CSC 380 Artificial Intelligence
Fall 2021
Final Report

Pipelayer Game with Artificial Intelligence Final Report

Sampling Heuristic

Members:

Summer Martin
Lalima Bhola
Dom Lamastra

Abstract

Most board games have been able to be automated using artificial intelligence agents. Automated games include Tic Tac Toe, Chess, Checkers, and Go. Pipelayer is an obscure, unpopular game and there has been no prior research conducted on how to automate this game. How should pipelayer, a game with no previous game playing studies, be implemented using artificial intelligence? In this project, we determine random and heuristic agents that effectively automate the game. The random agent acts as the baseline test while the heuristic agent is the focus of the data and our ultimate goal. We additionally collect data to check the significance of having the heuristic agent play against the random agent versus the heuristic agent playing against the heuristic agent.

Keywords: automated games, random agent, heuristic agent, artificial intelligence

1. Introduction/Background

Pipelayer is an obscure two-player game where each player creates lines using different shaped icons. The game is played with two grids of icons that are slightly offset from each other. One player utilizes the circles and another player utilizes the squares. The players take turns moving by connecting two of their respective icons. The icons must be adjacent from one another. A player can only connect icons that are adjacent horizontally or vertically. If the connecting icons are not directly across from one another the move is invalid. Furthermore, icons have to be of the same shape to allow for a line to be drawn between them. No move may cross over a previous move made by the other player, a player can only block the opponent's continuous line. The objective of Pipelayer is to create a continuous line from one end of the game board to the opposite side. The player with circles wins if they create a continuous line from the left of the game board to the right, while the player with the squares wins if they make a continuous line from top to bottom. Although the continuous line for each player needs to go from left to right and top to bottom respectively, each player does not have to start on any side in particular. Players may start anywhere on the board and may draw further lines in any valid position on the board thereafter. New lines do not necessarily have to connect to previous lines drawn. Pipelayer is generally a strategy game. How do you make a continuous line from one end of the board to the other while also preventing your opponent from doing the same on their end? That is the ensuing question. As of this project, there has been no attempt to automate Pipelayer in the past. Moreover, there has been no efforts made to create an artificially intelligent agent that can play the game at the same level or higher than a human being.

For the purpose of this project, we define heuristic agent as an agent that makes decisions based on either a set of rules or on what it learns from implemented methodology. The implementation of this heuristic agent is the overall goal of the project.

2. Problem Definition

The overall goal of this project is to create an artificially intelligent agent who will be able to optimally play Pipelayer. More specifically, based on the current state of the board this agent must find, using some method, the best, next line to draw in order to increase the agent's

chance of winning. Our goal is not to make it impossible to win against this agent, but to make an agent that makes intelligent decisions at a level equal to that of a human. It is not necessary for the agent to be unbeatable for this result to occur. Humans themselves, while often intelligent, and capable of making intelligent decisions, are not infallible. Further goals include creating a user interface for the game and allowing it to be manually playable by two users. Creating a random agent which, as its name implies, only makes random decisions about where to move, and adding capability for the users to play against the random and optimal heuristic artificial intelligence agent are additional goals to be considered and achieved.

Because Pipelayer is fully observable, the AI agent can learn or see all possible moves of the game. It will always be aware of the current game board state when the players make a move. The agent also knows the entirety of their surroundings in their finite environment. From this we can determine the best strategy in which to implement the artificial intelligence agent.

Our original hypothesis is that an agent that searches for moves optimally will never lose if it goes first, which should be true based on the nature of the Pipelayer. A further hypothesis is that an agent that uses a heuristic will perform better than an agent that makes random moves.

One problem comes from the lack of prior studies, research or implementation for Pipelayer. Because of this, we must also determine the most efficient way for the agent to make decisions. Some search algorithms or strategies which are commonly used to implement artificial intelligence game playing agents may not work in our case due to size constraints, time complexity or other factors.

