Durak Solver Agent

By: Drew Blackman

My project was to create an agent that could play the game Durak or the fool. I created a simplified version of it, as the game can be very comples with the amount of added rules there are. During the creation of this project, I still had a lot of difficulties with creating an AI as there as the nature of Durak and the version that I made made it difficult to create something I was 100 percent happy with.

I will be referring to a lot of types of cards in this paper, a cards face being what is displayed on it, in this exaple being 6, 7, 8, 9, 10, jack, queen, king, ace higher numbers being more important than lower numbers. Suits are spades, diamons, hearts, and clubs.

Game Rules

Durak is a Russian card game that is typically played with 2-6 players. The Deck typically consists of 36 cards for faster games. These cards are 6 to king and Ace of a normal playing card deck. This rule speeds up the game, but you can play with more if needed. The closest American card game I can relate it to is war, but with a lot of extra rules. To start, each person is delt 6 cards to their hand and the suit of the card on top of the deck is called the “trump suit” and placed at the bottom of the deck. Each round goes counter clockwise, the starting player being the one with the lowest card of the trump suit (so if the trump suit is spades, the player with the lowest spades goes first). This person is the first attacker. On the attackers turn, he places cards of the same face down against the defender. The defender then has to play cards that are higher than the attackers, they also have to be either the same suit or the trump suit. For example, let’s say the trump suit is hearts, the attacker places a 7 of spades and 7 of diamonds on the defender. The defender has to either place an 8 of spades or higher on the first one, an 8 of diamonds or higher on the second one, or any card that is a hearts. If the defender successfully defends, then all played cards are discarded and the defender becomes the next attacker. If the defender cannot defend or does not want to, he can pick up. This adds the attackers cards to his hands and his turn is skipped. At the end of an engagement, everyone draws up to six cards. The point of this game is to run out of cards and not be the last one out, the last person out is the only one who loses and is deemed ‘durak’ or the fool.

Difficulties

The main difficulties I had with this game come from there being unknown knowledge at all times, the rotating roles, and the multiple players.

As is typical of a card game, everyone has their hand hidden from the others. The only way to gleam what cards others can have is by remembering what has been discarded and what cards each player has picked up. Meaning I knew what cards are played and how many a player has, but not exactly which cards they had.

With there being two types of moves a player can have, it made it difficult to keep track of which player was playing what and generating possibilies for further game states.

Most of the algorithms and problems that we were used to only had 2 players, and because of the multiple moves everyone could make, it made it difficult to keep track of peoples hands, moves and give a proper score to each move based on the probabilities.

Creating the engine

I ended up creating my own simplified version of the game. It took in a max of 4 players and resolved the games quickly when there was no human player.

To make it easier to select move and create agents, I made two functions called get_valid_attack_moves and get_valid_defense_moves. These were to mimic a few of the labs by allowing the agents to pick from a list of moves, instead of generating them by themselves. This was the hardest part of implementing the game.

Valid attacks require you to play cards that are of the same suit, but you can play any amount

of them. If an attackers has four king cards, then they can play anywhere between one and four of those cards. The exception to this is they are capped by the amount of cards the defender has, so if the defender only has 2 cards in their hand left, the attacker can only place down 2 cards. This meant there were at least 6 moves an attacker could do, up to around 12 if they only had 6 cards in their hand. Generating all these moves was difficult for me to do because it included and unknown amount of unknown length arrays that had to be generated.

Valid defenses were more difficult to generate because they have more options. The highest card in durak is the Ace of the trump suit, so it can beat any card. If an attacker attacks with four cards then that Ace of trump suit can be placed on any of those cards to defeat it and can only be in one of those places. The order that cards are used is important, because if you can't defend against one of those cards the only option left is to pick up. I did this by sorting into arrays all the cards that could defend against each cards, then create every combination between those arrays.

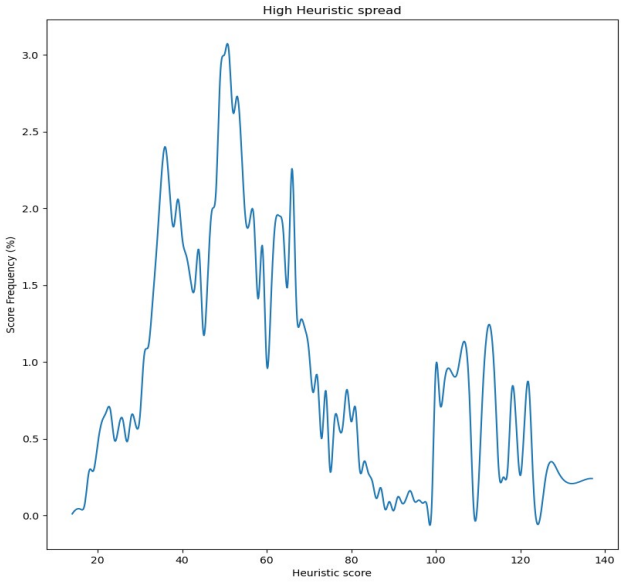

Figure 1

I did get those functions working and it helped a lot with creating the agents, one all the bugs were gone. The rest of the game was simple to make and mostly prompted for moves and updated the rest of the players as to which cards each player played and what actions they took during their turn.

