Strategy Selection in Schnapsen

Improving the Performance of Knowledge Base Bot Through the Strategy Selection

Sara Abesova, Yozlem Ramadan, Malgorzata Zdych

Vrije Universiteit Amsterdam, De Boelelaan 1105, The Netherlands

Abstract. This research paper analyses the impact of the quality and quantity of playing strategies on the overall performance of a learning agent in the card game of Schnapsen. A knowledge-based bot was equipped with various playing approaches obtained from already existing learning agents. The strategies were also derived from an analysis of the competent and logical choices within a game, which were determined by our research team. The results were achieved by setting up several tournaments to compare the performance of the knowledge base bot and its advanced versions created during this study.

Keywords: Schnapsen · Propositional logic · Knowledge based · Bots · Strategy · Artificial intelligence.

1 Introduction

Designing computers capable of executing complex tasks and embedding them to our daily lives is what artificial intelligence works towards. Solving problems like searching, sorting and mathematical dilemmas can already be achieved by computers. Improving the abilities of machines to perform activities that humans can instinctively execute became a common practise in this domain [11]. These kind of progresses are remarkably significant, as nowadays intelligent agents are capable of winning board games even against professional human players. The example of this could be Reversi or Connect 4 games [7]. These accomplishments are crucial, because they largely contribute to the development of new innovations that can be relevant outside the scope of game-playing [9].

The goal of our research is to evaluate already existing agents playing the Austrian card game of Schnapsen, analyse their strategies and eventually come up with a better version of them. In this context, an intelligent agent can interpret knowledge obtained from the game data and select actions depending on the given information [7].

In the past, an improved version of an intelligent agent which uses Perfect Information Monte Carlo Sampling (PIMC), was tested against human specialists and outperformed them [5].

However, our agent is developed based on propositional logic and logical strategies employed by human players. It also makes use of the knowledge base

combined with the states of the game that are utilized by the bot to implement certain approaches. Applying strong strategies is one of the most significant aspects of playing an efficient game. Logic can be used in representing different variants of information to the agent, resulting in highly valuable techniques in the games [8].

With the aim to answer our study objective, we have developed five district bots versions with varied strategy combinations. It was essential to understand how different strategies interact with one another. Furthermore, four experiments were conducted for each of the bots. Additional information about the tests and the results have been provided in section 4 and 5 of the paper.

2 Background Information

2.1 The rules of Schnapsen

Schnapsen is a two-player turn-based card game, similar to the game sixty-six. The goal of the game is to earn 66 points through different combinations of cards before the opponent does.[1] The deck of the game consists of Ace, Ten, King, Queen and Jack of every suit. The values for the cards are eleven, ten, four, three and two, respectively. At the beginning of the game one suit is determined as the trump suit. Each player starts with 5 cards in hand. The remaining cards which are in the stock are used to replenish the hands after a card is played. The game has two phases: phase one is semi-observable and phase two is fully observable.

The game starts when one of the players places a card face up in the center of the table. For each trick, the player must follow the same suit or play the trump card if possible. The player with the highest material value of a same suit obtains the winning points. The winner of the last turn leads the next trick. When on lead, if the player possesses the King and Queen of the same suit, they might declare a marriage. Obtaining a non-trump marriage earns 20 points, however a trump marriage earns 40 points. Moreover, when on lead, one can exchange the trump card for the trump of Jack if possible.

At the end of a round, points are assigned based on the difference the score. One point is obtained if the difference is greater than 33. Two points are given when the opponent has less than 33 trick points. Finally, a player is awarded with 3 points in case their opponent did not win any trick.

Schnapsen is a game with basic rules that allows us to perform different, often complex, experiments. After learning the essential rules, it is crucial to obtain the knowledge of effective strategies. This is meaningful due to fact that performing in Schnapsen depends to a great extent on strategy choice rather than on pure good luck.

2.2 The strategies in the Schnapsen game

Game strategy refers to the game plan, plan of action or tactics. There are many approaches that one can select to play a good game. However, in Schnapsen

specifically, these are dependant on which stage of the game they are implied. When the other player is in the lead and the stack is open, the main strategy is to follow the suit in the non-trump tricks and in the meantime keeping potential marriage. If the opponent played a non-trump card of Ten or Ace, the beneficial move would be to win the trick with the trump. Moreover, if a player has more cards of the same suit, they should use a higher one to win the trick and eventually gain the points. In some instances, one may not wish to win the trick for strategic reasons. In these scenarios, the most efficient move would be to play Jack, Queen or King, given that the marriage was already declared.

The fundamental strategy when being on lead is to declare a marriage or make an exchange when having Jack of trump suit. It is also essential to play a non-trump Jack, Queen or King that has no possibility of marriage anymore. In general, it is suggested to eliminate lower cards from player's hand and thus not give the opponent an opportunity to win high points. [2]

2.3 General description of bots

The Intelligent Systems course has provided main bots as examples of agents that can play Schnapsen. They operate on a variety of approaches and search strategies. How they function is clarified in the sections below.

2.3.1 Rand The random bot does not follow any particular strategy. This agent selects a random move from all possible ones. The legal moves that the bot can make are based on the rules of the game, cards in hand and game state. Game status depends on which player is in the lead, game phase or the card played by the opponent.

2.3.2 Kbbot The knowledge base agent determines its strategy via propositional logic. Its operators can be either True or False. Firstly, general information about the game is added to the knowledge base, followed by significant clauses. Later, information about the specific approach one can take are included.

2.3.3 Rdeep The strategy which rdeep uses is called Perfect Information Monte Carlo sampling (PIMC). This technique eliminates the problem of imperfect information by randomly selecting perfect information worlds. Thus, the agent assumes that all opponents are making a random choice and it subsequently applies this assumption to an end state. The bot later ranks the game moves based on the heuristic value of the resulting state [3]. The best overall score is utilized to choose which move to make for the current game state.

2.3.4 Bully The bully bot has an aggressive approach to the game. It starts with checking whether trump moves are available. In these scenarios, it is obligated to play it. In other cases, the bot has to follow the suit of the opponent regardless of the rank of the card. When none of these alternatives are available

the agent plays the highest rank possible, regardless of the suit of the card. This approach demonstrates that the bully bot is always attempting to play the high cards in order to acquire points as soon as possible.

3 Research Question

The aim of this research paper is to answer the question: How can the performance of knowledge base bot be improved through the strategy selection in the game of Schnapsen?

As stated above, kbbot is a simple bot applying only Jack strategy focusing on playing Jack of any suit when possible. In all other cases, it plays a random card. We attempt to expand already existing knowledge base, consisting of only four Jack cards, by adding all remaining cards: Queens, Kings, Tens and Aces. Our aim is to play the tournaments against the agents of varying difficulty level. Bot using PIMC and attacking bot, subsequently known as rdeep and bully, are considered as more complex ones, whereas the knowledge base bot and randomly playing bot are perceived as rather simple. In following tournaments, obtained points and won games will be considered when comparing newly designed bots. Those two measures will provide us with more comprehensive overview on the selected strategies and their impact on the performance of bots.

Our plan is to implement combination of strategies for our bots from already existing ones. We will also take into consideration our current knowledge of Schnapsen to logically analyse the moves that usually lead the human players to reach the goal state. Thus, the general pseudocode of the most logical strategies was developed and it can be found in Appendix G.

The main focus in enhancing kbbot will be to differentiate between the offensive and defensive part of the game. Therefore, different strategies will be applied when our bot is on the lead and when it is the opponent's turn.

We created five separate bots and each one of them takes use of similar strategies, but the way they perform wary due to the combinations of these. Four of them are more developed ones with the consideration of the defense/offense strategy, as well as the trump/non-trump suit approach. However, one of our agents - abot - will be only slightly improved version of the kbbot. Such strategy selection will demonstrate if it is even attainable to improve the performance of kbbot by expanding its approach. It will show us whether it is possible that one strategy is already sufficient to improve the performance of kbbot against other agents, like bully, and also kbbot itself.

The two of four more developed bots - cbot and dbot - operate based on the same strong strategy when it comes to defence, but when it comes to offense, they have slightly different approach. Both of them have the knowledge base of non-trump moves implemented and it enables them to play next higher card when the player does not have a lower card. However, when a player does not have any non-trump moves available, cbot plays any trump card, whereas dbot plays the highest trump card. On the other hand, the two remaining bots - bbot and ebot - diverge from each other when it comes to the defence. The strategy

of the ebot differentiates between the cases when player is in the possession of trump cards and when it is not. Whereas, bbot focuses on playing the highest rank card available no matter the suit.

