Evolving intelligence: genetic programming

Lecture 07.02

Genetic programming

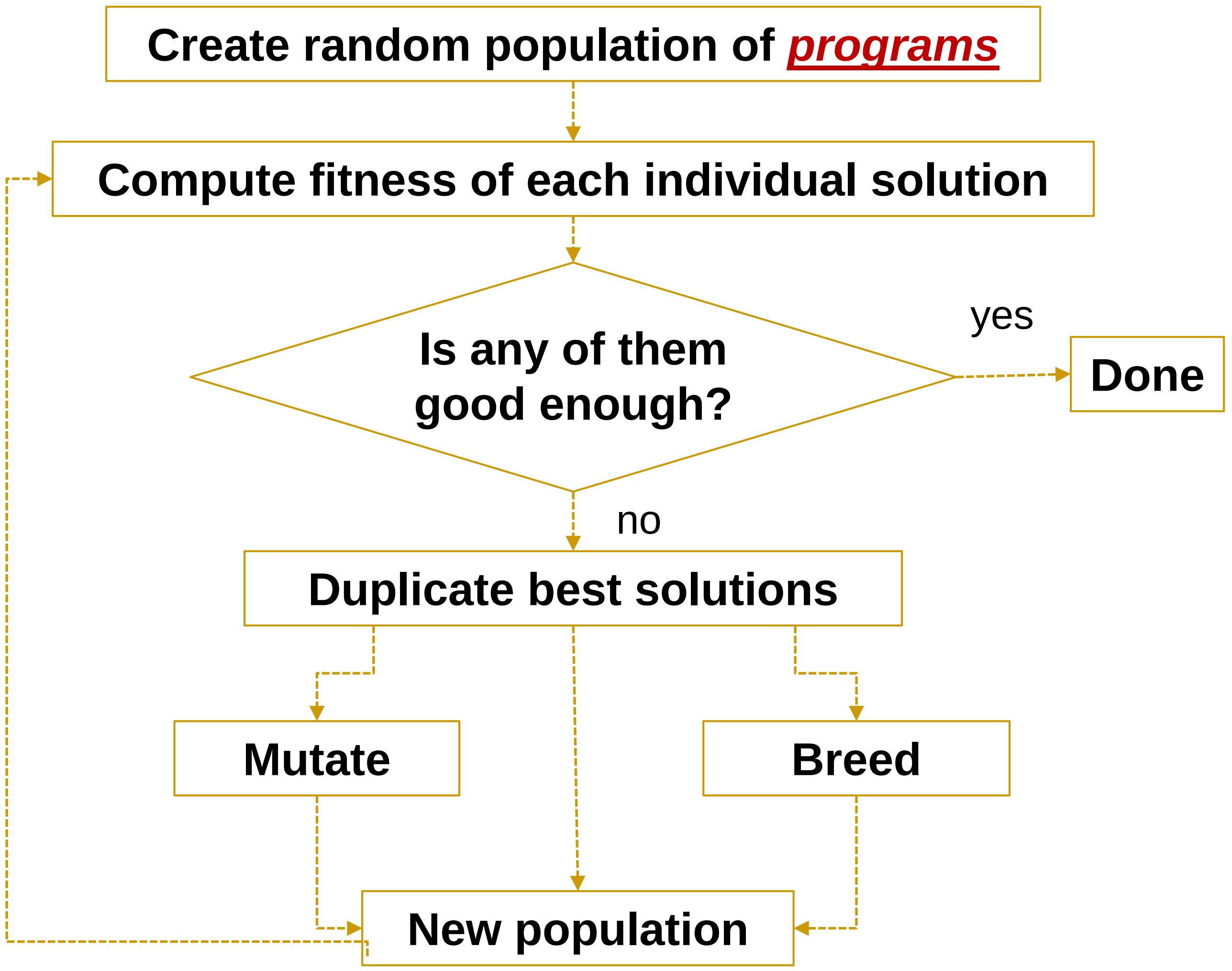
- Application of Genetic Algorithm to the case where solution space consists of computer programs
- The goal is to find a program which performs well at a predefined task
  - Instead of choosing an algorithm that is the best for a predefined task, we make a program that will create such algorithm: we design an algorithm which creates algorithms
- In some cases the algorithm finds programs that are human-competitive

