Essentials of Metaheuristics

A Set of Undergraduate Lecture Notes by

Sean Luke
Department of Computer Science
George Mason University

Second Edition
Online Version 2.2
October, 2015

Figure 0 The Mona Lisa, estimated with the $(5+1)$ Evolution Strategy. The objective is to find a set of fifty polygons which most closely approximates the original image. After Roger Alsing.

 ISBN: 978-1-300-54962-8

Thanks to Carlotta Domeniconi, Kenneth De Jong, John Grefenstette, Christopher Vo, Joseph Harrison, Keith Sullivan, Brian Hrolenok, Bill Langdon, R. Paul Wiegand, Brian Absetz, Jason Branly, Jack Compton, Stephen Donnelly, William Haddon, Beenish Jamil, Eric Kangas, James O'Beirne, Peshal Rupakheti, Nicholas Payette, Lee Spector, "Markus", Don Miner, Brian Ross, Mike Fadock, Ken Oksanen, Asger Ottar Alstrup, Joerg Heitkoetter, Don Sofge, Akhil Shashidhar, Jeff Bassett, Guillermo Calderón-Meza, Hans-Paul Schwefel, Pablo Moscato, Mark Coletti, Yuri Tsoy, Faisal Abidi, Ivan Krasilnikov, Yow Tzu Lim, Uday Kamath, Murilo Pontes, Rasmus Fonseca, Ian Barfield, Forrest Stonedahl, Muhammad Iqbal, Gabriel Catalin Balan, Joseph Zelibor, Daniel Carrera, Maximilian Ernestus, Arcadio Rubio Garcia, Kevin Molloy, Petr Pošík, Brian Olson, Matthew Molineaux, Bill Barksdale, Adam Szkoda, Daniel Rothman, Khaled Ahsan Talukder, Len Matsuyama, Andrew Reeves, Liang Liu, Pier Luca Lanzi, Leidy Patricia Garzon Rodriguez, Michael Feveile Mariboe, Muhammed Alper Cinar, Kevin Andrea, Sam McKay, Nikolaus Hansen, and Vittorio Ziparo.

Get the latest version of this document or suggest improvements here:

http://cs.gmu.edu/∼sean/book/metaheuristics/

Cite this document as:

Sean Luke, 2013, *Essentials of Metaheuristics*, Lulu, second edition, available at http://cs.gmu.edu/∼sean/book/metaheuristics/

Always include the URL, as this book is primarily found online. Do *not* include the online version numbers unless you must, as Citeseer and Google Scholar may treat each (oft-changing) version as a different book.

BibTeX:
```
@Book{ Luke2013Metaheuristics,
  author =    { Sean Luke },
  title =     { Essentials of Metaheuristics},
  edition =   { second },
  year =      { 2013 },
  publisher = { Lulu },
  note =      { Available for free at http://cs.gmu.edu/$\sim$sean/book/metaheuristics/ } }
```


Certain art and text is not mine. Figure 43 is copyright 2008 by Oskar Sigvardsson, and is distributed under the Creative Commons Attribution 3.0 License. Figure 33 is by Wikipedia User "Solkoll" and is in the public domain. The top Mona Lisa (in Figure 0) is from Wikipedia and is in the public domain. The bottom Mona Lisa is mine but is inspired by Roger Alsing's method (see http://rogeralsing.com/2008/12/07/genetic-programming-evolution-of-mona-lisa/). Note to Roger: it's not Genetic Programming. The data in Table 4 is from the *NIST/SEMATECH e-Handbook of Statistical Methods*, http://itl.nist.gov/div898/handbook/ and is in the public domain.

Cover art for the second print edition is a time plot of the paths of particles in Particle Swarm Optimization working their way towards the optimum of the Rastrigin problem.

This document is was produced in part via National Science Foundation grants 0916870 and 1317813.

Contents

List of Algorithms

0 Introduction

This is a set of lecture notes for an undergraduate class on **metaheuristics**. The first version of the notes was written for a course I taught in Spring of 2009. As these are *lecture notes* for an *undergraduate* class on the topic, which is unusual, these notes have certain traits. First, they're informal and contain a number of my own personal biases and misinformation. Second, they are light on theory and examples: they're mostly descriptions of algorithms and handwavy, intuitive explanations about why and where you'd want to use them. Third, they're chock full of algorithms great and small. I think these notes would best serve as a complement to a textbook, but can also stand alone as rapid introduction to the field.

I make **no guarantees whatsoever** about the correctness of the algorithms or text in these notes. Indeed, they're likely to have a lot of errors. *Please* tell me of any errors you find (and correct!). Some complex algorithms have been presented in simplified versions. In those cases I've noted it.

0.1 What is a Metaheuristic?

Metaheuristics is a rather unfortunate[1] term often used to describe a major subfield, indeed the primary subfield, of **stochastic optimization**. Stochastic optimization is the general class of algorithms and techniques which employ some degree of randomness to find optimal (or as optimal as possible) solutions to hard problems. Metaheuristics are the most general of these kinds of algorithms, and are applied to a very wide range of problems.

What kinds of problems? In *Jacobellis v. Ohio* (1964, regarding obscenity), the United States Supreme Court Justice Potter Stewart famously wrote,

> I shall not today attempt further to define the kinds of material I understand to be embraced within that shorthand description; and perhaps I could never succeed in intelligibly doing so. **But I know it when I see it**, and the motion picture involved in this case is not that.

Metaheuristics are applied to *I know it when I see it* problems. They're algorithms used to find answers to problems when you have very little to help you: you don't know beforehand what the optimal solution looks like, you don't know how to go about finding it in a principled way, you have very little heuristic information to go on, and brute-force search is out of the question because the space is too large. *But* if you're given a candidate solution to your problem, you *can* test it and assess how good it is. That is, you know a good one when you see it.

For example: imagine if you're trying to find an optimal set of robot behaviors for a soccer goalie robot. You have a simulator for the robot and can test any given robot behavior set and assign it a quality (you know a good one when you see it). And you've come up with a definition for what robot behavior sets look like in general. But you have no idea what the optimal behavior set is, nor even how to go about finding it.

[1] Ordinarily I'd call the subfield *stochastic optimization*. But that's too general a term; it includes important algorithms like **Markov Chain Monte Carlo (MCMC)** or **Gibbs Sampling**, which are *not* in this category. Metaheuristics has lately been the term of use, but I think it's profoundly misleading and weird. When I hear "metadiscussion" I think: *a discussion about discussions*. Likewise when I hear "metaheuristic" I think: *a heuristic about (or for) heuristics*. That's not at all what these algorithms are about! Perhaps the lesser-used term **black box optimization** would be better, though it too comes with some additional baggage. **Weak methods** is also too broad a term: it doesn't imply stochasticity. Sometimes the term **stochastic search** is used: but search algorithms are meant intended for problems where you are searching *for* something specific, like a Rubik's Cube solution or a way to win a Tic-Tac-Toe game: and either you find the solution or you don't. We're not doing search; we're doing optimization.

The simplest thing you could do in this situation is **Random Search**: just try random behavior sets as long as you have time, and return the best one you discovered. But before you give up and start doing random search, consider the following alternative, known as **Hill-Climbing**. Start with a random behavior set. Then make a small, random modification to it and try the new version. If the new version is better, throw the old one away. Else throw the new version away. Now make another small, random modification to your current version (which ever one you didn't throw away). If this newest version is better, throw away your current version, else throw away the newest version. Repeat as long as you can.

Hill-climbing is a simple metaheuristic algorithm. It exploits a heuristic belief about your space of candidate solutions which is usually true for many problems: that similar solutions tend to behave similarly (and tend to have similar quality), so small modifications will generally result in small, well-behaved changes in quality, allowing us to "climb the hill" of quality up to good solutions. This heuristic belief is one of the **central defining features of metaheuristics**: indeed, nearly all metaheuristics are essentially elaborate combinations of hill-climbing and random search.

The "I know it when I see it" problems tackled by metaheuristics are a subclass of **inverse problems**. An inverse problem is one in which you have a test function f which takes a candidate solution and produces an assessment of it, but in which it's difficult or impossible to construct the inverse function f^{-1} which takes an assessment and returns a candidate solution which would have had that assessment.[2] In our example, our robot simulator and test procedure is f. But what we *really* want is an inverse function f^{-1} which takes an assessment and returns a robot behavior set. That way, if we were lucky, we could plug in the optimal assessment value into f^{-1} and get the optimal robot behavior set.

Optimization methods (such as metheuristics) are designed to overcome inverse problems. But many classic optimization techniques, such as **Gradient Ascent** (Algorithm 1) make strong assumptions about the nature of f: for example, that we also know its first derivative f'. Metaheuristics make far weaker assumptions, and sometimes make none at all. This means that metaheuristics are very general, but also means that they're often best thought of as last-ditch methods, used when *no other known technique works*. As it so happens, that's the case for an enormous, important, and growing collection of problems.

0.2 Algorithms

The lecture notes have a lot of algorithms, great and small. Everything from large evolutionary computation algorithms to things as simple as "how to shuffle an array". Algorithms appear for even the most trivial and obvious of tasks. I strove to be pedantic in case anyone had any questions.

Algorithms in this book are written peculiarly. If an algorithm takes parameters, they will appear first followed by a blank line. If there are no parameters, the algorithm begins immediately. In some cases the algorithm is actually several functions, each labelled **procedure**. Sometimes certain shared, static global variables are defined which appear at the beginning and are labelled **global**. In a few cases, different parts of the algorithm are meant to be performed at different times or rates. In this case you may see some parts labelled, for example, **perform once per generation** or **perform every *n* iterations**, versus **perform each time**. Here is an example of a simple algorithm:

[2]Inverse problems are also notable in that f^{-1} may not be a valid function at all. f^{-1} may be **overspecified**, meaning that there are multiple candidate solutions which all yield a given assessment (which should be returned?). And f^{-1} may also be **underspecified**, meaning that some assessments have *no* candidate solution at all which have them (what should be returned then?).

Algorithm 0 *Bubble Sort*

1: $\vec{v} \leftarrow \langle v_1, ..., v_l \rangle$ vector to sort ▷ User-provided parameters to the algorithm appear here
▷ Then a blank space
2: **repeat** ▷ Algorithm begins here
3: *swapped* ← false ▷ ← always means "is set to"
4: **for** i from 1 to $l - 1$ **do** ▷ Note that l is defined by v_l in Line 1
5: **if** $v_i > v_{i+1}$ **then**
6: Swap v_i and v_{i+1}
7: *swapped* ← true
8: **until** *swapped* = false ▷ = means "is equal to"
9: **return** $\vec{v}$ ▷ Some algorithms return nothing, so there is no **return** statement

Note that the parameters to the function are only loosely specified: and sometimes when we call a function, we don't explicitly state the parameters, if it's obvious what needs to be provided. Yeah, I could have been more formal in a lot of places. So sue me.[3]

0.3 Notation[3]

There's little special here. But just to dot our i's and cross our t's:

- **Numbers** and **booleans** are denoted with lower-case letters, greek symbols, or words (n, λ, *min, popsize*). The default "empty" or **"null" element** is denoted with □. **Ranges** of numbers are often described like this: from 1 to n inclusive. Ranges can be of integers or real values. The symbol ← always means "is set to", and the symbol = usually means "equals".
- **Candidate solutions** (sometimes called **individuals, particles,** or **trails**) are indicated with upper-case letters or words (*Best*, S, P_i). Some candidate solutions are actually vectors and are described like vectors below. Others consist of a number of **components**, often designated $C_1, ..., C_j$. Candidate solutions may be associated with some kind of **quality** (**fitness**), usually via a function like Quality(S) or Fitness(P_i). Quality can be set as well. Usually quality is a single number; but can in some cases (for multiobjective optimization) be a group of numbers called **objectives**. The value of objective O_j, assigned to individual P_i, is accessed via a function like ObjectiveValue(O_j, P_i). In certain cases various other attributes may be assigned to individuals or to other objects.
- **Collections** (or bags, or groups, or pools, or lists, or multisets) are groups of objects where the objects usually don't have to be unique. In fact, the lecture notes rarely use sets, and abundantly use collections. A collection is denoted with a capital letter like P and contains some number of elements in braces $\{P_1, ..., P_n\}$. The size of P is $||P||$ or (in this case) n. Membership (or lack thereof) is indicated with $\in$ or $\notin$. Usually there is an implicit order in a collection, so you can refer to its elements uniquely (P_4 or P_i) and can scan through it like this (**for** each element $P_i \in P$ **do** ... P_i) or this (**for** i from 1 to n **do** ... P_i). Collections are generally read-only, though their elements may allow internal modification.

[3]Don't sue me. Thanks.

[3]This is *always* the most boring part of a book! Why are you reading this?

The union operator ($\cup$) is abused to indicate concatenating collections (like $P \leftarrow P \cup Q$). This is often used to add an element to a collection like $P \leftarrow P \cup \{R_j\}$. The minus sign is abused to indicate removing all elements that are in another collection, as in $P - M$, or removing a specific element from a collection ($P \leftarrow P - \{P_2\}$). In all these cases, presume that the new collection retains the implicit order from the old collection or collections.

The most common collections are the ones used for **populations**. Usually I denote populations with P or Q. Occasionally we need to have a collection of populations, denoted like this: $P^{(1)}, ..., P^{(n)}$. An individual numbered j in population $P^{(i)}$ would be $P^{(i)}_j$.

Sometimes children will be denoted C_a and C_b, etc. This doesn't imply the existence of a collection called C (though it's generally harmless to do so).

- **First-in, First-out Queues** are treated like collections, with the additional ability to add to the end of them and remove elements from the front or from an arbitrary location.
- **Vectors** are denoted with an over-arrow ($\vec{x}, \overrightarrow{Best}$) and contain some number of elements in angle brackets $\langle x_1, ..., x_n \rangle$. Unlike collections, vectors are modifiable. An element in a vector can be replaced with another object at the same location. Slots may not be simply deleted from a vector, but vectors can be extended by adding elements to the end of them. I use vectors instead of collections when we must explicitly change elements in certain locations.
- **Tuples** are vectors with named slots like $\vec{t} \leftarrow \langle t_{\text{lock}}, t_{\text{data}} \rangle$, rather than numbered slots.
- Two-dimensional **Arrays** or **Matrices** are denoted with capital letters (A) and their elements can be referred to in the usual manner: $A_{i,j}$. Like vectors, array elements can be replaced.
- **Probability Distributions** and other **Models** are denoted with a capital letter like T. Distributions and Models are constructed or updated; then we select random numbers under them. Along those lines, variances are denoted with σ^2, standard deviations with σ, and means are often denoted with μ.
- When passed around as data, **Functions** are in lower-case, as in f or $f(node)$.