3. Proposing AI Method/Heuristics/Implementation Details

3.1 Proposing AI Method and Heuristics

Our goal is to create an Artificial Intelligence agent which is capable of making intelligent decisions about which move to make next in the game. But how are we meant to accomplish this? For this discussion, line will refer to the continuous winning line from one end of the board to the other, or the progress towards that line, which is the goal state. Link will be used to refer to one move on the board; the connection of two circles or squares made by a player. It is also important to note that the functionality of the agent was implemented in a combination of multiple Javascript files and Python. Passing variables between these files is implemented using eel [4]. Further notes include that when playing against a human player, the agent always goes second. When playing a random agent against a heuristic agent, the random agent goes first. We started off by creating a random agent as a baseline. This agent has no functionality other than making random decisions about where to move next while continuing to follow the rules of Pipelayer. The purpose of this agent is to have a baseline to test against. Once the random agent was created, further consideration was needed for how the intelligent agent should be making decisions. Our proposed method is to use a sampling/ simulation strategy.

Before we consider the sampling method, the agent relies on a standard set of rules for its first two moves. These moves allow for some population of the board and allow our sampling strategy to work more efficiently as discussed in detail further on. Each of these rules created to determine these moves work to increase the chance of the heuristic agent winning. The agent's first move will be in relation to the other player's first move, regardless of what type of player this is. The rule works to determine where the other player drew their link and immediately block them to prevent them from continuing to create a straight line across the board and winning

quickly. The other player will now be required to go around the heuristic agent to win, automatically increasing the difficulty. The second rule pertains to the agent's second move and will most often extend the heuristic agent's overall line by adding a link to either end of the one it already created. This both increases the agent's reach across the board, getting it one step closer to winning and decreases the other player's chances by taking away a path it could utilize to win. If the agent is unable to add a link on either side of the one it previously created, it will instead start to go around or to the side of whatever is blocking it. This helps to increase the agent's chances by furthering it's progress across the board. Once these rules have been exhausted, the agent employs the implemented sampling method to determine further moves.

Using this sampling method, the first thing the agent does is take in the current board state. This means that every link on the board made by either player is copied into another place to be used later. All places where a link has not been created is also considered and recorded. This is done by sending all the links from our Javascript code to the Python file as they are created using eel [4]. The python file stores these links for future use in a board or 2D array created using numpy [2]. Once it is time to utilize the sampling strategy, this board is considered as the current board state as it mirrors the game board seen by the players. Behind the scenes, the program takes this current state and simulates continuing to play the game as many times as it can in 3 seconds. The minimum amount of simulations is around 500 games. As the actual game advances and more links are added onto the board, the number of simulated games increases. This is one purpose of including the opening moves or rules in the game. They allow for more samples to be generated once we start using the sampling method. As the game nears its end, up to 4000 samples may be used to determine the next link (Figure 1).

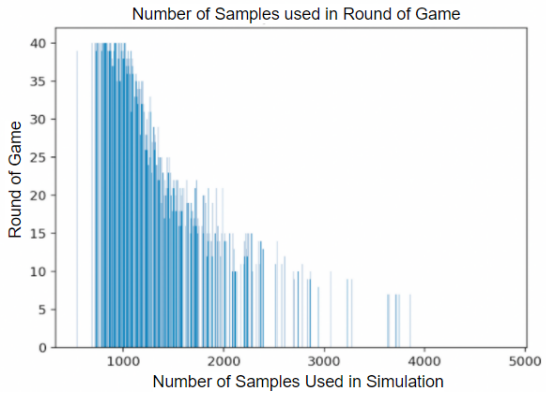


Figure 1: Shows the number of smaples used as game progresses Generated using matplotlib [3].

