Tight Genes:

Intro to Genetic Algorithms

by Dave Aronson

T.Rex-2025@Codosaur.us

twitter.com/DaveAronson

linkedin.com/in/DaveAronson

github.com/CodosaurusLLC/tight-genes

Speaker notes

NOTE TO SELF: click on timer to reset it at the start

Current time: ~37, speaking medium/fast, with lots of ad-libs.

DDEU wants 45, leave ~5-10 for Q&A, so net 35-40, seems about right, SLOW DOWN but use fewer ad-libs.

Sveiki, Vilniau!

(Hello, Vilnius!)

Speaker notes

SveikEE VILnio!

Aš esu Dave Aronson,

(I'm Dave Aronson,)

Image: me speaking at JSConf Hawai'i 2020!

Speaker notes

AHSH EHsoo Dave Aronson,

T. Reksas iš Codosaurus,

CODOSAURUS

(the T. Rex of Codosaurus,)

Image: my company logo!

Speaker notes

teeRANnoSOWrus RRREKsas ish Codosowrus,

ir atskridau čia

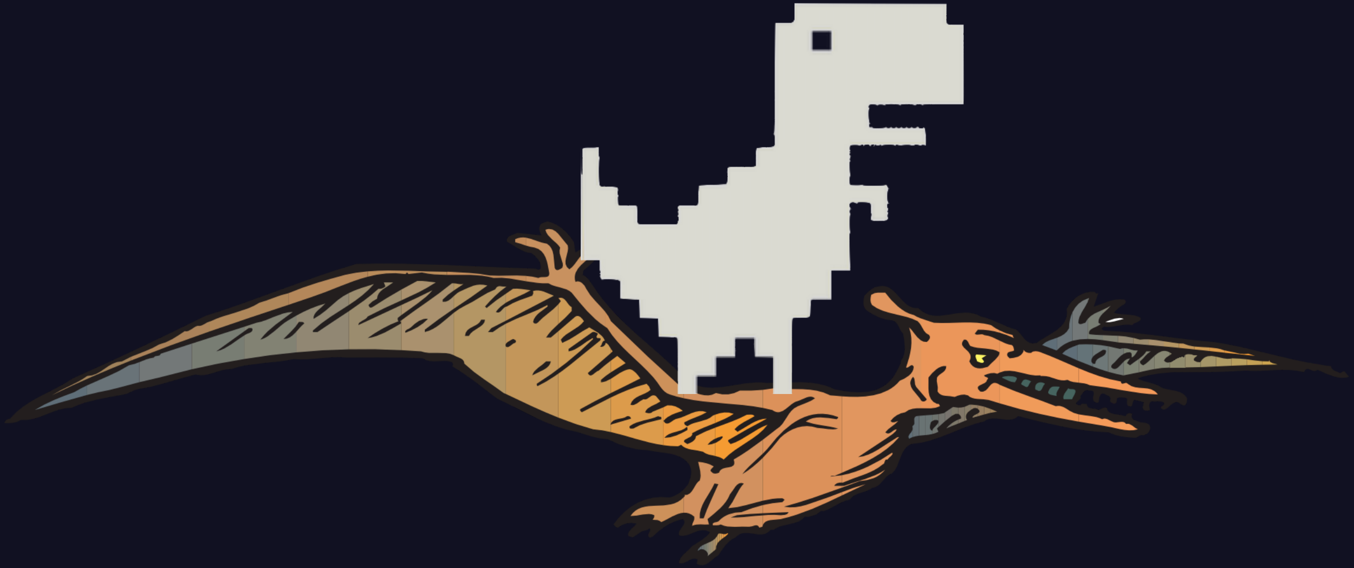
• • •

(and I flew here)

Speaker notes

ir atskrihDOH cheh

ant savo augintinio pterodaktilio



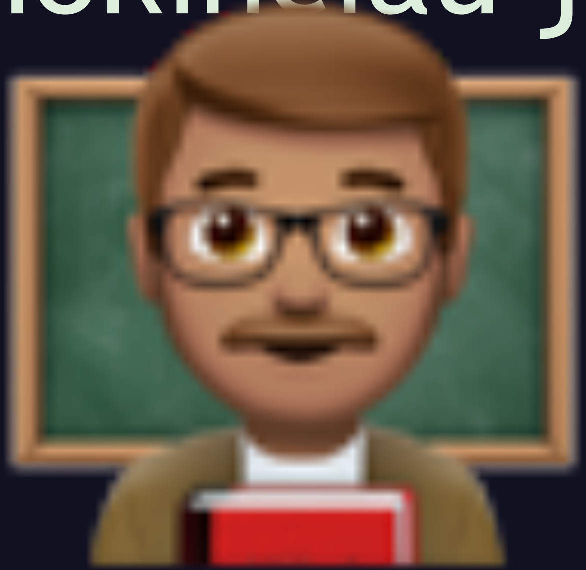
(on my pet pterodactyl)

Images: https://pixabay.com/vectors/dinosaur-tyrannosaurus-t-rex-6273164/
and https://pixabay.com/vectors/bird-flying-wings-dinosaur-ancient-44859/

Speaker notes

ahnt sahvo auGIHNtihnyo PterodakTIHlyo

kad išmokinčiau jus apie



(to teach you about)

Image: standard emoji

Speaker notes

kad ishmoKIHNchyo YOOS ah-PEE-eh

Genetiniai Algoritmai.

(Genetic Algorithms.)

Image: https://pixabay.com/vectors/genetic-testing-gene-panel-genetics-2316642

Speaker notes

geNETinee AlgoRITHmai.

Visgi . . .

(But . . .)

Speaker notes

VIZgih . . .

toliau tesiu pasakojima angliškai.

(I will do it in English.)

Image: standard emoji

Speaker notes

tohIYO TESSu PassaKoimah AHNglishkay.

So . . .

Image: https://www.pexels.com/photo/pensive-girl-with-school-notebook-thinking-16117755/

Speaker notes

. . . what are genetic algorithms in the first place? They are . . .

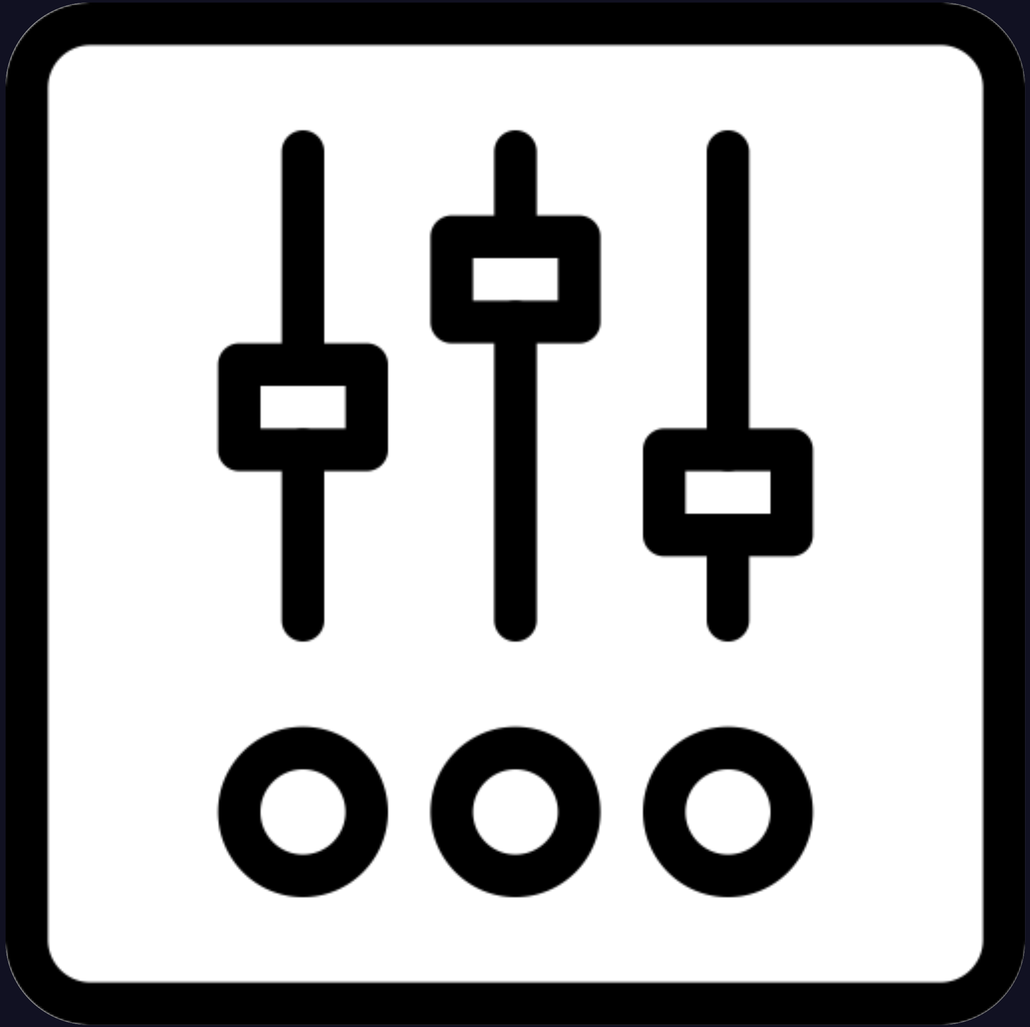


Image: https://icon-icons.com/download/49925/PNG/512/

Speaker notes

... optimization heuristics, . . .

WAT?!

Speaker notes

. . . which is fancy-talk for . . .

Optimization Heuristic:

shortcut to find
"good enough" solutions
(ideally the best,
but OK if not).

Speaker notes

. . . shortcuts to find solutions to hard problems. Ideally they'll find the best solution, but in reality we usually have to settle for something "good enough", due to constraints like time or money.

There are many kinds of optimization heuristics, but genetic algorithms are uniquely inspired by . . .

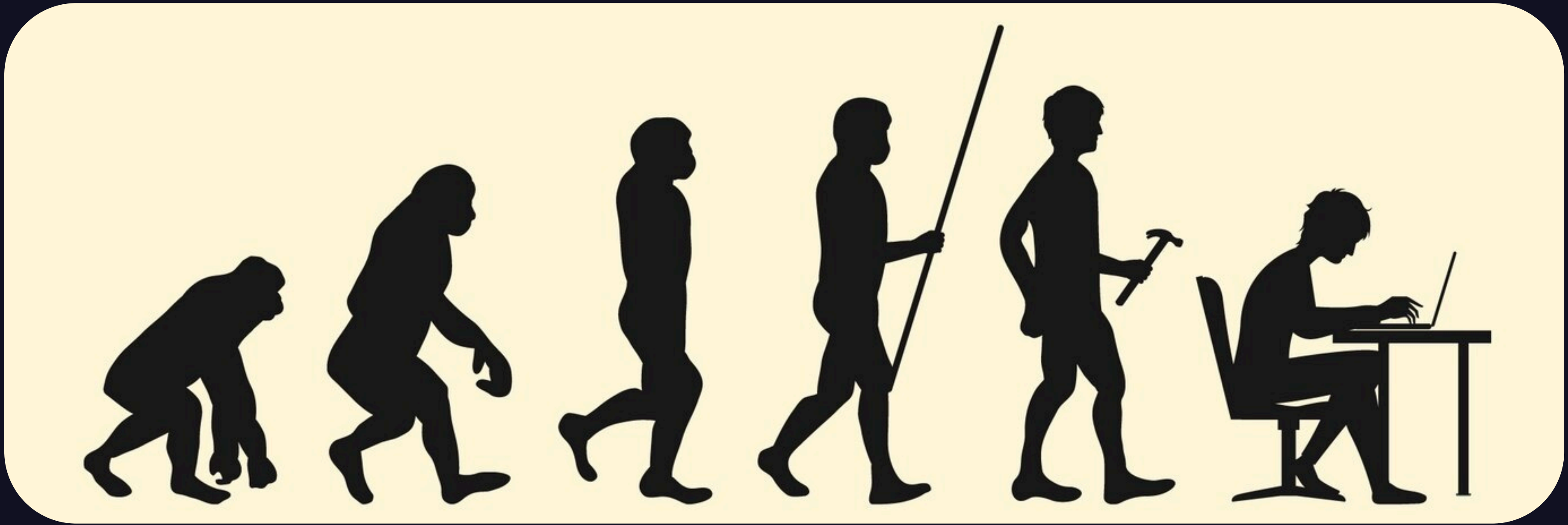
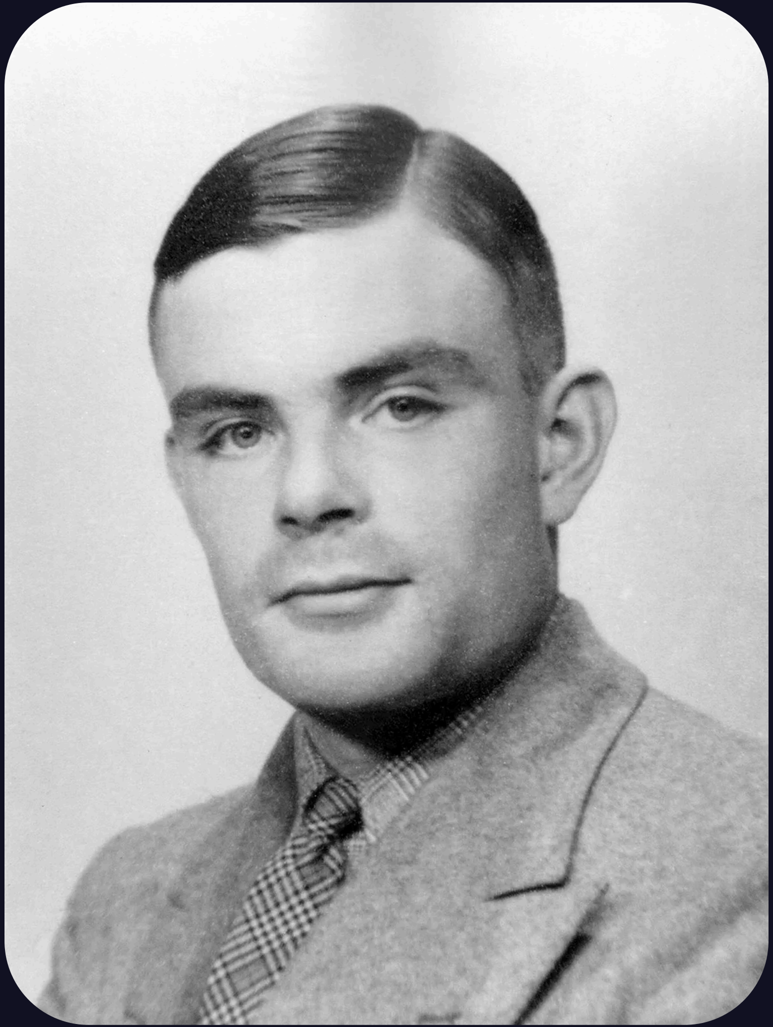


Image: https://www.freepik.com/free-vector/human-workers-evolution-silhouettes_725773.htm

Speaker notes

. . . real-world biological evolution, mainly the principles of survival of the fittest, random combination of old sets of genes (no, not like I'm wearing) into one new set, and random mutation.

