Unit-IV

Genetic Algorithms, Learning Sets of Rules, Reinforcement Learning

- Genetic Algorithms
  - Motivation
  - Genetic Algorithms
  - An Illustrative Example
  - Hypothesis Space Search
  - Genetic Programming
  - Models of Evolution and Learning
  - Parallelizing Genetic Algorithms
- Learning Sets of Rules
  - Introduction
  - Sequential Covering Algorithms
  - Learning Rules Sets: Summary
  - Learning First-Order Rules
  - Learning Sets of First Order Rules: FOIL
  - Induction as Inverted Deduction
  - Inverting Resolution
- Reinforcement Learning
  - Introduction
  - The Learning Task
  - Q Learning
  - Non Deterministic Rewards and Actions
  - Temporal Difference Learning
  - Generalizing from Examples
  - Relationship to Dynamic Programming

Genetic Algorithms-Motivation

- Genetic algorithms (GAs) provide a learning method motivated by an analogy to biological evolution.
- GAs 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 of the population by offspring of the most fit current hypotheses.
- The process forms a generate-and-test beam-search of hypotheses, in which variants of the best current hypotheses are most likely to be considered next.

Genetic Algorithms-Motivation

- 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

- Representing Hypotheses
- Genetic Operators
- Fitness Function and Selection

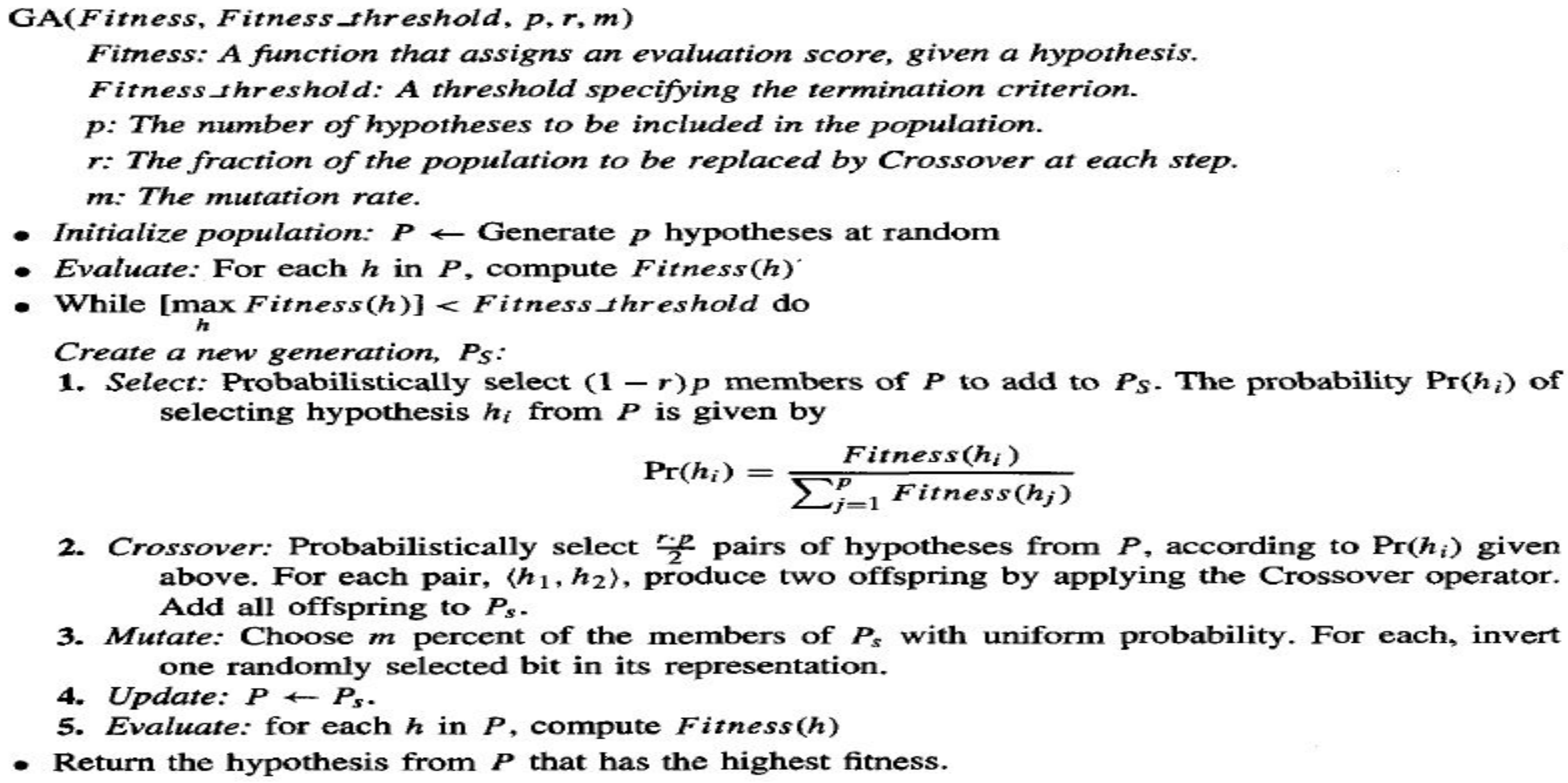
Genetic Algorithms

- 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.***
- 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.

Genetic Algorithms

- The Genetic Algorithms share the following structure:
    - The algorithm operates by iteratively updating a pool of hypotheses, called the population.
    - In 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.

Genetic Algorithms



```
GA($Fitness$, $Fitness\_threshold$, $p$, $r$, $m$)
        Fitness: A function that assigns an evaluation score, given a hypothesis.
        Fitness_threshold: A threshold specifying the termination criterion.
        p: The number of hypotheses to be included in the population.
        r: The fraction of the population to be replaced by Crossover at each step.
        m: The mutation rate.
• Initialize population: $P \leftarrow$ Generate $p$ hypotheses at random
• Evaluate: For each $h$ in $P$, compute $Fitness(h)$
• While $[\max_h Fitness(h)] < Fitness\_threshold$ do
    Create a new generation, $P_S$:
    1. Select: Probabilistically select $(1-r)p$ members of $P$ to add to $P_S$. The probability $\Pr(h_i)$ of
            selecting hypothesis $h_i$ from $P$ is given by
                $\Pr(h_i) = \frac{Fitness(h_i)}{\sum_{j=1}^{p} Fitness(h_j)}$
    2. Crossover: Probabilistically select $\frac{r \cdot p}{2}$ pairs of hypotheses from $P$, according to $\Pr(h_i)$ given
            above. For each pair, $\langle h_1, h_2 \rangle$, produce two offspring by applying the Crossover operator.
            Add all offspring to $P_s$.
    3. Mutate: Choose $m$ percent of the members of $P_s$ with uniform probability. For each, invert
            one randomly selected bit in its representation.
    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.
```

Representing Hypotheses

- Hypotheses in GAS are often represented by bit strings, so that they can be easily manipulated by genetic operators such as mutation and crossover.
- Let us look at an example:

$$(Outlook = Overcast \vee Rain) \wedge (Wind = Strong)$$

Outlook	*Wind*
011	10

IF *Wind* = *Strong* THEN *PlayTennis* = *yes*

would be represented by the string

Outlook	*Wind*	*PlayTennis*
111	10	10

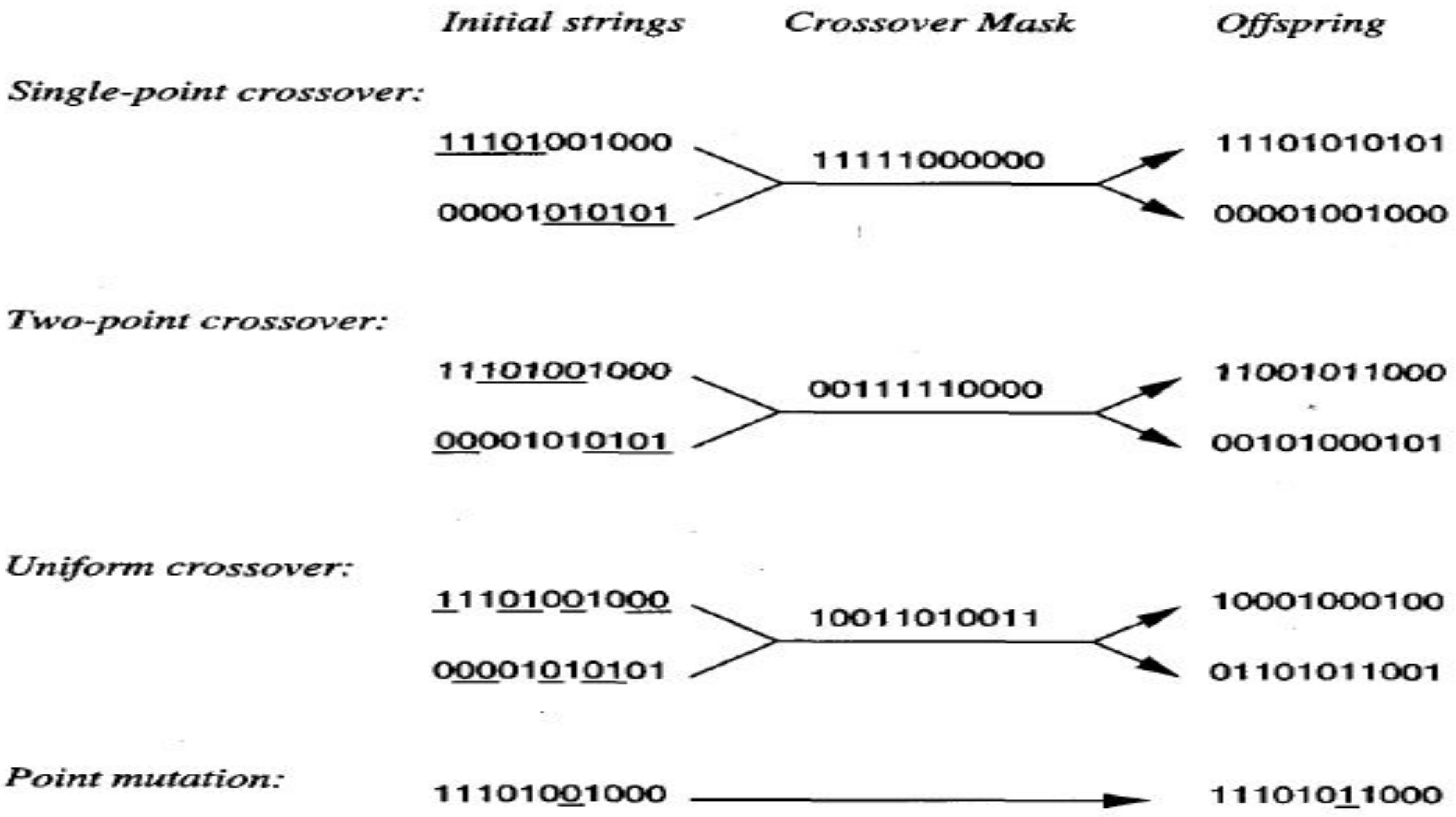
Representing Hypotheses

- While designing a bit string encoding for some hypotheses space, it is useful to make sure that every syntactically legal bit string represents a well defined hypothesis.
- According to the given rules in the previous example the bit string 111 10 11 does not represent a legal hypothesis.(11 is does not represent any target value).
- It will be good if we use one bit for playtennis in the previous example.
- We must take care that every bit string must be syntactically legal.

Genetic Operators

- Two most common operators used in Genetic Algorithms are:
  - crossover
  - mutation
- Crossover operator: It produces two new offspring from two parents by copying selected bits from each parent.
- The choice of which parent contributes the bit for position i is determined by an additional string called the crossover mask.
- Mutation operator: This operator produces small random changes to the bit string by choosing a single bit at random then changing its value.

Genetic Operators



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.
- If the task is to learn classification rules, then the fitness function typically has a component that scores the classification accuracy of the rule over a set of provided training examples.
- When the bit string hypothesis is interpreted as a complex procedure the fitness function may measure the overall performance of the resulting procedure rather than performance of individual rules.
- The method used in the previous algorithm is sometimes called ***fitness proportionate selection,*** or roulette wheel selection.

Fitness Function and Selection

- There are other methods like:
  - Tournament selection
  - Rank selection
- In **tournament selection** two hypotheses are first selected at random from the current population. With a probability of p the more fit of these two is selected and with a probability of (1-p) the less fit hypothesis is selected.
- In **rank selection,** the 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.

An Illustrative Example

- A genetic algorithm can be viewed as a general optimization method that searches a large space of candidate objects seeking one that performs best according to the fitness function.
- GAs succeed in finding an object with high fitness.
- In machine learning, Gas have been applied both to function-approximation problems and to tasks such as choosing the network topology for artificial neural network learning systems.
- Let us look at an example system GABIL for concept learning described by Dejong.

An Illustrative Example

- The algorithm used is same as the one described earlier.
- The parameter r is set to .6 and m is set to .001. The population p varies from 100 to 1000.
- **Representation:**
  - Let us consider a hypothesis space in which rule preconditions are conjunctions of constraints over two Boolean attributes, ***a1*** and *a2.*
  - The rule post-condition is described by a single bit that indicates **the** predicted value of the target attribute c.
  - The hypothesis consisting of the two rules:
    - IF a1=T∧a2=F then c=T ; IF a2=T then c=F
    - The representation of the above in bit string is:

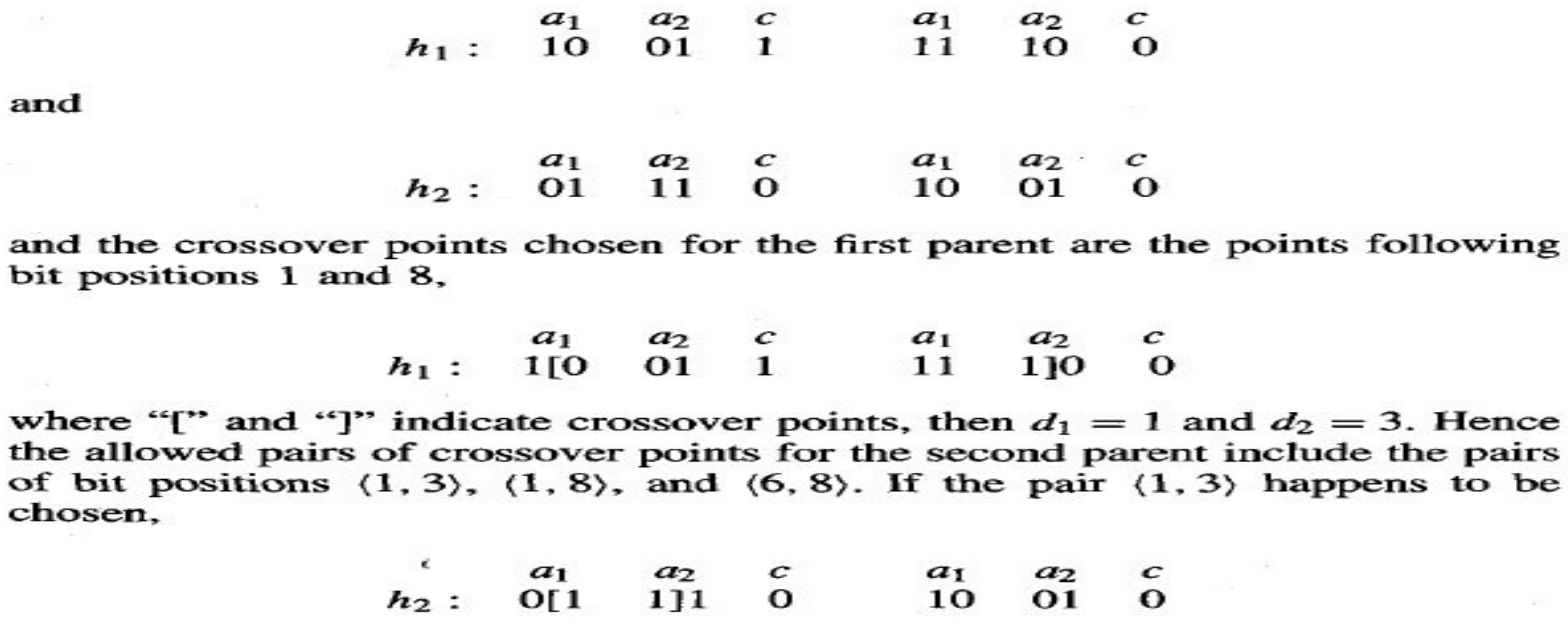
a_1	a_2	c	a_1	a_2	c
10	01	1	11	10	0

An Illustrative Example

- **Genetic Operators:**
  - The mutation operator chooses a random bit and replaces it by its compliment.
  - The crossover operator is an extension to the two point crossover operator described earlier.
  - To perform a crossover operation on two parents, two crossover points are first chosen at random in the first parent string.
  - Let d1 (d2) denote the distance from the leftmost (rightmost) of these two crossover points to the rule boundary immediately to its left.
  - The crossover points in the second parent are now randomly chosen, subject to the constraint that they must have the same d1 and ***d2*** value.

