Maze Adventure: An Application of Maze Algorithm in Role-playing Game Development by Python

Rui yang[1], Conger He[2]

[1]ryang69@wisc.edu,
[2]congerhedesign@gmail.com

[1]University of Wisconsin Madison College of Letters and Science, Wisconsin 53715, United States,
[2]Fashin Institution of Design & Merchandising, Los Angeles, California 90015, United States,

Abstract This paper talks about the development and algorithms of a 2D role-playing game coded in Python: Maze Adventure. The game is developed based on the Aldous Broder maze generation algorithm, a random walk algorithm, which is a way to produce uniform spanning trees. With the maze that is drawn by these uniform spanning trees, this paper introduces each part of code that has different functions, which works together to form a complete game. It summarizes the basic algorithms for not only the maze algorithm, but how to apply and link the function with the game's structure. It includes the generation and modification of different types of sprites, a Misty algorithm that hides most of the mazes for players to explore, and various ways of connecting collision detection along with the maze, in order to let players fight monsters or pick up treasures. It's an exploration of how to combine a traditional maze algorithm with new media like games, and by finishing Maze Adventure, it remarkably proved that Python language has strong fluidity, which is capable of using one of the newest media: Games.

Keywords: Python, Aldous Broder algorithm, Spirit, Collision

1 Introduction

In any coding education website that can be found, Python is the simplest to understand, and most suitable for novices to learn. After the development of Pygame, creating games in the Python language has soon become a fashion. The maze generation algorithm was invented and kept developed by different engineers for a long time. With the ability of Pygame to display graphics, it is possible to create a maze-exploring game completely under python language. This paper begins with a comparison of basic maze algorithms, then discusses the method about how to combine generated maps with character collisions. Subsequently, it focuses on user interfaces, digging into the detail of gaming elements, such as the calculation of experience and level, misty fog, and character animation.

The first part of the article discusses the basic structure of the game, including the way to build up Maze Adventure step by step from the maze generation algorithm to the sprite, and the

ICIDC 2022, June 17-19, Qingdao, People's Republic of China
Copyright © 2022 EAI
DOI 10.4108/eai.17-6-2022.2322876

collision detection method used in the coding.

The second part illustrates the point which can be improved in the future. Due to the time limit of the development circle and the inadequacy of the number of groupmates, many of the decisions are the simplest solution but not the best to solve the problem, such as collision detection function and the randomization for the monster and the treasure box.

2 Algorithms Used

2.1 Why Aldous Broder algorithm?

Firstly, the Aldous Broder algorithm is a random walk algorithm with no bias. Since a new maze is needed to randomly generate for the game each time when the player clicks on the "Play" bottom, it is necessary to find an appropriate random walk algorithm. Among all the different kinds of algorithms, the team's first decision was to use the Binary Tree algorithm as the maze-generated algorithm. However, The Binary Tree algorithm can also generate a random maze, but the problem is that the first row and the last column of the grid will always be connected as shown in Figure 1. Considering the playability of a game, the Binary Tree algorithm clearly cannot be the best choice for the game. Due to the fact that the game will be not interesting if the player can easily reach that end by simply going through the top row and the left column. Clearly, the algorithm needs to be changed to a non-bias type, such as the Aldous Broder algorithm. Compared to the Binary Tree, it can develop a well-randomized maze with no shortcut for the player.

2.1.1Basic implement of the Maze

There are three basic classes for maze implementation. The first one is called the cell, the most basic element in the maze representing every square in the maze. The four most important fields in the cell class are called North, South, East, and West. Each field contains another cell element, which is like a link list, and that cell element shows the permutation of all of the cells. There are a bunch of important methods in this class, including a method called Link. This method can add another cell to a list in the field of this cell, which contains all of the cells connected to this cell. Another important method for the cell class is called Neighbor which can return a list of all geographical neighboring cells, regardless of any connections. By using this method, the algorithm can easily access the neighbor of that cell and get prepared to build up the future maze.

The second class is a combination of all of the cells, which is called the grid. The grid is basically a two-dimensional array. Every element in this array is a cell element. A method called connect cells play an important role to build up the unlinked maze. It traverses every cell element in the two-dimensional array and sets every cell's neighbor according to the field North, South, East, and West of that cell.