Before the first simulation, a blank board is generated to keep track of the weights for each possible link. All the weights start at 2. For each simulated game, the game is continued from the current state board until a winner occurs. After a winner occurs we have to then find the winning line of the game. To do this we used the library collections [5], and the class deque, which is a doubly ended queue, and allows us to find the winning line faster than a linked list or an array. Once the winning line (continuous line from one end to the other) of the game is found, the weights of the links included in the winning line are incremented in the separate game board. Each simulation may have a different winning line. However, there may be links that are found in two different winning lines whose weights are incremented each time. After all the simulations, we check the cumulative weights of every link and pick the link with the highest weight. That is the link that will be played in the actual game. This simulation process is used for each of the heuristic agent's moves after the first two and continues until a winner occurs in the actual game.

In the case of two heuristic agents playing each other. The rules for the first agent change a bit. The first move will be entirely random but the second and third move will then follow the rules as discussed previously. This is because, when the heuristic agent is not playing another

heuristic agent, it always goes second so it can base its first move off the move of the other player. When the heuristic agent goes first, it's first move should be random.

In addition to the sampling implementation, we have included functionality for both ensuring the move being made by the respective player is valid and within the rules of the game and functionality for continuously checking if there is a winner. To ensure the move being made is valid we use Javascript to check that no are not crossing an existing link, not redrawing a link that has been made already, and not trying to create a link between two circles or squares that is more than one horizontal or vertical position away. In the case of human players, there are also preventative measures against moves using the other players icon. As discussed previously, circles are used by player one to draw links and squares are used by player two. This function ensures that this cannot be bypassed or avoided.

To check if a winner has occurred, we use a function in python which then returns whether or not there is a winner to the javascript using eel [4]. This checkWinner function will continuously check the current game board state to see if any player has made it from one end of the board to the other in a continuous line. To prevent infinite loops in the case of cycles, we have functionality to prevent the method from checking more than four right or four left turns in a row, indicating a cycle. If the function finds that a winner has occurred, it passes back a value indicating who won. The game will then be stopped, winner screens shows and statistics about the game displayed if selected. These statistics include the number of moves each player made and how long the game took.

3.2 UI Implementation

Before creating artificial intelligence agents to automate Pipelayer, we designed a user interface that opens on a separate game window. The game screens and design were created using HTML and CSS; the functionality was implemented using Javascript and Python. In the manual game, Player 1 is represented by a purple line and connects circle icons together. Player 2 is represented by a yellow line and connects square icons together. The game board originated from a dots and boxes game and was heavily modified to form the offset circles and squares grid and the remaining functionality [1]. The home screen contains a simple game menu with the options of viewing instructions, playing against another human, playing against a random agent, or playing against a heuristic agent. The last two menu options are watching either the random and heuristic agents or heuristic agents play against each other. On each game screen, the left-hand side indicates the turn of each player, along with their respective colors and icons. There are also two buttons for restarting the game or returning to the game menu. The contrasting colors make it visually appealing to play and easy to differentiate between players. Upon winning, the screen either switches to a purple screen or yellow screen to indicate a win for Player 1 or 2 respectively. On the winning screen, the user has the option to view game statistics which contain number of moves for each player as well as time it took to complete the game. Game statistics are displayed by first outputting the data to a text file from Python, then taking the text files as input for an alert function in Javascript. The window pop-up to display the statistics is a convenient way for the user to navigate the game data. Then the player can return to the menu or manually exit the game by exiting out of the window.