Our hypothesis presumes that bots with more elaborated defense strategy will outperform kbbot. Those approaches take into consideration whether the opponent played trump or non-trump suit card and they additionally check for the same cards in the player's hand, and based on that they choose the best move possible. As states, the strategies that include playing higher ranked cards at the beginning of the game allow the player to score more points in a rapid way and thus result in a win. What is more, playing the lowest cards available, when it is convenient or impossible to win the trick, prevents the opponent from gaining high points. It also saves the player from wasting valuable cards that can be used later in the game. Thus, it can be stated that cbot and dbot have a high probability of being better performing version of kbbot, since their defence strategy is more developed and sophisticated. They make use of almost all approaches mentioned above [10]. Thus, it can be certain that such a strong strategy selection will lead to their better performance, in comparison to kbbot. Respectively, it can be assumed that ebot will win more games and gain more points, contrary to the bbot, as the defensive strategy of the first one is more complicated and chooses moves depending on the availability of trump cards. Whereas, bbot does not even consider such a scenario. Accordingly, the strategy selection for bbot and ebot will bring about only slight improvement in their performance and particularly it will be the case with latter one. Eventually, it can be specified that abot with its rather uncomplicated strategy will have comparable results to the kbbot and thus will not outperform it.

4 Experimental Set-up

To find an answer for our research question, we will conduct five experiments to test five of our bots and see their performance against the same four agents: kbbot, rand, rdeep and bully. Additionally, the tournament with kbbot against rand, rdeep and bully will be conducted to observe how kbbot performs in the first place. Each one of learning agents: abot, bbot, cbot, dbot and ebot will take part in 30 tournaments with 1000 games in tournament. Therefore, every bot will play four times 30 000 games, since there are four different opponents. Additionally, the tournaments will take place twice, as first time the points gained during game will be taken into account and second time, the games won will be counted.

4.1 The strategy selection

The newly created bots adopt different combinations of the strategies. Short summaries of the various approaches and the information about its application in the particular bot can be found in the table.

Table 1. Description of strategies used in newly developed bots

strategy	**description**	**bot:**
jack_strategy	The jack card is chosen if it is in the hand of the player.	abot
queen_king	The queen card is chosen if it is in the hand of the player.	abot
random choice	The random move is chosen from the legal moves.	abot
highest_trump_suit_card	The cards of the trump suit are appended in a list, which is sorted in ascending order. The first element is returned.	bbot, dbot, ebot
list_of_non_trump_moves	The list of all non trump legal moves.	bbot, cbot, dbot, ebot
knowledge_base_non_trump	This function implements the knowledge base of non-trump moves. It starts with the Jack cards, however the next higher card is played, when the player does not have a lower card. In the scenario, when the player does not have any non-trump moves available, cbot plays any trump card, whereas bbot, dbot and ebot play the highest trump card.	bbot, cbot, dbot, ebot
knowledge_base_trump	It implements a knowledge base of trump moves. As the It begins with the least valuable move possible. When trump move is not available it uses knowledge_base_non_trump.	cbot, dbot
highest_rank_available	This method returns a card with the highest rank available of any suit.	bbot
same_suit_as_opponent	This function checks the suit of played opponent's card. It examines if the player has the moves of this suit and if so, they are appended in a list moves_same_suit which is sorted in descending order. The first element is compared to opponent's card. If the value of this card is higher than the player's, this move is chosen and returned.	cbot, dbot
higher_trump_suit_card	The available moves of the trump suit are appended in a list. The list is sorted in descending order and again the first element of the list is compared to the opponent's card. If the index of the opponent's card is higher than the player then this move is chosen and returned.	cbot, dbot
kb_consistent	Plays the first move that makes the knowledge base inconsistent, thus one of the Jack cards, as it applies knowledge about the jack_strategy.	all bots,
kb_consistent1	Plays the first move that makes the knowledge base inconsistent, thus one of the Ace cards, as it applies knowledge about the ace_strategy	bbot, cbot, dbot, ebot
kb_consistent2	Plays the first move that makes the knowledge base inconsistent, thus one of the Ten cards, as it applies knowledge about the ten_strategy	bbot, cbot, dbot, ebot
kb_consistent3	Plays the first move that makes the knowledge base inconsistent, thus one of the Queen cards, as it applies knowledge about the queen_strategy	all bots
kb_consistent4	Plays the first move that makes the knowledge base inconsistent, thus one of the King cards, as it applies knowledge about the king_strategy	bbot, cbot, dbot, ebot

4.2 Abot

The performance of our very first agent - abot - will consider Jack strategy and additionally when there is no Jack cards in the player's hand, the Queen strategy will apply. This approach assumes that any card of the Queen rank will be played whenever it is possible. Although, when it is not the case the random move would be performed. (see Appendix A)

4.3 Bbot

The bbot always plays the highest_rank_available when the opponent leads the trick. However, when the bot itself is on the lead it applies knowledge_base_non_trump. The knowledge base implemented and loaded in this function and knowledge_base_trump can be found under Appendix F. If there are no legal moves available from non-trump suit, then highest_trump_suit_card method is implemented. (see Appendix B)

4.4 Cbot and Dbot

Cbot and dbot both use knowledge_base_non_trump method when it is their turn to start the trick. However, cbot plays any trump card and dbot plays the highest one in the scenario of not having any non-trump suit cards. When it is opponent's turn to start the trick, both bots take the same approach. Firstly, they check if opponent played trump suit card and it is true, then player's hand is checked for the availability of trump cards. If there is a possibility of playing one higher_trump_suit_card method is applied and when there is no such option knowledge_base_non_trump is returned. On the other hand, there can be situation where opponent plays non-trump suit card. In this case, when the rank of a card is Ten, Ace of the same suit will be played or one of the moves from knowledge_base_trump function. When the Ace is played, the bot will respond with knowledge_base_trump card as well. Lastly, in the case of any other card performed by the opponent the same_suit_as_opponent will be applied. (see Appendix C and D)

4.5 Ebot

In the defense, Ebot firstly checks whether the trump suit card is in hand and when it is the case it plays the highest one. In the scenario of only non-trump moves ebot will play the lowest card possible, as knowledge_base_non_trump is applied there, as well as in the offense part. (see Appendix E)

5 Results and Findings

For each experiment, the total of 30 000 games was conducted. We decided to represent the outcomes results in forms of table and graph representations.

P-value refers to the level of statistical significance, which is in our research compared to alpha = 0.05. If the p-value is lower than alpha, it is considered to be a strong proof against the null hypothesis, implying that the data was not obtained randomly.[4] The p-value higher than alpha would indicate that it is not scientifically acceptable and thus the null hypothesis should be recognised.[6]

In Table 2, the p-values obtained through the tournaments are demonstrated. The games against kbbot can be seen in the first row. The p-values of all of them, except bbot, are below 0.05. The p-value of the bbot is equal to 0.223, thus it signifies the null hypothesis can not be rejected. Moreover, it can not be concluded that bbot performs better than kbbot. However, the p-values for abot, cbot, dbot and ebot are below 0.05, so the null hypothesis can be rejected. The p-value of ebot against rand is higher than alpha as well, with the value of 0.786, which implies that data could be obtained in a random way. The results of the tournaments against rdeep signify that p-value is lower than alpha, thus it is a strong proof against null hypothesis. What is more, the hypothesis should be retained in the case of abot and ebot against bully, as the p-values are respectively 0.190 and 0.112, hence above alpha value.

Table 2. The p-value of the two-tailed test with alpha = 0.05 obtained from the tournaments between already existing bots and newly created ones.

bots	abot	bbot	cbot	dbot	ebot
kbbot	0.00226	0.223	2e-323	4.16e-65	1.99e-17
rand	2.37e-7	1.20e-9	2e-323	5.35e-145	0.786
rdeep	2e-323	2e-323	2e-323	2e-323	2e-323
bully	0.190	2e-323	1.39e-33	4.48e-25	0.112

Table displayed below allow us to observe the final results obtained throughout the tournaments between already existing and our designed bots.

First of all, it was crucial to truly understand why do already existing bots operate as they do and how would they perform against each other. In the first column it can be observe the performances of kbbot against other bots. Kbbot's results are slightly better than rand's. However, such a minor difference can be considered as negligible. We can notice that in games held between kbbot and rdeep, the latter performs nearly four times better. In comparison with bully, kbbot won moderately more games.

