Lecture 1:

2 books:
- R. Poli et al, A Field Guide to Genetic Programming
- S. Luke, Essentials of Metaheuristics


Genetic Programming (GP)

- In a nutshell, using evolutionary algorithms to design computer programs
  - Or other ‘executable structures’, e.g. circuits, equations
  - Generally small programs that do specific things
  - So we wouldn’t expect to evolve Microsoft Office

Genetic Programming (GP)

In a nutshell…

- Create a population of random programs
- Then repeat:
  - Evaluate them
  - Kill off the (really) bad ones
  - Keep the (relatively) good ones
  - Use them to breed the next generation (by using mutation and recombination operators)
- Until the problem is (hopefully!) solved

Genetic Programming (GP)

Why use evolutionary algorithms?

- Good at solving global optimisation problems
- Flexible in how solutions are represented
- However, focus on EAs is in part historical
- Other optimisers may, in principle, be used

Also a slightly iffy bio-inspired argument

- Biological systems are evolved
- Biological systems are, in a sense, complex computers
- Therefore complex computations can be evolved
Why do we want to evolve programs?

- Sometimes because we’re lazy!
- More often because we don’t know how to write a program to solve a particular problem
- Or we want to do better than an existing solution

Evolutionary ‘black box’

- Often portrayed as a form of automatic innovation
  - “Humies” is an annual contest for human-beating results
  - $10,000 in prizes every year

- Previous Humies winners include:
  - Games controllers
  - Circuit designs/designers
  - Image analysis algorithms
  - Software engineering tools
  - Medical diagnostics tools
Genetic Programming (GP)

- There are a number of varieties of GP
  - You’ll see lots of these over the coming lectures

- They differ in how they represent programs
  - Syntax: control structures, modules, language
  - Also their degree of bio-inspiration

- Representation is important
  - The programs we write are fragile
  - Imagine “mutating” one ➔
  - Can we remove this fragility??
    (this is a big research question)

Evolvability

This is the capacity for a program to improve its fitness as a result of an evolutionary process (i.e. mutation and recombination).

For genetic programming, there’s little value in being theoretically able to express a program if it can not be discovered by evolution.
Koza Tree-Based GP

- Invented by John Koza
  - Also invented the scratch card
  - Earliest successful form of GP
  - (Though arguably not the first)
  - Still the most widely used form

- Programs are represented by trees
  - Also known as syntax trees or parse trees
  - Nodes are sampled from a function set
  - Leaves are sampled from a terminal set

Parse Trees

```
+ 
  \times 
  \times 
  y 5 
  t sin
  \theta

if
  <
  lRsensor
  10
  left
  right
  move

progn3
WRITE
  2
WRITE
  0
READ
  4
READ
  2

forwhile
  +
  9
  WRITE
  2
  3
  1
  i
```
To create a mathematical expression

- Function set = \{ +, -, \times, \div, \sin, \cos \}
- Terminal set = \{ y, t, \theta, \text{constant} \}

- e.g. \((y/5)\times(t \sin \theta)\):

```
      ×
     /   \
    ÷       \
   /         \
  /           \
 /             \
/               \

y  5  t  sin
```

- Other initialisation methods exist
  - E.g. ramped half-and-half: see Field Guide! θ
Recombination

Sub-tree crossover:

Child 1

Child 2

Mutation

Sub-tree mutation:
Sub-tree mutation:

```
Parent
  +
  /  
/    
+     
|     
|     
|     
y   t   sin
  5    θ

Child
  +
  /  
/    
+     
|     
|     
|     
y   t   θ
  5
```

Point mutation (less disruptive):

```
Parent
  +
  /  
/    
+     
|     
|     
|     
y   t   sin
  5    θ

Child
  +
  /  
/    
+     
|     
|     
|     
y   t   sin
  5
```
Fitting a mathematical expression to data
- A common use of genetic programming
- Useful when little is known about the generating function

Curve Fitting Example

https://www.youtube.com/watch?v=37D3QpFvrgs
Symbolic regression is a popular application of GP
  ▶ But mathematical expressions aren't programs
  ▶ Or at least, not very interesting programs!

