A Method for Generating Mazes with Length Constraint using Genetic Programming

Kiotaka Okano
Iwate University
Iwate, Japan
Email: tongari1123@outlook.jp

Katsutsugu Matsuyama
Iwate University
Iwate, Japan
Email: m18u@iwate-u.ac.jp

***Abstract*—We examine a method to automatically generate mazes with length constraint. Assuming that the user creates a text including branches, our prototype arranges the input text on a maze space. In this study, we employ genetic programming and define commands to structure our maze generation program. We implemented our program, and as a result, we were able to generate the desired maze for simple input text.**

***Keywords*-maze generation; genetic programming; text arrangement;**

I. Introduction

A mazes is a game or puzzle that aims to reach a goal through complicated paths such as bending and branching. Algorithms for generating mazes have been studied for a long time, and there are many maze generation algorithms [1]. There are some research focusing on adding secondary functions to mazes. For example, Xu and Kaplan proposed a method that generates mazes resemble to the input image [2].

OHANASHI MEIRO (Story Maze series) created by Akira Sugiyama [3] is one of mazes with secondary functions. In Story Maze series, one letter is assigned to one cell and the player follows the letters and proceeds through the maze while reading the text. Fig. 1 shows an example of Story Maze generated by our method. When the player reaches the goal, one (main) story, such as a fairy tale, is completed. At a location where the passage branches, a story branch also occurs. That is, when the player reaches a dead end other than the goal, a story different from the main story is completed.

This study examines a method to support the creation of story mazes. Assuming that the user creates a ”text tree” including ”story branches”, we examine a prototype of a system that arranges the input text tree on a maze. If we focus on arranging the input text tree on a maze without considering its story, this corresponds that the maze creator specifies the passage lengths of the maze. In this paper, we consider a method for automatically generating mazes with length constraint.

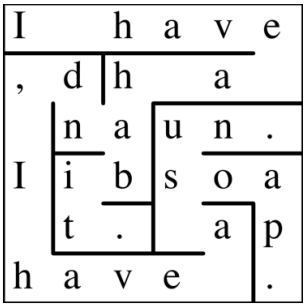

Figure 1: an example of Story Maze generated by our method

II. Related works

Algorithms for generating mazes using computers have been studied for a long time, and there are many maze generation algorithms [1]. There are some researches focusing on adding secondary functions to mazes, including methods to generate a maze similar to the input image [2, 4], a maze in which the shape of the solution is similar to the input image [5], and a maze having the above two features [6].

There are several maze generation methods that consider difficulty [7], and some of them takes into account the length of the maze [8]. [8] introduces the length of passages and the number of dead ends to the fitness function of the Genetic Algorithm. However, the purpose of [8] is to generate a maze according to the degree of difficulty, and it is not possible to generate a maze with the target lengths. We could not find methods that generate a maze considering length constraint, which is the purpose of our study.

III. Basics of maze generation

This section describes the basic maze generation methods. Data structures and terms used for maze generation are also described. For more details, see [1, 2].

For simplicity, we assume that the walls of a maze are in a planar grid (Fig. 2 (left)). The planar grid is called a ”grid”, and the rectangular surfaces constituting the grid is

978-1-7281-8771-6/20/$31.00 ©2020 IEEE
DOI 10.1109/NicoInt50878.2020.00014

called a ”cells”. A combination of multiple cells that do not share a wall is called a ”passage”. Many maze generation methods create passages by repeatedly erasing walls.

Introducing a cell graph, some of maze generation studies reduce their problem into graph theory. Fig. 2 (right) shows an example of a cell graph. Each node is corresponding to one cell. If there is no wall between cells, the corresponding cell graph nodes are connected.

A maze containing no cycles is called ”perfect”. An example maze in Fig. 2 is also perfect.

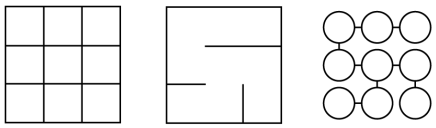

Figure 2: data structure of mazes

IV. Our method

In this study, we consider a method for automatically generating mazes with length constraint such as Story Maze series. Our system proposed in this paper consists of: (1) representation of a text tree which includes story branches, and (2) generating mazes. Details of them are described in the following sections. In this study, we assume the maze is in a planar grid.

A. Representation and structuring of story branches

In this study, we introduce a tag $< split >< /split >$ to indicate branch parts in text. The text below is an example of tag usage. The parts enclosed by the split tags are considered as branch parts and our system interprets it as tree structure shown in Fig. 3. In the tree structure, one node has one letter.

The path from the first letter of the input text (”a” in Fig. 3) to the last letter (”e” in Fig. 3) in tree structure is called a ”main path”, and represents a ”main story”. Paths other than the main path are called ”branch paths”.