The performance of our bots against kbbot can be considered as improved, particularly in the case of cbot, as its result was 18343. However, one of the bots - dbot - performed also relatively well. From obtained data it can be concluded that all of our bots performed better against rand, in contrast to kbbot. Ebot is excluded from this comparison since there is a high probability that the score was randomly obtained (see Table 2). To sum up, cbot outperformed kbbot significantly. In fact, it can be even stated that the strategies selected enhanced its performance. Against rdeep all of the bots lost, however cbot's performance

Table 3. Representation of the number of games won by kbbot and our bots against kbbot, rand, rdeep and bully. W signifies winner, L represents loser.

bots	kbbot	abot	bbot	cbot	dbot	ebot
kbbot	-	15265 W	14894 L	18343 W	16475 W	14264 L
rand	15330 W	15448 W	15527 W	18407 W	17217 W	15024 W
rdeep	6162 L	6262 L	4417 L	7455 L	6629 L	5019 L
bully	15765 W	15114 W	9775 L	13954 L	15896 W	14862 L

was moderately better than the others, especially the kbbot's one. In the case of bully, the abot and ebot were not considered, since their null hypotheses needed to be retained. However, only dbot performed better than the kbbot, but not to a significant extent.

Presented graphs allow us to visualize how different strategy implementations influenced the score.

The performance and efficiency of the designed bots (abot, bbot, cbot, dbot, ebot) had to be tested. We have achieved this by conducting tournaments of mentioned bots and with the original ones. The amount of points won as total was recorded and is stored in four graphs below. (see Fig.1-4)

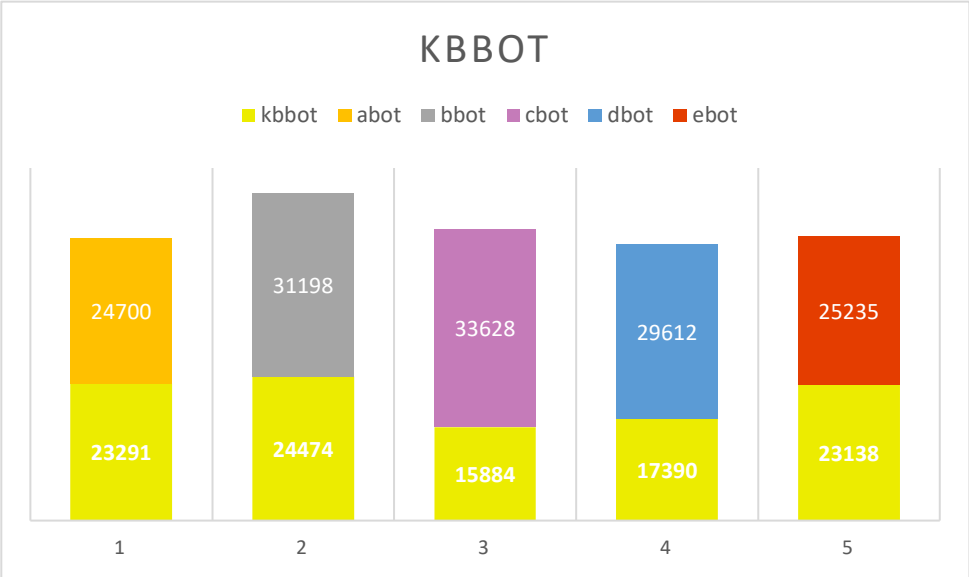

Fig. 1. Kbbot has less points against each of our bots. The biggest difference is when it plays against cbot.

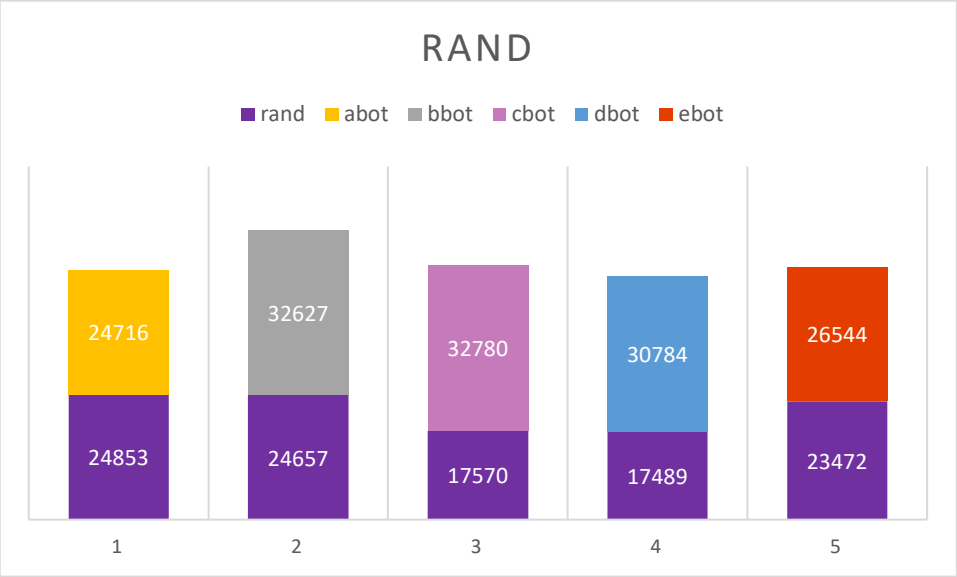

Fig. 2. Bbot, cbot and dbot are the strongest opponents when it comes to playing against rand. The difference in points gained throughout the whole game between rand and abot is rather irrelevant.

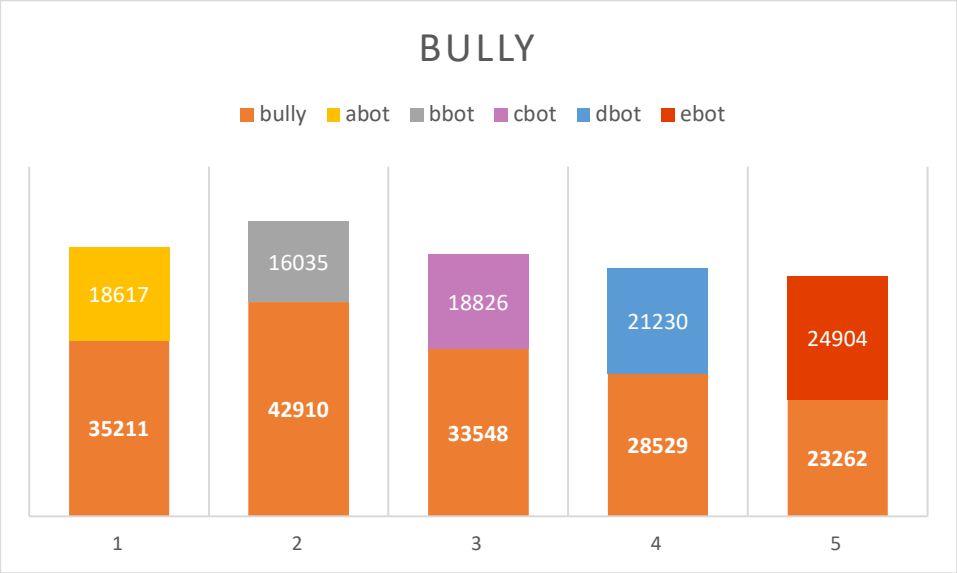

Fig. 3. Bully played the most balanced games against ebot. In tournaments with other bots, bully was advantaged significantly. Bbot was the weakest.

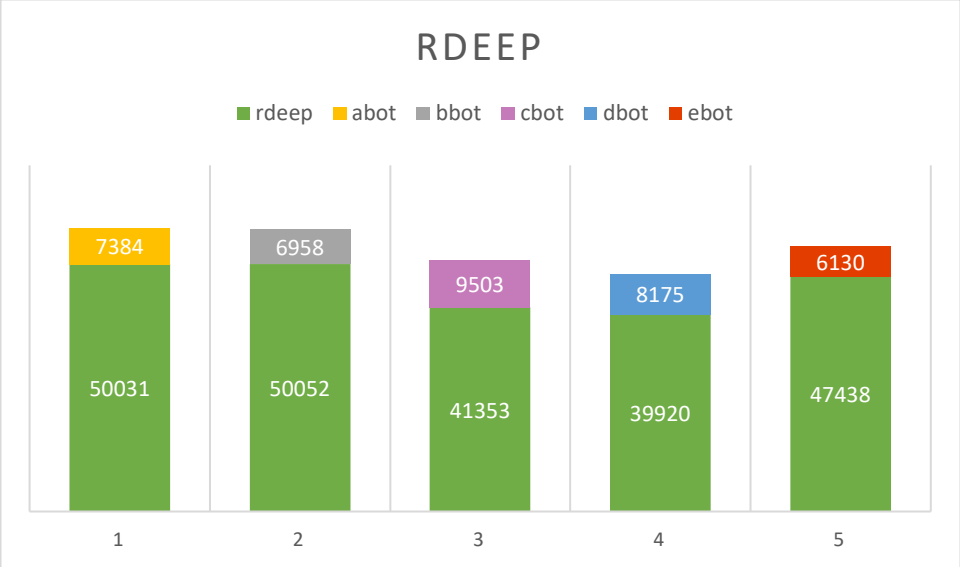

Fig. 4. Rdeep is the strongest opponent, none of our bots managed to outperform it.