1 Gradient-based Optimization

Before we get into metaheuristics, let's start with a traditional mathematical method for finding the maximum of a function: **Gradient Ascent**.[4] The idea is to identify the slope and move up it. Simple! The function we're going to maximize is $f(x)$. This method doesn't require us to compute or even know $f(x)$, but it *does* assume we can compute the slope of x, that is, we have $f'(x)$.

The technique is very simple. We start with an arbitrary value for x. We then repeatedly add to it a small portion of its slope, that is, $x \leftarrow x + \alpha f'(x)$, where α is a very small positive value. If the slope is positive, x will increase. If the slope is negative, x will decrease. Figure 1 roughly illustrates this. Ultimately x will move up the function until it is at the peak, at which point the slope is zero and x won't change any more.

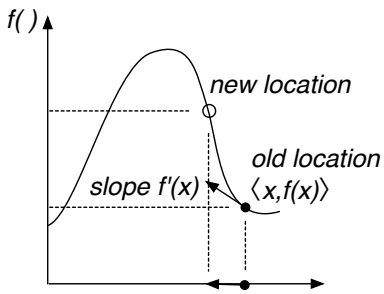

Figure 1 Gradient Ascent with a negative slope. x is decreasing.

We're usually not interested in simple one-dimensional functions like this: more generally, we'd like to find the maximum of a multidimensional function. To do this we replace x with the vector $\vec{x}$, and replace the slope $f'(x)$ with the *gradient* of $\vec{x}$, $\nabla f(\vec{x})$. As a reminder: the gradient is simply a vector where each element is the slope of $\vec{x}$ in that dimension, that is, $\langle \frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, ..., \frac{\partial f}{\partial x_n} \rangle$. So basically we're going up the slope in all dimensions at once. Here's the Gradient Ascent algorithm in its full five-line glory:

Algorithm 1 *Gradient Ascent*

1: $\vec{x} \leftarrow$ random initial vector
2: **repeat**
3: $\quad \vec{x} \leftarrow \vec{x} + \alpha \nabla f(\vec{x})$ ▷ In one dimension: $x \leftarrow x + \alpha f'(x)$
4: **until** $\vec{x}$ is the ideal solution or we have run out of time
5: **return** $\vec{x}$

Note that the algorithm runs until we've found "the ideal solution" or "we have run out of time". How do we know that we've got the ideal solution? Typically when the slope is 0. However there are points besides **maxima** where this is the case: the **minima** of functions (of course) and also **saddle points** such as in Figure 2.

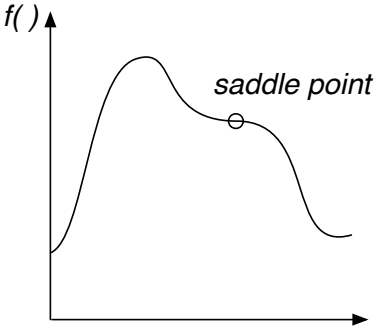

Figure 2 A saddle point.

One issue with Gradient Ascent is **convergence time**. As we get close to the maximum of the function, Gradient Ascent will *overshoot* the top and land on the other side of the hill. It may overshoot the top many times, bouncing back and forth as it moves closer to the maximum. Figure 3 shows this situation.

One of the reasons for this is that the size of the jumps Gradient Ascent makes is entirely based on the current slope. If the slope is very steep the jump will be large even if it's not warranted. One

[4] Actually, the method is usually called **Gradient Descent** because it's used to find the *minimum* of a function. To do that, we just *subtract* the gradient or slope rather than *add* it, that is, Algorithm 1 has its line changed to $\vec{x} \leftarrow \vec{x} - \alpha \nabla f(\vec{x})$. But in our later examples we're always finding maxima, so we're going to be consistent here.

way to deal with this is to tune Gradient Ascent for your problem, by adjusting the value of α. A very small value of α and Gradient Ascent won't overshoot hills but it may take a long time to march up the hills and converge to the top. But a very big value of α will cause Gradient Ascent to constantly overshoot the hills which *also* causes it to take a long time to converge to the maximum, if at all. We're looking for a value of α which is "just right".

We could also modify the algorithm to consider other factors. For example, if we could compute not only $f'(x)$ but also $f''(x)$, we could use **Newton's Method**.[5] This variation on Gradient Ascent includes an additional $\frac{-1}{f''(x)}$ like so: $x \leftarrow x - \alpha\frac{f'(x)}{f''(x)}$. This modification dampens α as we approach a zero slope. Additionally Newton's method no longer converges to just maxima: because of the second derivative, it'll head towards any kind of zero-slope point (maxima, minima, or saddle points).

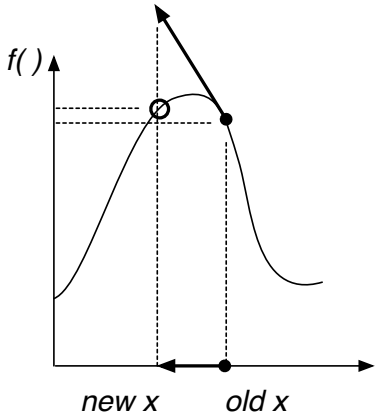

Figure 3 Gradient Ascent overshooting the maximum.

And the multidimensional situation is not as simple as in Gradient Ascent. The multidimensional version of a first derivative $f'(x)$ is the gradient $\nabla f(\vec{x})$. But the multidimensional version of a *second* derivative $f''(x)$ is a complicated matrix called a **Hessian** $H_f(\vec{x})$ consisting of partial second derivatives along each dimension. The Hessian is shown in Figure 4.

To make matters worse, we're *dividing* by the second derivative, which in the multidimensional case involves finding the inverse of this matrix. Overall, the method looks like this:

Algorithm 2 *Newton's Method (Adapted for Optima Finding)*

1: $\vec{x} \leftarrow$ random initial vector
2: **repeat**
3: $\quad \vec{x} \leftarrow \vec{x} - \alpha[H_f(\vec{x})]^{-1}\nabla f(\vec{x})$ ▷ In one dimension: $x \leftarrow x - \alpha\frac{f'(x)}{f''(x)}$
4: **until** $\vec{x}$ is the ideal solution or we have run out of time
5: **return** $\vec{x}$

Because it employs the second derivative, Newton's Method generally converges faster than regular Gradient Ascent, and it also gives us the information to determine if we're at the top of a local maximum (as opposed to a minimum or saddle point) because at a maximum, $f'(x)$ is zero and $f''(x)$ is negative.

But even so, this doesn't get around the *real* problem with these methods: they get caught in **local optima**. Local optima of a function are the optima (in our case, maxima) of a local region. **Global optima** are the optima of the entire function. Figure 5 shows the trace of Gradient Ascent getting caught in a local optimum. Gradient Ascent and Newton's Method are **local optimization algorithms**.

$$H_f(\vec{x}) = \begin{bmatrix} \frac{\partial}{\partial x_1}\frac{\partial f}{\partial x_1} & \frac{\partial}{\partial x_1}\frac{\partial f}{\partial x_2} & \cdots & \frac{\partial}{\partial x_1}\frac{\partial f}{\partial x_n} \\ \frac{\partial}{\partial x_2}\frac{\partial f}{\partial x_1} & \frac{\partial}{\partial x_2}\frac{\partial f}{\partial x_2} & \cdots & \frac{\partial}{\partial x_2}\frac{\partial f}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial}{\partial x_n}\frac{\partial f}{\partial x_1} & \frac{\partial}{\partial x_n}\frac{\partial f}{\partial x_2} & \cdots & \frac{\partial}{\partial x_n}\frac{\partial f}{\partial x_n} \end{bmatrix}$$

Figure 4 The Hessian $H_f(\vec{x})$.

[5] As in *Sir Isaac* Newton, 1642–1727. This method is normally used to find the zeros (roots) of functions, but it's easily modified to hunt for optima, as we've done here.

How do you escape local optima? With the tools we have so far, there's really only one way: change α to a sufficiently large value that the algorithm potentially overshoots not only the top of its hill but actually lands on the *next hill*.

Alternatively, we could put Gradient Ascent in a big loop: each time we start with a random starting point, and end when we've reached a local optimum. We keep trying over and over again, and eventually return the best solution discovered. To determine what the "best solution discovered" is, we need to be able to compute $f(x)$ (something we've not required up till now) so we can compare results. Assuming we have that, we can now construct a **global optimization algorithm**.

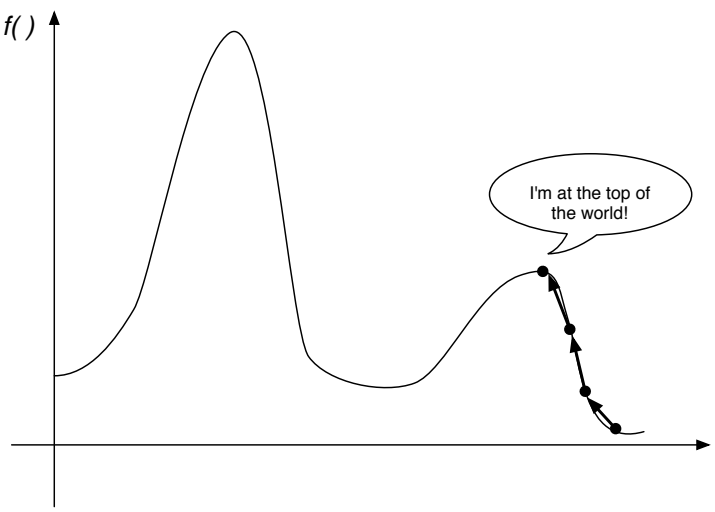

Figure 5 Gradient Ascent stuck in a local optimum.

Algorithm 3 *Gradient Ascent with Restarts*

$\vec{x} \leftarrow$ random initial value
$\vec{x}^* \leftarrow \vec{x}$ ▷ $\vec{x}^*$ will hold our best discovery so far
repeat
  repeat
    $\vec{x} \leftarrow \vec{x} + \alpha \nabla f(\vec{x})$ ▷ In one dimension: $x \leftarrow x + \alpha f'(x)$
  until $||\nabla f(\vec{x})|| = 0$ ▷ In one dimension: **until** $f'(x) = 0$
  if $f(\vec{x}) > f(\vec{x}^*)$ **then** ▷ Found a new best result!
    $\vec{x}^* \leftarrow \vec{x}$
  $\vec{x} \leftarrow$ random value
until we have run out of time
return $\vec{x}^*$

A global optimization algorithm is guaranteed to find the global optimum if it runs long enough. And at the limit, at some point Gradient Ascent with Restarts *will* discover the optimum because, at the very least, some restart will randomly land right on the optimum, just like random search. More realistically, a random restart will eventually start on the globally optimal hill, allowing gradient ascent to climb to the optimum.

Note that we'll likely never wind up with $f'(x)$ *precisely equal to* 0. So we'll have to fudge it: if $-\epsilon < f'(x) < \epsilon$ for some very small value of ϵ, we'll consider that "close enough to zero".[6]

[6]There is a gotcha with the algorithms described here: what happens when part of the function is totally flat? There's no gradient to ascend, which leads to some problems. Let's say you're in a perfectly flat valley (a local minimum) of the function. All around you the slope is 0, so Gradient Ascent won't move at all. It's stuck. Even worse: the second derivative is 0 as well, so for Newton's Method, $\frac{f'(x)}{f''(x)} = \frac{0}{0}$. Eesh. And to top it off, $f'(x) = f''(x) = 0$ for flat minima, flat saddle points, *and* flat maxima. Perhaps adding a bit of randomness might help in some of these situations: but that's for the next section....

2 Single-State Methods

Gradient-based optimization makes a big assumption: that you can compute the first (or even the second) derivative. That's a *big* assumption. If you are optimizing a well-formed, well-understood mathematical function, it's reasonable. But in most cases, you can't compute the gradient of the function *because you don't even know what the function is*. All you have is a way of creating or modifying inputs to the function, testing them, and assessing their quality.

For example, imagine that you have a humanoid robot simulator, and you're trying to find an optimal loop of timed operations to keep the robot walking forward without falling over. You have some n different operations, and your candidate solutions are arbitrary-length strings of these operations. You can plug a string in the simulator and get a quality out (how far the robot moved forward before it fell over). How do you find a good solution?

All you're given is a black box (in this case, the robot simulator) describing a **problem** that you'd like to optimize. The box has a slot where you can submit a **candidate solution** to the problem (here, a string of timed robot operations). Then you press the big red button and out comes the assessed **quality** of that candidate solution. You have no idea what kind of surface the quality assessment function looks like when plotted. Your candidate solution doesn't even have to be a vector of numbers: it could be a graph structure, or a tree, or a set of rules, or a string of robot operations! Whatever is appropriate for the problem.

To optimize a candidate solution in this scenario, you need to be able to do four things:

- Provide one or more initial candidate solutions. This is known as the **initialization procedure**.
- Assess the Quality of a candidate solution. This is known as the **assessment procedure**.
- Make a Copy of a candidate solution.
- Tweak a candidate solution, which produces a *randomly slightly different* candidate solution. This, plus the Copy operation, are collectively known as the **modification procedure**.

To this the algorithm will typically provide a **selection procedure** that decides which candidate solutions to retain and which to reject as it wanders through the space of possible solutions to the problem.

2.1 Hill-Climbing

Let's begin with a simple technique, **Hill-Climbing**. This technique is related to gradient ascent, but it doesn't require you to know the strength of the gradient or even its direction: you just iteratively test new candidate solutions in the region of your current candidate, and adopt the new ones if they're better. This enables you to climb up the hill until you reach a local optimum.

Algorithm 4 *Hill-Climbing*

$S \leftarrow$ some initial candidate solution ▷ The Initialization Procedure
repeat
  $R \leftarrow \text{Tweak}(\text{Copy}(S))$ ▷ The Modification Procedure
  if $\text{Quality}(R) > \text{Quality}(S)$ **then** ▷ The Assessment and Selection Procedures
    $S \leftarrow R$
until S is the ideal solution or we have run out of time
return S

Notice the strong resemblance between Hill-Climbing and Gradient Ascent. The only real difference is that Hill-Climbing's more general Tweak operation must instead rely on a **stochastic** (partially random) approach to hunting around for better candidate solutions. Sometimes it finds worse ones nearby, sometimes it finds better ones.

