Mario Level Generation

Procedural content generation is the use of algorithms (procedures) to create novel, and sometimes customized, game content from scratch. Examples of PCG include generation of levels, maps, tree, cityscapes, weapons, monsters, and quests. PCG is often used as a design-time tool to roughly sketch out level content to be refined by human designers. PCG can also be done at run-time to incorporate individual player differences such as skills or preferences. In this project, we look at run-time PCG to create Mario Bros. game levels customized to individual players' play styles. This includes (a) learning a model of the player's play style, and (b) using the model to create a custom level. Fortunately, the first part is already done for you. You must focus on designing and implementing algorithms that use the player information to create something that will evaluate well.

This project will be using the 2011 IEEE Super Mario Bros. Competition infrastructure (Level Generation Track). The description of the competition and some documentation can be found there.

You will write a procedural content generator in the provided Mario Bros. game engine that optimizes level content for different types of players such as those who like to jump, like to collect coins, or like to kill enemies. You will implement a genetic algorithm to tune the layout of the Mario Bros. level.

What you need to know

The Mario Bros. engine is written in Java. You will find the source code in the src/ directory.

You will modify MyLevel.java, MyDNA.java, and MyLevelGenerator.java to implement a genetic algorithm in order to satisfy a variety of "player profiles". Each player profile is an evaluation function focused on a specific type of potential player. You are provided with four player profiles:

- Collector: loves coins
- Jumper: loves jumping
- Killer: loves having to kill enemies
- CloudClimber: loves to be high in the sky
- KillerCollector: loves killing enemies and collecting coins

Each evaluation function for each player profile returns a value between 0-1 (inclusive) to demonstrate how much that player profile "likes" a given level.

dk.itu.mario.engine.level.DNA:

This object contains the chromosome representation used by the genetic algorithm. It represents an individual in the population. This is an abstract base class for MyDNA.

Member variables:

- chromosome: the default string representation for the DNA.
- fitness: the fitness of the individual represented by this DNA.
- length: the length of the level to be generated from this DNA. Not used.

Member functions:

- getGenotype(): return the genotype string.
- setGenotype(string): set the genotype string.
- getFitness(): return the fitness.
- setFitness(double): set the fitness.
- setLength(int): set the level length.
- getLength(): return the level length.
- compareTo(DNA): Return 0 if this object has the same fitness as the argument passed in. Return -1 if this object has lower fitness than the argument passed in . Return +1 if this object has greater fitness than the argument passed in. Used by Collections.sort().

dk.itu.mario.engine.level.MyDNA:

This is a specific version of DNA that will be used for your Mario level generation implementation. You must complete this class. You may choose to use the default string-based representation or add new member variables and functions as necessary.

- mutate(): Return a copy of this DNA object that has been changed in some small way. Do not change the MyDNA object by side-effect. You should make a copy of the current object, make a change to the copy, and return the copy. **You will complete this function.**
- crossover(mate): Cross this MyDNA object with another MyDNA object passed in. Return one or more children. **You will complete this function.**
- toString(): convert this object into a string. By default, this returns the genotype string. However, for debugging purposes you may want to modify this function. **Modifying this function is optional.**
- compareTo(DNA): Return 0 if this object has the same fitness as the argument passed in. Return -1 if this object has lower fitness than the argument passed in . Return +1 if this object has greater fitness than the argument passed in. Used by Collections.sort(). **Modifying this function is optional; it only needs to be modified if MyDNA uses some means of computing fitness other than storing it in the fitness member variable inherited from DNA.**

dk.itu.mario.engine.level.MyLevel:

This object places the blocks of the level. It converts a MyDNA object into a structure that can be rendered. In genetic algorithm terms, MyDNA is the genotype and MyLevel is the phenotype, the organisms that manifests when the chromosome is activated.

- create(dna, type): Converts the MyDNA into a level. **You will complete this function.**
- setBlock(x,y,byte-block-value): Sets the value of the current x,y value with a constant value determining the block type. You can see a list of all of these values in Level.java and see examples of this function being called in MyLevelGenerator.java

- setSpriteTemplate(x,y,SpriteTemplate): Sets an enemy of the passed in SpriteTemplate class to the x,y position. You can see a list of all enemy types in Enemy.java and see an example of this function being called in MyLevelGenerator.java

In addition, this class contains a number of extra functions that show examples of how to create complicated level structures.

dk.itu.mario.engine.level.generator.MyLevelGenerator:

This class implements the genetic algorithm. The basic structure of the genetic algorithm is given, but the details remain to be completed.

- generateLevel(playerProfile): Called by the underlying game engine. Passes in the playerProfile, which contains the fitness evaluation function. This function calls the genetic algorithm and then hands the solution off to MyLevel to create the phenotype level.
- geneticAlgorithm(playerProfile): Genetic algorithm implementation. The playerProfile is the fitness function. This function does not need to be modified, but it relies on a number of functions that must be customized.

