Going further

Bookmark this page

If you want to go further in the programming part of this MOOC, we provide you with additional material at the end of each part.

Note that this extra content does not count towards grading. You can choose to ignore it and proceed directly to the next part.

Why would you want to go further ?

The course material in this section will help you to implement AIs in the PyRat maze game on your local computer. This will allow you to create AIs to compete against your friends or other AIs.

The PyRat maze game

General principles

The PyRat software works essentially as follows. The core of the program is managed by the `pyrat.py` file, which manages all the fundamental elements: labyrinth creation, player management, display management, statistics generation ...

When executed, this program starts a "thread", i.e. a parallel process that takes care of the display. It communicates with him through lines of communication. Therefore, the display has no direct impact on the core of PyRat: if it is slowed down it does not affect the main program (unless of course the machine resources are insufficient).

It also starts two "processes", one for each player. These programs are more autonomous than threads and allow to manage both players almost independently. Communications between the main program and the processes managing the players are also done through communication queues. Again, everything is done to ensure that one player's program does not affect the other's.

Player management

When the game starts, the main program calls the `preprocessing` function of the player's file. If no file is provided for the player, the file `dummyplayer.py` is used instead (this file is available in the "imports" folder). Then, as the game progresses, the main program calls the turn functions of both players to find out their movements.

So it is not directly the file of your artificial intelligence that is called by python, but the main program that uses the functions you defined. That's why running your files directly with python doesn't give anything interesting.

Get the software

The PyRat software can simply be installed on your machine (preferably Linux). All you have to do is follow these steps:

- Download the source code on the GitHub page of the project.
- Unzip the archive.
- Read the README.md file.

Start a game

You can now use the commands to start games with PyRat. These are described in detail in the README.md file. For example, the following command starts a game where the rat tries to pick up all the pieces of cheese by moving randomly:

```
python pyrat.py --rat AIs/random.py
```

On Windows computers :

```
python pyrat.py --rat AIs\random.py
```

A window then shows the rat which picks up the pieces of cheese in the maze by moving randomly.

After the game

Once the game is finished, a string of characters appears in the terminal, summarizing the game :

```
{
        "miss_python": 0.0
        "miss_rat": 114.0
```

```
        "moves_python": 123.0
        "moves_rat": 21.0
        "prep_time_python": 3.0994415283203125e-06
        "prep_time_rat": 0.0017397403717041016
        "score_python": 21.0
        "score_rat": 5.0
        "stucks_python": 17.0
        "stucks_rat": 5.0
        "turn_time_python": 0.0019042918352576775
        "turn_time_rat": 4.34830075218564e-06
        "win_python": 1.0
        "win_rat": 0.0
}
```

It contains a certain amount of information for each player, among which :

- The number of additional movements caused by mud (stucks).
- The number of moves performed, equal to the number of not missed timings.
- The number of movements missed due to a too long calculation or moving against a wall (miss).
- The number of pieces of cheese collected (score).
- A win indicator that is 1 if the game was won by that player, 0.5 if tied and 0 otherwise.

To obtain **average statistics** on several games, (which is interesting, especially if they contain some random factor), use the "`-tests`" parameter and the last line will indicate the average obtained for each of the criteria mentioned above. You should use it in combination with "`--nodrawing --synchronous`" for faster computations.

Details on PyRat AIs

To make it easier to learn the PyRat game, we provide the teachers and the students with a Python source file skeleton. This is located in the `AIs` folder, and is called `template.py`. Reading the code contained in this file, you will realize that it is subdivided into subsections:

- **Pre-defined constants**: These are useful constants. The movements defined there (MOVE_XXX) are to be returned by the turn function to tell PyRat how to move. In addition, the name of your AI is to be defined in the TEAM_NAME constant.
- **Free expression area**: You are not restricted to the two functions useful to PyRat (preprocessing and turn). You can write all the code you want here.
- **Pre-processing function**: At the beginning of any part of PyRat, the preprocessing function is called. This allows you to do some calculations beforehand, in order to prepare your decisions during the game turns.
- **Game turn function**: Once the pre-processing phase is completed, the game turns are sequenced, and the turn function is called regularly. Each time this function returns a movement decision, the turn is applied.