Obtained results from tournaments run between different versions of our bot are displayed on graphs where y-axis represents the won games out of 1000 and x-axis depicts tournaments played in thousand (30 implies 30 000 games in total). These can be found in Appendix H.

6 Conclusions

The purpose of this research was to improve the performance of kbbot by developing five new bots and comparing their performance with primary kbbot. Our hypothesis turned out to be correct to a medium extent, as cbot and dbot won most of the games. Cbot won greatly with kbbot, also its performance against rand was much better than kbbot's and the same case was with dbot. The tournament with bully was not that sucessful and only dbot managed to win it and thus outperformed kbbot. Although, when it came to the rdeep none of our bots succeeded, which was not correctly stated in our hypothesis, however it could be predicted. The rdeep's strategy assumes very complex heuristic that essentially learns how the bot acts, what ensures its winning.

Moreover, when it came to the points obtained during the 30 000 games, the results complied largely with our hypothesis. Cbot and dnbot both had definitive advantage with kbbot and rand, but it was also the case with bbot, which was not predicted. Contrarily, we even specified that ebot will be more improved than bbot and it does not apply in this case. However, the ebot turned out to be the most successful against bully and in this scenario it outperformed kbbot.

This aspect agrees with the part of our hypothesis where we concluded that ebot will gain more points than bbot. Nevertheless, the performance of our bots against rdeep was not presumed precisely, including both the number of points scored and games won. To sum up, only cbot and dbot managed to meet most of the assumptions that we recognised in our hypothesis.

7 Future work

Undeniably, our work has many opportunities for future improvements, particularly if it comes to the performance of the bots. Throughout this research we have conducted a set of experiments which resulted in significant outcomes. However, there are still many ways to advance and expand the possibilities of our agents. One way of enhancing our bots would be to establish other new strategies. Even if our current methods show a positive outcome, adding more would be profitable to the performance. Valuable element to examine could be to run more experiments to acquire more accurate data. To conclude, with the sufficient amount of time we could focus more on the strategies applied. Moreover, we would increase the number of tournaments conducted.

References

1. Schnapsen and sixty-six rules, http://psellos.com/schnapsen/rules.html
2. Winning strategy for schnapsen or sixty-six, http://psellos.com/schnapsen/strategy.html
3. Bulitko, V., Björnsson, Y., Lawrence, R.: Case-based subgoaling in real-time heuristic search for video game pathfinding. Journal of Artificial Intelligence Research **39**, 269–300 (2010)
4. Gravetter, F.J., Wallnau, L.B., Forzano, L.A.B., Witnauer, J.E.: Essentials of statistics for the behavioral sciences. Cengage Learning (2020)
5. Haufe, S., Michulke, D., Schiffel, S., Thielscher, M.: Knowledge-based general game playing. KI-Künstliche Intelligenz **25**(1), 25–33 (2011)
6. McLeod, S.: What a p-value tells you about statistical significance. Simply psychology pp. 1–4 (2019)
7. Millington, I., Funge, J.: Artificial intelligence for games. CRC Press (2018)
8. Russell, S., Norvig, P.: Artificial intelligence: a modern approach (2002)
9. Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., et al.: A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. Science **362**(6419), 1140–1144 (2018)
10. Tompa, M.: Winning Schnapsen: From Card Play Basics to Expert Strategy. CreateSpace Independent Publishing Platform (2015), https://books.google.nl/books?id=beoqjgEACAAJ
11. Yannakakis, G.N., Togelius, J.: Artificial intelligence and games. Springer (2018)

8 Appendices

8.1 Appendix A - code for abot

```
from api import State, util
import random
from bots.new_bot import load
from bots.queen_strategy.kb import KB, Boolean
from api import Deck


class Bot:

    def __init__(self):
        pass

    def get_move(self, state):

        moves = state.moves()
        random.shuffle(moves)

        for move in moves:
            if not self.kb_consistent(state, move):  # jack strategy
                return move
            elif not self.kb_consistent3(state, move):  # queen strategy
                return move
            else:
                return random.choice(moves)

    def kb_consistent(self, state, move):
        # type: (State, move) -> bool

        kb = KB()

        load.general_information(kb)

        load.jack_strategy(kb)

        index = move[0]

        variable_string = "pj" + str(index)
        strategy_variable = Boolean(variable_string)

        kb.add_clause(~strategy_variable)

        return kb.satisfiable()
```

```
    def kb_consistent3(self, state, move):
        # type: (State, move) -> bool

        kb = KB()

        load.general_information(kb)

        load.queen_strategy(kb)

        index = move[0]

        variable_string = "pj" + str(index)
        strategy_variable = Boolean(variable_string)

        kb.add_clause(~strategy_variable)

        return kb.satisfiable()
```

8.2 Appendix B - code for bbot

```
from api import State, util
import random
from . import load
from .kb import KB, Boolean, Integer
from api import Deck


class Bot:
    def __init__(self):
        pass


    def highest_trump_suit_card(self, state):
        moves = state.moves()
        chosen_move = moves[0]
        moves_same_suit = []

        # Get all moves of the trump suit
        for index, move in enumerate(moves):
            if move[0] is not None and Deck.get_suit(move[0]) == state.get_trump_suit():
                moves_same_suit.append(move)
        moves_same_suit.sort(key=lambda a: a[0])
        # we are sorting the list in the ascending order
        if len(moves_same_suit) > 0:
            chosen_move = moves_same_suit[0]
```

```
        return chosen_move


def highest_rank_available(self, state):
    moves = state.moves()
    chosen_move = moves[0]
    for index, move in enumerate(moves):
        if move[0] is not None and move[0] % 5 <= chosen_move[0] % 5:
            chosen_move = move
    return chosen_move

def knowledge_base_non_trump(self, state):

    # All legal moves
    moves = state.moves()
    moves_not_trump_suit = []

    # Get all non-trump suit moves available
    for index, move in enumerate(moves):

        if move[0] is not None and Deck.get_suit(move[0]) != state.get_trump_suit():
            moves_not_trump_suit.append(move)

    list_of_non_trump_moves = moves_not_trump_suit

    moves = state.moves()

    random.shuffle(moves)

    for move in moves:
        if move in list_of_non_trump_moves:
            if not self.kb_consistent(state, move):
                return move
            elif not self.kb_consistent3(state, move):
                return move
            elif not self.kb_consistent4(state, move):
                return move
            elif not self.kb_consistent2(state, move):
                return move
            elif not self.kb_consistent1(state, move):
                return move
        else:
            return self.highest_trump_suit_card(state)
```

```
    def get_move(self, state):

        opponents_card = state.get_opponents_played_card()
        moves = state.moves()

        random.shuffle(moves)

        if opponents_card is None:  # when our bot is playing

            return self.knowledge_base_non_trump(state)

        else:   # when opponent is playing

            return self.highest_rank_available(state)

    def kb_consistent(self, state, move):
        # type: (State, move) -> bool

        kb = KB()

        load.general_information(kb)

        load.jack_strategy(kb)

        index = move[0]

        variable_string = "pj" + str(index)
        strategy_variable = Boolean(variable_string)

        kb.add_clause(~strategy_variable)

        return kb.satisfiable()

    def kb_consistent1(self, state, move):
        # type: (State, move) -> bool


        kb = KB()

        load.general_information(kb)

        load.ace_strategy(kb)

        index = move[0]
```

```
        variable_string = "pj" + str(index)
        strategy_variable = Boolean(variable_string)

        kb.add_clause(~strategy_variable)

        return kb.satisfiable()

    def kb_consistent2(self, state, move):
        # type: (State, move) -> bool

        kb = KB()

        load.general_information(kb)


        load.ten_strategy(kb)

        index = move[0]

        variable_string = "pj" + str(index)
        strategy_variable = Boolean(variable_string)

        kb.add_clause(~strategy_variable)

        return kb.satisfiable()

    def kb_consistent3(self, state, move):
        # type: (State, move) -> bool

        kb = KB()

        load.general_information(kb)

        load.queen_strategy(kb)
        index = move[0]

        variable_string = "pj" + str(index)
        strategy_variable = Boolean(variable_string)

        kb.add_clause(~strategy_variable)

        return kb.satisfiable()

    def kb_consistent4(self, state, move):
```

```
        # type: (State, move) -> bool

        kb = KB()

        load.general_information(kb)

        load.king_strategy(kb)

        index = move[0]

        variable_string = "pj" + str(index)
        strategy_variable = Boolean(variable_string)

        kb.add_clause(~strategy_variable)


        return kb.satisfiable()
```