We can make this algorithm a little more aggressive: create n "tweaks" to a candidate solution all at one time, and then possibly adopt the best one. This modified algorithm is called **Steepest Ascent Hill-Climbing**, because by sampling all around the original candidate solution and then picking the best, we're essentially sampling the gradient and marching straight up it.

Algorithm 5 *Steepest Ascent Hill-Climbing*

```
1: n ← number of tweaks desired to sample the gradient

2: S ← some initial candidate solution
3: repeat
4:     R ← Tweak(Copy(S))
5:     for n − 1 times do
6:         W ← Tweak(Copy(S))
7:         if Quality(W) > Quality(R) then
8:             R ← W
9:     if Quality(R) > Quality(S) then
10:        S ← R
11: until S is the ideal solution or we have run out of time
12: return S
```

A popular variation, which I dub **Steepest Ascent Hill-Climbing with Replacement**, is to not bother comparing R to S: instead, we just replace S directly with R. Of course, this runs the risk of losing our best solution of the run, so we'll augment the algorithm to keep the best-discovered-so-far solution stashed away, in a reserve variable called *Best*. At the end of the run, we return *Best*. In nearly all future algorithms we'll use the store-in-*Best* theme, so get used to seeing it!

Algorithm 6 *Steepest Ascent Hill-Climbing With Replacement*

```
1: n ← number of tweaks desired to sample the gradient

2: S ← some initial candidate solution
3: Best ← S
4: repeat
5:     R ← Tweak(Copy(S))
6:     for n − 1 times do
7:         W ← Tweak(Copy(S))
8:         if Quality(W) > Quality(R) then
9:             R ← W
10:    S ← R
11:    if Quality(S) > Quality(Best) then
12:        Best ← S
13: until Best is the ideal solution or we have run out of time
14: return Best
```

2.1.1 The Meaning of Tweak

The initialization, Copy, Tweak, and (to a lesser extent) fitness assessment functions collectively define the **representation** of your candidate solution. Together they stipulate what your candidate solution is made up of and how it operates.

What might a candidate solution look like? It could be a vector; or an arbitrary-length list of objects; or an unordered set or collection of objects; or a tree; or a graph. Or any combination of these. Whatever seems to be appropriate to your problem. If you can create the four functions above in a reasonable fashion, you're in business.

One simple and common representation for candidate solutions, which we'll stick to for now, is the same as the one used in the gradient methods: a **fixed-length vector of real-valued numbers.** Creating a random such vector is easy: just pick random numbers within your chosen bounds. If the bounds are *min* and *max* inclusive, and the vector length is l, we could do this:

Algorithm 7 *Generate a Random Real-Valued Vector*

1: $min \leftarrow$ minimum desired vector element value
2: $max \leftarrow$ maximum desired vector element value

3: $\vec{v} \leftarrow$ a new vector $\langle v_1, v_2, ...v_l \rangle$
4: **for** i from 1 to l **do**
5: $\quad v_i \leftarrow$ random number chosen uniformly between min and max inclusive
6: **return** $\vec{v}$

To Tweak a vector we might (as one of many possibilities) add a small amount of random noise to each number: in keeping with our present definition of Tweak, let's assume **for now** that this noise is no larger than a small value. Here's a simple way of adding bounded, uniformly distributed random noise to a vector. For each slot in the vector, if a coin-flip of probability p comes up heads, we find some bounded uniform random noise to add to the number in that slot. In most cases we keep $p = 1$.

Algorithm 8 *Bounded Uniform Convolution*

1: $\vec{v} \leftarrow$ vector $\langle v_1, v_2, ...v_l \rangle$ to be convolved
2: $p \leftarrow$ probability of adding noise to an element in the vector ▷ Often $p = 1$
3: $r \leftarrow$ half-range of uniform noise
4: $min \leftarrow$ minimum desired vector element value
5: $max \leftarrow$ maximum desired vector element value

6: **for** i from 1 to l **do**
7: $\quad$ **if** $p \geq$ random number chosen uniformly from 0.0 to 1.0 **then**
8: $\quad\quad$ **repeat**
9: $\quad\quad\quad n \leftarrow$ random number chosen uniformly from $-r$ to r inclusive
10: $\quad\quad$ **until** $min \leq v_i + n \leq max$
11: $\quad\quad v_i \leftarrow v_i + n$
12: **return** $\vec{v}$

We now have a knob we can turn: r, the size of the bound on Tweak. If the size is very small, then Hill-Climbing will march right up a local hill and be unable to make the jump to the next hill because the bound is too small for it to jump that far. Once it's on the top of a hill, everywhere it jumps will be worse than where it is presently, so it stays put. Further, the rate at which it climbs the hill will be bounded by its small size. On the other hand, if the size is large, then Hill-Climbing will bounce around a lot. Importantly, when it is near the top of a hill, it will have a difficult time converging to the peak, as most of its moves will be so large as to overshoot the peak.

Thus small sizes of the bound move slowly and get caught in local optima; and large sizes on the bound bounce around too frenetically and cannot converge rapidly to finesse the very top of peaks. Notice how similar this is to α used in Gradient Ascent. This knob is one way of controlling the degree of **Exploration versus Exploitation** in our Hill-Climber. Optimization algorithms which make largely local improvements are *exploiting* the local gradient, and algorithms which mostly wander about randomly are thought to *explore* the space. As a rule of thumb: you'd *like* to use a highly exploitative algorithm (it's fastest), but the "uglier" the space, the more you will have no choice but to use a more explorative algorithm.

2.2 Single-State Global Optimization Algorithms

A **global optimization algorithm** is one which, if we run it long enough, will *eventually* find the global optimum. Almost always, the way this is done is by guaranteeing that, at the limit, every location in the search space will be visited. The single-state algorithms we've seen so far cannot guarantee this. This is because of our definition (for the moment) of Tweak: to "make a small, bounded, but random change". Tweak wouldn't ever make big changes. If we're stuck in a sufficiently broad local optimum, Tweak may not be strong enough to get us out of it. Thus the algorithms so far have been **local optimization algorithms**.

There are many ways to construct a global optimization algorithm instead. Let's start with the simplest one possible: **Random Search**.

Algorithm 9 *Random Search*

```
1: Best ← some initial random candidate solution
2: repeat
3:     S ← a random candidate solution
4:     if Quality(S) > Quality(Best) then
5:         Best ← S
6: until Best is the ideal solution or we have run out of time
7: return Best
```

Random Search is the extreme in exploration (and global optimization); in contrast, Hill-Climbing (Algorithm 4), with Tweak set to just make very small changes and never make large ones, may be viewed as the extreme in exploitation (and local optimization). But there are ways to achieve reasonable exploitation and still have a global algorithm. Consider the following popular technique, called **Hill-Climbing with Random Restarts**, half-way between the two. We do Hill-Climbing for a certain random amount of time. Then when time is up, we start over with a new random location and do Hill-Climbing again for a different random amount of time.[7] And so on. The algorithm:

[7] Compare to Gradient Ascent with Restarts (Algorithm 3) and consider why we're doing random restarts now rather than gradient-based restarts. How do we know we're on the top of a hill now?

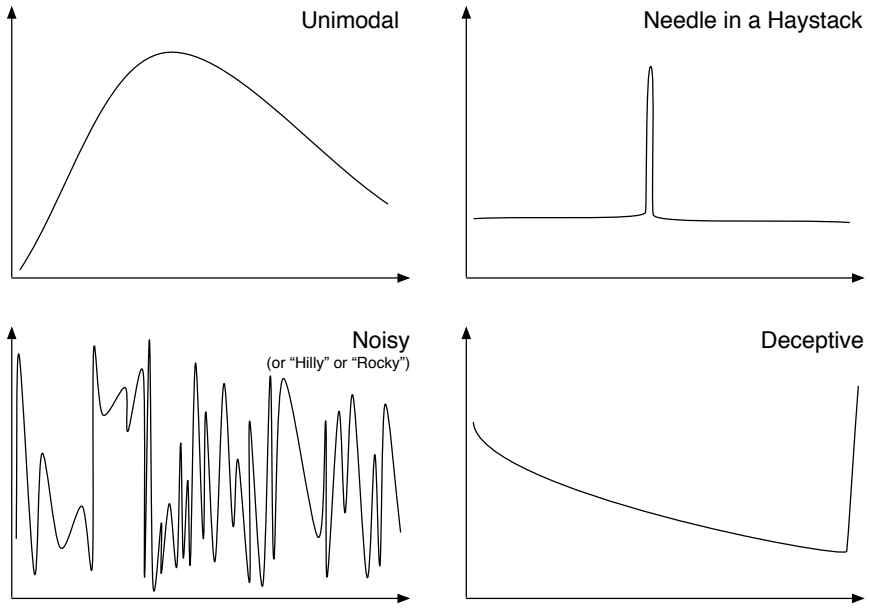

Figure 6 Four example quality functions.

Algorithm 10 *Hill-Climbing with Random Restarts*

```
T ← distribution of possible time intervals

S ← some initial random candidate solution
Best ← S
repeat
    time ← random time in the near future, chosen from T
    repeat
        R ← Tweak(Copy(S))
        if Quality(R) > Quality(S) then
            S ← R
    until S is the ideal solution, or time is up, or we have run out of total time
    if Quality(S) > Quality(Best) then
        Best ← S
    S ← some random candidate solution
until Best is the ideal solution or we have run out of total time
return Best
```

If the randomly-chosen time intervals are generally extremely long, this algorithm is basically one big Hill-Climber. Likewise, if the intervals are very short, we're basically doing random search (by resetting to random new locations each time). Moderate interval lengths run the gamut between the two. That's good, right?

It depends. Consider Figure 6. The first figure, labeled *Unimodal*, is a situation where Hill-Climbing is close to optimal, and where Random Search is a very bad pick. But for the figure labelled *Noisy*, Hill-Climbing is quite bad; and in fact Random Search is expected to be about

as good as you can do (not knowing anything about the functions beforehand). The difference is that in *Unimodal* there is a strong relationship between the distance (along the x axis) of two candidate solutions and their relationship in quality: similar solutions are generally similar in quality, and dissimilar solutions don't have any relationship per se. In the *Noisy* situation, there's no relationship like this: even similar solutions are very dissimilar in quality. This is often known as the **smoothness** criterion for local search to be effective.

This isn't sufficient though. Consider the figure labeled *Needle in a Haystack*, for which Random Search is the only real way to go, and Hill-Climbing is quite poor. What's the difference between this and *Unimodal*? After all, *Needle in a Haystack* is pretty smooth. For local search to be effective there must be an **informative gradient** which generally leads towards the best solutions. In fact, you can make highly *uninformative* gradients for which Hill-Climbing is spectacularly bad! In the figure labeled *Deceptive*, Hill-Climbing not only will not easily find the optimum, but it is actively *let away from the optimum*.

Thus there are some kinds of problems where making small local greedy changes does best; and other problems where making large, almost random changes does best. Global search algorithms run this gamut, and we've seen it before: **Exploration versus Exploitation**. Once again, as a rule of thumb: you'd *like* to use a highly exploitative algorithm (it's fastest), but the "uglier" the space, the more you will have no choice but to use a more explorative algorithm.

Here are some ways to create a global search algorithm, plus approaches to tweaking exploration vs. exploitation within that algorithm:

- **Adjust the Modification procedure** Tweak occasionally makes large, random changes.

  Why this is Global If you run the algorithm long enough, this randomness will cause Tweak to eventually try every possible solution.

  Exploration vs. Exploitation The more large, random changes, the more exploration.

- **Adjust the Selection procedure** Change the algorithm so that you can go down hills at least some of the time.

  Why this is Global If you run the algorithm long enough, you'll go down enough hills that you'll eventually find the right hill to go up.

  Exploration vs. Exploitation The more often you go down hills, the more exploration.

- **Jump to Something New** Every once in a while start from a new location.

  Why this is Global If you try enough new locations, eventually you'll hit a hill which has the highest peak.

  Exploration vs. Exploitation The more frequently you restart, the more exploration.

- **Use a Large Sample** Try many candidate solutions in parallel.

  Why this is Global With enough parallel candidate solutions, one of them is bound to be on the highest peak.

  Exploration vs. Exploitation More parallel candidate solutions, more exploration.

Let's look at some additional global optimizers. We'll focus on what I'm calling *single-state* optimizers which only keep around one candidate solution at a time. That is: no large sample.

2.3 Adjusting the Modification Procedure: (1+1), (1+λ), and (1, λ)

These three oddly named algorithms are forms of our Hill-Climbing procedures with variations of the Tweak operation to guarantee global optimization. They're actually degenerate cases of the more general (μ, λ) and $(\mu + \lambda)$ evolutionary algorithms discussed later (in Section 3.1).

The goal is simple: construct a Tweak operation which *tends* to tweak in small ways but *occasionally* makes larger changes, and *can* make any possible change. We'll mostly hill-climb, but also have the ability to, occasionally, jump far enough to land on other peaks. And there is a chance, however small, that the Hill-Climber will get lucky and Tweak will land right on the optimum.

For example, imagine that we're back to representing solutions in the form of fixed-length vectors of real numbers. Previously our approach to Tweaking vectors was Bounded Uniform Convolution (Algorithm 8). The key word is *bounded*: it required you to choose between being small enough to finesse local peaks and being large enough to escape local optima. But a **Gaussian**[8] (or **Normal**, or bell curve) distribution $N(\mu, \sigma^2)$ lets you do both: usually it makes small numbers but sometimes it makes large numbers. Unless bounded, a Gaussian distribution will *occasionally* make very large numbers indeed. The distribution requires two parameters: the mean μ (usually 0) and variance σ^2. The degree to which we emphasize small numbers over large ones can be controlled by simply changing the variance σ^2 of the distribution.

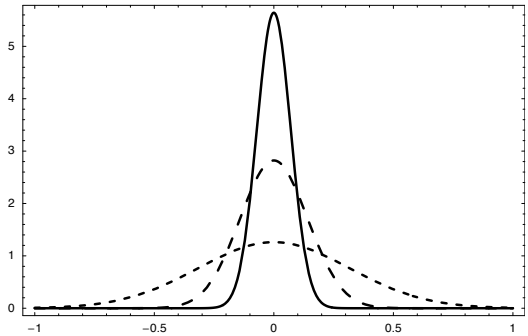

Figure 7 Three Normal or Gaussian distributions $N(\mu, \sigma^2)$ with the mean $\mu = 0$ and the variance σ^2 set to $\sigma^2 = 0.005$: ——, $\sigma^2 = 0.02$: – – –, and $\sigma^2 = 0.1$: - - - -.

We can do this by adding to each number in the vector some random noise under a Gaussian distribution with a mean $\mu = 0$. This is called **Gaussian convolution**.[9] Most noise will be near 0, so the vector values won't change much. But occasional values could be quite large.

