Lecture 7: Genetic Algorithms

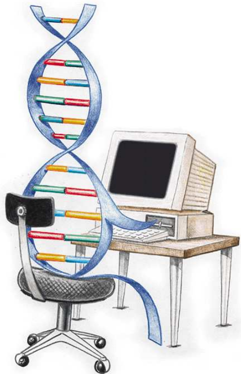
Cognitive Systems II - Machine Learning

Part II: Special Aspects of Concept Learning

Genetic Algorithms, Genetic Programming, Models of Evolution

last change November 29, 2007

Motivation



- Learning methods are motivated by analogy to biological evolution

- rather than search from general-to-specific or from simple-to-complex, genetic algorithms generate successor hypotheses by repeatedly mutating and recombining parts of the best currently known hypotheses
- at each step, a collection of hypotheses, called the current population, is updated by replacing some fraction by offspring of the most fit current hypotheses

Motivation

- reasons for popularity
  - evolution is known to be a successful, robust method for adaption within biological systems
  - genetic algorithms can search spaces of hypotheses containing complex interacting parts, where the impact of each part on overall hypothesis fitness may be difficult to model
  - genetic algorithms are easily parallelized
- genetic programming $\approx$ entire computer programs are evolved to certain fitness criteria
- evolutionary computation $=$ genetic algorithms $+$ genetic programming

Genetic Algorithms

- **problem:** search a space of candidate hypotheses to identify the best hypothesis
- the best hypothesis is defined as the one that optimizes a predefined numerical measure, called **fitness**
  - e.g. if the task is to learn a strategy for playing chess, fitness could be defined as the number of games won by the individual when playing against other individuals in the current population
- **basic structure:**
  - iteratively updating a pool of hypotheses (**population**)
  - on each iteration
    - hypotheses are evaluated according to the **fitness function**
    - a new population is generated by selecting the most fit individuals
    - some are carried forward, others are used for creating new offspring individuals

Genetic Algorithms

GA$(Fitness, Fitness_threshold, p, r, m)$

Fitness: fitness function, Fitness_threshold: termination criterion,
p: number of hypotheses in the population, r: fraction to be replaced by crossover,
m: mutation rate

- Initialize population: $P \leftarrow$ Generate p hypotheses at random
- Evaluate: For each h in P, compute $Fitnes(h)$
- While $[\max_h Fitness(h)] < Fitness_threshold$, Do
  1. **Select:** Probabilistically select $(1 - r) \cdot p$ members of P to add to P_S
  2. **Crossover:** Probalistically select $\frac{r \cdot p}{2}$ pairs of hypotheses from P. For each pair $< h_1, h_2 >$ produce two offspring and add to P_S
  3. **Mutate:** Choose m percent of the members of P_S with uniform probability. For each, invert one randomly selected bit
  4. **Update:** $P \leftarrow P_S$
  5. **Evaluate:** for each $h \in P$, compute $Fitness(h)$
- Return the hypothesis from P that has the highest fitness.

Remarks

- as specified above, each population P contains p hypotheses
- $(1-r) \cdot p$ hypotheses
  - are selected and added to P_S without changing
  - the selection is probabilistically
  - the probability is given by $Pr(h_i) = \frac{Fitness(h_i)}{\sum_{j=1}^{p} Fitness(h_j)}$
- $\frac{r \cdot p}{2}$ pairs of hypotheses
  - are selected and added to P_S after applying the crossover operator
  - the selection is also probabilistically

$\Rightarrow$ $(1-r) \cdot p + 2 \cdot \frac{r \cdot p}{2} = p$ where $r + (1-r) = 1$

Representing Hypotheses

- hypotheses are often represented as **bit strings** so that they can easily be modified by genetic operators
- represented hypotheses can be quite complex
- each attribute can be represented as a subtring with as many positions as there are possible values
- to obtain a fixed-length bit string, each attribute has to be considered, even in the most general case
  - $(Outlook = Overcast \vee Rain) \wedge (Wind = Strong)$
  - is represented as: $Outlook\ 011$, $Wind\ 10 \Rightarrow 01110$