The `preprocessing` and `turn` functions have a **limited time** allotted. If your pre-processing is too long, you will miss the first rounds of play. Similarly, if your turn is too long, you may move every other round!

You can find a complete tutorial on the MOOC to guide you through writing a simple program. Feel free to follow it to better understand how to use the software.

In addition, the code of the `programs/template.py` file is well documented, and details among other things the types of parameters of the `preprocessing` and `turn` functions, as well as their usage.

Also, take the time to discover PyRat's options (launch "`python pyrat.py -help`") to better understand them.

Understanding the data structures used in PyRat

Using the data structures

Launch python in a terminal (python3 if you are using Ubuntu).

Then copy and paste the following code:

```
mazeMap = {(0, 0): {(1, 0): 1}, (0, 1): {(0, 2): 1, (1, 1): 1}, (0, 2): {(0, 1): 1, (1, 2): 3, (0, 3): 1}, (0, 3): {(0, 2):
1}, (1, 0): {(0, 0): 1, (2, 0): 1}, (1, 1): {(0, 1): 1, (2, 1): 1}, (1, 2): {(0, 2): 3, (1, 3): 1}, (1, 3): {(2, 3): 1, (1,
2): 1}, (2, 0): {(1, 0): 1, (3, 0): 1}, (2, 1): {(3, 1): 1, (1, 1): 1}, (2, 2): {(2, 3): 8}, (2, 3): {(1, 3): 1, (2, 2): 8,
(3, 3): 1}, (3, 0): {(2, 0): 1, (4, 0): 1, (3, 1): 1}, (3, 1): {(2, 1): 1, (4, 1): 1, (3, 2): 1, (3, 0): 1}, (3, 2): {(3,
1): 1, (4, 2): 1}, (3, 3): {(2, 3): 1}, (4, 0): {(3, 0): 1}, (4, 1): {(3, 1): 1}, (4, 2): {(3, 2): 1, (4, 3): 1}, (4, 3):
{(4, 2): 1}}
mazeWidth = 5
mazeHeight = 4
playerLocation = (0, 0)
opponentLocation = (4, 3)
piecesOfCheese = [(1, 0), (4, 2), (0, 3)]
```

```
timeAllowed = 2000
```

You now have an environment that looks like the one that will be available inside the `preprocessing` or `turn` function when they are called.

Let's start with the simple things. The `playerLocation` and `opponentLocation` variables are pairs of integers. you can access each of the coordinates as follows :

```
>>> playerx, playery = playerLocation
>>> print("the coordinates of the player are (" + str(playerx) + ", " + str(playery) + ")")
the coordinates of the player are (0, 0)
```

The cheese pieces are stored in a list of pairs. Here are some manipulations of this structure:

```
>>> piecesOfCheese[1]
(4, 2)
>>> piecesOfCheese[-1]
(0, 3)
>>> lastCheesex, lastCheesey = piecesOfCheese[-1]
>>> lastCheesex
0
```

The maze map is a more complex structure, it is a dictionary associating couples with dictionaries, which associate couples with integers. Here are some examples of manipulations:

```
>>> mazeMap[(2,1)] # gives us the neighbours of cell (2,1) with the corresponding weights
{(3, 1): 1, (1, 1): 1}
>>> for (a,b) in mazeMap[(2,1)]:
...     print(str(mazeMap[(2,1)][(a,b)]))
...
1
1
>>> mazeMap[(2,1)][(2,0)]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: (2, 0)

>>> for (x,y) in mazeMap:
...     for (xx,yy) in mazeMap[(x,y)]:
...             if mazeMap[(x,y)][(xx,yy)] > 1:
...                     print("There is mud between (" + str(x) + "," + str(y) + ") and (" + str(xx) + "," + str(yy) + ")")
...
There is mud between (0,2) and (1,2)
There is mud between (1,2) and (0,2)
There is mud between (2,2) and (2,3)
There is mud between (2,3) and (2,2)
```

Feel free to manipulate these structures to understand how they work.

Statistics and plots

Computer science is a difficult science: concepts are often linked to discrete mathematics and proving intuitive results can be complex.

Rather than proving a property with a mathematical formalism, it often happens that we try to support a property by numerous and well thought simulations.