The history goes back to 1950, when . . .



Alan Turing

Image: https://cdn.britannica.com/81/191581-050-8C0A8CD3/Alan-Turing.jpg

Speaker notes

. . . Alan Turing, as in Turing Test, Turing Machine, and so on, proposed a "learning machine" in which he thought that the mechanism of learning would be similar to evolution. Nothing much ever came of that, and it took a few decades for genetic algorithms in general to get some traction. The first commercial product based on genetic algorithms, a mainframe toolkit for . . .

Image: https://pxhere.com/en/photo/267871

Speaker notes

. . . industrial processes, came out in the late 1980's, from General Electric. These days, MATLAB and such tools have some genetic algorithm facilities built-in, and most major programming languages have genetic algorithm libraries available. However, the actual uses of genetic algorithms remain mostly hidden, and in my opinion frankly rather boring, used by companies in their internal industrial processes, logistics, scheduling, and so on.

But once in a while, they do get used for something more interesting, and more publicly known. Most famously, in 2005 . . .

Image: https://www.nasa.gov/sites/default/files/thumbnails/image/nasa-logo-web-rgb.png

Speaker notes

. . . NASA used a genetic algorithm to design an . . .

ST5 antenna, and US quarter for scale

Image: https://www.jpl.nasa.gov/nmp/st5/IMAGES/st5-antenna.jpg

Speaker notes

. . . antenna for the ST5 series of satellites, launched in 2006. (No, that's not just a paperclip bent up by someone fidgeting in a boring meeting.) The NASA Jet Propulsion Laboratory website says: "Its unusual shape is expected because most human antenna designers would never think of such a design." And that is one of the great advantages of this approach.

So, how do genetic algorithms work? They consist of a simple series of steps:

Initialize

Speaker notes

First, we create an initial population of candidates. In Genetic Algorithm terms, these are called "chromosomes", but since most living beings contain many chromosomes in each and every cell, I don't like that term, I think it leads to confusion, so I'm just going to say "candidates". I've also heard them called individuals, solutions, or phenotypes, to use another actual genetic term, but most people don't know that word, so I'll skip that one too.

The next step is to . . .

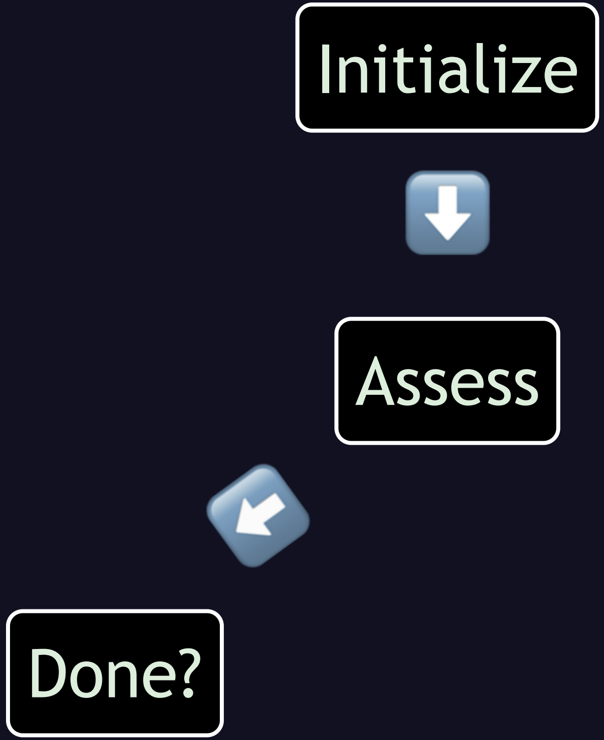
Initialize

⬇️

Assess

Speaker notes

. . . assess the "fitness" of each candidate, according to whatever criteria we want to apply. We do it here mainly because it supplies the data usually used in the next step, which is to ask, are we . . .


Initialize
Assess
Done?

Speaker notes

. . . done yet? This is usually based on the fitness, but could be based on other criteria, or a combination. If we're not done, then next we . . .

Initialize

Assess

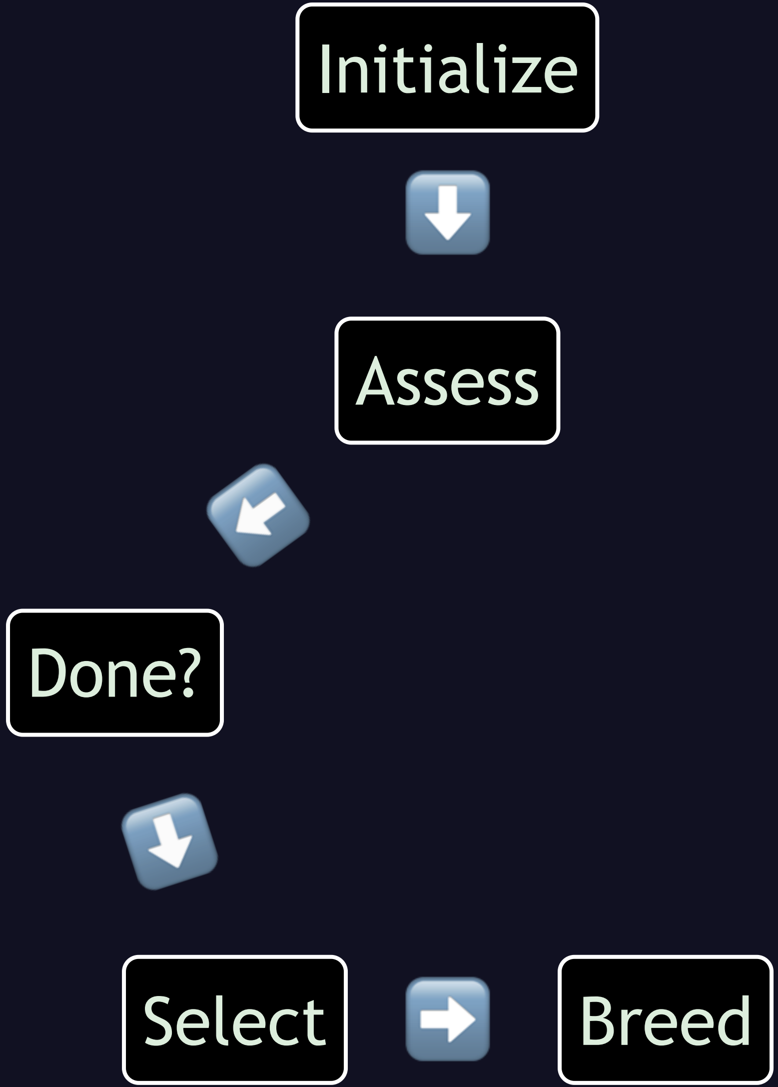
Done?

Select

Speaker notes

. . . select some candidates to breed the next generation. This is also usually based on the fitness, to simulate survival of the fittest.

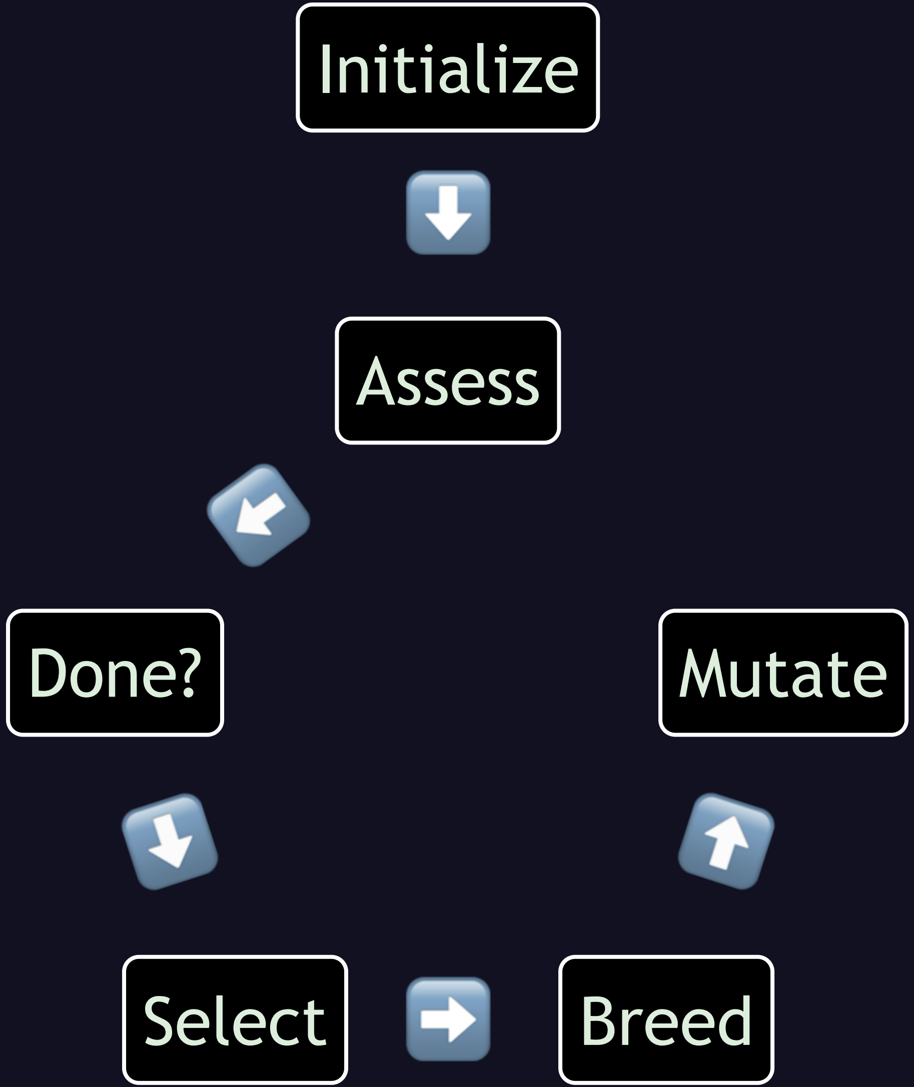
After that, as you may have guessed, we use those candidates we just selected . . .


Initialize
Assess
Done?
Select
Breed

Speaker notes

. . . to breed a new population. Some of the previous population, especially the fittest ones, may be carried over into this new population, but usually not.

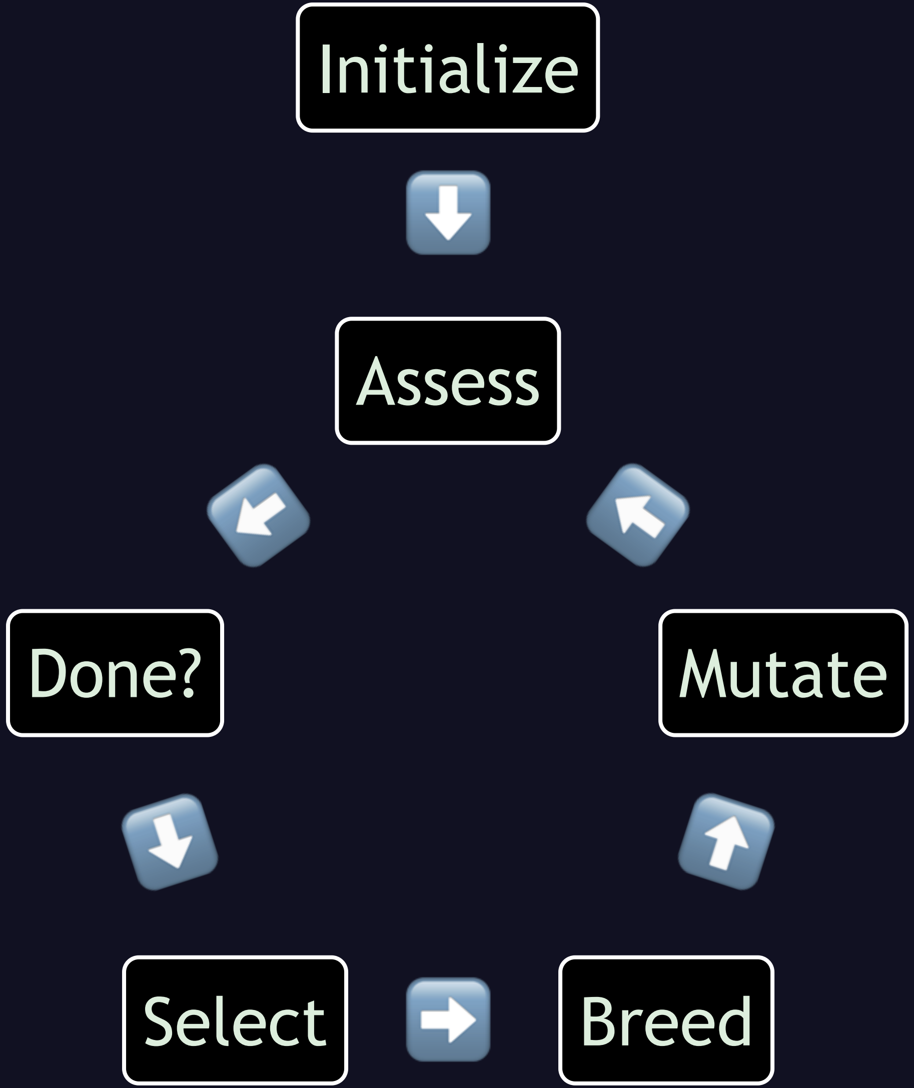
Next is a very important but easily forgotten step, which is to . . .


Initialize
Assess
Done?
Mutate
Select
Breed

Speaker notes

. . . mutate those new candidates, for more diversity in the gene pool.

Finally, we . . .