8.3 Appendix C - code for cbot

```
from api import State, util
import random
from . import load
from .kb import KB, Boolean, Integer
from api import Deck


class Bot:
    def __init__(self):
        pass

    def same_suit_as_opponent(self, state):
        moves = state.moves()
        chosen_move = moves[0]
        moves_same_suit = []

        # Get all moves of the same suit as the opponent's played card
        for index, move in enumerate(moves):
            if move[0] is not None and Deck.get_suit(move[0])
            == Deck.get_suit(state.get_opponents_played_card()):
                moves_same_suit.append(move)
        moves_same_suit.sort(key=lambda a: a[0], reverse=True)
        # we are sorting the list in the descending order

        if len(moves_same_suit) > 0:
            for index, move in enumerate(moves_same_suit):
```

```
                if move[0] is not None and state.get_opponents_played_card()
                > moves_same_suit[index][0]:
                    chosen_move = moves_same_suit[index]
            return chosen_move
        else:
            return self.knowledge_base_non_trump(state)

    def higher_trump_suit_card(self, state):
        moves = state.moves()
        chosen_move = moves[0]
        moves_same_suit = []

        # Get all moves of the trump suit
        for index, move in enumerate(moves):
            if move[0] is not None and Deck.get_suit(move[0]) == state.get_trump_suit():
                moves_same_suit.append(move)
        moves_same_suit.sort(key=lambda a: a[0], reverse=True)
        # we are sorting the list in the descending order

        if len(moves_same_suit) > 0:
            for index, move in enumerate(moves_same_suit):
                if move[0] is not None and state.get_opponents_played_card()
                > moves_same_suit[index][0]:
                    chosen_move = moves_same_suit[index]
            return chosen_move
        else:
            return self.knowledge_base_non_trump(state)

    def trump_suit_card(self, state):
        moves = state.moves()
        chosen_move = moves[0]
        moves_trump_suit = []

        # Get all moves of the trump suit
        for index, move in enumerate(moves):
            if move[0] is not None and Deck.get_suit(move[0]) == state.get_trump_suit():
                moves_trump_suit.append(move)

        if len(moves_trump_suit) > 0:
            chosen_move = moves_trump_suit[0]
            return chosen_move


    def knowledge_base_non_trump(self, state):
```

```
        # All legal moves
        moves = state.moves()
        moves_not_trump_suit = []

        # Get all non-trump suit moves available
        for index, move in enumerate(moves):

            if move[0] is not None and Deck.get_suit(move[0]) != state.get_trump_suit():
                moves_not_trump_suit.append(move)

        list_of_non_trump_moves = moves_not_trump_suit

        moves = state.moves()

        random.shuffle(moves)

        for move in moves:
            if move in list_of_non_trump_moves:
                if not self.kb_consistent(state, move):
                    return move
                elif not self.kb_consistent3(state, move):
                    return move
                elif not self.kb_consistent4(state, move):
                    return move
                elif not self.kb_consistent2(state, move):
                    return move
                elif not self.kb_consistent1(state, move):
                    return move
            else:
                return self.trump_suit_card(state)


    def knowledge_base_trump(self, state):
        # All legal moves
        moves = state.moves()
        moves_trump_suit = []

        # Get all trump suit moves available
        for index, move in enumerate(moves):

            if move[0] is not None and Deck.get_suit(move[0]) == state.get_trump_suit():
                moves_trump_suit.append(move)

        list_of_trump_moves = moves_trump_suit
```

```
        moves = state.moves()

        random.shuffle(moves)

        for move in moves:
            if move in list_of_trump_moves:
                if not self.kb_consistent(state, move):
                    return move
                elif not self.kb_consistent3(state, move):
                    return move
                elif not self.kb_consistent4(state, move):
                    return move
                elif not self.kb_consistent2(state, move):
                    return move
                elif not self.kb_consistent1(state, move):
                    return move
            else:
                return self.knowledge_base_non_trump(state)

    def get_move(self, state):

        opponents_card = state.get_opponents_played_card()
        moves = state.moves()

        random.shuffle(moves)

        if opponents_card is None:  # when our bot is playing

            return self.knowledge_base_non_trump(state)

        else:  # when opponent is playing
            if Deck.get_suit(state.get_opponents_played_card()) == state.get_trump_suit():
                for move in moves:
                    suit_of_the_card = Deck.get_suit(move[0])
                    if suit_of_the_card == state.get_trump_suit():
                        return self.higher_trump_suit_card(state)
                    else:
                        return self.knowledge_base_non_trump(state)

            else:
                suit_of_opponents_card = Deck.get_suit(state.get_opponents_played_card())
                rank_of_opponents_card = Deck.get_rank(state.get_opponents_played_card())
                if rank_of_opponents_card == "10":
                    for move in moves:
                        rank_of_the_card = Deck.get_rank(move[0])
```

```
                    suit_of_the_card = Deck.get_suit(move[0])
                    if "A" == rank_of_the_card and suit_of_the_card
                    == suit_of_opponents_card:
                        return move
                    else:
                        return self.knowledge_base_trump(state)
                elif rank_of_opponents_card == "A":
                    return self.knowledge_base_trump(state)
                else:
                    return self.same_suit_as_opponent(state)



    def kb_consistent(self, state, move):
        # type: (State, move) -> bool

        kb = KB()

        load.general_information(kb)

        load.jack_strategy(kb)

        index = move[0]

        variable_string = "pj" + str(index)
        strategy_variable = Boolean(variable_string)

        kb.add_clause(~strategy_variable)

        return kb.satisfiable()

    def kb_consistent1(self, state, move):
        # type: (State, move) -> bool

        kb = KB()

        load.general_information(kb)

        load.ace_strategy(kb)

        index = move[0]

        variable_string = "pj" + str(index)
        strategy_variable = Boolean(variable_string)
```

```
        kb.add_clause(~strategy_variable)

        return kb.satisfiable()

    def kb_consistent2(self, state, move):
        # type: (State, move) -> bool

        kb = KB()

        load.general_information(kb)

        load.ten_strategy(kb)

        index = move[0]

        variable_string = "pj" + str(index)
        strategy_variable = Boolean(variable_string)

        kb.add_clause(~strategy_variable)

        return kb.satisfiable()

    def kb_consistent3(self, state, move):
        # type: (State, move) -> bool

        kb = KB()

        load.general_information(kb)


        load.queen_strategy(kb)

        index = move[0]

        variable_string = "pj" + str(index)
        strategy_variable = Boolean(variable_string)

        kb.add_clause(~strategy_variable)

        return kb.satisfiable()

    def kb_consistent4(self, state, move):
        # type: (State, move) -> bool

        kb = KB()
```

```
        load.general_information(kb)

        load.king_strategy(kb)

        index = move[0]

        variable_string = "pj" + str(index)
        strategy_variable = Boolean(variable_string)

        kb.add_clause(~strategy_variable)

        return kb.satisfiable()
```

