Overview

- Introduction to the essentials of evolutionary search
  - Program representations
  - Fitness functions, selection mechanisms
  - Recombination via crossover and mutation
  - Experimental considerations
- Applications to graphic design and visual arts
  - Human-centric evolution, GUI considerations
  - Specifying design tasks for an evolutionary approach (tutorial)
- Art/design and other case studies throughout
Evolutionary Approaches

to AI tasks Involving Search

- Rather than writing software directly, we evolve codings in some representation which can be interpreted as programs

- Sometimes, the point of the exercise is to find a single coding which solves a problem
  e.g., *optimising parameters for an aeroplane wing design*

- Sometimes, the point of the exercise is to generate the best programs (in terms of efficiency, etc.)
  e.g., *evolving a sort algorithm*

- Sometimes the point of the exercise is to actually run the evolved programs to see their outputs
  e.g., *most design tasks*

- Worth remembering that it is just another search technique

Evolutionary Approaches

Overview of the Method and Terminology

1. Generate a *population* of *genotypes* (codings) randomly, which can each be interpreted, compiled and run as a program and assessed as a *phenotype* (the program itself and/or the output from the program)

2. Evaluate each of the *individuals* in the population
   - A person could do the *evaluation* here

3. Produce a new *generation* by allowing the *crossover* of coding material from pairs of parents into *offspring*, and performing *mutations* on the children
   - Allow the *fittest* individuals to produce offspring more often than the less fit ones

4. Check for *termination conditions*, i.e., whether the search has *converged* on a good solution, or time is up, etc
   - If we aren’t finished, start again at 2 with the new population
The Canonical Genetic Algorithm

Einstein Effect

Initial Population

Selection

Mating

Recombination

Notify user

Choose best ever

yes

no

Termination check

Three Major Questions

1. How do we represent codings?
   - Need codings to (a) be interpreted as programs
   - (b) be randomly generatable and
   - (c) be manipulatable by genetic operators

2. How do we select pairs of individuals to be parents?
   - Need to specify (a) fitness function and
   - (b) selection mechanisms

3. How do we produce offspring from parents?
   - Need to specify (a) crossover operations and
   - (b) mutation operations on the codings
Types of Coding Representations

- Actual artefacts we want to see
- Codings specific to the task at hand
- Parameters for a fixed program
- Tree based representations of programs
- Actual program code (in C, Java, Prolog, etc.)

- Generally: evolving parameters is known as *genetic algorithms*, whereas evolving programs is known as *genetic programming*.

Example

Evolving Artefacts Directly

- Evolve placements/shapes/sizes/rotations of images according to correlations

http://www.doc.ic.ac.uk/~sgc/papers/colton_evomusart08.pdf
Example

Task Specific Representation

- Evolve image filter trees to approximate Photoshop filters


Evolving Parameters For Static Programs

- Here, we already know the program we require, but we don’t know certain parameters for it
- Parameters are usually numbers, but could be strings, booleans, other data types, etc.
- Codings to be evolved are usually lists of parameters
  - We can randomly generate the lists; crossover material from lists; and mutate the elements of a list
- Advantages: convergence can be quick because the range of the variables may be quite limited
- Disadvantages: have to specify the majority of the program structure in advance; solutions are not likely to surprise you
Static Programs

Simple Example

- Evolve parameters A, B and C, so that the simple method produces the following input/output pairs:
- \((17, 18)\) \((6, 36)\) and \((7, 7)\)

```c
simple(int x){
    if (x > A) return x + 1;
    else if (x < B) return x*x;
    else if (x * C < 10) return x;
    return 0;
}
```

Static Programs

Design Example

See later lecture on shape grammars

Video courtesy of Kazjon Grace

/* $R = [-360, 360] */
/* $HUE = [0, 255] */
/* $SAT = [-1, 1] */
/* $RTWO = [-360, 360] */
/* $RTHREE = [-360, 360] */
/* $HUECHANGE = [-0.1, 0.5] */
/* $STARX = (0.25, 0.75) */
/* $SIZX = [0.99, 0.999] */
/* $REPS = [1, 8] */
Binary Representations