Algorithm 11 *Gaussian Convolution*

$\vec{v} \leftarrow$ vector $\langle v_1, v_2, ...v_l \rangle$ to be convolved
$p \leftarrow$ probability of adding noise to an element in the vector ▷ Often $p = 1$
$\sigma^2 \leftarrow$ variance of Normal distribution to convolve with ▷ Normal = Gaussian
$min \leftarrow$ minimum desired vector element value
$max \leftarrow$ maximum desired vector element value

for i from 1 to l **do**
    if $p \geq$ random number chosen uniformly from 0.0 to 1.0 **then**
        repeat
            $n \leftarrow$ random number chosen from the Normal distribution $N(0, \sigma^2)$
        until $min \leq v_i + n \leq max$
        $v_i \leftarrow v_i + n$
return $\vec{v}$

[8] Karl Friedrich Gauss, 1777–1855, kid genius, physicist, and possibly the single most important mathematician ever.

[9] A popular competitor with Gaussian convolution is **polynomial mutation**, from Kalyanmoy Deb and Samir Agrawal, 1999, A niched-penalty approach for constraint handling in genetic algorithms, in *Proceedings of the International Conference on Artificial Neural Networks and Genetic Algorithms*, pages 235–243, Springer. Warning: polynomial mutation has many variants. A popular one is from Kalyanmoy Deb, 2001, *Multi-objective Optimization Using Evolutionary Algorithms*, Wiley.

(1+1) is the name we give to basic Hill-Climbing (Algorithm 4) with this probabilistic-modified Tweak. **(1+λ)** is the name we give to a similarly modified Steepest Ascent Hill-Climbing (Algorithm 5). And **(1, λ)** is the name we give to the modified Steepest Ascent Hill-Climbing with Replacement (Algorithm 6). These names may seem cryptic now but will make more sense later (in Section 3.1).

As it turns out, Gaussian Convolution doesn't give us just one new knob (σ^2) to adjust exploration vs. exploitation, but *two* knobs. Consider the Steepest Ascent Hill-Climbing with Replacement algorithm (Algorithm 6), where the value n specified how many children are generated from the parent candidate solution through Tweak. In the "global" version of this algorithm, $(1, \lambda)$, the value of n interacts with σ^2 in an important way: if σ^2 is large (noisy), then the algorithm will search crazier locations: but a high value of n will aggressively weed out the poor candidates discovered at those locations. This is because if n is low, a poor quality candidate may still be the best of the n examined; but if n is high, this is much less likely. Thus while σ^2 is pushing for more exploration (at the extreme: random search), a high value of n is pushing for more exploitation. n is an example of what will later be called **selection pressure**. Table 1 summarizes this interaction.

		Noise in Tweak	
		Low	High
Samples	Few		Explorative
	Many	Exploitative	↗

Table 1 Simplistic description of the interaction of two factors and their effect on exploration versus exploitation. The factors are: degree of noise in the Tweak operation; and the samples taken before adopting a new candidate solution.

Many random number generators provide facilities for selecting random numbers under Normal (Gaussian) distributions. But if yours doesn't, you can make two Gaussian random numbers at a time using the **Box-Muller-Marsaglia Polar Method**.[10]

Algorithm 12 *Sample from the Gaussian Distribution (Box-Muller-Marsaglia Polar Method)*

$\mu \leftarrow$ desired mean of the Normal distribution ▷ Normal = Gaussian
$\sigma^2 \leftarrow$ desired variance of the Normal distribution

repeat
  $x \leftarrow$ random number chosen uniformly from -1.0 to 1.0
  $y \leftarrow$ random number chosen uniformly from -1.0 to 1.0 ▷ x and y should be independent
  $w \leftarrow x^2 + y^2$
until $0 < w < 1$ ▷ Else we could divide by zero or take the square root of a negative number!
$g \leftarrow \mu + x\sigma\sqrt{-2\frac{\ln w}{w}}$ ▷ It's σ, that is, $\sqrt{\sigma^2}$. Also, note that ln is $\log_e$
$h \leftarrow \mu + y\sigma\sqrt{-2\frac{\ln w}{w}}$ ▷ Likewise.
return g and h ▷ This method generates two random numbers at once. If you like, just use one.

Some random number generators (such as java.util.Random) only provide Gaussian random numbers from the **standard normal distribution** $N(0, 1)$. You can convert these numbers to a Gaussian distribution for any mean μ and variance σ^2 or standard deviation σ you like very simply:

$$N(\mu, \sigma^2) = \mu + \sqrt{\sigma^2}N(0, 1) = \mu + \sigma N(0, 1)$$

[10]The method was first described in George Edward Pelham Box and Mervin Muller, 1958, A note on the generation of random normal deviates, *The Annals of Mathematical Statistics*, 29(2), 610–611. However the polar form of the method, as shown here, is usually ascribed to the mathematician George Marsaglia. There is a faster, but not simpler, method with a great, and apt, name: the **Ziggurat Method**.

2.4 Simulated Annealing

Simulated Annealing was developed by various researchers in the mid 1980s, but it has a famous lineage, being derived from the **Metropolis Algorithm**, developed by the ex-Manhattan Project scientists Nicholas Metropolis, Arianna and Marshall Rosenbluth, and Augusta and Edward Teller in 1953.[11] The algorithm varies from Hill-Climbing (Algorithm 4) in its decision of when to replace S, the original candidate solution, with R, its newly tweaked child. Specifically: if R is better than S, we'll always replace S with R as usual. But if R is worse than S, we may *still* replace S with R with a certain probability $P(t, R, S)$:

$$P(t, R, S) = e^{\frac{\text{Quality}(R) - \text{Quality}(S)}{t}}$$

where $t \geq 0$. That is, the algorithm sometimes goes down hills. This equation is interesting in two ways. Note that the fraction is negative because R is worse than S. First, if R is much worse than S, the fraction is larger, and so the probability is close to 0. If R is very close to S, the probability is close to 1. Thus if R isn't *much* worse than S, we'll still select R with a reasonable probability.

Second, we have a tunable parameter t. If t is close to 0, the fraction is again a large number, and so the probability is close to 0. If t is high, the probability is close to 1. The idea is to initially set t to a high number, which causes the algorithm to move to every newly-created solution regardless of how good it is. We're doing a **random walk** in the space. Then t decreases slowly, eventually to 0, at which point the algorithm is doing nothing more than plain Hill-Climbing.

Algorithm 13 *Simulated Annealing*

$t \leftarrow$ temperature, initially a high number
$S \leftarrow$ some initial candidate solution
Best $\leftarrow S$
repeat
    $R \leftarrow$ Tweak(Copy(S))
    if Quality(R) > Quality(S) or if a random number chosen from 0 to 1 $< e^{\frac{\text{Quality}(R) - \text{Quality}(S)}{t}}$ **then**
        $S \leftarrow R$
    Decrease t
    if Quality(S) > Quality(*Best*) **then**
        Best $\leftarrow S$
until *Best* is the ideal solution, we have run out of time, or $t \leq 0$
return *Best*

The rate at which we decrease t is called the algorithm's **schedule**. The longer we stretch out the schedule, the longer the algorithm resembles a random walk and the more exploration it does.

[11] Nicholas Metropolis, Arianna Rosenbluth, Marshall Rosenbluth, Augusta Teller, and Edward Teller, 1953, Equation of state calculations by fast computing machines, *Journal of Chemical Physics*, 21, 1087–1091. And yes, Arianna and Marshall were married, as were Augusta and Edward. Now *that's* a paper! This gang also developed the **Monte Carlo Method** widely used in simulation. Edward Teller later became a major advocate for nuclear testing and is believed to be one of the inspirations for Dr. Strangelove. To make this Gordion knot even more convoluted, Augusta and Edward's grandson Eric Teller, who goes by Astro Teller, did a fair bit of early work in Genetic Programming (Section 4.3)! Astro also developed the graph-structured Neural Programming: see Footnote 56.

A later paper on Simulated Annealing which established it as a real optimization algorithm is Scott Kirkpatrick, Charles Daniel Gelatt Jr., and Mario Vecchi, 1983, Optimization by simulated annealing, *Science*, 220(4598), 671–680.

Simulated Annealing gets its name from **annealing**, a process of cooling molten metal. If you let metal cool rapidly, its atoms aren't given a chance to settle into a tight lattice and are frozen in a random configuration, resulting in brittle metal. If we decrease the temperature very slowly, the atoms are given enough time to settle into a strong crystal. Not surprisingly, t means **temperature**.

2.5 Tabu Search

Tabu Search, by Fred Glover,[12] employs a different approach to doing exploration: it keeps around a history of recently considered candidate solutions (known as the *tabu list*) and refuses to return to those candidate solutions until they're sufficiently far in the past. Thus if we wander up a hill, we have no choice but to wander back down the other side because we're not permitted to stay at or return to the top of the hill.

The simplest approach to Tabu Search is to maintain a **tabu list** L, of some maximum length l, of candidate solutions we've seen so far. Whenever we adopt a new candidate solution, it goes in the tabu list. If the tabu list is too large, we remove the oldest candidate solution and it's no longer taboo to reconsider. Tabu Search is usually implemented as a variation on Steepest Ascent with Replacement (Algorithm 6). In the version below, we generate n tweaked children, but only consider the ones which aren't presently taboo. This requires a few little subtle checks:

Algorithm 14 *Tabu Search*

$l \leftarrow$ Desired maximum tabu list length
$n \leftarrow$ number of tweaks desired to sample the gradient

$S \leftarrow$ some initial candidate solution
$Best \leftarrow S$
$L \leftarrow \{\}$ a tabu list of maximum length l ▷ Implemented as first in, first-out queue
Enqueue S into L
repeat
    if Length(L) $> l$ **then**
        Remove oldest element from L
    $R \leftarrow$ Tweak(Copy(S))
    for $n - 1$ times **do**
        $W \leftarrow$ Tweak(Copy(S))
        if $W \notin L$ and (Quality(W) $>$ Quality(R) or $R \in L$) **then**
            $R \leftarrow W$
    if $R \notin L$ **then**
        $S \leftarrow R$
        Enqueue R into L
    if Quality(S) $>$ Quality($Best$) **then**
        $Best \leftarrow S$
until $Best$ is the ideal solution or we have run out of time
return $Best$

[12] "Tabu" is an alternate spelling for "taboo". Glover also coined the word "metaheuristics", and developed Scatter Search with Path Relinking (Section 3.3.5). Tabu Search showed up first in Fred Glover, 1986, Future paths for integer programming and links to artificial intelligence, *Computers and Operations Research*, 5, 533–549.

3 Population Methods

Population-based methods differ from the previous methods in that they keep around a *sample* of candidate solutions rather than a single candidate solution. Each of the solutions is involved in tweaking and quality assessment, but what prevents this from being just a parallel hill-climber is that candidate solutions affect *how* other candidates will hill-climb in the quality function. This could happen either by good solutions causing poor solutions to be rejected and new ones created, or by causing them to be Tweaked in the direction of the better solutions.

It may not be surprising that most population-based methods steal concepts from biology. One particularly popular set of techniques, collectively known as **Evolutionary Computation (EC)**, borrows liberally from population biology, genetics, and evolution. An algorithm chosen from this collection is known as an **Evolutionary Algorithm (EA)**. Most EAs may be divided into **generational** algorithms, which update the entire sample once per iteration, and **steady-state** algorithms, which update the sample a few candidate solutions at a time. Common EAs include the **Genetic Algorithm (GA)** and **Evolution Strategies (ES)**; and there are both generational and steady-state versions of each. There are quite a few more alphabet soup subalgorithms.

Because they are inspired by biology, EC methods tend to use (and abuse) terms from genetics and evolution. Because the terms are so prevalent, we'll use them in this and most further sections.

Definition 1 *Common Terms Used in Evolutionary Computation*

individual	a candidate solution
child and parent	a *child* is the Tweaked copy of a candidate solution (its *parent*)
population	set of candidate solutions
fitness	quality
fitness landscape	quality function
fitness assessment or evaluation	computing the fitness of an individual
selection	picking individuals based on their fitness
mutation	plain Tweaking. This is often thought as "asexual" breeding.
recombination or crossover	a special Tweak which takes two parents, swaps sections of them, and (usually) produces two children. This is often thought as "sexual" breeding.
breeding	producing one or more children from a population of parents through an iterated process of selection and Tweaking (typically mutation or recombination)
genotype or genome	an individual's data structure, as used during breeding
chromosome	a genotype in the form of a fixed-length vector
gene	a particular slot position in a chromosome
allele	a particular setting of a gene
phenotype	how the individual operates during fitness assessment
generation	one cycle of fitness assessment, breeding, and population reassembly; or the population produced each such cycle

Evolutionary Computation techniques are generally **resampling techniques**: new samples (populations) are generated or revised based on the results from older ones. In contrast, **Particle Swarm Optimization**, in Section 3.5, is an example of a **directed mutation** method, where candidate solutions in the population are modified, but no resampling occurs per se.

The basic generational evolutionary computation algorithm first constructs an initial population, then iterates through three procedures. First, it **assesses the fitness** of all the individuals in the population. Second, it uses this fitness information to **breed** a new population of children. Third, it **joins** the parents and children in some fashion to form a new next-generation population, and the cycle continues.

Algorithm 17 *An Abstract Generational Evolutionary Algorithm (EA)*

```
1: P ← Build Initial Population
2: Best ← □                                          ▷ □ means "nobody yet"
3: repeat
4:     AssessFitness(P)
5:     for each individual P_i ∈ P do
6:         if Best = □ or Fitness(P_i) > Fitness(Best) then   ▷ Remember, Fitness is just Quality
7:             Best ← P_i
8:     P ← Join(P, Breed(P))
9: until Best is the ideal solution or we have run out of time
10: return Best
```

Notice that, unlike the Single-State methods, we now have a separate AssessFitness function. This is because typically we need all the fitness values of our individuals before we can Breed them. So we have a certain location in the algorithm where their fitnesses are computed.

Evolutionary algorithms differ from one another largely in how they perform the Breed and Join operations. The Breed operation usually has two parts: **Selecting** parents from the old population, then Tweaking them (usually **Mutating** or **Recombining** them in some way) to make children. The Join operation usually either completely replaces the parents with the children, or includes fit parents along with their children to form the next generation.[16]

Population Initialization All the algorithms described here basically use the same initialization procedures, so it's worthwhile giving some tips. Initialization is typically just creating some n individuals at random. However, if you know something about the likely initial "good" regions of the space, you could **bias** the random generation to *tend* to generate individuals in those regions. In fact, you could **seed** the initial population partly with individuals of your own design. Be careful about such techniques: often though you *think* you know where the good areas are, there's a good chance you don't. Don't put all your eggs in one basket: include a significant degree of uniform randomness in your initialization. More on this later on when we talk about representations (in Section 4.1.1).

