Intelligent Systems 2020: 4rd practical assignment

Logical Agents

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 work with logical agents on the Schnapsen platform

1. Be able to apply a SAT solver to check for satisfiability and entailment (bots)
2. Apply a SAT solver for build an knowledge-based agent for playing Schnapsen.
3. Build own variants of a knowledge-based agent for Schnapsen.

Preliminaries

In this worksheet we'll use Propositional Logic and a specific version of a Propositional Logic solver (SMT solver) to build a fully knowledge based agent playing Schnapsen. The idea is to represent the strategies humans apply to play the game and to apply logical reasoning to identify good, bad, safe or risky moves, whatever you wish to model. This kind of reasoning---setting up logical statements and proving unsatisfiability.

First things first, let's deal with dependencies for using the kb bot, namely numpy and scipy. They should be installed fairly easily via "pip install numpy", "pip install scipy".

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+Assignment4.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 bots.kbbot.kb import KB, Boolean, Integer, Constant
from api import State, engine, util
```

SAT solving and Entailment checking

We will start with some exercises to use an existing SAT solver to check for satisfiability and entailment.

In [2]:
```
# Define our symbols
A = Boolean('A')
B = Boolean('B')
C = Boolean('C')
D = Boolean('D')
P = Boolean('P')
Q = Boolean('Q')
R = Boolean('R')

# Create a new knowledge base
kb = KB()

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

This file first defines the three boolean symbols we will use (A, B and C), creates an empty knowledge base, and then adds the four following clauses:

> A or B or C
> (not A) or B
> (not B) or C
> B or not C

Task 1

Are there any models (assignments of values to these variables that make all clauses true)? Write down all the models of the knolwedge base in the following cell.

A & B & C -A & B & C

Now run the script and report the output in the cell after the code. Explain what it means.

In [3]:
```
# Print all models of the knowledge base
for model in kb.models():
    print(model)
```

```
# Print out whether the KB is satisfiable (if there are no models, it is not satisfiabl
print(kb.satisfiable())
```

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

The output is all the models that make all the clauses true. Therefore, these two models are the ones that are satisfiable.

Now it is time to adapt your first knowledge-base, by adding stuff....

Tasks 2:

Add a clause to the knowledge base to that it becomes unsatisfiable. Report the line of code you added, and in a separate line, the result that you get from the SAT solver and your script.

KB Clause Added: kb.add_clause(~B, ~C) Output: False

Now you can already to build your own knowledge base, and do reasoning/inferencing with it. For example, check satisfiability of one of the questions of the working group automatically.

Task 3:

Let KB be the knowledge base introduced in Exercise 7 of the Worksheet on Logical Agents: KB=

> (P v Q) ^ (Q -> R) ^ (R -> -P)
>
> -(P <-> - Q)

Write a new version of the script above to prove or disprove whether KB is satisfiable or not. You can put both the knowledge base construction (you will have to translate the axioms into CNF by hand before adding them to the KB), the printing of the model and the satisfiability check in the next cell.

In [4]:
```
P = Boolean('P')
Q = Boolean('Q')
R = Boolean('R')

kb.add_clause(P, Q)

kb.add_clause(~Q, R)

kb.add_clause(~R, ~P)

kb.add_clause(P, ~Q)

kb.add_clause(Q, ~P)

# Print all models of the knowledge base
for model in kb.models():
    print(model)

# Print out whether the KB is satisfiable (if there are no models, it is not satisfiabl
print(kb.satisfiable())
```

```
False
```

Task 4:

Exercise 9 of this week's work session on logical agents contained the following knowledge base KB:

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

Convert it to clause normal form, and write a script that creates this knowledge base as you did before. There are different subtasks: 1) Print out its models and report them.
2) As seen in the exercise, the knowledge base entails A ^ C ^ D. Explain in your own words what this says about the possible models for the knowledge base?

A = Boolean('A') B = Boolean('B') C = Boolean('C') D = Boolean('D') kb.add_clause(A, B) kb.add_clause(~B, A) kb.add_clause(~A, C) kb.add_clause(~A, D) Output: {A: True, D: True, C: True, B: False} {A: True, D: True, C: True, B: True} True Report: There are two models. Therefore, A & C & D is true for KB.

