Progress Report:

3D Tic-Tac-Toe QBF
Howe Miller Termaat Pomeroy

For our project, we chose to use a QBF (Quantified Boolean Formula) solver to solve a version of 3 dimensional tic-tac-toe. In our research, we discovered that just last year a team in Germany developed a golang encoder which takes in a simplified representation of a ‘positional’ game and outputs a QBF formula which can be solved to find a winning move if one exists. At first we were a bit disappointed that our initial goal had seemingly already been done, but looking further we realized it would be worthwhile to actually use their work to encode a game, and outline our steps start to finish for future researchers to follow. Although the most low level part of the process was already complete, there was a lot of work to be done to get the encoder running properly, to get an actually QBF solver running and interpret the results, and to devise a conversion from a physical game to the format expected by the encoder. None of these steps had any useful documentation, so we spent a lot of time getting each part running in the hopes that someone following along could repeat our process much faster for a game of their choice.

We believe this type of ‘meta research’ is very important, because it allows these fascinating results about logically solving positional games (which includes simple games like tic-tac-toe, as well as complex games like Go) to be brought to many more people. We could not find anything online about actual implementations of the central research our work is using, and through our struggles we have shown that for most of the population the process would be tedious and might prevent them from reaching their goal. With a clear guide to follow containing how to represent your game in the right format, and how to run each required piece of software, many more people can have the chance to solve their favorite game!

So far, we have created a complete python program which takes a 3 dimensional array and generates an output file which can be passed to the golang encoder. We have also done some debugging of the encoder, and added a configuration file to get that running and returning a second output in QDIMACS format, the format required by QBF solvers. We still need to finish writing, revise, and properly format our final report, once we have finished the rest of the work. The final part that we are still working on is getting a QBF solver to run successfully on the encoded output, and return the next move. We have tried many solvers so far, GhostQ being the most promising, but lack of documentation and obscure language use is getting to us. With focused effort on this in the coming days we are confident that this barrier will be broken and we will get the solver running.

Github Link:

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

Report Draft:

Tic-Tac-Go

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

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 a winning move for a given player if one exists. 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 and finding the next move a player should make.

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 exists 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 it to work 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 that we consider being able to use a QBF solver on a given gamestate of our 3D tic-tac-toe game. Therefore, we consider this project to be successful.

Walkthrough

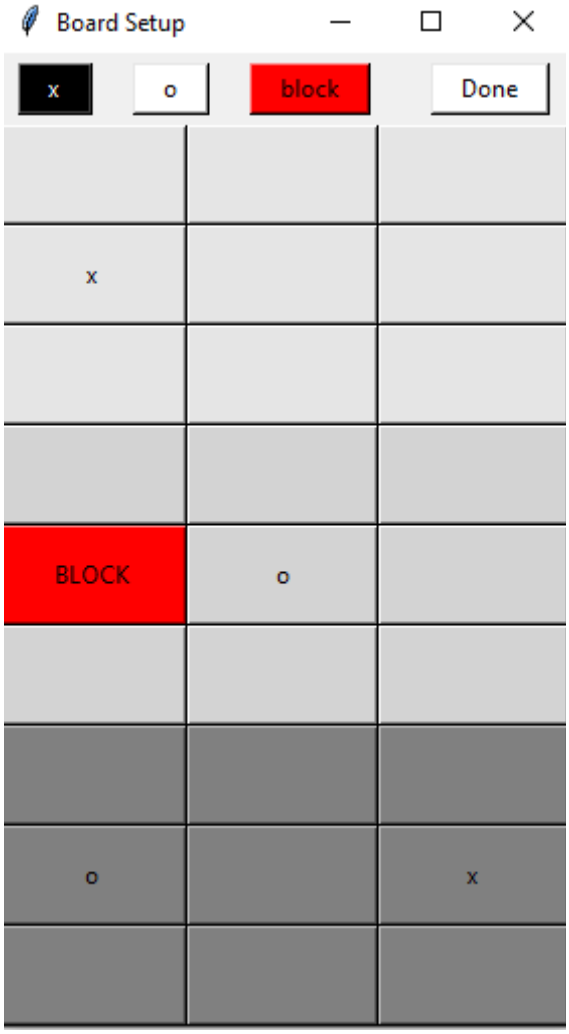

Figure 1: The board setup GUI

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. Thus far we have not been able to get the QBF solver to run. This QBF solver then gives the next best move.

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. As it stands we are about two thirds of the way through, as we just have to find a working QBF solver with enough documentation to run our input and get an output in a manageable form that we can see what the next move should be. We have already spent lots of time finding and testing QBF solvers, but without any success.

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 we went with the simple place of piece gameplay. Our largest issue then became the lack of documentation for both the encoder and the solver. The solver is currently where we are finding the most difficulty in getting it to

run. 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. Luckily we have not yet tried to prove any claims that were in fact invalid.

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 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. Our process can be repeated for other positional games using vale1410's encoder, and we 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. At this point we have been unsuccessful in our attempts to pass the encoder's output successfully through a QBF solver.

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.

Appendix