Author: Taylor Ecton
Date: 2017-09-24

Gear Ball Write-Up

Contents

1. Introduction

This is my solution to Puzzle 2: A* for CS 463. The program models a gear ball puzzle, represented graphically using characters corresponding to the color of the piece (see *section 2.2.1* for more detailed information about this graphical representation), and allows users to both randomize the gear ball puzzle and use the A* algorithm to solve the puzzle (see *section 2.1* for a detailed description of interaction allowed with the program).

1.1. Included Files

in addition to this PDF report, the submission should include an Excel file and a file named "GearBall.tar.gz". The Excel file contains data from my program and is not required (the results are highlighted in *section 3*). After extracting the source files, this should be everything included in the submission:

```
- GearBallWriteUp.pdf
- GearBallData.xlsx
+ GearBall
|----- GearBall.iml
|---- + out
|---- + src
      |----- GearBall.java
      |----- GearBallAStar.java
      |----- GearBallFace.java
      |----- GearBallRandomizer.java
      |----- GearBallSimulator.java
```

In the above file structure, '-' indicates a file, and '+' indicates a directory. I did not expand the "out" directory, because it is output produced by the IDE, and not important to the submission.

1.2. Compiling and Running Program

The source files can be extracted using the command:

```
$ tar -zxvf GearBall.tar.gz
```

from a command line prompt (where '$' is the prompt). The files included in this compressed directory correspond to an IntelliJ IDEA project (the IDE in which the program was developed). To run the program, either open the project folder using IntelliJ (this is what the GearBall.iml file above is needed for) and run the program from inside the IDE, or navigate to the 'src' directory, and use the following two commands to compile and run the code:

```
$ javac GearBallSimulator.java
$ java GearBallSimulator
```

2. Program Description

In this section, I will give a high-level overview of using the program, a breakdown of the individual classes and data structures utilized, and a description of the heuristic function used.

2.1. Overview

On startup, the program greets the user with the following output:

```
Welcome to the Gear Ball Simulator!

Please choose from the following options:
1. Randomize Gear Ball
2. Solve Gear Ball
3. Print Gear Ball
4. Reset Gear Ball
5. Check if Solved
6. Verify Moves
7. Help
8. Quit

(Type 1, 2, 3, 4, 5, 6, 7, or 8 and press 'ENTER'):
```

Below is a description of what each option does.

Number	Option Name	Description
1.	Randomize Gear Ball	This option applies random movements to the gear ball to put it in a random configuration. The program prompts the user for a number of moves to make when this option is selected.
2.	Solve Gear Ball	This option attempts to put the gear ball into a solved configuration using the A* algorithm. See section 3 for a discussion of results.
3.	Print Gear Ball	Prints the current configuration of the gear ball.
4.	Reset Gear Ball	Resets the gear ball to a solved state.
5.	Check if Solved	Prints a statement saying either “The gear ball is in a solved configuration!” or “The gear ball is NOT in a solved configuration!”
6.	Verify Moves	This makes a sequence of pre-determined moves and then makes the inverse sequence of moves to verify that the rotation functions are doing what they are supposed to do.
7.	Help	Prints out a description of each option.
8.	Quit	Terminates the program.

Table 2.1—Menu Options: A description of what each menu option does.

2.2. Classes and Data Structures

The following sections provide descriptions of each class in the program. If the reader is familiar with Puzzle 1 and is primarily concerned with Puzzle 2, feel free to skip to section 2.2.4.

2.2.1. The GearBall Class

The GearBall class (contained in "GearBall.java") represents the gear ball puzzle as a whole. It consists of six GearBallFace objects representing the six sides of the gear all. There is a GearBallFace array that contains each face and can be retrieved using a public member function, and there is also a public member function for printing the graphical representation of the gear ball. The most significant portions of code contained in this class are the methods for performing rotations to the gear ball. Comments in the code provide more information on utilizing these functions.

The printed representation of the gear ball puzzle takes the following format:

```
  _
 |T|
_|F|_
|L|R|
 |B|
 |b|
```

Figure 2.2.1a—Gear Ball Representation

where T is the top face, L is the left, F is the front, R is the right, B is the bottom, and b is the back. The below image is a printout from the program of the gear ball in a solved configuration:

```
               ***************
               * BBB BBB BBB *
               * BBB BBB BBB *
               * BBB BBB BBB *
               * BBB BBB BBB *
               * BBB BBB BBB *
               * BBB BBB BBB *
               * BBB BBB BBB *
               * BBB BBB BBB *
               * BBB BBB BBB *
*****************************************
* RRR RRR RRR * YYY YYY YYY * PPP PPP PPP *
* RRR RRR RRR * YYY YYY YYY * PPP PPP PPP *
* RRR RRR RRR * YYY YYY YYY * PPP PPP PPP *
* RRR RRR RRR * YYY YYY YYY * PPP PPP PPP *
* RRR RRR RRR * YYY YYY YYY * PPP PPP PPP *
* RRR RRR RRR * YYY YYY YYY * PPP PPP PPP *
* RRR RRR RRR * YYY YYY YYY * PPP PPP PPP *
* RRR RRR RRR * YYY YYY YYY * PPP PPP PPP *
*****************************************
               * GGG GGG GGG *
               * GGG GGG GGG *
               * GGG GGG GGG *
               * GGG GGG GGG *
               * GGG GGG GGG *
               * GGG GGG GGG *
               * GGG GGG GGG *
               * GGG GGG GGG *
               * GGG GGG GGG *
               ***************
               * OOO OOO OOO *
               * OOO OOO OOO *
               * OOO OOO OOO *
               * OOO OOO OOO *
               * OOO OOO OOO *
               * OOO OOO OOO *
               * OOO OOO OOO *
               * OOO OOO OOO *
               * OOO OOO OOO *
               ***************
```

