Advanced Algorithms II

Todd Davies

May 24, 2016

Overview

This course will explore certain classes of algorithms for modelling and analysing complex systems, as arising in nature and engineering. These examples include: flocking algorithms - e.g., how schools of fish or flocks of birds synchronised; optimisation algorithms; stability and accuracy in numerical algorithms.

Aims

By the end of the course, students should:

- Appreciate the role of using nature-inspired algorithms in computationally hard problems.
- Be able to apply what they learnt across different disciplines.
- Appreciate the emergence of complex behaviours in networks not present in the individual network elements.

Syllabus

- PART 1: OPTIMISATION AND NATURE-INSPIRED ALGORITHMS (8 hours)
  - Introduction to optimisation algorithms (2 hours) applications for optimisation algorithms local and global optimisation methods based on derivatives 1.4 - direct search methods
  - Stochastic optimisation (2 hours) grid search, random searches, multistart Simulated annealing particle swarm
  - Evolutionary algorithms (4 hours) 3.1 - basics of genetics and evolution 3.2 - evolutionary programming 3.3 - genetic algorithms 3.4 - evolution strategies 3.5 - genetic programming
- PART 2: COMPLEX NETWORKS AND COLLECTIVE BEHAVIOUR (8 hours)
  - Complex networks are groups of systems (normally, a big number of them) interconnected in a non-trivial and non-regular way Introduction to complex networks (2 hours) Where are the networks and the complexity? Characterisation of complex networks Basic network properties and terminology. Topological analysis
  - Complex network models. The structure of the network (2 hours) Regular networks Random-graph networks Small-world networks Scale-free networks
  - Network dynamics and collective behaviour (3 hours) Distributed/local versus centralised/global The concept of self-organisation Synchronisation in complex dynamical networks Consensus over complex networks Swarm dynamics Consensus protocols Flocking algorithms
- PART 3: NUMERICAL STABILITY AND ACCURACY OF COMPUTATIONS (8 hours)
  - Introduction to finite precision computation (2 hours)
  - Floating point arithmetic (examples that include error analysis insummation, evaluation of polynomials, recurrences, and basic linear algebra)(3 hours)
  - Mixed precision algorithms (basic concept of iterative refinement,different speed of execution on different architectures, linear algebraexamples) (1 hour)
  - Numerical solution of initial value problems (explicit/implicit methods,multistep methods, consistency, stability, convergence) (2 hours)

Attribution

These notes are based off of both the course notes (found on Blackboard). Thanks to the course staff (Eva Navarro-Lopez, Milan Mihajlovic and Pedro Mendes) for such a good course! If you find any errors, then I'd love to hear about them.

Contribution

Pull requests are very welcome: `https://github.com/Todd-Davies/third-year-notes`

Contents

1 Initial Value Problems

An initial value problem is an ordinary differential equation with a given value called the initial condition, which is the value of the unknown function (that we're trying to model) at a given point in the domain of the solution.

In the notes, the example of a chef leaving a large pot of soup to cool is given. If the pot starts at $T_0 = 100$, and the desired temperature of the soup is $T_e = 20$, then how long should the chef leave it cooling?

We can't know this until we have the initial conditions, which is that the ambient temperature is $T_\alpha = 10$, and the given point in the solution domain; the chef left the soup for ten minutes and the temperature was $T_i = 60$.

The cooling of the soup is modelled by $\frac{dT}{dt}$ (change in temperature over time), and we can use Newton's law of cooling and heating to change this to: $\frac{dT}{dt} = f(T - T_\alpha)$. Since we know that the ambient temperature is larger than the starting temperature ($T > T_\alpha$) and that the soup is actually cooling (as evidenced by $T_i < T_0$), we know that $k < 0$.

