Unit V

Genetic Algorithms

Overview of Genetic Algorithms(GAs)

- GA is a learning method motivated by analogy to biological evolution.
- GAs search the hypothesis space by generating successor hypotheses which repeatedly mutate and recombine parts of the best currently known hypotheses.
  - Rather than search from general-to-specific hypotheses, or from simple-to-complex
- At each step, a collection of hypotheses called the current population is updated by replacing some fraction of the population by offspring of the most fit current hypotheses.
- Genetic algorithms and genetic programming are two of the more popular approaches in a field that is sometimes called evolutionary computation.

- The popularity of GAs is motivated by a number of factors including:
  - Evolution is known to be a successful, robust method for adaptation within biological systems.
  - GAS 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 and can take advantage of the decreasing costs of powerful computer hardware.

Genetic Algorithms

- GAs is to search a space of candidate hypotheses to identify the best hypothesis.
- In GAs the "best hypothesis" is defined as the one that optimizes a predefined numerical measure for the problem at hand, called the hypothesis fitness.
- For example, if the learning task is the problem of approximating an unknown function given training examples of its input and output, then fitness could be defined as the accuracy of the hypothesis over this training data.
  - 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.

- The algorithm operates by iteratively updating a pool of hypotheses, called the population.
- On each iteration, all members of the population are evaluated according to the fitness function. A new population is then generated by probabilistically selecting the most fit individuals from the current population. Some of these selected individuals are carried forward into the next generation population intact. Others are used as the basis for creating new offspring individuals by applying genetic operations such as crossover and mutation.

Prototypical Genetic Algorithm

- **Initialize Population**: P← generate p hypotheses at random.
- **Evaluate**: for each h in P, compute fitness(h)
- While Max_h Fitness(h) < Threshold do
- Create a new generation P_{New}
  - **Select**: probabilistically select a fraction of the best p's in P to add to P_{New} .Call this new generation PNew
  - **Crossover**: probabilistically form pairs of the selected p's and produce two offsprings by applying the crossover operator. Add all offsprings to P_{New}.
  - **Mutate**: Choose m% of P_{New} with uniform probability. For each, invert one randomly selected bit in its representation.
  - **Update**: P ← P_{New}
  - **Evaluate**: for each p in P_{New}, compute fitness(p)
- Return the hypothesis from P that has the highest fitness.

Hypotheses Representation

- Hypotheses in GAs are often represented by bit strings, so that they can be easily manipulated by genetic operators such as mutation and crossover.
- For example, sets of if-then rules can easily be represented in this way, by choosing an encoding of rules that allocates specific substrings for each rule precondition and postcondition.
- Let see how to use a bit string to describe a constraint on the value of a single attribute.
- Consider the attribute Outlook, which can take on any of the three values Sunny, Overcast, or Rain.

- One obvious way to represent a constraint on Outlook is to use a bit string of length three, in which each bit position corresponds to one of its three possible values.
- Placing a 1 in some position indicates that the attribute is allowed to take on the corresponding value
  - For example, the string 010 represents the constraint that Outlook must take on the second of these values,or Outlook = Overcast.
  - The string 011 represents the more general constraint that allows two possible values, or (Outlook = Overcast v Rain)

- Knowing how to represent the constraints on single attribute, now conjunctions of constraints on multiple attributes can easily be represented by concatenating the corresponding bit strings.
- For example, consider a second attribute, Wind, that can take on the value Strong or Weak.
- A rule precondition such as (Outlook = Overcast V Rain) A (Wind = Strong) can then be represented by the following bit string of length five:

Outlook	Wind
01 1	10

- Rule postconditions (such as PlayTennis = yes) can be represented in a similar fashion.
- An entire rule can be described by concatenating the bit strings describing the rule preconditions, together with the bit string describing the rule postcondition.
- For example, the rule

  IF Wind = Strong THEN PlayTennis = yes

  would be represented by the string

  Outlook Wind PlayTennis

  111 10 10

where the first three bits describe the "don't care" constraint on ***Outlook***

- In designing a bit string encoding for some hypothesis space, it is useful to arrange for every syntactically legal bit string to represent a well-defined hypothesis.

Genetic Operators

- Genetic operators used in genetic algorithms maintain genetic diversity. Genetic diversity or variation is a necessity for the process of evolution.
- Genetic operators are analogous to those which occur in the natural world:
  - **Reproduction** (or Selection) ;
  - **Crossover** (or Recombination); and
  - **Mutation**.
- In addition to these operators, there are some parameters of GA. One important parameter is Population size.
  - Population size says how many chromosomes are in population (in one generation).
  - If there are only few chromosomes, then GA would have a few possibilities to perform crossover and only a small part of search space is explored.
  - If there are many chromosomes, then GA slows down.
- Research shows that after some limit, it is not useful to increase population size, because it does not help in solving the problem faster. The population size depends on the type of encoding and the problem