In the previous question, you have shown that A ^ C ^ D is entailed by KB semantically (following the definition of entailment), where you used the prover to give you all the models. Now let us see whether you can actually prove entailment using the SAT solver directly.

Task 5:

Show entailment of A ^ C ^ D by a proof by refutation. Write the script like above with the knowledge base your create in the following executable cell, including the control statements.

In [5]:
```
kb.add_clause(A, B)
kb.add_clause(~B, A)
kb.add_clause(~A, C)
kb.add_clause(~A, D)
kb.add_clause(~A, ~C, ~D)

# Print all models of the knowledge base
for model in kb.models():
    print(model)

# Print out whether the KB is satisfiable (if there are no models, it is not satisfiabl
print(kb.satisfiable())
```

```
False
```

Explain in your own words what you have done, and what you can conclude from the output of your script for Task 5.

By adding -A v -C v -D the KB and not sentence was False, thus it is entailed.

Implementing Schnapsen with an existing SAT solver

Now, we are ready to create a bot that used a knowledge base. Let us first see how we can represent some of the knowledge needed for making strategic decisions in propositional logic.

The idea is that you check with a SAT reasoner, which we provide, whether a knowledge base entails that a certain move adhere to a certain strategy. In the followng example this strategy is trivial, namely a card adheres to the PlayJack strategy if it is a Jack. So, all we have to model is which card is a Jack (cards with index 4,9,14 and 19) and that

> For all x the following holds PlayJack(x) <-> Jack(x)

This formula is, of course, not in Propositional Logic, but needs to be translated, which results in many different clauses (once transferred into Clause Normal Form). Once this is done, we can check that PlayJack(4) is entailed by the knowledge base, while PlayJack(5) is not. We do this by adding the negated propositional variables pj4 or pj5 to the knowledge base and check for satisfiability (proof by refutation) as we did before.

Task 6

Copy the outcome of the entailment test for pj4 and pj5 into the cell after the code/result, and explain in your own words what it means.

In [6]:
```
import sys
from bots.kbbot.kb import KB, Boolean, Integer, Constant

# Define our propositional symbols
# J1 is true if the card with index 1 is a jack, etc
# You need to initialise all variables that you need for you strategies and game knowle
# Add those variables here.. The following list is complete for the Play Jack strategy.
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')

# Create a new knowledge base
kb = KB()

# GENERAL INFORMATION ABOUT THE CARDS
# This adds information which cards are Jacks
kb.add_clause(J4)
kb.add_clause(J9)
kb.add_clause(J14)
kb.add_clause(J19)
# Add here whatever is needed for your strategy.

# DEFINITION OF THE STRATEGY
# Add clauses (This list is sufficient for this strategy)
# PJ is the strategy to play jacks first, so all we need to model is all x PJ(x) <-> J(
# In other words that the PJ strategy should play 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)
# Add here other strategies

# check for entailment (add negation of the to be entailed formula)
kb.add_clause(~PJ5)
# print all models of the knowledge base
for model in kb.models():
    print(model)

# print out whether the KB is satisfiable (if there are no models, it is not satisfiabl
print(kb.satisfiable())
```

```
{j4: True, pj4: True, j9: True, pj9: True, j14: True, pj14: True, j19: True, pj19: True,
pj5: False}
True
```

PJ4 Output: False Entailed.

PJ5 Output: {j4: True, pj4: True, j9: True, pj9: True, j14: True, pj14: True, j19: True, pj19: True, pj5: False} True Not entailed, because J5 is not a Jack. Since PJ is a strategy to play a Jack this is not entailed.

Task 7

Provide the executable cell below a new version of the above script with the knowledge for a strategy PlayAs, always playing an Ace first. Check whether you can do reasoning to check whether a card is entailed by the knowledge base or not. Again, print the result for a check with an Ace as well as with a card differen from an Ace.

