



MODULE 5

- Learning Concepts: Version space search. Back propagation learning.
- Social and emergent models of learning-genetic algorithm, classifier systems and genetic programming

- One of the most often heard criticism of AI is that machine cannot be called intelligent until they are **able to learn new things and to adapt to new situations**, rather than simply doing as they are told to do.

ability to adapt to new surroundings and to solve new problems

Knowledge acquisition activities

- Role learning-Simple storing of computed information
- Advice from others
- Problem solving experience
- Learning from examples

Rote Learning

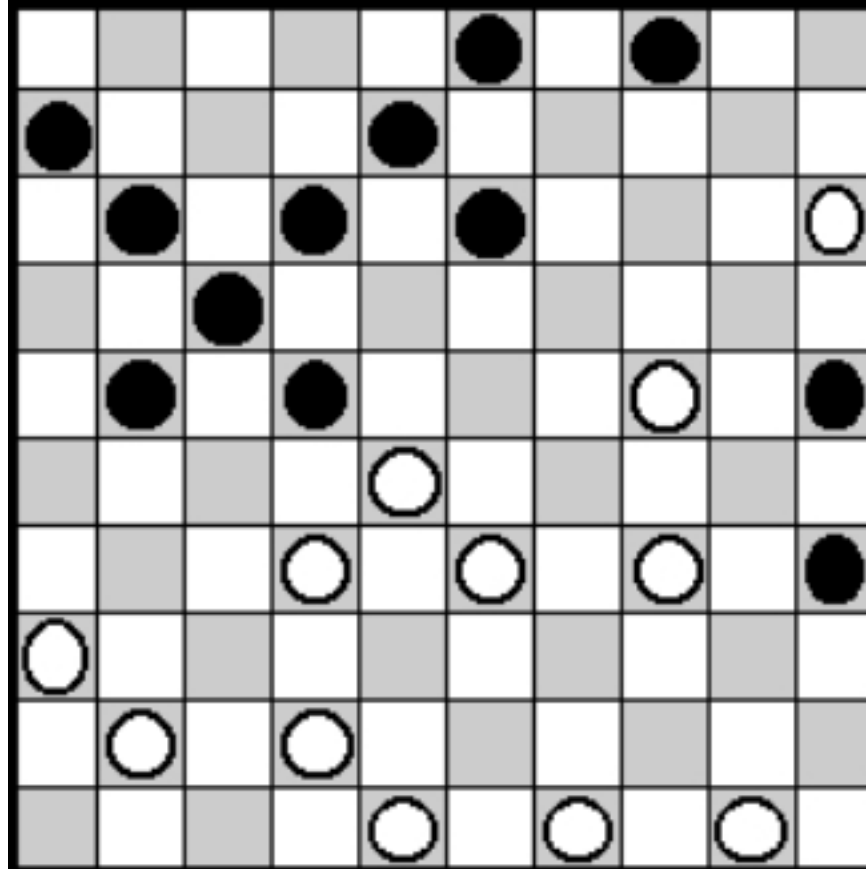


Prepared by Sharika T R, SNGCE

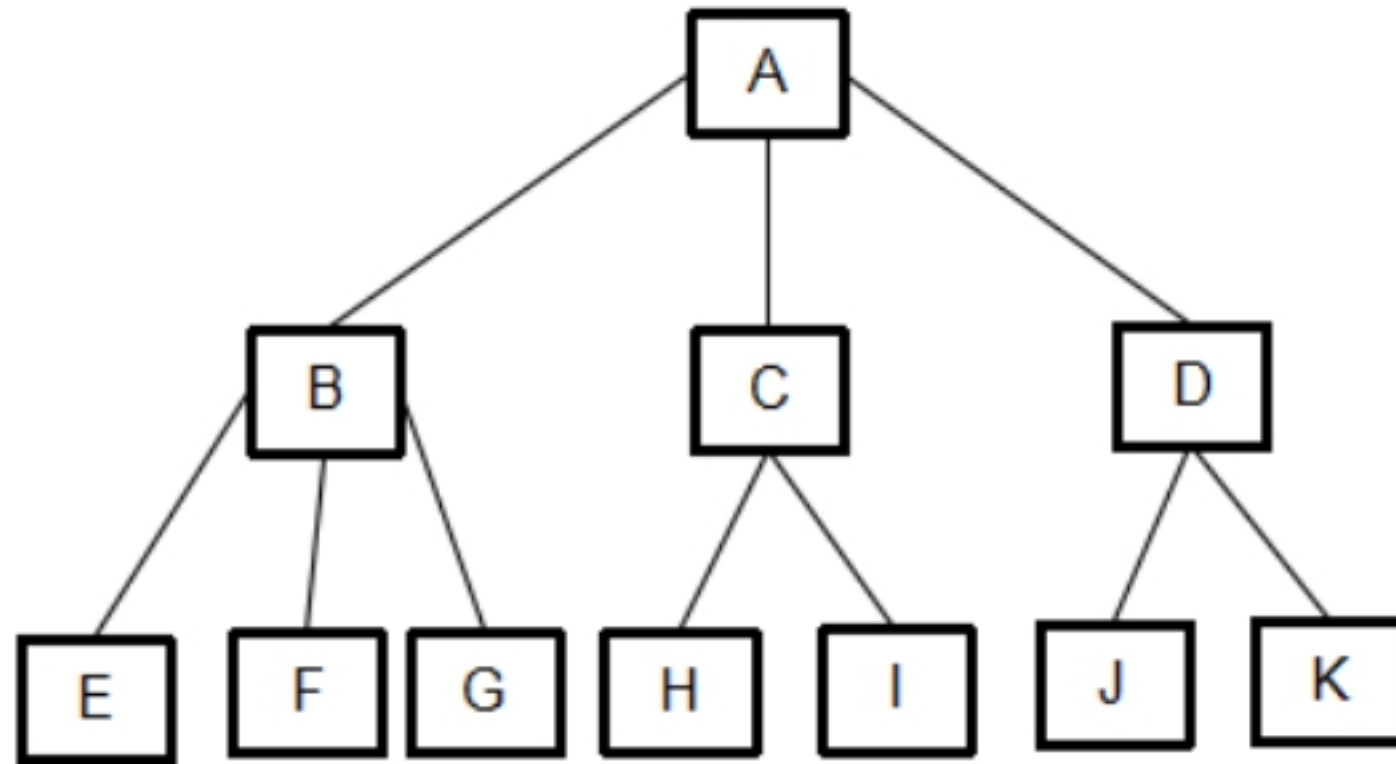
- In Data Caching, we store computed values so that we do not have to recompute them later.
- When computation is more expensive than recall, this strategy can save a significant amount of time.
- Caching has been used in AI programs to produce some surprising performance improvements.
- Such caching is known as rote learning.

Samuel's Checker Program

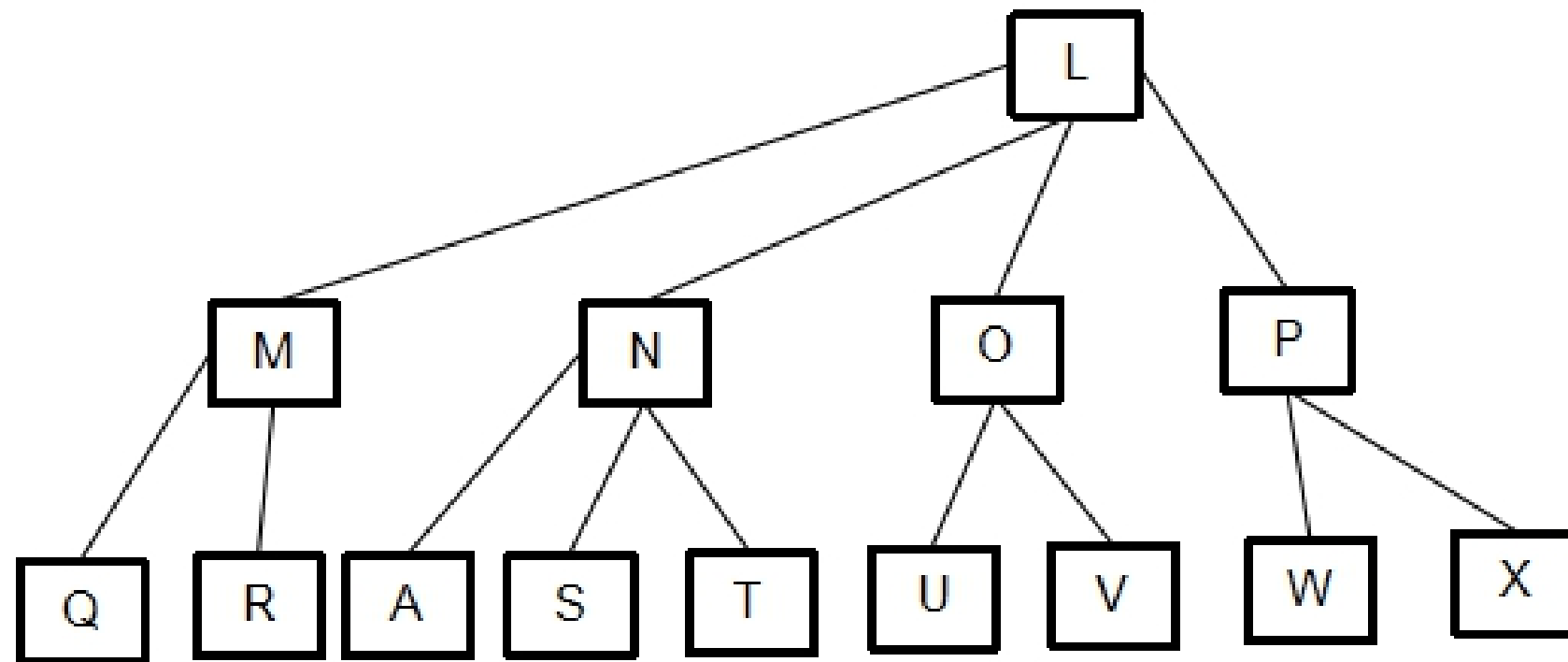
- This program learned to play checker well enough to beat its creator. It exploited two kind of learning:
 1. Rote learning and
 2. Parameter adjustments



- Samuel's program used the **minmax procedure** to explore checker game trees.
- time constraints permitted to search only a few levels in the tree.
- When it could search no deeper,
 - it applied the static evaluation function to board positions and use that score to continue its search of game tree.
- When it finished searching the tree and propagating the values backward, it had a score for the position represented by the root of the tree.
- It could choose the best move and make it.
- But it also recorded the board position at the root of the tree and the backed up score that had just been computed for it.
- But it also recorded the board position at the root of the tree and backed up score that had just been computed for it.



