Rebecca Andrews and Yina Wang
May 12th 2019

Sorry! Project Write Up

The basic premise for this project is to construct a two player version of the Hasbro game "Sorry!" in a similar manner to the chess game created by Professor Hunsberger, including the addition of minimax and alpha-beta pruning. This game involves chance, so it would require us to create a stochastic version of minimax, as described in our book (Russel & Norvig, 2016).

The primary goal of this project would be to further explore adversarial search algorithms by creating a minimax algorithm that is stochastic and incorporates details of the game Sorry!, and also by producing a useful version of a evaluation function for this particular game. Both of these goals will help us improve our understanding of adversarial gameplay and how it becomes more complicated with chance. In addition, our secondary goal would be to create a version of this game that is actually fun to play by yourself against a computer player. To our knowledge, no one has ever made a version of Hasbro's Sorry that can be played in this manner.

Overview of Sorry

The game of Sorry! involves moving pieces around a square board based on the selected card, where each card has some probability of being pulled from the deck. An example board is depicted in Figure 1.

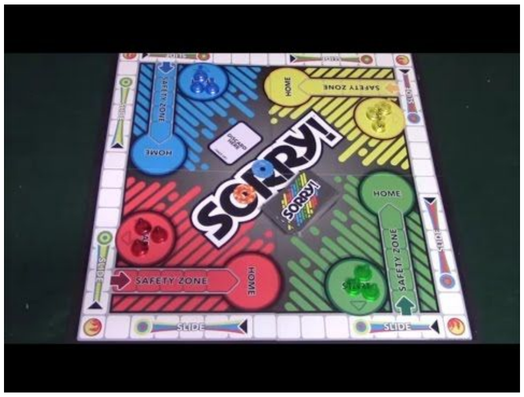
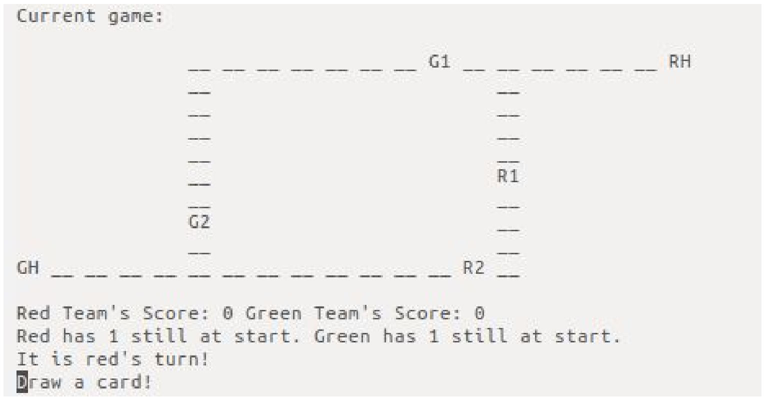


Figure 1. Example Sorry! board game.

There are 11 kinds of cards: the sorry, 1, 2, 3, 4, 5, 7, 8, 10, 11, and 12. In the original game, every card has an equal probability of being selected except for the 1 card, which is slightly more likely. For any card, it may be possible to move any one (or several) of the pieces. Most of the cards allow you to move the number of spaces forward (clockwise) that is listed on the card, but a few of the cards (such as the Sorry) have different rules. You also have the ability to send opponent pieces back to the starting place by landing on their piece. The goal of this game is to move all your pieces into the “home” space (the circle at the end of the path inside the main board) before your opponent. In many ways, this game resembles the games Parcheesi or Ludo.

To create our implementation, we simplified game play by removing some of the cards and just playing with the 1, 5, 8, 10, and Sorry. This was to help keep the branching factor to a more tractable size so that the minimax algorithm could still work reasonably quickly. We did however, leave in both uses of the 10 card (a -1 and a 10). We also decided to use an infinite deck to help minimize the additional complexity of our implementation. Finally, we choose to make the Sorry! game 2 player so that it would be compatible with minimax. An example board is shown below (Fig. 2).