"Human-competitive results produced by genetic programming"

How does it work

- We start with a large set of programs (*population*), which are either [randomly generated] or [hand-designed to be somewhat good solutions]
- The programs then compete in performing some user-defined task:
  - A game in which the programs compete against each other and the performance is measured by the number of wins
  - A known set of inputs and outputs and the best program (function) perfectly maps inputs to outputs

Genetic programming flowchart



Create random population of *programs*
Compute fitness of each individual solution
Is any of them good enough?
yes
Done
no
Duplicate best solutions
Mutate
Breed
New population

Same steps as in GA

- After evaluating each program using the *fitness test*, we produce *a ranked list of programs*
- The best programs are **replicated** and **modified** in two different ways
  - *Mutation*: certain parts of a program are altered slightly in a random manner in hope that this will make a good solution even better
  - *Crossover* (*breeding*): exchange the portions of best programs
- This replication and modification procedure creates many new programs which are evaluated until the best solution is found

Programs get better with each new generation

- Since the size of the population is kept constant, many of the *worst* programs are eliminated from the population to make room for new programs
- The new population is referred to as “the next generation”
- Because the best programs are being kept and only slightly modified, it is expected that with each generation they will get better and better

When to stop evolving

- New generations are created until a termination condition is reached:
  - The perfect solution is found
  - A good enough solution is found
  - The solution did not improve for several generations
  - The number of generations reached a specified limit

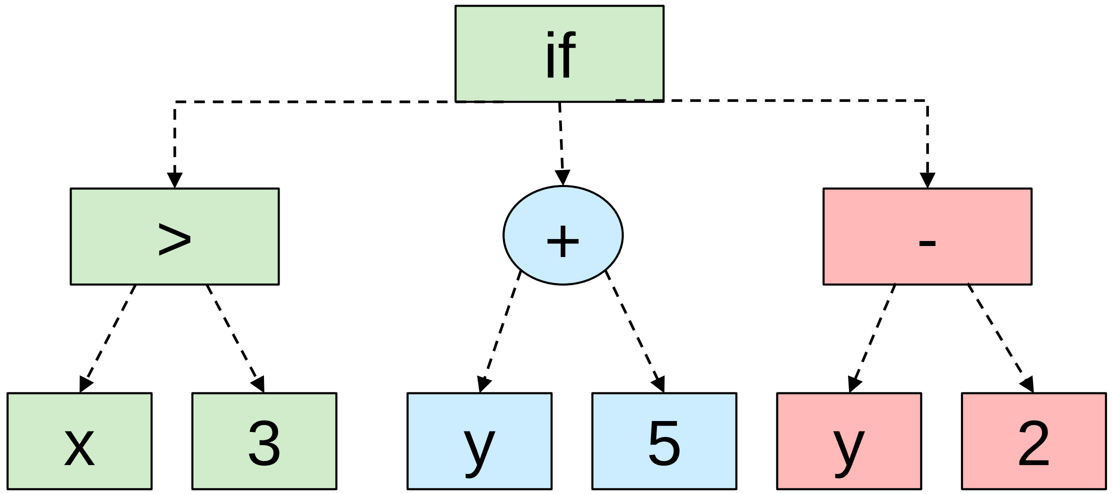
Genetic Programming (GP) vs. Genetic Algorithm (GA)

- GA is an optimization technique
  - As with any optimization, you have already selected an algorithm or metric and you’re trying to find the best parameters for it
- In GP the solutions are not just a best set of parameters applied to a given algorithm:
  - The algorithm itself and all its parameters are designed automatically by means of evolutionary pressure

Representing a solution

- We need to create an input for GP
- The input is a population of programs
- How do we represent programs?
  - The most commonly used is a ***tree representation***
  - Representing programs as trees is natural, because programs in most programming languages, when compiled or interpreted, are first turned into a parse tree