An Illustrative Example

- For example, if the two parent strings are:



	a_1	a_2	c	a_1	a_2	c
h_1 :	10	01	1	11	10	0

and

	a_1	a_2	c	a_1	a_2	c
h_2 :	01	11	0	10	01	0

and the crossover points chosen for the first parent are the points following bit positions 1 and 8,

	a_1	a_2	c	a_1	a_2	c
h_1 :	1[0	01	1	11	1]0	0

where "[" and "]" indicate crossover points, then $d_1 = 1$ and $d_2 = 3$. Hence the allowed pairs of crossover points for the second parent include the pairs of bit positions $\langle 1, 3\rangle$, $\langle 1, 8\rangle$, and $\langle 6, 8\rangle$. If the pair $\langle 1, 3\rangle$ happens to be chosen,

	a_1	a_2	c	a_1	a_2	c
h_2 :	0[1	1]1	0	10	01	0

An Illustrative Example

- The resulting two offspring will be:

	a_1	a_2	c
h_3 :	11	10	0

and

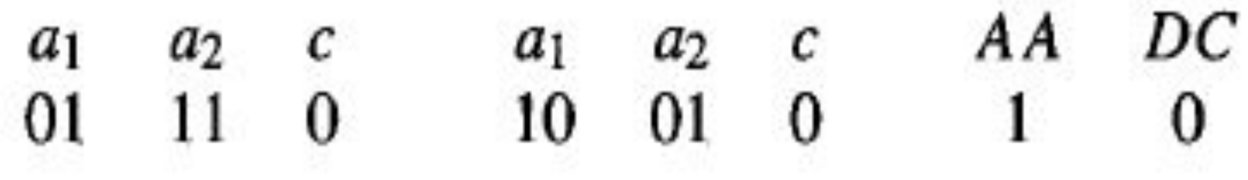
	a_1	a_2	c	a_1	a_2	c	a_1	a_2	c
h_4 :	00	01	1	11	11	0	10	01	0

- **Fitness Function:**
  - The fitness of each hypothesized rule set is based on its classification accuracy over the training data. In particular, the function used to measure fitness is:

$$Fitness(h) = (correct(h))^2$$

Extensions

- There are two other operators that can be used:
- **AddAlternative:**
  - This is a constraint on a specific attribute which changes a 0 to 1.
- **DropCondition:**
  - It replaces all bits of a particular attribute by 1.



a_1	a_2	c	a_1	a_2	c	AA	DC
01	11	0	10	01	0	1	0

Hypothesis Space Search

- GAs employ a randomized beam search method to seek a maximally fit hypothesis.
- Let us compare hypothesis space search of Gas with the gradient descent of backpropogation to understand the difference in the learning algorithms.
- Gradient descent moves smoothly from one hypothesis to a new hypothesis that is very similar.
- GA search moves abruptly replacing a parent hypothesis by an offspring that may be radically different from the parent.
- GA search is less likely to fall into the same kind of local minima as gradient descent.

Hypothesis Space Search

- 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.
- The negative impact of crowding is that it reduces the diversity of the population, thereby slowing further progress by the GA.
- The following strategies can be used to overcome crowding:
  - Alter the selection function, using criteria such as tournament selection or rank selection.
  - Another strategy is "fitness sharing," in which the measured fitness of an individual is reduced by the presence of other, similar individuals in the population.
  - A third approach is to restrict the kinds of individuals allowed to recombine to form offspring.

Population Evolution and the Schema Theorem

- An interesting question is whether we can mathematically characterize the evolution over time of the population within a GA.
- The schema theorem which is based on the concept of schemas that describe sets of bit strings is one such characterization.
- A schema is any string composed of 0s,1s and *s, where * is interpreted as don’t care.
- The schema theorem characterizes the evolution of the population within a GA in terms of the number of instances representing each schema.

Population Evolution and the Schema Theorem

- Let m(s,t) denote the number of instances of schema s in the population at time t.
- 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 crossover step, and the mutation step.
- Let us start by considering just the effect of the selection step.

Population Evolution and the Schema Theorem

- Let f (h) denote the fitness of the individual bit string h.
- $\bar{f}(t)$ denote the average fitness of all individuals in the population at time t.
- Let n be the total number of individuals in the population.
- Let h ϵ s∩p_t indicate that the individual h is both a representative of schema s and a member of the population at time t.
- Let $\hat{u}(s,t)$denote the average fitness of instances of schema ***s*** in the population at time t.

Population Evolution and the Schema Theorem

- We are interested in calculating the expected value of m(s, t + l), which we denote E[m(s, t + I)].
- The probability distribution for selection as discussed already is:

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

- 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:

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

Population Evolution and the Schema Theorem

- As we know:

$$\hat{u}(s,t) = \frac{\sum_{h \in s \cap p_t} f(h)}{m(s,t)}$$

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

- The above equation gives the probability that a single hypothesis selected by the GA will be an instance of schema s.
- 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.

$$E[m(s,t+1)] = \frac{\hat{u}(s,t)}{\bar{f}(t)} m(s,t)$$

Population Evolution and the Schema Theorem

- While the above analysis considered only the selection step of the *GA,* the crossover and mutation steps must be considered as well.
- The full schema theorem thus provides a lower bound on the expected frequency of schema *s,* as follows:

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

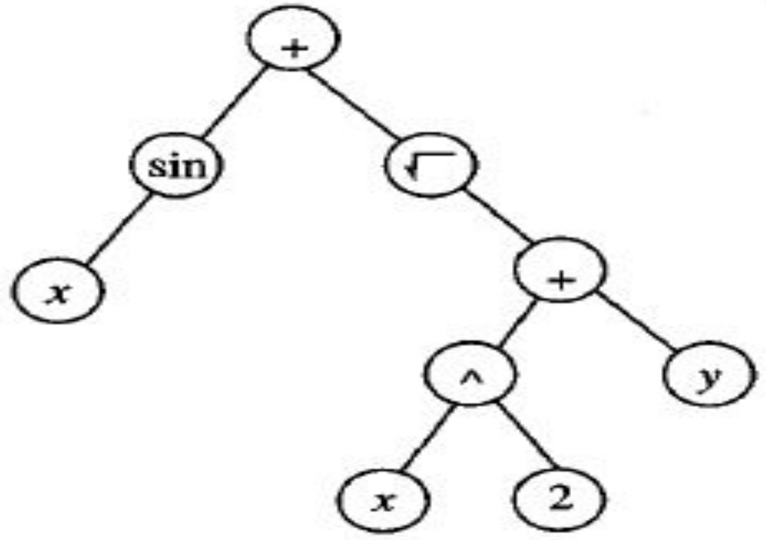
- p_c is the probability that the single-point crossover operator will be applied to an arbitrary individual.
- p_m is the probability that an arbitrary bit of an arbitrary individual will be mutated by the mutation operator.
- *o(s)* is the number I of *defined bits* in schema *s, d(s)* is the distance between the leftmost and rightmost defined bits in *s and l* is the length of the individual bit strings in the population.

Genetic Programming

- Representing Programs
- Illustrative Example
- Remarks on Genetic Programming

Representing Programs

- 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.

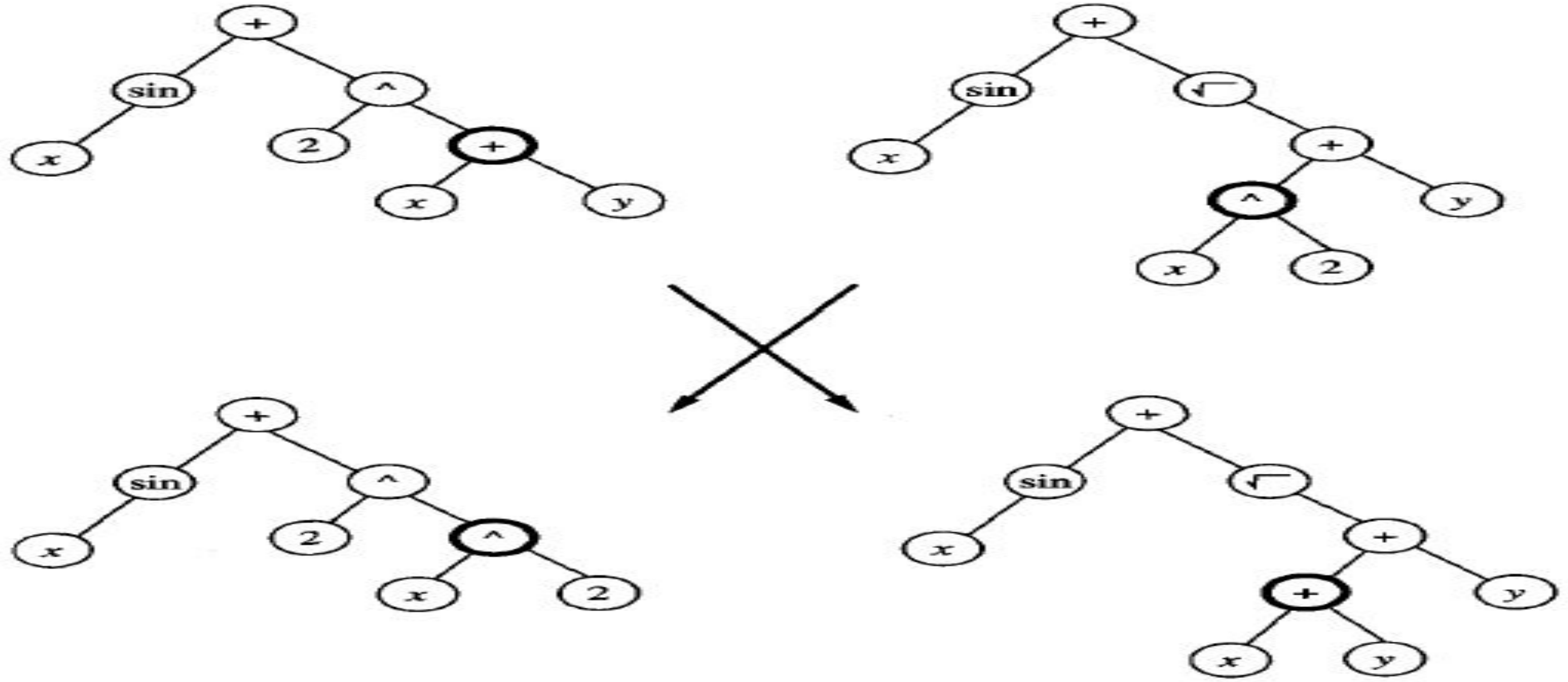


Representing Programs

- The prototypical genetic programming algorithm maintains a population of individuals.
- 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.

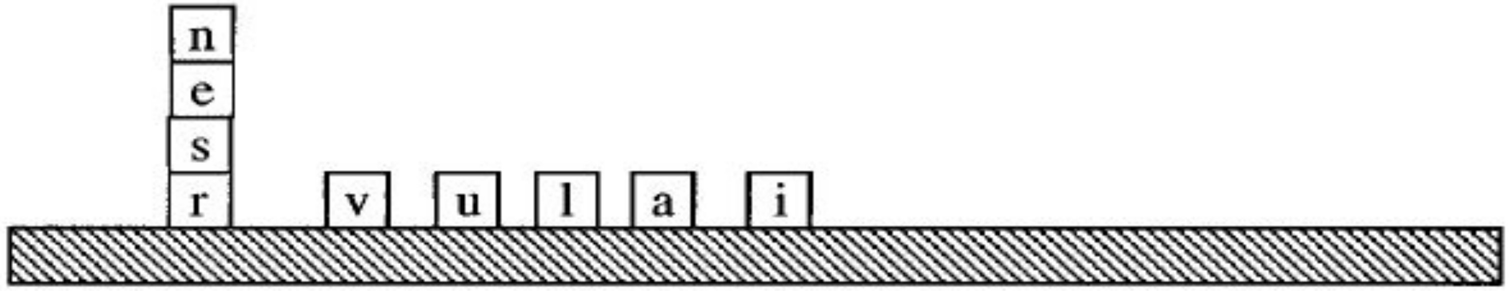
Representing Programs

- Crossover operations are performed by replacing a randomly chosen subtree of one parent program by a subtree from the other parent program.



Illustrative Example-Koza

- The task is to develop a general algorithm for stacking the blocks into a single stack that spells the word "universal,” independent of the initial configuration of blocks in the world.



- The actions available for manipulating blocks allow moving only a single block at a time.

Illustrative Example-Koza

- Here the primitive functions used to compose programs for this task include the following three terminal arguments:
    - CS (current stack), which refers to the name of the top block on the stack, or F if there is no current stack.
    - TB (top correct block), which refers to the name of the topmost block on the stack, such that it and those blocks beneath it are in the correct order.
    - NN (next necessary), which refers to the name of the next block needed above TB in the stack, in order to spell the word "universal," or F if no more blocks are needed.

Illustrative Example-Koza

- In addition to these terminal arguments, the program language in this application included the following primitive functions:
  - (MS x) (move to stack), if block x is on the table, this operator moves x to the top of the stack and returns the value T. Otherwise, it does nothing and returns the value F.
  - (MT x) (move to table), if block ***x*** is somewhere in the stack, this moves the block at the top of the stack to the table and returns the value T. Otherwise, it returns the value F.
  - (EQ x y) (equal), which returns T if x equals ***y,*** and returns F otherwise.
  - (NOT x), which returns T if x = F, and returns F if x = T.
  - (DU x y) (do until), which executes the expression ***x*** repeatedly until expression y returns the value T.

Illustrative Example-Koza

- The algorithm was provided a set of 166 training example problems representing a broad variety of initial block configurations, including problems of differing degrees of difficulty.
- The fitness of any given program was taken to be the number of these examples solved by the algorithm.
- The population was initialized to a set of 300 random programs.
- After 10 generations, the system discovered the following program, which solves all 166 problems.

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

Remarks on Genetic Programming

- Genetic programming extends genetic algorithms to the evolution of complete computer programs.
- Despite the huge size of the hypothesis space it must search, genetic programming has been demonstrated to produce good results in a number of applications.
- In most cases, the performance of genetic programming depends crucially on the choice of representation and on the choice of fitness function.
- For this reason, **an** active area of current research is aimed at the automatic discovery and incorporation of subroutines that improve on the original set of primitive functions.
- This allows the system to dynamically alter the primitives from which it constructs individuals

Models of Evolution and Learning

- A very important question 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?”
- Here we discuss two types of evolutions:
    - Lamarckian Evolution
    - Baldwin Effect

Lamarckian Evolution

- Lamarck 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 would allow for more efficient evolutionary progress than a generate-and-test process. The current scientific evidence overwhelmingly contradicts Lamarck's model.
- Recent computer studies have shown that Lamarckian processes can sometimes improve the effectiveness of computerized genetic algorithms

Baldwin Effect

- 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 who first suggested the idea.

