Biologically Inspired Methods: Cheatsheet

Vishnu

Tuesday 30th April, 2019

1 What is Optimisation?

Optimisation is adjusting the inputs to a problem to find an optimal output. Putting this into mathematical terms, given a function $f(x)$ find a value for x that maximises or minimises $f(x)$. This value is normally subject to various constraints ($g(x)$). Most optimisation algorithms deal with minimisation, so to convert a maximisation problem to a minimisation we simply multiply it by -1 .i.e $-1 * f(x)$.

1.1 Local vs. Global Minima

Optimisation methods try to find the **global minima** for a problem: this is the lowest value of $f(x)$ in the search space (the range of values permitted by the constraints). However, while doing so they may find **local minima**, which are the lowest value of $f(x)$ according to their neighbours. A real-world example is the lowest point on earth: a well is lower than its immediate surroundings, but it's very shallow compared to the Mariana Trench. If an optimisation method finds a local minima, it may assume that it is the global minima and stop searching: so algorithms have to take care to avoid this.

1.2 Exploitation vs. Exploration

All guided searching algorithms try to strike a balance between these.
Exploitation is taking known factors and using them to intensify them search: for example, on finding a good potential solution, search the area around it for better solutions. This allows us to follow trends in the search space, but often leads to being stuck in local minima. *Exploration* involves diversifying the search by visiting previously unknown areas and trying different values to discover potentially better solutions. Doing this too much leads to an effectively random search, but in moderation this allows us to identify and break out of local minima.

1.3 Example: Least Squares Problem

$$\text{Minimise } f(x) = \|Ax - B\|_2^2$$
$$\text{Where A, x and B are vectors/matrices, and } \|x\|_p^q = \left(\sqrt[p]{\textstyle\sum_{x\in X} x^p}\right)^q$$

Since we're using squares, we can safely say the minimum value of $f(x)$ is 0: so if we can find $x \Rightarrow f(x) = 0$, we've found the global minima. Using matrix maths:

$$\begin{aligned}
f(x) &= (Ax - B)^T \cdot (Ax - B) = (x^T A^T - B^T) \cdot (Ax - B) \\
&= x^T A^T Ax - x^T A^T B - B^T Ax + B^T B \\
&= x^T A^T Ax - 2 * x^T A^T B + B^T B \; (x^T A^T B = B^T Ax \text{ in scalar values i.e f(x)}) \\
\triangle f(x) &= \frac{\triangle x^T A^T Ax}{\triangle x} - \frac{\triangle 2 * x^T A^T B}{\triangle x} + \frac{\triangle B^T B}{\triangle x} \\
&= x \cdot (A^T A + AA^T) - 2 * A^T B = 2 * (xA^T A - A^T B) \\
&= 0 \text{ at } x = (A^T A)^{-1} A^T B
\end{aligned}$$

Therefore for this problem, rather than using an algorithm we can simply solve for $x = (A^TA)^{-1}A^TB$. For more complex problems (e.g find the best pairing of body features for an insect to survive the Sahara Desert), this isn't quite as simple.

1.4 Example: Linear Programming

Minimise f(x) subject to constraints g(x)
Where f(x) $= C^Tx$, $i \geq 0 \; \forall i \in x$ and constraints are of the form $a^Tx \leq b$

There are efficient analytical methods to solve to solve linear programming problems, but are out of scope.

2 Analytical Optimisation

For any derivable function f(x), we can find the global minima by using f'(x) and f"(x). The extreme points of any functions can be found at f'(x) = 0, and these are local maxima/minima if f"(x) < 0/> 0 respectively. If f"(x) = 0, this is a saddle point and we ignore it. To find the global minima, we can simply solve for every extreme point, identify the minima, and choose the lowest value. e.g.

$$f(x) = 5x^3 + 2x^2 - 3x$$

$$f'(x) = 15x^2 + 4x - 3$$

This is zero at x = $-3/5$ and x = 1/3.

$$f''(x) = 30x + 4$$

f"(-3/) = -14 , f"(1/3) = 14

f"(1/3) > 0, therefore x = 1/3 is the global minima.

2.1 Constrained Optimisation

If there are constraints as well (g(x)=c), we can convert the whole problem to a Lagrange function to be solved:

$$L(x, \lambda) = f(x) + \lambda * (g(x) - c) \tag{1}$$

2.2 Advantages and Disadvantages

Advantages:

- Very quick
- Will give a perfect answer
- Based off existing math (calculus)

Disadvantages:

- Doesn't work with discrete values
- Function must be differentiable
- Every maxima/minima has to be checked
- Doesn't have an easy way to set boundaries

3 Numerical Methods

3.1 Exhaustive Search

This method simply searches the entire space to find the global minima. Obviously, this can only be done on a search space that's finitely large: if a precision isn't given for real numbers, this method can go on infinitely. To reduce the amount of values searched a coarser sample can be taken to find a promising region, then the search refined in that area.

This method can't get stuck in local minima, makes no assumptions on the cost function, and can handle discrete/continuous values, but takes a very long time. It also might miss the global minima if the coarser sample chooses the wrong region.

3.2 Nelder-Mead Downhill Simplex

Unlike later methods, this doesn't need f(x) to be differentiable, but does need continuity. It can be applied to an N-dimensional problem (N is the number of inputs to the problem), but this explanation only covers 2-dimensional (f(x,y)). The algorithm keeps track of 3 points at each stage (the simplex), and adjusts these as required till an acceptable solution has been found. These adjustments are detailed in the following sections, followed by the algorithm flow.

At each stage the 3 points are ranked: B(est), G(ood) and W(orst) respectively. Also calculated are M = (B+G)/2 (the midpoint of the line BG), and R = 2M-W (the reflection of W in BG).

3.2.1 Simplex Flip

Since we've assumed that optimisation problems are minimisation, W will have a higher value than B or G. Therefore, the reflection of W in BG should have a lower value: so the algorithm sets W = R. Flipping follows the gradient of the problem, but doesn't directly use it.

3.2.2 Simplex Extend

If the algorithm would flip, but finds that the new direction is a good one (continuing in it gives further optimisation), it can extend the simplex further in that direction by setting W = 2R-M (effectively 3M-W).

3.2.3 Contraction

If flipping leads to a worse result, that goes against the gradient of the problem: but since it usually follows it, that means that the simplex is near/surrounds a local minima. The algorithm brings W closer to BG by setting it to either (W+M)/2 or (R+M)/2. This either contracts W towards BG, or flips it then contracts it: since it's not sure which side of BG the minima is, the algorithm checks both.

3.2.4 Shrinking

If the point the algorithm would obtain by contraction is worse than leaving it where it is, the algorithm simply shrinks the simplex towards B: it sets W and G to the midpoints of WB and GB respectively.

3.2.5 Overall Flow

```
A. Start the simplex in a random position and rank the points as B(est),G(ood),W(orst)
B. While f(B) $>$ $const_1$ AND step_num $<$ $const_2$ :
    1. Calculate M = (B+G)/2, R = 2M-W
    2. if f(R) < f(G) (is it worth flipping)
        i. if f(R) < f(B) && f(2R-M) < f(B) (is it worth extending)
           EXTEND (W = 2R-M)
        ii. else
           FLIP (W=R)
    3. else
        i. if f(R) < f(W) (which side is better for contraction)
           W=R
        ii. C = (W+M)/2 (contraction point)
        iii. if f(c) < f(w) (does contraction improve the simplex)
           CONTRACT (W=C)
        iv. else
           SHRINK (W = (W+B)/2, G = (G+B)/2)
C. take B as the optimised solution
```

3.3 Gradient Descent

This method requires f(x) to be differentiable, and simply follows the gradient of the search space down to a minima in multiple steps. With X_k being the considered value at step k, h_k being the step size (a scaling factor > 0) and $\nabla f(x)$ being the derivative of f(x):

$$X_{k+1} = X_k - h_k * \nabla f(X_k) \tag{2}$$

The algorithm stops when a set number of generations has passed or $\nabla f(X_k)$ becomes small enough (indicating convergence). Determining h_k is tricky: too small and the search will take a long time, too large and the algorithm may never converge. Gradient Descent has no way to overcome local minima, though a large step size may unintentionally overcome them.

