Play Nine Solitaire

Ilkka Kokkarinen

Chang School of Continuing Education
Toronto Metropolitan University

Python programming project for the course *CCPS 721 Artificial Intelligence*.

Version of June 13, 2022

Rules of the game

This programming project concerns a solitaire version of a commercially sold multiplayer card game of Play Nine. Its simple rules make this game a good first programming project in this author's course *CCPS 721 Artificial Intelligence*. All necessary files needed to complete this project under Python 3 are provided in the author's GitHub repository `ikokkari/PlayNineSolitaire`. Students who are not already familiar with this fun little card game for children of all ages can first consult the original multiplayer version rules available in English, but this is not necessary for following the presentation below.

The gameplay involves a series of decisions made in situations that are a good mix of both the determinism of known cards and the uncertainty of drawing random cards from the deck. Tactical decisions made early in the hand will affect the possibilities and choices that will be made later in the hand. This makes this game an interesting testing ground for various decision making techniques of artificial intelligence.

Instead of the familiar deck of cards that consists of four suits of thirteen ranks each, a special set of cards is used with no suits at all. The ranks of these cards are integers from 0 to 12. The deck also contains some special cards whose rank equals minus five. To keep the mathematics simple by avoiding **card removal effects**, our solitaire version of this game is played with an **infinite deck** that contains all ranks from 0 to 12 in equal proportions, except that there are only half as many cards of rank -5 as there are of other ranks.

The play of one hand is done on a **board** that consists of two rows of cards, the **top row** and the **bottom row**. Both rows have the same number of columns. In the original multiplayer version of Play Nine, the number of columns is always four, but in our solitaire version of the game, the number of columns for the given hand can range from four and ten. Playing with more columns than just four gives the player script more possibilities and strategic leeway in the beginning to optimize the expectation for the future decisions.

At the start of each hand, the cards on the board have been dealt face down, except that one randomly chosen card from each of the rows has been turned face up. As the hand proceeds, one by one these cards on the board get turned over or replaced by other cards, depending on the actions chosen by the player during the hand. Once all the cards have been turned face up, or when the player has performed the given number of draws, the hand is over. All cards on the board are turned over, and the score for the hand is computed as explained soon below. The goal of the game is to minimize the total score over all the hands played during the run.

Using the output of the automated testing framework `play_nine_runner.py` along with the example AI player stub `play_nine_player.py` that always passes the buck to the human

user as needed to decide the action taken at each step, the initial state of the game might be something like the following:

```
Starting hand #1 with 4 columns on board.
There are 9 draws remaining. Kitty card is 12.
Row 0: [  *   *   *   5 ]
Row 1: [  4   *   *   * ]
```

Asterisks are used to indicate a card that is face down, whereas cards that have been turned up are shown with their numerical ranks. As long as there are draws remaining and some cards are still face down on the board, the player must first choose whether to accept the known **kitty card**, or to draw a random card from the infinite deck. (In the output below, the input entered by the human user is highlighted in green.)

```
Will you take the (k)itty card or (d)raw from deck? d
You have chosen to draw from the deck. You are holding a 7.
```

Whichever way the player chooses to draw, the next action is to either use that card to **replace** any one of the cards on the board (the replaced board card can be either face up or face down), or to ignore the drawn card completely and just **turn over** any one of the cards that must be currently face down on the board. Either way, the card in the chosen position is known and therefore ends up being face up.

```
Will you (r)eplace or (t)urn over a board card? t
Enter row number of card to turn over: 1
Enter column number of card to turn over: 2
You are turning over card in row 1 and column 2.
There are 8 draws remaining. Kitty card is 4.
Row 0: [  *   *   *   5 ]
Row 1: [  4   *   2   * ]
```

Note that following the convention of computer science and programming, rows and columns are numbered from zero, instead of from one the way that normal people usually count things.

After these two steps, the play of the hand goes back to drawing the next card from either the deck or the kitty, and making the same decision of how and where to use that card. Once there are no more draws or when all the cards on the board have been turned face up, that hand is over and the only thing that remains is counting the score. As in the game of golf, the smaller the score, the better. (Unlike in golf, the score of some hand may even end up being negative!)

Each card contributes its numerical rank to the total score. However, whenever a column contains the same rank in both its top and bottom rows, those matching cards cancel each other out and are not added to the score. For example, the following board scores only 4 points,

instead of 48 points, since the columns 0, 2 and 3 with matching ranks 4, 12 and 6 are not counted in the score.

```
Row 0: [  4   3   12   6 ]
Row 1: [  4   1   12   6 ]
```

A column that contains a -5 in both its top and bottom rows does not get cancelled out, but is counted as -10. For example, the following board scores 2 points, not 12.

```
Row 0: [  1   -5   10   2 ]
Row 1: [  6   -5   10   3 ]
```

If more than one columns match their top and bottom rows using the exact same rank, each one of those columns gets counted as -5 points each, not zero. These matching columns that use the same ranks are not required to be in consecutive positions. For example, the following board with two matching columns with the same rank of 6 is scored as 5 points, not 15.

```
Row 0: [  6   2   6   1 ]
Row 1: [  6   9   6   3 ]
```

If the matching columns use the same rank of -5, each of those columns gets counted as a whopping -15 points. For example, the following board with two such matching columns scores as -9 points instead of 1 point.

```
Row 0: [  2   -5   -5   4 ]
Row 1: [ 10   -5   -5   5 ]
```

The player script

The instructor provides the complete framework of Python script `play_nine_runner.py` that takes care of all the game mechanics and the scoring of the hand. With the burden of game mechanics out of the way, students only need to provide the functions that perform the actual decision making during the game. The artificial agent that makes these decisions must be written as a Python module that is named exactly `play_nine_player.py`.

