

# CE213 Artificial Intelligence – Lecture 12

## Decision Tree Induction: Part 2

An example of decision tree induction using information gain to select best attributes

Knowledge discovery by machine learning

Some issues in decision tree induction

# Pseudocode for Decision Tree Induction Revisited

```
METHOD buildDecTree(samples,atts)
    Create node N if necessary; //starting as a node, ending as a tree
    IF samples are all in same class
        THEN RETURN N labelled with that class;
    IF atts is empty
        THEN RETURN N labelled with modal class 1;
    bestAtt = chooseBestAtt(samples,atts); 
    label N with bestAtt;

    FOR each value  $a_i$  of bestAtt //each branch from node N
         $s_i$  = subset of samples with bestAtt =  $a_i$ ;
        IF  $s_i$  is not empty
            THEN
                newAtts = atts - bestAtt;
                subtree = buildDecTree( $s_i$ ,newAtts); //recursive
                attach subtree as child of N;
            ELSE
                Create leaf node L;
                Label L with modal class;
                attach L as child of N;
    RETURN N;
```

{Note 1: Model class is the class of the group with the maximum number of samples or highest frequency)

# An Example of Decision Tree Induction

Suppose we have a training sample dataset derived from weather records, containing **four attributes**:

Attributes	Possible Values
Temperature	Warm, Cool
Cloud Cover	Overcast, Cloudy, Clear
Wind	Windy, Calm
<i>Precipitation</i>	<i>Rain, Dry</i>

We want to build a decision tree based on this sample dataset, which can predict precipitation from the other **three** attributes (Attribute Precipitation is used as class in this example).

## An Example ... (2)

The training sample dataset is as follows (8 samples):

Temperature	Cloud Cover	Wind	Precipitation (class)
[ warm,	overcast,	windy;	rain ]
[ cool,	overcast,	calm;	dry ]
[ cool,	cloudy,	windy;	rain ]
[ warm,	clear,	windy;	dry ]
[ cool,	clear,	windy;	dry ]
[ cool,	overcast,	windy;	rain ]
[ cool,	clear,	calm;	dry ]
[ warm,	overcast,	calm;	dry ]

What is the modal class of this dataset?

Can you guess which attribute would be the best for predicting precipitation?

(Note: For practical applications, there could be thousands or more samples and hundreds or more attributes, but the decision tree induction procedure would be the same except for more recursive loops, as described in the pseudo code, and require more computational load.)

# An Example ... (3)

**Initial information (or uncertainty) about the class (Precipitation):**

First we consider the initial information:

$$p(\text{rain}) = 3/8; p(\text{dry}) = 5/8$$

So for the **whole set**,  $\text{Inf} = -(3/8)\log_2(3/8)-(5/8)\log_2(5/8) = 0.954 \text{ bits}$

**Choosing the best attribute:**

Next we must choose the best attribute as the root node of the decision tree and then build up its branches.

There are three to choose from:

Temperature

Cloud Cover

Wind

# An Example ... (4)

Standard procedure for calculating information gain from an attribute

## Information Gain from attribute Temperature

**Cool** samples: 5 out of 8

There are 5 of these; 2 rain and 3 dry. (rain and dry are precipitation values)

So for this **subset**,  $p(\text{rain}) = 2/5$  and  $p(\text{dry}) = 3/5$

Hence,  $\text{Inf}_{\text{cool}} = -(2/5)\times\log_2(2/5)-(3/5)\times\log_2(3/5) = 0.971$  bits

**Warm** samples: 3 out of 8

There are 3 of these; 1 rain and 2 dry.

So for this **subset**,  $p(\text{rain}) = 1/3$  and  $p(\text{dry}) = 2/3$

Hence,  $\text{Inf}_{\text{warm}} = -(1/3)\times\log_2(1/3)-(2/3)\times\log_2(2/3) = 0.918$  bits

**Average Information about the class given value of Temperature:**

$$(5/8)\times\text{Inf}_{\text{cool}} + (3/8)\times\text{Inf}_{\text{warm}} = 0.625\times0.971+0.375\times0.918 = 0.951 \text{ bits}$$

Hence, Information Gain from Temperature is

$$\begin{aligned} &\text{Initial Information} - \text{Average Information given value of Temperature} \\ &= 0.954 - 0.951 = 0.003 \text{ bits} \quad (\text{Very small}) \end{aligned}$$

# An Example ... (5)

## Information Gain from attribute Cloud Cover

Following a similar calculation procedure gives 0.5 bits as average information about the class given value of Cloud Cover.

Hence, Information Gain from Cloud Cover is

$$\begin{aligned} & \text{Initial Information} - \text{Average Information given value of Cloud Cover} \\ & = 0.954 - 0.5 = 0.454 \text{ bits. (Large)} \end{aligned}$$

## Information Gain from attribute Wind

Following a similar calculation procedure gives 0.607 bits as average information about the class given value of Wind.

Hence, Information Gain from Wind is

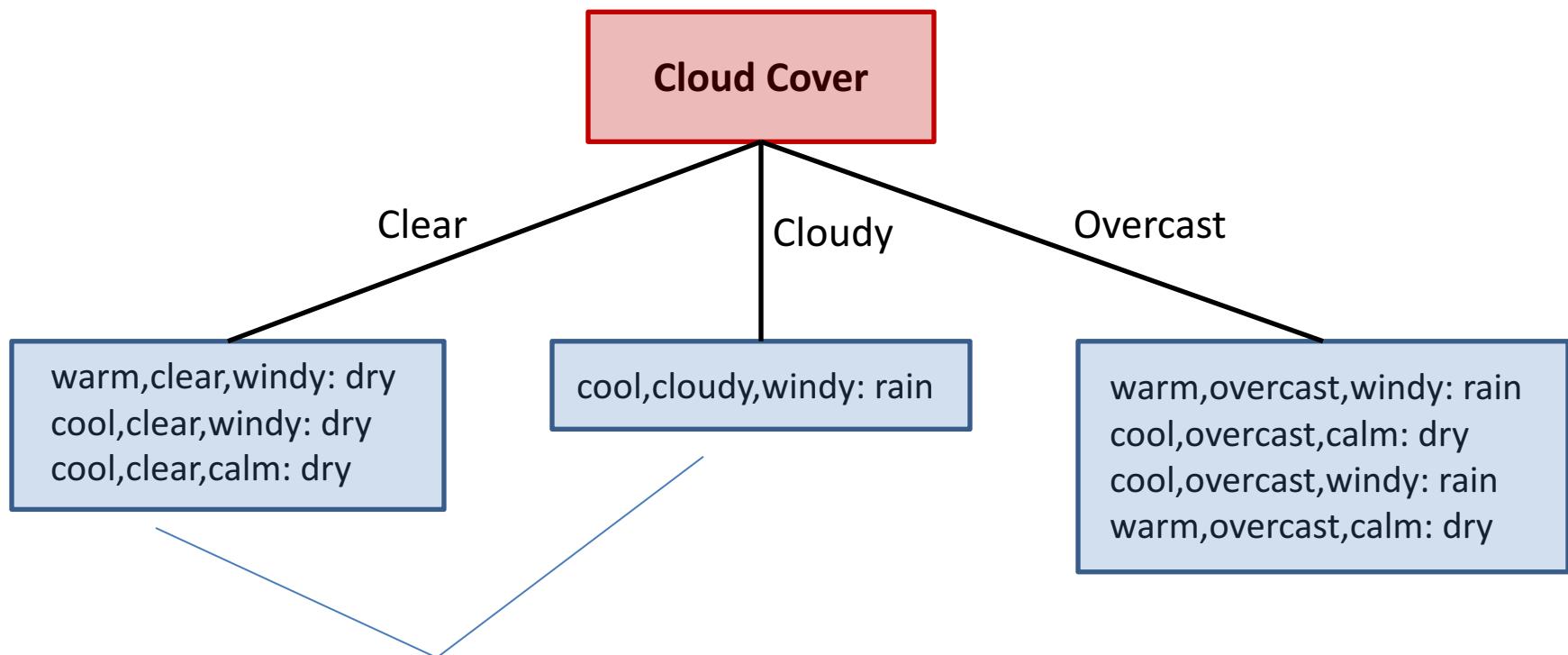
$$\begin{aligned} & \text{Initial Information} - \text{Average Information given value of Wind} \\ & = 0.954 - 0.607 = 0.347 \text{ bits. (Quite large)} \end{aligned}$$

## Conclusion

Cloud Cover gives the greatest information gain and is therefore the best attribute for predicting precipitation.

## An Example ... (6)

Building up the decision tree with the first best attribute as root node:



In each subset, samples are  
all in same class

# Pseudocode for Decision Tree Induction Revisited

```
METHOD buildDecTree(samples,atts)
    Create node N if necessary; //starting as a node, ending as a tree
    IF samples are all in same class
        THEN RETURN N labelled with that class; 
    IF atts is empty
        THEN RETURN N labelled with modal class;
    bestAtt = chooseBestAtt(samples,atts); 
    label N with bestAtt;

    FOR each value  $a_i$  of bestAtt //each branch from node N
         $s_i$  = subset of samples with bestAtt =  $a_i$ ;
        IF  $s_i$  is not empty
            THEN
                newAtts = atts - bestAtt;
                subtree = buildDecTree( $s_i$ ,newAtts); //recursive 
                attach subtree as child of N;
            ELSE
                Create leaf node L;
                Label L with modal class;
                attach L as child of N;
    RETURN N;
```

## An Example ... (7)

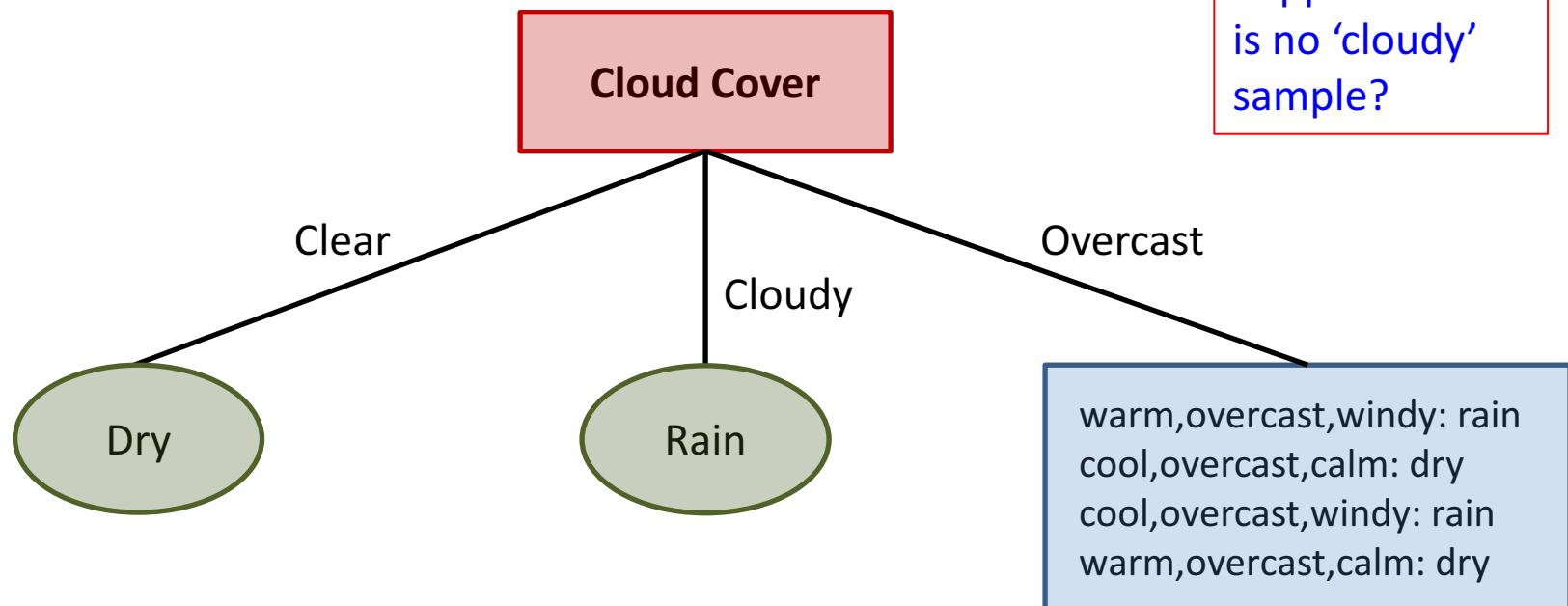
All the samples on the “Clear” branch or subset belong to the same class, i.e., dry, so no further expansion is needed.

It can be terminated with a leaf node labelled “dry” (see pseudo code)

Similarly, the single sample on the “Cloudy” branch or subset necessarily belongs to one class, i.e., rain.

It can be terminated with a leaf node labelled “rain”.

What would happen if there is no ‘cloudy’ sample?



## An Example ... (8)

**Extending the Overcast subtree (see pseudo code)**

The “Overcast” branch has both rain and dry samples.

So we must attempt to extend the tree from this node.

There are **4 samples in this subset**: 2 rain and 2 dry

So for this subset,  $p(\text{rain}) = p(\text{dry}) = 0.5$  and the initial information is 1 bit.

There are two remaining attributes: temperature and wind. Which is the best?

# An Example ... (9)

**Information Gain from attribute Temperature:**

**Cool** samples: 2 out of 4

There are 2 of these; 1 rain and 1 dry.

So for this **sub-subset**,  $p(\text{rain}) = 1/2$  and  $p(\text{dry}) = 1/2$

$$\text{Hence, } \text{Inf}_{\text{cool}} = -(1/2)\log_2(1/2)-(1/2)\log_2(1/2) = 1 \text{ bit}$$

**Warm** samples: 2 out of 4

There are also 2 of these; 1 rain and 1 dry.

So again for this **sub-subset**,  $p(\text{rain}) = 1/2$  and  $p(\text{dry}) = 1/2$

$$\text{Hence, } \text{Inf}_{\text{warm}} = -(1/2)\log_2(1/2)-(1/2)\log_2(1/2) = 1 \text{ bit}$$

**Average Information about the class given value of Temperature:**

$$(1/2)\times\text{Inf}_{\text{cool}} + (1/2)\times\text{Inf}_{\text{warm}} = 0.5 \times 1.0 + 0.5 \times 1.0 = 1 \text{ bit}$$

Hence, Information Gain from Temperature is zero!

(Does it make good sense? – examine the 4 samples)

# An Example ... (10)

**Information Gain from attribute Wind:**

**Windy** samples: 2 out of 4

There are 2 of these; 2 rain and 0 dry.

So for this **sub-subset**,  $p(\text{rain}) = 1$  and  $p(\text{dry}) = 0$

Hence,  $\text{Inf}_{\text{windy}} = -1 \times \log_2(1) - 0 \times \log_2(0) = 0$

$$0 \times \log_2(0)=0$$

**Calm** samples: 2 out of 4

There are also 2 of these; 0 rain and 2 dry.

So again  $\text{Inf}_{\text{calm}} = -1 \times \log_2(1) - 0 \times \log_2(0) = 0$

**Average Information about the class given value of Wind:**

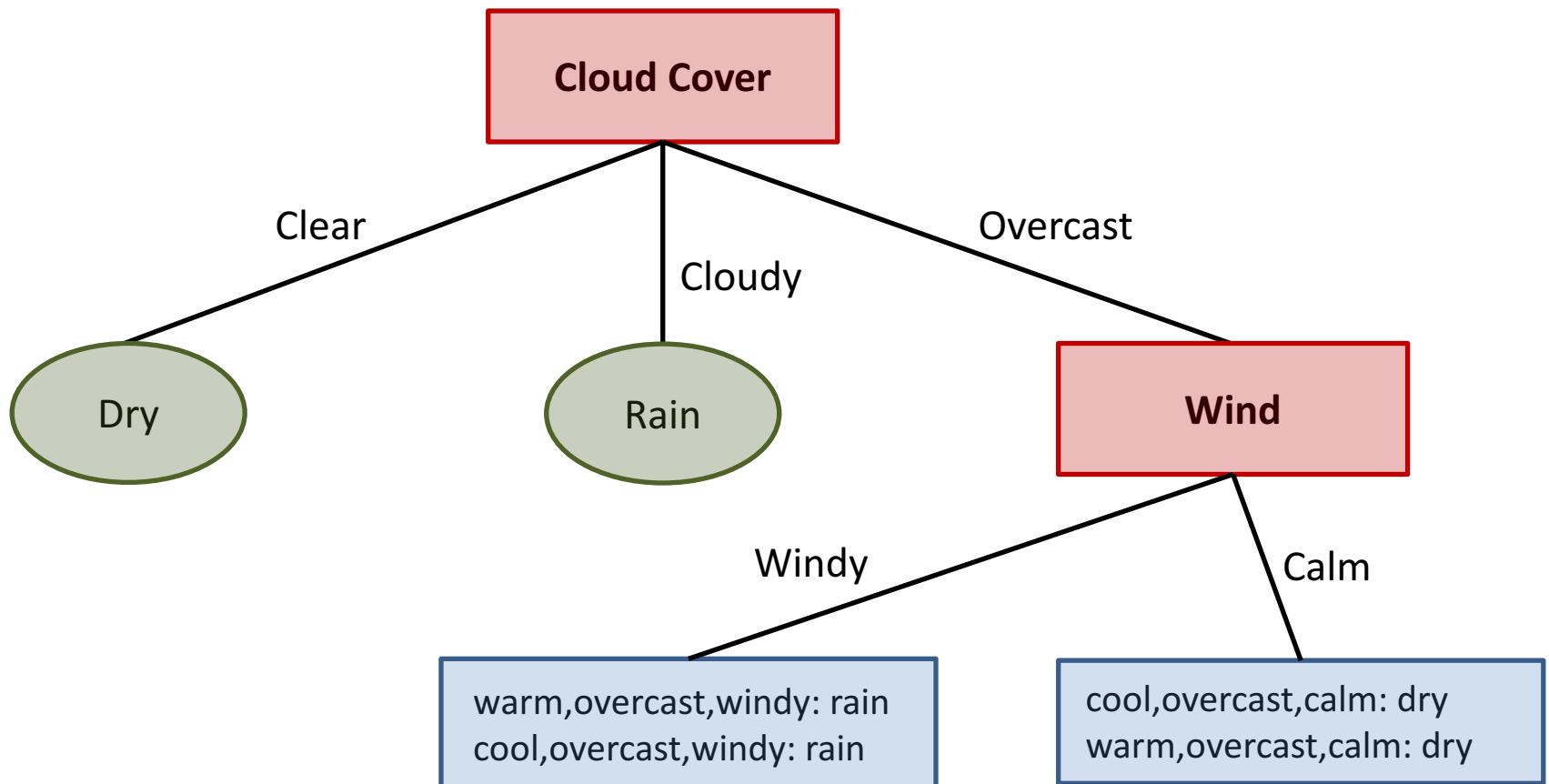
$$(1/2) \times \text{Inf}_{\text{windy}} + (1/2) \times \text{Inf}_{\text{calm}} = 0.5 \times 0 + 0.5 \times 0 = 0$$

Hence, Information Gain from Wind is 1.

Note: This reflects the fact that wind is a perfect predictor of precipitation for this sub-subset of samples.

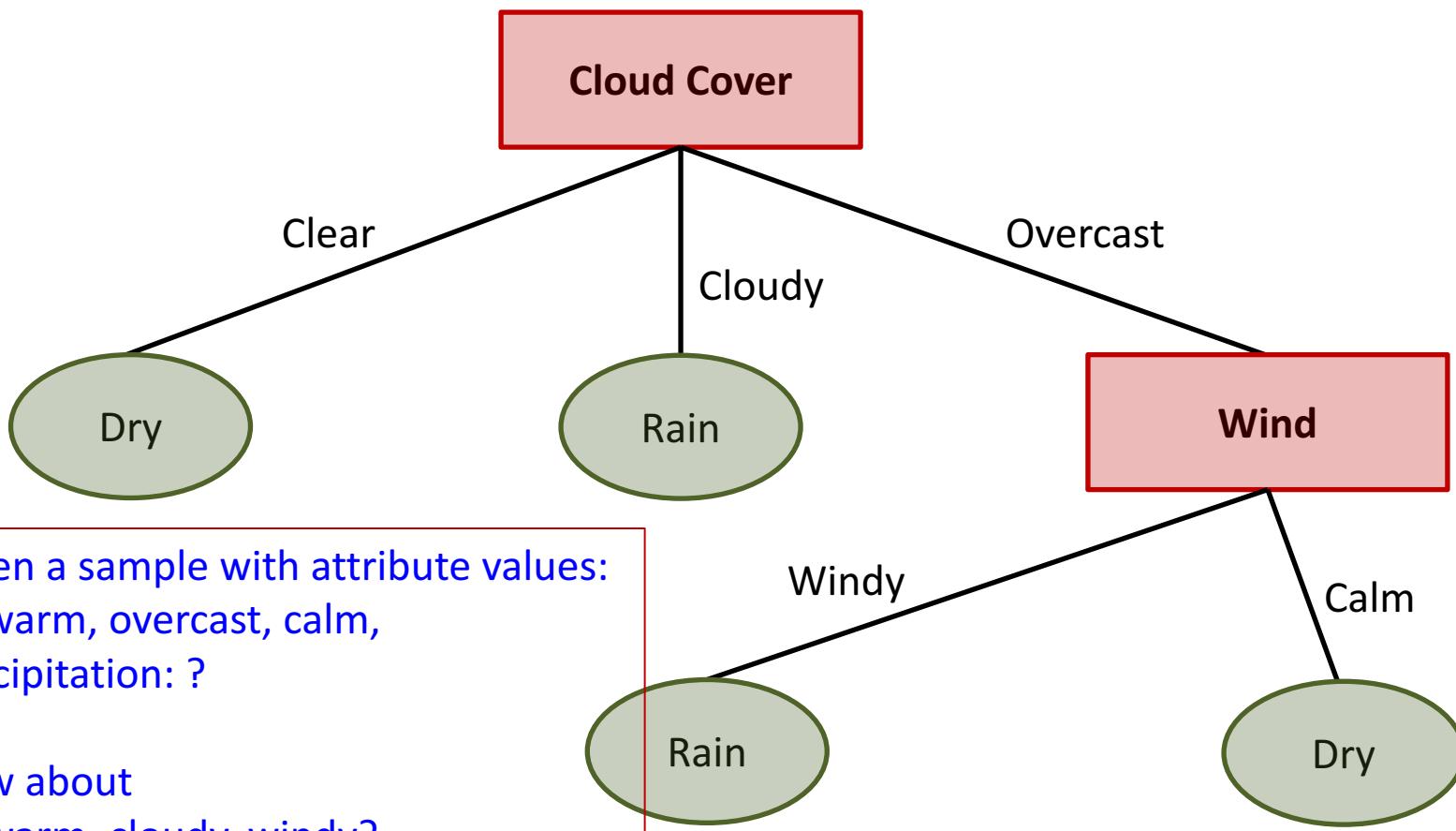
## An Example ... (11)

Obviously wind is the best attribute, so we can now extend the tree as follows:



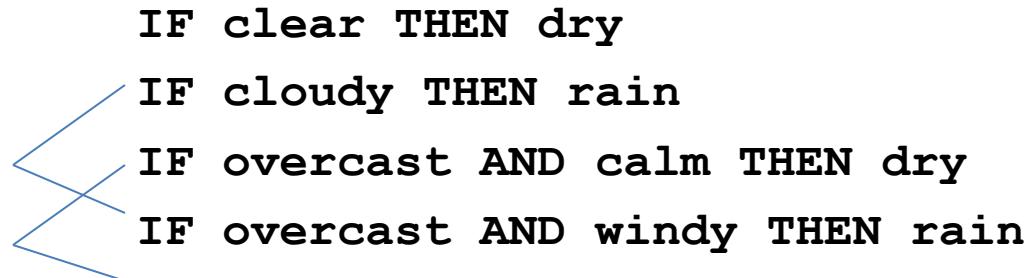
## An Example ... (12)

All the samples on both the new branches belong to the same class, so they can be terminated with appropriately labelled leaf nodes.



# From Decision Trees To Production Rules

Decision trees can easily be converted into sets of IF-THEN rules as follows:



(knowledge discovery)

Such rules are usually easier to understand than the corresponding decision tree.

Large trees produce large sets of rules.

It is often possible to simplify these rule sets considerably.

In some cases these simplified rule sets are more accurate than the original tree because they reduce the effect of overfitting – a topic we will discuss later.

Is it necessary to have conflict resolution strategies here?

Which rule interpreter is better, forward chaining or backward chaining?

# Some Issues in Decision Tree Induction

## 1) Inconsistent / Contradictory Training Data

It is possible (and not unusual) to arrive at a situation in which the samples associated with a leaf node belong to **more than one class**, because there are ***no more attributes*** available or useful to further subdivide the samples.

A simple method for handling this is to label the leaf node with the modal class. This means that some training samples will be misclassified by the induced decision tree.

# Some Issues in Decision Tree Induction (2)

## 2) Numeric Attributes

The number of possible attribute values can be very large, creating too many branches in the decision tree.

A simple solution is to partition the value into a small number of contiguous subranges and then treat the membership of each subrange as a **categorical variable**, e.g., small, medium, large.

# **Some Issues in Decision Tree Induction (3)**

## **3) Overfitting**

### **Question**

What would happen if a set of completely random data were used for a decision tree induction program?

### **Answer**

The program would build a decision tree.

If there were many variables and plenty of data, it could be a large tree.

### **Question**

Would the decision tree be any good as a classifier?

Would it do better than the simple strategy of always picking the modal class?

### **Answer**

No, because random dataset contains no useful information for classification.

Note that if a new set of random data were used we would get an entirely different decision tree.

# Some Issues in Decision Tree Induction (4)

## 3) Overfitting (continued)

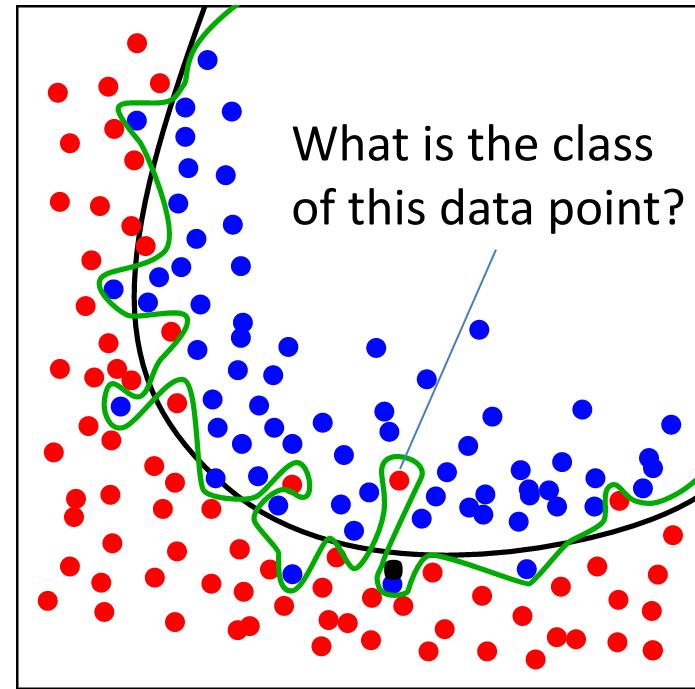
### Question

Could the same sort of thing happen with non-random data?

### Answer

Yes, when the training data is small, noisy, or corrupted, the deduced decision tree will fit the noise.

An example of overfitting with non-random data: see the figure.



The samples nearby separation boundary could be mislabelled due to noise.

The separation boundary in green is bad, which is a good example of overfitting.

# Some Issues in Decision Tree Induction (5)

## 3) Overfitting (continued)

What are the reasons for overfitting?

A decision tree is a mathematical model of some *population* of samples, but the tree is built on the basis of *a small fraction of the population* – the training dataset. What a decision tree induction program really does is building *a model of the training dataset*.

The induced decision tree could reflect relationships that are *true for the population as a whole*, when the training dataset is representative, but it may only reflect relationships that *are peculiar to the particular training dataset*, when the training dataset is too small or of poor quality (e.g., the example in this lecture, with one ‘cloudy’ sample only).

This phenomenon of modelling the training data rather than the population it represents is called *overfitting*. The reasons for overfitting include that the training data is not representative or the model is too flexible or powerful.

# Some Issues in Decision Tree Induction (6)

## 3) Overfitting (continued)

**How to combat overfitting?**

**- by better model and/or better training data**

There are two basic ways of reducing or preventing overfitting in decision tree induction:

- Stop tree growth before it happens: ***Pre-pruning***.
- Remove parts of the tree due to overfitting after it has been constructed: ***Post-pruning***.

The pruning methods are beyond the scope of this module.

Of course, increasing the number of high-quality training samples (if available/possible) is a more direct approach to overfitting prevention.

# **Summary**

## **The Weather Data Example of Decision Tree Induction**

Steps in the pseudo code

Procedure for choosing best attributes using  
information gain

## **From Decision Trees to Production Rules**

Knowledge discovery from data by machine learning

## **Some Issues in Decision Tree Induction**

Inconsistent Data

Numeric Attributes

Overfitting (to be addressed again in Neural Networks)

# CE213 Artificial Intelligence – Lecture 13

## Neural Networks: Part 1

What Is a Neural Network?

Why Do We Need Neural Networks?

How to Design or Train a Neural Network?

McCulloch-Pitts Neuron and Neural Networks (NN Part 1)

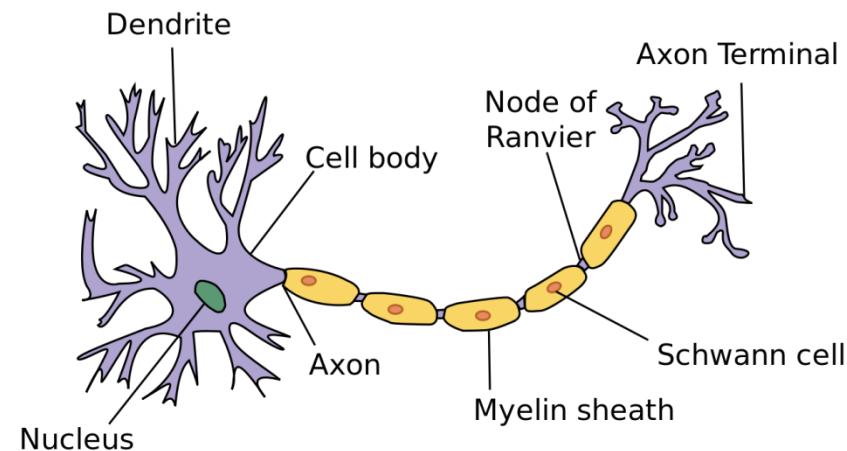
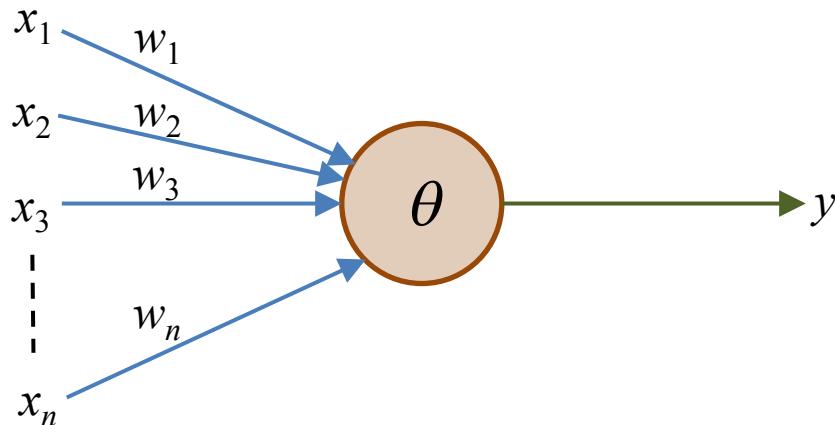
Learning Using The Delta Rule (NN Part 2)

The Generalised Delta Rule & Back-Propagation Networks (NN Part 3)

# McCulloch-Pitts Neuron Model

It is called an MP neuron (figure on the left) for simulating a typical biological neuron (figure on the right).

The relationship between the input and output of an MP neuron can be described by the following diagram:



where

$x_i$  is the  $i^{\text{th}}$  input

$w_i$  is the weight of the  $i^{\text{th}}$  input

$\theta$  is a threshold

$y$  is the output

A typical biological neuron  
(Approximately 100 billion neurons in the human brain)

## McCulloch-Pitts Neuron Model (2)

The output of an MP neuron is computed from the input as follows:

- 1) Calculate the weighted summation of inputs:

$$\sum_{i=1}^n w_i x_i = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n$$

- 2) Compare the weighted sum of inputs with the threshold to generate output:

$$\sum_{i=1}^n w_i x_i \geq \theta \rightarrow y = 1$$

$$\sum_{i=1}^n w_i x_i < \theta \rightarrow y = 0$$

The output of an MP neuron is binary. Its input takes continuous values in general, but binary values for logic.

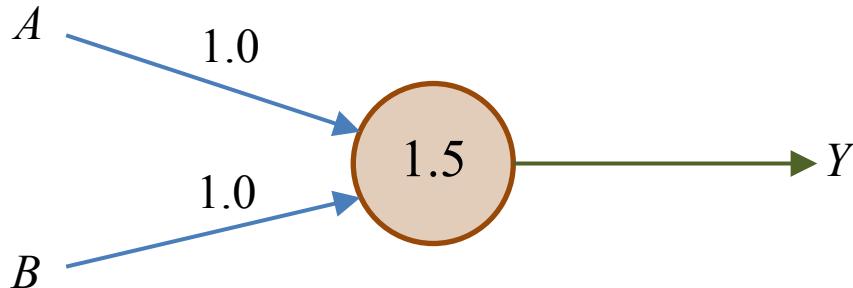
Basic information processing in an MP neuron: **weighted sum** and **thresholding**, which could be quite powerful (e.g., fault diagnosis).

This is more like a logic gate than a biological neuron.

# What Can an MP Neuron Compute?

Assume inputs are binary (i.e., 1 or 0)

## Implementation of AND



A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

If  $A + B \geq 1.5$ ,  $Y = 1$ .

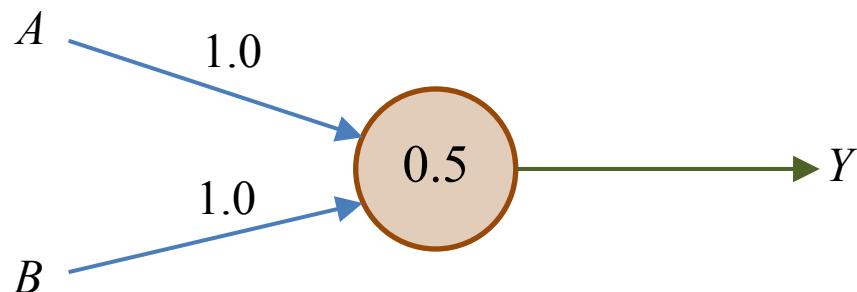
Otherwise,  $Y = 0$ .

Of course, there could be many other correct designs, e.g., changing 1.0 to 1.1 or 1.2 would do.

( $A$  and  $B$  correspond to input  $x_1$  and  $x_2$  in the MP neuron model presented in the previous slides. The weighted sum is  $w_A \cdot A + w_B \cdot B$ , where  $w_A = w_1 = 1.0$  and  $w_B = w_2 = 1.0$ .)

# What Can an MP Neuron Compute? (2)

## Implementation of OR



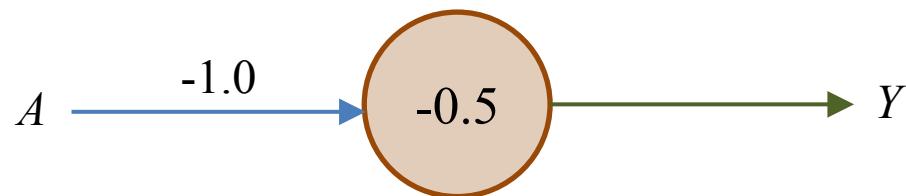
If  $A + B \geq 0.5$ ,  $Y = 1$ .

Otherwise,  $Y = 0$ .

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

# What Can an MP Neuron Compute? (3)

## Implementation of NOT



If  $-A < -0.5$  or  $A > 0.5$ ,  $Y = 0$ .

Otherwise,  $Y = 1$ .

A	Y
0	1
1	0

# What Can an MP Neuron Compute? (4)

Are there any Boolean functions that an MP neuron cannot compute?

How about implementation of XOR (exclusive OR)?

A	B	A xor B
1	1	0
1	0	1
0	1	1
0	0	0

(Truth table of XOR)

# What Can an MP Neuron Compute? (5)

Suppose the weights are  $w_A$  and  $w_B$ , and weighted sum of inputs to the MP neuron is  $w_A \cdot A + w_B \cdot B$ .

Then the following equations should be satisfied for computing XOR (check the equations on slide 3):

- |                      |                         |
|----------------------|-------------------------|
| $w_A + w_B < \theta$ | (1) (for A=1, B=1, Y=0) |
| $w_A \geq \theta$    | (2) (for A=1, B=0, Y=1) |
| $w_B \geq \theta$    | (3) (for A=0, B=1, Y=1) |
| $0 < \theta$         | (4) (for A=0, B=0, Y=0) |

According to (2), (3) and (4),  $\theta$ ,  $w_A$  and  $w_B$  must be positive. If both (2) and (3) are true, then (1) must be false,

Therefore, there is no combination of values for weights and threshold that can create an MP neuron for computing the XOR function.

**An MP neuron cannot compute XOR!**

# What Can an MP Neuron Compute? (6)

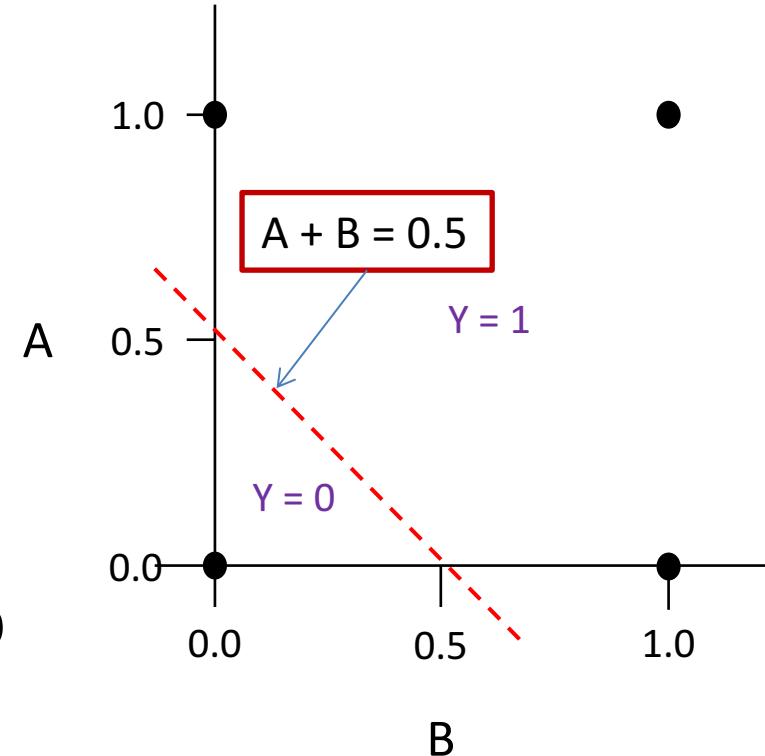
## A Graphical Interpretation

Let's consider the implementation of **OR**:  
the values of weights and threshold  
of the MP neuron are:

$$w_A = w_B = 1; \theta = 0.5$$

What values of A and B will make  $Y = 1$ ?

