Connect Four — Artificial Intelligence (AI)

The objective of this document is to present you different ways to achieve a program that will automatically play against you at Connect Four.

First, recall that a skeleton of a Connect Four Python program is on my website (you probably already used it): http://www.barsamian.am/2021-2022/S7ICTB/skeleton_connect_four.py.

The rules of this game can be found on https://en.wikipedia.org/wiki/Connect_Four.

1 Random

A first, simple, way of building your AI is to make it play randomly. This is not as simple as it may seem. Each time the AI wants to play, it will put its token inside a valid random column. A column is valid if it is not yet full.

2 First step

A first "intelligent" idea is the following: you can make the computer always win on its turn when it can win and always avoid to lose on next turn when it can avoid to lose.

You evaluate if, on the current grid, it can win. If so, make it play this move. If the computer can't win, you make it play what you want (with your current algorithm, random play is ok), except that, if it would lead to a winning move for you, the computer has to play something else.

This can be simulated by copying the full grid, making a move, evaluating if the player would win. Listing 1 gives a sample implementation of the "grid copy" function.

```
"""
Copy a grid passed as parameter.
@param grid : the grid that has to be copied.
@return grid : a new grid with the same values than the one passed as input.
"""
def grid_copy(grid):
    copy = build_grid()
    for i in range(7):
        for j in range(7):
            copy[i][j] = grid[i][j]
    return copy
```

Listing 1: Sample code for a "grid copy" function.

You might wonder why we need to write a new function to just copy a grid. After all, when dealing, for example, with numeric variables, Listing 2 creates a new variable `b` which is a copy of `a`, and everything is just fine.

```
a = 5
b = a
b = 2 * b + 5
print(a)
print(b)
```

Listing 2: Sample code that creates a new integer variable.

Let us now look at what happens if we do a similar thing with arrays: copy / paste the code in Listing 3 and run it with Python.

```
a = [5]
b = a
b.append(4)
print(a)
print(b)
```

Listing 3: Sample code that creates a new array variable.

What went wrong this time? Well, when we execute the line `b = a`, Python does not create a copy of the array. It creates a new name, `b`, that references the same object in memory than `a`. In the memory, there is only one array, but we can now access it with two different names. Thus, when we execute an append on `b`, this append is done on this unique array, thus the change is also reflected when we use `a`. This is why we need to make a full copy of the array. We can either do it by hand, either call the `copy` function provided in Python, see Listing 4.

```
a = [5]
b = a.copy()
b.append(4)
print(a)
print(b)
```

Listing 4: Sample code that creates a new 1d array variable.

However, you have to be careful when using this `copy` function when using arrays of arrays, like we do, see what happens in Listing 5.

```
a = [[5], [4]]
b = a.copy()
b.append([3])
b[0].append(4)
print(a)
print(b)
```

Listing 5: Sample code that creates a new 2d array variable.

This time, we are at first happy: `b.append([3])` does the append on the new array that is behind the variable `b`, and not on the array behind the variable `a`. However, `b[0].append(4)` is also reflected on the array behind the variable `a`. What happened? The `copy` function made a new array, yes, but the copy was not recursive, which means that each element in the new array was initialized, not with a copy, but with a simple equality, like before. Thus, in this new array behind the variable `b`, the array behind `b[0]` is the same than the one behind `a[0]`.

To avoid that, we have two ways: we can either do it by hand (as in Listing 1), or we use the `deepcopy` function, see Listing 6. This time, all modifications we will make on `b` or any of its sub-arrays will not be done on `a`. All that for that!

```
import copy
a = [[5], [4]]
b = copy.deepcopy(a)
b.append([3])
b[0].append(4)
print(a)
print(b)
```

Listing 6: Sample code that creates a new 2d array variable.

Remark : you can just remember to be careful with arrays, and to do new copies by hand.

3 Second step

In the first step, our strategy was just to check for a winning move. In fact, almost all strategies in 2-players games involve a way of evaluating the current game state, in more details. An example is described for another popular game, Reversi[1], in the book we began to read, "Invent Your Own Computer Games with Python" [1, Chapter 15] (also in Teams). The function that interests us is on p.233: "Getting a List of the Highest-Scoring Moves" (lines 173–182 of the program).

