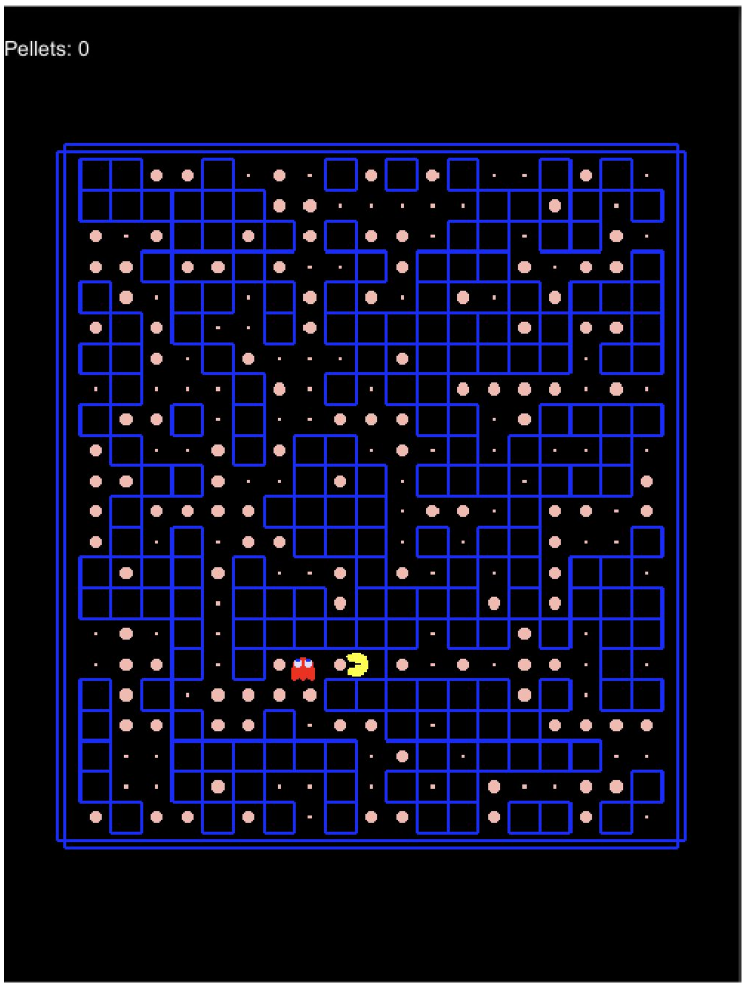
Assignment 5: Genetic Algorithm for Procedural Level Design

The purpose of this assignment is to make you familiar with procedural content generation for level design. Genetic Algorithms are one common method for creating game content that adapts to player behavior (e.g. Darwin's Demons or Petalz). In this assignment you'll implement a genetic algorithm to produce levels to match several AI "evaluators" as a stand-in for different types of human players. Essentially, creating a level generator that adapts its levels to better suit potential players.



Above you can see an initial random level generated for this assignment. You'll implement code to take initially random levels and adapt them to fit a particular player preference. For example, some players may like levels with tight corridors, levels with more collectibles, or levels with less challenge.

What you need to know

There's only one script that you'll be editing for this assignment (LevelOptimizer.cs). In addition, there's a script (Level.cs) that you'll be working with directly and another type of script (Evaluator.cs) that you'll be working with indirectly. Different classes inheriting from Evaluator.cs will give different criteria that will require different level (Level.cs) values. These are the player preferences discussed above. Your job will be to alter LevelOptimizer such that it can produce Level values that fit any given Evaluator and goal value.

The basic idea is that an Evaluator will give a score 0-1 for a level. You'll also have a goalValue, which is the value you want the Evaluator to score your level at. For example, if I have an Evaluator (HungryEvaluator) that loves Pellets, and I pass in a goalValue of 0.5, then that indicates a perfect level should have exactly half the number of possible pellets. You don't have to do this by hand, changing a random level to fit these criteria is what the genetic algorithm is for.

LevelOptimizer

LevelOptimizer handles the genetic algorithm, and all your procedural content generation code for this assignment. Unless you choose to use your own GridHandler, it should be the **only thing you turn in for this assignment**.

