Hex Solver Visualizer

Github: `https://github.com/dpankratz/Cmput497Hex`

1 Introduction

This report outlines the features and stages of development of a Hex strategy visualizer tool. In particular, this visualizer focuses on strategies encoded in an iteratively decomposing graph where edges are taken based on the moves of the white player and each node contains a winning black move. The goal of this tool is to greatly increase the productivity when developing or understanding such a strategy and this goal is accomplished in the following ways: It is possible to rapidly experiment in depth by rapidly diving into lines of play as well as to experiment in breadth by, at a glance, understanding the decomposition of a given state. Productivity is also improved by the introduction of a cross-platform application that can be built on the platform suited to the needs of the user.

2 Development

2.1 Unity Project

The development of this tool began at the beginning of this semester and was based on a Unity project built in the 2018 session of Cmput 497. The project originally featured the ability to create arbitrary hex states as well as player vs player and player vs random games on those states. In particular boards of any size could be created dynamically as well as boards containing stone counts impossible in a regular game (example Figure 1. These boards could be exported and imported from the clipboard and the project supported undoing and redoing moves in the player vs player mode.

The features of dynamically creating boards of varying sizes, the human player, and undoing became very important in the latter developments of the project.

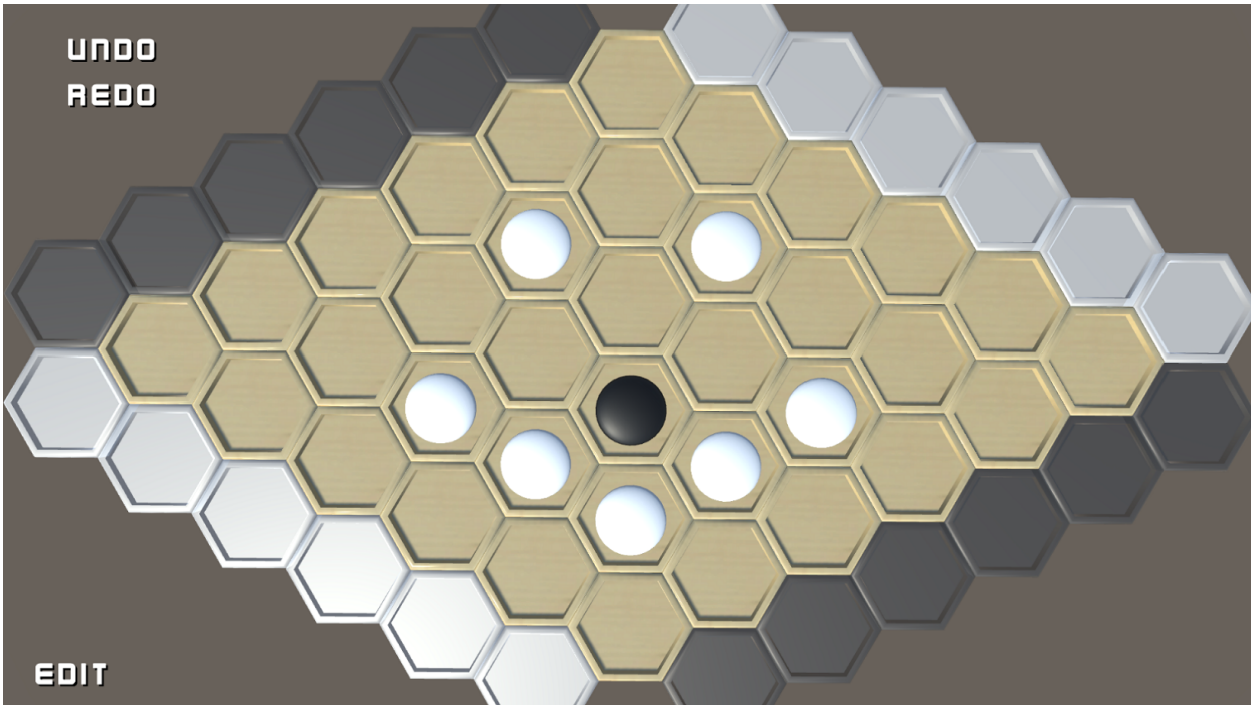

Figure 1: **Original project.**

2.2 Hex Player Backend

In the view of the author the remaining missing piece of the project was the capability for a human player to play against a strong hex bot. This feature has a number of benefits such as acting as a tutor to new players as well as being able to show the *thinking* of the bot through the power of a game-engine. Thus to accomplish the task of adding a hex bot to the Unity project the Benzene project was selected which features strong and well-tested bots. The first bot that caught my eye was the JingYang solver for 9x9 boards since it was mentioned this solver once had a Javascript front-end that lacked usability features such as undoing moves and lacked visualization.