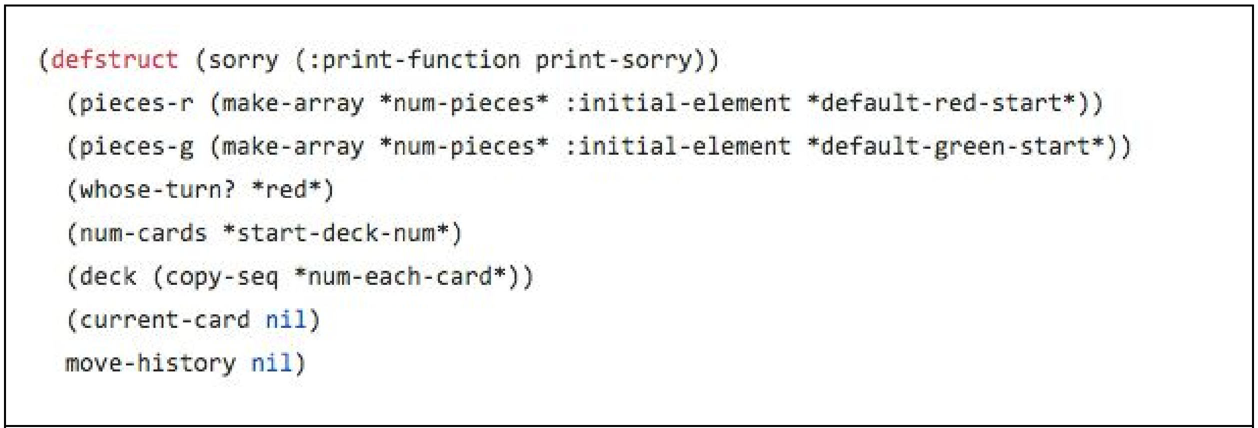


```
Current game:

               __ __ __ __ __ __ __ G1 __ __ __ __ __ __ RH
               __                         __
               __                         __
               __                         __
               __                         __
               __                         R1
               __                         __
               G2                         __
               __                         __
GH __ __ __ __ __ __ __ __ __ __ __ __ R2 __

Red Team's Score: 0 Green Team's Score: 0
Red has 1 still at start. Green has 1 still at start.
It is red's turn!
Draw a card!
```

Figure 2. An example Sorry! game in our implementation. RH represents the home for the red team and GH represents the green team home.

Game Representation and Legal Moves

To represent each game state, we kept track of the current location of each of the pieces for each player in two vectors, the current turn (represented by a number), the number of cards in the deck, the number of cards remaining for each type of card in a vector, the current card (represented by a number) and a list of the move history. The structure is listed below (Figure 3). The number of cards and the deck were included as fields in the structure to make it easier for us to extend to a finite deck, but ultimately we did not get to this during this semester.

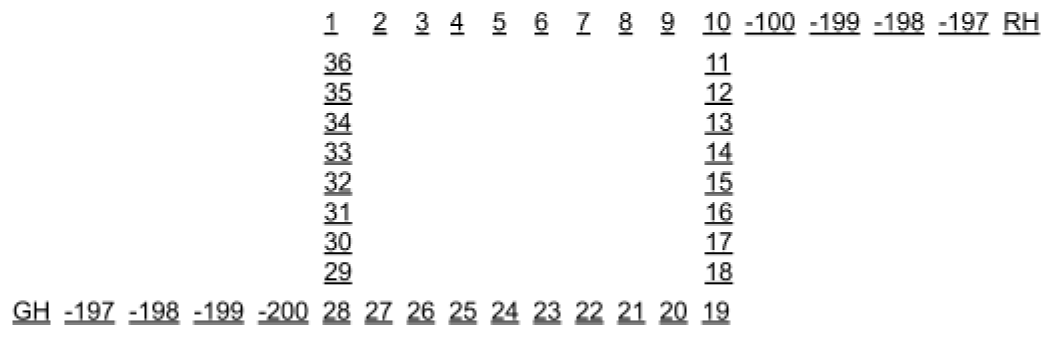


```
(defstruct (sorry (:print-function print-sorry))
  (pieces-r (make-array *num-pieces* :initial-element *default-red-start*))
  (pieces-g (make-array *num-pieces* :initial-element *default-green-start*))
  (whose-turn? *red*)
  (num-cards *start-deck-num*)
  (deck (copy-seq *num-each-card*))
  (current-card nil)
  move-history nil)
```

Figure 3. The sorry structure used in our implementation.

We choose to represent the game in this manner because unlike in the chess or go implementations, there are very few pieces that need to have their locations recorded. Thus it seemed to make sense to explicitly just record the location of each piece rather than to more fully represent the board and record if there is a piece there or not. This made it easy to generate the possible moves because each card can be applied to each piece to generate a possible move rather than having to search an array or use a more complicated representation to generate legal moves.