It's also worthwhile to enforce diversity by guaranteeing that every individual in the initial population is unique. Each time you make a new individual, don't scan through the whole population to see if that individual's already been created: that's $O(n^2)$ and foolish. Instead, create a hash table which stores individuals as keys and anything arbitrary as values. Each time you make an individual, check to see if it's already in the hash table as a key. If it is, throw it away and make another one. Else, add the individual to the population, and hash it in the hash table. That's $O(n)$.

[16] Though it's usually simpler than this, the Join operation can be thought of as kind of selection procedure, choosing from among the children and the parents to form the next generation. This general view of the Join operation is often called **survival selection**, while the selection portion of the Breed operation is often called **parent selection**.

3.1 Evolution Strategies

The family of algorithms known as **Evolution Strategies (ES)** were developed by Ingo Rechenberg and Hans-Paul Schwefel at the Technical University of Berlin in the mid 1960s.[17] ES employ a simple procedure for selecting individuals called **Truncation Selection**, and (usually) only uses mutation as the Tweak operator.

Among the simplest ES algorithms is the **(μ, λ)** algorithm. We begin with a population of (typically) λ number of individuals, generated randomly. We then iterate as follows. First we assess the fitness of all the individuals. Then we delete from the population all but the μ fittest ones (this is all there's to Truncation Selection). Each of the μ fittest individuals gets to produce λ/μ children through an ordinary Mutation. All told we've created λ new children. Our Join operation is simple: the children just replace the parents, who are discarded. The iteration continues anew.

In short, μ is the number of parents which survive, and λ is the number of kids that the μ parents make in total. Notice that λ should be a multiple of μ. ES practitioners usually refer to their algorithm by the choice of μ and λ. For example, if $\mu = 5$ and $\lambda = 20$, then we have a "(5, 20) Evolution Strategy". Here's the algorithm pseudocode:

Algorithm 18 *The (μ, λ) Evolution Strategy*

$\mu \leftarrow$ number of parents selected
$\lambda \leftarrow$ number of children generated by the parents

$P \leftarrow \{\}$
for λ times **do** ▷ Build Initial Population
    $P \leftarrow P \cup \{\text{new random individual}\}$
Best $\leftarrow \square$
repeat
    for each individual $P_i \in P$ **do**
        AssessFitness(P_i)
        if *Best* $= \square$ or Fitness(P_i) $>$ Fitness(*Best*) **then**
            Best $\leftarrow P_i$
    $Q \leftarrow$ the μ individuals in P whose Fitness( ) are greatest ▷ Truncation Selection
    $P \leftarrow \{\}$ ▷ Join is done by just replacing P with the children
    for each individual $Q_j \in Q$ **do**
        for λ/μ times **do**
            $P \leftarrow P \cup \{\text{Mutate}(\text{Copy}(Q_j))\}$
until *Best* is the ideal solution or we have run out of time
return *Best*

Note the use of the function Mutate instead of Tweak. Recall that population-based methods have a variety of ways to perform the Tweak operation. The big two are **mutation**, which is just like the Tweaks we've seen before: convert a single individual into a new individual through a (usually small) random change; and **recombination** or **crossover**, in which multiple (typically two) individuals are mixed and matched to form children. **We'll be using these terms in the algorithms from now on out** to indicate the Tweak performed.

[17] Ingo Rechenberg, 1973, *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*, Fromman-Holzbook, Stuttgart, Germany. In German!

The (μ, λ) algorithm has three knobs with which we may adjust exploration versus exploitation. Figure 8 shows the effect of variations with these operations.

- The size of λ. This essentially controls the sample size for each population, and is basically the same thing as the n variable in Steepest-Ascent Hill Climbing With Replacement. At the extreme, as λ approaches ∞, the algorithm approaches exploration (random search).
- The size of μ. This controls how *selective* the algorithm is; low values of μ with respect to λ push the algorithm more towards exploitative search as only the best individuals survive.
- The degree to which Mutation is performed. If Mutate has a lot of noise, then new children fall far from the tree and are fairly random regardless of the selectivity of μ.

The second Evolution Strategy algorithm is called $\boldsymbol{(\mu + \lambda)}$. It differs from (μ, λ) in only one respect: the Join operation. Recall that in (μ, λ) the parents are simply replaced with the children in the next generation. But in $(\mu + \lambda)$, the next generation consists of the μ parents plus the λ new children.[18] That is, the parents compete with the kids next time around. Thus the next and all successive generations are $\mu + \lambda$ in size. The algorithm looks like this:

Algorithm 19 *The $(\mu + \lambda)$ Evolution Strategy*

$\mu \leftarrow$ number of parents selected
$\lambda \leftarrow$ number of children generated by the parents

$P \leftarrow \{\}$
for λ times **do** ▷ Or perhaps $\lambda + \mu$. See Footnote 18.
    $P \leftarrow P \cup \{$new random individual$\}$
Best $\leftarrow \square$
repeat
    for each individual $P_i \in P$ **do**
        AssessFitness(P_i)
        if *Best* $= \square$ or Fitness(P_i) $>$ Fitness(*Best*) **then**
            Best $\leftarrow P_i$
    $Q \leftarrow$ the μ individuals in P whose Fitness( ) are greatest
    $P \leftarrow Q$ ▷ The Join operation is the only difference with (μ, λ)
    for each individual $Q_j \in Q$ **do**
        for λ / μ times **do**
            $P \leftarrow P \cup \{$Mutate(Copy(Q_j))$\}$
until *Best* is the ideal solution or we have run out of time
return *Best*

The $(\mu + \lambda)$ algorithm may re-evaluate the μ parents along with their λ kids. If your fitness function is deterministic, there's no need to do that: just keep their original fitness values.

[18] It's common to use $\mu + \lambda$ initial individuals rather than the λ I have chosen to use here. I suspect the reasoning behind using $\mu + \lambda$ is that this way every population will have $\mu + \lambda$ individuals. But this has a downside: assuming a deterministic fitness function (so you don't have to reevaluate the μ parents each time), this means the first generation *evaluates* $\mu + \lambda$ individuals but successive generations only evaluate λ individuals, which is both a bit funky, and also inconsistent with the (μ, λ) Evolution Strategy. Pick your poison: it's pretty minor.

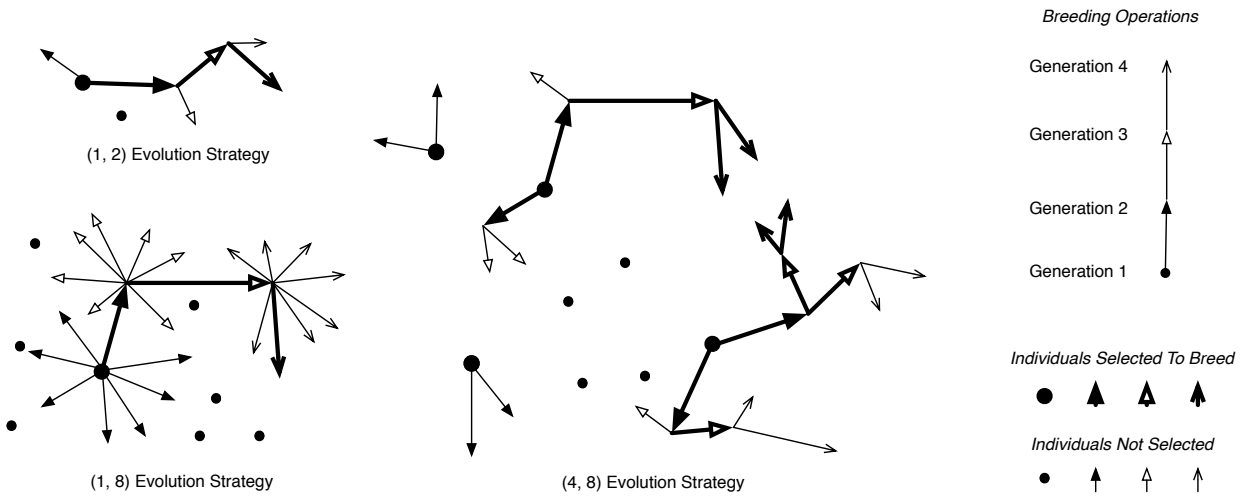

Figure 8 Three (μ, λ) Evolution Strategy variations. Each generation, μ individuals are selected to breed, and each gets to create λ/μ children, resulting in λ children in total.

Generally speaking, $(\mu + \lambda)$ may be more exploitative than (μ, λ) because high-fitness parents persist to compete with the children. This has risks: a sufficiently fit parent may defeat other population members over and over again, eventually causing the entire population to **prematurely converge** to immediate descendants of that parent, at which point the whole population has been trapped in the local optimum surrounding the parent.

If you think about it, $(\mu + \lambda)$ resembles Steepest Ascent Hill-Climbing in that both of them allow the parent to compete against the children for supremacy in the next iteration. Whereas (μ, λ) resembles Steepest Ascent Hill-Climbing with Replacement in that the parents are replaced with the best children. This is more than a coincidence: the hill-climbers are essentially degenerate cases of the ES algorithms. Recall that with the right Tweak operator, plain Hill-Climbing becomes the $(1 + 1)$ algorithm, Steepest Ascent Hill-Climbing with Replacement becomes $(1, \lambda)$, and Steepest Ascent Hill-Climbing becomes $(1 + \lambda)$. Armed with the explanation of the algorithms above, it should be a bit clearer why this is.

3.1.1 Mutation and Evolutionary Programming

Evolution Strategies historically employ a representation in the form of a fixed-length vector of real-valued numbers. Typically such vectors are initialized using something along the lines of Algorithm 7. Mutation is typically performed using Gassian Convolution (Algorithm 11).

Gaussian Convolution is controlled largely by the distribution variance σ^2. The value of σ^2 is known as the **mutation rate** of an ES, and determines the noise in the Mutate operation. How do you pick a value for σ^2? You might pre-select its value; or perhaps you might slowly decrease the value; or you could try to adaptively change σ^2 based on the current statistics of the system. If the system seems to be too exploitative, you could increase σ^2 to force some more exploration (or likewise decrease it to produce more exploitation). This notion of changing σ^2 is known as an **adaptive mutation rate**. In general, such **adaptive** breeding operators adjust themselves over time, in response to statistics gleaned from the optimization run.[19]

[19] Evolution Strategies have also long been associated with **self-adaptive operators** which are stochastically optimized along with individuals. For example, individuals might contain their own mutation procedures which can themselves be mutated along with the individual.

One old rule for changing σ^2 adaptively is known as the **One-Fifth Rule**, by Ingo Rechenberg,[20] and it goes like this:

- If more than $\frac{1}{5}$ children are fitter than their parents, then we're exploiting local optima too much, and we should increase σ^2.
- If less than $\frac{1}{5}$ children are fitter than their parents, then we're exploring too much, and we should decrease σ^2.
- If exactly $\frac{1}{5}$ children are fitter than their parents, don't change anything.

This rule was derived from the results of experiments with the $(1+1)$ ES on certain simple test problems. It may not be optimal for more complex situations: but it's a good starting point.

You don't have to do ES just with vectors. In fact, a little earlier than ES, an almost identical approach was developed by Larry Fogel at the National Science Foundation (Washington DC) and later developed in San Diego.[21] The technique, called **Evolutionary Programming (EP)**, differs from ES in two respects. First, it historically only used a $(\mu+\lambda)$ strategy with $\mu = \lambda$. That is, half the population was eliminated, and that half was then filled in with children. Second, EP was applied to most any representation. From the very start Fogel was interested in evolving graph structures (specifically finite state automata, hence the "programming"). Thus the Mutate operation took the form of adding or deleting an edge, adding or deleting a node, relabeling an edge or a node, etc.

Such operations are reasonable as long as they have two features. First, to guarantee that the algorithm remains global, we must guarantee that, with some small probability, a parent can produce *any* child. Second, we ought to retain the feature that *usually* we make *small* changes likely to not deviate significantly in fitness; and only occasionally make *large* changes to the individual. The degree to which we tend to make small changes could be adjustable, like σ^2 was. We'll get to such representational issues for candidate solutions in detail in Section 4.

3.2 The Genetic Algorithm

The **Genetic Algorithm (GA)**, often referred to as *genetic algorithms*, was invented by John Holland at the University of Michigan in the 1970s.[22] It is similar to the (μ, λ) Evolution Strategy in many respects: it iterates through fitness assessment, selection and breeding, and population reassembly. The primary difference is in how selection and breeding take place: whereas Evolution Strategies select all the parents and *then* create all the children, the Genetic Algorithm little-by-little selects a few parents and generates a few children until enough children have been created.

To breed, we begin with an empty population of children. We then select two parents from the original population, copy them, cross them over with one another, and mutate the results. This forms two children, which we then add to the child population. We repeat this process until the child population is entirely filled. Here's the algorithm in pseudocode.

[20] Also in his evolution strategies text (see Footnote 17, p. 33).

[21] Larry Fogel's dissertation was undoubtedly the first such thesis, if not the first major work, in the field of evolutionary computation. Lawrence Fogel, 1964, *On the Organization of Intellect*, Ph.D. thesis, University of California, Los Angeles.

[22] Holland's book is one of the more famous in the field: John Holland, 1975, *Adaptation in Natural and Artificial Systems*, University of Michigan Press.

