Tic-Tac-Go

Hartley Howe, John Miller, Morgan TerMaat, and Benjamin Pomeroy

howe.ha@northeastern.edu
miller.john@northeastern.edu
termaat.m@northeastern.edu
pomeroy.b@northeastern.edu

GitHub Repo:
https://github.com/jackmillerNEU/CS2800-Final-Project

Abstract:

Positional games are a mathematical class of 2 player games of varying complexity, all the way from tic-tac-toe to the immensely complex game of Go. Every positional game can be represented as a graph containing vertices and hyperedges, where vertices are spaces on the 'game board' players can claim, and hyperedges are sets of vertices one player must own to win the game. In tic-tac-toe the vertices are each square on the 3x3 grid, and the hyperedges contain the winning runs of 3 (horizontal, vertical, and diagonal lines). In this paper we will follow the research done by Valentin Mayer-Eichenburger and Abdallah Saffidine which can translate a representation of a positional game into QDIMACS format, which can then be solved by a QBF (Quantified Boolean Formula) solver to return whether a board can be solved in the next move. We will decipher and simplify their research, and detail the entire pipeline. From choosing a novel positional game, to converting the game into a format expected by the encoder mentioned above, to running a QBF solver on the encoded file to see whether it can be won in the next move and the attempt to finding that next move, this process will be made clear.

Introduction:

Quantified Boolean Formulae are a generalization of boolean formula to include existential (there exists) and universal (for all) qualifiers on variables. This makes them especially helpful for two player games, where 'solving' a game for one player involves considering every possible move the other player could make in response. In this way, we would find a winning move for player 1 by asking if there exists a move on turn 1, such that for all moves player 2 can make on turn 2, there exists another move player 1 can make on turn 3, such that… and on and on until the number of moves calculated is the maximum length of the game in question. $\exists m1\ -> \forall m2, \exists m3\ -> \forall m4, \exists m5\ -> p1\, wins\ ==\ true$. Another important thing to know about positional games is that the majority of them fall into two sub groups, maker-maker and maker-breaker games. In a maker-maker game the first player to claim every vertex in a hyperedge wins the game, while in a maker-breaker game if the game ends with no one claiming a whole hyperedge, the second player wins. So sometimes this formula will end with "there exists a move such that player 1 wins, and sometimes it will end with "there exists a move such that player 1 has not won". In practice, with real non-trivial games, these formulas become massive. This is largely because every move has to be logically encoded so that neither play can cheat or otherwise make an illegal move, and so that each player *must* play on their assigned turns (Mayer-Eichenburger).

During the investigative portion of our research, we found that the direct encoding process to QBF was many times more complex than we expected, and contained more nuance than a SAT encoding would. The previous research we found which encodes positional games is 2300 lines of low level Go code, which was largely indecipherable to us. Instead of reproducing this extensive work, which has already done what we initially wanted to accomplish, in another programming language, we decided to do a more meta analysis using this work to go from a specific positional game to actually solving the next moves for a game instance. This involved many moving parts, including writing our encoder of a specific positional game into the format expected by the general positional game encoder, getting the Go code to run (which turned out to be a big challenge), as well as getting a QBF solver written in C++ to run on the output of the golang encoder. We have worked through this process and provide documentation of reproducible steps future researchers can take to solve other positional games.

Background

Our initial idea of a game to solve would be nonogram but since we found solutions to that we looked towards similar games. Jim had created a sliding form of three dimensional tic-tac-toe for a previous course. This became the inspiration for our project. Our game has one deviation from 3D tic-tac-toe, which is that a number of the 27 cubes that make up the board have blocks that may not be claimed by either player. We were originally going to try to encode a variant of this game where players could push one of the blocks to an adjacent unclaimed space as well as claim a space. We decided against pursuing this when we realised that it would necessarily involve modifying the golang encoder significantly, which was not feasible for us in the allotted time.

We started out our research by looking for a QBF encoder that already existed for a normal game of tic-tac-toe played in 2D. We found a paper about a generalized QBF encoder for games of generalized tic-tac-toe, a game played on any size board and with an arbitrary shape as a winning configuration, by Valintin Mayer-Eichberger and Abdallah Saffidine. We were able to download the Go program from the first paper that did the QBF encoding, and we eventually got the encoder to work and output a QDIMACS file, on the provided examples.