Let us show you on an example how such a scientific approach is implemented to analyze a problem.

Problem

The problem we are considering here is that of the famous Syracuse sequence.

A Syracuse sequence is obtained by recurrence as follows:

- The first term is a parameter u_0, which is a strictly positive integer
- The u_{i+1} term is calculated as follows:
  - if u_i is even, then $u_{i+1} = u_i/2$
  - if u_i is odd, then $u_{i+1} = 3 \cdot u_i + 1$

As soon as a 1 is encountered in this sequence, the cycle 1,4,2,1,4,2,... repeats itself indefinitely.

It is supposed that for any choice of u_0, the sequence will end up passing through 1 and thus repeating this cycle. However, no one has proved this.

We will try to analyze the impact of u_0 on the number of values the sequence takes before meeting the value 1.

Python Program

In order to analyze this impact, we start by writing the function which calculates the Syracuse sequence :

```
# The syracuse function takes as an argument an integer n strictly higher than 0
# It outputs the number of steps before encountering 1
def syracuse (n) :
  if n == 1 :
    return 0
  else :
    if n % 2 == 0 :
      return 1 + syracuse(n / 2)
    else :
      return 1 + syracuse(3 * n + 1)
```

Let us save this function in a file called `syracuse.py`.

Small values of u_0

First, we will look at what happens with small values of u_0. To do this, we are going to make our program interfaceable with a script, i.e. ease its execution with different parameters.

The Python sys module is used for this:

```
import sys

# We retrieve the first argument as u0
u0 = int(sys.argv[1])

# The syracuse function takes as an argument an integer n strictly higher than 0
# It outputs the number of steps before encountering 1
def syracuse (n) :
  if n == 1 :
    return 0
  else :
    if n % 2 == 0 :
      return 1 + syracuse(n / 2)
    else :
      return 1 + syracuse(3 * n + 1)

# We execute the function and output its result
print(syracuse(u0))
```

Thanks to this module, we can now start our program from the terminal by setting u_0 as argument. To do this, we write in a terminal:

```
python3 syracuse.py 14
```

The output printed in the terminal is 17.

Average results

We will execute our script for values of u_0 between 1 and 100 to see how it behaves. We therefore write in a terminal :

```
for i in {1..100}; do python3 syracuse.py $i; done
```

Note that this line would only work in a unix environment (e.g. Linux). If you are using Windows instead, it could be replaced by (thanks to Colin_McNicholl):

```
for ($i = 1; $i -le 20; $i++) {py -3 syracuse.py $i}
```

The output is not very readable ...

Instead we would prefer to have a plot. To do this, we will create a file in csv format with the result as a function of u_0 . Modify the display of the result in the file `syracuse.py` :

```
print(u0, syracuse(u0))
```

The command line in the terminal becomes as follows, where `>> results.csv` allows to redirect the output of the program to the file `results.csv` :

```
rm -f results.csv
for i in {1..100}; do python3 syracuse.py $i >> results.csv; done
```

The first line ensures that the old `results.csv` file, which might exist, is removed before new data is added to the file.

