Intelligent Systems 2020: 3rd practical assignment

Games and Adversarial Search

An introduction to the Schnapsen platform

Your name: Seeun Park

Your VUnetID: spk760

If you do not provide your name and VUnetID we will not accept your submission.

Learning objectives

At the end of this exercise you should be able to use the Schnapsen platform, run basic games between agents, and run tournaments in order to evaluate rational agents (also called bots). You should also be able to identify the working patterns of the MiniMiax algorithm in this platform and the improvements with Alpha/Beta pruning.

1. Understand the main functionality of the Schnapsen platform (playing games and running tournements)
2. Implement your own rational agents (bots)
3. Follow the individual steps and explain the MiniMax algorithm
4. Make small modifications of the code to see the effect on the search algorithms
5. Make small adaptations to the algorithm to study the computational properties

Practicalities

Follow this Notebook step-by-step.

Of course, you can do the exercises in any Programming Editor of your liking. But you do not have to. Feel free to simply write code in the Notebook. Please use your studentID+Assignment3.ipynb as the name of the Notebook.

Note: unlike the courses dedicated to programming we will not evaluate the style of the programs. But we will, however, test your programs on other data that we provide, and your program should give the correct output to the test-data as well.

As was mentioned, the assignment is graded as pass/fail. To pass you need to have either a full working code or an explanation of what you tried and what didn't work for the tasks that you were unable to complete (you can use multi-line comments or a text cell).

Initialising

First, let us make sure the necessary packages are installed, and imported. Run the following code:

```
In [1]: import sys, random

        from api import State, engine, util
```

Playing the first games

The basic engine comes with three basic bots: rand, bully and rdeep (the rest you can ignore for now). To try them out, just run the following bit of code.

```
In [2]: # Choose your first player
        player1 = "rand"
        player2 = "bully"
        # Decide in which phase you want to start the game.
        startphase = 1
        # Decide whether you want verbose output or not
        verbose=True

        #And here you run a game on the engine.
        engine.play(util.load_player(player1),util.load_player(player2), state=State.generate(p
```

```
player1: <bots.rand.rand.Bot object at 0x0000025628611C70>
player2: <bots.bully.bully.Bot object at 0x0000025628611CD0>
*   Player 2 plays: JH
The game is in phase: 1
Player 1's points: 0, pending: 0
Player 2's points: 0, pending: 0
The trump suit is: H
Player 1's hand: AC KC JC KH JS
Player 2's hand: 10C AD JH AS KS
There are 10 cards in the stock
Player 2 has played card: J of H

*   Player 1 plays: JS
The game is in phase: 1
Player 1's points: 0, pending: 0
Player 2's points: 4, pending: 0
The trump suit is: H
Player 1's hand: AC KC QC JC KH
Player 2's hand: 10C AD 10D AS KS
There are 8 cards in the stock

*   Player 2 plays: AS
The game is in phase: 1
Player 1's points: 0, pending: 0
Player 2's points: 4, pending: 0
The trump suit is: H
Player 1's hand: AC KC QC JC KH
Player 2's hand: 10C AD 10D AS KS
There are 8 cards in the stock
Player 2 has played card: A of S

*   Player 1 plays: QC
The game is in phase: 1
Player 1's points: 0, pending: 0
Player 2's points: 18, pending: 0
The trump suit is: H
Player 1's hand: AC KC JC KH QS
Player 2's hand: 10C AD 10D AH KS
There are 6 cards in the stock
```

```
*   Player 2 plays: AH
The game is in phase: 1
Player 1's points: 0, pending: 0
Player 2's points: 18, pending: 0
The trump suit is: H
Player 1's hand: AC KC JC KH QS
Player 2's hand: 10C AD 10D AH KS
There are 6 cards in the stock
Player 2 has played card: A of H

*   Player 1 plays: QS
The game is in phase: 1
Player 1's points: 0, pending: 0
Player 2's points: 32, pending: 0
The trump suit is: H
Player 1's hand: AC KC JC QD KH
Player 2's hand: 10C AD 10D 10H KS
There are 4 cards in the stock

*   Player 2 plays: 10H
The game is in phase: 1
Player 1's points: 0, pending: 0
Player 2's points: 32, pending: 0
The trump suit is: H
Player 1's hand: AC KC JC QD KH
Player 2's hand: 10C AD 10D 10H KS
There are 4 cards in the stock
Player 2 has played card: 10 of H

*   Player 1 plays: JC
The game is in phase: 1
Player 1's points: 0, pending: 0
Player 2's points: 44, pending: 0
The trump suit is: H
Player 1's hand: AC KC QD JD KH
Player 2's hand: 10C AD 10D 10S KS
There are 2 cards in the stock

*   Player 2 plays: AD
The game is in phase: 1
Player 1's points: 0, pending: 0
Player 2's points: 44, pending: 0
The trump suit is: H
Player 1's hand: AC KC QD JD KH
Player 2's hand: 10C AD 10D 10S KS
There are 2 cards in the stock
Player 2 has played card: A of D

*   Player 1 plays: AC
The game is in phase: 2
Player 1's points: 0, pending: 0
Player 2's points: 66, pending: 0
The trump suit is: H
Player 1's hand: KC QD JD KH QH
Player 2's hand: 10C 10D KD 10S KS
There are 0 cards in the stock

Game finished. Player 2 has won, receiving 3 points.
```

```
Out[2]: (2, 3)
```