The novel part of our work is a Python script that uses a simple GUI to allow for the user to specify a gamestate (who has moved where, and where there are blocks) and outputs an input file for the QBF encoder which we can then use to solve the given gamestate. Our metric for success with this project is being able to use a QBF solver on a given gamestate of our 3D tic-tac-toe game, and receive some kind of output. Therefore, we consider this project to be successful.

Walkthrough

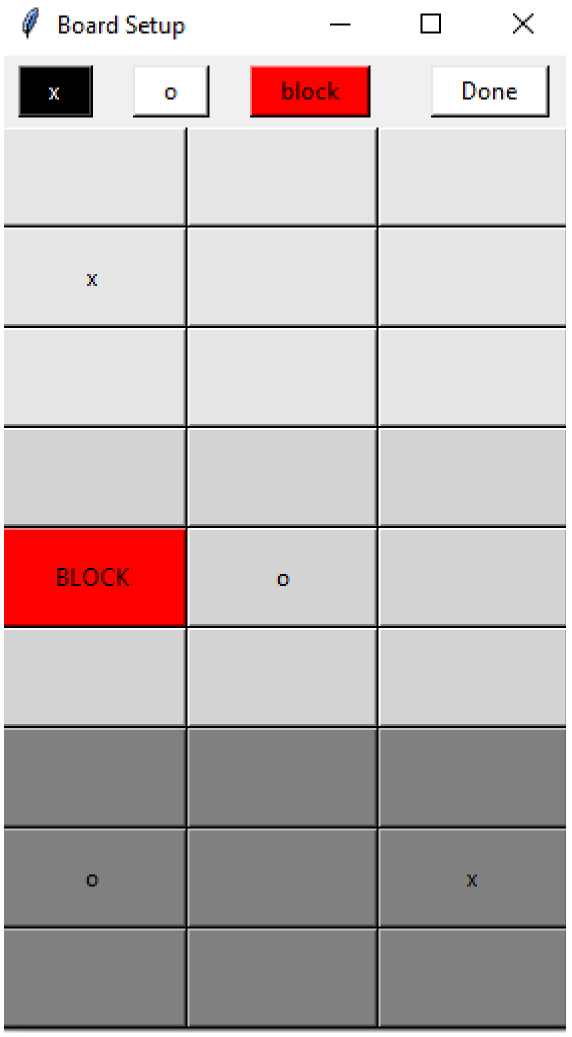
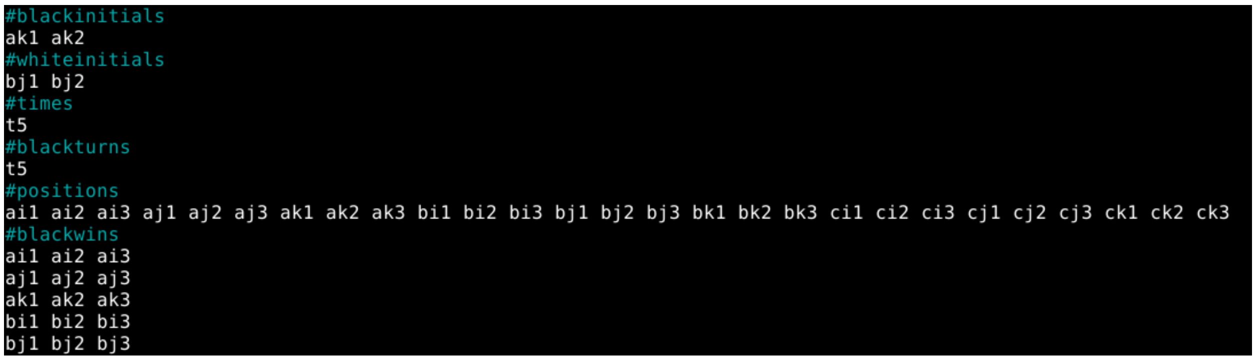


Figure 1: The board setup GUI

The key to getting to the QBF solver was initially to write a Python script, shown above in Figure 1. We wrote this script to take a three dimensional, stacked game board and correctly format it. The correct format is output to the file that gets passed along to the vale1410 encoder. This is where we ran into our first large struggle as there was no documentation on how to install or run the project. On top of the difficulties installing the project the installation of golang itself was difficult and the project required a specific version. Importantly the encoder was missing the go.mod file it required to run. Fortunately we were able to solve all of these issues and proceed to the next step in our chain.