3.4 Line Minimisation

This is similar to Gradient Descent, but h_k is determined rather than being a user constant. The proof is as below:

Expanding the Taylor series at point $f(X_k)$ and ignoring higher order terms:

$$\begin{aligned} f(x) &= f(x_k) + f'(x_k) * (x - x_k) + 1/2 * f''(x_k) * (x - x_k)^2 \\ \frac{f(x) - f(x_k)}{x - x)k} &= f'(x_k) + 1/2 * f''(x_k) * (x - x_k) \\ \frac{d(f(x))}{dx} &= f'(x_k) + 1/2 * f''(x_k) * (x - x_k) \end{aligned}$$

The minima of f(x) must be at $\frac{d(f(x))}{dx} = 0$. On re-arranging the above:
$X = X_k - \frac{f'(x_k)}{f''(x_k)}$

Therefore at each step we can use $X_{k+1} = X_k - \frac{f'(x_k)}{f''(x_k)}$. Comparing this to gradient descent, $h_k = \frac{1}{f''(x_k)}$.
Line Minimisation is robust and the steps are clearly explained/easy to influence, but as it uses a gradient it also gets stuck in local minima and requires a continuous differentiable f(x).

4 Random Optimisation

4.1 Random Walk

This algorithm tracks one point in the N-dimensional space. At each iteration, this algorithm chooses a random direction/step size and compares the result of using it to the current position. If it provides a better result it moves the algorithm's position, if it's worse then there's a chance that it does move. This chance (C) can break the algorithm out of local minima, and can be changed at each generation: if it reduces as the generations increase, the algorithm becomes simulated annealing.

For an N-dimensional problem, D (direction) is an NxN diagonal matrix with values of 1 or -1 and h (step size) is a vector of length N. The two are combined by the following equation:

$$X_{k+1} = X_k + D_k \cdot h_k \tag{3}$$

If X_{k+1} would be outside of the search space, it is either re-done with new random values or set to the closest boundary value. The algorithm stops when a maximum number of generations has been made, the change in X $< \varepsilon_1$, or the change in f(X) ε_2: the latter two usually indicate a plateau in the data, and the random walk will continuously iterate with no improvement.

The overall flow of the algorithm is:

```
A Create a random initial point $X_0$, $X_{best} = X_0$
B While stopping criteria are unmet:
    1 Generate $D_k$ and $h_k$,
    2 $T = X_k + D_k \cdot h_k$
    3 if f(T) > f($x_k$) && rand(0,1) < C (Worse point but random chance to move)
      $x_{k+1}$ = T
    4 Else
        i. $x_{k+1}$ = T
        ii. if f($X_{best}$) ¿ f(T) $x_{best}$ = T
C $X_{best}$ is the most optimal solution found
```

As mentioned in the title, random optimisation is very unpredictable: it might provide great results, but there's no guarantee and it could completely miss every minima of the search space. Most of the following algorithms work to use randomness in a controlled manner.

5 Genetic Algorithms

Genetic Algorithms work by evolving a pool of candidate solutions, optimising them in the process till it eventually converges to an optimal enough solution. Optimising *usually* means minimising, unless otherwise specified, and this is the assumption throughout these notes. To convert from a maximisation to a minimisation problem, just multiply the cost function by -1 . Each solution is represented as a **chromosome**, which contains values for the variables of the given problem (each variable is known as a **gene**). For example, if we're optimising a function of the form $f(x, y)$, then each chromosome contains two genes. **Binary Genetic Algorithms** represent chromosomes as a binary string, **Continuous Genetic Algorithms** represent them as a tuple (or list or array) of genes.

The general form of a genetic algorithm has 4 main parts:

1. Natural Selection
2. Pair Selection
3. Crossover
4. Mutation

Combining all of these together, the overall flow of the algorithm is:

A Create a random initial population

B While stopping criteria are unmet:

    1 Filter population with Natural Selection
    2 Create pairs via Pair Selection
    3 Make new chromosomes via Crossover
    4 Apply mutation to the population

C Select the best performing chromosome as the solution

For problems that have constraints or multiple objectives to be optimised, having the cost function be a weighted sum of each individual requirement is sufficient. Note that all must be maximisation/minimisation: these can be converted with negative weights if required. Constraints should degrade the cost if not met (e.g add a large integer with absolute value higher than the lowest possible cost), and should be designed such that a solution violating a constraint cannot have a lower cost than a function that satisfies it. There is no such restriction for preferences.

5.1 Stopping Criteria

Any/all of the below can be used: it depends on the situation:

1. An acceptable (low enough cost) solution has been found
2. The mean/standard deviation/range of the chromosomes is within a target value
3. No change in chromosomes/best/cost/worst cost after X iterations
4. N iterations have elapsed

5.2 Binary Strings

For the **Binary Genetic Algorithm**, we also have to determine how to convert to and from binary strings. Using binary strings is mostly historical, back when the memory used by each chromosome was important. It's also fairly simple to implement.

5.2.1 Binary String Length

The overall length of the binary string is dependent on the number of possible values our gene can take. For example: for an equation $f(x, y)$, if x has 100 possible values and y has 30, the overall length of the binary string representation will be $7 + 5 = 12$.
Normally though, we're given a range and precision that our genes must adhere to. Rephrasing our earlier example: $0 \leq x \leq 1$ with a precision of 2 decimal places, $5 \leq y \leq 8$ with a precision of 1 decimal place. To convert from these to the above, we use the following equation:

$$Number\_of\_Values = \frac{x_{hi} - x_{lo}}{10^{-d}} \tag{4}$$

This equation is the same as saying $integer\_range * 10^{decimal\_places}$. Now that we have the number of possible values a gene can take, we need to find the minimum length of a binary string that can represent that many distinct values. A binary string of length **n** can represent $2^n - 1$ distinct values.

For example: using the above equation, we see there are 30 distinct values for y. $2^4 - 1$ is 15, $2^5 - 1$ is 31. Therefore we choose a binary string of length 5 to represent y. We could have also calculated this as $ceil(log_2(30))$.

The above can be also be summarised as:

$$\frac{x_{hi} - x_{lo}}{10^{-d}} \leq 2^m - 1 \tag{5}$$

Where m is the required length. To get the full string, do this for each variable and add them together.

5.2.2 Binary Encoding and Decoding

Our binary representation roughly contains one unique string for each possible value. It's not going to be a perfect match though, as binary can't perfectly split decimal. To ease in the conversion, we calculate the **Binary Value**:

$$\frac{x_{hi} - x_{lo}}{2^m - 1}$$

This gives us the decimal value of a 1 in the binary string, and we'll represent it as b. Representing the binary string as BS and the decimal value as D, the conversions are:

$$Encoding : BS = binary(round(\frac{D - x_{lo}}{b})) \tag{6}$$

$$Decoding : D = x_{lo} + decimal(BS) * b \tag{7}$$

With these, x_{lo} is represented as $0 \ldots 0$ and x_{hi} as $1 \ldots 1$ for every gene. The complete binary string for a chromosome is just the concatenation of the strings for each gene.

5.2.3 Grey Codes

One problem with binary strings is adjacent numbers can have very different representations. For example: 3 and 4 have a difference of 1 in decimal, but their binary representations have every bit different (011 and 100). We can quantify this as the **Hamming Distance**, which is the number of bits that are different between two binary numbers (e.g 011 and 100 have a h.d of 3, 001 and 101 have a h.d of 1).

Grey codes are an alternate representation that guarantee consecutive numbers to have a Hamming Distance of 1. Representing the binary string as b and the grey code as g, both with length n, the conversions are as follows:

Binary to Grey:

$$g[i] = \begin{cases} b[i], & \text{if } i = n \\ b[i] \, XOR \, g[i+1], & \text{otherwise} \end{cases} \tag{8}$$

Grey to Binary:

$$b[i] = \begin{cases} g[i], & \text{if } i = n \\ b[i+1] \, XOR \, g[i], & \text{otherwise} \end{cases} \tag{9}$$

5.3 Natural Selection

This part of the algorithm determines which chromosomes are carried forward to the next generation.

5.3.1 X_{rate}

Only the best survive. The top (lowest cost) N_{keep} chromosomes are chosen, and the rest discarded. This method guarantees a fixed amount from each generation are kept, though this can be varied by changing X_{rate} (a user-defined percentage) at each iteration. With N_{pop} as the size of the population, N_{keep} can be calculated as:

$$N_{keep} = N_{pop} * X_{rate}$$

5.4 Thresholding

Only chromosomes with a score below a given threshold survive. This has the advantage of not having to sort and rank the chromosomes, but there is no guarantee on how many chromosomes survive. This can be fixed by changing the threshold at each iteration, either to an average or to a reasonable value.

5.5 Pair Selection

This part of the algorithm determines how chromosomes are paired up for crossovers (creating new chromosomes). This pairing up is important as it can influence how genes spread throughout the population, and can prevent good genes being overshadowed by poor ones. All of the below are independent of the cost function.i.e they don't require it to be of a certain form, such as differentiable.

5.5.1 Adjacency

Just pair adjacent chromosomes from top to bottom. If the population is sorted, this pairs chromosomes according to their ranks, which produces chromosomes without introducing much diversity (exploitation as opposed to exploration). It's very simple to implement, and every chromosome is used in crossovers.