Scored value of A:10



10

Storing backed up values

- Organized storage of information:
 - It is faster to use a stored value than it would be to recompute it, there must be a way to access the stored values quickly.
 - Eg, indexing positions.
- Generalization:
 - the number of distinct objects that might potentially be stored can be very large.
 - To keep the number of stored objects down to a manageable level, some kind of generalization is necessary.

Learning by taking advice

- When a programmer writes a series of instructions into a computer, a fundamental kind of learning taking place.
- The programmer is sort of a teacher.
- Computer is sort of a student.
- After being programmed the computer is now able to do something it previously could not.
- Some interpreter or compiler must intervene to change the programmer's instruction into code that machine can execute directly.

Learning the game of hearts

- FOO(First Operational Operationaliser) tries to convert high level advice into effective executable procedure

- It is a game played as series of tricks.
- One player who has the lead plays a card
- Other players follow in turn and plays a card
 - The player must follow suit
 - If he cannot he play any of his card
- The player who plays the highest value card wins the trick and the lead.
- Winning player takes the card played in the trick
- The aim is to avoid taking points. Each heart counts as one point. The queen of spade is worth 13 points.
- The winner is the person that after all tricks have been played and has the lowest points score



- Hearts is a game of partial information with no known algorithm for winning.
- Although the possible situations are numerous general advice can be given such as
 - Avoid taking points
 - Do not lead a high card in suit in which an opponent is void
 - If an opponent has the queen of spade try to flush it.

A human user first translate the advice from english to a representation that FOO can understand.

- Avoid taking points.

`(avoid(take_points me) (trick))`

- FOO operationalises the advice by translating it into expression it can use in the game.
- UNFOLD an expression by replacing some term by its definition.

- By unfolding the definition of `avoid` FOO comes up with `(acheive (not (during (trick) (takepoints me))))`
- FOO considers the advice to apply to the player called “me”.

- Next, FOO UNFOLDS the definition of trick

(acheive (not (during

(scenario

(each pl(player) (play_card pl) --->(1)

(take-trick (trick-winner))) --->(2)

(take-point me))))

- In other words the player should avoid taking points during the scenario consisting of
 1. player playing cards and
 2. One player taking the trick

- FOO then uses case analysis to determine which step could cause one to **take points**.
- so FOO UNFOLDS the definition of take points.

```
(acheive (not (there-exists c1 (card-played)
                          (there-exists c2 (point-cards)
                          (during (take (trick-winner) c1)
                          (take me c2))))))
```

Advice says that **player should avoid taking point-cards during the process of trick-winner taking the trick.**

- Question for FOO: under what condition does **(take me c2)** occur during?

(take (trick-winner) c1) ?

- By using a technique called **partial match**, FOO hypothesized that points will be taken if

me=trick_winner and c2=c1

- Advice: “Do not win a trick that has points”.

**(acheive (not (and (have-points (card-played))
 (=(trick-winner) me))))**

- **Advice:** When playing a card that is the same suit as the card that was played first, if the trick possibly contains points, then play a low card.

```
(acheive (>=(and (in-suit-led(card-of me))  
                  (possible(trick-has-points)))  
         (low (card-of me))))
```

- At last FOO has translated the rather vague advice “avoid taking points” into a specific useable heuristic.
- FOO is able to play a better game of hearts after receiving this advice.
- A human can watch FOO play, detect new mistakes correct them through yet more advice, such as “Play high cards when it is safe to do so”
- The ability to operationalize knowledge is critical for systems that learn from a teacher’s advice.
- It is also an important component of **explanation-based learning**.

Can a program get better without the aid of the programmer?
It can by generalizing from its own experiences.

1. Learning by Parameter Adjustment

- Many programs rely on an **evaluation procedure** that combines information from several sources into a single summary statistics
- Game playing programs do this in their static evaluation functions in which a variety of factors, such as
 - piece advantage and
 - mobility
- it is often difficult to know a priori how much weight should be attached to each feature being used.
- Features that appear to be good predictors of overall success will have their weights increased, while those that do not will have their weights decreased.

Samuel's Checkers Problem

- Its static evaluation function, the program used a polynomial of the form

$$C_1t_1 + C_2t_2 + \dots + C_{16}t_{16}$$

- The term t are the values of the sixteen features that contribute to the evaluation.
- The term C are coefficients(weights) that are attached to each of these values.
- As learning progresses, the C values will change.

- The most important question in the design of a learning program based on parameter adjustment is:
 1. When should the value of a coefficient be increased and when should it be decreased?
 2. How much the value should be changed?
- The simple answer to the first question is that
 - coefficients of term that predicted the final outcome accurately should be increased,
 - while the coefficients of poor predictors should be decreased.
- The problem of appropriately assigning responsibility to each of the step that led to a single outcome is known as the **credit assignment problem**.

2. Learning with Macro Operators

The idea is to avoid expensive recomputation.

- suppose you are faced with the problem of getting to the postoffice.
- Your solution may involve
 1. getting in your car
 2. Starting it
 3. Driving along a certain route

- Substantial planning may go into choosing the appropriate route, but you need not plan about how to go about starting your car.
- You are free to choose `START_CAR` as an atomic action, even though it really consist of several actions:
 1. sitting down
 2. Adjusting the mirror
 3. Inserting key
 4. Turning the key

- Sequence of actions that can be treated as a whole are called **MACRO-OPERATORS**.
- Macro-Operators were used in early problem solving system STRIPS.

In artificial intelligence, STRIPS (Stanford Research Institute Problem Solver) is an automated planner developed by Richard Fikes and Nils Nilsson in 1971 at SRI International. A STRIPS instance is composed of:

- An initial state;
- The specification of the goal states – situations which the planner is trying to reach;
- A set of actions. For each action, the following are included:
 - preconditions (what must be established before the action is performed);
 - postconditions (what is established after the action is performed).

- STRIPS has a learning component.
- After each problem solving episode, the learning component takes the computed plan and stores it away as macro-operator or **MACROP**.

A MACROP is just like a regular operator except it consist of a sequence of actions, not just a single one.

- A MACROP 's precondition are the initial conditions of the problem just solved, and its postconditions correspond to the goal just achieved.

Robot Arm to Manipulate Blocks

- The actions are

1. **UNSTACK(A,B):**

- pickup block A from its current position on block B.
- The arm must be empty and block A must have no blocks on top of it.

2. **STACK(A,B):**

- Place block A on block B.
- The arm must already be holding and the surface of must be clear.

3. **PICKUP(A):**

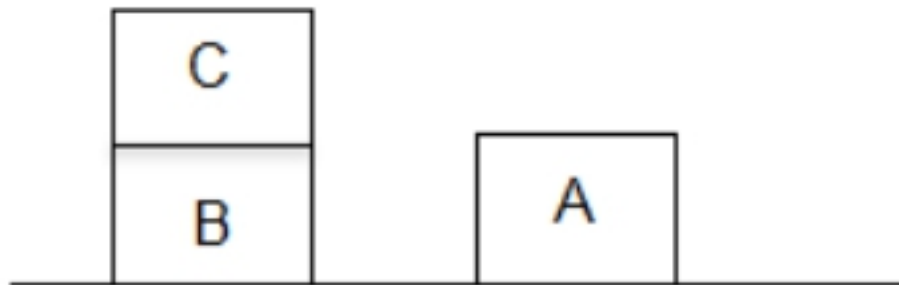
- Pick up block A from the table and hold it.
- The arm must be empty and there must be nothing on top of block A

4. **PUTDOWN(A):**

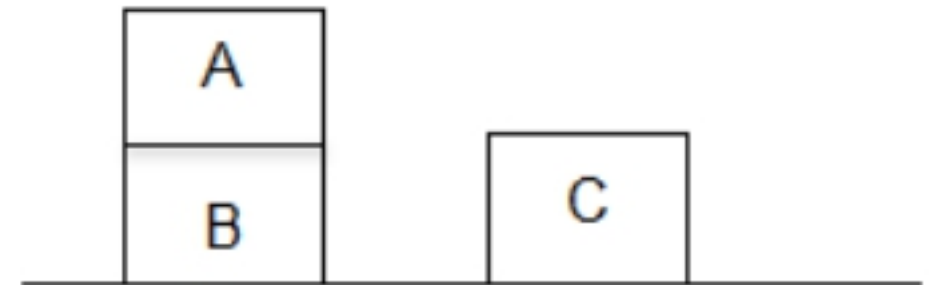
- Put block A down on the table.
- The arm must have been holding block A.

- Result Predicates

1. **ON(A,B)**: Block A is on Block B
2. **ONTABLE(A)**: Block A on table
3. **CLEAR(A)**: There is nothing on top of block A
4. **HOLDING(A)**: The arm is holding block A
5. **ARMEMPTY**: The arm is holding nothing



START:ON(C,B)