Fitness Function and selection

- The fitness function defines the criterion for ranking potential hypotheses and for probabilistically selecting them for inclusion in the next generation population.
- The probability that a hypothesis will be selected is given by the ratio of its fitness to the fitness of other members of the current population
- Reproduction is usually the first operator applied on population. From the population, the chromosomes are selected to be parents to crossover and produce offspring.
- The problem is how to select these chromosomes ?
- According to Darwin's evolution theory "survival of the fittest" – the best ones should survive and create new offspring.
  - The Reproduction operators are also called Selection operators.
  - Selection means extract a subset of genes from an existing population, according to any definition of quality. Every gene has a meaning, so one can derive from the gene a kind of quality measurement called fitness function. Following this quality (fitness value), selection can be performed.
  - Fitness function quantifies the optimality of a solution (chromosome) so that a particular solution may be ranked against all the other solutions. The function depicts the closeness of a given ‘solution’ to the desired result.

- Many reproduction operators exists and they all essentially do same thing. They pick from current population the strings of above average and insert their multiple copies in the mating pool in a probabilistic manner.
- The most commonly used methods of selecting chromosomes for parents to crossover are :
- Roulette wheel selection, - Rank selection
- Boltzmann selection, - Steady state selection.
- Tournament selection,

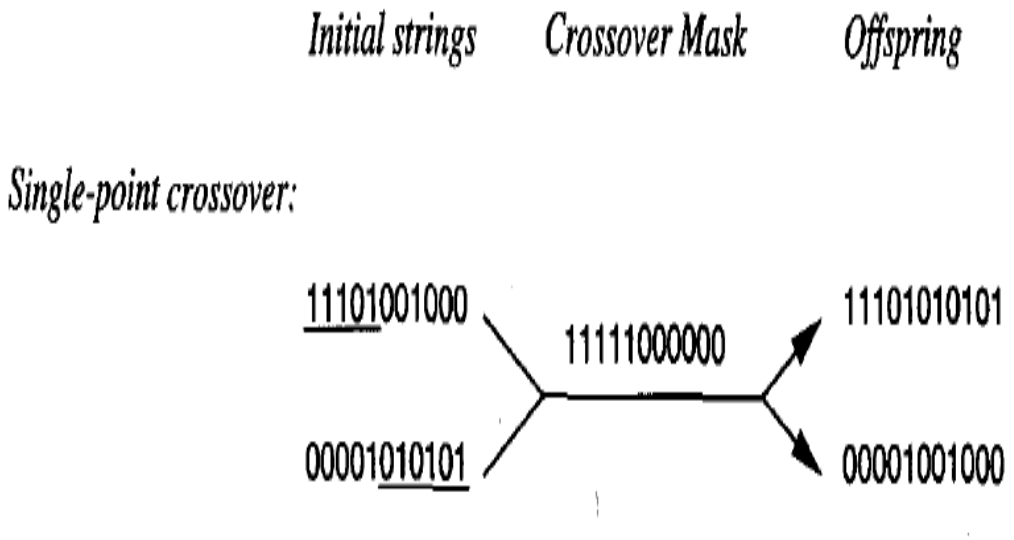
- Roulette Wheel Selection
  - Fitness Proportionate Selection is one of the most popular ways of parent selection. In this every individual (hypothesis) can become a parent with a probability which is proportional to its fitness.
  - In a roulette wheel selection, the circular wheel is divided into n pies, where n is the number of individuals in the population. Each individual gets a portion of the circle which is proportional to its fitness value.
  - A fixed point is chosen on the wheel circumference and the wheel is rotated. The region of the wheel which comes in front of the fixed point is chosen as the parent. For the second parent, the same process is repeated.
    - It is clear that a fitter individual has a greater pie on the wheel and therefore a greater chance of landing in front of the fixed point when the wheel is rotated. Therefore, the probability of choosing an individual depends directly on its fitness.

- Tournament Selection
  - In tournament selection, we select tow(or K) individuals from the population at random and select the best out of these to become a parent. The same process is repeated for selecting the next parent.
- Rank Selection
  - In rank selection, the individuals(hypotheses) in the current population are first sorted by fitness. The probability that a hypothesis will be selected is then proportional to its rank in this sorted list, rather than its fitness.