We can work out the value of k from the analytical solution:

$$T(t) = T_\alpha + (T_0 - T_\alpha)e^{-kt}$$

By subbing in the values $t = 600, T(600) = 60, T_\alpha = 10$ we gan get k:

$$\begin{aligned}
60 &= 10 + (100 - 10)e^{-600\times k} \\
50 &= 90e^{-600\times k} \\
\frac{5}{9} &= e^{-600\times k} \\
log(\frac{5}{9}) &= -600k \\
k &= 9.796 \times 10^{-4}
\end{aligned}$$

To see how long the cook should leave the soup for, we can simply sub in k and sub in 20 for the desired ending temperature:

$$\begin{aligned}
T(t) &= T_\alpha + (T_0 - T_\alpha)e^{-kt} \\
20 &= 10 + 90e^{-9.796\times 10^{-4}t} \\
\frac{1}{9} &= e^{-9.796\times 10^{-4}t} \\
t &= \frac{log(\frac{1}{9})}{-9.796 \times 10^{-4}} \\
t &= 2242s
\end{aligned}$$

2 Optimisation and nature inspired algorithms

To start, lets recap some stuff you should already know; a minima is a point of a function where all points in its vicinity have a higher value than itself,and a maxima is the opposite; a point where all points in its vicinity have a lower value than itself.

Since we can invert a function by putting a minus in front of it, we don't really need to differentiate between maxima and minima, since we can always express on in terms of the other.

Global optimums are the best values for the entire domain, while local optimums are best in some bounded region.

One dimensional functions are easiest to visualise; they are just a graph, where the y value is the value of the function and the x value is the input. As the dimensionality increases, the functions get harder to visualise; 2d functions are visualised on a 2d graph, where the intensity of the colour in each square represents the value of the function at that coordinate.

An optimisation problem is one where we attempt to maximise or minimise a function. This involves finding the input that will produce the largest or smallest value. Optimisation problems are really search problems; given a domain, find an input that produces the smallest output.

If we know exactly what the function we're trying to optimise is (i.e. we have an equation), then it's an explicit problem, but if we just have some inputs and outputs, then it's a black box problem.

Similarly, there are two different approaches to solving these problems:

Single solution:
: This is where there is a single candidate solution that is incrementally improved by the algorithm throughout the procedure.

Population based solution:
: This is where there is a set of candidate solutions (the population) and an iterative operation combines the best ones to improve the quality of the population.

All of the optimisation methods follow a cycle:

- Guess some parameters for the initial solution
- While we're not satisfied:
  - Evaluate how the current parameters perform
  - If we're satisfied, then we've finished
  - Otherwise, then determine new parameters based on the current ones and their evaluation.

We can use derivatives to try and find maxima and minima in a function. If we do not have an explicit function for the derivative, then we can calculate them using **finite differences**:

Forward difference:

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

Central difference:

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

Here, the h parameter is the range over which we're calculating the differential.

2.1 Minimising univariate functions

For one dimensional functions, we can use two approaches:

- Interval reduction, where we iteratively reduce the range of values that we think the optimum is inside.
- Interpolation, where we try to find an approximate function and use the optimum of that function to iterate to find a better approximate function.

2.1.1 Bisection algorithm

This is the easiest algorithm, we start with the full range in the domain, and then cut it in half based on the value of the differential at the midpoint at the range.

```
bisection(f'(x), a, b, e) {
  while (b - a >= e) {
    c = (a + b) / 2;
    if (f'(c) < 0) a = c;
    else if(f'(c) > 0) b = c
    else return c;
  }
}
```

2.1.2 Quadratic interpolation algorithm

Here, we make a quadratic function that goes through three points a, b and c, then we either drop a or b depending on which way we should move the quadratic function. Here, **we do not need the differential** which is good since we won't always have it!

```
Sudo insert psudo code
```

2.1.3 Stopping criteria

Eventually, we will have to stop our optimisation algorithms, but some of them will run indefinitely. We need criteria to make them stop after a sane amount of time. The simple approach is to stop after n iterations, and is applicable to all iterative functions. The more clever approach is to only stop when we reach a certain threshold, i.e. when a method converges asymptotically on an optimum.