4. Experimental Outcome

With our Heuristic agent created, the next step in our process is determining the efficiency of our agent by testing it against other agents. We decided to see how it would fair playing against itself. Using a script file, we simulated playing 500 games for both the random agent against the heuristic agent and the heuristic agent against itself (another heuristic agent). From the following results we can see that the heuristic agent always wins against the random agent proving part of our hypothesis (Figure 2). The possibility of the random agent winning against the heuristic agent is possible, however in that case, the random agent would have to randomly counter every heuristic move perfectly and the chances of that happening are near to zero. It is almost negligible and not worth considering. However, some interesting results can be seen from simulating the heuristic agent playing against another heuristic agent of the same make (essentially playing itself) (Figure 3). One might expect the results to show that each agent wins about 50% of the time as both are near optimal agents and play at the same level and with the same strategy. Additionally, in our hypothesis we stated that an agent that moves optimally would always win if it went first. However, from the given results, we can conclude that going first gives the first heuristic agent a slight advantage over the other agent but does not guarantee a win. The fact that the first agent would always either have one more or the same number of links on the game board as the other agent which may be the reason for the advantage. The reason the first agent occasionally loses may also be because it is not 100% optimal. The term optimal implies that it always chooses the best link for the current state, which has not been proven to be true for our agent.

Figure 2: Number of Wins Over 500 Games

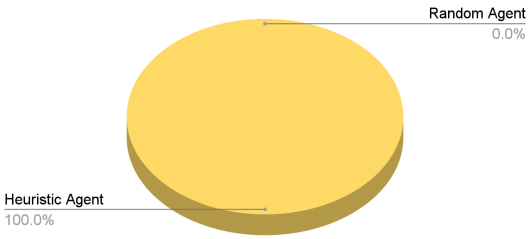
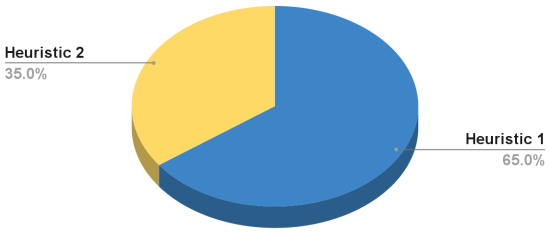


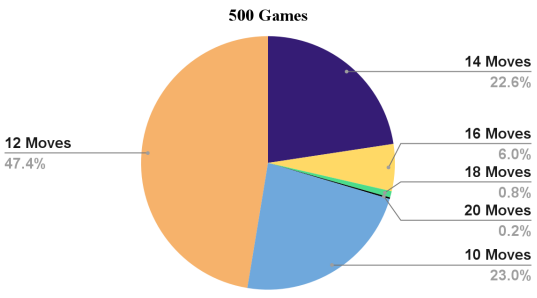
Figure 3: Number of Wins Over 500 Games



Another case that was not covered was the case of our heuristic agent playing against a human. These results were not explored as they would require a large N to find any conclusive evidence. From personal exploration, we can determine that the heuristic agent is not unbeatable, although it is a challenging opponent.

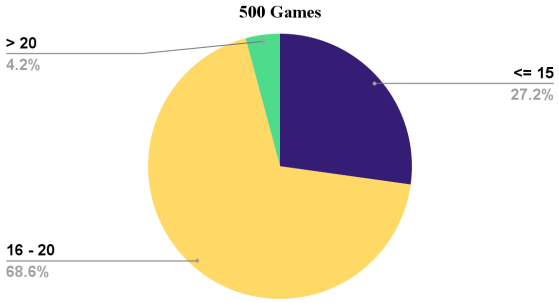
Additional information provided from our simulated games shows other interesting results about the length of the game and the cumulative number of moves made by the agents in a game. When playing a random agent against a heuristic agent, the games both take only a few moves and end quickly. The average amount of time it takes is about 16.04 second and the average sum of moves is around 12.296 moves. As you can see from Figure 4, over 70% of all games finish in 12 seconds or less. Less than 30% of games take more time than that (Figure 5). Furthermore, nearly 70% of all games take between 16 - 20 moves. This is the sum of moves made by both players. One reason for

Figure 4: Total Moves per Game
500 Games



these results is that the random agent spends negligible time making decisions because those decisions are uninformed. The heuristic agent will still take 3 seconds to make a decision because of the simulations required. These games are also quicker because there are fewer rounds. The heuristic agent can win quite quickly most of the time because it is rarely blocked by the random agent. The heuristic agent rarely needs 10 or more moves (20 or more moves for both agents combined) to win, as can be seen from our results.