Figure 2.2.1b—Gear Ball Printout from Program

2.2.2. The GearBallFace Class

The GearBallFace class implements the representation for each face of the gear ball. The face is a 9x9 char array where the char at index *[i, j]* represents the color at that index. Every three rows/columns of this array can be thought of as a single row/column of the actual face on the gear ball (see *Figure 2.2.1b* for clarity).

The reason for using a 9x9 array instead of a 3x3 array is to allow for representation of the gear states on the edges of the face. The class contains an array of the state of each gear. The gear states are shown below.

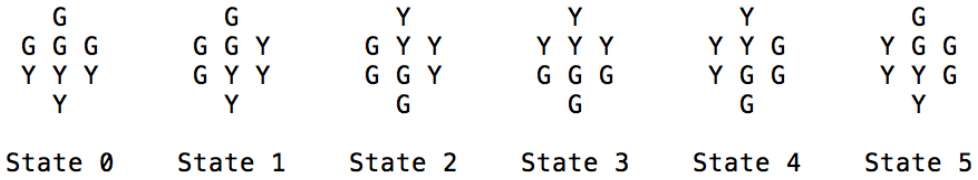

Figure 2.2.2—Gear States: This image shows the six different gear states tracked by each GearBallFace object. Starting at state 0 and rotating clockwise takes the gear to state 1, then 2, etc. Counterclockwise goes from state 5 to 4, to 3, etc.

2.2.3. The GearBallRandomizer Class

The GearBallRandomizer class contains the logic for randomizing the gear ball puzzle. Calling the randomize() method will prompt the user for a number of random rotations to make. The class instantiates a random number generator from Java's Math library and seeds it with the current system time in milliseconds. The random number generator then repeatedly generates a random move number to apply to the gear ball. In Puzzle 1, this class was also the driver for the program, but that functionality has been moved to the GearBallSimulator class (*section 2.2.5*).

2.2.4. The GearBallAStar Class

The GearBallAStar class implements the A* search algorithm for the gear ball puzzle. The class uses Java's built in PriorityQueue for ordering the nodes to search, and also has a TreeMap of these nodes for fast access to node information. The key used for the TreeMap is a string representation of the GearBall object's current configuration. For the list of explored nodes, the class utilizes an ArrayList and only holds onto the string representation of the configuration of that node (since that is all the information needed from it after it is explored) in order to cut down on memory usage. The class also tracks the current node in the tree.

The class contains a nested class, the GBNode class. The GBNode class represents a single node on the search tree. The class implements the Comparable interface by overriding the Object class's compareTo() function. This creates a "natural ordering" for the class, and allows the PriorityQueue data structure to order the objects. The class contains the configuration of the gear ball at this node and also contains an ArrayList of nodes representing the path from the root to this node.

2.2.5. The GearBallSimulator Class

This is the driving class for the program. It prints the menu upon startup and makes calls to the corresponding functions in the other classes.

2.3. Heuristic

This section discusses the heuristic used by my A* search algorithm. I utilize two heuristic values and take the maximum of the two values.

For the first, I take a relaxed view of the gear ball. In this relaxed view, I disregard the gear states. Not counting the gear states, a single rotation misplaces 24 squares—that is 24 squares are not on the same face as their corresponding center square (it would be more if the gears were being considered, but they are not in this view). After two rotations, 40 squares are out of place for this view of the gear ball. Thus, $h_1(puzzle)$ can be defined as:

$$h_1(puzzle) = \left\lceil \frac{\#squares\ out\ of\ place}{24} \right\rceil \qquad \textbf{(1)}$$

The second heuristic function considers the gears that are ignored in the first heuristic, to ensure that they are given consideration. There are 12 gears in total, and it takes 3 rotations to have the 12 gears be in a state other than state zero (gears on a face that is rotated but that remain un-rotated themselves are still considered to be in state zero). The second heuristic, $h_2(puzzle)$ used is defined as:

$$h_2(puzzle) = \left\lfloor \frac{\#\ gears\ not\ in\ state\ 0}{3} \right\rfloor \qquad \textbf{(2)}$$

The final heuristic combines the two into:

$$h(puzzle) = \max(h_1, h_2) \qquad \textbf{(3)}$$

3. Results

This section details the results from running the program. Let k be a randomizing factor representing the number of random rotations applied to the puzzle. For each $k \in \{5, 6, 7, \ldots, 20\}$, the program was run 5 times and data was collected. The values recorded after each instance were: the depth of the solution, the number of nodes in the explored list, and the system time of the execution.

