Can multiple heuristic measures outperform a single heuristic measure in grading a game state of Schnapsen effectively and accurately?

Harsh Khandelwal(hkl250) – Arian Babori(hbi450) – Eustatiu Dima(eda440)

Contents:

1. Abstract

The amount of different heuristics that can be implemented in Schnapsen or any game in general is plenty, and is only limited by the extent of the logic and rules of the game. Even in a simple card game such as Schnapsen the amount of game states that can be reached is estimated to be around a quintillion, even though a round consists of just twenty cards and ten moves. Hence it's important to have a good heuristic to accurately pick the best possible option of the vast amount available. In this paper we will discuss the properties of heuristics used in the different standard bots we worked with in this course of our own designed heuristics. We will experiment with different combinations of heuristics to discover why certain options work better than others and finally answer the question: what makes a good heuristic?

2. Introduction

When creating an agent to play the game of Schnapsen, there are many criteria that can be implemented to base each next play or decision off of. Both in machine learned implementations and agents based on human insights, heuristic values are evaluated from progressing game states to determine the next best move to play. In this paper we compare the heuristic methods in the different logic based default bots of the course, specifically rdeep and alphabeta, with some of our own designed heuristics to first find out whether the use of different heuristic methods has influence on the overall performance of the bot and next whether combinations of different heuristics provide improvements in performance compared to a single heuristic evaluation. We will start off with an explanation into the topic of our research and discuss the setup of our experiment. We performed experiments using a certain weight of each of the heuristic functions and observed that it improves the performance of the rdeep bot and outperforms the native rand, bully and rdeep.

3. Background Information

a. Schnapsen – Game Description

Schnapsen is a point trick taking card game for 2 players. It is an Austrian variant of the card game 66. Played using 20 cards from a standard card deck, Schnapsen uses the Ace, 10s, Kings, Queens, and Jacks of all four suits. The game is played in rounds. Each round players are dealt five cards, replenishing from stock after each trick, with one turned up to indicate the distinguished trump suit. Each round is split up into 2 phases. 1st phase indicates the 'liberal' phase where a player is free to play any card he wishes. 2nd phase indicates the 'restricted' phase where a player may only play the same suit as his opponent. The game starts in 1st phase and the 2nd phase is reached wither when the stock is closed or empty.

The goal in each deal is to accumulate sixty-six trick points, where the card values are A=11, T=10, K=4, Q=3, J=2. 20 points can be won by declaring a marriage, where a marriage is having a king and a queen of the same suit in hand. Depending on the margin by which a player wins the round, that player scores 1-3 game points. A player who reaches 7 points wins the whole game. A better overview of the rules of the game can be found here: https://www.pagat.com/marriage/schnaps.html. As can be seen, Schnapsen is a complex mix of randomness and strategical gameplay in which there are multiple factors that can affect the flow of the game. This is why we are interested to create and evaluate heuristics which can assure to lead a player or an AI to a good position in the game.

b. What is a Heuristic?

Humans are intelligent. Humans can look at a scenario and immediately associate it with a good or bad aspect. The excellent pattern recognition and analytical skills of humans makes it easy for them to infer if a certain game position in chess is eventually good or bad. But computers aren't intelligent yet. They do not possess rationalizing skills. To be able to assess a certain scenario, they need to compare it with a scale or some other scenario or an end result. This other thing in talk is called a heuristic.

A heuristic, in game playing algorithms, refers to a certain value which is associated to a game state, in order for the algorithm to select a move. It is the factor which biases a computer to pick a certain move over another. Heuristics take into consideration numerous factors about the game. Naively speaking, a heuristic might consider a move which leads the player to having higher points and the opponent to have lower points. A heuristic might also be shaped to get affected by the amount of high cards a player has (in a card game). It is usually tailored to the rules and usage of the game, taking into consideration all the intricate factors of the game.

c. Simple Heuristics

Imagine a game of tic tac toe (crosses and circles). An algorithm would create the various game states possible, given a starting state. A plausible heuristic might be to get most of your moves in a single line, either vertically, horizontally or diagonally. Then attach this heuristic to the particular game state. Do this for all game states and then your algorithm has a value to grade a game state. It then goes through all the game state and chooses the one with the best heuristic value. However, this heuristic has a fallacy. It would lead an algorithm to play a move in a line which has one of the player's moves and one of opponents'. Therefore the heuristic would have to be reengineered to eliminate such lines from consideration which already have an opponent's move in it. This too, doesn't lead to a perfect heuristic. It would not counter cases where opponent has 2 moves in a line and needs one more to win. Therefore, an addition would be to move on such position with the highest priority. Some more additions, for example, are making gambit moves, etc. In this way, every intricate law of the game is stitched in the heuristic to make it grade a game state more accurately.

d. Complex Heuristics

Tic tac toe is a very simple game where there is no external randomness involved. However, Schnapsen is not. While there exist strategies that lead a player to have a better position in the game, Schnapsen still has the factor of drawing random cards from the deck. This factor of randomness adds difficulty in designing a heuristic for the game. Even if a player had been following a certain heuristic to play and score good points, the opponent can receive an ace of trump from the deck and can turn the game around. On the other hand, if the player is restricted to play good moves in order to wait for the ace of trump to be revealed, the player fails to follow the heuristic of scoring high points and loses after all. Furthermore, the player does not know if a certain card is with the opponent or in the deck. If the player has a queen and wants the king of the same suit, it does not know if the opponent has it or he can get it from the deck. In such a case, would it be better to wait for a king and save the queen, or would it be rather better to get rid of a low-points card (queen)? Therefore, in Schnapsen, heuristics could be pointed towards the direction of having marriages, having trump cards, having cards of high ranks and having better game points. In this paper, we experiment with combinations of these factors towards building a robust heuristic for an algorithm to pick the best move while playing Schnapsen.

e. What is a good Heuristic?