- Represent all the parameters as binary numbers with the same number of digits and concatenate them as a long bit string.
  - E.g. \([16, 7, 1]\) becomes \([10000, 00111, 00001]\)
    - which in turn becomes: \(10000011100001\)
- Still affords crossover and mutation.
- Advantages: can use a binary representation for any parameter generation problem, so we can use the same evolutionary system.
- Disadvantages: crossover might be deleterious; bit strings might need to be very long; not particularly intuitive.
- Early papers on genetic algorithms tend to use this approach.

Tree Based Program Representations

- Representing programs as trees is a common technique, which is good for genetic programming, because:
  - It’s fairly easy to translate trees into program code.
  - It’s fairly easy to generate trees randomly.
  - It’s possible to swap branches for crossover.
  - It’s possible to mutate trees.
- Advantages: more expressive search space; possibility for surprise; less need to worry about the program structure.
- Disadvantages: search space might be too large, so search might not converge quickly; might need a lot of experimentation with ingredients.
Example

Program Tree Fragment

```
IF_LT_E

+  Y  abs  √
   √  Y  +
 -  X  Y  X  Y
```

if ($(\sqrt{(x-y)} + y) < y$ then return $|x*y|$ else return $\sqrt{(x+y)}$

Example

Alternate Formulation

```
IF_E

<  Y  abs  √
   +  Y  +
   √  Y  *
 -  X  Y  X  Y
```

if ($(\sqrt{(x-y)} + y) < y$ then return $|x*y|$ else return $\sqrt{(x+y)}$
Another Example

Program Tree Fragment

if \((\sin(y \times x) + y) < \sqrt{y}\) then return \(\cos(x \times y)\) else return \(x\)

Design Example

```java
int width = 647;
int height = 400;

public void setup() {
  size(width, height);
  background(hsv(0.0f, 0.0f, 0.0f));
  for (float y=0; y<1; y+=1/(float)height) {
    for (float x=0; x<1; x+=1/(float)width) {
      float h = y;
      float s = x;
      float v = (x)+(y)+(x)*(y);
      pixel(x, y, hsv(h, s, v));
    }
  }
}
```
Genetic Programming Ingredients

- Wrapper code - to turn evolved fragments into compilable code
  - In our design example: two nested for loops specifying x and y co-ordinates; method definition code

- Terminal set = set of constants and variables you want to see in the programs (trees), usually integers, floats, etc., and variables occurring in the wrapper code. These are found as the leaf nodes of trees
  - In our design example: floats 0.1 to 0.9, integers 1 to 10, co-ord variables x and y

- Function set = set of function symbols you want to see in the programs (trees), usually arithmetic, geometric functions, general mathematical functions, and functions specific to the task at hand, e.g., neighbouring pixel information. These are not normally leaf nodes (unless they have input arity zero, e.g., a function generating a random number)
  - In our design example: + * / - √ sin cos tan

- Need to choose these carefully as an experimental consideration (see later)

Random Generation of Individuals

- For parameter lists and bitstrings, it is straightforward, subject to being given ranges of values for each parameter

- For trees, one way to generate random ones is to grow them by iteratively replacing a terminal node with a function node and randomly chosen terminals as input

- Subject to typing constraints and often a limit on the tree size and/or depth
Randomly Growing Trees

Example

Start with a Y

Replace Y with X-Y

Replace X with $\sqrt{X}$

And so on...

Survival of the Fittest

- We now know some ways to represent individuals in a population
- Recall that we want to produce successive generations of individuals, each one derived from the current generation
  - Usually keep a fixed population size, but might allow it a small amount of fluctuation (have to avoid populations ballooning)
- We want to simulate Darwinian natural selection of the fittest individuals, to drive a best-first search
  - Need to define a fitness function for each project
  - And choose a mechanism for selecting pairs of individuals to produce an offspring for the next generation
Fitness Functions

- A fitness function needs to differentiate individuals with respect to their ability at the task at hand. Ideally, it needs to really match your expectations and properly separate the good from the bad individuals.