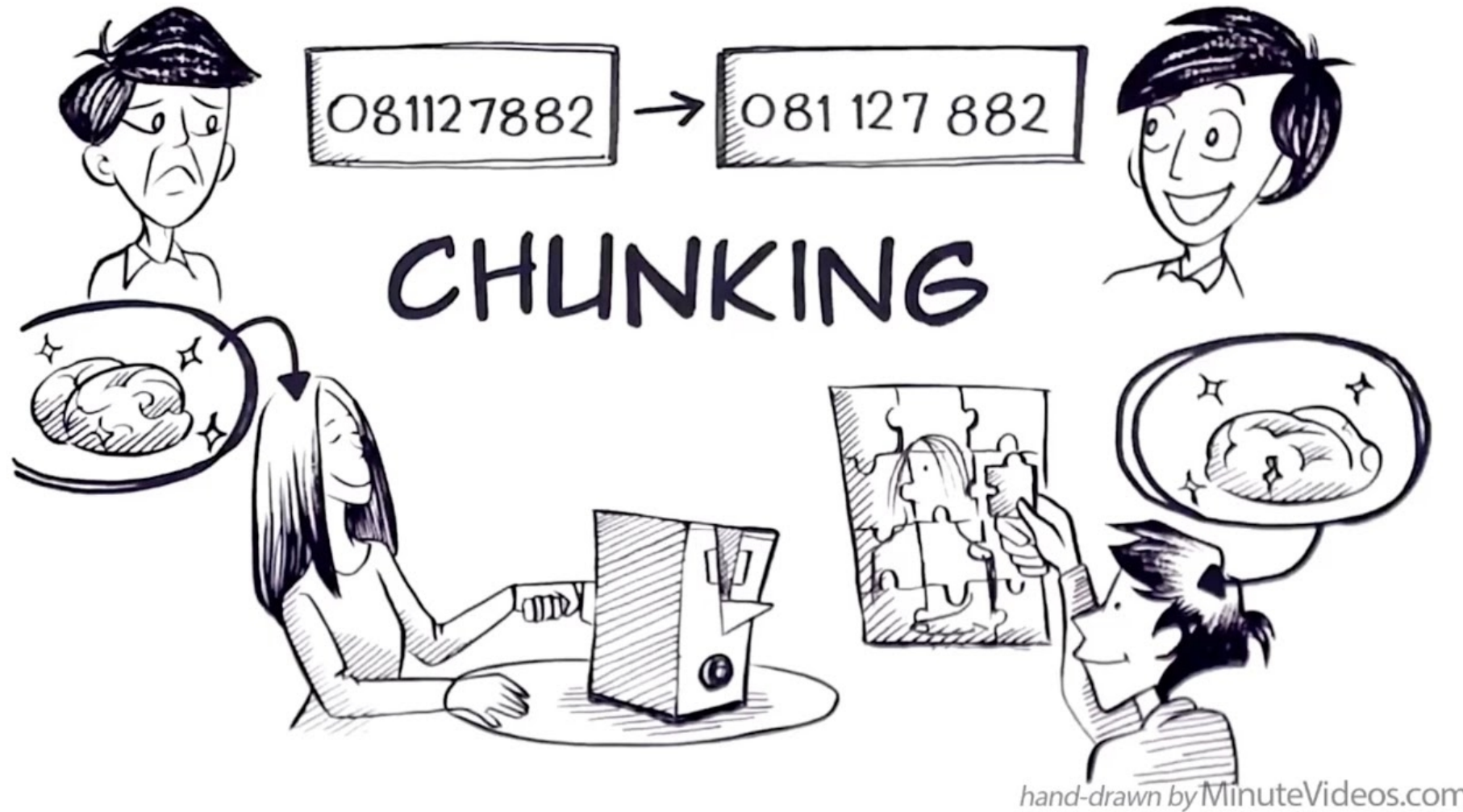


GOAL:ON(A,B)

- Suppose we are given an initial blocks world situation in which $ON(C,B)$ and $ON(A, Table)$ are both true.
- STRIPS can achieve the goal $ON(A,B)$ by devising a plan with four steps:
 1. $UNSTACK(C,B)$
 2. $PUTDOWN(C)$
 3. $PICKUP(A)$
 4. $STACK(A,B)$

- STRIPS now builds a MACROP with
 - **Precondition** ON(C, B), ON(A, Table) and
 - **Postcondition** ON(C, Table), ON(A, B)
- The body of the MACROP consist of the four steps.
- In future planing, STRIPS is free to use this complex macro-operator just as it would use any other operator.
- But rarely will STRIPS see the exact same problem again twice.
- We need **genaralization of MACROPs** before storing them

- Instead of storing the MACROP, STRIPS can generalize the plan to consist of the steps
 - 1.UNSTACK(x1,x2)
 - 2.PUTDOWN(x1) C=x1, B=x2, A=x3
 - 3.PICKUP(x3)
 - 4.STACK(x3, x2)
- This plan can then be stored with
 - **Preconditions:** ON(x1,x2), ON(x3, Table) and
 - **Postconditions:** ON(x1, Table), ON(x2, x3)



- Chunking is a process similar in flavor to macro-operators.
- The idea of chunking comes from psychological literature on memory and problem solving.
- Computational basis is production system.

Soar is a general cognitive architecture for developing systems that exhibit intelligent behavior. Researchers all over the world, both from the fields of artificial intelligence and cognitive science, are using Soar for a variety of tasks. It has been in use since 1983, evolving through many different versions to where it is now Soar, Version 9

SOAR is a general architecture for building intelligent systems.
SOAR is based on a set of specific, cognitively motivated hypothesis about the structure of human problem solving.

- SOAR system use of control knowledge.
- SOAR also exploit chunking so that its performance can increase with experience.
- The designer of SOAR hypothesis that chunking is a universal learning method, it account for all types of learning in intelligent systems.

1. **Long term memory** is stored as a set of production or rules
2. **Short term memory or working memory** is a **buffer** that is affected by perception and serves as a **storage area** for facts deducted by rules in long term memory. Working memory is analogous to **state description** in problem solving

3. All problem solving activity takes place as **state space traversal**.
 - There are **several classes** of problem solving activities including **reasoning** about which states to explore,
 - which **rules to apply** in a given situation and what effects those rules will have.
4. All intermediate and final results of problem solving are remembered for future references.

- SOAR solves problems by firing productions which are stored in long term memory.
- Some of those firing turns out to be more useful than others.

When SOAR detects a useful sequence of production firing, it creates a chunk which is essentially a large production that does the work of an entire sequence of smaller ones.

- Chunks are generalized before they are used.

- several chunks may encode a single macro operator and chunk may participate in a number of macro sequences.
- Chunk are generally applicable towards any goal state.
- This contrast with macro tables, which are structured towards searching a particular goal state from any initial state.
- Chucking emphasis how learning can occur during problem solving while macro tables are usually built during preprocessing stage.

Learning From Example: Induction

- **Classification** is the process of assigning to a particular input, the name of the **class** to which it belongs.
- Classification is an important component of many problem solving tasks.
- In its simplest form, it is presented as a straightforward **recognition task**.
- The idea of producing a classification program that can evolve its own class definition is appealing.

This task of constructing class definition is called concept learning or induction.

1. Isolate a set of features that are relevant to the task domain. Define each class by **weighted sum of values of these features**.
 - Each class is then defined by a scoring function that looks very similar to the scoring functions often used in other situations.
$$C_1t_1 + C_2t_2 + C_3t_3 + \dots$$
 - t: value of relevant parameter
 - c: weight
 - Negative weight can be used to indicate features whose presence usually constitutes negative evidence for a given class.
 - Done by Parameter Adjustment

2. Isolate a set of features that are **relevant to the task domain**. Define each class as a structure component of those features.
 - For eg; if the task is to identify animals, the body of each type of animal can be stored as a structure with various features representing such things as color, length of neck and feathers.

- To define classes structurally the three techniques used are
 1. Winston's Learning Program
 2. Version Spaces
 3. Decision Trees

Winston's Learning Program

- Winston describes an early structural concept learning program.
- This program operated in a simple blocks world domain.
- Its goal was to **construct representation of definition of concepts in the block domain.**

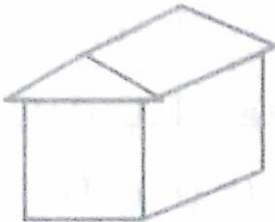
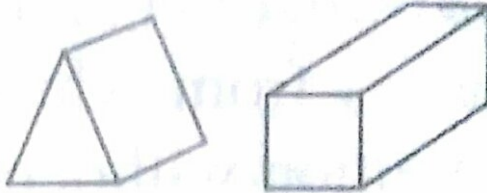
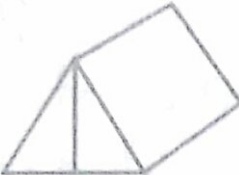

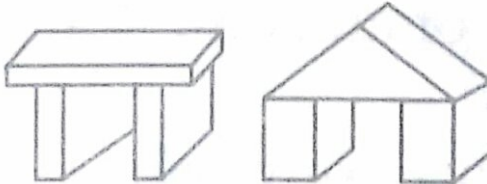
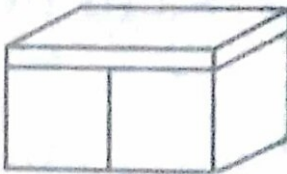
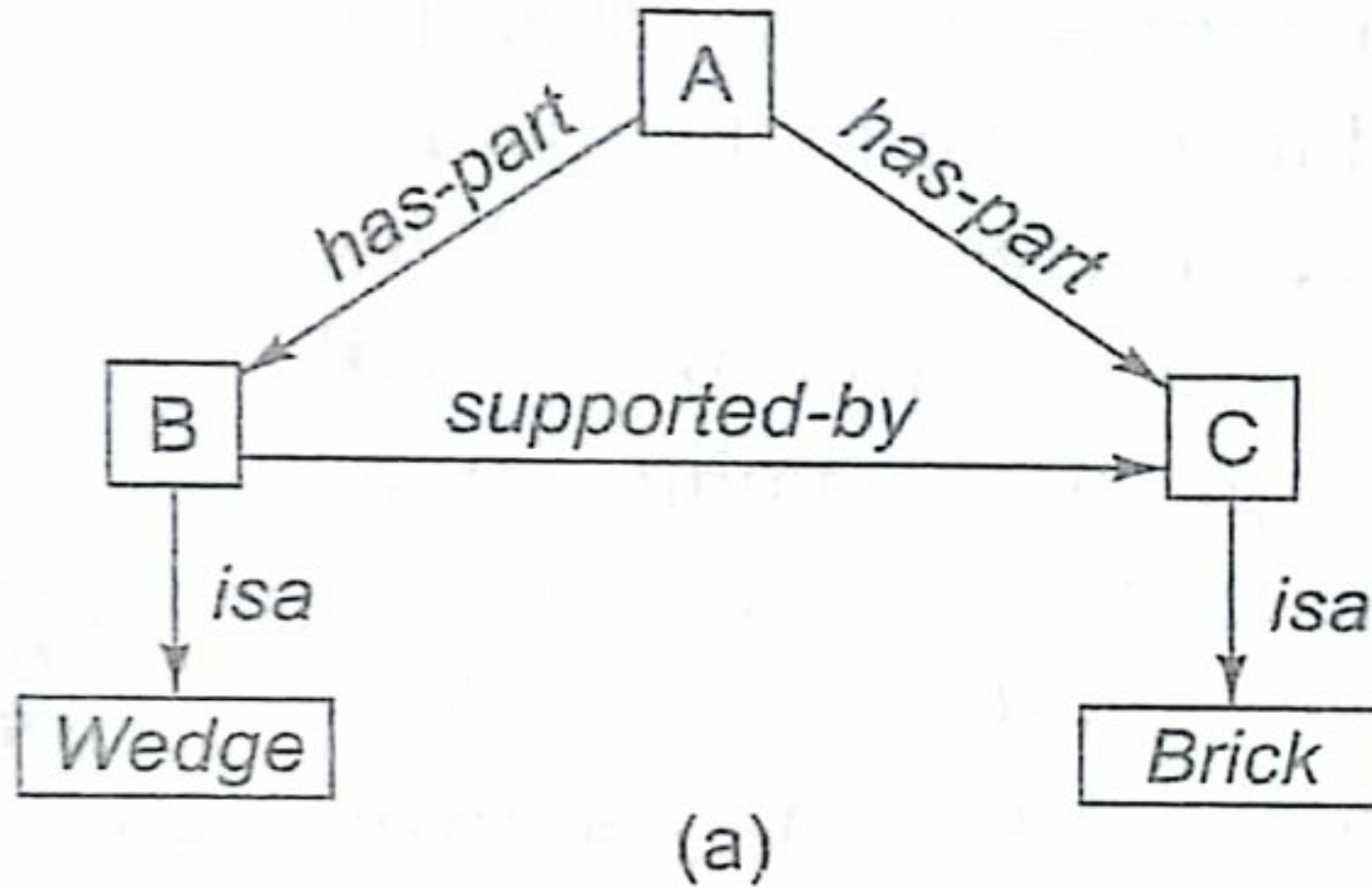
	Concept	Near Miss
House		
Tent		
Arch		