Member variables:

- width: Half of the width of each level.
- height: Half of the height of each level.
- gridSize:The standard grid size for each level
- POPULATION_SIZE: A variable you may use to track the size of your population.
- MUTATION_RATE: A variable you may use to determine with what probability to mutate a member of your population.
- NUM_CROSSOVERS: A variable you may use to determine how many crossovers you'll have (equivalent to how many children to produce) in each iteration.
- MAX_ATTEMPTS: A variable that you may use to track the max number of attempts for your genetic algorithm (maximum number of generations).
- squareObstacle: A simple set of points that can be used to define a square obstacle exactly the size of a grid cell. (You may wish to create other basic obstacle shapes like this, but it's not necessary to get a 10/10 on the assignment).

Member functions:

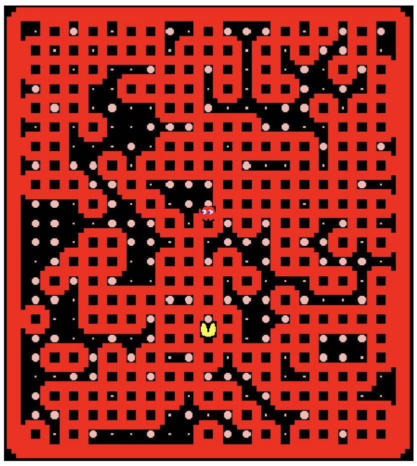
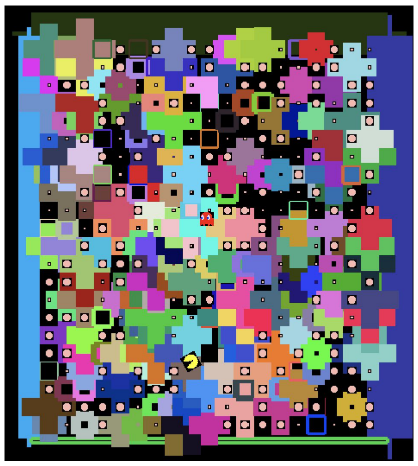
- Level GetLevel(Evaluator evaluator, float goalValue): The main function of this assignment, and where you'll need to implement your genetic algorithm. It takes in an evaluator and goalValue that can be used to determine the quality (fitness) of a particular level. It should return the best possible level relative to that evaluator and goalValue. **You will need to alter this function so that it correctly implements a**

genetic algorithm to convert from random levels into levels that satisfy the given criteria (Evaluator and goalValue).

- float Fitness(Level l, Evaluator evaluator, float goalValue): The basic fitness function for this assignment. Takes in a level, evaluator, and goalValue and returns a float (from 0-1) that indicates how close this level is to matching the particular evaluator and goalValue. You may want to alter this function for the extra credit, but it's not a requirement.
- Level Mutate(Level level): Takes in a level object and returns a slightly different (or mutated) level. You will have to use this function or similar code in GetLevel. You may find it helpful to modify this function, but it is not necessary for this assignment (though it may be helpful for the extra credit).
- Level Crossover(Level parent1, Level parent2): Takes in two levels and outputs a level that is a mix or "child" of these two levels. You will have to use this function or similar code in GetLevel. You may find it helpful to modify this function, but it is not necessary for this assignment (though it may be helpful for the extra credit).
- Level Sample(List<Level> population, Evaluator evaluator, float goalValue): Grabs a value from the passed in population by sampling based on the different Level fitness values. You will have to use this function or similar code in GetLevel.
- List<Level> Reduce(List<Level> population, int populationSize, Evaluator evaluator, float goalValue) Takes in a population and a populationSize and passed back a List of populationSize of only the best levels in terms of their fitness. You will have to use this function or similar code in GetLevel.
- Level GetRandomLevel(): Generates a random level. You will have to use this function or similar code in GetLevel. You may find it helpful to modify this function, but it is not necessary for this assignment (though it may be helpful for the extra credit).