After the encoder is successfully running it will give us the appropriate QDIMACS input that then gets passed to the actual QBF solver (labeled boardMap.qdimacs). This solver, similar to the encoder, lacked documentation. Additionally it used an older version of C++. At first we were unable to even determine which file to run. At first we thought we were unable to get a QBF solver to run but determined that the QBF solver was not the issue but instead it was the encoder. Thus we went back to the encoder and determined the correct way to run it and determined that we also had to run the ground.go file.

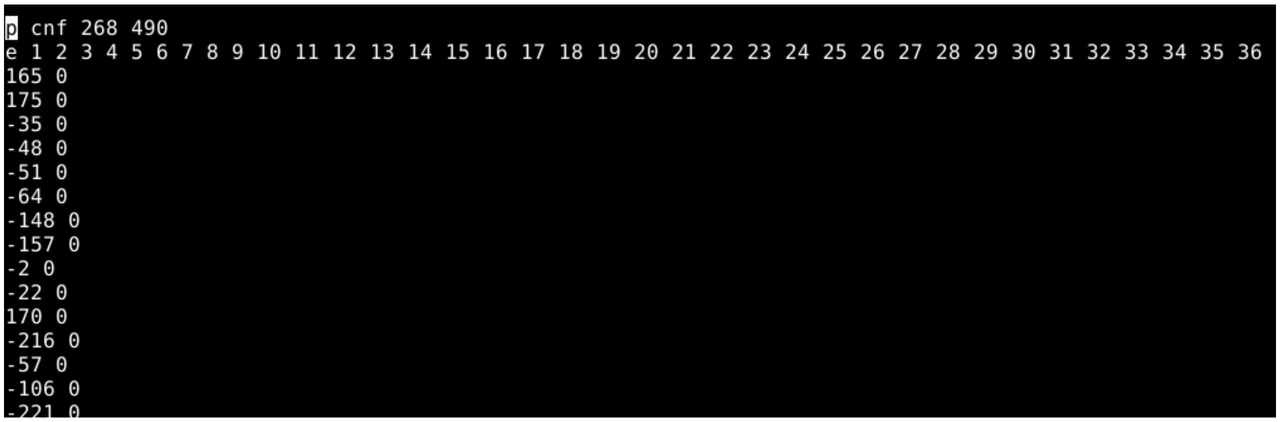
The following is a step by step look at a single run of the program. First the user is able to run the Python program (via the line ./build-a-board.py) which will display a board similar to the one shown in Figure 1. In this run the Python script saves the contents shown in Figure 2 to a file. Figure 2 shows the first portion of this file however there are several more lines of in the #blackwins section.



```
#blackinitials
ak1 ak2
#whiteinitials
bj1 bj2
#times
t5
#blackturns
t5
#positions
ai1 ai2 ai3 aj1 aj2 aj3 ak1 ak2 ak3 bi1 bi2 bi3 bj1 bj2 bj3 bk1 bk2 bk3 ci1 ci2 ci3 cj1 cj2 cj3 ck1 ck2 ck3
#blackwins
ai1 ai2 ai3
aj1 aj2 aj3
ak1 ak2 ak3
bi1 bi2 bi3
bj1 bj2 bj3
```

Figure 2: Python script output

This file was then given to the encoder and then run through the ground program as well (via ./run_encoder.sh) to output what is displayed in Figure 3 to a QDIMACS file. On this particular run the file output was a little under 500 lines long. The run_encoder.sh is a shell we adapted from a benchmarking shell to run our inputs. It copies the output to the depqbf solver directory.



```
p cnf 268 490
e 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36
165 0
175 0
-35 0
-48 0
-51 0
-64 0
-148 0
-157 0
-2 0
-22 0
170 0
-216 0
-57 0
-106 0
-221 0
```