Legal moves were generated simply by applying the given card to each piece in turn for the current player. However, special care was taken to prevent duplicate moves from appearing when multiple pieces were at the start for a given player. For example, only one legal move is generated to move a piece from the start to the board even if there are a few pieces at start because there is really no reason to differentiate between which piece is placed on the board as long as it is from the same starting point.



```
                            1  2  3  4  5  6  7  8  9  10 -100 -199 -198 -197 RH
                            36                          11
                            35                          12
                            34                          13
                            33                          14
                            32                          15
                            31                          16
                            30                          17
                            29                          18
GH -197 -198 -199 -200      28 27 26 25 24 23 22 21 20  19
```

Figure 4. Numbered board to show you what is underlying the generation of legal moves.

The spots on the board are numbered in the way demonstrated above (Fig 4). Essentially, the spots are numbered starting from 1 and moving clockwise up to the 36, except for the safe zones for each player which have their own numbering, starting from either -100 (red) or -200 (green) until their home.

<u>Using the Expectiminimax Algorithm</u>

We used expectiminimax as our chosen game playing algorithm (Fig. 5). In expectiminimax, there are chance nodes in addition to min and max nodes. Just like each branch from each max or min node represents a possible action, each branch from a chance node represents a possible random outcome. For our game, each random outcome is the selection of the card. This meant that in our implementation, each chance node had 5 branches representing each of the five possible cards.

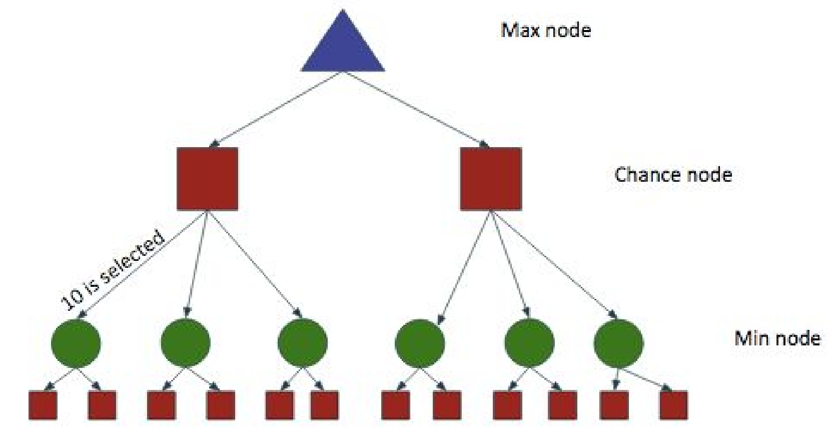


Figure 5. An example of the expectiminimax algorithm.

At each chance node, the average value of all the random actions was calculated and passed up to the parent. This average was calculated by summing up the resulting values of each child of the chance node weighted by the probability of selecting the card that produced that outcome.

We incorporated alpha-beta pruning into our minimax algorithm, but we chose not to prune the lower probability actions because we are using an infinite deck in which the probability of a certain card getting chosen does not change. Had we had time to use a finite deck, we would have utilized this and we set up our minimax algorithm to be able to easily incorporate this.

In our particular game, there were also many instances were only a single move is possible for a particular player. In these instances, we did not compute the root node value and instead simply return the only available move. When this happens, we print "ONLY ONE MOVE SO NO NEED TO EXPLORE" after root node value to indicate this to the user.

Evaluation Functions

All of our evaluation functions included the following features: number of your own pieces in the home, number of pieces in your opponent's home, the number of your own pieces on the board, number of your opponent's pieces on the board, how close you are to your home, and how close your opponent is to their home.

In our initial exploration of this topic, we found that there has been extensive research about strategies for the game "Ludo," which is very similar to Sorry. For this reason, we created our evaluation functions primarily based on this research, though we could not directly utilize the same strategies because of the low level differences between the games and the fact that the research we did find did not utilize any form of minimax (Alvi & Ahmed, 2011; Chhabra, & Tomar, 2015). These paper primarily highlighted the difference between defensive or offensive strategies, i.e. strategies that highly value protecting your own pieces from being sent to start versus strategies that highly value sending your opponent's pieces to start.

We implemented these same kind of strategies as well by highly valuing features of the game in ways that correspond to the strategies. For the defensive strategy, we highly valued keeping your own pieces on the board and getting your pieces into the home base. For the offensive strategy, we highly valued preventing your opponent from leaving pieces on the board and preventing your opponent from getting pieces close to their home base. We also created a baseline strategy that fell between the two that we used to help build the others around. This strategy was constructed by modifying a randomly created evaluation function until it could win significantly above chance against the random player. In addition, we also created a strategy that valued getting your own pieces close to the home more than anything else, as this is a common strategy for people playing Sorry! The values used are shown below in Table 1.