Level.cs

This script holds the Level class, which is the basic representation of a specific level. You **will not** be making any changes to this script as its only for storing level data, but you will work with it. A particular Level object will specify a particular level. You can get pretty wild with this (see examples below).

Member public variables:

- pellets: A list of ProtoPellet objects that represent the pellets in this level in terms of their location (Vector3 worldPosition) and whether or not it is a power pellet (bool powerPellet). Your code will be changing this value in LevelOptimizer.
- obstacles: A list of ProtoObstacle objects that represent the obstacles in this level. For the base assignment you only need to specify these in terms of their location (Vector3 WorldPosition) and shape (Vector2[] shape), but you have the option to impact the obstacle colour (lineColor) and line width (lineWidth) as well, which may be helpful for extra credit.
- ghostStartPos: The start positions of the ghosts for this level. It is not necessary to change this value, but it may be helpful for the extra credit.
- pacmanStartPos: The start position of Pacman for this level. It is not necessary to change this value, but you may find it helpful for certain Evaluators, but it may be helpful for the extra credit.
- fitness: Used only for the Reduce function in LevelOptimizer, meant to store the fitness for this level. You will not need to interact with this variable at all.

Member functions:

- Level: There are many constructors for Level, which basically determine how much information will be specified. You do not need to use any of them in your implementation, but you may find it helpful.
- Level Clone(): Returns a clone of this level. Useful for the Mutate function in LevelOptimizer.
- Vector2[][] GetObstaclePoints(): Returns all of the obstacle points of this level, which can be used in rendering this level. You will not need to interact with this function at all.

Evaluator.cs

Scripts that inherit from Evaluator will be how your genetic algorithm in GetLevel of LevelOptimizer knows what kind of level to produce. There’s a single function EvaluateLevel, which takes in a level and outputs a float, which can vary from 0-1. This value plus the goalValue will guide your genetic algorithm. There are three scripts that inherit from Evaluator given to you. (1) ClaustrophobicEvaluator, which returns 0 for a level full of obstacles and 1 for a level free of any obstacles, (2) ObstacleLoverEvaluator, which returns 0 for a level with no obstacles and 1 for a level full of obstacles (except it will return 0 if Pacman spawns in an obstacle), and (3) HungryEvaluator, which returns 0 if the level has no pellets and 1 if the level is full of pellets.

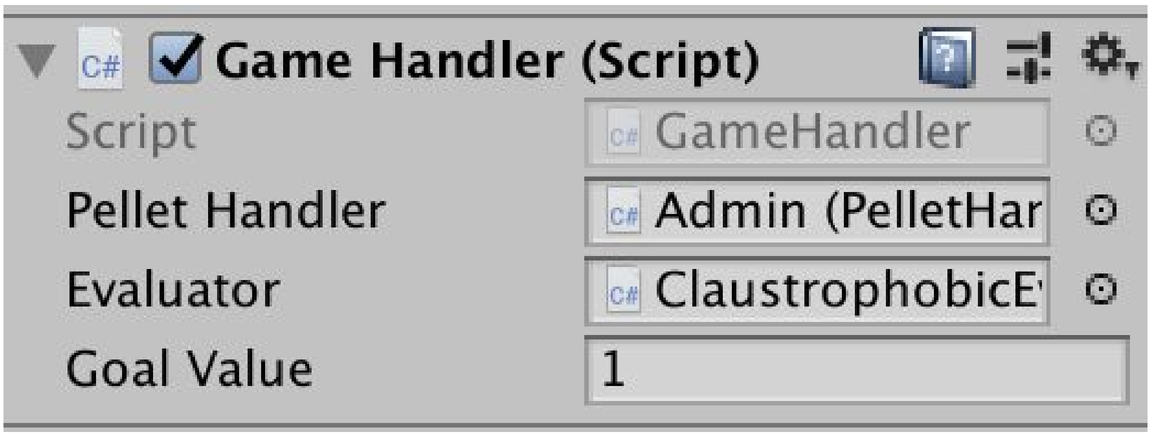
Instructions

By default the project will create ten random levels and then return the random level with the highest fitness given a particular Evaluator and goalValue. Your job is to instead optimize these levels with a genetic algorithm (implemented in GetLevel) such that they fit the Evaluator and goalValue as closely as possible.