Due to games primarily running on Windows, compiling and running Benzene on Windows was investigated but proved to be very hostile. The multiple dependencies and overall CMake structure did not play well with Windows and so that path was abandoned. Luckily, Unity Editor is available for Ubuntu and Benzene was very easy to compile on Ubuntu so exploring Linux was the next course of action. Very quickly, the project was running and Benzene was running so all that remained was to connect them. For this a custom API was developed that allows the front-end programmer to send arbitrary Benzene commands to the underlying executable. Armed with this API it was easy to integrate it into the Unity front-end to allow a human to play against the JingYang player.

3 Visualization

3.1 Counter move

Once the human vs JingYang match was functional, a novice player could attempt to play this bot and see very strong moves as an answer to whatever they were attempting. However the goal to disseminate how the bot produces these strong moves in a way that is intuitive to Hex players of varying levels of experience remains. The first visualization developed was the real time

representation of which black counter move Figure 2 black would play in response to the current white move being hovered.

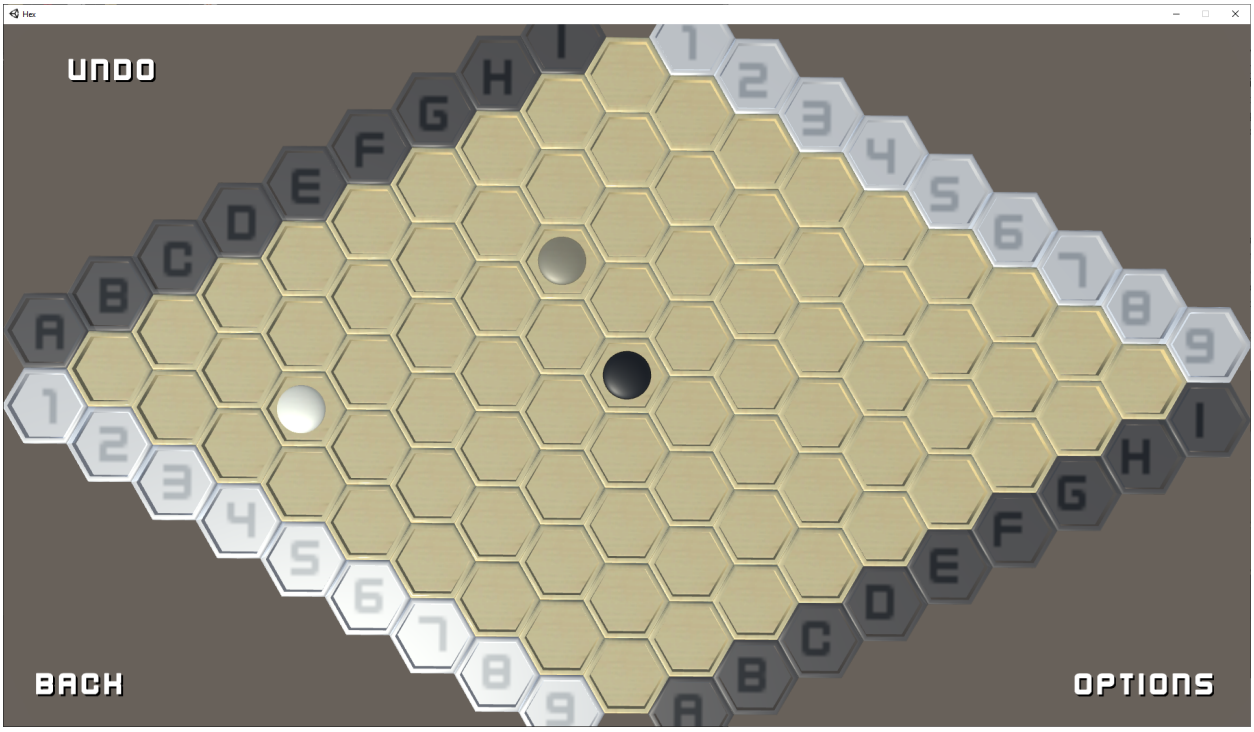

Figure 2: **Black counter move F3 for hovered white move B3.**

The counter move visualization gives depth to the visualization by allowing the human to quickly simulate a 2-ply sequence simply by moving the mouse. It also informs the user about black's upcoming strategy which allows novice players to learn much more quickly by simulating many moves before committing to a given one. However, this visualization is not helpful in discerning how black arrives at the move.