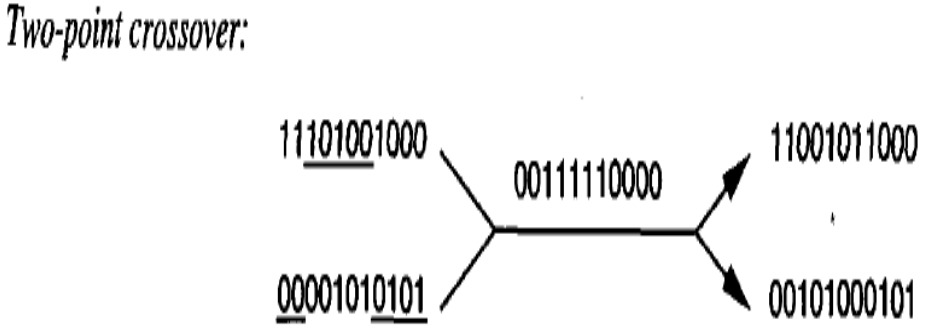
Crossover

- Crossover is a genetic operator that combines (mates) two chromosomes (parents) to produce a new chromosome (offspring).
- The idea behind crossover is that the new chromosome may be better than both of the parents if it takes the best characteristics from each of the parents.
- Crossover occurs during evolution according to a user-definable crossover probability.
- Crossover selects genes from parent chromosomes and creates a new offspring.
- The bit at position i in each offspring is copied from the bit at position i in one of the two parents. The choice of which parent contributes the bit for position i is determined by an additional string called the crossover mask
- The Crossover operators are of many types.
  - One-Point crossover.
  - Two Point
  - Uniform

- One-Point crossover operator randomly selects one crossover point and then copy everything before this point from the first parent and then everything after the crossover point copy from the second parent
- Consider the two parents selected for crossover.
  - Parent 1 1 1 0 1 1 | 0 0 1 0 0 1 1 0 1 1 0
  - Parent 2 1 1 0 1 1 | 1 1 0 0 0 0 1 1 1 1 0
- Interchanging the parents chromosomes after the crossover points - The Offspring produced are :
  - Offspring 1 1 1 0 1 1 | 1 1 0 0 0 0 1 1 1 1 0
  - Offspring 2 1 1 0 1 1 | 0 0 1 0 0 1 1 0 1 1 0



- Two-Point Crossover
  - Two-Point crossover operator randomly selects two crossover points within a chromosome then interchanges the two parent chromosomes between these points to produce two new offspring
- Consider the two parents selected for crossover :
  - Parent 1 1 1 0 1 1 | 0 0 1 0 0 1 1 | 0 1 1 0
  - Parent 2 1 1 0 1 1 | 1 1 0 0 0 0 1 | 1 1 1 0
- Interchanging the parents chromosomes between the crossover points - The Offspring produced are :
  - Offspring 1 1 1 0 1 1 | 0 0 1 0 0 1 1 | 0 1 1 0
  - Offspring 2 1 1 0 1 1 | 0 0 1 0 0 1 1 | 0 1 1 0



Uniform crossover operator decides (with some probability – know as the mixing ratio) which parent will contribute how the gene values in the offspring chromosomes. The crossover operator allows the parent chromosomes to be mixed at the gene level rather than the segment level (as with one and two point crossover).

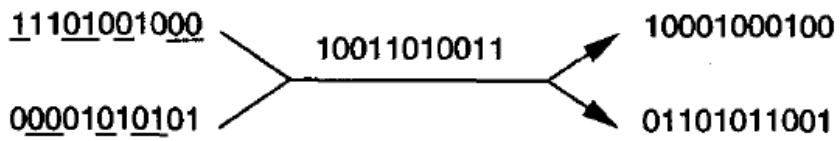
Consider the two parents selected for crossover.

Parent 1	1	1	0	1	1	0	0	1	0	0	1	1	0	1	1	0
Parent 2	1	1	0	1	1	1	1	0	0	0	0	1	1	1	1	0

If the mixing ratio is **0.5** approximately, then half of the genes in the offspring will come from parent **1** and other half will come from parent **2**. The possible set of offspring after uniform crossover would be:

Offspring 1	1_1	1_2	0_2	1_1	1_1	1_2	1_2	0_2	0_1	0_1	0_2	1_1	1_2	1_1	1_1	0_2
Offspring 2	1_2	1_1	0_1	1_2	1_2	0_1	0_1	1_1	0_2	0_2	1_1	1_2	0_1	1_2	1_2	0_1

Uniform crossover:



- Mutation
  - After a crossover is performed, mutation takes place.
  - Mutation is a genetic operator used to maintain genetic diversity from one generation of a population of chromosomes to the next.
  - The mutation operator produces small random changes to the bit string by choosing a single bit at random, then changing its value.
  - Mutation occurs during evolution according to a user-definable mutation probability, usually set to fairly low value, say 0.01 a good first choice.
  - Mutation alters one or more gene values in a chromosome from its initial state. This can result in entirely new gene values being added to the gene pool. With the new gene values, the genetic algorithm may be able to arrive at better solution than was previously possible.
  - Mutation is an important part of the genetic search, helps to prevent the population from stagnating at any local optima. Mutation is intended to prevent the search falling into a local optimum of the state space

- The Mutation operators are of many type.
  - Flip Bit.
  - Boundary, Non-Uniform, Uniform, and Gaussian.
- The operators are selected based on the way chromosomes are encoded .
- In the Flip bit mutation, we select one or more random bits and flip them.
  - The mutation operator simply inverts the value of the chosen gene.
    - i.e. 0 goes to 1 and 1 goes to 0.
  - This mutation operator can only be used for binary genes.