Initialize
Assess
Done?
Mutate
Select
Breed

Speaker notes

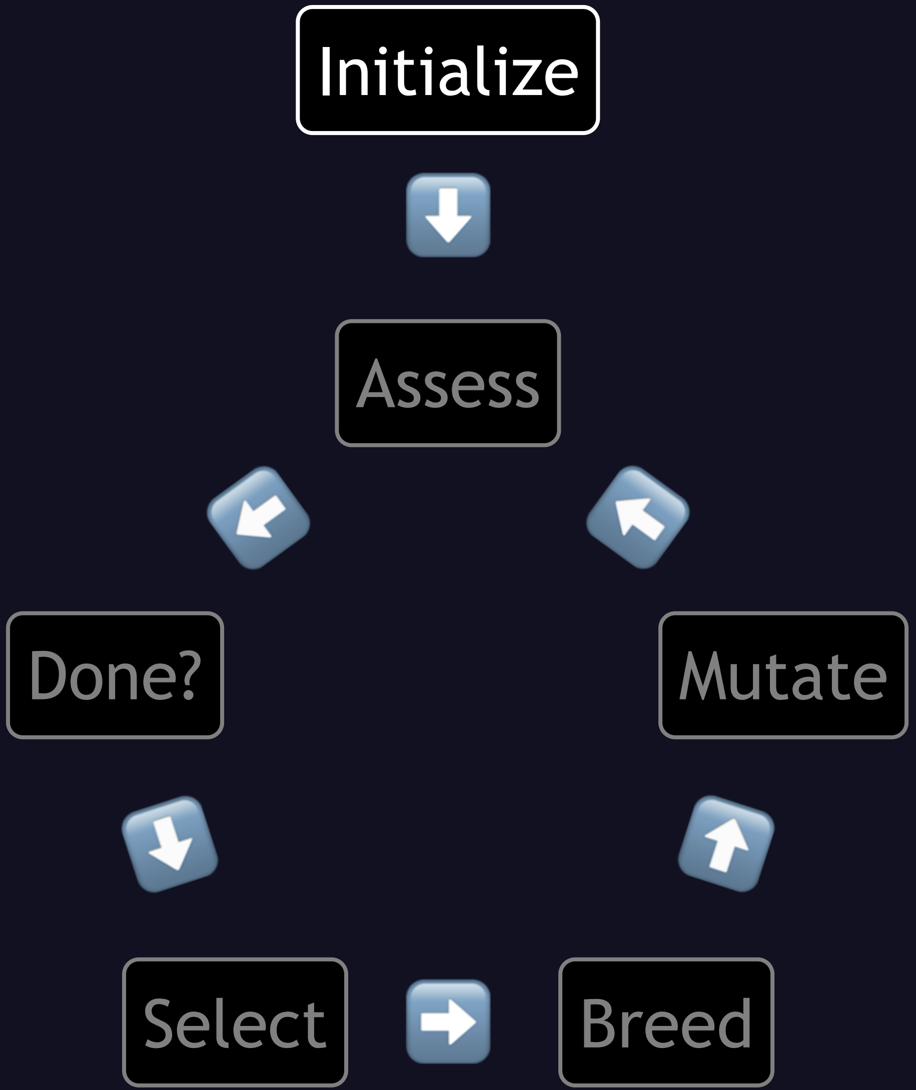
. . . go back to step 2, assessing their fitness. This sequence could be represented at a high level with some rather simple code, like so:

```
how_many = 10  # or however big we want
pop = initial_pop(how_many)
evaluate(pop)
while not done?(pop)
  breeders = select_breeders(pop)
  pop = breed(breeders, how_many)
  mutate(pop)
  evaluate(pop)
end
```

Speaker notes

This code is in Ruby. I chose that to reduce boilerplate overhead, and because it reads so close to plain English, so even if you don't know Ruby, I'm confident you'll understand the ideas, and maybe even the actual code.

Now let's take a closer look at what goes on in each step, by working through an example.


Initialize
Assess
Done?
Mutate
Select
Breed

Speaker notes

First we create an initial population of candidates. But what is a candidate, and how do we create one? These are different solutions to some problem, usually represented as different instances of the same data structure. They could be any data structure we want, so long as we can evaluate their fitness, combine old ones to make a new one, and mutate them. The simplest common type of candidate is . . .

01001000
01100101
01101100
01101100
01101111
00100000
01110111
01101111
01110010
01101100
01100100
00100001

Speaker notes

. . . a simple string of bits. This will do fine for candidates that consist of a simple series of yes/no decisions. This may sound simplistic, but there is a huge class of problems that boil down to this, called . . .

Knapsack / Rucksack / Backpack / Whatever!

Image: https://publicdomainvectors.org/en/free-clipart/Rucksack-vector-image/9910.html

Speaker notes

. . . knapsack problems. The canonical example is that you have a knapsack, and many things you want to carry in it, but they won't all fit, so or the total weight is more than you can carry, or some such similar constraint, or combination of constraints. So you want to find the combination of items, that will fit the constraints, and has the maximum value. That could be the literal cash value, or something more metaphorical. To look at a concrete example, suppose we know . . .

Image: standard emoji

Speaker notes

. . . a farmer, with a smallish truck, and he needs to decide what to take to market each week. And on this farm he has . . .

Speaker notes

. . . some cows. (E-I-E-I-O!) So among the things he can take to market are:







Images: standard emoji plus https://www.rawpixel.com/image/6130389/

Speaker notes

. . . cows, milk, cheese, butter, ice cream, meat, and leather. For the sake of simplicity, we won't differentiate between price and profit, nor dairy versus meat cows, and he can only take a set amount of each item, and always has that amount on hand. His truck has room to take all the items, but it can only carry so much weight, so that's our constraint. His choices are as follows:

What	Unit	Qty	Pounds	Value
Cow	cow	1	1,500	$2,000
Milk	1-gal jug	200	1,720	$800
Cheese	5-lb wheel	200	1,000	$12,000
Butter	1-lb block	1,000	1,000	$3,000
Ice Cream	1-gal tub	200	1,000	$2,000
Meat	side	4	1,280	$8,000
Leather	hide	20	1,100	$6,000
TOTAL WEIGHT:			8,600	

Speaker notes

You don't need to remember all that, just notice that it totals 8,600 pounds. But, his truck's suspension can only handle two tons, in other words, 4,000 pounds.

So let's see what happens if we use a genetic algorithm to determine a "good enough" truckload. First we need a way to represent each candidate. In code, we could represent them as a class, and create one randomly, like so:

```ruby
class Truckload
  attr_reader :contents
  def initialize()
    @contents = rand(128)
  end
end
```

Speaker notes

Whoa, that looks like we're just making a random number! That's right! Making a random number from 0 to 2^7-1, gives us a random 1 or 0 for each of our seven possible items. We could get as complex as we want in this function, like dictating a minimum or maximum number of items, but let's keep it simple.

To create an initial population, we can just create a bunch of candidates and stuff them into an array, . . .

```
def self.initial_population(how_many)
  population = []
  for i in 1..how_many
    population.append(self.new)
  end
  return population
end
```

Speaker notes

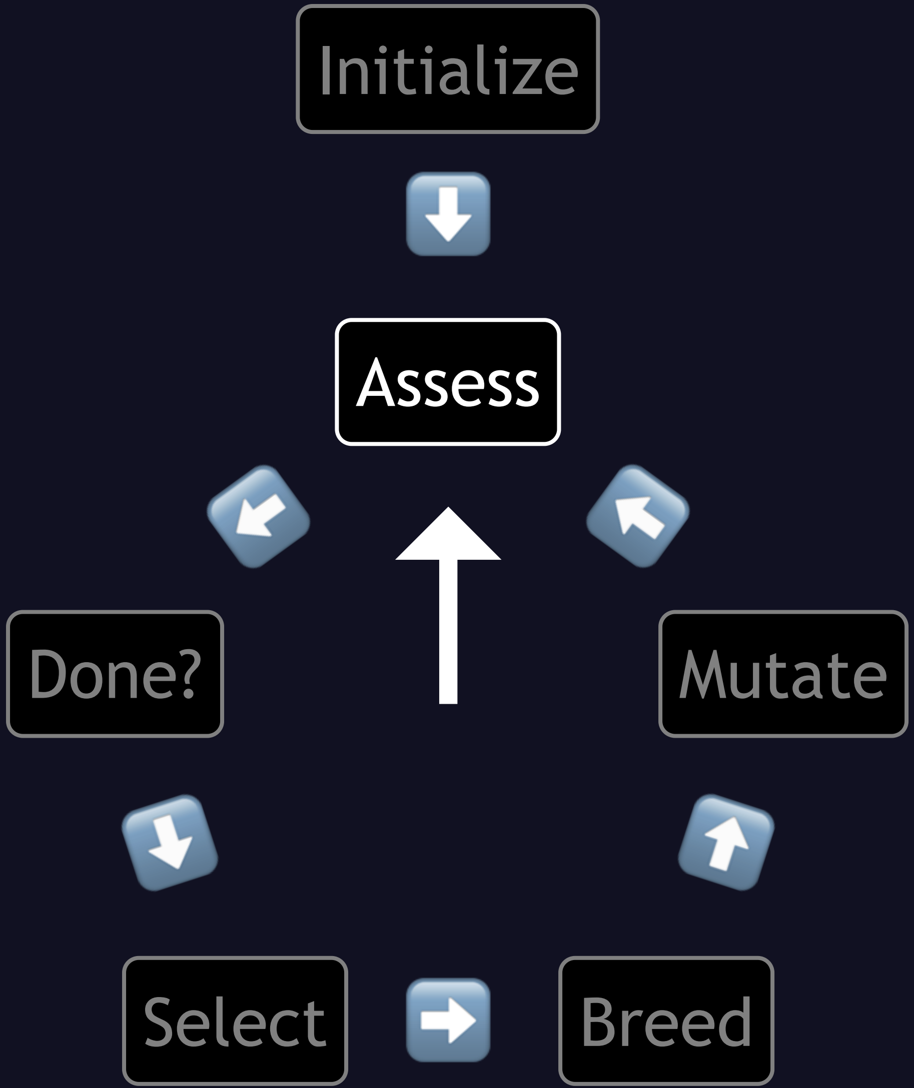
. . . like so. (This could actually be done in much more idiomatic Ruby, so those of you who do know Ruby, please don't scold me for that, I'm just trying to keep it easily understandable by people who don't know Ruby.) So if we create a population of ten Truckloads, we might wind up with a list like this:

Cow	Milk	Cheese	Butter	Ice Cream	Meat	Leather
Y	N	N	Y	N	Y	Y
N	N	N	Y	Y	N	N
N	Y	N	N	N	Y	N
N	Y	Y	N	Y	N	N
Y	Y	Y	N	Y	Y	N
Y	Y	N	Y	N	N	N
Y	N	N	Y	N	Y	N
Y	Y	N	N	N	N	N
N	N	Y	Y	Y	Y	Y
N	N	Y	N	N	Y	N

Speaker notes

Why ten? Because that's what fits on the screen in a decently readable size. If I were doing this for real, I might use a hundred, a thousand, a million, or even more.

We can get from random numbers to those combinations, by iterating over the bits and seeing which are turned on, but in the interests of time, I'll handwave over those details. Next, we . . .


Initialize
Assess
Done?
Mutate
Select
Breed

Speaker notes

. . . assess how "fit" each of these truckloads is. We do this with something called a "fitness function". (Surprise!) Just like how biological creatures might be perfectly fit for one environment but a lousy fit for another, this should reflect how fit a candidate is for some particular purpose. In this case, we already know we want the total cash value, BUT, any load that's too heavy for the truck, is worthless. In Ruby, that would look like this:

```ruby
def fitness()
  items = (0...ITEMS.count).
    select { |idx| bit_on?(idx) }.
    map { |idx| ITEMS[idx] }
  weight = items.map(&:weight).sum
  if weight > 4000
    return 0
  else
    return items.map(&:value).sum
  end
end
```

Speaker notes

We decode which items we want to take (abstracting away the actual bit-checking again for simplicity), then sum up their weights. If that exceeds the truck's capacity, we return zero, else we sum up their values.

Again, we could get as complex as we want in this function, and NASA's antenna fitness function certainly must have been. For instance, we could take into account the costs of refrigerating or freezing any items that need it.

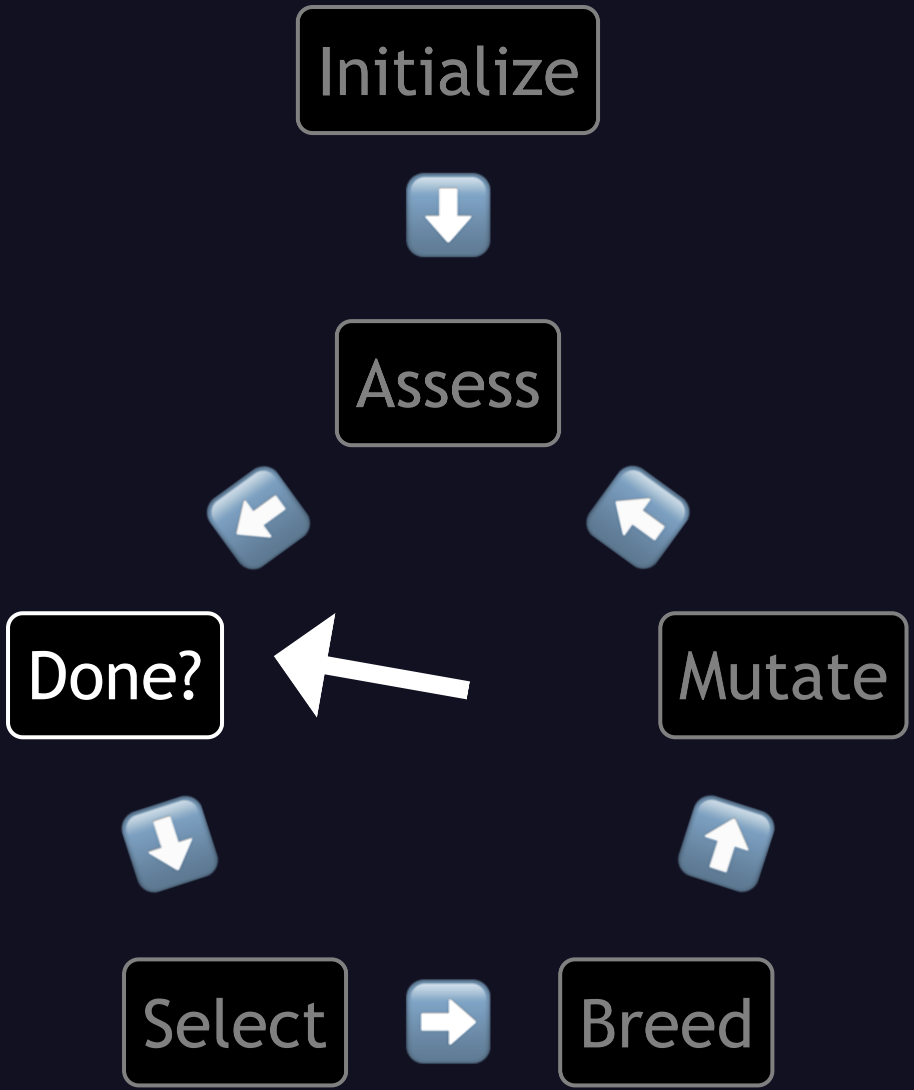
If we run this fitness function on our population, and sort on fitness descending, to make it easy to find the best, we get this:

Cow	Milk	Cheese	Butter	Ice Cream	Meat	Leather	Fitness
N	N	Y	N	N	Y	N	20,000
N	Y	Y	N	Y	N	N	14,800
Y	N	N	Y	N	Y	N	13,000
N	Y	N	N	N	Y	N	8,800
N	N	N	Y	Y	N	N	5,000
Y	Y	N	N	N	N	N	2,800
Y	N	N	Y	N	Y	Y	0
Y	Y	Y	N	Y	Y	N	0
Y	Y	N	Y	N	N	N	0
N	N	Y	Y	Y	Y	Y	0

Speaker notes

Remember that 20,000 figure.

So now that we've assessed their fitness, we . . .


Initialize
Assess
Done?
Mutate
Select
Breed

Speaker notes

. . . check if we're done. So what are our criteria? The function can be simple, but it can take some thinking to figure out what the function should do. With a knapsack problem, a good solution, especially the best, can be made totally worthless by adding just one more . . .