Figure 5: Total Elapsed Time per Game (seconds)



As for the heuristic vs heuristic, we see contrasting results in comparison to the results seen previously from the random vs heuristic. Games played between two heuristic agents take significantly longer to play (Figure 7). There are several reasons for this. The first is that each agent is making intelligent decisions. They each take three seconds to simulate the games and choose the next link to play. Each agent is continuously blocking the other from winning while trying to win themselves. Because each agent prevents the other from winning too quickly, there are more rounds per game and, thus, more overall moves per game (Figure 6). This combination results in a significantly longer game. The average length of one game is approximately 96 seconds while the average sum of total moves is about 34 moves. The majority of games take more than 30 moves to complete, rarely does a game end with less. Because of this, over 50% of games take over 100 seconds to complete.

Figure 6: Total Moves per Game

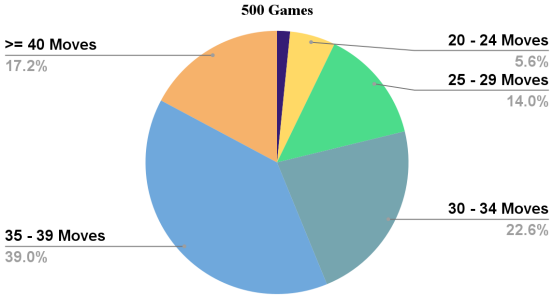
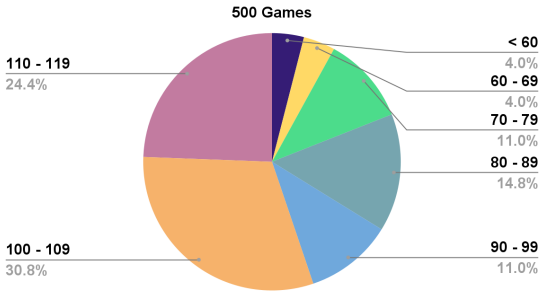


Figure 7: Total Elapsed Time per Game (seconds)



Between these two sets of results we can conclude that playing two intelligent agents against each other results in a significantly longer and more complex game than playing a random agent against one that makes intelligent decisions.

5. Discussion

The following section contains discussion regarding additional aspects of our project. Included is further discussion regarding our sampling/simulation agent. Also considered are the limitations of our project and how future works could improve upon our current implementation. We also briefly explore our reason for not using a methodology more commonly seen when creating game playing agents, such as using minimax to create a Tic Tac Toe agent. These topics are expanded upon and discussed in the following section.

5.1 Success of Sampling Agent

One function of an optimal agent is the ability to tell when it can win and make that move if possible. If our agent has the ability to finish the game and win on its next turn, during the simulated sample games the weight of that link should be high because it should come up most frequently in the winning links that have not been played yet. Consider the following board shown on the right (Figure 8). There are 14 possible moves left to make in the game where no other space is occupied, labeled accordingly. It can be seen that it is the second player's turn (the squares) since it has one less link on the board than player 1 (the circles). It can also be seen that there are two possibilities for the game at this current state. The first option is that the second player chooses a link that is not in space 14 and player one wins on the next turn. The second option is that the second player does move into space 14 and the circles have no choice but to go around to try and win. The second player could then try to choose the link in space 12 and then the link at space 2 for their next two turns but the circles may have blocked those locations by that time. The possibilities of both of these actions happening are 7% and 0.04% respectively.