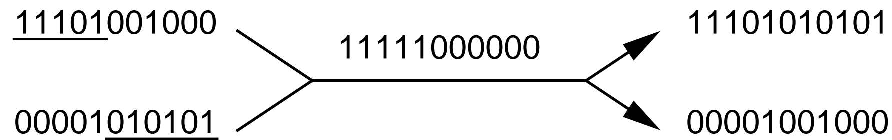
Genetic Operators

- generation of successors is determined by a set of operators that recombine and mutate selected members of the current population
- operators correspond to idealized versions of the genetic operations found in biological evolution
- the two most common operators are **crossover** and **mutation**

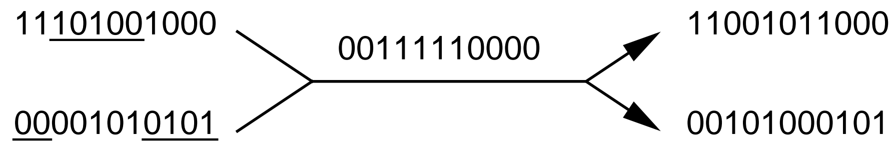
Genetic Operators

Initial strings *Crossover Mask* *Offspring*

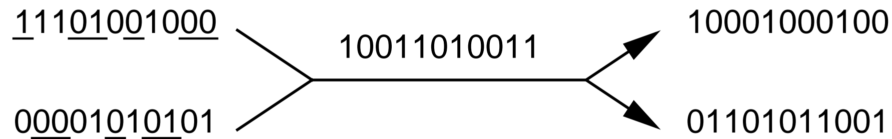
Single-point crossover:



Two-point crossover:



Uniform crossover:



Point mutation:

Genetic Operators

- **Cossover:**
  - produces two new offspring from two parent strings by copying selected bits from each parent
  - bit at position i in each offspring is copied from the bit at position i in one of the two parents
  - choice which parent contributes bit i is determined by an additional string, called **cross-over mask**
    - **single-point crossover:** e.g. 11111000000
    - **two-point crossover:** e.g. 00111110000
    - **uniform crossover:** e.g. 01100110101
- **mutation:** produces bitwise random changes

Illustrative Example (GABIL)

- GABIL learns boolean concepts represented by a disjunctive set of propositional rules
- **Representation:**
  - each hypothesis is encoded as shown before
  - hypothesis space of rule preconditions consists of a conjunction of constraints on a fixed set of attributes
  - sets of rules are represented by concatenation
  - e.g. a_1, a_2 boolean attributes, c target attribute
    IF $a_1 = T \wedge a_2 = F$ THEN $c = T$;
    IF $a_2 = T$ THEN $c = F$

    $\Rightarrow 10\ 01\ 1\ 11\ 10\ 0$
    (bei zweiter Regel kein Constraint auf a_1, d.h. alle Werte erlaubt, kodiert als 11)

Illustrative Example (GABIL)

- **Genetic Operators:**
  - uses standard mutation operator
  - crossover operator is a two-point crossover to manage variable-length rules
  - **Fitness function:**
    - $Fitness(h) = (correct(h))^2$
    - based on classification accuracy where $correct(h)$ is the percent of all training examples correctly classified by hypothesis h

Hypothesis Space Search

- method is quite different from other methods presented so far
- neither general-to-specific nor simple-to-complex search is performed
- genetic algorithms can move **very abruptly**, replacing a parent hypothesis by an offspring which is radically different
- so this method is less likely to fall into some local minimum
- practical difficulty: **crowding**
  - some individuals that fit better than others reproduce quickly, so that copies and very similar offspring take over a large fraction of the population