Baldwin Effect

- The Baldwin effect is based on the following observations. The first observation is:
  - 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, non-learning individuals whose fitness is fully determined by their genetic makeup will operate at a relative disadvantage.

Baldwin Effect

- The second observation is:
  - Those individuals who are able to learn many traits will rely less strongly on their genetic code to "hard-wire" traits.
  - Those who are able to learn can support a more diverse gene pool, relying on individual learning to overcome the "missing" or "not quite optimized" traits in the genetic code.
  - A more diverse gene pool can, in turn, support more rapid evolutionary adaptation.
  - The ability of individuals to learn can have an indirect accelerating effect on the rate of evolutionary adaptation for the entire population.

Baldwin Effect

- 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.
- Survivability increases the chance that the species will evolve genetic, non-learned traits that better fit the new environment.

Parallelizing Genetic Algorithms

- GAs are naturally suited to parallel implementation, and a number of approaches to parallelization have been explored.
- Here we shall discuss about two approaches:
  - Coarse grain
  - Fine grain

Coarse Grain

- Coarse grain approaches to parallelization subdivide the population into somewhat distinct groups of individuals, called demes.
- Each deme is assigned to a different computational node, and a standard GA search is performed at each node.
- Communication and cross-fertilization between demes occurs on a less frequent basis than within demes.
- Transfer between demes occurs by a migration process, in which individuals from one deme are copied or transferred to other demes.
- One benefit of such approaches is that it reduces the crowding problem often encountered in nonparallel GAs.
- Examples of coarse-grained parallel GAS are described by Tanese (1989) and by Cohoon (1987).

Fine Grain

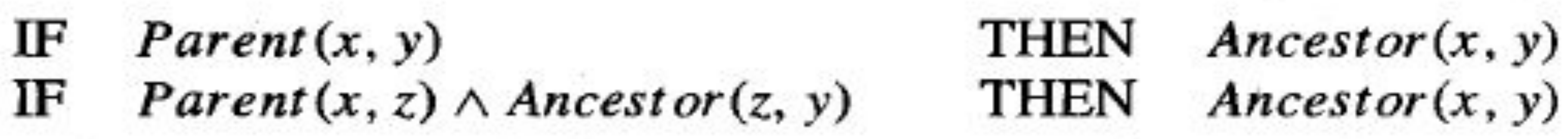
- Fine-grained implementations assign one processor per individual in the population. Recombination then takes place among neighboring individuals.
- Several different types of neighborhoods have been proposed, ranging from planar grid to torus.
- Examples of such systems are described by Spiessens and Manderick (1991).

Learning Sets of Rules

- One of the most expressive and human readable representations for learned hypotheses is sets of if-then rules.
- We already discussed about a few ways in which sets of rules can be learnt:
  - First learn a decision tree, then translate the tree into an equivalent set of rules, one rule for each node in the tree.
  - Use a genetic algorithm that encodes each rule set as a bit string and uses genetic search operators to explore this hypothesis space.

Learning Sets of Rules

- Here we explore a variety of algorithms that directly learn rule sets and differ from the algorithms we already learnt in two key aspects:
  - They are designed to learn sets of first-order rules that contain variables.
  - They use sequential covering algorithms that learn one rule at a time to incrementally grow the final set of rules.
- Let us look at an example to describe the target concept Ancestor.



IF $Parent(x, y)$ THEN $Ancestor(x, y)$
IF $Parent(x, z) \wedge Ancestor(z, y)$ THEN $Ancestor(x, y)$

- Here we discuss about learning algorithms capable of learning such rules given appropriate sets of training examples.

Learning Sets of Rules

- Here we start with algorithms to learn disjunctive sets of rules.
- We later extend these algorithms to learn first-order rules.
- We then discuss general approaches to inductive logic programming.
- Later we explore the relation between inductive and deductive inference.

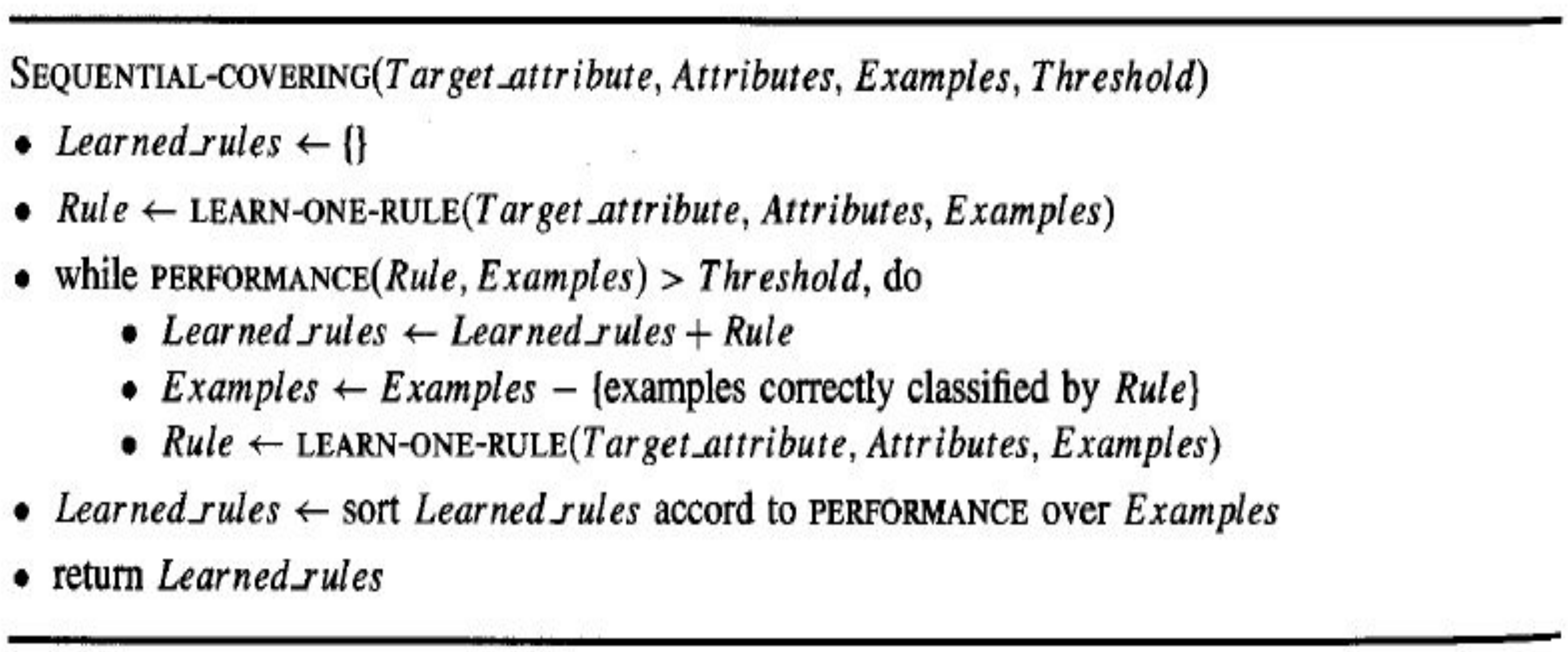
Sequential Covering Algorithms

- General to Specific Beam Search
- Variations

Sequential Covering Algorithms

- A family of algorithms for learning rule sets based on the strategy of learning one rule, removing the data it covers, then iterating this process are called sequential covering algorithms.
- The sequential covering algorithm reduces the problem of learning a disjunctive set of rules to a sequence of simpler problems, each requiring that a single conjunctive rule be learned.
- It performs a greedy search, formulating a sequence of rules without backtracking.
- It is not guaranteed to find the smallest or best set of rules that cover the training examples.

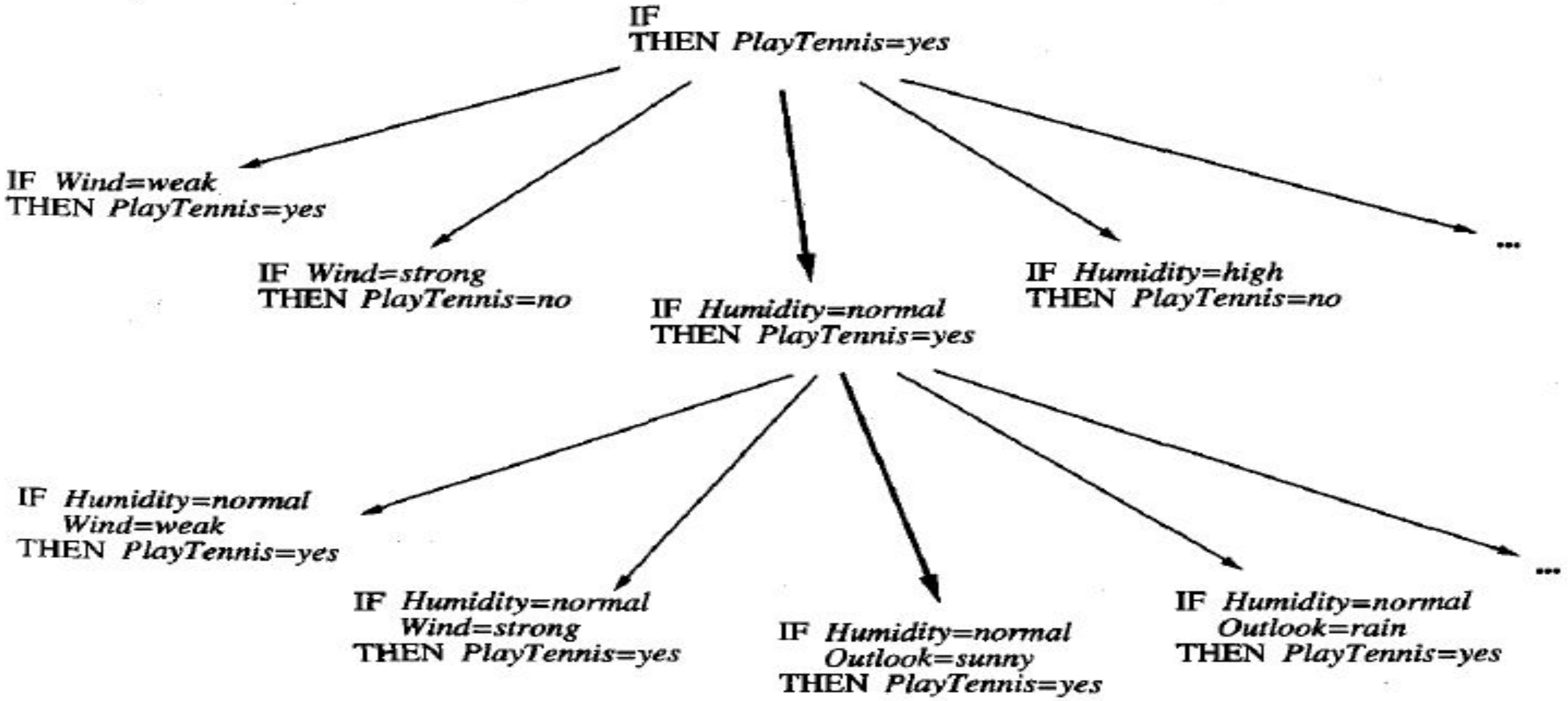
Sequential Covering Algorithms



```
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 accord to PERFORMANCE over Examples
• return Learned_rules
```

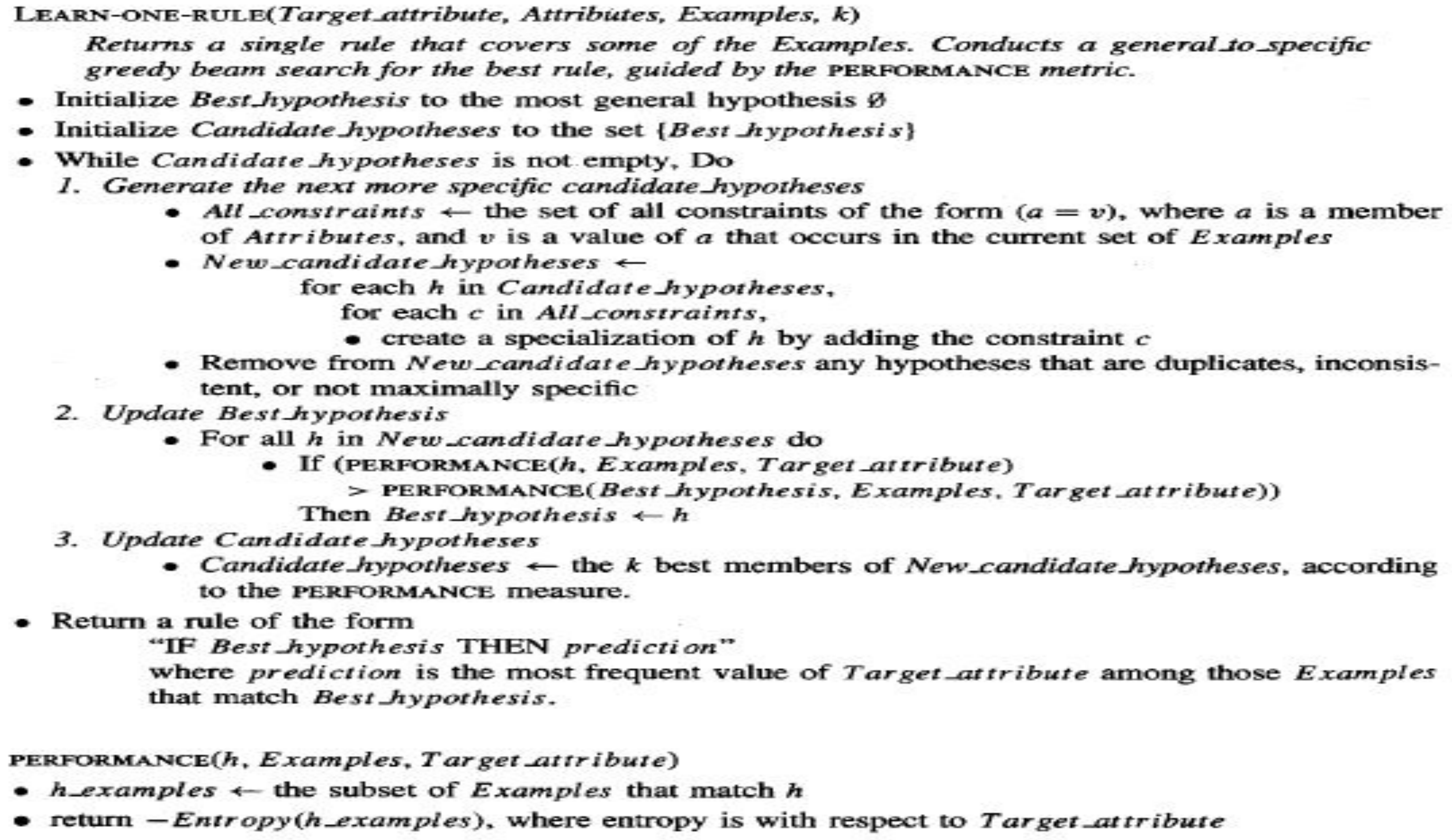
General to Specific Beam Search

One effective approach to implementing Learn-One-Rule to organize the hypothesis space search in the same general fashion as the ID3 algorithm, but to follow only the most promising branch in the tree at each step.



IF THEN PlayTennis=yes
IF Wind=weak THEN PlayTennis=yes
IF Wind=strong THEN PlayTennis=no
IF Humidity=normal THEN PlayTennis=yes
IF Humidity=high THEN PlayTennis=no
...
IF Humidity=normal Wind=weak THEN PlayTennis=yes
IF Humidity=normal Wind=strong THEN PlayTennis=yes
IF Humidity=normal Outlook=sunny THEN PlayTennis=yes
IF Humidity=normal Outlook=rain THEN PlayTennis=yes
...