Consider the two original off-springs selected for mutation.

Original offspring 1	1	1	0	1	1	1	1	0	0	0	0	1	1	1	1	0
Original offspring 2	1	1	0	1	1	0	0	1	0	0	1	1	0	1	1	0

Invert the value of the chosen gene as **0** to **1** and **1** to **0**

The Mutated Off-spring produced are :

Mutated offspring 1	1	1	0	**0**	1	1	1	0	0	0	0	1	1	1	1	0
Mutated offspring 2	1	1	0	1	1	0	**1**	1	0	0	1	1	0	1	**0**	0

Hypothesis Space search

- GA search can move very abruptly (as compared to Backpropagation, for example), replacing a parent hypothesis by an offspring that may be radically different from the parent.
  - In Backpropagation we moved smoothly from one hypothesis to a new hypothesis that is very similar.
- One practical difficulty in some GA applications is the problem of crowding.
- Crowding is a phenomenon in which some individual that is more highly fit than others in the population quickly reproduces, so that copies of this individual and very similar individuals take over a large fraction of the population.
  - The negative impact of crowding is that it reduces the diversity of the population, thereby slowing further progress by the GA.

Population Evolution and the Schema Theorem

- Can mathematically characterize the evolution over time of the population within a GA
  - Schema theorem serves as the analysis tool for the GA process
- The Schema Theorem is based on the concept of schemas, or patterns that describe sets of bit strings.
- A schema is any string composed of Os, 1s, and *'s.
- Each schema represents the set of bit strings containing the indicated 0s and Is, with each "*" interpreted as a "don't care.“
  - For example, the schema 0*10 represents the set of bit strings that includes exactly 0010 and 01 10

- An individual bit string can be viewed as a representative of each of the different schemas that it matches.
  - For example, the bit string 0010 can be thought of as a representative of 24 distinct schemas including 00**, O* 10, ****, etc.
- The schema theorem characterizes the evolution of the population within a GA in terms of the number of instances representing each schema.
- Let m(s, t) denote the number of instances of schema s in the population at time t (i.e.,during the t^{th} generation).
- The schema theorem describes the expected value of m(s, t + 1) in terms of m(s, t) and other properties of the schema, population, and GA algorithm parameters.

- The evolution of the population in the GA depends on
  - the selection step,
  - the recombination step,
  - and the mutation step.

- Let us start by considering just the effect of the selection step.
  - Let f (h) denote the fitness of the individual bit string h and $\bar{f}(t)$ denote the average fitness of all individuals in the population at time t.
  - Let n be the total number of individuals in the population.
  - Finally, let $\hat{u}(s, t)$ denote the average fitness of instances of schema s in the population at time t.
  - E[m(s, t + I)] denotes the expected value of m(s,t+1)

- Let m(s, t) denote the number of instances of schema s in the population at time t (i.e.,during the t^{th} generation)
  - m= m(s,t)
- Let f (h) denote the fitness of the individual bit string h and f(t) denote the average fitness of all individuals in the population at time t. Let n be the total number of individuals in the population.
- During reproduction, an individual bit stringgets selected according to its fitness with probability

$$\Pr(h) = \frac{f(h)}{\sum_{i=1}^{n} f(h_i)} = \frac{f(h)}{n\bar{f}(t)}$$

- Now if we select one member for the new population according to this probability distribution, then the probability that we will select a representative of schema s is

$$\Pr(h \in s) = \sum_{h \in s \cap p_t} \frac{f(h)}{n\bar{f}(t)}$$

$$= \frac{\hat{u}(s,t)}{n\bar{f}(t)} m(s,t)$$

- where u^(s, t) denote the average fitness of instances of schema s in the population at time t.

- The average fitness of the instances of the schema is
  - $$\hat{u}(s,t) = \frac{\sum_{h \in s \cap p_t} f(h)}{m(s,t)}$$

- Therefore, the expected number of instances of s resulting from the n independent selection steps that create the entire new generation is just n times this probability. $$E[m(s,t+1)] = \frac{\hat{u}(s,t)}{\bar{f}(t)} m(s,t)$$

- It states that the expected number of instances of schema ***s*** at generation ***t*** + 1 is proportional to the average fitness ***i(s, t)*** of instances of this schema at time ***t*** **,** and inversely proportional to the average fitness ***f(t)*** of all members of the population at time t.

- A crossover site is selected uniformly among m-1 possible sites, where m is the length of the bitstring representing an individual. Hence d(S) / (m-1) is the probability that the schema S is destroyed if a certain string (or chromosome) undergoes crossover, and thus
- $p_s >= 1 - p_c . d(s)/(l-1)$

- Where p_c is the probability of crossover.