2.2 Minimisation of multivariate functions

There are two different classes of minimisation methods for multivariate functions:

Based on derivatives:

Moves are determined based on information from derivatives of the multivariate function.

A *directional derivative* is a derivative indicating the rate of change in a specific dimension:

$$\Phi'_r = lim_{h \to 0} \frac{f(x + hr) - f(x)}{h}$$

Where r is the direction. If Φ'_r is positive, then the direction is ascending, if it's negative, then the direction is descending and if it's equal to zero, then there is no slope.

Finding the gradient of a vector of a multidimensional function $f(x)$ (and hence the vector of all partial derivatives) is:

$$\frac{\delta f}{\delta x_j} = lim_{h \to 0} \frac{f(x + he_j) - f(x)}{h}$$

e_j is the unit vector in the direction of axis j, so to construct the whole gradient for the vector, we need to do:

$$\Delta f(x) = \left[\frac{\delta f}{\delta x_1}(x) \quad \frac{\delta f}{\delta x_2}(x) \quad \ldots, \frac{\delta f}{\delta x_n}(x) \right]^T$$

The **Hessian** is the matrix containing the second order partial derivatives, and is always of size $N \times N$:

$$Hf(x) = \begin{bmatrix} \frac{\delta^2 f}{\delta x_1 \delta x_1}(x) & \dots & \frac{\delta^2 f}{\delta x_1 \delta x_n}(x) \\ \vdots & \ddots & \vdots \\ \frac{\delta^2 f}{\delta x_n \delta x_1}(x) & \dots & \frac{\delta^2 f}{\delta x_n \delta x_n}(x) \end{bmatrix}$$

Most algorithms have some kind of *parameter update scheme*, where they will move in a given direction r for a length of α every iteration, and r, α change based on the current position:

$$x_{i+1} = x_i + r_i \alpha_i$$

2.2.1 The alternating variable method

A conceptually easy method, you start from an arbitrary position x, and then for each dimension d:

- Find the differential for that dimension
- Find the minimum according to that differential
- Move the value of x_d to be equal to that minimum.

Then repeat for all dimensions until no progress is made.

2.2.2 Newton's method

As soon as Newton invented calculus, he also started inventing methods for finding minima. This method requires a function to be twice differentiable:

$$x_{i+1} = x_i - \alpha \frac{\Delta f(x_i)}{Hf(x_i))}$$

The trouble with Newton's method is that it will only converge if the initial point is close to the solution, and the memory requirements for the Hessian matrix is n^2.

Direct search:
Moves are determined using methods other than derivatives, for example, using geometric concepts.

2.3 Simplex methods

If you did Advanced Algorithms 1, you're either groaning now or you're jumping for joy at being able to skip a section. However, these aren't the simplex methods you learned about last semester.

For this, you construct a simplex (a shape of $n + 1$ vertices in n dimensions), and then observe that you can always form a new simplex from an existing one by adding a new point.

The **Spendley, Hext and Himsworth method** is like so:

1. Create $n + 1$ vectors for a regular simplex (where all sides are the same length).
2. For each vector, the value of the objected point is calculated for that point.
3. The vector with the highest point is removed and substituted by a new one.

4. If the worst vector is also the most recently introduced one, then the next worst one is used.
5. The newly substituted vector is the mirror image of the old one along the axis of the remaining vectors.
6. When the simplex simply rotates around a point, then the vertex in the middle is the minimum.

The maximum age of a vector is $1.65n + 0.05n^2$, and if a vertex exceeds age a, then the search stops, or is restarted using a smaller simplex. If the search was restarted, then the initial (smaller) simplex is started from the site of the rotating.

The **Nelder and Mead method** is similar, though the simplices are no longer regular (though they can be). It involves having rules that either expand, contract or reflect the simplex (or any combination of them), which leads to quicker results and better approximations of the solution.