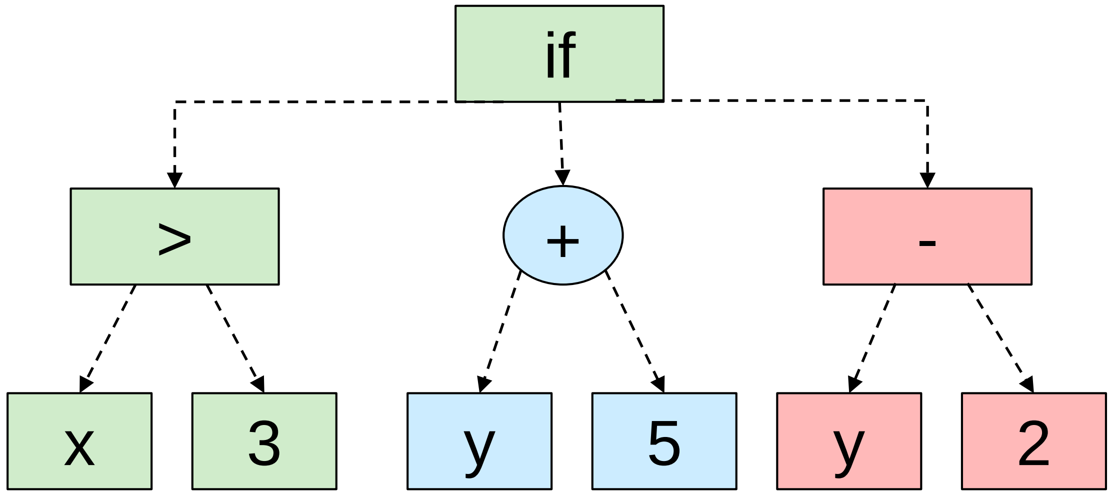
Sample program tree



- Each non-leaf node represents an operator applied to its child nodes
- Each leaf node represents an operand (variable parameter or a constant value)
- Once a leaf node is evaluated, it is given to the node above it, which applies its operator to its branches
- The “if” operator has 3 child branches: if leftmost branch evaluates to true, return the center branch; if it doesn’t, return the rightmost branch.

What is the program?

Sample program tree



```
def func(x,y)
   if x>3:
      return y + 5
   else:
      return y - 2
```

Initializing program population

- It’s possible to hand-create initial population, but most of the time the initial population is a set of random programs
- This makes the process easier to start, since it’s not necessary to design several programs that almost solve a problem
- It also creates much more *diversity*—if initial programs are designed by a single programmer they are likely to be very similar, and although they may give answers that are almost correct, the ideal solution may look quite different

Define basic function types

```
class fwrapper:
    def __init__(self,function,childcount,name):
        self.function=function
        self.childcount=childcount
        self.name=name
```

```
addw=fwrapper(lambda l:l[0]+l[1],2,'add')

subw=fwrapper(lambda l:l[0]-l[1],2,'subtract')

mulw=fwrapper(lambda l:l[0]*l[1],2,'multiply')

ifw=fwrapper(lambda l:  l[1] if l[0]>0 else l[2],3,'if')

gtw=fwrapper(lambda l: 1 if l[0]>l[1] else 0,2,'isgreater')
```

Also define Node classes to connect functions into a tree

```
class node:
    def __init__(self,fw,children):
        self.function=fw.function
        self.name=fw.name
        self.children=children

    def evaluate(self,inp):
        results=[n.evaluate(inp) for n in self.children]
        return self.function(results)

    def display(self,indent=0):
        print ((' '*indent)+self.name)
        for c in self.children:
            c.display(indent+4)
```

```
class paramnode:
        …
```

```
class constnode:
        …
```

Making a random program tree

- Creating a random program consists of creating a root node with a random associated operator and then creating as many random child nodes as necessary, which in turn may have their own associated random child nodes
- Like most functions that work with trees, this is easily defined recursively

See ***make_random_tree()*** in *gp.py*

The way to generate solutions

Try this:

The tree from the example

```
>>> from gp import *

>>> t = exampletree()
>>> t.display()
if
    isgreater
        p0
        3
    add
        p1
        5
    subtract
        p1
        2
>>>
```

Random tree

```
>>> rtree = make_random_tree(2)
>>> rtree.display()
multiply
    add
        0
        10
    add
        add
            add
                p0
                p1
            if
                p0
                p0
                6
        3
```

Evaluating solutions

- We can now build programs automatically
- Generating random programs until one is correct would be ridiculously impractical because there are infinite possible programs
- How do we test a solution to see if it's correct, and if it's not, how do we determine how close it is?