5.5.2 Random

Randomly choose pairs. To generate the pair, a chromosome is chosen randomly, twice. This gives us unmatched diversity (exploration over exploitation), which gives a higher chance to find better quality offspring. Again very simple to implement, but for a completely random selection there's a chance we choose the same chromosome twice in a single pair.

5.5.3 Rank-Weighted Roulette

Choose pairs with probabilities according to their ranks. To generate the pair, two chromosomes are chosen according to the probabilities and paired together. If random can be taken as given each chromosome an equal chance to be chosen, Rank Weighted Pairing gives each chromosome a chance proportional to their rank in the population, which requires us to sort the population. This favours chromosomes with better costs. With a population of size N_{keep}, this probability for chromosome C_n of rank n is calculated as:

$$P(C_n) = \frac{N_{keep} + 1 - n}{\sum_{i=1}^{N_{keep}} i} \tag{10}$$

This gives probability $1/sum(N_{keep})$ to the last chromosome and $N_{keep}/sum(N_{keep})$ to the first. This gives a large difference in probability between adjacent chromosomes, regardless of their actual weights. For small populations, this gives a very skewed probability.

5.5.4 Cost-Weighted Roulette

Choose pairs with probabilities according to their weights. This doesn't need us to sort the population, but we do need to account that some costs may be negative. This is fixed by normalising the costs:

$$NCost(C_n) = Cost(C_n) - C_{N_{keep}+1} + 1 \tag{11}$$

$C_{N_{keep}+1}$ is the cost of the best chromosome not chosen. For X_{rate} selection this is obvious, for threshold selection or where $N_{keep} = 0$, instead use 2 * the cost of the lowest chromosome. As this value is guaranteed to be $\geq$ the lowest chromosome, subtracting it guarantees that all Ncosts are < 0. With the normalised costs, probability for each chromosome is calculated as:

$$P(C_n) = \frac{NCost(C_n)}{\sum_{i=1}^{N_{keep}} NCost(C_i)} \tag{12}$$

This gives a more accurate reflection of the relationships between the weights of the chromosomes, but is more computationally expensive.

5.6 Tournament Selection

A small subset of chromosomes is randomly chosen, and the best performing chromosome from this subset. This is done twice to get a pair, and the size of each subset is normally 2-3. A variation involves taking a larger subset, ranking them and then performing *rank-weighted roulette* to choose two chromosomes.

Both variations favour chromosomes with better costs. Also, since in each instance only a trivial subset has to be sorted, it reduces computation on sorting and thus works well for problems with large populations.

5.7 Crossover

Once the parents are selected, to generate offspring we take parts from each and combine them to make new chromosomes. Other than a few exceptions these don't generate new values: just re-arrange the existing ones. Repeatedly doing just crossovers only covers a limited section of the search space, so isn't sufficient to find an optimal solution. Crossover methods can be combined and altered: for example, a Single Point crossover followed by Blending, or only considering a subset of indices for Extrapolation.

Only the first three crossovers are applicable to the Binary Genetic Algorithm. Crossovers for ordering problems are detailed in a separate section.

5.7.1 Single Point

This breaks each parent into two parts, and creates offspring by swapping the second part for each. A random number $0 \leq n < L$ is generated, and used to split the chromosomes. L is the length of the binary string / number of genes for binary and continuous genetic algorithms respectively. Using slice notation, this can be shown as:

$$Child_1 = Parent_1[0:n] + Parent_2[n:end]$$
$$Child_2 = Parent_2[0:n] + Parent_1[n:end]$$

5.7.2 Double Point

This breaks each parent into 3 parts, and swaps the middle parts for each. This middle section is denoted by two crossover points, which are the start and end of the section respectively. These follow $0 \leq n_1 < n_2 \leq L$. Using slice notation, this can be shown as:

$$Child_1 = Parent_1[0:n_1] + Parent_2[n_1:n_2] + Parent_1[n_2:end]$$
$$Child_2 = Parent_2[0:n_1] + Parent_1[n_1:n_2] + Parent_2[n_2:end]$$

5.7.3 Uniform Crossover

This uses a variable $\mu \in (0,1)$ to determine how much crossover occurs. For each index of the binary string/gene tuple, generate a random number n $\in (0,1)$. If $n < \mu$, the bit/gene in that index is swapped in the children, otherwise are directly copied from the parent. The addition of μ gives us a tuning parameter, but if it's too high or low effectively no crossover will happen (no swaps or all swaps).

5.7.4 Blending

This uses a variable $\beta \in (0,1)$ to combine the genes from the parents (P) into new values as below:

$$Child_1[i] = \beta * P_1[i] + (1-\beta) * P_2[i]$$
$$Child_2[i] = \beta * P_2[i] + (1-\beta) * P_1[i]$$

These values are in the range $(P\_1[i], P\_2[i])$, so while these generate new values they are still limited to a subset of the search space. β can change from generation to generation, etc.

5.7.5 Extrapolation

This also uses a variable β, but there's no limit on its value. Children are generated using the below:

$$\begin{aligned} Diff_i &= abs(P_1[i] - P_2[i]) \\ Child_1[i] &= P_1[i] - \beta * Diff_i \\ Child_2[i] &= P_2[i] + \beta * Diff_i \end{aligned}$$

With some re-arrangement these are the same equations as Blending, but as there's no limit on β this can explore the entire search space without mutation. β can change from generation to generation, and if needed we can artificially limit the range of values generated.

5.8 Mutation

Mutation is the process that actually drives evolution: by changing the chromosomes into new values, new solutions can be evaluated and used. It is possible to have an algorithm that consists entirely of mutation: this would perform very similar to random walk or (1+1)ES. Crossovers move the genes into potentially better solutions, mutations discover new solutions.

5.8.1 How many Mutations?

Mutation is determined using a mutation rate, μ. To prevent good solutions from being mutated, we can designate the top n chromosomes as *elite*, and not consider them for mutation. n can be left as 0, in which case any chromosome can be mutated.

With N_{pop} as the size of the population, the standard formula is:

$$M = (N_{pop} - n) * \mu * N_{bits} \tag{13}$$

N_{bits} is the length of the binary string for a chromosome for BGA, and the number of genes in a chromosome for CGA.

5.8.2 Binary Mutation

Disregarding the top n chromosomes, choose M bits randomly across the population and flip them (0→1, 1→0).

5.8.3 Numeric Mutation

This uses a variable σ to control how much a gene mutates. Disregarding the top n chromosomes, choose M genes randomly and apply:

$$Gene = Gene + \sigma * N_n(0, 1) \tag{14}$$

σ is usually a constant, but for a dynamic system can be changed between generations. N_n is a random number generator, but generates numbers according to a standard distribution (i.e 0.5 is most likely to be generated, 0 and 1 are very unlikely).

5.9 Permutation Problems

Certain problems (such as the **Travelling Salesman Problem**) require their solution to be an ordering of a given set of variables. This means traditional mutation and crossover methods won't work, since they change the values and discover new ones respectively. Chromosomes take the form of a tuple of values, similar to CGA, but every chromosome has the same values in a different order.

5.9.1 Partially Matched Crossover

Perform two-point crossover, then swap duplicated genes *outside* of the two points. This can be done by going through the children, and swapping the first duplicate in Child$_1$ with the first in Child$_2$ and so on .e.g.

$$\begin{aligned} Parent1 : (3,4,6,2,1,5)\ ;\ Parent2 : (4,1,5,3,2,6) \\ ChosenPoints : 1,3 \\ InitialChildren : (3|1,5|2,1,5); (4|4,6|4,2,6) \\ FixedChildren : (3,1,5,2,4,6); (1,4,6,3,2,5) \end{aligned}$$

This doesn't preserve the existing ordering in the slightest: so two very good orderings could potentially produce two terrible ones.

5.9.2 Ordered Crossover

Similar to two-point crossover, choose two points and add the middle section to the children. Then fill in the empty spaces with the genes from the parent starting at the first crossover point, skipping the duplicates. e.g.

$$\begin{aligned} Initial : Parents - (3|46|215), (4|15|326) \\ Phase1 : Children - (x|15|xxx), (x|46|xxx) \\ Phase2 : Children - (4|15|623), (1|46|532) \end{aligned}$$

For Child$_1$, the x's are filled in with 4623 (starting at crossover point 1, skipping duplicates, looping around at the end). For Child 2, 1532 is taken from the parent. An alternative is to delete the duplicates from the parents then directly fill in the children, which saves skipping the duplicates.

This method preserves the relative ordering, but not the absolute. For TSP this is fine since the start is irrelevant, and the sequence can just be rotated to get the desired start point.