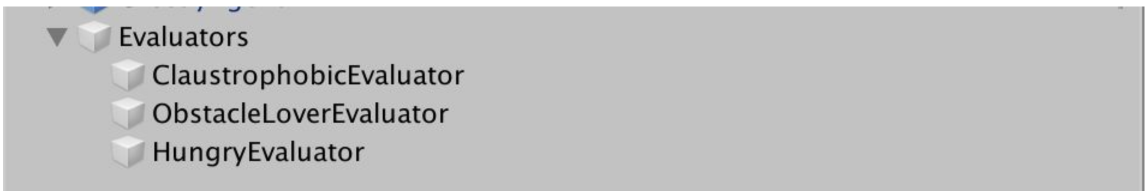
Step 1: Download the Assignment5.zip file from eclass, unzip it, and open it via Unity. You can open it by either selecting the unzipped folder from Unity or double clicking the “Game” scene.

Step 2: Open “Game” located inside Assets/HW5. Hit the play button. You should see a random level appear. Hit the play button a few more times to see what kinds of random levels are produced by default. Note the changing fitness print out at the bottom left. If you wait long enough, Blinky will spawn and begin to try to chase Pacman (though the level may make this impossible).

Step 3: Swap out the Evaluator and goalValue and see how this changes the fitness. By default the ClaustrophobicEvaluator is being used with a goalValue of 1 (this means that a fitness of 1 would mean that the level was entirely free of obstacles). If you change this value to 0.5 then you’ll get a fitness value of 1 if half of the level is covered in obstacles, etc. You can find the other Evaluators in the scene via the Hierarchy view (nested beneath an “Evaluator” object. Swap out the Evaluator by clicking and dragging it onto the GameHandler component of the Admin object in the Hierarchy view.


Game Handler (Script)

Script	GameHandler
Pellet Handler	Admin (PelletHar
Evaluator	ClaustrophobicE
Goal Value	1


Evaluators
ClaustrophobicEvaluator
ObstacleLoverEvaluator
HungryEvaluator

Step 4 (Optional): Import your own previous GridHandler, AStarPathFinder, and ghost code from the prior assignments, using the instructions from those assignments to do so. The only one of these that might impact the grading for this assignment is GridHandler, and even then, only if you get really creative with your generated levels.

Step 5: Implement a genetic algorithm in the GetLevel function of LevelOptimizer.cs We will go over this approach in more detail in an upcoming lecture video. But for reference the pseudocode for a genetic algorithm is as follows:

```
population = initialize random initial population of size X
attempts = 0

while(best_fitness(population)<threshold and attempts<MAX_ATTEMPTS){

        attempts++
        population = MutatePopulation(population)
        population +=CrossoverPopulation(population)
        population = Reduce(population, X)

}

return best_fitness_object(population)
```

MutatePopulation should check each member of the population and with some probability mutate it. A mutation in a genetic algorithm is some small change to that member of the population.

CrossoverPopulation produces the next “generation” of the population by choosing two members of the population based on their fitness (a measure of quality, think like a heuristic), and creating a child that is a mix of their attributes.

Reduce: Takes some population and reduces it back to some size X, taking only the X best members based upon their fitness.

Step 6: Test your code to ensure that it can generate levels that are able to achieve >=0.9 with the default fitness function for any Evaluator and goalValue.

Step 7 (Optional, Extra Credit): Make changes to your level editor so that it produces levels that are as much like the original Pacman levels as possible or alternatively that represent entirely new kinds of level designs compared to the original Pacman levels.