Fig. 17.2 *Some Blocks World Concepts*

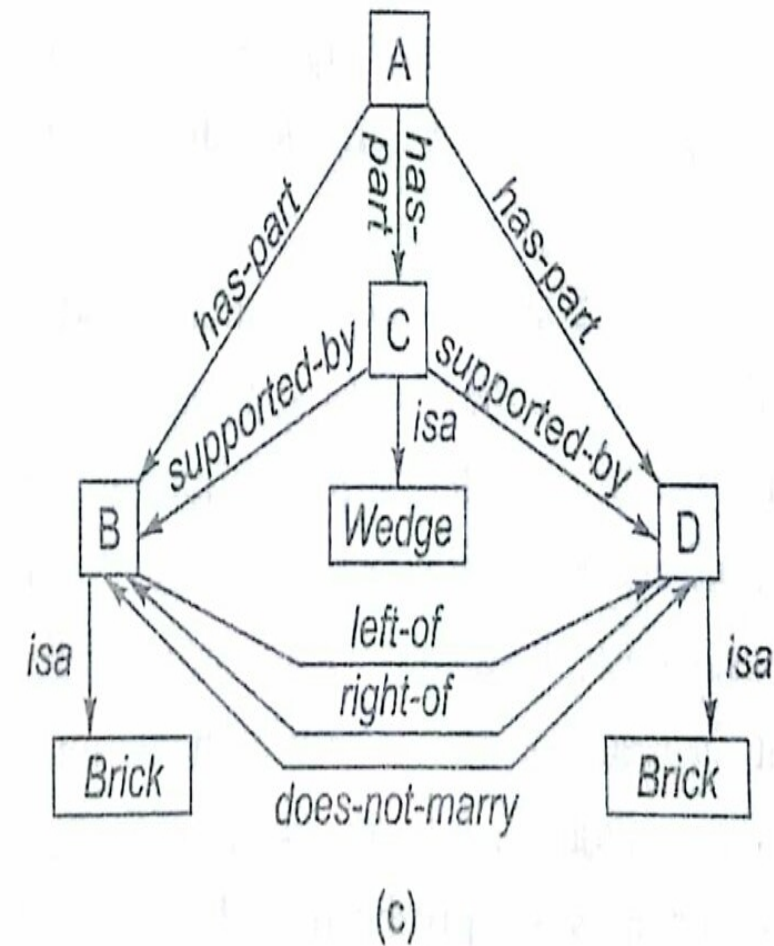
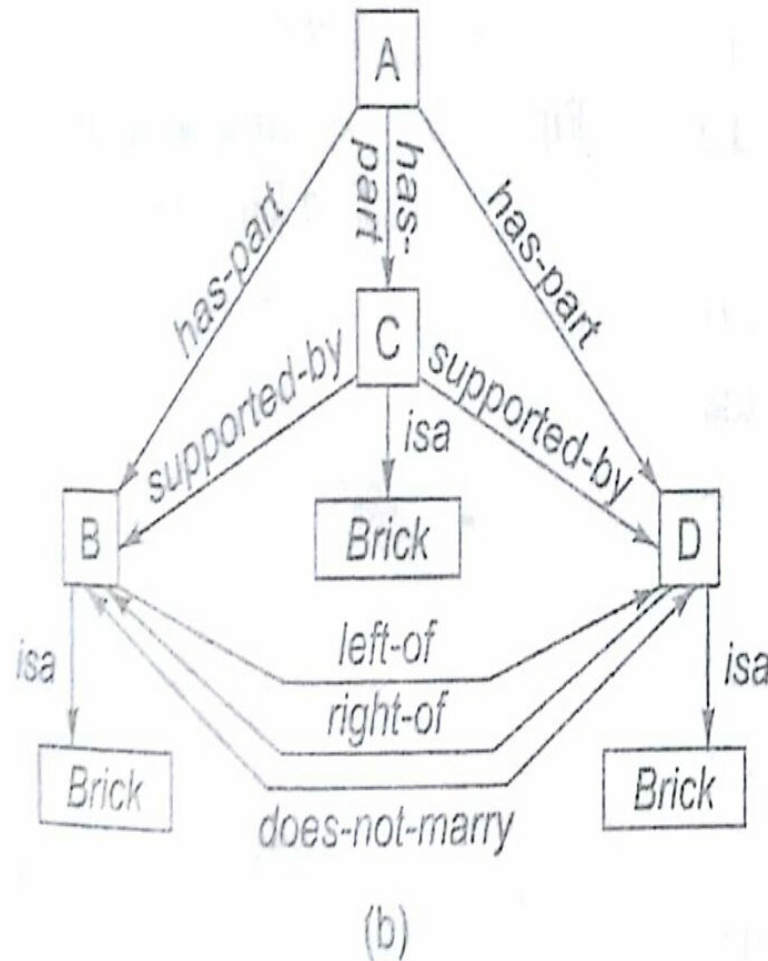
- it learned the concept House, Tent and Arch.
- The figure shows an example of a near miss for each concept.
- **Near miss** is an object that is **not an instance of concept** in the question but is very **similar to the instances**.

- The program started with a line of drawing of blocks world structure and constructs a **sematic net representation of structural description** of the objects.
- This **structural description** was then provided as **input** to learning program.

House: Structural Description



Arch: Structural Description



1. Begin with a structural description of one known instance of the concept. Call that description the concept definition
2. Examine description of other known instances of the concept. Generalize the definition to include them
3. Examine descriptions of near miss of the concept. Restrict the definition to exclude these.

- Goal
 - To produce a description **that is constant with all positive examples but no negative example** in the training set.
- Version space work by maintaining a set of possible descriptions and evolving that set as a new example and near misses are present.

An Example Concept Car

- Car023
 - Origin: Japan
 - Manufacturer: Honda
 - Color: Blue
 - Decade: 1970
 - Type : Economy

- Now suppose that each slot may contain only the discrete values.

<i>origin</i>	∈	{Japan, USA, Britain, Germany, Italy}
<i>manufacturer</i>	∈	{Honda, Toyota, Ford, Chrysler, Jaguar, BMW, Fiat}
<i>color</i>	∈	{Blue, Green, Red, White}
<i>decade</i>	∈	{1950, 1960, 1970, 1980, 1990, 2000}
<i>type</i>	∈	{Economy, Luxury, Sports}

Fig. 17.8 Representation Language for Cars

- The choice of features and value is called **bais of learning system**.
- By being embeded in a particular program and by using particular representation, every learning system is baised, because it learns some things more easily than others.
- In this example bais is fairly simple- eg, we can learn concepts that have to do with car manufacturers but not owners.
- In more complex system, the bais is less obivous.
- A clear statement of bias of a learning system is very important to its evaluation.

- For example the concept “Japanese Economy Car” can be represented as

Origin:	Japan
Manufacturer:	x1
Color:	x2
Decade:	x3
Type :	Economy

Given a representation language and given a positive and negative training example how can we produce a **concept description** that is consistent with all training examples?