Where $y_0^{(r)}$ is the best value in the previous simplex, and $y_{n-1}^{(r)}$ is the worst, the new simplex results from:

$y_0^{(r)} < y_i^{(r+1)} < y_{n-1}^{(r)}$:
The simplex is reflected like in the SHH method.

$y_i^{(r+1)} < y_0^{(r)}$:
The simplex is reflected and expanded.

$y_i^{(r+1)} > y_{n-1}^{(r)}$:
The simplex is reflected and contracted.

2.4 Optimisation and nature inspired algorithms

As we have seen, a global optimum is the best set of admissible conditions to achieve an objective (i.e. minimising the function) under a set of constraints.

Finding such an optimum is non-trivial, mainly because the search space is often continuous, so there are an infinite number of parameters to test. The following algorithms try to find global optimums:

Grid Search:
We overlay a grid onto the search space (in as many dimensions as we need to), and test each point on the grid. We can change the resolution of the grid by making the space between the grid lines larger or smaller. The algorithm scales exponentially in the number of dimensions, but linearly in the number of points tested (dependent on the grid resolution), therefore the runtime is $O(n^d)$.

The quality of the solution depends on the density of the grid, but getting a high quality solution is usually impractical, since the search space grows so quickly (even in two dimensions, it is quadratic).

Random search:
Here, we simply generate n random points and sample the objective function at these points. The point that gives the best value for the objective function is remembered for possible later use (since that's the optimum). Obviously this algorithm runs in $O(n)$ time, and its quality depends on the number of random points chosen. It is possible that the random search method could outperform all the other methods if it got lucky, though this is unlikely.

Due to its ease of implementation, random search is often used as a reference for other methods.

Multistart:

Here, we guess n random points again, but for each one, we find the local minimum. This improves the quality (since there's more chance of finding the global minimum, since it's definitely a local minimum too), but it can degrade performance.

This method often visits the same local minima multiple times (if optimisations aren't made to avoid doing this) so it can be inefficient.

Random walk:
We can generate a single random start point, but then generate subsequent points by choosing a random direction and length for a vector and adding that to the current point.

Using nature to help minimise:
If a crystal is being formed, and it cools slowly, then the atoms in the crystal will vibrate around until they are in a local minima, which results in some cool shapes. We can use techniques to emulate this to find minima in our functions.

Furthermore in genetics, non-optimal genes are 'selected out' of the population by natural selection and evolution, leading to a genotype that is optimum for the environment.

The remaining algorithms use nature as inspiration:

Metropolis algorithm:
This is an algorithm that emulates an ensemble of particles in equilibrium at a certain temperature. It uses the Boltzmann probability density function:

$$p.d.f = e^{\frac{-E}{k_B T}}$$

To give the probability that a certain particle configuration with an energy E has a certain temperature T.

In nature, perfect crystals are formed by the cooling down from being molten very slowly so that the material can reach an equilibrium at each temperature. If the cooling is too fast, then an equilibrium will not be reached at each temperature, and the crystal will have imperfections.

Local optimisation algorithms have parallels with letting crystals cool too fast!

The algorithm is as follows:

- Start with a random position for an atom x^0
- Create a small random displacement to obtain x^1 and calculate the difference in energy $\Delta E = E^1 - E^0$.
- If $\Delta E < 0$ accept the new position, otherwise accept it with a probability of $P(\Delta E) = e^{\frac{-\Delta E}{k_B T}}$
- Iterate a lot of times, simulating the thermal motion of particles in a heat bath of temperature T.

The Metropolis algorithm works because it lets the energy of the particle increase even though that's a 'step back' in terms of optimisation. Sometimes the algorithm needs to get out of a local minima in order to find the global optimum.

Simulated Annealing:
This takes advantage of the fact that we can see the energy of a particle was similar to the value of an objective function we're trying to maximise, and the coordinates that it is at as similar to the parameters of the function we're finding an optimum for.