In broad terms, a good heuristic function is a function that rates the game state in a way that leads the player towards victory. However, in some games, there are cases where it is beneficial for the player to lose the current hand / trick/ round, in order to win the entire game. A good heuristic function must recognize such conditions and accordingly, lead the player towards making not only the locally optimal, but the globally optimal choice. We consider a heuristic to be good if it takes into consideration all the factors of the game and performs at par against all the bots that are provided to us.

f. Various bots and their functionalities

We were provided with several bots which play the game Schnapsen using different strategies. These bots include rand, bully, rdeep, minimax, alphabeta, kbbot, and mlbot. However, our research revolves around using simple bots like rand, bully and rdeep. We did this because alphabeta and minimax were programmed to play only the second phase of the game, and mlbot and kbbot are not be compatible with our system of heuristics. So, the bots we play with and against are rdeep, rand and bully. Below, we describe what each of these bots do.

rand: rand is a very simple bot. It gets a state where it must make a move. It randomly chooses a move out of all the legal moves possible. Therefore, rand is not following any strategy and is used to play against our naive heuristics.

bully: bully is a development upon rand. It does what its name suggests. Bully's strategy can be broken down into two parts. It would, if it has one, play the trump of the highest rank. If, however, the opponent makes the first move, then it would play the card of the highest rank of the same suit. If it does not have any trumps or the same suit as that played by opponent, then it would play a card with the highest rank (irrespective of the suit).

rdeep: rdeep is by far, the most strategic bot considered. For every state, it assumes future game states if the game is in phase 1. If the game is in phase 2, rdeep continues testing moves based on the perspective of the state. If the game is in phase 1, rdeep generates possible game states, given the game state input and goes into a certain depth of the possibilities of every move begin played. At each such depth, it uses heuristics to rate the game state and returns the states with best heuristics, then it picks the best move. This process makes rdeep the most analytical and successful bot in our collection.

4. Thesis Statement

Our research question for the paper is: “Can multiple heuristic measures outperform a single heuristic measure in grading a game state of Schnapsen effectively and accurately?.” We will experiment with different heuristics, which evaluate specific parts of the logic and strategy of the game and we will discover whether a combination of these heuristics, to calculate an evaluation of game states, will lead to any significant changes in performance of the agent.

For effectiveness, this strategy shall be valid for any possible game state the agent might come across, in which case the combined heuristic, can calculate a valid evaluation of the state. It must also consider all the important factors of the game. For accuracy, we want the combined heuristic to be correctly informed about all the strategic insights it takes into consideration. For our research we started out with three simple, shallow heuristics which tackle very specific strategic aspects of Schnapsen. We make the assumption that these heuristic values will not show good results individually, and later on will prove this. Furthermore we want to research what causes any difference in the performance of a bot when these heuristics are mixed together into a single combined heuristic.

5. Experimental Setup

To make observations about the performance and functionality of our devised heuristics we compared results in a progressive fashion starting with playing with the heuristics naively on their own. To make sure the game results were significantly accurate we let the bot play around 100 games with each alternative composition of the heuristics and their weight. We chose this number because around this frequency the performance ratio tends to reach the actual winning rate and leaves little room for lucky or odd outliers. We alternate evenly between starting players to remove any starting bias from the results.

We first look at every single heuristic separately to first get an impression of the strategic influence each one has to offer. Only after this we can combine them together to see which strategic value are derived.

To properly measure the practicality of our heuristics there are many dimensions of the gathered data we could compare, such as measuring difference in time or space complexity is an available option, but they don’t provide any useful insight to what makes a heuristic good in our situation, because it doesn’t tell us anything about the strategic insights. Also we don’t have enough understanding of the game at our disposal to concentrate our observations towards individual moves, to classify those as being “good” position-improving moves or not. As a result of this we focus our experiment on round results. We base our criticism of heuristics on success rates of entire rounds. Another constraint is that we don’t take into consideration the amounts of points with which individual games are won as this isn’t a part of the framework we perform our experiments on (tournament.py) and our heuristics are designed to take a short term gain as a priority and don't differentiate between attempting to win with as many points as it could, by securing a setting with as less possible opponent points.

We started working with individual heuristics to see if our assumption, that individual heuristics would fail to consider every possibility of a game state, even the ones where they are not valid, is correct. The performance should not be a lot better than rand in this case. We continued by adding the naïve heuristics to the default rdeep and calculating a combined heuristic by adding weights to each separately calculated heuristic and adding them together to a single combined heuristic. This way we will test whether taking multiple aspects of the game strategy into consideration will lead to better performance rates compared to just a single heuristic. We went on to keep tweaking weights and testing different combinations to discover eccentric results.

6. Heuristic Functions

a. Opponent’s Points

In this heuristic, game states are evaluated on the basis of opponent having the least amount of points. The maximum number of points possible to make in one round is 128. So we use the formula (1 - (opponent_points/128)) to assign a value (0-1) to evert game state. The lower the opponent’s points, the higher will be the heuristic value.

Claim: Since this heuristic looks only at the opponent’s points and does not take into consideration the scenario of the player itself, we predicted it to perform poorly.

Experiment: We conducted a tournament of the rdeep bot using this heuristic vs the other bots. 100 games of rdeep vs rand, 100 of rdeep vs bully and 100 games of rdeep vs rdeep (with original heuristics) were played.

Observation: When plugged into the rdeep’s heuristic function, this heuristic function performed poorly. Against rand, it won 40% of the games. Against bully, it won 33% of the games. Against rdeep(with original heuristics), it could secure only 15 % of the games.

Inference: This heuristic is indeed a shallow measure of a game state. This means that it could lead the algorithm to a state where the player has 0 points and the opponent has points greater than 0. It would still, in that case, perform its function of following the path where opponent has relatively lower points, but only against itself, not against the player. Considering the player’s points in addition to the opponent would be a development on it.

b. Marriage

In this heuristic, we look at all the moves of the players and count the number of marriages. The maximum number of marriages that a player can have in a hand is 2. Therefore, the maximum number of points the player can achieve from marriages is 60 (considering a royal marriage). Therefore the formula (marriage_points / 60) could be used to give a 0-1 scale of marriage points. The higher this value, the better the position the player has.

Claim: Since this heuristic looks only at the marriage points and does not take into consideration the scenario where the player does not have any marriages, we predicted it to perform poorly.

Experiment: We conducted a tournament of the rdeep bot using this heuristic vs the other bots. 100 games of rdeep vs rand, 100 of rdeep vs bully and 100 games of rdeep vs rdeep (with original heuristics) were played.

Observation: When plugged into the rdeep's heuristic function, this heuristic function performed poorly. Against rand, it won 54% of the games. Against bully, it won 40% of the games. Against rdeep, it could secure only 18 % of the games.

