The benefits of an extended feature space in a machine learning bot for the game Schnapsen

Lucas Faijdherbe, Daan Schrage and Ruben van der Ham

Intelligent Systems
Vrije Universiteit Amsterdam

January 25, 2018

Abstract

Abstract paragraph with abstract shit about abstract things

1 Introduction

Artificial Intelligence is becoming a huge subject, and we see it appearing more and more in daily life: voice assistants on smartphones, self-driving Tesla's, AlphaGo defeating Go masters, and so on. ... We can make a distinction between strong and weak AI [1]: *weak AI* is focused on mastering problems in one limited area, whereas *strong AI* is considered to be Artificial Intelligence with *actual* intelligence, and even self-awareness. For this paper, we will be focussing on weak AI in games, more specifically in the game of Schnapsen. In section 2 we will describe Schnapsen in more detail, alongside the bots used for the game. The research question will be described in section 3. In section 4, we will talk more about the experiment. ...

2 Background Information

2.1 Schnapsen

Schnapsen is an Austrian card game, played with 20 cards; Only the Jacks, Queens, Kings, 10s and Aces are in the deck. The goal of the game is to win tricks, and to be the first player to score 66 points. Each game has a trump suit, which is determined randomly. A card from the trump suit lets you automatically win the trick, provided that there is no higher trump card in the game.

The game has a multi-agent (we have an opponent), deterministic (the environment is determined by the current state and executed action), static (the environment does not change when an agent is choosing an action) and discrete (the game is turn-based) environment. The game consists of two phases: In phase one, when there are still cards in stock, we have an imperfect information game. In phase two, when the stock is depleted, we have a perfect information game, since we know exactly what the cards of our opponent are. The point count in the game is as follows:

J	**Q**	**K**	**10**	**A**
2	3	4	10	11

A lot of points can also be won by marriages. If the player is leading the trick and has a king or a queen of the same suit, it can play one card, and show the other, so the opponent knows that there is a marriage.

Normal marriage (non-trump suit)	**Royal marriage (trump suit)**
20	40

The game starts in phase one, an imperfect-information game. Each agent has 5 cards. Player 1 plays a card, thereby determining the suit that has to be played. To win the trick, player 2 has to play a higher-ranking card from the same suit, or a trump card. Player 2 does not have to follow suit in phase 1, but he/she will lose the trick in that case. The winner of the trick now becomes player one and gets the cards, with the amount of points the two cards represent. This goes on until the stock is depleted; then we enter phase 2, a perfect-information game. Player 2 now has to follow suit. If it cannot do that, the player has to trump, and otherwise play any card. If a player reaches 66 points or more, the game is over.

2.2 Intelligent Systems Schnapsen Framework

An API in Python has been provided, together with a framework for the Schnapsen game. Furthermore, we have two scripts: one where bots can play against each other in a single game where all moves in the game can be seen, and a script where bots can play a tournament against each other - with a specifiable number of the number of games to be played - and where the number of won games per robot can be seen. There are already multiple bots already present in the framework to play against:

Minimax / Alphabeta: This bot only works in a perfect-information game (phase 2 of Schnapsen). It recursively looks through all the states, and decides what the best move to play will be, provided that the opponent plays optimally, and will try to minimize the outcome of the game for the player.

Bully: Plays by a defined set of rules. If it has a trump card, it will play that. Otherwise, it will play a card of the same suit player 1 played. If it cannot do either of those things, it will play the highest ranking card of any suit.

Rand: Chooses a random move out of all of the legal moves, and plays that.

Rdeep: Functions quite like a Monte-Carlo sampling bot. It looks through multiple states, and averages the heuristics of a state, thereby ranking the moves. It then picks one of the highest ranking moves to play.

KBbot: Logical reasoning bot. The strategy can be defined in propositional logic. The current strategy of the bot is to always play cheap cards (i.e kings, queens, jacks) first.

2.3 Machine Learning Bots

For this paper, we will use machine learning bots with different set features to determine what the best features are.

3 Research Question

In our research, we want to find out if an extended feature space in a machine learning bot will improve its performance in Schnapsen. There is already a machine learning bot present, with what we will call, the *default feature space*, provided by the framework. Our goal is to extend and improve this feature space, and see if the bot will have a better performance. Our hypothesis is that an extended feature space will lead to a better performance, since .....

The *default feature space* contains the following features:

- Perspective information
- Player 1's point count
- Player 2's point count
- Player 1's pending points
- Player 2's pending points
- The trump suit
- The phase
- The stock size
- The leader
- Whose turn it is currently
- The played card of the opponenet

4 Experimental setup

For the testing of the bots, multiple Schnapsen tournaments were held. Each bot played a 100 games against every other bot. The competing bots were:

- Minimax
- Bully
- Rand
- Rdeep
- KBBot
- ML

- MLAdvanced

Both ML, and MLAdvanced were trained on the same bot. First, bully, then rand, then rdeep, then kbbot, etc.