Running engine.play provides some textual output to show how the game is progressing. At every plie (half a turn) you will see what move the player made and a concise overview of the board.

Something like this (when you run a game in verbose mode):

> Player 1 plays: KC The game is in phase: 1
>
> Player 1's points: 21, pending: 0
>
> Player 2's points: 25, pending: 0
>
> Player 1's hand: 10C JC AD QD QH
>
> Player 2's hand: AC KD 10H KS QS
>
> There are 2 cards in the stock

The first line signifies that player 1 has played the King of Clubs card. Internally these cards are represented by indices from 0 to 19. To make the translation between indices and card names, use util.get_card_name(index), which returns the rank and the suit of the card as a tuple. Alternatively, use util.get_rank(index) and util.get_suit(index) for each property alone. In this case it is a King of Clubs. Have a look at the GitHub readme or at the top of __deck.py to see the convention used for encoding cards into indices.

In [3]:
```
util.get_card_name(2)
```

Out[3]:
```
('K', 'C')
```

You can also run the Python programmes provided from the command line, or in your favourite editor.
Run

> python play.py -1 rand -2 bully

to run the rand bot against the bully bot, or

> python play.py -h

to see other options.

There is a lot of randomness involved in the game when the cards are distributed to the players and the pile. To get an accurate sense of whether one player is better than another, you'll need to play a number of different games. The following code will play a tournament between bully and rand where every pair of participants plays 10 matches.

In [4]:
```
botnames = []
verbose = False
myphase = 1
myrepeats = 10

# Create player 1
player1 = util.load_player("rand")
player2 = util.load_player("bully")

bots = [player1,player2]

n = len(bots)
wins = [0] * len(bots)
matches = [(p1, p2) for p1 in range(n) for p2 in range(n) if p1 < p2]

totalgames = (n*n - n)/2 * myrepeats
```

```
playedgames = 0

print('Playing {} games:'.format(int(totalgames)))
for a, b in matches:
    for r in range(myrepeats):

        if random.choice([True, False]):
            p = [a, b]
        else:
            p = [b, a]

        # Generate a state with a random seed
        state = State.generate(phase=myphase)

        winner, score = engine.play(bots[p[0]], bots[p[1]], state, 1000, verbose, True)

        if winner is not None:
            winner = p[winner - 1]
            wins[winner] += score

        playedgames += 1
        print('Played {} out of {:.0f} games ({:.0f}%): {} \r'.format(playedgames, tota

print('Results:')
for i in range(len(bots)):
    print('    bot {}: {} points'.format(bots[i], wins[i]))
```

```
Playing 10 games:
Played 1 out of 10 games (10%): [0, 3]
Played 2 out of 10 games (20%): [2, 3]
Played 3 out of 10 games (30%): [3, 3]
Played 4 out of 10 games (40%): [3, 6]
Played 5 out of 10 games (50%): [4, 6]
Played 6 out of 10 games (60%): [4, 8]
Played 7 out of 10 games (70%): [4, 11]
Played 8 out of 10 games (80%): [4, 14]
Played 9 out of 10 games (90%): [4, 16]
Played 10 out of 10 games (100%): [4, 19]
Results:
    bot <bots.rand.rand.Bot object at 0x0000025628696190>: 4 points
    bot <bots.bully.bully.Bot object at 0x0000025628696F40>: 19 points
```

