Artificial Intelligence Programming

Local Search

Cindi Thompson

Department of Computer Science

University of San Francisco

Genetic Algorithms

- Can be thought of as a form of parallel hill-climbing search.
- Basic idea:
  - Select some solutions at random.
  - Combine the best parts of the solutions to make new solutions.
  - Repeat.
- Successors are a function of *two* states, rather than one.

GA applications

- Function optimization
- Job Shop Scheduling
- Factory scheduling/layout
- Circuit Layout
- Molecular structure

GA terminology

- Chromosome - a solution or state
- Trait/gene - a parameter or state variable
- Fitness - the “goodness” of a solution
- Population - a set of chromosomes or solutions.

A Basic GA

```
pop = makeRandomPopulation
while (not done)
   new-pop = []
   foreach p in pop
      p.fitness = evaluate(p)
   for i = 1 to size(pop) by 2:
     parent1, parent2 = select random solutions from pop
     child1, child2 = reproduce (parent1, parent2)
     if rand() > epsilon:
        mutate child1, child2
   replace old population with new population
```

Analogies to Biology

- Keep in mind that this is *not* how biological evolution works.
  - Biological evolution is much more complex.
  - Diploid chromosomes, phenotype/genotype, nonstationary objective functions, ...
- Biology is a nice metaphor.
  - GAs must stand or fail on their own merits.

Encoding a Problem

- Encoding a problem for use by a GA can be quite challenging.
- Traditionally, GA problems are encoded as bitstrings
- Example: 8 queens. For each column, we use 3 bits to encode the row of the queen = 24 bits.
- 100 101 110 000 101 001 010 110 = 4 5 6 0 5 1 2 6
- We begin by generating random bitstrings, then evaluating them according to a *fitness function* (the function to optimize).
  - 8-queens: number of nonattacking pairs of queens (max = 28)

Generating new solutions

- Successor function will operate on two solutions.
- Called *crossover*.
- Pick two solutions to be parents, $p1$ and $p2$
  - Go into *how* to choose parents in a bit
- Pick a random point on the bitstrings. (locus)
- Merge the first part of $p1$ with the second part of $p2$ (and vice versa) to produce two new bitstrings.

Crossover Example

s1: 100 101 110 000 101 001 010 110 = 4 5 6 0 5 1 2 6
s2: 001 000 101 110 111 010 110 111 = 1 0 5 6 7 2 6 7

- Pick locus = 9
- s1: (100 101 110) (000 101 001 010 110)
- s2: (001 000 101) (110 111 010 110 111)

Crossover:

- s3: (100 101 110) (110 111 010 110 111)
  = 4 5 6 6 7 2 6 7
- s4: (001 000 101) (000 101 001 010 110)
  = 1 0 5 0 5 1 2 6

Mutation

- Next, apply mutation.
- With probability m (for small m) randomly flip one bit in the solution.
- After generating a new population of the same size as the old population, discard the old population and start again.

So what is going on?

- Why would this work?
- Crossover: recombine pieces of partially successful solutions.
- Genes closer to each other are more likely to stay together in successive generations.
  - This makes encoding important.
- Mutation: inject new solutions into the population.
  - If a trait was missing from the initial population, crossover cannot generate it unless we place the locus within a gene.

Selection

How should we select parents for reproduction?

Selection

- How should we select parents for reproduction?
- Use the best n percent?
  - Want to avoid premature convergence
  - No genetic variation
  - Also, sometimes poor solutions have promising subparts.
- Purely random?
  - No selection pressure

Roulette Selection

- *Roulette Selection* weights the probability of a chromosome being selected by its relative fitness.
- $P(c) = \frac{fitness(c)}{\sum_{chr \in pop} fitness(chr)}$
- This normalizes fitnesses; total relative fitnesses will sum to 1.
- Can directly use these as probabilities.

Example

- Suppose we want to maximize $f(x) = x^2$ on $[0, 31]$
  - Let's assume integer values of x for the moment.
- Five bits used to encode solution.
- Generate random initial population

String	Fitness	Relative Fitness
01101	169	0.144
11000	576	0.492
01000	64	0.055
10011	361	0.309
Total	1170	1.0

Example

- Select parents with roulette selection.
- Choose a random *locus*, and crossover the two strings

String	Fitness	Relative Fitness
0110 | 1	169	0.144
1100 | 0	576	0.492
01000	64	0.055
10011	361	0.309
Total	1170	1.0

- Children: 01100, 11001

Example

- Select parents with roulette selection.
- Choose a random *locus*, and crossover the two strings

String	Fitness	Relative Fitness
01101	169	0.144
11| 000	576	0.492
01000	64	0.055
10| 011	361	0.309
Total	1170	1.0

- Children: 01100, 11001
- Children: 10000, 11011

Example

- Replace old population with new population.
- Apply mutation to new population.
  - With a small population and low mutation rate, mutations are unlikely.
- New generation:
  - 01100, 11001, 11011, 10000
- Average fitness has increased (293 to 439)
- Maximum fitness has increased (576 to 729)

What's really going on here?