See instructions for the functions in MyLevelGenerator that need to be completed.

Instructions

You must implement a genetic algorithm that produces level content that is "liked" by different player profiles as given by the evaluation functions: Collector, Killer, and Jumper, CloudClimber, KillerCollector.

Step 1: Acquire and install Apache Ant (http://ant.apache.org/).

Step 2: In the homework7 directory, build the game engine:

```
> ant
```

Step 3: Modify the following files: homework7/src/dk/itu/mario/engine/level/MyDNA.java, homework7/src/dk/itu/mario/engine/level/MyLevel.java, and homework7/src/dk/itu/mario/engine/level/generator/MyLevelGenerator.java. Complete the following functions:

MyDNA:

- crossover(): This function should return one or more new children from combining the object the function is called on and the object passed in.
- mutate(): This function should return a single MyDNA instance, which is a copy of the object the function is called on. This copy is randomly modified in some small way and the original should not be modified by side-effect.
- toString(): optional (for debugging purposes). The default is to return the default chromosome string, but if the MyDNA implementation is more complex, this will need to be altered to provide useful information.

You can create new member variables and functions as necessary to give the genotype representation the complexity needed to represent Super Mario Bros. levels. At a minimum, you will want to define a string symbol language that represents level elements that can be produced by MyLevel.

MyLevel:

- create(dna, type): This function should take a MyDNA object and translate the genotype into playable level through the use of setBlock() calls. That is, it should read the MyDNA genotype representation and create the phenotype.

If the genotype uses a default chromosome string, then create() *parses* the string and translates elements in the string into setBlock() calls. You may make any new member functions or member variables necessary.

MyLevelGenerator:

- getPopulationSize(): returns a number, the maximum population size. This function needs only set the return value.
- numberOfCrossovers(): returns a number, the number of times crossover will occur per iteration. This function needs only set the return value.
- terminate(): Return true if the search is complete. The population and the number of iterations is passed in.
- generateRandomIndividual(): Create a new individual with random configuration.
- selectIndividualsForMutation(): given a population, select and return an ArrayList of individuals that will undergo mutation.
- pickIndividualForCrossover(): given a population, pick a single individual that will undergo crossover. The excludeMe parameter is an individual that should NOT be picked (this prevents the same individual from being picked twice).
- competeWithParentsOnly(): return true if the population will be reduced by competing children against parents (i.e., use competeWithParents()). Return false if the population will be reduced by competing all individuals against all other individuals (i.e., use globalCompetition()).
- competeWithParents(): return a population of the proper size by testing whether each child is more fit than its parents. The original population, the population after mutations and crossover, and a hash that gives the parents of each individual.
- globalCompetition(): returns a population of the proper size by selecting the most fit individuals from the old population and the new population.
- evaluateFitness(): optional, can use this as is or make any changes necessary.
- postProcess(): optional, use this to perform any additional processes on the final solution DNA.

Step 4: Run your level generator from the homework7 directory:

To run the codebase, you must first compile any changes you've made with the command "ant" in the parent directory. Then, if the code compiled correctly, you can run the game by calling:

```
> java -cp bin dk.itu.mario.engine.PlayCustomized <Profile_Name>
```

With the five player profiles you have access to:

```
> java -cp bin dk.itu.mario.engine.PlayCustomized Scrooge
> java -cp bin dk.itu.mario.engine.PlayCustomized Jumper
> java -cp bin dk.itu.mario.engine.PlayCustomized Killer
> java -cp bin dk.itu.mario.engine.PlayCustomized CloudClimber
> java -cp bin dk.itu.mario.engine.PlayCustomized KillerCollector
```

When you run the application, an image (output_image.png) will be generated in the parent directory which shows the entire generated level. It will also drop a Java GUI window which allows you to play through the level using the arrow keys, 'a' to run, and space bar to jump.

If you die 3 times, Mario will respawn at the top of the screen and fly, allowing you to see the entire level.

Grading

- 5 points: your code generates levels that score >0.8 for all test player profiles (Collector, Killer, CloudClimber, Jumper, KillerCollector),
- 5 points: all of the generated levels should be playable. Playability will be evaluated by an A* agent that can play a perfect game. Additionally, no levels generated should have unreachable blocks surfaces or power blocks, enemies spawning in mid-air, nor incomplete pipes (e.g., without tops). These will be evaluated by visual inspection.

Hints

Make sure that MyLevel.create() is linear time, or close to it. It will need to be called to turn a genotype into phenotype every time an individual is created (via mutation or crossover) and evaluated.

You may wish to create member variables in MyGenerateLevel to keep track of the global fitness of the population and how it has changed over time. You can use this to terminate the search if the search is approach an asymptote or if the global fitness (or max fitness) hasn't changed in a while.

Submission

To submit your solution, upload your modified MyDNA.java, MyLevel.java, and MyLevelGenerator.java.

DO NOT upload the entire game engine.