# Quixo 3D Notes

## Possible Moves

Generally, I’ve come up with the formula:

Where n is the number of pieces in the rows, and d is the dimension (either 2D or 3D).

So, for a 5x5 board, that’s , or , or 847288609443, or 8.5 \* 1011. For a 5x5x5 board, the total number of positions is, or , or 4.4 \* 1059. These numbers aren’t entirely true. In a 2D game, the board can be translated such that a board position is repeated 4 times, and in the 3D world, it’s 24. Still, that doesn’t reduce the space much. And there are also numerous states that simply cannot occur (i.e. the entire board is filled with Xs or Os), but I’m not sure how that can be determined. Divide by 2? In any event, it doesn’t decrease the search space much.

For comparison purposes, checkers is 1020, chess is around 1050, and go is around 10200. To surpass go in complexity, you’d need to have an 8x8x8 board, which is about 1.9 \* 10244.

## Move Generation and Random Thoughts

Well, I made progress tonight. I basically figured out that our search space is HUGE, and doing anything more than a search 2-ply deep (i.e. a move and the responses to that move) is infeasible.

Here's my reasoning. With a blank board, you have 438 moves at your disposal. Now, that number will start to decrease with each move and it depends on what you opponent did on the previous move. From move 1 to 2 it will go down anywhere from 433 to 435. So, with just a 2-ply search, that's approx 180,000 moves it has to visit. Now, my Negascout search does wonderfully with this - it only calculates (i.e. determines a score for the board state) 869 of those positions (due to how it prunes the tree), which is great. It still have to go through all 180,000 moves but the point is that you do not have to calculate each move at depth 2. My test shows on average that I can find the "right" move in 0.3 seconds. When the depth is pushed to 3, things get out of control quickly. Now I have to go through 77,400,000 moves, although I'm only calculating 180,000 of them. 77 million may not sound like a lot, but it's 2 orders of magnitude and it shows: my result time grows to 36 seconds. So this sucks.

BTW with chess you start out with 18 valid moves. So does the opponent. After that it starts to vary widely. I know that Deep Thought (the IBM computer that beat Kasparov) could do 200,000,000 board calculations per second, but that was a huge supercomputer with specialized CPUs designed specifically with a chess board in mind. We don't have that at our disposal.

The problem I also have is that my evaluation returns the SAME DAMN VALUE FOR ALL MOVES AT A CERTAIN DEPTH. This sucks, because that means even with Negascout (and iterative deepening added!) all nodes will be evaluated. I have to get my “real” evaluation function in; otherwise, I’m sunk. I think I also need to do something that says for the first 6-8 moves, perform some kind of simple strategy and only kick in the deep search after a certain point in the game.

## Evaluation Functions

My first idea is to do what I originally did, except transformed for the 3D world: For each slice, go through each row, horizontally and vertically, and the 2 diagonals, and do a calculation on continuations (i.e. if the next piece was the same as the previous one, that’s a “good” connection). This means, though that I have to visit every single piece at least once, if not more. For each slice, that’s 25 (horizontally) + 25 (vertically) + 10 (diagonally), which is 60. There are 15 slices, so that’s 60 \*15, or 900. Plus, there’s the 4 diagonals through the cube, so that’s 4 \* 5, or 20, which makes it 620 visitations.

Ugh.

Here’s an idea. Go through each coordinate once. If the piece is non-blank, then figure it its’ radiancy. That is, figure out all the lines that it could make a winning line with. Evaluate those lines, and eliminate ALL of the pieces you just evaluated with that radiancy.

Well, it’s not quite that simple. I have to keep track of the lines that have been visited for that piece.

So, let’s say I start at 0, 0, 0. If the piece is NOT blank, I call some function GetLines(Coordinate c) and it gives me a List<Line>, where a Line has a Start and End. For each Line, I submit it to some object called VisitedLines, and say, HasLineBeenVisited(Line, Player). If false, then I go through the line, and evaluate it for connectedness. After the evaluation, I tell VisitedLines, LineHasBeenVisited(Line, Player), and move on.

With x, x, b, o, x

I would visit 7 lines (1 in x direction, 1 in y direction, 1 in z direction, 1 diagonal on 3 faces, and 1 diagonal through the cube) for x at 0, 0, 0. For 1, 0, 0, I would only need to visit 2 lines (NOT 3), one in the y direction, and one in the z direction. For 2, 0, 0 I just move on. For 3, 0, 0, I need to visit 3 lines because this is o, not x. For 4, 0, 0, it’s 6 lines, not 7.

What seems nice about this is that I only visit the line once for that…

Hey!

If I find a non-blank piece, evaluate the lines for both X and O. Then the API for VisitedLines just becomes a thin veneer over List<Lines> (remember that 0, 0, 0 – 4, 0, 0 is the same as 4, 0, 0 – 0, 0, 0) – i.e. I just check to see if a line is in the list. If it isn’t, then evaluate it and add the line to the list.

SWEEET!

Then…

With x, x, b, o, x

I would visit 7 lines (1 in x direction, 1 in y direction, 1 in z direction, 1 diagonal on 3 faces, and 1 diagonal through the cube) for x at 0, 0, 0. For 1, 0, 0, I would only need to visit 2 lines (NOT 3), one in the y direction, and one in the z direction. For 2, 0, 0 I just move on. For 3, 0, 0, I would only need to visit 2 lines (NOT 3). For 4, 0, 0, it’s 6 lines, not 7.

Creating the lines. There’s a standard way to get them for all the outer shell pieces. Also, there’s an easy way to get the 3 straight lines. For the middle pieces, you have at least 3 lines and as many as 13 (for the center). So generate the 3 lines, and then try to generate the 10 diagonal lines and catch the LineException if the diagonal isn’t long enough.

## TODOs

Revisit private constructors.

Add IEquatable to MoveHistory, and (potentially) to Board (that’s debatable, should it be equal based solely on current position? Probably…)

Test code (and some framework and engine code) assume a dimension of 5. Change this. To that end, having an odd dimension simplifies things (e.g. an even dimension yields 4 inner pieces on the faces and center that are part of the diagonals), so consider that to either be a rule or change code to accommodate dimension independence.

## References

* Checkers Complexity: <http://weblog.fortnow.com/2007/08/checkers-clarification-by-schaefer.html>
* Chess vs. Go: <http://www.everything2.com/index.pl?node_id=1310582>
* Computer Chess: <http://en.wikipedia.org/wiki/Computer_chess>
* MinMax Explained: <http://ai-depot.com/articles/minimax-explained/>
* MinMax Search: <http://www.seanet.com/~brucemo/topics/minmax.htm>
* An Improvement of the Scout Tree Search Algorithm (Negascout): <http://www.zib.de/reinefeld/bib/83icca.pdf>
* Iterative Deepening
  + <http://en.wikipedia.org/wiki/Iterative_deepening_depth-first_search>
  + <http://www.seanet.com/~brucemo/topics/iterative.htm>
  + <http://www.cs.usask.ca/resources/tutorials/csconcepts/1998_3/IDS/2-4.html>
  + <http://www.cs.mu.oz.au/303/slides/week02h.pdf>
  + <http://www.doc.gold.ac.uk/~mas02md/2AI/uninformedsearch.ppt>
  + <http://acm.fzu.edu.cn/reference/Search%20Techniques.htm>
  + <http://webcourse.cs.technion.ac.il/236501/Winter2006-2007/ho/WCFiles/UninformedSearch.ppt>