Programmatic expressions also typically have:
  ▶ Command sequences: command; command; ...
  ▶ Conditional execution: if ... then ... else
  ▶ Iteration: for ..., do ... while
  ▶ Memory, variables: int i = 0 ...
  ▶ Functions, modules: foo = bar(x, y)
Santa Fe Trail Problem
- A control problem commonly used to benchmark GP
- Guide an ‘ant’ to ‘eat’ all the ‘food’ in minimum time

Function and terminal sets
- Functions: \{ if-food-ahead, progn2, progn3 \}
- Terminals: \{ left, right, move \}

```
  if_foodAhead
     /     \
    /       \
   move     progn2
          /     \
         /       
        right   if_foodAhead
               /     \
              /       
             move     right
```
Crossover Problem

Parent 1

```
if IRsensor < 10
  left right move
progn2
```

Child 1

```
if
  progn2
    right move
    progn2
    left
  progn2
    right move
```

'progn2' has a return type of void

'if' expects a return type of Boolean

Parent 2

```
if IRsensor 10
  left right move
progn2
```

Child 2

```
if
  progn2
    <
      IRsensor 10
      IRsensor 10
```

`
Traditional tree-based GP requires closure

- All functions must be able to do something with whatever input they may receive
- i.e., their input types must be more general than any other function or terminal's output type

Function set with closure – good 😊
- \{ \text{AND, OR, NAND, NOR, NOT} \}

Function set without closure – bad 😞
- \{ +, -, \text{AND, OR, progn2, sin, cos} \}

Can we avoid closure?

Penalise invalid solutions

- A common approach in EAs
- Easy to implement
- Can lead to search space bias
- Inefficient use of population if invalidity occurs often

Repair invalid solutions

- Another common EA approach
- Maintains population efficiency
- Can be time consuming

Fitness = 0 😞

Mutate sub-tree until valid
Type-Constrained Operators

- Constrain initialisation and variation operators
  - By taking into account the return types of branches
  - e.g. only allow crossover points at type-compatible points
  - The preferred approach to handling mixed types in GP

```
compatible crossover points
```
A variant of GP [Montana, 1995]
- Builds upon the idea of type constraints
- Every terminal and function is assigned a type
- Provides scope for type hierarchies
- Also supports generic functions with flexible types
- Paper discusses mixing scalars, vectors and matrices:

Bloat is a big problem for genetic programming
- Tendency for trees to grow large during evolution
- In standard GP, growth has quadratic complexity
- Leads to inefficient uninterpretable programs

From Langdon, 2003, Quadratic Bloat in Genetic Programming

Many theories for why bloat occurs:
- There are more big programs than small programs
- GP operators tend to explore larger trees (operator bias)
- Programs protect themselves with non-functional code
  - [See §11.3 of “Field Guide”]

For small programs, uncontrollable big programs (Langdon, 1998)

There are various ways to control bloat
- Easiest way is to apply depth constraints
  - e.g., only pick crossover points below depth N
- Parametric pressure involves penalising large programs
  - e.g., subtract a term from their fitness in proportion to size
- Code editing involves removing parts of large programs
  - e.g., remove the bits that don’t do anything
- An extra objective can be added to a multiobjective EA
  - e.g., second objective of minimizing number of nodes

For more info, read [Luke, 2006]
- http://dl.acm.org/citation.cfm?id=11681881368

Lecture 2:
A framework for evolutionary computing

- Supports common evolutionary algorithms
  - GAs, evolution strategies, GP, PSO, ...
  - You just need to implement a `Problem` subclass

- Individual components are configurable
  - Using parameter files
  - Representations, operators, selection mechanisms

- Relatively easy to evolve non-standard things
  - New representations subclass `Individual` and `Species`
  - New variation operators subclass `BreedingPipeline`

Expressiveness

This is the capacity for a program representation to express different kinds of behaviours.

For genetic programming, you can’t evolve a program if you can’t express it.

In practice, there is often a trade-off between expressiveness and evolvability.
Adding Memory

- There are various ways of adding memory to GP
  - However, in practice these are not widely used

- Consider the approach used by Astro Teller [1994]
  - This introduces a memory, of width $M$:

    
    \[
    \begin{array}{ccccccccc}
    0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & \ldots & M \\
    \end{array}
    \]

  - And 2 new functions:
    - READ($X$) – read value from memory location $X$
    - WRITE($Y$, $X$) – write value $Y$ to memory location $X$

Consequence: Usual problems of accessing memory that is not there – crash!