3.2 Strategies

For the next visualization the output of the command `show_jypattern_list` is used. This command was originally included in the benzene player and displays which cells on the board currently have a rule encoded to counter a white response. Thus when a cell is colored it represents a move that requires consideration by the strategy. Conversely, when a cell is not colored then black can take a random edge in the strategy and still win. An example of this visualization is shown in Figure 3.

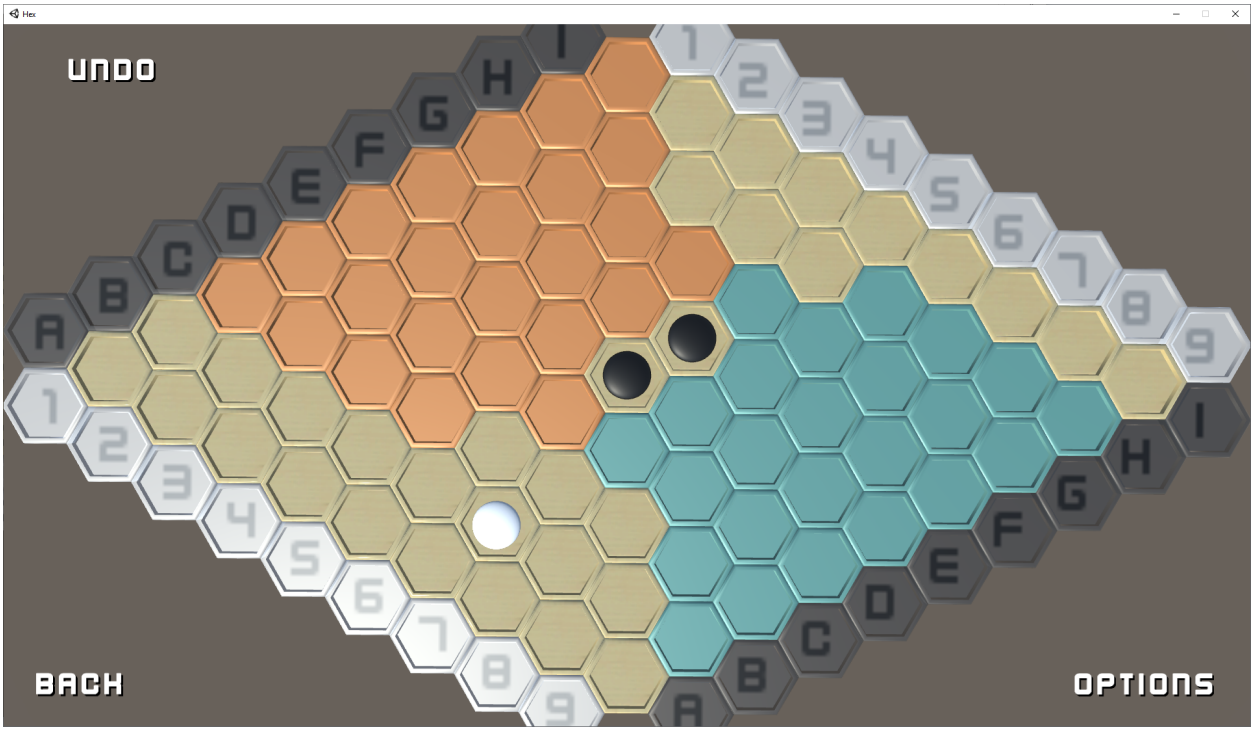

Figure 3: **Patterns after white B6. Uncolored cells indicate trivial black strategies.**

The pattern visualization can be used as a rough heuristic of the strength of a move, the more cells that are left colored after a move, then more consideration is required by the black strategy. Using both this feature in conjunction with the counter move proved to be an excellent visualization as it allowed a number of advances in the project: The first advance was using it to quickly identify a bug in the Benzene player where the undo command in `JYEngine.cpp` where the board failed to unflip when undoing the first move. Secondly, using the aforementioned heuristic it was easy for a novice player to discover a much longer line of play Figure 4 than they had prior to the visualization.