Task 1:

The previous code runs a tournament between rand and bully, but you can adapt the script by testing the performance of these bots with the third default bot, rdeep, as the opponent. The general idea of rdeep was extensively discussed under the header PIMS (Perfect Information Monte Carlo Sampling). Report in the following Markdown Cell on the results you get from two-player tournaments including rdeep, rand and bully (rdeep vs. rand; rdeep vs. bully). Describe which games you played, and who won. Add the (non-verbose) output of the script.

When playing all the combinations for the bots, rand scored lower than the two other bots. bully won over rand, however was unable to wind rdeep. Thus, rdeep would have the highest possibility to win against the other two bots.

Playing 10 games: Played 1 out of 10 games (10%): [0, 2] Played 2 out of 10 games (20%): [0, 5] Played 3 out of 10 games (30%): [0, 7] Played 4 out of 10 games (40%): [0, 9] Played 5 out of 10 games (50%): [2, 9] Played 6 out of 10 games (60%): [3, 9] Played 7 out of 10 games (70%): [3, 11]

Played 8 out of 10 games (80%): [3, 14] Played 9 out of 10 games (90%): [3, 17] Played 10 out of 10 games (100%): [4, 17] Results: bot <bots.rand.rand.Bot object at 0x0000018C3539D8E0>: 4 points bot <bots.bully.bully.Bot object at 0x0000018C3539D880>: 17 points

Task 2:

The previous code runs a tournament between two bots only, but you can easily adapt the script above to play round-robin tournament. All you have to do is to add a third player to the bots list. Report in the following Markdown Cell on the results you get from three-player tournament including rdeep, rand and bully. Add the (non-verbose) output of the script. Report on the results of the tournament and try to explain in your own words what do the results mean.

Each bot plays 10 games against each bot, therefore a total of 30 games have been played. The outcome displays the total sum of the points from the games, therefore presenting the outcome more efficiently in comparison to having to run the game 10 times for each bot manually.

Playing 30 games: Played 1 out of 30 games (3%): [2, 0, 0] Played 2 out of 30 games (7%): [2, 1, 0] Played 3 out of 30 games (10%): [2, 4, 0] Played 4 out of 30 games (13%): [3, 4, 0] Played 5 out of 30 games (17%): [4, 4, 0] Played 6 out of 30 games (20%): [4, 7, 0] Played 7 out of 30 games (23%): [5, 7, 0] Played 8 out of 30 games (27%): [6, 7, 0] Played 9 out of 30 games (30%): [7, 7, 0] Played 10 out of 30 games (33%): [7, 9, 0] Played 11 out of 30 games (37%): [7, 9, 2] Played 12 out of 30 games (40%): [7, 9, 5] Played 13 out of 30 games (43%): [7, 9, 8] Played 14 out of 30 games (47%): [7, 9, 9] Played 15 out of 30 games (50%): [7, 9, 12] Played 16 out of 30 games (53%): [7, 9, 14] Played 17 out of 30 games (57%): [7, 9, 15] Played 18 out of 30 games (60%): [7, 9, 17] Played 19 out of 30 games (63%): [8, 9, 17] Played 20 out of 30 games (67%): [8, 9, 18] Played 21 out of 30 games (70%): [8, 10, 18] Played 22 out of 30 games (73%): [8, 12, 18] Played 23 out of 30 games (77%): [8, 12, 21] Played 24 out of 30 games (80%): [8, 12, 22] Played 25 out of 30 games (83%): [8, 12, 24] Played 26 out of 30 games (87%): [8, 12, 25] Played 27 out of 30 games (90%): [8, 12, 26] Played 28 out of 30 games (93%): [8, 12, 28] Played 29 out of 30 games (97%): [8, 12, 30] Played 30 out of 30 games (100%): [8, 12, 32] Results: bot <bots.rand.rand.Bot object at 0x0000018C3539DAF0>: 8 points bot <bots.bully.bully.Bot object at 0x0000018C3539DA90>: 12 points bot <bots.rdeep.rdeep.Bot object at 0x0000018C3539DC10>: 32 points

Inspecting the code

Now let's have a look at how the bots work. Open the file bots/rand/rand.py in PyCharm. Each bot is a class called Bot.