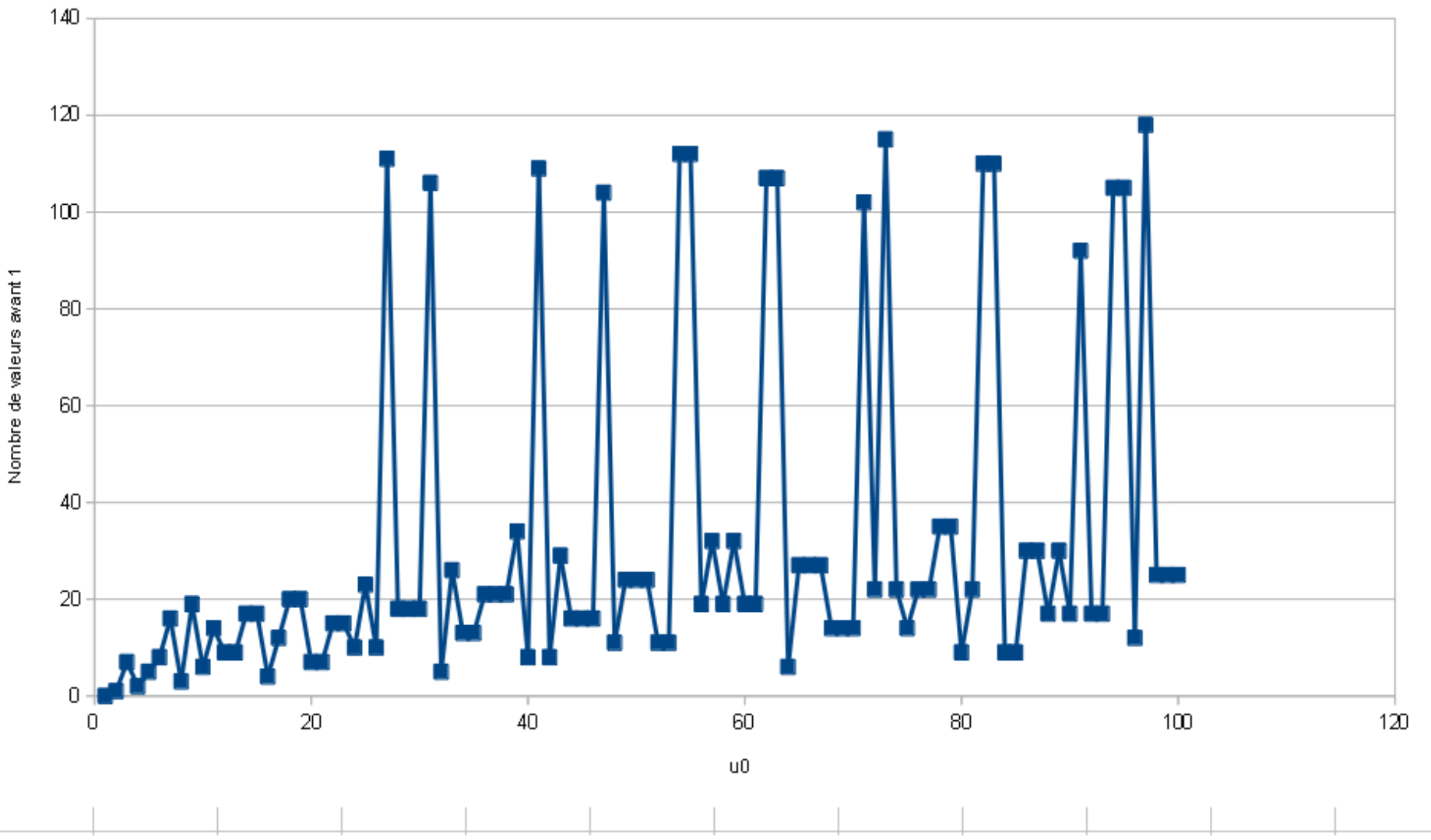
Again, for Windows users, the command is a bit different (courtesy of Colin_McNicholl):

```
for ($i=1; $i -le 100; $i++) {py -3 syracuse.py $i | Out-File results.csv -Append}
```

To plot these data, we use LibreOffice :

```
libreoffice results.csv
```

We specify that the data are separated by spaces and we can then draw the following plot :



Behaviour analysis

What we have obtained is already interesting. It seems that globally our curve is increasing even if there are strong local variations.

In order to verify this statement, we propose to check the situation for larger values of u_0 . We cannot be exhaustive so we will look at what happens for the first multiples of 1,000,000. The problem is that given the variations, we cannot just draw a plot with a point for every multiple of 1,000,000.

What we propose then is to look at a large number of numbers close to multiples of 1,000,000 and average them. For that, we will use the `random` module of Python :

```
import random

# We perform 10.000 tests
TESTS = 10000

# We study a neighborhood of ± 50.000 around u0
NEIGHBORHOOD = 50000

# We start the computations and store the results in total
total = 0
for _ in range(TESTS) :
    total = total + syracuse(u0 + random.randint(-NEIGHBORHOOD, NEIGHBORHOOD))

# We write the output
print(u0, total / TESTS)
```

If we run the tests multiple times with

```
python3 syracuse.py 1000000
```

we notice that the values do not change much, which means that 10,000 tests for each value is reasonable.