Partial Ordering of Description

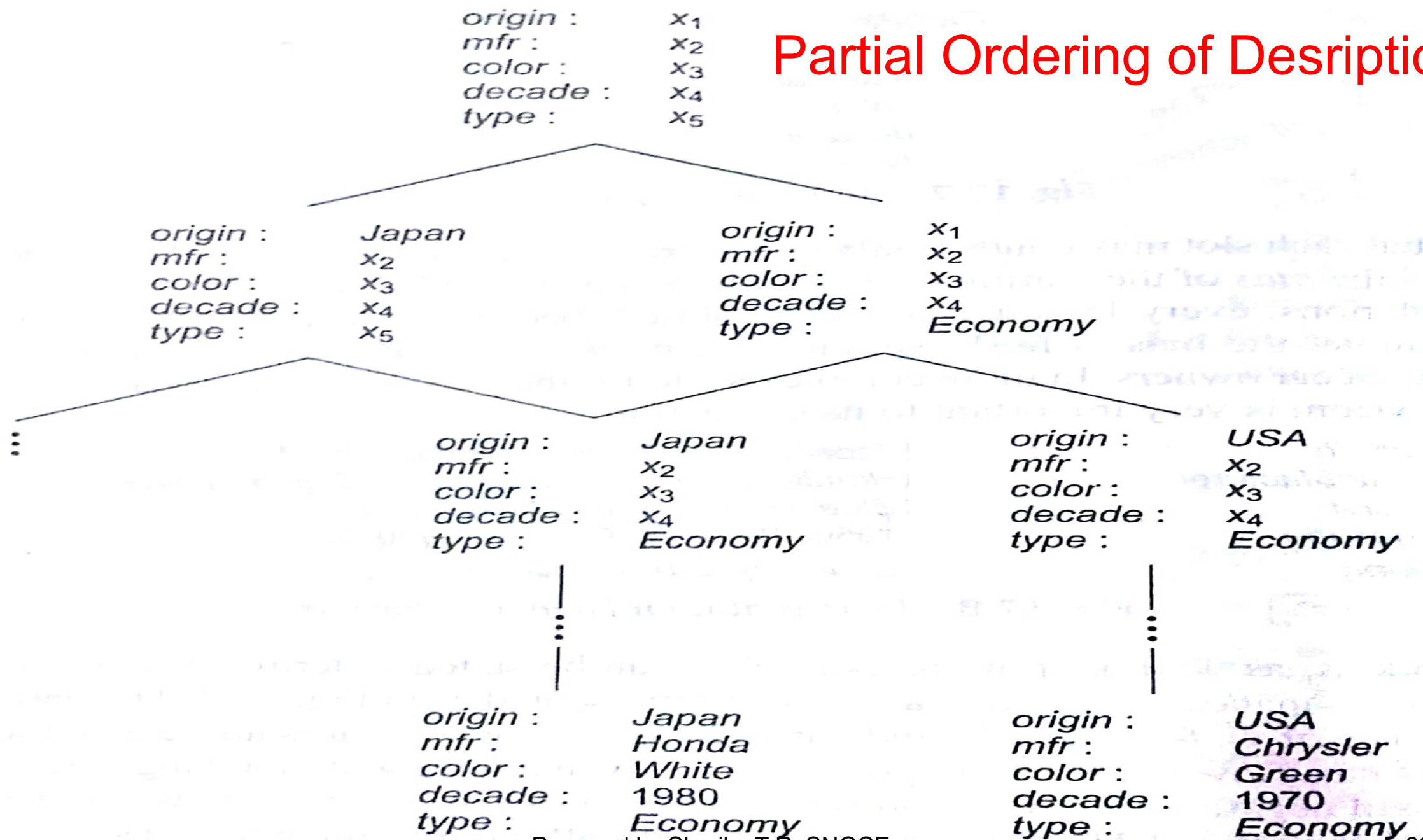


Fig. 17.10 Partial Ordering of Concepts Specified by the Representation Language

Concept and Version space

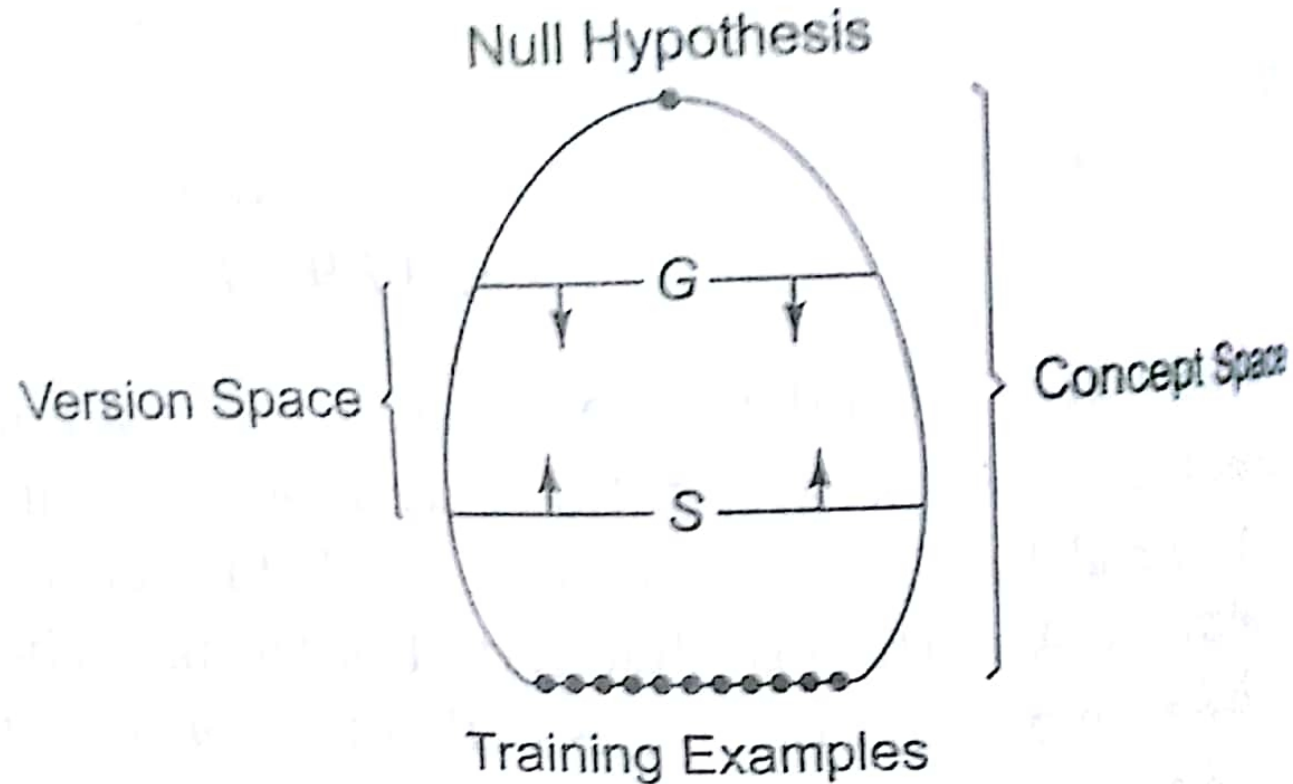


Fig. 17.11 *Concept and Version Spaces*

- At the top of the concept space is the null description consisting only of variables and at bottom are all the possible training instances, which contain no variables.
The version space is simply a set of description so initial idea is to keep an explicit list of those description.
- Unfortunately the number of descriptions in the context space is exponential in number of features and values. But Version Space has a concise representation.

- It consist of two subset of concept space:
 1. Subset G contain the most general descriptions consistent with training example
 2. Subset S contains most specific descriptions consistent with training examples.

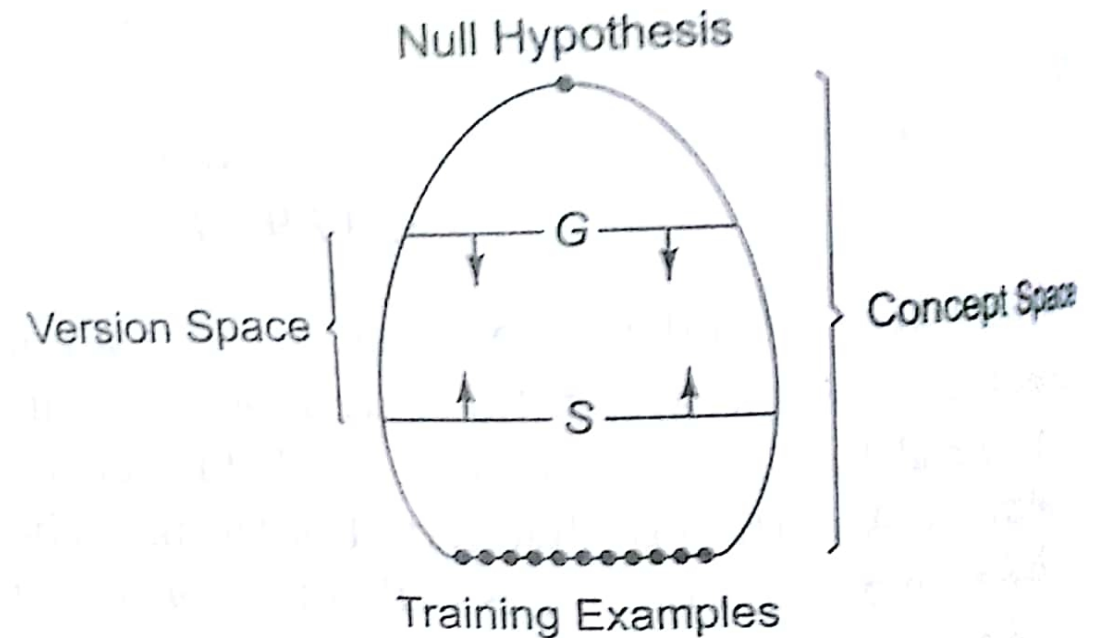


Fig. 17.11 *Concept and Version Spaces*

- Version space is the set of all descriptions that lie between some element of G and some element of S in partial order of concept space.
- This representation of version space is not only effective for storage **but also for modification**.
- Intuitively each time we receive a **positive training example** we want to make the S set more general.
- **Negative training example** serves to make the G more specific.
- If the S and G set coverage, our range of hypothesis will **narrow to a single concept description**.
- The algorithm for narrowing the version space is called **candidate elimination algorithm**.

Candidate Elimination Algorithm

- **Given:**
 - A representation language and set of positive and negative examples expressed in that language
- **Compute:**
 - A concept description that is constant with all the positive examples and none of the negative examples

1. Initialize G to contain one element: the null description (all features are variables)
2. Initialize S to contain one element: the first positive example

3. Accept a new training example:

– If it is a positive example,

- first **remove from G** any descriptions that do not cover the example.
- Then **update the S** set to contain the most specific set of descriptions in the version space **that cover the example** and the current example of the S set. i.e
- generalize the elements of S as little as possible so that they cover the new training example.

– If it is a negative example

- first **remove from S** any descriptions that cover the example.
- Then, **update the G** set to contain the most general set of descriptions in the version space that **do not cover the example**.
- i.e, specialize the element of G as little as possible so that negative example is no longer covered by any element of G

4. if S and G are both singleton sets, then if they are identical, output their value and halt.
 - If they are both singleton sets but they are different then the training cases were inconsistent.
 - Output this result and halt. Otherwise go to step 3

Learn “Japanese Economy Car” form examples

(1)

Origin: Japan
Manufacturer: Honda
Color: Blue
Decade: 1980
Type : Economy

(+)

- S and G both start out as singleton sets:
- $G \leq$ null description
- $S \leq$ first positive training example
- $G = \{(x_1, x_2, x_3, x_4, x_5)\}$
- $S = \{(\text{Japan}, \text{Honda}, \text{Blue}, 1980, \text{Economy})\}$

(2)

Origin: Japan
 Manufacturer: Toyota
 Color: Green
 Decade: 1970
 Type : Sports

(-)

$G = \{(\cancel{x1}, \cancel{x2}, \cancel{x3}, \cancel{x4}, x5)\}$
 $S = \{(\text{Japan}, \text{Honda}, \text{Blue}, 1980, \text{Economy})\}$

- G set must be specialized in such a way that the negative example is no longer in version space
- $G = \{ (x1, \text{Honda}, x3, x4, x5), (x1, x2, \text{Blue}, x4, x5), (x1, x2, x3, 1980, x5), (x1, x2, x3, x4, \text{Economy}) \}$
- The set **S** is **unaffected** by the negative example.

(3)

Origin: Japan
 Manufacturer: Toyota
 Color: Blue
 Decade: 1990
 Type : Economy

(+)

~~$G = \{ (x1, Honda, x3, x4, x5), (x1, x2, Blue, x4, x5), (x1, x2, x3, 1980, x5), (x1, x2, x3, x4, Economy) \}$~~

- $G = \{ (x1, x2, Blue, x4, x5), (x1, x2, x3, x4, Economy) \}$
- now generalize the S set to include the new example.
- This involves replacing constants with variables
- $S = \{ (Japan, x2, Blue, x4, Economy) \}$
- $S \Rightarrow$ Japanese blue economy car
- $G \Rightarrow$ either blue car or economy car.

(4)

Origin: USA
Manufacturer: Chrysler
Color: Red
Decade: 1980
Type : Economy

(-)

$G = \{ (x1, x2, \text{Blue}, x4, x5), (x1, x2, x3, x4, \text{Economy}) \}$
 $S = \{ (\text{Japan}, x2, \text{Blue}, x4, \text{Economy}) \}$

- The set S is unaffected, but the G set must be specialized to avoid covering the new example
- $G = \{ (\text{Japan}, x2, \text{Blue}, x4, x5), (\text{Japan}, x2, x3, x4, \text{Economy}) \}$

(5)

Origin: Japan
Manufacturer: Honda
Color: White
Decade: 1980
Type : Economy

(+)

- $G = \{(\text{Japan}, x_2, x_3, x_4, \text{Economy})\}$
- $S = \{(\text{Japan}, x_2, x_3, x_4, \text{Economy})\}$
- G and S are both singletons so the algorithm has converged on target concept. No more examples are needed.