Image: https://images.squarespace-cdn.com/content/v1/5dace41dae7efb0296d47681/1617070720932-SD9B9QT535LA555Q2U19/python.png (used for edu fair use)

Speaker notes

. . . waffer-theen item, and thereby exceeding the constraints. So, we're going to record the best we've seen, and stop if we don't see anything better within 100 generations.

Why 100? Pretty much random. It seems like enough for a good chance for improvement, and since what we're doing is so simple, and our population is so small, using lots of generations isn't very slow. In Ruby, that would look like this:

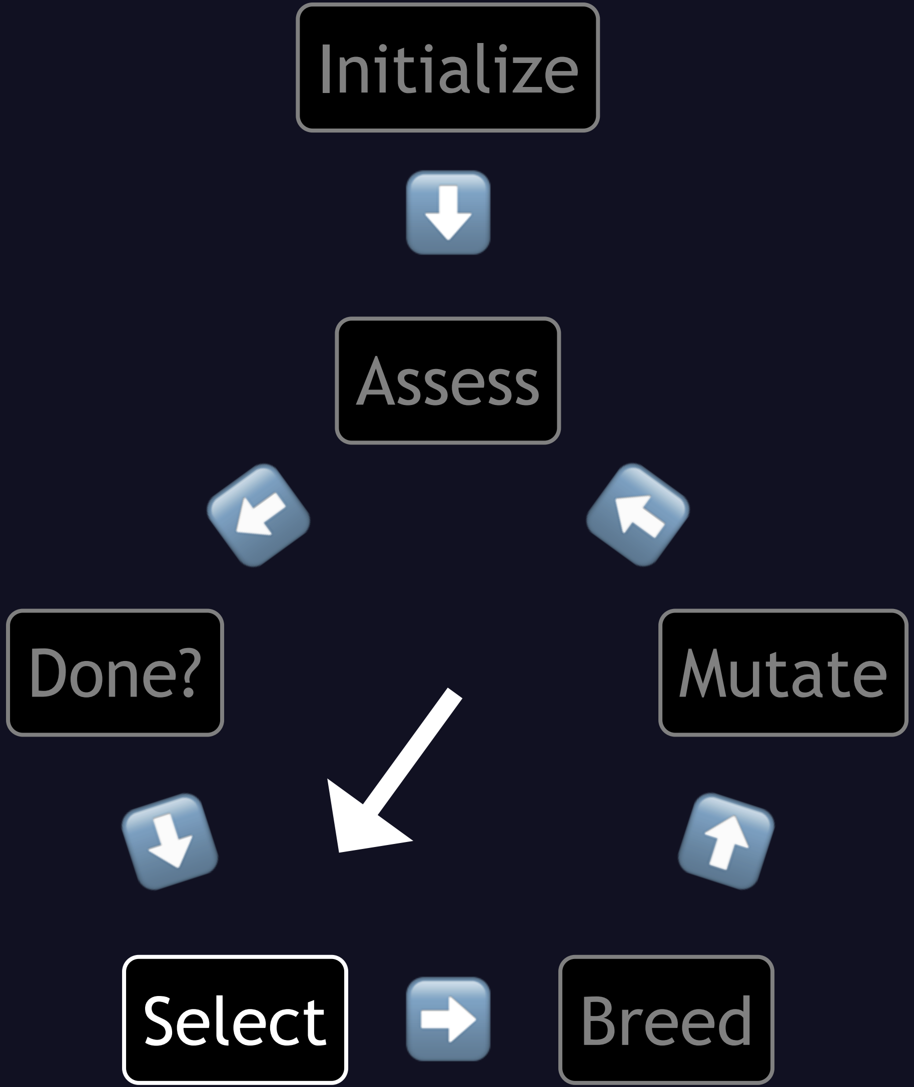
```ruby
@@best_combo = self.new(0)
@@generations = 0

def self.done?(population)
  @@generations += 1
  better = population.
    select { |c| c.fitness > @@best_combo.fitness }
  if better.any?
    @@best_combo = better.sort_by(&:fitness).last
    @@generations = 0
    return false
  else
    return @@generations >= 100
  end
end
```

Speaker notes

When this code is initially run, to define the function, we set the initial best combo as empty, and we set how many generations it's been since we saw that, as zero, both as class variables. When the function is called, we increment the number of generations, look at the fitness of the current candidates, and select the ones with a better fitness than our benchmark. If there are any better candidates, we make the fittest one our new benchmark, reset the generation counter, and return false. Else if it's been 100 generations since the best one, we return true, else we return false.

Again, we can get as complex as we want, not only in checking the maximum fitness, but we could look at other stopping criteria, like the average or minimum fitness, or achieving some specific level of maximum fitness, some maximum number of generations, or amount of time, (whether clock time or CPU or whatever), or let the user click a STOP button, or many other ways, or a combination of ways.

Since we're not done, the next step is to . . .


Initialize
Assess
Done?
Mutate
Select
Breed

Speaker notes

. . . select some candidates to breed the next generation. The obvious way is to take the top two most fit, like so:

```ruby
def self.select_breeders(population)
  return population.
    sort_by(&:fitness).
    reverse.
    take(2)
end
```

Speaker notes

We take the population, sort them by fitness in descending order, and take the first two. Out of our current population, that would choose:

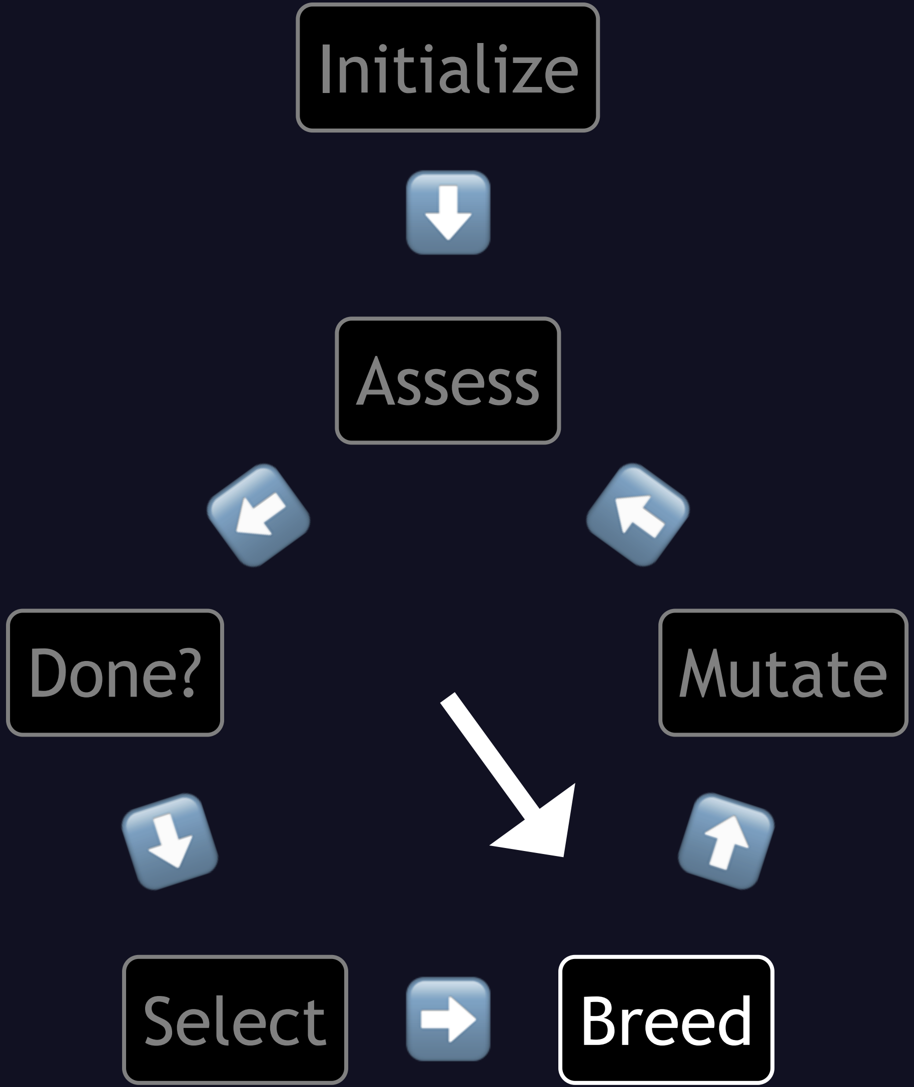
Cow	Milk	Cheese	Butter	Ice Cream	Meat	Leather	Fitness
N	N	Y	N	N	Y	N	20,000
N	Y	Y	N	Y	N	N	14,800

Speaker notes

these two. As usual, we also could get more complicated, and there are some more complex common alternatives, that we will look into later.

We could also take more than two, whether to combine more than two at once or to breed all pairs in that set, or larger subsets, such as all trios out of a set of five breeders.

Now that we've chosen our breeders, next we . . .


Initialize
Assess
Done?
Mutate
Select
Breed

Speaker notes

. . . breed them. The usual way is called crossover. This consists of taking the data points from one parent, up to some randomly chosen crossover point, then switching to the other parent. In Genetic Algorithm terms, each place in the list is called a "gene" (that's where the name comes from), and the actual value there is called an allele. In Ruby, that could look like this:

```ruby
def self.breed(p1, p2)
  cross_point = rand(ITEMS.count + 1)
  list = (0..ITEMS.count).
    map { |index|
      parent = index < cross_point ? p1 : p2
      parent.contents & (1 << index)
    }.
    sum
  return self.new(list)
end
```

Speaker notes

We establish the crossover point for each new candidate, as a random number between zero and how many items there are, inclusive. Then we iterate through the list of items. If we haven't yet hit the crossover point, we get the decision for that item from the first parent, else we get it from the other parent. This means that it could be all copied from one parent or the other, or it could switch at some point. We could just copy the sets of bits, but the code for that would be more complex and non-portable than I want to explain here.

As usual, we could get as complex as we want, like making some crossover points more or less likely, or even mandatory or forbidden. But we're going to keep it simple.

If we use this function once, with a crossover point of 3, so we take 3 values from the first parent, and the rest from the other, that would get us a result like this:

Cow	Milk	Cheese	Butter	Ice Cream	Meat	Leather
N	N	Y	N	N	Y	N
			+			
N	Y	Y	N	Y	N	N
			=			
N	N	Y	N	Y	N	N

Speaker notes

But this is just one of ten results, because we're making a whole new population, like so:

```ruby
def self.new_population(p1, p2, how_many)
  population = []
  for i in 1..how_many
    population.append(self.breed(p1, p2))
  end
  return population
end
```

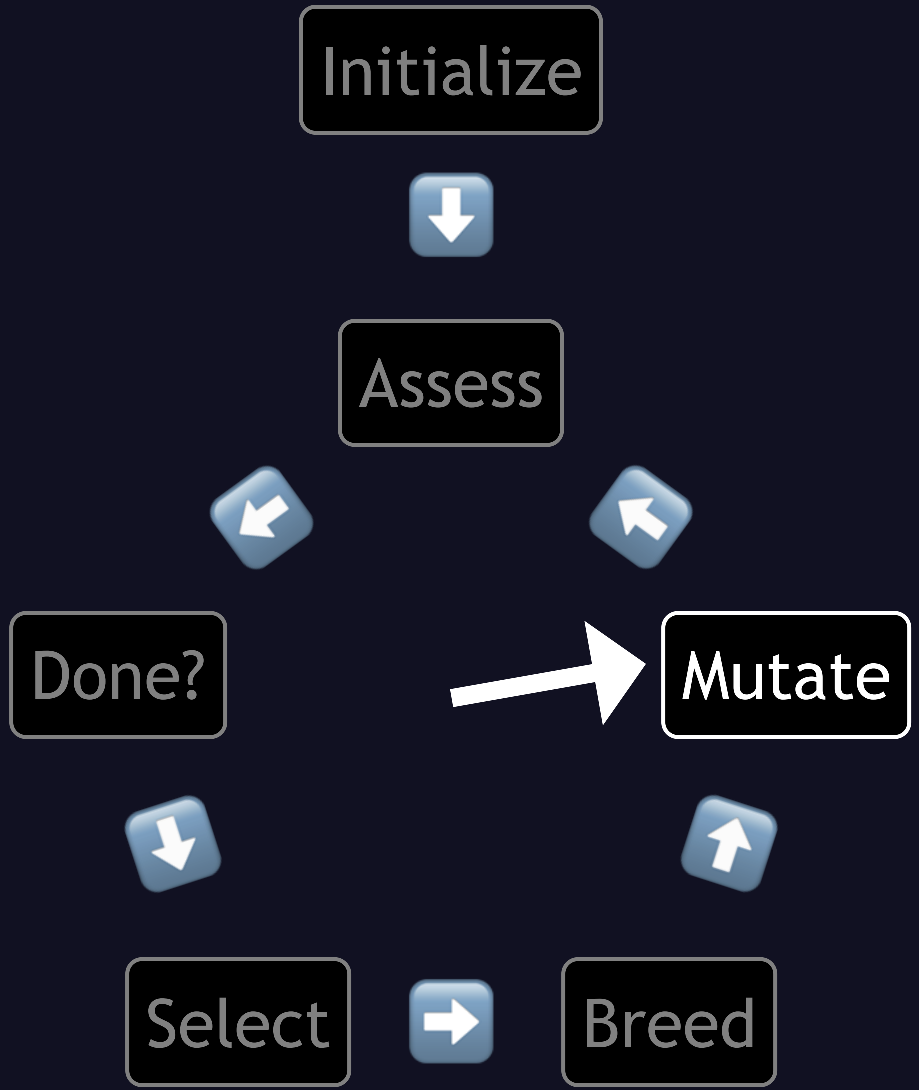
Speaker notes

This is just like how we created the initial population, except that instead of each candidate being made from scratch, they're the product of breeding our chosen breeders. The whole list might look like this:

Cow	Milk	Cheese	Butter	Ice Cream	Meat	Leather
N	N	Y	N	Y	N	N
N	Y	Y	N	Y	N	N
N	N	Y	N	Y	N	N
N	N	Y	N	N	Y	N
N	N	Y	N	Y	N	N
N	N	Y	N	Y	N	N
N	N	Y	N	N	N	N
N	N	Y	N	Y	N	N
N	N	Y	N	Y	N	N
N	N	Y	N	N	Y	N

Speaker notes

Lots of family resemblance there, eh? None of these loads include a cow, butter, or leather, and they all include cheese. That's because both of our two breeders were like that. Or in Genetic Algorithms terms, both chromosomes had the same alleles for each of those genes. If we were to just continue breeding the fittest of each generation, we wouldn't ever see any loads including a cow, butter, leather, or no cheese, but we fix that in the next step, which is to . . .