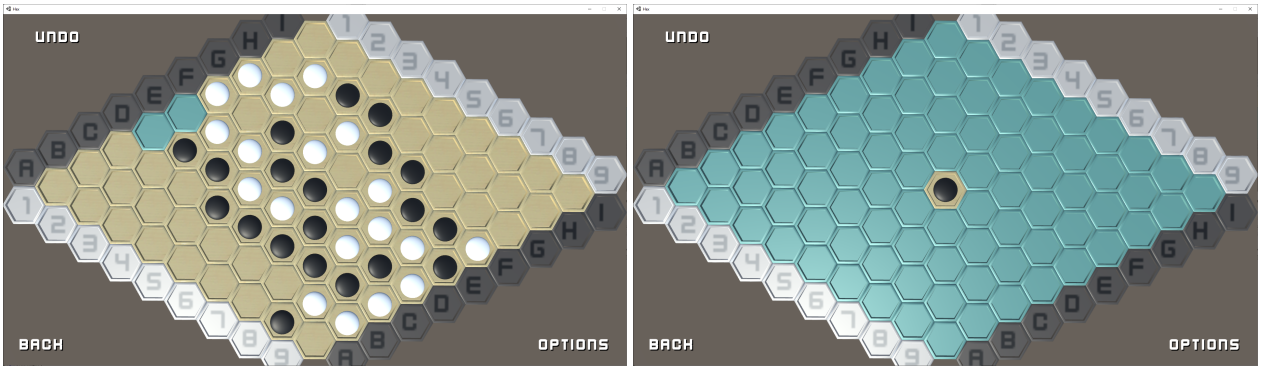

Figure 4: **Left: long line of play. Right: pattern for first move.**

3.3 Decomposition

This visualization could still be improved further since, although it gives some sense of the strength of a white move, it does not enumerate exactly how Black understands differences in potential

white moves. This issue can be seen in Figure 4 where the entire board is highlighted on the first white move which does not convey that there are stronger and weaker starting white moves. The solution to this was two-fold.

The first solution was to differentiate black strategies since a group of white cells could be answered by the same black strategy. These strategies are also referred to as `branches` (or as 'star' nodes in Chao's notation) and each branch describes how to decompose the current rule into subrules for each white move (the white moves are not exhaustive since in the case of weak white moves, a random strategy can be selected). These strategies or branches are shown for the first move in Figure 5. It can be seen that there are precisely 14 black strategies for the first white move and also that many white moves may be answered by the same black strategy.

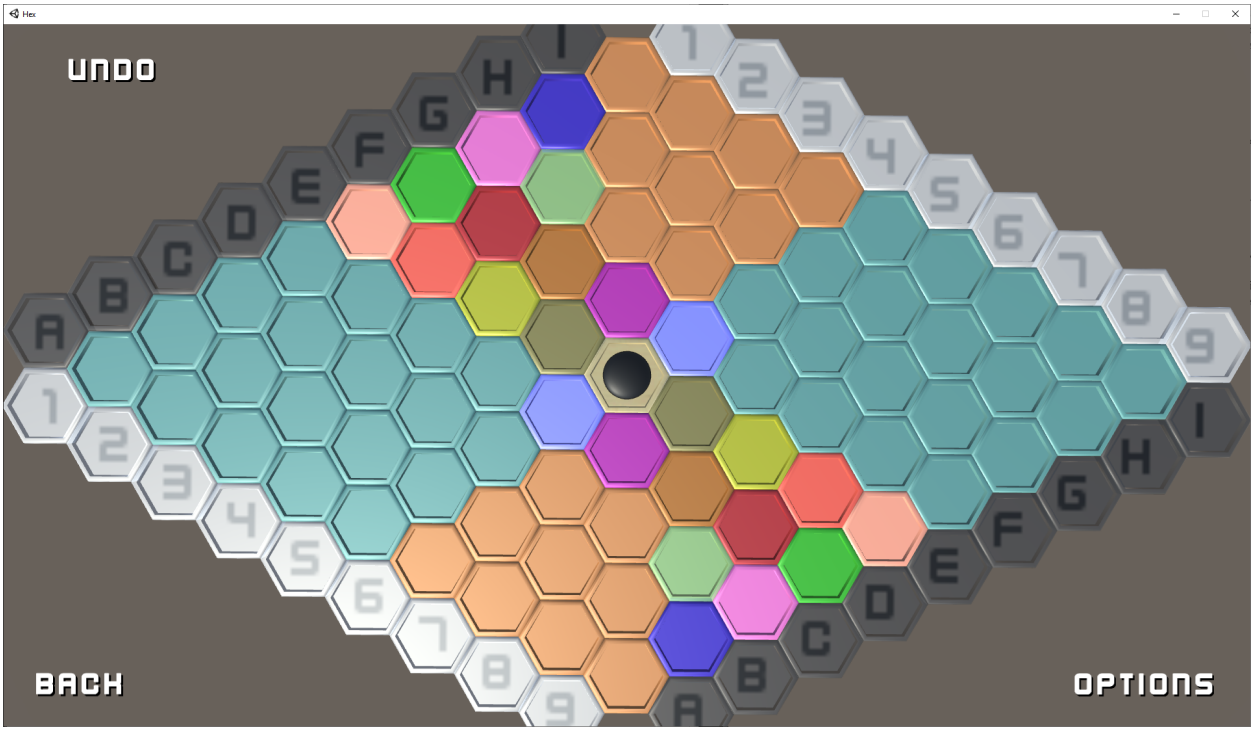

Figure 5: **Decomposed first strategy.**

After adding the decomposition visualization it becomes much more clear how black differentiates between different groups of white moves. However, one interesting observation is that the JingYang 9x9 player has 5 counter moves (10 after reflection) to white's first move but as shown previously there are 14 strategies. This indicates that there is some aliasing between strategies and the moves they make which is clearly shown in the next visualization Figure 6.