In [7]:
```
import sys
from bots.kbbot.kb import KB, Boolean, Integer, Constant

# Define our propositional symbols
# J1 is true if the card with index 1 is a jack, etc
# You need to initialise all variables that you need for you strategies and game knowle
# Add those variables here.. The following list is complete for the Play Jack strategy.
```

```python
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')

# Create a new knowledge base
kb = KB()

# GENERAL INFORMATION ABOUT THE CARDS
# This adds information which cards are Jacks
kb.add_clause(J0)
kb.add_clause(J5)
kb.add_clause(J10)
kb.add_clause(J15)
# Add here whatever is needed for your strategy.

# DEFINITION OF THE STRATEGY
# Add clauses (This list is sufficient for this strategy)
# PJ is the strategy to play jacks first, so all we need to model is all x PJ(x) <-> J(
# In other words that the PJ strategy should play a card when it is a jack
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)
# Add here other strategies

# check for entailment (add negation of the to be entailed formula)
kb.add_clause(~PJ1)
# print all models of the knowledge base
for model in kb.models():
    print(model)

# print out whether the KB is satisfiable (if there are no models, it is not satisfiabl
print(kb.satisfiable())
```

```
{j0: True, pj0: True, j5: True, pj5: True, j10: True, pj10: True, j15: True, pj15: True,
pj1: False}
True
```

PJ0: False Entailed, thus proves its an Ace. PJ1: {j0: True, pj0: True, j5: True, pj5: True, j10: True, pj10: True, j15: True, pj15: True, pj1: False} True Not entailed, thus proves its not an Ace and a Ten instead.

Task 8

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

In [8]:
```
import sys
from bots.kbbot.kb import KB, Boolean, Integer, Constant

# Define our propositional symbols
# J1 is true if the card with index 1 is a jack, etc
# You need to initialise all variables that you need for you strategies and game knowle
# Add those variables here.. The following list is complete for the Play Jack strategy.
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')
```

```python
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')

# Create a new knowledge base
kb = KB()

# GENERAL INFORMATION ABOUT THE CARDS
# This adds information which cards are Jacks
kb.add_clause(J4)
kb.add_clause(J9)
kb.add_clause(J14)
kb.add_clause(J19)
# Add here whatever is needed for your strategy.
kb.add_clause(J3)
kb.add_clause(J7)
kb.add_clause(J13)
kb.add_clause(J17)

kb.add_clause(J2)
kb.add_clause(J8)
kb.add_clause(J12)
kb.add_clause(J18)

# DEFINITION OF THE STRATEGY
# Add clauses (This list is sufficient for this strategy)
# PJ is the strategy to play jacks first, so all we need to model is all x PJ(x) <-> J(
# In other words that the PJ strategy should play 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)
# Add here other strategies
kb.add_clause(~J2, PJ2)
kb.add_clause(~J7, PJ7)
kb.add_clause(~J12, PJ12)
kb.add_clause(~J17, PJ17)
kb.add_clause(~PJ2, J4)
kb.add_clause(~PJ7, J7)
kb.add_clause(~PJ12, J12)
kb.add_clause(~PJ19, J19)

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)

# check for entailment (add negation of the to be entailed formula)
kb.add_clause(~PJ3)
# print all models of the knowledge base
for model in kb.models():
    print(model)

# print out whether the KB is satisfiable (if there are no models, it is not satisfiabl
print(kb.satisfiable())
```

```
False
```

PJ1: {j4: True, pj4: True, j9: True, pj9: True, j14: True, pj14: True, j19: True, pj19: True, j3: True, pj3: True, j7: True, pj7: True, j13: True, pj13: True, j17: True, pj17: True, j2: True, pj2: True, j8: True, pj8: True, j12: True, pj12: True, j18: True, pj18: True, pj1: False} True Not entailed, proving that it is not one of the cheapest and is true since PJ1 is a Ten.

PJ3: False Entailed, proving that it is one of the cheapest and is true since PJ3 is a Queen.

Implementing a Logic-based Agent *kbbot.py*

The final task is to implement and run a rational game agent using the logic-based approach introduced in the previous section of this assignment.

Look at the code implemented in the knowledge based bot (kbbot.py) in the codebase and try to understand the code. All it does is to go through the 5 possible moves and to call a method kb_consistent with arguments state and move.

This method initialises a knowledge base and loads the needed game information from a file load.py. This is where the knowledge is represented about the game and the strategy. If you play the bot kbbot, it will play according to the PlayJack strategy, as this is what is represented in load.py.