General to Specific Beam Search



```
LEARN-ONE-RULE(Target_attribute, Attributes, Examples, k)
    Returns a single rule that covers some of the Examples. Conducts a general_to_specific
    greedy beam search for the best rule, guided by the PERFORMANCE metric.
• Initialize Best_hypothesis to the most general hypothesis ∅
• Initialize Candidate_hypotheses to the set {Best_hypothesis}
• While Candidate_hypotheses is not empty, Do
    1. Generate the next more specific candidate_hypotheses
        • All_constraints ← the set of all constraints of the form $(a = v)$, where $a$ is a member
          of Attributes, and $v$ is a value of $a$ that occurs in the current set of Examples
        • New_candidate_hypotheses ←
                for each h in Candidate_hypotheses,
                    for each c in All_constraints,
                        • create a specialization of h by adding the constraint c
        • Remove from New_candidate_hypotheses any hypotheses that are duplicates, inconsis-
          tent, or not maximally specific
    2. Update Best_hypothesis
        • For all h in New_candidate_hypotheses do
            • If (PERFORMANCE(h, Examples, Target_attribute)
                    > PERFORMANCE(Best_hypothesis, Examples, Target_attribute))
              Then Best_hypothesis ← h
    3. Update Candidate_hypotheses
        • Candidate_hypotheses ← the k best members of New_candidate_hypotheses, according
          to the PERFORMANCE measure.
• Return a rule of the form
        "IF Best_hypothesis THEN prediction"
        where prediction is the most frequent value of Target_attribute among those Examples
        that match Best_hypothesis.

PERFORMANCE(h, Examples, Target_attribute)
• h_examples ← the subset of Examples that match h
• return −Entropy(h_examples), where entropy is with respect to Target_attribute
```

General to Specific Beam Search

- Let us analyze the Learn-One-Rule algorithm:
    - Each hypothesis considered in the main loop of the algorithm is a conjunction of attribute-value constraints.
    - Each of these conjunctive hypotheses corresponds to a candidate set of preconditions for the rule to be learned and is evaluated by the entropy of the examples it covers.
    - The search considers increasingly specific candidate hypotheses until it reaches a maximally specific hypothesis that contains all available attributes.
    - The rule that is output by the algorithm is the rule encountered during the search whose PERFORMANCE is greatest, not necessarily the final hypothesis generated in the search.
    - The algorithm constructs the rule post-condition to predict the value of the target attribute that is most common among the examples covered by the rule precondition.

Variations

- The Sequential-Covering algorithm, together with the Learn-One-Rule algorithm, learns a set of if-then rules that covers the training examples.
- One variation on this approach is to have the program learn only rules that cover positive examples and to include a default that assigns a negative classification to instances not covered by any rule.
- In order to learn such rules that predict just a single target value, the Learn-One-Rule algorithm can be modified to accept an additional input argument specifying the target value of interest.

Variations

- Another variation is provided by a family of algorithms called AQ.
- AQ seeks rules that cover a particular target value learning a disjunctive set of rules for each target value in turn.
- AQ algorithm while conducting a general to specific beam search for each rule uses a single positive example to focus this search.
- It considers only those attributes satisfied by the positive example as it searches for progressively more specific hypotheses.

Learning Rule Sets: Summary

- *Sequential covering(CN2)* algorithms learn one rule at a time, removing the covered examples and repeating the process on the remaining examples.
- Decision tree algorithms such as ID3 learn the entire set of disjuncts simultaneously as part of the single search for an acceptable decision tree. These type of algorithms are called as simultaneous covering algorithms.
- At each search step ID3 chooses among alternative *attributes* by comparing the *partitions* of the data they generate.
- CN2 chooses among alternative *attribute-value* pairs, by comparing the *subsets* of data they cover.

Learning Rule Sets: Summary

- To learn a set of n rules, each containing k attribute-value tests in their preconditions sequential covering algorithms will perform n*k primitive search steps.
- Simultaneous covering algorithms will make many fewer independent choices.
- Sequential covering algorithms such as CN2 make a larger number of independent choices than simultaneous covering algorithms such as ID3.
- The answer to the question “which should we prefer?” depends on the size of the data.
- If the data is larger prefer sequential covering.

Learning Rule Sets: Summary

- A second dimension along which approaches vary is the direction of search in the Learn-One-Rule, whether it is general to specific or specific to general.
- A third dimension is whether the Learn-One-Rule search is generate and test or example-driven.
- A fourth dimension is whether and how rules are post-pruned.

Learning Rule Sets: Summary

- A final dimension is the particular definition of rule performance used to guide the search in Learn-One-Rule. Various evaluation functions have been used:
- **Relative Frequency:** Let ***n*** denote the number of examples the rule matches and let $\boldsymbol{n_c}$ denote the number of these that it classifies correctly. The relative frequency estimate of rule performance is: n_c/n
- **m-estimate of accuracy:** Let p be the prior probability that a randomly drawn example from the entire data set will have the classification assigned by the rule. Let m be the weight then the m-estimate of rule accuracy is:

$$\mathbf{n_c+mp/n+m}$$

- **Entropy:**

$$-Entropy(S) = \sum_{i=1}^{c} p_i \log_2 p_i$$

Learning First-Order Rules

- Here we consider learning rules that contain variables in particular, learning first-order Horn theories.
- Inductive learning of first-order rules or theories is referred to as ***inductive logic programming***, because this process can be viewed as automatically inferring **PROLOG** programs from examples.
- **PROLOG** is a general purpose, Turing-equivalent programming language in which programs are expressed as collections of Horn clauses.

First-Order Horn Clauses

- Let us look at an example:

$$\begin{array}{lll} \langle Name_1 = Sharon, & Mother_1 = Louise, & Father_1 = Bob, \\ Male_1 = False, & Female_1 = True, & \\ Name_2 = Bob, & Mother_2 = Nora, & Father_2 = Victor, \\ Male_2 = True, & Female_2 = False, & Daughter_{1,2} = True \rangle \end{array}$$

- This is a training example for the target concept Daughter$_{1,2}$.
- If this type of examples are provided to a propositional rule learner like CN2 or ID3 the result would be:

$$\begin{array}{ll} \text{IF} & (Father_1 = Bob) \wedge (Name_2 = Bob) \wedge (Female_1 = True) \\ \text{THEN} & Daughter_{1,2} = True \end{array}$$

- The problem with propositional representation is that they do not offer any general way to describe the essential relations among the values of the attributes.

First-Order Horn Clauses

- A program using first-order representation could learn the following general rule:

$$\text{IF} \quad Father(y,x) \wedge Female(y), \quad \text{THEN} \quad Daughter(x,y)$$

where x and y are variables that can be bound to any person.

- First-order Horn clauses may also refer to variables in the precondition that do not occur in the post-conditions.

$$\begin{array}{ll} \text{IF} & Father(y,z) \wedge Mother(z,x) \wedge Female(y) \\ \text{THEN} & GrandDaughter(x,y) \end{array}$$

- When variables occur only in the precondition then they are assumed to be existentially qualified.

Terminology

- In first order logic all expressions are composed of **constants** (Ravi, Raja), **variables** (x,y), **predicate symbols** (Married, Greater-Than), and **functions** (age).
- The difference between predicates and functions is that predicates take on values True or False where s function take on any values.

Terminology

- Every well-formed expression is composed of *constants* (e.g., $Mary$, 23, or Joe), *variables* (e.g., x), *predicates* (e.g., $Female$, as in $Female(Mary)$), and *functions* (e.g., age, as in $age(Mary)$).
- A *term* is any constant, any variable, or any function applied to any term. Examples include $Mary$, x, $age(Mary)$, $age(x)$.
- A *literal* is any predicate (or its negation) applied to any set of terms. Examples include $Female(Mary)$, $\neg Female(x)$, $Greater\_than(age(Mary), 20)$.
- A *ground literal* is a literal that does not contain any variables (e.g., $\neg Female(Joe)$).
- A *negative literal* is a literal containing a negated predicate (e.g., $\neg Female(Joe)$).
- A *positive literal* is a literal with no negation sign (e.g., $Female(Mary)$).
- A *clause* is any disjunction of literals $M_1 \vee \ldots M_n$ whose variables are universally quantified.
- A *Horn clause* is an expression of the form

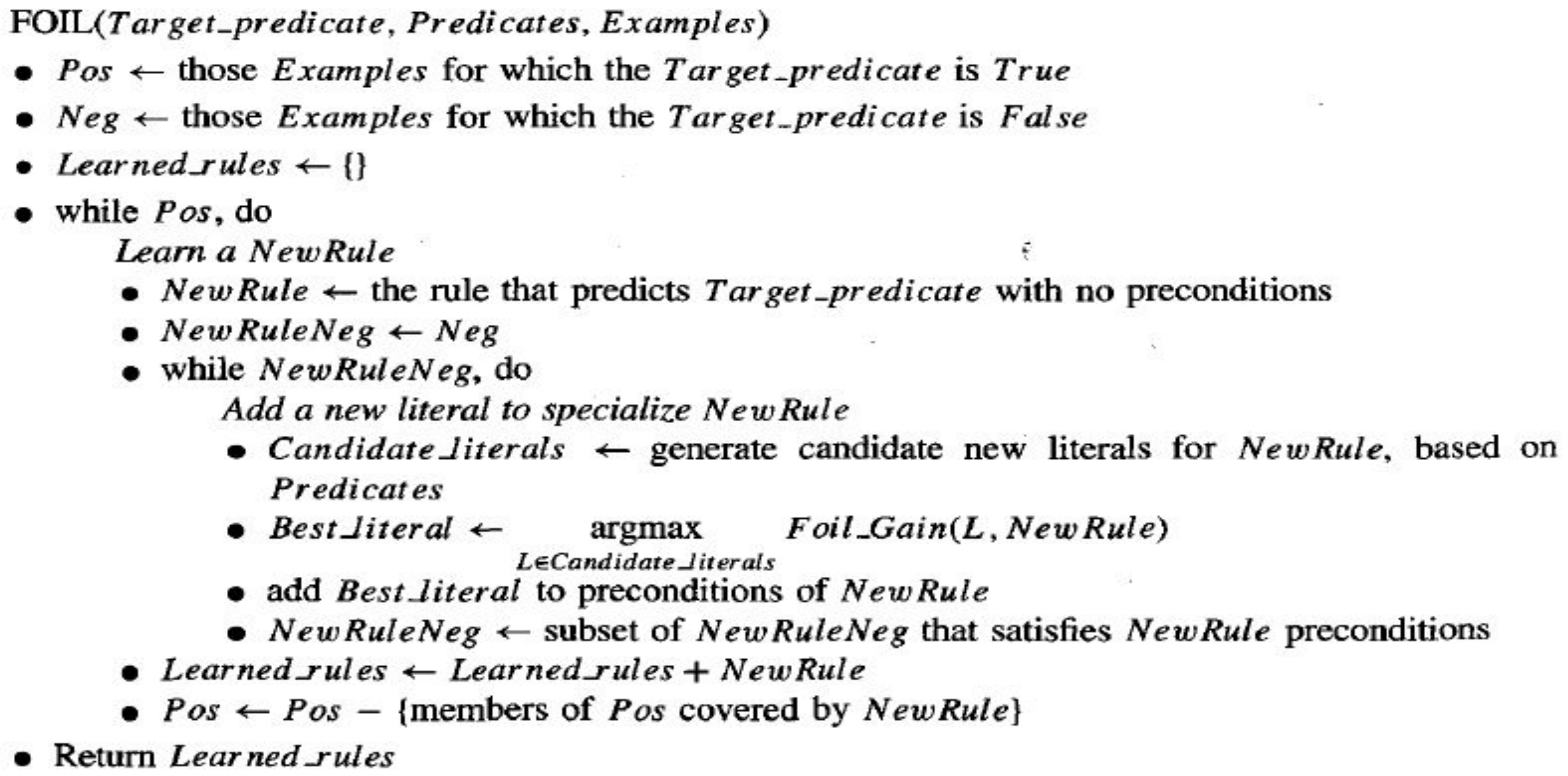
  $$H \leftarrow (L_1 \wedge \ldots \wedge L_n)$$

  where $H, L_1 \ldots L_n$ are positive literals. H is called the *head* or *consequent* of the Horn clause. The conjunction of literals $L_1 \wedge L_2 \wedge \ldots \wedge L_n$ is called the *body* or *antecedents* of the Horn clause.
- For any literals A and B, the expression $(A \leftarrow B)$ is equivalent to $(A \vee \neg B)$, and the expression $\neg(A \wedge B)$ is equivalent to $(\neg A \vee \neg B)$. Therefore, a Horn clause can equivalently be written as the disjunction

  $$H \vee \neg L_1 \vee \ldots \vee \neg L_n$$

- A *substitution* is any function that replaces variables by terms. For example, the substitution $\{x/3, y/z\}$ replaces the variable x by the term 3 and replaces the variable y by the term z. Given a substitution θ and a literal L we write $L\theta$ to denote the result of applying substitution θ to L.
- A *unifying substitution* for two literals L_1 and L_2 is any substitution θ such that $L_1\theta = L_2\theta$.

Learning Sets of First Order Rules: FOIL



```
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 on
          Predicates
        • Best_literal ← $\underset{L \in Candidate\_literals}{\operatorname{argmax}}$ 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
```

Learning Sets of First Order Rules: FOIL

- The most important differences between FOIL and our earlier sequential covering and learn-one-rule are as follows:
  - 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 shown for LEARN-ONE-RULE. This difference follows from the need to distinguish between different bindings of the rule variables and from the fact that FOIL seeks only rules that cover positive examples.

Generating Candidate Specializations in FOIL

- Let us assume the current rule being considered is:

$$P(x_1,x_2,\ldots\ldots x_k) \leftarrow L_1,L_2\ldots\ldots L_n$$

Where L1,L2…….Ln are literals forming the current rule preconditions and where P(x1,x2…..xk) is the literal that forms the rule head, or post conditions.

Generating Candidate Specializations in FOIL

- FOIL generates candidate specializations of this rule by considering new literals L_{n+1} that fit one of the following forms:
  - ***$Q(v_1, \ldots, v_n)$***, where ***Q*** is any predicate name occurring in Predicates and where the ***v_i*** are either new variables or variables already present in the rule. At least one of the ***v_i*** in the created literal must already exist as a variable in the rule.
  - ***$Equal(x_j, x_k)$***, where ***x_j*** and ***x_k*** are variables already present in the rule.
  - The negation of either of the above forms of literals.

Generating Candidate Specializations in FOIL

- Let us look at an example to predict the target literal GrandDaughter(x,y) where the other predicates used to describe examples are Father and Female.
- The general-to-specific search in FOIL begins with the most general rule:

  GrandDaughter(x,y) □

- *Equal (x , y) , Female(x), Female(y), Father(x, y), Father(y, x), Father(x, z), Father(z, x), Father(y, z), Father(z, y),* and the negations of each of these literals (e.g., *-Equal(x, y)) are generated literals as candidate additions to the rule precondition.*

Generating Candidate Specializations in FOIL

- Now suppose that among the above literals FOIL greedily selects ***Father(y , z)*** as the most promising, leading to the more specific rule

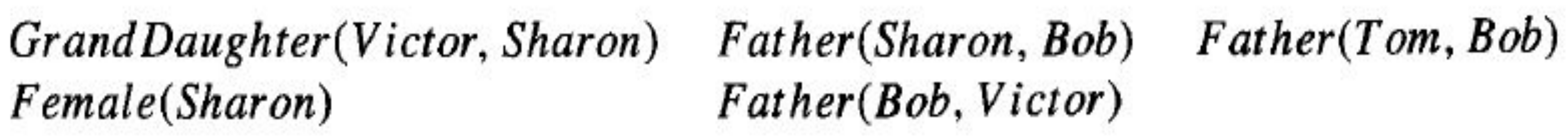
  GrandDaughter(x, y)□ Father(y, z)

- A new variable z is added here which is to be considered while selecting the next literal.
- The final rule will be:

  GrandDaughter(x, y)□ Father(y, z)∧Father(z,x)∧Female(y)

Guiding the Search in FOIL

- To select the most promising literal from the candidates generated at each step, FOIL considers the performance of the rule over the training data.
- Let us look at the same example of GrandDaughter(x,y). Here P(x,y) can be read as “The P of x is y”.