Algorithm 20 *The Genetic Algorithm (GA)*

popsize ← desired population size ▷ This is basically λ. Make it even.
$P \leftarrow \{\}$
for *popsize* times **do**
    $P \leftarrow P \cup \{\text{new random individual}\}$
Best ← □
repeat
    for each individual $P_i \in P$ **do**
        AssessFitness(P_i)
        if *Best* = □ or Fitness(P_i) > Fitness(*Best*) **then**
            Best ← P_i
    $Q \leftarrow \{\}$ ▷ Here's where we begin to deviate from (μ, λ)
    for *popsize*/2 times **do**
        Parent $P_a \leftarrow$ SelectWithReplacement(P)
        Parent $P_b \leftarrow$ SelectWithReplacement(P)
        Children $C_a, C_b \leftarrow$ Crossover(Copy(P_a), Copy(P_b))
        $Q \leftarrow Q \cup \{\text{Mutate}(C_a), \text{Mutate}(C_b)\}$
    $P \leftarrow Q$ ▷ End of deviation
until *Best* is the ideal solution or we have run out of time
return *Best*

Though it can be applied to any kind of vector (and indeed many representations) the GA classically operated over fixed-length vectors of *boolean values*, just like ES often were applied to ones of *floating-point values*. For a moment, let's be pedantic about generation of new individuals. If the individual is a vector of floating-point values, creating a new random vector could be done just like in ES (that is, via Algorithm 7). If our representation is a boolean vector, we could do this:

Algorithm 21 *Generate a Random Bit-Vector*

$\vec{v} \leftarrow$ a new vector $\langle v_1, v_2, ...v_l \rangle$
for i from 1 to l **do**
    if 0.5 > a random number chosen uniformly between 0.0 and 1.0 inclusive **then**
        $v_i \leftarrow$ true
    else
        $v_i \leftarrow$ false
return $\vec{v}$

3.2.1 Crossover and Mutation

Note how similar the Genetic Algorithm is to (μ, λ), except during the breeding phase. To perform breeding, we need two new functions we've not seen before: SelectWithReplacement and Crossover; plus of course Mutate. We'll start with Mutate. Mutating a real-valued vector could be done with Gaussian Convolution (Algorithm 11). How might you Mutate a boolean vector? One simple way is **bit-flip mutation**: march down the vector, and flip a coin of a certain probability (often $1/l$, where l is the length of the vector). Each time the coin comes up heads, flip the bit:

Algorithm 22 *Bit-Flip Mutation*

$p \leftarrow$ probability of flipping a bit ▷ Often p is set to $1/l$
$\vec{v} \leftarrow$ boolean vector $\langle v_1, v_2, ...v_l \rangle$ to be mutated

for i from 1 to l **do**
    if $p \geq$ random number chosen uniformly from 0.0 to 1.0 inclusive **then**
        $v_i \leftarrow \neg(v_i)$
return $\vec{v}$

Crossover is the Genetic Algorithm's distinguishing feature.[23] It involves mixing and matching parts of two parents to form children. How you do that mixing and matching depends on the representation of the individuals. There are three classic ways of doing crossover in vectors: **One-Point**, **Two-Point**, and **Uniform Crossover**.

Let's say the vector is of length l. One-point crossover picks a number c between 1 and l, inclusive, and swaps all the indexes $< c$, as shown in Figure 9. The algorithm:

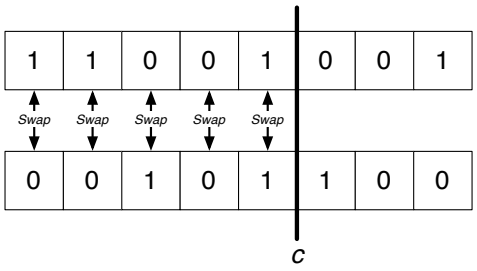

Figure 9 One-Point Crossover.

Algorithm 23 *One-Point Crossover*

$\vec{v} \leftarrow$ first vector $\langle v_1, v_2, ...v_l \rangle$ to be crossed over
$\vec{w} \leftarrow$ second vector $\langle w_1, w_2, ...w_l \rangle$ to be crossed over

$c \leftarrow$ random integer chosen uniformly from 1 to l inclusive
if $c \neq 1$ **then**
    for i from c to l **do**
        Swap the values of v_i and w_i
return $\vec{v}$ and $\vec{w}$

If $c = 1$ no crossover happens. This empty crossover occurs with $\frac{1}{l}$ probability. If you'd like to instead control this probability, you can pick c from between 2 to l inclusive and decide on your own when crossover will occur.

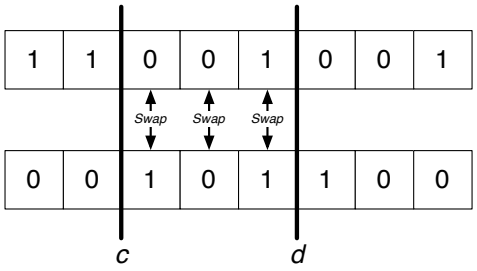

Figure 10 Two-Point Crossover.

The problem with one-point crossover lies in the possible **linkage** (also called **epistasis**) among the elements in the vector. Notice that the probability is high that v_1 and v_l will be broken up due to crossover, as almost any choice of c will do it. Similarly, the probability that v_1 and v_2 will be broken up is quite small, as c must be equal to 2. If the organization of your vector was such that elements v_1 and v_l had to work well in tandem in order to get a high fitness, you'd be constantly breaking up good pairs that the system discovered. Two-point crossover is one way to clean up the linkage problem: just pick *two* numbers c and d, and swap the indexes between them. Figure 10 gives the general idea, and the pseudocode is below:

[23] Though it's long since been used in various ways with Evolution Strategies as well.

Algorithm 24 *Two-Point Crossover*

$\vec{v} \leftarrow$ first vector $\langle v_1, v_2, ...v_l \rangle$ to be crossed over
$\vec{w} \leftarrow$ second vector $\langle w_1, w_2, ...w_l \rangle$ to be crossed over

$c \leftarrow$ random integer chosen uniformly from 1 to l inclusive
$d \leftarrow$ random integer chosen uniformly from 1 to l inclusive
if $c > d$ **then**
    Swap c and d
if $c \neq d$ **then**
    for i from c to $d - 1$ **do**
        Swap the values of v_i and w_i
return $\vec{v}$ and $\vec{w}$

As was the case for one-point crossover, when $c = d$ you get an empty crossover (with $\frac{1}{l}$ probability). If you'd like to control the probability of this yourself, just force d to be different from c, and decide on your own when crossover happens.

It's not immediately obvious two-point crossover would help things. But think of the vectors not as vectors but as *rings* (that is, v_l is right next to v_1). Two-point crossover breaks the rings at two spots and trades pieces. Since v_l is right next to v_1, the only way they'd break up is if c or d sliced right between them. The same situation as v_1 and v_2.[24]

Even so, there's still a further linkage problem. v_1 and v_l are now being treated fairly, but how about v_1 and $v_{l/2}$? Long distances like that are still more likely to be broken up than short distances like v_1 and v_2 (or indeed v_1 and v_l). We can treat all genes fairly with respect to linkage by crossing over each point independently of one another, using Uniform crossover. Here we simply march down the vectors, and swap individual indexes if a coin toss comes up heads with probability p.[25]

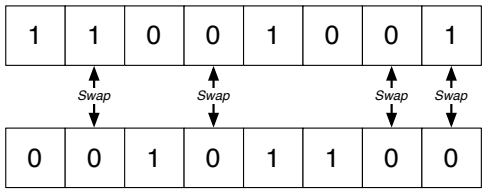

Figure 11 Uniform Crossover.

Algorithm 25 *Uniform Crossover*

$p \leftarrow$ probability of swapping an index ▷ Often p is set to $1/l$. At any rate, $p \leq 0.5$
$\vec{v} \leftarrow$ first vector $\langle v_1, v_2, ...v_l \rangle$ to be crossed over
$\vec{w} \leftarrow$ second vector $\langle w_1, w_2, ...w_l \rangle$ to be crossed over

for i from 1 to l **do**
    if $p \geq$ random number chosen uniformly from 0.0 to 1.0 inclusive **then**
        Swap the values of v_i and w_i
return $\vec{v}$ and $\vec{w}$

[24] We can generalize two-point crossover into a **Multi-Point Crossover**: pick n random points and sort them smallest first: $c_1, c_2, ..., c_n$. Now swap indexes in the region between c_1 and c_2, and between c_3 and c_4, and likewise c_5 and c_6, etc.

[25] The original uniform crossover assumed $p = 1/2$, and was first proposed in David Ackley, 1987, *A Connectionist Machine for Genetic Hillclimbing*, Kluwer Academic Publishers. The more general form, for arbitrary p, is sometimes called *parameterized* uniform crossover.

Crossover is not a global mutation. If you cross over two vectors you can't get every conceivable vector out of it. Imagine your vectors were points in space. Now imagine the hypercube formed with those points at its extreme corners. For example, if your vectors were 3-dimensional, they'd form the corners of a cube (or box) in space, as shown in Figure 12. All the crossovers so far are very constrained: they will result in new vectors which lie at some other corner of the hypercube.

By extension, imagine an entire population P as points in space (such as the three-dimensional space in Figure 12). Crossover done on P can only produce children inside the bounding box surrounding P in space. Thus $P's$ bounding box can never increase: you're doomed to only search inside it.

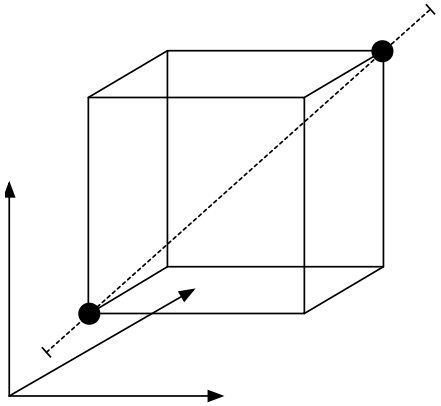

Figure 12 A box in space formed by two three-dimensional vectors (black circles). The dashed line connects the two vectors.

As we repeatedly perform crossover and selection on a population, it may reach the situation where certain alleles (values for certain positions in the vector) have been eliminated, and the bounding box will collapse in that dimension. Eventually the population will **converge**, and often (unfortunately) **prematurely converge**, to copies of the same individual. At this stage there's no escape: when an individual crosses over with itself, nothing new is generated.[26] Thus to make the Genetic Algorithm global, you also need to have a Mutate operation.

What's the point of crossover then? Crossover was originally based on the premise that highly fit individuals often share certain traits, called **building blocks**, in common. For fixed-length vector individuals a building block was often defined as a collection of genes set to certain alleles. For example, in the boolean individual 10110101, perhaps ***101*1 might be a building block (where the * positions aren't part of the building block). In many problems for which crossover was helpful, the fitness of a given individual is often at least partly correlated to the degree to which it contains various of these building blocks, and so crossover works by spreading building blocks quickly throughout the population. Building blocks were the focus of much early genetic algorithm analysis, formalized in an area known as **schema theory**.

That's the idea anyway. But, hand-in-hand with this building-block hypothesis, Crossover methods also assume that there is some degree of **linkage**[27] between genes on the chromosome: that is, settings for certain genes in groups are strongly correlated to fitness improvement. For example, genes A and B might contribute to fitness only when they're *both* set to 1: if either is set to 0, then the fact that the other is set to 1 doesn't do anything. One- and Two-point Crossover also make the even more tenuous assumption that your vector is structured such that highly linked genes are located near to one another on the vector: because such crossovers are unlikely to break apart closely-located gene groups. Unless you have carefully organized your vector, this assumption is probably a bug, not a feature. Uniform Crossover also makes some linkage assumptions but does not have this linkage-location bias. Is the general linkage assumption true for your problem? Or are your genes essentially independent of one another? For most problems of interest, it's the former: but it's dicey. Be careful.

[26] Crossovers which don't make anything new when an individual crosses over with itself are called **homologous**.

[27] One special kind of linkage effect has its own term stolen straight from biology: **epistasis**. Here, genes A and B are linked because gene B has an *effect* on the expression of gene A (on the other hand, A may not affect B). The term "epistasis" can also be used more generally as a synonym for linkage.

In theory, you could perform uniform crossover with several vectors at once to produce children which are the combination of all of them.[28] To avoid sheer randomization, probably you'd want only a bit of mixing to occur, so the probability of swapping any given index shouldn't be spectacularly high. Something like this is very rare in practice though. To do it, we first need to define how to uniformly randomly shuffle a vector. Surprisingly, it's not as obvious as you'd think.

Algorithm 26 *Randomly Shuffle a Vector*

$\vec{p} \leftarrow$ elements to shuffle $\langle p_1, ..., p_l \rangle$

for i from l down to 2 **do** ▷ Note we don't go to 1
    $j \leftarrow$ integer chosen at random from 1 to i inclusive
    Swap p_i and p_j

Armed with a random shuffler (we'll use it in future algorithms too), we can now cross over k vectors at a time, trading pieces with one another, and producing k children as a result.

Algorithm 27 *Uniform Crossover among K Vectors*

$p \leftarrow$ probability of swapping an index ▷ Ought to be very small
$W \leftarrow \{W_1, ..., W_k\}$ vectors to cross over, each of length l

$\vec{v} \leftarrow$ vector $\langle v_1, ..., v_k \rangle$
for i from 1 to l **do**
    if $p \geq$ random number chosen uniformly from 0.0 to 1.0 inclusive **then**
        for j from 1 to k **do** ▷ Load $\vec{v}$ with the ith elements from each vector in W
            $\vec{w} \leftarrow W_j$
            $v_j \leftarrow w_i$
        Randomly Shuffle $\vec{v}$
        for j from 1 to k **do** ▷ Put back the elements, all mixed up
            $\vec{w} \leftarrow W_j$
            $w_i \leftarrow v_j$
            $W_j \leftarrow \vec{w}$
return W

3.2.2 More Recombination

So far we've been doing crossovers that are just swaps: but if the vectors are of floating-point values, our recombination could be something fuzzier, like averaging the two values rather than swapping them. Imagine if our two vectors were points in space. We draw a line between the two points and choose two new points between them. We could extend this line somewhat beyond the points as well, as shown in the dashed line in Figure 12, and pick along the line. This algorithm, known as **Line Recombination**, here presented in the form given by Heinz Mühlenbein and Dirk Schlierkamp-Voosen, depends on a variable p which determines how far out along the line we'll allow children to be. If $p = 0$ then the children will be located along the line within the hypercube (that is, between the two points). If $p > 0$ then the children may be located anywhere on the line, even somewhat outside of the hypercube.

[28] There's nothing new under the sun: this was one of the early ES approaches tried by Hans-Paul Schwefel.