Data Structures

- Using addressable memory is problematic
- An alternative is to use data structures
  - These are less expressive than ‘full memory’
  - But less susceptible to bad mutations
  - e.g. using a stack:

```
progn3
  /
 /  
PUSH  PUSH  POP
     /
  /
PUSH +
   /
  /
2 + 2
     /
POP
```
Other Memory Approaches

◊ “Soft memory” [Poli et al 2009]
  ▶ Blends new assignments with old memories
  ▶ Intended to be more evolvable
  ▶ http://cswww.essex.ac.uk/staff/poli/papers/PoliMcPheeCitiCraneJAEA2009.pdf

◊ Cultural memories [Spector & Luke 1996]
  ▶ Memory is shared amongst the population
  ▶ Allows programs to communicate

Adding Loops

◊ This is possible, but…
  ▶ Requires caution!
  ▶ Halting Problem: can’t predict termination
  ▶ Need to use constraints to prevent infinite loops
  ▶ (e.g. max iterations)
  ▶ Or use a time-out during solution evaluation

Alan designed the perfect computer
http://www.cooperatens.com
Recap

◊ It is possible to add syntactic features to GP:
  ◦ Types ✓
  ◦ Memory ✓
  ◦ Loops ✓

◊ But this has consequences:
  ◦ More complex initialisation and variation operators
  ◦ More constraints during evolution
  ◦ Possible biases within the search landscape

  ◦ So, only use them when necessary
A program is defined as a forest of trees
- One “result producing branch” (RPB), i.e. main()
- One of more ADFs, each with one or more arguments

![Diagram of ADFs and RPB](image)

ADF's must be defined before a GP run
- Both the number of ADFs, and the number of arguments for each ADF must be specified in advance
- This is a prominent limitation of ADFs

Some heuristics for choosing these values:
- *A priori* knowledge of problem decomposition
- Over-specification, i.e. more than will ever be needed
- Affordable capacity, since ADFs tend to increase the complexity and hence the execution time of programs

Good things about ADFs
- They can reduce overall program size
- They make it easy to solve modular problems
- They have been used to solve hard problems
  - See Koza’s books!

Bad things about ADFs
- They can increase program complexity
- Incorrect parameter settings hinder evolution
- Modular dependencies may hinder evolution...
Modular Dependencies

- This is also seen in software engineering
  - Two routines make use of shared code
  - The routines’ requirements diverge
  - The shared code must be copied and modified
  - i.e. it is no longer shared code!

Removing Dependencies

- Koza introduced a similar mechanism to GP
  - Called ‘case splitting’
  - Part of a group of ‘architecture altering operators’
  - Basically mutation operators that refactor code
diamond Another group of methods for achieving modularity
  - These attempt to identify useful code blocks
  - And then turn them into modules
  - For potential use elsewhere in evolving programs

diamond Adaptive Representation through Learning (ARL)
  - A successful form of analytical modularity
  - First, it identifies programs with improved fitness
  - Then extracts the most-executed code blocks
  - Placing these in a library shared between programs

diamond Recap

diamond It is possible to add syntactic features to GP:
  - Types ✓
  - Memory ✓
  - Loops ✓
  - Modularity ✓

diamond But this has consequences:
  - More complex initialisation and variation operators
  - More constraints during evolution
  - Possible biases within the search landscape
  - So, only use them when necessary
Some limitations of GP are due to using trees:

There are advantages to using graphs instead

- Instant reuse!
There are a number of graph-based GPs
- PADO, PDGP, GNP, CGP, ...

Cartesian GP (CGP) is the best known [Miller]
- Functions are arranged on a Cartesian grid

Other notable properties of CGP
- Constrained grid limits program size (no bloat!)
- Mutation can connect/disconnect nodes
- Disconnected nodes are a form of redundancy
- Redundancy has evolutionary advantages
Other notable properties of CGP

- Constrained grid limits program size (no bloat!)
- Mutation can connect/disconnect nodes
- Disconnected nodes are a form of redundancy
- Redundancy has evolutionary advantages

Lecture 3:
Real Languages?

