Introduction to Artificial Life and Genetic Algorithms

CS405

What is Artificial Life?

- Sub-Discipline of Complex Adaptive Systems
- Roots from Artificial Intelligence
  - Bottom-Up rather than Top-Down
- Studies application of computational techniques to biological phenomena
- Studies application of biological techniques to computational problems
- Question: Can we build computers that are intelligent and alive?

Some Requistes for Life

- Autonomy
- Metabolism
- Survival Instinct
- Self-Reproduction
- Evolution
- Adaptation

Let’s focus on Self-Reproduction, Evolution, and Adaptation

Self-Reproduction in Computers

- An old mathematical problem, to write a program that can reproduce (e.g., print out a copy of itself) leads to infinite regress.
- Attempt in a hypothetical programming language:

```
Program copy
    print(“Program copy”)
    print(“      print(“Program copy”)”)
    print(“      print(“    print(“Program copy”)”)”)
```

Solution to Infinite Regress

- Solution in the von Neumann computer architecture. He also described a “self-reproducing automaton”
- Basic idea
  - Computer program stored in computer memory
  - A program has access to the memory where it is stored
  - Let’s say we have an instruction MEM that is the location in memory of the instruction currently being executed

A Working Self-Reproducing Program

```
Program copy
      L = MEM +1
      print(“Program copy”)
      print(“L = MEM + 1”)
      LOOP until line[L] = “end”
            print(line[L])
            L=L+1
      print(“end”)
end
```

Self-Reproducing Program

- Information used two ways
  - As instructions to execute
  - As data for the instructions
- Could make an analogy with DNA
  - DNA strings of nucleotides
  - DNA encodes the enzymes that effect copying: splitting the double helix, copying each strand with RNA, etc.

Evolution in Computers

- Genetic Algorithms – most widely known work by John Holland
- Based on Darwinian Evolution
  - In a competitive environment, strongest, "most fit" of a species survive, weak die
  - Survivors pass their good genes on to offspring
  - Occasional mutation

Evolution in Computers

- Same idea in computers
  - Population of computer program / solution treated like the critters above, typically encoded as a bit string
  - Survival Instinct – have computer programs compete with one another in some environment, evolve with mutation and sexual recombination

GA's for Computer Problems

Population of critters → Population of computer solutions
Surviving in environment → Solving computer problem
Fitness measure in nature → Fitness measure solving computer problem
Fit individuals life, poor die → Play God and kill computer solutions that do poorly, keep those that do well. i.e. "breed" the best solutions typically Fitness Proportionate Reduction
Pass genes along via mating → Pass genes along through computer mating
Repeat process, getting more and more fit individuals in each generation.
Usually represent computer solutions as bit strings.

The Simple Genetic Algorithm

1. Generate an initial random population of M individuals (i.e. programs)
2. Repeat for N generations
   1. Calculate a numeric fitness for each individual
   2. Repeat until there are M individuals in the new population
      1. Choose two parents from the current population probabilistically based on fitness (i.e. those with a higher fitness are more likely to be selected)
      2. Cross them over at random points, i.e. generate children based on parents (note external copy routine)
      3. Mutate with some small probability
      4. Put offspring into the new population

Crossover

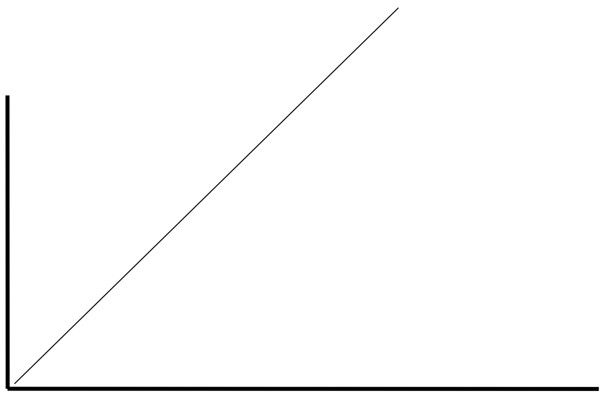
Typically use bit strings, but could use other structures

Bit Strings: Genotype representing some phenotype

Individual 1:	00101**0001**	Individual 2:	**10011**0110
New child :	**100110001**	has characteristics of both parents, hopefully better than before	

Bit string can represent whatever we want for our particular problem; solution to a complex equation, logic problem, classification of some data, aesthetic art, music, etc.