Dataset generated by an unknown program

x	y	output
4	2	10
2	3	7
5	1	11

- Can you guess which function was used to generate the data?
- Guessing the function which describes relationship between attributes and the numeric value is the task of *regression*
- We will use GP to find the best function

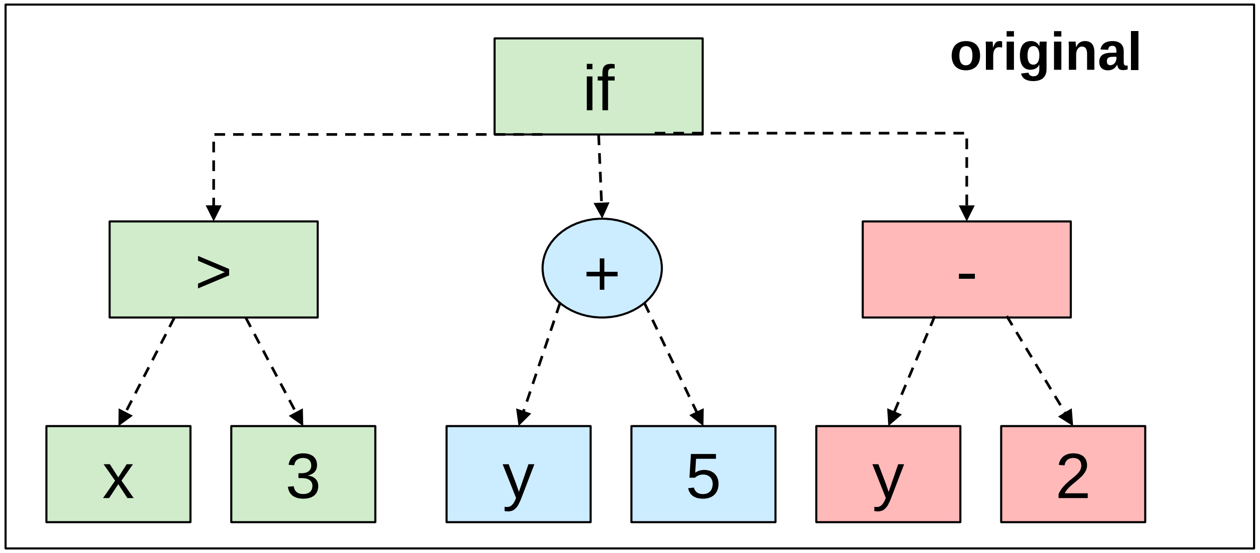
Fitness function

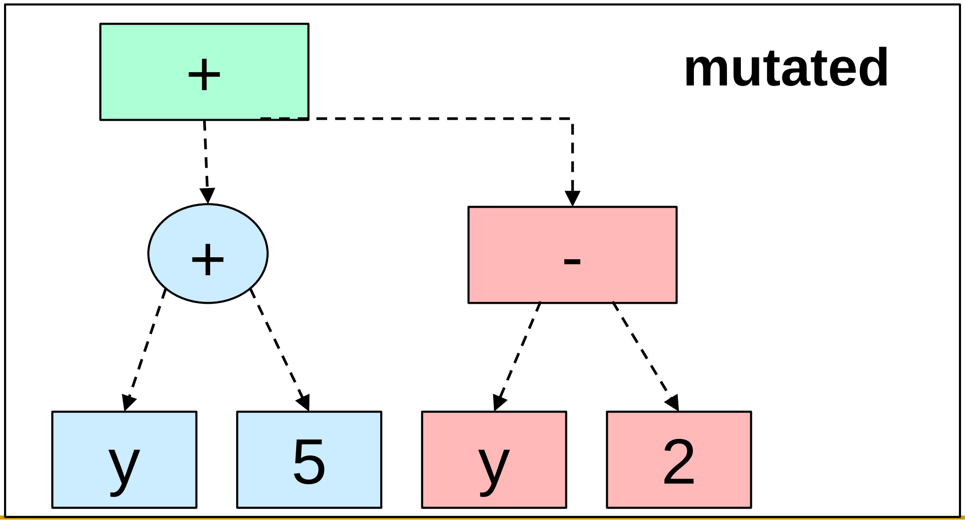
- Our fitness function will check every row in the dataset, calculating the output from a given candidate function and comparing it to the real result
- It will add up all the differences, giving lower values for better programs—a return value of 0 indicates that the function got every result correct
- Since we only generated a few random programs, the chance that one of them is actually the correct function is vanishingly small
- But now we have a way to evaluate how close we are to predicting a mathematical function, which is important for deciding which programs make it to the next generation