The only other variable is the temperature, which acts as a control parameter with the same units as the objective function. Simulated annealing starts by melting the objective function at a high temperature, then using the Metropolis algorithm to calculate the equilibrium of the objective function at temperature decreases. Here's the algorithm:

- Start with a high temperature T^0 at some random position $x^{(0,T^0)}$.
- Apply the Metropolis algorithm to determine the average equilibrium value of the objective function and parameter values $x^{(Ex,T^0)}$.
- keep reducing T and repeating the previous step, and only stop when the function freezes; you've reached the global minimum.

Note that the start position is irrelevant, since there are many iterations required for each temperature, and an equilibrium is found for each. The start temperature, T^0 is very important though; if its too low, then a local minimum will be found, and if its too high, then the algorithm will take too long.

Random and grid searches are also asymptotically complete, but they converge at much slower rates.

Simulated annealing can be reformulated computationally as a Markov Chain, and a proof has been given that states the algorithm will converge to a global optimum in infinite time, which makes simulated annealing *asymptotically complete*.

Press' modification:
In 1989, Press et al. suggested that each temperature cycle should last for a predetermined number of iterations N. After each cycle, the temperature should be reduced by a constant factor p:

$$T^{n+1} = pT^n \quad < 0 \leq p \leq 1$$

If after a cycle has finished, there have been no successful moves, then the algorithm stops. If we increase N, then the accuracy increases, but the execution time increases faster too. Similarly if we increase p, the solution will improve (and the reliability too), but the rate of cooling drops so it takes longer.

There is also a slight modification to the displacement step; a single parameter is changed to a random value within the boundaries of the parameter.

Corana's modification:
Corana et al. suggested that the displacement step should be based off a vector; each parameter is changed according to the same coordinate of a vector v_i. After N_s rounds, v_i is changed so half the new candidate parameters are changed. Just like Press' modification, after N_r rounds, the temperature is reduced.

As a result, the annealing will adapt to the shape of the function, but it is also slower; N in Press' algorithm is equal to $N_s \times N_r$ in Corana's algorithm.

2.5 Evolutionary algorithms

Natural populations evolve by having variation amongst the members of the population and having selection (or unfit members of the population dying before they get chance to reproduce). Evolutionary algorithms are a class of optimisation algorithms that evolve candidate solutions as a group (ensemble) rather than looking at one solution at a time.

If you took a biology course at some point, then you may be able to figure this out, but there is a mapping from biology terminology to CS terminology:

- Gene - Parameter
- Chromosome - All parameters
- Individual - Candidate solution
- Generation - One iteration
- Fitness - Value of the objective function

Evolutionary algorithms evolve their populations. This means there will be n points in the parameter space, and each iteration will drop some of the points and create new ones (in hopefully better locations). Each individual in the population is a single candidate solution, and is stored

as a *chromosome*. This is a sequence of bits, split into m sections called *genes*. As described above, each gene maps onto a parameter.

Fogel's algorithm for evolutionary programming is as such:

1. Generate a random initial population of n individuals
2. Calculate the fitness of each individual in the population (using the objective function)
3. Each individual from the current population generates and offspring by copying its own genes
4. Mutate each gene for each chromosome in the offspring by a small variance.
5. Put the offspring in a new population, $2n$ large
6. Probabilistically select n individuals as a function of fitness to be removed (now we have a population of n again).
7. Go back to step 2, or stop.

In this algorithm, we described mutation (in step 4) and selection (in step 6). For mutation, we add a small random number to the original value of a gene.

For selection, we want to select n individuals, where better individuals have a greater chance of remaining in the population. A roulette wheel is a manual way of doing this, but stochastic ranking (e.g. a sort that has a chance of being wrong) or a tournament where each individual is compared against a small number of others and receives a score based off of its performance; the n lowest scoring individuals are pruned.