Initialize
Assess
Done?
Mutate
Select
Breed

Speaker notes

. . . mutate them. Again, I'm going to keep it very simple, and give each gene a 1 in 4 chance of flipping. In code, that looks like this:

```ruby
def maybe_mutate()
  (0..ITEMS.count).each do |index|
    if rand(4) == 0
      @contents ^= (1 << index)
    end
  end
end
```

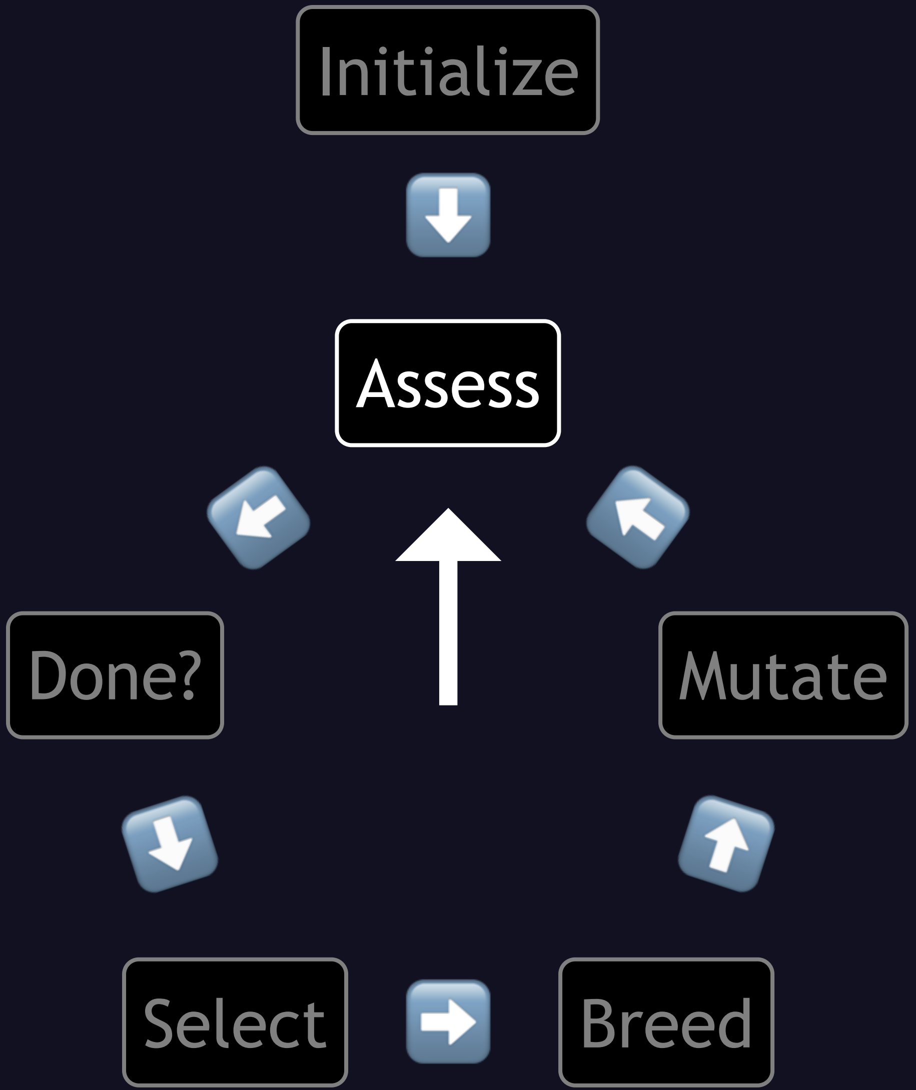
Speaker notes

We iterate through the item numbers, and for each one, if a random number from zero to three is a zero, we flip that bit. Again, we could get as complex as we want, like having some genes more or less likely to mutate than others, or having some minimum or maximum number of mutations per candidate, or all kinds of other options. If we run this mutation function on these new candidates, we might wind up with something like this:

Cow	Milk	Cheese	Butter	Ice Cream	Meat	Leather
N	Y	N	Y	N	N	Y
N	N	Y	Y	N	N	N
Y	N	Y	Y	Y	Y	N
Y	Y	Y	Y	Y	Y	N
Y	Y	Y	Y	N	N	Y
N	Y	N	Y	N	N	Y
Y	Y	N	Y	N	N	Y
N	N	Y	N	Y	N	N
Y	N	N	Y	Y	N	N
N	N	Y	N	Y	N	N

Speaker notes

. . . where green means that it changed. You can see that we now DO have some truckloads that include a cow, butter, or leather, or no cheese. Now we go back to . . .


Initialize
Assess
Done?
Mutate
Select
Breed

Speaker notes

. . . assessing the fitness of these new candidates, and we get this:

Cow	Milk	Cheese	Butter	Ice Cream	Meat	Leather	Fitness
N	N	Y	Y	N	N	N	15,000
N	N	Y	N	Y	N	N	14,000
N	N	Y	N	Y	N	N	14,000
N	Y	N	Y	N	N	Y	9,800
N	Y	N	Y	N	N	Y	9,800
Y	N	N	Y	Y	N	N	7,000
Y	Y	Y	Y	N	N	Y	0
Y	Y	Y	Y	Y	Y	N	0
Y	N	Y	Y	Y	Y	N	0
Y	Y	N	Y	N	N	Y	0

Speaker notes

Oh noes! Our maximum fitness actually went down! As you may recall, our previous best one scored 20,000. But don't worry, as you may recall from our "are we done yet" function, we hang onto the best one, and just try to outdo it, so we haven't lost it.

But that's not really the best approach. Remember how I said that sometimes the fittest members of a population might be carried over into the next one? If I had carried over the fittest one, the maximum fitness would never go down, so the "are we done" function could have been a bit simpler, and we'd be done a bit faster. But, I didn't think of it until I had already made all the slides for this talk, and I didn't want to redo them. Also, I think this version is still worth exploring, to make the point that the fitness can go down and then come back up. So let's keep going.

The next generation might look like this:

Cow	Milk	Cheese	Butter	Ice Cream	Meat	Leather	Fitness
N	Y	Y	N	N	Y	N	20,800
N	N	Y	N	N	Y	N	20,000
N	N	Y	N	N	Y	N	20,000
N	N	N	N	N	Y	Y	14,000
N	N	Y	N	N	N	N	12,000
N	Y	N	N	Y	N	N	2,800
Y	N	Y	Y	N	Y	N	0
Y	Y	Y	N	Y	Y	N	0
N	Y	Y	N	Y	Y	N	0
N	Y	Y	Y	N	Y	N	0

Speaker notes

. . . a small improvement over our prior best! So, we set that top one as our benchmark, and reset the counter of generations since we saw it. If we let this run to completion, we might wind up with something like this:

Cow	Milk	Cheese	Butter	Ice Cream	Meat	Leather	Fitness
N	N	Y	N	N	Y	Y	26,000
N	N	Y	Y	N	N	Y	21,000
N	Y	N	N	Y	Y	N	10,800
Y	N	N	N	N	N	Y	8,000
N	N	N	N	N	Y	N	8,000
N	N	N	Y	Y	N	N	5,000
N	Y	N	N	N	N	N	8,00
N	Y	N	N	N	N	N	8,00
Y	Y	N	Y	N	N	Y	0
Y	Y	N	Y	Y	Y	Y	0

Speaker notes

. . . with our best truckload scoring 26,000, made up of cheese, meat, and leather.

So that's one complete run of a genetic algorithm. If we wanted to check whether that was the best that this algorithm could produce, we could just run it again, as many times as we like, within reason, since it's so much faster than brute force. Okay, maybe writing all this code is not so much faster when we've only got seven items, and such simple criteria, but if we had to choose among many more items, with more complex criteria, for many truckloads a day, creating a genetic algorithm might well be worthwhile.

Now, suppose we want to evolve solutions to a different huge class of problems: . . .

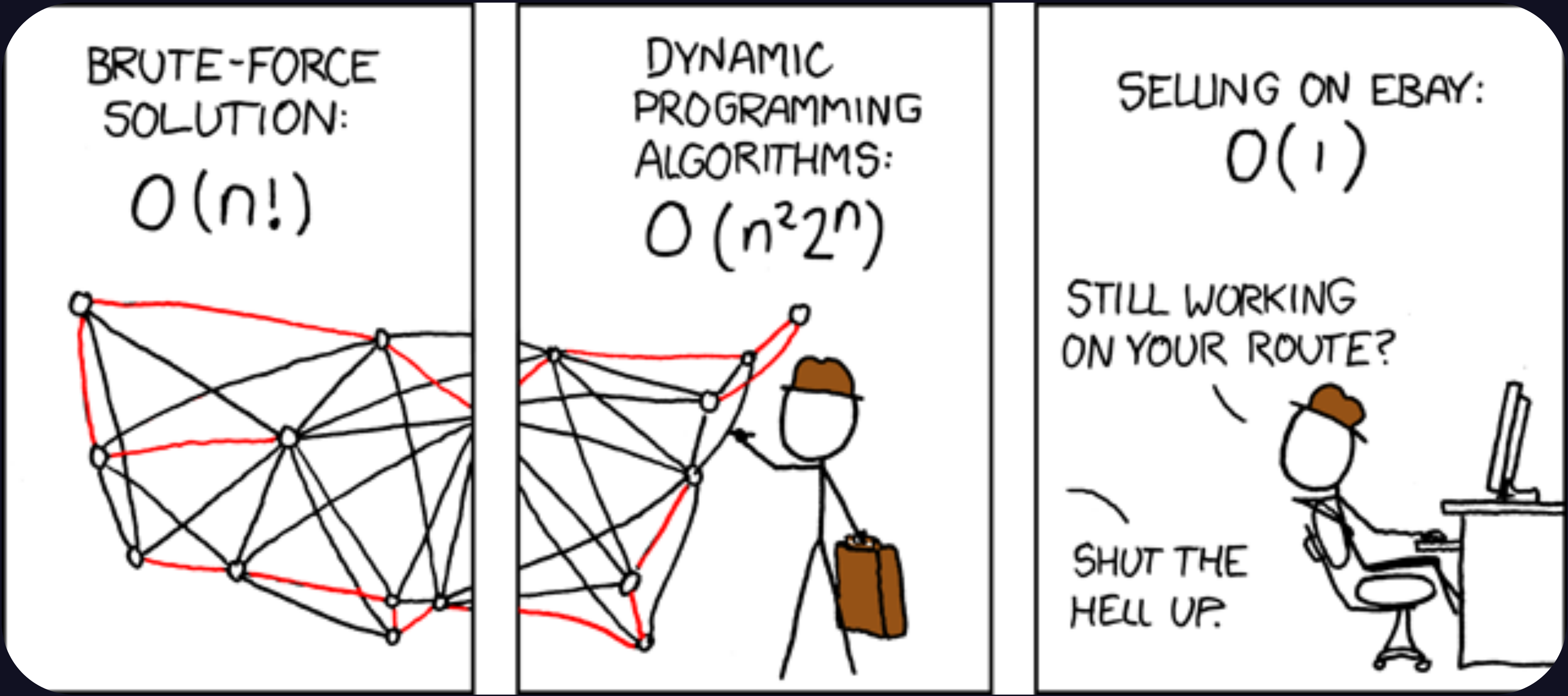


Image: https://xkcd.com/399/

Speaker notes

. . . Traveling Salesman problems. The canonical example is that you're literally a traveling salesman, and you want to find the shortest route to visit a list of cities. Realistically, we may want to include other factors, such as the time or money it takes to get there, which may not be proportional to the distance, and the expected time to spend in each city. But to keep this example simple, we'll just look at the distance.

Our list of cities will be the capitals of mainland Europe, west of Germany. In alphabetical order, that's:

