Search-based procedural level generation for a 1D "platformer"

Part I:

1. Introduction

For this task I have chosen to implement a simple evolutionary level creation algorithm, which is designed to produce as interesting of an 1D "platformer" level, where there are certain obstacles like bushes and birds, for which the player has to move accordingly. The level design is loosely inspired by the Chrome browser dinosaur game, where the dinosaur has two different types of obstacles, which it can such as a pterodactyl (in my case a bird) and a cactus (in my case a bush).

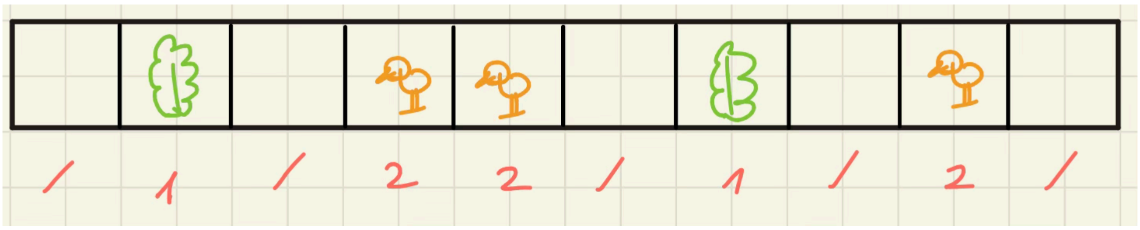

Figure 1: World example sketch

In the example shown in Figure 1, we can see a simple sketch of a possible world generation. Underneath every obstacle (green being the bush and orange being the bird, sorry for the bad sketches:) you can see what "height" does **not** represent a legal move. So if a tile is blank it does not matter what height is the player, if there is a bush the player has to "jump" or in this case have a height of 2 and in the case of a bird the player has to "duck" and have a height of 1.

2. Explanation of the algorithm and implementation

This algorithm starts by creating a population of level layouts - **genomes**, each represented as an array where each element can be empty space, a bush, or a bird obstacle. The population is initialized randomly based on a seed.

Each level in the population is evaluated using a fitness function that scores layouts based on if it the level is **traversable**, which gives a heavy penalty, **balanced**, meaning there are not too many obstacles, where an ideal ratio is set (for this example it is set to 30%), **diverse** and also **interesting**, which in its current state is just defined as having a combination of tiles like **[empty, bush/bird, empty].** The fitness score in its state is very rudimentary and arbitrary, since I put more emphasis on the evolution part of the algorithm itself, because I did not have much time for this project.

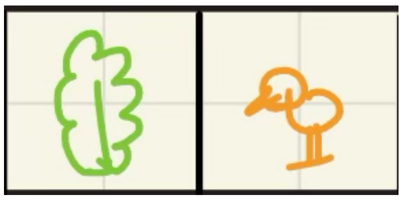
Figure 2: Example of an non-traversable level section

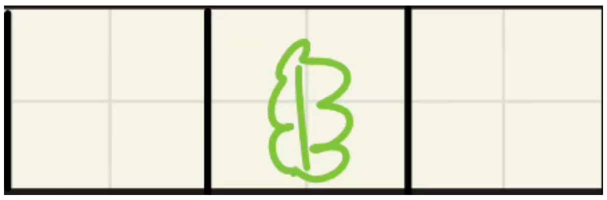
Figure 3: Example of a “interesting” level section

The evolution happens over multiple generations:

- **Tournament selection**: Selects parents by comparing fitness values, choosing the better layouts.
- **Crossover**: Combines two parent layouts by randomly picking each cell’s content from one of the parents to create a child layout.
- **Mutation**: Randomly changes some cells in the child to maintain variation, with a mutation rate that can increase if progress stalls.
- **Elitism**: The top 10% best layouts are preserved directly into the next generation to keep good solutions.

Key functions include:

- **generateWorldGenome(seed, size) :** Generates a random level layout of given size using the seed, assigning obstacles as integers.
- **evaluateWorld(world) :** Calculates the fitness score of a level layout based on obstacle balance, transitions, diversity, and more.
- **crossover(parent1, parent2, rand) :** Produces a child layout by randomly choosing cells from two parent layouts.
- **mutate(genome, mutationRate, rand)** : Randomly alters cells in a layout based on the mutation rate.
- **tournamentSelect(fitness, tournamentSize, rand) :** Selects a parent layout based on tournament selection from fitness scores.
- **evolveWorlds(population, generations, mutationRate, rand) :** Runs the entire evolutionary process over many generations, applying selection, crossover, mutation, elitism, and tracking fitness.
- **plotEvolutionGraph(avg, best, worst) :** Displays a graphical plot of fitness values over generations.

- **visualizeBestWorlds(bestWorlds) :** Shows a color-coded image of the best layouts evolving throughout the generations.

3. PCG taxonomy of the algorithm

The algorithm is **offline**, as it runs ahead of gameplay for level optimization. It is **adaptive**, since it iteratively refines solutions based on feedback. It is **stochastic**, due to the probabilistic genetic operations. Given that the human designer defines the goal-setting parameters (fitness function and structure) while the algorithm autonomously searches and improves solutions, it can be classified as **semi-autonomous** or **human-in-the-loop**, representing a **mixed-authorship** approach with user-guided objectives guiding autonomous search.