This module must define and implement the following three functions that the framework will then be calling at appropriate moments. These functions must work correctly for any number of *k* columns on the board, not just for the special case of *k* = 4 of the original game.

```
def get_author_info():
```

Returns the author information as a two-tuple of strings that contain the name and the student ID of the student. For example, `('Joe Shmoe', '123456789')`.

```
def choose_drawing_action(top_concealed, bottom_concealed, draws_left, kitty_card):
```

Given the current board as two lists `top_concealed` and `bottom_concealed` whose each element is either the integer rank of a face-up card or the character `'*'` to denote an unknown card still face down, the number of remaining `draws_left` and the current `kitty_card`, this function must return precisely one of the two one-letter strings `'d'` or `'k'` for the corresponding actions of **d**rawing a random card from the deck, or accepting the **k**itty card.

```
def choose_replacement_action(top_concealed, bottom_concealed, draws_left, current_card):
```

Given the same three arguments `top_concealed`, `bottom_concealed` and `draws_left` as the previous function, along with a new argument `current_card` for the card drawn after the previous step, this function must return its answer as a three-tuple `(action, row, column)` where action is one of the two strings `'r'` or `'t'` for **r**eplacing the board card or **t**urning it over, and `row` and `column` are integers that determine the position of the card being replaced or turned over. For example, returning the tuple `('r', 1, 2)` would replace the third card of the bottom row with the current card, since the row and column positions start from zero.

Your functions may use anything from the standard library of Python 3.9, plus the recent versions of the **numpy** and **scipy** frameworks.

The GitHub repository contains an example implementation of the `play_nine_player.py` agent script that doesn't make any decisions on its own, but consults the human player to provide all the decisions as they come up. Students can use this example implementation to play this game and that way get a feel of its mechanism and tactical decisions, before they then start replacing these methods with their own implementations that make decisions without ever consulting or needing a human by their side.

The `play_nine_runner.py` framework contains some parameters that students may edit to customize the behaviour of the framework. These parameters are indicated between the comments that delimit this group. Since the instructor will be grading all submissions with the unmodified test runner framework, no student can possibly gain anything by modifying the rest of the runner script. The parameters that can be freely changed during your development stage are the following:

```
TOTAL_HANDS = 10
```

How many hands are played in total to determine the total score. During fuzz testing runs done overnight, this can be set to several thousands.

```
VERBOSE = True
```

If `True`, the steps of each hand are printed out during the run. Otherwise, only the final total score is printed out in the end.

```
SEED = 4242
```

The seed of the random number generator that is used to deal the board and determine the cards drawn from the deck and the kitty.

Other than these parameters clearly marked as such, do not modify the framework script in any way.

Grading

Submit only your `play_nine_player.py` script on D2L, and no other files.

All submitted `play_nine_player.py` scripts will be graded by the instructor using the exact same private `SEED` value, with the setting `TOTAL_HANDS=1000` to measure each submission for its performance over one thousand random hands to make the effect of the luck of the draws negligible for the grading.

The graded submissions are listed in order of their total scores. The submission with the lowest total score is the winner that automatically gets the full 20 marks for the project. Whichever submission down the line along these sorted results is still considered to be acceptable for having passed this project will get 10 marks for the project. The project marks of all other submissions will then be linearly interpolated from these two known fixed points.

The proof of the pudding is in the eating. Grading of your script is determined solely by the score it achieves over the large test run. Style and quality of your code, and the algorithms and techniques used inside it, will not affect your project mark in either direction.

Silence is golden. During the execution, **your functions should not print anything whatsoever on the console**. You can have your own debugging outputs in the code during the development stage, but remember to remove all of them before submitting your code.

It is absolutely essential that your functions return legal results in every possible situation that they can be in. The tester contains a bunch of assertions about your returned results. **If the runner framework crashes for any reason during the test run of your submission, your submission cannot be graded and will receive a zero mark.** Make sure to test your final submission over several thousands of hands during overnight runs over several different `SEED` values to ensure that your functions return some legal result in all possible situations that they can possibly find themselves in.

If your script uses a random number generator of its own for some of its decisions, make sure to initialize your RNG with some fixed seed value that is hardcoded in your script. This makes the runs **repeatable** for the same fixed `SEED` value in the runner, which is essential for any effective testing and debugging.

Your `play_nine_player.py` script must play the game fully above the board, with no cheating allowed. Any attempt whatsoever to interfere with the `play_nine_runner.py` framework used to test and grade your submissions, including but not limited to adjusting its internal score counter or by peeking or changing the cards, i**s automatically considered academic misconduct**, regardless of the specific technique that you used and whether the

attack is made possible by a bug or a security hole in the instructor's runner script. Any such tomfoolery, regardless of whether the exploit was intended as "just a prank, bro", will immediately result in a zero mark for the entire project, and will be reported to the appropriate channels and handled thereafter according to the Toronto Metropolitan University Policy 60 of Academic Integrity.