$Y=1$  at all points where  $w_A \cdot A + w_B \cdot B \geq \theta$   
i.e., whenever  $A + B \geq 0.5$



Therefore, from the figure on the right we can see that

- All points above or on the line described by  $A + B = 0.5$  lead to  $Y = 1$
- All points below the line lead to  $Y = 0$

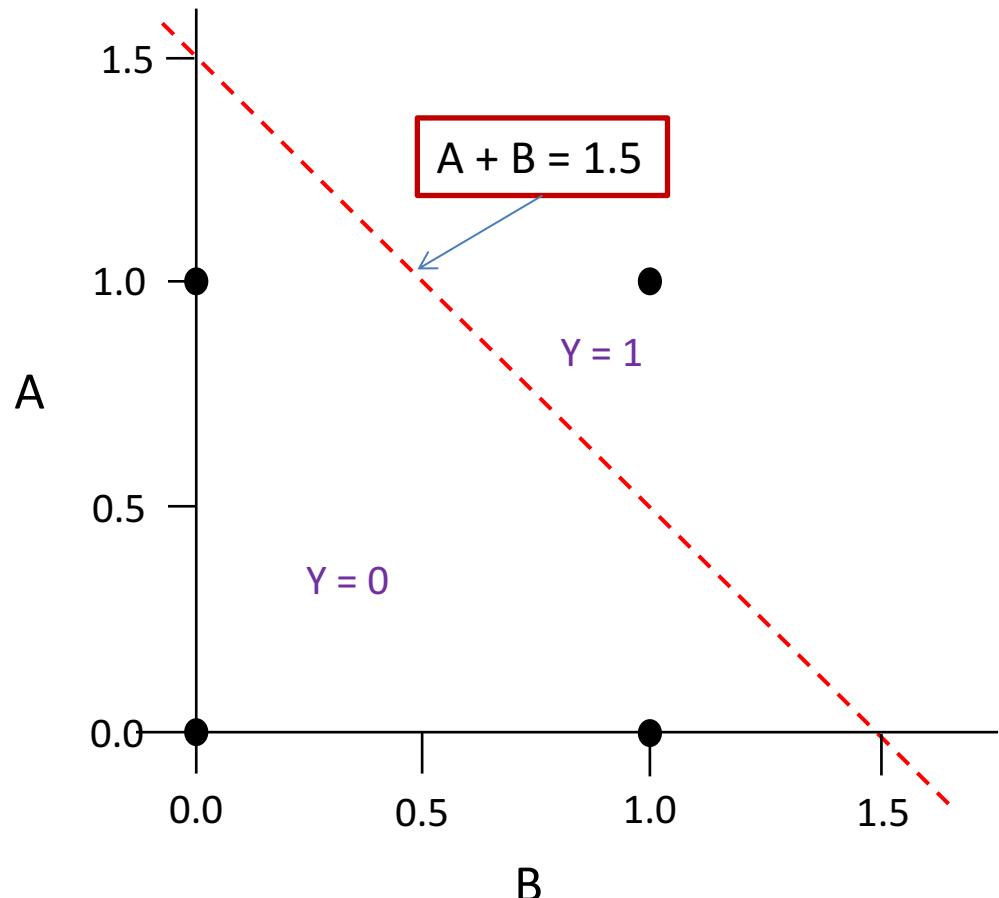
# What Can an MP Neuron Compute? (7)

Similarly, for the implementation of **AND**, the values of weights and threshold of the MP neuron are:

$$w_A = w_B = 1 \text{ and } \theta = 1.5,$$

which define a line in the figure on the right:

$$w_A \cdot A + w_B \cdot B = \theta, \text{ or } A+B = 1.5.$$



Therefore,

All points above or on the line described by  $A + B = 1.5$  lead to  $Y = 1$

All points below the line lead to  $Y = 0$ .

# What Can an MP Neuron Compute? (8)

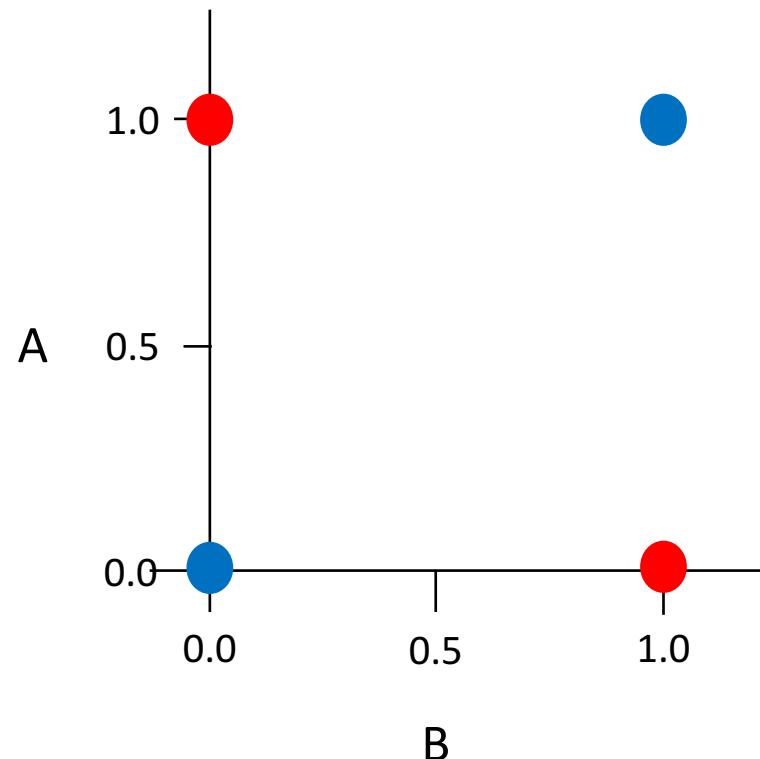
So, what about implementation of XOR?

$Y$  should be 1 at red spots.

$Y$  should be 0 at blue spots.

There is no straight line that can separate the blue spots ( $Y=0$ ) from the red spots ( $Y=1$ ).

Therefore, it is impossible for an MP neuron to implement logic XOR !



# What Can an MP Neuron Compute? (9)

**When an MP neuron has more than 2 Inputs (one neuron only)**

The function implemented by such an MP neuron depends on

$$\sum_{i=1}^n w_i x_i = \theta$$

This is the equation of a **hyperplane** in  $n$ -dimensional space.

**So, an MP neuron divides the space into two regions separated by a hyperplane. On one side of this hyperplane,  $Y=0$ , and on the other side,  $Y=1$ .**

**The values of the weights and threshold of the MP neuron determine the position and orientation of this hyperplane. – This may solve complicated classification problems in high-dimensional space! (This had led to the first golden age of neural network research in 1960s)**

# Linear Separability

The functions that an MP neuron with  $n$  inputs can implement are those defined by a linear surface:

when  $n = 2$  it is a straight line

when  $n = 3$  it is a plane

when  $n > 3$  it is a hyperplane

Such functions are said to be ***linearly separable***.

For this reason, MP neurons are sometimes called:

***Linear Separation Units (LSUs)***

(They are not linear units, but linear separation units)

# McCulloch-Pitts Neural Networks

## - More Than One MP Neuron

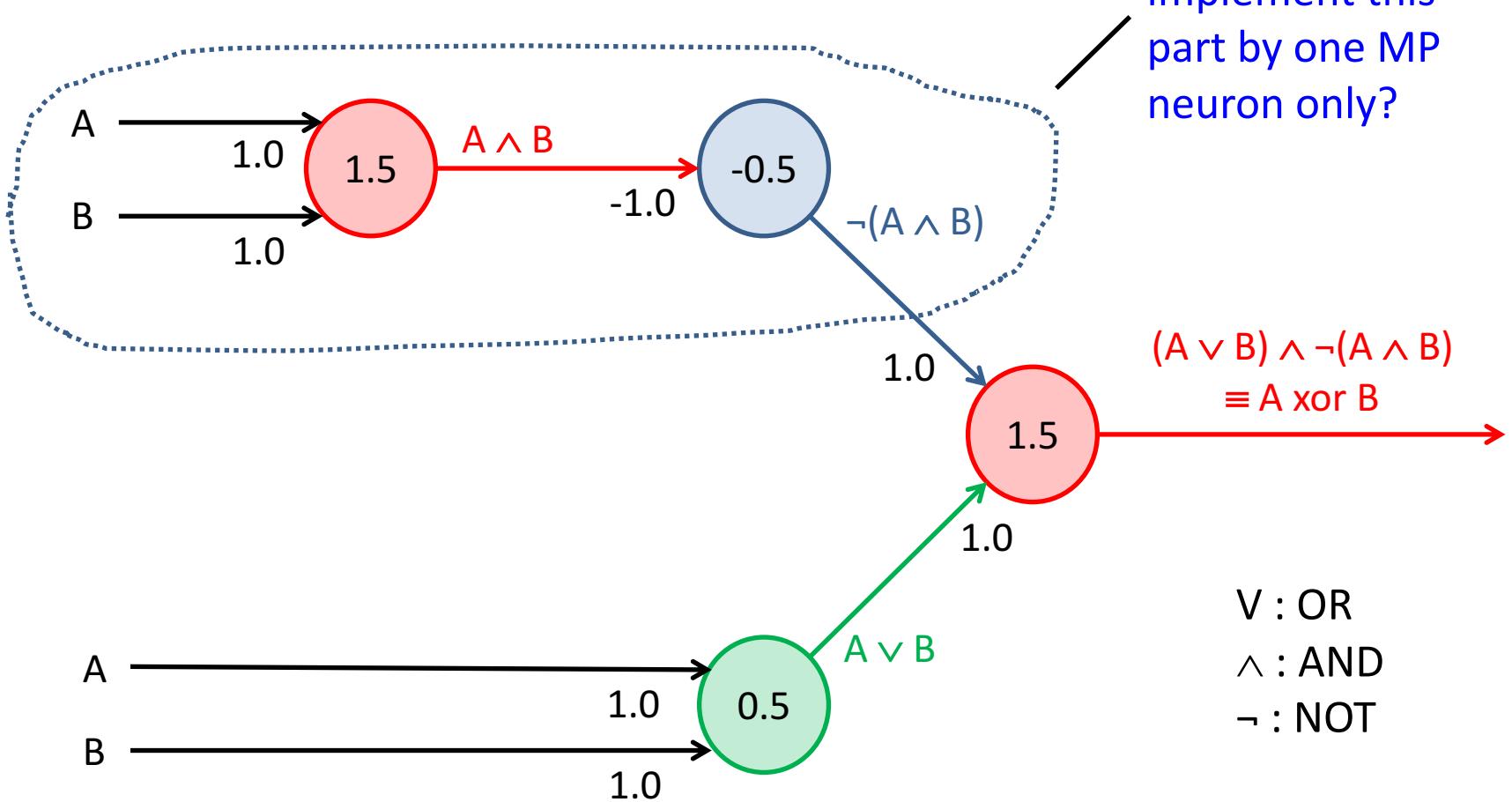
We could obviously use the output of one MP neuron as an input to another MP neuron.

In such a way we could build an *artificial neural network*.

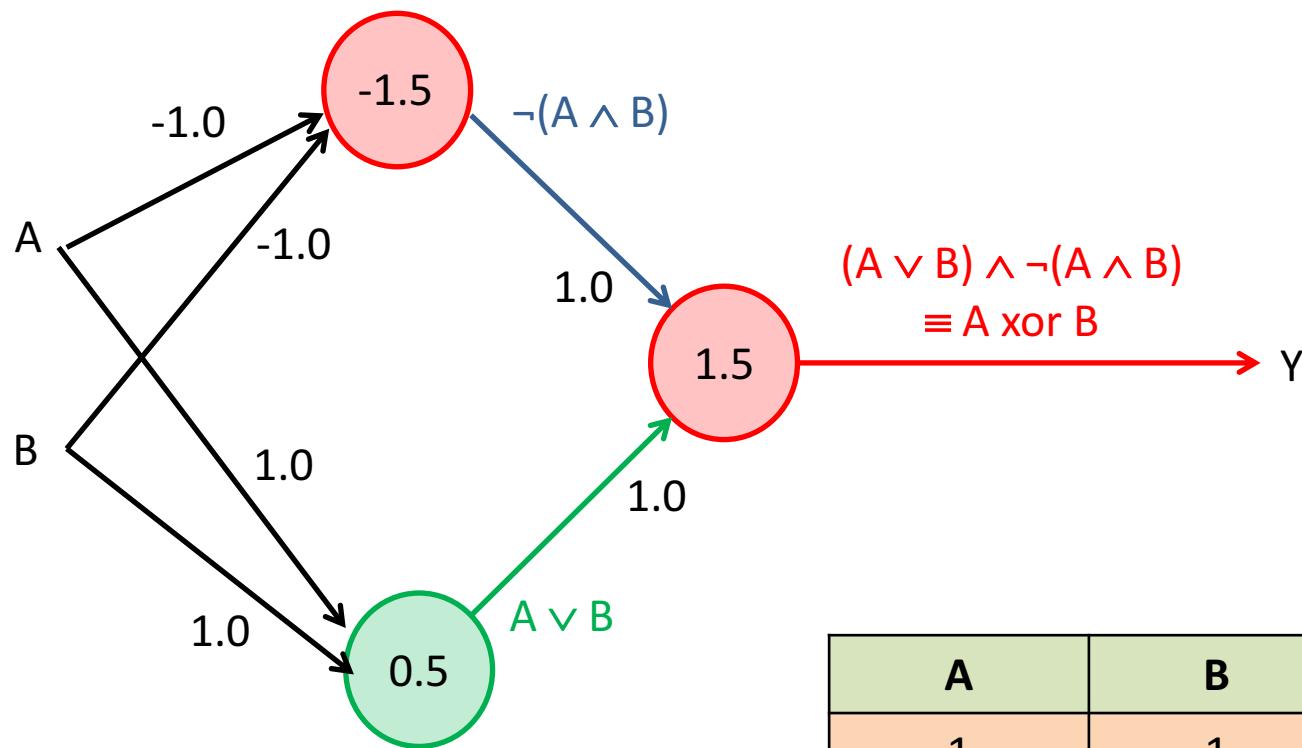
Could such a neural network (more than one neuron) compute functions that a single neuron could not? - Yes

# McCulloch-Pitts Neural Networks (2)

## Implementation of XOR with 4 MP neurons



# A Neural Network with 3 MP Neurons for Computing XOR:



Minimum number  
of MP neurons  
required for  
computing XOR?

Calculate the network's output Y with different values of A and B to prove it solves the XOR problem.

A	B	Y
1	1	?
1	0	?
0	1	?
0	0	?

# Summary

## McCulloch-Pitts Neuron and Neural Networks

MP Neuron (described by a diagram or equations)

What an MP Neuron Can Compute

Linear Separability – One Neuron with More Than 2 Inputs

Neural Network - More Than One Neuron

*Designing an MP neural network like the one for computing XOR seems difficult, especially for high-dimensional input problems.*

*Solution – Learning from data/experience (topic for next lecture)*

# CE213 Artificial Intelligence – Lecture 14

## Neural Networks: Part 2

### Learning Rules

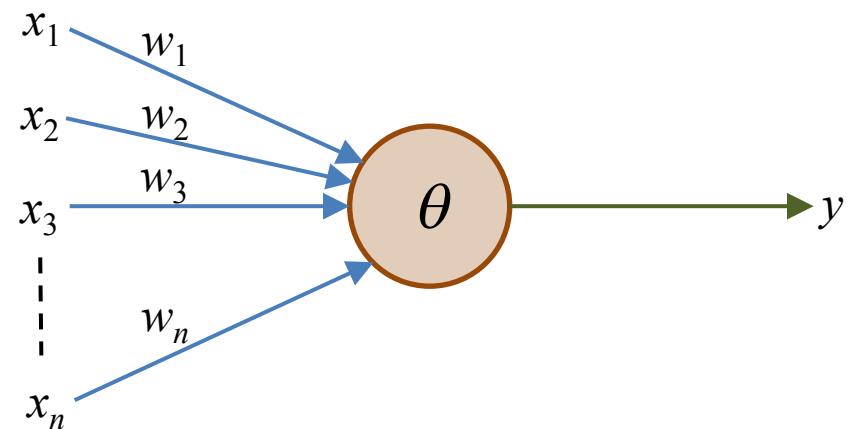
- Hebb Rule
- Perceptron Rule
- Delta Rule

Neural Networks Using Linear Units without Thresholding

# Learning in Neural Networks

Let's consider learning in one MP neuron first before considering learning in a neural network.

It can be described by a diagram:



or described by equations:

If  $\sum_{i=1}^n w_i x_i \geq \theta$ , then  $y = 1$

If  $\sum_{i=1}^n w_i x_i < \theta$ , then  $y = 0$

The function of the MP neuron is determined by the values of its weights and threshold.

Given a labelled sample dataset, how to determine the values of  $w_i$  and  $\theta$  so that the MP neuron will classify or fit the sample data correctly?

- 1) Analytical way (usually when  $n \leq 3$ )
- 2) Machine learning

## Learning in Neural Networks (2)

The function of an MP neuron can be constructed by **appropriate choice of weight and threshold values**, e.g., making the following hold if  $(x_1, x_2)$  and  $(x'_1, x'_2)$  are from different classes:

$$w_1 x_1 + w_2 x_2 > \theta \rightarrow y=1 \text{ (class 1)}$$

$$w_1 x'_1 + w_2 x'_2 < \theta \rightarrow y=0 \text{ (class 2)}$$

This is feasible when there are 2 inputs only, but with a large number of inputs we have to **train** a neuron or neural network to achieve the desired weights by fitting a set of samples through machine learning.

**What information is available from samples and the MP neuron?**

The current input:  $x_1 \dots x_n$   from the samples

The desired output (label):  $z$  

The current weights:  $w_1 \dots w_n$  

The actual output :  $y$   from the MP neuron

**How to make use of the above information to learn – to change weight values?**

# Learning in Neural Networks (3)

A general solution to machine learning in neural networks is to use the following learning rule:

$$w_i(t + 1) = w_i(t) + \Delta(x_i, w_i, y, z)$$

where

$w_i(t)$  is the weight of the  $i$ th input at time  $t$

$\Delta$  is a function that determines the change in weight.

A learning procedure would simply be:

**Initialisation of weight values;**

**REPEAT**

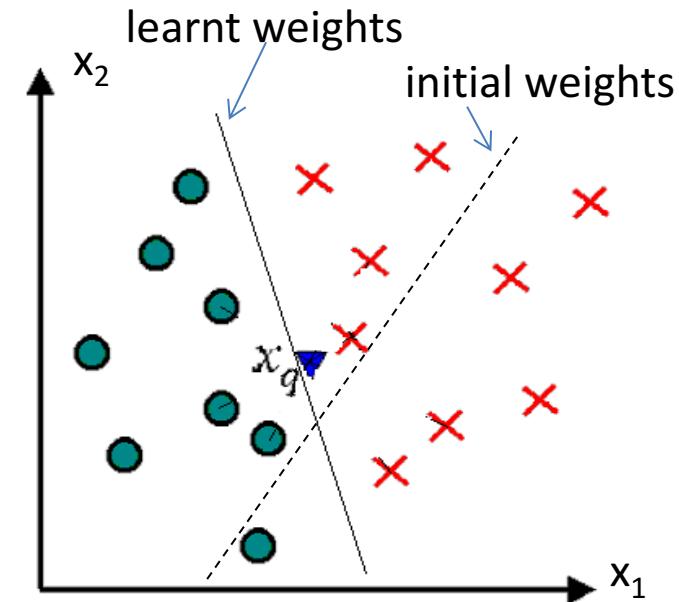
**Present next sample;**

**Adjust all weights using  $\Delta$ ;**

**UNTIL Training complete;**

(There could be different completion criteria.)

<https://www.youtube.com/watch?v=vGwemZhPlsA>



# Learning in Neural Networks (4)

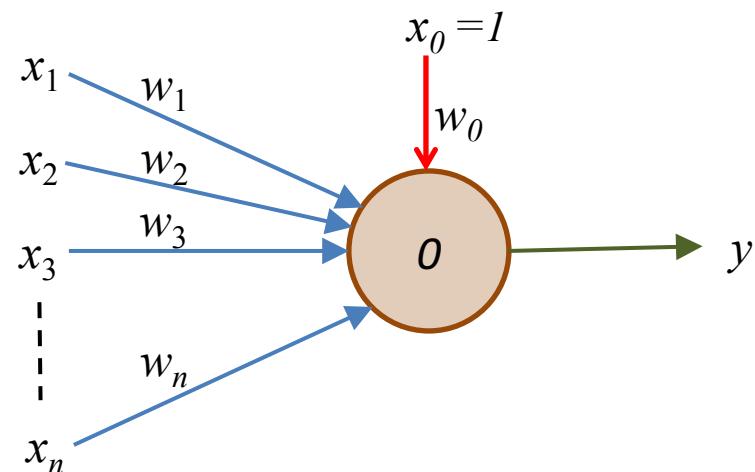
**How to update the value of the threshold through machine learning?**

If we set threshold to zero and add an extra input  $x_0$  with weight  $w_0$ , then the threshold can be regarded as a special weight.

Let  $x_0$  be always 1, then the MP neuron can be described as follows:

If  $\sum_{i=0}^n w_i x_i = \sum_{i=1}^n w_i x_i + w_0 \geq 0$  then  $y = 1$

If  $\sum_{i=0}^n w_i x_i = \sum_{i=1}^n w_i x_i + w_0 < 0$  then  $y = 0$



Now there is no  $\theta$  in the diagram or the equations of the MP neuron. This is equivalent to set  $\theta = -w_0$

## Learning in Neural Networks (5)

So, the weight with fixed input is essentially an adaptive threshold, because the equations describing the MP neuron are as follows:

If  $\sum_{i=1}^n w_i x_i \geq -w_0$  then  $y = 1$

If  $\sum_{i=1}^n w_i x_i < -w_0$  then  $y = 0$

( $-w_0$  plays the role of threshold  $\theta$ )

In this way, we can arrange for the threshold to be modified in the same way as the weights.

Now let's focus on **how to determine  $\Delta(x_i, w_i, y, z)$** .

# Hebb Rule

Hebb (1949) proposed that learning could take the form of “strengthening the connections” between any pair of neurons that were **simultaneously active**.

This implies a learning rule of the form ( $x_i$  may be the output of another neuron)

$$w_i(t + 1) = w_i(t) + \alpha x_i y$$

where  $\alpha$  is a constant that determines the rate of learning.

This rule was modified later for training MP neurons.

During learning, the output of the neuron ( $y$ ) is forced to the desired value ( $z$ ). Hence the learning rule becomes

$$w_i(t + 1) = w_i(t) + \alpha x_i z$$

This is now known as the “**Hebb Rule**”.

Is this supervised learning?

# Perceptron Rule

Perceptron Rule was the second learning rule in the history of machine learning, which was proposed by Prof. Marvin Minsky in late 1950s when he was a PhD student.

The basic idea is to adjust the weights only when the neuron would have given the wrong output.

**The Perceptron Rule:**

$$w_i(t + 1) = w_i(t) + \alpha x_i z \quad \text{only if } y \neq z$$

which is actually the Hebb rule restricted to **error correction**.

However, such a minor change to the Hebb rule made a big difference.

# Delta Rule

The delta rule generalises the idea of **error correction** introduced in the perceptron rule.

Rather than having a rule that is only applied if an error occurs, delta rule includes the error in it:

$$w_i(t + 1) = w_i(t) + \alpha x_i(z - y) = w_i(t) + \alpha x_i \delta$$

This delta rule was proposed in 1960, also known as the Widrow-Hoff rule.

Can  $\delta$  be  $(y-z)$  ?

Why is it called the delta rule?

Because the difference or error  $(z - y)$  is often written as  $\delta$ .

## Delta Rule (2)

Delta rule can be used to train neural networks for both classification and function approximation.

Let's consider the task of training a ***function approximator*** First.

If we take away the thresholding, we have a linear sum unit:

$$y = \sum_{i=0}^n w_i x_i \quad (\text{normally } x_0 = 1)$$

The input and output could be real numbers, not just 1 or 0.

Such a computing unit clearly computes a hyperplane in  $n$ -dimensional space.

## Delta Rule (3)

**How will this computing unit behave if it is trained?**

Suppose training data can be described by  $z = f(x_1, \dots, x_n)$ .

Then the weights will adapt to values such that the computing unit provides a **linear** approximation to the function  $f$ .

Therefore, the computing unit can be used for function approximation as well as for classification.

Function approximation and classification are closely related:

**Function approximation** is a **surface** in  $n$ -dimensional space.

This surface separates two classes of points in  $n$ -dimensional space.

**For function approximation, it can be proved that the delta rule is an optimal strategy for changing the weights.** (Strict proof is beyond the scope of CE213)

# Limitations of the Delta Rule

## Two major limitations

- It can only learn linear functions or linear separation for classification.
- It is equivalent to linear regression.

However, standard techniques for calculating regressions are orders of magnitude faster; And there is no need to choose an appropriate value of learning rate  $\alpha$ .

Of course, delta rule can start training neurons as soon as data is available.

## Choosing appropriate value for learning rate $\alpha$

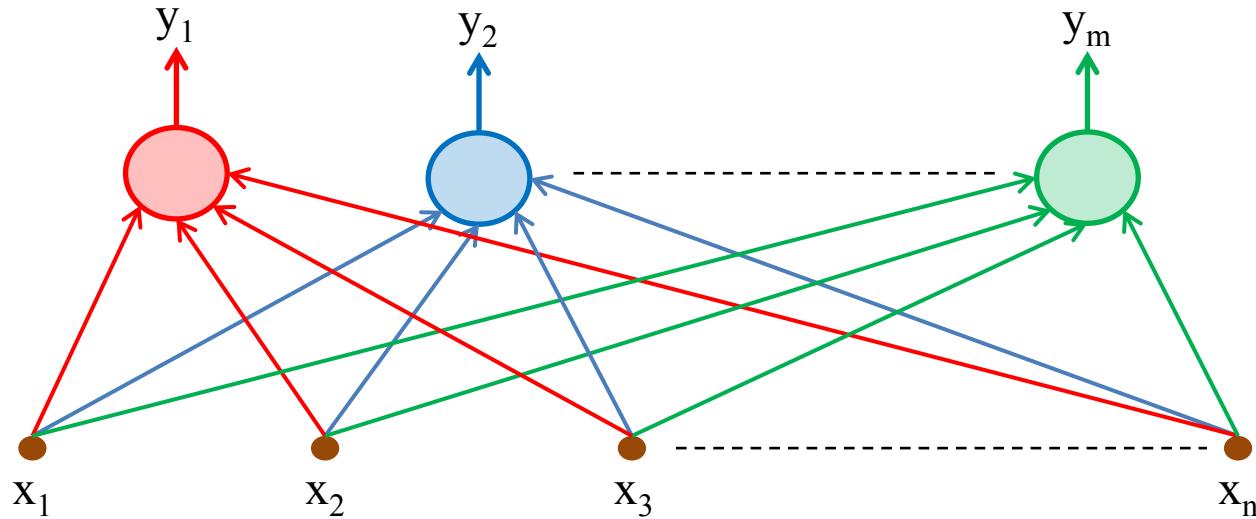
If  $\alpha$  is too small, learning will be very slow.

If  $\alpha$  is too large, the final weight values will cycle around the optimum values.

In many applications, ‘trial and error’ could be the only practical method for choosing appropriate learning rate.

# More Than One Output (Network)

A Single Layer Linear Network (consisting of linear units):



The behaviour of this network can be most conveniently expressed by

$$\bar{y} = \mathbf{W}\bar{x}$$

where

$\bar{x}$  is the input vector

$\bar{y}$  is the output vector

$\mathbf{W}$  is the weight matrix ( $w_{ij}, i=1,2,\dots,m; j=1,2,\dots,n$ )

## More Than One Output (2)

The weight of the  $j^{\text{th}}$  input to the  $i^{\text{th}}$  neuron is element  $[i,j]$  of the weight matrix  $\mathbf{W}$ .

So the delta learning rule now takes the following form

$$\mathbf{W}(t+1) = \mathbf{W}(t) + \alpha \bar{\delta} \bar{x}^\top \quad (\top - \text{transpose})$$

where

$$\bar{\delta} \equiv (\bar{z} - \bar{y}) = (\bar{z} - \mathbf{W}\bar{x})$$

$\bar{\delta}, \bar{x}, \bar{y}, \bar{z}$  are vectors, e.g.,  $\bar{x} = [x_1, x_2, \dots, x_n]^\top$

For each element in the matrix  $\mathbf{W}$ , the delta learning rule is as follows:

$$w_{ij}(t+1) = w_{ij}(t) + \alpha x_j(z_i - y_i)$$

## More Than One Output (3)

**How does such a single layer network with multiple outputs behave?**

Clearly it can be viewed as  $m$  single neurons operating in parallel:

There are no connections that could allow the behaviour of one neuron to affect another.

A single computing unit finds an approximation to a scalar function

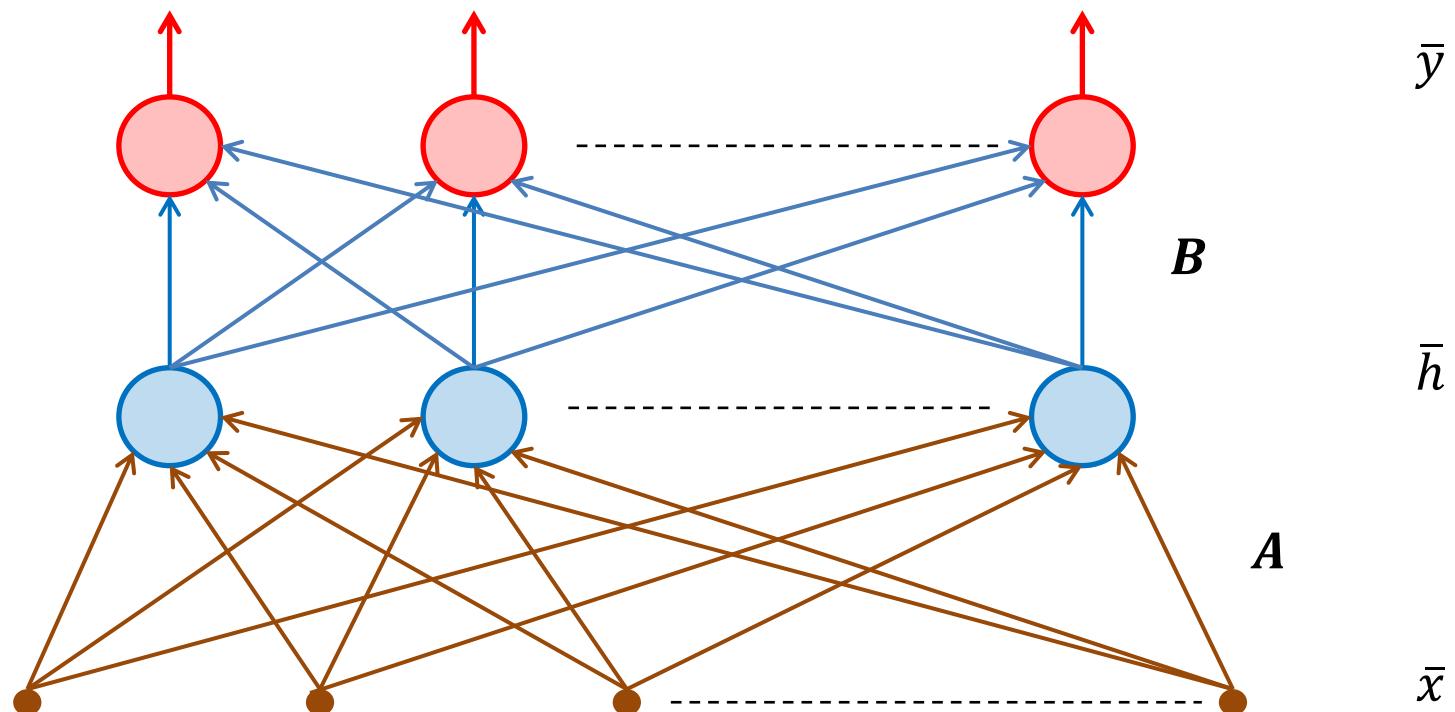
$$y = f(\bar{x})$$

**The multiple output network simply extends that to the case where the function has vector rather than scalar values.**

$$\bar{y} = \mathbf{f}(\bar{x})$$

# Multiple Layer Network Using Linear Units

Let's try adding another layer of linear computing units  
(not linear separation units, because there is no thresholding):



This is called a ***feed-forward network*** because signals flow in forward direction only.

## Multiple Layer Network Using Linear Units (2)

Now let's see what this multilayer network does:

$$\bar{h} = \mathbf{A}\bar{x} \quad (\text{lower layer})$$

$$\bar{y} = \mathbf{B}\bar{h} \quad (\text{higher layer})$$

Therefore, the relationship between the output and input of the multilayer network can be described by

$$\bar{y} = \mathbf{B}(\mathbf{A}\bar{x}) = (\mathbf{BA})\bar{x}$$

But the product of two matrices is another matrix.

So let

$$\mathbf{W} = \mathbf{BA}$$

Hence

$$\bar{y} = \mathbf{W}\bar{x}$$

*A multilayer linear network can be equivalently implemented by a single layer network.*

# Multiple Layer Network Using Linear Units (3)

## Conclusion

There is no point building a multilayer feed-forward network if the computing units perform linear transformations of their inputs.

Therefore

**The only way to build multilayer neural networks that are more powerful than single layer networks is to use computing units (neurons) with non-linear output functions.**

But

**The delta rule is designed to find the best weights for linear units only.**

**Need of new learning rule/algorithm!**

# Summary

## Learning in Neural Networks Using the Delta Rule

Hebb Rule

Perceptron Rule

**Delta Rule**

Limitations of the Delta Rule

Multilayer Networks Using Linear Computing Units

→ Need of Nonlinear Computing Units or Neurons  
in Multilayer Neural Networks and More Powerful  
Learning Algorithms

# CE213 Artificial Intelligence – Lecture 15

## Neural Networks: Part 3

Multilayer Feedforward Neural Networks

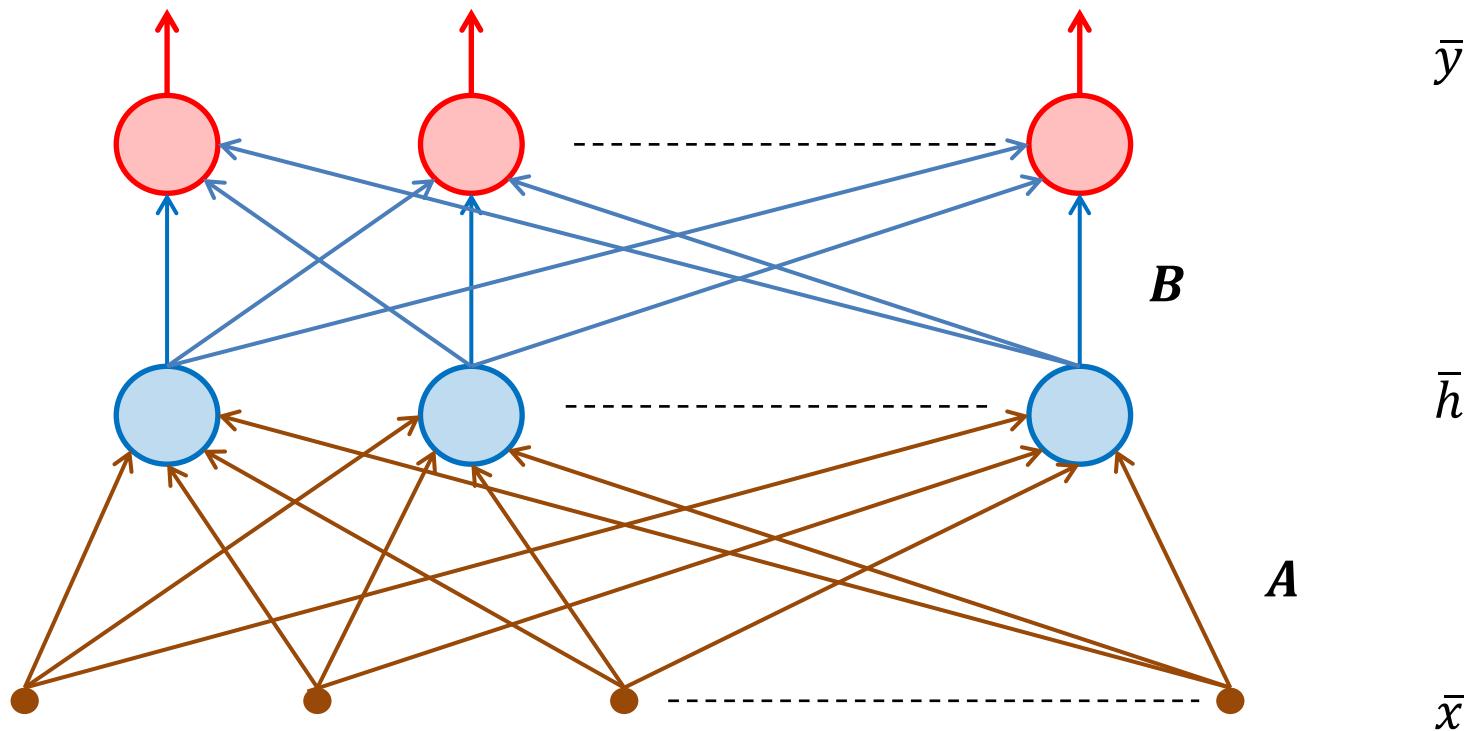
**Error Back-Propagation Learning Algorithm**

Overfitting Problem

[ Warning: It may be difficult to understand the error back-propagation idea and related equations. ]

# Multiple Layer Network Using Linear Units

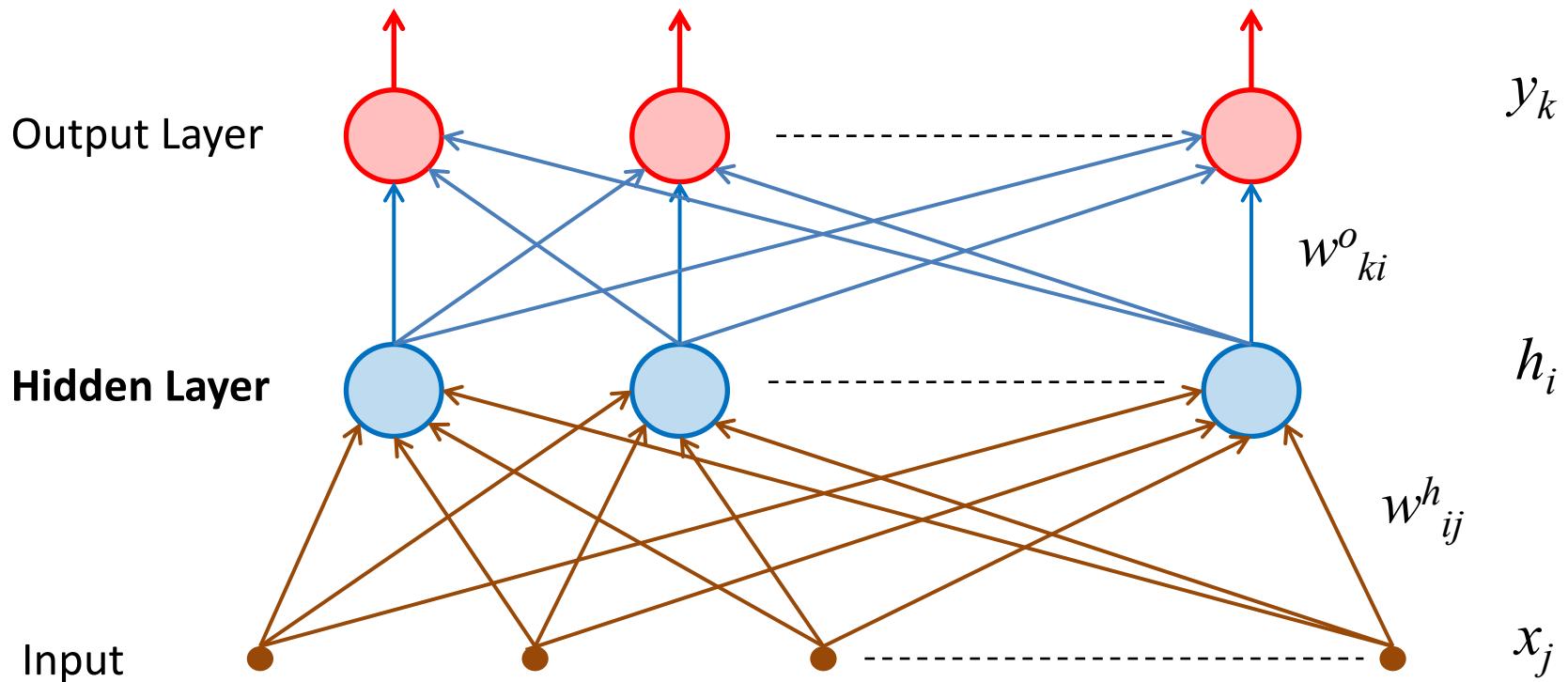
There is no point building a two-layer network of **linear** units, because it is functionally equivalent to a single layer network.



$$\bar{y} = B\bar{h} = B(A\bar{x}) = (BA)\bar{x} = W\bar{x}$$

# Multilayer Feedforward Neural Network

However, a two-layer neural network of **non-linear** neurons can be much more powerful than a single layer network.



$$y_k = f\left(\underbrace{\sum_{i=1}^{n_h} w_{ki}^o \cdot h_i}_{\text{red wavy underline}} - \theta_k^o\right) = f\left(\sum_{i=1}^{n_h} w_{ki}^o \cdot f\left(\underbrace{\sum_{j=1}^n w_{ij}^h \cdot x_j}_{\text{red wavy underline}} - \theta_i^h\right) - \theta_k^o\right)$$

# Multilayer Feedforward Neural Network (2)

**There are two problems:**

- The delta rule works for single layer neural networks or multilayer networks with linear units only.  
How do we modify it so that it can work with non-linear neurons?  
**Solution: Modify the delta rule → The generalised delta rule.**
- We don't know what the desired outputs or errors from the hidden layer would be.  
How do we train the hidden layer (update its weights)?  
**Solution: Reasonably estimate the ‘errors’ of the hidden neurons based on errors of output neurons → The error back-propagation learning algorithm.**

# The Generalised Delta Rule

The delta rule is defined as

$$w_j(t+1) = w_j(t) + \alpha \delta x_j$$

where  $x_j$  is the input to the neuron and

$$\delta \equiv z - y$$

Now suppose we replace the linear function  $y = \bar{w}\bar{x}$  with some **non-linear function**  $f$  so that

$$y = f(\bar{w}\bar{x})$$

***How can we modify the delta rule so that it will maximise the error reduction, no matter what the form of  $f$  is ?***

(N.B.  $\bar{w}$  and  $\bar{x}$  are vectors. In some following slides,  $\bar{w}\bar{x}$  is simply written as  $wx$ )

## The Generalised Delta Rule (2)

We modify the delta rule to give the following ***generalised delta rule***:

$$\delta x_j \Rightarrow \delta \frac{dy}{dw_j} = (z - y) \frac{dy}{dw_j}$$

why  $\frac{dy}{dw_j}$  ?