Figure 3: QDIMACS file

Then the QDIMACs file is given to the QBF solver which first is run via ./depqbf $filename.qdimacs and returns SAT or UNSAT. This step can also be run with ./depqbf --qdo $filename.qdimacs which gives this simplified output.

```
s cnf 1 248 448
V 1 0
V -2 0
V -3 0
V -4 0
V -5 0
V -6 0
V -7 0
```

……… this continues several hundred lines, with no clear structure or meaning

This output file is the same type as the input, a .qdimacs file. We have researched this type of file extensively, but the documentation is considerably lacking. The .dimacs file type that it is built off of does have some better documentation, but was still unclear. The output, shown above, and in the examples in the repo are obnoxiously large series of numbers, somewhere in which might be information that leads to the players next move. However, we were not able to figure this out under the time constraints.

We may have not yet been successful in extracting the position that a player should go from this QDIMACS output file, but we are able to determine whether X has won or can win on the next turn as indicated by the 1. The next steps if we were to continue this project further would be to run the same process on a gamestate where the options are limited to determine how to interpret this output. Additionally we could simplify to one dimensional tic-tac-toe to complete this process.

Results

The ultimate goal for our project would be to convert a 3D tic-tac-toe board to a QBF solver. This would effectively allow us to solve our three dimensional version of the game. If we can achieve this, we will have succeeded by our own metric of success. We have successfully accomplished this and have been able to produce output that determines if player X is able to win on the next turn, as shown below.