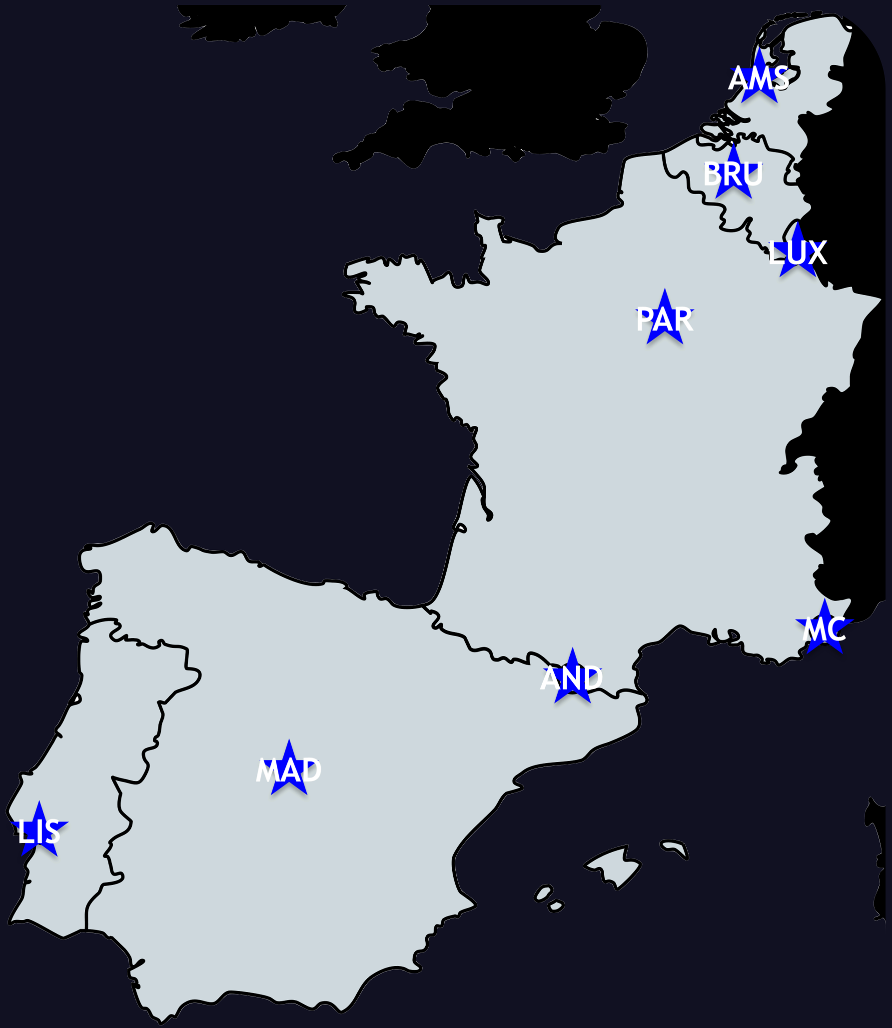


Image: modified from https://www.mapchart.net/europe.html

Speaker notes

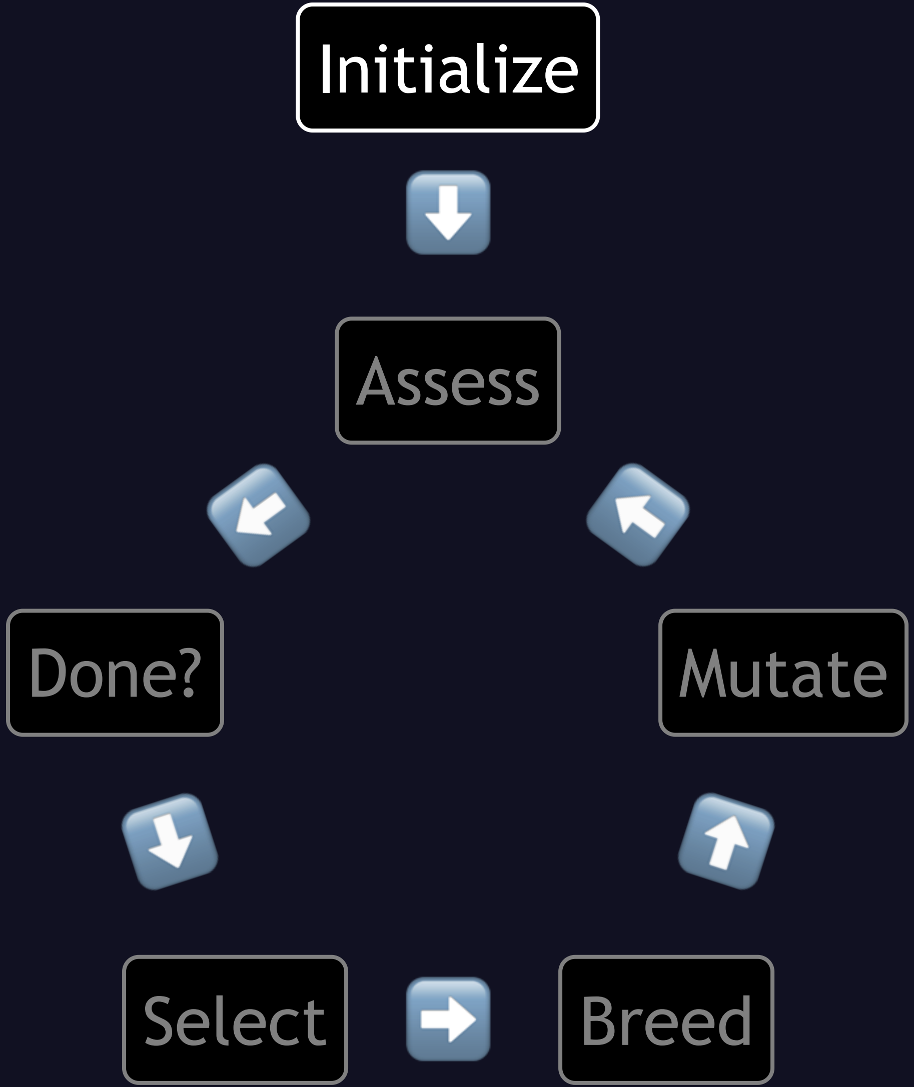
Amsterdam, Andorra la Vella, Brussels, Lisbon, Luxembourg, Madrid, Monte Carlo, and Paris. We have the . . .

To From	AND	BRU	LIS	LUX	MAD	MTC	PAR
AMS	1357	210	2233	417	1773	1421	502
AND	-	1162	1232	1178	613	653	862
BRU	-	-	2038	213	1577	1200	307
LIS	-	-	-	2153	625	1838	1739
LUX	-	-	-	-	1691	1041	386
MAD	-	-	-	-	-	1288	1278
MTC	-	-	-	-	-	-	956

Speaker notes

. . . distances between them, as shown here in kilometers. This is the minimum driving distance according to Google Maps, and to keep things simple we'll assume it's the same in either direction.

So let's dive back into our process. That starts with . . .


Initialize
Assess
Done?
Mutate
Select
Breed

Speaker notes

. . . creating an initial population. So how do we create a route? It could be done quite easily, like this:

```ruby
class Route
  CITIES = %w(AMS AND BRU LIS LUX MAD MTC PAR)

  attr_reader :stops
  def initialize(stops=CITIES.shuffle)
    @stops = stops
  end
end
```

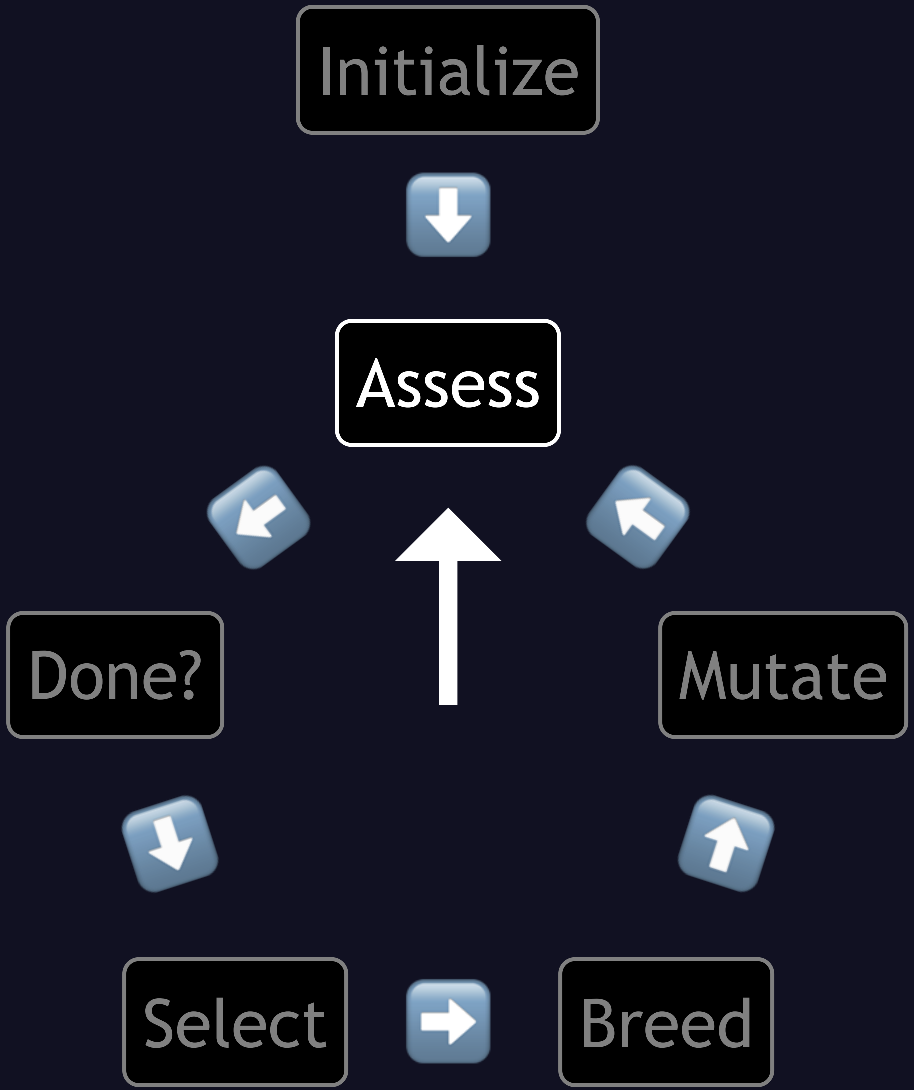
Speaker notes

Again we're going to make a class, this time called Route, with the list of city abbreviations as a constant. If we create ten of these and put them in an array, it might look like this:

1st	2nd	3rd	4th	5th	6th	7th	8th
AND	MAD	AMS	LUX	PAR	BRU	MTC	LIS
MTC	PAR	LUX	AND	MAD	LIS	AMS	BRU
MAD	LUX	BRU	AMS	LIS	AND	MTC	PAR
AND	PAR	LUX	BRU	AMS	LIS	MAD	MTC
LIS	LUX	MTC	AMS	AND	BRU	PAR	MAD
BRU	LUX	LIS	PAR	AND	MAD	MTC	AMS
AND	LUX	PAR	MTC	BRU	MAD	AMS	LIS
LUX	BRU	AND	MAD	PAR	LIS	AMS	MTC
AND	AMS	MTC	PAR	LIS	BRU	LUX	MAD
PAR	MAD	AMS	BRU	AND	MTC	LIS	LUX

Speaker notes

So that's the Initialize step done. Now we have to . . .


Initialize
Assess
Done?
Mutate
Select
Breed

Speaker notes

. . . determine how fit each route is. The measure we have is better when smaller and worse when larger, but that's easy to take care of. We can subtract it from a constant, divide a constant by it, or all kinds of other solutions. I decided to subtract the total distance of each one from . . .

```ruby
WORST_ROUTE = Route.new(%w(AMS
                           LIS
                           LUX
                           MAD
                           BRU
                           MTC
                           PAR
                           AND))

# this works out to 12_029
WORST_DISTANCE = WORST_ROUTE.fitness()
```

Speaker notes

. . . the fitness of the worst route I could easily construct manually. I started in Amsterdam, and then did the opposite of the usual heuristic, repeatedly going to the furthest unvisited city, until I had included them all, then back to Amsterdam. The fitness function is fairly straightforward:

```ruby
def fitness = WORST_DISTANCE - total_distance

def total_distance
  stops.
    each_cons(2).
    to_a.
    map { |src, dst| distance(src, dst) }.
    sum +
    distance(stops.first, stops.last)
end
```

Speaker notes

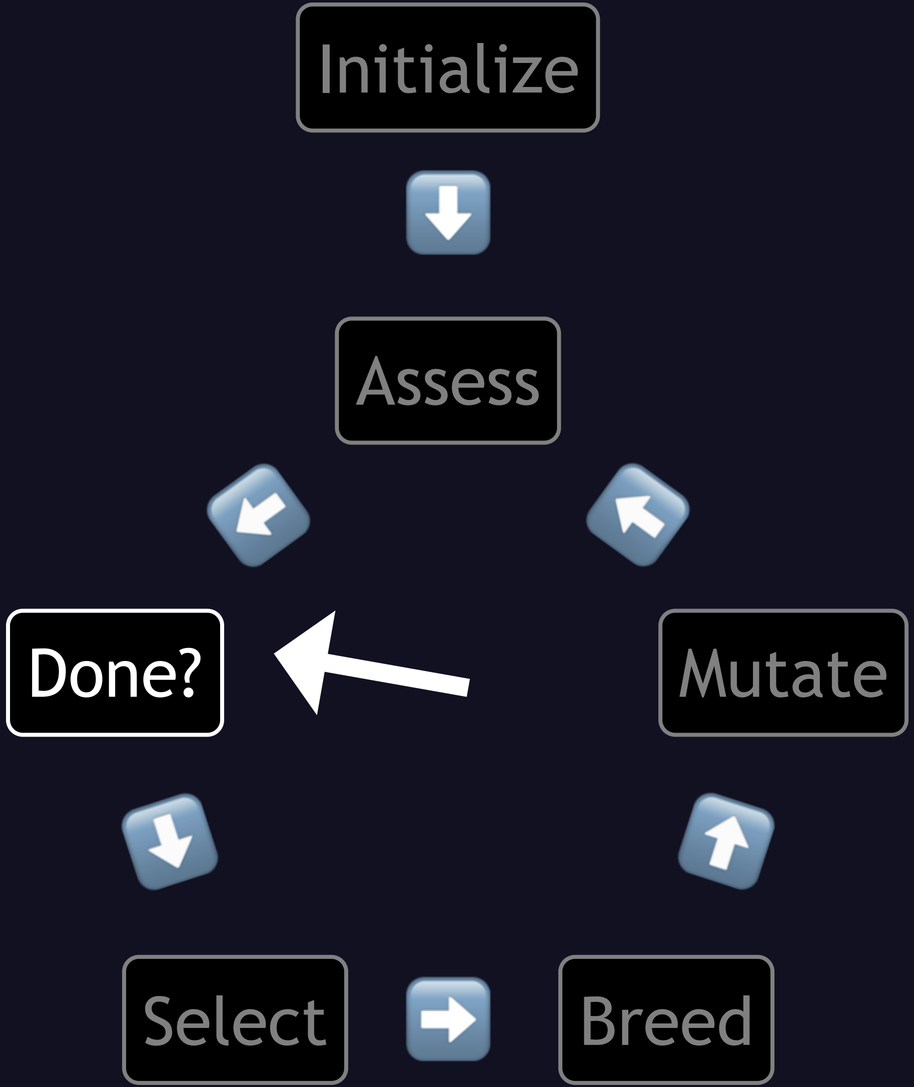
. . . we calculate the total distance by taking the stops, extracting each consecutive pair, then we map each pair to its distance by calling a function that looks up the distance in that table (but in the interests of time I'll hand-wave over that code), add them all up, and finally add the distance back to the starting point. Then to get the fitness we finally subtract that total distance from the worst distance.

If we run this on our current population, and sort on fitness descending, we get:

1st	2nd	3rd	4th	5th	6th	7th	8th	Fit
AND	PAR	LUX	BRU	AMS	LIS	MAD	MTC	5559
MTC	PAR	LUX	AND	MAD	LIS	AMS	BRU	4628
AND	MAD	AMS	LUX	PAR	BRU	MTC	LIS	4263
MAD	LUX	BRU	AMS	LIS	AND	MTC	PAR	3563
BRU	LUX	LIS	PAR	AND	MAD	MTC	AMS	3530
LIS	LUX	MTC	AMS	AND	BRU	PAR	MAD	2685
PAR	MAD	AMS	BRU	AND	MTC	LIS	LUX	2576
LUX	BRU	AND	MAD	PAR	LIS	AMS	MTC	2329
AND	AMS	MTC	PAR	LIS	BRU	LUX	MAD	2001
AND	LUX	PAR	MTC	BRU	MAD	AMS	LIS	1494

Speaker notes

. . . this. The best route of this generation is 5_559 km shorter than the worst route I could easily construct manually, and the worst of this generation is still 1_494 km shorter than that same worst route. So now we can use this information to decide . . .