5.9.3 Cycle Crossover

Swap the left-most genes, then keep swapping duplicates from left to right in child$_1$ till there are none left. This method has a relatively simple implementation, as only child$_1$ is checked for duplicates and the checking index only advances from left to right (looping around if necessary).e.g.

$$
\begin{aligned}
&Initial : Parents - (|346215), (|415326)\\
&Swap1 : Children - (4|46215), (3|15326)\\
&Swap2 : Children - (41|6215), (34|5326)\\
&Swap3 : Children - (41622|5), (34531|6)\\
&Swap4 : Children - (4163|25), (3452|16)
\end{aligned}
$$

5.9.4 Coding for Crossovers

Rather than a direct crossover method, the chromosomes are encoded to a numeric string, can then have any crossover method applied to them, then decoded back to a chromosome.

Encoding:

A Make a reference list of the genes

B For idx in range(0,len(chromosome)):

  1 coded[idx] = index of chromosome[idx] in reference list

  2 remove chromosome[idx] from the reference list

e.g.

Epoch	Coded	Chromosome	Reference List
1		346215	1,2,3,4,5,6
2	3	46215	1,2,4,5,6
3	33	6215	1,2,5,6
4	334	215	1,2,5
5	3342	15	1,5
6	33421	5	5
7	334211		

Decoding:

A Make a reference list of the genes

B For idx in range(0,len(coded)):

  1 chromosome[idx] = index of coded[idx] in reference list

  2 remove coded[idx] from the reference list

e.g.

Epoch	Coded	Chromosome	Reference List
1	351311		(,2,3,4,5,6
2	51311	3	1,2,4,5,6
3	1311	3,6	1,2,4,5
4	311	3,6,1	2,4,5
5	11	3,6,1,5	2,4
6	1	3,6,1,5,2	4
7		3,6,1,2,5,4	

5.9.5 Mutation

As generating new values breaks the problem, the following methods can be used:

1. Invert/Reverse a randomly selected subset of the chromosome
2. Move/Swap a randomly selected subset within the chromosome
3. Randomly swap positions: this may break an encoded chromosome, so it must be decoded first

5.10 Genetic Programming

This application of GA allows it to build a program for us. It is provided an input and a desired output, and the algorithm builds us a program to convert between to two.

Rather than strings or tuples, chromosomes are represented as tress of varying sizes and shapes. This requires extra overhead for each chromosome, since the size and shape has to be tracked as well. Lead nodes are variables and constants, while internal tree nodes are functions that combine these.

The algorithm also has to be provided a set of possible values for variables/constants, a set of functions, and semantics on how to combine the two. e.g. the SUM function requires two numerics and outputs a numeric, the OR function requires two binary strings and outputs a binary string, etc.

5.10.1 Crossovers

The simplest crossover is to swap and replace subtrees between chromosomes. A single child can be generated by replacing a subtree in one parent by one from the other parent, and two children can be obtained by swapping subtrees.

5.10.2 Mutation

Node Mutation Swap a node with an appropriate alternative
e.g replace SUM with DIFF, replace 4 with 11

Swap Mutation Swap the order of terminal nodes provided to a function
e.g a DIFF b → b DIFF a

Grow Mutation Replace a terminal node with a new subtree that provides the same type as the replaced node e.g. replace 5 with a SUM 7

Gaussian Mutation Add a value determined by a gaussian ($N_n * \sigma$) to a terminal node

Trunc Mutation Replace a subtree with an appropriate terminal node
e.g replace 5 MOD 2 with 11

5.11 Schema Theorem

This theorem is used to demonstrate that GAs are effective in discovering good solutions, and increase the quality of the solutions with more generations. The proof uses the binary genetic algorithm, but the result proves effectiveness for all GAs.

5.11.1 What is a Schema

A schema (or scheme) is a template that represents binary strings, similar to how regular expressions represent regular strings. They consist of '0','1' and '*' (wildcard).

The order of a schema is the number of fixed positions it contains (O(s)), and the defining length (δ(s)) is the distance between the first and last fixed positions. e.g.

$$\begin{aligned} &**00**0***\\ Order &: 3\\ \text{Defining Length} &: 7-3=4\\ \text{Examples of matching Strings} &: 1100010101, 0000000000, 0100100111 \end{aligned}$$

For this proof, we define the average cost of a schema s at generation t (f(s,t)) as the average cost of all the chromosomes that match the schema at that generation, and n(s,t) is the number of chromosomes at t that match s.

5.11.2 Schema Pair Selection

Using cost-weighted roulette, the probability that a chromosome C is chosen is $\frac{Cost(C)}{\sum_{i=1}^{N_{keep}} Cost(C_i)}$. Therefore, for a chromosome matching a schema at the generation, the average probability of being chosen is $\frac{f(s,t)}{\sum_{i=1}^{N_{keep}} Cost(C_i)}$. The probability for **any** chromosome matching the schema being chosen is $\left(n(s,t) * \frac{f(s,t)}{\sum_{i=1}^{N_{keep}} Cost(C_i)}\right)$.

This selection happens N_{pop} times. If we assume that only selection takes place and no crossover/mutation occurs, we obtain:

$$n(s,t+1) = n(s,t) * \frac{f(s,t)}{\sum_{i=1}^{N_{keep}} Cost(C_i)} * N_{pop} = n(s,t) * \frac{f(s,t)}{\text{Avg. Cost at gen t}} \tag{15}$$

With some re-arrangement, we can re-write these as the following:

$$\begin{aligned} n(s,t+1) &= n(s,t) * (1+\varepsilon(s,t)) && (16)\\ &= n(s,t-1) * (1+\varepsilon(s,t-1)) * (1+\varepsilon(t))\\ &= n(s,1) * \prod_{i=1}^{t}(1+\varepsilon(s,t)) && (17)\\ \varepsilon(s,t) &= \frac{f(s,t) - Avg.cost(t)}{Avg.cost(t)} && (18) \end{aligned}$$

5.11.3 Schema Crossover

After being selected as a parent, a schema is said to survive a crossover if one of the child chromosomes also matches it. The only circumstance a schema breaks in single-point crossover is if the break point is inside the defined length of the schema: as the externals are wild cards it doesn't matter. Using this, and the knowledge that the last index can't be selected for a single-point crossover, the probability that a schema survives a single-point crossover is:

$$P(survival) = 1 - \frac{\delta(s)}{len(s) - 1} \tag{19}$$

5.11.4 Schema Mutation

Schemas are resilient to mutation as wildcards match both 0 and 1, so they only break if one of the fixed positions is mutated. If P_m is the probability for a bit to be mutated, the probability a schema survives mutation is:

$$P(survival) = (1 - P_m)^{O(s)} \tag{20}$$

5.11.5 Conclusion

Combining the above three sections, we obtain the schema growth equation:

$$\begin{align} P(survival) &= n(s,t) * (1 + \varepsilon(s,t)) * \left(1 - \frac{\delta(s)}{len(s) - 1}\right) * (1 - P_m)^{O(s)} \tag{21} \\ &= n(s,t) * \frac{\text{Avg.Cost of Matches}}{\text{Overall Avg. Cost}} * (1 - \frac{DefiningLength}{SchemaLength}) * (1 - P_m)^{Order(s)} \tag{22} \end{align}$$

Therefore, we can say that above-average, short, low-order schema propagate well. The more above average the better, so with enough generations eventually the weaker solutions disappear and the more optimal ones remain.

6 Evolution Strategies

Evolution Strategies are similar to Genetic Algorithms, with two main differences:

1. ES can only handle real values
2. ES uses a set of values, called **Strategy Parameters** to handle mutations for each gene

The overall flow of the algorithm is:

- A Create a random initial population and strategy parameters
- B While stopping criteria are unmet:
  - 1 Create pairs via Pair Selection
  - 2 Make new chromosomes via Crossover
  - 3 Apply Mutation to the children and parameters
  - 4 Select new population
- C Select the best performing chromosome as the solution

6.1 Strategy Parameters

Strategy Parameters (σ) for ES are the same as the step size in gradient descent. One is created for each variable in the optimisation problem: so there will be an equal number of parameters and genes.

A larger σ will move further in the current population direction, exploiting a good trend, and a smaller σ will search locally to find a good new direction to move the population. The σ are mutated alongside the population, and the mutations apply even if the offspring are rejected.

6.2 ES Notation

Evolution strategies are depicted with the (μ, λ) or $(\mu + \lambda)$ notation. μ represents the size of the population at each generation, and λ is the number of children generated at each stage.

The , and + represent two methods of selection. With , the new population is selected only from the children, and with + the new population is selected from a combination of the parents and children. e.g. (1,1)ES will always choose the child (effectively random walk), and (1+1)ES will choose the better of the child and parent (smarter random walk).