8.4 Appendix D - code for dbot

```
from api import State
import random
from . import load
from bots.queen_strategy.kb import KB, Boolean
from api import Deck


class Bot:
    def __init__(self):
        pass


    def highest_rank_available(self, state):
        moves = state.moves()
        chosen_move = moves[0]
        for index, move in enumerate(moves):
            if move[0] is not None and move[0] % 5 <= chosen_move[0] % 5:
                chosen_move = move
        return chosen_move

    def same_suit_as_opponent(self, state):
        moves = state.moves()
        chosen_move = moves[0]
        moves_same_suit = []

        # Get all moves of the same suit as the opponent's played card
        for index, move in enumerate(moves):
            if move[0] is not None and Deck.get_suit(move[0])
            == Deck.get_suit(state.get_opponents_played_card()):
                moves_same_suit.append(move)
```

```
        moves_same_suit.sort(key=lambda a: a[0], reverse=True)
        # we are sorting the list in the descending order

        if len(moves_same_suit) > 0:
            for index, move in enumerate(moves_same_suit):
                if move[0] is not None and state.get_opponents_played_card()
                > moves_same_suit[index][0]:
                    chosen_move = moves_same_suit[index]
            return chosen_move
        else:
            return self.knowledge_base_non_trump(state)


    def higher_trump_suit_card(self, state):
        moves = state.moves()
        chosen_move = moves[0]
        moves_same_suit = []

        # Get all moves of the trump suit
        for index, move in enumerate(moves):
            if move[0] is not None and Deck.get_suit(move[0]) == state.get_trump_suit():
                moves_same_suit.append(move)
        moves_same_suit.sort(key=lambda a: a[0], reverse=True)
        # we are sorting the list in the descending order

        if len(moves_same_suit) > 0:
            for index, move in enumerate(moves_same_suit):
                if move[0] is not None and state.get_opponents_played_card()
                > moves_same_suit[index][0]:
                    chosen_move = moves_same_suit[index]
            return chosen_move
        else:
            return self.knowledge_base_non_trump(state)

    def highest_trump_suit_card(self, state):
        moves = state.moves()
        chosen_move = moves[0]
        moves_same_suit = []

        # Get all moves of the trump suit
        for index, move in enumerate(moves):
            if move[0] is not None and Deck.get_suit(move[0]) == state.get_trump_suit():
                moves_same_suit.append(move)
        moves_same_suit.sort(key=lambda a: a[0])
        # we are sorting the list in the descending order
```

```
        if len(moves_same_suit) > 0:
            chosen_move = moves_same_suit[0]
            return chosen_move
        else:
            return self.knowledge_base_non_trump(state)


    def knowledge_base_non_trump(self, state):

        # All legal moves
        moves = state.moves()
        moves_not_trump_suit = []

        # Get all non-trump suit moves available
        for index, move in enumerate(moves):

            if move[0] is not None and Deck.get_suit(move[0]) != state.get_trump_suit():
                moves_not_trump_suit.append(move)

        list_of_non_trump_moves = moves_not_trump_suit

        moves = state.moves()

        random.shuffle(moves)

        for move in moves:
            if move in list_of_non_trump_moves:
                if not self.kb_consistent(state, move):
                    return move
                elif not self.kb_consistent3(state, move):
                    return move
                elif not self.kb_consistent4(state, move):
                    return move
                elif not self.kb_consistent2(state, move):
                    return move
                elif not self.kb_consistent1(state, move):
                    return move
            else:
                return self.highest_trump_suit_card(state)


    def knowledge_base_trump(self, state):
        # All legal moves
        moves = state.moves()
        moves_trump_suit = []
```

```
        # Get all trump suit moves available
        for index, move in enumerate(moves):

            if move[0] is not None and Deck.get_suit(move[0]) == state.get_trump_suit():
                moves_trump_suit.append(move)

        list_of_trump_moves = moves_trump_suit

        moves = state.moves()

        random.shuffle(moves)

        for move in moves:
            if move in list_of_trump_moves:
                if not self.kb_consistent(state, move):
                    return move
                elif not self.kb_consistent3(state, move):
                    return move
                elif not self.kb_consistent4(state, move):
                    return move
                elif not self.kb_consistent2(state, move):
                    return move
                elif not self.kb_consistent1(state, move):
                    return move
            else:
                return self.knowledge_base_non_trump(state)

    def get_move(self, state):

        opponents_card = state.get_opponents_played_card()
        moves = state.moves()

        random.shuffle(moves)

        if opponents_card is None:  # when our bot is playing

            return self.knowledge_base_non_trump(state)

        else:  # when opponent is playing
            if Deck.get_suit(state.get_opponents_played_card()) == state.get_trump_suit():
                for move in moves:
                    suit_of_the_card = Deck.get_suit(move[0])
                    if suit_of_the_card == state.get_trump_suit():
                        return self.higher_trump_suit_card(state)
```

```
                else:
                    return self.knowledge_base_non_trump(state)

            else:
                suit_of_opponents_card = Deck.get_suit(state.get_opponents_played_card())
                rank_of_opponents_card = Deck.get_rank(state.get_opponents_played_card())
                if rank_of_opponents_card == "10":
                    for move in moves:
                        rank_of_the_card = Deck.get_rank(move[0])
                        suit_of_the_card = Deck.get_suit(move[0])
                        if "A" == rank_of_the_card and suit_of_the_card
                        == suit_of_opponents_card:
                            return move
                        else:
                            return self.knowledge_base_trump(state)
                elif rank_of_opponents_card == "A":
                    return self.knowledge_base_trump(state)
                else:
                    return self.same_suit_as_opponent(state)


    def kb_consistent(self, state, move):
        # type: (State, move) -> bool

        kb = KB()

        load.general_information(kb)

        load.jack_strategy(kb)

        index = move[0]

        variable_string = "pj" + str(index)
        strategy_variable = Boolean(variable_string)

        kb.add_clause(~strategy_variable)

        return kb.satisfiable()

    def kb_consistent1(self, state, move):
        # type: (State, move) -> bool

        kb = KB()

        load.general_information(kb)
```

```
        load.ace_strategy(kb)

        index = move[0]

        variable_string = "pj" + str(index)
        strategy_variable = Boolean(variable_string)

        kb.add_clause(~strategy_variable)

        return kb.satisfiable()

    def kb_consistent2(self, state, move):
        # type: (State, move) -> bool

        kb = KB()

        load.general_information(kb)


        load.ten_strategy(kb)

        index = move[0]

        variable_string = "pj" + str(index)
        strategy_variable = Boolean(variable_string)

        kb.add_clause(~strategy_variable)

        return kb.satisfiable()

    def kb_consistent3(self, state, move):
        # type: (State, move) -> bool

        kb = KB()

        load.general_information(kb)

        load.queen_strategy(kb)

        index = move[0]

        variable_string = "pj" + str(index)
        strategy_variable = Boolean(variable_string)
```

```
        kb.add_clause(~strategy_variable)


        return kb.satisfiable()

    def kb_consistent4(self, state, move):
        # type: (State, move) -> bool

        kb = KB()

        load.general_information(kb)


        load.king_strategy(kb)

        index = move[0]


        variable_string = "pj" + str(index)
        strategy_variable = Boolean(variable_string)

        kb.add_clause(~strategy_variable)


        return kb.satisfiable()
```

8.5 Appendix E - code for ebot

```
from api import State
import random
from . import load
from bots.queen_strategy.kb import KB, Boolean
from api import Deck

class Bot:
    def __init__(self):
        pass


    def highest_trump_suit_card(self, state):
        moves = state.moves()
        chosen_move = moves[0]
        moves_same_suit = []

        # Get all moves of the trump suit
```

```
        for index, move in enumerate(moves):
            if move[0] is not None and Deck.get_suit(move[0]) == state.get_trump_suit():
                moves_same_suit.append(move)
        moves_same_suit.sort(key=lambda a: a[0])
        # we are sorting the list in the descending order
        if len(moves_same_suit) > 0:
            chosen_move = moves_same_suit[0]
            return chosen_move
        else:
            return self.knowledge_base_non_trump(state)

    def knowledge_base_non_trump(self, state):

        # All legal moves
        moves = state.moves()
        moves_not_trump_suit = []

        # Get all non-trump suit moves available
        for index, move in enumerate(moves):

            if move[0] is not None and Deck.get_suit(move[0]) != state.get_trump_suit():
                moves_not_trump_suit.append(move)

        list_of_non_trump_moves = moves_not_trump_suit

        moves = state.moves()

        random.shuffle(moves)

        for move in moves:
            if move in list_of_non_trump_moves:
                if not self.kb_consistent(state, move):
                    return move
                elif not self.kb_consistent3(state, move):
                    return move
                elif not self.kb_consistent4(state, move):
                    return move
                elif not self.kb_consistent2(state, move):
                    return move
                elif not self.kb_consistent1(state, move):
                    return move
            else:
                return self.highest_trump_suit_card(state)
```

```
    def get_move(self, state):

        opponents_card = state.get_opponents_played_card()
        moves = state.moves()

        random.shuffle(moves)

        if opponents_card is None:              # when our bot is playing

            return self.knowledge_base_non_trump(state)

        else:                    # when opponent is playing

            moves = state.moves()

            random.shuffle(moves)
            for move in moves:
                suit_of_the_card = Deck.get_suit(move[0])
                if suit_of_the_card == state.get_trump_suit():
                    return self.highest_trump_suit_card(state)
                else:
                    return self.knowledge_base_non_trump(state)


    def kb_consistent(self, state, move):
        # type: (State, move) -> bool

        kb = KB()

        load.general_information(kb)

        load.jack_strategy(kb)

        index = move[0]

        variable_string = "pj" + str(index)
        strategy_variable = Boolean(variable_string)

        kb.add_clause(~strategy_variable)

        return kb.satisfiable()

    def kb_consistent1(self, state, move):
        # type: (State, move) -> bool
```

```
        kb = KB()

        load.general_information(kb)

        load.ace_strategy(kb)

        index = move[0]

        variable_string = "pj" + str(index)
        strategy_variable = Boolean(variable_string)

        kb.add_clause(~strategy_variable)

        return kb.satisfiable()

    def kb_consistent2(self, state, move):
        # type: (State, move) -> bool

        kb = KB()

        load.general_information(kb)

        load.ten_strategy(kb)

        index = move[0]

        variable_string = "pj" + str(index)
        strategy_variable = Boolean(variable_string)

        kb.add_clause(~strategy_variable)

        return kb.satisfiable()

    def kb_consistent3(self, state, move):
        # type: (State, move) -> bool

        kb = KB()

        load.general_information(kb)

        load.queen_strategy(kb)

        index = move[0]

        variable_string = "pj" + str(index)
```

```
        strategy_variable = Boolean(variable_string)

        kb.add_clause(~strategy_variable)

        return kb.satisfiable()

    def kb_consistent4(self, state, move):
        # type: (State, move) -> bool

        kb = KB()

        load.general_information(kb)


        load.king_strategy(kb)

        index = move[0]

        variable_string = "pj" + str(index)
        strategy_variable = Boolean(variable_string)

        kb.add_clause(~strategy_variable)

        return kb.satisfiable()
```