Holland and DeJong's Genetic algorithm is as follows. It encodes parameters in binary, and genes are therefore strings of binary digits.

1. Generate n random individuals for the population
2. Calculate the fitness of each individual in the population
3. Choose two parent individuals from the current population as a probability of their fitness
4. Cross them over (see below) at a random locus to produce two offspring
5. Mutate each locus (gene) in the offspring with a small probability
6. Put the offspring in the new population
7. Go to step two, or stop.

The mutation operation flips bits in a gene with a small probability. The cross over operator swaps the latter halves of two chromosomes at a random index.

The differences in the two algorithms are as follows:

Evolutionary programming	**Genetic algorithms**
Genes are encoded as floating point numbers	Genes are encoded in binary.
Mutation is addition of a random value	Mutation is bit flipping.
Asexual reproduction (two parent makes one child)	Sexual reproduction (two parents make two children.)
No cross over	Cross over by swapping subsections of the chromosome.

Rechenberger and Schwefel made another algorithm:

1. Generate n random individuals
2. Calculate the fitness of each
3. Randomly pair up individuals
4. Recombine them to create two offspring

5. Mutate each gene locus with a small probability
6. Put the offspring in the new population ($2n$ large)
7. Select the best n
8. Go back to step 2, or stop.

I should figure out how std. deviation incorporates in here...

It is important to note the differences between selection and variation so we can get the balance right:

- Selection is responsible for keeping improvements in the population and not discarding them
- Very strong selection limits the variation and all individuals will be the same
- Mutation and cross over are responsible for introducing variation and will make the parameters/gene pool change
- Very strong variation results in good solutions, but they will be replaced by random ones and not retained.

Many genetic algorithms progress at very slow rates, but then have short bursts of very rapid progress. In general, it is not common for these algorithms to have proven convergence properties, but some of them do when applied to simple problems.

Deciding when to stop a genetic algorithm isn't always simple; stopping after a predetermined number of generations requires lots of trial and error to get the number right, waiting until a desired fitness is reached is okay if you know what the optimal fitness is, but it may take an (infinitely) long amount of time, and waiting for the fitness variations from one population to the next to become stable isn't a good idea either because of the propensity for genetic algorithms to progress very slowly then have a burst of progress.

2.6 Genetic Programming

With evolutionary programming, we optimised the parameters of the input function in order to find the ones that produce the best solution. On the other hand, genetic programming optimises the function until it fits our purpose. This may seem like a strange approach, but when you consider that any computer program is just a big mathematical function, then genetic programming just becomes automatic computer programming.

In this sense, programs evolve to become better at solving the problem they are being evaluated against. If you think about it, programmers have two 'phrases' when to producing code; developing new code, and editing existing code. This maps cleanly onto genetic programming, since the algorithms first generate entirely new pseudo-random code, and then mutate/cross it over with other code.

We can represent programs as trees, where internal nodes are functions (which includes arithmetic/logical operations) with arguments, and where leaf nodes correspond to constant values, functions with no arguments, or inputs to the program.

The genetic algorithm must be given a set of functions and a set of terminals that it can use to generate the tree. Example functions include:

- Boolean functions (`AND`, `OR`,...)
- Arithmetic functions (+,%,...)
- Transcendental[1] functions (sin, cos, log,...)
- Conditional statements (if, switch, etc)

[1] A transcendental function is one that cannot be described with a finite sequence of algebraic operations; i.e. it transcends algebra.

- Loops (while, for, etc)
- Variable manipulation (set, etc)
- Subroutines[2] (http-request, read-sensor, etc)

The set of terminal nodes is smaller, and can include constants[3], functions such as `rand()`, variables etc.

In order to create valid programs, the genetic algorithm should implement some type checking to ensure that only valid inputs are given to functions. This may include automatic type conversions to increase the number of valid trees that can be generated.