- Assuming independence of the reproduction and crossover operations,

$$E[m(s,t+1)] \geq \frac{\hat{u}(s,t)}{\bar{f}(t)} m(s,t) \left(1 - p_c \frac{d(s)}{l-1}\right)$$

- Effect of Mutation
  - Mutation is the random alteration of a single position with probability p_m
  - A **single gene** survives with a **probability**(1-$\mathbf{p_m}$)
  - Since each of the mutations is statistically independent, a particular schema S survives when each of the o(S)fixed positions within the schema survives.
  - The survival probability is multiplied by itself o(S) times:

$$(1 - p_m)^{o(s)}$$

  - The full schema theorem thus provides a lower bound on the expected frequency of schema s, as follows

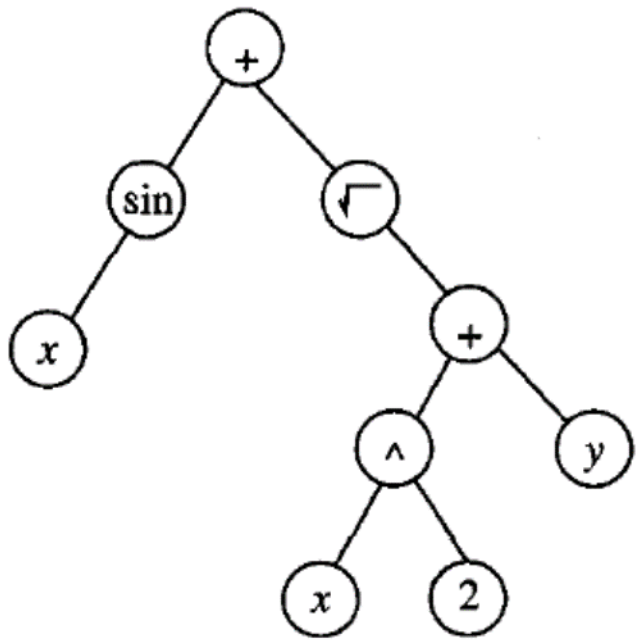
$$E[m(s, t+1)] \geq \frac{\hat{u}(s,t)}{\bar{f}(t)} m(s,t) \left(1 - p_c \frac{d(s)}{l-1}\right) (1 - p_m)^{o(s)}$$

- For small values of pm (pm << 1), we can write:
  - $1\text{-}O(s).p_m$

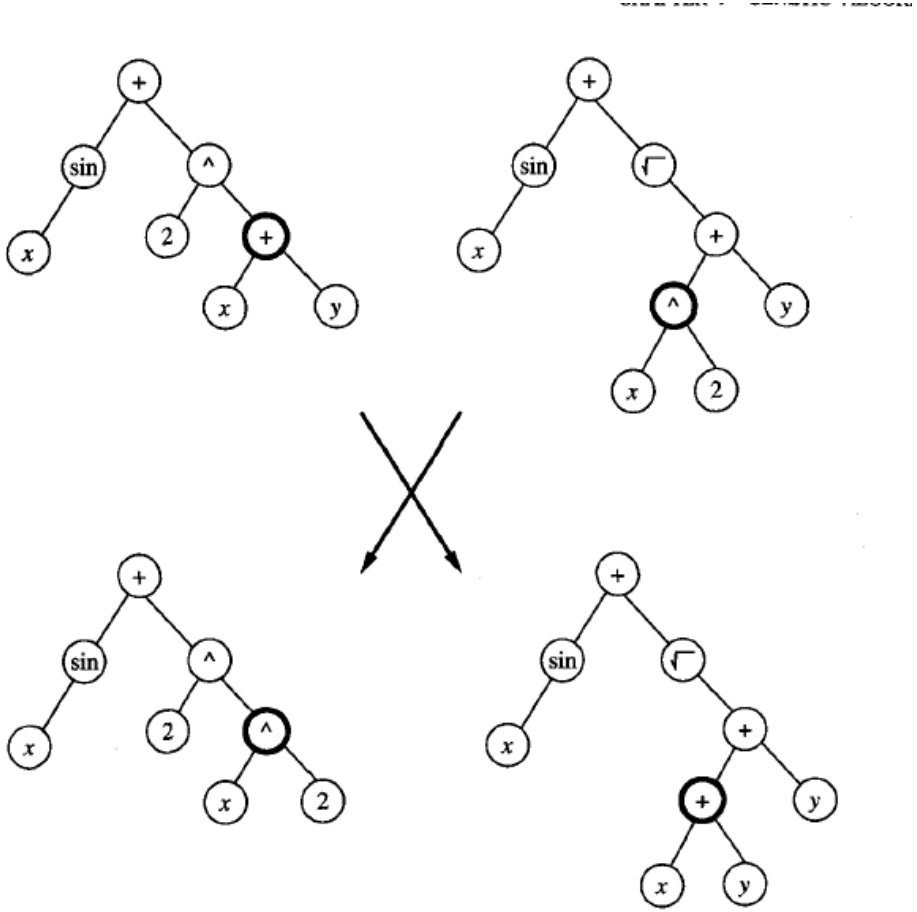
Genetic Programming

- Difference between the genetic algorithm and genetic programming is the representation of the solution.
  - Genetic Programming creates computer programs in the lisp or scheme computer languages as the solution.
  - Genetic algorithms create a string numbers that represent the solution.
- Genetic programming (GP) is a form of evolutionary computation in which the individuals in the evolving population are computer programs rather than bit strings.
- Koza (1992) describes the basic genetic programming approach and presents a broad range of simple programs that can be successfully learned by GP