$dy/dw_j$  indicates how change in  $w_j$  will lead to change in  $y$ . In the case of a linear unit, this reduces to the simple delta rule since  $y = \bar{w}\bar{x}$  and  $dy/dw_j = x_j$ .

This restricts our choice of  $f$  to differentiable functions because the function must have a derivative defined at all points.

So we cannot use a step function (  ) as in the original MP neuron.

It can be proved that the generalized delta rule (similar to gradient descent algorithm) has the same property as the original delta rule, i.e.,

**It modifies the weights to minimize  $|z - y|$ .**

(Strict proof is beyond the scope of CE213. Will give a graphical interpretation later.)

# The Generalised Delta Rule (3)

## Choosing a suitable non-linear function

We need a function that is

- differentiable (so we can find its derivative at any point)
- Monotonically increasing (so increased input always produces increased output)

We would prefer a function that

- Is asymptotic for large values of  $\bar{w}\bar{x}$ , so that the output remains within limits (similar to the step function used in the MP neuron).
- Has a readily computable first derivative.

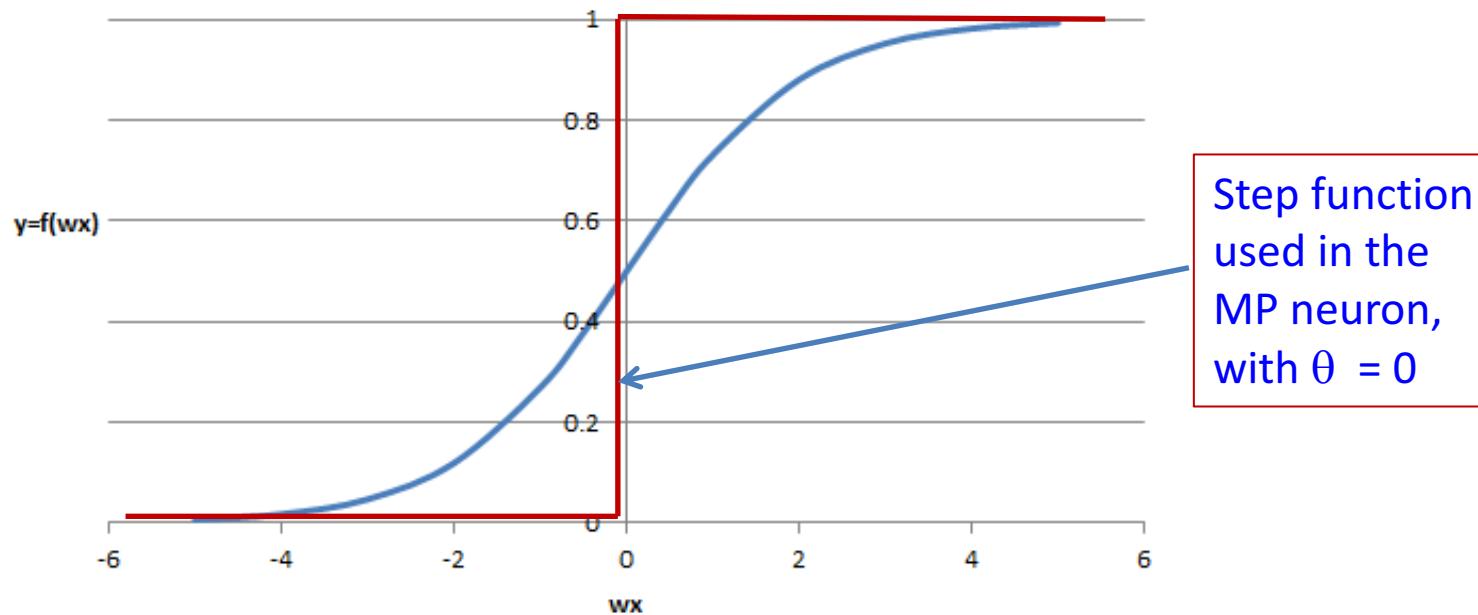
# The Generalised Delta Rule (4)

## The Logistic Function

$$y = \frac{1}{1 + e^{-wx}}$$

It has a sigmoid shape:

Choosing the logistic function for neuron modelling led to a breakthrough in machine learning in 1980s .



Step function  
used in the  
MP neuron,  
with  $\theta = 0$

# The Generalised Delta Rule (5)

**Properties of the logistic function:**

- Differentiable
- Monotonically increasing
- $f(wx) \rightarrow +1$  as  $wx \rightarrow +\infty$  (asymptotic)
- $f(wx) \rightarrow 0$  as  $wx \rightarrow -\infty$
- $f(wx) = 0.5$  when  $wx = 0$  (good for thresholding if needed)
- Derivative has a simple form:

$$f(wx) \times (1 - f(wx)) \quad \text{or} \quad y(1-y)$$

# Training the Hidden Layer

## The 2<sup>nd</sup> Problem

How can we train the hidden neurons when we do not know what their desired outputs (or errors) would be?

( for output neurons we have  $\delta \equiv z - y$  )

## Solution

Devise a way of making a plausible guess at what the errors of the hidden neurons would be.

[Training a neuron means finding appropriate values of its weights so that it will output as desired or the error between desired output and actual output will be minimised.]

# Training the Hidden Layer (2)

## Basic idea: Feeding back the error from the output layer

Make the estimated errors of hidden neurons **proportional** to the errors of the output neurons. This is reasonable because:

If the output is perfect we won't mess it up by changing the hidden layer (no error, no change).

If the output error is small we will only make small changes to the hidden layer.

## How much of the error should be fed back?

Each hidden neuron can contribute to each of the output neurons.

The **contribution of a particular hidden neuron** to a particular output neuron will depend on the **weight** of the connection between them.

Therefore, it is reasonable to use weights to proportion the error feedback.

# Training the Hidden Layer (3)

So, the solution to the 2<sup>nd</sup> problem (estimating errors of hidden neurons) is:

- Making the **estimated error of a hidden neuron** simply the **weighted sum of the errors of the output neurons**.
- Using the weights of the connections from the hidden neuron to the output neurons to form this weighted sum.

We can express this solution by introducing **a modified form of the generalised delta rule** for hidden neurons, described by the following equations:

This process of passing back a weighted error is called ***error back-propagation***.

$$w_{ij}^h(t+1) = w_{ij}^h(t) + \alpha \delta_i^h x_j$$

$$\delta_i^h = \left( \sum_{k=1}^m \delta_k^o \cdot w_{ki}^o \right) \cdot h_i \cdot (1 - h_i)$$

$$\delta_k^o = (\underline{z_k - y_k}) \cdot y_k \cdot (1 - y_k)$$

$k$  : index for output units, (Back propagation)

$i$  : index for hidden units,

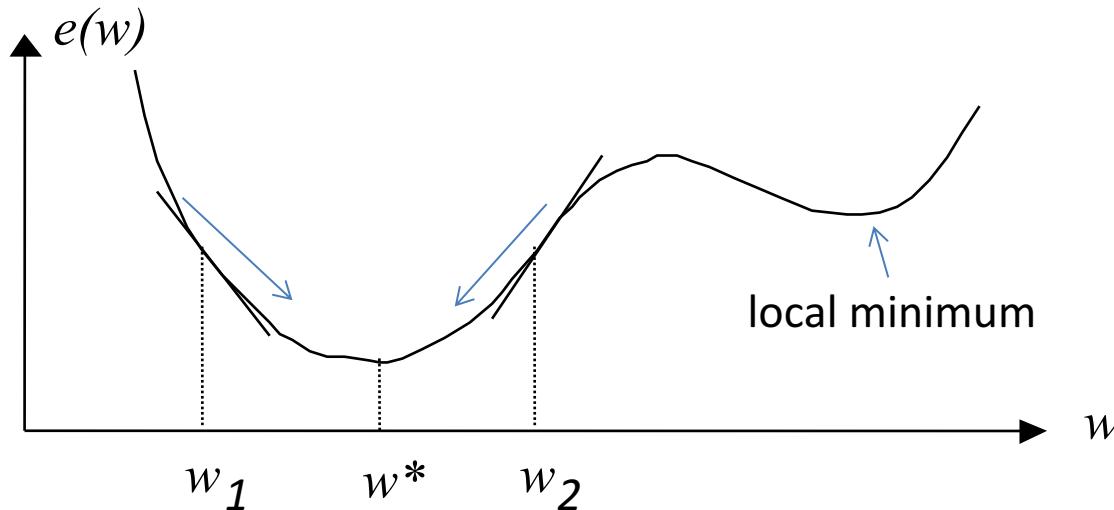
$j$  : index for inputs

[ Difficulty warning! ]

# Error Back-Propagation Learning Process Regarded as Error Gradient Descending

The delta rule and all its variants used in error back-propagation can be regarded as ***error gradient descending*** procedures, because

$$e = \frac{1}{2}(z - y)^2, \quad \Delta w = -\alpha \frac{de}{dw} = \alpha(z - y) \frac{dy}{dw}$$



Weight updating by  $\Delta w$  always reduce error if  $\alpha$  is small enough.

$$\left( \frac{de}{dw} < 0, \Delta w > 0 \right) \quad \left( \frac{de}{dw} > 0, \Delta w < 0 \right)$$

# When to Stop Error Back-Propagation?

Error back-propagation learning is an iterative process. How do you determine when the learning process should stop?

An obvious approach:

Monitor the average error for the output layer,  $(z - y)$ .

Stop learning when this falls below a small value  $\varepsilon$ .

Problem: It may happen that after a long period of learning we still have  $|z - y| > \varepsilon$ .

Alternative stopping criteria include:

Stop when the average error change is very small.

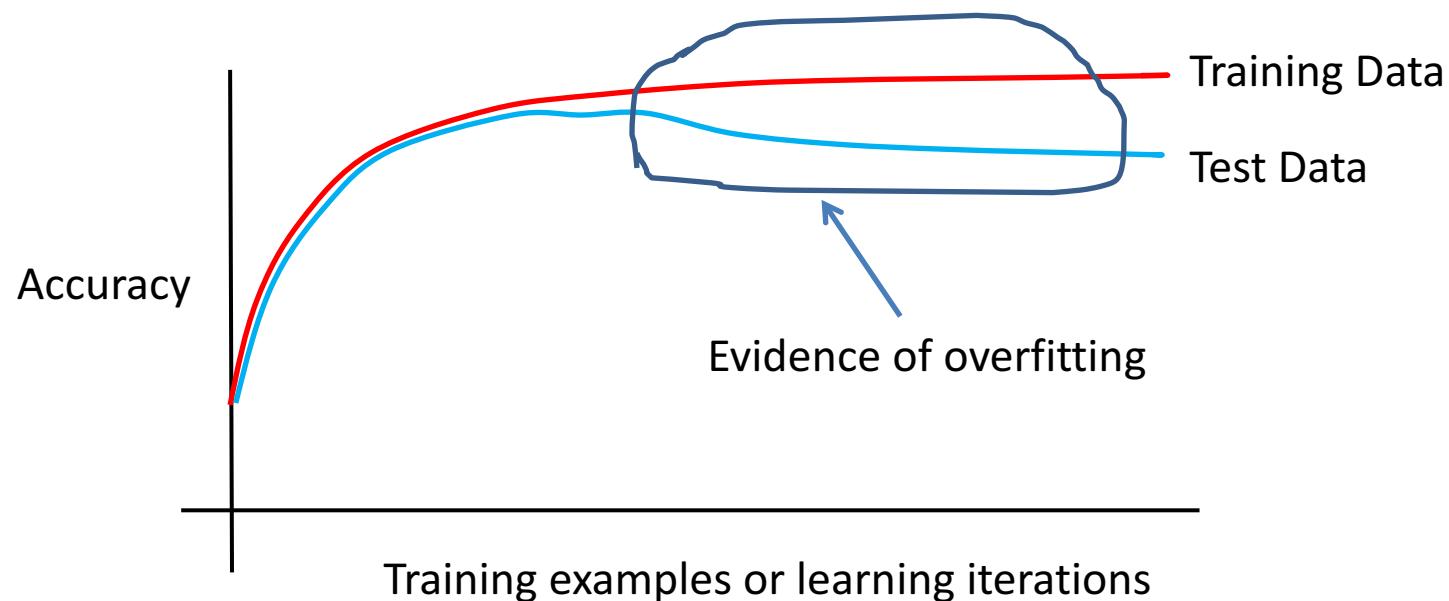
or

Stop when a preset number of iterations have been reached.

# Overfitting in Error Back-Propagation Networks

As in other forms of learning (e.g., decision tree induction), overfitting is a potential problem in error back-propagation neural networks.

Overfitting occurs because a built model is supposed to reflect the characteristics of the whole population, but it may reflect the characteristics peculiar to the training dataset only.



# Overfitting in Error Back-Propagation Networks (2)

## Main reasons for overfitting:

- Model used for machine learning is too flexible and powerful
- Number of training samples is too small (not representative)
- Training samples are too noisy (of low quality)

## Methods for reducing overfitting:

- Keep the number of hidden units (and thus number of weights) small
- Add weight decay
  - Decrease weight values by a small factor each iteration.
  - Thus weights tend to stay small.
- Use a large number of high-quality representative training samples.

The first two techniques **restrict the complexity** of the model that can be developed.

# Overfitting in Error Back-Propagation Networks (3)

## Dealing with overfitting through cross-validation:

Apart from the methods mentioned in the previous slide, cross-validation is a useful technique for reducing overfitting.

Split the available training data into two datasets:

***Training dataset:*** Used for weight updating.

***Validation dataset:*** Used to assess performance throughout learning.

Assess performance using validation dataset throughout learning.

Save weights for best performance so far.

Continue learning until performance on validation dataset has clearly dropped.

Restore weights to values saved from best performance.

Finally, assess performance using a separate ***Test dataset***.

Cross-validation can be also used to determine optimal neural network structure, leading to ***parametric learning combined with structural learning***.

# Applications of Error Back-Propagation Networks

Error back-propagation neural networks are very powerful in approximation, prediction, and classification.

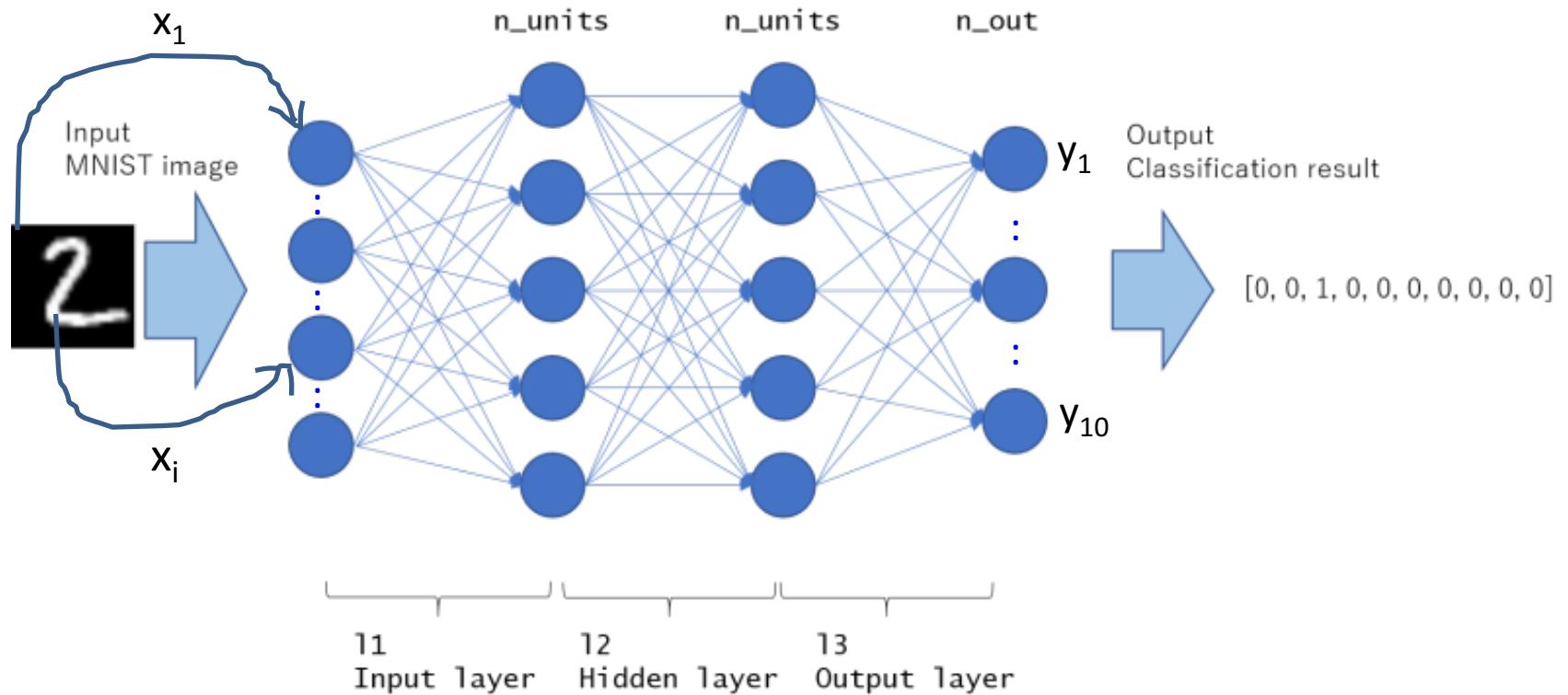
For classification problems, the desired output is like 0 ... 0 1 0 ... 0, with the only 1 representing a certain class. As actual output  $y$  takes continuous values, for classification problems,  $y$  should be converted to binary values by thresholding, e.g., 1 if it is greater than 0.5, and 0 if it is less than 0.5.

Unfortunately, it is impossible to demonstrate, step by step, the learning procedure of an error back-propagation neural network using a small set of training samples, like what we did for decision tree induction.

A demo: <https://www.youtube.com/watch?v=hB3b1CoZuR4> (classification)

Lab exercise 3 at your own time provides another example.

# An Application Example: Handwritten Digit Recognition



What would be happening during machine learning?  
What would be different for face recognition?

# Summary

## Multilayer Neural Networks Using Error Back-Propagation

Necessity of non-linear neurons

The generalized delta rule

Error back-propagation for training hidden neurons

## Overfitting in Error Back-Propagation Neural Networks

What is overfitting?

How to deal with overfitting?



# CE213 Artificial Intelligence – Lecture 16

## Clustering

Unsupervised learning – Learning from unlabeled data

It is different from decision tree induction and error back-propagation learning in neural networks due to the unavailability of class labels or desired output in the training sample data.

# A Brief Review of Machine Learning Concepts and Methods Taught in the Past Weeks

## KEY COMPONENTS OF MACHINE LEARNING:

- a task and an associated performance measure,
- a learning environment or sample dataset,
- a model (structure + parameters),
- a learning algorithm.

## Decision Tree Induction: supervised structural learning

Decision tree; Information gain based best attribute selection.

## Learning in Artificial Neural Networks: supervised learning (mostly parametric)

MP neuron model, neuron model with logistic function,  
multilayer feedforward neural network;  
Hebb rule, Perceptron rule, Delta rule, generalised Delta rule,  
error back-propagation learning algorithm.

# What is Clustering?

## THE TASK

Given a set of *unlabelled* training samples:

- Find a good way of partitioning the training samples into classes/groups.
- Construct a representation model that enables the class of any new sample to be determined.

Although the two subtasks are logically distinct, they are usually performed together.

## Terminology

Statisticians call this *clustering*.

Neural network researchers usually call it *unsupervised learning*, but unsupervised learning is more than clustering (e.g., data compression by machine learning, learning in generative adversarial network (GAN)).

# The Basic Problem in Clustering

## – Performance Measure

Classification learning programs are successful if the predictions they make are correct, i.e., if they agree with the externally defined class labels.

In clustering, *there is no externally defined notion of correctness*.

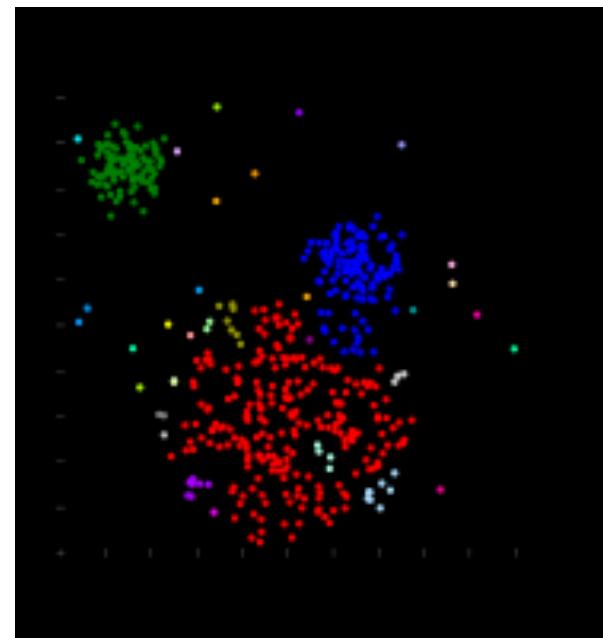
However, there are many ways in which a training dataset could be partitioned.

Some of these are better than others.

e.g., how to partition the data shown

in the figure on the right?

In your opinion, how many  
clusters are in this dataset?



# Partitioning Criteria

Common sense suggests that members of a class should resemble each other more than they resemble members of other classes (or clusters).

Therefore, a good partition should:

- maximise similarity within classes (**Criterion 1**)
- minimise similarity between classes (**Criterion 2**)

Note that this implies the existence of a similarity metric.

Are these two criteria enough to identify good partitions?

No.

Consider the partitioning in which every item is assigned to its own class.

Such a partition would be of no use, but meet the above criteria.

This suggests a further criterion:

- minimise the number of classes created (**Criterion 3**)

Clearly there will be a trade off between this and the other two criteria.

# Partitioning Criteria (2)

## How do we find the right balance?

Consider why we form classes,

i.e., what do we gain by assigning two samples to the same class?

One important reason for grouping individuals into classes is that being told the class of an item conveys a lot of information about it.

An example:

Suppose I tell you that Fido is a dog:

Immediately you are reasonably confident of the following:

Fido has four legs

Fido barks

Fido has sharp teeth

Fido probably chases cats

etc.

Class membership may contain a lot of information.

## Partitioning Criteria (3)

Motivated by the above example, from a different perspective, we could define a good partition as one that

**maximises the ability to predict unknown attribute values of an item from its class membership.**

This may be difficult to implement. We still need to consider individual criteria in certain balanced manner.

There may be no perfect solution to this balance problem. Let's see how specific clustering algorithms take these partitioning criteria into account in a balanced manner.

# Approaches to Clustering

Numerous methods have been devised for clustering.

We will look at the following two techniques:

Agglomerative Hierarchical Clustering

K-Means Clustering

Other techniques, including competitive learning, such as self-organising map, are beyond the scope of CE213.

# Agglomerative Hierarchical Clustering

The Basic Idea (pseudo code):

Assign each sample to its own cluster.

WHILE there are at least two clusters

    Find the most similar pair of clusters

(if there are more than one pairs with the highest similarity, choose one of them randomly)

    Merge them into a new larger cluster

Results are usually presented as a tree called a **dendrogram** (an example in the next slide).

This approach

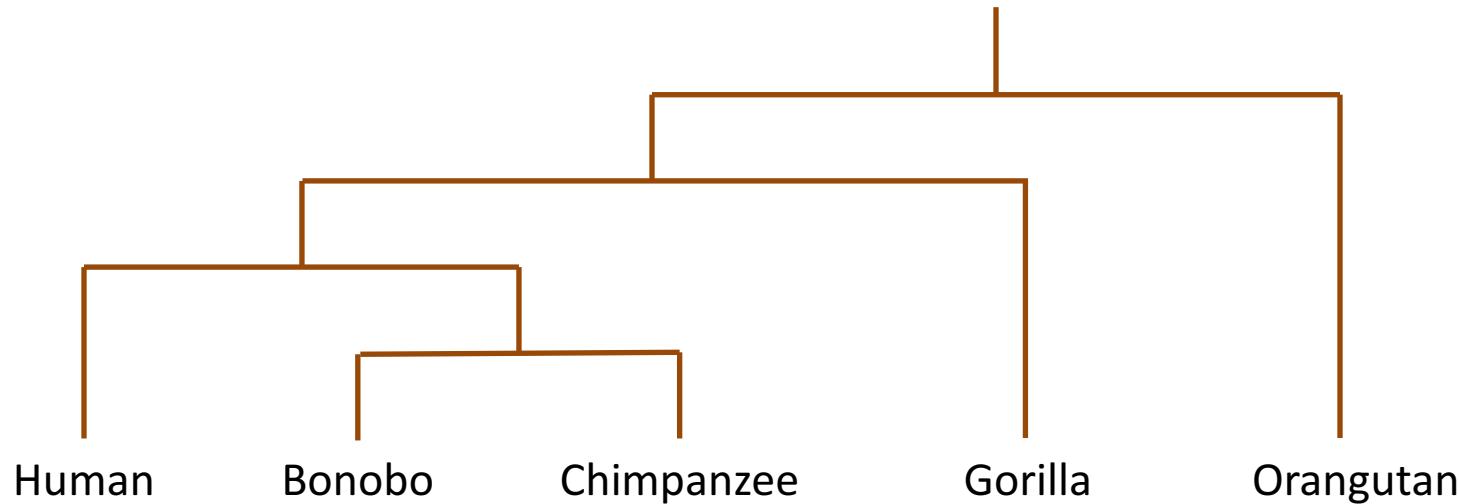
- Requires all samples to be available at the start.
- Requires a **similarity metric** that can determine the similarity between groups/clusters.
- Requires a **human analyst** to decide on the optimal number of classes/clusters (the third criterion).

[NB. Maximising similarity could be implemented by minimising distance.]

# Agglomerative Hierarchical Clustering (2)

An example dendrogram:

Here is a dendrogram for clustering great apes using similarity between given DNAs of the apes.



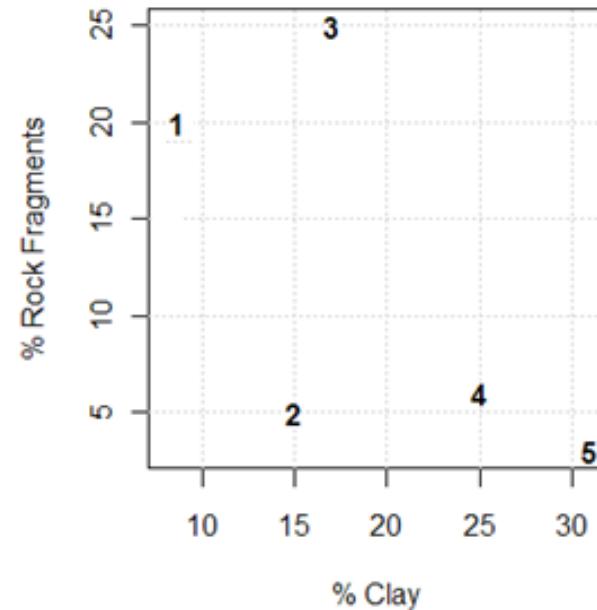
In your opinion (if you are the human analyst), what should be the optimal number of clusters/classes: 1, 2, 3, 4, or 5?

# Agglomerative Hierarchical Clustering (3)

Another example : using similarity/distance between soils' attributes

Five soil samples

Soil	%Rock Fragments	%Clay
S1	20	5
S2	5	15
S3	25	17
S4	6	25
S5	3	31

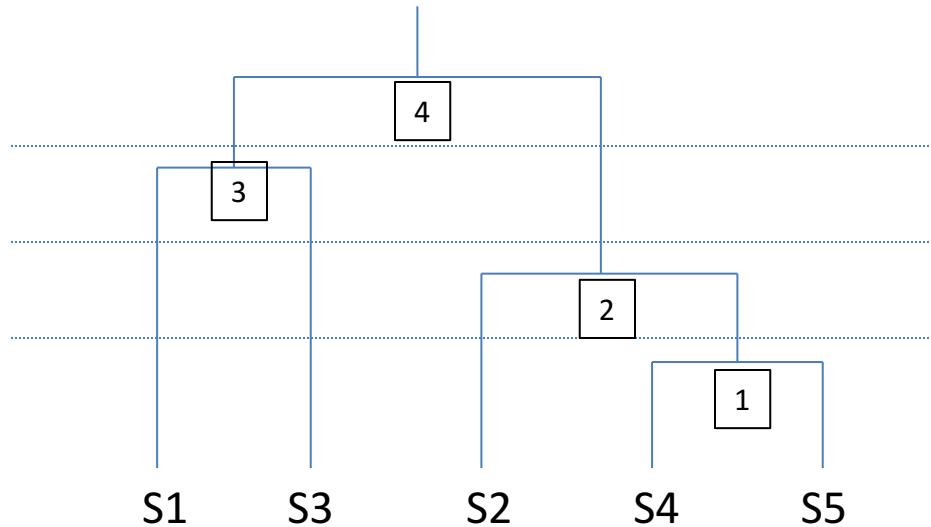


The **similarity of two soils** is defined as  $1/($ Euclidean distance between two attribute vectors $)$ , and the **similarity of two clusters or groups** of soils is defined as the similarity of the most similar pairs of soils where each member of the pair is from a different group.

# Agglomerative Hierarchical Clustering (4)

Soil Pair	Distance
S1, S2	18.03
S1, S3	13
S1, S4	24.41
S1, S5	31.06
S2, S3	20.10
S2, S4	10.05
S2, S5	16.12
S3, S4	20.62
S3, S5	22.36
S4, S5	6.71

(minimum distance =  
maximum similarity)



For 2 clusters: S1 and S3 in cluster 1,  
S2, S4 and S5 in cluster 2.

For 3 clusters: S1 in cluster 1,  
S3 in cluster 2,  
S2, S4 and S5 in cluster 3.

For 4 classes: ...

Optimal number of clusters?

# K-Means Clustering Method

The Basic idea can be described by the following pseudo code:

```
METHOD K-Means (samples, K) //K is number of clusters to be formed;  
Choose K samples randomly as initial cluster centroids  
REPEAT  
    Assign each sample to a cluster whose centroid is the  
    closest to it  
    Update the cluster centroid to the mean value of all  
    samples currently in that cluster  
UNTIL no sample changes the existing clusters  
Return K cluster centroids  
//one sample may be used many times, and it may be assigned to different  
//clusters in different iterations.
```

Input to the program is a set of unlabelled samples and the number of clusters to be formed. Output of the program is the K cluster centroids.

Number of iterations needed will depend on how well the formed clusters are.

# K-Means Clustering Method (2)

Demos (Pay attention to centroid initialisation and updating):

<https://www.youtube.com/watch?v=BVFG7fd1H30>

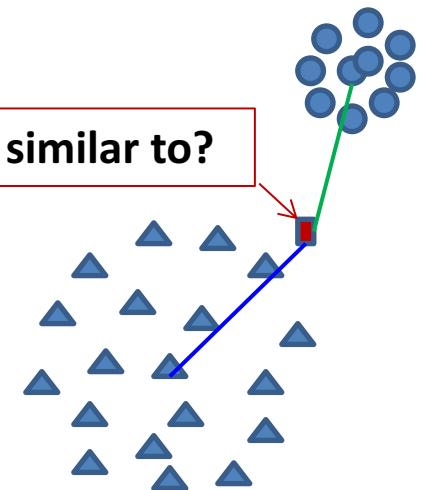
<https://www.youtube.com/watch?v=5FmnJVv73fU>

Which cluster is this new sample more similar to?

Discussions:

Cluster representation: mean, variance, .....

Similarity metric: statistical distance based, .....



If the number of clusters is unknown, how to conduct K-means clustering? – growing, pruning, separability checking, .....

# **Summary**

## **Clustering**

The task of clustering

Three partitioning criteria

## **Agglomerative Hierarchical Clustering**

Similarity metric

Dendrogram

Number of clusters

## **K-Means Clustering Method**

Similarity metric or distance measure

Cluster centroids

Number of clusters



# CE213 Artificial Intelligence – Lecture 19

## Genetic Algorithms

Nature-inspired Problem Solvers or Learning Algorithms:

*Can be regarded as problem solvers, directly applied to problem solving*

*Or regarded as learning algorithms, applied to build problem solvers such as neural networks*

# Biological Basis

## Darwin's Contribution to the Theory of Evolution

Contrary to popular opinion, Darwin did *not* invent the theory of evolution.

He proposed a ***mechanism*** of evolution :

*Source of Variation + Selection → Adaptation*

Adaptation will still occur even if the source of variation is completely **random**.

# Biological Basis (2)

## Mendelian Genetics

Organisms contain **coded descriptions** of all the features of the organism.

They are used to construct the organism during development and passed on to the organism's offspring.

Such coded descriptions are called ***genes***.

Genes are grouped into linear sequences called ***chromosomes***.

# Biological Basis (3)

## Sources of Variation

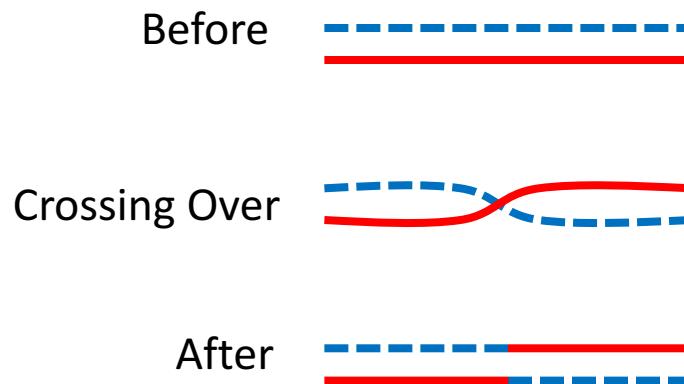
### Mutation:

Modifies genes randomly – usually detrimental, occasionally beneficial.

### Crossover:

Generates new combinations of genes.

Corresponding portions of **two chromosomes** are exchanged.



# Biological Basis (4)

## Natural Selection

If

An organism produces more offspring than can possibly survive

And

These offspring are only approximate copies, differing **in random ways**

Then

Those offspring that are best able to survive will be selected

## Fitness as selection criterion

The likelihood that an individual will survive to reproduce is called its ***fitness***.

Therefore, it is the “Survival of the Fittest” selection criterion makes improvement through evolution, generation by generation.

# A Basic Genetic Algorithm

## Suppose

- We want to search for the solution to some problem;
- We can encode candidate solutions to the problem as a string of symbols (***problem formalisation***).

Then we could create a population of candidate solutions by randomly generating a set of such strings that represent ***chromosomes***.

## Suppose we also have

- A function that could measure how good each candidate solution is, which could be used as a fitness function.

Then we could proceed to construct better solutions using the following procedure: cycles of “**Evaluation – Selection – Reproduction**”

# A Basic Genetic Algorithm – Pseudo Code

Initialise a population of chromosomes; //as candidate solutions

REPEAT //generation by generation, ‘evaluation-selection-reproduction’

    Determine fitness of each chromosome; //evaluation

    REPEAT

        //selection

        Select a pair of chromosomes (parents) with probability proportional to fitness;

        //reproduction

        Create two new chromosomes (children) using *Crossover*;

        Apply *Mutation* to randomly change the new chromosomes.

    UNTIL enough children have been generated

    Replace the least fit members of the population with the children;

UNTIL required number of generations has been reached.

The chromosome with the highest fitness represents the optimal solution.

# Mutation

The representation scheme (i.e., encoding) will define a set of values for representing genes in chromosomes.

e.g., {0,1} for a binary chromosome

Mutation simply replaces the current value in a gene with a randomly selected member of the value set.

## **Mutation Rate (to control how often a gene may be replaced.)**

Mutation Rate = Probability that mutation will occur at a given gene

Typical value:  $10^{-3}$

Mutation rates are typically very low since they introduce random change that is usually harmful (occasionally beneficial).

# Mutation (2)

## Effect of Mutation

Major benefit: It can introduce new values into the gene pool. It plays a sort of role of exploration.

A system in which mutation is the **only source of variation** would adapt, but only very slowly.

Such a system would lack any means to combine two partial solutions to make a better solution.

# Crossover

Crossover operators create new chromosomes by combining components of two existing chromosomes.

## Uniform One-Point Crossover

There are many varieties of crossover.

The simplest is uniform one-point crossover:

**Randomly select a single point somewhere along the chromosome;**

**Exchange the portions of the two chromosomes beyond the selected crossover point.**

The crossover is *uniform* if each point in a chromosome is equally likely to be selected.

# Crossover (2)

**Crossover Rate (to control how often chromosomes may be selected for crossover)**

Crossover rate = Probability that a given pair of chromosomes will crossover.

Typical values: 0.5 ~ 0.8

Such values allow

A significant number of chromosomes to be passed on to the next generation without modification

And a significant number of chromosomes to be recombined

# Crossover (3)

## Effect of Crossover

Major benefit: It can generate new gene combinations (new chromosomes) from the gene pool. It is more about exploitation.

A system in which crossover is the **only source of variation** would adapt, but would have no way of replacing any gene by something absent from the gene pool.

Hence, such a system might not find as good a solution as a system that also includes mutation.

## Relationship between Mutation and Crossover

Mutation and crossover are *complementary*.

They do different jobs:

Mutation ensures the whole solution space can be searched.

Crossover accelerates the search process.

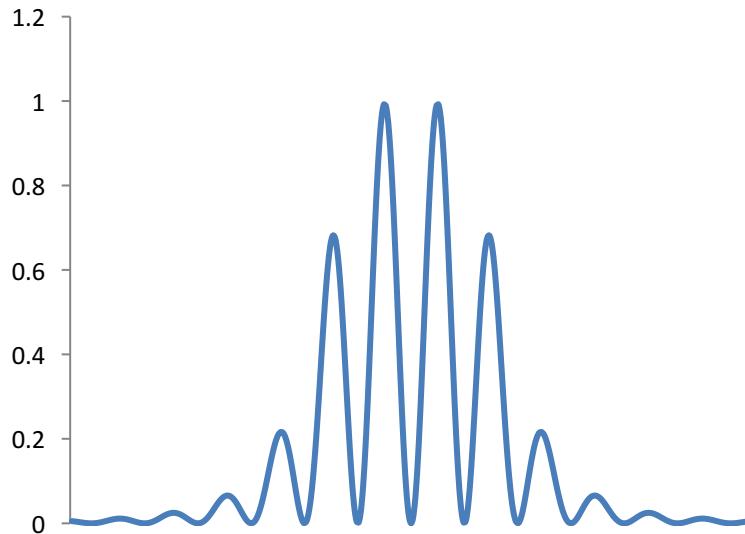
A Genetic Algorithm needs both, so that exploitation and exploration are combined.

# An Example (finding the maximum point)

Consider the two argument function:

$$f(x, y) = \frac{(\sin \sqrt{x^2 + y^2})^2}{1 + 0.001(x^2 + y^2)^2}$$

A plot of this function as  $x$  is varied looks something as shown in this figure:



What  $x$  and  $y$  make  $f(x, y)$  maximum?

This function is difficult to maximise because it has a very large number of local maxima and minima.

# Problem Formalisation for a GA Solution

## Encoding candidate solutions

Assume we know that around the maximum value of  $f(x,y)$  the values of  $x$  and  $y$  are:

$$-100 < x < 100$$

$$-100 < y < 100$$

We could represent a candidate solution as a binary string of length  $2L$ :

The first  $L$  bits represent  $x$ , and  
the remaining  $L$  bits represent  $y$ .

## Fitness function

Simple: Use the actual value of  $f(x,y)$ .

In general, **problem formalisation** includes: 1) solution representation,  
2) fitness function.

# Results

Using a population of 100 chromosomes (or candidate solutions represented as strings of  $2L$  bits,  $L=16$ ), run genetic algorithm for 40 generations, with evaluation-selection-reproduction in each generation.

Fitness of best five chromosomes:

After 1 generation

0.9903, 0.9893, 0.9100, 0.8697, 0.8241

After 4 generations

0.9823, 0.9823, 0.9774, 0.9758, 0.9758

After 40 generations

0.9930, 0.9926, 0.9925, 0.9925, 0.9923

The corresponding chromosomes give the values of  $x$  and  $y$  at the maximum point.

# How about Training a Neural Network?

## Encoding candidate solutions

Assume there are  $N$  connection weights (including thresholds) in the neural network. Use  $L$  bits to represent one weight and thus  $N \times L$  bits to represent all the weights. A chromosome of  $N \times L$  bits can represent a candidate neural network.

## Fitness function

Can be simply the accuracy of the neural network on the training dataset.

## Use of genetic algorithm

Initialisation: Generate a population of  $m$  random chromosomes (candidate neural networks)

Cycles of ‘evaluation-selection-reproduction’ ( $n$  generations)

(The values of  $N$ ,  $L$ ,  $m$ ,  $n$  depend on the size of the neural network and the experimental setup for the genetic algorithm)

# Genetic Programming (Optional)

Genetic programming (GP) is a development from genetic algorithm (GA), in which chromosomes represent computer programs.