Inference: The results match our expectation. The scenarios where it performs poorly are the ones where the player has no marriage. The probability to not have a marriage is very high. In this case, the heuristic would judge every other game state equally, this is why it performance is low. It is a strong factor of the game and could be combined with other heuristics to achieve a better result.

c. High Cards

This heuristic makes a measure of how high a player's cards are. The maximum number of points to have in a hand is 54. We loop through all the possible cards in the players hand and sum up the cards' points. The formula (total_card_points / 54) is then used to return a heuristic value between 0-1. The higher this value is, the better hand a player has.

Claim: We expect this heuristic to perform decently since it appears obvious that the higher the cards a player has, the better are his chances to win.

Experiment: We conducted a tournament of the rdeep bot using this heuristic vs the other bots. 100 games of rdeep vs rand, 100 of rdeep vs bully and 100 games of rdeep vs rdeep (with original heuristics) were played.

Observation: When plugged into the rdeep's heuristic function, this heuristic function performed poorly. Against rand, it won 44% of the games. Against bully, it won 30% of the games. Against rdeep, it could secure only 10 % of the games.

Inference: The results reveal a very important aspect about the game. W/e did not consider cases where if we play high, the opponent can trump those cases and secure the points by itself. It would leave the player with lower cards, lower points and not the position of the leader. We might combine this function with a possible trump heuristic function. If the player has some trump cards, then the probability that the opponent has trump cards as well is low. Therefore, the function can perform better in that case.

d. Trump Cards

The trump heuristic takes into consideration the amount of trump cards a player has. Schnapsen revolves around having good trump cards since they are the most powerful ones and can win over any other suit. We count the number of trump cards in a player's moves and the formula (total_trumps / 5) is then used to return a value between 0 – 1. The higher the value, the higher the chances of the player winning the hand, and eventually, the game.

Claim: We expect this function to perform decently when compared to the ones above. This is because trump cards can win most of the exchanges and are an asset to possess as the game advances.

Experiment: We conducted a tournament of the rdeep bot using this heuristic vs the other bots. 100 games of rdeep vs rand, 100 of rdeep vs bully and 100 games of rdeep vs rdeep (with original heuristics) were played.

Observation: When plugged into the rdeep's heuristic function, this heuristic function performed fairly well. Against rand, it won 70% of the games. Against bully, it won 45% of the games. Against rdeep, it could secure only 25 % of the games.

Inference: The results match our expectation. The scenarios where it performs poorly are the ones where the player has no trumps. In other scenarios, it did win a fair number of matches when compared to the other heuristic functions. The reason it performed well is, as stated above, that trumps can win most of the exchanges. However, the probability of having a trump is not as high, and this, we recognize, as the main reason of it not having an overwhelming victory.

e. Ratio

This is the default heuristic used by the rdeep bot which was provided to us. The ratio heuristic has two advantages. It directs the algorithm towards a state where the player has more points, but also, the opponent has lesser points. It returns a value between 0 – 1 which is calculated by the formula (player_points / (player_points + opponent_points)). The higher the value, the better.
Claim: We expect this function to perform a lot better than the others. This is because this function guarantees to lead the player to a state where he has more points and the opponent has less.

Experiment: We conducted a tournament of the rdeep bot using this heuristic vs the other bots. 100 games of rdeep vs rand, 100 of rdeep vs bully and 100 games of rdeep vs rdeep (with original heuristics) were played.

Observation: When plugged into the rdeep's heuristic function, this heuristic function had an excellent performance. Against rand, it won 90% of the games. Against bully, it won 80% of the games. Against rdeep, it could secure only 50 % of the games.

Inference: The results exceed our expectations. Upon long reflection, we come to the conclusion that this function strikes at the core of the purpose of the game. The purpose of the game is to have most points. And this function leads the player to the states where his points are maximized, and the opponent's is minimized.

7. Developments

a. Combination of Heuristic functions

From the above observations, it becomes certain that we must use a fair combination of all the heuristic functions to obtain a strong position in the game. Using single heuristics is not a good strategy because it only covers a part of the strategy to achieve high points. All the individual heuristics when used together, we predict, would come in harmony to fill the voids exposed by other heuristics and perform better overall.

b. Experiments with combinations

We will perform 2 kinds of experiments with combination of heuristics.

In the first kind, we will perform matches of rdeep vs other bots where rdeep will use a combination of the 3 basic heuristic functions of Trump Cards, High Cards and Marriages. We avoid using the Opponent's Heuristic any further in our research as it is flawed conceptually. The Opponent's points are considered in the ratio points anyway. We will call the heuristic functions and multiply their value with a certain weight which will be unique for every heuristic. Then we will take the average of the sum of heuristic values and divide that by sum of weights to achieve an average heuristic value between 0 – 1.

In the second variation, we will perform matches of rdeep vs other bots where rdeep will use a combination of the 3 basic heuristic functions of Trump Cards, High Cards and Marriages in addition with the dominant Ratio heuristic. We will call the heuristic functions and multiply their value with a certain weight which will be unique for every heuristic. Then we will take the average of the sum of heuristic values and divide that by sum of weights to achieve an average heuristic value between 0 – 1.

c. Balancing Heuristic weights

To obtain an effective overall heuristic value, we need to distribute the weights effectively amongst the individual heuristic functions. We plan to provide the highest weight to the Ratio heuristic because of its outstanding performance amongst others and since it focuses on maximizing the player's points and minimizing the opponent's points. We will experiment with varied weights of other heuristics to see how our results vary.