For , notation, λ must be $\geq$ than μ: otherwise the population requirement won't be hit. (μ, λ) results in faster convergence, $(\mu + \lambda)$ gives more exploration.

6.3 Crossovers

Some variations don't use crossovers (the children are just the parents), but otherwise there are four main types of crossovers. They can be distinguished in the following categories:

- Source:

  Local 2 parents, 1 child

  Global >2 parents, 1 child

- Output:

  Discrete Children contain elements from the parents

  Intermediate Children contain elements derived from the parents, such as a weighted average.

To generate more children, the order of the parents is changed. Examples are shown below:

	Discrete	Intermediate
Local	Uniform Crossover	Blending
Global	Non-Uniform Crossover	Uniform Blending

6.3.1 Non-Uniform Crossover

Rather than using a single μ, a rank-based cost roulette is used amongst the parents. Otherwise this is the same as uniform crossover, selecting each index of the children from a parent. The top-ranked parent (P_1) has $P = 0.5$, and the others are generated according to:

$$P(P_2) > P(P_3) > \ldots > P(P_n)$$

6.3.2 Uniform Blending

$$Child[i] = \frac{1}{\text{Num. of Parents}} * \sum_{k=1}^{n} parent_k[i]$$

6.4 Mutation

6.4.1 Chromosomes

For a chromosome c, the mutation of each gene c[i] depends on the strategy parameter σ_i. This is similar to CGA, but as σ_i adjusts as well it allows for dynamic growth, allowing the algorithm to choose between exploring and exploiting. As before, N_n gives a random number according to a gaussian distribution.

$$c[i] = c[i] + \sigma_i * N_n(0, 1) \tag{23}$$

6.4.2 Strategy Parameters

These change according to how well the population is doing: if the quality of the solutions is steadily improving then the algorithm's speed is increased, and if solutions aren't improving then the speed is decreased in favour of local search. Determining an improvement could be checking if the average cost lowers, or if more children are chosen than parents.

$$\sigma_i(t+1) = \begin{cases} \sigma_i(t) * e^{\frac{1}{3}}, & \text{if t=0 or previous mutation was successful} \\ \sigma_i(t) * e^{\frac{-1}{12}}, & \text{otherwise} \end{cases} \tag{24}$$

6.5 (1+1)ES

Effectively a smarter random walk, this chooses the better of the parent and child at each generation. No crossover occurs, and the offspring is simply a copy of the parent with mutation.

6.5.1 Parameter Mutation Variants

1. No mutation: this is similar to a GA

2. If, in the last X generations, the good mutation rate $\geq 20\%$, then increase σ. Else, decrease it.

3. With n_x being the number of genes, n_m being the number of good mutations in the last $10n_x$ generations and $0 < \alpha < 1$, perform the following every $10n_x$ generations:

$$\sigma_i(t+1) = \begin{cases} \sigma_i(t) * \alpha, & n_m < 2n_x \\ \sigma_i(t)/\alpha, & n_m > 2n_x \\ \sigma_i(t), & n_m = 2n_x \end{cases}$$

6.6 (μ+1)ES

The population only potentially changes by 1 individual each time. Since there are multiple chromosomes, the offspring is obtained by crossover and thus there is a diversity in output.

7 Differential Evolution

Differential Evolution is similar to Evolution Strategies, but uses a guided form of mutation rather than adding a randomised value. It does this by generating difference vectors, and using these to mutate the generated offspring.
Unlike previous algorithms, this allows chromosomes to work together to achieve a solution: previously each chromosome was independent, now their mutation is interdependent. Another crucial difference is that each chromosome is now regarded as a point/particle in an n-dimensional space (n being the number of genes). This has no effect on the structure of the chromosome (it's still a tuple of values), but allows us to use vector arithmetic on it (any point in space is a vector from the origin to that point).

The overall flow of the algorithm is:

A Create a random initial population and control parameters

B While stopping criteria are unmet:

   1 Apply mutation to each parent to generate a trial vector

   2 Make new chromosomes via Crossover with parent+trial vector

   3 Select new population from parents and offspring

C Select the best performing chromosome as the solution

The two control parameters are the scale factor (β) and the crossover parameter (P_r). These are detailed in the following sections. DE hybrids are discussed in a later chapter.

7.1 Mutation

A mutated vector, known as a **trial vector**, is generated for each parent. The basic skeleton is:

$$U_i(t) = X_{i_1}(t) + \beta * (X_{i_2}(t) - X_{i_3}(t)) \tag{25}$$

$$\text{Trial Vector} = \text{Target Vector} + \text{Scale Factor} * \text{Difference Vector}$$

Where i is the index of the parent vector, and i_1, i_2 and i_3 are the indices randomly selected vectors from the population $\neq$ i. β can take any positive value, but is usually in the range (0.5,2).
This mutation automatically scales between exploration and exploitation, as the difference vector shrinks as the particles get closer together. The scale factor helps with this, but early on in the search the difference vectors will be large and may dramatically spread the particles out. Another disadvantage is that the trial vector has nothing to do with the parent, though it is combined with the parent to make the offspring.

7.2 Crossover

Once the trial vector is generated, it is mated with the parent vector to produce an offspring. Though standard methods like Uniform Crossover, Blending or Double Point Crossover can be used, the usual method for DE is:

$$X'_{ij}(t) = \begin{cases} X_{ij}(t) & \text{, if j}\in\text{J} \\ U_{ij}(t) & \text{, otherwise} \end{cases} \tag{26}$$

Where $X_{ij}(t)$ is the j^{th} gene of the parent, and $U_{ij}(t)$ is the j^{th} gene of the trial vector. J is a list of indices, and generation methods are shown below (most of which use P_r). This method limits the overall amount of change, as opposed to uniform crossover, and blends the entire vector together, as opposed to clearly defined subsections being from each vector. Later in the search we can expect most genes to be roughly correct, and P_r gives us a tuning parameter to reduce the amount of trial vector influence.

7.2.1 Random Generation

J is a random list of indices $\in (0, n_x)$, where n_x is the number of genes. In extreme cases this may include every/no indices, and there's no tuning parameter.

7.2.2 Binomial Crossover (bin)

This approach is similar to uniform crossover, but uses P_r as opposed to μ. This approach guarantees at least one value in J (so some mutation), and P_r gives us a tuning parameter: a high P_r means mostly the offspring is mostly from the parent, a low P_r means mostly from the trial vector.

```
1. Randomly add a value x $\in (0, n_x)$ to J
2. for i in range(0, $n_x$):
    (a) if ( i!= x && rand() < $P_r$)
            Add i to J
```

7.2.3 Exponential Crossover (exp)

This provides an adjacent sequence of indices (similar to Double Point Crossover), but this range is generated using P_r.

```
1. Randomly add a value i $\in (0, n_x - 1)$ to J
2. Do
    (a) Add i+1 to J
    (b) i = (i+1)%$n_x$
   while (rand() < $P_r$ and len(J) < $n_x$ )
```

This is exponential as a sequence of length 2 has probability $(P_r)^0$, of length 3 has $(P_r)^1$, and length n $(P_r)^{n-2}$. A high P_r gives us more influence from the parent vector, a low P_r more influence from the trial vector.

7.3 DE Notation

Differential Evolution methods are denoted as DE x/y/z, where x is the method of selecting the target vector in mutation, y is the number of difference vectors and z is the crossover method used.

7.3.1 Target Vector Selection

It was previously mentioned that the target vector for mutation was randomly selected (rand), but this isn't fixed. Another popular method is selecting the best vector ($\hat{X}$)from the population (best).
Rand-to-Best sets the target vector as $\alpha * \hat{X} + (1-\alpha) * X_{i_1}(t)$: α then allows us to tune between exploration (the random vector) and exploitation (the best vector). α can be set to dynamically increase from 0 to 1 as generations increase.
Lastly, current-to-best sets it as $\beta * (\hat{X} - X_{i_p}(t))$, where $X_{i_p}(t)$ is the parent vector. This allows the parent to influence the trial vector, and minimises the divergence.

7.3.2 Difference Vectors

The basic formula has a single difference vector, but any number can be chosen, denoted as n_v. Using the current-to-best target vector adds another difference, so this is denoted as $1 + n_v$.

7.3.3 Examples

1. DE rand-to-best/2/exp

   Trial Vector Equation $\alpha * \hat{X} + (1-\alpha) * X_{i_1}(t) + [\beta * (X_{i_2}(t) - X_{i_3}(t)) + \beta * (X_{i_4}(t) - X_{i_5}(t))]$

2. DE current-to-best/1+2/bin

   Trial Vector Equation $\beta * (\hat{X} - X_{i_p}(t)) + [\beta * (X_{i_2}(t) - X_{i_3}(t)) + \beta * (X_{i_4}(t) - X_{i_5}(t))]$

7.4 Switching DE

Studies have shown that DE rand/1/bin is good for exploring and DE current-to-best/2/bin is good for exploiting. We could swap between them after a fixed number of generations, but the usual method uses probabilities P_1 and $P_2 (= 1 - P_1)$ to decide which method to use at a generation.