The third class is called the Markup class which is used for drawing all of the cells onto the screen. This class can easily get access to every cell in a specific row and column, and represent the final mazes as a graph in the window.

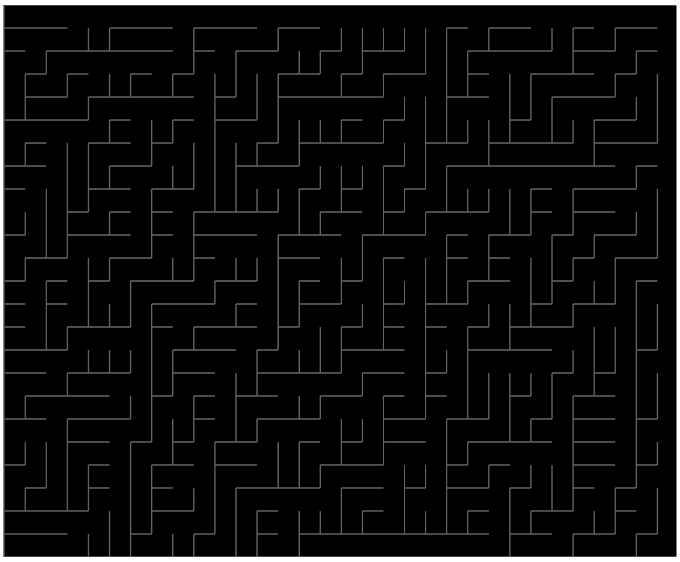
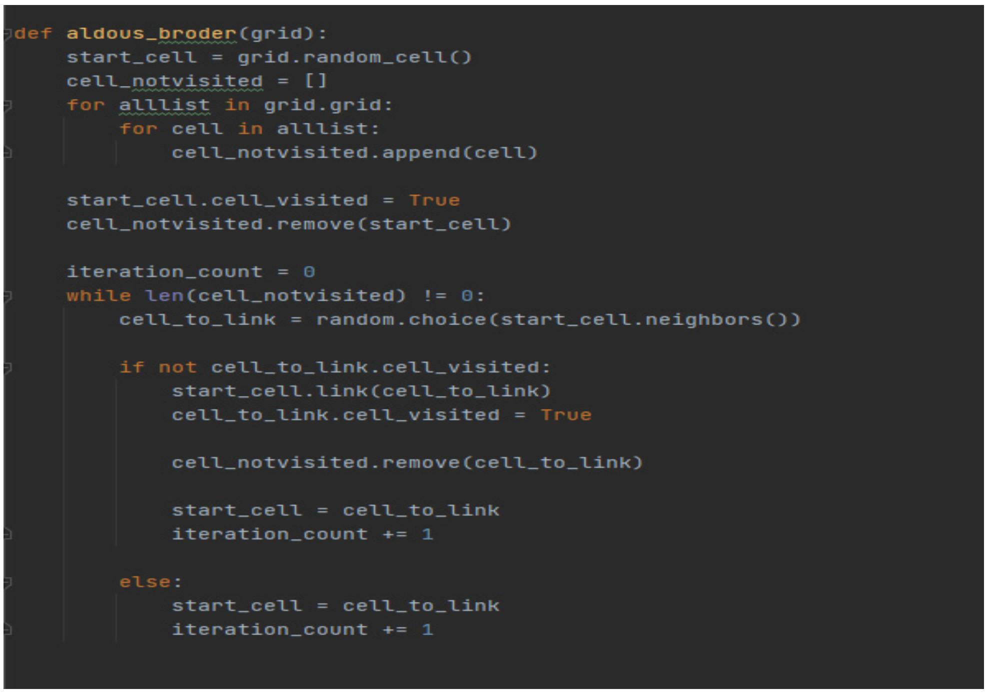