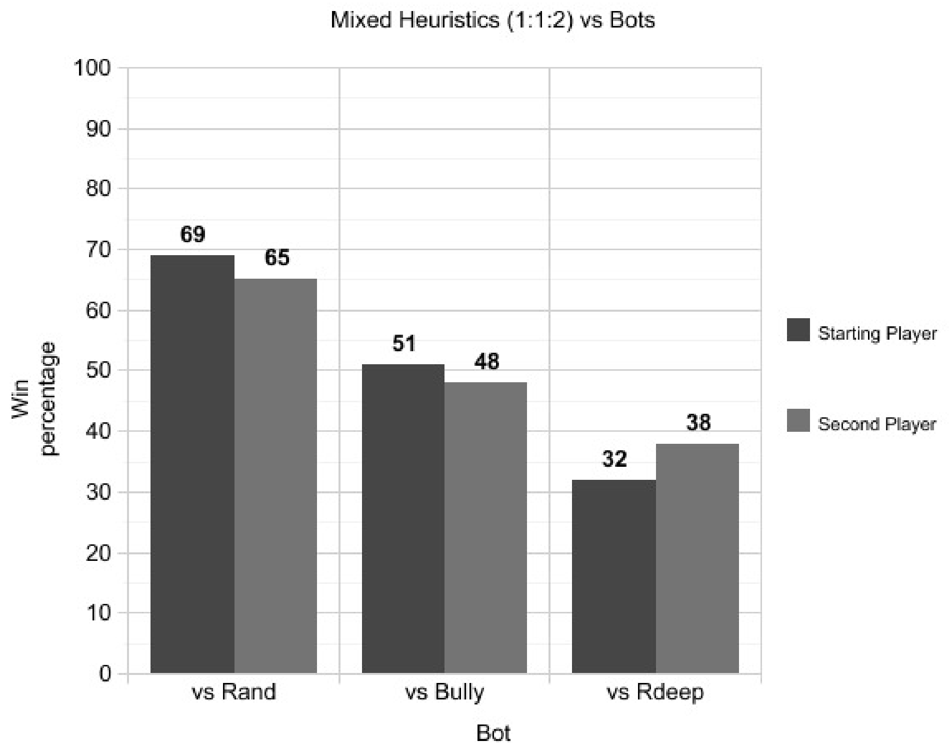
We will make a further attempt to use the concept of adaptive weights. This means that if a certain heuristic function provides low values, we will lower that heuristic's weight. On the other hand, we will increase a heuristic's weight if we achieve better values from it. This seems to be an advanced logic since it always tries to take the best heuristic out of the best individual heuristics. In other words, it covers up the cases where some individual heuristics bring down the overall heuristic value. For example, if a person has no marriages, then the marriage heuristic function would return a very low value. The player, however, may have good trump cards and so, in this case, we will consider less of marriage value and more of trump value. By doing this, we ensure that the advantage is made from every situation that a player is in. Unless the case where a player has poor cards and cannot win anyway.

To counter the cases where the player has poor cards and can't secure a victorious stance, we tweaked the Ratio weights. If a player has less than 33 points, then the Ratio function achieves maximum weight. This means that if a player has poor cards and less than 33 points, then maximum effort is put to make the player win more points and cross the threshold of 33, so that if at all it loses, it loses with the least damage.

With the advanced adaptive weight system and an improved Ratio function, we aim to achieve better results than rdeep (which is using only Ratio function) in every situation against every bot.

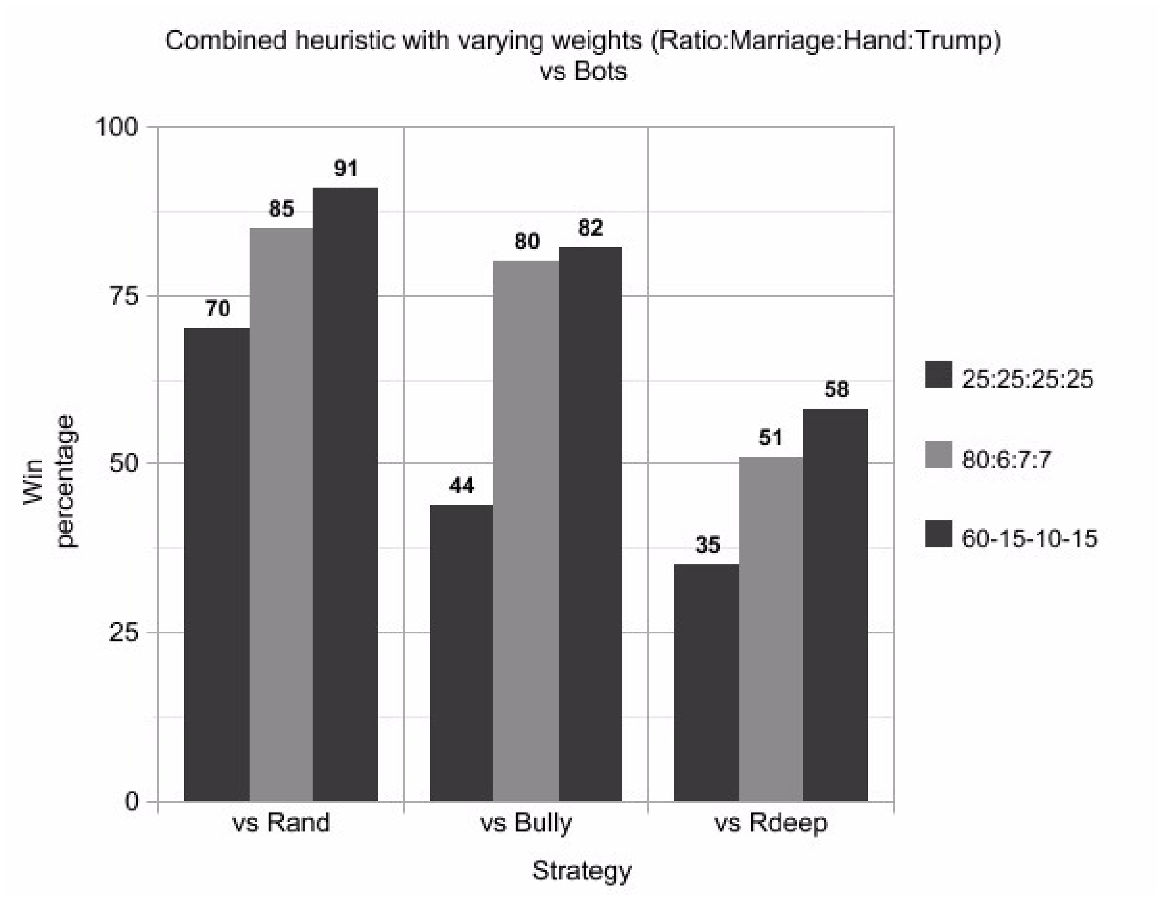
8. Results

After combining the heuristics together into a single weighted heuristic we again played a considerate amount of games against the default bots.



This graph depicts the result of the first kind of our combination heuristic experiment. The one where we combined the 3 heuristic functions: Trump cards, High cards and Marriage values. We made each heuristic influence the evaluation value with an almost equal degree, with an extra margin given to the Trump card heuristic. This is done to take into consideration that the probabilities of gaining a royal marriage or a hand with many high cards increases as a consequence of reaching a situation where there is high ratio of trump cards in the agent's hand compared to the opponent's hand. We came to this observation after testing many permutations of the weight distribution. The weight values used were 1 for Marriage, 1 for High cards and 2 for Trumps.