Table 1. Comparison of Evaluation Functions

Strategy name	Home-pt	Op-pt	On-board-pt	Op-on-board-pt	My-home-close-pt	my-op-close-pt
Default	700	-600	50	-20	50	-50
Defensive	2000	-600	500	-20	50	-50
Offensive	700	-600	50	-200	50	-300
Runner	700	-600	50	-20	500	-50

Testing

In order to test our implementation, we made several different comparisons between different variations of our algorithm. Firstly, we played our algorithm using depths 4 and 6 against an algorithm that simply selected a random move on each turn. We ran these tests multiple times to account for the random selection of cards throughout the game that could make it difficult for even a very good algorithm to win every time. We ran the algorithm at depth four against the random opponent 100 times and we ran the algorithm at depth six 40 times (this took somewhat longer to run). We also did 40 runs of depth 4 versus depth 6. We felt this was the most appropriate comparison because depth 2 is really nearly the same as random and depth 8 was too slow to really run many times against another depth.

We also tested our various evaluation functions against each other at depth 4. Depth 4 was used so that we could run many tests, and it did not seem like there was a tremendous difference in performance in just a two depth increase, as we will mention in the results. Given that we had 4 evaluation functions (including the default), we ran 6 different tests comparing each evaluation combination. We ran each of these tests 100 times.

Finally, we did some manual testing where we simply played against the algorithm ourselves a few times at different depths and using different evaluation functions. We only played a few games of each because it takes a reasonable amount of time to play through a full game.

Results and Discussion

Our algorithm easily outperformed the random player at both depths 4 and 6. At depth 4, the player using expectiminimax with alpha beta pruning won 71% of the games played (71 out of 100), while at depth 6, the player using expectiminimax with alpha beta pruning won 85% of the games played (34 out of 40). This seems to suggest that our algorithm is certainly superior to randomly deciding moves as at both depths the winning percentage was way above chance, and it was particularly promising that the advantage seemed to increase as the depth increased.

However, we found that our algorithm did not differentiate between depths 4 and 6 very much. The player using depth 6 won 57% of the games played (17 out of 30), indicating that the two depths are relatively even. Perhaps at a greater depth there would be a larger differentiation, but we felt that we could not do enough runs of depth 8 to compare it very well with depth 4.

The evaluation functions were also relatively even when played against each other. The default performed evenly against all evaluations functions (won 50% of games versus offensive, won 45% of games versus defensive, and won 46% of games versus runner). This was the case even though the offensive, defensive, and runner evaluation functions were quite different from the default. The offensive and defensive evaluation functions were also relatively even against each other, with the defensive winning 44% of games. The runner, however, did not fare as well against the defensive and offensive strategies. The runner won only 36% of games against the offensive player and 31% of games against the defensive player, indicating that it seems to lose more than win against these two strategies.

When we played manually against the minimax algorithm, we found that it was relatively difficult to defeat the algorithm. For example, when Rebecca was playing against it, she often did relatively well at first and then struggled when the opponent got Sorry! Cards, even when playing just at depth 4. She believed that this was likely to do the fact that because Sorry! cards are not common, she does not play the game counting on selecting this card, when in fact, this can really sway the game in your favor (or out of your favor) if done correctly.

There were certainly moments where the algorithm selected moves that seemed somewhat counterintuitive at first, yet given the advantage the game seemed to have over us when we played, perhaps we just do not recognize what moves most often increase the chance of victory.

Overall Summary

Overall, we were relatively successful in creating an algorithm that could play Sorry! reasonably effectively. One of the biggest challenges we had was determining what our evaluation function should look like because there are so many features of the game that we could account for. In our research, we discovered that evaluation functions for games of chance are particularly challenging, and often they are created based on information about the actual probability of winning the game from a particular state (Klein, 2010).

For us in particular, it was difficult to deal with the fact that a state that looks good currently could turn out to be quite bad if the other user selects a particular card (such as a Sorry!, which was often game-changing). We even first created a baseline evaluation function that ran quite well against a random opponent but poorly when the opponent was using an informed algorithm at a lower depth. This forced us to go back and improve our evaluation function so that it would also work with a player who was somewhat informed in their decision making. We were, however, able to do so.