Grading

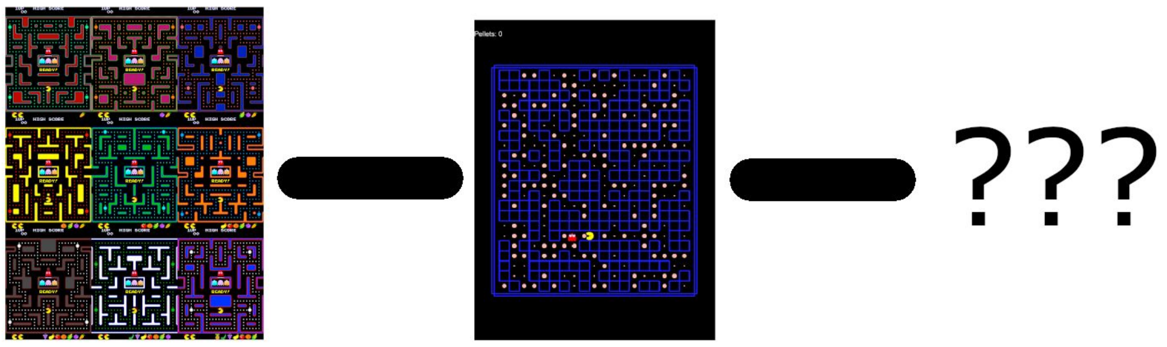
This homework assignment is worth 10 points. Your solution will be graded by an autograder. The point breakdown is as follows:

4 points: A correct implementation of a genetic algorithm. This will be tested by monitoring your code's calls to the Evaluator and whether the fitness rises and falls in ways that demonstrate a population that is iteratively being improved, and occasionally becoming slightly worse through a random mutation. As long as you don't just implement greedy search and don't just generate a large number of random levels and return the best one, you should be fine.
6 points: Your code should be able to satisfy an arbitrary Evaluator and goalValue. You will gain 1 point for each Evaluator your code successfully satisfies. Specifically, the autograder will test six Evaluators with 10 goal values (60 tests). If your code is able to achieve >0.9 fitness for that Evaluator and value you will gain 0.1 points. You are given three of the Evaluators that will be used for this test, but three more will be withheld from you.

In addition you can receive up to 2 extra credit points on this assignment.

For the first point of extra credit, your levels should all be *playable*, meaning that it should be possible for a Pacman agent to collect all of the available pellets in a level. Of the 60 levels the autograder will have your approach produce you will gain a proportion of 1.0 extra credit point for each level where it is possible to collect all the available pellets. Note: levels with no pellets this will automatically pass this check.

You have two options to receive the other extra credit point. Either you can ensure that your output levels match the level design style of the original Pacman levels (walls of consistent lengths and colour, horizontal symmetry, regular pellet placement, ghost spawn position unchanged, pacman spawn position unchanged, consistent challenge to the original levels) or you can produce something entirely distinct from the original Pacman levels but with a consistent "style". You can imagine the extra credit as a linear scale with both ends of the scale worth 1 point and only using the basic building blocks with 0 extra credit points, as you can see below:



For the left side I will be looking to ensure that your levels match the original Pacman in terms of the features stated above, with points given based on to what extent it matches. For the right side, I will use the same features as above, but I will check for the consistency of the differences to the original levels. Those who go for the right side should have generators that produce a consistent, but entirely distinct, style of level. You can get really creative with this, creating what is functionally an entirely new kind of game.

Hints

You have a lot of freedom for this assignment. However, the instructor solution gets a 10/10 from the autograder and only makes changes to the GetLevel function of LevelOptimizer.cs, though notably it receives no extra credit points.

A good implementation can run for a long time. The instructor solution runs for two minutes on a 2019 MacBook pro.

Look to Piazza for general help and email for specific questions.

You might find it helpful to implement your own scripts that inherit from Evaluator.cs for testing purposes, you can look at the three examples as a basis for this.

You will be using a lot of Random values for this assignment. Note that Unity has its own Random library, which you can find the documentation for here: https://docs.unity3d.com/ScriptReference/Random.html

Submission

To submit your solution, upload your modified LevelOptimizer.cs file. If you used your own GridHandler.cs, you should submit that as well. If you went for the extra credit and you believe it’s necessary for your implementation, you may upload your AStarPathFinder and your code from Assignment 4 (the custom ghost, pinky, and clyde code).

You should not modify or upload any other files in the game engine.

DO NOT upload the entire game engine.