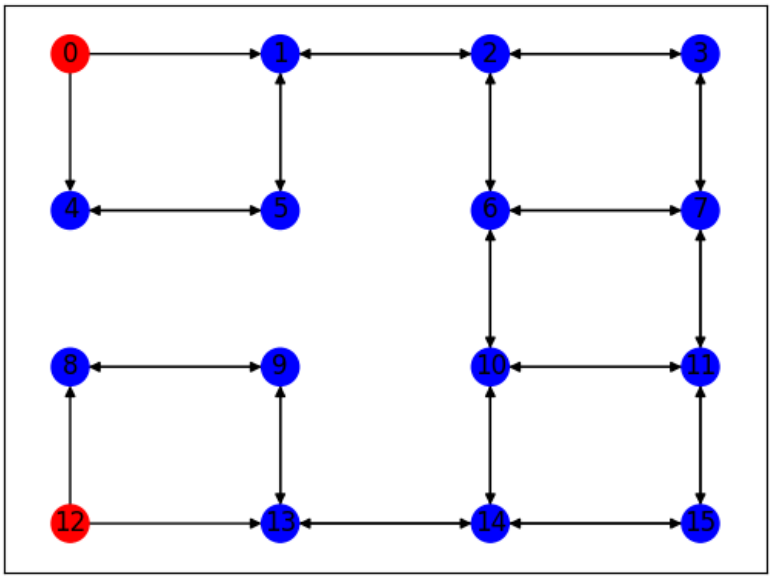
SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS
DEPUIS 1257

Sorbonne University

Python Project

Some algorithms for solving graph path finding problems



Kenneth Assogba - Alexis Squarcioni

Master 2 IMPE-Meca
2019-2020

Table des matières

1 Data Structure **3**
1.1 Possible displacements 3

2 Algorithms **4**
2.1 Pure random 4
2.2 Pure greedy 4
2.3 Epsilon-greedy method 5
2.4 Epsilon-Gamma-greedy method 5
2.5 Small improvements 5

3 Results and performance **6**
3.1 Results 6
3.2 Improvements considered 6

Introduction

The objective of the project is to code in Python3 an algorithm which will give guidance instructions to one or few robots based on a room blueprint. The robot(s) will have to go over each box at least once, but with a minimal number of moves.

To achieve this goal, we must first read and process the data from the plan of the room. Then, it will be necessary to store this data in adequate data structures. Finally, it will be a question of writing an algorithm which will give the movement instructions to the robots in order to clean the part in an optimal way. In addition, we will have to check and validate our code using test cases.

1 Data Structure

We have several information to store and update as we move, including : the position of the robots, the cleaned boxes and possible movements. We will take the exemple of Case_Aspi_R_2 to illustrate.

•Robot's position is represent as dict having for key : robotColor and for value : robotPosition.

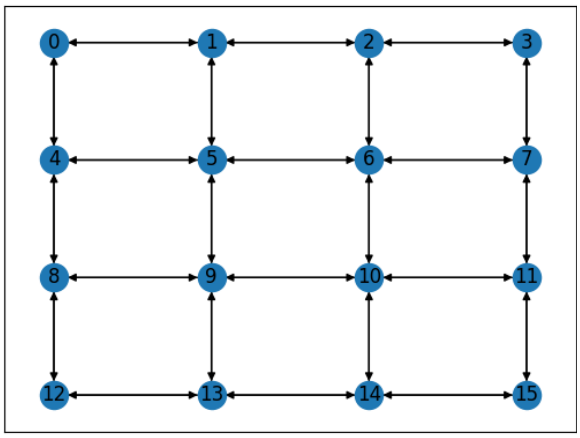
```
robots = {'B': 0, 'R': 12}
```

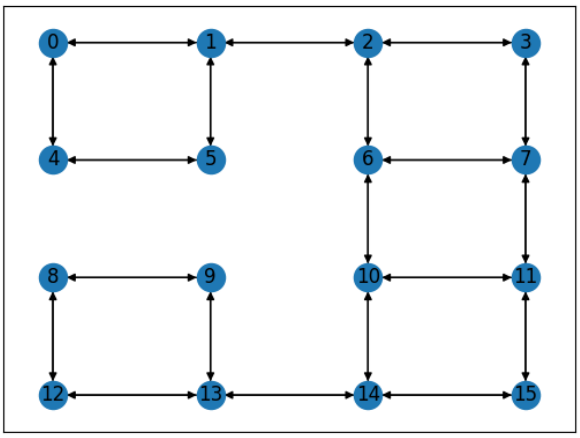
•Cleaned cases is represent by a list. Cases are numbered from 0 with row-major order when value 1 is for cleaned case and else 0 :

```
cleanedBoxes = [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0]
```