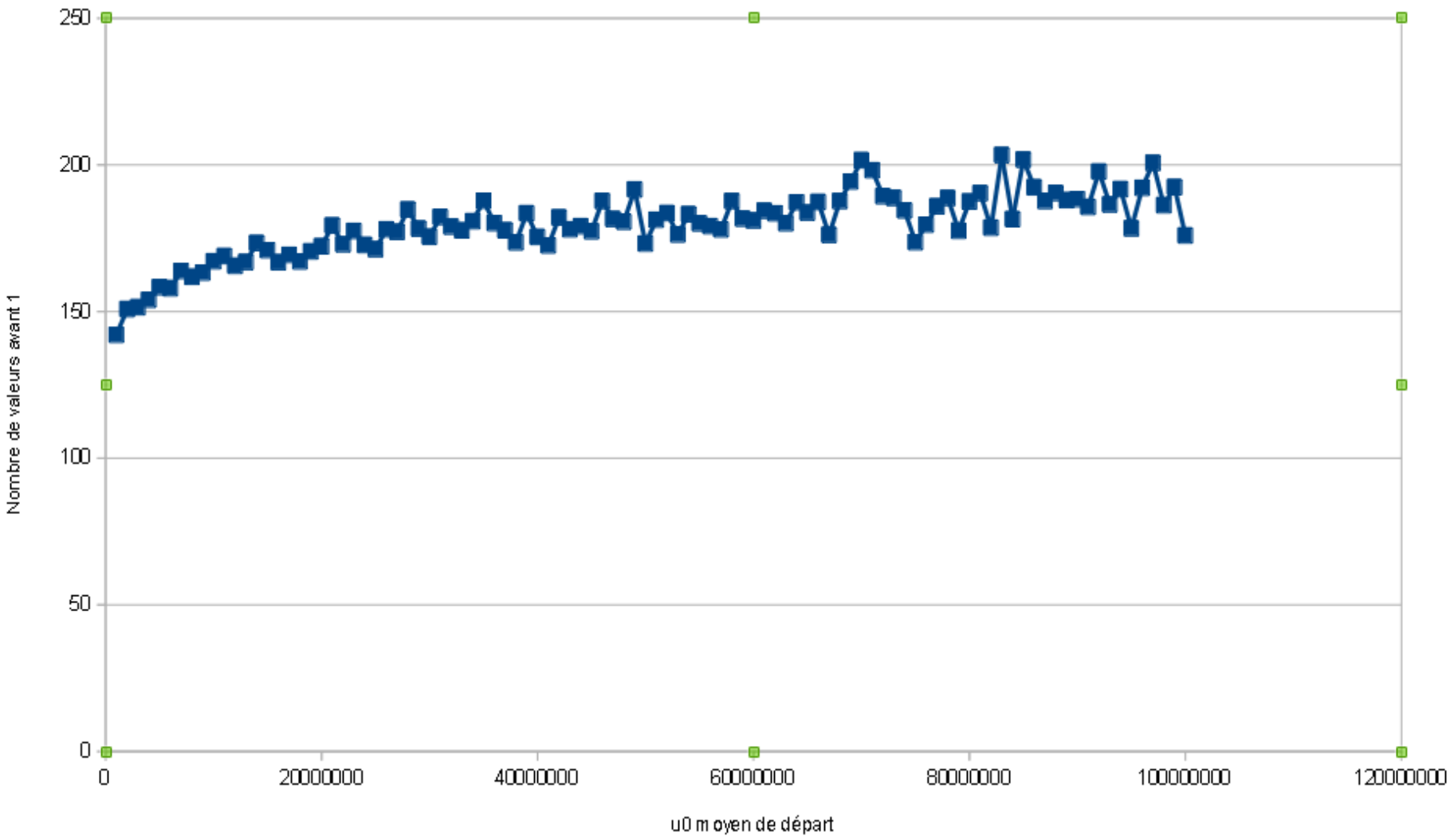
We still have to calculate the result for our different values, and write the result in a file `results.csv` :

```
rm -f results.csv
for i in {1..100}; do python3 syracuse.py ${i}000000 >> results.csv; done
```

For Windows users (courtesy of Colin_McNicholl) (note that the new file is names results2.csv this time):

```
for ($i=1; $i -le 100; $i++) {py -3 syracuse.py ${i}000000 | Out-File results2.csv -Append}
```