Backpropagation learning

- start at the output layer and propagate error backwards through the hidden layers

A neural network propagates the signal of the input data forward through its parameters towards the moment of decision, and then backpropagates information about the error through the network so that it can alter the parameters one step at a time.

Example



- You could compare a neural network to a large piece of **artillery** that is attempting to strike a distant object with a shell.
- When the neural network makes a guess about an instance of data, it fires, a cloud of dust rises on the horizon, and the gunner tries to make out where the shell struck, and how far it was from the target.
- That distance from the target is the **measure of error**.
- The measure of error is then applied to the angle of and direction of the gun (parameters), before it takes another shot.

- Backpropagation takes the error associated with a wrong guess by a neural network, and uses that error to adjust the neural network's parameters in the direction of less error.
- How does it know the direction of less error?

- A gradient is a slope whose angle we can measure.
- Like all slopes, it can be expressed as a relationship between two variables: “y over x”, or rise over run.
- In this case, the y is the error produced by the neural network, and x is the parameter of the neural network.
- The parameter has a relationship to the error, and by changing the parameter, we can increase or decrease the error.
- So the gradient tells us the change we can expect in y with regard to x.

- To obtain this information, we must use differential calculus, which enables us to measure instantaneous rates of change, which in this case is the **tangent of a changing slope** expressed the relationship of the parameter to the neural network's error.
- Obviously, a neural network has many parameters, so what we're really measuring are the **partial derivatives** of **each parameter's contribution** to the **total change in error**.


- neural networks have parameters that process the input data sequentially, one after another.
- Therefore, **backpropagation** establishes the relationship between the neural network's error and the parameters of the net's last layer;
- then it establishes the relationship between the parameters of the neural net's last layer those the parameters of the second-to-last layer, and so forth,
- in an application of the **chain rule of calculus**.

MultiLayer Network

- Take relatively amorphous mass of neuronlike elements and teach it to perform useful task
- we need it to be fast and resistant to damage
- we like it to generalize from input it see
- we would like to build the neural mass on very large scale
- and would like them to be able to learn efficiently

What can Multilayer network Compute

- ANYTHING
- given a set of inputs we can use summation threshold units as simple AND, OR, NOT gates
 - by appropriately setting threshold and connection weights
- we can build any arbitrary combinational circuit out of those basic logical units
- if we are allowed to use feedback loops we can build a general purpose computer with them

- 
- the multilayer network must learn their own representation because programming them by hand is impossible
 - Back propogation is a step in that direction

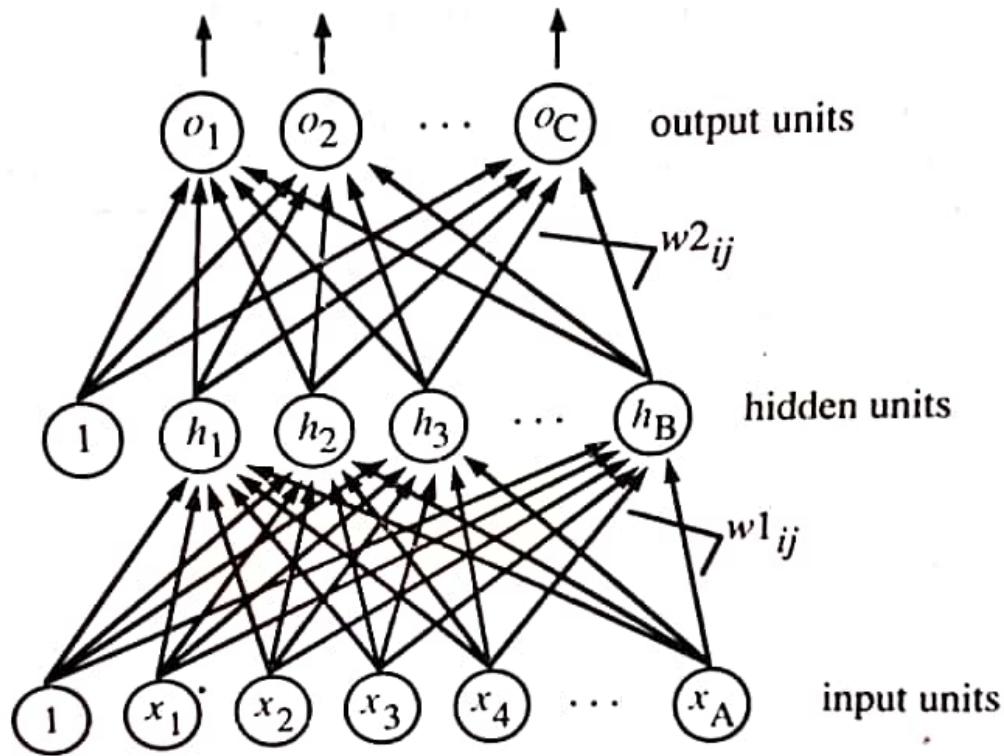
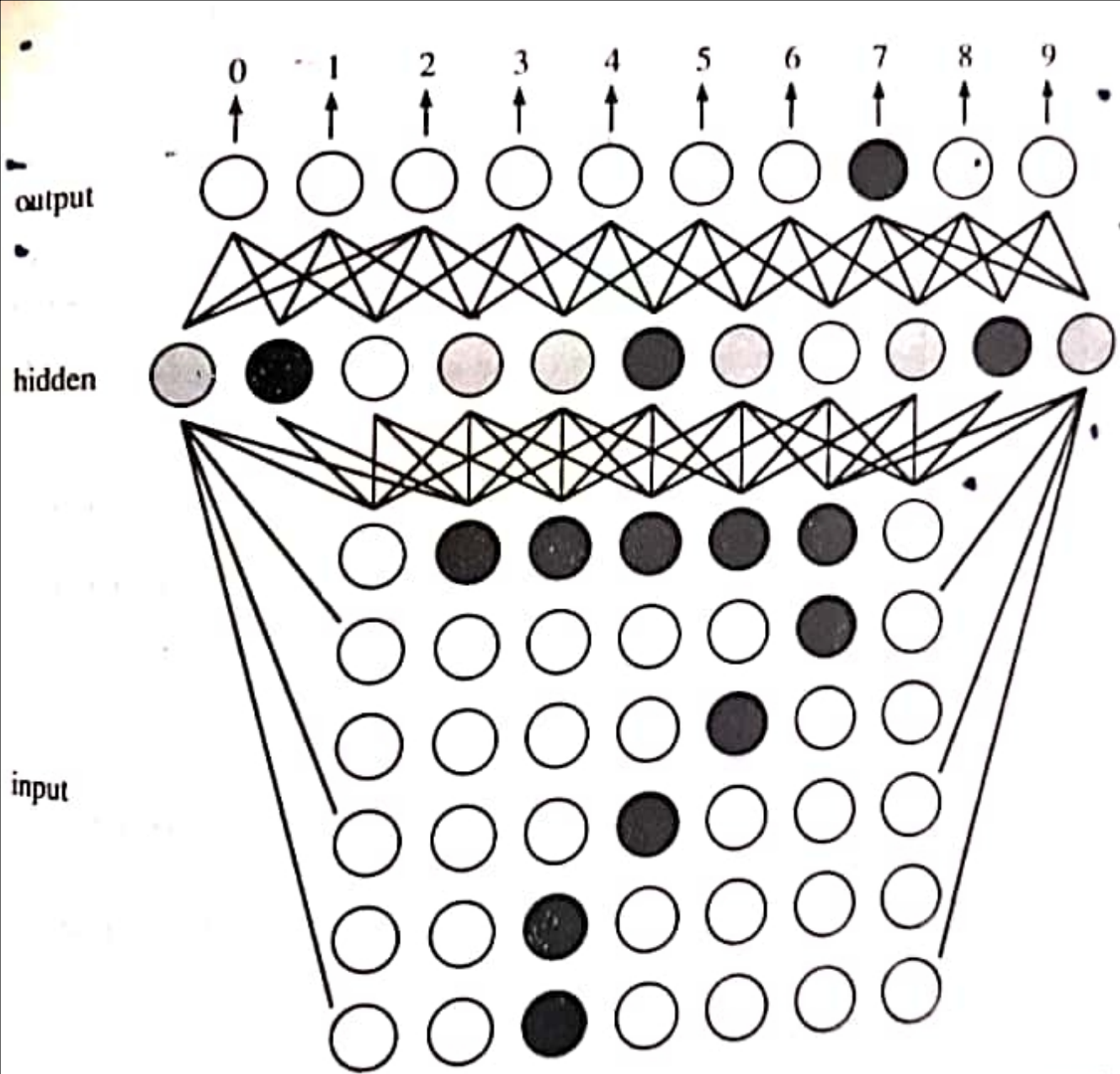


Figure 18.14: A Multilayer Network


- x_i = unit activation levels of input
- h_i = hidden units
- o_i = output units
- $w1_{ij}$ = weight on connection between input and hidden layer
- $w2_{ij}$ = weight on connection between hidden and output layer

- Each unit in one layer is connected in forward direction to every unit in next layer
- Activation flow from input layer through the hidden layer then on to output layer
- the knowledge of network is encoded in the weights on connection between units
- Backpropagation algorithm performs simpler computation
 - because activation flow only in one direction there is no need for iterative relaxation process
- Activation levels of units in output layer determine output of network
- the existence of hidden units allow the network to develop complex feature detectors, or internal representations



Multilayer Network to
learn Classify
Handwritten Digits

Figure 18.15: Using a Multilayer Network to Learn to Classify Handwritten Digits

- 
- The two dimensional grid containing the numeral “7” form the input layer
 - A single hidden unit might be strongly activated by a horizontal line in the input, or perhaps a diagonal
 - behaviour of these hidden unit is automatically learned not preprogrammed

- The hope in problems like handwritten recognition is that the neural network will not only learn to classify the input it is trained on but that it will generalize and be able to classify input it has not yet seen

- The approach taken by the backpropagation algorithm is to start at the output layer and **propagate error backwards through the hidden layers**
- All the information needed to update the weights on a neuron was local to that neuron, except for the amount of error
- For **output nodes**, this is easily computed as the **difference between the desired and actual output values**.
- For nodes in the **hidden layers**, it is considerably **more difficult to determine the error** for which a node is responsible

- The activation function for backpropagation is usually the logistic function:

$$f(\text{net}) = 1/(1 + e^{-\lambda * \text{net}}), \text{ where } \text{net} = \sum x_i w_i$$

- This function is used for four reasons:
 1. First, it has the sigmoid shape.
 2. Second, as a continuous function, it has a derivative everywhere.
 3. Third, since the value of the derivative is greatest where the sigmoidal function is changing most rapidly, the assignment of the most error is attributed to those nodes whose activation was least certain.