The result of this combined heuristic is that the performance rate gained an improvement in comparison to the individual heuristics. It now performs better than rand in a slight majority of the games played and competes with bully on an equal level. This demonstrates that some of the lacking strategic insight of individual heuristics are accounted for when combined, so the bot has become more effective in determining more optimal states from others, which in turn has increased its odds of winning games against any of the other default bots by a significant amount. The current bot is however still outmatched by the regular heuristic of rdeep as the current evaluations are not sufficient to compete against the Ratio heuristic stand alone. Ratio heuristic is a very dominant factor in wining the game as it tries to maximize the player's points and minimize the opponent's points.



When Ratio heuristic was added to our combination of heuristics, the overall performance matched our expectations. The above graph shows a measure of the functions performance against the 3 standard bots. Against rand, the performance increased by 50% more win ratio. Against bully, the performance increased by nearly 100% more wins and against rdeep, the performance increased by 50% win ratio than before.

In the right side of the graph, the various weight values can be seen. The first ratio corresponds to the Ratio function, the second ratio corresponds to the Marriage function, the third ratio corresponds to the high cards function and finally, the fourth function corresponds to the Trump cards function. We experimented using several ratio values and have graphed the 3 most successful ones.

We achieve the best results when we use 60% weight for Ratio, 15% for Marriage, 10% for High cards and 15% for Trumps. We have mentioned our rationale behind using a high ratio value for Ratio function above. Having a nearly balanced weight for the other 3 heuristic functions leads to a better result because of the fact that they are each other's compliment. They try to fill the voids left open by the others. So in cases where a player does not have trump, but has marriages, the fact that equal weight has been assigned to them grades the state equally well and the players is confirmed to have a better position.

9. Conclusion

To conclude, a combination of heuristic function does perform better than a single heuristic function to grade a game state in Schnapsen effectively and accurately. Used with the Ratio function, the Trump Cards, High Cards and the Marriage function outperform the three bots rand, bully and rdeep.

Our combination of heuristic functions performs better than others because of the fact that we covered most dominant factors of the game in it. Our method of adaptive weights made it certain to use the best out of every scenario a player can possibly land into.

In retrospect, we think we could have modified the tournament.py script and the alphabeta and minimax bots to develop our strategies upon them. However, our results speak concretely of our logical reasoning behind it.

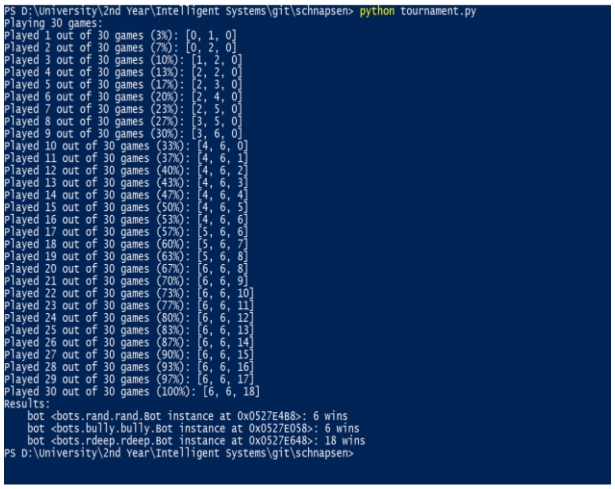
For future possibilities of implementing heuristic functions, we recommend going in the direction of creating a function which generates the ratio of trumps or high cards in player’s hand to the trumps and high cards as unseen in the perspective.

10. Appendix

Worksheet 1a

Question 1) Which of the three default bots does the best? Add the output to your report.

Out of the three bots ‘rdeep’, ‘rand’, and ‘bully’, ‘rdeep’ performs the best. Out of 10 tournaments (each consisting a total of 30 games) that we conducted between the 3 bots, ‘rdeep’ won all of them. The screenshot of one such tournament is attached below:



```
PS D:\University\2nd Year\Intelligent Systems\git\schnapsen> python tournament.py
Playing 30 games:
Played 1 out of 30 games (3%): [0, 1, 0]
Played 2 out of 30 games (7%): [0, 2, 0]
Played 3 out of 30 games (10%): [1, 2, 0]
Played 4 out of 30 games (13%): [2, 2, 0]
Played 5 out of 30 games (17%): [2, 3, 0]
Played 6 out of 30 games (20%): [2, 4, 0]
Played 7 out of 30 games (23%): [2, 5, 0]
Played 8 out of 30 games (27%): [3, 5, 0]
Played 9 out of 30 games (30%): [3, 6, 0]
Played 10 out of 30 games (33%): [4, 6, 0]
Played 11 out of 30 games (37%): [4, 6, 1]
Played 12 out of 30 games (40%): [4, 6, 2]
Played 13 out of 30 games (43%): [4, 6, 3]
Played 14 out of 30 games (47%): [4, 6, 4]
Played 15 out of 30 games (50%): [4, 6, 5]
Played 16 out of 30 games (53%): [4, 6, 6]
Played 17 out of 30 games (57%): [5, 6, 6]
Played 18 out of 30 games (60%): [5, 6, 7]
Played 19 out of 30 games (63%): [5, 6, 8]
Played 20 out of 30 games (67%): [6, 6, 8]
Played 21 out of 30 games (70%): [6, 6, 9]
Played 22 out of 30 games (73%): [6, 6, 10]
Played 23 out of 30 games (77%): [6, 6, 11]
Played 24 out of 30 games (80%): [6, 6, 12]
Played 25 out of 30 games (83%): [6, 6, 13]
Played 26 out of 30 games (87%): [6, 6, 14]
Played 27 out of 30 games (90%): [6, 6, 15]
Played 28 out of 30 games (93%): [6, 6, 16]
Played 29 out of 30 games (97%): [6, 6, 17]
Played 30 out of 30 games (100%): [6, 6, 18]
Results:
    bot <bots.rand.rand.Bot instance at 0x0527E4B8>: 6 wins
    bot <bots.bully.bully.Bot instance at 0x0527E058>: 6 wins
    bot <bots.rdeep.rdeep.Bot instance at 0x0527E648>: 18 wins
PS D:\University\2nd Year\Intelligent Systems\git\schnapsen>
```

Question 2) Have a look at the code: what strategy does the bully bot use?

Bully's strategy can be broken down into two parts. It would, if it has one, play the trump of the highest rank. If, however, the opponent makes the first move, then it would play the card of the highest rank of the same suit. If it does not have any trumps or the same suit as that played by opponent, then it would play a card with the highest rank (irrespective of the suit).