- Choosing a fitness function is usually task specific, and is perhaps the most important experimental consideration. The actual values don’t matter as much as the relative comparison of individuals they enable.

- Three generic aspects that can be included in fitness function calculations:
  - Aspects of the coding, e.g., graph size and structure, etc.
  - Aspects of the derived program e.g., runtime efficiency, programmatic structure, amount of recursion, complexity, etc.
  - Aspects of the output from the program, e.g., predictive accuracy for machine learning, correctness of outputs for given inputs, aesthetic and utilitarian considerations, etc.

Example Fitness Functions

- UG project of Ricardo Hoare and Joanne Penner on optimising traffic flow

\[
\text{wait} = \sum_{i \in \text{Cars}} \frac{w_i}{d_i + w_i}
\]

Figure 3b: Fitness function for sequence evaluation where \(w_i\) denotes total waiting time and \(d_i\) denotes total driving time for car \(i\).

Figure 4b: Improvement in fitness over 130 generations.
Design Example #1

**Fitness Function**

- Evolving Manhattan-style skylines
- Weighted sum of correlations between rectangle properties
  - Distance from centre negatively correlated with height, width, saturation and positively correlated with y co-ordinate
  - Depth positively correlated with height and brightness, negatively correlated with saturation

http://www.doc.ic.ac.uk/~sgc/papers/colton_evomusart08.pdf

Design Example #2

**Fitness Function**

- Evolving pixel shaders for a game city world (Subversion) to match a user’s colour specification given via GUI
  - Calculation performed on the colour qualities of the generated image (matching to requirements)
Selection Mechanisms

- Suppose we have a way to assess individuals relative to each other (fitness function). We need to use this to choose individuals to derive new ones from
  - Standard approach is to kill off all the old generation and replace with new
  - Desired properties (simulating natural selection):
    - Fitter individuals are more likely to produce offspring, but even quite unfit individuals have a possibility (which helps to keep the gene pool healthy)
  - Possible selection mechanisms:
    - Elitism, truncation, fitness proportionate and tournament selection, + others
    - Can also use an intermediate population
    - The selection mechanism is also an important experimental consideration

Elitism and Truncation

- Only certain individuals which are above a certain fitness threshold are used to produce new offspring
- Elitism: an exact copy of the individuals go unchanged into the next generation
  - May require an age constraint, whereby individuals who have been around for too many generations are deleted
- Truncation: top individuals are allowed to generate new individuals via recombination of their genes (see later)
- Possible disadvantage: premature convergence due to a limited gene pool
Fitness Proportionate Selection
Calculation

- Each individual is selected with a probability proportionate to its fitness
- Calculate probability of an individual being selected by dividing its fitness by the sum of all fitnesses, i.e., by calculating:

\[ p(i) = \frac{f(i)}{\sum_{j=1}^{N} f(j)} \]

where \( p(i) \) is the selection probability of individual \( i \), and \( N \) is the size of the population
- Use this to define a probability distribution with which to choose individuals to produce offspring from. Note that unfit individuals still have a (small probability) of selection.

Fitness Proportionate Selection
Implementation

- Line up the individuals sequentially along the number line from 0 to 1, with the length of each being its probability \( p(i) \)
- Generate a random number, then find and select the individual occupying that position on the line

```
0 ← A B C D E F → 1
   ↑ R3 = 0.1
   ↑ R1 = 0.3
   ↑ R2 = 0.7
```
Fitness Proportionate Selection
Roulette Wheel Analogy

R1 = 0.3
R2 = 0.7
R3 = 0.1

Tournament Selection

• Choose a tournament size, T
• Repeatedly select T individuals for a tournament
• The winner of the tournament is the fittest one in the set, and this one is selected