4. Finally, the derivative is easily computed by a subtraction and multiplication:

$$f'(net) = (1/(1 + e^{-\lambda * net})) = \lambda(f(net) * (1 - f(net))).$$

- Backpropagation training uses the generalized delta rule.
- This uses the gradient descent approach
- For nodes in the hidden layer we look at their contribution to the error at the output layer.


The generalized delta rule is a mathematically derived formula used to determine how to update a neural network during a (back propagation) training step. A neural network learns a function that maps an input to an output based on given example pairs of inputs and outputs

Social and emergent models of learning

Social and emergent models of learning

- Considers learning algorithms patterned after the processes underlying evolution: shaping a population of individuals through the survival of its most fit members.
- Emergent models of learning simulate nature's most elegant and powerful form of **adaptation**: the evolution of plant and animal life forms.
- Through this simple process of introducing variations into successive generations and **selectively eliminating less fit individuals**, adaptations of increasing capability and diversity emerge in a population.
- Evolution and emergence occur in populations of embodied individuals, whose actions affect others and that, in turn, are affected by others.

- Genetic algorithm has three distinct stages:
 - first, the individual potential solutions of the problem domain are **encoded** into representations that support the necessary variation and selection operations; often, these representations are as simple as bit strings.
 - In the second stage, **mating and mutation algorithms**, analogous to the sexual activity of biological life forms, produce a new generation of individuals that recombine features of their parents.
 - Finally, a **fitness function judges** which individuals are the “best” life forms, that is, most appropriate for the eventual solution of the problem.

- 
- These individuals are favored in survival and reproduction, shaping the next generation of potential solutions.
 - Eventually, a generation of individuals will be interpreted back to the original problem domain as solutions for the problem.

- They view **learning as a competition** among a population of evolving candidate problem solutions.
- A “fitness” function evaluates each solution to decide whether it will contribute to the next generation of solutions.
- Then, through operations analogous to gene transfer in sexual reproduction, the algorithm **creates a new population of candidate solutions**.

General form of genetic algorithm

Let $P(t)$ define a population of candidate solutions, x_i^t , at time t :

$$P(t) = \{x_1^t, x_2^t, \dots, x_n^t\}$$

begin

 set time $t := 0$;

 initialize the population $P(t)$;

 while the termination condition is not met do

 begin

 evaluate fitness of each member of the population $P(t)$;

 select members from population $P(t)$ based on fitness;

 produce the offspring of these pairs using genetic operators;

 replace, based on fitness, candidates of $P(t)$, with these offspring;

 set time $t := t + 1$

 end

end.

- The genetic algorithm initializes $P(0)$ to a **population of candidate patterns**.
 - Typically, initial populations are selected randomly.
- Evaluation of candidate solutions assumes a fitness function, $f(x_i^t)$, that returns a **measure of the candidate's fitness at time t** .
- A common measure of a candidate's fitness tests it on a set of training instances and returns the **percentage of correct classifications**.

- Using such a fitness function, an evaluation assigns each candidate solution the value:

$$f(x_i^t)/m(P, t)$$

- where $m(P, t)$ is the average fitness over all members of the population.
- It is also common for the fitness measure to change $f(x_i^t)$ across time periods, thus fitness could be a function of the stage of the overall problem solution .

- After evaluating each candidate, the **algorithm selects pairs for recombination**.
- Recombination uses **genetic operators** to produce new solutions that **combine components of their parents**.
- As with natural evolution, the **fitness** of a candidate determines the **extent to which it reproduces**,
- with those candidates having the **highest evaluations** being given a greater probability of reproducing.

- Selection is often probabilistic, where **weaker members** are given a smaller likelihood of reproducing, but are **not eliminated outright**.
- That **some less fit candidates survive** is important since they can still contain some essential component of a solution, for instance part of a bit pattern, and reproduction may extract this component.

- There are a number of genetic operators that produce offspring having features of their parents; the most common of these is **crossover**.
- Crossover takes **two candidate solutions and divides them, swapping components to produce two new candidates**.
- The operator **splits them in the middle and forms two children whose initial segment comes from one parent and whose tail comes from the other**.
- Note that splitting the candidate solution in the middle is an arbitrary choice.
- This **split may be at any point** in the representation, and indeed, this splitting point may be randomly adjusted or changed during the solution process

Input Bit Strings:

1	1	#	0	:	1	0	1	#	#	1	1	0	:	#	0	#	1
			⋮									⋮					
			⋮									⋮					
			⋮									⋮					

Resulting New Strings:

1	1	#	0	#	0	#	1	#	1	1	0	1	0	1	#
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Figure 12.1 Use of crossover on two bit strings of length eight. # is don't care.

- Mutation is another important genetic operator.
- Mutation takes a single candidate and randomly changes some aspect of it.
- For example, mutation may randomly select a bit in the pattern and change it, switching a 1 to a 0 or #.
- Mutation is important in that the initial population may exclude an essential component of a solution.

- In our example,
 - if no member of the initial population has a 1 in the first position, then crossover, because it preserves the first four bits of the parent to be the first four bits of the child, cannot produce an offspring that does.
 - Mutation would be needed to change the values of these bits.
- Other genetic operators, for example inversion, could also accomplish this task
- The genetic algorithm continues until some termination requirement is met, such as having one or more candidate solutions whose fitness exceeds some threshold.

- Early research in genetic algorithms focused almost exclusively on low-level representations, such as strings of {0, 1, #}.
- Holland (1986) developed a problem-solving architecture called classifier systems that **applies genetic learning to rules in a production system**.
- A classifier system includes the familiar elements of a production system:
 - production rules (here called classifiers)
 - working memory
 - input sensors (or decoders)
 - and outputs (or effectors).

- Unusual features of a classifier system include the use of competitive bidding for conflict resolution, genetic algorithms for learning, and the bucket brigade algorithm to assign credit and blame to rules during learning.
- Feedback from the outside environment provides a means of evaluating the fitness of candidate classifiers, as required in genetic learning.

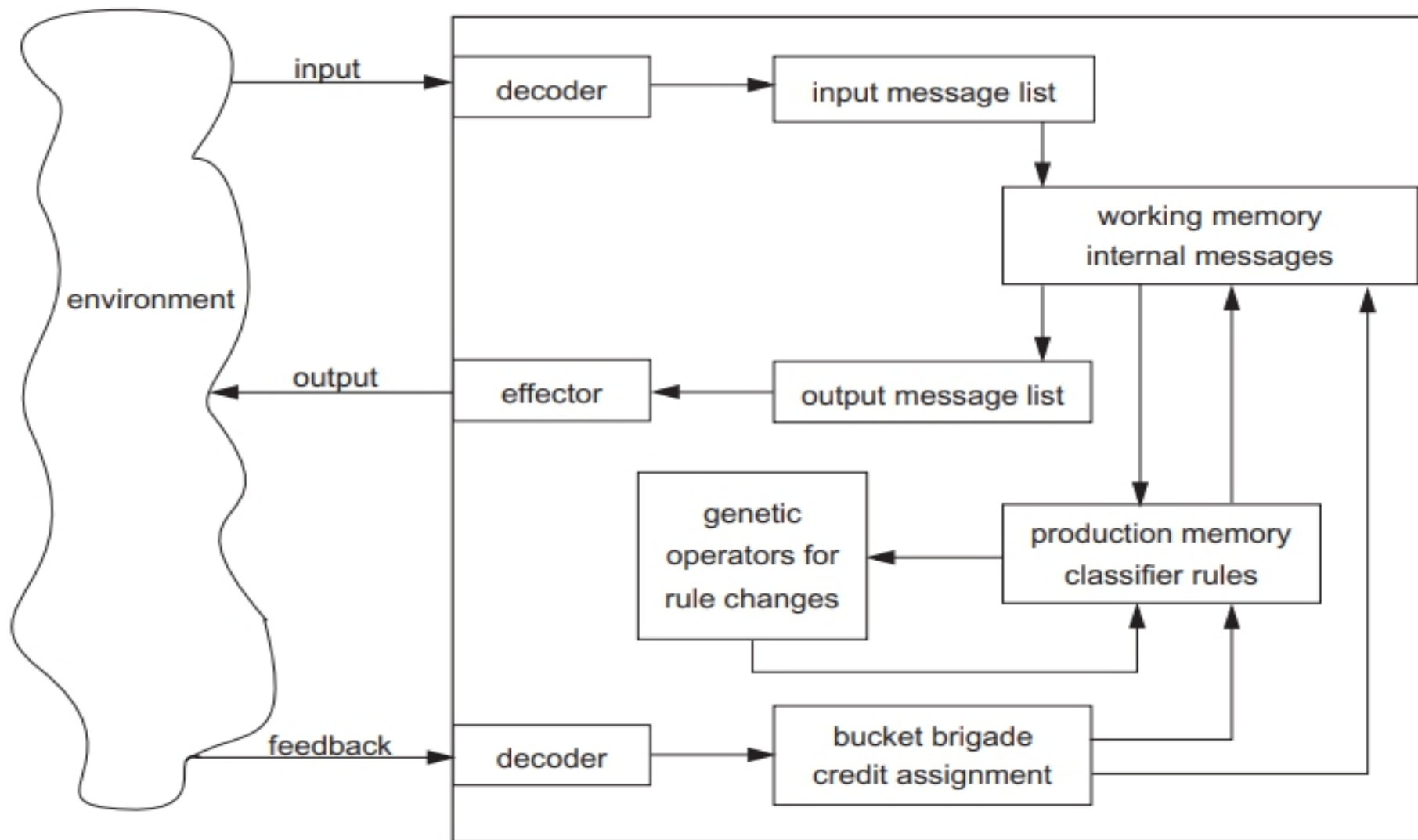


Figure 12.3 A classifier system interacting with the environment, adapted from Holland (1986).

- The classifier system has the following major components
 - **Detectors of input messages** from the environment.
 - **Detectors of feedback** messages from the environment.
 - **Effectors translating results** of rule applications back to the environment.
 - A **production rule set** made up of a population of classifiers. Each classifier has an associated **fitness measure**.
 - A **working memory** for the classifier rules. This memory integrates the results of production rule firing with input information.
 - A **set of genetic operators** for production rule modification.
 - A **system for giving credit** to rules involved in producing successful actions.

- In problem solving, the classifier performs as a traditional production system.
- The **environment sends a message**, perhaps a move in a game, to the classifier system's detectors.
- This **event is decoded and placed** as a pattern on the internal message list, the working memory for the production system.
- These messages, in the **normal action** of data-driven production system, **match** the condition patterns of the classifier rules.
- The selection of the **“strongest activated classifiers”** is determined by a **bidding scheme**

- a **bid** is a function of both the **accumulated fitness** of the classifier and the **quality of the match** between the input stimulus and its condition pattern.
- The classifiers with the **closest match add messages** (the action of the fired rules) to working memory.
- The **revised message list** may send messages to the **effectors** which act upon the environment or activate new classifier rules as the production system processing continues.