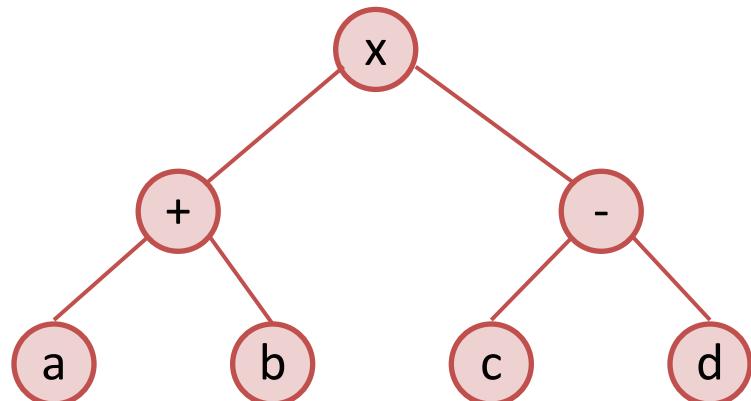
If GA is directly used, a program could be viewed as a **linear sequence of characters**, but this removes most of the structure of the program.

In GP, programs are typically represented as syntax trees.

Consider the following expression:

$$(a + b) \times (c - d)$$

This could be represented as a tree as shown in the figure on the right.



Therefore, in GP the population that evolves is made up of **trees rather than strings. (different problem representations)**

# Operators for Genetic Programming

## Crossover

The simple biologically inspired mechanism of crossover is fine for **strings** but must be modified to deal with **syntax trees**.

Given two parent trees,  $T_A$  and  $T_B$ .

Randomly select a **node** in each tree,  $N_A$  and  $N_B$ .

Swap the **subtrees** starting at  $N_A$  and  $N_B$  to make two new trees.

Many nodes of a syntax tree could be leaf nodes.

Swapping these produces only minor change.

Therefore, node selection in GP is not uniform.

It is biased to select non-leaf nodes.

# Operators for Genetic Programming (2)

## Mutation

### Subtree mutation

Mutation point in the tree is replaced by a randomly generated subtree.

### Point mutation

Changes the content of the chosen node only, such as compatible operands or arguments.

# **Summary**

## **Biological Basis of Genetic Algorithm**

Gene and Chromosome

Mutation and Crossover

Natural Selection and Fitness

## **A Basic Genetic Algorithm**

Cycles of “Evaluation – Selection – Reproduction”

(a new approach to “generate and evaluate”)

## **Operators for Reproduction**

Mutation

Crossover

## **Genetic Programming (optional)**

Syntax Trees

Operators (mutation and crossover)

# CE213 Artificial Intelligence – Lecture 20

## Intelligent Agents

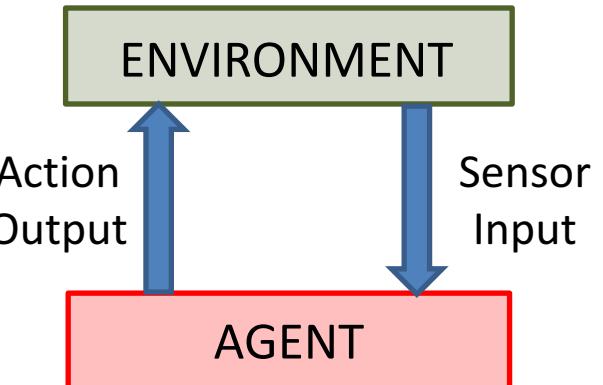
Integration of various AI approaches to perception / environment understanding (NN ), state-action mapping (reinforcement learning, NN), and decision making (decision tree, expert system, NN).

## Architectures for Intelligent Agents and Multi-agent Systems

# What Are Agents?

An agent is a system (hardware or software) that is

- situated in an environment
- capable of autonomous action in that environment to achieve goals or objectives



**Autonomy** (autonomous action) implies that it can act without the intervention of humans or other agents.

Agents must be able to:

- **perceive/understand their environments**
- **act upon their environments**

Basic functions of an agent are perception, decision making, and action. Therefore, various AI methods may be required in an agent.

# Simple Examples of Agents

## Central heating thermostat

Environment: Building whose heating it regulates.

Sensor: Temperature sensor

Actions: Turn heating on  
Turn heating off

## Pressure cooker safety valve

Environment: Pressure cooker and its surroundings.

Sensor: Pressure sensor

Actions: Open valve  
Close valve

# Simple Examples of Agents (2)

## New e-mail demon

Environment:	Operating systems and data structures
Sensor:	Unread message checker (software)
Actions:	Initiate audio warning signal

## Simple spinal withdrawal reflex

Environment:	Animal's body and surroundings
Sensor:	Pain sensors in skin
Actions:	Activate flexor muscles

*Of course, robots are agents in general.*

# Intelligent Agents

## What agents are Intelligent?

Criterion 1: Whether they can pass the Turing Test

This is too demanding to be a useful criterion

Criterion 2: Whether they are capable of ***flexible autonomous behaviour***

What do we mean by “flexible”?

- They are Pro-active:

Can take the initiative in order to achieve goals.

- They are conditionally reactive:

Can react in a timely fashion to changes in environment by selecting appropriate response from a range of alternatives.

- They are socially interactive:

Can interact with other agents (possibly including humans) to achieve collective goals.

# Agents and Expert Systems

Are expert systems intelligent agents?

Most people would say “No” because:

They do not perceive their environments directly: they rely on the human user.

They do not act directly on the environment: they give advice to the user.

However, some expert system applications could be regarded as examples of intelligent agents.

e.g., expert systems equipped with sensors and actuators for real-time monitoring and control.

# A Formal Representation of Agents

## Environment (sensor input)

Let the environment be a set of states **S** where

$$S = \{s_1, s_2, \dots\}$$

At a given instant, the environment is in one of these states.

## Actions

Let the actions available to the agent be a set of actions **A** where

$$A = \{a_1, a_2, \dots\}$$

## Agent

Then the *agent* can be viewed as a function that maps a **sequence of environment states  $S^*$**  to actions: (e.g., neural networks, control policies, etc.)

$$S^* \rightarrow A$$

## Environmental Change (action output)

The effect of actions and other sources of environmental change can then be modelled as a function:

$$S \times A \rightarrow S$$

# Types of Intelligent Agents

## Reactive Agents

Decision making is implemented as a direct mapping from *current* situation to action:

$$\begin{aligned} \textit{purely reactive agent} : S &\rightarrow A \\ & \quad (S \text{ rather than } S^*) \end{aligned}$$

A purely reactive agent has no memory of earlier environment states and thus responds only to the immediate situation.

(Similar to the Markov decision process described in reinforcement learning)

# Types of Intelligent Agents (2)

## Agents with Internal State

An agent that has some form of memory can base its action choices on both the current environment and its internal state.

We could represent this as:

$$\text{agent} : \mathbf{S} \times \mathbf{M} \rightarrow \mathbf{A}$$

where **M** is the agent's set of internal states (memory).

However, it is clearer what is happening if we represent the perception and action of the agent separately as follows:

$$\text{agent\_perception} : \mathbf{S} \times \mathbf{M} \rightarrow \mathbf{M}$$

$$\text{agent\_action} : \mathbf{M} \rightarrow \mathbf{A}$$

# Subsumption Architecture

A reactive agent architecture developed by Rodney Brooks at MIT  
([https://en.wikipedia.org/wiki/Subsumption\\_architecture](https://en.wikipedia.org/wiki/Subsumption_architecture))

Two main ideas:

## 1. *Task Accomplishing Behaviours*

Agent's decision making is realised through a set of behaviours that determine or select individual actions.

Each of these behaviours is a simple response to the current situation.

## 2. *Subsumption Hierarchy*

Behaviours are arranged in *layers*.

Will have an example later.

Lower layers can inhibit higher layers.

Thus when two or more behaviours are appropriate for the current situation, the one in lowest layer will be chosen. (conflict resolution)

Higher layers represent more abstract aspects of the problem domain or relate to longer term objectives.

# Pros and Cons of Reactive Agents

## Advantages

- *Cheap*: Computational power required for each agent is low.
- *Robust*: Loss of single agent does not seriously disrupt operation of entire set (no centralised communication/control).

## Disadvantages

- All decisions are based on purely local information, so agents must be able to obtain sufficient information locally to select best action.
- All behaviours are “short-term”. Cannot take account of information regarding states encountered earlier.
- Learning from experience is not possible (no memory).

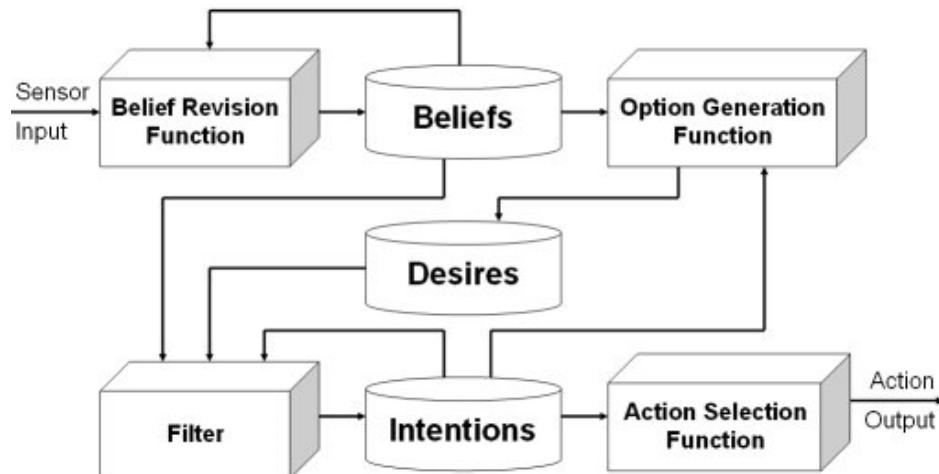
# Architecture for Agents with Internal State

(An Example Only, for self study)

## Belief-Desire-Intention (BDI) Architecture

Originally proposed by Bratman in 1987, trying to explain the human reasoning with the following attitudes: belief, desire and intention.

Decision making depends on the **manipulation of various internal data structures** representing beliefs, desires and intentions of the agent.



Belief: informational state of the agent (memory)

Desire: motivational state of the agent (memory)

Intention: deliberative state of the agent (memory)

They are updated continuously through perception

$$S \times M \rightarrow A \quad \text{or} \quad S \times M \rightarrow M, M \rightarrow A$$

# Multi-agent Systems

An important (and very interesting) class of problems are those involving more than one agent operating in the same environment.

## Interaction

They may *cooperate* or they may *compete*.

If they cooperate they may need to *construct plans* to achieve collective goals. Such planning may be done *centrally* or *distributed* across the agents.

If they compete they may need to *negotiate* to resolve conflicts.

## Communication

In both cases, cooperate or compete, they are likely to need to *communicate* with each other.

Such communication can be either

### *Direct communication:*

Agents exchange messages with each other.

or

### *Indirect communication:*

Agents communicate by acting upon the environment.  
e.g., making marks in the environment.

# An Example: Planetary Exploration (Multi-agent system in Subsumption Architecture)

## Scenario:

- The system is designed based on foraging behaviour of ants.
- A set of autonomous vehicles (agents) are landed on Mars.
- Their task is to collect samples of a rare type of rock and bring them back to the mothership.
- Locations of the rock are not known in advance, but tend to fall in clusters.
- No map of the planet is available.
- The vehicles cannot communicate directly with each other.

# Planetary Exploration (2)

## Navigation

To aid vehicles in returning, the mothership broadcasts a steady signal.

The intensity of this signal reduces as the distance from the mothership increases.

Vehicles can thus find their way back by moving in the direction that most increases the signal strength.

This navigational mechanism is called a *gradient field*.

# Planetary Exploration (3)

## Individual Behaviours (represented as production rules)

The behaviour in lowest layer should enable an agent to avoid obstacles:

IF       **detect an obstacle**  
THEN     **change direction**                    *[Layer 1]*

The next two important behaviours ensure that an agent that has found rock samples will take them back to the mothership (base):

IF       **carrying samples**  
AND      **at base**  
THEN     **drop samples**                    *[Layer 2]*

# Planetary Exploration (4)

**IF** carrying samples  
**AND** not at base  
**THEN** travel up gradient [Layer 3]

(Note that 'obstacle avoidance' can inhibit 'return to base' behaviours.)

The next behaviour ensures that an agent picks up samples that it finds.

**IF** detect a sample  
**THEN** pick up sample [Layer 4]

*(Note that by layering architecture this behaviour is inhibited once the agent has picked up samples)*

..... (some other behaviours, e.g. for communication)

The final behaviour in the highest layer, with lowest priority, produces exploratory behaviour.

**IF** true  
**THEN** make random move [Layer 10]

# Planetary Exploration (5)

## Cooperation (no competition)

The set of behaviours designed so far enables the vehicles (agents) to operate independently to perform the sample collection task.

It lacks a means to enable them to cooperate:

That is, to pass on the information they have discovered about where samples are to be found.

We can address this limitation by providing them with a simple facility for **indirect communication**.

We therefore assume that the vehicles can also:

- Drop crumbs of rock samples
- Detect crumbs of rock samples
- Pick up crumbs of rock samples

# Planetary Exploration (6)

First we can extend one of the existing rules, so that agents will lay a trail of crumbs as they return to base:

IF       **carrying samples**  
AND      **not at base**  
THEN     **drop 2 crumbs**  
            **travel up gradient**                  [*Layer 3*]

Then we can add a new rule for dealing with the situation when crumbs are encountered during exploratory behaviour:

IF       **sense crumbs**  
THEN     **pick up 1 crumb**  
            **travel down gradient**                  [*Layer 5*]

*More behaviours/rules?*

**Discussions: What are the differences from general expert systems?**

**How about perception? – Detection of obstacles, samples, crumbs, ..... (neural networks may be applied here)**

# Summary

## What Are Agents

Simple examples of agents

## Intelligent Agents

Pro-active, Conditionally reactive, Socially interactive

## A Formal Representation of Agents

Reactive agents, Agents with internal state

## Subsumption Architecture

Task-accomplishing behaviours arranged in layers

## Pros and Cons of Reactive Agents

## BDI Architecture for Agents with Internal State

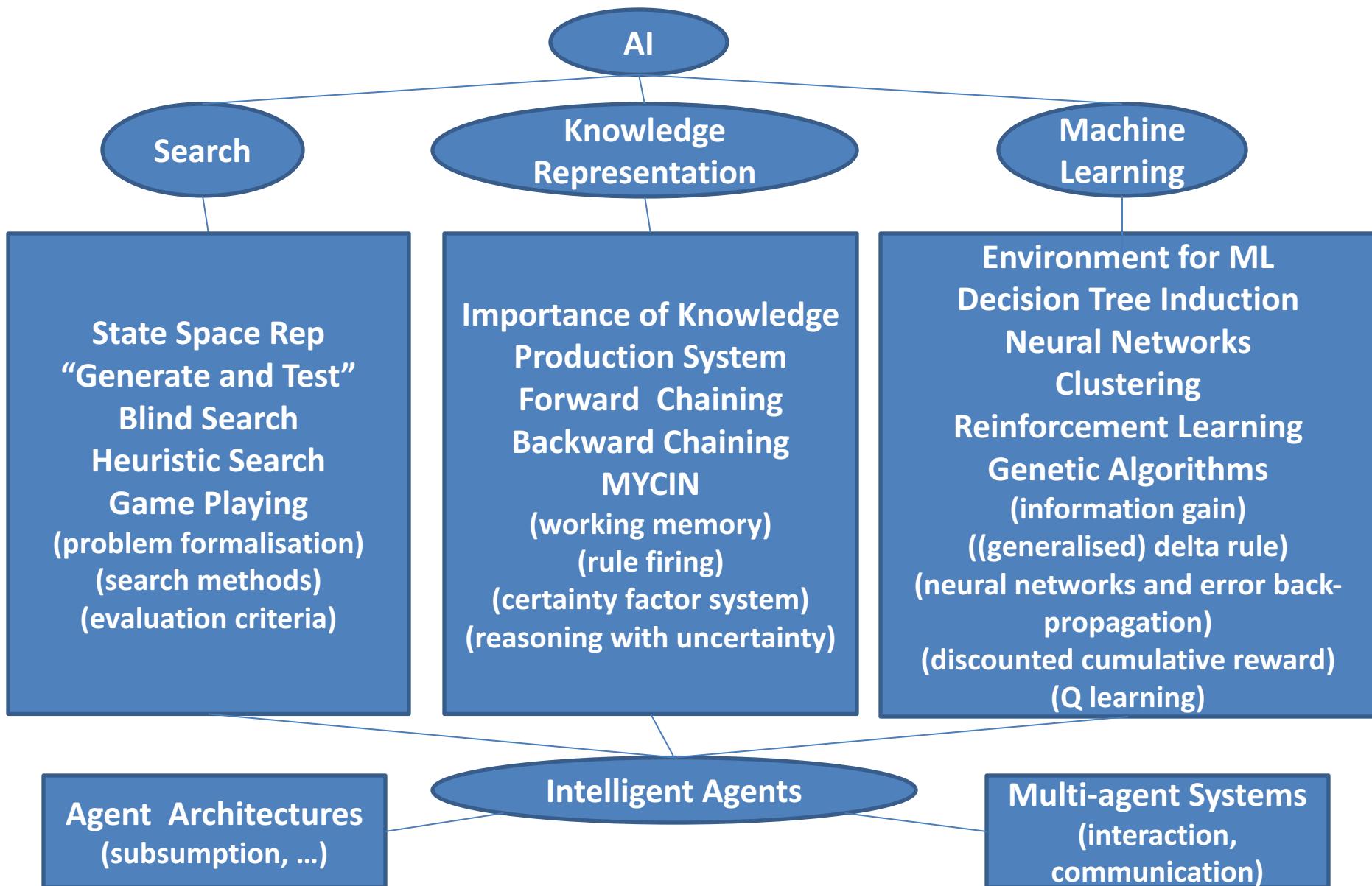
## Multi-agent Systems

Interaction: Cooperation or competition

Communication: Direct or indirect

Planetary exploration example

# Basic AI Concepts and Methods



# If you want to learn more about AI .....

Hopefully, CE213 has laid a solid foundation for possible future modules:

CE215 Robotics

CE217 Computer Game Design

CE310 Evolutionary computation and Genetic Programming

CE316/CE866 Computer Vision

CE801 Intelligent Systems and Robotics

CE802 Machine Learning and Data Mining

CE811 Game Artificial Intelligence

CE889 Artificial Neural Networks and Deep Learning.

# CE213 Artificial Intelligence – Lectures 5&6

## Game Playing

**Game playing is an excellent example of problem solving by state space search.**

**It is more complex than general problem solving due to**

- 1) the competition from the opponent;**
- 2) large size of search space;**
- 3) lack of accurate heuristics or evaluation functions.**

# Game Playing

We will be concerned with games that, like chess or Go, have the following characteristics:

- **Two players**
- **Turn-taking**
- **Deterministic**
  - The same move in the same situation always has the same effect.
- **Perfect information**
  - The entire game state is always available to both players.

Other examples include: Draughts (Checkers in US), Noughts and Crosses (Tic-Tac-Toe in US). Games like poker and bridge are not considered here.

## Game Playing (2)

Choosing the best move in such games has much in common with solving problems by state-space search:

- **Positions** in the game (game states) are **states** in the space of all possible positions.
- The starting arrangement is the **initial state** (e.g., empty board).
- Winning positions or winning endgames are **goal states**.
- Legal moves are the possible **transitions/operations**.

However, there is one big difference between the search strategies for game playing and for general problem solving.

# Adversarial Search – Challenge 1

## The opponent

- Normally has very different goals.
- Selects every other move.
- Will try to stop us reaching our goal.

This form of state space search taking the opponent into account is called ***adversarial search***.

There is usually another challenge in developing search strategies for game playing.

# Large Search Space – Challenge 2

The puzzles and navigation problems we considered when discussing state space search had small state spaces:

Road map problem	11 distinct states (11 towns)
Corn goose fox problem	16 distinct states
Three jugs problem	24 distinct states (for the one we discussed)
(Eight puzzle	9! distinct states)
(Fifteen puzzle	16! distinct states)

Non-trivial games have vastly larger state spaces:

Chess has about  $10^{40}$  distinct states.

Go has  $3^{361}$  distinct states, and  $2.08168199382 \times 10^{170}$  legal game positions.

Even noughts and crosses has  $3^9 = 19683$  distinct states (although many of these are illegal, in which ' $| \text{no. of noughts} - \text{no. of crosses} | \leq 1$ ' is not true.)

# Solutions

Adversarial search: Minimax search

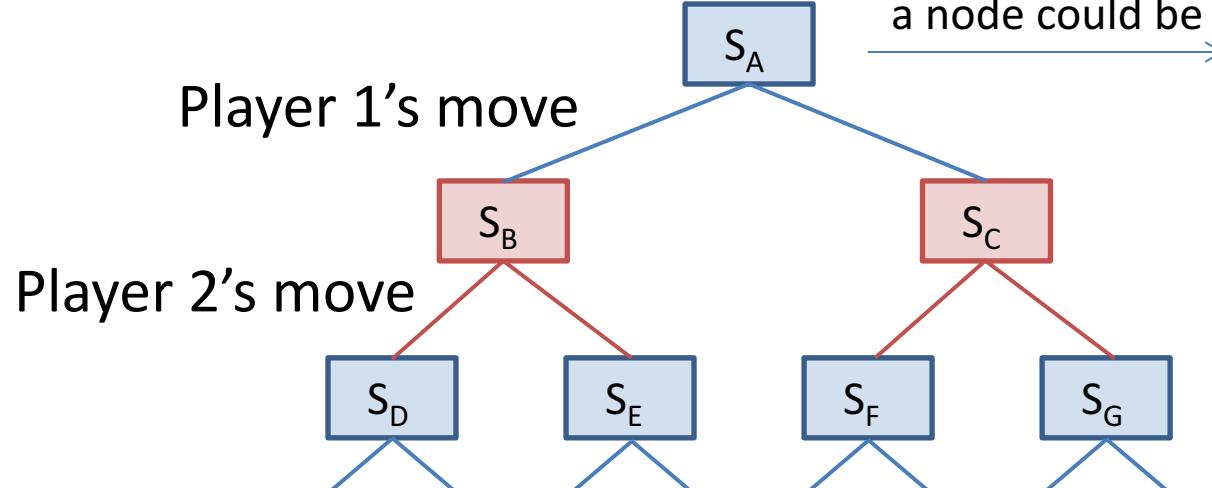
Large state spaces: Evaluation functions (Heuristics)

*Challenge 2 is mainly about how to evaluate game states  
(In this lecture we mainly address Challenge 1)*

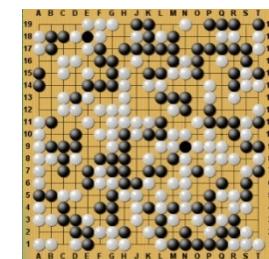
*Effective and efficient approach to “generate and evaluate”  
is a key AI research topic.*

# Game Tree – Search Tree for Game Playing

A simplest game tree for the first two moves of a possible game:



or



There are two types of moves that aim at different goals!  
(For convenience, let's assume Player 1 is AI and Player 2 is Opponent)

# Minimax Search

If the opponent could be trusted to help us, then the problem would be easy:

An exhaustive search to the depth limit would reveal the best move.

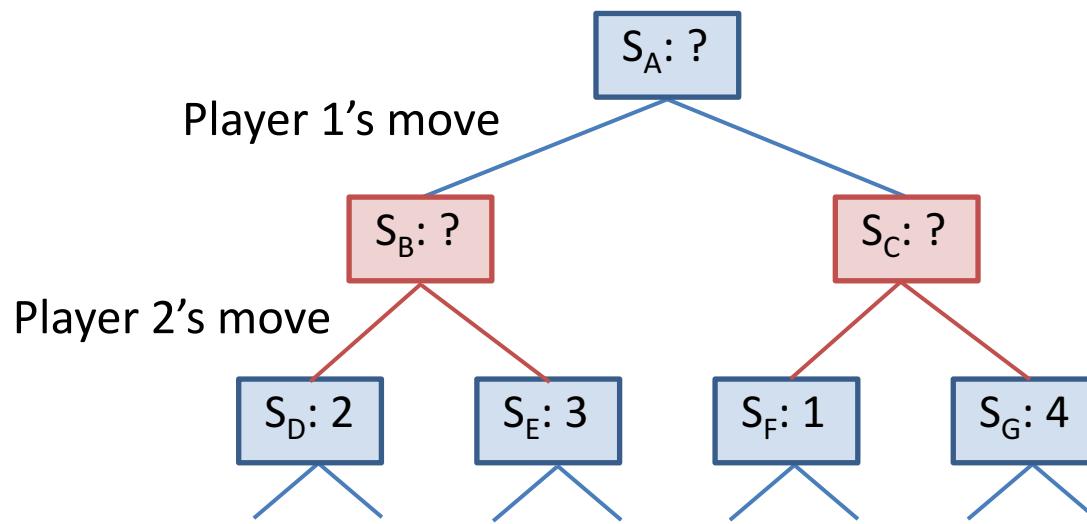
But in fact we expect the opponent to *hinder* us.

How can we take account of this while carrying out the search?

Different from the search strategies learnt so far, in minimax search we need to check the nodes at deeper levels before choosing which move to make or which node to expand at the current depth in game tree construction.

## Minimax Search (2)

It is well-known that it is easier to evaluate game states deeper in the game tree as they are closer to endgame. So, suppose that an evaluation function has given the following values to the nodes at depth 2, as shown in the game tree, but it is hard to evaluate the values of the nodes at depth 1 directly. We need to get the values of states  $S_A$ ,  $S_B$ , and  $S_C$  by minimax search.



Note that all the values are to Player 1 (the larger the better to Player 1). (You may ignore how these values are generated at the moment. They are game-dependent. Here, these values are hypothetical only.)

## Minimax Search (3)

Player 2 will choose a move so as to *minimise* the value to Player 1.

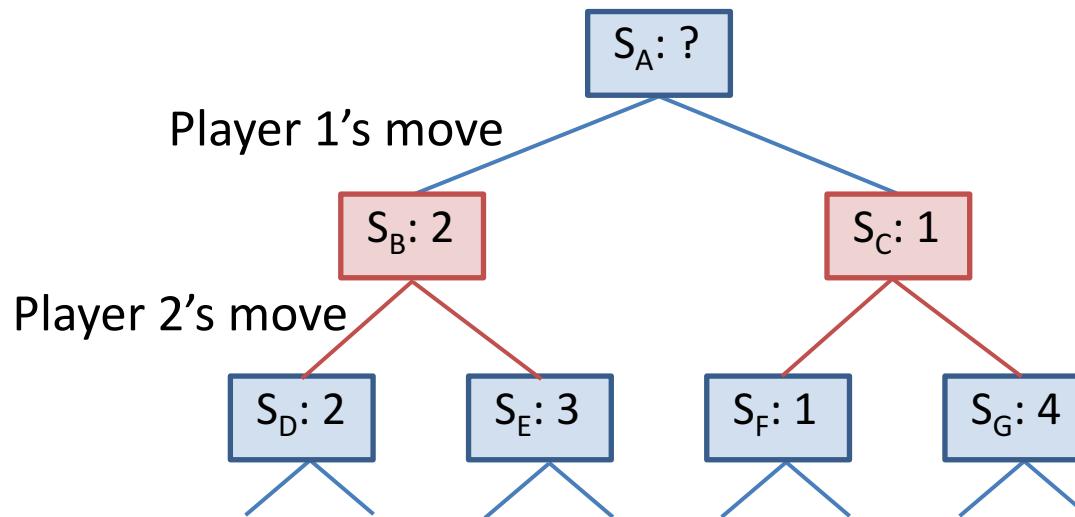
In game state  $S_B$ , Player 2 will choose to go to  $S_D$ .

Thus the potential value of  $S_B$  to Player 1 is only 2.

In game state  $S_C$ , Player 2 will choose to go to  $S_F$ .

Thus the potential value of  $S_C$  to Player 1 is only 1.

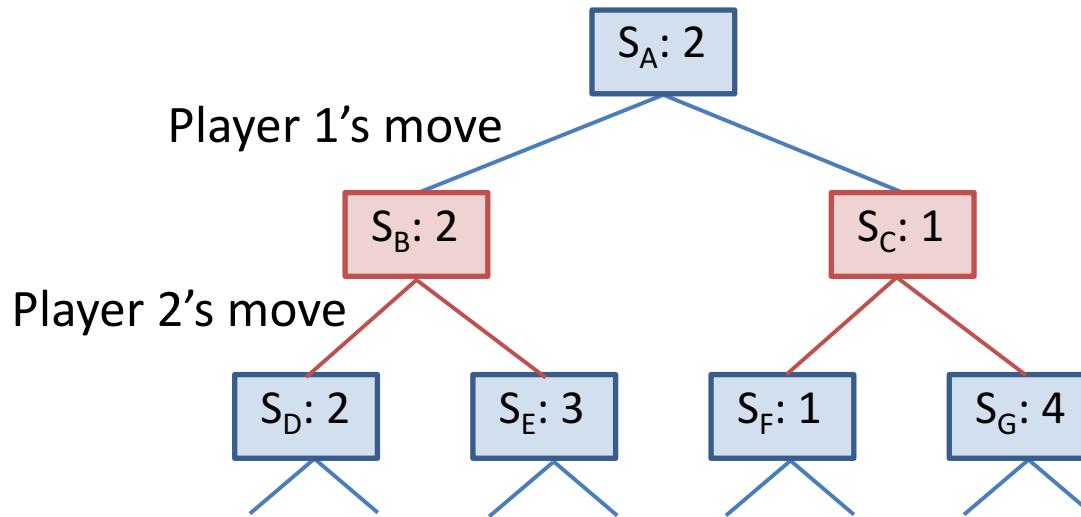
So we can include these values of  $S_B$  and  $S_C$  in the search tree:



## Minimax Search (4)

Player 1 should choose a move so as to *maximise* the potential value of the chosen game state. Clearly in this case Player 1 should choose to go to  $S_B$ .

Thus the potential value of  $S_A$  to Player 1 is 2, as shown in the game tree now.



# Minimax Search (5)

This process of passing the evaluation values of game states (nodes) back up the tree is called ***minimaxing***, or ***minimax*** search:

For nodes where opponent makes the move, pass back the minimum value.

For nodes where AI player makes the move, pass back the maximum value.

Minimaxing can be applied to any depth, with iterations of minimax search for every 2 depths. The deeper, the better, because the evaluation of deeper nodes is more accurate in general in game playing.

Note that minimaxing places a lower bound on how badly the AI player will do, because it is assumed that the opponent would make perfect moves – the worst case scenario.

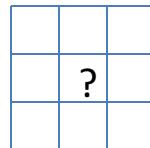
If the opponent departs from minimaxing, the AI player may do better. Therefore, AI player is guaranteed not to do worse.

# Minimax Search (6)

**Why not just apply the evaluation function to the directly reachable game states?**

Doing this would be fine if the evaluation function is perfect.

In practice it will only give an approximate indication of how good a game state is. The evaluation would be more accurate in more depth where the game states are closer to endgame.



Which of the 9 possible first moves in Tic-Tac-Toe is better?

o	o	x
x	x	o
		x

Which of these 2 game positions is better for Player 'x'?  
(see next slide)

o	o	x
x		o
x		x

**How deep should the search go?**

How much time have you got? The deeper the better!

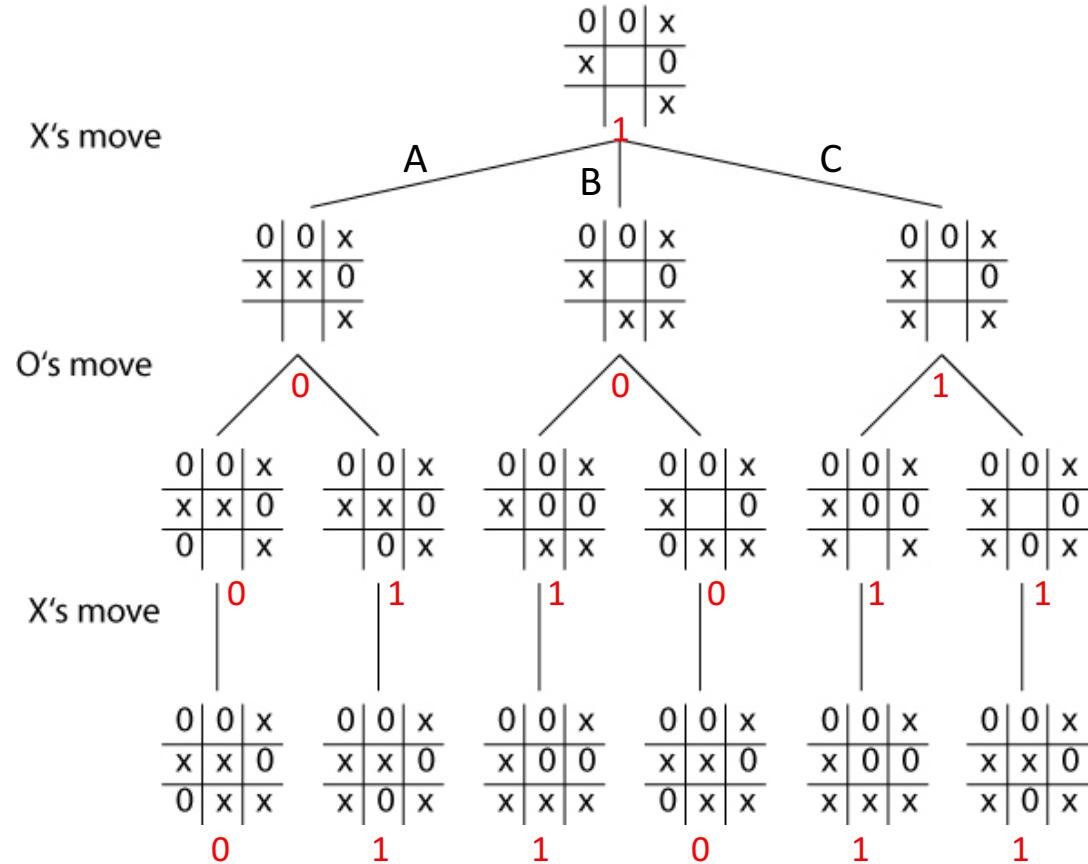
(It is easy to understand this depth requirement if you play chess.)

# An Example of Minimax Search

There are 3 possible moves for player X (player O is the opponent here). Which move is the best?

We can expand nodes to reach endgame and evaluate endgame states: 1 for win, 0 for draw, and -1 for loss.

Then conduct the minimaxing procedure: maximising when it is X's move; minimising when it is O's move.



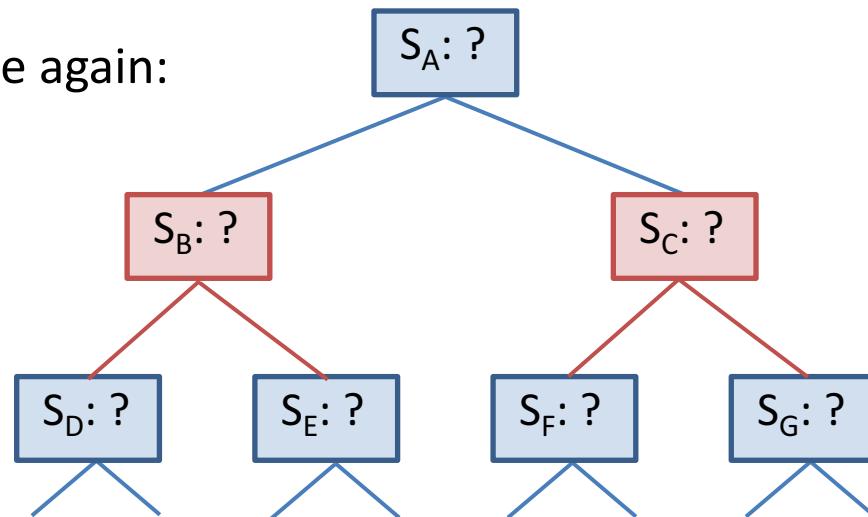
You may try to build up a game tree by starting with an empty board. How many nodes would be there in such a tree?  $\sum_{n=0}^9 \frac{9!}{(9-n)!}$

Some nodes may represent same game states!

# alpha-beta Pruning (how to avoid unnecessary evaluation?)

For many practical games, there are a large number of game states, and minimax search may be too slow in finding best moves. However, it is not always necessary to consider every node in the game tree. Taking this into account, the alpha-beta pruning algorithm can speed up minimax search.

Consider this simple game tree again:



Assume we will work from left to right.

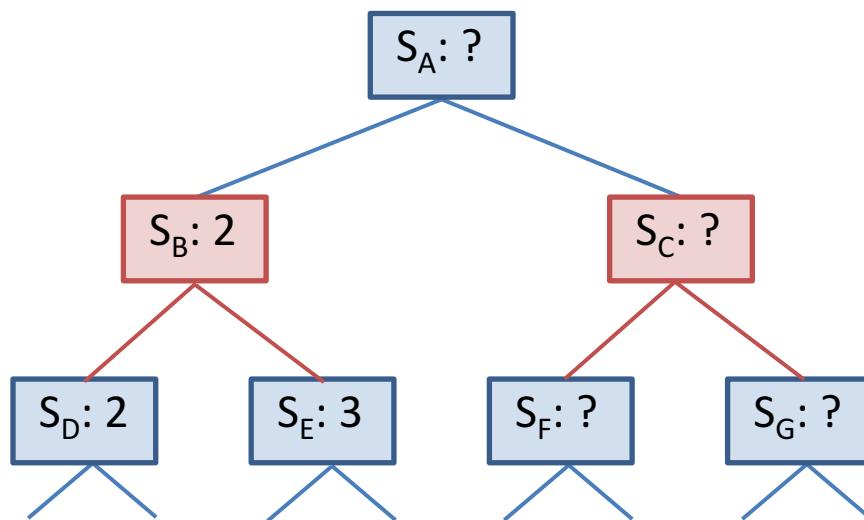
Which nodes at the bottom (at depth 2) need to be evaluated in order to get values returned to nodes S<sub>B</sub> and S<sub>C</sub>?

## alpha-beta Pruning (2)

First we back up the appropriate value to node  $S_B$  by **minimising**.

This means we must evaluate nodes  $S_D$  and  $S_E$ .

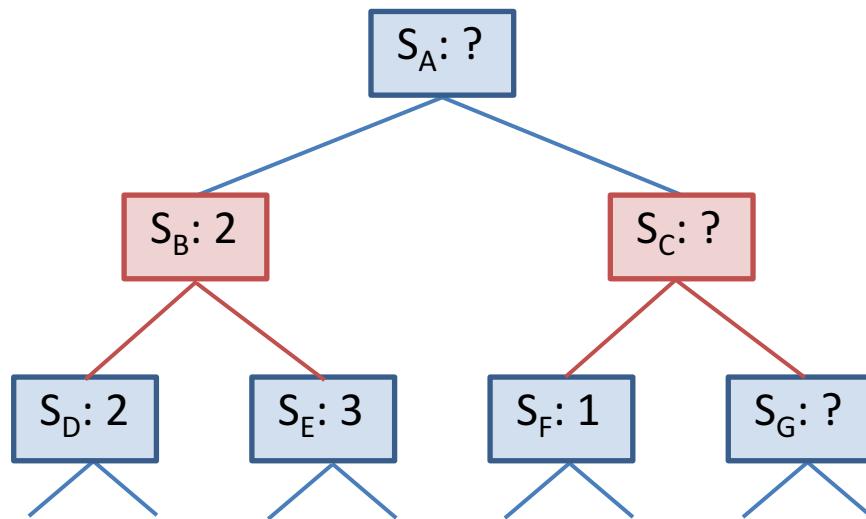
Let us **assume** this leads to the situation, as shown in the following figure. So, the value returned to  $S_B$  is 2.



N.B. Whenever possible, return min or max value back to parent node before evaluating new nodes at the deeper level.

## alpha-beta Pruning (3)

Next we consider node  $S_C$ , which depends on the values of nodes  $S_F$  and  $S_G$ . Suppose node  $S_F$  has a value of 1:



N.B. The value of  $S_F$  is compared to the value of  $S_B$  (not the values of the nodes at the same depth).

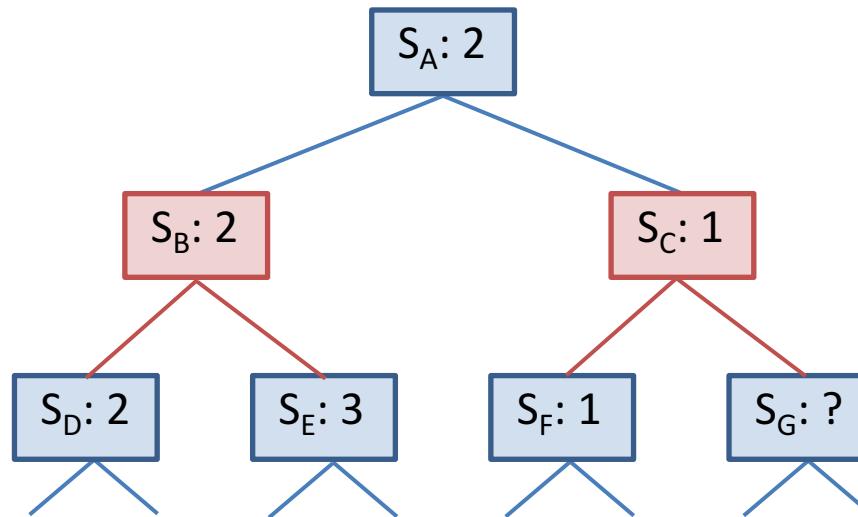
Since we are minimizing, we now know that the value of  $S_C$  is equal to or smaller than 1.

