P5: Evolving Mario Levels

1 Introduction

In this assignment, you are to write a program that finds interesting and playable Super Mario Bros. levels through an evolutionary algorithm. You will explore two different encodings of the Mario level, tweaking fitness functions and evolutionary operators in both encodings to produce cool levels, and ultimately play your colleagues' levels to rank them.

In order to do this, you will need to implement a selection strategy, crossover and mutation operators, fitness calculations, and possibly population initialization code for each of the given encodings. But before this, you will need to install a few dependencies.

2 Prerequisites

We need a few different Python libraries, first of all. Using either `pip` or `pip3` as appropriate,

```
pip install numpy scipy.
```

You'll also need (any version of) Unity3D installed. Why Unity? Well, we have a Mario simulator which requires the Unity editor to run. Once you have Unity installed, open up the `Player` project in Unity, load the GameWorldScene, and hit the play button in the editor to play around.

3 Genetic Algorithms

Genetic algorithms (sometimes called evolutionary algorithms) are a type of stochastic search algorithm suitable for searching large possibility spaces to find high-quality and diverse solutions. The key idea behind genetic algorithms is that two solutions might be good in some ways and bad in other ways, and combining these solutions could lead to a better solution.

In genetic algorithms, we describe a problem in terms of an *encoding* or *genome,* which is necessarily problem dependent. We start with a population of *individuals* with distinct and diverse genomes, determine the *fitness* of each individual, then repeatedly *select* which individuals should reproduce, *crossover* genomes between reproducing pairs, and *mutate* new individuals to produce a new population. Optionally, we may choose to stop after some condition is met (e.g., no improvement in the last ten iterations, or fitness exceeds some threshold).

In this project you are given two encodings. The first (`Individual_Grid`) treats a level as a grid of tiles represented as characters, similar to the included `level.txt file`. The second (`Individual_DE`) treats a level as a sequence of *design elements* like platforms, gaps, or blocks, and produces a level on the fly based on this description. In genetic algorithms, as in many search problems, the *representation* we use has a huge impact on our overall performance; a substantial part of this assignment lies in exploring the tradeoffs between these two representations: balancing predictability, controllability, surprise, novelty, computational efficiency, and other considerations in the pursuit of good, interesting Mario levels.

3.1 The Grid Encoding

First, let's explain the `level.txt` format, which is identical to the `Individual_Grid` encoding. We assume that the leftmost and rightmost columns are fixed, that the rows are ordered with 0 at the top and height (16 in our levels) at the bottom, and that all levels are the same size (`width*height`). The allowed values for a character are:

- An empty space

X A solid wall

? A question mark block with a coin inside

M A question mark block with a mushroom inside

B A breakable block (if you are big Mario)

o A coin floating in the air

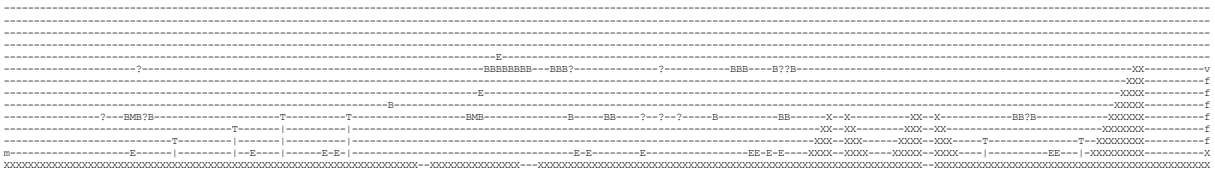
| A vertical pipe segment (the position to the right of this will be overwritten with the other side of the pipe automatically)

T A pipe top segment (the position to the right of this will be overwritten with the other side of the pipe automatically) f A goal flag (do not generate this in your level)

v A goal flagpole (do not generate this in your level)

m Mario's start position (do not generate this in your level)

Mario 1-1 as such a grid:



And as a set of tiles (ignore the underground coin room please):

To determine the fitness of such a level, we call out to a metrics module to get some measurements on the level, then multiply some of those measurements according to some weights we essentially pulled out of a hat. The built-in metrics include the following list. Please see Summerville et al for details.

- `length` (Should always be `width`)
- `negativeSpace`
- `pathPercentage`
- `emptyPercentage`
- `decorationPercentage`
- `leniency`
- `meaningfulJumps`
- `jumps`
- `meaningfulJumpVariance`
- `jumpVariance`
- `linearity`
- `solvability (1 if solvable, 0 otherwise)`

It is your job to implement crossover for this type of individual. You can do this in `generate_children` by deciding, at each position in the new child's genome, whether to take the left parent or the right parent's gene. One approach is to randomly pick one or the other parent's gene at each position, possibly with some bias; this is called *uniform crossover.* Another is to pick a position in the genome (or multiple positions) and take the first parent's genome up to the position and switch to the second parent's genome afterwards; this is called respectively *single-point* or *multiple-point crossover.* Note that `generate_children` can return 0 or more new children; either of the above techniques could produce a second child which has all the pieces of each parent which were "left behind" by the first child.