> We will use more advanced features of Python than what you have seen so far in Introduction to Python (don’t worry), so for more details have a look at: https://www.learnpython.org/en/Classes_and_Objects

The rand bot contains nothing but an empty constructor, and one method: get_move(self, state). This is the only method you need to implement to get a working bot. It receives a description of a game state, and returns a move. A move is always a pair of two elements, each of which can be either an integer or None. Note that it is not an option to pass, therefore (None, None) is not a valid move.

As you can see, in the rand bot, the state object does almost all the work: state.moves() gives you a list of legal moves. The rand Bot simply makes a random choice from this list using the function random.choice() from the python library.

If you want to see what happens when you make a given move, just do next_state = state.next(move) And you get a state representing the outcome.

What else can the State object do for you? You can look at the code in api/_state.py.

bully.py

Bully is a deterministic bot: given the same state it will always do the same thing. We've removed part of the explanation from the comments.

Task 3:

Have a look at the code: describe in your own words what strategy does the bully bot use?

There are a total of three strategies. Bully checks the moves available, in order to play the trump suit to win the trick. When the above is not applicable, bully will look at the card that has been played by the player to play the same suit. However, when bully matches the suit it does not give the opportunity to check whether the point is higher. The final strategie is used when bully does not have any suits that match, therefore plays the highest point card it has.

rdeep.py

The lectures discuss the hill-climbing strategy: look one move ahead and pick the move that leads to the best heuristic. The heuristics we use is the ratio of the player points w.r.t. to the total points currently assigned in the game. The higher this value, the better the state is for us. Imagine doing hill-climbing with this heuristic. This strategy would not work here. Why not?

In order to avoid this issue, we need to loook further ahead than the hill climbing strategy does. rdeep.py does this in the simplest way we could think of. Make eight random moves and look at the value of the resulting state. Do this a few times and average the values found. This method is called Perfect-Information Monte-Carlo Sampling (PIMC).

You just ran a tournament between rdeep and the other two bots. Most likely, rdeep will have won a few more games. But does the difference really mean rdeep is better? It might just be that rdeep is no better than rand and won by pure luck.

Task 4

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

Repeat the games a number of times and create a graph out of the outcome and compare another set of repeated games played the same number of times and compare that graph to the others as well.

mybot.py

It's time to write your own bot. Think of a simple strategy that is easy to implement. To create the bot follow these steps:

1. Create the directory bots/mybot (in the directory, not this notebook!)
2. Add an empty file **init**.py, or copy it from one of the other bot directories.
3. Copy rand.py to the directory mybot, and rename it to mybot.py
4. Change the implementation of get_move(state). Keep the method signature (line 16) exactly as it is.

Make sure your bot always returns either a pair elements that are each either int or None. Try playing a tournament against rand. See if you can get a decent margin.

If your bot has parameters (like a search depth, or a pre-programmed probability of doing nothing) you can add these to the constructor. Have a look at rdeep.py to see how this is done.

To get some examples of how to talk to the API, see the README.md

Task 5

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

Please write your code here (in raw text, to avoid an error), as well as the results in the following cell:

def get_move(self, state): # type: (State) -> tuple[int, int] """ Function that gets called every turn. This is where to implement the strategies. Be sure to make a legal move. Illegal moves, like giving an index of a card you don't own or proposing an illegal mariage, will lose you the game. TODO: add some more explanation :param State state: An object representing the gamestate. This includes a link to the states of all the cards, the trick and the points. :return: A tuple of integers or a tuple of an integer and None, indicating a move; the first indicates the card played in the trick, the second a potential spouse. """ # All legal moves moves = state.moves() chosen_move = moves[0] #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) #If I hold two trump_suit cards, play the highest if len(moves_trump_suit) > 1: moves_for_trump = moves_trump_suit[0] if move[0] is not None and move[0] % 5 <= chosen_move[0] % 5: chosen_move = move return chosen_move #Get move with lowest rank available, of any suit 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 return chosen_move -- Output player1: player2: Bot returned a move None that was not a pair (i.e. (2,3)) Game finished. Player 1 has won, receiving 3 points. (1, 3) Seems to be error in output?