This means that  $S_B$  (with value of 2) is bound to be a better choice when we come to maximise for the value of  $S_A$ .

***So we do not need to know the value of  $S_G$ .***

## alpha-beta Pruning (4)

Hence, we can get the value of  $S_A$  without evaluating  $S_G$ .



This process of skipping unnecessary evaluations is called ***alpha-beta pruning***. (alpha, beta correspond to minimising and maximising processes)

It can be performed at every level of the tree and thus may save considerable time, especially when the branching factor is large.

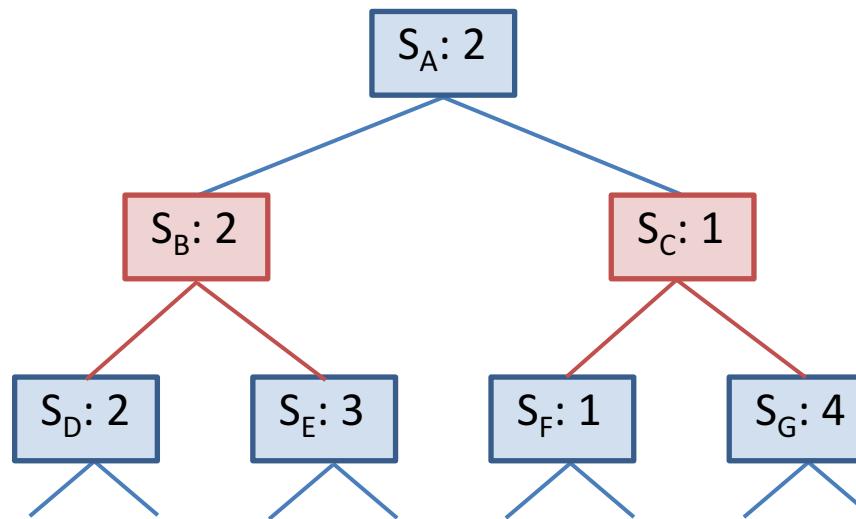
# alpha-beta Pruning (5)

**How much effort does alpha-beta pruning save?**

It depends upon the order in which the nodes are considered.

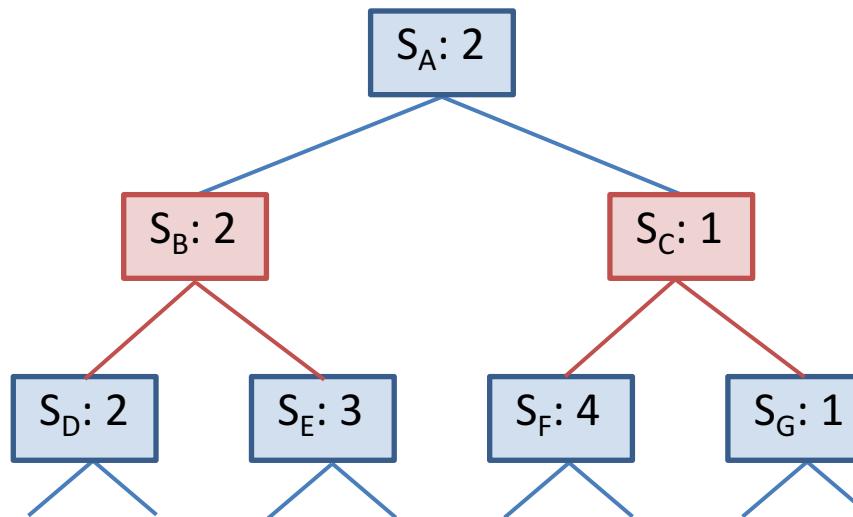
Suppose that  $S_G$  actually has a value of 4.

As we have seen, in this case there would be no need to evaluate it.



## alpha-beta Pruning (6)

However, if the values of  $S_F$  and  $S_G$  are transposed, then there would be no saving through alpha-beta pruning in this simple case, because both  $S_F$  and  $S_G$  would have to be evaluated in order to return the correct value to  $S_C$  (Because the value of  $S_F$  is larger than the value of  $S_B$ , it cannot be returned to  $S_C$  without knowing the value of  $S_G$ ).



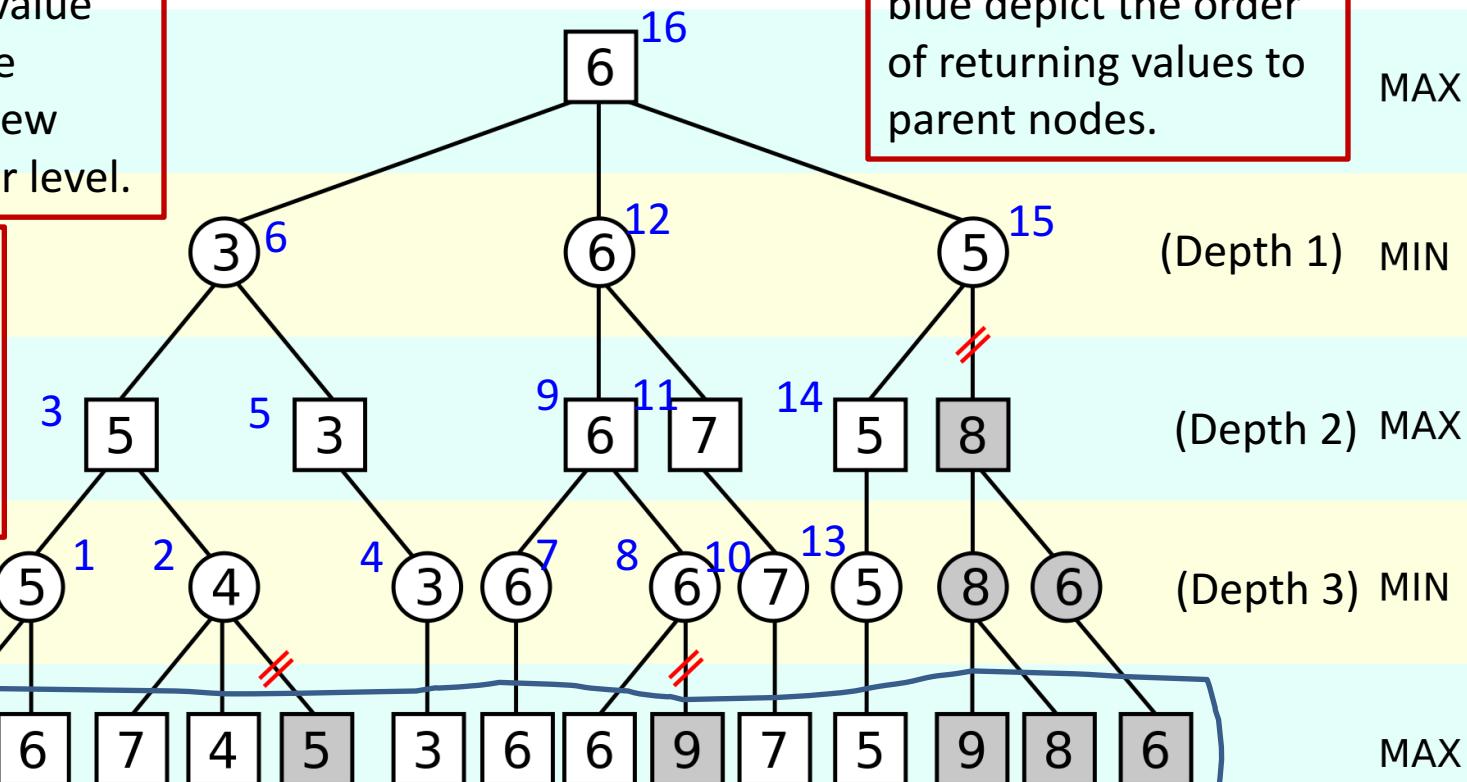
# An Example of Minimax Search with alpha-beta Pruning

N.B. Whenever possible, return min or max value back to parent node before evaluating new nodes at the deeper level.

N.B. Compare value of currently evaluated node with values of its 'uncle nodes'.

N.B. The numbers in blue depict the order of returning values to parent nodes.

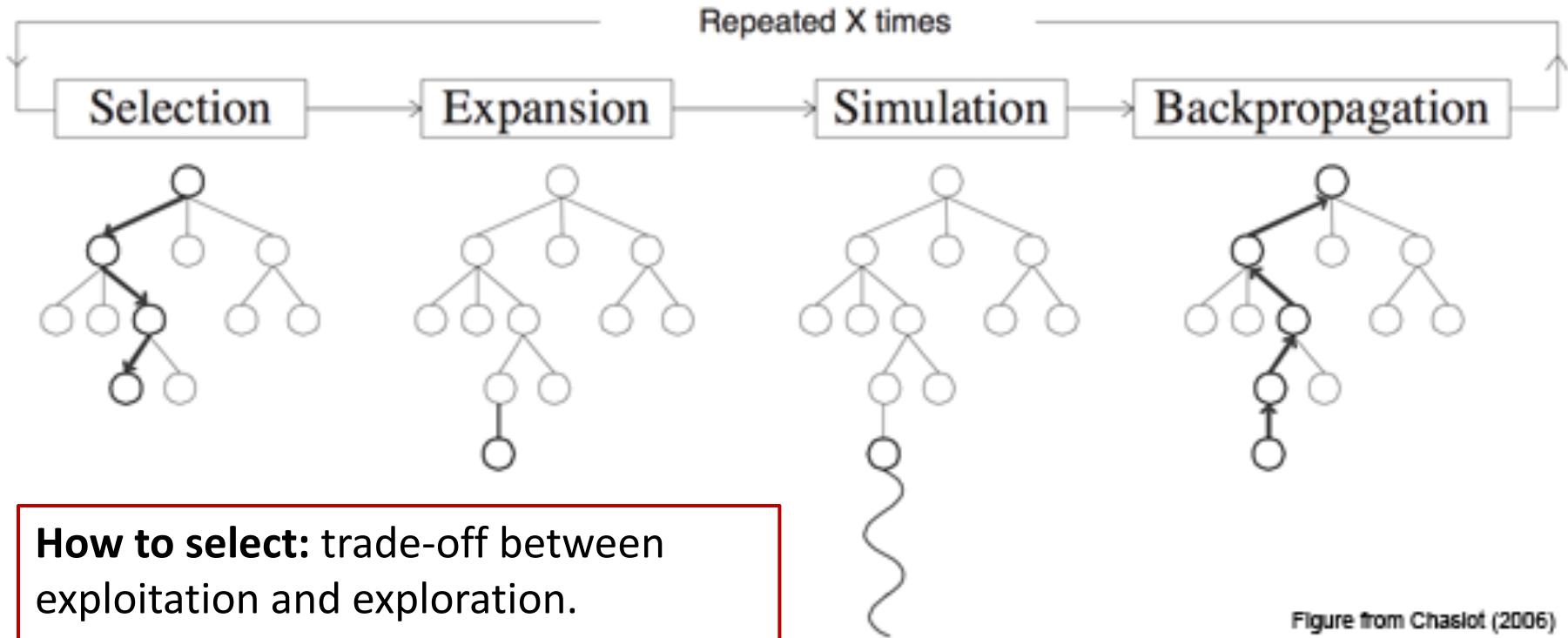
Only these values are directly obtained by heuristics.



The effectiveness and efficiency of minimax search with alpha-beta pruning heavily depend on how good the game state evaluations are. Monte-Carlo tree search provides game state evaluation by running simulated games.

# Monte-Carlo Tree Search (MCTS) – Basic Ideas

- Updating values of states by running simulated games



**How to select:** trade-off between exploitation and exploration.

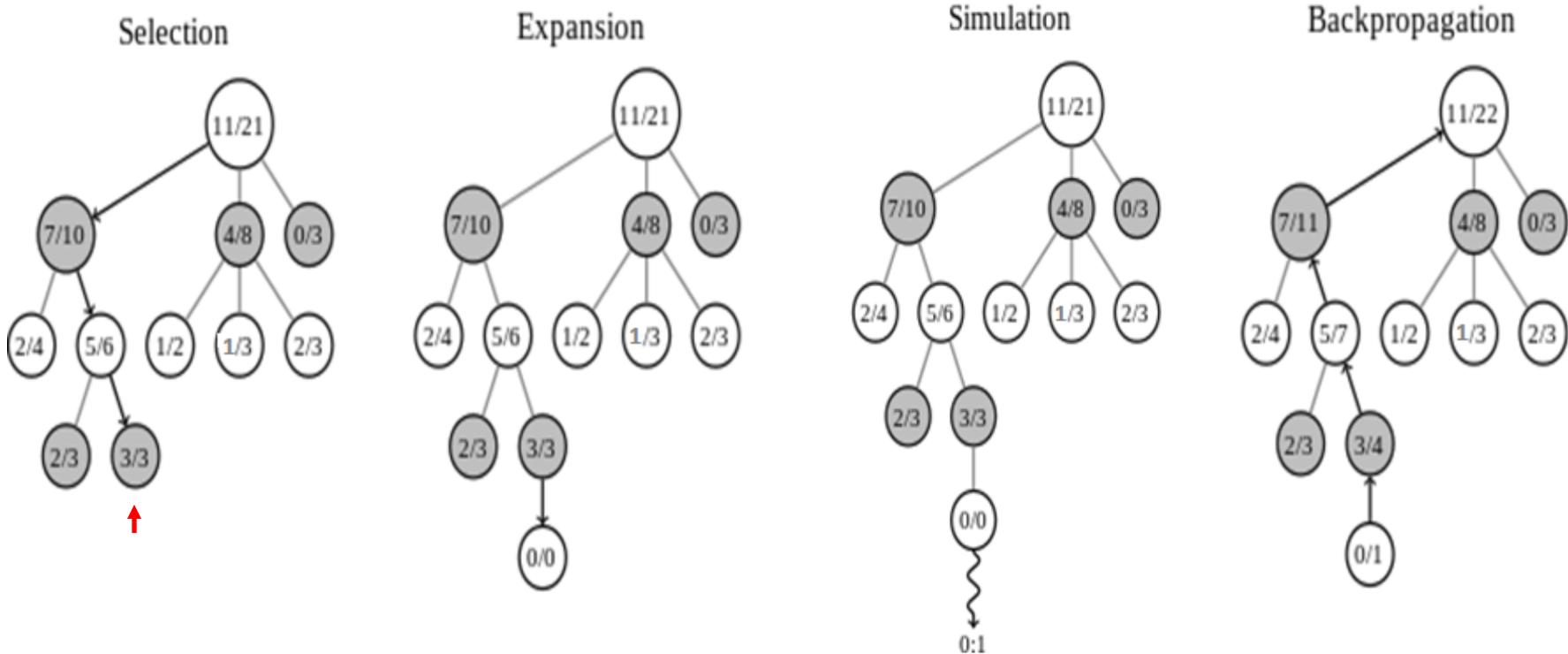
**How to expand:** possible new legal moves.

**How to simulate:** randomly making moves from the selected node to end of game, resulting in win, loss or draw.

**How to back-propagate:** updating number of wins and number of visits for each node that has been visited in the simulated game.

Figure from Chaslot (2006)

# An Example of a Round of MCTS



One of the key issues is still which node to **select** for expansion. Two methods:

1. Select the child node of the root node, which has the largest **value** (The node's value may be defined in various ways. See next slide.), then select the child node of the previously selected node, which has the largest value, and repeat this until a leaf node is selected.
  2. Select the node that has the largest value among the unexpanded nodes.
- The second approach is simpler, but may not work well with some game trees!**

# Selection in MCTS Using Upper Confidence Bound

Exploration-Exploitation Tradeoff:

$$UCB = \frac{w_i}{n_i} + C \sqrt{\frac{\ln(t)}{n_i}}$$

Exploitation      Exploration

$\ln$  : natural logarithm.

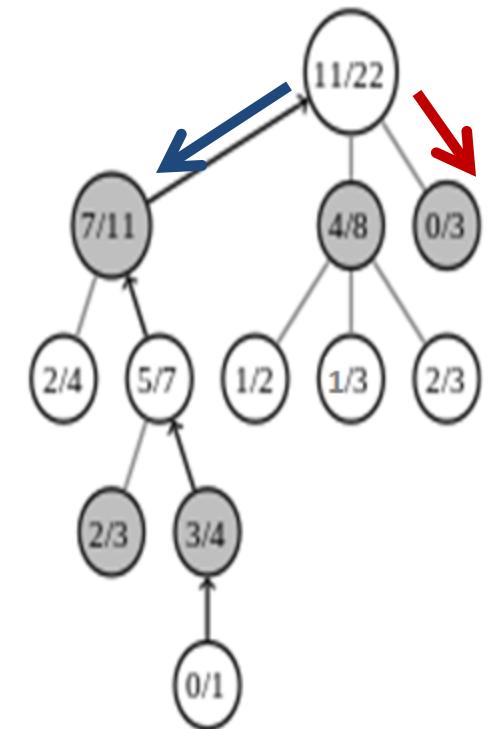
$w_i$  : no. of wins after visiting node  $i$

$n_i$  : no. of times node  $i$  has been visited.

$C$  : exploration factor. In theory,  $C = \sqrt{2}$ .

$t$  : no. of times the parent of node  $i$  has been visited,  
i.e.,  $t$  is equal to the sum of  $n_i$ .

(e.g., for the nodes in depth 1:  $w_1=7, n_1=11, w_2=4, n_2=8, w_3=0, n_3=3, t=22$ )

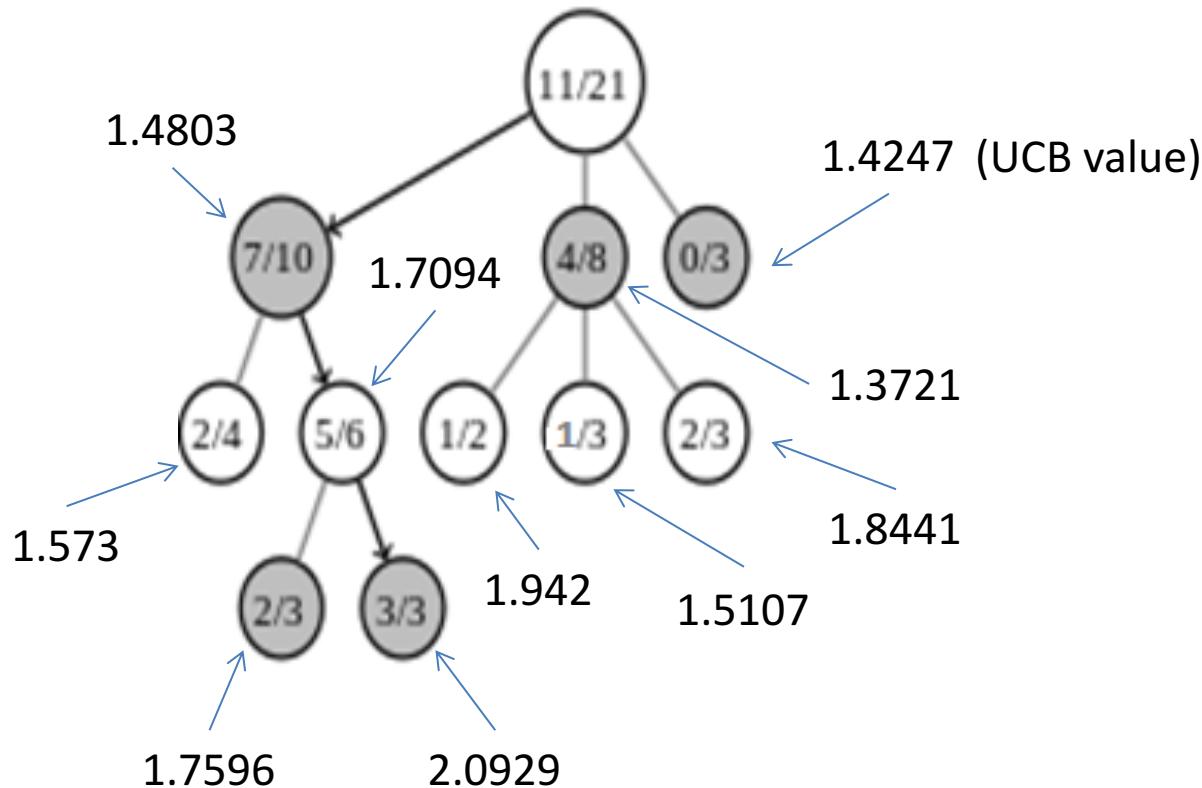


Emphasising exploitation: selection in favour of nodes with higher average win ratio, e.g., as shown by the blue arrow in the figure.

Emphasising exploration: selection in favour of nodes with fewer visits, e.g., as shown by the red arrow .

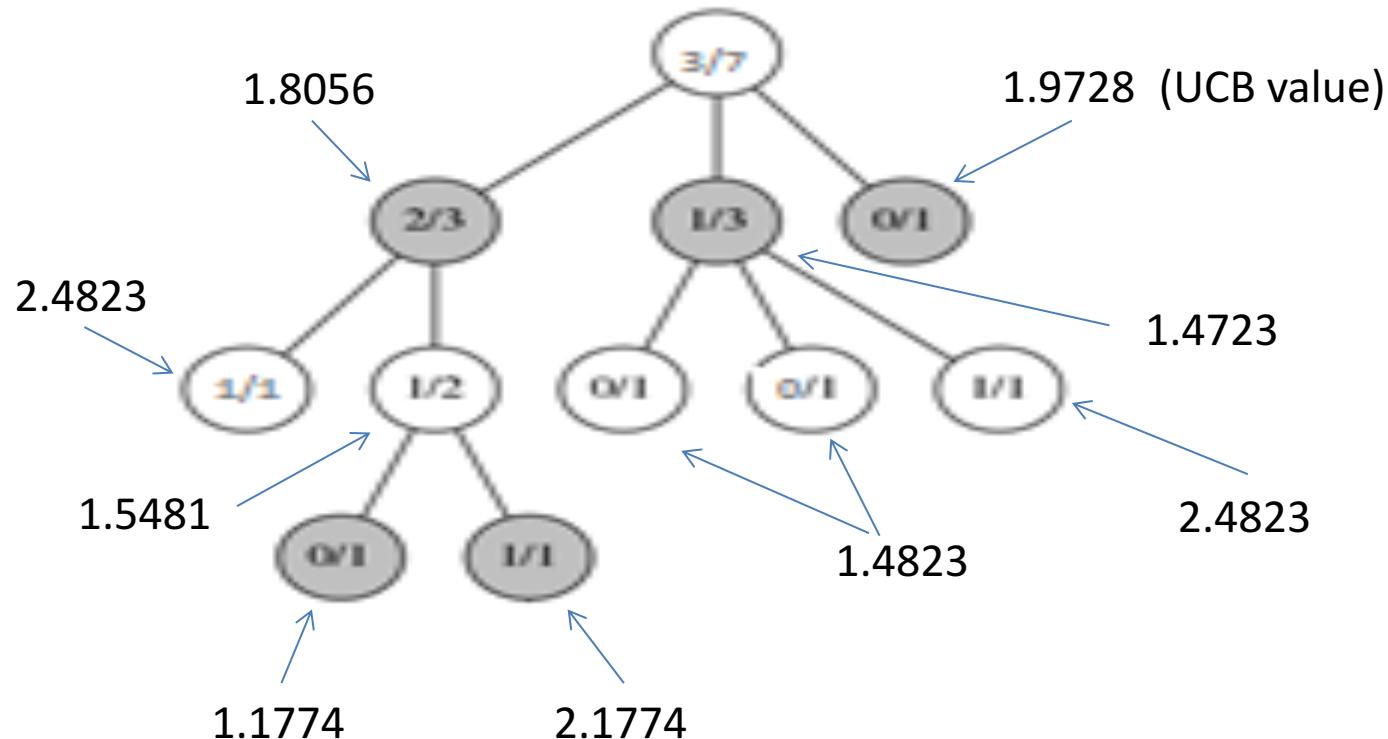
## An Example of Node Selection in MCTS

Why is the node with win ratio of 3/3 selected in slide 23?



Both methods described in slide 23 obtain the same result:  
The node with win ratio of 3/3 is selected.

## Another Example of Node Selection in MCTS

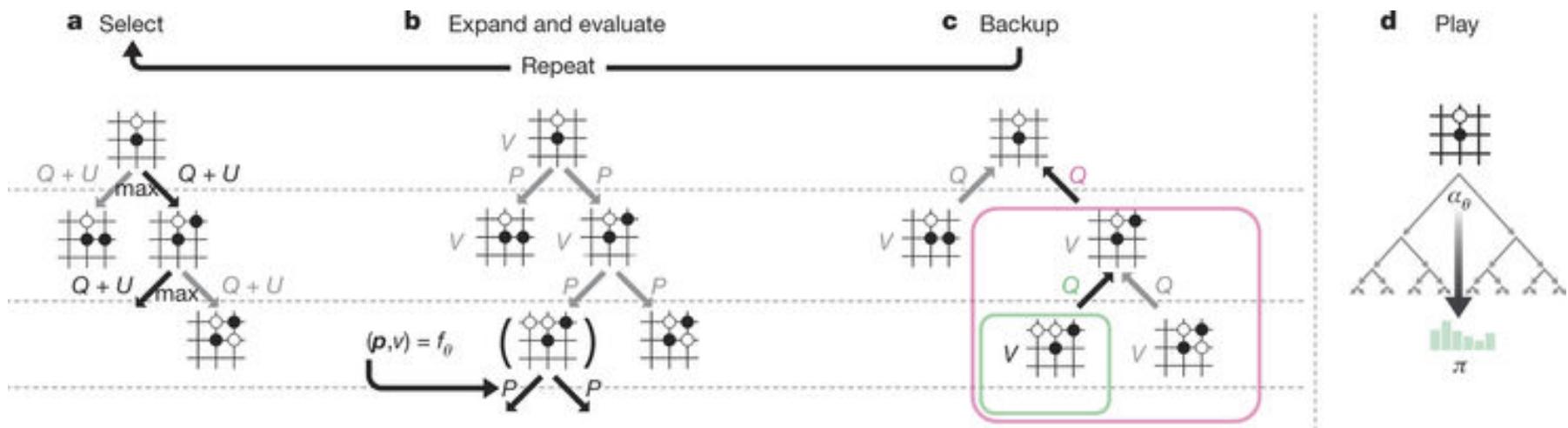


If we select among the child nodes of the root node first, the child node with  $UCB=1.9728$  should be selected for expansion. It is unnecessary to consider the other nodes because this child node of the root node is unexpanded.

However, if we select the node that has the largest UCB value among the unexpanded nodes, then either of the two nodes with  $UCB=2.4823$  can be selected for expansion next. The two methods select different nodes in this case.

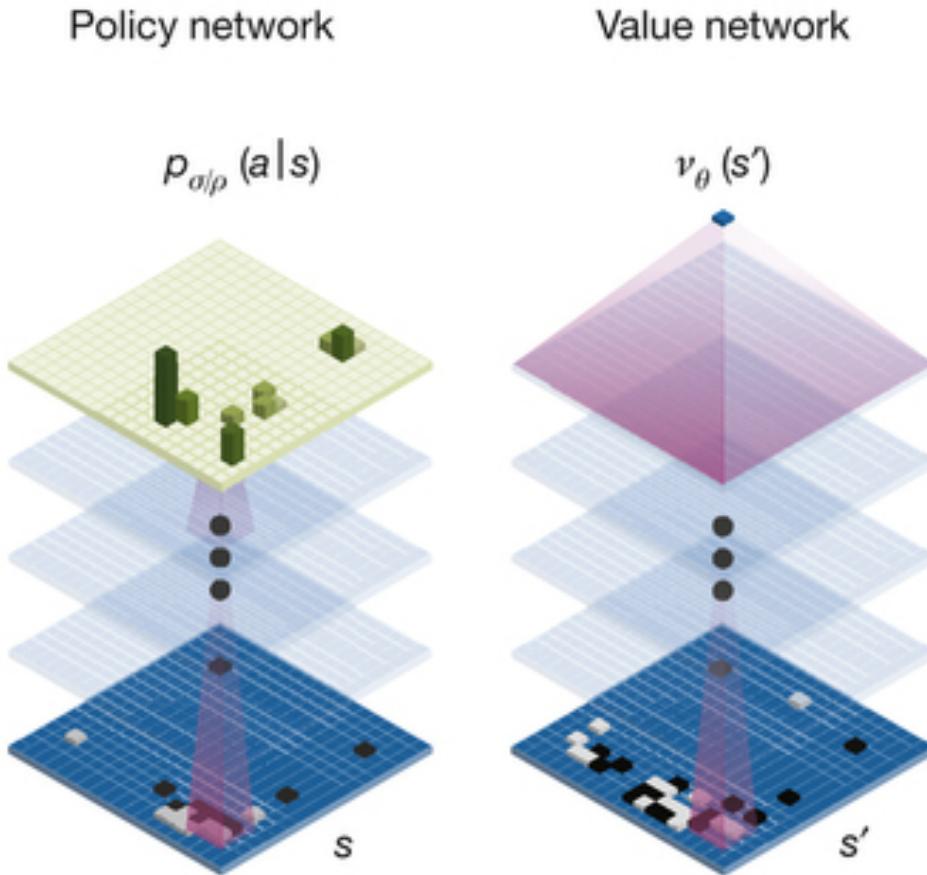
# Application of MCTS in AlphaGo Zero

David Silver et al, Mastering the game of Go without human knowledge  
Nature, 550, 354–359, 19 October 2017



Value neural network (for selection) and policy neural network (for expansion and play) are combined with MCTS in AlphaGo, making MCTS more powerful.

# Success of AlphaGo: MCTS + Machine Learning



Reinforcement learning, deep learning, Monte-Carlo Tree Search are combined for game playing in AlphaGo.

They are based on huge amount of data/experience from Go experts and self-playing.

Powerful cloud computing makes the complex machine learning feasible.

After learning, the value network could outperform world-leading Go experts in evaluating game positions.

The learnt policy network can choose optimal moves given current game position.

## Further Reading on MCTS

MCTS has been a hot research topic in the past decade. There have been many modified MCTS methods, such as new node selection criteria and game state evaluation methods, especially those in the development of AI player for the game of Go.

Here are two survey articles on MCTS:

[https://moodle.essex.ac.uk/pluginfile.php/797117/mod\\_resource/content/1/Monte-Carlo Tree Search - A New Framework for Game AI 2008.pdf](https://moodle.essex.ac.uk/pluginfile.php/797117/mod_resource/content/1/Monte-Carlo%20Tree%20Search%20-%20A%20New%20Framework%20for%20Game%20AI%202008.pdf) (short)

[https://moodle.essex.ac.uk/pluginfile.php/796529/mod\\_resource/content/2/mcts-survey.pdf](https://moodle.essex.ac.uk/pluginfile.php/796529/mod_resource/content/2/mcts-survey.pdf) (long)

# How to program a computer to beat (almost) anyone at chess/go?

Three key ideas in traditional AI can be useful:

- Minimax search

- Alpha-beta pruning

- Evaluation functions for evaluating game positions

Modern approaches focus on game position evaluation:

- Monte-Carlo Tree Search and/or machine learning  
(no need of explicit evaluation function).

# Grandmaster Chess Programs (for self study)

In 1997, IBM's Deep Blue beat Kasparov, the then world champion, in an exhibition match.

How was this possible?

Like all chess playing programs, Deep Blue is based upon the ideas we have already discussed.

The rules of chess tournaments impose time limits, so such programs do their minimaxing using an iterative deepening technique.

They keep going deeper until time runs out.

They examine the whole search tree to their depth limit.

[This slide and the next three are for information only.]

# **Grandmaster Chess Programs (2)**

Deep Blue made very effective use of these basic techniques in two ways:

## **Evaluation function: (the most difficult part)**

Very sophisticated

8000 features

(indicating what are good or bad game positions / states)

## **Special Purpose Hardware (huge speed and memory)**

30 RS/6000 processors doing the alpha-beta search

480 custom VLSI processors doing move generation and ordering.

Searched 30 billion positions per move

Typical search depth: 14 moves

In some circumstances the depth reached 40 moves

# Grandmaster Chess Programs (3)

Deep Blue also used another technique that dates back to the 1950s:

Databases

Deep Blue had:

An opening book of 4000 positions

An endgame database containing solutions for all positions with 5 or fewer pieces

Copies of 700,000 grandmaster games

All of these enable much searching to be eliminated by immediately indicating the best move for a given position.

***Making good use of information could greatly improve search efficiency!***

# Grandmaster Chess Programs (4)

More recent programs running on special purpose hardware have continued to beat grandmasters:

Hydra ([https://en.wikipedia.org/wiki/Hydra\\_\(chess\)](https://en.wikipedia.org/wiki/Hydra_(chess)))

Deep Fritz ([https://en.wikipedia.org/wiki/Fritz\\_\(chess\)](https://en.wikipedia.org/wiki/Fritz_(chess)))

Rybka (<http://www.rybkachess.com/>)

Rybka has even beaten grandmasters after giving a pawn advantage.

All of this has led to the view that success at computer chess requires special purpose hardware.

But, programs written for standard PC's have won the World Computer Chess Championship.

How? – more efficient search strategies!

# Summary

## **Minimax search:**

Most popular method for adversarial search

## **alpha-beta pruning (efficiency is essential for a huge search tree):**

Reduces effort required in state space search

## **Monte-Carlo tree search – basic ideas and steps:**

Evaluation of game positions by running simulated games

Application of MCTS in AlphaGo

## **Evaluation functions (most difficult part):**

Heuristics for evaluating game states (game positions)

Improve adversarial search in large state space

# CE213 Artificial Intelligence – Lectures 17&18

## Reinforcement Learning

No training data available: Learning from experience/exploration rather than from sample data

To learn what to do next or find a sequence of actions

Applications: game playing and robotic control

Focus: evaluation of states and actions

[Warning: It may be difficult to understand the Q learning concept and procedure]

# Reinforcement Learning

## WHY IS REINFORCEMENT LEARNING DIFFICULT?

Consider the following learning tasks:

- A child learning to ride a bicycle.
- A person/computer learning to play chess.
- A person learning how to find a way out of a complex maze.
- A robot learning how to navigate in a complicated environment.
- .....

Are these learning tasks more difficult than classification or prediction, e.g., face recognition, medical diagnosis?

- Yes, due to the involvement of time/dynamics and no immediate teaching signal available. This would be the same for human learning.

Despite the different domains, these learning tasks have much in common.

# Reinforcement Learning (2)

**What do these learning tasks have in common?**

- This type of learning is to choose a sequence of actions that will lead to a reward (usually long-term goal, similar to game playing).
- The ultimate consequence of an action may not be apparent until the end of a sequence of actions (no immediate teaching signal).
- When a reward is achieved, it may not be due to the last action performed, but due to one earlier in the sequence of actions.
- In contrast to classification or clustering, there is no pre-defined set of training samples. The experiences that form the basis of reinforcement learning are derived through the *exploration in a learning environment*, instead of training data. So, *reinforcement learning is not supervised learning or unsupervised learning*.

# Markov Decision Process as Problem Representation

- An agent (e.g., a robot or an AI game player) is operating in a domain that can be represented as a set of distinct states, represented as a set  $\mathbf{S}$ .
- The agent has a set of actions that it can perform, represented as a set  $\mathbf{A}$ .
- Time advances are in discrete steps, with a fixed time step.
- At time  $t$  the agent knows the current state  $s_t \in \mathbf{S}$  and must select an action  $a_t \in \mathbf{A}$  to perform.
- When the action  $a_t$  is carried out, the agent will receive a reward  $r_t$  (it could be 0 if there is no reward) and the agent enters a new state  $s_{t+1}$ .
- The reward, which may be positive, negative or zero, depends on both the state and the action chosen, so  $r_t \equiv r(s_t, a_t)$ , where  $r$  is the **reward function**.
- The new state also depends on both the state and the action chosen, so  $s_{t+1} = T(s_t, a_t)$ , where  $T$  is the **transition function**.

In a **Markov decision process**, the functions  $r$  and  $T$  depend only on the **current** action and state. (no memory of the past)

# Control Policy

## – what action to take at a given state?

The agent in reinforcement learning must learn how to choose the **best action at each state**. This is the key issue in reinforcement learning.

Therefore, it must acquire a ***control policy***  $\pi$ , which is a mapping from  $S \rightarrow A$ .

That is,  $a_t = \pi(s_t)$ , for any state at time t.

So what do we mean by “best action”?

- This is related to the reward of an action or values of states as the consequence of a sequence of actions.

**Learning to evaluate actions and states is a main task of reinforcement learning.**

(It could help you understand better if you associate reinforcement learning with game playing or robot navigation.)

# What is the “best action”?

If we define the best action as the one leading to the greatest **immediate reward**, this would produce a good short-term payoff, but might not be optimal in the long run. Also, immediate reward may not be clear or available.

It usually makes more sense to maximise the **total payoff** over time.

We could define the payoff of a sequence of actions starting from the current state  $s_0$  to goal state  $s_N$  as the sum of all the rewards corresponding to these states:

$$V \equiv \sum_{i=0}^N r_i = r_0 + r_1 + \dots + r_N$$

where the sum is taken over all the states involved in the sequence of actions.

However, this makes a reward in the very distant future just as valuable as one received immediately, which is often unrealistic.

# Discounted Cumulative Reward

So, we need some way of weighting the rewards so that the more distant they are the less they are worth.

Quite naturally, we can introduce a constant  $\gamma$ , as a discount factor, to indicate the relative values of immediate and delayed rewards:

$$V \equiv \sum_{i=0}^N \gamma^i r_i = \gamma^0 r_0 + \gamma^1 r_1 + \gamma^2 r_2 + \dots + \gamma^N r_N$$

where  $0 < \gamma \leq 1$ ,  $r_i$  is the reward at state  $s_i$ ,  $i=0$  represents the current state or time (time as discrete steps). The total payoff calculate in this way is called the ***discounted cumulative reward***.

*There may be more than one sequence of actions starting from a state to the goal state, corresponding to different routes. The V value should be the one calculated using the optimal route, i.e., the maximum value.  
(see example later)*

# Optimal Control Policy

The optimal control policy, represented as  $\pi^*$ , is clearly the one that **maximises the discounted cumulative reward for each state**, thus

$$\pi^* \equiv \text{argmax}_{\pi} V^{\pi}(s) \text{ for all } s$$

where  $V^{\pi}$  denotes the discounted cumulative reward function when the actions in the sequence are chosen using control policy  $\pi$ .  
Different control policies lead to different sequences of actions.

The discounted cumulative reward given by the optimal control policy  $\pi^*$  is denoted as  $V^*(s)$ .

# The Learning Task

We can now define what we want to learn in reinforcement learning:

**The learning task is to discover the optimal control policy  $\pi^*$ ,  
i.e., the best action at each state.**

This is similar to find the optimal route or optimal sequence of operations /moves in state space search. However, the focus in reinforcement learning is evaluation of states and actions, rather than search strategies.

Indirectly, the learning task is **to find out the maximum discounted cumulative reward values of all states**. As a matter of fact, this is the focus of reinforcement learning (most of the remaining lecture on reinforcement learning is about how to calculate or estimate the discounted cumulative reward values).

# An Example Domain/Scenario

Suppose we want to write a program to learn to play an extremely simple “adventure” game (Similarly, we can handle a complex one for practical applications if we have time).

The domain may be represented as follows:

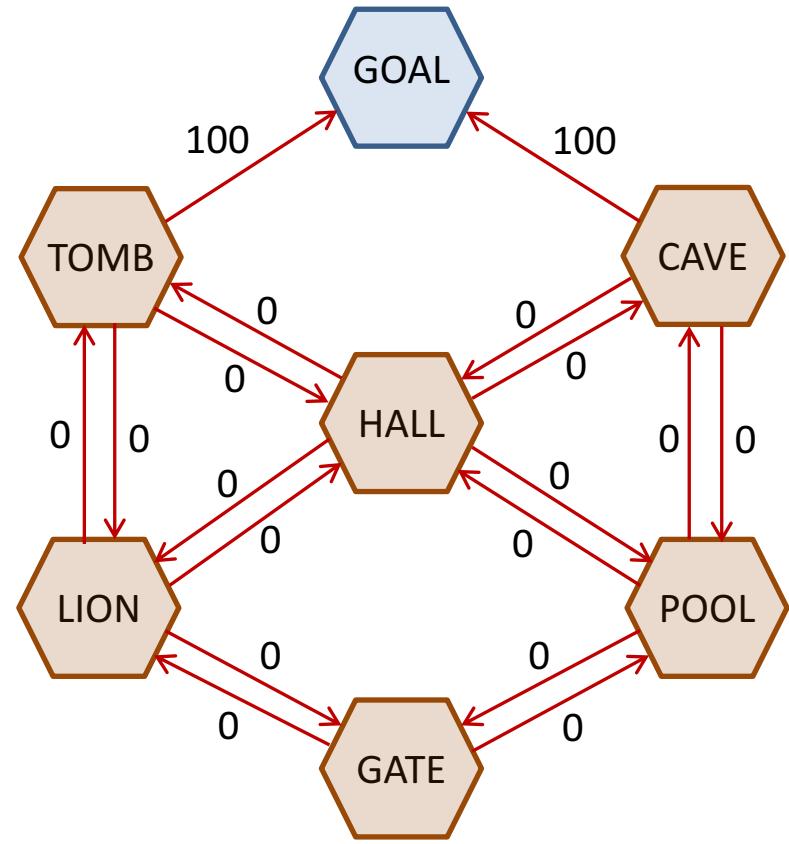
Hexagons denote **states**.