3.4 Development Assists

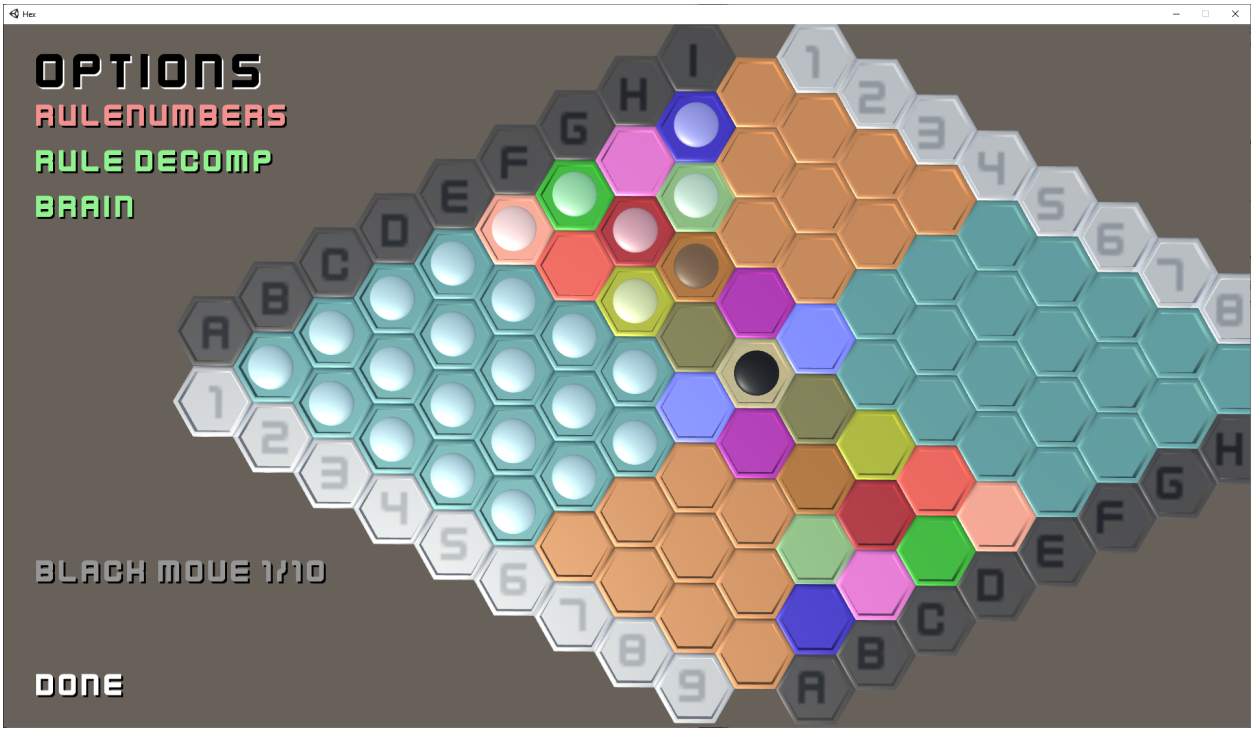

Figure 6: **All white moves which are countered by the black move F3.**

It is apparent after counting the number of strategies that are overlapped by the single black move F3 why there are 5 black counter moves and 14 strategies. This visualization is designed to provide another dimension with which to understand how the underlying strategy functions. It is likely that someone designing such creating a new solver would wonder if any two strategies could be joined into one and this visualization would be the first step in answering that question.