Also defined are n_1 and n_2 (the total number of offspring used after methods 1 and 2 respectively) and f_1 and f_2 (the total number of offspring discarded after methods 1 and 2 respectively).

The first X (normally 50) generations are set as a learning period, with $P_1 = P_2 = 0.5$ and recording values for n_1, n_2, f_1 and f_2. After generation X, the following equations are used:

$$\begin{aligned} P_1 &= \frac{n_1 * (n_2 + f_2)}{n_1 * (n_2 + f_2) + n_2 * (n_1 + f_1)} \\ P_2 &= 1 - P_1 \end{aligned} \tag{27}$$

With some re-arrangement, this is simply $P_1 = \frac{\text{Method 1 success \%}}{\text{Method 1 success \% + Method 2 success \%}}$

8 Swarm Optimisation

Swarm optimisation/intelligence takes DE a step further: rather than a group of solutions deriving from each other, a group of simple agents collectively find a single optimal solution. A highlight of this method is there is no central control or global model: each agent is a simple individual, and their local interactions cause global patterns to emerge. Some further characteristics are:

Flexible it can handle internal/external problems and unexpected events

Resilient even if some agents fail to complete, the overall system will continue

Self-Organised Paths to solutions aren't predefined: the system discovers the correct method

8.1 Self-Organisation

By using simple individual mechanisms, the interactions of the agents form structures on the global level of the system without external planning or control.

1. Positive Feedback: good solutions are highlighted and reinforced
2. Negative Feedback: introduce a time scale to the algorithm by reducing the effects of older feedback, prevents early/premature convergence on local optima
3. Exploration and diversity are amplified, by moving randomly then falling into the positive feedback loops
4. Agents communicate with each other, directly or indirectly

8.2 Stigmergy

Stigmergy is a method of indirect agent communication by modifying the environment the agents are in. These may trigger certain actions from the receiving agents, or simply be positive/negative feedback. There are two main types:

Sematectonic Communicate via changing the physical environment (the current state of the solution).e.g. well-travelled routes show more wear and tear

Sign-Based Communicate via signalling mechanisms.e.g laying a trail of breadcrumbs on the travelled route

8.3 Binary Bridge Experiment

An example of the above is the binary bridge experiment: there are two paths from the start to the goal, one longer than the other. The agents in this case are ants, which leave a pheromone behind them as they travel (Sign-Based Stigmergy). This pheromone attracts other ants, which increases the probability that future ants pick that path.
The ants reach the goal then return to the source via the same path. As the shorter path will have ants completing their journey faster, more pheromones will be deposited, which

attracts more ants until eventually every agent takes the shorter path. The pheromones dissipate after a while (negative feedback), so the longer path will slowly lose attractiveness while the shorter is constantly reinforced.

There is no guarantee that the ants will always pick the path with more pheromones though: a small random chance is added to pick a previously unexplored path, to prevent a scenario where the initial ants only traverse the longer path, and all ants only pick that path.

9 Ant Colony Optimisation

An type of swarm optimisation where the agents mimic ants, using sign-based stigmergy. Though the base form solves shortest path and TSP problems, a large number of problems can be reduced to these forms. Unlike previous methods, the paths taken by the ants are the solutions, not the ants themselves.
Problems are given in a graph structure, with each edge (i,j) having a cost/length and pheromone concentration ($\tau_{ij}(t)$) associated with it. Also defined are $L^k(t)$, which is the length of the path taken by ant k at generation t, and $N_k^i(t)$, the list of nodes reachable from node i for ant k at gen t. The latter two will become more clear in the next section. The overall flow of the algorithm is:

A Create random initial pheromone concentrations ($\tau_{ij}(0)$)

B While stopping criteria are unmet:

  1 Create and evaluate a path for each ant

  2 Evaporate the Pheromones

  3 Apply feedback to the Pheromones

C Select the path with the most ants/ the most optimal path

9.1 Creating an Ant Path

At each generation/iteration, a path is created for each ant according to the transition probabilities. These probabilities vary for each method (and are detailed later), but the overall method for shortest path problems is:

```
I $X^k(t)$ = {start_node}, i = start_node
1. While i != goal_node:
    (a) If $N_k^i(t)$ is empty, append the predecessor nodeto i
    (b) Select a node j from $N_k^i(t)$ according to the transition probabilities
    (c) Append (i,j) to $X^k(t)$, i = j
2. Remove all loops from $X^k(t)$
3. Calculate $L^k(t)$
```

Removing the loops from $X^k(t)$ affects $L^k(t)$, so care must be taken: for example, there may be two conflicting loops of different lengths, so depending on which one is removed this changes the attractiveness of the path.

9.2 Simple Ant Colony Optimisation

9.2.1 Transition Probability

SACO just uses a pheromone-weighted roulette for its transition probabilities: these are evaluated at each node while creating a path. The probability that ant k chooses edge (i,j) from node i at generation t is:

$$P_{ij}^{k}(t) = \begin{cases} \frac{\tau_{ij}(t)}{\sum_{ab \in N_k^i(t)} \tau_{ab}(t)} & , if j \in N_k^i(t) \\ 0 & , otherwise \end{cases} \tag{28}$$

9.2.2 Pheremone Evaporation

This uses a variable, $\rho \in (0, 1)$, to determine how much the pheremones evaporate at each stage. The evaporation removes the effects of older pheromones on current probabilities, so for a path to be consistently picked the pheromones on it need to be reinforced.

$$\tau_{ij}(t) = (1 - \rho) * \tau_{ij}(t) \tag{29}$$

9.2.3 Pheremone Updates

SACO uses a simple update system: pheromones for an edge are increased based on the number of ants that included that edge in their path (path for ant k is $X^k(t)$), and this increase is based on the quality of the ants' solutions. It uses a constant, Q, to moderate the increase in the pheremones.

$$\tau_{ij}(t + 1) = \tau_{ij}(t) + \sum_{k \in ants} \Delta\tau_{ij}^{k}(t) \tag{30}$$

$$\Delta\tau_{ij}^{k}(t) = \begin{cases} \frac{Q}{L^k(t)} & , if (i, j) \in X^k(t) \\ 0 & , otherwise \end{cases} \tag{31}$$

9.3 Ant System

AS builds off SACO by adding more heuristics for the transition probability and implementing elitism. Most versions also pre-process a list of connections from each node rather than calculating the probability for every edge at every node (even the obviously unreachable ones).

9.3.1 Transition Probability

As the system pre-processes $N_k^i(t)$, this doesn't need to be checked at each stage. Instead it uses the inverse of the cost ($\eta_{ij}(t) = 1/\text{cost(i,j)}$) and tuning parameters α and β. There are two possible methods, both of which are weighted roulettes using $\tau_{ij}(t)$ and $\eta_{ij}(t)$:

Exponent-Based

$$P_{ij}^k(t) = \frac{\tau_{ij}(t)^\alpha * \eta_{ij}(t)^\beta}{\sum_{ab \in N_k^i(t)} \tau_{ab}(t)^\alpha * \eta_{ab}(t)^\beta} \tag{32}$$

Fraction-Based

$$P_{ij}^k(t) = \frac{\alpha * \tau_{ij}(t) + (1-\alpha) * \eta_{ij}(t)}{\sum_{ab \in N_k^i(t)} \alpha * \tau_{ab}(t) + (1-\alpha) * \eta_{ab}(t)} \tag{33}$$

9.3.2 Pheremone Updates

AS keeps the same overall equation, but uses a different calculation for $\Delta\tau_{ij}^k(t)$. Again there are three different calculations, each gives slightly different results:

Regular (Ant-Cycle)

$$\Delta\tau_{ij}^k(t) = \begin{cases} \frac{Q}{L^k(t)} & , if(i,j) \in X^k(t) \\ 0 & , otherwise \end{cases} \tag{34}$$

Density-Based

$$\Delta\tau_{ij}^k(t) = \begin{cases} Q & , if(i,j) \in X^k(t) \\ 0 & , otherwise \end{cases} \tag{35}$$

Quantity-Based

$$\Delta\tau_{ij}^k(t) = \begin{cases} \frac{Q}{cost(i,j)} & , if(i,j) \in X^k(t) \\ 0 & , otherwise \end{cases} \tag{36}$$

9.3.3 Elite Pheromone Updates

On top of one of the above updates, AS also gives a special bonus to edges that were on the best path ($X^e(t)$). This is achieved by adding $n_e * \Delta_{ij}^e(t)$ to all paths, where n_e is the number of ants that took the best path.

$$\Delta_{ij}^e(t) = \begin{cases} \frac{Q}{L^{X^e}(t)} & , if(i,j) \in X^e(t) \\ 0 & , otherwise \end{cases} \tag{37}$$

9.4 Ant Colony System

ACS builds even further of AS, providing more tuning parameters and increasing the system's exploration. It also slightly changes the functions of Pheromone Evaporation and Pheromone Updates: they become global and local updates respectively, where global affects all pheromones and local only those on the best path.