After crossover, you must implement a mutation operator. Mutation is a way to produce more diversity in the population. Generally you determine a mutation rate either for an individual or for each gene and you perform some action to perturb the genome based on a random chance tied to that mutation rate. You are responsible for implementing mutation in the mutate function of Individual_Grid.

Test your changes to the first encoding by running `python ga.py`, hitting ctrl-c when you're finished. The output of each generation will be emitted into `levels/last.txt` and some samples from the last generation will be placed in other timestamped files in that directory. Before you can proceed, however, you will need to implement generate_successors, which selects (based on fitness) some combination of the previous population to breed new individuals (by calling `generate_children`).

Selection is a key function in genetic algorithms and must be written to balance having a diverse population with having an increasingly more fit population (you might recognize this as the explore/exploit tradeoff). You are responsible for researching and implementing at least two selection strategies in `generate_successors`, filling up the results array with a new population. Some suggestions include random selection, roulette-wheel selection, elitist selection, or tournament selection, but feel free to find and use others—just explain yourself in your writeup! It is *your responsibility* to look up these selection criteria and implement them in ga.py.

3.2 The Design Element Encoding

The second encoding, `Individual_DE`, is quite different. Instead of being a fixed grid, its genome consists of a variable-length sequence of *design elements* per Sorenson & Pasquier. This puts a lot more domain knowledge in and can obtain better-looking results faster, but the output may look more predictable or dull. You can imagine that the genotype is this sequence while the phenotype is a gridded level. Starting with the assumption of a flat ground plane, a design element at position x layered on top of this is one of:

- A hole of width `w` in the ground plane
- A platform of width w at y-position y made up of blocks of type t (either `?`, `X`, or `B`)
- An enemy
- A coin at height `y`
- A block at height `y` which may or may not be `breakable`
- A question-mark block at height y which may or may not `contain-powerup`
- A stairs structure (ascending/descending blocks) with a `height` and `direction` (-1 for left, +1 for right)
- A pipe of height `h`

Note that turning a list of design elements into a level grid applies all of the holes from left to right, then all of the platforms, then all of the enemies, and so on, and later design elements may overwrite earlier ones (in the code, these type names include numerical indices for tiebreaking). DE parameters are constrained within the width and height and are not allowed to modify the first or last column; their effects are simply cut off at the edges. This figure from the referenced paper shows the six basic element types (we have added question mark blocks and free-floating coins):

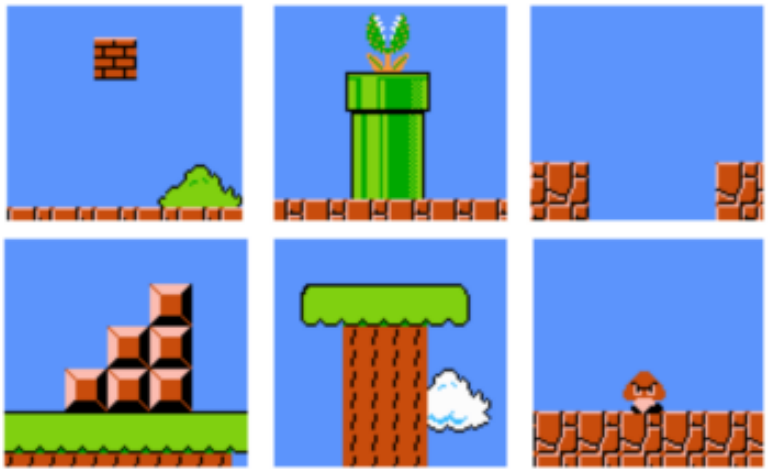


Fig. 1. The 6 DEs for SMB level design. Clockwise from top-left: block, pipe, hole, enemy, platform, and staircase DE.

You are given an implementation of *variable-point crossover* and are responsible for explaining the implementation and why it effectively produces diverse levels with different design elements and different numbers of design elements. Pay special attention to the design element genome and how it's represented (referring to the Sorenson & Pasquier paper will help). You are also given a mutation operator for this encoding and are responsible for explaining its operation as well (the `offset_by_upto` function takes a value and a variance and offsets the value by up to the variance positively or negatively according to a normal distribution, clipping the result within a given range).