- Classifier systems implement a form of **reinforcement learning**
- Based on **feedback** from a teacher or fitness evaluation function, the **learner computes the fitness** of a population of candidate rules and **adapts this population** using a variation of genetic learning.

Classifier systems learn in two ways.

- First, there is a **reward system** that adjusts the fitness measures of the classifier rules, **rewarding successful rule firings and penalizing errors**.
 - The **credit assignment algorithm** passes part of the reward or penalty back to any classifier rules that have contributed to the final rule firing.
 - This distribution of differential rewards across interacting classifiers, as well as those that enabled their firing, is often implemented in a bucket brigade algorithm.
 - The **bucket brigade algorithm** addresses the problem of **assigning credit or blame** in situations where the system's output may be the product of a sequence of rule firings.
 - The bucket brigade algorithm allocates both **credit and blame across a sequence of rule applications** according to measures of each rule's contribution to the final conclusion.

- The **second form** of learning **modifies the rules themselves** using **genetic operators** such as **mutation and crossover**.
- This allows the most **successful rules to survive** and **combine to make new classifiers**, while unsuccessful rule classifiers disappear.

- Each classifier rule consists of three components:
 1. First, the **rule's condition matches data in the working memory** in the typical production system sense. In learning, genetic operators can modify both the conditions and the actions of the production rules.
 2. The second component of the rule, **the action, can have the effect of changing the internal message list** (the production memory).
 3. Finally, each rule has a **fitness measure**. This parameter is changed, both by successful as well as by unsuccessful activity. This measure is originally assigned to each rule on its creation by the genetic operators; for example, it may be set as the average fitness of its two parents.

GENETIC PROGRAMMING

- Koza has suggested that a **successful computer program might evolve through successive applications of genetic operators.**
- In genetic programming, the structures being adapted are **hierarchically organized segments of computer programs.**
- The learning algorithm maintains a population of candidate programs.
- The fitness of a program will be measured by its ability to solve a set of tasks
- programs are modified by applying crossover and mutation to program subtrees.
- **Genetic programming searches a space of computer programs of varying size and complexity**
- the search space is the space of all possible computer programs composed of functions and terminal symbols appropriate to the problem domain.
- **this search is random, largely blind and effective**

- Genetic programming **starts with an initial population** of randomly generated programs made up of **appropriate program pieces**.
- These pieces, suitable for a **problem domain**, may consist of standard arithmetic operations, other related programming operations, and mathematical functions, as well as logical and domain-specific functions.
- Program components include data items of the usual types: Boolean, integer, floating point, vector, symbolic, or multiple-valued.

- After initialization, thousands of computer programs are **genetically bred**.
- The production of new programs comes with **application of genetic operators**.
- **Crossover, mutation, and other breeding algorithms** must be customized for the production of computer programs.
- The **fitness** of each new program is then determined by seeing how well it performs in a particular problem environment.
- The **nature of the fitness measure will vary according** to the problem domain.
- Any program that does well on this fitness task will survive to help produce the children of the next generation.

- To summarize, genetic programming includes six components, many very similar to the requirements for GAs:
 1. A set of **structures** that **undergo transformation** by genetic operators.
 2. A set of **initial structures** suited to a problem domain.
 3. A **fitness measure**, again domain dependent, to evaluate structures.
 4. A set of **genetic operators** to transform structures.
 5. **Parameters and state descriptions** that describe members of each generation.
 6. A set of **termination conditions**.

- Genetic programming **manipulates hierarchically organized** program modules
- Lisp was (and still remains) the primary representation for the programming language components:
- Koza represents program segments as Lisp symbol expressions, or **s-expressions**.
- **Genetic operators manipulate s-expressions.**
- operators map tree structures of s-expressions, (Lisp program segments), into new trees, (new Lisp program segments).

- Genetic programming will **construct useful programs**, given that the atomic pieces and evaluable predicates of the problem domain are available
- first **analyze what terminals** are required for units in its solution as well as what functions are necessary to produce these terminals.

- To initialize the structures for adaptation by genetic operators, we must create two sets:
 - **F the set of functions** and
 - **T the set of terminal values** required for the domain.
- F can be as simple as $\{+, *, -, /\}$ or may require more complex functions such as $\sin(X)$, $\cos(X)$, or functions for matrix operations.
- T may be the integers, reals, matrices, or more complex expressions. The symbols in T must be closed under the functions defined in F.

- Next, a **population of initial “programs”** is generated by **randomly** selecting elements from the **union of sets F and T**.
- For example, if we begin by selecting an element of T, we have a degenerate tree of a single root node.
- when we start with an element from F, say +, we get a root node of a tree with two potential children.
- Suppose the initializer next selects * (with two potential children) from F, as the first child, and then terminal 6 from T as the second child.
- Another random selection might yield the terminal 8, and then the function + from F. Assume it concludes by selecting 5 and 7 from T

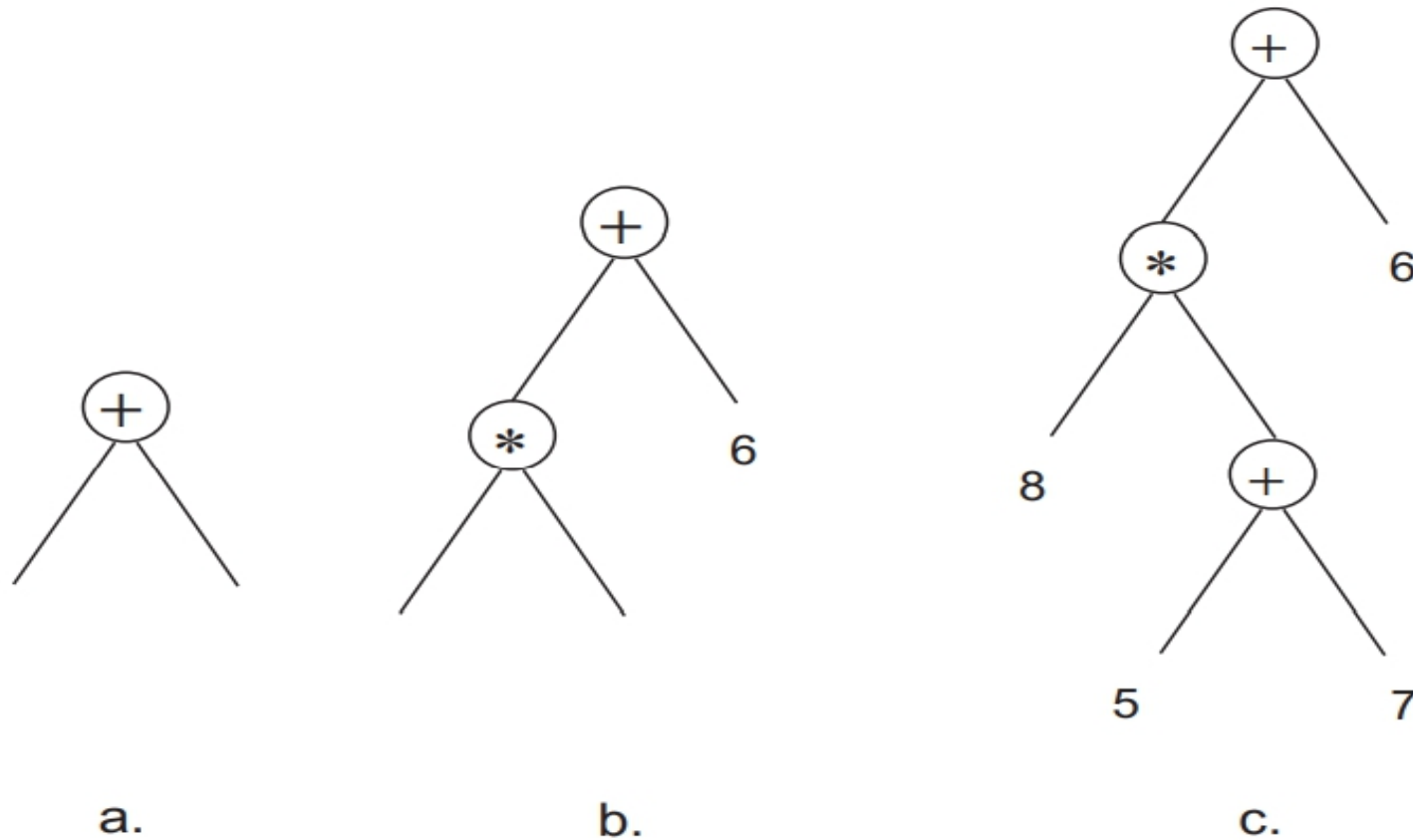


Figure 12.4 The random generation of a program to initialize.
The circled nodes are from the set of functions.

- The program we have randomly produced is represented in Figure 12.4.
- Figure 12.4a gives the tree after the first selection of +,
- 15.4b after selecting the terminal 6, and
- 15.4c the final program.
- A population of similar programs is created to initialize the genetic programming process.
- **Sets of constraints**, such as the maximum depth for programs to evolve, can help **prune this population**.


- Next, we require a **fitness measure** for populations of programs.
- The fitness measure is **problem domain dependent** and usually consists of a set of tasks the evolved programs must address.
- The fitness measure itself is a function of how well each program does on these tasks.
- A **simple raw fitness score** would add the differences between what the program produced and the results that the actual task from the problem domain required.
- Thus, raw fitness could be seen as the sum of errors across a set of tasks.

- Other fitness measures are also possible.
- Normalized fitness divides raw fitness by the total sum of possible errors and thus puts all fitness measures within the range of 0 to 1.
- Normalization can have an advantage when trying to select from a large population of programs.
- A fitness measure can also include an adjustment for the size of the program, for example, to reward smaller programs.

- Genetic operators on programs include both **transformations on a tree** itself as well as the **exchange of structures between trees**.
- Koza (1992) describes the primary transformations as **reproduction and crossover**.
- Reproduction simply **selects programs from the present generation and copies them** (unchanged) into the next generation.
- **Crossover exchanges subtrees between the trees representing two programs.**

- There are a number of secondary, and much less used, genetic transforms of program trees.
- These include **mutation**, which simply **introduces random changes in the structures of a program**.
- For example, replacing a terminal value with another value or a function subtree.
- The permutation transform, similar to the inversion operator on strings, also works on single programs, exchanging terminal symbols, or subtrees, for example.

- The state of the solution is reflected by the current generation of programs.
- There is no record keeping for backtrack or any other method for skipping around the fitness landscape.
- The genetic programming paradigm parallels nature in that the evolution of new programs is a continuing process.
- Nonetheless, lacking infinite time and computation, termination conditions are set.
- These are usually a function both of program fitness and computational resources.



MODULE 5
END