- Tree GP doesn’t use standard languages
  - Requires special interpreters
  - Challenging to integrate with existing code
  - Typically not Turing-complete
  - Language features appear a little *ad hoc*

- Other approaches do use conventional languages
  - Machine languages (e.g. x86 assembler): **Linear GP**
  - Imperative languages (e.g. C): **Grammatical evolution**
  - Functional languages (e.g. Haskell): PolyGP
  - Logic languages (e.g. Prolog): DCTG-GP

Linear GP

- Evolves lists of machine language instructions
  - So, works a lot like a genetic algorithm
  - Variation operators less likely to break syntax
  - Often no need for an interpreter or a compiler
  - However, semantic (e.g. runtime) errors are possible

```
push r1 mov r1, r2 mov 2, r1 mov r1, r2 pop r2 ...
```

*mutate*

```
push r1 mov r1, r2 mov 2, r2 mov r1, r2 pop r2 ...
```
FINCH [Orlov & Sipper 2010]

- Special crossover/mutation operators preserve type-compatibility and stack depth-compatibility:

Parent A

Parent B

Offspring x

(incorrect)

Offspring y

(incorrect)

Offspring z

Evolves lists of grammar transitions [Ryan 1998]

- Programming language is expressed by a grammar
- Described in Backus Naur form
- GE can then evolve any program in that grammar
Grammatical Evolution (GE)

\[
\begin{align*}
<\text{exp}> & ::= \ 0<\text{exp}>\langle\text{op}\rangle<\text{exp}> \mid 1\{<\text{exp}>\langle\text{op}\rangle<\text{exp}>\} \mid 2<\text{pre-op}><\text{exp}> \mid 3<\text{var}> \\
<\text{op}> & ::= \ 0\text{AND} \mid 1\text{OR} \mid 2\text{XOR} \\
<\text{pre-op}> & ::= \ 0\text{NOT} \\
<\text{var}> & ::= \ 0\text{IN1} \mid 1\text{IN2} \mid 2\text{IN3}
\end{align*}
\]

At each step, match the left-most unmatched term

\[
\begin{align*}
\text{Evolved} = & \langle\text{exp}\rangle \\
\langle\text{pre-op}\rangle & \langle\text{exp}\rangle \\
\text{NOT} & \langle\text{exp}\rangle
\end{align*}
\]

10 18 41 27 6 39 21 47 13 31 19 8

18 mod 1 = 0

Number of transitions in current grammar rule

Apply transition 0 to the current term

A flexible and expressive approach
  - Since grammars can be defined for all languages
  - Programs have been evolved in C, for example

Though modern languages are problematic
  - Complicated syntax and large APIs
  - Typically only a subset of a language is used

Some concerns about evolvability
  - Sensitive to mutations at the left of a chromosome
  - These have a large effect upon the final expression
Indirect Encodings

♦ Evolve something that constructs something else
  ▶ Grammatical evolution does this, in a limited sense
     i.e. evolve grammar transitions that construct a program
  ▶ You also saw it with genetic algorithms
     e.g. rule sets that construct bin packing solutions

![Diagram: EA trains Constructor builds Construct](image)

Developmental GP

♦ Developmental GPs also use indirect encodings
  ▶ Evolve programs that construct other entities, e.g.
     structures, circuits, or other programs
  ▶ Often inspired by biological models of development
  ▶ Often more scalable, able to solve bigger problems

![Diagram: EA trains Program (genotype) builds Construct (phenotype)](image)

This is often referred to as a ‘developmental process’ or a ‘genotype-phenotype mapping’
Scalability

- Scalability is a problem for genetic programming
  - Search space grows exponentially with program size
  - Development allows programs to be compressed
  - Particularly when solutions have repetitive structure:

Even-4-parity circuit:

http://tams-www.informatik.uni-hamburg.de/applets/hades/webdemos/index.html

Even-8-parity circuit:

Example: Self-Modifying CGP

- CGP programs that modify themselves [Harding'11]
  - Contains normal functions and self-modifying functions
    - E.g. delete nodes, duplicate nodes, change connection …
  - A new program is created at each program iteration
    - Can generate general solutions, e.g. even-n-parity circuit:

Colours represent different functions in the CGP program

Step 1 solves for 2 inputs
Step 2 solves for 3 inputs
This program solves 12-input parity
Autoconstructive Evolution