Initialize
Assess
Done?
Mutate
Select
Breed

Speaker notes

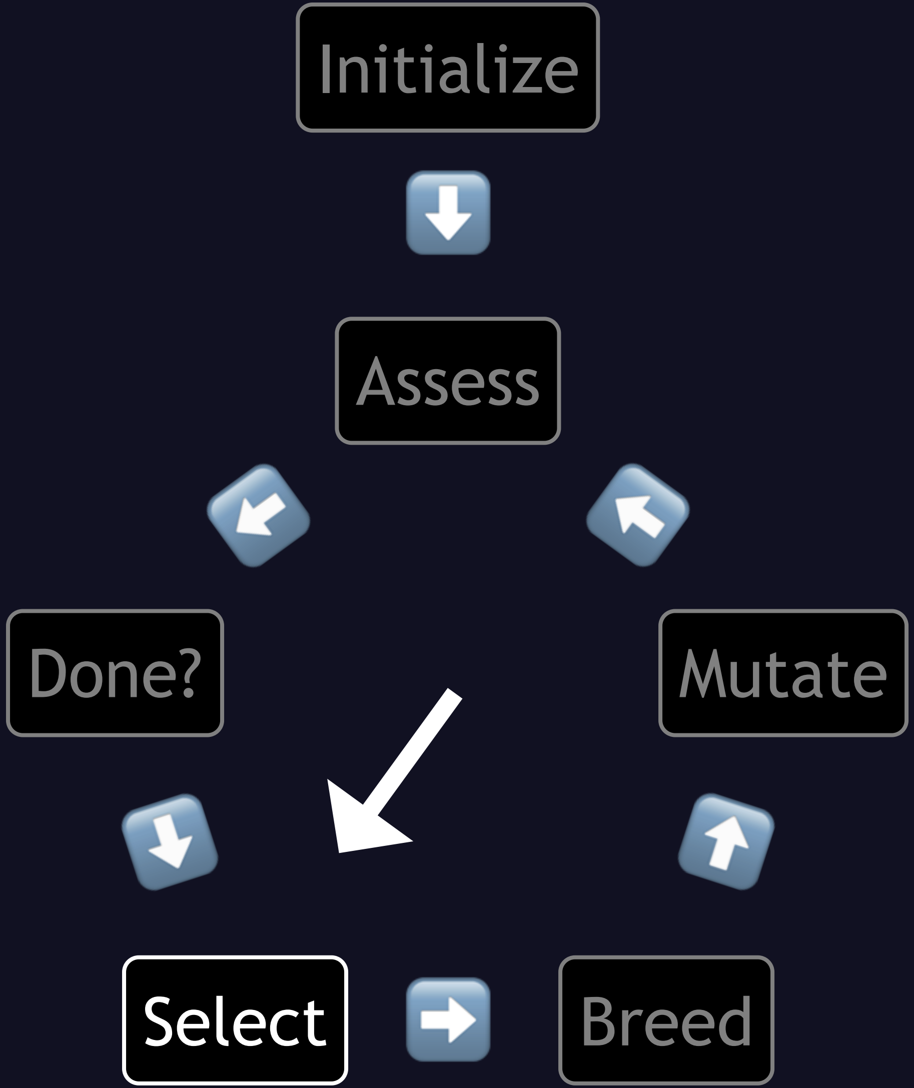
. . . are we done? What are our criteria? For now I'm going to stick with the idea of declaring a winner if it hasn't been outdone in 100 generations, so of course we're not done on the first pass. The code . . .

```ruby
@@best_route = WORST_ROUTE
@@generations = 0

def self.done?(population)
  @@generations += 1
  better = population.
    select { |r| r.fitness > @@best_route.fitness }
  if better.any?
    @@best_route = better.sort_by(&:fitness).last
    @@generations = 0
    return false
  else
    return @@generations >= 100
  end
end
```

Speaker notes

. . . is exactly the same as last time, except that now we're using Routes instead of Truckloads.

Since we're not done, we now . . .


Initialize
Assess
Done?
Mutate
Select
Breed

Speaker notes

. . . pick some breeders, and this time we're going to delve into Roulette Wheel selection. The concept there is that every candidate has a chance to be selected, but the size of the chance is based on the candidate's fitness. (So I think this is actually a misnomer; one would hope that an actual roulette wheel has an equal chance for all the numbers! But, they didn't ask my opinion when naming this.) The simplest version is that it's just equal, or at least directly proportional, to the fitness. The . . .

```ruby
def self.select_breeders(pop)
  p1 = pick_winner(pop)
  p2 = pick_winner(pop - [p1])
  [p1, p2]
end

def self.pick_winner(pop)
  total = pop.map(&:fitness).sum
  target = rand(total)
  so_far = 0
  pop.each do |p|
    so_far += p.fitness
    return p if so_far > target
  end
end
```

Speaker notes

. . . code for that is of course more complex than last time, when we simply chose the top two. Basically what we're doing is summing up all the fitnesses, generating a random number from 0 to the sum minus 1, and figuring out which route's range that falls in. To do that, we iterate over the routes, summing the fitnesses again, until we exceed the random number. When that happens, it means that the one whose fitness we just added is the lucky winner. Then we do that again, without the one we already picked.

As usual we could make it even more complex, such as by applying some function to the actual fitness. We might amplify the fitter routes' chances by squaring the fitness, or diminish them by taking the square root, or a logarithm or whatever, or we could make those things part of the definition of the fitness in the first place. We could also bias it by generating that random number differently, with an uneven distribution, whether favoring the best or the middle or whatever.

If we run this on our current population, we just might randomly wind up with . . .

1st	2nd	3rd	4th	5th	6th	7th	8th	Fit
MAD	LUX	BRU	AMS	LIS	AND	MTC	PAR	3563
LUX	BRU	AND	MAD	PAR	LIS	AMS	MTC	2329

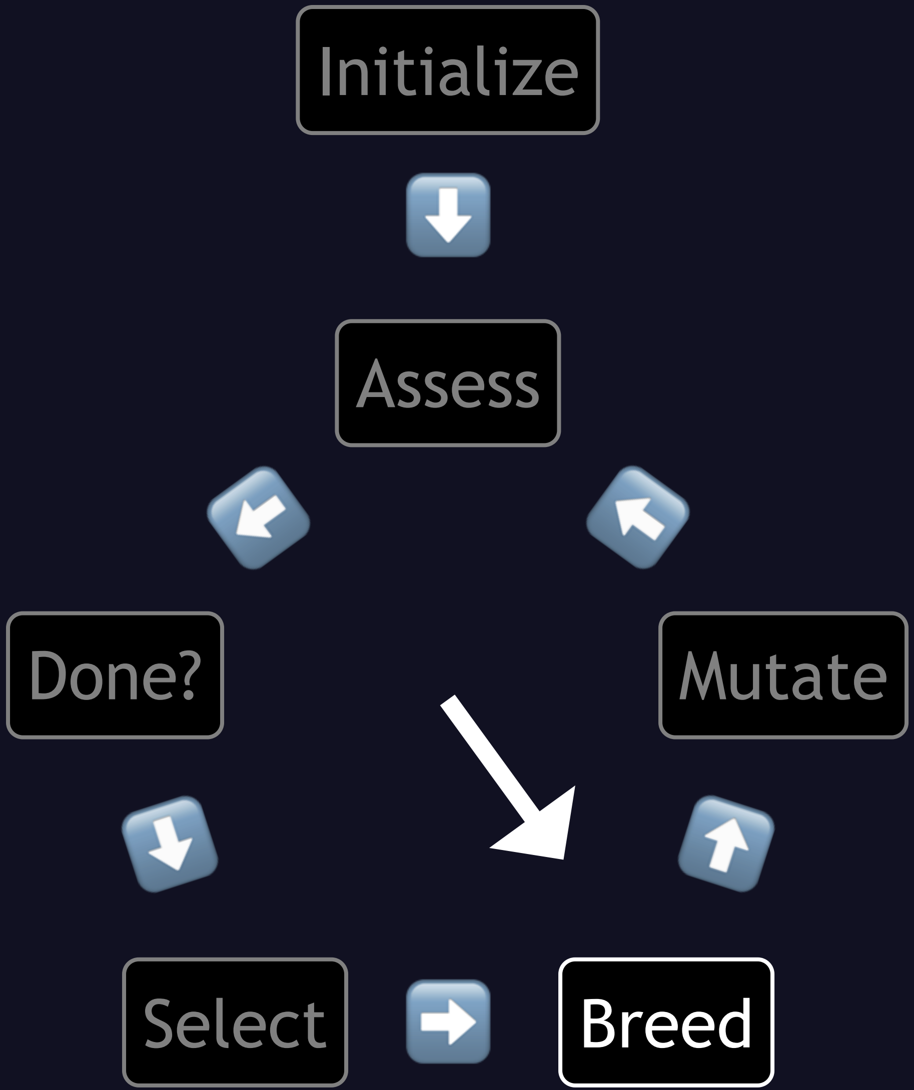
Speaker notes

. . . these two. These weren't the top two, in fact they were #4 and #8, which is fairly poor, averaging out to #6 out of 10. But just like in real life, to quote the late great . . .

Image: https://commons.wikimedia.org/wiki/File:Tom_Petty_(8191710373).jpg

Speaker notes

. . . Tom Petty, even the losers get lucky sometimes. So now it's time to . . .


Initialize
Assess
Done?
Mutate
Select
Breed

Speaker notes

. . . breed them together. Breeding Traveling Salesman routes is much more complex than breeding Knapsack contents. It's actually still an area of ongoing research! But the simplest way to breed Traveling Salesman routes is fairly similar to ordinary crossover: we . . .

```ruby
def self.breed(p1, p2)
  xover = rand(CITIES.length + 1)
  cities = []
  cities[0 .. (xover - 1)] =
    p1.stops.slice(0, xover)
  cities[xover .. (CITIES.length - 1)] =
    p2.stops.reject { |city| cities.member?(city) }
  return Route.new(cities)
end
```

Speaker notes

. . . take a random number, between 0 and the number of cities, inclusive, copy that many from the first parent, and fill the rest (if any) from the second parent. However, we don't do it by copying them straight down in place, like with regular crossover, but by using the missing cities, in the order they appear in the second parent, no matter exactly where they appeared. So, supposing we have a crossover point of 3, the result would look like . . .

1st	2nd	3rd	4th	5th	6th	7th	8th
MAD	LUX	BRU	AMS	LIS	AND	MTC	PAR
			+				
LUX	BRU	AND	MAD	PAR	LIS	AMS	MTC
			=				
MAD	LUX	BRU	AND	PAR	LIS	AMS	MTC

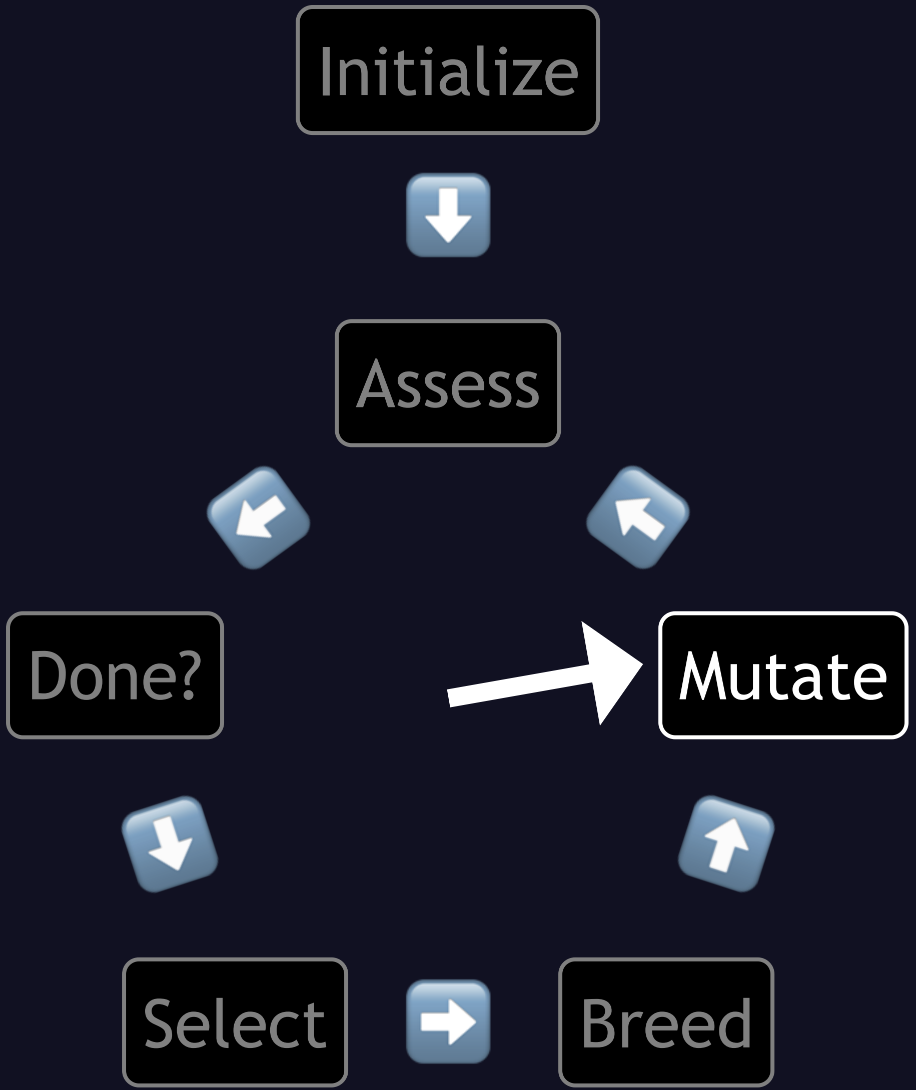
Speaker notes

. . . this, with Madrid, Luxembourg, and Brussels copied straight down, and then Andorra la Vella, Paris, Lisbon, Amsterdam, and Monte Carlo filled in in the order they appear in the second parent. It happens to match their order in the first parent, but that's actually unusual, and they're not consecutive in the second one, interrupted by Madrid. Do you see how that works? (PAUSE FOR CONFIRMATION, EXPLAIN IF NEEDED.) Next, as you may recall, this is just one of ten results, as we're making a whole new population. One possible result might be . . .