Question 3) For our game, this strategy would be no good. Why not?

This strategy just looks one move ahead and picks the one with the highest ratio of points. But that does not necessarily guarantee a victory. It is analogous to a game of chess where the player captures opponent's knight (3 points advantage) as it would have been the move that provides the highest ratio of points in the current game state. However, in the next move, the opponent would capture the player's queen (9 points disadvantage) which could lead the player to lose the game.

Consider this example from a game of Scnapsen ni phase 2:

player 1: JD JS KC and player 2: QH QD QC. Trump: C

Player 1's best hill climbing strategy when thinking 1 state ahead would be to play the king of trump, which will lead to him eventually succumbing in this game.

A development upon this idea would be to look 4-5 moves ahead and then pick the move which leads to the highest ratio. But whether this strategy also leads to guaranteed victory (if there can be), is arguable.

Question 4) If you wanted to provide scientific evidence that `rdeep` is better than `rand`, how would you go about it?

While rand and rdeep both pickup random moves, rdeep maintains a heuristic for every choice it makes. It makes 8 rnadom choices and then chooses the one which leads to the highest heuristic. Therefore, rdeep makes a choice that is certain to lead it to a better state. Whereas rand just chooses a random move without looking forward and seeing the consequences of the move. Therefore, rdeep performs better than rand.

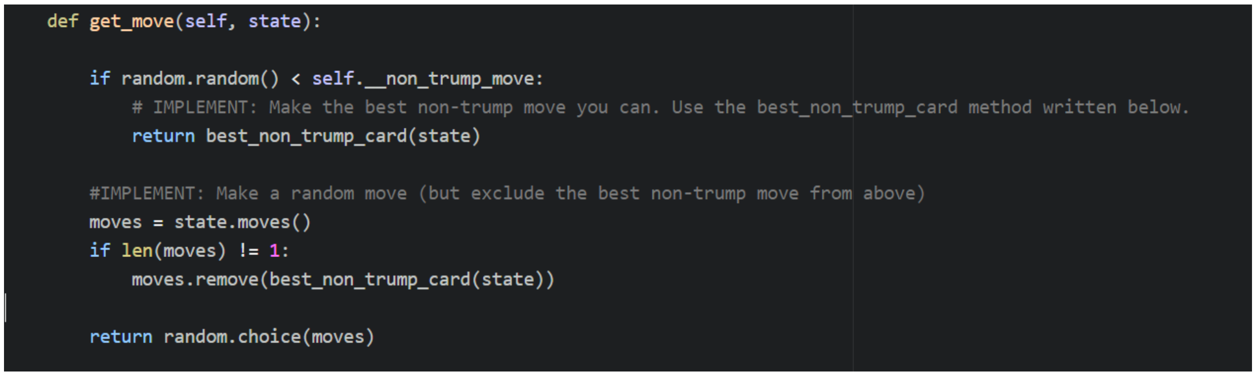
Question 5) Add your implementation of get_move() and the result of a tournament against rand to your report.

improved bully: now keeps trump cards in later stage of the game(points>33) to have a better hand in the second phase

```
for index, move in enumerate(moves):
                        sum_points = state.get_points(1) + state.get_points(2)
                        if move[0] is not None and Deck.get_suit(move[0]) ==
state.get_trump_suit():
if (move[0] % 5 == 3 or move[0] % 5 == 4) and sum_points < 33:
            print sum_points
            continue
            moves_trump_suit.append(move)
```

Worksheet 1b

Question 6) Two simple bits of code are missing (indicated with lines starting with #IMPLEMENT). Read the whole script carefully and finish the implementation of the Bot. Run the experiment. It should output a file experiment.pdf containing a heatmap. Add this heatmap to your report, and discuss briefly what it means.

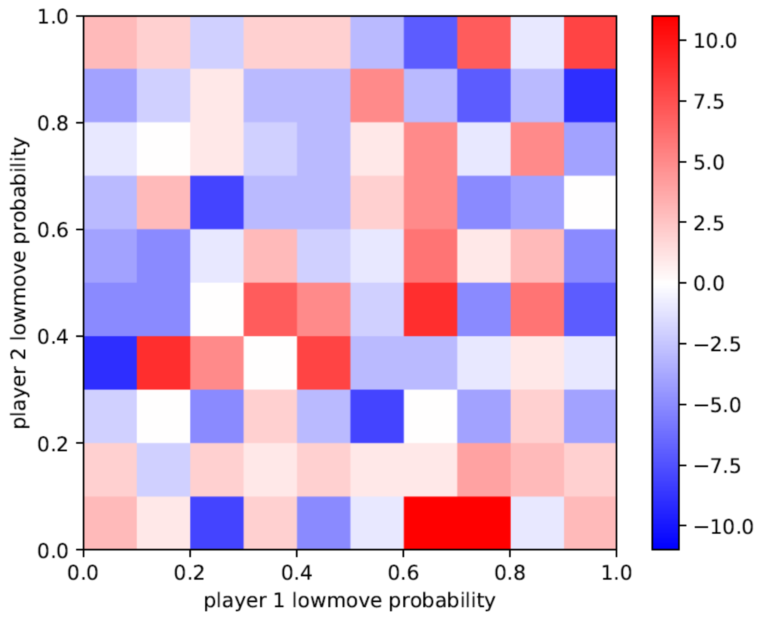


```
    def get_move(self, state):

        if random.random() < self.__non_trump_move:
            # IMPLEMENT: Make the best non-trump move you can. Use the best_non_trump_card method written below.
            return best_non_trump_card(state)

        #IMPLEMENT: Make a random move (but exclude the best non-trump move from above)
        moves = state.moves()
        if len(moves) != 1:
            moves.remove(best_non_trump_card(state))

        return random.choice(moves)
```



Every cell of the heatmap shows the number of games won by player 1 / player 2 when used the respective probability of playing the highest non trump card that the cell corresponds to.

Question 7) All you need to do to finish the minimax bot is to add one line of code on line 58. Take your time to really understand the minimax algorithm, recursion, and the rest of the code. Finish the bot, and let it play against `rand (use flag to start in phase 2)`. Add the line you wrote and the results of the tournament to the appendix of your report.

value, m = self.value(next_state, depth+1)

Question 8) Once again, crucial parts of the implementation are missing. Finish the implementation of the alphabeta bot. The script check_minimax.py lets you see if you implemented alphabeta and minimax correctly. What does it do? Run it and add the output to the appendix of your report.

value, m = self.value(next_state, alpha, beta, depth+1)

if beta <= alpha:

Question 9) What heuristic do these implementations use? Try to come up with a better one. Implement it and see how it does.