The algorithm must also attempt to stop things like `RuntimeException` occuring; i.e. the program must produce some usable output. This can be done by having functions return a default value if they fail with invalid inputs[4], or by making functions that fail score very low when they are evaluated (so that they are selected out of the population).

2.7 Cramer & Koza's Genetic Programming

As previously stated, the program is encoded as a tree, using functions and terminals. Cramer and Koza's algorithm is as follows:

1. Initialise a random population of n individuals
2. Calculate the fitness of each individual
3. Select two parents at random proportionally to their fitness and reproduce them together, producing two children
4. Apply cross over with a certain probability
5. Apply mutation with a certain probability
6. Remove the parents
7. Loop back to step 2 unless satisfied.

While the algorithm may seem well specified, there are a few additional things we need to clear up; namely initialisation and mutation/cross over.

Initialisation is done using one of two methods:

Full method:
Choose a function node at random, and either fill its children with terminal nodes if we have reached the required depth, or loop and choose more function nodes.

Grow method:
Choose a function or a terminal node at random, and fill its children with child nodes if the maximum depth has been reached, or loop back and choose child or function nodes.

Both of these methods generate trees of up to a specified depth, though the full method always generates a tree where all branches are that depth, while the grow method generates branches *up to* that depth.

Mutation is applied to single individuals; a node is chosen at random and is substituted with a randomly generated sub-tree. Different implementations generate the sub-tree differently, some will maintain the depth of the tree to avoid it growing too big or generate a tree that can be used as input for the parent node (to avoid creating an invalid tree post-mutation), while others have no such constraints.

[2] Think of these like library calls. The evolutionary algorithm is really just trying to string these together.

[3] Random constants are determined at the start of the program (i.e. once per run).

[4] Type checking can't catch everything (though some functional languages have type systems that integrate proof systems and can check all kinds of funky stuff; Google for 'dependent types'), divide by zero is an example of this.

Cross over is when children inherit information from both their parents. For our purposes, this involves making a tree with a sub-tree from one parent and a sub-tree from another parent. The node within each parent to cross over at is chosen at random (meaning that there could be a case when most of the tree is inherited from one parent and little of it is inherited from another.).

2.8 Bloat

Bloat is when programs created by genetic algorithms get larger and larger trees without significant improvement to the fitness of the program for the function it is trying to perform. This is undesirable, since more memory is used, the expressions are evaluated slower and the population evolves more slowly.

Bloat is caused by a variety of sources, such as when cross over is very unbalanced, and through random mutations. Bloat often 'survives' and accumulates because it is neutral to fitness; many bloated sub-trees will have no effect on the fitness (these are celled introns).

Examples of introns include expressions like $x - x$, x && true etc (basically the identity functions).

Controlling bloat is hard; but we can do so in a few different ways:

- Enforce tree size and depth limits, maybe by returning a parent of a tree if its child violates a limit.
- Anti bloat operations could be implemented such as making sure that the second subtree in cross over is small enough to not violate size constraints, or by making sure that a mutated tree is of the same size as the original one was.
- Selection can be used to reduce bloat; the fitness of trees can be reduced proportional to their size (parsimony pressure), or the fitness of trees violating the size constraints could simply be set to zero (the tarpeian method).

All evolutionary algorithms have a requirement for diversity and variability in their population in order to evolve. When all individuals in a population are similar, cross-over won't effect much change and mutation is unlikely to move individuals far in the 'solution space'. Unfortunately, due to selection, populations tend to evolve to be relatively uniform.

There are attempts to make evolutionary algorithms more similar to nature in the sense that there can be multiple populations of the same species in different areas with little mixing between them. Algorithms mimicking this effect are called *Distributed EA's*, and evolve multiple populations in near-isolation (some mixing will occur) in order to try and get heterogeneous sets of effective genes. Some swarm algorithms are an example of this (multiple swarms can be present in the solution space).

3 Complex Networks

Eva's notes and lectures are really very good, use her notes, not mine :)