Simple example: Find MAX of a function



To keep it simple, use y=x so bigger X is better

Chromosome Representation

Let's make our individuals just be numbers along the X axis, represented as bit strings, and initialize them randomly:

Individual 1	:	000000000
Individual 2	:	001010001
Individual 3	:	100111111
….		
Individual N	:	110101101

Fitness function: Y value of each solution. This is the fitness function. Note that even for NP complete problems, we can often compute a fitness (remember that solutions for NP Complete problems can be verified in Polynomial time).
Say for some parents we pick: 100111111 and 110101101

Crossover

Crossover: Randomly select crossover point, and swap code
100111111 and 110101101

Individual 1:	10011**1111**	Individual 2:	**11010**1101
New child :	110101111	has characteristics of both parents, hopefully better than before	

Or could have done:

Individual 1:	**10011**1111	Individual 2:	11010**1101**
New child:	100111101	; not better in this case	

Mutation

Mutation: Just randomly flip some bits ; low probability of doing this

Individual:	**0**11100101
New:	**1**11100101

Mutation keeps the gene pool active and helps prevent stagnation.

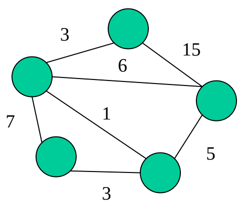
Demos

- Online: Minimum of a function
  - http://www.obitko.com/tutorials/genetic-algorithms

Second Example : TSP

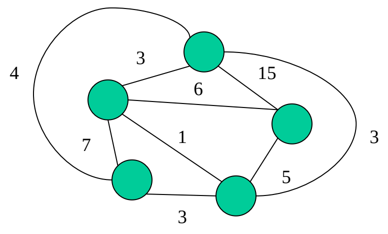
- NP-Complete
- NP-Complete problems are good candidates for applying GA's
  - Problem space too large to solve exhaustively
  - Multiple "agents" (each individual in the population) provides a good way to probe the landscape of the problem space
  - Generally not guaranteed to solve the problem optimally

- Formal definition for the TSP
  - Start with a graph G, composed of edges E and vertices V, e.g. the following has 5 nodes, 7 edges, and costs associated with each edge:



  - Find a loop (tour) that visits each node exactly once and whose total cost (sum of the edges) is the minimum possible

- Easy on the graph shown on the previous slide; becomes harder as the number of nodes and edges increases



- Adding two new edges results in five new paths to examine
- For a fully connected graph with n nodes, n! loops possible
  - Impractical to search them all for more than about 25 nodes
- Excluding degenerate graphs, an exponential number of loops possible in terms of the number of nodes/edges

- Guaranteed optimal solution to TSP
  - Evaluate all loops
- Approximation Algorithms
  - May achieve optimal solution but not guaranteed
  - Nearest Neighbor
  - Find minimum cost of edges to connect each node then turn into a loop
  - Heuristic approaches, simulated annealing
  - Genetic Algorithm

- A genetic algorithm approach
  - Randomly generate a population of agents
    - Each agent represents an entire solution, i.e. a random ordering of each node representing a loop
      - Given nodes 1-6, we might generate 423651 to represent the loop of visiting 4 first, then 2, then 3, then 6, then 5, then 1, then back to 4
      - In a fully connected graph we can select any ordering, but in a partially connected graph we must ensure only valid loops are generated
  - Assign each agent a fitness value
    - Fitness is just the sum of the edges in the loop; lower is more fit
  - Evolve a new, hopefully better, generation of the same number of agents
    - Select two parents randomly, but higher probability of selection if better fitness
    - New generation formed by crossover and mutation

- Crossover
  - Must combine parents in a way that preserves valid loops
  - Typical cross method, but invalid for this problem

    Parent 1 = 423651 Parent 2 = 156234

    Child 1 = 423234 Child 2 = 156651
  - Use a form of order-preserving crossover:

    Parent 1 = 423651 Parent 2 = 156234

    Child 1 = 123654

    - Copy positions over directly from one parent, fill in from left to right from other parent if not already in the child
- Mutation
  - Randomly swap nodes (may or may not be neighbors)

- Traveling Salesman Applet:

  Generates solutions using a genetic algorithm

  http://www.generation5.org/jdk/demos/tspApplet.html

Fun example: Smart Rockets

http://www.blprnt.com/smartrockets/

Eaters:

http://math.hws.edu/xJava/GA/

Why does this work?