Tournament 1

Selected

p(A) = 0.4
p(B) = 0.2
p(C) = 0.09
p(D) = 0.1
p(E) = 0.11
p(F) = 0.05

Tournament 2

p(A) = 0.4
p(B) = 0.2
p(C) = 0.09
p(D) = 0.1
p(E) = 0.11
p(F) = 0.05

Tournament 3

p(A) = 0.4
p(B) = 0.2
p(C) = 0.09
p(D) = 0.1
p(E) = 0.11
p(F) = 0.05
Using an Intermediate Population

- In natural evolution, fit individuals get a chance of finding a partner, but no guarantee of producing offspring
- Populate an intermediate population (IP) in a fitness proportionate way (see next slide)
- The IP may have multiple copies of an individual
- We then choose pairs from the set randomly for breeding and allow these to produce offspring

![Diagram showing population dynamics]

Populating the Intermediate Population

- For each individual, we calculate a guaranteed number of copies of it which can go into the intermediate population and a probability of another copy going in
- Calculation:
  \[ E(i) = \frac{f(i)}{\text{average}} = \frac{f(i)}{(\sum_{j=1}^{N} f(j))/N} \]
- Then, for each i, the integer part of E(i) is the guaranteed number and the fraction part is the probability of another copy
Example Calculation of an Intermediate Population

- Suppose we have 6 individuals: A, B, C, D, E, F such that
  - \( f(A) = 40, f(B) = 20, f(C) = f(D) = 10, f(E) = 5, f(F) = 1 \)
  - Average fitness = \( \frac{40+20+10+10+5+1}{6} = \frac{86}{6} = 13.3 \)
  - \( E(A) = \frac{40}{13.3} = 3.01 = 3 \text{ guaranteed, } P = 0.01 \)
  - \( E(B) = \frac{20}{13.3} = 1.50 = 1 \text{ guaranteed, } P = 0.5 \)
  - \( E(C) = E(D) = \frac{10}{13.3} = 0.75 = 0 \text{ guaranteed, } P = 0.75 \)
  - \( E(E) = \frac{5}{13.3} = 0.38 = 0 \text{ guaranteed, } P = 0.38 \)
  - \( E(F) = \frac{1}{13.3} = 0.08 = 0 \text{ guaranteed, } P = 0.08 \)

Producing New Generations

- We know ways to represent individuals and we know ways to select them for breeding
- We need to specify how genetic operators can produce offspring from two parents
- Two main mechanisms:
  - Crossover
  - Mutation
Evolving Parameters

One Point Crossover

- Suppose our coding is of \( n \) numerical parameters
- One-point crossover of two parents:
  - Parent1 = \([X_1, X_2, ..., X_n]\) and Parent2 = \([Y_1, Y_2, ..., Y_n]\)
  - Choose the same position \( P \) in the list coding of both parents and swap the left and right sides to create these two children:
    - Child1 = \([X_1, ..., X_P, Y_{P+1}, ..., Y_n]\)
    - Child2 = \([Y_1, ..., Y_P, X_{P+1}, ..., X_n]\)

### Evolving Parameters - Example

**One Point Crossover**

\[
\begin{align*}
\text{Parent1} & = [1, 17, 8, 9, 10, 5, 6] \\
\text{Parent2} & = [7, 9, 10, 17, 1, 3, 8]
\end{align*}
\]

\[
\begin{align*}
\text{Child1} & = [1, 17, 8, 9, \text{P=4}, 10, 5, 6] \\
\text{Child2} & = [7, 9, 10, 17, 1, 3, \text{P=4}, 8]
\end{align*}
\]
Evolving Parameters

Two Point Crossover

- Suppose our coding is of n numerical parameters
- Two-point crossover of two parents:
  - Parent1 = [X₁, X₂, ..., Xₙ] and Parent2 = [Y₁, Y₂, ..., Yₙ]
  - Choose the same positions P and Q (such that Q > P), in the list coding of both parents and swap the two central sections to create these two children:
    - Child1 = [X₁, ..., Xₚ, Yₚ₊₁, ..., Yₚ, Xₚ₊₁, ..., Xₙ]
    - Child2 = [Y₁, ..., Yₚ, Xₚ₊₁, ..., Xₚ, Yₚ₊₁, ..., Yₙ]