9.4.1 Transition Probability

It uses a user-provided variable $r_0 \in (0,1)$ to balance between exploration and exploitation (higher $r_0 \rightarrow$ more exploitation). From an ant k at node i, the edge taken is decided by:

$$\text{Next edge for ant k} = \begin{cases} \text{Lowest cost edge} \in N_k^i(t) & \text{, if rand(0,1)} < r_0 \\ \text{Use Exponent-based from Ant System} & \text{, otherwise} \end{cases} \tag{38}$$

9.4.2 Pheromone Evaporation (Local Update)

Unlike previous systems, this both evaporates and adds to the pheromone concentration. This local addition gives the system more chance to explore since all edges get a boost, but this addition remains constant so its effects is reduced in later generations when the system is closer to finding an optimal path. It uses the user-provided constants $\rho_L \in (0,1)$ and $\tau_o > 0$. If either were 0 then it would be regular pheromone evaporation, and if P_L was 1 then the pheromone intensities would be reset at every generation.

$$\tau_{ij}(t) = (1 - \rho_L) * \tau_{ij}(t) + \rho_L * \tau_o \tag{39}$$

9.4.3 Pheromone Updates (Global Update)

The global updates gives the edges on the global best path an extra bonus, similar to the elite pheromone updates from AS, and thus pushes the system towards exploitation. It also adds an extra layer of evaporation, which reduces the universal bonus given in the previous step. The global best path ($X^+(t)$) can either be the best in this generation ($\tilde{x}(t)$) or the best from the first to the current generation ($\hat{x}(t)$). This update also uses a user-provided constant $\rho_G \in (0,1)$, which again isn't 0 or 1 for the reasons mentioned above.

$$\tau_{ij}(t+1) = (1 - \rho_G) * \tau_{ij}(t) + \rho_G * \Delta\tau_{ij}(t) \tag{40}$$

$$\Delta\tau_{ij}(t) = \begin{cases} \frac{1}{L^{X^+(t)}} & , if(i,j) \in X^+(t) \\ 0 & , otherwise \end{cases} \tag{41}$$

10 Co-ordinated Collected Behaviour

CCB a behaviour model that was proposed to interpret the movement of swarms or groups of animals (such as a flock of birds or herd of cows). The individuals in these groups only have local knowledge, limited mental facilities and are bound by the laws of physics, but can seamlessly make complicated decisions as a group (for an example, see Starlings in BBC's The Code). By following three simple rules, the behaviour of these groups can be modelled:

Separation If an agent is too close to a neighbour, it moves away

Alignment Each agent aligns itself towards the average alignment of its neighbours (tries to face the average direction)

Cohesion Each agent tries to move towards the average position of its neighbours

When using this technique in optimisation we can share more data between agents than physical animals can, so agents can use alignments and positions from non-neighbouring agents to find a more optimal solution.

10.1 Particle Swarm Optimisation

PSO is another type of swarm optimisation, but is closer to DE in that each agent is a particle/vector in n-dimensional space and a potential solution. Each agent also has a velocity associated with it, which guides the speed and direction of the search.

The overall flow of the algorithm is:

A Create initial agents (with random positions and zero velocities), find the best agent from these

B While stopping criteria are unmet:

  1 Update velocities for each agent

  2 Update positions for each agent

  3 Find the personal best for each agent and the local/global best agents

C Select the best agent as the solution

As previously mentioned, since the algorithm has the data for all of the agents data can be shared between non-neighbouring agents, or neighbours of an agent don't have to be restricted by distance.

10.1.1 Global Best PSO

Every agent receives information from the entire swarm: this converges quicker (exploitation over exploration). In the implementation of the algorithm, the best particle of each generation ($\hat{y}(t)$) is tracked. Each particle also keeps a track of its personal best position($y_i(t)$ for particle i) since the first generation. Global Best uses less memory and involves less computation than Local Best.

10.1.2 Local Best PSO

Each agent only receives information from its neighbours: this is achieved by forming sub-swarms of particles that share information amongst themselves. The local best for each neighbourhood ($\hat{y}_s(t)$ for neighbourhood s) is tracked, along with the personal best for each particle. As mentioned before, neighbourhoods can be formed in multiple ways:

1. Agents are grouped by their indices: these groups are easily formed, and groups can share information from all over the search space
2. Agents are grouped by spacial similarity: this is very computationally expensive (usually k-means clustering is used), but groups stay close together and often converge faster

Neighbourhoods can also be generated through other methods, by following architectures like Von Neumann/Pyramid or even randomly. The number of neighbourhoods to be generated is another tuning lever (1 neighbourhood is the same as global best) Agents can also be part of multiple neighbourhoods at once: this spreads information between groups for faster convergence. Local Best gives more diversity to solutions and is less likely to be trapped in local minima.

10.1.3 Velocity Updates

An n-dimensional agent has n velocities, and these are all updated at the same time. With $V_i(t)$ as an agent's velocity, $X_i(t)$ as the agent's position and C_1 and C_2 as acceleration constants, the following is applied to every velocity j of every agent:

$$V_{ij}(t+1) = V_{ij}(t) + C_1 * rand(0,1) * (y_{ij}(t) - X_{ij}(t)) + C_2 * rand(0,1) * (\hat{y}_j(t) - X_{ij}(t)) \quad (42)$$

$\hat{y}(t)$ is either the local or global best, depending on the variant. The first addition is the *cognitive component* (using the agent's personal best), and the second is the *social component* (using data from other agents).

10.1.4 Position Updates

Like velocity updates, position updates happen to all agents simultaneously.

$$X_i(t+1) = X_i(t) + V_i(t+1) \quad (43)$$

10.1.5 Best Updates

First, the personal bests for all agents are updated. As optimisation problems are generally regarded as minimisation, a lower valued agent is considered as more optimal.

$$y_i(t+1) = \begin{cases} x_i(t+1) & , f(x_i(t+1)) < f(y_i(t)) \\ y_i(t) & , otherwise \end{cases} \quad (44)$$

The global/local bests are both generated the same way: local best does it once per neighbourhood, global best does it once for the entire swarm. Note that they take the best *personal best*, not the best current agent: it is entirely possible that the global/local best stays constant over multiple generations.

$$\hat{y}_i(t+1) = arg\, min_{a \in swarm}(f(y_a(t+1))) \quad (45)$$

10.1.6 Stopping Criteria

1. X iterations have been run
2. An acceptable solution has been found
3. No change in the global best after Y iterations
4. Global swarm radius (with global best at center) $\leq$ R .i.e the agents have converged
5. For local best, average sub-swarm size $\geq$ S .i.e agents have converged enough into groups
6. Rate of change of global best $(\frac{f(\hat{y}(t))-f(\hat{y}(t-1))}{f(\hat{y}(t))}) \leq \varepsilon$

10.1.7 Velocity Clamping

A control mechanism for PSO, this prevents the velocity from getting too large. As the velocity only grows at each stage (V(t+1) = V(t) + ...), after enough generations velocities will explode in size and cause agents to heavily diverge and only search the boundaries of the solution space. This can be limited with the following step after velocity updates:

$$V_{ij}(t+1) = min(V_{max\,j}, V_{ij}(t+1)) \tag{46}$$

While this stops velocities from becoming too large, it also means that after a point, the magnitude of the velocities becomes constants: this gives agents a limited number of position updates, so doesn't search the entire solution space. V_{max} can be dynamically updated:

1. If $\hat{y}(t)$ doesn't improve after X generations, reduce V_{max}, so the algorithm uses local search to find a better direction
2. Exponentially decay the maximum velocity using n_t (max number of generations) and α (user value > 0)

$$V_{max\,j}(t+1) = \left(1 - \left(\frac{t}{n_t}\right)^{\alpha}\right) * V_{max\,j}(t)$$

Alternatively, to cap the velocities at V_{max} but still allow some variance (effectively normalise them to the range [0, V_{max}]), the following formula is used:

$$V_{ij}(t+1) = V_{max\,j} * \tanh\left(\frac{V_{ij}(t+1)}{V_{max\,j}}\right)$$

10.1.8 Inertia Weight

Inertia Weight is another control mechanism for PSO to prevent velocity explosions, but unlike Velocity Clamping it doesn't outright restrict the maximum velocity. Using an inertia weight ω, the velocity update step is changed to the following:

$$V_{ij}(t+1) = \omega * V_{ij}(t) + \ldots \tag{47}$$

Usually ω is kept less than 1, to decrease the velocity memory of the system, but by making $\omega > 1$ the velocity growth rate can be temporarily increased to speed up the search. Some methods for dynamically updating ω are:

1. Randomly changing it at each generation

2. Linearly decreasing it from $\omega(0)$ to $\omega(n_t)$, where n_t is again the max. number of generations
$$\omega(t) = \omega(n_t) + \frac{n_t - t}{n_t} * (\omega(0) - \omega(n_t))$$

3. Non-linearly decreasing with $\omega(0) = X_1$ (e.g 0.9) and another user value X_2(e.g 0.4) to control the decrease
$$\omega(t+1) = \frac{(\omega(t) - X_2) * (n_t - t)}{(n_t + X_2)}$$

11 Hybrid Methods

As we've seen throughout, many of the algorithms are fairly similar in nature and thus we can apply teachings from one to the other. This section focuses on Hybrid methods for Differential Evolution, but some things can go both ways.