- Programs manipulated by a GP are typically represented by trees corresponding to the parse tree of the program.
- Each function call is represented by a node in the tree, and the arguments to the function are given by its descendant nodes.
- For example, this tree representation for the function sin(x) + $\sqrt{x^2 + y}$.
- To apply genetic programming to a particular domain, the user must define the primitive functions to be considered (e.g., sin, cos, $\sqrt{}$,+, -, exponentials), as well as the terminals (e.g., x, y, constants such as 2).
- The genetic programming algorithm then uses an evolutionary search to explore the vast space of programs that can be described using these primitives.


+
sin
√
x
+
^
y
x
2

- The prototypical genetic programming algorithm maintains a population of individuals (in this case, program trees).
- On each iteration, it produces a new generation of individuals using selection, crossover, and mutation.
- The fitness of a given individual program in the population is typically determined by executing the program on a set of training data.
- Crossover operations are performed by replacing a randomly chosen subtree of one parent program by a subtree from the other parent program



Crossover operation applied to two parent program trees (top). Crossover points (nodes shown in bold at top) are chosen at random. The subtrees rooted at these crossover points are then exchanged to create children trees (bottom).

- Koza (1992) describes a set of experiments applying a GP to a number of
- applications.
- In his experiments, 10% of the current population, selected probabilistically according to fitness, is retained unchanged in the next generation.
- The remainder of the new generation is created by applying crossover to pairs of programs from the current generation, again selected probabilistically according to their fitness.
- The mutation operator was not used in this particular set of experiments
- Some of the applications of GP are curve fitting, data modelling, feature selection,symbolic regression, classification etc.,

MODELS OF EVOLUTION AND LEARNING

- In many natural systems, individual organisms learn to adapt significantly during their lifetime.
- At the same time, biological and social processes allow their species to adapt over a time frame of many generations.
- One interesting question regarding evolutionary systems is

  "What is the relationship between learning during the lifetime of a single individual, and the longer time frame species-level learning afforded by evolution?'

- Lamarckian Evolution
  - Lamarck was a scientist who proposed that evolution over many generations was directly influenced by the experiences of individual organisms during their lifetime.
  - He proposed that experiences of a single organism directly affected the genetic makeup of their offspring: If an individual learned during its lifetime to avoid some toxic food, it could pass this trait on genetically to its offspring, which therefore would not need to learn the trait.
  - This is an attractive conjecture, because it would presumably allow for more efficient evolutionary progress than a generate-and-test process (like that of GAS and GPs) that ignores the experience gained during an individual's lifetime.

- Baldwin Effect
  - Although Lamarckian evolution is not an accepted model of biological evolution, other mechanisms have been suggested by which individual learning can alter the course of evolution.
  - One such mechanism is called the Baldwin effect, after J. M. Baldwin (1896), who first suggested the idea.

- The Baldwin effect is based on the following observations:
  - If a species is evolving in a changing environment, there will be evolutionary pressure to favor individuals with the capability to learn during their lifetime. For example, if a new predator appears in the environment, then individuals capable of learning to avoid the predator will be more successful than individuals who cannot learn. In effect, the ability to learn allows an individual to perform a small local search during its lifetime to maximize its fitness. In contrast, nonlearning individuals whose fitness is fully determined by their genetic makeup will operate at a relative disadvantage.

- Those individuals who are able to learn many traits will rely less strongly on their genetic code to "hard-wire" traits. As a result, these individuals can support a more diverse gene pool, relying on individual learning to overcome the "missing" or "not quite optimized" traits in the genetic code. This more diverse gene pool can, in turn, support more rapid evolutionary adaption.

- Thus, the Baldwin effect provides an indirect mechanism for individual learning to positively impact the rate of evolutionary progress. By increasing survivability and genetic diversity of the species, individual learning supports more rapid evolutionary progress, thereby increasing the chance that the species will evolve genetic, nonlearned traits that better fit the new environment.

Learning set of rules

- One of the most expressive and human readable representations for learned hypotheses is sets of if-then rules.
  - special case involves learning sets of rules containing variables, called first-order Horn clauses
- In many cases it is useful to learn the target function represented as a set of if-then rules that jointly define the function.
- Rules can be derived from other representations (e.g., decision trees) or they can be learned directly.
  - one way to learn sets of rules is to first learn a decision tree, then translate the tree into an equivalent set of rules-one rule for each leaf node in the tree.
  - A second method, is to use a genetic algorithm that encodes each rule set as a bit string and uses genetic search operators to explore this hypothesis space

- Given this LEARN-ONE-RULE sub-routine for learning a single rule, one obvious approach to learning a set of rules is to invoke LEARN-ONE-RULE on all the available training examples, remove any positive examples covered by the rule it learns, then invoke it again to learn a second rule based on the remaining training examples.
- This procedure can be iterated as many times as desired to learn a disjunctive set of rules that together cover any desired fraction of the positive examples.
- This is called a sequential covering algorithm because it sequentially learns a set of rules that together cover the full set of positive examples.
- The final set of rules can then be sorted so that more accurate rules will be considered first when a new instance must be classified.