Figure 1 A Maze Example of the Binary Tree algorithm

2.2 The implementation of Aldous Broder maze algorithm

Figure 2 shows the functional codes of the Aldous Broder. The code follows the rule of the basic Aldous Broder algorithm. Firstly, it picks a random cell and marks it as visited, then adds all of the cells in the grid into a list. The list is functioned to hold all the none-visited cells, and the first random cell will be marked as the Currents Cell, then removed from the list. After that, it will choose a random direction, if the cell at that direction is not visited yet, the current cell and that cell will be linked, and then set the second cell as the Current Cell. The algorithm will repeat previous moves until there is no cell in the none-visited cell list, which means the maze is perfectly generated [1].



```
def aldous_broder(grid):
    start_cell = grid.random_cell()
    cell_notvisited = []
    for alllist in grid.grid:
        for cell in alllist:
            cell_notvisited.append(cell)

    start_cell.cell_visited = True
    cell_notvisited.remove(start_cell)

    iteration_count = 0
    while len(cell_notvisited) != 0:
        cell_to_link = random.choice(start_cell.neighbors())

        if not cell_to_link.cell_visited:
            start_cell.link(cell_to_link)
            cell_to_link.cell_visited = True

            cell_notvisited.remove(cell_to_link)

            start_cell = cell_to_link
            iteration_count += 1

        else:
            start_cell = cell_to_link
            iteration_count += 1
```

Figure 2 the Code Sample for Aldous Broder

2.3 Sprite

The sprite module in Pygame is what the game system uses to store information about characters and monsters [2]. Also, the level, attack, health, mana value, and the picture used on the actual game window are all built in this class.

Sprite is an in-build class in the Pygame. Several classes are included in this module for use within games. In addition to the main Sprite class, there are several Group classes that contain Sprites. Using these classes is entirely optional when coding with Pygame, classes are lightweight and only provide a starting point for the code that is common to most games [3].

Sprites serve as a base class for all types of objects in the game. There is also a Sprite Group class that simply stores sprites. Games could create new Group classes that operate on specially customized Sprite instances they contain. [3]

Finally, this module contains several collision functions. They are useful for finding sprites inside multiple groups that have intersecting bounding rectangles. Sprites and Groups manage their relationships with the *add* () and *remove* () methods. Surfacer attributes are required to find collisions. They can accept single or multiple targets. The default initializers for these classes also accept a single target or a list of targets for initial membership. Repeatedly adding and removing the same Sprite from a Group is fine. [3]

2.4 Collision detection

For the collision detection part, since the sprite system is implemented, which is a Pygame module in the python library so that it would be most easy to use the in-build collision detection function in the sprite module. The decision is to make all of the monsters and treasures into a sprite group and use pygame.sprite. spritecollide [4] to make a simple test of if a sprite intersects anything in any group. There do have some other better collision detection functions that build in with the Pygame system, but considering both the size of the graphs and the actual running time, keeping the basic detection function is the best for this game.

2.5 Misty algorithm

The Misty algorithm is an algorithm that was developed by the authors. It hides most parts of the maze when the character that the player controls is created at the upper left corner as the game starts. The misty fog will stay until the character moves nearby, which creates a sense of mystery. In this case, the player needs to walk around to explore their way to the end, and cannot get to see all parts of the maze unless they have moved to every corner of the maze.

To achieve this goal, the team developed a special algorithm. In order to implement this feature, a special attribute is added in the cell class called "visible" which is initially false and takes the coordinates of the player character as parameters. Every time the player moves, it updates the visible attribute of the nearby cells, changing it from false to true. In the end, when the whole maze is drawing onto the screen, the system would only draw the part of cells that has the true value on the visible attributes.

2.6 Monster generation

In order to achieve the randomness of the whole maze, it is also needed to randomly generate monsters in a specific position. However, if the generation of the monster is truly random, another problem surfaces. Some monsters may crowd in a small area, but other regions will be totally blank. Clearly, it will break the balance of the game. To solve this problem, the team developed a pseudo-random algorithm, which divides the whole maze into separate parts. In this case, the maze is totally 19 by 20, after removing the last row as a line for the destination, the maze will be divided into a total of 30 of 3 by 4 little squares. The monsters and treasure boxes will be generated at a random position in each little square. Also, it's random for the algorithm to decide if it's a monster or a treasure box. By doing a pseudo-random, it ensures that each generated maze is completely random, and prevents the situation that too many monsters were generated in the same area, causing the game to be too difficult or simple.