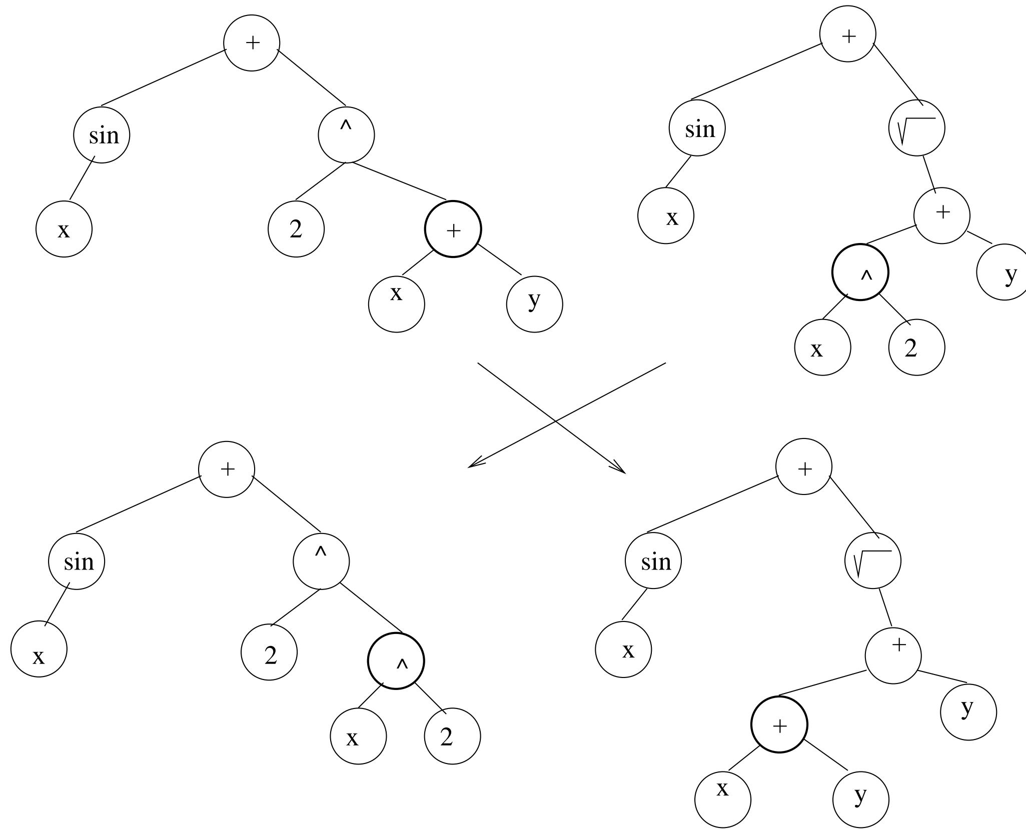
  $\Rightarrow$ reduced diversity of population

  $\Rightarrow$ slower progress of the genetic algorihms

Genetic Programming

- individuals in the evolving population are computer programs rather than bit strings
- has shown good results, despite vast H
- **representing programs**
  - typical representations correspond to parse trees
    - each function call is a node
    - arguments are the descendants
  - fitness is determined by executing the programm on the training data
  - crossover are performed by replacing a randomly chosen subtree between parents

Cross-Over



Approaches to Genetic Programming

- Learning from input/output examples
- Typically: functional programs (term representation)
- Koza (1992): e.g. Stacking blocks, Fibonacci
- Roland Olsson: ADATE, evolutionary computation of ML programs

Koza's Algorithm

1. Generate an initial population of random compositions of the functions and terminals of the problem.
2. Iteratively perform the following sub-steps until the termination criterium has been satisfied:
   (a) Execute each program in the population and assign it a fitness value according to how well it solves the problem.
   (b) Select computer program(s) from the current population chosen with a probability based on fitness.
   Create a new population of computer programs by applying the following two primary operations:
      i. Copy program to the new population (Reproduction).
      ii. Create new programs by genetically recombining randomly chosen parts of two existing programs.
3. The best so-far individual (program) is designated as result.

Genetic Operations

Mutation: Delete a subtree of a program and grow a new subtree at its place randomly.
This “asexual” operation is typically performed sparingly, for example with a probability of 1% during each generation.

Crossover: For two programs (“parents”), in each tree a cross-over point is chosen randomly and the subtree rooted at the cross-over point of the first program is deleted and replaced by the subtree rooted at the cross-over point of the second program.
This “sexual recombination” operation is the predominant operation in genetic programming and is performed with a high probability (85% to 90 %).

Genetic Operations

Reproduction: Copy a single individual into the next generation.
An individuum "survives" with for example 10% probability.

Architecture Alteration: Change the structure of a program.
There are different structure changing operations which are applied sparingly (1% probability or below):

- **Introduction of Subroutines:** Create a subroutine from a part of the main program and create a reference between the main program and the new subroutine.
- **Deletion of Subroutines:** Delete a subroutine; thereby making the hierarchy of subroutines narrower or shallower.

Genetic Operations

Architecture Alteration: Subroutine Duplication: Duplicate a subroutine, give it a new name and randomly divide the preexisting calls of the subroutine between the old and the new one. (This operation preserves semantics. Later on, each of these subroutines might be changed, for example by mutation.)

Argument Duplication: Duplicate an argument of a subroutine and randomly divide internal references to it. (This operation is also semantics preserving. It enlarges the dimensionality of the subroutine.)