Heuristics

The goal of Durak is to get rid of all your cards and not be the last person out. I thought it would be easier to focus on trying to be out first instead of not being last. There is also the strategy of keeping the best cards, if you get rid of the lower cards it becomes easier to attack and defend against players later in the game. You also want to have multiple cards of the same face in your hand, as attacking with four 10's is better than attacking with one as there are more cards to defend against. I took these option in consideration when creating the different heuristics.

Get_high_heuristics is the name I called my first heuristic idea. The idea was to gage the score of the cards that a player had in their hands, the higher the score the better (I named it after trying to get the highest score). I gave weights to number of cards, number of trump cards, and smaller weights to higher cards. I gave the amount of cards the weight of 100 / number of cards. Meaning the less cards you had the better and gave 0 cards, or a winning hand, the weight of 1000. I then added 15 score for every card a hand had of the trump suit. Each face was given a score in the importance of the card. The 6 face card was given 0 points, and the Ace was given 8. I made higher cards more important by rasing them to the power of 1.5, as I thought this

would given them a nice curve in importance. I multiplied each of these by the amount of instances there were in your hand, I then take the average score of your hand. If my hand consisted of three 8 face cards without a trump suit, I would do 100 / 3 = 33.3 plus 0 plus (2 ^ 1.5) * 3 / 3. With a total score of 36.13.

My goal for this heuristic was to get a right skewed distribution. The average hand would be around the lower end of the spectrum and the higher hands would have more differences between them. I took 10,000 different hands and got their scores and plotted them, shown in figure 1. This was the closest I could get to my goal, the average being around 50 points and there being a large dip around 80 to 100 with it picking back up.

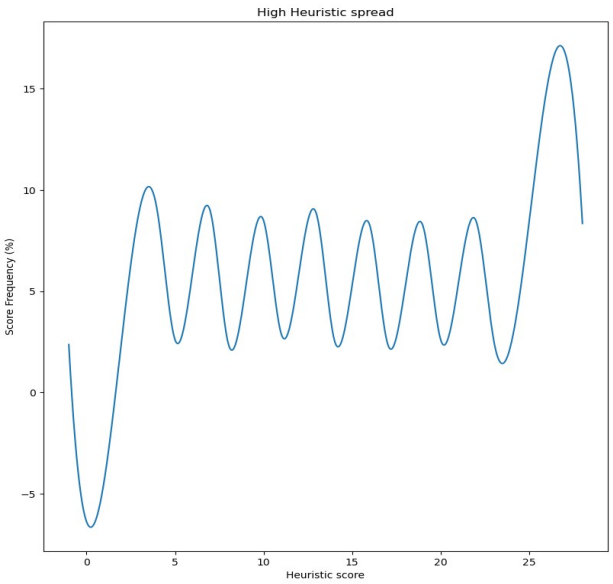

Figure 2

Get_low_heuristic was the pairing to get_high_heuristic. The goal was similar, but I thought it would be easier to weigh lower hand counts with high card numbers. The Ace of trump would have a weight of 1 and 6 would have a higher weight. Negative numbers would have the absolute value taken of them and compared. I never implemented this one because it seemed redundant and didn't have much improvements over get_high_heuristic.

Get_play_heuristic focused on the cards being played, not the cards still in your hand. This would be used to find if it was a good attack or good defense. These weights could be negative, intending 0 to be if the action was picking up cards. I wanted to focus on getting rid of the worse cards first and as many of them as possible. I did something similar to get_high_heuristic where I gave a score to every hard and then took the average. This time I took the amount of cards and squared it and added it to the heuristic. I changed the weight so the smaller cards were weighted more, 6 had a weight of 8 and Ace had a weight of 0. I cubed these results and took the average of the cards being played. I then subtraced 5 for every trump suit that was being played in that hand.

I didn't quite hit my goal with the distribution, I wish there were more cards that were weighted negative. Currenlty it is very rare that the best weighted action is to pick up cards, even though that is sometimes a really good action to collect more pairs. I also did the same tests to see what the distribution was and you can see it in figure 2. The way it is set up is kinda deceiving. For this heuristic, we want the average score to be higher because it mean more bad cards are getting used first and it's keeping more of the good cards.

AI Agents

There were two types of agents that I created, naive and algorithmic. The naive group only looked at the current hand and current play. It tried to take the best move for the current hand according to its heuristic. Algorithmic attempted

to search through different states of the game and select the best outcome.

