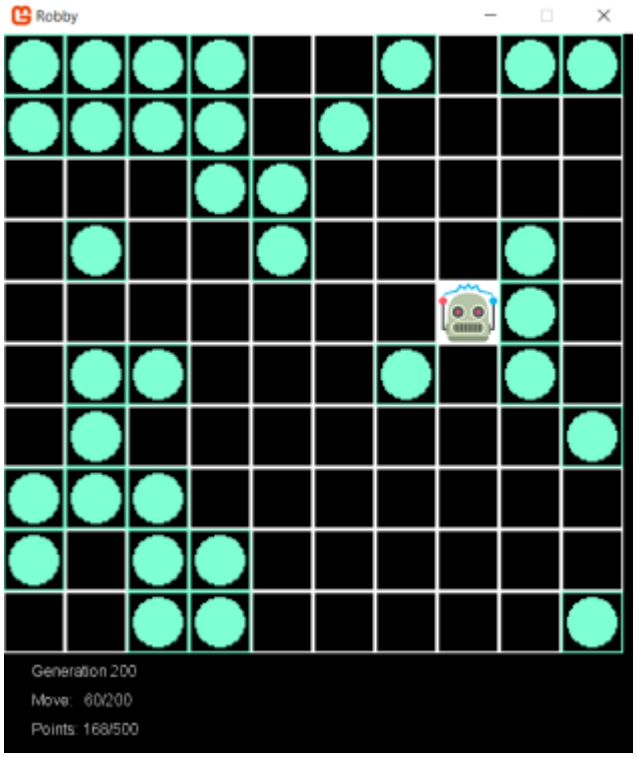
Advanced .NET 420-411-DW

Project Two: Genetic algorithms and Robby the Robot

Due date: Via Gitlab submission March 31st, end of day.

Teams of two required

Robby the Robot is described in Chapter 9 of Melanie Mitchell's book Complexity: A Guided Tour. Robby's task is to collect empty soda cans that lie scattered around his square grid world, by following instructions encoded in an array of 243 genome enums. A number of implementation of this algorithm can be found on-line given that it is a popular problem in introductory AI courses. Unfortunately none are following the same specs as this one! Sorry not sorry…



In this assignment, you will:

- code and test generic classes that can be reused in other genetic algorithm projects with minimal changes
- code specific classes and methods to implement a genetic algorithm to evolve control strategies for Robby the Robot
- use the Monogame framework to help visualize how Robby improves over generations.

Summary (from Melanie Mitchell's' book):

Genetic Algorithms

In a genetic algorithm (GA), the desired output is a solution to some problem….

The input to the GA has two parts: a population of candidate programs, and a fitness function that takes a candidate program and assigns to it a fitness value that measures how well that program works on the desired task.

Here is the recipe for the GA.

Repeat the following steps for some number of generations:

1. Generate an initial population of candidate solutions. The simplest way to create the initial population is just to generate a bunch of random programs (strings), called "individuals."

2. Calculate the fitness of each individual in the current population.

3. Select some number of the individuals with highest fitness to be the parents of the next generation.

4. Pair up the selected parents. Each pair produces offspring by recombining parts of the parents, with some chance of random mutations, and the offspring enter the new population. The selected parents continue creating offspring until the new population is full (i.e., has the same number of individuals as the initial population). The new population now becomes the current population.

5. Go to step 2.

Robby the Robot

I have a robot named "Robby" who lives in a (computer simulated, but messy) two-dimensional world that is strewn with empty soda cans. I am going to use a genetic algorithm to evolve a "brain" (that is, a control strategy) for Robby.

Robby's job is to clean up his world by collecting the empty soda cans. Robby's world consists of 100 squares (sites) laid out in a 10 × 10 grid. Let's imagine that there is a wall around the boundary of the entire grid. Various sites have been littered with soda cans (but with no more than one can per site). Robby isn't very intelligent, ***he has no memory of his past moves****, and his eyesight isn't that great. From wherever he is, he can see the contents of one adjacent site in the north, south, east, and west directions, as well as the contents of the site he occupies. A site can be empty, contain a can, or be a wall.*

For each cleaning session, Robby can perform exactly 200 actions. Each action consists of one of the following seven choices: move to the north, move to the south, move to the east, move to the west, choose a random direction to move in, stay put, or bend down to pick up a can. Each action may generate a reward or a punishment. If Robby is in the same site as a can and picks it up, he gets a reward of ten points. However, if he bends down to pick up a can in a site where there is no can, he is fined one point. If he crashes into a wall, he is fined five points and bounces back into the current site.

Clearly, Robby's reward is maximized when he picks up as many cans as possible, without crashing into any walls or bending down to pick up a can if no can is there.

The first step is to figure out exactly what we are evolving; that is, what exactly constitutes a strategy? In general, a strategy is a set of rules that gives, for any situation, the action you should take in that situation. For Robby, a "situation" is simply what he can see: the contents of his current site plus the contents of the north, south, east, and west sites. For the question "what to do in each situation," Robby has seven possible things he can do: move north, south, east, or west; move in a random direction; stay put; or pick up a can.

Therefore, a strategy for Robby can be written simply as a list of all the possible situations he could encounter, and for each possible situation, which of the seven possible actions he should perform.

How many possible situations are there? Robby looks at five different sites (current, north, south, east, west), and each of those sites can be labeled as empty, contains can, or wall. This means that there are 243 different possible situations (see the notes for an explanation of how

I calculated this). Actually, there aren't really that many, since Robby will never face a situation in which his current site is a wall, or one in which north, south, east, and west are all walls. There are other "impossible" situations as well. Again, being lazy, we don't want to figure out what all the impossible situations are, so we'll just list all 243 situations, and know that some of them will never be encountered.

To decide what to do next, Robby simply looks up this situation in his strategy table, and finds that the corresponding action is MoveWest, for example. So he moves west.

There are two parts to this assignment: The GA application and related test cases, and the gui layer. The *GA application* is where the genetic algorithm runs for 1000 generations. Certain classes here need to be thoroughly unit tested. The *gui* layer, which uses the Monogame framework, reads in the generations top specimens from a file and is simply for visualization and is relatively straightforward to code.

Solution's structure

You should in the end have (at least) three visual studio projects, all in a single solution:

- RobbyGeneticAlgo project: This should be created as a "console application" so that you can run it over 1000 generations.
- RobbyGeneticAlgoUnitTests: In here, you should pull all the classes for unit tests. This project should refer to the RobbyGeneticAlgo project.
- Robby Monogame project : In here, you will put all of the logic for the GUI part of the code. This is the only project that should have a reference to the Monogame framework. You should add here a reference to the RobbyGeneticAlgo project as well. Make sure this refers to .NET framework 4.6.1 so that it is compatible with the Console Application.
-

Team Coding Standards

This project must be done in teams of two. Like in previous assignments, you should create a team branch "staging" and each individual person should have their own feature branches to which they will contribute:

- Each person should freely make commits and pushes within their feature branch (nothing changes here), but make sure that you pull in new changes from staging (or rebase)
- Whenever you wish to *merge* your code in to staging, you will create a merge request (as before)
- The merge request needs to be *code reviewed*. This means that *someone other than the author* should review the merge request and give constructive comments as to whether the request is good or not.
    - In addition to providing text comments, they should give either a "thumbs up" or a "thumbs down" (in GitLab) to indicate whether they approve of the merge or not
    - You should only give a thumbs up if the code is good to go (other than extremely minor changes, such as renaming a variable)
- Once the merge request has received a thumbs up, the *original author* of the change should merge the code into the staging branch

- Note that in the case that there are other changes made to the staging branch in the interim, they may need to make a new merge request.

Robby Genetic Algorithm Classes, Structs, Enums and Delegates

For the overall list of types you need to use, please see the UML diagram at the end of this document.

The UML diagram is not complete, in that some fields and methods are missing. It is meant to give you an idea of how you should code this, but it isn't meant to be exhaustive. You will have to figure some stuff out on your own and make some adjustments as needed.

You may also choose to add methods, etc, but you need to be careful about not violating encapsulation. The important question to keep in mind in determining if a change is breaking is to think about *what class is responsible for what things* in the program. You are encouraged to go over your additional ideas with other classmates and your teacher to see if they are good ideas or not.

You must unit test the Chromosome and Generation classes. It is not necessary to verify via unit tests simple get/set properties.

Some additional notes/hints on the UML diagram. There's a lot here, so we will go over this in class. 😊

The Chromosome class and Allele enum

This class represents a single chromosome, or a single solution to a GA problem. It contains an array of Alleles, which are basically the genes. The Alleles can have values North, South, East, West, Nothing, PickUp, or Random: they are basically the moves that Robby can do upon any turn.

- the first Chromosome constructor instantiates the array of Alleles to contain length genes, and sets them all to random numbers. Use the Random class that is in the Helpers class. Remember that enums are ints behind the scenes, ranging from 0 to the number of enums. You can find the number of enums with this code snippet, where you replace Name_of_enum with the specific enum: `Enum.GetNames(typeof(Name_of_Enum)).Length`
  and you can convert an int to an enum by casting.
- the second constructor makes a deep copy of the array of Alleles
- The Reproduce method takes the second Chromosome as well as a Crossover delegate object and mutation rate. It uses to Crossover function to create two offspring, then iterates through the two children Chromosomes' Alleles, changing them to random alleles according to the mutation rate
- The EvalFitness method takes a Fitness delegate object, invokes it on this, and sets the result as the Fitness property
- a readonly indexer is used to get a specific Allele
- Chromosome implements the IComparable<Chromosome> interface, which means a CompareTo method that returns an int s required. Since doubles are IComparable, simply return the result of comparing your fitness with the other's Fitness (in other words, Chromosomes are compared based on their Fitness).
- Override the ToString method to return the contents of the Alleles array, joined with a comma ","

- Finally, the Chromosome class contains two methods that meet the delegate Crossover's signature. The SingleCrossover method find a random crossover point and returns the two offspring. The DoubleCrossover finds the first crossover point randomly in the first half of the chromosomes, and the second crossover point in the second half and returns the two resulting offspring.

You are required to unit test this class. Hint: you may want to use a seed when you instantiate the Random object in the Helpers class (and remove the seed when testing is complete). When you used the same seed in a pseudo-random generator, you are sure that the sequence of numbers returned is the same, which enables you to predict the correct expected behavior.

The Generation Class

The Generation class represents a generation with a population size of Chromosome members. It contains an array of Chromosomes.

- the first constructor initializes a population of random Chromosomes.
- the second constructor makes a deep copy of an array of Chromosomes
- The EvalFitness method takes a Fitness delegate object, invokes it on all the Chromosome members, and then sorts the array of members (you can use Array.Sort since Chromosomes implement IComparable) and the reverses the array (you can use Array.Reverse) in order to get the highest fitness at the front.
- a readonly indexer is used to get a specific Chromosome
- Finally, the SelectParent method is used to select a parent for the next generation. It will be invoked only after EvalFitness is invoked so you can assume that the array of Chromosomes is sorted in descending order of Fitness (meaning the more desirable parents have a smaller index). So simply get 10 random indexes return the Chromosome at the smallest random index.

You are required to unit test this class.

The RobbyRobotProblem Class

This class is the one that gets all the other classes interacting correctly. You will notice that the Main method provided in the Helpers class starts the Console Application by instantiating a RobbyRobotProblem, subscribing to events, and starting the problem.

- The constructor initializes many instance variables (see the UML diagram). It instantiates an array of Contents[,] which will hold all the randomly generated test grids
- Parameters to use with your RobbyRobotProblem constructor:

  numActions = 200
  numTestGrids = 100
  gridSize = 10
  numGenes = 243
  eliteRate = 0.05
  mutationRate = 0.05

- the Start method in invoked to start the GA process. It instantiates the first current generation (needs to be an instance variable) to random values then loops through the generations:

    - evaluating the fitness of the current generation
    - invoking the event
    - creating the next generation
- EvalFitness generates a set of test grids using the helper method in the Helpers class and evaluates the fitness of the current generation
- GenerateNextGeneration creates the next generation by retaining the elite and filling the remaining positions in the population with offspring, based on the Crossover strategy it is given. Make sure that you take an even number of elite, since there are always pairs of offspring and we assume population size is even. To procreate, select 2 parents and have them reproduce, adding the offspring to the next generation's population. Once the population of Chromosomes is complete, instantiate a new Generation and set the current generation to refer to it.
- RobotFitness calculates and returns the average fitness of a given Chromosome by going through all test grids, invoking Helpers RunRobbyInGrid, and returning the average Fitness

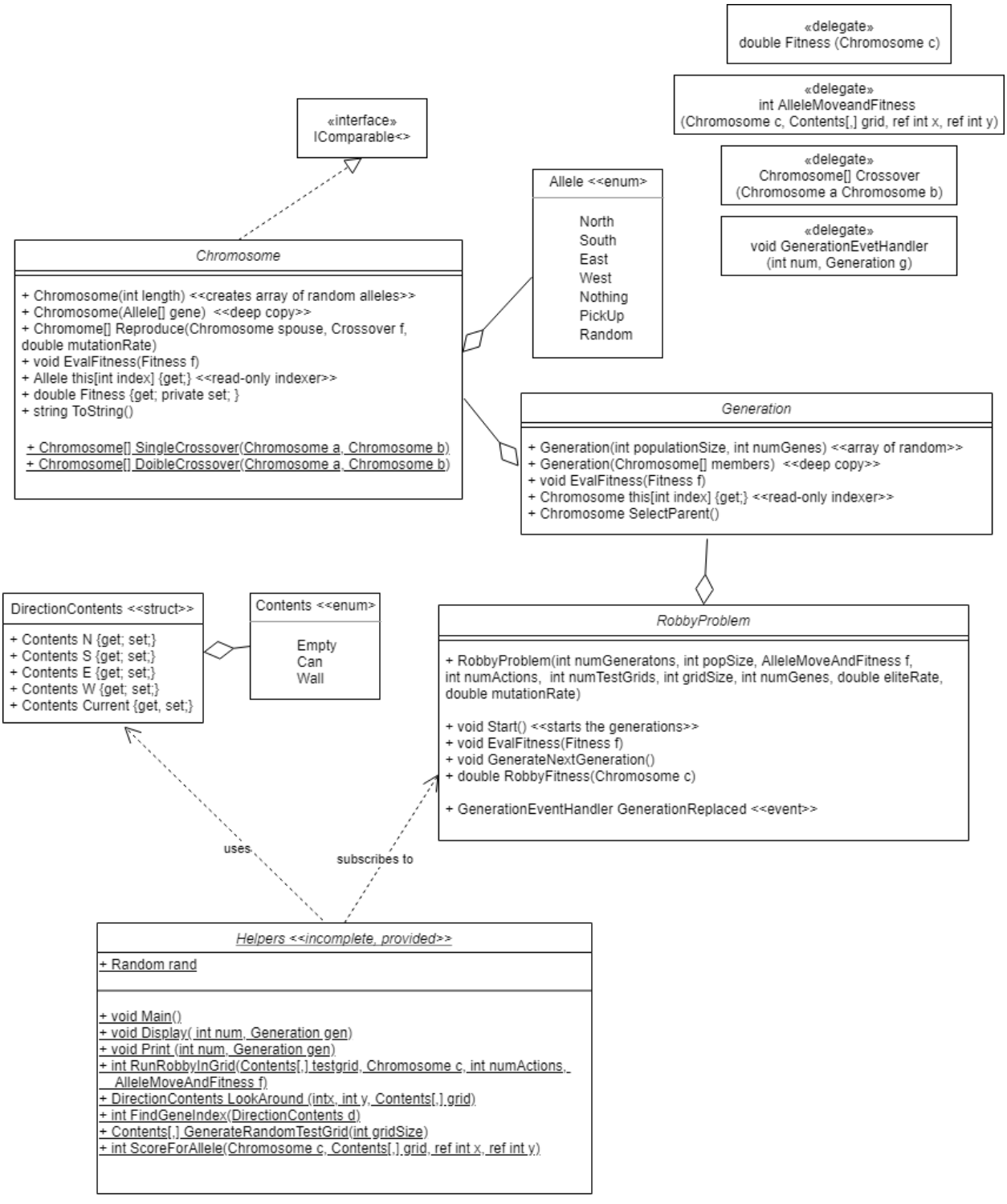
The Helpers class, DirectionContents struct, Contents enum

Finally, the Helpers static class contains static members and methods. You have been provided an incomplete class: read through this code carefully and fill in the TODO sections

- the rand object should be used by all classes which require Random numbers
- the Main method starts the process. Fill in the lines to subscribe to the RobbyRobotProblem's GenerationReplaced event with the Display and the Print methods
- The Display method should print to the Console the generation number, and the fitness of the top (i.e., first Chromosome).
- The Print method (provided) prints the info of the 1st, 20th, 100, 200, 500 and 1000th generation to a file. This is used by the Monogame project. Note that the System.IO.File class has a WriteAllText method, you will write the best chromosome of each generation in a separate file
- The RunRobbyInGrid method (provided) runs Robby in a given testgrid, keeping track of its fitness score.
- The LookAround method (provided) is used to fill a DirectionContents struct based on what Robby sees around him
- the FindGeneIndex and getIndexForDirection methods (provided) are used to get the index (0 to 242) of the gene in the Chromosome array
- you must code the GenerateRandomTestGrid method, which generates and returns a grid. Use a counter to place Contents.Can in exactly 50% of the positions.
- finally, ScoreForAllele (provided) make a move and calculates and returns the fitness score for that move based on the grid, Robby's position and the Chromosome

Notes:

When you run the Console application, be patient. Your best Robby should be able to achieve 100 points by 500 generations. If this isn't the case, double-check your code and see me.


«delegate»
double Fitness (Chromosome c)
«delegate»
int AlleleMoveandFitness
(Chromosome c, Contents[,] grid, ref int x, ref int y)
«delegate»
Chromosome[] Crossover
(Chromosome a Chromosome b)
«delegate»
void GenerationEvetHandler
(int num, Generation g)
«interface»
IComparable<>
Allele <<enum>
North
South
East
West
Nothing
PickUp
Random
Chromosome
+ Chromosome(int length) <<creates array of random alleles>>
+ Chromosome(Allele[] gene) <<deep copy>>
+ Chromome[] Reproduce(Chromosome spouse, Crossover f, double mutationRate)
+ void EvalFitness(Fitness f)
+ Allele this[int index] {get;} <<read-only indexer>>
+ double Fitness {get; private set; }
+ string ToString()
+ Chromosome[] SingleCrossover(Chromosome a, Chromosome b)
+ Chromosome[] DoibleCrossover(Chromosome a, Chromosome b)
Generation
+ Generation(int populationSize, int numGenes) <<array of random>>
+ Generation(Chromosome[] members) <<deep copy>>
+ void EvalFitness(Fitness f)
+ Chromosome this[int index] {get;} <<read-only indexer>>
+ Chromosome SelectParent()
DirectionContents <<struct>>
+ Contents N {get; set;}
+ Contents S {get; set;}
+ Contents E {get; set;}
+ Contents W {get; set;}
+ Contents Current {get, set;}
Contents <<enum>
Empty
Can
Wall
RobbyProblem
+ RobbyProblem(int numGeneratons, int popSize, AlleleMoveAndFitness f, int numActions, int numTestGrids, int gridSize, int numGenes, double eliteRate, double mutationRate)
+ void Start() <<starts the generations>>
+ void EvalFitness(Fitness f)
+ void GenerateNextGeneration()
+ double RobbyFitness(Chromosome c)
+ GenerationEventHandler GenerationReplaced <<event>>
uses
subscribes to
Helpers <<incomplete, provided>>
+ Random rand
+ void Main()
+ void Display(int num, Generation gen)
+ void Print (int num, Generation gen)
+ int RunRobbyInGrid(Contents[,] testgrid, Chromosome c, int numActions, AlleleMoveAndFitness f)
+ DirectionContents LookAround (intx, int y, Contents[,] grid)
+ int FindGeneIndex(DirectionContents d)
+ Contents[,] GenerateRandomTestGrid(int gridSize)
+ int ScoreForAllele(Chromosome c, Contents[,] grid, ref int x, ref int y)

Graphical Simulation:

Once the generations have been created and written to file, the Monogame project will display Robby's strategy for these generations so you can visualize the improvement. Monogame will take care of the game loop, updates and drawing.

Add a MonoGame Windows Project to your solution. You will need a reference to your RobbyGeneticAlgo project since you will use Helpers ScoreForAllele method. Within the game project, you will add a SimulationSprite class: this class will read the files in its Initialize method. Make sure you add this file properly in visual studio by setting it to "copy always" (see https://stackoverflow.com/questions/4596508/vs2010-how-to-include-files-in-project-to-copy-them-to-build-output-directory-a). It will load image content related to the empty tiles, soda can, and Robby.

In each update (after a counter throttles), it will perform the required move, and update the score. Use the Helpers.ScoreForAllele method to calculate the score on each Update.

The Draw method draws the images and writes the fitness and moves. All pngs should be 32 x 32, and these dimensions can be used to calculate where to draw each png. For example, to draw a can at position [x,y], the Draw method will look something like:

```
spriteBatch.Draw(imageCan, new Rectangle(x * 32, y * 32, 32, 32),
Color.White);
```

(nb: the last argument indicates the colour to tint. Color.White means no tinting).

The Game1 class instantiates the sprite and adds it to its components list.