GrandDaughter(Victor, Sharon) *Father(Sharon, Bob)* *Father(Tom, Bob)*
Female(Sharon) *Father(Bob, Victor)*

- Those not listed above are assumed to be false(!GrandDaughter(Tom,Bob))

Guiding the Search in FOIL

- Let us look at the initial step when the rule is:

  GrandDaughter(x,y) □

- The rule variables ***x*** and ***y*** are not constrained by any preconditions and may therefore bind in any combination to the four constants ***Victor, Sharon, Bob,*** and ***Tom.***
- There are 16 possible variable bindings for this initial rule.
- The binding ***{xlvictor, ylSharon}*** corresponds to ***a*** positive example binding, because the training data includes the assertion ***GrandDaughter(Victor, Sharon).***

Guiding the Search in FOIL

- If a literal is added that introduces a new variable, then the bindings for the rule will grow in length (e.g., if ***Father(y, z)*** is added to the above rule, then the original binding ***{xlvictor, y/Sharon)*** will become the more lengthy ***{xlvictor, ylSharon, z/Bob}.***
- The evaluation function used by FOIL to estimate the utility of adding a new literal is based on the numbers of positive and negative bindings covered before and after adding the new literal.

Guiding the Search in FOIL

- Let R' be the rule created by adding literal L to rule ***R.*** The value ***Foil_Gain(L, R)*** of adding ***L*** to R is defined as:

$$Foil\_Gain(L, R) \equiv t \left(\log_2 \frac{p_1}{p_1 + n_1} - \log_2 \frac{p_0}{p_0 + n_0} \right)$$

- P0 is the number of positive bindings to rule R
- n0 is the number of negative bindings to rule R
- P1 is the number of positive bindings to rule R’
- n1 is the number of negative bindings to rule R’
- ***t*** is the number of positive bindings of rule R that are still covered after adding literal ***L*** to ***R.***

Learning Recursive Rule Sets

- Recursive rules are rules that use the same predicate in the body and the head of the rule.
- Example:

$$\begin{array}{lll} \text{IF} & Parent(x, y) & \text{THEN} \quad Ancestor(x, y) \\ \text{IF} & Parent(x, z) \wedge Ancestor(z, y) & \text{THEN} \quad Ancestor(x, y) \end{array}$$

- The second rule is among the rules that are potentially within reach of FOIL'S search, provided ***Ancestor*** is included in the list ***Predicates*** that determines which predicates may be considered when generating new literals.
- Whether this particular rule would be learned or not depends on whether these particular literals outscore competing candidates during FOIL'S greedy search for increasingly specific rules.

Summary of FOIL

- To learn each rule FOIL performs a general-to-specific search, at each step adding a single new literal to the rule preconditions.
- The new literal may refer to variables already mentioned in the rule preconditions or post-conditions, and may introduce new variables as well.
- At each step, it uses the ***Foil_Gain*** function to select among the candidate new literals.
- In the case of noise-free training data, FOIL may continue adding new literals to the rule until it covers no negative examples.
- FOIL post-prunes each rule it learns, using the same rule post-pruning strategy used for decision trees

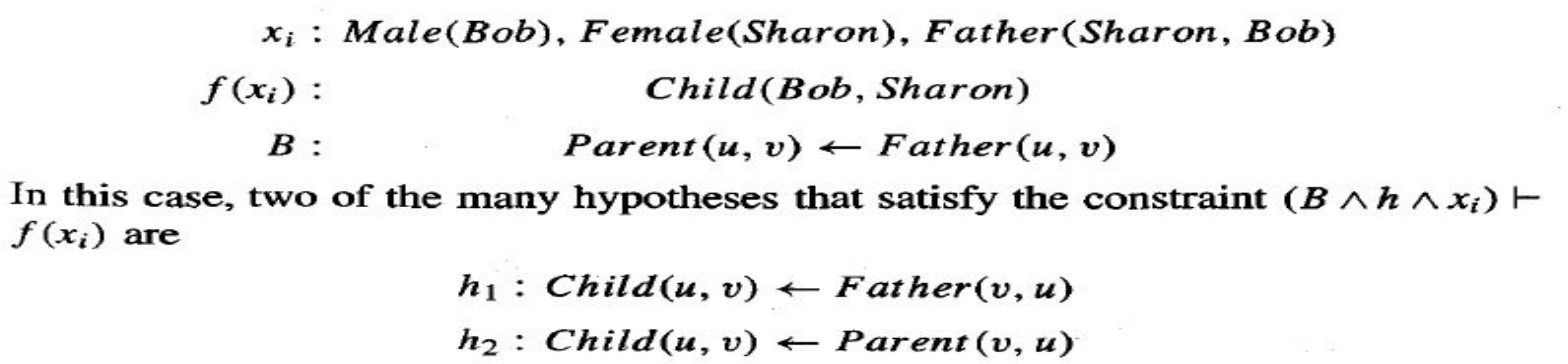
Induction as Inverted Deduction

- Another approach to inductive logic programming is based on the simple observation that induction is just the inverse of deduction.
- Learning can be described as:

$$(\forall \langle x_i, f(x_i) \rangle \in D)\ (B \wedge h \wedge x_i) \vdash f(x_i)$$

- Where B is the background knowledge and X⊦Y can be read as “Y follows deductively from X”.

Induction as Inverted Deduction



$$x_i : Male(Bob), Female(Sharon), Father(Sharon, Bob)$$

$$f(x_i) : \quad Child(Bob, Sharon)$$

$$B : \quad Parent(u, v) \leftarrow Father(u, v)$$

In this case, two of the many hypotheses that satisfy the constraint $(B \wedge h \wedge x_i) \vdash f(x_i)$ are

$$h_1 : Child(u, v) \leftarrow Father(v, u)$$

$$h_2 : Child(u, v) \leftarrow Parent(v, u)$$

- The process of augmenting the set of predicates based on background knowledge is referred to as **constructive induction**.

Induction as Inverted Deduction

- Now we shall explore the view of induction as inverse of deduction.
- Here we will be interested in designing inverse entailment operators.
- An inverse entailment operator, ***O(B, D)*** takes the training data ***D* = *{(xi,f (xi)) }*** and background knowledge ***B*** as input and produces as output a hypothesis h satisfying the Equation:

$$O(B, D) = h \text{ such that } (\forall \langle x_i, f(x_i) \rangle \in D) \; (B \wedge h \wedge x_i) \vdash f(x_i)$$

- One common heuristic in ILP (Inductive Logic Programming) for choosing among such hypotheses is to rely on the heuristic known as the Minimum Description Length principle.

Induction as Inverted Deduction

- There are some interesting features in formulating a learning task for the above equation:
  - This formulation is different from the common definition of learning as finding some general concept that matches a given set of training examples.
  - By incorporating the notion of background information B, this formulation allows a more rich definition of when a hypothesis may be said to "fit“ the data.
  - The inverse resolution procedure described in the following section uses background knowledge to search for the appropriate hypothesis.

Induction as Inverted Deduction

- Let us look at a few practical difficulties in finding the hypothesis:
  - The requirement in the previous equation does not naturally accommodate noisy training data.
  - The language of first order logic is so expressive and the number of hypotheses that satisfy the given equation is so large that the search through the space of hypotheses is intractable in general case.
  - Despite our intuition that background knowledge B should help constrain the search for a hypothesis, in most **ILP** systems the complexity of the hypothesis space search ***increases*** as background knowledge B is increased.

Inverting Resolution- A Inverse Entailment Operator

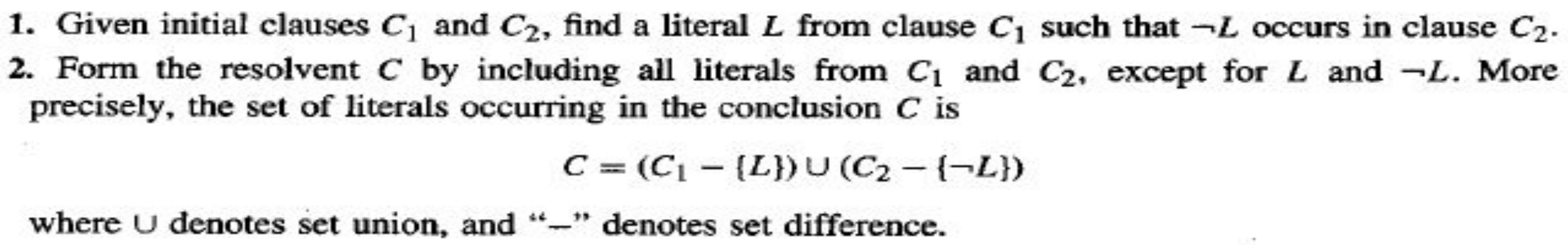
- A general method for automated deduction is the ***resolution rule*** introduced by Robinson.
- Let us try to understand resolution rule:

$$\frac{\begin{array}{ccc} P & \vee & L \\ \neg L & \vee & R \end{array}}{\begin{array}{ccc} P & \vee & R \end{array}}$$

- The resolution rule is a sound and complete rule for deductive inference in first-order logic.
- Now we look at the possibility whether we can invert the resolution rule to form an inverse entailment operator.

Inverting Resolution

- The general form of the propositional resolution operator is as follows:



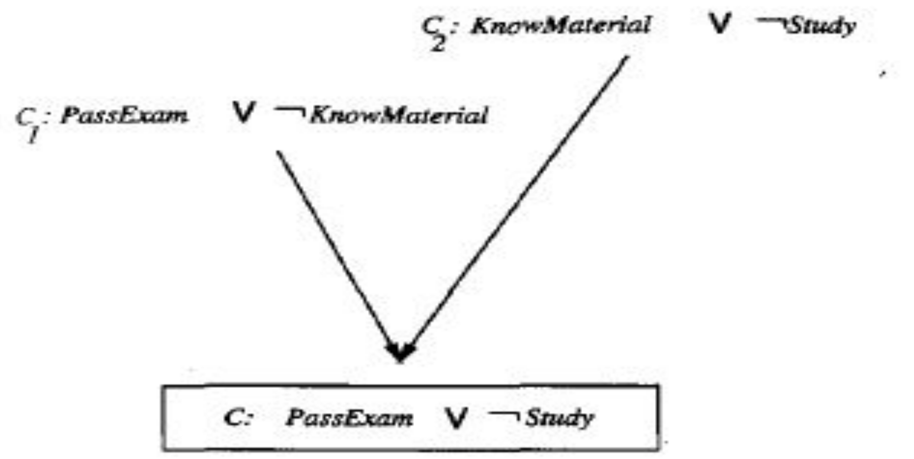
1. Given initial clauses C_1 and C_2, find a literal L from clause C_1 such that $\neg L$ occurs in clause C_2.
2. Form the resolvent C by including all literals from C_1 and C_2, except for L and $\neg L$. More precisely, the set of literals occurring in the conclusion C is

$$C = (C_1 - \{L\}) \cup (C_2 - \{\neg L\})$$

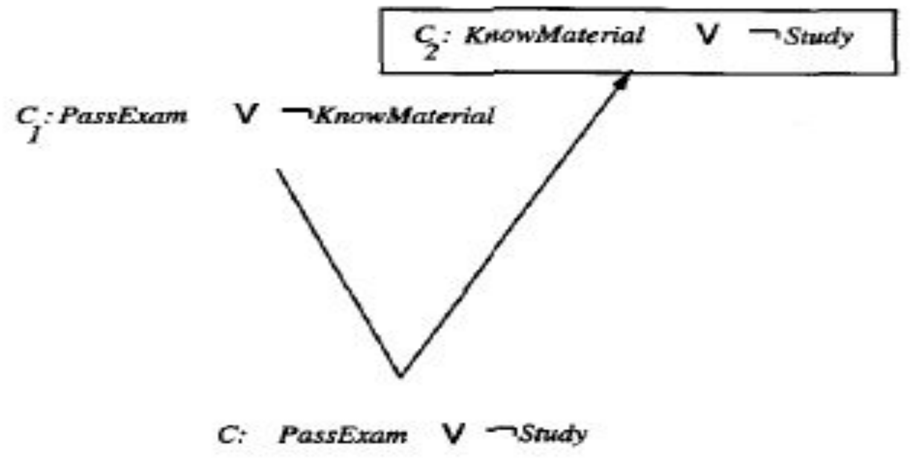
where $\cup$ denotes set union, and "$-$" denotes set difference.

- Given two clauses ***C1*** and ***C2,*** the resolution operator first identifies a literal ***L*** that occurs as a positive literal in one of these two clauses and as a negative literal in the other.
- It then draws the conclusion given by the above formula.

Inverting Resolution



C_2: KnowMaterial $\vee$ $\neg$Study
C_1: PassExam $\vee$ $\neg$KnowMaterial
C: PassExam $\vee$ $\neg$Study



C_2: KnowMaterial $\vee$ $\neg$Study
C_1: PassExam $\vee$ $\neg$KnowMaterial
C: PassExam $\vee$ $\neg$Study

1. Given initial clauses C_1 and C, find a literal L that occurs in clause C_1, but not in clause C.
2. Form the second clause C_2 by including the following literals

$$C_2 = (C - (C_1 - \{L\})) \cup \{\neg L\}$$

First Order Resolution

- The resolution rule extends easily to first-order expressions.
- First-order expressions also take two clauses as input and produces a third clause as output.
- The key difference from the propositional case is that the process is now based on the notion of ***unifying*** substitutions.
- We define a ***substitution*** to be any mapping of variables to terms.
- Example:
- Let L be the literal Father(x,Bill), let substitution θ={x/Bob,y/z} then L$\theta = Father(Bob, Bill)$.

First Order Resolution

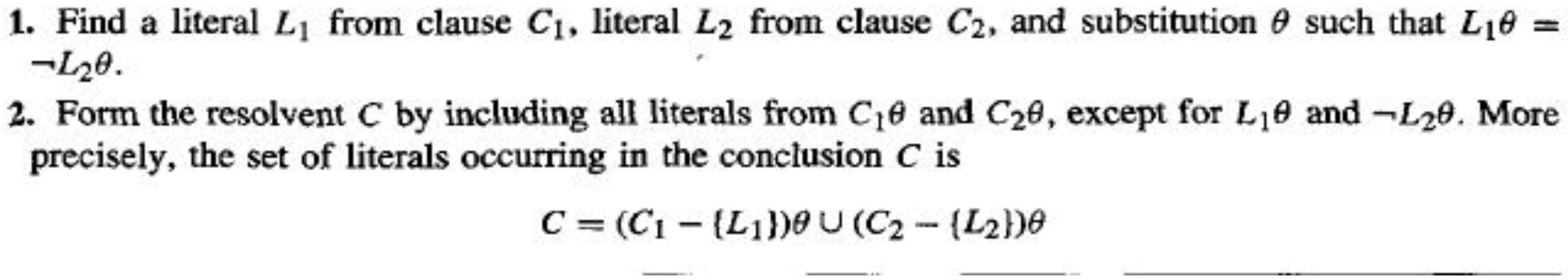
- We say that θ is a *unifying substitution* for two literals *L1* and *L2,* provided $L_l\theta$=$L_2\theta$.
- *Example:*
- L_1= *Father(x, y),* L_2= *Father(Bil1, z),* and θ*=(x/Bill, z/y},* then θ is a unifying substitution for L_1 and L_2 because $L_l\,\theta$=$L_2\theta$*=Father(Bil1, y).*
- The significance of a unifying substitution is this:
  - In the propositional form of resolution, the resolvent of two clauses ***C1*** and ***C2*** is found by identifying a literal ***L*** that appears in ***C1*** such that ***!L*** appears in ***C2.***
  - In firstorder resolution, this generalizes to finding one literal ***L1*** from clause ***C1*** and one literal ***L2*** from ***C2,*** such that some unifying substitution θ can be found for ***L1*** and ***!L2*** (i.e., such that ***LI*** θ = !***L2*** θ***).***

First Order Resolution

- The resolution rule then constructs the resolvent ***C*** according to the equation:

$$C = (C_1 - \{L_1\})\theta \cup (C_2 - \{L_2\})\theta$$

- The general statement of the resolution rule is as follows:



1. Find a literal L_1 from clause C_1, literal L_2 from clause C_2, and substitution θ such that $L_1\theta = \neg L_2\theta$.
2. Form the resolvent C by including all literals from $C_1\theta$ and $C_2\theta$, except for $L_1\theta$ and $\neg L_2\theta$. More precisely, the set of literals occurring in the conclusion C is

$$C = (C_1 - \{L_1\})\theta \cup (C_2 - \{L_2\})\theta$$

First Order Resolution

- Let us look at an example:
- suppose C1 = White(x) $\leftarrow$ Swan(x) and suppose C2 = Swan(Fred)
- Now we express C1 in an equivalent form C1=white(x) V !swan(x)
- Now we apply the resolution rule:
- We find the literal L1=!swan(x) in C1, L2=swan(Fred) in C2. Now we choose unifying substitution θ={x/Fred} then the two literal L1θ=!L2θ =!swan(Fred).
- Finally C is the union of (C1-{L1})θ=white(Fred) and (C2-{!L2})θ=Ø C=white(Fred).