- Programs build their own offspring [Spector 2010]
  - i.e. they include code to copy and modify themselves
  - Based on the idea that programs can learn good patterns or regions of variation, for instance mutation hotspots
  - e.g. Autopush: autoconstructive evolution using Push
  - Modifies the program’s “code” stack to construct a child

```python
((integer.stackdepth (boolean.and code.map)) (integer.sub
integer.stackdepth (integer.sub (in (code.wrap (code.if (code.noop)
Boolean/fromfloat (2) integer/fromfloat) (code.rand integer/rot)
exec.swap code.append integer/mult))))))
```

Append to the code stack

Generate some random code

Developmental GP

- A way of transitioning between representations
  - e.g. using a tree to represent a graph
  - Choose an evolvable representation, rather than one that is required by the problem domain

- Also a way of achieving scalability
  - Genotype space can be smaller than phenotype space
  - Human analogy: 30,000 genes encode 100 trillion cells

- And potentially for achieving complexity
  - Not limited to repetitive modular structures
Conventional
- Centralised
- Top-down
- Halting
- Static
- Exact
- Fragile
- Synchronous

Biological
- Distributed
- Bottom-up (emergent)
- Ongoing
- Dynamical
- Inexact
- Robust
- Asynchronous

See Mitchell, “Biological Computation,” 2010
http://www.santafe.edu/media/workingpapers/10-09-021.pdf

Cellular Automata (CA)

What is a cellular automaton?
- A model of “emergence”
  - complex behaviour emerges from interactions between simple rules
  - Emergent behaviour occurs widely in biological systems
- Originally developed by Ulam & von Neumann in the 1940s/50s
- Popularised by John Conway’s work on the ‘Game of Life’ in the 1970s
- Significant later work by Stephen Wolfram from the 1980s onwards
Definition

- Computation takes place on a grid, which may have 1, 2 or more dimensions, e.g. a 2D CA:

![Grid Diagram]

Definition

- At each grid location is a **cell**
  - Which has a **state**
  - In many cases this is binary:

![Cell Diagram]
Each cell contains an automaton

- Which observes a neighbourhood around the cell

If one neighbour is on, turn on, else turn off

If one neighbour is on, turn on, else turn off
Definition

- The CA is run over a number of discrete time steps
  - At each time step, each automaton applies its update rule
  - Time = 0

If one neighbour is on, turn on, else turn off

---

Definition

- The CA is run over a number of discrete time steps
  - At each time step, each automaton applies its update rule
  - Time = 1

If one neighbour is on, turn on, else turn off
The CA is run over a number of discrete time steps
- At each time step, each automaton applies its update rule
- Time = 2

If one neighbour is on, turn on, else turn off

The CA is run over a number of discrete time steps
- At each time step, each automaton applies its update rule
- Time = 3

If one neighbour is on, turn on, else turn off

The CA is run over a number of discrete time steps
- At each time step, each automaton applies its update rule
- Time = 4

If one neighbour is on, turn on, else turn off
A number of different neighbourhoods are used in CAs. This is called a Moore neighbourhood.

A number of different neighbourhoods are used in CAs. This is called a von Neumann neighbourhood.
A number of different neighbourhoods are used in CAs
- This is called an extended von Neumann neighbourhood

At the edges, toroidal neighbourhoods are often used
- Also known as periodic boundary conditions
**Cellular Automata (CA)**

- **What are cellular automata used for?**
  - Modelling spatial processes
    - e.g. forest fires, disease spread
  - Modelling physical processes
    - e.g. crystal formation, thermodynamics
  - Modelling biological processes
    - e.g. pattern formation, self-replication
  - Solving computational problems
    - e.g. random number generators, ciphers
  - Parallel processing architectures
    - e.g. systolic arrays, Connection Machine

**Conway’s Game of Life**

- **Developed by John Conway in the 1970s**
  - A simple model of self-replication
  - Surprisingly complex behaviour
  - Led to wider interest in CAs

- **2 states (live, dead), Moore neighbourhood, 4 rules:**
  - A live cell with <2 live neighbours dies (*under-population*)
  - A live cell with 2-3 live neighbours remains alive
  - A live cell with >3 live neighbours dies (*over-crowding*)
  - A dead cell with 3 live neighbours becomes a live cell (*reproduction*)

Spaceships, Guns
Methuselahs

- Patterns that grow and take a long time to stabilise
  - Complexity emerges from simple rule and initial state
  - Can be seen as carrying out a complex computation