Run a tournament between rand, kbbot and rdeep, and report on how this naive knowledge-based bot performs:

kbbot wins against rand and loses against rdeep the majority of the time.

Interpret the results in your own words:

Results: bot : 34 points bot : 16 points bot : 15 points The strategy to play Jacks first is more efficient in scoring in comparison to rand because it minimizes the possibility to win a trick by using a high score card by playing the Jack instead.

Task 9

Replace the knowledge and strategy you modelled in one of the previous questions into a new file load2.py. You might want to create a new kbbot2, which loads your alternative strategy load2, so that you can play kbbot against kbbot2.

Task 9a) Run some games between one of the other bots until you find an example that your strategy works (e.g. that the bot indeed plays always a Jack first if you play the JackFirst strategy.

Due to error I was unable to run the game. However, I have tried to create a strategy where I play either the Jack, Queen or Kings first since they are one of the cheapest options, in comparison to the Ten and Ace.

Please copy-paste below parts of an example printout where your strategy works:

Playing 30 games: Played 1 out of 30 games (3%): [2, 0, 0] Played 2 out of 30 games (7%): [4, 0, 0] Played 3 out of 30 games (10%): [5, 0, 0] Played 4 out of 30 games (13%): [5, 1, 0] Played 5 out of 30 games (17%): [7, 1, 0] Played 6 out of 30 games (20%): [10, 1, 0] Played 7 out of 30 games (23%): [12, 1, 0] Played 8 out of 30 games (27%): [12, 2, 0] Played 9 out of 30 games (30%): [12, 3, 0] Played 10 out of 30 games (33%): [15, 3, 0] Played 11 out of 30 games (37%): [17, 3, 0] Played 12 out of 30 games (40%): [19, 3, 0] Played 13 out of 30 games (43%): [22, 3, 0] Played 14 out of 30 games (47%): [23, 3, 0] Played 15 out of 30 games (50%): [26, 3, 0] Played 16 out of 30 games (53%): [28, 3, 0] Played 17 out of 30 games (57%): [30, 3, 0] Played 18 out of 30 games (60%): [31, 3, 0] Played 19 out of 30 games (63%): [33, 3, 0] Played 20 out of 30 games (67%): [36, 3, 0] Played 21 out of 30 games (70%): [36, 3, 1] Played 22 out of 30 games (73%): [36, 5, 1] Played 23 out of 30 games (77%): [36, 5, 2] Played 24 out of 30 games (80%): [36, 6, 2] Played 25 out of 30 games (83%): [36, 6, 4] Played 26 out of 30 games (87%): [36, 7, 4] Played 27 out of 30 games (90%): [36, 7, 6] Played 28 out of 30 games (93%): [36, 7, 8] Played 29 out of 30 games (97%): [36, 8, 8] Played 30 out of 30 games (100%): [36, 8, 11] Results: bot : 36 points bot : 8 points bot : 11 points

Task 9b) Finally, run the tournament again (copy the tournament code in the cell below):