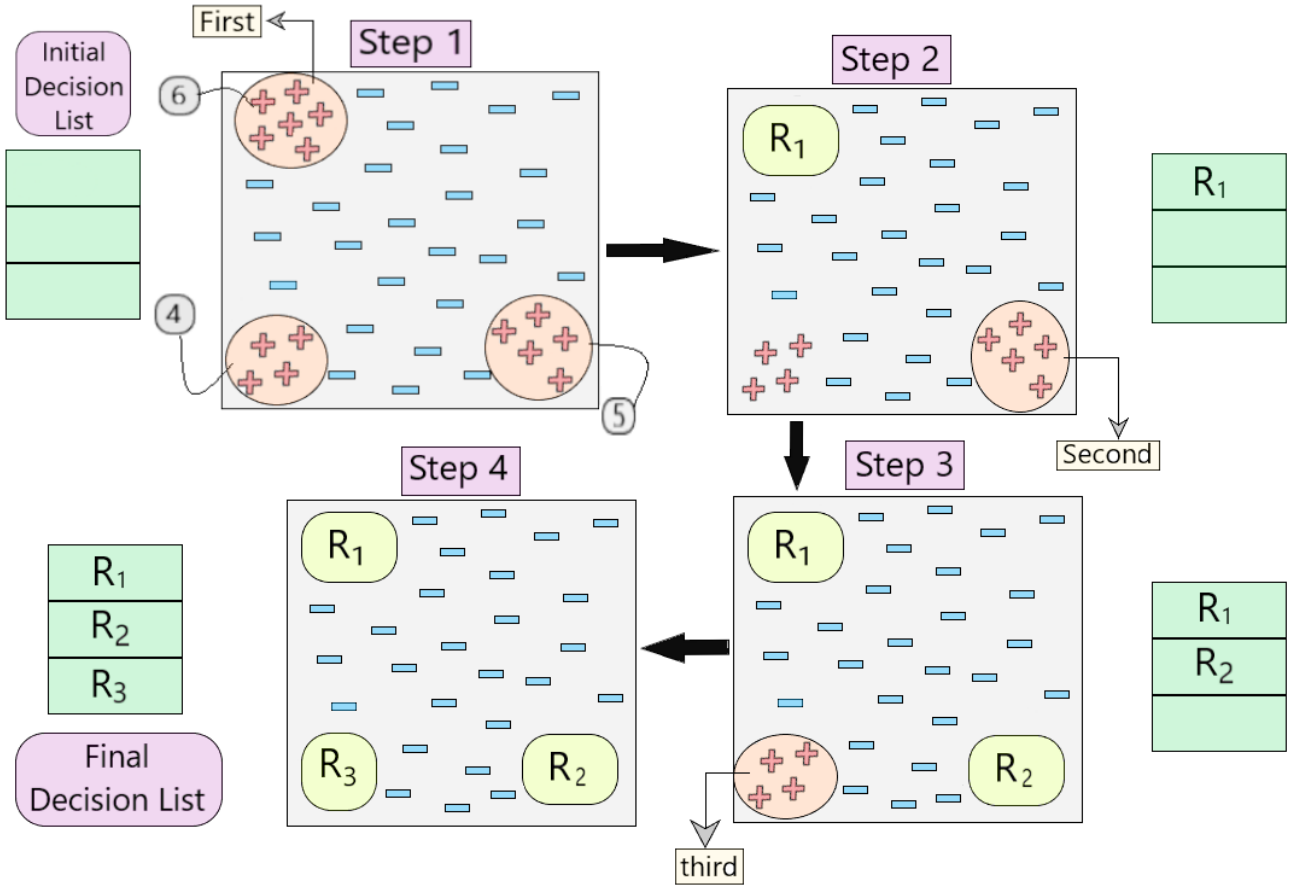
- An important aspect of direct rule-learning algorithms is that they can learn sets of first-order rules which have much more representational power than the propositional rules that can be derived from decision trees. Learning first-order rules can also be seen as automatically inferring PROLOG programs from examples.