Algorithm 28 *Line Recombination*

$p \leftarrow$ positive value which determines how far long the line a child can be located ▷ Try 0.25
$\vec{v} \leftarrow$ first vector $\langle v_1, v_2, ...v_l \rangle$ to be crossed over
$\vec{w} \leftarrow$ second vector $\langle w_1, w_2, ...w_l \rangle$ to be crossed over

$\alpha \leftarrow$ random value from $-p$ to $1 + p$ inclusive
$\beta \leftarrow$ random value from $-p$ to $1 + p$ inclusive
for i from 1 to l **do**
    $t \leftarrow \alpha v_i + (1 - \alpha) w_i$
    $s \leftarrow \beta w_i + (1 - \beta) v_i$
    if t and s are within bounds **then**
        $v_i \leftarrow t$
        $w_i \leftarrow s$
return $\vec{v}$ and $\vec{w}$

We could extend this further by picking random α and β values for each position in the vector. This would result in children that are located within the hypercube or (if $p > 0$) slightly outside of it. Mühlenbein and Schlierkamp-Voosen call this **Intermediate Recombination**.[29]

Algorithm 29 *Intermediate Recombination*

$p \leftarrow$ positive value which determines how far long the line a child can be located ▷ Try 0.25
$\vec{v} \leftarrow$ first vector $\langle v_1, v_2, ...v_l \rangle$ to be crossed over
$\vec{w} \leftarrow$ second vector $\langle w_1, w_2, ...w_l \rangle$ to be crossed over

for i from 1 to l **do**
    repeat
        $\alpha \leftarrow$ random value from $-p$ to $1 + p$ inclusive ▷ We just moved these two lines!
        $\beta \leftarrow$ random value from $-p$ to $1 + p$ inclusive
        $t \leftarrow \alpha v_i + (1 - \alpha) w_i$
        $s \leftarrow \beta w_i + (1 - \beta) v_i$
    until t and s are within bounds
    $v_i \leftarrow t$
    $w_i \leftarrow s$
return $\vec{v}$ and $\vec{w}$

Since we're using different values of α and β for each element, instead of rejecting recombination if the elements go out of bounds, we can now just repeatedly pick a new α and β.

Why bother with values of $p > 0$? Imagine that you have no Mutate operation, and are just doing Intermediate or Line Recombination. Each time you select parents and generate a child,

[29] Okay, they called them *Extended* Line and *Extended* Intermediate Recombination, in Heinz Mühlenbein and Dirk Schlierkamp-Voosen, 1993, Predictive models for the breeder genetic algorithm: I. continuous parameter optimization, *Evolutionary Computation*, 1(1). These methods have long been in evolutionary computation, but the terms are hardly standardized: notably Hans-Paul Schwefel's original Evolutionary Strategies work used (among others) line recombination with $p = -0.5$, but he called it *intermediate recombination*, as do others. Schwefel also tried a different variation: for each gene of the child, two parents were chosen at random, and their gene values at that gene were averaged.

that child is located somewhere within the cube formed by the parents (recall Figure 12). Thus it's impossible to generate a child *outside the bounding box of the population*. If you want to explore in those unknown regions, you need a way to generate children further out than your parents are.

Other Representations So far we've focused on vectors. In Section 4 we'll get to other representations. For now, remember that if you can come up with a reasonable notion of Mutate, any representation is plausible. How might we do graph structures? Sets? Arbitrary-length lists? Trees?

3.2.3 Selection

In Evolution Strategies, we just lopped off all but the μ best individuals, a procedure known as **Truncation Selection**. Because the Genetic Algorithm performs iterative selection, crossover, and mutation while breeding, we have more options. Unlike Truncation Selection, the GA's SelectWithReplacement procedure can (by chance) pick certain Individuals over and over again, and it also can (by chance) occasionally select some low-fitness Individuals. In an ES an individual is the parent of a fixed and predefined number of children, but not so in a GA.

The original SelectWithReplacement technique for GAs was called **Fitness-Proportionate Selection**, sometimes known as **Roulette Selection**. In this algorithm, we select individuals in proportion to their fitness: if an individual has a higher fitness, it's selected more often.[30] To do this we "size" the individuals according to their fitness as shown in Figure 13.[31] Let $s = \sum_i f_i$ be the sum fitness of all the individuals. A random number from 0 to s falls within the range of some individual, which is then selected.

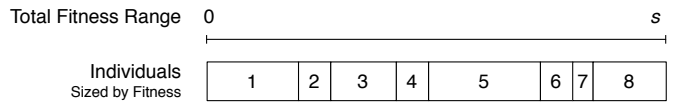

Figure 13 Array of individual ranges in Fitness Proportionate Selection.

Algorithm 30 *Fitness-Proportionate Selection*

perform once per generation
  global $\vec{p} \leftarrow$ population copied into a vector of individuals $\langle p_1, p_2, ..., p_l \rangle$
  global $\vec{f} \leftarrow \langle f_1, f_2, ..., f_l \rangle$ fitnesses of individuals in $\vec{p}$ in the same order as $\vec{p}$ ▷ Must all be ≥ 0
  if $\vec{f}$ is all 0.0s **then** ▷ Deal with all 0 fitnesses gracefully
    Convert $\vec{f}$ to all 1.0s
  for i from 2 to l **do** ▷ Convert $\vec{f}$ to a CDF. This will also cause $f_l = s$, the sum of fitnesses
    $f_i \leftarrow f_i + f_{i-1}$
perform each time
  $n \leftarrow$ random number from 0 to f_l inclusive
  for i from 2 to l **do** ▷ This could be done more efficiently with binary search
    if $f_{i-1} < n \leq f_i$ **then**
      return p_i
  return p_1

[30] We presume here that fitnesses are ≥ 0. As usual, higher is better.
[31] Also due to John Holland. See Footnote 22, p. 36.

Notice that Fitness-Proportionate Selection has a preprocessing step: converting all the fitnesses (or really copies of them) into a cumulative distribution. This only needs to be done once per generation. Additionally, though the code I provided searches linearly through the fitness array to find the one we want, it'd be smarter to do that in $O(\lg n)$ time by doing a binary search instead.

One variant on Fitness-Proportionate Selection is called **Stochastic Universal Sampling** (or **SUS**), by James Baker. In SUS, we select in a fitness-proportionate way but biased so that fit individuals always get picked at least once. This is known as a *low variance resampling* algorithm and I include it here because it is now popular in other venues than just evolutionary computation (most famously, **Particle Filters**).[32]

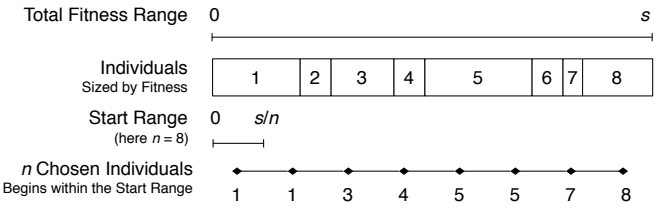

Figure 14 Array of individual ranges, start range, and chosen points in Stochastic Universal Sampling.

SUS selects n individuals at a time (typically n is the size of the next generation, so in our case $n = l$). To begin, we build our fitness array as before. Then we select a random position from 0 to s/n. We then select the individual which straddles that position. We then increment the position by s/n and repeat (up to n times total). Each increment, we select the individual in whose fitness region we landed. This is shown in Figure 14. The algorithm is:

Algorithm 31 *Stochastic Universal Sampling*

perform once per n individuals produced ▷ Usually $n = l$, that is, once per generation
  global $\vec{p} \leftarrow$ copy of vector of individuals (our population) $\langle p_1, p_2, ..., p_l \rangle$, shuffled randomly
    ▷ To shuffle a vector randomly, see Algorithm 26
  global $\vec{f} \leftarrow \langle f_1, f_2, ..., f_l \rangle$ fitnesses of individuals in $\vec{p}$ in the same order as $\vec{p}$ ▷ Must all be ≥ 0
  global $index \leftarrow 0$
  if $\vec{f}$ is all 0.0s **then**
    Convert $\vec{f}$ to all 1.0s
  for i from 2 to l **do** ▷ Convert $\vec{f}$ to a CDF. This will also cause $f_l = s$, the sum of fitnesses.
    $f_i \leftarrow f_i + f_{i-1}$
  global $value \leftarrow$ random number from 0 to f_l/n inclusive
perform each time
  while $f_{index} < value$ **do**
    $index \leftarrow index + 1$
  $value \leftarrow value + f_l/n$
  return p_{index}

There are basically two advantages to SUS. First, it's $O(n)$ to select n individuals, rather than $O(n \lg n)$ for Fitness-Proportionate Selection. That used to be a big deal but it isn't any more, since the lion's share of time in most optimization algorithms is spent in assessing the fitness of individuals, not in the selection or breeding processes. Second and more interesting, SUS

[32] And they never seem to cite him. Here it is: James Edward Baker, 1987, Reducing bias and inefficiency in the selection algorithm, in John Grefenstette, editor, *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms (ICGA)*, pages 14–21, Lawrence Erlbaum Associates, Hillsdale.

guarantees that if an individual is fairly fit (over s/n in size), it'll get chosen for sure, sometimes multiple times. In Fitness-Proportionate Selection even the fittest individual may never be selected.

There is a big problem with the methods described so far: they presume that the actual fitness *value* of an individual really means something important. But often we choose a fitness function such that higher ones are "better" than smaller ones, and don't mean to imply anything more. Even if the fitness function was carefully chosen, consider the following situation, where a fitness function goes from 0 to 10. Near the end of a run, all the individuals have values like 9.97, 9.98, 9.99, etc. We want to finesse the peak of the fitness function, and so we *want* to pick the 9.99-fitness individual. But to Fitness-Proportionate Selection (and to SUS), all these individuals will be selected with nearly identical probability. The system has converged to just doing random selection.

To fix this we could **scale** the fitness function to be more sensitive to the values at the top end of the function. But to really remedy the situation we need to adopt a **non-parametric** selection algorithm which throws away the notion that fitness values mean anything other than bigger is better, and just considers their rank ordering. Truncation Selection does this, but the most popular technique by far is **Tournament Selection**,[33] an astonishingly simple algorithm:

Algorithm 32 *Tournament Selection*

1: $P \leftarrow$ population
2: $t \leftarrow$ tournament size, $t \geq 1$

3: *Best* $\leftarrow$ individual picked at random from P with replacement
4: **for** i from 2 to t **do**
5: *Next* $\leftarrow$ individual picked at random from P with replacement
6: **if** Fitness(*Next*) $>$ Fitness(*Best*) **then**
7: *Best* $\leftarrow$ *Next*
8: **return** *Best*

We return the fittest individual of some t individuals picked at random, with replacement, from the population. That's it! Tournament Selection has become the **primary selection technique used for the Genetic Algorithm** and many related methods, for several reasons. First, it's not sensitive to the particulars of the fitness function. Second, it's dead simple, requires no preprocessing, and works well with parallel algorithms. Third, it's tunable: by setting the **tournament size** t, you can change how selective the technique is. At the extremes, if $t = 1$, this is just random search. If t is very large (much larger than the population size itself), then the probability that the fittest individual in the population will appear in the tournament approaches 1.0, and so Tournament Selection just picks the fittest individual each time (put another way, it approaches Truncation Selection with $\mu = 1$).

In the Genetic Algorithm, the most popular setting is $t = 2$. For certain representations (such as those in Genetic Programming, discussed later in Section 4.3), it's common to be more selective ($t = 7$). To be *less* selective than $t = 2$, but not be totally random, we'd need some kind of trick. One way I do it is to also allow real-numbered values of t from 1.0 to 2.0. In this range, with probability $t - 1.0$, we do a tournament selection of size $t = 2$, else we select an individual at random ($t = 1$).[34]

[33] Tournament Selection may be a folk algorithm: but the earliest usage I'm aware of is Anne Brindle, 1981, *Genetic Algorithms for Function Optimization*, Ph.D. thesis, University of Alberta. She used binary tournament selection ($t = 2$).

[34] You could generalize this to any real-valued $t \geq 1.0$: with probability $t - \lfloor t \rfloor$ select with size $\lceil t \rceil$, else with size $\lfloor t \rfloor$.

nodes and edges were beneficial to program. NP would then make it more likely that less desirable nodes or edges were more likely to be swapped out via crossover, or to be mutated.

We've not even gotten to how to make sure that your particular graph constraint needs (no self-loops, no multiple edges, etc.) are kept consistent over crossover or mutation. What a mess. As a representation, graphs usually involve an awful lot of ad-hoc hacks and domain specificity. The complete opposite of vectors.

4.3 Trees and Genetic Programming

Genetic Programming (GP) is a research community more than a technique per se. The community focuses on how to use stochastic methods to search for and optimize small *computer programs* or other computational devices. Note that to *optimize* a computer program, we must allow for the notion of *suboptimal* programs rather than programs which are simply right or wrong.[57] GP is thus generally interested in the space where there are *lots* of possible programs (usually small ones) but it's not clear which ones outperform the others and to what degree. For example, finding team soccer robot behaviors, or fitting arbitrary mathematical equations to data sets, or finding finite-state automata which match a given language.

Because computer programs are variable in size, the representations used by this community are also variable in size, mostly **lists** and **trees**. In GP, such lists and trees are typically formed from basic functions or CPU operations (like + or *if* or *kick-towards-goal*). Some of these operations cannot be performed in the context of other operations. For example, $4 +$ *kick-towards-goal()* makes no sense unless *kick-towards-goal* returns a number. In a similar vein, certain nodes may be restricted to having a certain *number* of children: for example, if a node is *matrix-multiply*, it might be expecting exactly two children, representing the matrices to multiply together. For this reason, GP's initialization and Tweaking operators are particularly concerned with maintaining **closure**, that is, producing *valid* individuals from previous ones.

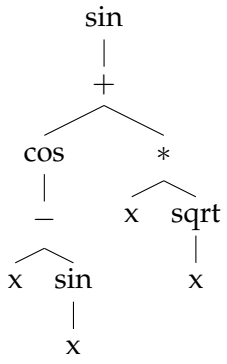

Figure 18 A Symbolic Regression parse tree.

One of the nifty things about optimizing computer programs is how you assess their fitness: run them and see how they do! This means that the *data* used to store the genotypes of the individuals might be made to conveniently correspond to the *code* of the phenotypes when run. It's not surprising that the early implementations of GP all employed a language in which code and data were closely related: Lisp.

The most common form of GP employs **trees** as its representation, and was first proposed by Nichael Cramer,[58] but much of the work discussed here was invented by John Koza, to whom a lot of credit is due.[59]

[57] John Koza proposed exactly this notion in his book *Genetic Programming*: "...you probably assumed I was talking about writing a *correct* computer program to solve this problem.... In fact, this book, focuses almost entirely on *incorrect* programs. In particular, I want to develop the notion that there are gradations in performance among computer programs. Some incorrect programs are very poor; some are better than others; some are approximately correct; occasionally, one may be 100% correct." (p. 130 of John R. Koza, 1992, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, MIT Press.)

[58] In a single paper Cramer proposed both tree-based GP and a list-based GP similar to that discussed in Section 4.4. He called the list-based version the JB Language, and the tree-based version the TB Language. Nichael Lynn Cramer, 1985, A representation for the adaptive generation of simple sequential programs, in John J. Grefenstette, editor, *Proceedings of an International Conference on Genetic Algorithms and the Applications*, pages 183–187.