Argument Deletion: Delete an argument; thereby reducing the amount of information available to a subroutine (“generalization”).

Genetic Operations

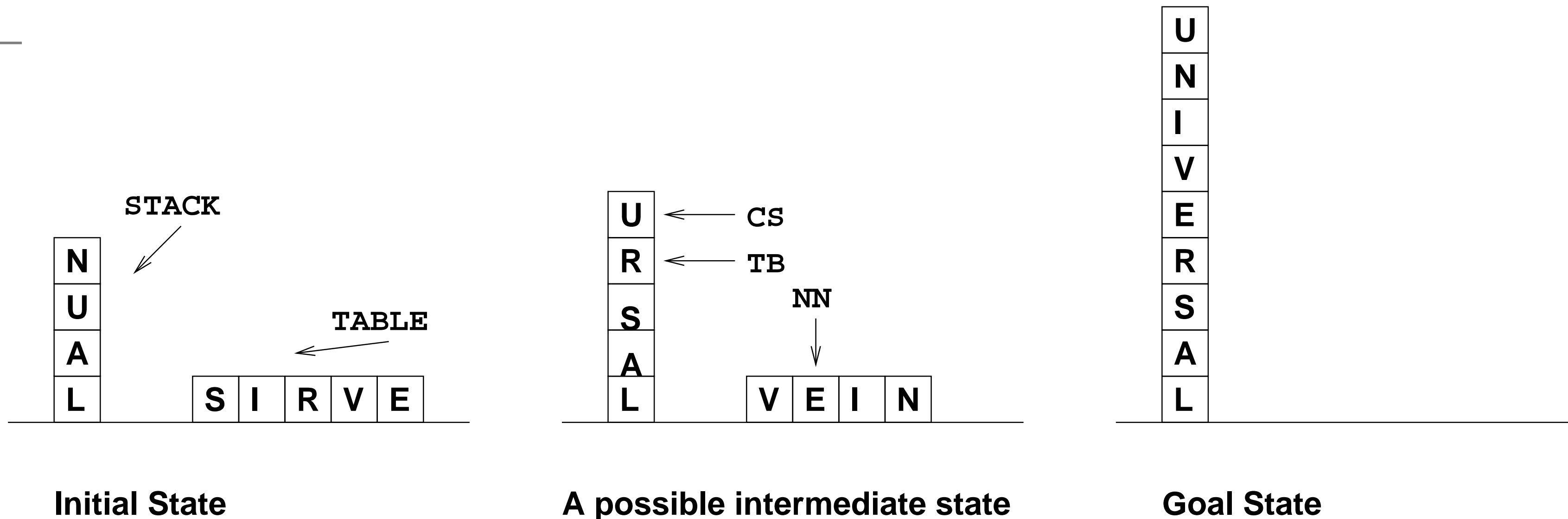
Automatically Defined Iterations/Recursions: Introduce or delete iterations (ADIs) or recursive calls (ADRs). Introduction of iterations or recursive calls might result in non-termination. Typically, the number of iterations (or recursions) is restricted for a problem. That is, for each problem, each program has a time-out criterium and is terminated “from outside” after a certain number of iterations.

Automatically Defined Stores: Introduce or delete memory (ADSs).

Fitness

- Quality criteria for programs synthesized by genetic programming:
  - correctness: defined as 100% fitness for the given examples
  - efficiency
  - parsimony
- The two later criteria can be additionally coded in the fitness measure.

Learning to Stack Blocks



Terminals $T = \{CS, TB, NN\}$

***CS*:** A sensor that dynamically specifies the top block of the *Stack*.

***TB*:** A sensor that dynamically specifies the block in the *Stack* which together with all blocks under it are already positioned correctly. (“top correct block”)

***NN*:** A sensor that dynamically specifies the block which must be stacked immediately on top of *TB* according to the goal. (“next needed block”)

Learning to Stack Blocks

Set of functions $\{MS, MT, NOT, EQ, DU\}$

***MS*:** A move-to-stack operator with arity one which moves a block from the *Table* on top of the *Stack*.

***MT*:** A move-to-table operator with arity one which moves a block from the top of the *Stack* to the *Table*.

***NOT*:** A boolean operator with arity one switching the truth value of its argument.

***EQ*:** A boolean operator with arity two which returns true if its arguments are identical and false otherwise.

***DU*:** A user-defined iterative “do-until” operator with arity two. The expression `DU *Work* *Predicate*` causes `*Work*` to be iteratively executed until `*Predicate*` is satisfied.