An example of input text

ab$< split >$fg$< /split >$
c$< split >$h$< /split >$de

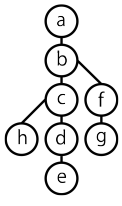

Figure 3: text in tree structure

B. Generating mazes

We generate a maze by arranging a text tree on a maze. It is necessary to form a maze while assigning one letter to one cell.

In this study, we employ genetic programming [10]. Genetic programming is an optimization algorithm that applies evolutionary pressure. The difference from the genetic algorithm is that a tree structure is treated as a gene. Therefore, the genetic programming can handle programs with complicated contents. We will explain details in section 5.

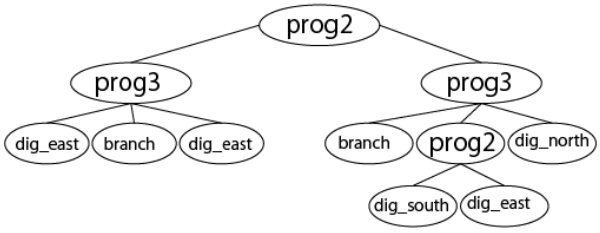

Figure 4: tree structure representing program

V. Maze generation using genetic programming

This section describes our maze-generating program and evaluation functions.

A. Maze-generating program

In order to generate a maze by genetic programming, it is necessary to define a maze generation program in tree structure. This section describes the commands defined in our maze generation program.

Commands to erase walls (create passages):
We define commands that create a passage from the ”current cell” to an adjacent cell. In the case of a grid maze, there are four adjacent cells from the current cell. Therefore, we define commands to erase the wall in each direction (create a passage) and then move to the next cell. Specifically, four commands dig_north, dig_south, dig_west, dig_east: erase the wall of the north, south, west, and east direction respectively and move to the cell in that direction. These four commands are collectively called ”dig commands”.

Fig. 5 shows an example of the dig commands. The black cell on the cell graph represents the ”current cell”. By the command dig_east, the program creates a link to the east cell and moves to it. Then, by the command dig_south, it creates a new link to the south cell and moves to it.

If the wall to be erased has already been erased (Fig. 6 upper), if cycles occur (Fig. 6 lower), if the wall to be erased

is outer wall of the maze, or if the length of the passage exceeds the input length (Fig. 7), the commands are not executed. In case of Fig. 7, the command dig_east is not executed, since the length of the cell graph will exceed 5, the length of the main path.

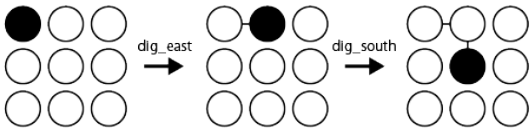

Figure 5: an example of the dig commands

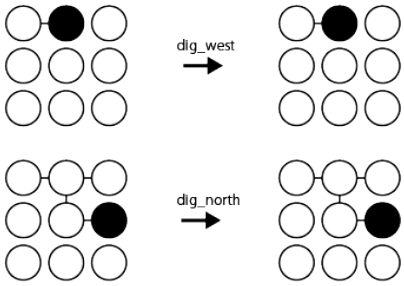

Figure 6: case not executed: a wall already erased (upper) and cycles (lower)

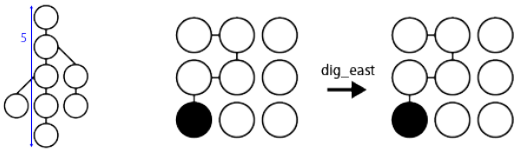

Figure 7: case not executed: the length of the main path

Commands to create a maze fork:

In order to create a maze fork, we create a queue to store branch source cells. By the dig commands, the nodes of the input text tree are arranged in the cell one by one. Here, when a node where a branch occurs is arranged in a cell, the cell is enqueued.

Using the queue and the information of the current cell, we create a maze fork. We define a $branch$ command as a command to change (jump) the current cell. The $branch$ command operates differently depending on the number of times it is executed. Specifically, in the odd-numbered execution, the cell is dequeued, and the dequeued cell is set as the current cell. In the even-numbered execution, the cell at the end of the main path in the cell graph is set to the current cell (Fig. 8).

If the number of executions exceeds twice the number of branches in the input text tree, if there are no cells in the queue, the command branch is not executed.

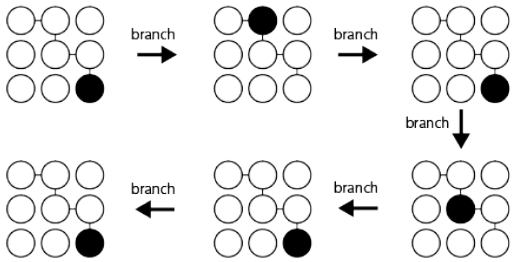

Figure 8: an example of the command branch.