- How does a GA differ from random search?
  - Pick best individuals and save their "good" properties, not random ones
- What information is contained in the strings and their fitness values, that allows us to direct the search towards improved solutions?
  - Similarities among the strings with high fitness value suggest a relationship between those similarities and good solutions.
    - A schema is a similarity template describing a subset of strings with similarities at certain string positions.
    - Crossover leaves a schema unaffected if it doesn't cut the schema.
    - Mutation leaves a schema unaffected with high probability (since mutation has a low probability).
    - Highly-fit, short schema (called building blocks) are propagated from generation to generation with high probability.
  - Competing schemata are replicated exponentially according to their fitness value.
  - Good schemata rapidly dominate bad ones.

Advantages of GA's

Easy to implement
Easy to adapt to many problems
Work surprisingly well
Many variations are possible
(elitism, niche populations, hybrid w/other techniques)
Less likely to get stuck in a local minima due to randomness

Dangers of GA's

Need diverse genetic pool, or we can get inbreeding :
stagnant population base

No guarantee that children will be better than parents
could be worse, could lose a super individual

elitism- when we save the best individual

Other Problems Addressed By GA's

1. Classification Systems - Plant disease, health, credit risk, etc.
2. Scheduling Systems
3. Learning the boolean multiplexer.

S0	S1	D0	D1	D2	D3
0	0	X			
0	1		X		
1	0			X	
1	1				X

Boolean Multiplexer

System was able to learn the 11 function boolean multiplexer from training data.

Hard problem for humans! Ex:

```
00101010101  1
11010101001  0
…
```

Genetic Programming

Invented by John Koza of Stanford University in the late 80's.

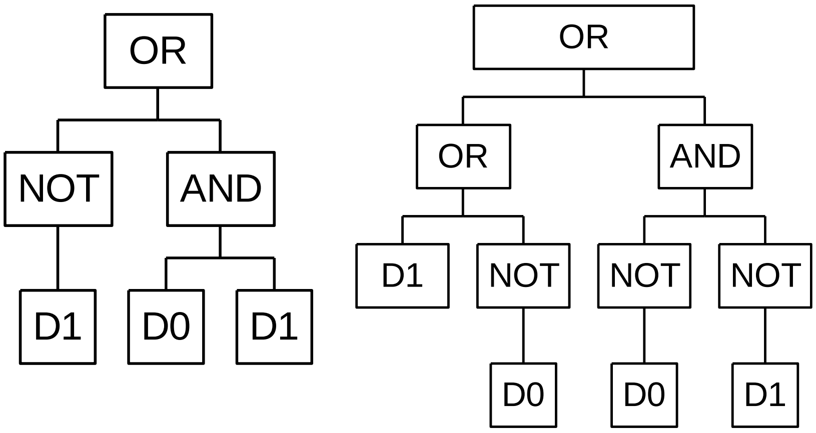
What if instead of evolving bit strings representing solutions, instead we directly evolve computer programs to solve our task?

Programs are typically represented in LISP. Here are some example Lisp expressions:

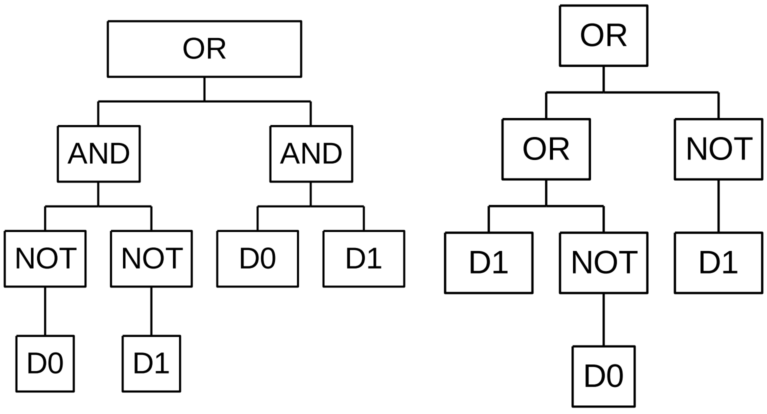
```
(Or (Not D1) (And D0 D1))

(Or (Or D1 (Not D0))
    (And (Not D0) (Not D1))
```