Mutating programs

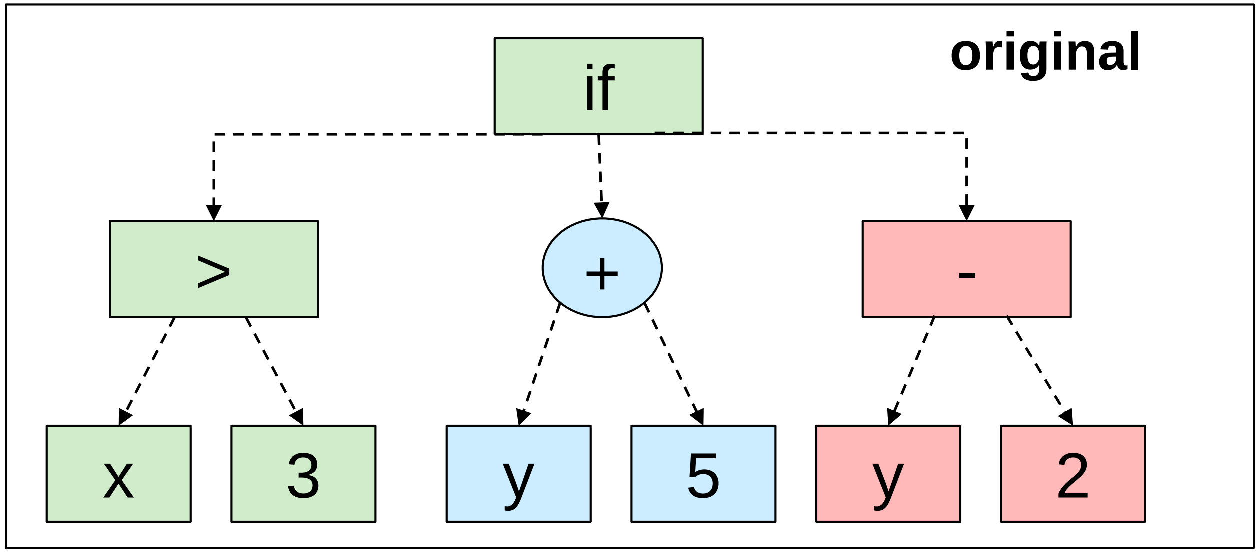
- Mutation takes a single program and alters it slightly
- The programs can be altered in different ways:
  - Changing the operator on a node
  - Replacing a subtree with a completely new subtree