Evolving Parameters - Example

Two Point Crossover

P=2  Q=4  
[1,17, 8,9, 10,5,6]  [7,9, 10,17, 1,3,8]  
[1,17,10,17,10,5,6]  [7,9,8,9,1,3,8]
Example

Crossovers for Bitstrings

\(a, b, X, Y \in \{0, 1\}\)

One point crossover

Two point crossover

Crossover for Program Trees

- Essentially, just swap branches in a tree
- Choose a node in the first parent tree and a node in the second tree and swap the branches including the nodes
- Usually subject to some typing constraints, i.e., function nodes expecting input of a certain type must continue to get that type of input
- Often, we might also impose a tree size limit, or a tree depth limit, to stop programs ballooning (which might take a relatively long time to execute)
Crossover for Program Trees
Example - Individuals

Individual 1

Crossover for Program Trees
Example - Individuals

Individual 2
Crossover for Program Trees
Example - Chosen Nodes

Crossover for Program Trees
Example - Child 1
Crossover for Program Trees
Example - Child 2

Crossover for Program Trees
Example - Code for Child 1

if \((x + y) < y\) then return \(|x \cdot y|\) else return \(\sqrt{x + y}\)
Crossover for Program Trees

Example - Code for Child 2

```
if ((\sin(y\times x) + y) < \sqrt{y}) then return \cos(\sqrt{(x-y) + y}) else return x
```

Crossover

Design Example

Parents

Children
Crossover - Final Notes

- Both 1 and 2 point crossover, and tree crossover preserve fairly large areas of the *genome* (coding)
- This is motivated by natural evolution, where we inherit genes from our parents, i.e., we are not an average of our parents
- Program *bloat* has to be managed, as the programs can get very difficult to understand, and sometimes slow to execute (although sometimes understandability isn’t all that important)
- See ftp://cs.ucl.ac.uk/genetic/papers/WBL_fairxo.pdf for experiments with different crossover techniques aimed at reducing bloat

Mutation

Overview

- Again, simulating natural evolution: when crossover occurs, random mutations happen
- This is usually deleterious (deadly), but it can be an evolutionary driving force
- In GA and GP approaches, we used mutation to help the search get out of local maxima
- We specify a mutation rate $M$ (as a probability) in the evolutionary setup, which determines the amount of mutation which is required
Mutation in Genetic Algorithm Approaches

- For each child which is just about to be put into the next generation, loop through each parameter in its coding list (or bit in the bitstring)
  - Then, with probability $M$, change the parameter value for another value in the range of the parameter
  - (Or flip the bitstring, i.e., from 1 to 0 or 0 to 1)

Mutation in Genetic Programming Approaches

- For each child which is just about to be put into the next generation, loop through each of its tree nodes, and with probability $M$, mutate the node
- Mutation can be done in one of three ways, with roughly increasing mutation occurring:
  - Change a terminal for another terminal
  - Change a function for another function
  - Replace one subtree with a randomly generated one
Replace terminal X with 10
if (10 + y) < y then return |x*y| else return √(x+y)

Replace function abs with sin
if (x + y) < y then return sin(x*y) else return √(x+y)

Replace subtree √X+Y with sin(abs(X+Y))
Experimental Considerations

- AI is an empirical science - we have to experiment to find the best ways to do things. In an evolutionary setting, we want to determine a setup which:
  - Maximises the overall fitness of the best individual(s) produced
  - Minimises the time taken to converge on such good individuals
  - Sessions can take a long time, so you will need to think hard about what factors to experiment with
  - Initial experimentation is usually undertaken with the overall interpretation of the task, i.e., trying different coding representations and different fitness functions. Does the output match your expectations? Is the fitness function a good approximation of what you want?
  - Once you are fairly happy with this, you can move on to experimenting with the evolutionary search parameters
Search Parameters

- Main search parameters to experiment with:
  - Population size
  - Crossover style
    (1 or 2 point, applied once, or repeatedly - dependent on the coding/tree size)
  - Mutation rate and mutation style
  - Selection mechanism
- Also need to experiment with the termination conditions
  - If only one solution is required, then specify a minimum fitness of the best individual in the current generation, when this is achieved, stop
  - If multiple solutions are required, then specify a minimum average fitness of the latest population, or minimum average fitness of the best 10% ever seen, etc.
  - Number of generations to run
  - Maximum time allowed

Some Rules of Thumb

- 100 generations with 100 in the population is a good starting point - you should be able to see the average fitness increasing
- If the search is converging prematurely, and you think the fitness could be increased, try altering the population size and/or the mutation rate, to increase the genetic diversity. Note that the search will probably take much longer to converge
- Systematic experimentation always pays off
  - Possibly distribute your sessions
Measurements to Take

Behaviour Tree Example

- Fitness versus generation number (Max, min, mean, best ever, average of the top 10% ever seen)
- Expect to see an increase, followed by a plateau at convergence

Watch out for local plateaus: run the search for longer if possible

Scene Generation Example

- Max fitness (and time taken) versus search setup
User-Centric Evolution

- The user can guide the search by specifying an objective (mathematical) fitness function
  - Perhaps via a GUI in advance of search
- They can also act directly as the fitness function during the session
  - After each generation is produced, they can make boolean choices about which individuals are good and which are bad
  - Or they could make value judgements and give numerical values to each individual
  - This is usually based on the output of the programs rather than the code

Example

User-Centric Approach

- The Elvira Ludic Evolutionary Art iPad App

Case Study #1

Evolving Building Designs

- Representation: bespoke scripts which define shape protrusions and contractions for towers in a game world (Subversion)
  - Can be represented in tree form
- User-centric interface with 3D environment, mouse controls the view of all the phenotypes (buildings). Large view enabled on mouse click
- Strong and weak forms of both crossover and mutation (experimental considerations)
  - Sliders for the strength of each operator

Case Study #2

The Avera System

- Generic user-guided evolutionary system
- Applied to a number of art projects
- Pixel-based, particle-based, spirographs, image filters
- GUI keeps previous generations; allows favourites and mixing of individuals from different generations; allows the generation of different phenotypes from the same genotype (e.g., different sized pictures); and lets viewers see the tree and the code produced
- Enables the I/O storage of individuals and entire sessions

http://www.doc.ic.ac.uk/~sgc/papers/hull_cc07.pdf
Case Study #3
Interactive Shape Grammar Evolver

• Uses CFDG software from www.contextfreeart.org
• Students built their own design grammar templates, which have numerical parameters in them
• Parameters are randomly assigned to start with, and then the user expresses choices through the UI
• Version 1 of the interface was very basic - choose grammars by looking at the designs, press space to evolve them. Can change design grammar template
• Version 2 is more useful

GUI Design Issues

• Need to experiment with the nature and application of the genetic operators
• To make sure that each chosen individual has an equal chance of getting developed
• To make sure that the experience is satisfying
• Two ways in which it won’t be satisfying are: (i) cannot return to previous generations, or mix favourites from different generations (ii) it feels pointless making any choices of individuals - either because the children look nothing like their parents or the children look too much like their parents. Need to experiment with the strength of crossover and mutate, and in particular enable users to focus on ones they like
• Also think about throwing in randomly generated individuals from time to time, to give people ideas when they have focused too much. Might also need to manage the lack of interesting phenotypes at the start of the session (e.g., give from a library)
• Need to experiment with users to assess their level of enjoyment and achievement
• See later in the course for more generic issues

Check Out These Projects

NEvAr by Penousal Machado
Check Out These Projects

The Work of Jon McCormack

Check Out These Projects

The Mutators Research Group
http://doc.gold.ac.uk/mutators/
Check Out These Projects

The Work of Karl Sims
http://www.karlsims.com/

More Links

- http://www.jhlabs.com/java/art.html
- http://picbreeder.org/
- http://www.wickedbean.co.uk/cfdg/index.html