Inverting Resolution: First Order Case

- We can derive the inverse resolution operator analytically, by algebraic manipulation of the previous Equation which defines the resolution rule.
- We first factor the unifying substitution θ into $\theta 1$ and $\theta 2$.
- This factorization is possible because C1 and C2 will always begin with distinct variable names.
- Now the previous equation becomes:

$$C=(C_1-\{L_1\})\theta_1 \cup (C_2-\{L_2\})\theta_2$$

The equation becomes $C-(C_1-\{L_1\})\,\theta_1=(C_2-\{L_2\})\,\theta_2$

Inverting Resolution: First Order Case

- Finally we use the fact that by definition of the resolution rule L2 =!L1 θ_1 θ_2^{-1} and solve for C2 to obtain:

Inverse resolution:

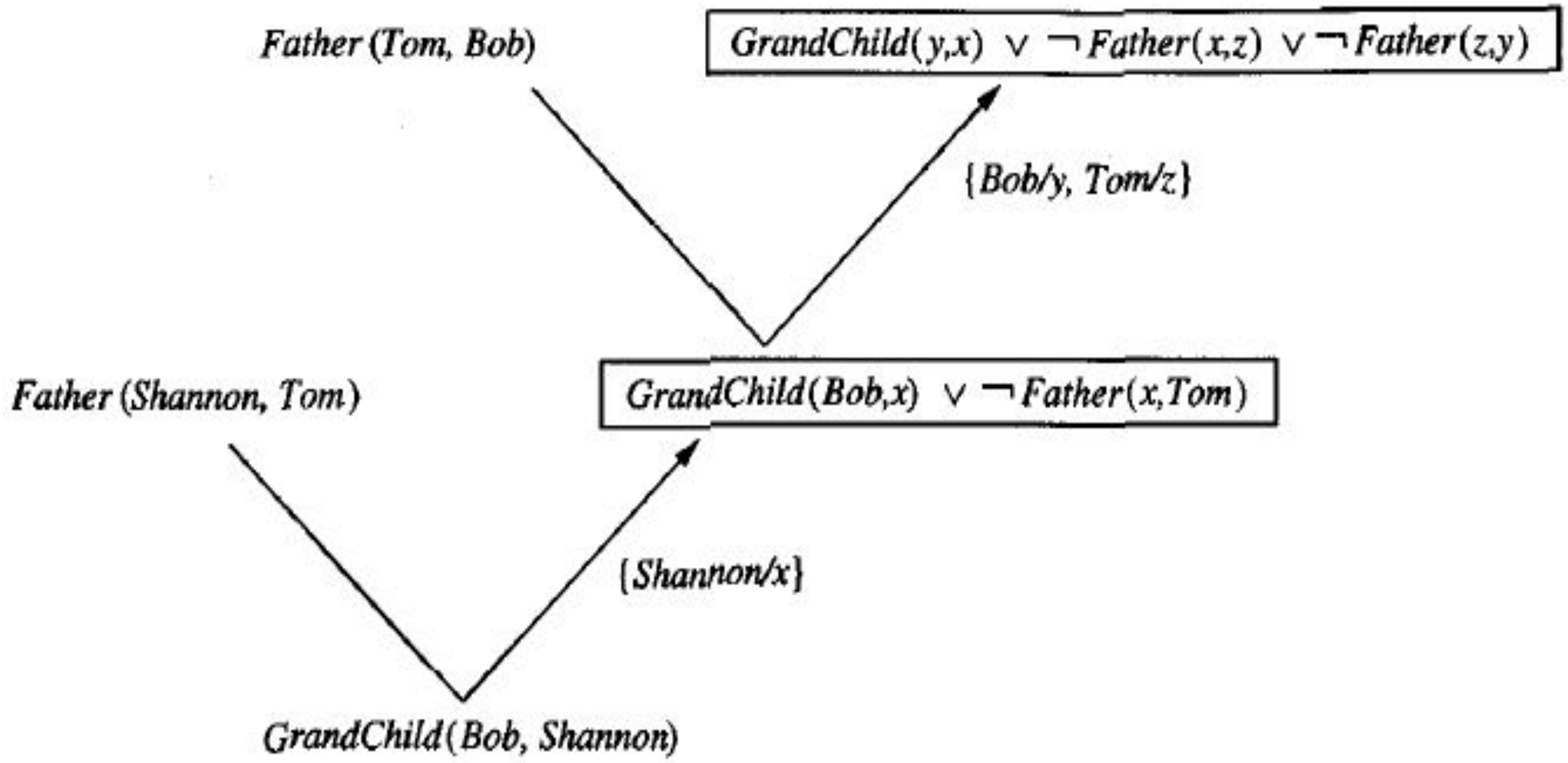
$$C_2 = (C - (C_1 - \{L_1\})\theta_1)\theta_2^{-1} \cup \{\neg L_1\theta_1\theta_2^{-1}\}$$

- Let us look at an example:
- Here we wish to learn the target predicate GrandChild(y,x). The given training data is D=GrandChild(Bob,Shanon). The background information B={Father(Shanon,Tom), Father(Tom,Bob)}

Inverting Resolution: First Order Case

- Here C=GrandChild(Bob,Shanon) and C1=Father(Shanon,Tom)
- To apply the inverse resolution operator we have only one choice L1=Father(Shanon,Tom).
- Let us choose inverse substitution θ_1^{-1}={}, θ_2^{-1}={Shanon/x}
- The resultant clause C2=(C-(C1-{L1}) θ_1) θ_2^{-1}=(C θ_1) θ_2^{-1}= GrandChild(Bob,x)
- And the clause {!L1 θ_1 θ_2^{-1}}=!Father(x,Tom).
- The resultant clause is GrandChild(Bob,x) v !Father(x,Tom) which is equivalent to GrandChild(Bob,x)$\leftarrow$Father(Tom,x)

Inverting Resolution: First Order Case



Summary of Inverse Resolution

- Inverse resolution provides a general approach to automatically generating hypotheses ***h*** that satisfy the constraint $(B \wedge h \wedge x_i) \vdash f(x_i)$.
- Beginning with the resolution rule and solving for the clause ***C2,*** the inverse resolution rule is easily derived.
- The inverse resolution rule has the advantage that it generates ***only*** hypotheses that satisfy $(B \wedge h \wedge x_i) \vdash f(x_i)$.
- The generate-and-test search of FOIL generates many hypotheses at each search step, including some that do not satisfy this constraint.
- One disadvantage is that the inverse resolution operator can consider only a small fraction of the available data when generating its hypothesis at any given step, whereas FOIL considers all available data to select among its syntactically generated hypotheses.

Generalization, $\boldsymbol{\theta}$-Subsumption, and Entailment

- In the earlier discussion we pointed out the correspondence between induction and inverse entailment.
- Since we are using general-to-specific ordering let us consider the relationship between more-general-than and inverse entailment.
- Let us consider the following definitions:
- **more-general-than:** Given two boolean-valued functions hj(x) and hk(x), we say that hj $>=_g$ hk if and only if $(\forall x)hk(x) \rightarrow hj(x)$.
- **$\boldsymbol{\theta}$-subsumption**: Consider two clauses Cj and Ck, both of the form H v ***L1*** v . . . Ln, where H is a positive literal, and the ***Li*** are arbitrary literals. Clause Cj is said to θ -subsume clause Ck if and only if there exists a substitution θ such that Cjθ $is\ contained\ in$ Ck.

Generalization, $\boldsymbol{\theta}$-Subsumption, and Entailment

- **Entailment:** Consider two clauses Cj and Ck. Clause Cj is said to entail clause Ck (written Cj $\vdash$ Ck) if and only if Ck follows deductively from Cj.

PROGOL

- An alternative approach to inverse resolution is to use inverse entailment to generate just the single most specific hypothesis that along with the background information fits the data.
- The most specific hypothesis can then be used to bound a general-to-specific search through the hypothesis space similar to that used by FOIL.
- There is an additional constraint that the only hypotheses considered are hypotheses more general than this bound.

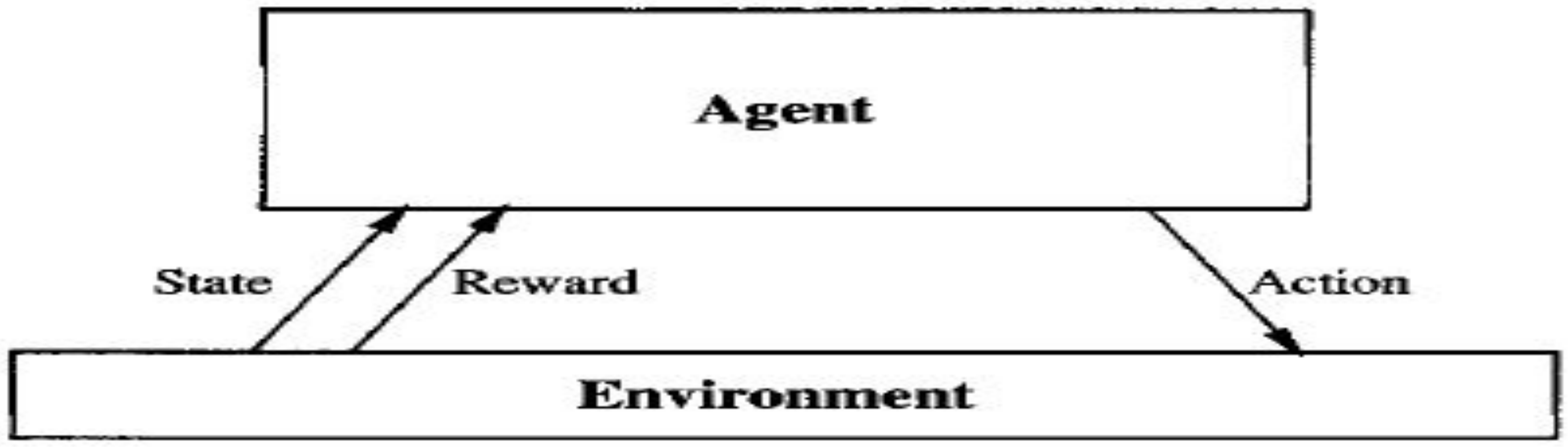
PROGOL

- The specified approach is employed by PROGOL system, whose algorithm is summarized as follows:
    - The user specifies a restricted language of first-order expressions to be used as the hypothesis space H.
    - Restrictions are stated using "mode declarations,” which enable the user to specify the predicate and function symbols to be considered, and the types and formats of arguments for each.
    - **PROGOL** uses a sequential covering algorithm to learn a set of expressions from H that cover the data.
    - **PROGOL** then performs a general-to-specific search of the hypothesis space bounded by the most general possible hypothesis and by the specific bound $\boldsymbol{h}_i$ calculated in the previous step.
    - Within this set of hypotheses, it seeks the hypothesis having minimum description length.

Reinforcement Learning

- Consider building a learning robot. The robot, or ***agent,*** has a set of sensors to observe the ***state*** of its environment, and a set of ***actions*** it can perform to alter this state.
- Sensors can be camera, sonars and actions can be move forward or move backward.
- The goals of the agent can be defined by a **reward** function that assigns a numerical value-an immediate payoff-to each distinct action the agent may take from each distinct state.
- The problem of learning a control policy to maximize cumulative reward is very general and covers many problems beyond robot learning tasks.

Reinforcement Learning



$$s_0 \xrightarrow[r_0]{a_0} s_1 \xrightarrow[r_1]{a_1} s_2 \xrightarrow[r_2]{a_2} \dots$$

Goal: Learn to choose actions that maximize

$r_0 + \gamma r_1 + \gamma^2 r_2 + \dots$, where $0 \leqslant \gamma < 1$

Reinforcement Learning

- The target function to be learned in this case is a control policy, π : ***S*** → ***A,*** that outputs an appropriate action ***a*** from the set ***A,*** given the current state ***s*** from the set ***S.***
- The reinforcement learning problem differs from other function approximation tasks in several important respects.
  - **Delayed Reward:**
  - The task of the agent is to learn a target function ***n*** that maps from the current state ***s*** to the optimal action ***a*** = π***(s).*** In reinforcement learning, however, training information is not available in this form.
  - The trainer provides only a sequence of immediate reward values as the agent executes its sequence of actions. The agent, therefore, faces the problem of ***temporal credit assignment:*** determining which of the actions in its sequence are to be credited with producing the eventual rewards.

Reinforcement Learning

- ***Exploration.***
- In reinforcement learning, the agent influences the distribution of training examples by the action sequence it chooses.
- This raises the question of which experimentation strategy produces most effective learning.
- The learner faces a tradeoff in choosing whether to favor ***exploration*** of unknown states and actions, or ***exploitation*** of states and actions that it has already learned will yield high reward.
- ***Partially observable states.*** Although it is convenient to assume that the agent's sensors can perceive the entire state of the environment at each time step, in many practical situations sensors provide only partial information.

Reinforcement Learning

- ***Life-long learning.*** Unlike isolated function approximation tasks, robot learning often requires that the robot learn several related tasks within the same environment, using the same sensors.

The Learning Task

- Here we formulate the problem of learning sequential control strategies more accurately.
- We define one general formulation of the problem based on Markov decision processes.
- In a Markov decision process (MDP) the agent can perceive a set ***S*** of distinct states of its environment and has a set ***A*** of actions that it can perform.
- At each discrete time step *t,* the agent senses the current state s_t, chooses a current action a_t, and performs it.

The Learning Task

- The environment responds by giving the agent a reward $r_t = r(s_t, a_t)$ and by producing the succeeding state $s_{t+1} = \delta(s_t, a_t)$.
- Here the functions ***δ*** and ***r*** are part of the environment and are not necessarily known to the agent.
- In an MDP, the functions $\delta(s_t, a_t)$ and $r(s_t, a_t)$ depend only on the current state and action, and not on earlier states or actions.
- First let us consider ***δ and r to be deterministic.***

The Learning Task

- The task of the agent is to learn a ***policy, π*** : ***S*** → A, for selecting its next action ***a,*** based on the current observed state ***s_t;*** that is, ***π(s_t)*** = a_t.(How is the learning going to be?)
- We define the cumulative value ***$V^\pi(s_t)$*** achieved by following an arbitrary policy ***π*** from an arbitrary initial state ***s_t*** as follows:

$$V^\pi(s_t) \equiv r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots$$
$$\equiv \sum_{i=0}^{\infty} \gamma^i r_{t+i}$$

- Here $0<\gamma<1$

The Learning Task

- The quantity $V^{\pi}(s_t)$ defined by the Equation is often called the **discounted cumulative reward** achieved by policy π from initial state s.
- There are other definitions of total reward like:
- **Finite horizon reward** $\sum_{i=0}^{h} r_{t+i}$ which considers the undiscounted sum of rewards over a finite number h of steps.
- **Average reward** $\lim_{h \to \infty} \frac{1}{h} \sum_{i=0}^{h} r_{t+i}$, which considers the average reward per time step over the entire lifetime of the agent.
- Here we consider only the discounted cumulative reward.