MINIMAX Adversarial Games

Now we will build some bots that search the game tree by using the MiniMax algorithm and show genuinely intelligent behavior. Because we only have partial information, and as there would be too many belief-states, we play these bots on the 2nd phase of the game only, when the stock has been exhausted, such that the state of all cards is known and no assumptions have to be made.

All you need to do to finish the minimax bot is to add one line of code on line 58. The skeleton given in the code differs slightly from the pseudocode from the lecture in that instead of having two sub-methods, all is implemented in one method. Take your time to really understand the minimax algorithm, recursion, and the rest of the code. The pseudocode would then look like as follows

function VALUE(state) returns a utility value, move # First we initialise the values for best_value (which we need to calculate the MIN and MAX later. best_value = -infinity if MAXMOVE, +infinity if MINMOVE # then we check for the terminal states of our search tree, either if there are no more successors, or in case we ran over the maximal depths (set in minimax.py) if TERMINAL-TEST(state) then return UTILITY(state) if DEPTH = max_depth then return heuristic_value(state) # Now we calculate the values of the successor states, and choose the maximal one (or minimal) depending in which state we are. for move in SUCCESSORS(state) do next_move = SUCCESSOR(move) value = VALUE(next_move) # This is a recursive call, in which we calculate the optimal value for the opponent to play if MAXMOVE # This is information provided by the game-engine as a function in the state class. best_value is now calculated as the maximum between the value of the next_move and the previously calculated value (!) best_move is set to the the move with the current best value (!) else best_value is now calculated as the minimum between the value of the next_move and the previously calculated value (!) best_move is set to the the move with the current best value (!) return best_value, best_move

Task 6

Implement the parts of the algorithm marked "???" in the code of the minimax.py program in the bot directory. Please write down your new code in the following raw cell.

Once you have done this, you can check whether you algorithm runs correctly.

for move in moves: next_state = state.next(move) value, _ = self.value(next_state) if maximizing(state): # we are in the Maximize part of the algorithm if value > best_value: best_value = value best_move = move else: # we are in the Maximize part of the algorithm if value < best_value: best_value = value best_move = move return best_value, best_move

In [5]:
```python
# Choose your first player
player1 = "bully"
player2 = "minimax"
# Decide in which phase you want to start the game.
startphase = 2
# Decide whether you want verbose output or not
verbose=True

#And here you run a game on the engine.
engine.play(util.load_player(player1),util.load_player(player2), state=state.generate(p
```

```
player1: <bots.bully.bully.Bot object at 0x00000256286965B0>
player2: <bots.minimax.minimax.Bot object at 0x00000256286D3AF0>
*   Player 1 plays: 10D
The game is in phase: 2
Player 1's points: 39, pending: 0
Player 2's points: 39, pending: 0
The trump suit is: S
Player 1's hand: KC QC JC 10D JH
Player 2's hand: QD JD QH 10S JS
There are 0 cards in the stock
Player 1 has played card: 10 of D

*   Player 2 plays: JD
The game is in phase: 2
Player 1's points: 51, pending: 0
Player 2's points: 39, pending: 0
The trump suit is: S
Player 1's hand: KC QC JC JH
Player 2's hand: QD QH 10S JS
There are 0 cards in the stock
```

```
*   Player 1 plays: KC
*   Player 1 melds a marriage between KC and QC
The game is in phase: 2
Player 1's points: 51, pending: 20
Player 2's points: 39, pending: 0
The trump suit is: S
Player 1's hand: KC QC JC JH
Player 2's hand: QD QH 10S JS
There are 0 cards in the stock
Player 1 has played card: K of C

*   Player 2 plays: JS
The game is in phase: 2
Player 1's points: 51, pending: 20
Player 2's points: 45, pending: 0
The trump suit is: S
Player 1's hand: QC JC JH
Player 2's hand: QD QH 10S
There are 0 cards in the stock

*   Player 2 plays: QD
The game is in phase: 2
Player 1's points: 51, pending: 20
Player 2's points: 45, pending: 0
The trump suit is: S
Player 1's hand: QC JC JH
Player 2's hand: QD QH 10S
There are 0 cards in the stock
Player 2 has played card: Q of D

*   Player 1 plays: QC
The game is in phase: 2
Player 1's points: 51, pending: 20
Player 2's points: 51, pending: 0
The trump suit is: S
Player 1's hand: JC JH
Player 2's hand: QH 10S
There are 0 cards in the stock

*   Player 2 plays: QH
The game is in phase: 2
Player 1's points: 51, pending: 20
Player 2's points: 51, pending: 0
The trump suit is: S
Player 1's hand: JC JH
Player 2's hand: QH 10S
There are 0 cards in the stock
Player 2 has played card: Q of H

*   Player 1 plays: JH
The game is in phase: 2
Player 1's points: 51, pending: 20
Player 2's points: 56, pending: 0
The trump suit is: S
Player 1's hand: JC
Player 2's hand: 10S
There are 0 cards in the stock

*   Player 2 plays: 10S
The game is in phase: 2
Player 1's points: 51, pending: 20
Player 2's points: 56, pending: 0
The trump suit is: S
Player 1's hand: JC
Player 2's hand: 10S
There are 0 cards in the stock
Player 2 has played card: 10 of S

*   Player 1 plays: JC
The game is in phase: 2
Player 1's points: 51, pending: 20
Player 2's points: 68, pending: 0
The trump suit is: S
Player 1's hand:
Player 2's hand:
There are 0 cards in the stock

Game finished. Player 2 has won, receiving 1 points.
```