Commands to structure commands:

In this study, since we employ genetic programming, it is necessary to represent programs in tree structure. We define commands to contain multiple child nodes to structure commands. Specifically, commands $prog2$ and $prog3$ are defined, each having 2 or 3 child nodes. Commands $prog2$ and $prog3$ execute child nodes in order from left to right.

Fig. 4 shows an example of a program. When the program in Fig. 4 is executed, the cell graph shown in Fig. 9 is generated.

Genetic programming repeatedly applies crossover and mutation to tree-structured programs (genes) to optimize.

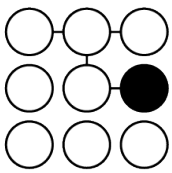

Figure 9: cell graph generated by Fig. 4 program

B. Evaluation function

In this study, we define an evaluation function E_{main} for the main path and an evaluation function E_{branch} for the branch paths.

The evaluation function E_{main} is

$$E_{main} = E_{goal} + E_{len}. \quad (1)$$

E_{goal} evaluates the difference from the goal position

$$E_{goal} = dist(GoalPos, EndPos)^2, \quad (2)$$

where, $dist()$ is the Manhattan distance function, $GoalPos$ is the goal position specified by the user, and $EndPos$ is the position of the terminal cell of the main path generated by the program. E_{len} evaluates the difference between maze lengths

$$E_{len} = (TgtMainLen - CrtMainLen)^2, \quad (3)$$

where, $TgtMainLen$ is the length of the main path specified by the user's input, and $CrtMainLen$ is the length of the main path generated by the program.

The evaluation function E_{branch} is

$$E_{branch} = \sum_{i=1}^{n} (TgtBrLen_i - CrtBrLen_i)^2, \quad (4)$$

where, $TgtBrLen_i$ is the length of the i-th branch path specified by the user's input, $CrtBrLen_i$ is the length of the i-th branch path generated by the program, and n is the number of branches.

Multi-objective optimization of these two evaluation functions E_{main}, E_{branch} is performed using genetic programming. A program with an evaluation value of 0 can generate a maze without defects.

VI. Execution results

We implemented the above program and executed for some input text. For example, Fig. 1 shows the execution result of inputting the text 1. The start position of the maze was specified in the upper left corner, and the goal position was specified in the lower right corner. The number of individuals in one generation was set to 100. The depth of the tree-structured programs was set in the range 1 to 8. The maze generation time was about 7 seconds and the number of generations was 58. The start and goal positions of the main path and the length of the maze matched the input information, and the desired maze was generated. The bend, which is a feature of the maze, could also be expressed.

A 6x6 size maze as shown in Fig. 1 could be generated with about 100 individuals. It is considered that a larger number of individuals is required to generate a maze of a larger size. We also realized that some input text structures can not fill in to the rectangle maze grid in principle (this itself is a well known problem).

text1

I ␣ have ␣ a ␣ ha< $split$ >bit.< /$split$ >nd, ␣ I ␣ have ␣ a ␣ s< $split$ >un.< /$split$ >oap.

VII. Conclusion and future works

In this paper, we examined a method for automatically generating a maze with length constraint using genetic programming. We executed our program on some input text, and as a result, for simple input text, we were able to generate the desired maze. Our future works include improving our current method, considering design of user interface to create mazes.

Acknowledgment

This work was supported by JSPS KAKENHI Grant Number JP17K00261.

References

[1] Pullen, W., Think Labyrinth!, Retrieved Jan. 20, 2020, from http://www.astrolog.org/labyrnth.htm

[2] XU, J., AND KAPLAN, C. S. 2007. Image-guided maze construction. ACM Trans. on Graphics (Proc. SIGGRAPH) 26, 3, 29.

[3] Sugiyama, A., NAZONAZO KOUBOU, Retrieved Jan. 20, 2020, from https://sugiyama-akira.jp

[4] Wan, L., Liu, X., Wong, T.T., Leung, C.S., Evolving Mazes from Images, IEEE Transactions on Visualization and Computer Graphics, 16, 2. 2010.

[5] Okamoto, Y., Uehara, R., How to make a picturesque maze, 21st Canadian Conference on Computational Geometry. 2009.

[6] Wong, F.J., Takahashi, S., Flow - Based Automatic Generation of Hybrid Picture Mazes, Computer Graphics Forum. 2009.

[7] Viana, B.M.F., Santos, S.R., A Survey of Procedural Dungeon Generation, 2019 18th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames), pp.391-400. 2019.

[8] Adams, C., Louis, S., Procedural maze level generation with evolutionary cellular automata, 2017 IEEE Symposium Series on Computational Intelligence (SSCI), pp.1-8. 2017.

[9] Segaran, T., Programming Collective Intelligence: Building Smart Web 2.0 Applications, O'Reilly Media. 2007.

[10] DEAP 1.3.0 documentation, Retrieved Jan. 20, 2020, from https://deap.readthedocs.io/en/master/