Sequential Covering Algorithms

- Sequential covering algorithms are algorithms for learning rule sets based on the strategy of learning one rule, removing the data it covers, then iterating this process.
- To elaborate, imagine we have a subroutine LEARN-ONE-RULE that accepts a set of positive and negative training examples as input, then outputs a single rule that covers many of the positive examples and few of the negative examples. We require that this output rule have high accuracy, but not necessarily high coverage.
  - By high accuracy, we mean the predictions it makes should be correct.
  - By accepting low coverage, we mean it need not make predictions for every training example

- Propositional Logic does not include variables and thus cannot express general relations among the values of the attributes.
- Example 1: in Propositional logic, you can write:
  - IF (Father=Bob) ^ (Name=Bob)^ (Female=True) THEN Daughter=True.
- This rule applies only to a specific family!
- Example 2: In First-Order logic, you can write:
  - IF Father(y,x) ^ Female(y), THEN Daughter(x,y)
- This rule (which you cannot write in Propositional Logic) applies to any family!

- Both approaches to learning are useful as they address different types of learning problems.
- Like Decision Trees, Feedforward Neural Nets , Propositional Rule Learning systems are suited for problems in which no substantial relationship between the values of the different attributes needs to be represented.
- In First-Order Learning Problems, the hypotheses that must be represented involve relational assertions that can be conveniently expressed using first-order representations such as horn clauses (H <- L1 ^…^Ln).


Initial Decision List
First
Step 1
6
4
5
Step 2
R_1
R_1
Second
Step 3
R_1
R_2
third
Step 4
R_1
R_3
R_2
R_1
R_2
R_3
Final Decision List

Prototypical Sequential Covering Algorithm

Sequential-Covering(Target_attribute, Attributes, Examples, Threshold)

- Learned_rules <-- { }
- Rule <-- Learn-one-rule(Target_attribute, Attributes, Examples)
- While Performance(Rule, Examples) > Threshold, do
  - Learned_rules <-- Learned_rules + Rule
  - Examples <-- Examples -{examples correctly classified by Rule}
  - Rule <-- Learn-one-rule(Target_attribute, Attributes, Examples)
- Learned_rules <-- sort Learned_rules according to Performance over Examples
- Return Learned_rules

Learning sets of First-Order rules: Foil

- A variety of algorithms has been proposed for learning first-order rules, or Horn clauses
- FOIL (Quinlan, 1990) is the natural extension of SEQUENTIAL-COVERING and LEARN-ONE-RULE to first order rule learning.
- FOIL learns first order rules which are similar to Horn clauses with two exceptions:
  - literals may not contain function symbols (reduces complexity of hypothesis space)
  - literals in body of clause may be negated (hence, more expressive than Horn clauses)

- Like SEQUENTIAL-COVERING, FOIL learns one rule at time and removes positive examples covered by the learned rule before attempting to learn a further rule.
- The two most substantial differences between FOIL and SEQUENTIAL-COVERING and LEARN-ONE-RULE algorithm are :
  - In its general-to-specific search to 'learn each new rule, FOIL employs different detailed steps to generate candidate specializations of the rule. This difference follows from the need to accommodate variables in the rule preconditions.
  - FOIL employs a PERFORMANCE measure, Foil-Gain, that differs from the entropy measure used for LEARN-ONE-RULE.

FOIL(*Target_predicate*, *Predicates*, *Examples*)

- *Pos* ← those *Examples* for which the *Target_predicate* is *True*
- *Neg* ← those *Examples* for which the *Target_predicate* is *False*
- *Learned_rules* ← {}
- while *Pos*, do

  Learn a NewRule

  - *NewRule* ← the rule that predicts *Target_predicate* with no preconditions
  - *NewRuleNeg* ← *Neg*
  - while *NewRuleNeg*, do

    Add a new literal to specialize NewRule

    - *Candidate_literals* ← generate candidate new literals for *NewRule*, based o *Predicates*
    - *Best_literal* ← $\underset{L \in Candidate\_literals}{\text{argmax}}$ *Foil_Gain*(*L*, *NewRule*)
    - add *Best_literal* to preconditions of *NewRule*
    - *NewRuleNeg* ← subset of *NewRuleNeg* that satisfies *NewRule* preconditions
  - *Learned_rules* ← *Learned_rules* + *NewRule*
  - *Pos* ← *Pos* − {members of *Pos* covered by *NewRule*}
- Return *Learned_rules*

- FOIL searches its hypothesis space via two nested loops:
  - The outer loop corresponds to a variant of the SEQUENTIAL-COVERlNG algorithm.
  - The outer loop at each iteration adds a new rule to an overall disjunctive hypothesis Learned_rules(i.e. rule1 Vrule2 V...)
  - The search is a specific-to-general search through the space of hypotheses,
    - starting with the most specific empty disjunctive hypothesis which covers no positive instances
    - stopping when the hypothesis is sufficiently general enough to cover all positive examples

- The inner loop of FOIL performs a finer-grained search to determine the exact definition of each new rule.
  - This inner loop searches a second hypothesis space, consisting of conjunctions of literals, to find a conjunction that will form the preconditions for the new rule
  - This loop may be viewed as a general-to-specific search
    - starting with the most general precondition (empty)
    - stopping when the hypothesis is specific enough to exclude all negative examples