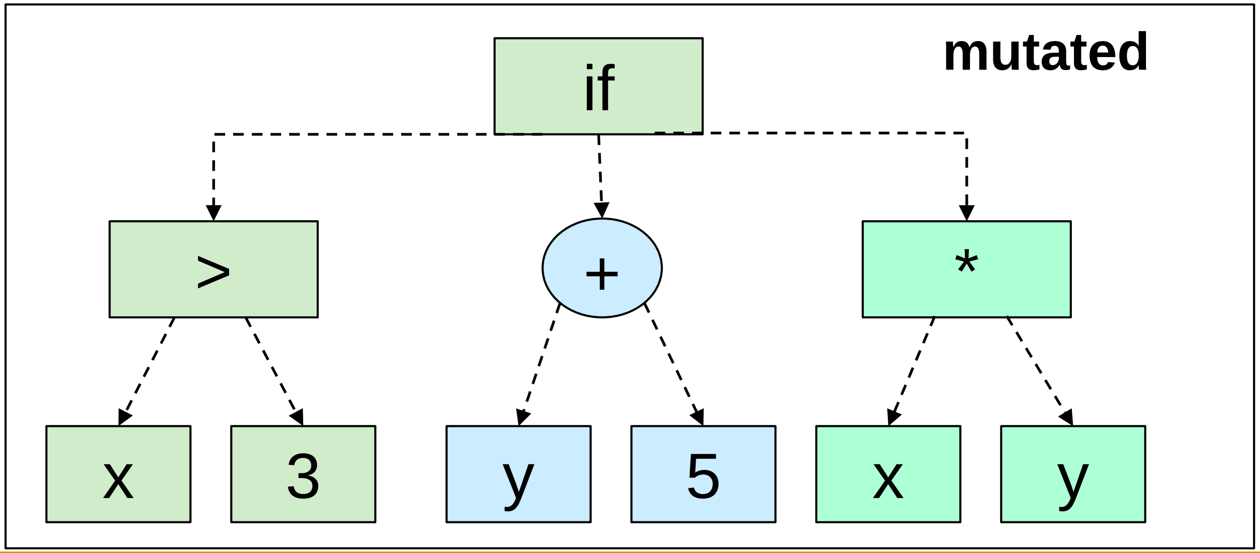
Mutation by changing node functions





Mutation by replacing a subtree





Implementing program mutations

- For simplicity, only the second type of mutations is implemented
- The *mutate()* function begins at the top of the tree and decides whether the node should be altered (according to mutation probability):
  - If yes, current node will be replaced with a random program
  - If not, it continues traversing the tree and calls *mutate* on each child
- It's possible that the entire tree will be mutated, and it's also possible to traverse the entire tree without changing it

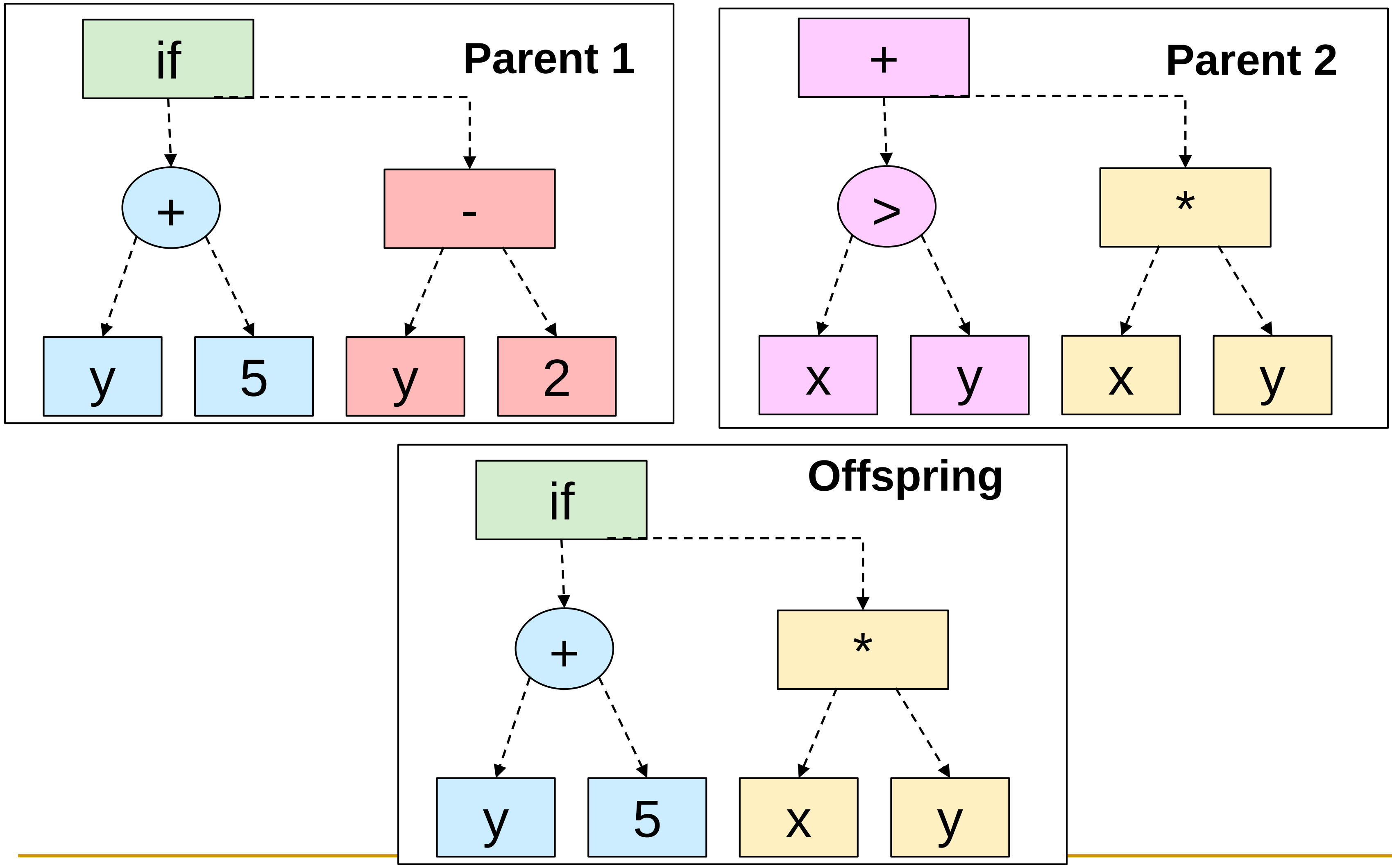
Mutations are random and not necessarily beneficial