Learning to Stack Blocks

All functions have defined outputs for all conditions: *MS* and *MT* change the *Table* and *Stack* as side-effect. They return *true*, if the operator can be applied successfully and *nil* (*false*) otherwise. The return value of *DU* is also a boolean value indicating whether `*Predicate*` is satisfied or whether the *DU* operator timed out.

Learning to Stack Blocks

- Terminals functions carefully crafted. Esp. the pre-defined sensors carry exactly that information which is relevant for solving the problem!
- 166 fitness cases:
  ten cases where zero to all nine blocks in the stack were already ordered correctly; eight cases where there is one out of order block on top of the stack; and a random sampling of 148 additions cases.
- Fitness was measured as the number of correctly handled cases.

Learning to Stack Blocks

- Three variants
  1. first, correct program first moves all blocks on the table and than constructs the correct stack. This program is not very efficient because there are made unnecessary moves from the stack to the table for partially correct stacks. Over all 166 cases, this function generates 2319 moves in contrast to 1642 necessary moves.
  2. In the next trial, efficiency was integrated into the fitness measure and as a consequence, a function calculating only the minimum number of moves emerged. But this function has an outer loop which is not necessary.
  3. By integrating parsimony into the fitness measure, the correct, efficient, and parsimonious function is generated.

Learning to Stack Blocks

Population Size: M = 500

Fitness Cases: 166

Correct Program:

Fitness: *166 - number of correctly handled cases*

Termination: Generation 10

```
(EQ (DU (MT CS) (NOT CS))
    (DU (MS NN) (NOT NN)))
```

Learning to Stack Blocks

Correct and Efficient Program:

Fitness: $0.75 \cdot C + 0.25 \cdot E$ with

$C = ($ *number of correctly handled cases*$/166) \cdot 100$

$E = f(n)$ *as function of the total number of moves over all 166 cases:*

with $f(n) = 100$ *for the analytically obtained minimal number of moves for a correct program (*$min = 1641$*);*

$f(n)$ *linearly scaled upwards for zero moves up to 1640 moves with* $f(0) = 0$

$f(n)$ *linearly scaled downwards for 1642 moves up to 2319 moves (obtained by the first correct program) and* $f(n) = 0$ *for* $n > 2319$

Termination: Generation 11

Learning to Stack Blocks

```
(DU (EQ (DU (MT CS) (EQ CS TB))
            (DU (MS NN) (NOT NN)))
       (NOT NN))
```

Learning to Stack Blocks

Correct, Efficient, and Parsimonious Program:

Fitness: $0.70 \cdot C + 0.20 \cdot E + 0.10 \cdot$ *(number of nodes in program tree)*

Termination: Generation 1

```
(EQ (DU (MT CS) (EQ CS TB))
    (DU (MS NN) (NOT NN)))
```

ADATE

Models of Evolution and Learning

- **observations:**
  - individual organisms learn to adapt significantly during their lifetime
  - biological and social processes allow a species to adapt over a time frame of many generations
- interesting question: What is the relationship between learning during lifetime of a single individual and species-level learning afforded by evolution?

Models of Evolution and Learning

- **Lamarckian Evolution:**
  - proposition that evolution over many generations was directly influenced by the experiences of individual organisms during their lifetime
  - direct influence of the genetic makeup of the offspring
  - completely contradicted by science
  - Lamarckian processes can sometimes improve the effectiveness of genetic algorithms
- **Baldwin Effect:**
  - a species in a changing environment underlies evolutionary pressure that favors individuals with the ability to learn
  - such individuals perform a small local search to maximize their fitness
  - additionally, such individuals rely less on genetic code
  - thus, they support a more diverse gene pool, relying on individual learning to overcome “missing” or “not quite well” traits

  $\Rightarrow$ indirect influence of evolutionary adaption for the entire population

Summary

- method for concept learning based on simulated evolution
- evolution of populations is simulated by taking the most fit individuals over to a new generation
- some individuals remain unchanged, others are the base for genetic operator application
- hypotheses are commonly represented as bitstrings
- search through the hypothesis space cannot be characterized, because hypotheses are created by crossover and mutation operators that allow radical changes between successive generations
- hence, convergence is not guaranteed