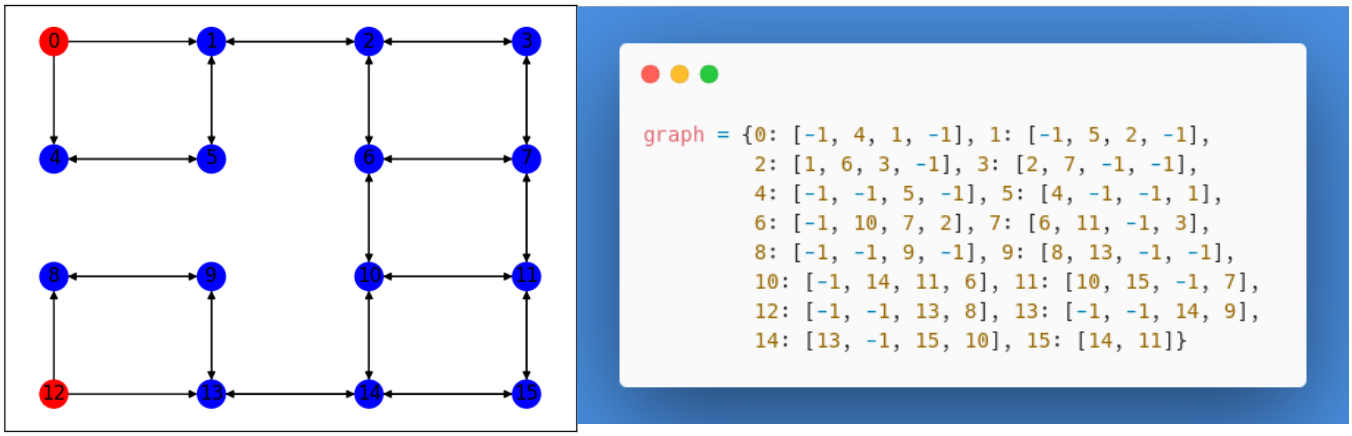
1.1 Possible displacements

We have chosen to represent our grid by an directed graph. This directed graph represents the possible displacements starting from a box while taking into account the internal walls, external walls and robots. We store it in a dict because : dicts and sets use hash tables so have $O(1)$ lookup performance [1]. Here we have the following graphs : 1) : without walls and without robot ; 2) : with walls and without robot ; 3) : with walls and robots.





1. 5 tips to speed up your Python code. https://pybit.es/faster-python.html



We finally create a class *Cleaning* containing our data and methods.

2 Algorithms

Robots are always randomly chosen from among those who can move. Since all of our algorithms are random, we rerun the *Cleaning.cleaning ()* method several times and only return the optimal path. This number of iterations depends on the position of the internal walls, the number of robots and their positions. After studying the test cases that have been submitted to us, we set the number of iterations for each of them via the *setParameters* function in order to avoid unnecessary calculations on simple cases. For unknown test cases, a number of security iterations = 500000 is set, the calculation time is still reasonable (<2min). In the following, we describe the strategies developed and how we play them.

2.1 Pure random

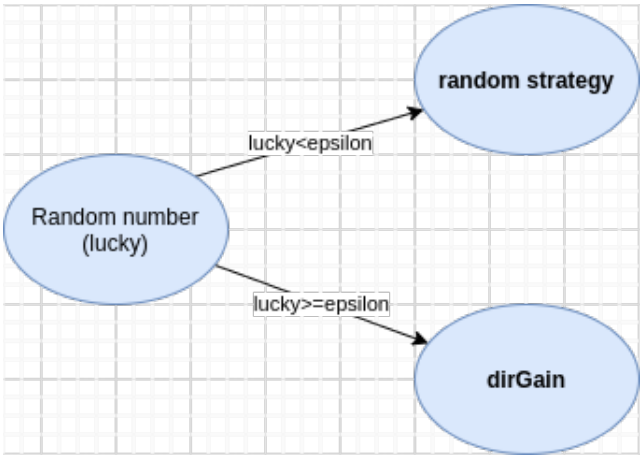
Here, after choosing the robot, a random direction among the possible directions is chosen. One can theoretically obtain an optimal answer for all the problems in infinite time and the performances in computation time are mediocre. Python uses the Mersenne Twister as the core generator. It produces 53-bit precision floats and has a period of $2^{19937} - 1$. An improvement would be to ban round trips.