It was found that on my system, with my implementation, that solutions at depths greater than 9 could not be found in a reasonable amount of time. Solutions found at depth 9 were recorded to take up to 15 minutes (more than 6 times longer than the longest time taken to find a solution at depth 8). One execution of the program at *k = 16* was left executing overnight and was unsuccessful at finding a solution. Another run at the same *k* executed for upwards of 2 hours without finding a solution. After these results, I added a print statement letting the user know the time at which a search was started for ease of reference. Subsequent runs were terminated if the execution time exceeded an hour for facilitation of data collection within a reasonable time frame.

In an attempt to reduce the amount of time taken to find a solution, I changed the way A* searches through the gear ball states by reducing the number of moves it could make (an attempt at pruning the search space), but this resulted in unexpected behavior (for instance, a solution was found at depth 7 after only applying 5 moves to the gear ball). This attempt may be revisited at a later date.

Below I have included a graph showing the average number of nodes explored as a function of solution depth. See the included Excel spreadsheet for the complete data collected, as well as a graph of the average system time as a function of solution depth.

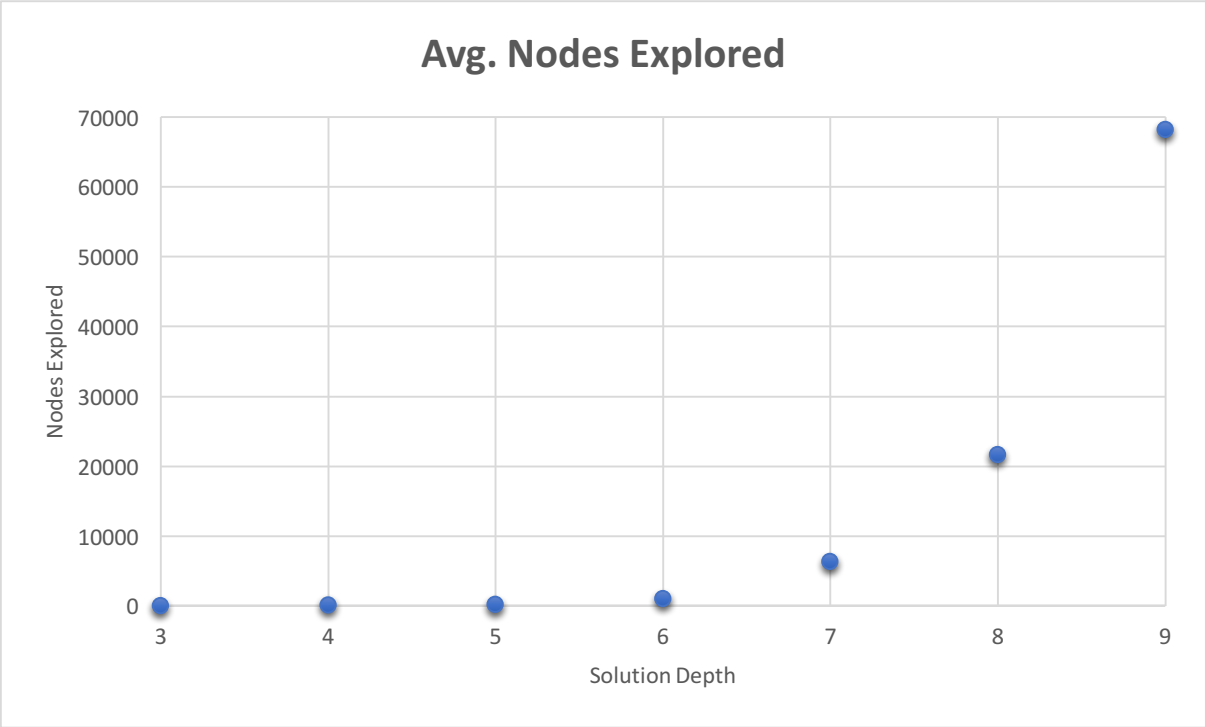

Figure 3.1—Avg. Nodes Explored: This graph shows the average number of nodes explored at each solution depth.

4. Learning Outcomes

From Puzzle 2, I learned:

- How quickly execution time increases in a polynomial time algorithm such as A*. Solutions at depth less than 9 were found quickly, but the growth takes off from there.
- How the representation of the puzzle and the heuristic function used can impact execution times. The initial heuristic used was not implemented correctly, resulting in a program that still worked, but took much longer to reach a solution.
- How programming languages, computer systems, and specific implementations lead to different execution times. Based on the discussion board, my program handled better than some but worse than others.