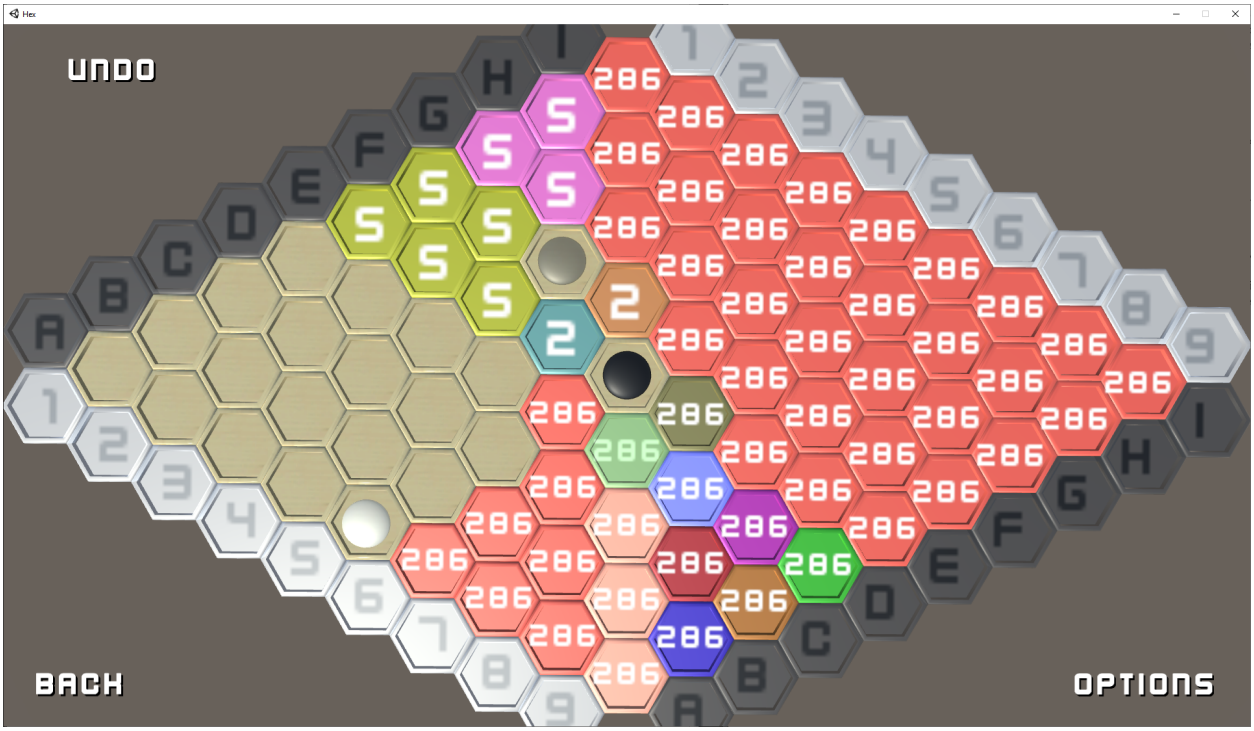

Figure 7: **Strategies annotated with their rule numbers. Bridge has label 2.**

The next visualization developed with the aim of indicating which rules the solver is currently considering. For example in Figure 7 the solver is considering a bridge labeled 2, a 432 labeled 5 and then a custom rule labelled 286. Again this would be helpful to someone who wishes to understand the functioning of the solver by allowing them to locate a rule of interest and quickly recognize common rules.

4 New solvers

Armed with all the visualizations of the previous section it is significantly easier to understand and test a given solver. However the JingYang solver is still highly intimidating since there are 26700 lines describing 715 rules. To ease this burden a 3 by 3 solver and 4 by 4 solver (shown in Figure 8) were developed and given first-class integration into the Unity project. The 3 by 3 solver has 38 lines and 2 rules and the 4 by 4 solver has 63 lines and 3 rules.

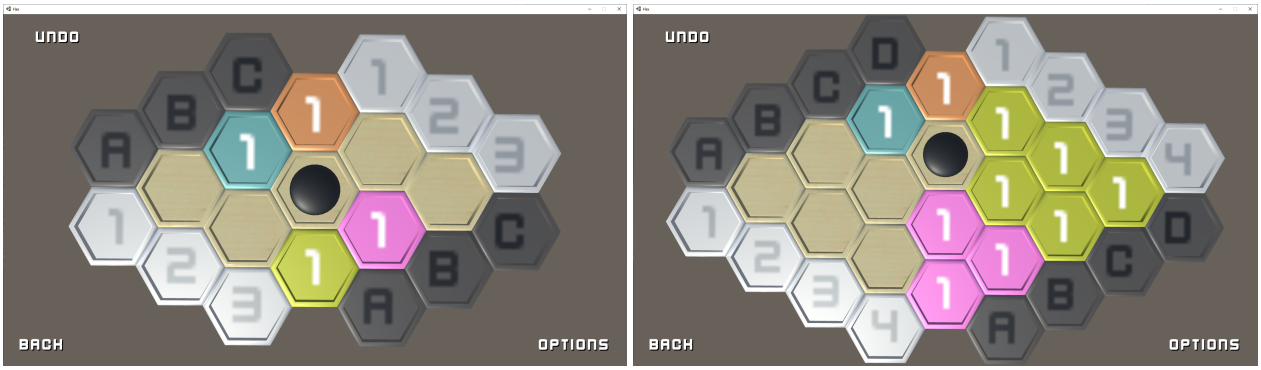

Figure 8: **Left: 3 by 3 strategy visualized. Right: 4 by 4 strategy visualized.**

