Intelligent Game Designer

Progress Report

Team Juliet

February 11, 2016

1 Level Design

1.1 Achieved So Far

1.1.1 The Genetic Algorithm for 'Evolving' Designs

We have implemented a feasible-infeasible two-population genetic algorithm for 'evolving' a population of level designs. This maintains two subpopulations - those that are 'feasible' and those that are 'infeasible'. The reason for this separation is that we can save computational costs during the 'fitness' evaluation of a population by not running simulated players on a design that is not yet aesthetically pleasing or even playable (i.e. not yet 'feasible').

1.1.2 Keeping the UI Responsive

Running the genetic algorithm is a lengthy computation. In order to avoid interfering with the Event-Dispatch Thread in Java Swing (the main thread handling UI events), we have wrapped the entire level design computation into a background process which notifies the user interface thread whenever significant changes have been made (e.g. the top 5 designs have changed) so that the user interface can continually display the evolution of the designs to the user.

1.2 Areas Needing Improvement / Completion

1.2.1 Running the Simulated Players More Intelligently

Currently, the LevelDesignerManager class (which handles running and evaluating the performance of simulated players on a particular design) spawns several simulations in multiple threads and takes the average 'difficulty' given by each performance. A smarter implementation would be to initially evaluate the design using the less intelligent (but also less computationally expensive) players, before proceeding to use the more expensive players. This will be implemented when the full range of simulated players are ready.

1.2.2 Making the Simulations Run Faster

If we have time, and we find that running a large number of simulations in parallel is too slow, we could improve the execution time by using MapReduce via Google's App Engine.

1.2.3 Constraints Checker

We need to complete this, which determines whether a design is actually playable.

2 Simulated Players

2.1 Achieved So Far

2.1.1 Basic Simulated Players

We have made two basic simulated players, one that picks the first move it finds and another that picks the move that would produce the largest predicted increase in score. We have also made progress on more sophisticated simulated players.

2.1.2 A* Search or Depth Oriented Searching

We have implemented an abstract simulated player that accepts the depth to which the lookahead search is going to discover, the number of game states in the pool that will be examined at each round (in case there are more only the top ones will be examined) and two functions: the game state metric (that indicates how far from the target we are at a given game state) and the game state potential (that indicates how likely it is that this state is going to lead us close to the target. This type of player also accepts the merging function that accepts the two metrics above and generates a combined metric that will be used for comparison with that of the other states.

This type of design allows for multiple types of simulated players to be created, such as the players that only examine one move at a certain depth or players that do not use the potential function at all (and perhaps do complete state space search). Greedy players also become a special type of this player (it is just that the metric function will be highly biased).

2.1.3 Designing Metric Functions

We are considering several different types of functions from basic ones that simply give weight only to certain moves, to more advanced that combine different characteristics. We are using certain functions to evaluate the difficulty of e.g. removing a certain jelly (since at certain positions it is harder than others).

2.2 Areas Needing Improvement / Completion

2.2.1 Reinforcement learning Simulated Players

We are going to use reinforcement learning techniques to build simulated players. However, in order to do this we need to be able to monitor their behaviour on actual games (i.e. all of the game features have to be implemented).

2.2.2 Simulated Players Assessment

In order to complete this task we need to have the core game mechanics complete and the automated game player complete.

2.2.3 Adding Variety to Simulated Players

We need to use our abstract search class to produce simulated players of different ability.

2.2.4 Simulated Player Manager

Once we have completed assessment of different simulated players we will have to create a manager that would return a simulated player of requested ability to the user interface.

3 Game Implementation

3.1 Achieved So Far

- So far, we have a functional game implementation, that includes swapping candies, clearing the board and generating new tiles.

- Additional special tiles have been implemented, including jellies, liquorice locks and icing.
- The special candies have also been implemented (wrapped, striped and colour bombs), along with their effects (including swapping two special candies).
- Support for ingredients have been added.

3.2 Areas Needing Improvement / Completion

- Ending the game when the turns have run out, or the objective has been met (in either a win or a loss).
- Add a record of progress with respect to objectives (clearing ingredients decreases the objective count etc.)
- Refreshing the board if there are no moves available.
- Make the candies behave correctly with the icing (do not allow candies to fall from above).
- Making the Candy Generators drop the correct number of ingredients over the course of a game.
- Add missing functionalities to the scoring system (e.g. the combos are not yet taken into account).
- Find out how does a colour bomb react when a bomb is detonated next to it.

4 User Interface

4.1 Achieved So Far

- The game interface is now up and running, allowing you to navigate between the different game screens.
- The level creator is complete, allowing you to create boards of variable sizes and save them.
- The level browser is complete, allowing you to delete levels and select levels to view/play.
- The 'watch simulated player' is implemented, allowing you to either call the next move by the click of a button, or to run unaided through the moves. Once more simulated players have been added, these can be added with ease.
- The 'play game' screen is implemented, allowing you to play a selected level design.
- The level requester has been implemented, giving requirements to the level manager. Further requirements can be added later.
- The 'generating level' screen has been implemented, displaying the current best designs created by the manager, which can be selected and viewed/played/saved. Display of additional statistics can be added later.

4.2 Areas Needing Improvement / Completion

- The appearance of the interface could be improved with the use of textures for tiles etc.
- Adding the implementation to select and run created unit tests.
- Implement analysis of designed levels (their predicted difficulties).
- Display further statistics to the 'Generating Level' screen.
- Testing of designed features.
- Handle the end of a played level (completion or loss).

5 Testing

5.1 Testing the Game Implementation

Testing the game implementation essentially boils down to specifying:

- The state of the game before a move
- A move
- The state of the game after a move

Hard-coding such tests is tedious, so instead Ben has incorporated a temporary game implementation unit-test creator into the user interface. This allows us to visually construct unit tests and save them to a file. We have also implemented a class which runs through all such test files and displays whether they succeeded or not.

We will also run real case tests to ensure that the class can operate well under long game periods. This means that we will have simulated players performing valid moves on the board, just to ensure that there are no crushes. The output of the games can also be verified by humans and be included for regression testing.

5.2 Testing the Level Design

We are using the JUnit framework to write and run unit tests for the level design. Most of the tests will be used to check that the genetic algorithm is iterating through generations of level design correctly.

5.3 Testing the Simulated Players

We are using the JUnit framework to write and run unit tests for the simulated players. Most of the tests will ensure that the appropriate metric functions are implemented correctly, the depth specified in the recursion is not exceeded and that the players execute without crashing.

5.4 Testing the User Interface

Currently, we are just testing the user interface by playing around with the application ourselves. For more thorough testing, we could make a list of common tests that we would run manually (e.g. clicking on the board when a move is made, or clicking out of the board). We are aware of ways to automate user interface unit-tests but frankly we believe this is overkill for the time-scale of our project.