These implementations use the ratio of a players' point to the total points as a heuristic. We tried to combine this heuristic with number of marriages a player can show, however, it did not have a significant impact on the score.

Worksheet 2

Question 1) Add a clause to the knowledge base to that it becomes unsatisfiable. Report the line of code you added.

Add : kb.add_clause(~C)

Question 2) Exercise 8 of this week's work session on logical agents contained the following knowledge base:

- **A -> B**
  B -> A
  A -> (C ^ D)

Convert it to clause normal form, and write a script that creates this knowledge base. Print out its models and report them. As seen in the exercise, the knowledge base entails A ^ C ^ D. What does that say about the possible models for the knowledge base?

```
# Add clauses

kb.add_clause(A, B)

kb.add_clause(~B, A)

kb.add_clause(~A, C)

kb.add_clause(~A, D)
```

The fact that this kb entails A^C^D means that whenever A^C^D is true, the kb also has to be true. This can be seen is valid from the results of kb obtained below:

```
{A: True, C: True, B: False, D: True}

{A: True, C: True, B: True, D: True}
```

Question 3) What does this mean for the values of x and y? Are there any integer values for x and y that make these three constraints true?

If x = y and x + y is larger than 2 but smaller than 5, then both values have to be 2.

Question 4) If we know that [x + y < 5] must be false, as in the second model, we know that x + y >= 5 must be true. Write each of these three models as three constraints that must be true.

[x=y] & ([x+y >2] OR [x+y < 5]) & ([x+y <=2] OR [x+y >= 5])

Question 5) Write this down in propositional logic first (two sentences). Then convert this to clause normal form. Now create a knowledge base with the required clause and report which models are returned. The script test3.py does almost all the work for you. All you need to do is fill in the blanks.

a = x + y > 2
b = x + y < 5
c = x + y < -2
d = x + y > -5

(A and B) or (C and D) **move conj outward**

((A and B) or C)) and ((A and B) or D) **move conj outward**

((C or A) and (C or B)) and ((D or A) and (D or B)) **split clauses**

(C or A), (C or B), (D or A), (D or B)

```
kb.add_clause(c, a)
kb.add_clause(c, b)
kb.add_clause(d, a)
kb.add_clause(d, b)
```

returned models:

```
{[y + x < 5]: True, [y + x > -5]: True, [y + x < -2]: False, [y + x > 2]: True, [(-y) + x ==
0]: True}

{[y + x < 5]: True, [y + x > -5]: True, [y + x < -2]: True, [y + x > 2]: False, [(-y) + x ==
0]: True}
```

Question 6) Look at the code in test4.py. (The long list in the beginning is just the variable instantiation, the real modelling starts at line 50.) Extend the document with the knowledge for a strategy PlayAs, always playing an As first. Check whether you can do reasoning to check whether a card is entailed by the knowledge base or not.

The cards A0, A5, A10, A15 and PA0, PA5, PA10, PA15 entail to the knowledge base.

Question 7) Build a more complex logical strategies. For examples, you can define the notion of a cheap card, as being either a jack, king or queen, and devise a strategy that plays cheap card first. Test whether you can use logical reasoning to check whether the correctness of a move w.r.t. this strategy is entailed by your knowledge base.

kbbot.py

```
from api import State, util
```

```
import random, load
from kb import KB, Boolean, Integer
class Bot:
    def __init__(self):
        pass
    def get_move(self, state):
        moves = state.moves()
        for move in moves:
            if move[0] == None:
                return move
        # random.shuffle(moves)
        # alist.sort(key=lambda x: x.foo)
        moves.sort(key=lambda x: (5 - x[0]%5))
        for move in moves:
            if not self.kb_consistent(state, move):
                # Plays the first move that makes the kb inconsistent. We do not take
                # into account that there might be other valid moves according to the strategy.
                # Uncomment the next line if you want to see that something happens.
                # print "Strategy Applied"
                return move
        # If no move that is entailed by the kb is found, play random move
        return random.choice(moves)
    # Note: In this example, the state object is not used,
    # but you might want to do it for your own strategy.
    def kb_consistent(self, state, move):
    # type: (State, move) -> bool
        # each time we check for consistency we initialise a new knowledge-base
        kb = KB()
        # Add general information about the game
        load.general_information(kb)
        # Add the necessary knowledge about the strategy
        load.strategy_knowledge(kb)
        # This line stores the index of the card in the deck.
        # If this doesn't make sense, refer to _deck.py for the card index mapping
        index = move[0]
        # This creates the string which is used to make the strategy_variable.
        # Note that as far as kb.py is concerned, two objects created with the same
        # string in the constructor are equivalent, and are seen as the same symbol.
        # Here we use "pj" to indicate that the card with index "index" should be played with the
        # PlayJack heuristics that was defined in class. Initialise a different variable if
        # you want to apply a different strategy (that you will have to define in load.py)
        variable_string_jacks = "pj" + str(index)
        strategy_variable_jacks = Boolean(variable_string_jacks)
        variable_string_queens = "pq" + str(index)
        strategy_variable_queens = Boolean(variable_string_queens)
        variable_string_kings = "pk" + str(index)
        strategy_variable_kings = Boolean(variable_string_kings)
        # Add the relevant clause to the loaded knowledge base
        kb.add_clause(~strategy_variable_jacks)
        kb.add_clause(strategy_variable_jacks, ~strategy_variable_queens)
        kb.add_clause(strategy_variable_jacks, strategy_variable_queens,
~strategy_variable_kings)
        # If the knowledge base is not satisfiable, the strategy variable is
        # entailed (proof by refutation)
        return kb.satisfiable()
```