1st	2nd	3rd	4th	5th	6th	7th	8th
MAD	LUX	BRU	AND	PAR	LIS	AMS	MTC
MAD	LUX	BRU	AND	PAR	LIS	AMS	MTC
MAD	LUX	BRU	AND	PAR	LIS	AMS	MTC
LUX	BRU	AND	MAD	PAR	LIS	AMS	MTC
MAD	LUX	BRU	AMS	AND	PAR	LIS	MTC
MAD	LUX	BRU	AND	PAR	LIS	AMS	MTC
MAD	LUX	BRU	AND	PAR	LIS	AMS	MTC
MAD	LUX	BRU	AND	PAR	LIS	AMS	MTC
LUX	BRU	AND	MAD	PAR	LIS	AMS	MTC
LUX	BRU	AND	MAD	PAR	LIS	AMS	MTC

Speaker notes

. . . this. This time we have not only strong family resemblance, but a fair bit of full duplication too. If you want to continue the biological analogy, you could call them twins. But we get some more variety in the next step:


Initialize
Assess
Done?
Mutate
Select
Breed

Speaker notes

mutation. So how do we mutate a Traveling Salesman route? It's actually pretty easy:

```ruby
def mutate()
  i1 = rand(CITIES.length)
  i2 = rand(CITIES.length)
  stops[i1], stops[i2] = [stops[i2], stops[i1]]
end
```

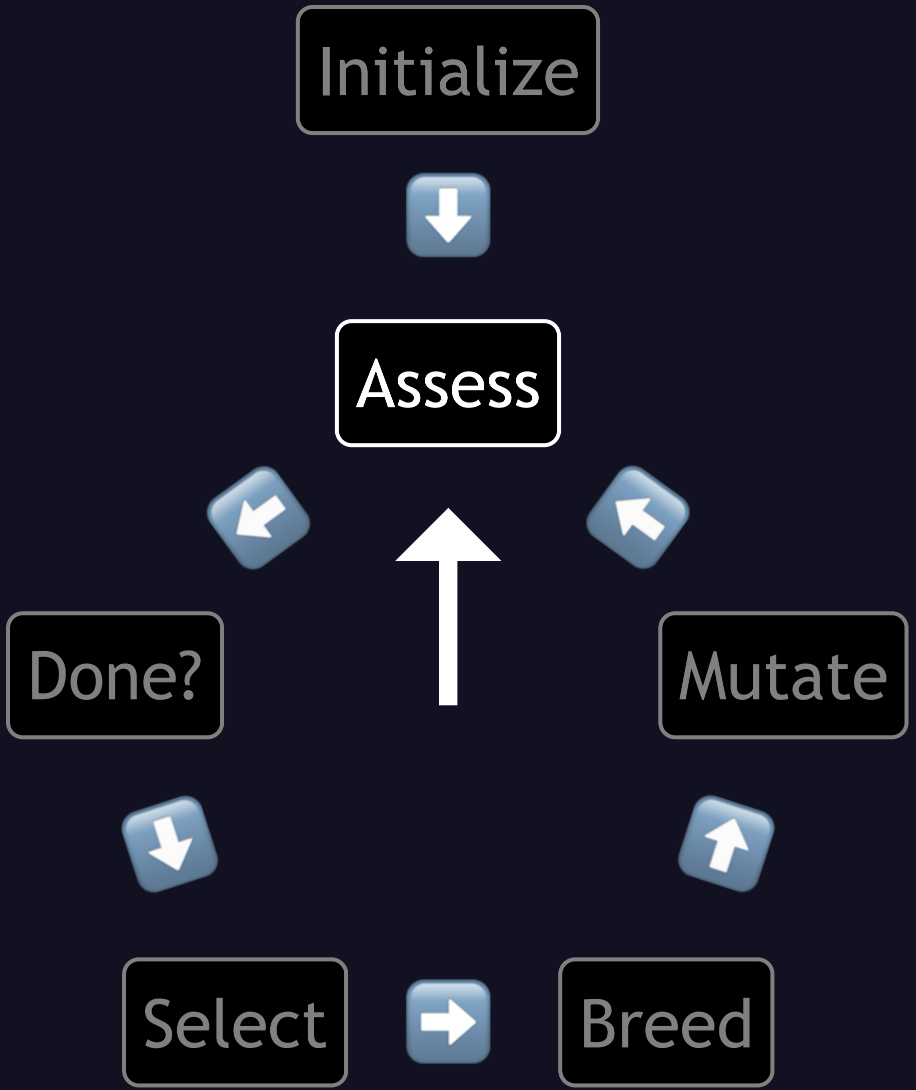
Speaker notes

we just pick two indices in the array, and swap the cities there. If we wanted to make the code more complex, we could make some cities or indices more or less likely to swap, or require that they be consecutive, or not consecutive, or we could have a probability of multiple mutations, but we're going to keep it simple and just do one swap each. The chance of the two numbers being the same will give us some chance of not really mutating. If we apply this to our current routes, we might wind up with . . .

1st	2nd	3rd	4th	5th	6th	7th	8th
BRU	LUX	MAD	AND	PAR	LIS	AMS	MTC
MAD	LUX	BRU	MTC	PAR	LIS	AMS	AND
MAD	LUX	BRU	AND	PAR	LIS	AMS	MTC
LUX	BRU	AND	MAD	PAR	LIS	AMS	MTC
MAD	LUX	BRU	AMS	AND	PAR	LIS	MTC
LIS	LUX	BRU	AND	PAR	MAD	AMS	MTC
AMS	LUX	BRU	AND	PAR	LIS	MAD	MTC
PAR	LUX	BRU	AND	MAD	LIS	AMS	MTC
LUX	BRU	MTC	MAD	PAR	LIS	AMS	AND
LUX	BRU	AND	PAR	MAD	LIS	AMS	MTC

Speaker notes

. . . this, where the green color means it changed. You can see that there are still many that somewhat resemble others, but there's no more exact duplication. That's not guaranteed though; some twins could undergo identical mutation, or some may mutate into twinship, though we could guarantee it with a much more complicated mutation function. Now that they're in their final forms, we can . . .


Initialize
Assess
Done?
Mutate
Select
Breed

Speaker notes

. . . start our cycle over again by asking how fit they are, and we get . . .

1st	2nd	3rd	4th	5th	6th	7th	8th	Fit
PAR	LUX	BRU	AND	MAD	LIS	AMS	MTC	4420
AMS	LUX	BRU	AND	PAR	LIS	MAD	MTC	4302
LUX	BRU	AND	PAR	MAD	LIS	AMS	MTC	3194
MAD	LUX	BRU	AMS	AND	PAR	LIS	MTC	2831
LUX	BRU	AND	MAD	PAR	LIS	AMS	MTC	2329
BRU	LUX	MAD	AND	PAR	LIS	AMS	MTC	2057
MAD	LUX	BRU	MTC	PAR	LIS	AMS	AND	2027
LUX	BRU	MTC	MAD	PAR	LIS	AMS	AND	1543
MAD	LUX	BRU	AND	PAR	LIS	AMS	MTC	1420
LIS	LUX	BRU	AND	PAR	MAD	AMS	MTC	1329

Speaker notes

. . . this, after sorting by fitness. Just as before, our best fitness went down! But, also just like before, we haven't lost that best one, it's recorded in that best_route class variable. Let's let it run to completion, producing . . .

1st	2nd	3rd	4th	5th	6th	7th	8th	Fit	Gens
LIS	AND	MTC	LUX	PAR	BRU	AMS	MAD	5802	22
LUX	BRU	PAR	MTC	AND	MAD	LIS	AMS	6012	18
LIS	MAD	PAR	AMS	BRU	LUX	MTC	AND	6275	26
AMS	BRU	PAR	LIS	MAD	AND	MTC	LUX	6424	100

Speaker notes

. . . these additional best routes, with those fitnesses, that stay as the best for that many generations. On a map, the final best one, that reigned supreme for a hundred generations, would look like . . .

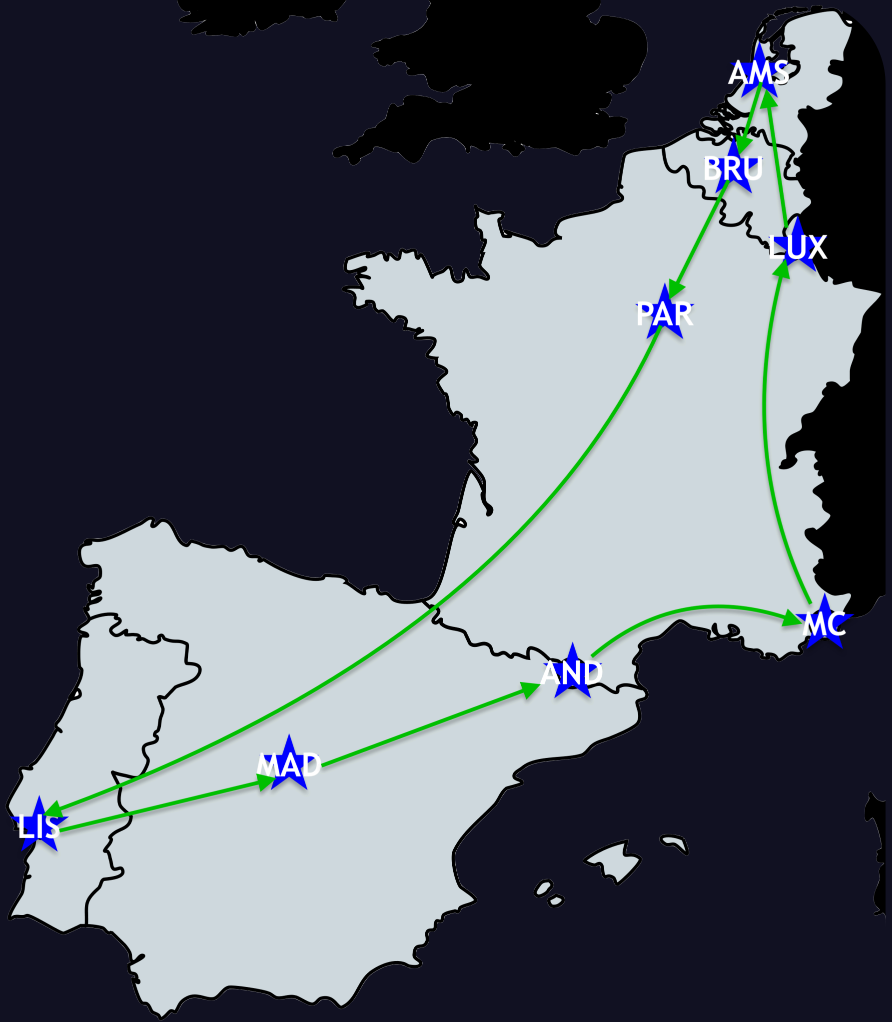


Image: Image: modified from https://www.mapchart.net/europe.html

Speaker notes

. . . this, and we can see it really makes sense, it's roughly what we would have thought of just by eyeballing it.

There are . . .

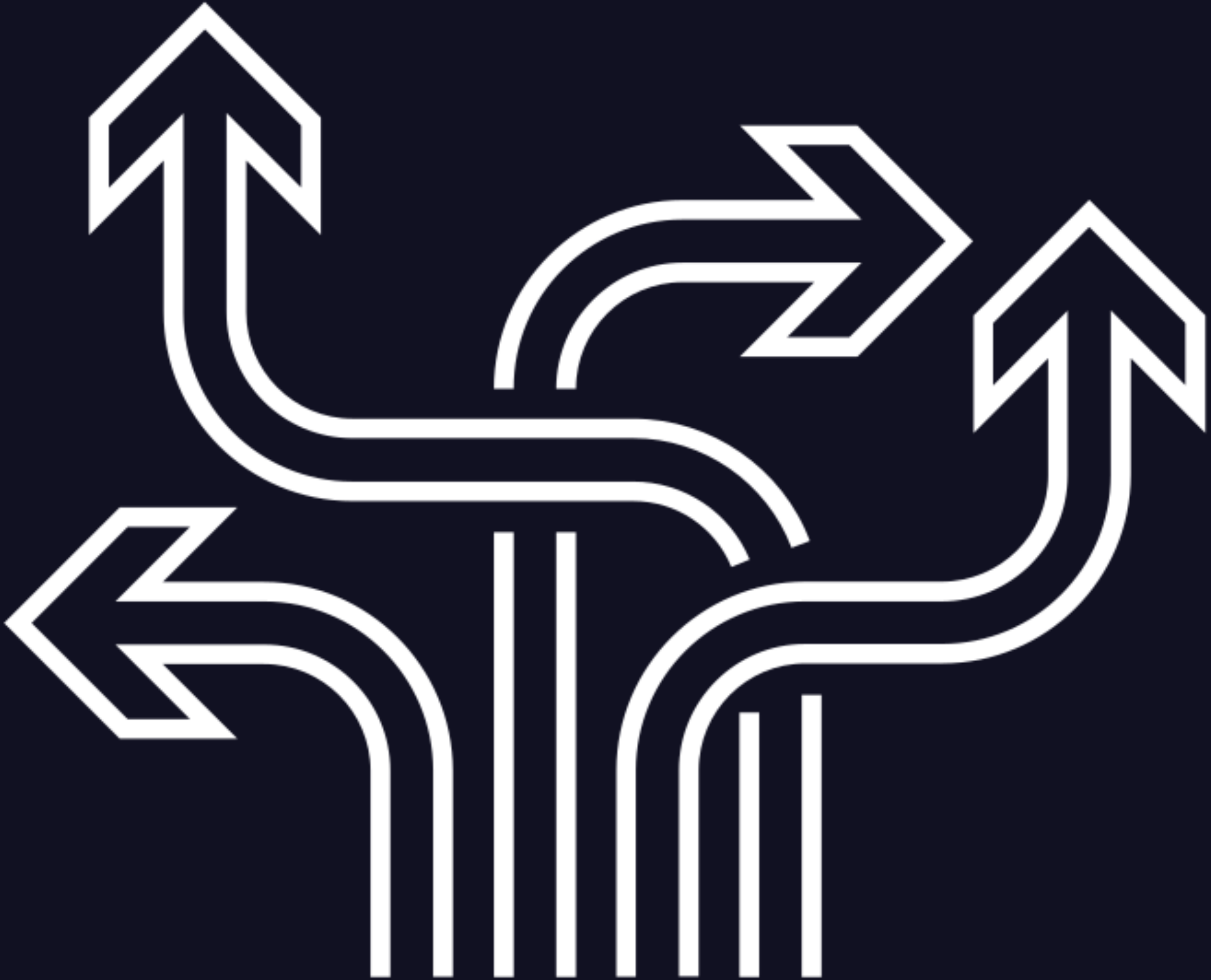


Image: https://commons.wikimedia.org/wiki/File:Noun_76041_-_Different_Ways.svg

Speaker notes

. . . many other ways we can use genetic algorithms. Longer versions of this talk show how to use them to generate Dungeons & Dragons character stats, and recipes for brewing mead. I'm now working on a system to use them to schedule the talks for a conference. Mey Beisaron has a talk on using one to schedule her college classes. They can create images and music, even code. So, think about it, and you might be able to use them for something.

To recap what you've learned here today:

Genetic Algorithms:

- are ~~optimization heuristics~~ shortcuts

Speaker notes

Genetic Algorithms are optimization heuristics, which is fancy-talk for shortcuts to finding good-enough solutions. They're . . .

Genetic Algorithms:

- are ~~optimization heuristics~~ shortcuts
- are simpler than you probably thought

Speaker notes

. . . simpler than you probably thought. They . . .

Genetic Algorithms:

- are ~~optimization heuristics~~ shortcuts
- are simpler than you probably thought
- can use very simple functions

Speaker notes

. . . can use very simple functions, but it . . .

Genetic Algorithms:

- are ~~optimization heuristics~~ shortcuts
- are simpler than you probably thought
- can use very simple functions
- can be tricky to figure out good functions

Speaker notes

. . . can be tricky to figure out exactly what the functions should do, for best results. This approach is also . . .

Genetic Algorithms:

- are ~~optimization heuristics~~ shortcuts
- are simpler than you probably thought
- can use very simple functions
- can be tricky to figure out good functions
- applicable to a wide variety of problems

Speaker notes

. . . applicable to a huge variety of problems, including ones so complex that . . .

Genetic Algorithms:

- are ~~optimization heuristics~~ shortcuts
- are simpler than you probably thought
- can use very simple functions
- can be tricky to figure out good functions
- applicable to a wide variety of problems
- can create solutions humans would not

Speaker notes

. . . a semi-random algorithm can come up with excellent solutions that we humans would never have thought of.

Now, if you have any . . .

? ? ? ? ?

T.Rex-2025@Codosaur.us
twitter.com/DaveAronson
linkedin.com/in/DaveAronson

Repo and Slides:
github.com/CodosaurusLLC/tight-genes
Codosaur.us/reds/gen-algs-dd-eu-25-slides

Speaker notes

. . . questions, I'll take them now, or at the contact info shown up there. As for the other URLs, the Github one is for the code, and slides in HTML, and the other one is for the slides as a PDF, complete with a full script... which I've mostly stuck to. Anyway, any questions?