Arrows indicate the possible **actions** for each state.

The game finishes when the player reaches the goal state and receives a reward of £100.

Numbers adjacent to arrows indicate values of the reward function  $r$ , i.e., the **immediate reward** associated with the state transition.

Only two non-zero reward values are shown in the figure. The other reward values are not known, and we assume they are all zero.



# Discounted Cumulative Rewards

For this simple game it is easy to determine the optimal control policy  $\pi^*$ , or the best actions at each state.

Suppose that the discount factor  $\gamma=0.8$ , then we can easily calculate the **discounted cumulative reward**  $V^*(s)$  for every state using

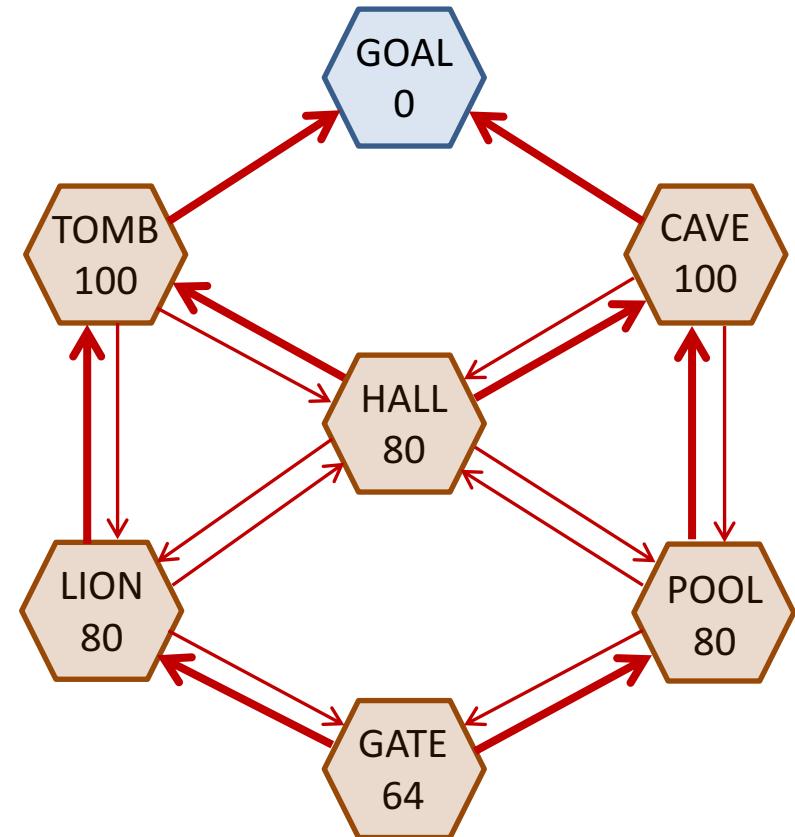
$$V \equiv \sum_{i=0}^N \gamma^i r_i$$

For example:

$\xrightarrow{\text{GATE to POOL}}$   $\xrightarrow{\text{POOL to CAVE}}$   $\xrightarrow{\text{CAVE to GOAL}}$

$$\begin{aligned} V^*(\text{GATE}) &= \gamma^0 r_0 + \gamma^1 r_1 + \gamma^2 r_2 \\ &= 0.8^0 \times 0 + 0.8^1 \times 0 + 0.8^2 \times 100 \\ &= 64 \text{ (optimal)} \end{aligned}$$

(there are other routes with smaller values)



The best actions at each state are indicated by thick arrows.

( Where to go at POOL? )

# Optimal Control Policy and Q Function

Suppose the agent in reinforcement learning knows:

- the transition function  $T(s, a)$
- the reward function  $r(s, a)$
- the discounted cumulative reward of each state  $V^*(s)$

Then the optimal control policy for state  $s$  would be

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

Therefore,  $V^*$  can be used as an evaluation function for actions or states.

However, for many applications we do not know the immediate reward values  $r_0, r_1, r_2, \dots, r_N$  for calculating discounted cumulative reward. Even for the simple game just considered earlier, we assume most of the  $r$  values are zero. From the point of view of machine learning, it could be a good idea for the agent to try to learn  $V^*$  if there is insufficient knowledge about it.

# Optimal Control Policy and Q Function (2)

Suppose an agent is in state  $s$  and is trying to select its next action.

If the agent does not know the transition function  $T$ , then no form of **action evaluation** that requires looking ahead is possible.

What information does such an agent need to know?

It is the total payoff that can be expected for each possible choice of action  $a$  in the current state  $s$ , i.e.,  $V^*(s)$ , but this may not be available.

So, let's *define* such an **evaluation function** as follows, which is called Q function and related to  $V^*$ :

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

where  $s'$  is the state resulted from action  $a$  in the current state  $s$ .

Thus, in place of  $V^*$ , which is a **function of state only and may not be available**, we now have  $Q$ , which is a **function of both state and action**.

# Why is Q function useful?

If we know  $Q$ , we no longer need to know transition function  $T$  and reward function  $r$  to determine the best actions because

$$\pi^*(s) = \text{argmax}_a Q(s, a)$$

The information, which could be obtained by looking ahead using  $T$  and  $r$ , has been absorbed into the Q function.

However, in many applications we do not have  $V^*(s)$  and do not have an analytical Q function either.

Is it possible to learn  $Q$ ? Yes, that's why the Q function is useful!

- Chris Watkins (<http://www.cs.rhul.ac.uk/~chrisw/>) proposed a Q learning algorithm in 1989 when he was a PhD student at Cambridge University. He proposed a formula to express the Q function and an iterative learning algorithm to estimate the Q values through iterative updating.

# Learning Q

## The Problem

The agent requires a procedure that will be able to form a good estimate of  $Q$  as a result of its experiences obtained from exploration in the domain.

Such experiences only provide direct information about **immediate rewards**.

But the true value of  $Q$  depends on a **sequence of immediate rewards**.

## The Solution

We can exploit the fact that the value of  $Q$  for a state  $s$  and an action  $a$  depends on the  $Q$  values of the neighbours of state  $s$ , because

$$Q(s, a) \equiv r(s, a) + \gamma V^*(s') := r(s, a) + \gamma \max_{a'} Q(s', a')$$

where  $\equiv$  means “is equivalent to ...” and  $:=$  means “is defined by ...”

The trick: Making  $Q$  iteratively updatable,  
so that it is possible to estimate  $Q$  through machine learning.

# The Q Learning Algorithm (pseudo code)

Suppose there are:

No need of functions  $r$  and  $T$

$m$  states  $s_1 \dots s_m$  and  $n$  actions  $a_1 \dots a_n$  //the domain

Create an  $m \times n$  array  $QE$  to hold estimates of  $Q(s,a)$  and an  $m \times n$  array  $rE$  to hold estimates of  $r(s,a)$ .

Initialise all entries in  $QE$  to zero, i.e.,  $QE(s,a)=0$ .

Initialise all entries in  $rE$  with given initial values or zero if not given.

Select an initial state  $s_i$  //could be randomly chosen from  $s_1 \dots s_m$

$s := s_i$

REPEAT

Select and execute an action  $a$  to reach a new state  $s'$  //could be randomly

$QE(s,a) := rE(s,a) + \gamma \times \max_{a'} QE(s',a')$  //estimate Q function value

$rE(s,a) := QE(s,a)$  //update immediate reward

If  $a'$  is not empty (action available at  $s'$ ), then  $s := s'$ .

Otherwise, set  $s$  to another initial state //could be randomly

UNTIL Termination Condition Satisfied

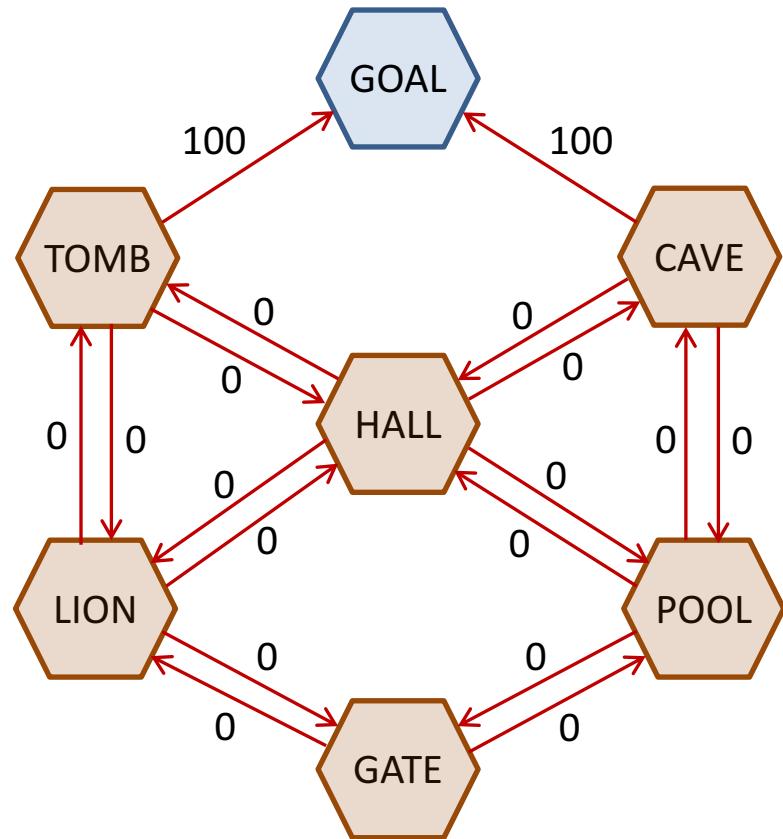
//pre-set number of iterations reached or  $QE$  and  $rE$  are stable.

# Q Learning in Action

Consider learning the Q values of states and actions in the same ‘adventure’ game domain as shown in the diagram.

Numbers adjacent to arrows indicate initial values of the estimates of immediate rewards (i.e.,  $r_E$ ).

Initially all the estimates of Q (i.e.,  $Q_E$ ) will be zero.



# Q Learning in Action (2)

To start iterative updating of Q values, **suppose** the agent begins at state HALL and selects the action To-CAVE (**kind of random**). Therefore, we have  $s=HALL$ ,  $a=To-CAVE$ ,  $s'=CAVE$ .

Applying Q-learning update procedure ...

Before updating:

$$rE(HALL, \text{To-CAVE}) = 0$$

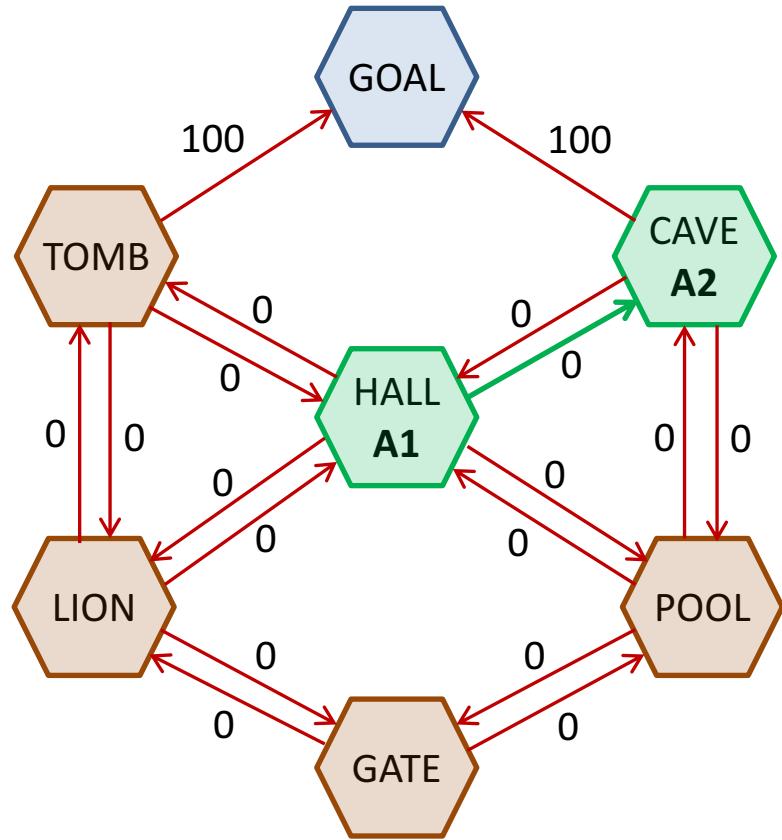
$$QE(CAVE, a') = 0 \text{ for all actions } a'$$

After updating:

$$QE(HALL, \text{To-CAVE}) = 0 + \gamma \times 0 = 0$$

$$rE(HALL, \text{To-CAVE}) = QE(HALL, \text{To-CAVE}) = 0$$

Nothing changed by this updating.



$$QE(s, a) := rE(s, a) + \gamma \times \max_{a'} QE(s', a')$$
$$rE(s, a) := QE(s, a)$$

# Q Learning in Action (3)

Now the agent is in state CAVE, suppose it selects the action To-GOAL. Therefore we have  
 $s=CAVE$ ,  $a=To\text{-GOAL}$ ,  $s'=GOAL$ .

Applying the update procedure again ...

Before updating:

$$rE(CAVE, \text{To-GOAL}) = 100$$

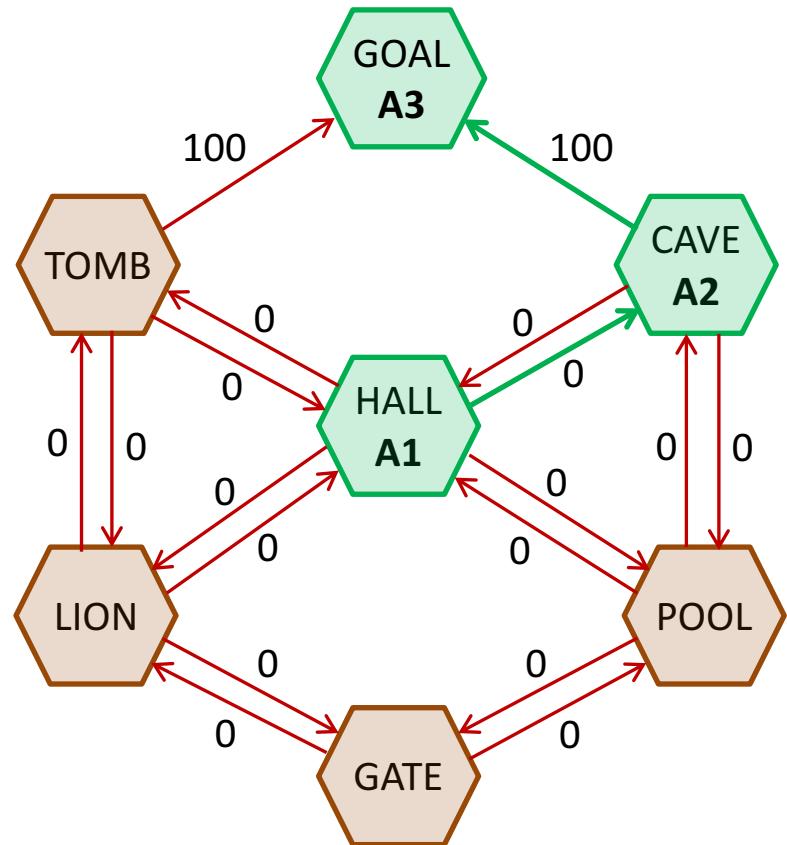
$$QE(GOAL, a') = 0$$

After updating:

$$QE(CAVE, \text{To-GOAL}) = 100 + \gamma \times 0 = 100$$

$$\begin{aligned} rE(CAVE, \text{To-GOAL}) &= QE(CAVE, \text{To-GOAL}) \\ &= 100 \end{aligned}$$

The estimated  $Q$  value for the CAVE-GOAL transition has been improved.



$$\begin{aligned} QE(s, a) &:= rE(s, a) + \gamma \times \max_{a'} QE(s', a') \\ rE(s, a) &:= QE(s, a) \end{aligned}$$

# Q Learning in Action (4)

Since GOAL is a sink state in this domain, the agent must start again with a new initial state . **Suppose** it begins once more at HALL, and once again selects the action To-CAVE. Therefore we have  $s=HALL$ ,  $a=To\text{-}CAVE$ ,  $s'=CAVE$ .

Before updating:

$$rE(HALL, \text{To-CAVE}) = 0$$

$$QE(CAVE, \text{To-GOAL}) = 100$$

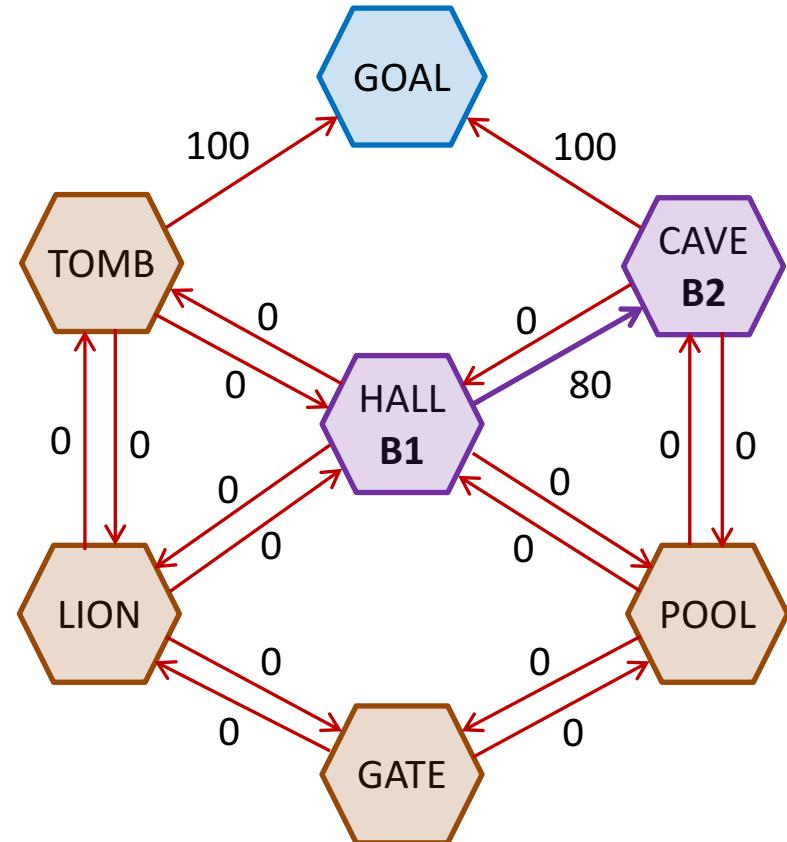
$$QE(CAVE, a') = 0 \text{ for all other actions}$$

$$\text{Max}_{a'}QE(CAVE, a') = 100$$

After updating ( $\gamma = 0.8$ ):

$$QE(HALL, \text{To-CAVE}) = 0 + \gamma * 100 = 80$$

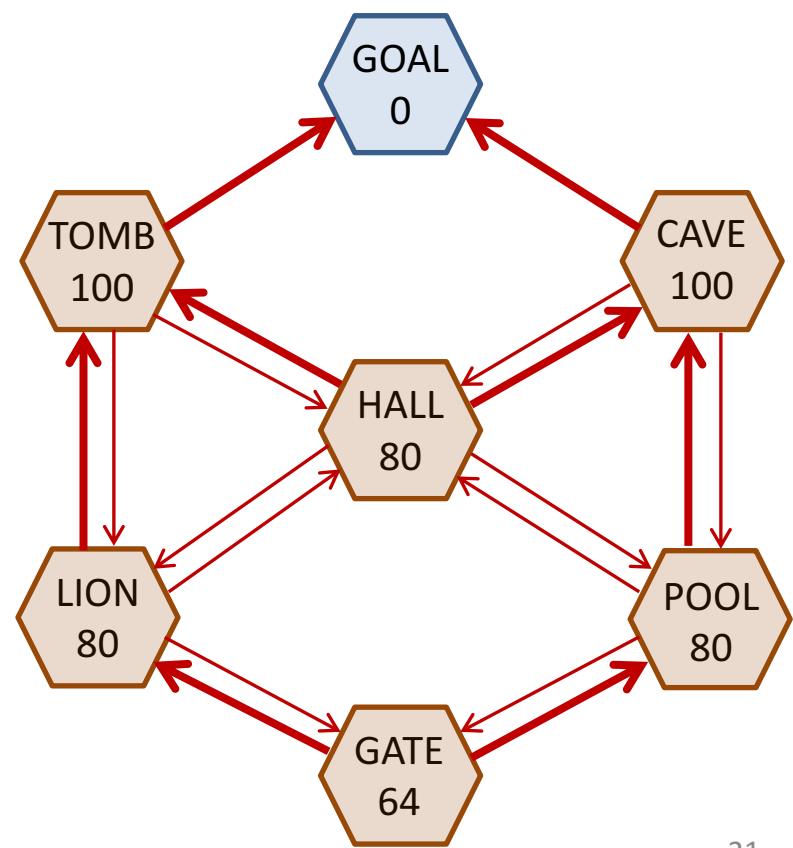
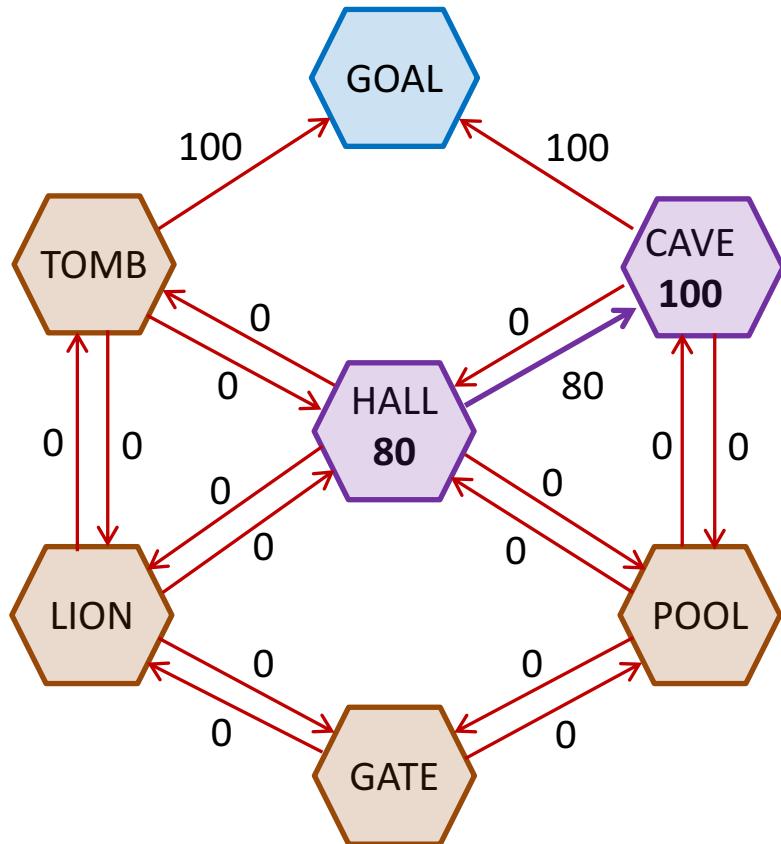
$$\begin{aligned} rE(HALL, \text{To-CAVE}) &= QE(HALL, \text{To-CAVE}) \\ &= 80 \end{aligned}$$



$$\begin{aligned} QE(s,a) &:= rE(s,a) + \gamma \times \max_{a'} QE(s',a') \\ rE(s,a) &:= QE(s,a) \end{aligned}$$

# Q Learning in Action (5)

After a few iterations, Q-learning has obtained estimated Q values for CAVE and HALL, which are the same as the previously calculated discounted cumulative reward values. Choosing different initial states and repeating Q-learning update procedure we can obtain estimated Q values for other states.



# The Q Learning Algorithm (pseudo code)

Suppose there are:

No need of functions  $r$  and  $T$

$m$  states  $s_1 \dots s_m$  and  $n$  actions  $a_1 \dots a_n$  //the domain

Create an  $m \times n$  array  $QE$  to hold estimates of  $Q(s,a)$  and an  $m \times n$  array  $rE$  to hold estimates of  $r(s,a)$ .

Initialise all entries in  $QE$  to zero, i.e.,  $QE(s,a)=0$ .

Initialise all entries in  $rE$  with given initial values or zero if not given.

Select an initial state  $s_i$  //could be randomly

$s := s_i$

REPEAT

Select and execute an action  $a$  to reach a new state  $s'$  //could be randomly

$QE(s,a) := rE(s,a) + \gamma \times \max_{a'} QE(s',a')$  //estimate Q function value

$rE(s,a) := QE(s,a)$  //update immediate reward

If  $a'$  is not empty (action available at  $s'$ ), then  $s := s'$ .

Otherwise, set  $s$  to another initial state //could be randomly

UNTIL Termination Condition Satisfied

//pre-set number of iterations reached or  $QE$  and  $rE$  are stable.

# Some Issues in Q Learning

As indicated in the Q learning algorithm (pseudo code), the steps for updating  $QE(s, a)$  and  $rE(s, a)$  can be done iteratively, with different initial states and different choices of actions at each state involved in the agent's learning trajectory. **How to choose initial states and actions at a specific state? Any better ways than random choice?**

In general, each state should be visited many times during the Q learning process. This means the number of iterations could be very large, especially when there are many states and actions. Therefore, **Q learning could be very slow. How to improve Q learning?**

**When should the Q learning stop?** - There could be different stop criteria. For example, when the pre-set maximum number of iterations have been conducted or when there has been no improvement/change of QE values by the Q learning iterations.

# Strategies for Selecting Initial States and Actions

The choice of initial states and actions at a given state determines the agent's learning trajectory through state space and hence its learning experience and outcome.

How should the agent choose initial states and actions during Q learning?

- Uniform random selection

Advantage: Will explore the whole state space with equal opportunity.

Disadvantage: May spend a great deal of time learning the values of transitions that are not on any optimal path.

- Select actions with highest expected cumulative reward (exploitation)

Advantage: Concentrates resources on apparently useful transitions.

Disadvantage: May ignore even better pathways whose values have not been explored.

How about trade-off between exploitation and exploration, as what is done for node selection in Monte-Carlo tree search?

# Learning Q Values by Neural Networks (if training data is available, e.g. from previous Q learning)

In many applications (e.g., game Go), the total number of state-action pairs can be very large. **Both time complexity and space complexity of Q learning could be too high.**

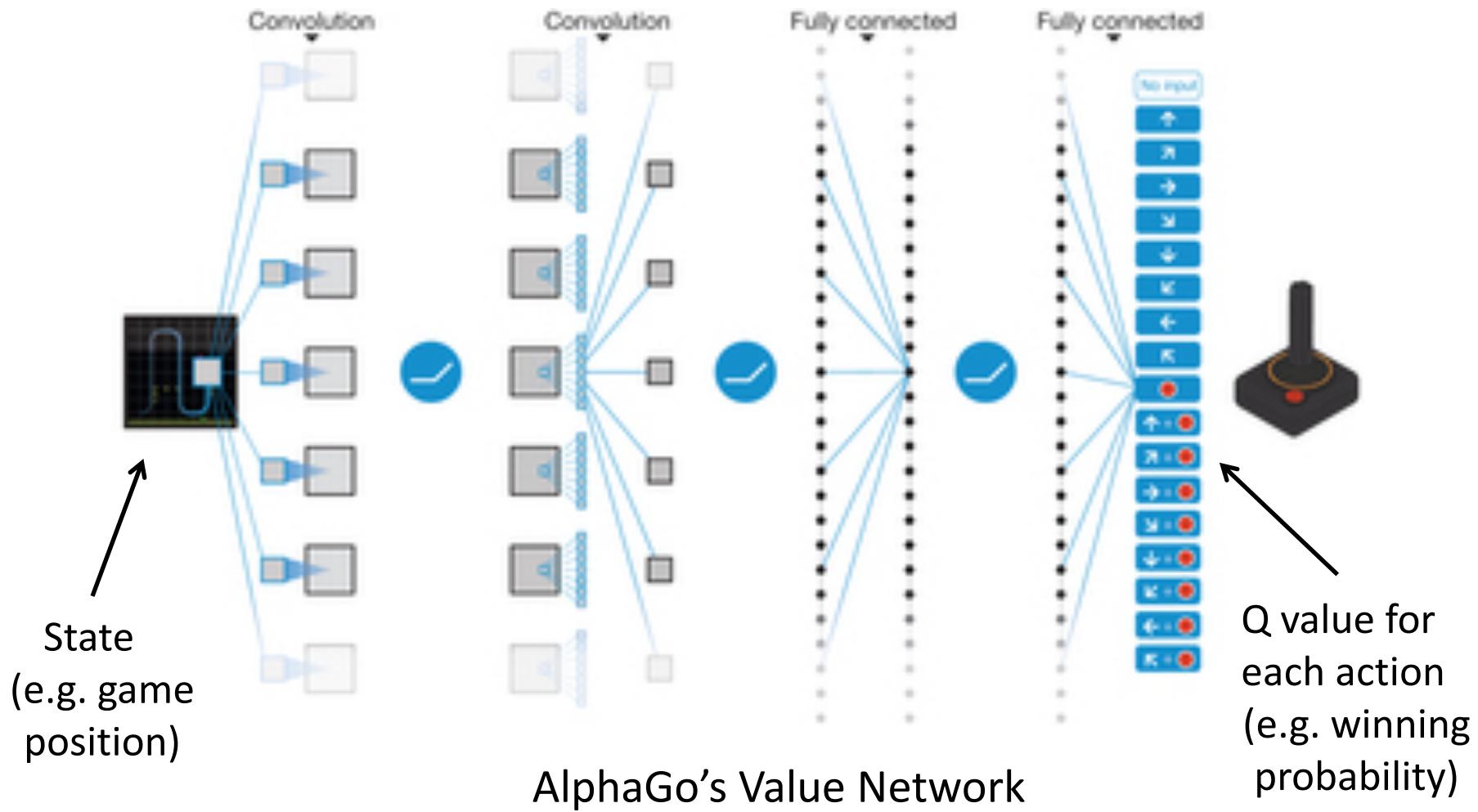
However, in many domains, **similar actions in similar states usually have similar consequences**. This fact can be exploited to generalize the results of Q learning by using multilayer neural networks to carry out function approximation.

Two approaches:

- Using one neural network: input is  $(s, a)$ , output is  $Q$ .
- Using  $n$  neural networks, where  $n$  is the number of actions: input is  $s$  for all the neural networks, output of each neural network is  $Q$  corresponding to a specific action. This approach is better in general.

**Problem: It requires labelled training data, which could be from the interim results of Q learning, leading to the combination of Q learning and learning in neural networks.**

# Learning Q Values by a Deep Neural Network



[This is equivalent to  $n$  neural networks, each with one output unit.]

# Summary

## Reinforcement Learning:

Tasks of reinforcement learning

## Markov Decision Process as Problem Representation:

It is similar to state space representation  
with extra reward function

Control policy

*Discounted cumulative reward*

## Q Learning:

The Q function ( replacing  $V^*(s)$  )

*The Q learning algorithm (pseudo code)*

Q learning in action – an example

Strategies for selecting initial states and actions

Learning Q values by neural networks

# CE213 Artificial Intelligence – Lecture 1

## Module supervisor:

Prof. John Gan, Email: jqgan, Office: 4B.524

Homepage: <https://www.essex.ac.uk/people/GANJO00207/john-qiang-gan>

## CE213 on Moodle: <https://moodle.essex.ac.uk/course/view.php?id=3651&section=0>

Module information, Assessment information,  
Teaching/reading materials, Other resources.

## Acknowledgement:

Some of the lecture notes for this module were originally prepared by Dr. Paul Scott and updated by Prof. John Gan.

# Information about this module

Lectures (Zoom webinars): Tuesdays, 2-4pm

Classes (Zoom meetings): Thursdays, 9-10am for CLAa01  
3-4pm for CLAa02

Lab Exercises: at your own time, with detailed instructions and sample code provided on CE213 Moodle. Each lab exercise will be explained briefly in an appropriate class. *These are optional, but could be very useful.*

Assignment: one programming assignment with deadline in week 8

Progress Tests: through Moodle, one in week 6, another in week 11

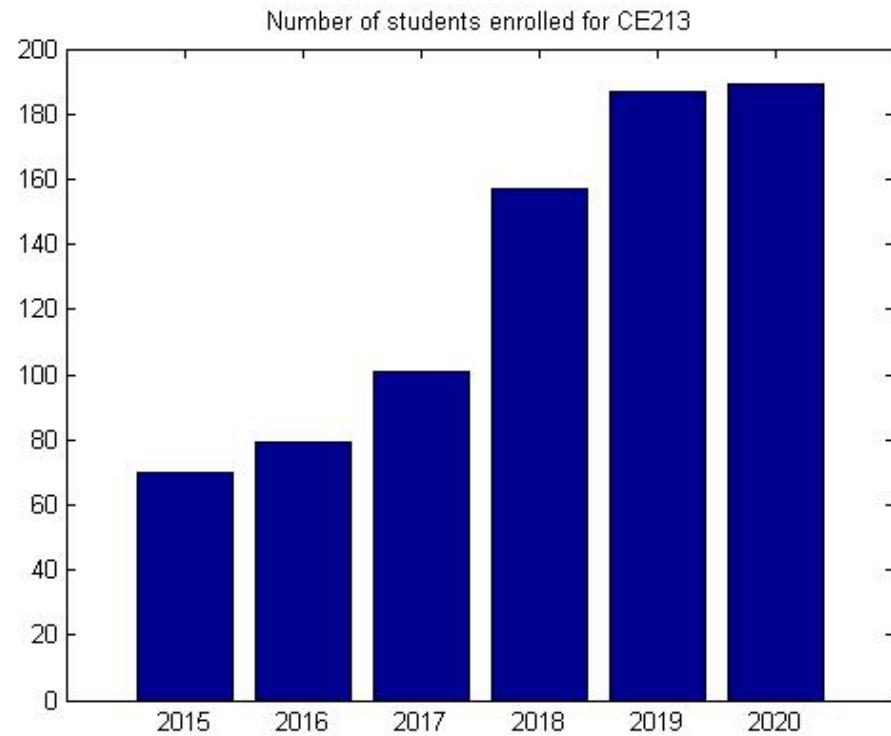
Let's go to CE213 Moodle website and highlight some information provided there: <https://moodle.essex.ac.uk/course/view.php?id=3651&section=0>

# Information about this module (2)

- CE213 is a challenging module, but we can work together to make it a success. I hope that you can do the following:

- 1) **Attend lectures and classes on time. Feel free to ask questions and take part in discussions.**
- 2) **Skim relevant lecture notes and class problem sheet before a lecture or class.**
- 3) **Have pen and paper at hand for problem solving exercises during classes.**
- 4) **For lab exercises at your own time, at least try to run and understand the provided solutions.**

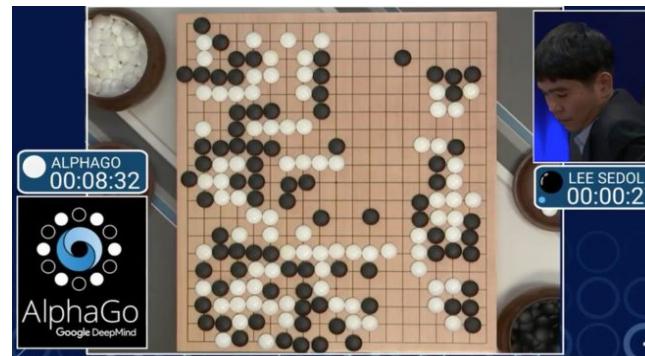
- Number of CE213 students in the past years:



- In the recent years, average mark of this module is over 60%, and average SAMT score is over 4 out of 5.

# Questions to be answered in this first lecture

1. What is AI?
2. Is AI possible?
3. How is AI possible?
4. What are the applications of AI?

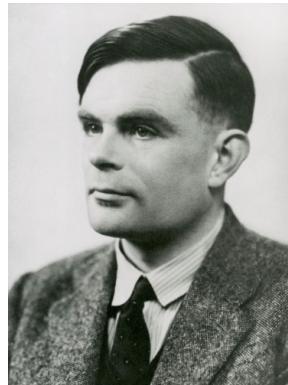


Do these pictures give you some ideas for answering the above questions?

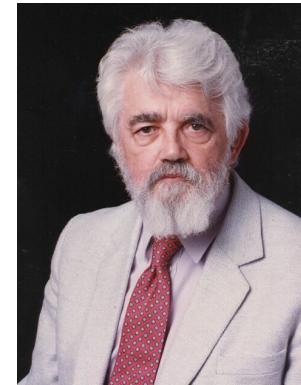
# What Is AI? – A little bit history first

- 4<sup>th</sup> Century BC: Aristotle invented syllogistic logic, the 1<sup>st</sup> formal deductive reasoning system.
- 19<sup>th</sup> Century: Babbage's analytical Engine (1<sup>st</sup> general-purpose computer) and Lady Lovelace's first computer program (In her opinion, computers cannot really think).
- 20<sup>th</sup> Century: Alan Turing argued that machine intelligence is possible and proposed 'Turing Test' in 1950. John McCarthy coined the term 'Artificial intelligence' in 1955 to publicise a summer workshop.

Alan Turing



John McCarthy



# What Is AI? – Some definitions

- There is still **no consensus** on the definition of AI. Here are two simple and famous definitions:
  - (The science and engineering of making) intelligent machines.  
(McCarthy's definition)
  - Machines that think. (Turing's definition)
- Strong AI: Machines that think.
- Weak AI: Machines that act as if they think.
- Specialised/Narrow AI: AI for one specific task.
- General AI or Artificial General Intelligence:  
Machines that can think, learn, act and feel like humans and are for multiple tasks. [https://www.youtube.com/watch?v=x-QfL\\_BmZVE](https://www.youtube.com/watch?v=x-QfL_BmZVE)

# **Is AI Possible?**

## **AI and The Philosophy of Mind**

It seems that a ‘yes’ answer to this question is obvious now. However, is there a machine that is genuinely capable of thinking now?

For several hundred years people have been arguing about whether it was possible, in principle, to make a machine that can think. (Mainly argued by philosophers at the time when no computer was available)

The arguments took on a less abstract form about 180 years ago due to the design of the first general-purpose computer.

# Babbage's Analytical Engine

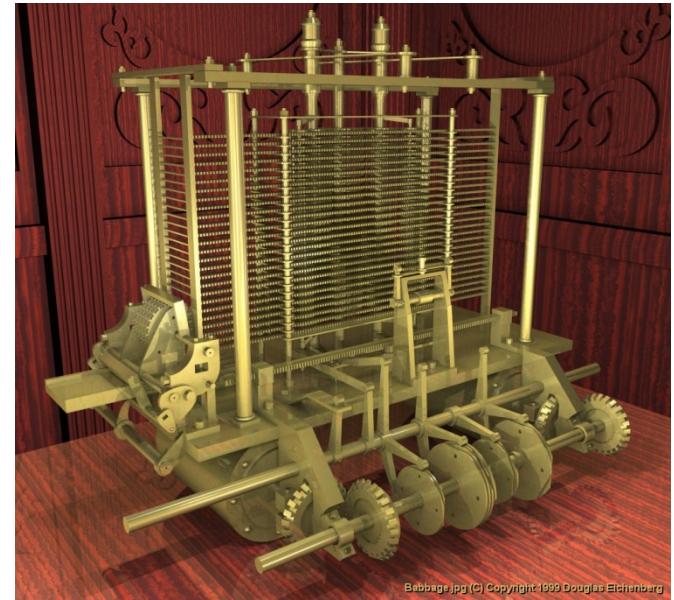
The first general-purpose computer (it is mechanical!), which made the AI argument less abstract.



Charles Babbage



Lady Lovelace



Part of Babbage's Analytical Engine

# The World's First Programmer

Lady Lovelace and Babbage collaborated on developing the Analytical Engine.

He designed the hardware.

She wrote the programs, was the world's first programmer.

Unfortunately the machine was never finished.

They ran out of money.

However Lady Lovelace did develop several key ideas that have been used by programmers ever since.

e.g., Subroutines

## The Lady Lovelace's Objection

In her paper describing the Analytical Engine she said the following about the possibility that it could “think”:

“The Analytical Engine has no pretensions whatever to *originate* anything. It can do whatever we know how to order it to perform. It can *follow* analysis, but it has no power of *anticipating* any analytical relations or truths.”

In other words:

Computers cannot really think because they can only do what their programmers tell them to do.

# Rebutting the Objection

In 1950, Alan Turing published [a paper](#) “Computing Machine and Intelligence”, addressing a range of objections to the idea that computers could think. (<http://www.csee.umbc.edu/courses/471/papers/turing.pdf>)

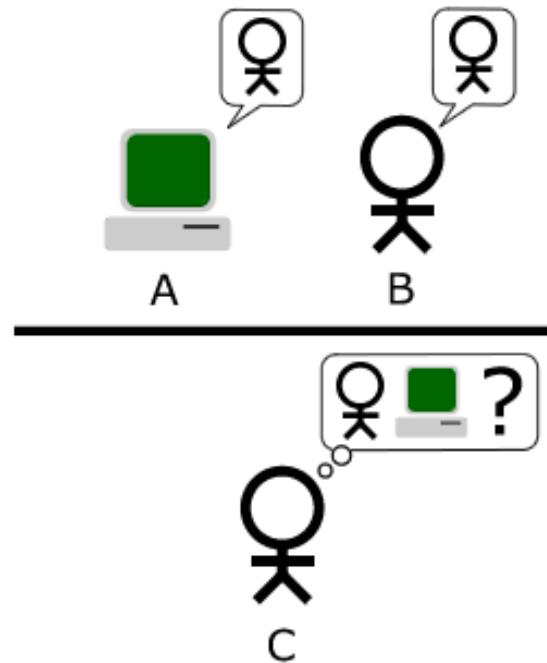
Including rebutting the Lady Lovelace’s Objection.