This script takes a little time, which is normal given the amount of calculations required. We obtain the following curve:



Let's optimize the program

Instead of recalculating everything each time, the results already obtained can be stored in memory to speed up calculations. This is at the expense of a much higher memory consumption (and potentially too high for your computers: if your computer runs out of memory, it will use the SWAP or worse, it will freeze. Do not hesitate to interrupt the program by pressing `CTRL+C` in case of problems):

```
import sys

# We retrieve the first argument as u0
u0 = int(sys.argv[1])

# The idea is to fill a dictionary to count the number of values
# before a 1 is encountered, and to refer to them instead of computing
# everything again for other values
values = {1:0}

# Now, finding the requested value can be done either by consulting the dictionary,
# or by computing it if not present
def syracuse (n) :
    if n in values.keys() :
        return values[n]
    else :
        if n % 2 == 0 :
            res = 1 + syracuse(int(n / 2))
        else :
            res = 1 + syracuse(3 * n + 1)
    values[n] = res
    return res

# Function to compute everything for a given u0 (Thanks to Ada_Tzereme for pointing out mistakes in this part)
def computations (u0) :

    # We compute the mean for a certain neighborhood
    # Remark: here, we compute exhaustively in the neighborhood rather than using a random function
    NEIGHBORHOOD = 50000

    # We start the computations and store the results in total
    total = syracuse(u0)
```

```
    for i in range(1, NEIGHBORHOOD):
        total = total + syracuse(u0 + i) + syracuse(u0 - i)

    return total / (2 * NEIGHBORHOOD - 1)

# We write the output
print(u0, computations(u0))
```

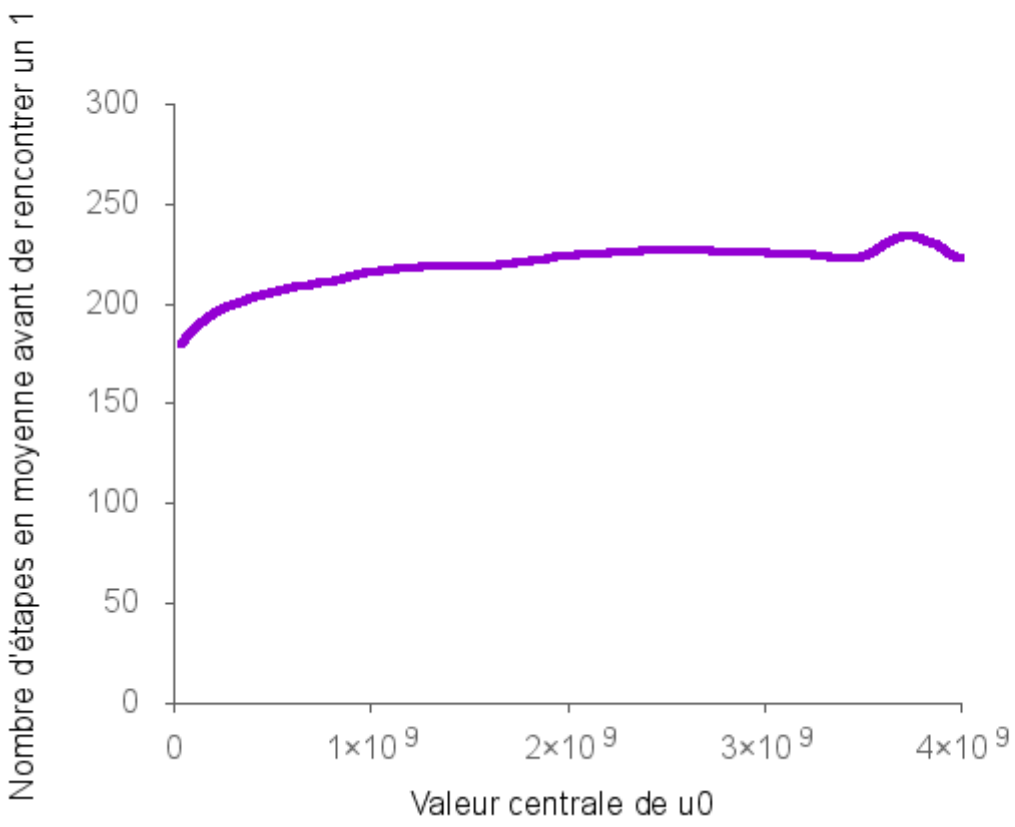
The program is called with large values of u_0 :

```
rm -f results.csv
for i in {1..100}; do echo $i; python3 syracuseopt.py $[ $i * 4 * 10000000 ] >> results.csv; done
```