11.1 Gradient-Based Differential Evolution

This method uses the base DE, but adds two more steps: **Acceleration**, which uses gradient descent to converge faster without compromising on diversity; and **Migration** which increases the population diversity to escape local minima.

With the new steps, the overall flow of the algorithm is:

A Create a random initial population and control parameters

B While stopping criteria are unmet:

  1 **Apply migration if necessary**

  2 Apply mutation to each parent to generate a trial vector

  3 Make new chromosomes via Crossover with parent+trial vector

  4 Select new population from parents and offspring

  5 **Apply acceleration if necessary**

C Select the best performing chromosome as the solution

11.1.1 Acceleration

This method requires the cost function to be differentiable: the gradient of the cost function (f) is represented as ∇f, and a learning rate $\eta(t)$ is also tracked. Acceleration is done right after selection, so chromosomes from generation t and t+1 are both available. With the usual notation of $\hat{x}(t)$ as the best chromosome of a generation, a new vector x(t) is generated:

$$x(t) = \begin{cases} \hat{x}(t+1) & \text{, if } f(\hat{x}(t+1)) < f(\hat{x}(t)) \\ \hat{x}(t+1) - \eta(t) * \nabla f & \text{, otherwise} \end{cases} \tag{48}$$

The worst vector of generation t+1 is then replaced with x(t): this either duplicates the best vector, or uses gradient descent to find a potentially better one. If the second option is used and it still isn't better than $\hat{x}(t)$, $\eta(t)$ is reduced: once it becomes too small, acceleration is halted since it won't make a difference.

11.1.2 Migration

Before mutation, if the population diversity is too low, migration is applied to spread the vectors out. 'Too low' is based on two user constants: ε_1 and ε_2 and the following formulae:

$$l_{ij}(t) = \begin{cases} 1 & , if \frac{(x_{ij}(t) - \hat{x}_j(t))}{\hat{x}_j(t)} > \varepsilon_2 \\ 0 & , otherwise \end{cases} \tag{49}$$

$$diversity(t) = \frac{\sum_{i \in vectors(t) - \hat{x}} \sum_{j \in n_x} l_{ij}(t)}{n_x * (n_s - 1)} \tag{50}$$

While this looks complex, it can be boiled down fairly simply. $l_{ij}(t)$ is 1 if gene j of vector i is diverse enough from gene j of the best vector, and 0 otherwise. $diversity(t)$ is the fraction of diversity, considering every gene except those in the best vector. If $diversity(t) < \varepsilon_1$, then migration is applied.

The migration operator itself diversifies around the best vector: if it is applied it means that a majority of vectors are already close enough, so it evenly distributes them in a radius around it. This radius is defined by $(x_{min\,j} and x_{max\,j})$, which are the min and max values for gene j respectively. Using these values, the following formulae are applied to genes/vectors as often as required:

$$P_j(t) = \frac{\hat{x}_j(t) - x_{min\,j}}{x_{max\,j} - x_{min\,j}}$$
$$x'_{ij}(t) = \begin{cases} \hat{x}_j(t) + rand(0,1) * (x_{min\,j} - \hat{x}_j(t)) & , if\ rand() < P_j(t) \\ \hat{x}_j(t) + rand(0,1) * (x_{max\,j} - \hat{x}_j(t)) & , \text{otherwise} \end{cases} \tag{51}$$

11.2 Evolutionary/DE Hybrids

1. Use bin/exp crossovers in BGA/CGA/ES - as mentioned before these are similar to Uniform and Double Point crossover

2. Use CGA/ES mutation to add noise to trial vectors in DE ($X_{max\,j}$ and $X_{min\,j}$ are max/min values for gene j):

$$U_{ij}(t) = U_{ij}(t) + (X_{max\,j} - X_{min\,j}) * N_n(0,1)$$

3. Rank-Based Crossover Mutation: effectively a DE with crossover before mutation, like GA/ES. Offspring vectors are generated through crossovers, mutated, then the usual selection takes place.

11.2.1 Rank-Based DE Crossovers

The vectors are ranked according to cost ascending (i.e for any i, i+1 has a worse cost, and). For N_s there is no i+1, so index 1 is used arbitrarily (to cycle around). $U_{ij}(t)$ is assigned the better of the parent and the offspring.

$$child_{ij}(t) = X_i(t) + rand(0,1) * (X_{i+1}(t) - X_i(t))$$
$$U_{ij}(t) = arg\ min(f(X_i(t)), f(child_{ij}(t))) \tag{52}$$

11.2.2 Rank-Based DE Mutation

After mutation, selection continues as usual. The mutation is designed to occur less for higher-ranked chromosomes (rank denoted with i) and to mutate a smaller amount for later generations (denoted by t, n_t is max generations).

$$P_i = \frac{N_s + 1 - i}{N_s}$$

$$X'_{ij}(t) = \begin{cases} X_{ij}(t) & , if\ rand(0,1) > P_i \\ \begin{cases} X_{ij}(t) + rand(0,1) * e^{\frac{-2t}{n_t}} * (X_{max\,j} - X_{ij}(t)) & , if\ rand(0,1) > 0.5 \\ X_{ij}(t) + rand(0,1) * e^{\frac{-2t}{n_t}} * (X_{min\,j} - X_{ij}(t)) & , otherwise \end{cases} & , otherwise \end{cases} \tag{53}$$

11.3 Particle Swarm DE

1. Using the same population, swap/alternate between PSO and DE
2. Update PSO Personal Bests using DE Mutation.

11.3.1 PSO Personal Bests with DE Mutation

After performing the usual personal best ($y_i(t)$) updates, this method generates a new value using DE mutation and potentially updates the personal best. It uses a user constant P_m, which controls the probability of mutating the personal best to be relative to the global best. The personal best for the next generation still checks if the DE mutation produces a better vector before switching to it, so this method can't degrade the solution quality.

$$y'_{ij}(t) = \begin{cases} \hat{y}_j(t) + 0.5 * (y_{1j}(t) - y_{2j}(t)) & , ifrand() < P_m \\ y_{ij}(t) & , otherwise \end{cases} \tag{54}$$

$$y_i(t+1) = arg\ min(\ f(y_i(t)),\ f(y'_i(t)\ )$$

11.4 Adaptive Differential Evolution

11.4.1 Dynamic β and P_r

These decrease as time goes on: reduces the size of difference vectors and crossovers favour the trial vector. Uses a user variable X and max. generations n_t.

$$P_r(t) = P_r(t-1) - \frac{P_r(0) - X}{n_t} \tag{55}$$

$$\beta_r(t) = \beta(t-1) - \frac{\beta(0) - X}{n_t} \tag{56}$$

11.4.2 Self-Adaptive β

β is the parameter that controls how much effect the differential vectors have on the trial vectors. Therefore, if the population is close to converging then the trial vectors should be small: so β needs to be reduced. We can tell if the population is converging by calculating $abs(\frac{f_{max}}{f_min})$, or $abs(\frac{f_{min}}{f_max})$ where f_{max} and f_{min} are the lowest and highest costs of the population. As these values start to approach 1 we can infer the population is converging, and thus reduce β. Two values are used as the costs could be positive, negative or a mix of the two: using a single value requires more complex formulae.

$$\beta(t) = \begin{cases} max(\beta_{min}, 1 - abs(\frac{f_{max}}{f_min})) & \text{, if } abs(\frac{f_{max}}{f_min}) < 1 \\ max(\beta_{min}, 1 - abs(\frac{f_{min}}{f_max})) & \text{, otherwise} \end{cases} \tag{57}$$