- Acorn: size 7, grows to 1057, lasts 5206 time steps
  - Stable pattern consists of 41 blinkers, 4 traffic lights, 34 blocks, 30 beehives, 1 honey farm, 13 gliders, 8 boats, 5 loaves, 3 ships, 2 barges, 2 ponds and 1 mango

http://www.conwaylife.com/wiki/Methuselah

Game of Life

- A computationally interesting cellular automata
  - Simple definition, complex behaviour
  - Unexpected emergent phenomena
  - i.e. spaceships, methuselah
  - Computationally universal
Various multi-valued state CAs have been studied
  - e.g. Langton’s loops model self-replication
    - Uses 8 states:
      ![Langton's loops](image1.png)
    - e.g. WireWorld models electron flow in circuits
      - Uses 4 states:
        ![WireWorld](image2.png)
Elementary Cellular Automata

1D binary CAs that take place on a single grid row
- Appear simple, but can be deceptively complex
- Probably the most studied form of CA
- Stephen Wolfram’s work on these is very well known

Space-time diagram

They are based around a neighbourhood of size 3:

Hence, it maps $2^3 = 8$ possible patterns to 0 or 1
- Meaning there are $2^8 = 256$ possible update rules

Rule 30, Rule 110
How do I compute using a CA?

- 1) Find a suitable rule
- 2) Encode the problem instance in the initial state
- 3) Execute the CA for a certain number of steps
- 4) Read the result from the final state of the CA

Lots of cryptography applications

- Google ‘cellular automata encryption’
- Lots of different CA models used
- Have a look!
In a Boolean network, each variable is regulated by others that serve as inputs. The dynamic behavior of each variable—that is, whether it will be on or off at the next moment—is governed by a logical switching rule called a Boolean function. The function specifies the activity of a variable in response to all the possible combinations of activities in the input variables. One such rule is the Boolean OR function, which says that a variable will be active if any of its input variables is active. Alternatively, the AND function declares that a variable will become active only if all its inputs are currently active.

One can calculate how many Boolean functions could conceivably apply to any binary element in a network. If a binary element has \( K \) inputs, then there are \( 2^K \) possible combinations of inputs it could receive. For each combination, either an active or inactive result must be specified. Therefore, there are \( 2 \) to the \( 2^K \) power possible Boolean switching rules for that element.

The mathematically idealized versions of biological systems I shall discuss are called autonomous random Boolean NK networks. They consist of \( N \) elements linked by \( K \) inputs per element; they are autonomous because none of the inputs comes from outside the system. Inputs and one of the possible Boolean functions are assigned at random to each element. By assigning values to \( N \) and \( K \), one can define an ensemble of networks with the same local features. A random network is one sampled at random from this ensemble.

Each combination of binary element activities constitutes one network “state.” In each state, all the elements assess the values of their regulatory inputs at that moment. At the next clocked moment, the elements turn on or off in accordance with their individual functions. (Because all the elements act simultaneously, the system is also said to be synchronous.) A system passes from one unique state to another. The succession of states is called the trajectory of the network.

A critical feature of random Boolean networks is that they have a finite number of states. A system must therefore eventually reenter a state that it has previously encountered. Because its behavior is determined precisely, the system proceeds to the same successor state as it did before. It will consequently cycle repeatedly through the same states.

Such state cycles are called the dynamic attractors of the network: once a network’s trajectory carries it onto a state cycle, it stays there. The set of states that flow into a cycle or that lie on it constitutes the “basin of attraction” of the state cycle. Every network must have at least one state cycle; it may have more.

Left to itself, a network will eventually settle into one of its state cycle attractors and remain there. Yet if the network is perturbed in some way, its trajectory may change. Two types of perturbation are worth discussing here: minimal perturbations and structural perturbations.

A minimal perturbation is a transient flipping of a binary element to its opposite state of activity. If such a change does not move a network outside its original basin of attraction, the network will eventually return to its original state cycle. But if the change pushes the network into a different basin of attraction, the trajectory of the network will change: it will flow into a new state cycle and a new recurrent pattern of network behavior.
A gene is a contiguous region of DNA

Each gene describes how to make a protein, which is a molecular machine
Genes are expressed when a **transcription complex** forms – this is a bit like a photocopier.

The transcription complex binds to conserved patterns that mark the start of genes, e.g. TATA.