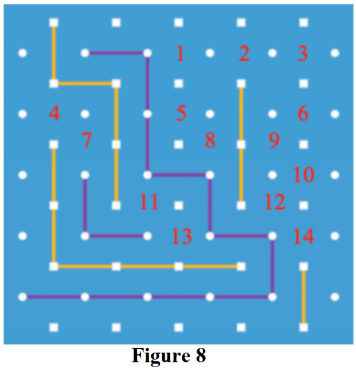


Figure 8

P(Option 1) = 13/14 possible actions * 1/13 possible actions = 7%

P(Option 2) = 1/14 possible actions * 12/13 possible actions * 1/12 possible actions * 11/12 possible actions * 1/11 possible actions = 0.04 %

The closer an agent is to winning, the more likely it will be for that goal state to be realized during simulation. The Law of Large Numbers states that as more samples are generated the sampled values or the weights for each line should be as we expected. That is, the weight of the link that would result in a winning game should be much higher than one that would not. If we generate over 2000 simulated games, we should expect to see option one fairly often since it has a higher chance of occurring compared to option two. However, we should see option two at least a few times.

5.2 Limitations/Future Work with Sampling Agent

Because much of the decision making relies on how many simulations the program can run in three seconds, this may cause potential issues or limitations. One of the limitations of our heuristic is based on the speed of the user's computer hardware. If it is slow, their system may not be able to generate as many sample simulations in the 3 seconds. Another issue is that at the very beginning of the game, with fewer moves on the game board, it takes a lot more steps in each simulation to finish a game. The allocated 3 seconds will not allow the agent to generate as many samples as would be optimal. The more rounds that have passed in the actual game, the better our agent should perform due to increased sample size and decreased link options. This is another reason to include some standard rules to populate the board before starting with the simulations. However, anything that affects the program's speed is a limitation. Future work can

continue to develop the heuristic more efficiently to avoid situations where optimal moves are less obvious. Further analysis can prove or disprove this algorithm.

As previously stated, the sampling agent has very strong results. Further improvements could be implemented using simulated annealing. Simulated annealing is where the likelihood of the agent playing a random move starts high and then exponentially decreases as the number of rounds in the game increases. Simulated annealing may have also had a positive effect in conjunction with our heuristic. We tested the heuristic with added simulated annealing on an agent that makes the first move but did not add simulated annealing to the opposing heuristic agent. The agent that moved first won 79.6% of the time. This is an improvement compared to the 65% winning rate shown in Figure 3. When the same test was run with the simulated annealing on the agent that went second, the agent's winning rate increased from 35% shown in Figure 3 to 38% shown in Figure 10. Further testing and analysis can find the significance of this although we can assume that when coupled with the advantage of going first, simulated annealing can increase the first agent's chances of winning. If this is deemed significant, the future work can involve optimizing the simulated annealing benefits of the agent, by modifying the temperature of the simulated annealing.

Figure 9: Simulated Annealing Agent Goes First

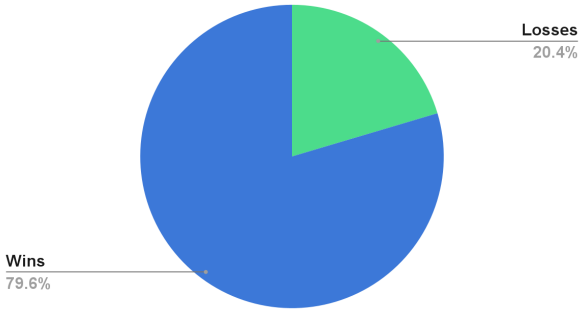
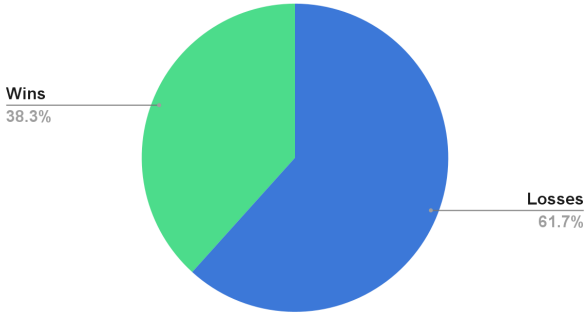


Figure 10: Simulated Annealing Agent Goes Second



5.3 Reasons Against Using Minimax with Alpha Beta Pruning as Heuristic Agent

Most adversarial search games use the Minimax algorithm with Alpha Beta Pruning and a useful heuristic function to make optimal decisions. With minimax, the agent makes its moves assuming the opposing agent is going to make optimal choices as well. Our agents did not use minimax. A Minimax agent in regards to this project was tested but the results were not ideal because the agent would occasionally lose against a random agent. While minimax is not being discredited, implementing minimax in our game has two specific problems.