8.6 Appendix F - the load method, initializes variables for game knowledge and strategies

```
from .kb import KB, Boolean, Integer

J0 = Boolean('j0')
J1 = Boolean('j1')
J2 = Boolean('j2')
J3 = Boolean('j3')
J4 = Boolean('j4')
J5 = Boolean('j5')
J6 = Boolean('j6')
J7 = Boolean('j7')
J8 = Boolean('j8')
J9 = Boolean('j9')
J10 = Boolean('j10')
J11 = Boolean('j11')
J12 = Boolean('j12')
J13 = Boolean('j13')
J14 = Boolean('j14')
J15 = Boolean('j15')
```

```
J16 = Boolean('j16')
J17 = Boolean('j17')
J18 = Boolean('j18')
J19 = Boolean('j19')
PJ0 = Boolean('pj0')
PJ1 = Boolean('pj1')
PJ2 = Boolean('pj2')
PJ3 = Boolean('pj3')
PJ4 = Boolean('pj4')
PJ5 = Boolean('pj5')
PJ6 = Boolean('pj6')
PJ7 = Boolean('pj7')
PJ8 = Boolean('pj8')
PJ9 = Boolean('pj9')
PJ10 = Boolean('pj10')
PJ11 = Boolean('pj11')
PJ12 = Boolean('pj12')
PJ13 = Boolean('pj13')
PJ14 = Boolean('pj14')
PJ15 = Boolean('pj15')
PJ16 = Boolean('pj16')
PJ17 = Boolean('pj17')
PJ18 = Boolean('pj18')
PJ19 = Boolean('pj19')



def general_information(kb):
    # GENERAL INFORMATION ABOUT THE CARDS
    # The Jack cards
    kb.add_clause(J4)
    kb.add_clause(J9)
    kb.add_clause(J14)
    kb.add_clause(J19)
    # The King cards
    kb.add_clause(J2)
    kb.add_clause(J7)
    kb.add_clause(J12)
    kb.add_clause(J17)
    # The Ace cards
    kb.add_clause(J0)
    kb.add_clause(J5)
    kb.add_clause(J10)
    kb.add_clause(J15)
```

```
    # The Ten cards
    kb.add_clause(J1)
    kb.add_clause(J6)
    kb.add_clause(J11)
    kb.add_clause(J16)
    # The Queen cards
    kb.add_clause(J3)
    kb.add_clause(J8)
    kb.add_clause(J13)
    kb.add_clause(J18)


def jack_strategy(kb):
    # PJ(x) <-> J(x),
    # Plays a card when it is a jack
    kb.add_clause(~J4, PJ4)
    kb.add_clause(~J9, PJ9)
    kb.add_clause(~J14, PJ14)
    kb.add_clause(~J19, PJ19)
    kb.add_clause(~PJ4, J4)
    kb.add_clause(~PJ9, J9)
    kb.add_clause(~PJ14, J14)
    kb.add_clause(~PJ19, J19)


def ace_strategy(kb):
    # PJ(x) <-> A(x),
    # Plays a card when it is an ace
    kb.add_clause(~J0, PJ0)
    kb.add_clause(~J5, PJ5)
    kb.add_clause(~J10, PJ10)
    kb.add_clause(~J15, PJ15)
    kb.add_clause(~PJ0, J0)
    kb.add_clause(~PJ5, J5)
    kb.add_clause(~PJ10, J10)
    kb.add_clause(~PJ15, J15)


def ten_strategy(kb):
    # PJ(x) <-> T(x),
    # Plays a card when it is a ten
    kb.add_clause(~J1, PJ1)
    kb.add_clause(~J6, PJ6)
    kb.add_clause(~J11, PJ11)
    kb.add_clause(~J16, PJ16)
```

```
    kb.add_clause(~PJ1, J1)
    kb.add_clause(~PJ6, J6)
    kb.add_clause(~PJ11, J11)
    kb.add_clause(~PJ16, J16)


def queen_strategy(kb):
    # PJ(x) <-> Q(x),
    # Plays a card when it is a queen
    kb.add_clause(~J3, PJ3)
    kb.add_clause(~J8, PJ8)
    kb.add_clause(~J13, PJ13)
    kb.add_clause(~J18, PJ18)
    kb.add_clause(~PJ3, J3)
    kb.add_clause(~PJ8, J8)
    kb.add_clause(~PJ13, J13)
    kb.add_clause(~PJ18, J18)


def king_strategy(kb):
    # PJ(x) <-> K(x),
    # Plays a card when it is a king
    kb.add_clause(~J2, PJ2)
    kb.add_clause(~J7, PJ7)
    kb.add_clause(~J12, PJ12)
    kb.add_clause(~J17, PJ17)
    kb.add_clause(~PJ2, J2)
    kb.add_clause(~PJ7, J7)
    kb.add_clause(~PJ12, J12)
    kb.add_clause(~PJ17, J17)
```

8.7 Appendix G - pseudocode of the logical strategies

```
if my turn:
        if the card is in non-trump moves:
                if has jack in hand
                        play jack
                elif has queen in hand and no marriage possibility:
                        play queen
                elif has king in hand and no marriage possibility:
                        play king
                else:
                        play the highest card in hand so trump


else:
```

```
if opponent plays trump card:
        if we have trump card:
                play trump higher than opponent's
        else:
                if the card is in non-trump moves:
                        if has jack in hand
                                play jack
                        elif has queen in hand and no marriage possibility:
                                play queen
                        elif has king in hand and no marriage possibility:
                                play king
                        else:
                                play the highest card in hand so trump


else:
        if opponent plays 10
                play A of the same suit or trump suit J, Q or K
        elif opponent plays A
                play trump suit J, Q or K, or 10 or A
        else:
                play higher card of the same suit or any J
```

8.8 Appendix H - tournaments

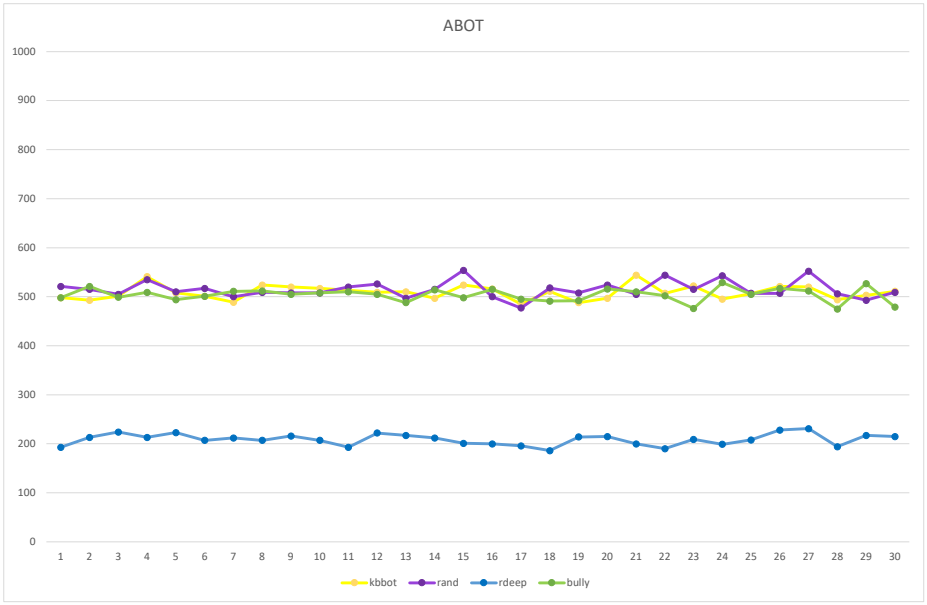

Fig. 5. Abot performance against kbbot, rand, rdeep and bully can be observed.