- The mutations are random, and they aren’t directed toward improving the solution
- The hope is simply that some small changes will improve the result
- These changes will survive, and over several generations the best solution will eventually be found

Breeding programs

- Crossover (breeding) is implemented as combining two most successful programs to create a new program, usually by replacing a subtree from one with a subtree from another
- The *crossover()* function takes two trees as inputs and traverses both of them simultaneously
- If a threshold (crossover probability) is reached, the function returns a copy of the first tree with one of its branches replaced by a branch in the second tree
- By traversing both trees at once, the crossover happens at approximately the same level on each tree

Breeding programs



Finding the best program tree that describes the data

- Armed with a measure of success and two methods of modifying the best programs, we’re ready to set up a competitive environment in which programs can evolve

Testing GP

```
def hidden_function(x,y):
  return x**2+2*y+3*x+5
```

- We know what function was used to generate the dataset
- The real test is whether genetic programming can reproduce it without being told
- Run *gp.py* to see if the random population of programs can evolve into a target function that best describes the relationship between input and output in the dataset

Evolution in action

```
…
generation 19 , score = 400
generation 20 , score = 200
generation 21 , score = 200
generation 22 , score = 200
generation 23 , score = 200
generation 24 , score = 0
add
    multiply
        p0
        p0
    add
        add
            p1
            5
        add
            p1
            multiply
                p0
                3
```

```
def hidden_function(x,y):
   return x**2+2*y+3*x+5
```

- The result may look more complex than the target function, but it is the same function!

The danger of inbreeding

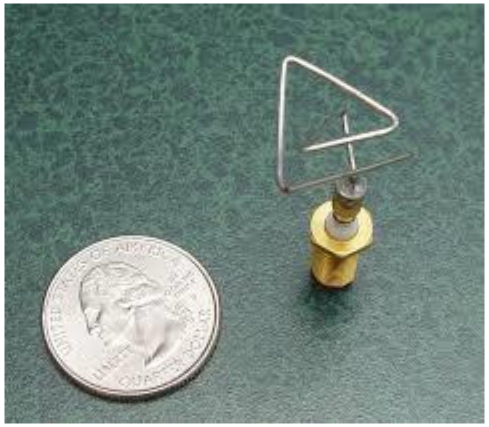
- The *evolve()* function ranks the programs in each generation from best to worst, so it might be tempting to just take 2-3 programs at the top and replicate and modify them for the new population
  - After all, why would you bother allowing anything less than the best to continue?
- The problem is that choosing only top solutions quickly makes the population extremely homogeneous (or inbred): the programs converge to the same set
- This again is a *local minima* problem: a state that is good but not quite good enough, and one in which small changes won’t improve the result

Adding diversity in each generation

- The evolve() function has 2 parameters which combat inbreeding:
- *pexp* - the probability of selecting lower-ranked programs
  - A higher value makes the selection process more stringent, choosing only programs with the best ranks to replicate
  - By lowering the value, you allow weaker solutions - turning the process from “survival of the fittest” to “survival of the fittest and luckiest”
- *pnew* - the probability that a completely new, random program is introduced

Successes of GP

- Designing antennas for NASA
- Developing programs for playing games, such as chess and backgammon
- Used in photonic crystals, optics, quantum computing systems, and other scientific inventions
- In 1998 a robot team that was programmed entirely using genetic programming which placed well in the Robo-Cup soccer contest



Automatic antenna design with evolutionary algorithms