Turing had two counterarguments:

- In practice, the results of programming often surprise us.  
(Lady Lovelace never got to run any of her programs!)
- The possibility that machines could be programmed to learn.

# The Turing Test

In the scenario shown in the figure, A is a computer, B is a human, and C is an interrogator. C asks A and B questions, and determines on the basis of their replies which one is the computer. Communication is only by typewritten text.



Turing argues that if C cannot distinguish between A and B then we would have to concede that the computer is as capable of thought as a human being. This is essentially a **behavioural** argument - If an entity behaves as if it thinks then it thinks.

*Should more interrogators be involved in Turing Test to make it more reliable? Even if so, is Turing Test valid for assessing AI? Is Turing Test itself a refutation of Lady Lovelace's Objection?*

## Searle's Chinese Room

([https://en.wikipedia.org/wiki/John\\_Searle](https://en.wikipedia.org/wiki/John_Searle))

The Turing Test is very famous, but also controversial. One of the criticism was given by John Searle. He proposed the so-called Searle's Chinese Room.

Consider the following two definitions of artificial intelligence:

*A machine that **thinks** like a person. (**strong AI**)*

*A machine that **acts as if it thinks** like a person. (**weak AI**)*

Searle has no quarrel with weak AI, but he profoundly disagrees with the view that Weak AI and Strong AI are equivalent, like what Turing argued for the Turing Test.

Furthermore, he does not believe Strong AI is possible.

## Searle's Chinese Room (2)

### The Chinese Room – A thought experiment

In order to demonstrate the difference between Weak AI and Strong AI, Searle developed an analogy known as the “Chinese Room”.

*Imagine an English speaking person locked in a room, who has no knowledge of Chinese or Russian.*

*Also in the room is a (large) set of rules, written in English, for manipulating the symbols of Chinese text and producing Russian translations.*

*Periodically this person is given texts written in Chinese.*

*By mechanically following these rules, the person produces Russian translations of the Chinese text.*

## Searle's Chinese Room (3)

### Searle argues

The room+person system behaves as if it understands Chinese. However, it is obvious that no understanding is involved. Hence, behaviour does not imply the existence of a mind that understands.

In other words, weak AI does *not* imply strong AI.

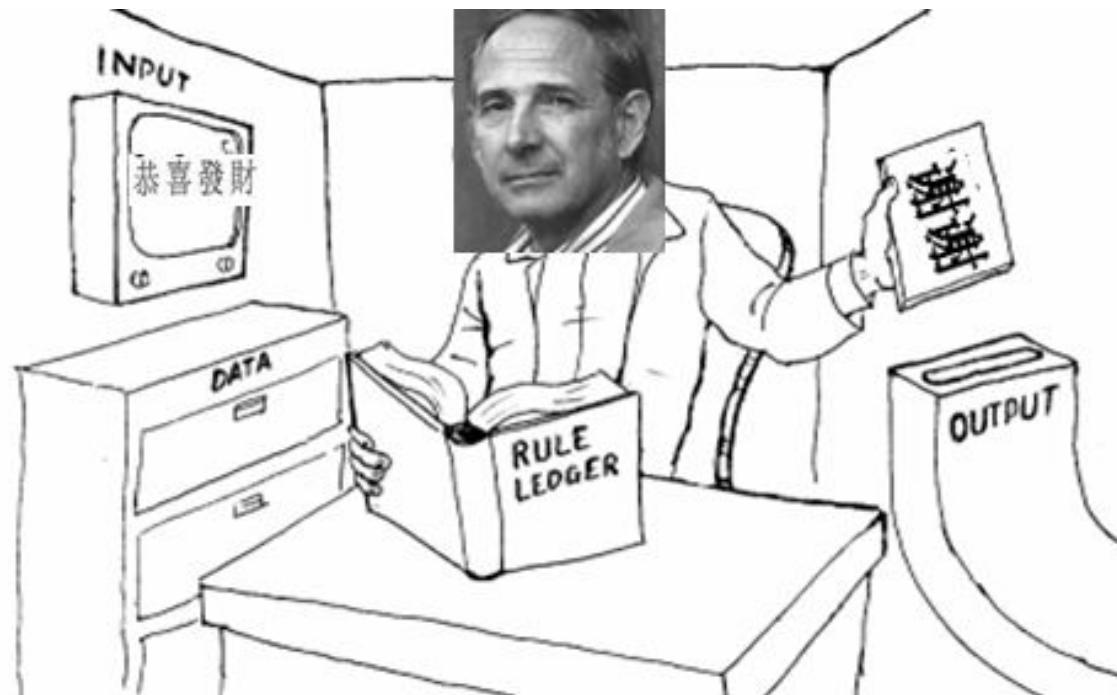
Thus, if we accept Searle's argument, then Turing test is not a valid criterion/method for assessing whether a machine can think.

***What would be a better way to test whether (strong) AI is possible?***

(Something like the experiment described in Ex Machina?)

(Robot college student test?)

## Searle's Chinese Room (4)



Another version of the Searle's Chinese Room: Someone or a computer can communicate in Chinese without understanding Chinese in the 'Chinese Room', who could pass the Turing Test without understanding/thinking involved.

# So, Is AI possible?

- There has been evidence of weak AI or specialised AI, e.g., AlphaGo, self-driving cars, Alexa, personalised search engine. We may say that weak AI is possible.
- However, no computer program has really passed the Turing Test ([https://en.wikipedia.org/wiki/Loebner\\_Prize](https://en.wikipedia.org/wiki/Loebner_Prize)).
- And there are no well-recognised/accepted criteria for assessing strong AI or general AI.
- Anyway, if AI is defined as intelligent machines then AI is possible, but how is AI possible and what can AI do?



# How Is AI Possible?

## ➤ Three fundamental approaches:

Search as general problem solver – Unit 1

The ‘generate and evaluate’ approach:

state space representation of problems, search strategies  
(e.g., A\*, minimax, MCTS), evaluation criteria

Expert systems – Unit 2

knowledge representation, rule interpretation, reasoning  
with uncertainty

Machine learning – Unit 3

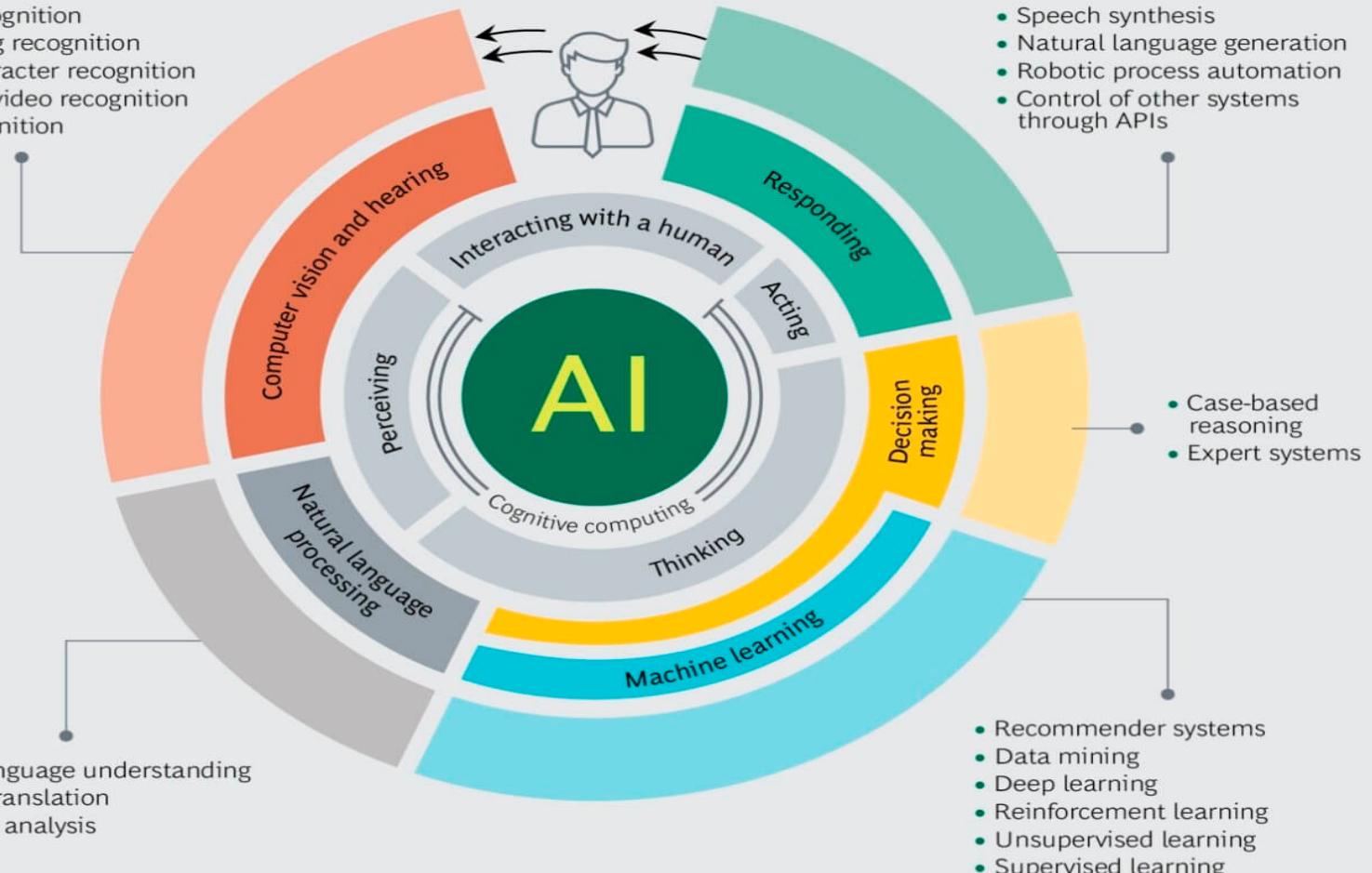
learning environment, models and algorithms

*About 90% of an AI course would try to answer this question!*

# AI Applications

## EXHIBIT 1 | AI and Robotics Technologies Come in Many Forms, Giving Rise to a Broad Variety of Applications

- Speech recognition
- Handwriting recognition
- Optical character recognition
- Image and video recognition
- Facial recognition



Source: BCG analysis.

Note: APIs = application programming interfaces.

# Summary

## What is AI?

Intelligent machines.

Machines that **think and learn** like people.

Machines that **act** rationally/intelligently.

## Is AI possible?

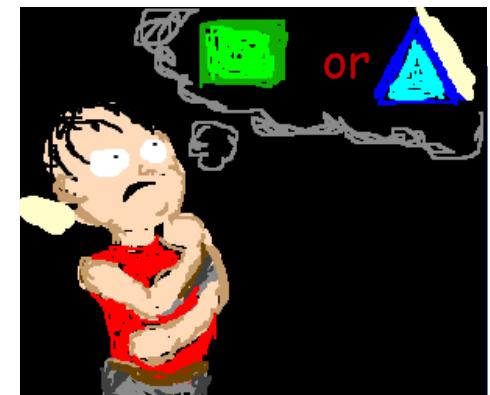
Lady Lovelace's Objection

The Turing Test

Searle's Chinese Room

## How is AI possible?

## What are the applications of AI?



*Have you got the answers to these questions now?*

*What are you going to learn next?*

# CE213 Artificial Intelligence – Lecture 2

## Problem Solving by State Space Search

Basic idea for general problem solver:  
“generate-and-evaluate”

How do we get a computer to solve problems?  
*How to represent the solution to a problem?*  
*How to generate potential solutions?*  
*How to evaluate them?*

# The Corn, Goose and Fox Problem

*A farmer is taking a sack of corn, a goose, and a fox to market. He has to cross a river under the following constraints:*

*There is a rowing boat but it is only large enough to carry the farmer (rower) and any one of the three things he is taking.*

*Unfortunately there are two problems:*

*The goose will eat the corn if the farmer is not present.*

*The fox will eat the goose if the farmer is not present.*

*(The goose or fox will not run away even when the farmer is not with them ☺)*

**How does the farmer get everything across the river?**

# Solving the Corn, Goose and Fox Problem

We begin by abstracting the **essentials** of the problem:

Clearly, any solution consists of a series of **moves / actions**, in which the farmer ferries either the corn, the goose, the fox, or just himself across the river.

So at any stage there are, at most, four things (moves) the farmer can do:

Take the corn across:  $c$

Take the goose across:  $g$

Take the fox across:  $f$

Go across alone:  $n$

We also need to describe the **situations** resulted from these moves:

- We need to know which animals are on which bank.

In this problem we can **ignore** what happens while the boat is crossing, and just consider the situations before and after each move.

- We need to keep track of the positions of *four* entities: the corn, the goose, the fox, and the farmer.

The farmer and the boat have to stay together.

Each must be on the left bank (*L*) or the right bank (*R*).

So we can represent any situation by a 4-tuple (a state representation):

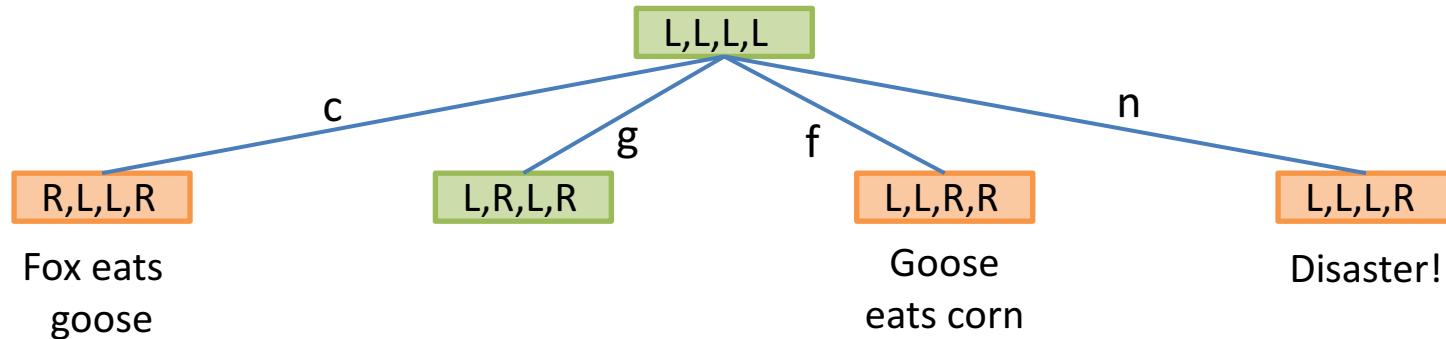
**[Corn place, Goose place, Fox place, Farmer place]**

e.g.,  $[L,L,L,R]$  indicates that the corn, the goose and the fox are all on the left bank, but the farmer is on the right bank.

# Searching for a solution

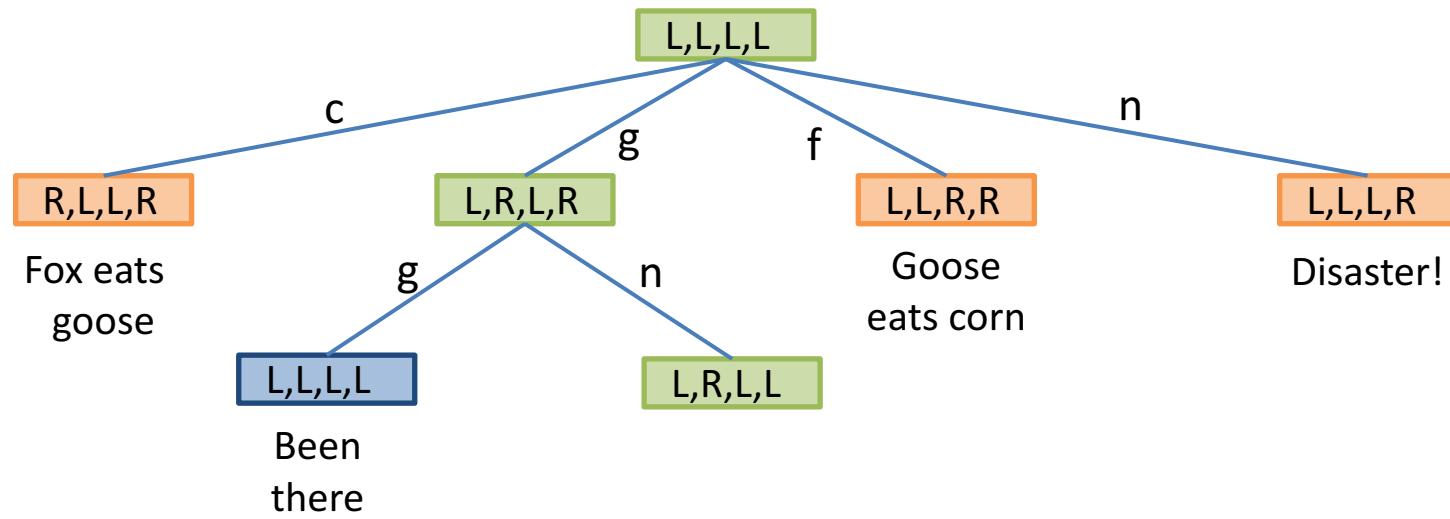
Initially, at  $[L,L,L,L]$  there are four possible moves:

Clearly, only one is a viable choice for the first move, that is  $g$ .



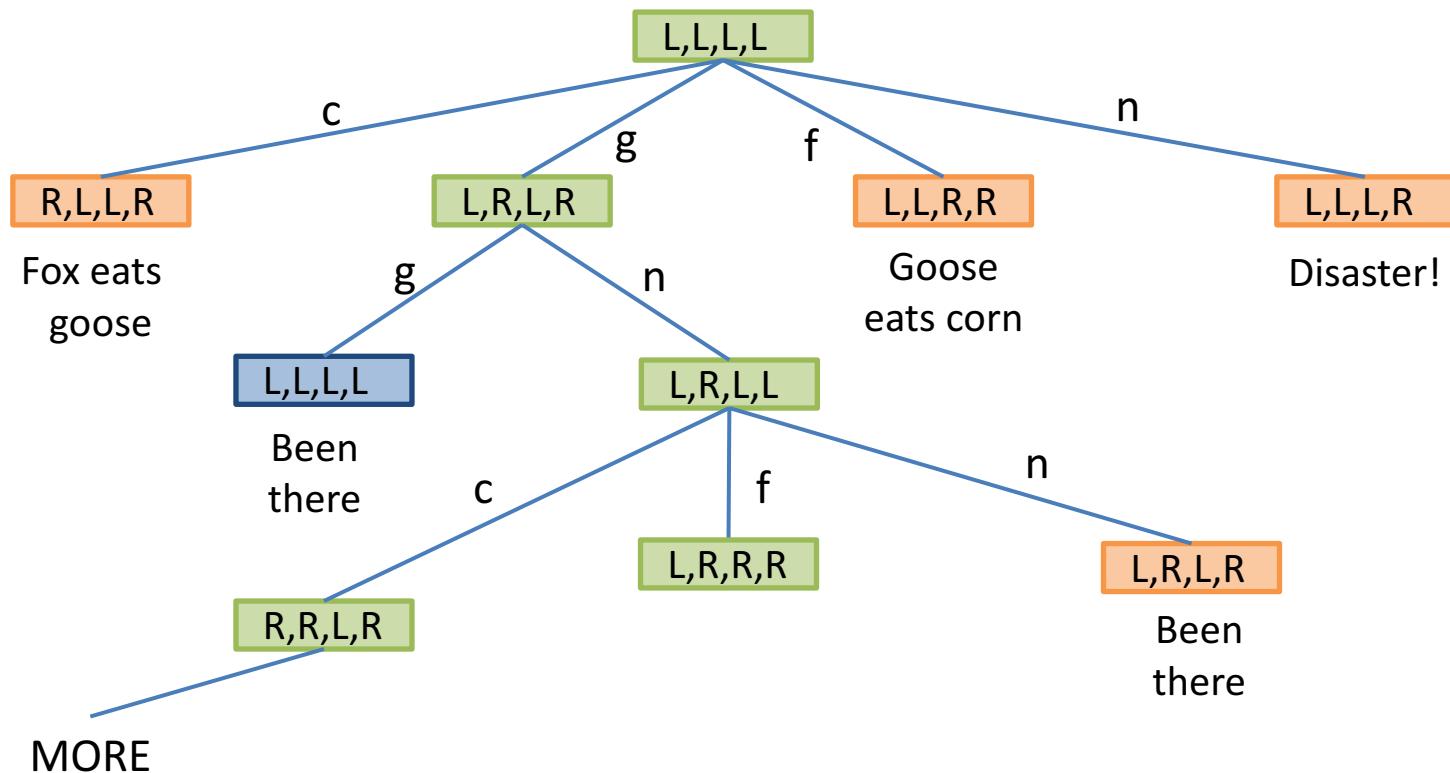
(Note: Initial state could be  $[R,R,R,R]$ , and search process would be similar)

For the second move, at [L,R,L,R] there are only two possibilities: *g* or *n*. Only one of them is worth doing.

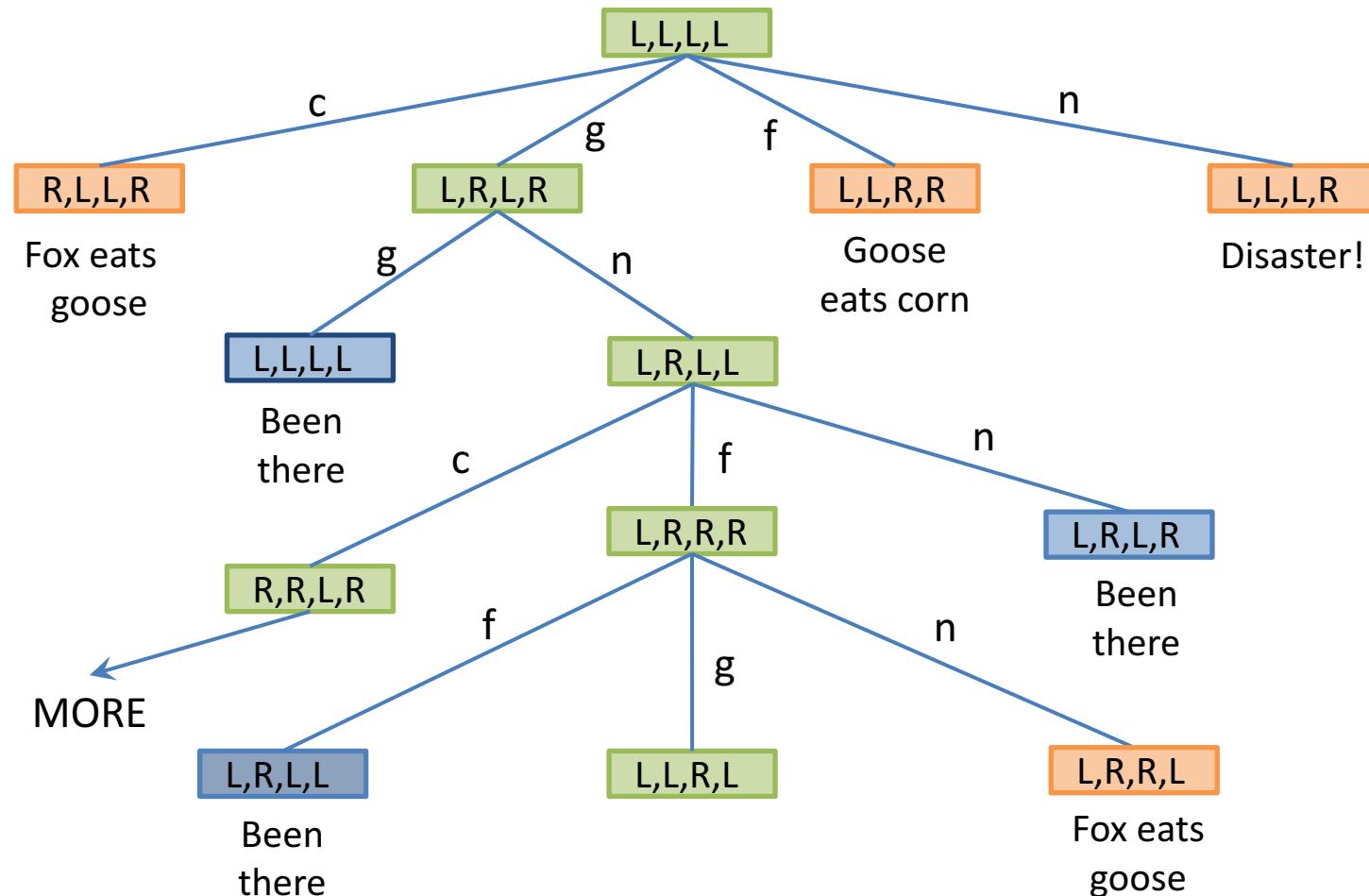


The structure we are developing is called a **search tree**.

For the third move, at  $[L,R,L,L]$  there are three possibilities:  $c$ ,  $f$ , or  $n$ . Two of them are viable. We will follow up one of them (by random choice).

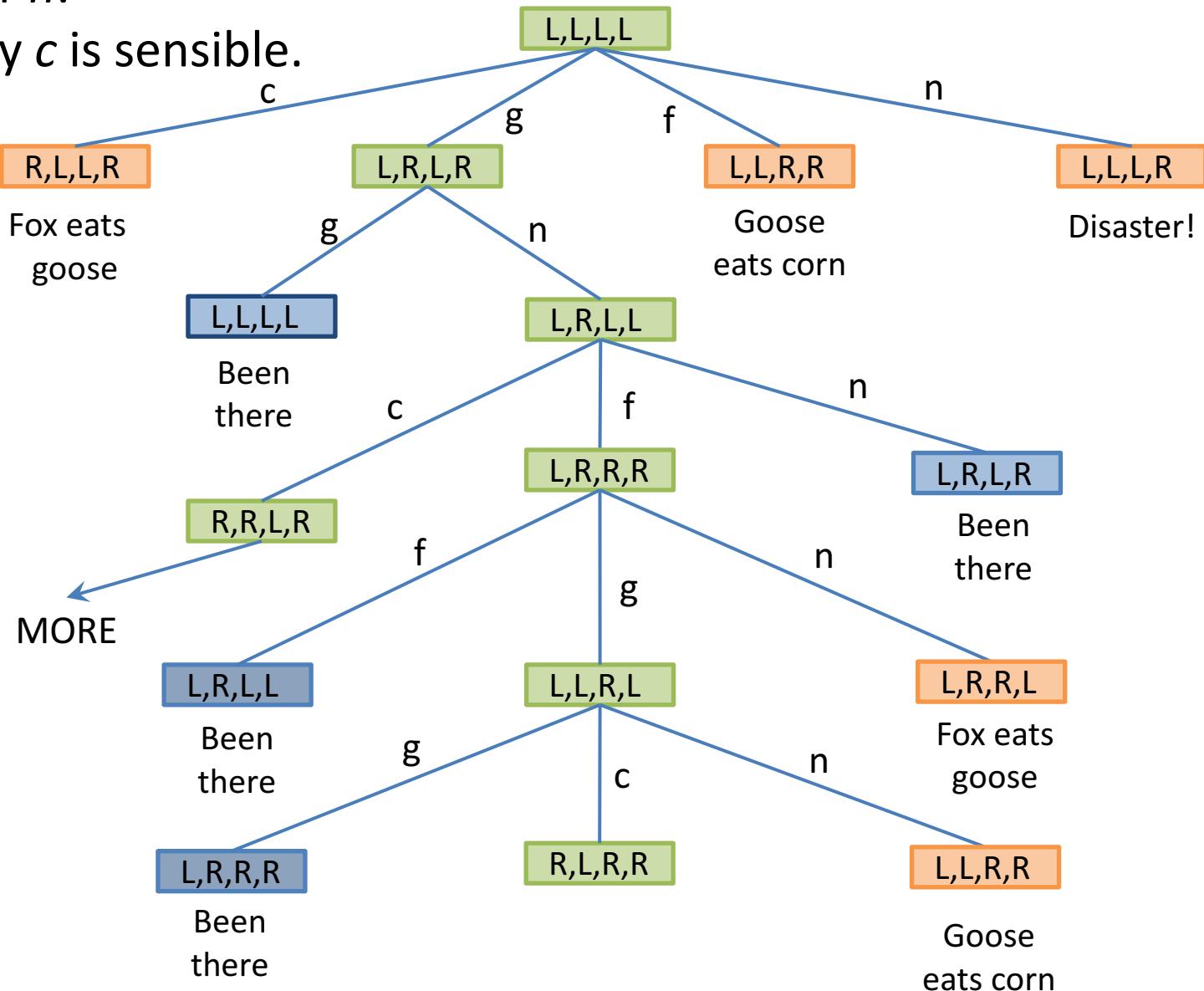


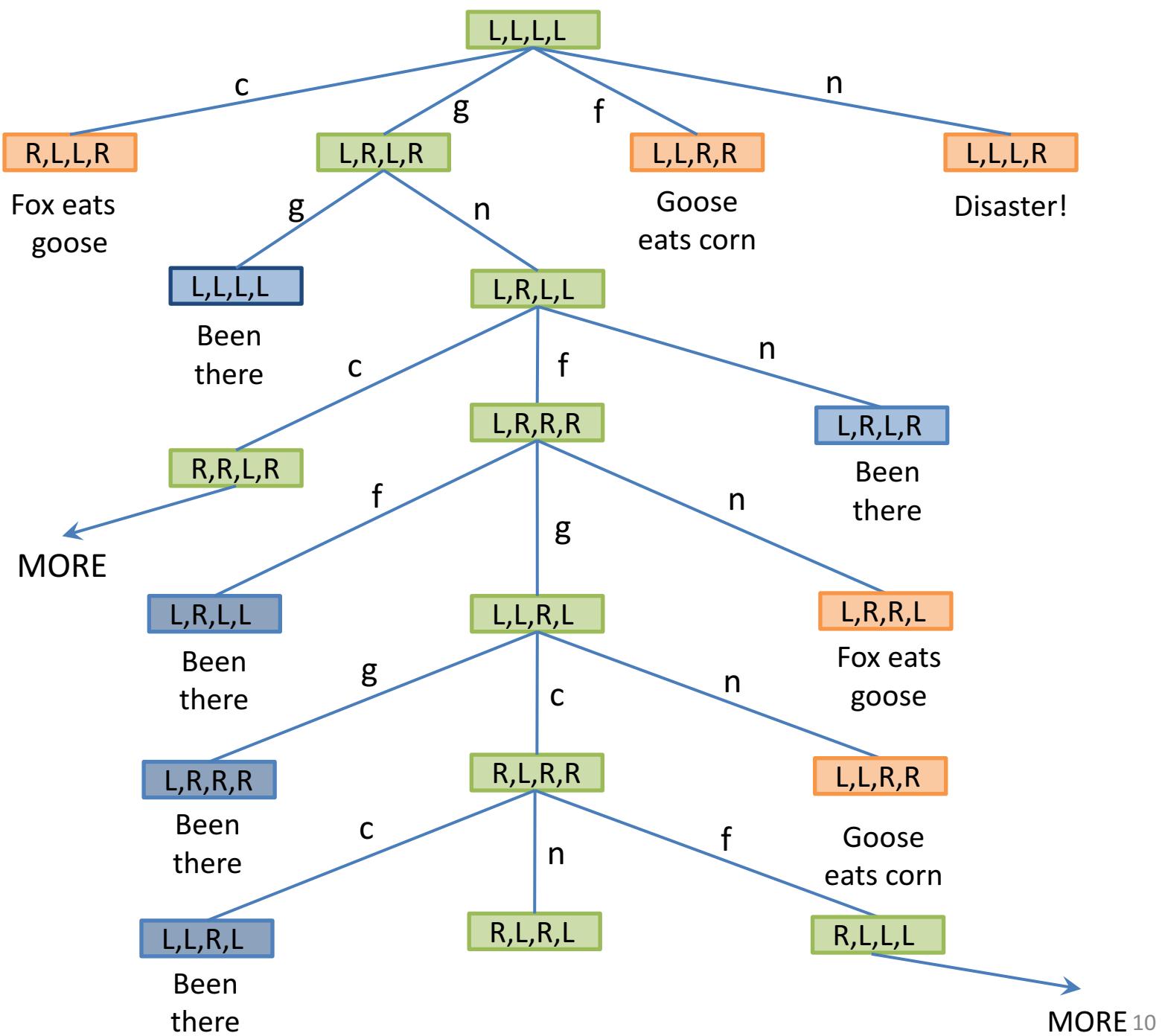
For the fourth move, at [L,R,R,R] there are three possibilities: *f*, *g*, or *n*. Only *g* is sensible.

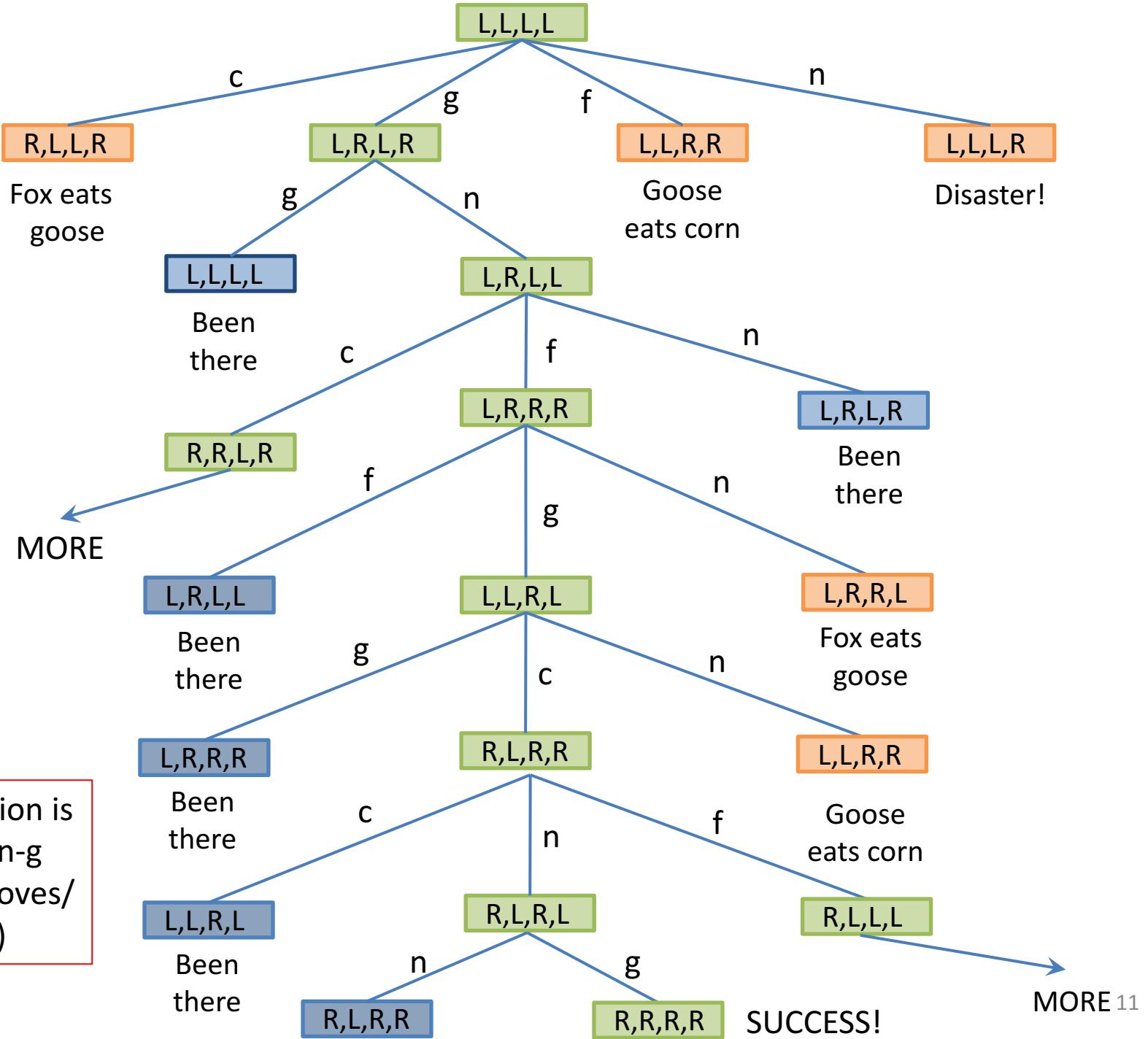


For the fifth move, at [L,L,R,L] there are three possibilities: *g*, *c*, or *n*.

Only *c* is sensible.





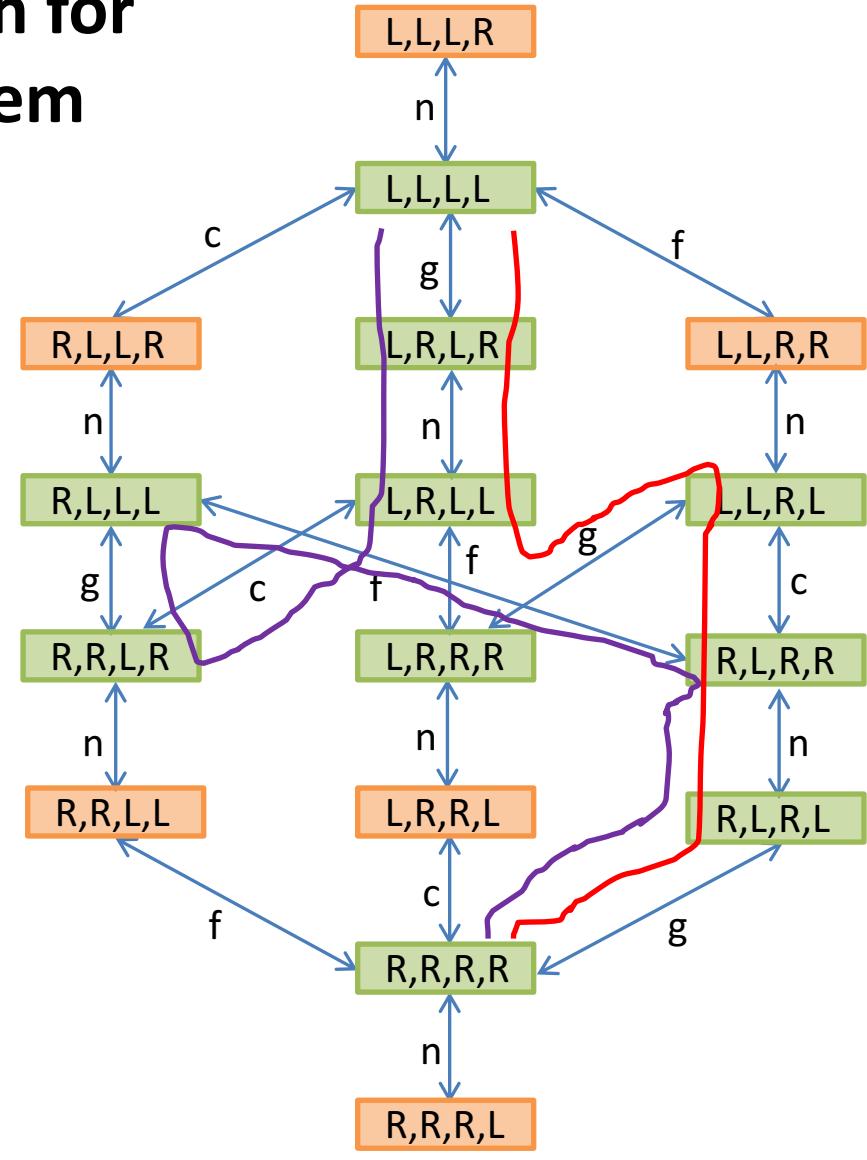


# State Space Diagram/Graph for Corn, Goose and Fox Problem

unviable

viable

There are 16 states in total, including 6 unviable states. There are 2 possible optimal solutions, each needing 7 crossings, and some non-optimal solutions (with more than 7 crossings).



# How we solved the problem

## Abstraction

We identified the **essential elements** of the problem.

## Problem/Knowledge Representation

We developed a simple way of representing any **situation / state** that could arise in the course of solving the problem:

In this case, a tuple indicating the location of each entity.

We also represented the **moves / actions** that could be made in solving the problem:

In this case four such moves that might be applicable in any situation.

Our representation was incomplete in that we did not specify what the **conditions and consequences** of these moves would be.

A representation suitable for a computer would have to include this.

## Search

We used a straightforward search method through the **possible moves** to find a solution to the problem (there are many advanced search strategies).

**PROBLEM/KNOWLEDGE REPRESENTATION**

**+**

**SEARCH**

**=**

**ARTIFICIAL INTELLIGENCE  
(TRADITIONAL)**

# State Space Representation

State space representation of problems has five key components:

## 1. A *State Space*, $S$

$S$  is a set of states. There is no limit to the number of states.

Each state is a distinct situation that could arise during attempts to solve the problem.

It is usually OK to also include states that could never arise.

## 2. An *Initial State*, $s_i \in S$

The state from which the problem solver begins

## 3. A Set of *Goal States*, $G \in S$

Note: This set may have only one member.