It was also somewhat difficult to deal with the fact that the random actions could really affect the course of the game. There were many times in our testing phase that the player we expected to win was doing quite well, only for a few unlucky moves to turn the tide. This was very much in contrast to the Othello or chess testing we did where we could be confident that the better player would always win, as even when a certain move is likely to produce a good outcome, an unlikely next random outcome could make this a poor choice. This was particularly difficult because the wide range of cards in the game: there is a big difference between selecting a 1 and a 10 in terms of how much progress you can make towards winning. We did our best to manage this, and I think our relative success can be seen in the ability of the algorithm to beat the random player.

Read Me (Copied from Read Me that is in Our Project Folder)

The basic premise for this project is to construct a two player version of the Hasbro game "Sorry!" in a similar manner to the chess game created by Professor Hunsberger, including the addition of minimax and alpha-beta pruning. The game of Sorry! involves moving pieces around a board based on the selected card, where each card has some probability of being pulled from the deck. For any card, it may be possible to move any one (or several) of the pieces. The goal of this game is to move all your pieces into the "home" space before your opponent, while also have the ability to send opponent pieces back to the starting space along the way. This game involves chance, so we modified minimax to include that.

You can play either with someone else or against a random computer using our AI that should suggest the best moves possible.

Instructions

To fire this up, you should open up a new buffer in ACLEMACS within the folder that our project
is in. To do this, type **aclemacs** into the terminal. Then **ESC + X**, then type **fi:common-lisp** and **ENTER** seven times to open up lisp in emacs.

In order to run our specific project, type **(load "basic-defns")**, and run **(maker)** which will allow you to compile and load all of the relevant files for our implementation of sorry.

Type **(setf g (make-sorry))** to create a new game.

To play manually against yourself

Type **(draw-card g)** to draw a card to start your turn. The only way to get out of the starting position is to draw a 1 card, so you need to pass.

Type **(pass g)** to pass if you have no available moves.

Type **(play-card g index &optional secondary)** to play your card on a particular piece (specify the piece using index).

When the card is the Sorry, the index is used to specify which piece belonging to the other player that you want to send to start.

You can use the optional index to specify the secondary use of the card if you have a 10 and you want to use it as a -1. To do this, just type -1 in the parameter spot for secondary. Example: *(play-card g 0 -1)*

Important note: When you specify an index to apply a card to, if you select a piece that is at the start, the piece at start with the lowest number will be selected (none of these are distinguishable anyway). This is only relevant when applying a 1 to a piece at the start.

Get some help playing your move

Type **(suggest-best-move g depth &optional eval-choice)** with a depth value (can do any depth value 8 or below, but 6 and 8 can be a little slow) and an optional numeric value for your choice of evaluation function. If no value is specified, the default is used. 1 indicates the offensive strategy, 2 the defensive, and 3 the runner. This will print out the best move and how to use it above the current game using the play-card function.

Type **(do-best-move g depth &optional eval-choice)** in order to just do the best move that would have been suggested in the same way as using suggest-best-move.

Play Against the AI

You can select your own moves manually, but for the other player, you will always have them use **(do-best-move g depth &optional eval-choice)** as shown above.

Competing at multiple depths or with multiple evaluation functions

For playing with another player, type **(compete red-depth green-depth g)** in which *red-depth* is the depth at which you want the search to go for the red player, *green-depth* is the same but for the green player, and *g* is your Sorry! game.

If you want to play against a random player, use **(compete-vs-random depth g)** in which *depth* is the depth at which you want to conclude the search and *g* is your Sorry! game.

If you want to play two evaluation functions against each other, use **(compete-diff-eval eval-red eval-green g)** in which *eval-red* is the evaluation function used by red, *eval-green* is the evaluation function used by green and *g* is your Sorry! game.

This should print out the **Root Node Value**, **Number of Moves done**, **Number of Nodes Pruned**, **Best Move**, and **Card Chosen.**

Citations

Alvi, F., & Ahmed, M. (2011). Complexity analysis and playing strategies for Ludo and its variant race games. *2011 IEEE Conference on Computational Intelligence and Games (CIG11)*. doi:10.1109/cig.2011.6031999

Chhabra, V. & Tomar, K. (2015). Artificial intelligence: Game techniques Ludo - A case study. *Advances in Computer Science and Information Technology*, *2*(6) pp. 549-553.

Klein, D. (2010). Expectiminimax Search [Powerpoint slide]. Retrieved from https://inst.eecs.berkeley.edu/~cs188/fa10/slides/FA10%20cs188%20lecture%207%20--%20expectimax%20search%20(2PP).pdf

Russell, S. J., & Norvig, P. (2016). Stochastic games In *Artificial intelligence: A modern approach*. (pp. 177-180). Upper Saddle River: Pearson.