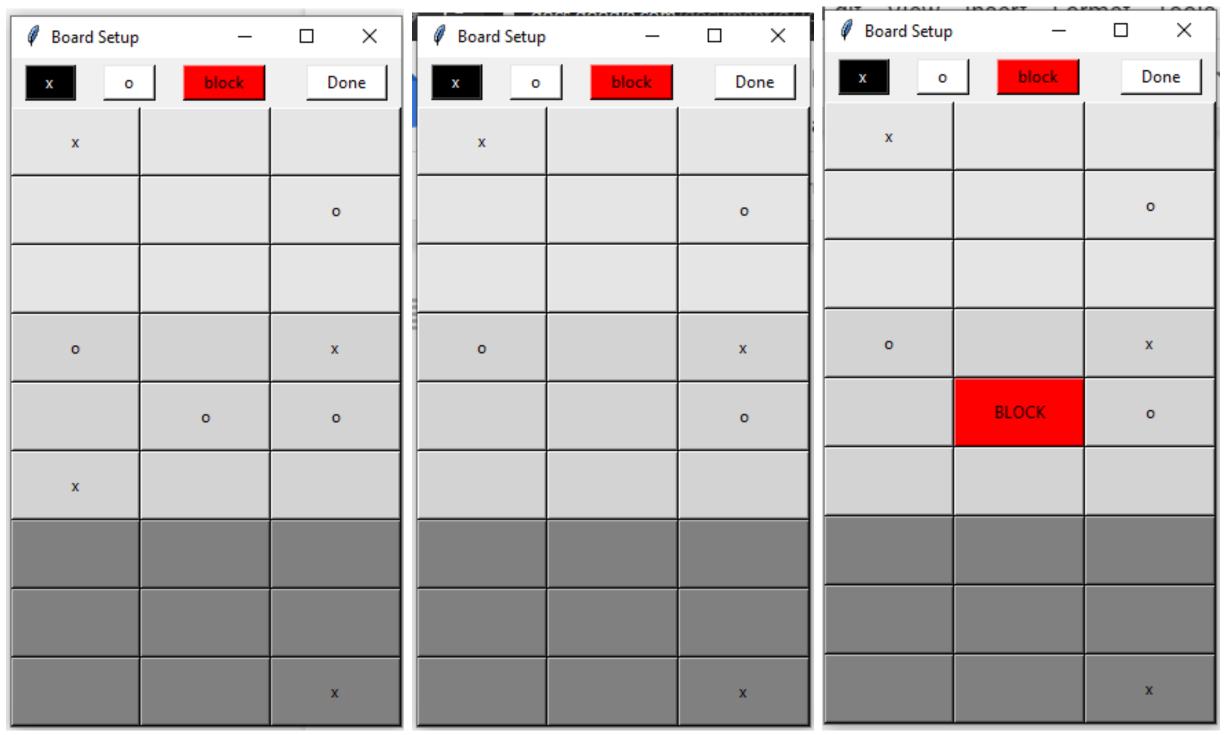


Figure 4: Boards 1, 2 and 3

The boards shown in figure 4 above show three boards, where x is trying to win by going from the top upper left diagonal to the bottom lower right, the most diagonal possible winning state. In the first example, this path is blocked by an O, and the path is unblocked in the second example. In the third example, the board is missing the central spot, so possible winning arrangements have to be adjusted. This GUI input is then outputted as a pg file, seen as exampleSAT.pg, exampleUNSAT.pg and exampleBLOCKED.pg in the repository. These are then run through the Go encoder and outputted as exampleSAT.qdimacs, exampleUNSAT.qdimacs and exampleBLOCKED.qdimacs. These are the inputs formatted to be inputted to the depqbf solver. The solver output can be seen in the txt files in the repo, but the boards are accurately solved. Only board two returns SAT when run through the solver, because it is the only one that has an unblocked path.

Personal Progress

Initially we had a more ambitious game setup where instead of placing pieces they would instead be pushed along the board. This proved to be much more difficult to encode than originally anticipated and thus we had to take a step back. Instead, our game is a version of 3D tic-tac-toe where certain spaces can be blocked out. Our largest issue then became the lack of documentation for both the encoder and the solver. We originally thought that our large issue was the QBF solver however this turned out to be an issue with the encoder instead. The issue was that we were running the encoder without also running the ground function after. Although we did not do a simpler version of the game there are several examples of solved tic-tac-toe with the simple 3-3 board. We first tested encoding basic 3D tic-tac-toe before including the blocked positions element of the game. Additionally we ran the encoder and qbf solver on their respective provided examples in the process. Luckily we have not yet tried to prove any claims that were in fact invalid. While we did run into a lot of problems, we learned a lot. We struggled our way through using and often learning from scratch, the following: Ubuntu, golang, c++, qbf solvers (ghostq, depqbf), python/python guis, and more.

Summary

The results of this project that will be most useful to others include our documentation of how to use tools such as vale1410's positional games encoder and QBF solvers such as ghostq or depqbf and in tangent to arbitrary positional games, in our case tic-tac-toe. From accomplishing this project we will have found a way to convert the board of a slightly modified version of 3D tic-tac-toe to QDIMACS and then feed it to a QBF solver. This has involved research on vale1410's positional games encoder, and getting a positional game from it's board state to a

QBF output that tells us what the next move should be, or at least whether it can be solved in the next move. Our process is relatively easy to repeat for other positional games using vale1410's encoder, since they have shown an approach that makes things more clear to those who do not have much background in SAT / QBF solving. The process we have demonstrated starts with entering the state of a board into a GUI which is then read by the "build-a-board" Python script. The Python program then outputs the code in a format used by the Golang positional games encoder. This encoder then outputs the code into QDIMACS format, which can be solved by most QBF solvers. This process now runs smoothly and produces reasonable results for the user to interpret. A simple SAT or UNSAT signals whether the board can be won by player X on the next turn.

References

Mayer-Eichberger, Valentin, and Abdallah Saffidine. "Positional Games and QBF: The Corrective Encoding." *Theory and Applications of Satisfiability Testing – SAT 2020*, 2020, pp. 447–463., doi:10.1007/978-3-030-51825-7_31.

Gent, Ian P, and Andrew G D Rowley. "Encoding Connect-4 Using Quantified Boolean Formulae." University of St. Andrews, University of St. Andrews, 2010, citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.5.9915&amp;rep=rep1&amp;type=pdf.

Shen, Zhihe. "USING QBF SOLVERS TO SOLVE GAMES AND PUZZLES." Https://Www.bc.edu/Content/Dam/bc1/Schools/Mcas/Cs/Pdf/Honors-Thesis/sample5.Pdf, Howard Straubing, 2014.

Appendix