In [15]:
```python
botnames = []
verbose = False
myphase = 1
myrepeats = 10

# Create player 1
player1 = util.load_player("rdeep")
player2 = util.load_player("kbbot2")
player3 = util.load_player("rand")
player4 = util.load_player("kbbot")

bots = [player1,player2,player3,player4]

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 60 games:
Played 1 out of 60 games (2%): [1, 0, 0, 0]
Played 2 out of 60 games (3%): [3, 0, 0, 0]
Played 3 out of 60 games (5%): [6, 0, 0, 0]
Played 4 out of 60 games (7%): [6, 1, 0, 0]
Played 5 out of 60 games (8%): [6, 3, 0, 0]
Played 6 out of 60 games (10%): [6, 4, 0, 0]
Played 7 out of 60 games (12%): [8, 4, 0, 0]
Played 8 out of 60 games (13%): [10, 4, 0, 0]
Played 9 out of 60 games (15%): [11, 4, 0, 0]
Played 10 out of 60 games (17%): [14, 4, 0, 0]
Played 11 out of 60 games (18%): [14, 4, 1, 0]
Played 12 out of 60 games (20%): [16, 4, 1, 0]
Played 13 out of 60 games (22%): [19, 4, 1, 0]
Played 14 out of 60 games (23%): [20, 4, 1, 0]
Played 15 out of 60 games (25%): [20, 4, 2, 0]
Played 16 out of 60 games (27%): [23, 4, 2, 0]
Played 17 out of 60 games (28%): [25, 4, 2, 0]
Played 18 out of 60 games (30%): [27, 4, 2, 0]
Played 19 out of 60 games (32%): [30, 4, 2, 0]
Played 20 out of 60 games (33%): [32, 4, 2, 0]
Played 21 out of 60 games (35%): [34, 4, 2, 0]
Played 22 out of 60 games (37%): [36, 4, 2, 0]
Played 23 out of 60 games (38%): [37, 4, 2, 0]
Played 24 out of 60 games (40%): [39, 4, 2, 0]
Played 25 out of 60 games (42%): [42, 4, 2, 0]
Played 26 out of 60 games (43%): [42, 4, 2, 1]
Played 27 out of 60 games (45%): [44, 4, 2, 1]
Played 28 out of 60 games (47%): [47, 4, 2, 1]
Played 29 out of 60 games (48%): [49, 4, 2, 1]
Played 30 out of 60 games (50%): [51, 4, 2, 1]
Played 31 out of 60 games (52%): [51, 4, 4, 1]
Played 32 out of 60 games (53%): [51, 4, 5, 1]
Played 33 out of 60 games (55%): [51, 5, 5, 1]
Played 34 out of 60 games (57%): [51, 5, 7, 1]
Played 35 out of 60 games (58%): [51, 5, 10, 1]
Played 36 out of 60 games (60%): [51, 7, 10, 1]
Played 37 out of 60 games (62%): [51, 7, 12, 1]
Played 38 out of 60 games (63%): [51, 7, 13, 1]
Played 39 out of 60 games (65%): [51, 7, 14, 1]
Played 40 out of 60 games (67%): [51, 8, 14, 1]
Played 41 out of 60 games (68%): [51, 9, 14, 1]
Played 42 out of 60 games (70%): [51, 9, 14, 2]
Played 43 out of 60 games (72%): [51, 9, 14, 3]
Played 44 out of 60 games (73%): [51, 9, 14, 4]
Played 45 out of 60 games (75%): [51, 10, 14, 4]
Played 46 out of 60 games (77%): [51, 13, 14, 4]
Played 47 out of 60 games (78%): [51, 14, 14, 4]
Played 48 out of 60 games (80%): [51, 16, 14, 4]
Played 49 out of 60 games (82%): [51, 16, 14, 6]
Played 50 out of 60 games (83%): [51, 16, 14, 8]
Played 51 out of 60 games (85%): [51, 16, 15, 8]
Played 52 out of 60 games (87%): [51, 16, 15, 10]
Played 53 out of 60 games (88%): [51, 16, 15, 12]
Played 54 out of 60 games (90%): [51, 16, 15, 13]
Played 55 out of 60 games (92%): [51, 16, 15, 15]
Played 56 out of 60 games (93%): [51, 16, 17, 15]
Played 57 out of 60 games (95%): [51, 16, 18, 15]
Played 58 out of 60 games (97%): [51, 16, 18, 16]
Played 59 out of 60 games (98%): [51, 16, 18, 17]
Played 60 out of 60 games (100%): [51, 16, 19, 17]
Results:
    bot <bots.rdeep.rdeep.Bot object at 0x00000199D31EB9D0>: 51 points
    bot <bots.kbbot2.kbbot2.Bot object at 0x00000199D31EB4C0>: 16 points
    bot <bots.rand.rand.Bot object at 0x00000199D31EB550>: 19 points
    bot <bots.kbbot.kbbot.Bot object at 0x00000199D31EB970>: 17 points
```

Report on how your new strategy performs with respect to the other bots, including the kbbot with the PlayJack strategy.

Winning order: rdeep > rand > kbbot > kbbot2 The strategy was shown to perform, however in terms of gaining points kbbot and kbbot2 were not efficient in comparison to rand and rdeep. Proving that the strategy to play Jacks first or Jacks, Queens and Kings will not earn many points than randomly playing the cards. Therefore, these two strategies are inefficient.