You can then mimic the behavior of this program in Connect Four: for each possible move (each non-full column), try to make a play in this column, compute the value of the board, and at the end, make the move that gives the best value. The difficult part, of course, is computing the value of the board. A simple example is to maximize the number of adjacent tokens that can lead to 4 in a row (i.e., do not put a third token when it's guaranteed that it's impossible to put the 4th one after). And, when there are different positions possible... always play in the center row — it maximizes the connections you can make, see this short video (5 minutes):

https://www.youtube.com/watch?v=yDWPi1pZOPo

4 Going further

The general idea that we just captured in the previous section can be set further: it is the Minimax algorithm. Instead of looking only at the next move, we look many moves in advance. We want to maximize the value of the board state for us, and to minimize it for the other player. The following video (15 minutes) explains this notion. But, to program this idea, it is best to use the "tree" data structure that we haven't seen yet (even though it is not mandatory), so maybe you won't be able to program it by your own:

https://www.youtube.com/watch?v=y7AKtWGOPAE

If you have a lot of time and really want to go into the details, you can watch the following video (1h30 !) that covers some things needed to write a Connect Four AI:

https://www.youtube.com/watch?v=8392NJjj8s0

You can find in the links after the video everything you need if you want to understand the code, improve the game to use `pygame` (program the game with beautiful graphics instead of playing only with a textual representation of the board).

5 Still further

In fact, this game is solved. It means that we know an optimal way of playing: if the players who begins (often named "White" because the Whites always begin in chess) plays perfectly, he wins.

This result was announced by James D. Allen on October 1st, 1988 and was independently announced by Victor Allis 15 days later [5, 6].

Another, more difficult, game that has been solved is the "Draughts" or "Checkers", in 2007 [3, 8].

More advanced games, like chess or go, are still far from being solved, even though today, top AIs win also against top human players (since "Deeper Blue" against Garry Kasparov in 1997 [2, 7] ; since "Alpha Go" against Lee Sedol in 2016 [4, 9].).

The research articles I cited are available in Teams if you want to read them (the links given in the bibliography usually require that you have an access which you probably don't have unless you know someone who is a researcher). They are hard to read, being research articles, but for each research article I also gave a press article that gives the general idea.

[1] https://en.wikipedia.org/wiki/Reversi

6 References

Books

[1] A. Sweigart. *Invent Your Own Computer Games with Python*. 4th edition. No Starch Press, 2017.
URL: https://inventwithpython.com/invent4thed/.

Press Articles

[2] C. Higgins. *A Brief History of Deep Blue, IBM's Chess Computer*. 2017.
URL: https://www.mentalfloss.com/article/503178/brief-history-deep-blue-ibms-chess-computer.

[3] J. Mullins. *Checkers 'solved' after years of number crunching*. 2007.
URL: https://www.newscientist.com/article/dn12296-checkers-solved-after-years-of-number-crunching/.

[4] BBC News. *Artificial intelligence: Google's AlphaGo beats Go master Lee Se-dol*. 2016.
URL: https://www.bbc.com/news/technology-35785875.

[5] P. Pons. *Solving Connect 4: how to build a perfect AI*. 2019.
URL: http://blog.gamesolver.org/solving-connect-four/01-introduction/.

Research Articles

[6] V. Allis. "A Knowledge-based Approach of Connect-Four. The Game is Solved: White Wins". Master's Thesis. Vrije Universiteit, 1988.
URL: http://www.informatik.uni-trier.de/~fernau/DSL0607/Masterthesis-Viergewinnt.pdf.

[7] M. Campbell, A. J. Hoane, and F.-H. Hsu. "Deep Blue". In: *Artificial Intelligence* 134.1 (2002), pp. 57–83.
DOI: 10.1016/S0004-3702(01)00129-1.

[8] J. Schaeffer, N. Burch, Y. Björnsson, A. Kishimoto, M. Müller, R. Lake, P. Lu, and S. Sutphen. "Checkers Is Solved". In: *Science* 317.5844 (2007), pp. 1518–1522.
DOI: 10.1126/science.1144079.

[9] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529 (2016), pp. 484–489.
DOI: 10.1038/nature16961.