Despite these seeming very simple, quite a lot of a work was required to develop them. The benzene implementation is baked to use the JingYang 9 by 9 solver and is therefore difficult to change in order to support solvers of other sizes. Furthermore depending on the Benzene executable requires that Benzene is built on every machine that wishes to run the visualization. The solution to both these issues is to develop a standalone solver and much of that work had already been undergone by Chao Gao. However, this standalone implementation had breaking bugs and needed to be expanded to fit the needs of the visualization.

After fixing the bugs and adding the functionality to clear the board, undo moves, detect the end of the game, and output all the required visualization data, the standalone could be integrated with the Unity Project. One benefit that was not anticipated was that since this standalone solver code is a single C++ file, it is actually possible to compile and run it on Windows. Thus a standalone build can be created for Mac, Linux and Windows and Benzene needn't be compiled at all. Overall developing this alternative backend brought with it many wins in terms of portability, flexibility and usability and was well worth the time investment.

5 Web build

While writing the future work section for this report, I realized that porting the aforementioned C++ standalone client to C# would bring further benefits. This is also noteworthy simply because the code would be a homogeneous Unity project and not require any sort of complicated process communication and cleaning up processes upon quitting the application. Performing the port also enables a web build which would allow anyone to use the project simply by navigating to a certain URL. This is obviously a much better prospect than what was originally the case, requiring benzene to be compiled for the visualization to work.

6 Conclusion

This report has enumerated the different functionalities introduced over the course of the semester. It moves the solver from a less user friendly front end to one that many useful features and rapid powerful visualizations. Beyond that the project is powerful in its cross platform compatibility and flexibility to support different solving agents and board sizes. The goal of creating a tutor is also accomplished which is proven by the developer of the project improving his own Hex ability through the use of the tool.

7 Retrospective

This project blew away my expectations going into the course. I was able to improve my skills in many areas involved which includes Unity, Hex Solvers, Hex, Benzene, C++, C#, and .Net to name a few. I believe this intersection of skills has the potential to produce much more than I have done in this course in the area of visualizations. In particular the capacity of Unity to display information in 3D seems very promising to me.

I'm elated that this project has the potential to aid future research such as creating a solver for larger hex board sizes. I'm looking forward to hearing about and contributing my help where needed to this effort. I'm also proud to have improved the quality of the project so much over the course of the semester. It was immensely rewarding to reap the tangible benefits of changing the back end from benzene to C++ to C#. I also believe that having put in that effort, the project is significantly better for it.

8 Future work

1. The strategies used in JingYang's 9x9 solver are not minimal as seen in Figure 9. One project could be to detect circumstances where the decomposition could be pruned such as in the figure.

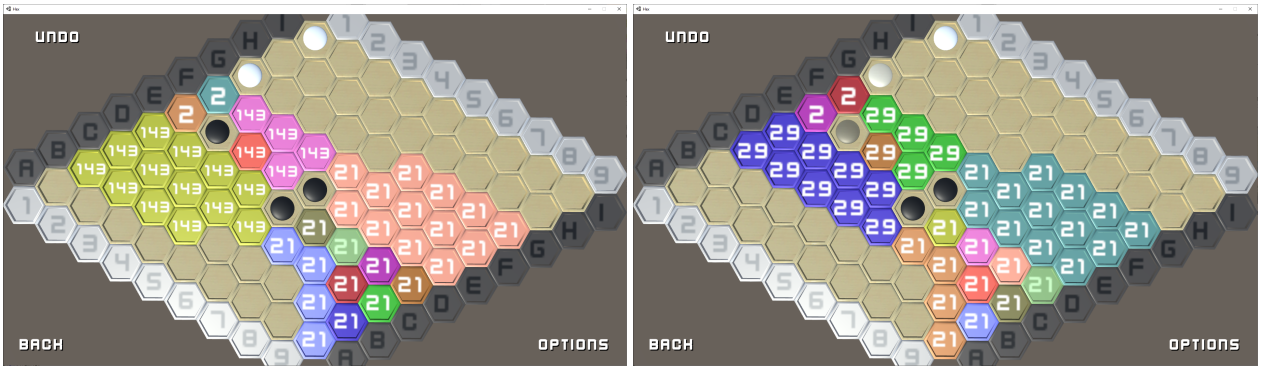

Figure 9: **Left: White move G1 then I1. Right: White move I1 then G1. Rule 143 could be replaced by Rule 29.**