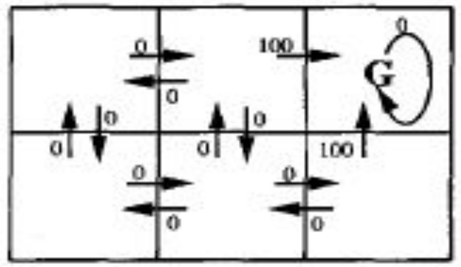
The Learning Task

- Now we define the agent's learning task.
- We require that the agent learn a policy π that maximizes V $^{\pi}$(s) for all states s. We will call such a policy an **optimal policy** and denote it by π *.

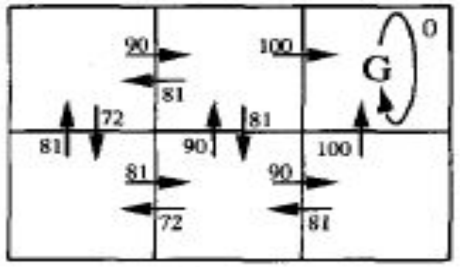
$$\pi^* \equiv \underset{\pi}{\operatorname{argmax}}\, V^{\pi}(s), (\forall s)$$

- To simplify notation, we will refer to the value function V $^{\pi *}$(s) of such an optimal policy as V*(s).
- V*(s) gives the maximum discounted cumulative reward that the agent can obtain starting from state s; that is, the discounted cumulative reward obtained by following the optimal policy beginning at state s.

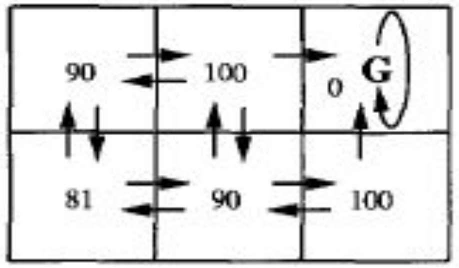
The Learning Task



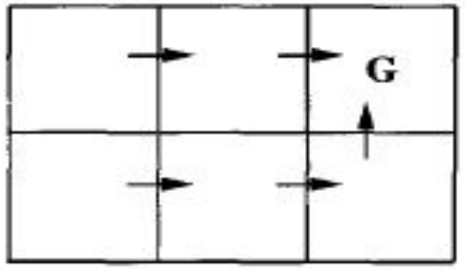
$r(s, a)$ (immediate reward) values



$Q(s, a)$ values



$V^*(s)$ values



One optimal policy

- Each arrow in the diagram represents a possible action the agent can take to move from one state to another.
- The number associated with each arrow represents the immediate reward r(s, a) the agent receives if it executes the corresponding state-action transition.
- G is the goal state and is also called **an absorbing state**.
- In this case we choose γ=.9
- The discounted reward from the bottom state of the diagram for V* is given below

$$0 + \gamma 100 + \gamma^2 0 + \gamma^3 0 + \cdots = 90$$

Q Learning

- It is difficult to learn the function π^*: S$\rightarrow$A because the available training data does not provide training examples in the form <s,a>.
- The training examples are provided in the form of rewards r<s_i,a_i> for I =1,2,…..
- Here we try to learn a numerical evaluation function defined over states and actions and then implement the optimal policy.
- One obvious choice as to which numerical evaluation function must be learned is V*.
- The agent policy must choose among the actions and not states.

Q Learning

- We can use V* in certain settings to choose among actions also.
- The optimal action in state s is the action a that maximizes the sum of immediate reward r(s,a) plus the value V* of the immediate successor state discounted by γ.

$$\pi^*(s) = \underset{a}{\operatorname{argmax}}[r(s,a) + \gamma V^*(\delta(s,a))]$$

- The agent can acquire the optimal policy by learning V* provided it has knowledge of immediate reward r and the state transition function δ.
- In many practical problems it is impossible to predict the exact outcome of an arbitrary action to an arbitrary state.

The Q Function

- In cases where δ or r is unknown learning V* is of no use for selecting optimal actions.
- The evaluation function Q is the one which we will be using.
- Let us now define the function Q(s,a).
- The value of Q is the reward received immediately upon executing action a from state s, plus the value of following the optimal policy thereafter.

$$Q(s,a) \equiv r(s,a) + \gamma V^*(\delta(s,a))$$

- We can rewrite the equation of π^* as

$$\pi^*(s) = \underset{a}{\operatorname{argmax}}\, Q(s,a)$$

The Q Function

- The agent needs to consider each available action a in its current state s and choose the action that maximizes Q(s,a).
- The agent can choose globally optimal action sequences by reacting repeatedly to the local values of Q for the current state.
- The function Q for the current state and action summarizes in a single number all the information needed to determine the discounted cumulative reward that will be gained in the future if action a is selected in state s. (Example)

An Algorithm for Learning Q

- The relationship between Q and V* is

$$V^*(s) = \max_{a'} Q(s, a')$$

- The equation of Q(s,a) can be rewritten as :

$$Q(s, a) = r(s, a) + \gamma \max_{a'} Q(\delta(s, a), a')$$

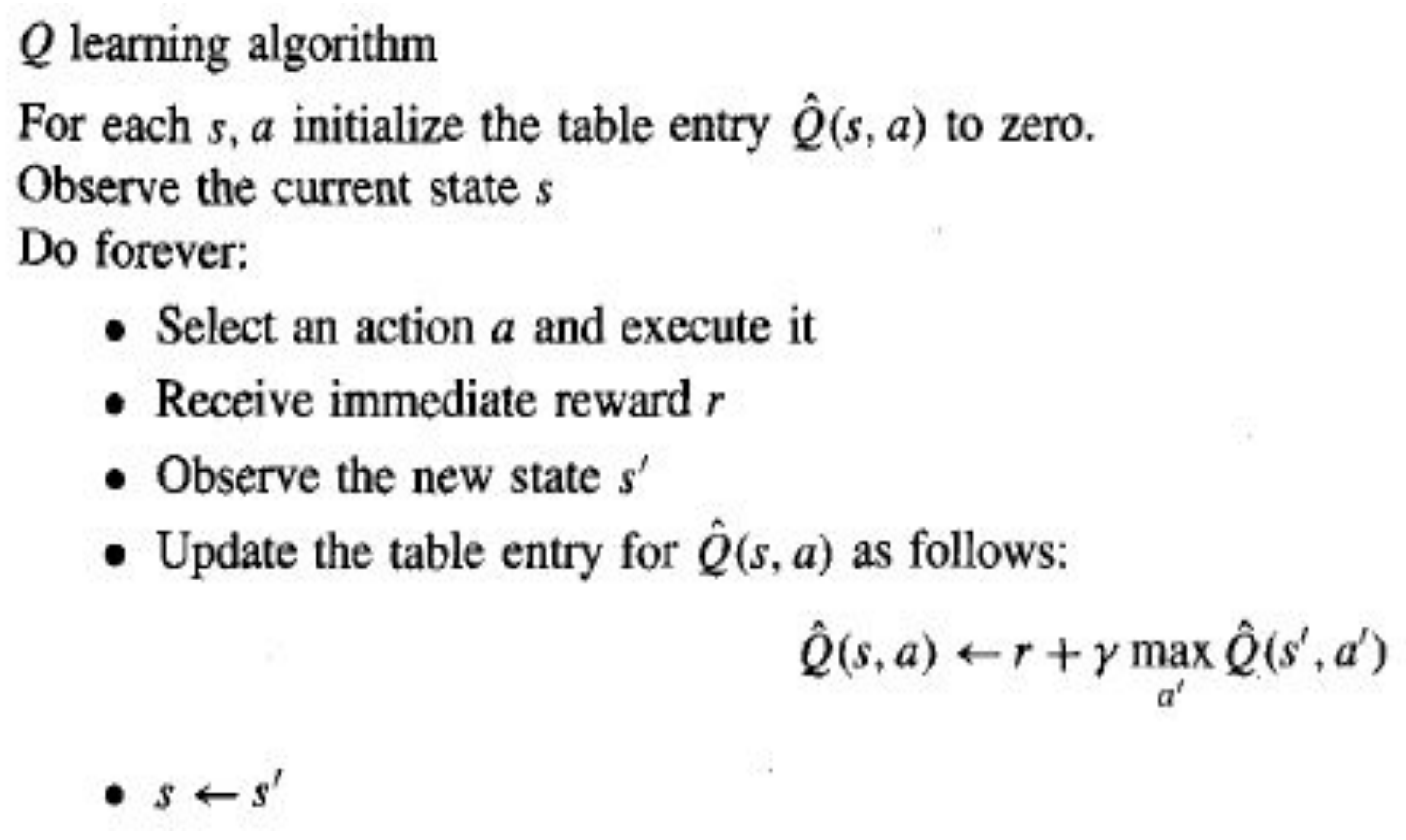
- The above recursive definition of Q provides the basis for algorithms that iteratively approximate Q.
- We use the symbol $\hat{Q}$ to refer to the learner's estimate or hypothesis of the actual Q function.
- The learner represents its hypothesis by a large table with a separate entry for each state-action pair.

An Algorithm for Learning Q

- The table entry for the pair <s,a> stores the value for $\hat{Q}(s,a)$ the learners current hypothesis about the actual but unknown value of Q(s,a).
- The table is initialized to any random value(zero).
- The agent chooses a state s executes an action a and observes the reward r=r(s,a) and the new state s'=δ(s,a).
- The entry for $\hat{Q}(s,a)$ is updated.

$$\hat{Q}(s,a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s',a')$$

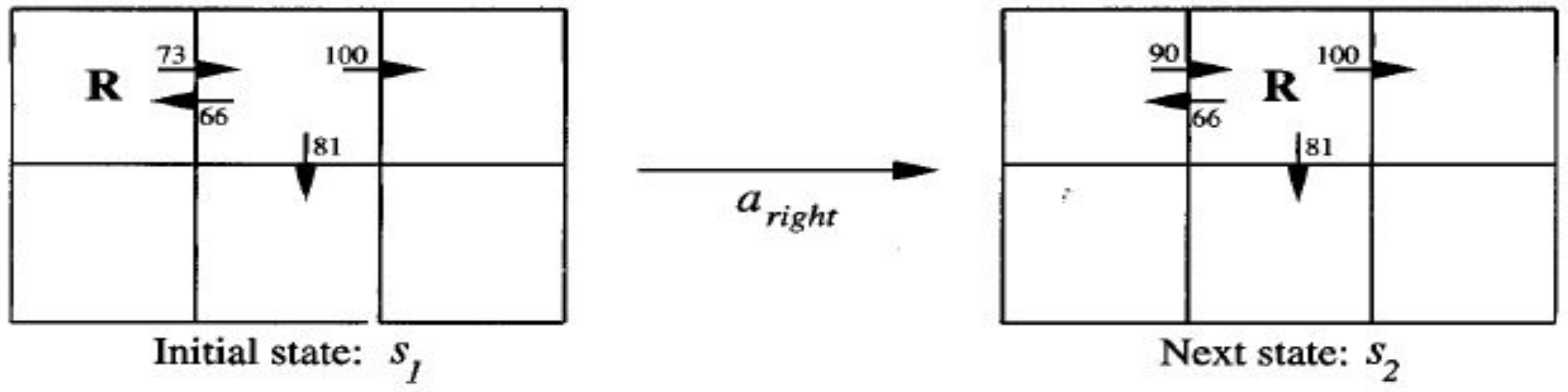
An Algorithm for Learning Q



Q learning algorithm

```
For each $s, a$ initialize the table entry $\hat{Q}(s, a)$ to zero.
Observe the current state $s$
Do forever:
    • Select an action $a$ and execute it
    • Receive immediate reward $r$
    • Observe the new state $s'$
    • Update the table entry for $\hat{Q}(s, a)$ as follows:
          $\hat{Q}(s, a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s', a')$
    • $s \leftarrow s'$
```

An Illustrative Example



Initial state: s_1

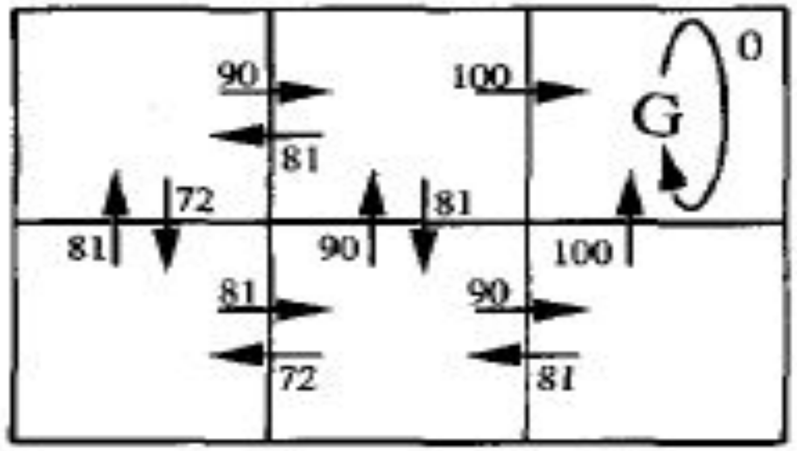
Next state: s_2

$$\begin{aligned} \hat{Q}(s_1, a_{right}) &\leftarrow r + \gamma \max_{a'} \hat{Q}(s_2, a') \\ &\leftarrow 0 + 0.9 \ \max\{66, 81, 100\} \\ &\leftarrow 90 \end{aligned}$$

An Illustrative Example

- Let us apply this algorithm to the previous grid world problem.
- Since the world consists of absorbing goal state we assume that training consists of a series of episodes.
- In each episode a random state is chosen and allowed to execute actions until it reaches the absorbing goal state.
- The table entry happens when the goal state is reached with a non-zero reward.
- Slowly in a step-by-step manner the entire table is filled.