5 Results

6 Findings

7 Conclusion

8 References

1. http://humanparagon.com/strong-and-weak-ai/

A Worksheet I

1. The *rdeep* bot performs best, winning 20 out of 30 games.
   Output:

```
Results:
bot <bots.rand.rand.Bot instance at 0x101a70fc8>: 7 wins
bot <bots.bully.bully.Bot instance at 0x101a71098>: 3 wins
bot <bots.rdeep.rdeep.Bot instance at 0x101a71170>: 20 wins
```

2. The bully bot operates as follows:
   - If it has a trump card, it will play that.
   - If it does not have a trump card, it will play a card from the same suit
   - If it cannot do either of those things, it will choose the highest ranking card of any suit.
3. Using this strategy means that you will try to get as many points as possible each move. However, this might also mean that early in the game, the bot will play valuable cards in order to get a high point ratio, but later in the game it has no good cards left.
4. Have the bots complete a large number of tournaments (preferably n ¿ 30), see if the rdeep bot still outperforms the rand bot, and how big the difference in wins is. If there is a significant difference, we can assume that rdeep is better than rand.
5. Bot:

6. The heat map: - Insert pdf here - This heatmap means:

7. The added line:

   ```
   value, move = self.value(next_state, depth + 1)
   ```

   Output:

   ```
   python tournament.py -p rand,minimax -s 2 -r 10
   Playing 10 games:
   Played 1 out of 10 games (10%): [0, 1]
   Played 2 out of 10 games (20%): [0, 2]
   Played 3 out of 10 games (30%): [0, 3]
   Played 4 out of 10 games (40%): [0, 4]
   Played 5 out of 10 games (50%): [1, 4]
   Played 6 out of 10 games (60%): [2, 4]
   Played 7 out of 10 games (70%): [2, 5]
   Played 8 out of 10 games (80%): [3, 5]
   Played 9 out of 10 games (90%): [4, 5]
   Played 10 out of 10 games (100%): [4, 6]
   Results:
   bot <bots.rand.rand.Bot instance at 0x102a70f80>: 4 wins
   bot <bots.minimax.minimax.Bot instance at 0x102a731b8>: 6 wins
   ```

8.

9. The heuristic is a value between -1.0 and 1.0, derived from the point ratio of the current player. If the player has a 1.0 value, the player has a definite win. If the value is -1.0, the opponent has a definite win.

B Worksheet II

1. The added clause that makes the knowledge base unsatisfiable:

   ```
   kb.add_clause(~B, ~C)
   ```

2. The three clauses converted to CNF:

   - A V B
   - -B V A
   - -A V C
   - -A V D

   The code that creates the variables and the knowledge base:

```
# Define our symbols
A = Boolean('A')
B = Boolean('B')
C = Boolean('C')
D = Boolean('D')

# Create a new knowledge base
kb = KB()

# Add clauses
kb.add_clause(A, B)
kb.add_clause(~B, A)
kb.add_clause(~A, C)
kb.add_clause(~A, D)
```

Because the knowledge base entails A & C & D, all three have to be true. For now, the model has two solutions:

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

B can be whatever it wants, but A, B and D have to be true.

3. Three constraints have to be met:

   I $X = Y$

   II $X + Y > 2$

   III $X + Y < 5$

   The model is satisfiable, as long as: $1 < x < 2.5$ and $1 < y < 2.5$

4. We now have three models:

```
(a) [x = y] = True, [x + y > 2] = True, [x + y < 5] = True
(b) [x = y] = True, [x + y > 2] = True, [x + y < 5] = False
(c) [x = y] = True, [x + y > 2] = False, [x + y < 5] = True
```

   The three constraints per model:

   (a) This model already has three constraints.
   (b) Only the last clause has to change, namely to $[x + y >= 5]$= True
   (c) Only the second clause has to change, namely to$[x + y <= 2] =$ True.

   Now we end up with this:

```
(a) [x = y] = True, [x + y > 2] = True, [x + y < 5] = True
(b) [x = y] = True, [x + y > 2] = True, [x + y >= 5] = True
(c) [x = y] = True, [x + y <= 2] = True, [x + y < 5] = True
```

5. What we want, is to find points on the line $x = y$, where the points either lie between $x + y > -5$ and $x + y < -2$, or where the points lie between $x + y > 2$ and $x + y < 5$.

   The clauses we need to add are:

   - $x == y$
   - $x + y > 2 V x + y < -2$
   - $x + y > -5 V x + y < 5$
   - $x + y > -5 V x + y > 2$
   - $x + y < -2 V x + y < 5$

   The code:

```
# Define our integer symbols
x = Integer('x')
y = Integer('y')

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

kb = KB()

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

6.

C Worksheet III

1. The added code:

```
value = self.heuristic(next_state)

p1_points = state.get_points(1)
```

```
p2_points = state.get_points(2)

p1_pending_points = state.get_pending_points(1)

p2_pending_points = state.get_pending_points(2)

trump_suit = state.get_trump_suit()

phase = state.get_phase()

stock_size = state.get_stock_size()

leader = state.leader()

whose_turn = state.whose_turn()

opponents_played_card = state.get_opponents_played_card()
```

The result of the played tournament against rand and bully:

```
bot <bots.ml.ml.Bot instance at 0x102271290>: 16 wins
bot <bots.rand.rand.Bot instance at 0x1022711b8>: 4 wins
bot <bots.bully.bully.Bot instance at 0x1059b71b8>: 10 wins
```

2. We let the mlbot learn by playing against rdeep. But since the games took way longer than the games of mlbot vs rand, we set the number of games to 1000 (Which still took quite some time).

   We get a lower output now:

```
bot <bots.ml.ml.Bot instance at 0x102a73248>: 15 wins
bot <bots.rand.rand.Bot instance at 0x102a73170>: 7 wins
bot <bots.bully.bully.Bot instance at 0x10599df80>: 8 wins
```

   ML now wins 15 times, where it won 16 times on the first tournament. This might have to do with the training set being smaller, and it was trained against rdeep, which does not compete in the tournament. However, the difference is not comparably worse.

   We decided to train the mlbot again against rdeep, but now with 5000 games. Even after this long wait, the robot does not play significantly better or worse.

```
bot <bots.ml.ml.Bot instance at 0x102a71248>: 14 wins
bot <bots.rand.rand.Bot instance at 0x102a71170>: 5 wins
bot <bots.rdeep.rdeep.Bot instance at 0x105aa2cf8>: 11 wins
```