2. The standalone C++ backend could be rewritten in C# to easily enable a web version of this project as well as Android or IOs. This could be a big win as hosting a web version for students to play with could potentially act as a great tutor for classes such as *Games,Puzzles, Algorithms*, *Combinatorial Game Theory* or *Search,Knowledge, Simulations*. **I decided to complete this since bringing this to students is very exciting and one of my original goals for the course.**

3. An ambitious expansion to the project could also be to create a tool that is designed such that a user can create their own solver from scratch. For example by selecting a region, a black move, and which patterns to use, a branch could be defined and then written to the strategy file. This way an entire strategy could be developed in a single unified environment. This would absolutely require front-end Unity and UI work which is likely to be the challenging to someone who lacks Unity experience.

4. Now that this app can run on the web, there might also be some utility in adding functionality for the user to upload a solver file to play against. I believe this could be implemented easily since in the original project I added the ability to upload/download game states which could be modified to include uploading/downloading solver files. The flow would likely be something like this:

   (a) parse clipboard text

   (b) use a animated popup display success and number of rules loaded or failure

   (c) change boardsize and then begin playing.

5. Another implication from having implemented a C# is that the information in the solver is much closer to all the scripts in the project so passing information over strings can be avoided. In a previous project I developed a real-time visualization of a MCTS chess program which showed how the DAG is constructed as the bot plays (**??**). This project also relies on a bot's knowledge of a graph and thus displaying the 'tree' used by a solver could be an interesting extension.

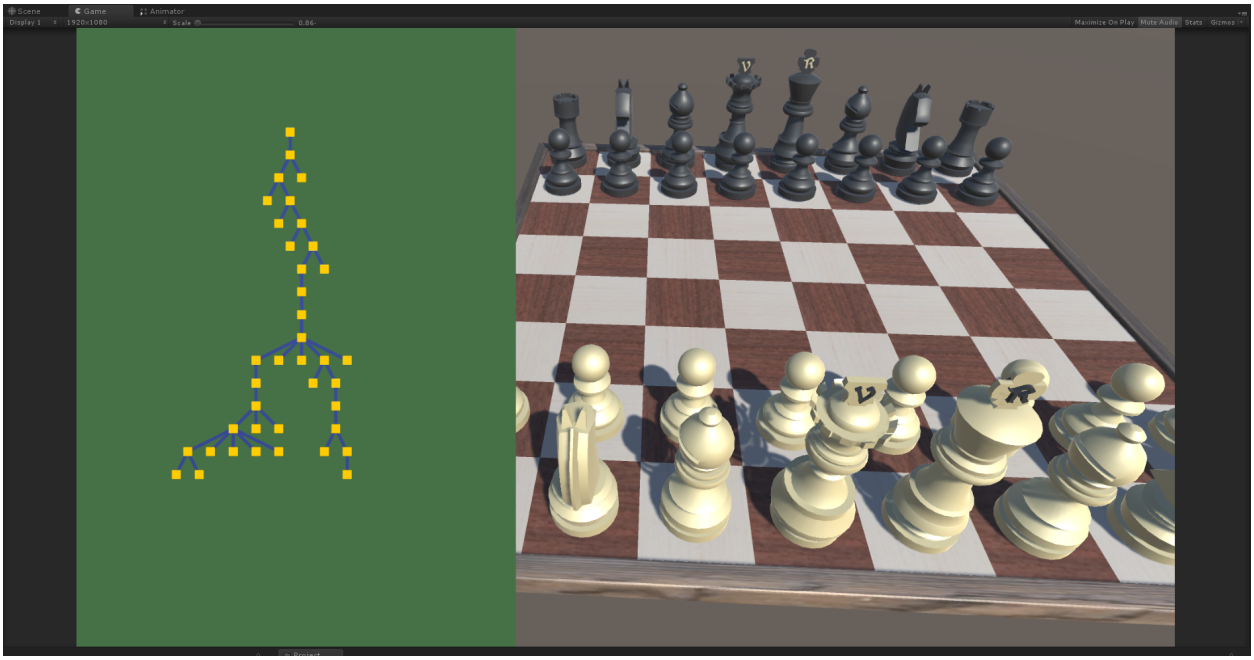

Figure 10: **Chess MCTS player. Left side shows current tree, right shows bot learning.**