2.7 Basic model

The basic model for this game is that the character the player controls have an initial level which is always level 1, and the health, mage, attack values will all depend on the character's level. In the beginning, when the level is 1, the player has 50 health, 50 mana, and the attack value is 10. As the player meets and defeats the monster, they will get experience (EXP), and the character's level will increase when getting enough amount of exp. Every level-up would lead to an increase in both health and mana. Every time the player defeats a monster, there is a so-called level up method that would run, and it would check whether or not the exp of the character meets the standard to level up. Also, there are some treasure boxes randomly distributed in the maze.

Every time the character meets the treasure box, a random event triggers. The player will get one of the prizes, including an apple, which can increase 10 health, an elixir, which can increase 10 mana, a sword, which can increase 5 attacks, or an exp bottle, which can add 20 exp to the character.

2.8 The character

The movement of the character is implemented by changing the coordinate of the character picture depending on the pixel size of each cell in the grid. Every time the player clicks on one of the keys in “w”, “a”, “s”, “d”, which represents moving “up”, “left”, “down” and “right”, the position of the picture will change towards that direction, redraw at the position that is one cell-size more than the original position. The same system is also combined with animation, every time a key is clicked, the next frame will be redrawn in the new location [5].

3 Improvement

There are many parts of the game algorithm that can be improved.

3.1 collision detection

The collision detection method of the game can be improved in the future since the current version is using the most basic detection module. This method only takes the sprite image as the input and uses the picture rectangle as a detection area. In this case, due to the different shapes of the monster and treasure box, and the character, there need to be different ways of collision detection.

Mentioned in the article by Mirtich, it proved a method of collision which is used in robot motion planning. Mirtich also says that their collision detection algorithms have roots in computational geometry, where the basic problem is to report intersections among a group of static objects [6]. This method can help us solve the problem when the object is moving and detect the collision. In this case, in the future, we can also set the monsters of our game moving in a specific way.

Mirtich illustrates the Two-phase collision detection in specific. Mirtich maintains that people can divide collision detection into two phases. The first one is called the broad phase, and the second one is called the narrow phase. For the broad phase, the method just culls away most of the pairs using a trivial rejection test based on bounding boxes, bounding spheres, octrees, or the like. And for the narrow phase, the method uses algorithms that are more refined to check for collision or to compute distances. In this case, the broad phase serves as a filter for the narrow phase, and the choice of algorithms can be independent [6]. Two-phase collision detection would be a great method to use in this game since it saves a lot of time computing the information from the narrow phase using the broad phase. If choosing only to use one specific algorithm to compute the collision, the method would either be too complicated, or the result of the algorithm would be too rough which is not enough for the output.

3.2 randomization for the monster and the treasure generation

The monster generation for the game is not truly random as we mentioned before. In this case, a way to improve the randomness of the game is to achieve true randomness. However, when using truly random algorithms to generate monsters and treasure boxes, it is necessary to prevent any possible generation problems, which is the duplicate of position when the monsters and treasure boxes are generated as previously mentioned.

The position is not the only fact that can be changed when randomly generating the monsters. The monster's health, attack, and level can also be randomly generated. By achieving that, the indeterminacy of the game needs to be improved and let the player meet much more different situations compared to the first version.

Park introduces a great reformed Genetic algorithm for monster generation, which is applied to Homologous Chromosomes. As Figure 3 shows, Park's genetic monster generation algorithm first randomly generates the first generation of the monster. Then he will evaluate the score of the monster-fighting with the character. Finally, select some of the monster's values and crossover the value of the monster, and create a mutation that generates the next generation [7].