An Illustrative Example



$Q(s, a)$ values

An Illustrative Example

- Let us consider two general properties of Q Learning algorithm that hold for any deterministic MDP in which the rewards are non-negative, assuming we initialize all $\hat{Q}$ values to zero.
- The first property is that under these conditions the $\hat{Q}$ value never decreases during training.

$$(\forall s, a, n) \quad \hat{Q}_{n+1}(s, a) \geq \hat{Q}_n(s, a)$$

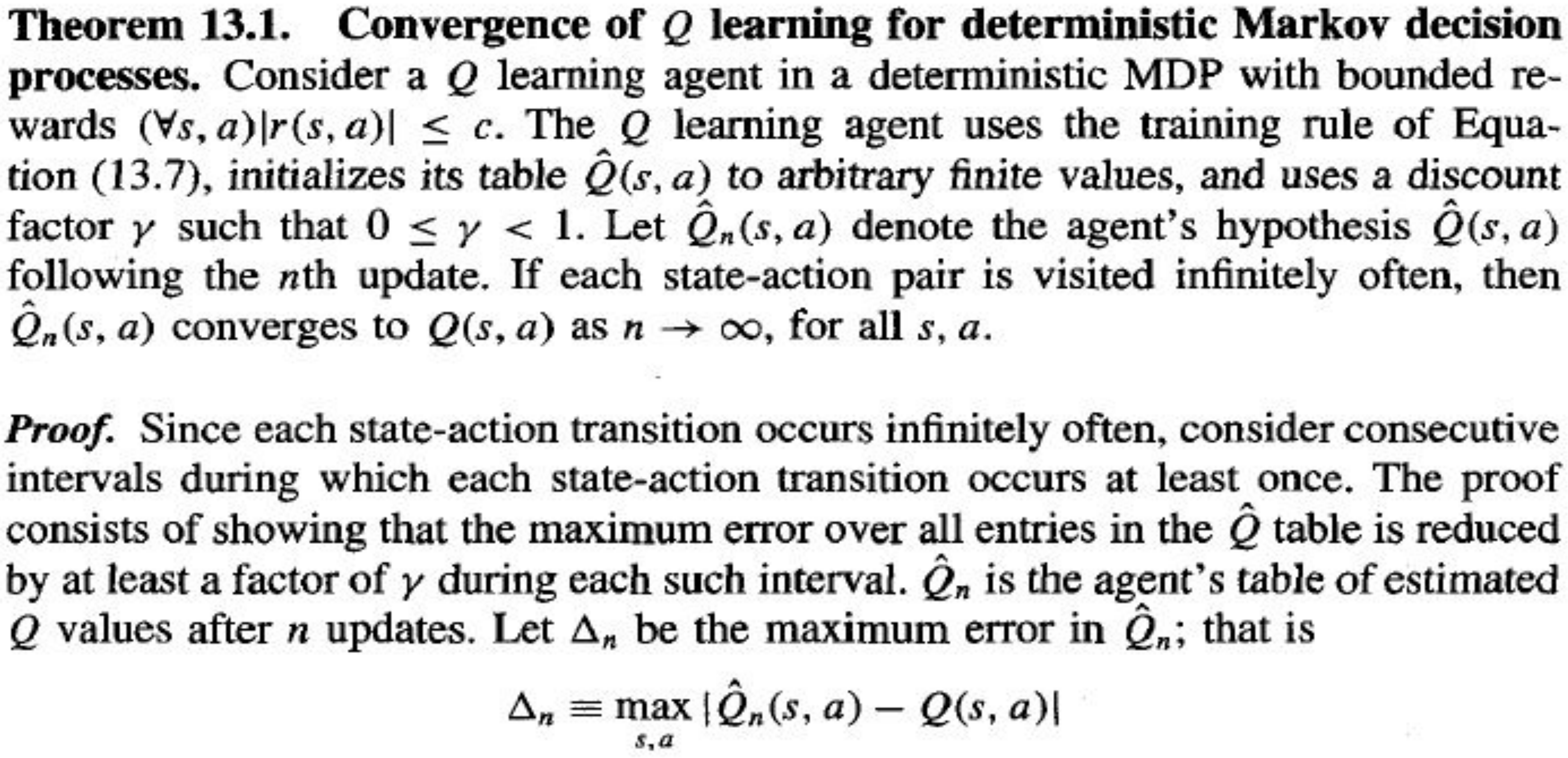
- The second property is that throughout the training process every $\hat{Q}$ value will remain in the interval between 0 and its true Q value.

$$(\forall s, a, n) \quad 0 \leq \hat{Q}_n(s, a) \leq Q(s, a)$$

Convergence

- The question to be answered is “will the algorithm discussed converge toward a equal to $\hat{Q}$ e true Q function?”
- The answer is yes in certain conditions:
  - We assume the system is a deterministic MDP.
  - We assume the immediate reward values are bounded.
  - We assume the agent selects actions in such a fashion that it visits every possible state-action pair infinitely often.
- The key idea underlying the proof of convergence is that the table entry $\hat{Q}$ **(s,**a) with the largest error must have its error reduced by a factor of *γ* whenever it is updated.

Convergence

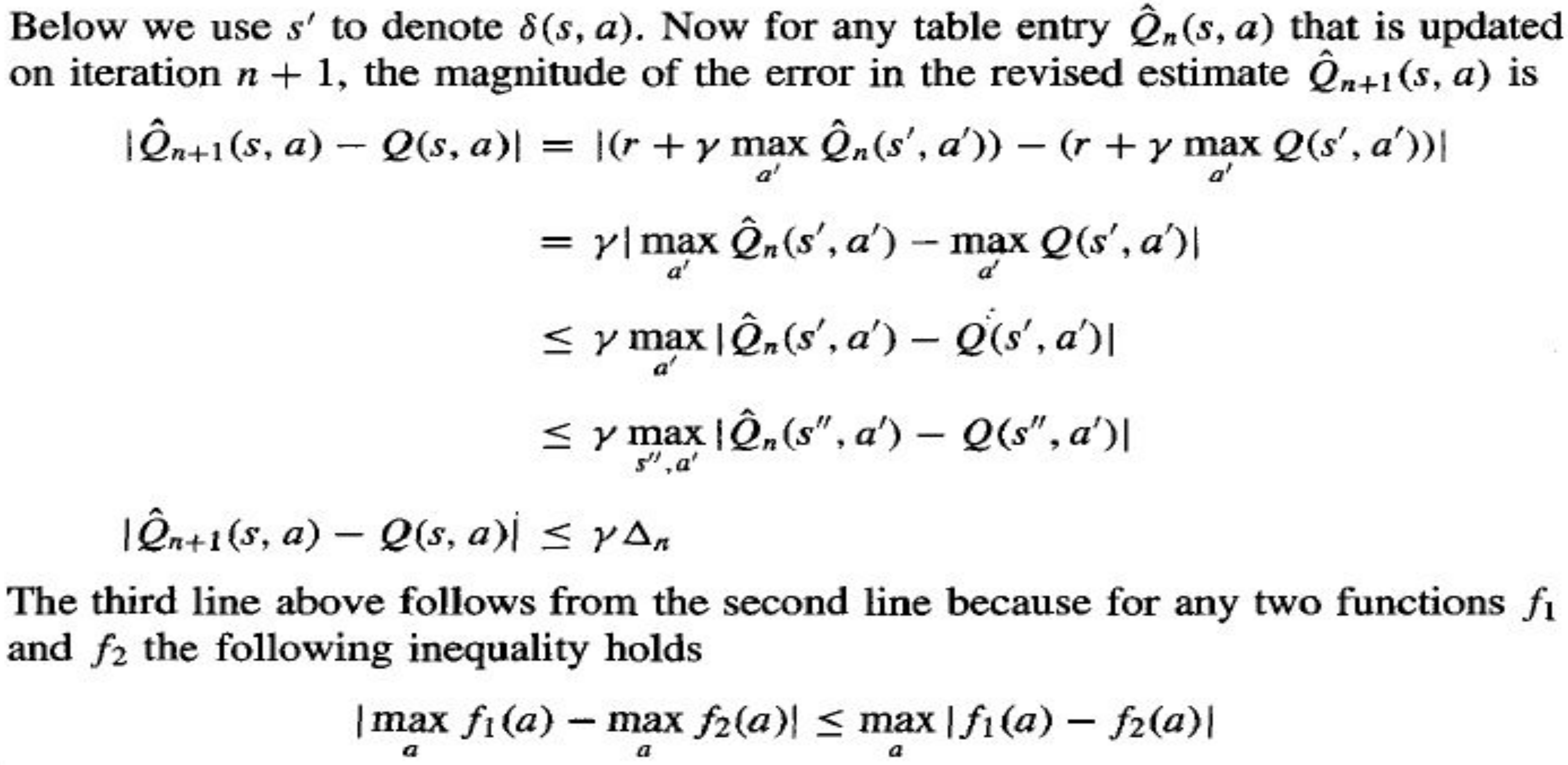


Theorem 13.1. **Convergence of Q learning for deterministic Markov decision processes.** Consider a Q learning agent in a deterministic MDP with bounded rewards $(\forall s, a)|r(s, a)| \leq c$. The Q learning agent uses the training rule of Equation (13.7), initializes its table $\hat{Q}(s, a)$ to arbitrary finite values, and uses a discount factor γ such that $0 \leq \gamma < 1$. Let $\hat{Q}_n(s, a)$ denote the agent's hypothesis $\hat{Q}(s, a)$ following the nth update. If each state-action pair is visited infinitely often, then $\hat{Q}_n(s, a)$ converges to $Q(s, a)$ as $n \rightarrow \infty$, for all s, a.

Proof. Since each state-action transition occurs infinitely often, consider consecutive intervals during which each state-action transition occurs at least once. The proof consists of showing that the maximum error over all entries in the $\hat{Q}$ table is reduced by at least a factor of γ during each such interval. $\hat{Q}_n$ is the agent's table of estimated Q values after n updates. Let Δ_n be the maximum error in $\hat{Q}_n$; that is

$$\Delta_n \equiv \max_{s,a} |\hat{Q}_n(s, a) - Q(s, a)|$$

Convergence



Below we use s' to denote $\delta(s, a)$. Now for any table entry $\hat{Q}_n(s, a)$ that is updated on iteration $n + 1$, the magnitude of the error in the revised estimate $\hat{Q}_{n+1}(s, a)$ is

$$\begin{aligned}
|\hat{Q}_{n+1}(s, a) - Q(s, a)| &= |(r + \gamma \max_{a'} \hat{Q}_n(s', a')) - (r + \gamma \max_{a'} Q(s', a'))| \\
&= \gamma |\max_{a'} \hat{Q}_n(s', a') - \max_{a'} Q(s', a')| \\
&\leq \gamma \max_{a'} |\hat{Q}_n(s', a') - Q(s', a')| \\
&\leq \gamma \max_{s'', a'} |\hat{Q}_n(s'', a') - Q(s'', a')|
\end{aligned}$$

$$|\hat{Q}_{n+1}(s, a) - Q(s, a)| \leq \gamma \Delta_n$$

The third line above follows from the second line because for any two functions f_1 and f_2 the following inequality holds

$$|\max_a f_1(a) - \max_a f_2(a)| \leq \max_a |f_1(a) - f_2(a)|$$

Experimentation Strategies

- Q learning uses a probabilistic approach to select actions.
- Actions with higher $\hat{Q}$ values are assigned higher probabilities, but every action is assigned a nonzero probability.
- One way to assign such probabilities:

$$P(a_i|s) = \frac{k^{\hat{Q}(s,a_i)}}{\sum_j k^{\hat{Q}(s,a_j)}}$$

- $P(a_i/s)$ is the probability of selecting action a_i, given that the agent is in state s.
- K>0 is a constant that determines how strongly the selection favors actions with high $\hat{Q}$ value. (exploit and explore)

Updating Sequences

- Q learning can learn the Q function while training from actions chosen completely at random at each step as long as the resulting training sequence visits every state-action infinitely often.
- We run repeated identical episodes to fill the $\hat{Q}$ table backwards from the goal state at the rate of one new state-action transition per episode.
- Now consider training on these same state-action transitions, but in reverse chronological order for each episode.
- This training process will clearly converge in fewer iterations, although it requires that the agent use more memory to store the entire episode before beginning the training for that episode.

Updating Sequences

- A second strategy for improving the rate of convergence is to store past state-action transitions, along with the immediate reward that was received, and retrain on them periodically.
- Throughout the above discussion we have kept our assumption that the agent does not know the state-transition function *δ(s, a)* used by the environment to create the successor state *s'* = *δ (s, a),* or the function *r(s, a)* used to generate rewards.
- If the two functions are known in advance , then many more efficient methods are possible.
- A large number of efficient algorithms from the field of dynamic programming can be applied when the functions *δ* and *r* are known.

Nondeterministic Rewards and Actions

- In the nondeterministic case we must first restate the objective of the learner to take into account the fact that outcomes of actions are no longer deterministic.
- The obvious generalization is to redefine the value V^π of a policy π to be the *expected value* of the discounted cumulative reward received by applying this policy.

$$V^\pi(s_t) \equiv E\left[\sum_{i=0}^{\infty} \gamma^i r_{t+i}\right]$$

Nondeterministic Rewards and Actions

- We define the optimal policy π^* to be the policy π that maximizes $V^{\pi}(s)$ for all states *s.*
- We now generalize our earlier definition of *Q* by taking its expected value.

$$\begin{aligned} Q(s,a) &\equiv E[r(s,a) + \gamma V^*(\delta(s,a))] \\ &= E[r(s,a)] + \gamma E[V^*(\delta(s,a))] \\ &= E[r(s,a)] + \gamma \sum_{s'} P(s'|s,a) V^*(s') \end{aligned}$$

- We can re-express Q recursively as:

$$Q(s,a) = E[r(s,a)] + \gamma \sum_{s'} P(s'|s,a) \max_{a'} Q(s',a')$$

Nondeterministic Rewards and Actions

- Now the training rule for the nondeterministic environment is:

$$\hat{Q}_n(s, a) \leftarrow (1 - \alpha_n)\hat{Q}_{n-1}(s, a) + \alpha_n[r + \gamma \max_{a'} \hat{Q}_{n-1}(s', a')]$$

where

$$\alpha_n = \frac{1}{1 + visits_n(s, a)}$$

Temporal Difference Learning

- ***Q*** learning can be treated as a special case of a general class of ***temporal difference*** algorithms that learn by reducing discrepancies between estimates made by the agent at different times.
- Let ***Q'(s,,a ,)*** denote the training value calculated by this one-step look ahead:

$$Q^{(1)}(s_t, a_t) \equiv r_t + \gamma \max_a \hat{Q}(s_{t+1}, a)$$

- The value for two steps is:

$$Q^{(2)}(s_t, a_t) \equiv r_t + \gamma r_{t+1} + \gamma^2 \max_a \hat{Q}(s_{t+2}, a)$$

Temporal Difference Learning

- For n steps it will be:

$$Q^{(n)}(s_t, a_t) \equiv r_t + \gamma r_{t+1} + \cdots + \gamma^{(n-1)} r_{t+n-1} + \gamma^n \max_a \hat{Q}(s_{t+n}, a)$$

- Sutton introduces a general method for blending these alternative training estimates, called ***TD(λ).*** The idea is to use a constant 0 <=λ<=1 combine the estimates obtained from various look-ahead distances as shown below:

$$Q^{\lambda}(s_t, a_t) \equiv (1-\lambda)\left[Q^{(1)}(s_t, a_t) + \lambda Q^{(2)}(s_t, a_t) + \lambda^2 Q^{(3)}(s_t, a_t) + \cdots\right]$$

An equivalent recursive definition for Q^{λ} is

$$Q^{\lambda}(s_t, a_t) = r_t + \gamma[\ (1-\lambda) \max_a \hat{Q}(s_t, a_t) \\ + \lambda\ Q^{\lambda}(s_{t+1}, a_{t+1})]$$

Temporal Difference Learning

- If λ=0 then we have $Q^{(1)}$ as the training estimate.
- If λ=1 then only the r_{t+1} values are considered.
- The motivation for the TD(λ) method is that in some settings training will be more efficient if more distant look-aheads are considered.
- For example, when the agent follows an optimal policy for choosing actions, then Q^{λ} with λ = 1 will provide a perfect estimate for the true Q value, regardless of any inaccuracies in Q.
- On the other hand, if action sequences are chosen sub-optimally, then the $\boldsymbol{r_{t+i}}$ observed far into the future can be misleading.

Generalizing from Examples

- The algorithms we discussed perform a kind of rote learning and make no attempt to estimate the Q value for unseen state-action pairs by generalizing from those that have been seen.
- This rote learning assumption is reflected in the convergence proof, which proves convergence only if every possible state-action pair is visited.
- This is clearly an unrealistic assumption in large or infinite spaces, or when the cost of executing actions is high.
- More practical systems often combine function approximation methods discussed in earlier with the Q learning training rules described here.

Generalizing from Examples

- We can use backpropogation into the Q learning algorithm by substituting a neural network for the lookup table and using each update as a training example: $\hat{Q}(s, a)$
  - We can encode the state s and action a as network inputs and train the network to output the target values of Q given by the training rules.
  - We can train a separate network for each action, using the state as input and Q as output.
  - We can train one network with the state as input, but with one Q output for each action.

Relationship to Dynamic Programming

- Reinforcement learning methods such as ***Q*** learning are closely related to a long line of research on dynamic programming approaches to solving Markov decision processes.
- Dynamic Programming assumes that the agent possesses perfect knowledge of the functions δ(s,a) and r(s,a).
- Q learning assumes that the agent does not have any knowledge about the above functions.
- Let us look at Bellman equation which forms the foundation for many dynamic programming approaches:

$$(\forall s \in S) V^*(s) = E[r(s, \pi(s)) + \gamma V^*(\delta(s, \pi(s)))]$$