First, the minimax algorithm needs a tree structure to check all the possible options to make an intelligent decision. Pipelayer has 41 possible links you could draw on the board, so any tree would be size 41! - X, where X is the number of different goal states. This number is too large for a computer to handle and would take too long to reasonably search every branch. The tree could be constructed dynamically, where nodes are added and deleted but this adds extra overhead. Even with alpha beta pruning, the tree does not see enough information to make an optimal decision.

Our heuristic would also not work in a tree structure because it relies on the repetitiveness of the samples. The closer the agent is to winning, the higher the weights of the links required to reach that goal state. An agent playing using the minimax algorithm with alpha

beta pruning would need a different heuristic function to calculate the weights of the links or states of the board then the one used by the sampling/simulation agent.

6. Conclusion

In this project, we stated two hypotheses regarding the performance of an agent that moves optimally and an agent that uses heuristics. The first hypothesis which states an agent that searches for moves optimally will never lose if it goes first was neither proven to be true nor false. While our agent is near optimal, it cannot be proven to be 100% optimal because we cannot, at this time, prove that it always makes the best decisions. However, when our near optimal faces another heuristic agent of the same make, it is not able to win each game. Furthermore, as seen in our previous discussion, even with added functionality such as simulated annealing to improve the agent, it still does not win every time. In our case, our goal was not to create an unbeatable agent but to create one that plays on the same level as a human. As humans cannot be said to always play 100% optimally, we can say that we accomplished our goal satisfactorily. Regarding the heuristic agents, the agents playing against each other are both using the same heuristic and have a chance to place strategic moves blocking the opponent. This means that neither player 1 nor player 2 can be guaranteed to win regardless of which player goes first. However, the data shows that since player 1 makes the first move in each game, it has a slight advantage over player 2; this is still not significant to claim a specific player will win. For our intended agent capability, this is more than sufficient. The second hypothesis states an agent that uses heuristics will perform better than an agent that makes random moves has found to be true. Due to the unpredictable and non-strategic moves made by the random agent, the heuristic has a straightforward chance of winning as a result of its own strategic moves. While the heuristic agent is almost guaranteed to win, there is an almost zero, yet existing, possibility that the random agent could win if it randomly made all the right moves for every move the heuristic agent made. However, this possibility is generally too small to consider and the chance of this occurrence is close to nothing. Overall, we can say that our heuristic agent performs at a level deemed more than satisfactory for our intended project.

List of References

[1] Davies, J., 2021. GitHub - *jwld/dots-and-boxes: Dots and boxes game made in HTML5 canvas.*. [online] GitHub. Available at: <https://github.com/jwld/dots-and-boxes> [Accessed 2021].

[2] Harris CR, Millman KJ, van der Walt SJ, Gommers R, Virtanen P, Cournapeau D, et al. Array programming with NumPy. Nature. 2020;585:357–62 [Accessed 2021].

[3] Hunter JD. Matplotlib: A 2D graphics environment. Computing in science & engineering. 2007;9(3):90–5 [Accessed 2021].

[4] Knott, C., 2020. GitHub - ChrisKnott/Eel: *A little Python library for making simple Electron-like HTML/JS GUI apps.* [online] GitHub. Available at: <https://github.com/ChrisKnott/Eel> [Accessed 2021].

[5] Docs.python.org. 2021. *collections — Container datatypes — Python 3.7.12 documentation.* [online] Available at: <https://docs.python.org/3.7/library/collections.html> [Accessed 2021].