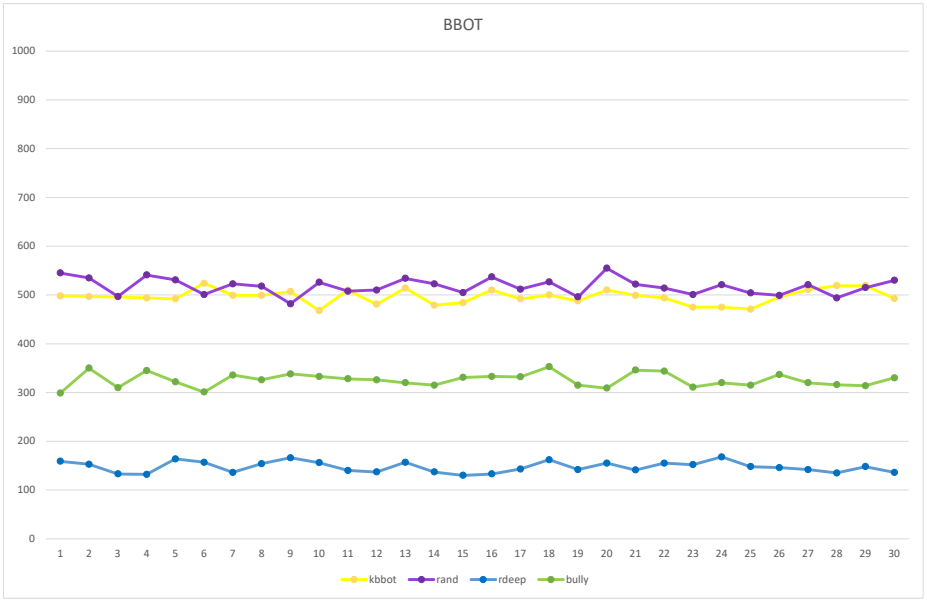

Fig. 6. Bbot performs slightly poorer against bully, and even weaker when it comes to rdeep than other our bots.

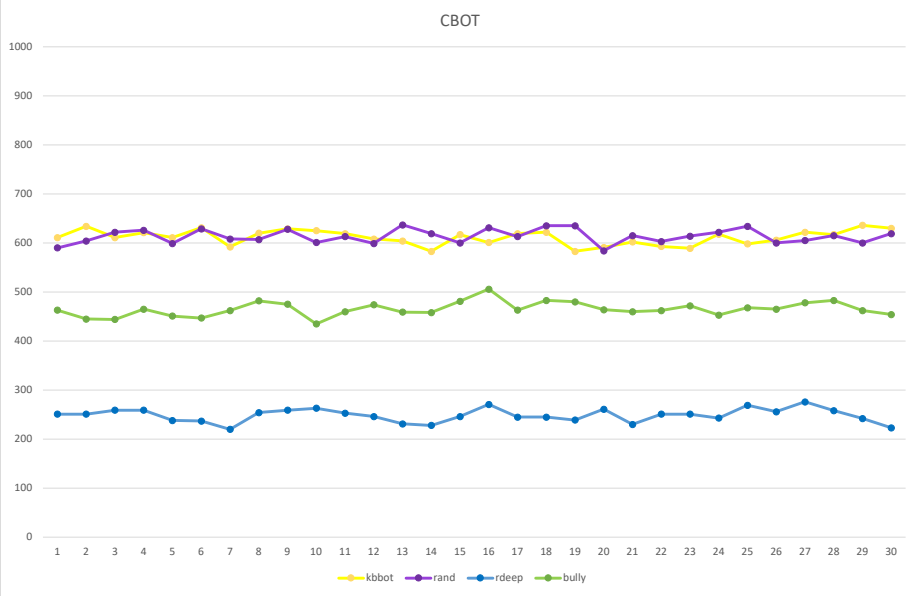

Fig. 7. Cbot performs rather poorly against rdeep and bully, but rand and kbbot is highly enhanced with 100 points more than previous bots.

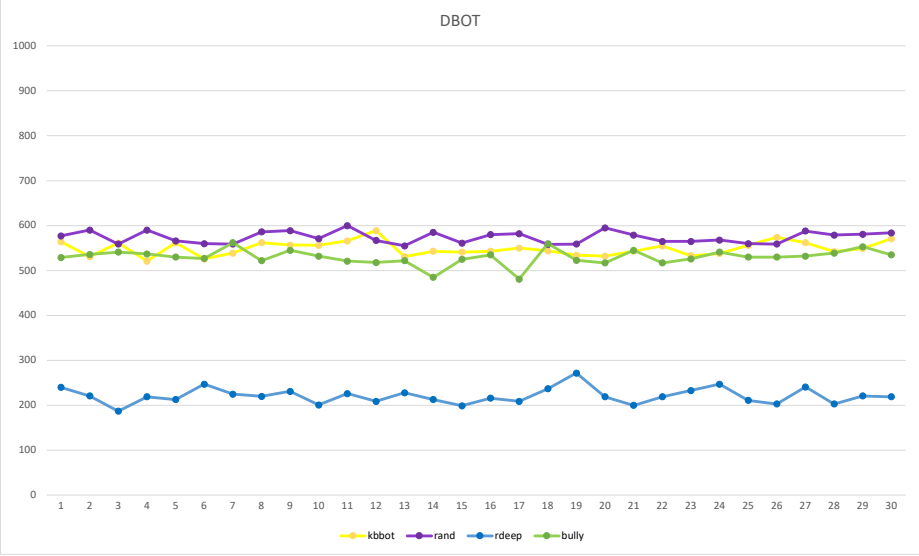

Fig. 8. Dbot's performance in tournaments against bully is strengthened significantly.

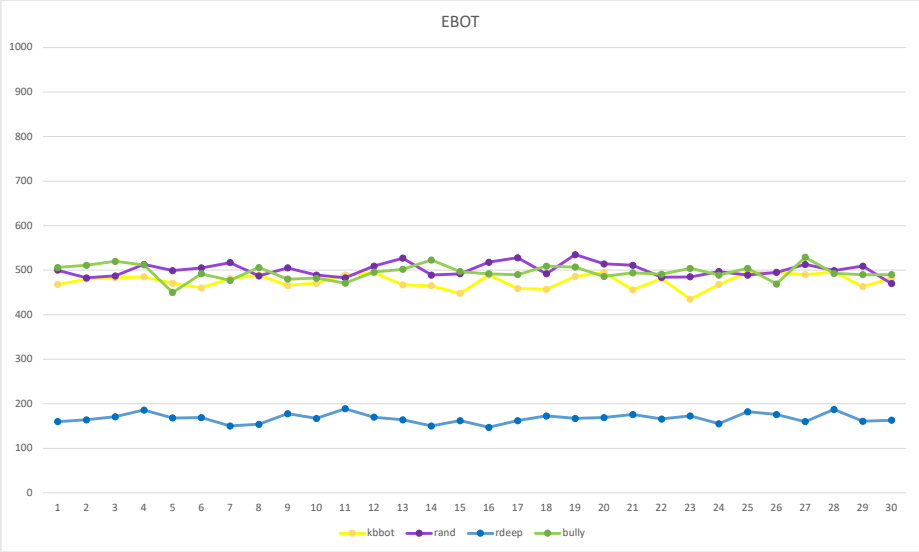

Fig. 9. Games won in tournaments with kbbot, rand and bully are almost even when it comes to opposing ebot. Surprisingly enough, its performance against rdeep is, together with bbot, insufficient.