Instead, it must be helped out by **proteins** called **transcription factors** (TFs).

These bind to both the DNA and the transcription complex, holding everything in place – a bit like a clamp.
Why is gene regulation of computational interest?
   - Gene regulatory networks (GRNs) underlie the complexity of biological systems, such as ourselves

Biological systems are **structurally complex**
   - This has led to an interest in how computational models of GRNs can be used to generate intricate structures

Biological systems are **dynamically complex**
   - GRNs produce robust and intelligent responses, leading to interest in whether computational models can do the same, e.g. for controlling robots

Also theoretical interest in **computability**
   - How do biological systems process information?
   - How does this differ from conventional computers?
   - Is it in some ways better? e.g. more compact and robust

And from an EA perspective **evolvability**
   - GRNs are known to be evolvable, i.e. able to respond robustly to mutation and crossover
   - Especially in comparison to computer programs
   - Potentially an evolvable representation for GP
This idea can be modelled as a Boolean Network

- A set of nodes (representing genes)
- Each with a binary state (expressed or not expressed),
- A set of input nodes (their regulating genes),
- And a Boolean function (their regulatory function)
- These are executed synchronously at each time step

\[ G_0 = \text{NOT}(G_2 \text{ AND } G_4) \]

Looks a bit like an elementary cellular automata

- But without a fixed neighbourhood
- And with a different update rule in each cell
- In fact, a Boolean network is a generalisation of a CA
- Therefore must be capable of universal computation
Random Boolean Networks

- Their behaviour can be studied statistically
  - By sampling and executing networks with particular sizes, connectivities and function sets
  - These are termed Random Boolean Networks (RBNs)

- Stuart Kauffman is known for this
  - He studied \textit{NK} networks:
    - RBNs with $N$ nodes
    - $K$ inputs per node
    - A random function for each node
    - Also called Kauffman networks

Kauffman’s NK Networks

- The behaviour of an RBN falls into 3 categories:

  \begin{center}
  \begin{tabular}{ccc}
  Ordered & Critical & Chaotic \\
  \begin{figure}
  \end{figure}
  \end{tabular}
  \end{center}

From [Gershenson, 2004] Introduction to Random Boolean Networks
http://uk.arxiv.org/abs/nlin.AO/0408006
Kauffman’s NK Networks

- Kauffman observed that, on average*

  Ordered when $K<2$  Critical when $K=2$  Chaotic when $K>2$

*But note this doesn’t mean that all $K=2$ networks are critical, or that critical networks can’t be found for $K>2$

Attractors

- Attractors are an important concept for RBNs
  - A finite number of nodes means that states must repeat
  - An attractor of length $L$ repeats every $L$ steps
  - e.g. $L=3$:

  ![Diagram of attractor cycle]

  - An attractor of length 1 is termed a point attractor

- Transients occur before an attractor is reached
  - e.g. “Acorn” is a transient leading to a stable attractor
Attractors

◊ Attractors have a biological interpretation
  ▶ Stable states for a cell
    e.g. cell types: neuron, liver, skin, …
  ▶ Cancer can also be seen as an attractor

◊ Boolean networks are used to study bio GRNs
  ▶ Captures their qualitative dynamics
  ▶ Usually using NOT, OR and AND functions
  ▶ See “Boolean modeling of biological regulatory networks: A methodology tutorial”:

Limitations of Boolean Networks

◊ From a practical perspective
  ▶ Inputs and outputs must be binary encoded
  ▶ Difficult to handle large/continuous/many values
  ▶ E.g. Pi in binary: 010000000100100100011111111101

◊ From a biological perspective
  ▶ Gene expression levels are not discrete
  ▶ Regulatory functions are not always Boolean functions

  However, Boolean networks are computationally efficient and can be implemented directly in hardware
GRNs can be used to generate structures
- Makes use of their ability to generate patterns
- Often hybridised with a model of cell division

"Maternal cell" containing a GRN

Generative code books for compression [Trefzer'10]
- GRN generates a series of 8x8 patterns
- Their indices are then used to encode an image

58, 843, 456, 25, 2, 956, 32, 56, 445, 322, 12, 100, 238, 685, ...
Models of gene regulatory networks
- Boolean networks and artificial genomes

Main applications
- Control, especially in robotics
- Generating complex structures
- Understanding biological systems