load.py

```
from kb import KB, Boolean, Integer
JACKS = 'j'
PLAYED_JACKS = 'pj'
j = [0] * 20
```

```
pj = [0] * 20
QUEENS = 'q'
PLAYED_QUEENS = 'pq'
q = [0] * 20
pq = [0] * 20
KINGS = 'k'
PLAYED_KINGS = 'pk'
k = [0] * 20
pk = [0] * 20
for i in range(20):
    j[i], pj[i] = Boolean(JACKS + str(i)), Boolean(PLAYED_JACKS + str(i))
    q[i], pq[i] = Boolean(QUEENS + str(i)), Boolean(PLAYED_QUEENS + str(i))
    k[i], pk[i] = Boolean(KINGS + str(i)), Boolean(PLAYED_KINGS + str(i))
def general_information(kb):
    for i in range(0, 20, 5):
        kb.add_clause(j[i + 4])
    for i in range(0, 20, 5):
        kb.add_clause(q[i + 3])
    for i in range(0, 20, 5):
        kb.add_clause(k[i + 2])
def strategy_knowledge(kb):
    for i in range(0, 20, 5):
        kb.add_clause(~j[i + 4], pj[i + 4])
        kb.add_clause(j[i + 4], ~pj[i + 4])
    for i in range(0, 20, 5):
        kb.add_clause(~q[i + 3], pq[i + 3])
        kb.add_clause(q[i + 3], ~pq[i + 3])
    for i in range(0, 20, 5):
        kb.add_clause(~k[i + 2], pk[i + 2])
        kb.add_clause(k[i + 2], ~pk[i + 2])
```

Worksheet 3

Question 1

First line

Value = self.heuristic(next_state)

The rest can be derived from the state.py file

Question 2

Tournament.py –p rand,bully,ml

```
Playing 30 games:
Played 1 out of 30 games (3%): [1, 0, 0]
Played 2 out of 30 games (7%): [2, 0, 0]
Played 3 out of 30 games (10%): [3, 0, 0]
Played 4 out of 30 games (13%): [4, 0, 0]
Played 5 out of 30 games (17%): [5, 0, 0]
Played 6 out of 30 games (20%): [6, 0, 0]
Played 7 out of 30 games (23%): [6, 1, 0]
Played 8 out of 30 games (27%): [7, 1, 0]
Played 9 out of 30 games (30%): [8, 1, 0]
Played 10 out of 30 games (33%): [8, 2, 0]
Played 11 out of 30 games (37%): [8, 2, 1]
Played 12 out of 30 games (40%): [8, 2, 2]
Played 13 out of 30 games (43%): [8, 2, 3]
Played 14 out of 30 games (47%): [8, 2, 4]
Played 15 out of 30 games (50%): [8, 2, 5]
Played 16 out of 30 games (53%): [8, 2, 6]
Played 17 out of 30 games (57%): [8, 2, 7]
Played 18 out of 30 games (60%): [8, 2, 8]
Played 19 out of 30 games (63%): [9, 2, 8]
Played 20 out of 30 games (67%): [9, 2, 9]
Played 21 out of 30 games (70%): [9, 2, 10]
Played 22 out of 30 games (73%): [9, 2, 11]
Played 23 out of 30 games (77%): [9, 2, 12]
Played 24 out of 30 games (80%): [9, 2, 13]
Played 25 out of 30 games (83%): [9, 2, 14]
Played 26 out of 30 games (87%): [9, 2, 15]
Played 27 out of 30 games (90%): [9, 2, 16]
Played 28 out of 30 games (93%): [9, 2, 17]
Played 29 out of 30 games (97%): [9, 2, 18]
Played 30 out of 30 games (100%): [9, 2, 19]
Results:
    bot <bots.rand.rand.Bot instance at 0x02112080>: 9 wins
    bot <bots.bully.bully.Bot instance at 0x02111F30>: 2 wins
    bot <bots.ml.ml.Bot instance at 0x02112878>: 19 wins
```

Question 3

Tournament between rdeep, rand and ml trained mlbots

```
C:\Users\aqp\Desktop\schnapsen2>tournament.py -p ml,mlrand,mlml
C:\Users\aqp\Desktop\schnapsen2\bots\ml/model.pkl
C:\Users\aqp\Desktop\schnapsen2\bots\ml/model.pkl
C:\Users\aqp\Desktop\schnapsen2\bots\mlrand/model2.pkl
C:\Users\aqp\Desktop\schnapsen2\bots\mlrand/model2.pkl
C:\Users\aqp\Desktop\schnapsen2\bots\mlml/model3.pkl
C:\Users\aqp\Desktop\schnapsen2\bots\mlml/model3.pkl
Playing 30 games:
Played 1 out of 30 games (3%): [1, 0, 0]
Played 2 out of 30 games (7%): [1, 1, 0]
Played 3 out of 30 games (10%): [1, 2, 0]
Played 4 out of 30 games (13%): [2, 2, 0]
Played 5 out of 30 games (17%): [3, 2, 0]
Played 6 out of 30 games (20%): [3, 3, 0]
Played 7 out of 30 games (23%): [4, 3, 0]
Played 8 out of 30 games (27%): [5, 3, 0]
Played 9 out of 30 games (30%): [6, 3, 0]
Played 10 out of 30 games (33%): [6, 4, 0]
Played 11 out of 30 games (37%): [6, 4, 1]
Played 12 out of 30 games (40%): [7, 4, 1]
Played 13 out of 30 games (43%): [8, 4, 1]
Played 14 out of 30 games (47%): [9, 4, 1]
Played 15 out of 30 games (50%): [9, 4, 2]
Played 16 out of 30 games (53%): [10, 4, 2]
Played 17 out of 30 games (57%): [10, 4, 3]
Played 18 out of 30 games (60%): [11, 4, 3]
Played 19 out of 30 games (63%): [11, 4, 4]
Played 20 out of 30 games (67%): [12, 4, 4]
Played 21 out of 30 games (70%): [12, 5, 4]
Played 22 out of 30 games (73%): [12, 6, 4]
Played 23 out of 30 games (77%): [12, 7, 4]
Played 24 out of 30 games (80%): [12, 8, 4]
Played 25 out of 30 games (83%): [12, 8, 5]
Played 26 out of 30 games (87%): [12, 8, 6]
Played 27 out of 30 games (90%): [12, 8, 7]
Played 28 out of 30 games (93%): [12, 8, 8]
Played 29 out of 30 games (97%): [12, 9, 8]
Played 30 out of 30 games (100%): [12, 9, 9]
```

Question 4

Added features:

last card played
Difference of player points

```
    feature_set.append(p1_points - p2_points + 128)

    lastplayed = state.get_opponents_played_card()
      if lastplayed ==  None:
                    lastplayed = 0
    feature_set.append(lastplayed)
```

Played tournament with old mlrand vs new mlrand

Old mlrand: 15 wins
New mlrand: 15 wins