4. Visualization of the algorithm

The algorithm is visualized in the windows created after the evolution is finished.

Firstly, the graph window, which visualizes the evolution process by plotting the average, best, and worst fitness across generations, providing a clear view of progress and diversity within the population. This comprehensive visualization helps in analyzing convergence and exploring the range of the level generation over time.

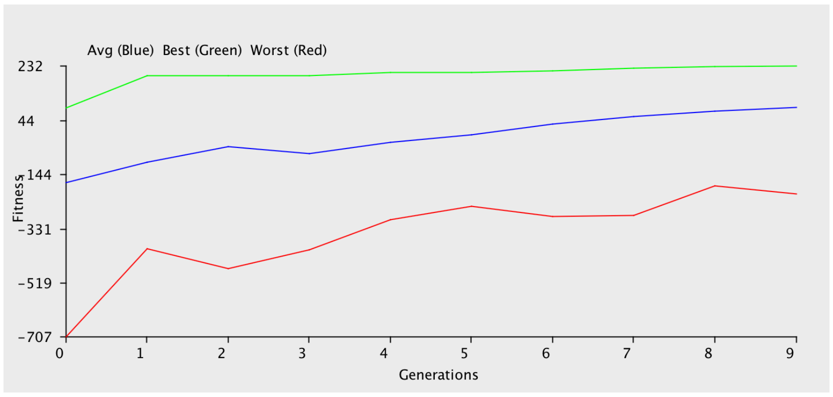

Figure 4: Fitness score graph per generation

And secondly, the algorithm also generates a graph, where it visualizes the the best scoring generated level per generation, where a white cell represents an empty cell, a blue one represents a bird and a green one a bush.

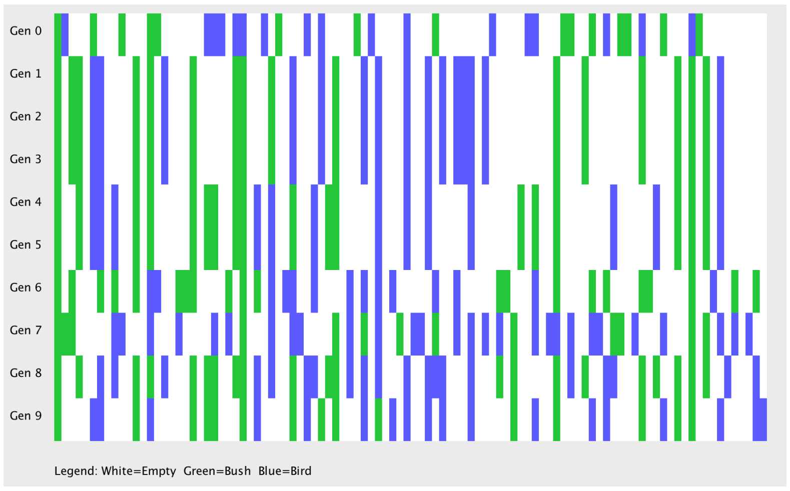

Figure 5: Top performing level visualizer per generation

5. Expressive range of the algorithm

The algorithm's expressive range is currently limited by the 1D platformer setup and simple move representation. It can generate varied levels within this restricted possible environment. However, its capacity to produce diverse or complex content is constrained. Expanding to higher-dimensional levels or richer world/obstacle representations would significantly increase the range of possible world that are generated.

Part II:

1. Critical reflection over the algorithm, content and evaluation

The evolutionary algorithm effectively generates and evolves 1D worlds composed of terrain elements such as empty spaces, bushes, and birds. It demonstrates core procedural content generation principles, using selection, crossover, and mutation to iteratively refine level design based on fitness evaluations. The fitness function balances several aesthetic and structural goals, rewarding obstacle diversity and penalizing unrealistic or repetitive patterns. However, the simplicity of the 1D representation limits the expressive potential of the generated worlds, and the evaluation metrics remain largely structural rather than gameplay-oriented. The visualization tools provide helpful insight into average, best, and worst fitness trends over generations, though deeper pattern-based analysis or player-centric evaluation would offer a more comprehensive understanding of level quality. Overall, it succeeds as a clear demonstration of evolving environmental content, but its design scope remains modest and could be expanded for more complex procedural generation tasks, but since I did not have the time I will leave it as is.

2. What would I do different, what worked, what didn't, what would I do next

The current implementation successfully evolves levels with improved obstacle distribution and structural diversity, showing steady progress across generations. However, the fitness function could be refined to better reflect gameplay qualities, such as playability, pacing, or challenge, rather than just aesthetic balance. Despite its limitations, the experiment worked well as a foundation for understanding evolutionary level generation and provides a solid framework for future improvements in both scale and sophistication.

3. The use of AI in the task

As in the previous task AI was used in this task for proof reading the report and to help generate some minor helper with discrete helper functions when manual effort could be better allocated elsewhere. Additionally, AI was used to help me with Java's GUI library in which I do not have much experience.