Code Trees

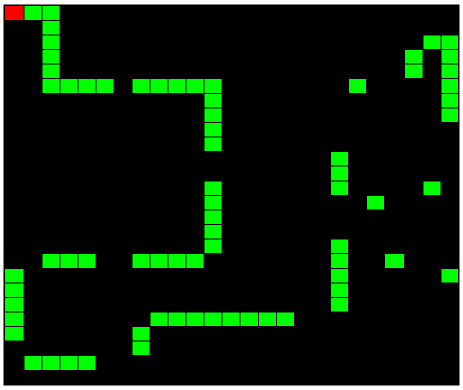


Crossover - Swap Tree Segments



Example Applications

- Santa-Fe Trail Problem



Fitness: How much food collected

Individual program on the previous slide generated on 7th generation solved the problem completely

Example GP Problem

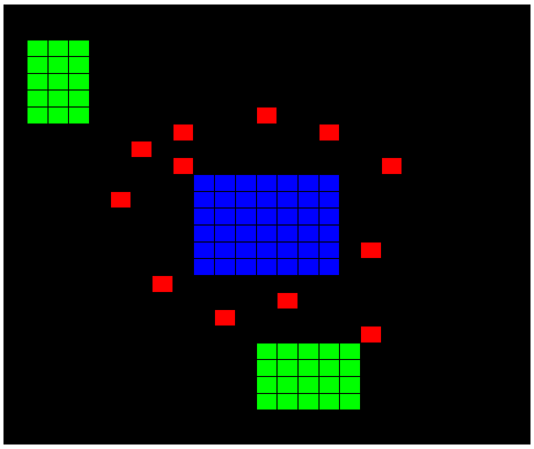
Examples: Artificial Ant Problem. Given a set environment with a trail of food, goal is to get as most of the food as possible in a given timeframe

Functions: IF-FOOD, PROGN
Terminals: ADVANCE, TURN LEFT, TURN RIGHT

After 51 generations with population of 1000, following individual emerged that got all the food:

```
(If-Food (Advance)
        (Progn (turn-right)
                (If-Food (Advance) (Turn-Left))
                (Progn (Turn-left)
                        (If-Food (Advance) (Turn-Right))
                                (Advance))))
```

Ants - Emergent Collective Behavior



Fitness: Food collected by all ants and returned to nest in given time period

Programs evolved to demonstrate collective intelligent behavior, lay pheromone trails

Other GA/GP Applications

- GA’s and GP has been used to solve many problems
  - Numerical optimization
  - Circuit Design
  - Factory Scheduling
  - Network Optimization
  - Robot Navigation
  - Economic Forecasting
  - Police Sketches
  - Flocking/Swarming Behavior
  - Evolutionary Art, Music
  - Windows NT?

Other Demos

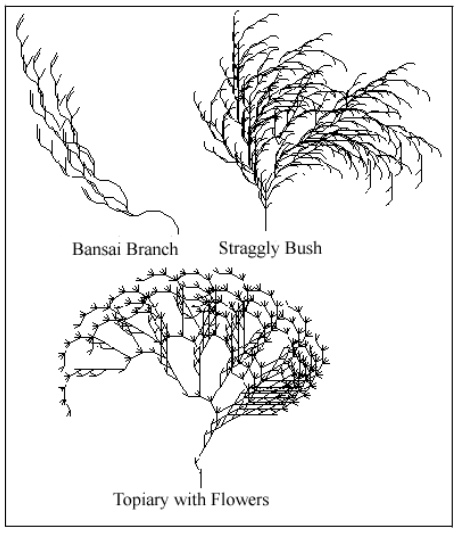
- Rule finding for boolean multiplexer
- Ant problem?
- Plants?

Extra Material

Where we are going...

Simple mechanism to evolve interesting computer-generated plants within the virtual environment.

Can play with applet from web page; http://www.math.uaa.alaska.edu/~afkjm



L-Systems

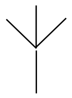
- Plants are based on L-Systems
  - Introduced by Lindenmayer in 1968 to simulate the development of multi-cellular organisms
- Simplified "Turtle-Logo" version implemented in Wildwood
  - Based on a context-free grammar
  - String re-writing rules
    - Recursively replace productions on RHS via transformation rules, ignoring productions at end.
    - **A=FfAFF** **Order(1) : A=FfFF**
    - **A=FfFfAFFFF** **Order(2) : A=FfFfFFF**
    - **A=FfFfFfAFFFFFF** **Order(3) : A=FfFfFfFFFFFF**

L-Systems Terminals

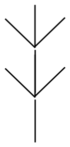
- Turtle-graphics interpretation of grammar terminals. Assume a pen is attached to the belly of a turtle.
  - F : Move forward and draw a line segment
  - f : Move forward without drawing a segment
  - - : Rotate left R degrees
  - + : Rotate right R degrees
  - [: Push current angle/position on stack
  -] : Pop and return to state of last push
- R set to 30 degrees
- Push and Pop comprise a bracketed L-System necessary for branching behavior

L-System Grammar Example

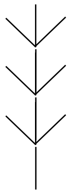
Initially, turtle pointing up.



Initial grammar: A=F[-F][F][+F]A
Order(1): F[-F][F][+F]



One rewrite rule:
A=F[-F][F][+F]F[-F][F][+F]A
Order(2):
F[-F][F][+F]F[-F][F][+F]



Two rewrite rules:
A=F[-F][F][+F]F[-F][F][+F]F[-F][F][+F]A
Order(2):
F[-F][F][+F]F[-F][F][+F]F[-F][F][+F]

Extremely simple formalism for generating complex structures

Genetic Algorithm Components

- Apply the traditional genetic algorithm paradigm to the evolution of L-System plants
- Use a single grammar rule as the chromosome
- Mutation: Randomly generates new grammar string and inserts into random location
- Crossover: Randomly swap subsections of the grammar string:

  Parent1: A=F[-fF++F]fAfFA
  Parent2: A=FFAFFF**AFFF**

  Child1: A=F[-fF**AFFF**]fAfFA
  Child2: A=FFAFFF**++F**

Also computes valid sequences that preserve bracket matching

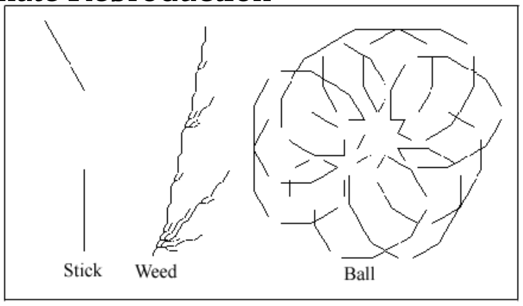
Genetic Operations

- Random String Generation
  - Rule length selected randomly between 4-20
  - Terms from set {"f","F","[","]","+", "-", "A"} selected at random with equal probability
  - probability of a "]" increases proportionally with respect to the remaining slots in the string and the number of right brackets necessary to balance the rule**.**
- Fitness Function
  - User-determined, e.g. genetic art, hand-bred
  - Height / Width

Hand-Bred Plants

- Humans assign fitness values
- Mutation = 0.05, Popsize=10, Order=4
- Fitness Proportionate Reproduction

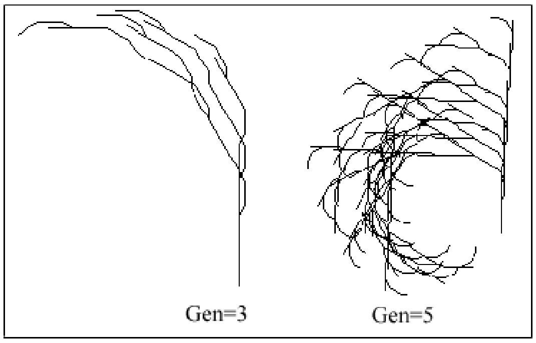
Generation 0:
Not very plant-like



Stick: A=Ff-F
Weed: A=F+F[+-A]Ff-AFFAF
Ball: A=F+f+AAFF+FA

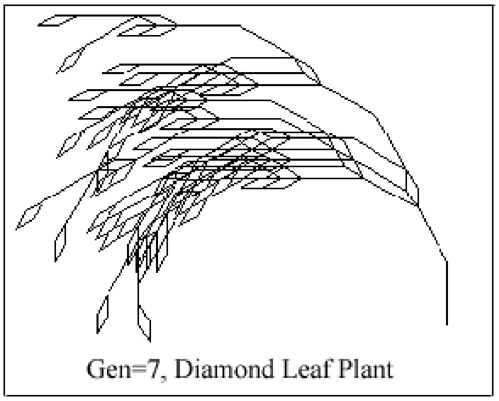
Hand-Bred Plants

- More plant-like individuals generated as evolution progresses.
  - Gen 3 : A=FF[F[A+F[-FF-AFF]]]
  - Gen 5 : A=FF[F[A+F[-FF-+F[-FF-FFA[-F[]A[]]]AFF]]]F



Final Plant

- Stopped at generation 7
  - Gen 7: A= FF[F-[F[A+F[-[A][]]]][A+F[-F-A[]]]]F


Gen=7, Diamond Leaf Plant