- The subsolutions 11*** and ***11 are recombined to produce a better solution.
- There's a correlation between strings and fitness.
  - Having a 1 in the first position is correlated with fitness.
  - This shouldn't be shocking, considering how we encoded the input.

Schemas (Schemata)

- A way to talk about strings that are similar to each other
- Add * (don't care) symbol to $0, 1$
- A schema is a template that describes a set of strings using $\{0, 1, *\}$
    - 111** matches 11100, 11101, 11110, 11111
    - 0*11* matches 00110, 00111, 01110, 01111
    - 0***1 matches ?
- Premise: Schemas are correlated with fitness
- In many encodings, only some bits matter to the solution
- Schemas provide a way to describe the important information

Schemas

- GAs process schemas rather than strings
- Crossover may or may not damage a schema
  - **11* vs 0***1
- Short, highly fit low-order schema are more likely to survive
  - Order: the number of fixed bits in a schema
    - 1**** - order ?
    - 0*1*1* - order ?

Schema Theorem

Building block hypothesis: GAs work by combining low-order schemas into higher-order schemas to produce progressively more fit solutions

Schema Theorem:
Short, low-order, above average fitness schemata receive exponentially increasing trials in subsequent generations

Theory vs. Implementation

- Schema Theorem shows us *why* GAs work.
- In practice, implementation details can make a big difference in the effectiveness of a GA.
- This includes algorithmic improvements and encoding choices.

Tournament Selection

- Roulette selection is nice, but can be computationally expensive.
  - Every individual must be evaluated.
  - Two iterations through entire population.
- *Tournament selection* is a much less expensive selection mechanism.
- For each parent, choose (small) n individuals at random.
- Highest fitness gets to reproduce.

Elitism

- In practice, discarding all solutions from a previous generation can slow down a GA.
  - Bad draw can destroy progress
  - You may need monotonic improvement.
- *Elitism* is the practice of keeping a fraction of the population from the previous generation.
- Can keep top N solutions, or else use roulette selection to choose a fraction of the population to carry over without crossover.
- Varying the fraction retained lets you trade current performance for learning rate.

Knowing when to stop

- Stop whenever the GA finds a "Good enough" solution
- What if we don't know what "Good enough" is?
  - How do we know we've found the best solution to TSP?
- Stop when population has 'converged'
  - Without mutation, eventually one solution will dominate the population
- After 'enough' iterations without improvement

Encoding

- The most difficult part of working with GAs is determining how to encode problem instances.
  - Parameters that are interrelated should be located near each other.
- N queens: Assume that each queen will go in one column.
- Problem: find the right row for each queen
- N rows requires $log_2 N$ bits
- Entire length of string: $N * log_2 N$

Encoding Real-valued numbers

- What if we want to optimize a real-valued function?
- $f(x) = x^2, x \in Reals[0, 31]$
- Decide how to discretize input space; break into m “chunks”
- Each chunk coded with a binary number.
- This is called *discretization*

Permutation operators

- Some problems don't have a natural bitstring representation.
- e.g. Traveling Salesman
  - Encoding this as a bitstring will cause problems
  - Crossover will produce lots of invalid solutions
- Encode this as a list of cities: [3,1,2,4,5]
- Fitness: MAXTOUR - tour length (to turn minimization into maximization.)

Crossover and Permutations

- We can't just crossover two solutions:
  - Result is very unlikely to be a permutation.
- Instead, we can use permuation crossover.
- With single-point:
  - Select a crossover point n.
  - For child 1, c[0:n] = parent1[0:n]
  - For c[n:], iterate through parent 2.
    - If a value is not already present in child1, add it.
  - Do the complementary action for child 2.

Crossover and Permutations

Example:

- p1 = [2,3,4,1,6,5,7], p2 = [7,1,3,4,5,2,6]
- We choose the point between index 3 and 4 to crossover.
- c1 initially gets [2,3,4,1] from p1.
- Then we iterate through p2 and add missing cities in order. [7,5,6]
- c1 = [2,3,4,1,7,5,6]
- c2 = [7,1,3,4,2,6,5]

Greedy Crossover

- For TSP, we can also use *greedy crossover*
- For c1, start with the city in position 0 in p1.
- Then examine the cities in position 1 for each parent and choose the one closest to position 0.
- If this causes a cycle, choose the other city.
- If this also causes a cycle, choose a non-cycling vertex at random.
- Repeat for all cities.
- Do the complementary operation for child 2

Pseudocode

```
let p1, p2 be parents
c[0] = p1[0]
for i = 1 ...len(p1) :
  if dist(c[i-1],p1[i]) < dist(c[i-1],p2[i]) :
     c[i] = p1[i] (if cycle, use p2[i])
  else :
      c[i] = p2[i] (if cycle, use p1[i])
  if still cycle, choose unused city for c[i]
```

Summary

- GAs use bitstrings to perform local search through a space of possible schema.
- Quite a few parameters to play with in practice.
- Representation is the hardest part of the problem.
- Very effective at searching vast spaces.