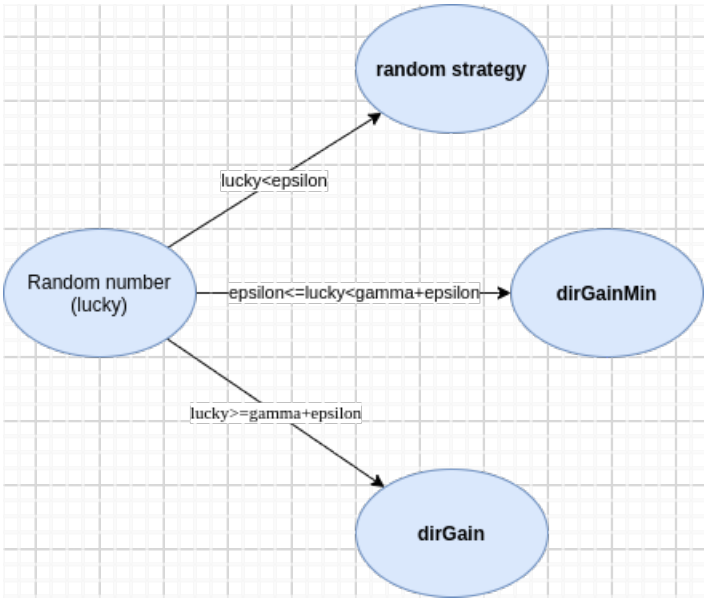
2.2 Pure greedy

Because the random strategy is very expensive, we thought of a heuristic to clean up as many boxes as possible. *dirGain* determines for a given robot the direction of maximum gain, that is the direction in which the robot cleans the most boxes. In case of equality, a random choice is made. This greedy strategy means making the optimal choice at each iteration. The computing times have been divided by 10, however this strategy is far from being optimal overall (or only in special cases) especially since upstream, the choice of robot is made randomly.

2.3 Epsilon-greedy method

We introduce a bit of randomness via a *epsilon* parameter. At each step, a random number $\in [0, 1)$ is generated by *random.random()* method. If the number was lower than epsilon in that step the robot chooses a random direction and if it was higher than epsilon in that step the robot chooses an action based on *dirGain*. An improvement would consist in varying epsilon according to the iteration in order to more or less influence the greedy.





2.4 Epsilon-Gamma-greedy method

Since we did not obtain satisfactory results on certain test cases, we decided to implement an additional heuristic. Indeed, the optimal solution of test case 4 necessarily involves making small movements at the beginning. This new heuristic : *dirGainMin* determines the direction of non-zero minimum gain. In the event of a tie, a random choice is made. To randomly play our 3 strategies, we introduce a new parameter γ. From now on, the random strategy will be played according to the probability ϵ, *dirGainMin* according to the probability γ. The gamma and epsilon parameters being variable according to the test case, we adjust them with the *setParameters* method.

2.5 Small improvements

Each time a path smaller in size than the previous ones is obtained, its size is saved in *lenoptimal*. As soon as a new path reaches the current *lenoptimal* it is stopped. This saves a lot of unnecessary calculations.
We also wrote a Destructor to delete copy of robot. The effectiveness of this seems minimal since Python already destroys unused objects. Finally we notice that the computation times are divided by 20 (even more) by commenting on the many print (verbose boolean), we noticed that print consumes a lot thanks to the command *python -m cProfile*.

3 Results and performance

3.1 Results

We present the results we have obtained. The algorithm being random, we can obtain a new optimal path for each execution. The number of iterations is chosen in order to greatly increase the probability of finding an optimal path without sacrificing the computation time. Tests are performed on a machine Debian 10 with Intel Core i5-7200U CPU @ 2.50GHz × 4 and 7.7 GiB of memory. IW=Internal Wall. The star test cases * are those where we obtain better results than in the unit tests provided by the professor. Finally, the execution time of all unit tests by *unittest* is **124.7s**.

	nx*ny*nd	**IW**	**Depl**	**iter**	ϵ	γ	**Time(s)**	**Path**
0	4*3*2	0	6	200	0.1	0	0.02	RE BE RS RW RN RE
1	4*4*2	0	6	200	0.1	0	0.02	RE RN BE RW BS BW
2	4*4*2	4	10	10000	0.1	0	1.71	BS BE RN BN RE BE BS RS BW BN
3	6*6*3	8	16	200000	0.15	0.2	67.81	YE YS BE BS YW RE RN BW RW RS YN BN BE BS BW RE
4	6*6*2	6	**12***	60000	0.15	0.8	14.77	**RE BS RN RE RS BE RE RN RW BN RE RS**
5	6*6*2	6	12	20000	0.1	0.1	4.97	RS BE BS RE BN BW BS BE BN RN RW RS
6	6*7*2	6	**14***	200000	0.2	0.1	64.42	**RS BE BS BW RW BN BW BS BE BN BE BS RE RN**
7	6*7*2	6	14	40000	0.15	0	11.87	BE RS RW RN RE RS BS RN RE RS BE BN BW BS

3.2 Improvements considered

An important issue is the judicious choice of the robot to move. We have not yet found an adequate strategy for this. Then, it would have been interesting to improve *setParameters* in order to obtain the number of iterations, the epsilon and gamma parameters based on the data in the grid rather than on the name of the test case.
Finally we thought of implementing an ant colony algorithm, that is to say mark in each box the number of times it has been covered and preferably go in the directions where we went the least. But we didn't know which strategy to apply at the start of the game.

Conclusion

From a results perspective, we have achieved the desired findings. That is, we have written an algorithm that determines the optimal path for a given part. These results were validated by eight test cases.
Furthermore, this project allowed us to improve our knowledge of Python. We were able to implement different algorithms that we tested each time. This allowed us to verify and validate our code in the same way as is done in a study office. Our code has also been documented in English and French. Finally, we prepared a detailed report on the results we obtained and the methods used.