Out[5]:
```
(2, 1)
```

Heuristics

What heuristic do these implementations use? Explain the heuristic function in your own words:

The heuristic function implements an expected value of the state.

Adapt the search depth (in the constructor of the Minimax bot, and adapt the Minimax procedure by uncommenting the respective function) to smaller values (search depths). Report on the performance of these different versions of using the heuristics against some other players (like rdeep and rand), and see if the performance (nr. of games won) differs with the different look-aheads.

Task 7

Report on your experiments with Minimax as compared to previous bots (or a copy of MiniMax with a different search depths)

The heuristics allowed Minimax to win rand, bully and rdeep the majority of the time.

Alphabeta

Finally let us look at Alphabeta pruning. Alphabeta pruning is a technique to make minimax faster. If implemented correctly, it will do exactly the same thing minimax does, but skip large parts of the search tree. We've provided a basic implementation in bots/alphabeta/alphabeta.py.

Task 8

Once again, one crucial bit of the implementation are missing, the decision on when to prune. Finish the implementation of the alphabeta bot. Copy the line of code you adapted in the skeleton file alphabeta.py into the following cell:

if alpha > beta: break

The following programme lets you see if you implemented alphabeta and minimax correctly. Run it, and check the outcome.

In [6]:
```python
from api import State, util
import random, time
```

```
from bots.alphabeta import alphabeta
from bots.minimax import minimax

REPEATS = 10
DEPTH = 6

ab = alphabeta.Bot(randomize=False, depth=DEPTH)
mm = minimax.Bot(randomize=False, depth=DEPTH)

mm_time = 0
ab_time = 0

# Repeat
for r in range(REPEATS):

    # Repeat some more
    for r2 in range(REPEATS):

        # Generate a starting state
        state = State.generate(phase=2)

        # Ask both bots their move
        # (and time their responses)

        start = time.time()
        mm_move = mm.get_move(state)
        mm_time += (time.time() - start)

        start = time.time()
        ab_move = ab.get_move(state)
        ab_time += (time.time() - start)


        if mm_move != ab_move:
            print('Difference of opinion! Minimax said: {}, alphabeta said: {}. State:
        else:
            #print('Agreed.')
            pass

print('Done. time Minimax: {}, time Alphabeta: {}.'.format(mm_time/REPEATS, ab_time/REP
print('Alphabeta speedup: {} '.format(mm_time/ab_time))
```

```
Done. time Minimax: 1.0585179567337035, time Alphabeta: 1.774511981010437.
Alphabeta speedup: 0.5965121498537111
```

In [ ]:

In [ ]:

In [ ]:

In [ ]: