Mohsen Amjadi - 810896043

project explanation:

Logic gates are the basic building blocks of logic circuits. In this project we’re going to use a genetic algorithm to combine logic gates into circuits that can do work.

Genetic algorithms are one of the tools we can use to apply machine learning to finding good, sometimes even optimal, solutions to problems that have billions of potential solutions. They use biological processes in software to find answers to problems that have really large search spaces by continuously generating candidate solutions, evaluating how well the solutions fit the desired outcome, and refining the best solutions.

So we define our logic gates including NOT , OR , AND , XOR in the circuits module and we combine genetic algorithm related classes and functions in the genetic module.

Also we have a main file which we define our tests and custom functions to feed into genetic algorithm and get the result.

When we want to use genetic algorithm to solve a problem we should model our problem with genetic concepts. We should define genes , chromosomes , selection , population , mutation and crossover.

We’ll use a tree node structure for the genotype, that contains the type of gate and indexes

to 2 child tree nodes (potential inputs) that you see how Node class has been defined. and chromosomes are logic circuits that are potential solutions for the problem. our phenotype should have behaviors, so we’ll use objects.

In addition to logic gates the circuit will also contain references to the actual A and B source inputs we’re testing. We need to be able to change the source values to check the fitness of the circuit, so we’ll give it a reference to a container whose contents we can modify externally. So we have our Source class for this purpose.

Now we’ll write the function that builds the circuit. Which is called (nodes_to_circuit).

It loops through all the nodes starting with the leaf nodes and rolling toward the root while connecting the logic gates together.

we can prevent recursion by design with the convention that child indexes are only valid if they are lower than the node index.

Lastly we update the circuit by creating the gate. The circuit we’ll use ends up fully connected and in the last index of the array. There may be latent circuits in the array as well when we’re done.

Answer to question 1)

To calculate the fitness we need to build the circuit from the nodes then use each rule's inputs to test the circuit and count how many rules the circuit can satisfy.

We have a csv file which is called truth_table.csv and contains our rules. It has 10 inputs and the last column is the output. In our fitness function we we can use ABCDEFGHIJ as source labels of the 10 inputs and every time circuit's output become equal with the last column of inputs we can say a rule has been passed and increment it by one.

We use unit test framework to test if the program runs correctly so we have CircuitTest class to execute the whole functionality by defining functions starting name with test , and populating the gene set with setUpClass .

Answer to question 3)

We will use fnCreateGene to call a function which create gene for us. It will pick child index values relative to the index where the node will be inserted so they're more likely to be valid when converted to a circuit. We'll also try to make the input indexes different so we reduce the waste from gates like And(A A).

Next we'll add a custom mutate function. So we have fnMutate to call our custom mutate function.To be efficient in the mutate function, we only want to change the nodes that we actually use in the circuit. We can accumulate those while we're building the circuit. The change in nodes_to_circuit is to add a tracking array where each element contains the set of node indexes that are used to build the corresponding circuit. now we can call nodes_to_circuit to get the list of node indexes to use as mutation candidates.

We also need a custom create function. It is simple enough to be added inline so in fnCreate function we call create gene for 100 times that is our maximum length of genes or nodes.

Answer to question 2)

Next we need to solve the problem of figuring out what the optimal solution is. Hill climbing is a popular problem space exploration technique where one feature of the Chromosome is

incrementally adjusted until a better solution is found or a local minimum or maximum is detected, at which point the process repeats with a different feature. Some variants change any feature as long as it only affects a single piece of data, which may be smaller than a gene. An example from our current project would be to only change the gate type or just one of the indexes in a node rather than replacing the entire node as we're doing now. hill climbing doesn't always find the optimal solution so we will introduce another techniqe later.

We'll implement hill climbing in the genetic module so that it is reusable. Starting with the function definition, it includes the optimization function it will call, a function to test whether an improvement has been found, a function to test whether the improvement is optimal so we can stop, a function that gets the next feature value, a display function, and the initial value of the feature we're trying to optimize.

Once we have a result we're going to enter a loop where we keep getting new results until we find an optimal one. When we find an improvement it becomes the new best value and we display it. If we find the optimal solution we return it.

Now we need to build the inputs we're going to pass to the hill_climbing function. We'll start by wrapping the current call to get_best in a new function. This will be the optimization function. We'll give it a maximum of 50 gates to work with and see what it finds.

I gave it 30 seconds to try to find an improvement, but it can be a lower value. The feature we're optimizing is the number of nodes in the circuit. We need to get that to the create_gene function. We'll do that by transferring it to the existing maxLength variable that is already passed to that function.

Next we need a function that can tell whether the new result is better than the current best. We have fnIsImprovement for that .we return True if all the rules pass and the number of gates used in the new result if less than that of the current best result, otherwise False.

We also need a function that can tell whether we've found the known optimal solution (expectedLength) and fnIsOptimal will do so.

As for display, we can simply add an optional parameter to the existing fnDisplay for the feature value. When it is set we show the number of nodes used in the new best circuit.

When an improvement is found we'll make the number of nodes in that circuit the new value to beat. (fnGetNextFeatureValue)

At the end we call the hill climbing function. Now we're finally able to finish the test implementation And call find circuit function to execute the test.

Answer to question 4)

after we execute the test a local maximum occurs so we use a techniqe called simulated annealing to try to find global maximum.

To fix the local minimum/local maximum issue we’re going to allow the current genetic line to die out. The first step in that is tracking how many generations have passed since the last improvement. We’ll call this its Age and define it in the chromosome class.

Next, we’ll add an optional parameter that allows us to set an upper limit on the age of a genetic line. That parameter will be passed through to _get_improvement.

Then we need to separate the best parent from the current parent so that we still have something to compare to when a genetic line dies out. We’re also going to keep a list of the fitnesses of the historical best parents.

Next, we want to make sure we retain the current functionality if maxAge is not provided.

However, when the child’s fitness is worse than that of its parent, the most traveled path through the code, and maxAge is provided, then we need to check whether or not the genetic line’s age has reached the maximum.

If so, we may allow the genetic line to die and replace it with something else. We’re going to do that with simulated annealing.

Annealing is a method used to reduce internal stresses in materials like metal. At a high level, it works by heating the metal to a high temperature then allowing it to cool slowly. As you know, heat causes metal to expand. So the metal expands, loosening the bonds between materials in the metal, allowing them to move around. Then the metal is allowed to cool slowly. As the metal cools the bonds tighten again and the impurities get pushed along by the increasing pressure until they find something to bond with or until there is no more wiggle-room, thus reducing overall stress in the system.

The standard global search algorithm only allows a parent to be replaced by a child with equal or better fitness.

This makes it very difficult to break out of a local minimum or maximum because the only way out is through chance discovery of an equal or better gene sequence that is different enough from the current gene sequence to escape the local minimum/maximum, an unlikely event.

Simulated annealing resolves this by slowly increasing the probability of replacing the parent with a different gene sequence, even if that gene sequence has a worse fitness, based in part on how long we’ve been using that parent to try to make the next generation. Thus, the implementation in this project uses the term age to control the simulated annealing process.