[59] Except as noted, the material in Section 4.3 is all due to John Koza. For the primary work, see Footnote 57.

Consider the tree in Figure 18, containing the mathematical expression $\sin(\cos(x - \sin x) + x\sqrt{x})$. This is the *parse tree* of a simple program which performs this expression. In a parse tree, a node is a function or if statement etc., and the children of a node are the arguments to that function. If we used only functions and no operators (for example, using a function subtract(x, y) instead of $x - y$), we might write this in pseudo-C-ish syntax such as:

```
sin(
    add(
        cos(subtract(x, sin(x))),
        multiply(x, sqrt(x))));
```

The Lisp family of languages is particularly adept at this. In Lisp, the function names are tucked *inside* the parentheses, and commas are removed, so the function foo(bar, baz(quux)) appears as (foo bar (baz quux)). In Lisp objects of the form (...) are actually singly-linked lists, so Lisp can manipulate code as if it were data. Perfect for tree-based GP. In Lisp, Figure 18 is:

```
(sin
    (+
        (cos (- x (sin x)))
        (* x (sqrt x))))
```

How might we evaluate the fitness of the individual in Figure 18? Perhaps this expression is meant to fit some data as closely as possible. Let's say the data is twenty pairs of the form $\langle x_i, f(x_i)\rangle$. We could test this tree against a given pair i by setting the return value of the x operator to be x_i, then executing the tree, getting the value v_i it evaluates to, and computing the some squared error from $f(x_i)$, that is, $\delta_i = (v_i - f(x_i))^2$. The fitness of an individual might be the square root of the total error, $\sqrt{\delta_1 + \delta_2 + ... + \delta_n}$. The family of GP problems like this, where the objective is to fit an arbitrarily complex curve to a set of data, is called **symbolic regression**.

Programs don't have to be equations: they can actually *do* things rather than simply *return values*. An example is the tree shown in Figure 19, which represents a short program to move an ant about a field strewn with food. The operator if-food-ahead takes two children, the one to evaluate if there is food straight ahead, and the one to evaluate if there isn't. The do operator takes two children and evaluates the left one, then the right one. The left and right operators turn the ant 90° to the left or right, forward moves the ant forward one step, consuming any food directly in front. Given a grid strewn with food, the objective is to find a program which, when executed (perhaps multiple times), eats as much food as possible. The fitness is simply the amount of food eaten. This is actually a common test problem called the **artificial ant**.

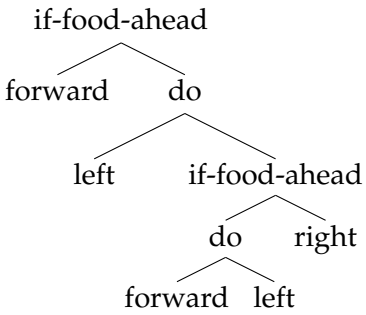

Figure 19 An Artificial Ant tree.

The code for Figure 19 in a pseudo-Lisp and C would look something like:

Pseudo-Lisp:

```
(if-food-ahead
    forward
    (do
        left
        (if-food-ahead
            (do forward left)
            right)))
```

Pseudo-C:

```
if (foodAhead) forward();
else {
    left();
    if (foodAhead) {
        forward();
        left();
    }
    else right();
}
```

Tree-style GP can be used with any optimization algorithm of course. But for no particular reason it has its own traditional algorithm, which was described earlier in Section 3.3.3.

4.3.1 Initialization

GP builds new trees by repeatedly selecting from a **function set** (the collection of items which may appear as nodes in a tree) and stringing them together. In the Artificial Ant example, the function set might consist of if-food-ahead, do, forward, left, and right. In the Symbolic Regression example, the function set might consist of $+$, $-$, $*$, sin, cos, sqrt, x, and various other mathematical operators. Note that the functions in the function set each have an **arity**, meaning, a pre-defined number of children. sin takes one child. do and $+$ take two children. x and forward take no children. Nodes with a zero arity (taking no children) are considered to be *leaf nodes* in the function set, and nodes with an arity ≥ 1 are *nonleaf nodes*. Algorithms which string nodes together generally need to respect these conventions in order to build a valid tree.

One common algorithm is the **Grow** algorithm, which builds random trees depth-first up to a certain depth:

Algorithm 53 *The Grow Algorithm*

```
1: max ← maximum legal depth
2: FunctionSet ← function set

3: return DoGrow(1, max, FunctionSet)

4: procedure DoGrow(depth, max, FunctionSet)
5:     if depth ≥ max then
6:         return Copy(a randomly-chosen leaf node from FunctionSet)
7:     else
8:         n ← Copy(a randomly-chosen node from the FunctionSet)
9:         l ← number of child nodes expected for n
10:        for i from 1 to l do
11:            Child i of n ← DoGrow(depth + 1, max, FunctionSet)
12:        return n
```

The **Full** algorithm is a slight modification of the Grow algorithm which forces full trees up to the maximum depth. It only differs in a single line:

Algorithm 54 *The Full Algorithm*

```
1: max ← maximum legal depth
2: FunctionSet ← function set

3: return DoFull(1, max, FunctionSet)

4: procedure DoFull(depth, max, FunctionSet)
5:     if depth ≥ max then
6:         return Copy(a randomly-chosen leaf node from FunctionSet)
7:     else
8:         n ← Copy(a randomly-chosen non-leaf node from FunctionSet)      ▷ The only difference!
9:         l ← number of child nodes expected for n
10:        for i from 1 to l do
11:            Child i of n ← DoFull(depth + 1, max, FunctionSet)
12:        return n
```

GP originally built each new tree by picking either of these algorithms half the time, with a max depth selected randomly from 2 to 6. This procedure was called **Ramped Half-and-Half**.

Algorithm 55 *The Ramped Half-and-Half Algorithm*

```
1: minMax ← minimum allowed maximum depth
2: maxMax ← maximum allowed maximum depth        ▷ ... if that name makes any sense at all...
3: FunctionSet ← function set

4: d ← random integer chosen uniformly from minMax to maxMax inclusive
5: if 0.5 < a random real value chosen uniformly from 0.0 to 1.0 then
6:     return DoGrow(1, d, FunctionSet)
7: else
8:     return DoFull(1, d, FunctionSet)
```

The problem with these algorithms is that they provide no control over the size of the trees: and indeed tend to produce a fairly odd distribution of trees. There are quite a number of algorithms with better control.[60] Here's a one of my own design, **PTC2**,[61] which produces a tree of a desired size, or up to the size plus the maximum number of children to any given nonleaf node. It's easy to describe. We randomly extend the horizon of a tree with nonleaf nodes until the number of nonleaf nodes, plus the remaining spots, is greater than or equal to the desired size. We then populate the remaining slots with leaf nodes:

[60] Liviu Panait and I did a survey of the topic in Sean Luke and Liviu Panait, 2001, A survey and comparison of tree generation algorithms, in Lee Spector, *et al.*, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 81–88, Morgan Kaufmann, San Francisco, California, USA.

[61] PTC2 was proposed in Sean Luke, 2000, Two fast tree-creation algorithms for genetic programming, *IEEE Transactions on Evolutionary Computation*, 4(3), 274–283. It's an obvious enough algorithm that it's been no doubt used many times prior in other computer science contexts.

Algorithm 56 *The PTC2 Algorithm*

$s \leftarrow$ desired tree size
FunctionSet $\leftarrow$ function set

if $s = 1$ **then**
    return Copy(a randomly-chosen leaf node from *FunctionSet*)
else
    $Q \leftarrow \{\ \}$
    $r \leftarrow$ Copy(a randomly-chosen non-leaf node from *FunctionSet*)
    $c \leftarrow 1$
    for each child argument slot b of r **do**
        $Q \leftarrow Q \cup \{b\}$
    while $c + ||Q|| < s$ **do**
        $a \leftarrow$ an argument slot removed at random from Q
        $m \leftarrow$ Copy(a randomly-chosen non-leaf node from *FunctionSet*)
        $c \leftarrow c + 1$
        Fill slot a with m
        for each child argument slot b of m **do**
            $Q \leftarrow Q \cup \{b\}$
    for each argument slot $q \in Q$ **do**
        $m \leftarrow$ Copy(a randomly-chosen leaf node from *FunctionSet*)
        Fill slot q with m
    return r

Ephemeral Random Constants It's often useful to include in the function set a potentially infinite number of constants (like 0.2462 or $\langle 0.9, -2.34, 3.14 \rangle$ or 2924056792 or "s%&e : m") which get sprinkled into your trees. For example, in the Symbolic Regression problem, it might be nice to include in the equations constants such as -2.3129. How can we do this? Well, function sets don't have to be fixed in size if you're careful. Instead you might include in the function set a special node (often a leaf node) called an **ephemeral random constant** (or **ERC**). Whenever an ERC is selected from the function set and inserted into the tree, it automatically transforms itself into a randomly-generated constant of your choosing. From then on, that particular constant never changes its value again (unless mutated by a special mutation operator). Figure 20 shows ERCs inserted into the tree, and Figure 21 shows their conversion to constants.

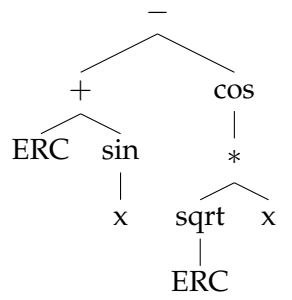

Figure 20 A tree with ERC placeholders inserted. See Figure 21.

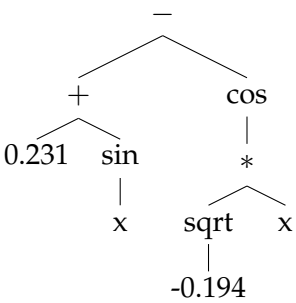

Figure 21 The tree in Figure 20 with ERC placeholders replaced with permanent constants.

4.3.2 Recombination

GP usually does recombination using **subtree crossover**. The idea is straightforward: in each individual, select a random subtree (which can possibly be the root). Then swap those two subtrees. It's common, but

hardly necessary, to select random subtrees by picking leaf nodes 10% of the time and non-leaf nodes 90% of the time. Algorithm 57 shows how select a subtree of a given type.

Algorithm 57 *Subtree Selection*

```
1: r ← root node of tree
2: f(node) ← a function which returns true if the node is of the desired type

3: global c ← 0
4: CountNodes(r, f)
5: if c = 0 then                          ▷ Uh oh, no nodes were of the desired type!
6:     return □                           ▷ "null" or "failure" or something
7: else
8:     a ← random integer from 1 to c inclusive
9:     c ← 0
10:    return PickNode(r, a, f)

11: procedure CountNodes(r, f)            ▷ This is just depth-first search
12:    if f(r) is true then
13:        c ← c + 1
14:    for each child i of r do
15:        CountNodes(i, f)

16: procedure PickNode(r, a, f)           ▷ More depth-first search!
17:    if f(r) is true then
18:        c ← c + 1
19:        if c ≥ a then
20:            return r
21:    for each child i of r do
22:        v ← PickNode(i, a, f)
23:        if v ≠ □ then
24:            return v
25:    return □                           ▷ You shouldn't be able to reach here
```

4.3.3 Mutation

GP doesn't often do mutation, because the crossover operator is **non-homologous**[62] and is highly mutative. Even so, there are many possibilities for mutation. Here are just a few:

- **Subtree mutation**: pick a random subtree and replace it with a randomly-generated subtree using the algorithms above. Commonly Grow is used with a max-depth of 5. Again, leaf nodes are often picked 10% of the time and non-leaf nodes 90% of the time.
- Replace a random non-leaf node with one of its subtrees.

[62] Recall that with homologous crossover, an individual crossing over with itself will just make copies of itself.

- Pick a random non-leaf node and swap its subtrees.
- If nodes in the trees are ephemeral random constants, mutate them with some noise.
- Select two subtrees in the individual such that neither is contained within the other, and swap them with one another.

Again, we can use Algorithm 57 to select subtrees for use in these techniques. Algorithm 57 is called **subtree selection** but it could have just as well been called *node selection*: we're just picking a node. First we count all the nodes of a desired type in the tree: perhaps we want to just select a leaf node for example. Then we pick a random number a less than the number of nodes counted. Then we go back into the tree and do a depth-first traversal, counting off each node of the desired type, until we reach a. That's our node.

4.3.4 Forests and Automatically Defined Functions

Genetic Programming isn't constrained to a single tree: it's perfectly reasonable to have a genotype in the form of a vector of trees (commonly known as a **forest**). For example, I once developed simple soccer robot team programs where an individual was an entire robot team. Each robot program was two trees: a tree called when the robot was far from the ball (it returned a vector indicating where to run), and another tree called when the robot was close enough to a ball to kick it (it would return a vector indicating the direction to kick). The individual consisted of some n of these tree pairs, perhaps one per robot, or one per robot class (goalies, forwards, etc.), or one for every robot to use (a homogeneous team). So a soccer individual might have from 2 to 22 trees!

Trees can also be used to define **automatically defined functions (ADFs)**[63] which can be called by a primary tree. The heuristic here is one of **modularity**. Modularity lets us search very large spaces if we know that good solutions in them are likely to be repetitive: instead of requiring the individual to contain all of the repetitions perfectly (having all its ducks in order)—a very unlikely result—we can make it easier on the individual by breaking the individuals into *modules* with an overarching section of the genotype define how those modules are arranged.

In the case of ADFs, if we notice that ideal solutions are likely to be large trees with often-repeated subtrees within them, we'd prefer that the individual consist of one or two *subfunctions* which are then called repeatedly from a main tree. We do that by adding to the individual a second tree (the ADF) and including special nodes in the original parent tree's function set[64] which are just function calls to that second tree. We can add further ADFs if needed.

For purposes of illustration, let's say that a good GP solution to our problem will likely need to develop a certain subfunction of two arguments. We don't know what it will look like but we believe this to be the case. We could apply this heuristic belief by using a GP individual representation consisting of two trees: the main tree and a two-argument ADF tree called, say, ADF1.

We add a new non-leaf node to its function set: ADF1(*child1*, *child2*). The ADF1 tree can have whatever function set we think is appropriate to let GP build this subelement. But it will need to have two additional leaf-node functions added to the main tree's function set as well. Let's call them ARG1 and ARG2.

[63] Automatically Defined Functions are also due to John Koza, but are found in his second book, John R. Koza, 1994, *Genetic Programming II: Automatic Discovery of Reusable Programs*, MIT Press.

[64] Every tree has its own, possibly unique, function set.