You are responsible for customizing both representations' fitness functions to achieve your design objectives. Note that for the DE encoding, you can use information about how many of each type of design element you might want, whether you want to enforce certain constraints between design elements, et cetera. If a single fitness function doesn't seem to help, refer to the Sorenson & Pasquier paper for an enhancement called FI-2POP which maintains two separate populations (one of solvable and one of unsolvable levels) and considers "fitness" to mean something different in each of them, placing new offspring in one or the other population according to whether it's solvable. Another aspect of the algorithm you may choose to customize is the population initialization. This can be an important way to seed a population with useful biodiversity. The provided implementation gives a mix of empty and random individuals, but you might choose to change the way these are produced or perhaps include portions of real Mario levels in the seeds (or compare against real Mario levels as part of your fitness calculation).

4 Provided code

- `metrics.py` defines some useful metrics for Mario levels, per Summerville et al.
- `pathfinding.py` is also due to Summerville, it's just a basic Dijkstra's implementation you can likely ignore.
- copy_level.py is a utility script to load up game levels into the Unity player. Run it and then hit Play in the editor for each level you want to try out.
- `ga.py` is where you will be making most of your changes.
  - Genetic algorithms are naturally parallelizable, and the skeleton you are given uses Python's `multiprocessing` module to calculate fitnesses in parallel (generally the most expensive step of a genetic algorithm).
  - As the search proceeds, you can look at `levels/last.txt` to see the best level from the most recent generation.

5 Your Job

- **Implement `generate_successors` using at least two selection strategies to build up the next population**. You can use elitist selection, roulette wheel selection, tournament selection, or any other approach considering the individuals based on their fitnesses (this is a key element of the search and you should play with different approaches and parameters to get good results). Eventually, it should call `generate_children`.

• **Implement crossover in `generate_children` for the Grid encoding**. You must describe in your writeup whether you use single-point, uniform, or some other crossover. Note that the skeleton provided produces just one child, but you can produce more than one if you want (or even have more than two parents). If you want to use single- or multi-point crossover, consider whether you're doing it by columns, by rows, or something else.
• **Implement mutation in `mutate` for the Grid encoding.** You must explain your mutation rate and operator in your writeup.
• **(Optional, for better results) Improve the provided `calculate_fitness` function for `Individual_Grid`**. If you like, add new metrics calculations to `metrics.py`.
• **(Optional, for better results) Improve the population initialization.** You can do this by modifying empty_individual, random_individual, and the line of code after the label STUDENT: (Optional) change population initialization.
• **Switch the encoding to `Individual_DE` and explore its outputs; write down an explanation of its crossover and mutation functions, with a diagram if possible.** Change the line `Individual = Individual_Grid` to `Individual = Individual_DE` to switch encodings. Consulting the Sorenson & Pasquier paper may help.
• I**mprove the fitness function and mutation operator (and potentially crossover) in `Individual_DE`.** It's OK if it's different from the Grid version; you may want to make assumptions about how many of the different design elements you want and use that to feed into your fitness function.
• **(Optional, for extra credit and better results) Implement FI-2POP from the Sorenson & Pasquier paper.** Extra credit. Will require some changes `to ga()`.
• **Pick 1 favorite level from either encoding.** Copy out a `.txt` file from `levels/` with a shape you like. You can also change the code in `ga.py` to output more/different levels from your population.
• **Play that level in the Unity player and make sure you can beat it.** Use `python3 copy-level.py lev.txt` to copy your level to the right place.

6 Submission

• Your modified `ga.py` code and `metrics.py` code (if you changed it).
• Your 1 favorite (beatable!) level, named `lastName1_lastName2.txt`.
• A writeup detailing:
  – What you changed from the template and why, especially related to your selection strategies, fitness functions, crossover and mutation operators, etc.
  – Something about each of your two favorite levels: Why do you like them? How many generations did it take and how many seconds to generate these levels?

Remember that a good writeup can make up for a lot of problems in the actual submission. After the initial submission, we will collect levels for a contest and establish a new submission: P5-Contest. For this, you must submit a text file with a particular format (described below).

7 Evaluation

• (7 points) The modified code for `ga.py` and `metrics.py` (if you changed it), meeting the requirements above (roughly: 4 points for `generate_successors`, 3 points for changes to each of the two encodings).
• (3 points) Your writeup and level files. Your writeup should also state whether your team intends to participate in the competition.
If your team enters into the competition, you will also be responsible for evaluating some other entrants' levels:
• Play 10 Mario levels made by other teams. You can use `python3 copy_level.py lev.txt` to copy each level into the right place, then run the level as usual. After playing a level, write down the level name, whether you could beat it, and two scores (fun and visual appeal; each score is 1-3).
• If you do not evaluate other teams' games, you will be disqualified from the competition.

When you're done, submit a single text file formatted like so. We will process these automatically so be

sure to stick to the format.

```
levelname.txt
Beat: Y
Fun to play: 3
Nice to look at: 2
levelname2.txt
Beat: Y
Fun to play: 2
Nice to look at: 2
levelname3.txt
Beat: N
Fun to play: 1
Nice to look at: 3
```

...

The team with the best levels according to student ratings will receive extra credit for the course. 5