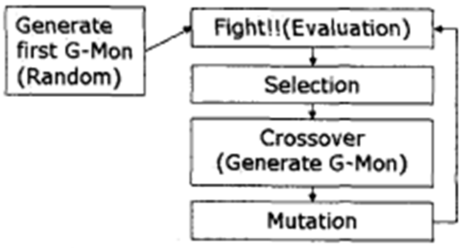
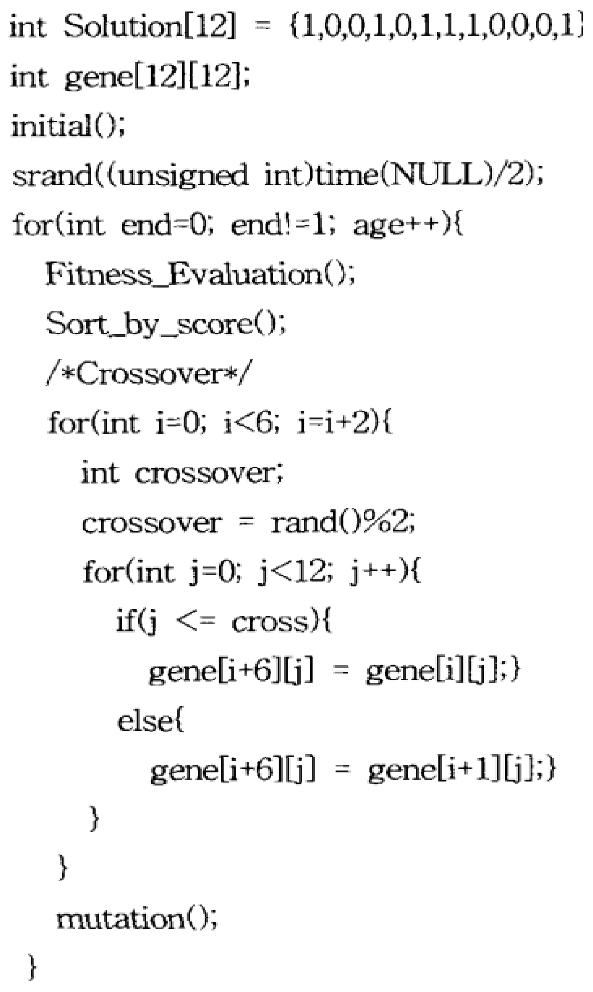


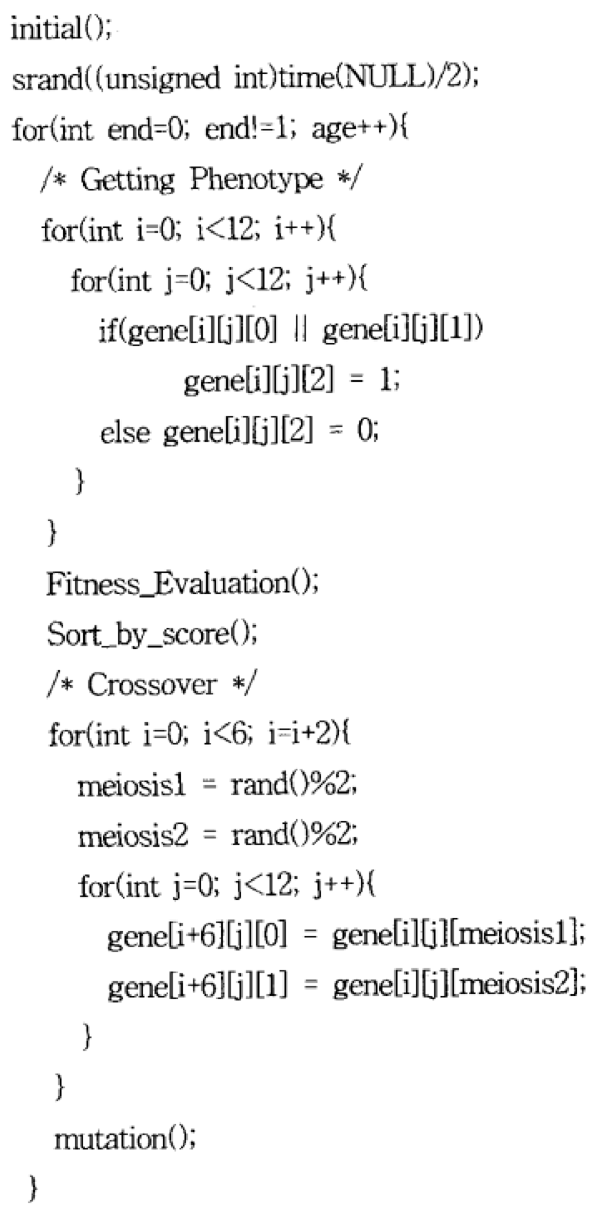
Figure 3 Structure of Park's Genetic Monster Generation Algorithm [7]