For Windows users (courtesy of Colin_McNicholl):

```
for ($i=1; $i -le 100; $i++) {py -3 syracuse_opt.py ($i * 4 * 10000000) | Out-File results3.csv -Append}
```

We obtain the following plot :



This plot is obtained with the Gnuplot tool and the following script :

```
# Header
set encoding utf8

# Labels
set xlabel "Central value of u0"
set ylabel "Number of steps before obtaining a 1"
set style line 11 lc rgb '#606060' lt 1
set border 3 back ls 11
set tics nomirror out scale 0.75

# Ranges
set yrange [0:]

# Tics
set xtics 1000000000

# Zoom
set size 0.8,0.8

# Caption position
set key at 0.7,0.95

# Output options
set terminal png
set output "syracuse.png"

# Lines definitions
set linestyle 1 lt 1 lw 4

# Plot
plot "results.csv" using 1:2 notitle with linespoints ls 1 smooth bezier
```

PyRat Lab 1

Let us now implement a first rat in the PyRat maze game.

Let us now implement a first rat in the PyRat maze game.

The goals of this lab are to :

1. **Program** : develop an algorithm that uniformly randomly selects directions while avoiding walls.
2. **Test** : check you program by running a large number of tests and making sure you can find the piece of cheese each time (fix the number of pieces of cheese to 1).

3. **Analyze** : draw plots showing the influence of various maze parameters on the number of movements before catching the piece of cheese.

Presentation of the algorithm

One idea to find a piece of cheese in the labyrinth is to move randomly. However, this may take time, as you will probably visit the same places several times. So we will move at random, but (a little) smarter. The algorithm we will use proceeds as follows, as long as the piece of cheese is not reached:

- We move on a square not visited yet.
- If we can't do it, we move at random.

The second step could be improved, for example by returning to the last known position with neighbours not yet visited.

Implementation

To implement this algorithm, we start from a copy of the `AIs/template.py` file, renamed to `AIs/improved_random.py`.

Let's change the name of the program, and delete the content of the `preprocessing` and `turn` methods (which just serve as demo).

Once that's done, let's think about what we need.

First, we will need to choose random directions. So we need to import the `random` Python library, and create a function `randomMove()` that returns a random direction. Note that

```
random.randint(0, 3)
```

draws uniformly an integer between 0 and 3 and returns it.

Then, at each step, we want to move to a position that has not yet been visited. It will therefore be necessary to remember the visited locations. For this, we choose to use a data structure `visitedCells`. Since we will use this structure for several distinct function calls, it must be declared as a global variable, outside any function body.

Let's move on to the two main functions. There is nothing particular to do at initialization. We will therefore use the Python `pass` keyword to indicate that the body of the `preprocessing` method is empty.

We will need a function, which for an origin position and a destination position, returns the movement to be performed. Let us call it `moveFromLocation. Note that` Python's `numpy` library provides a function to easily subtract pairs of integers, item by item:

```
difference = tuple(numpy.subtract(targetLocation, sourceLocation))
```

Now, let's go to the main part of the program: the definition of the behavior at each game turn. A first thing to do is to find the movements that can take us to a position not visited. To do this, we define a new function that returns the list of these movements. Let's call `listDiscoveryMoves.`

In the turn function, you can now use the `listDiscoveryMoves` function to select a move to a new position. However, if such a movement does not exist, one chooses to move in a random direction. **Attention!** Don't forget to declare `visitedCells` as a **global** variable to update it.

To check the behavior of your program, use the following command in a terminal, at the root of the PyRat directory :

```
python3 pyrat.py --rat AIs/improved_random.py
```

You should now see a Window representing the maze, and in which the rat moves around.

edX

Legal

Connect