Each agent has two functions that create its decisions. One is get_attack_move and get_defense_move. These do as their names suggest and get their respective moves as needed. They also use get_valid_attack_moves and get_valid_defense_moves respectively to get their move options to process.

Naive

Random is not really a good agent. It was the first one I made after finishing the game. It just picks at randomly and was more used as a proof of concept and a debugger for the game.

Naive Hand was my first attempt at creating a working agent that chooses its move in an intelligent way. This agent focuses on picking the move that will create the next best hand. It takes the generated list of moves and creates copies of each hand as if it were to play that move. It then used the get_high_heuristic to pick which hand has the best cards and returns that option.

Naive Move is my second attempt at an agent. Instead of using get_high_heuristic for each hand, it uses get_play_heuristic to find which cards it should play. It actually runs a lot faster than Naive Hand. Move only has to generate each possible move and then rank them, Hand has to create a copy of every possible hand from that point, making it slower from all the copying it has to do.

Algorithmic

Ab was the only arithmetic agent I tried to make and I was not able to complete the agent in the end, or make it not crash. The first problem that I ran into was multiple players. Trying to keep each players move made it so I had to have a depth limit of at least 4 to evaluate a full turn, which made it diffucult to find it useful. My solution to this was to cap the amount of players to two when using ab.

The next problem was creating each play state. I was trying to keep track of all the cards that the agent had seen in play then remove any cards that were discarded and remember any cards picked up by the other players. I think this was a good approach, but I couldn't translate that to generating all the possible hand combinations that an attacker or defender could have. I also had a hard time having nested games, because the agents have two types of moves, I had to keep track of who was doing what type of move, then resolve the next iteration to see how they would defend, if the defender had to pick up then would skip their next turn. I then tried to kind of cheat, instead of generating each possible move I would hand the agent a copy of the deck at the beginning of the game, so it would be able to know exactly which cards each player had and would draw in every circumstance. This made it simpler, but didn't solve all my issues.

I think that ab was the wrong approach. I was trying to fit it into a framework that didn't work for this game and that gave me a lot of problems and should have done more research to find other approaches.

Testing

I wanted to test each agent against the other agents in a 1v1 to see how they fared against each other. My methodology was to create one of each agent type, then shuffle their position to account for first player bias. I ran each group 10,000 times because the games go by fast and to remove random card draws. I also ran two Random actors against each other with shuffling and not shuffling to see if there was a first player bias. Here are the results:

Random vs NaiveHand

Name	Wins in %
Random	54.42%
NaiveHand	45.56%

Random vs NaiveMove

Name	Wins in %
Random	10.97%
NaiveMove	89%

NaiveHand vs NaiveMove

Name	Wins in %
NaiveHand	8.88%
NaiveMove	91.1%

Random vs Random (without shuffling)

Name	Wins in %
Random 1	50.4%
Random 2	49.58%

Random vs Random (with shuffling)

Name	Wins in %
Random 1	49.76%
Random 2	50.22%

Discussion

I think the most interesting part information is is how bad the NaiveHand actor did. It performed worse than random when playing against the random agent and comparing it to how it player against the NaiveMove. I do find it interesting because intuitively I though that trying to optimize for the cards in your hand would be better than trying to optimize for losing lower cards. I'm not sure if this is because get_high_heuristic was optimizing for the wrong things or because trying to keep use the cards you have as a way to gauge best moves is a bad idea. I also think it is interesting that there doesn't seem to be a first player bias because the difference in the random changes is only ~0.2.

When playing against these agents myself, I was able to notice the difference as the NaiveMove agent did beat me a few times, which really surprised me because of how simple the agent actually is and I wasn't expecting it to win.

What I would have changed

I think there were a few things that made it difficult to expand to other agents and contributed to my agents being so simple.

The first is my game wasn't very modular. The game loop itself didn't allow me to use it when creating the ab agent. That forced me to try and implement it in two different places and it would have allowed me to validate each play update with the logic of the actual game instead of monkey patching it in different ways.

Second I should have looked for a better approach instead of the ab agent. I made the mistake of it sounded good in my head and was excited from how we used it in our class projects, but was unable to get anything out of it.

Time Spent

Research: 5 hours

Game Creation: 2 hours

Getting Best Moves: 6 hours

Heuristics: 2.5 hours

NaiveMove agent: 2 hours

NaiveHand agent: 1 hour

Ab agent: 7 hours

Testing: 1 hour

Total ~26.5 hours

Code

The code can be found at https://github.com/eagledb14/durak-solver, the game can be started by running ‘python main.py’ and adding which agents you want to use. It even includes an agent template called ‘blank_player.py’ that should make it easy to create new agents, as it has all the functions that need to be implemented.