This genetic mutation is used to make sure the randomness among individual monsters and control the value of all monsters to a specific range. A sample code of achieving this generation is shown in Figures 4 and 5 [7].



```
int Solution[12] = {1,0,0,1,0,1,1,1,0,0,0,1}
int gene[12][12];
initial();
srand((unsigned int)time(NULL)/2);
for(int end=0; end!=1; age++){
  Fitness_Evaluation();
  Sort_by_score();
  /*Crossover*/
  for(int i=0; i<6; i=i+2){
    int crossover;
    crossover = rand()%2;
    for(int j=0; j<12; j++){
      if(j <= cross){
        gene[i+6][j] = gene[i][j];}
      else{
        gene[i+6][j] = gene[i+1][j];}
    }
  }
  mutation();
}
```

Figure 4 Code Sample of Park's Genetic Monster Generation Algorithm [7]



```
initial();
srand((unsigned int)time(NULL)/2);
for(int end=0; end!=1; age++){
  /* Getting Phenotype */
  for(int i=0; i<12; i++){
    for(int j=0; j<12; j++){
      if(gene[i][j][0] || gene[i][j][1])
            gene[i][j][2] = 1;
      else gene[i][j][2] = 0;
    }
  }
  Fitness_Evaluation();
  Sort_by_score();
  /* Crossover */
  for(int i=0; i<6; i=i+2){
    meiosis1 = rand()%2;
    meiosis2 = rand()%2;
    for(int j=0; j<12; j++){
      gene[i+6][j][0] = gene[i][j][meiosis1];
      gene[i+6][j][1] = gene[i][j][meiosis2];
    }
  }
  mutation();
}
```

Figure 5 Code Sample of Park's Genetic Monster Generation Algorithm, part 2 [7]

4 Conclusion

Even though Pygame provided a convenient way of graphic display, the main gaming system is still built-in python itself. To build a role-playing adventure game, it is important to combine graphics with the cell system, in order to give a reference of where the open road is. The advantage of the system is that all related elements can be easily controlled and changed within each moving key, but instead, all variables need to be remarked again in every used key, also. By developing Maze Adventure, it proves that as one of the easiest coding languages, Python and Pygame is not lacking in accomplishing game-related functions, but furthermore considered most of the possible situations, and built them in as a part of the language. With this new type of media which needs to spend a lot of time and effort to produce, it provides a quicker way to create a playable prototype, greatly broadening the possibilities and tentativeness of game design. Also, compare to C$^{\#}$ and C^{++}, Python is more convenient for newcomers to learn and get familiar with, for the group of people who's interested in game design but barely touched on programming, it's a great choice to actually achieve a small game model using Python and Pygame. Due to the simplicity and broad functions, it is possible to become the next trend of the game coding world – for both the big development teams and personal creators, to share their ideas with the public.

References

[1] Jamis Buck, 2011. Maze Generation: Aldous-Broder algorithm. https://weblog.jamisbuck.org/2011/1/17/maze-generation-aldous-broder-algorithm

[2] DEV1 documentation. (2021). Pygame.sprite - pygame v2.0.1. https://www.pygame.org/docs/ref/sprite.html#pygame.sprite.Sprite.groups.

[3] Hu, Y., Lyons, R., & Tang, P. (2021) A reverse Aldous–Broder algorithm. Annales de l'Institut Henri Poincaré - Probabilités et Statistiques, 57: pp. 890-900.

[4] DEVI documentation. (2021). Pygame. sprite. spritecollide - pygame v2.0.1 https://www.pygame.org/docs/ref/sprite.html#pygame.sprite.spritecollide

[5] Tech With Tim, 2018. Character Animation. https://www.techwithtim.net/tutorials/game-development-with-python/pygame-tutorial/pygame-animation/

[6] Mirtich, B. (1997). Efficient algorithms for two-phase collision detection. In: Kamal G. Angel P. D. (Eds.), Practical motion planning in robotics: current approaches and future directions. Wiley, Hoboken. 203-223.

[7] Park, S. W., & Lee, W. H. (2006). Genetic Algorithm for Game Monster Generation. Proceedings of the Korea Contents Association Conference 2006, 4: pp. 811-814.