# State Space Representation (2)

## 4. A Set of Operators, $O$

Members of this set are the actions that can be taken or operations that can be applied in solving the problem. In principle, one operator only is ok.

## 5. A Transition Function: $T: S \times O \rightarrow S$

This denotes the effect of the actions available for solving the problem.

It defines the state transition produced by applying any operator  $o$  in any state  $s$ . The result is also a state.

The transition function is often specified by defining the transformation produced by each operator individually (pre- and post- conditions).

(see examples later)

# Representing Operators and Transition Function

There are many ways for representing operators.

One of the most useful is to define each operator in terms of its pre-conditions and post-conditions.

## ***Pre-conditions***

Facts that must be true immediately before the operator is to be applied.

The operator cannot be applied if its pre-conditions are not satisfied.

## ***Post-conditions***

Facts that must be true immediately after the operator has been applied.

Thus the post-conditions describe the effects of the operator.

We could informally represent the operators in the Corn, Goose and Fox problem:

### ***Operator c: (g, f, and n could be defined similarly)***

Pre-conditions: Boat/farmer on the same bank as corn.

Post-conditions: Corn is on opposite bank. Boat/farmer on opposite bank.

# The Three Jugs Problem

*You have three jugs which can hold exactly 8, 5, and 3 litres of wine, respectively.*

*The 8 litre jug is full to the brim with wine, the other two jugs are empty.*

*You must divide the wine into two equal amounts of 4 litres each accurately.*

*It is assumed that you can pour (decant) without spilling, but you cannot use any other vessels.*

*There is no calibration on the jugs.*

**(Is this more difficult than the corn, goose and fox problem?)**

# Formalising the Three Jugs Problem

## The State Space

We denote any possible state as a 3-tuple  $\langle x, y, z \rangle$  where

$x$  is the amount of wine in the 8 litre jug

$y$  is the amount of wine in the 5 litre jug

$z$  is the amount of wine in the 3 litre jug

$x, y, z$  are integers.

Hence  $0 \leq x \leq 8$ ,  $0 \leq y \leq 5$ , and  $0 \leq z \leq 3$ .

## Initial State

$\langle 8, 0, 0 \rangle$

## Goal State

$\langle 4, 4, 0 \rangle$

# Formalising the Three Jugs Problem (2)

## Set of Operators

*Decant\_X\_to\_Y, Decant\_X\_to\_Z, Decant\_Y\_to\_X*

*Decant\_Y\_to\_Z, Decant\_Z\_to\_X, Decant\_Z\_to\_Y*

## Transition Function

Operator: *Decant\_X\_to\_Y*

Pre-conditions:  $\langle x, y, z \rangle$ ,  $x > 0$  and  $y < 5$

Post-conditions:  $\langle x', y', z' \rangle$

IF  $x < (5 - y)$  THEN  $x' = 0, y' = y + x, z' = z.$

ELSE  $x' = x - (5 - y), y' = 5, z' = z.$

$x$  denotes value of  $X$  before operation;  $x'$  its value after the operation.

Five other operations could be defined in a similar way (see Java code).

# Representing the Three Jugs Problem in Java

```
public class ThreeJugState{  
  
    private static int[] capacity = new int[3]; // Holds the capacities of each jug  
    private int[] content = new int[3]; // Hold the current contents of each jug
```

# Representing the Three Jugs Problem in Java (2)

```
/** Constructors ----- */  
  
/**  
 * Constructor that creates a new state leaving capacities of each jug unchanged.  
 * x, y, and z are the contents of each jug  
 */  
public ThreeJugState(int x, int y, int z) {  
    content[0] = x;  
    content[1] = y;  
    content[2] = z;  
}  
  
/**  
 * Constructor that creates a new state and resets the capacities.  
 * (Default values of capacities are zero so this constructor should be used  
 * when a new problem is begun).  
 * x, y, and z are the contents of each jug  
 */  
public ThreeJugState(int capX, int capY, int capZ, int x, int y, int z) {  
    this(x, y, z);  
    capacity[0] = capX;  
    capacity[1] = capY;  
    capacity[2] = capZ;  
}
```

# Representing the Three Jugs Problem in Java (3)

```
public ThreeJugState decant(int source, int dest) {  
    if (source == dest) { // Source and dest are same jug  
        System.out.println("WARNING: Attempt to decant from and to same jug: " + source);  
        return this;  
    } else if (content[source] == 0) { // Nothing to pour  
        return this;  
    } else if (content[dest] == capacity[dest]) { // Destination already full  
        return this;  
    } else {  
        int remainingVolume = capacity[dest] - content[dest];  
        int newDestContent, newSourceContent;  
        if (remainingVolume >= content[source]) { // Empty source into destination  
            newSourceContent = 0;  
            newDestContent = content[dest] + content[source];  
        } else { // Fill up destination from source leaving remainder in source  
            newDestContent = capacity[dest];  
            newSourceContent = content[source] - remainingVolume;  
        }  
        ThreeJugState newState = this.cloneState();  
        newState.setContent(source, newSourceContent);  
        newState.setContent(dest, newDestContent);  
        return newState;  
    }  
}
```

# Representing the Three Jugs Problem in Java (4)

```
/* Setting and Getting */

public int getContent(int j) { return content[j]; }

public void setContent(int j, int vol) { content[j] = vol; }

public int getCapacity(int j) { return capacity[j]; }

public void setCapacity(int j, int vol) { capacity[j] = vol; }

/**
 * cloneState creates a new state identical to current state
 *
 * @return a new state whose content values are same as those of current state.
 */
public ThreeJugState cloneState() {
    return new ThreeJugState(content[0], content[1], content[2]);
}

}
```

# The 8-Puzzle Problem

*The 8-puzzle consists of a 3 by 3 grid. On each grid square there is a tile, except for one square that remains empty. The 8 tiles are numbered from 1 to 8 respectively, so that each tile can be uniquely identified. A tile next to the empty square can be moved into the empty space, leaving its previous square empty in turn. The objective of the 8-puzzle problem is to find a solution, a sequence of tile moves, which leads from any initial grid configuration to a given goal grid configuration, e.g.,*

8	7	6
5	4	3
2	1	

Initial State

1	2	3
4	5	6
7	8	

Goal State

***Your assignment will be to write a Java program to solve this problem.***

# Summary

## Toy problems

Corn, Goose and Fox

Three Jugs

*Has this lecture helped  
improve your problem  
solving skills?*

## Finding a solution

Abstracting the essential features of a problem

Systematically searching for a solution

## State space representation (Problem formalisation)

State space

Initial and goal states

Operators

Transition function

*How to search effectively  
and efficiently?*



# CE213 Artificial Intelligence – Lecture 3

## Blind Search Strategies

We now have a way of representing problems:

Formulate them as state spaces

To solve them we need:

Some way of searching for a sequence of transitions  
that will take us from the initial state to a goal state

# Strategies for Searching State Spaces

The objective of a **search strategy** is to systematically explore the state space by constructing a **search tree**.

*Key question in search tree construction:*

*Which state (or node) to be expanded first and how?*

There are many ways in which this can be done, such as **blind search** and **heuristic search**.

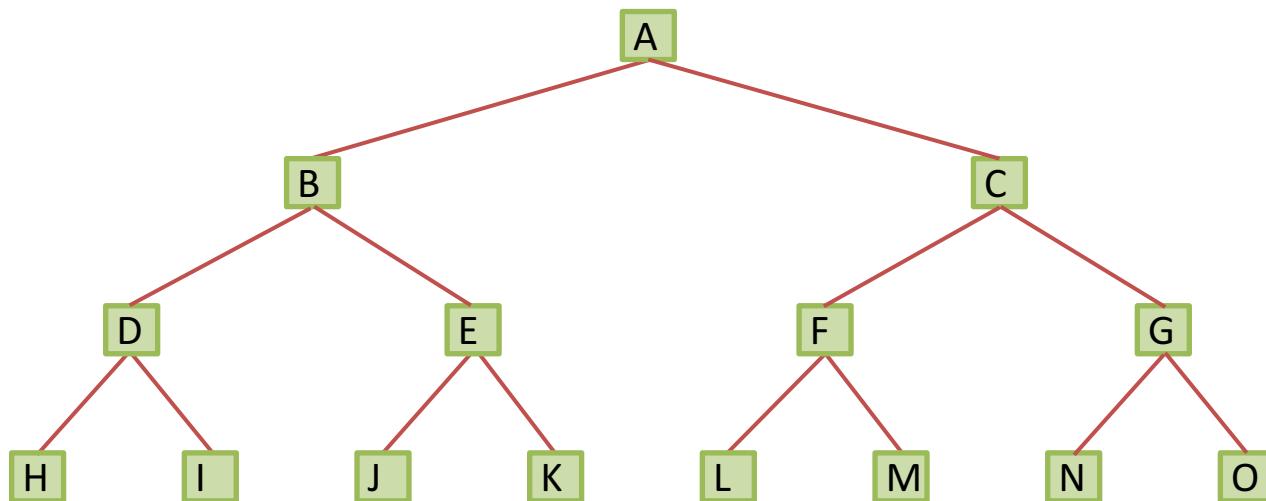
Note: The search space may be very large, possibly infinite.  
Exhaustive search may be unrealistic.

# A Binary Search Tree

We first consider the simplest possible search state space:

One in which there are only two possible transitions from each state.

This will lead to a binary search tree:



A search strategy determines the order in which the tree is constructed.

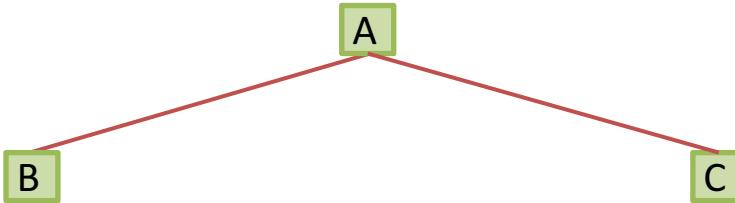
# Node Generation and Node Expansion

Basic elements in a search tree:

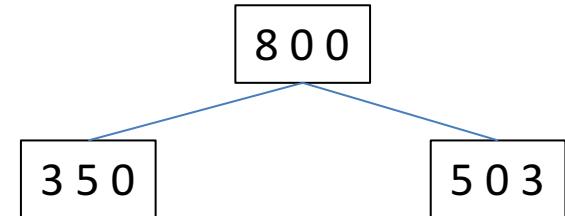
A **node** represents a state and its related information such as cost, heuristic value and parent node; A **branch** represents an action or move.

A node is **generated** when it is added to the tree.

A node is **expanded** when its successors (children) are added to the tree.



e.g. for the 3 jugs problem



At this point:

Node A has been expanded.

Nodes B and C have been generated but not expanded.

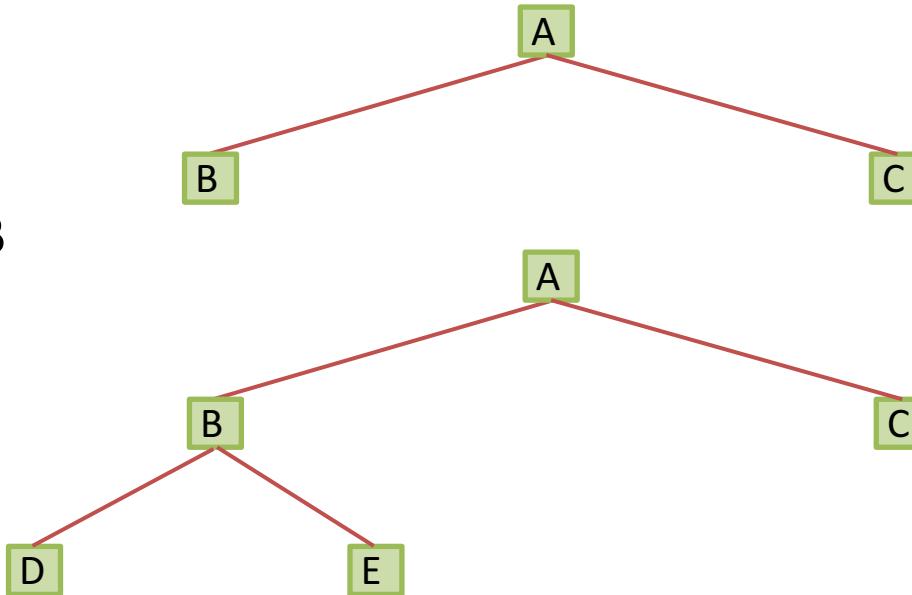
# Breadth First Search

In ***breadth first search***, all the nodes at depth n from the root must be expanded before any node at depth n+1.

In other words, **the *least recently generated*** unexpanded node is chosen for expansion.

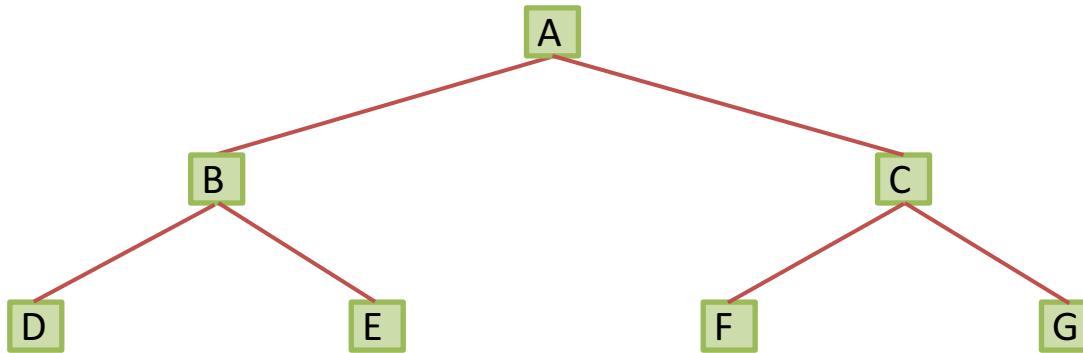
So we begin by expanding Node A:

Then Node B

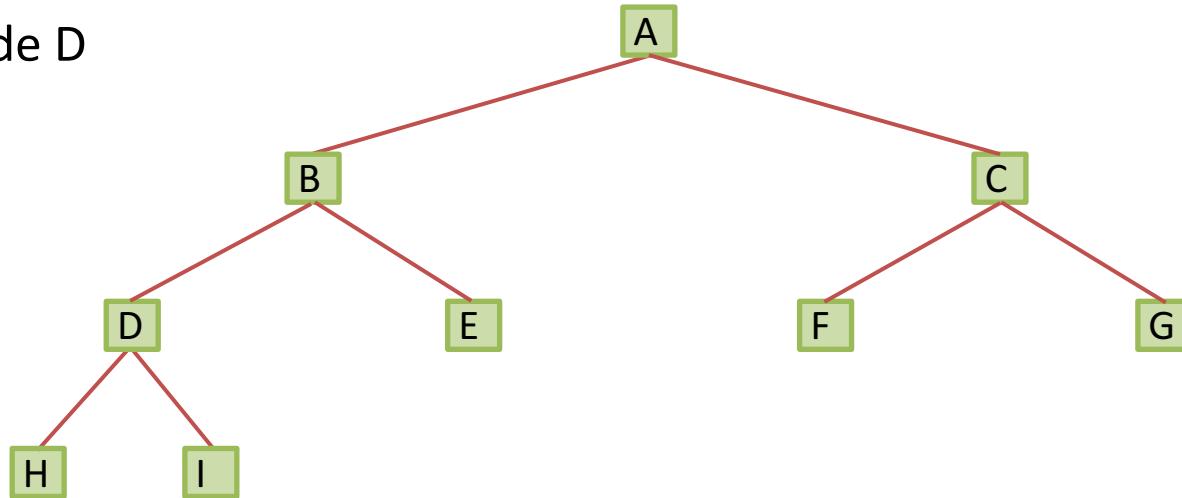


# Breadth First Search (2)

Then Node C

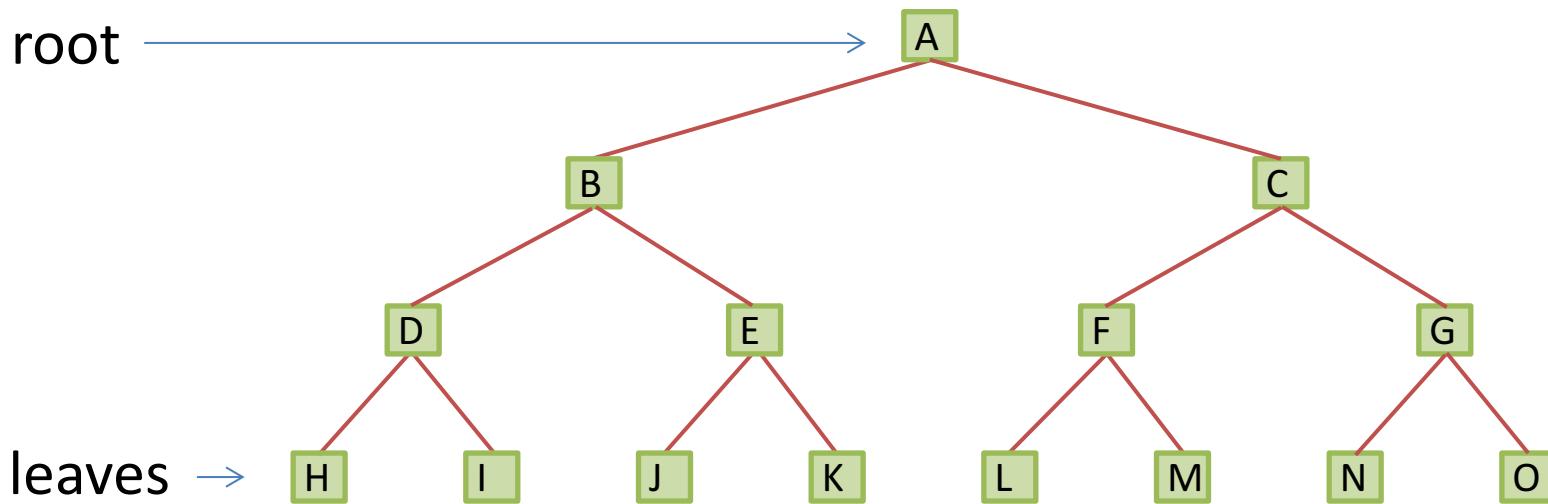


Then Node D



## Breadth First Search (3)

And then Node E, then Node F, then Node G to produce:



*A search strategy is therefore simply a rule for deciding which node to expand next.*

Note that the generated but unexpanded nodes are the leaves of the tree (H, I, J, K, L, M, N, O are leaves of the above search tree).

# Pseudo Code for Breadth First Search

```
Create an empty list UnexpNodes1);  
{Holds list of unexpanded nodes}  
Add initial node to UnexpNodes;  
WHILE (UnexpNodes not empty)  
    N = First item of UnexpNodes2)  
    Remove N from UnexpNodes  
    IF N is goal  
        THEN Produce Solution3) and Return(Success);  
    Expand N to produce list of successors of N4);  
    Add successors to tail of UnexpNodes;  
Return(Failure)
```

Would it be better  
to check whether  
any successor is  
goal here?

- 1) A node may contain state, cost, parent, etc.
- 2) Which node to be expanded next? – depending on search strategy adopted
- 3) How to produce a solution? – backtracking the parents from the goal node found
- 4) How to expand a node? – depending on the problem, e.g., possible operators

## Pseudo Code for Breadth First Search (2)

It would be possible to check for the goal state immediately after a new node is generated by expanding its parent.

This would save (a little) time.

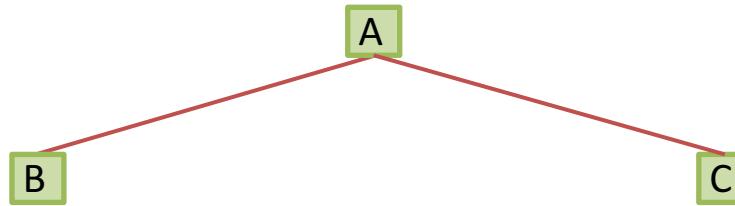
However, waiting until the node is chosen for expansion is important in some of the searches we will consider later.

e.g., uniform cost search, to ensure the solution found is the optimal one.

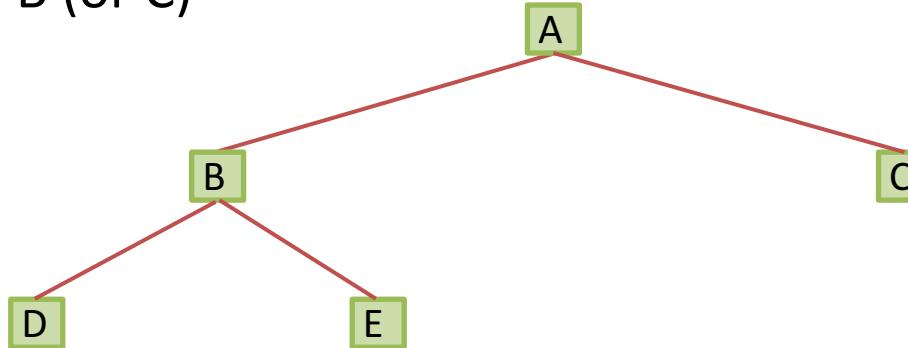
# Depth First Search

In *depth first search*, the *most recently generated* unexpanded node is chosen for expansion.

Again we begin by expanding Node A:

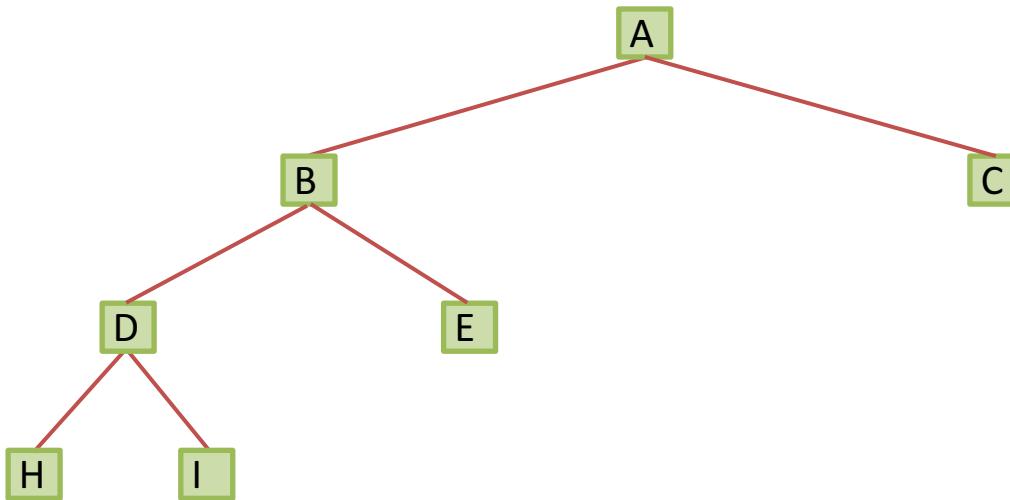


And then Node B (or C)



## Depth First Search (2)

Next Node D (or E) (not C) is chosen for expansion:

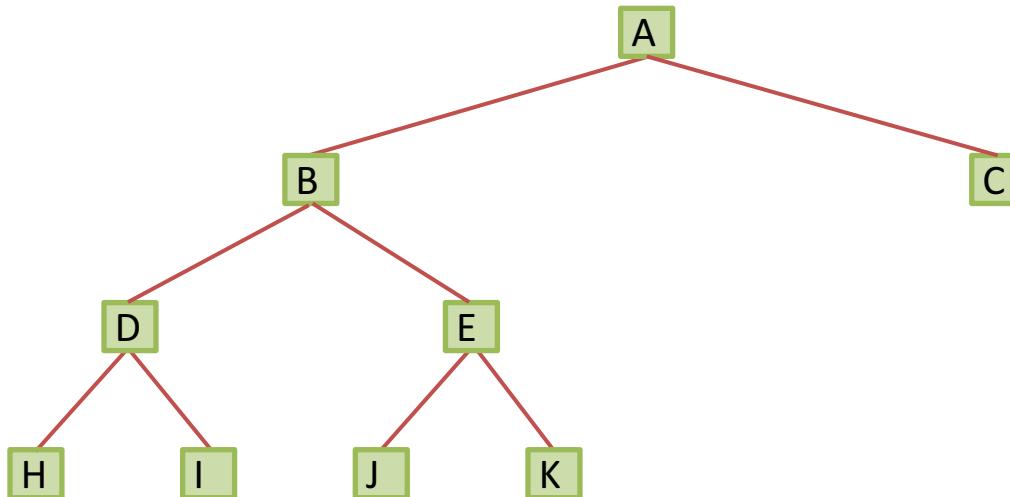


Assuming Nodes H and I have no successors, we cannot expand these. So the search process **backtracks** (very important concept in this search).

## Depth First Search (3)

The most recently generated unexpanded node is Node E.

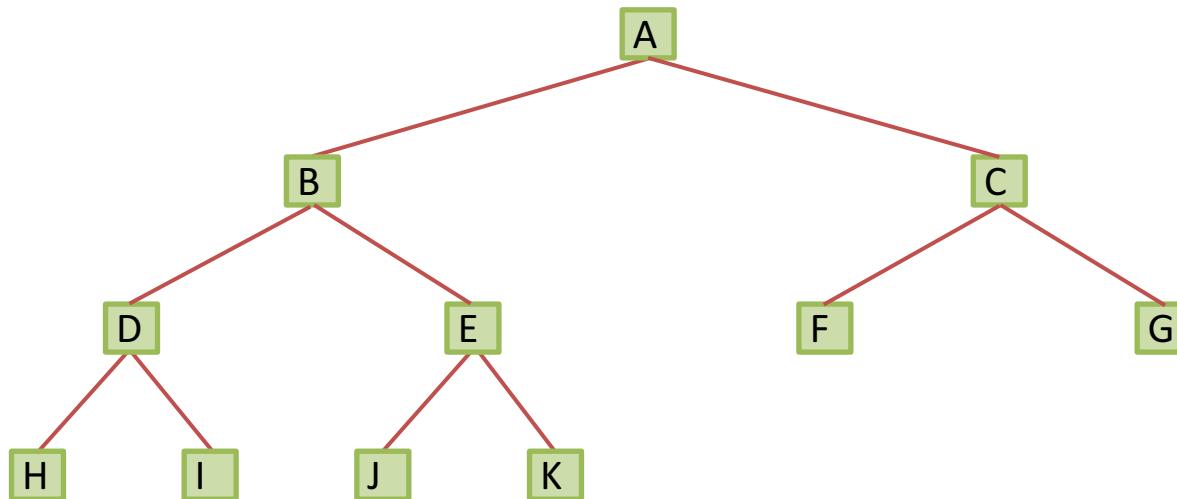
So this is chosen for expansion:



Note that Nodes H and I were removed from UnexpNodes before backtracking (see pseudo code later).

## Depth First Search (4)

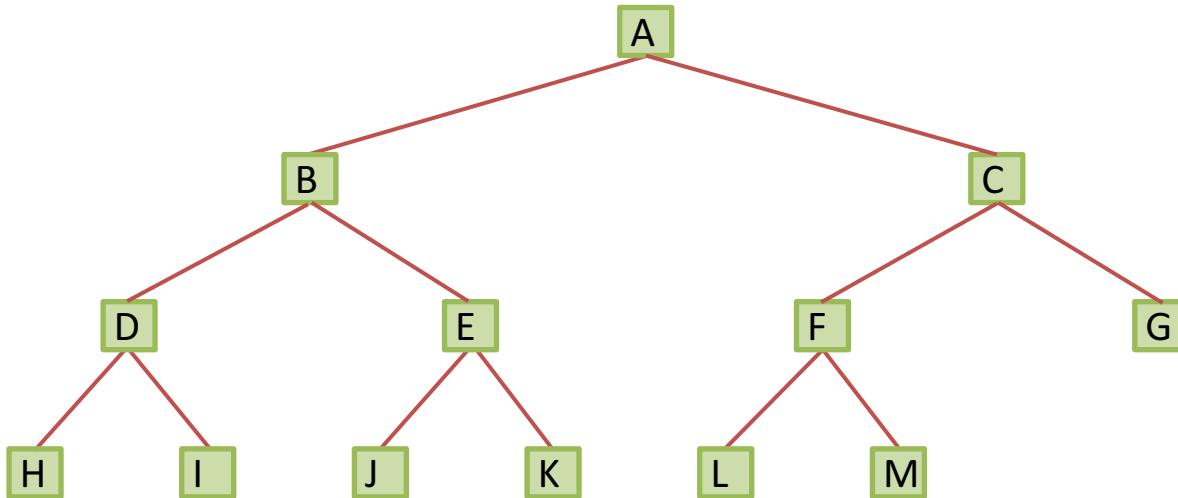
Assuming Nodes J and K have no successors. Once again we have to backtrack: Node C is now the only unexpanded node so this must be chosen next.



Note that Nodes J and K were removed from UnexpNodes before backtracking (see pseudo code later).

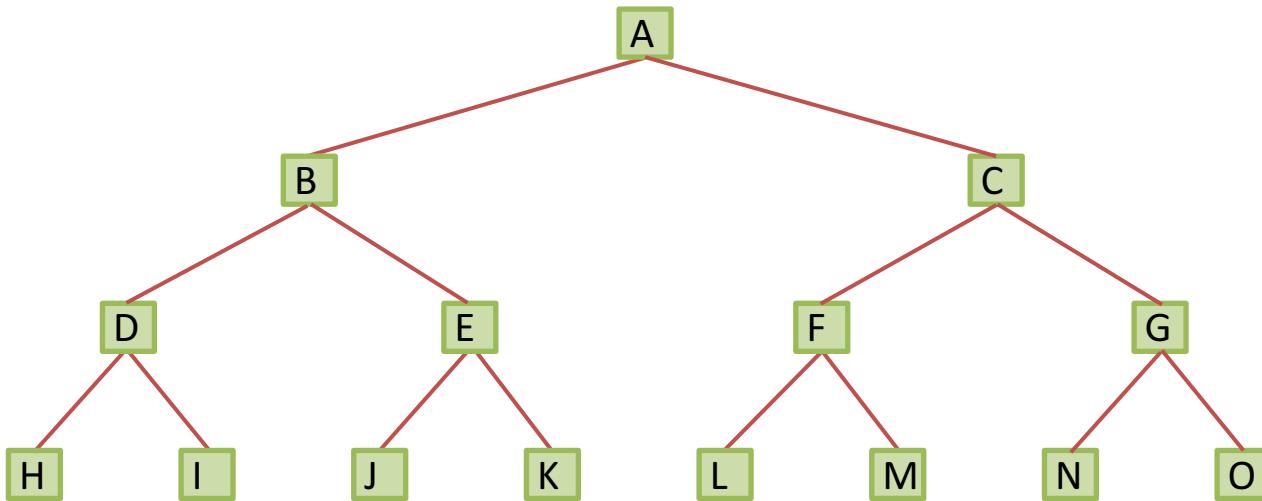
# Depth First Search (5)

Next Node F is chosen:



## Depth First Search (6)

Finally we backtrack again and expand Node G:



Note that Nodes L and M were removed from UnexpNodes before backtracking (see pseudo code later). Nodes N and O have no successor.

# Pseudo Code for Depth First Search

```
Create an empty list UnexpNodes;  
{Holds list of unexpanded nodes}  
Add initial node to UnexpNodes;  
WHILE (UnexpNodes not empty)  
    N = Last item of UnexpNodes //only change from BFS  
    Remove N from UnexpNodes  
    IF N is goal  
        THEN Produce Solution and Return(Success);  
    Expand N to produce list of successors of N;  
    Add successors to tail of UnexpNodes;  
Return(Failure)
```

# Blind Searching

Depth first search and breadth first search are called ***blind search*** strategies

They use no additional information in selecting which node to expand next

Blind search is also called ***uninformed search***.

# Comparing/Evaluating Search Strategies

We now have two search strategies – Which one is better?

## Four Criteria:

### *Completeness*

Is the strategy certain to find a solution if there is one?

### *Optimality*

If there is more than one solution, does the strategy find the best one?  
(Clearly this may depend on what is meant by ‘best’.)

### *Time Complexity*

How long does the strategy take to find a solution?

### *Space Complexity*

How much memory does the strategy require?

# Completeness

## *Breadth First*

Complete.

Suppose a goal state exists at depth  $d$  from the root.

The program will eventually expand all nodes at depth  $d$ .

## *Depth First*

Not complete, if the state space is infinite.

If the search space is infinite the program may never explore a branch containing a goal state.

However, if the search space is finite and the program checks for duplicate states on the current path then this method is complete.

# Optimality

Let a best solution be the one with a minimal number of moves.

## *Breadth First*

Optimal.

First solution encountered clearly has minimum number of moves.

## *Depth First*

Not optimal.

It depends entirely on which branches the program happens to explore first.

# Time Complexity

The time complexity and space complexity depend upon the size of the tree to be searched.

We define the size of a search tree as follows:

The branching factor for every node is  $b$ .

The goal state lies at depth  $d$  from the root.

The maximum depth is  $m$ .

For complexity, we usually consider the worst case scenario.

**Time complexity is about the time required for expanding nodes, depending on *the number of nodes expanded*.**

# Time Complexity (2)

## Breadth First

The number of nodes expanded will be at most (worst case scenario):

$$1 + b^1 + b^2 + \dots + b^d$$

Hence the time complexity is  $O(b^d)$ .

Thus the time complexity increases *exponentially* as the branching factor is increased.

(If you are not familiar with the Big O notation, please visit  
[https://en.wikipedia.org/wiki/Big\\_O\\_notation](https://en.wikipedia.org/wiki/Big_O_notation))

## Depth First

In the worst case scenario the solution could be found only when the last node is reached, so all the nodes may have to be expanded.

Hence the time complexity is  $O(b^m)$ .  $m \geq d$

Note, however, that if there are a large number of possible solutions, the actual time may be much less, and depth first search may be faster than breadth first search.

# Space Complexity

**Space complexity is about the memory required for storing nodes, depending on *the number of nodes unexpanded*.**

## **Breadth First**

It is necessary to keep all of the nodes at depth  $n$  in memory just prior to expanding the nodes at depth  $n + 1$ .

So the space complexity is also  $O(b^d)$ . (the same as its time complexity)

## **Depth First**

When the program backtracks it discards the nodes it has already visited.

So it need only store the unexpanded nodes on the path from the root to the current node and their immediate successors.

At each depth, at most  $b$  unexpected nodes need to be stored.

Hence the space complexity is only  $O(bm)$ . This is the only advantage of depth first search, compared to breath first search.

# The practical effect of exponential complexity

Assume: Branching factor  $b = 10$ ,

1 million nodes expanded per second,

Each node needs 1000 bytes of memory.

Depth	Nodes	Time	Memory
2	110	0.11 millisecs	107 Kbytes
4	11,110	11 millisecs	10.6 Mbytes
6	$10^6$	1.1 seconds	1 Gbytes
8	$10^8$	2 minutes	100 Gbytes
10	$10^{10}$	3 hours	10 Terabytes
12	$10^{12}$	13 days	1 Petabytes
14	$10^{14}$	3.5 years	100 Petabytes
16	$10^{16}$	350 years	10 Exabytes

# More Sophisticated Blind Search Strategies

- Depth Limited Search
- Iterative Deepening Search
- Uniform Cost Search  
(also known as Dijkstra's algorithm)

Basic ideas: Limiting the depth in depth first search or introducing a specific cost rather than the number of moves

→ ***Better than breadth first or depth first search?***

# Depth Limited Search

If depth first search makes a wrong choice near the start, it may waste a great deal of time exploring ever deeper in a portion of the tree that contains no solutions.

In the case of an infinite search space it will never terminate.

***Depth Limited Search*** addresses this problem.

**A maximum limit,  $m$ , is placed on the depth that can be explored before backtracking occurs.**

So, in the worst case, the program searches the entire tree to depth  $m$ .

Time complexity  $O(b^m)$

Space complexity  $O(bm)$ .

Provided there is a solution with path length less than  $m$ , it will be found.

# Iterative Deepening Search

If you cannot work out in advance what the maximum depth should be. You cannot safely use depth-limited search.

***Iterative Deepening Search*** provides a simple solution to this problem:

`m = 1;`

`REPEAT`

`Do depth limited search to depth m;`

`m = m + 1;`

`UNTIL Solution Found;`

Clearly, like breadth first, this strategy will eventually find an optimal solution. So iterative deepening search is

complete

optimal

low space complexity:  $O(bm)$ ,

high time complexity (repetition):  $O(b^m)$

An attractive  
search strategy

# Uniform Cost Search

So far we have assumed that an optimal solution is one that has the shortest path length, i.e., has the minimum number of moves/operations/actions.  
What happens if some operators have higher costs than others?

**Uniform Cost Search** takes operator costs into account. (It is not breath-first or depth-first search!)

Nodes are selected for expansion as follows:

Compute the cost of a node as ***the sum of the costs of the operations leading to it from the root.***

Always expand the cheapest node next.

Provided costs are never negative, the solution found will be the cheapest (optimal!).

# Pseudo Code for Uniform Cost Search

```
Create an empty list UnexpNodes;  
{Holds list of unexpanded nodes}  
Add initial node to UnexpNodes;  
WHILE (UnexpNodes not empty)  
    N = Lowest cost item of UnexpNodes  
    Remove N from UnexpNodes  
    IF N is goal  
        THEN Produce Solution and Return(Success);  
        Expand N to produce list of successors of N;  
        Compute costs of reaching successors by adding  
        transition costs to cost of N.  
        Add successors with costs to tail of UnexpNodes; 1)  
Return(Failure)
```

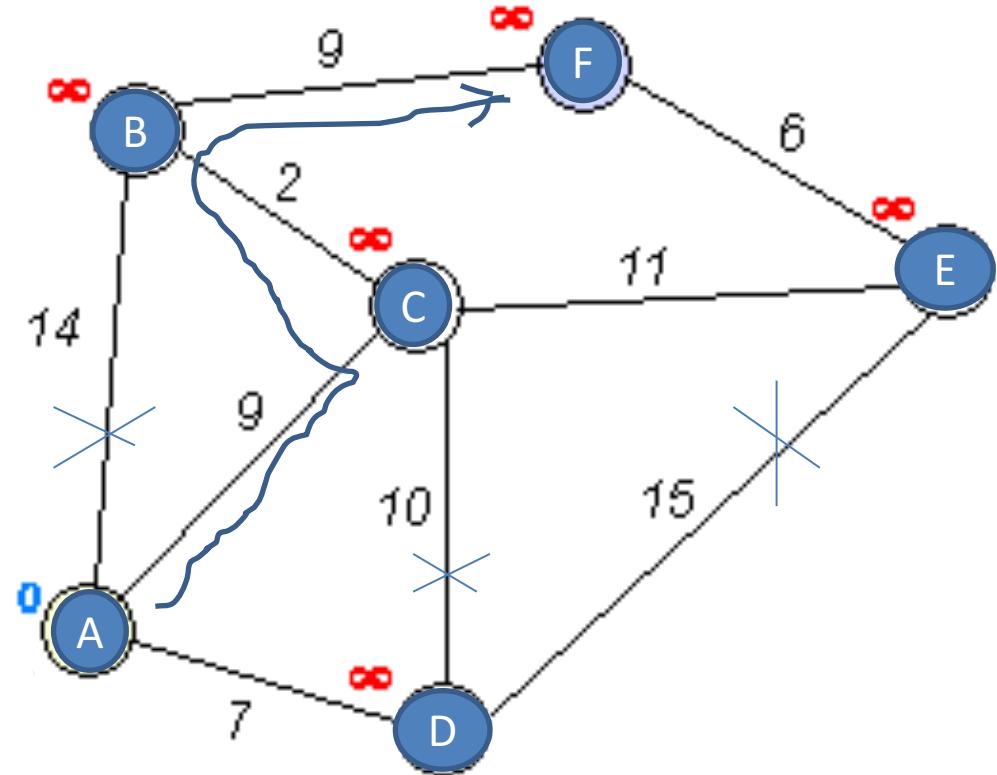
Will do exercises with toy problems in the class, which will help you a lot in understanding the pseudo code!

Note 1: If a cheaper route to a node in UnexpNodes has been found, replace its costlier parent node with the cheaper one.

# An Example of Uniform Cost Search

There are 6 towns: A, B, C, D, E, and F. The numbers alongside the lines indicate the distances between the towns. Find the shortest road from town A to town F using uniform cost search:

Expand node A: produce new nodes B, C, and D.  
Choose a node with minimum cost from nodes B, C, D to expand.  
Expend D: produce a new Node E.



Choose a node from nodes B, C, E to expand.  
Expand C: produce no new node, but update costs of nodes B (14->11) and E (22 -> 20).

.....

# Summary

## Blind (Uninformed) Search Strategies

- Breadth First Search
- Depth First Search
- Depth Limited Search
- Iterative Deepening Search
- Uniform Cost Search

## Comparison of Search Strategies

- Completeness
- Optimality
- Time Complexity
- Space Complexity



# CE213 Artificial Intelligence – Lecture 4

## Heuristic Search Strategies

None of the blind search strategies does something that any human solver would do:

i.e., consider how close states appear to be to the goal when deciding what to try next.

To do this we will have to provide two things:

A way of assessing how close a state is to the goal

A way of using this additional information in the search

# Heuristic and Evaluation Function

Suppose that in addition to the state space formalisation of a problem, the solver also has a function that can estimate how close a state is to a goal state.

Such function is called ***evaluation function or heuristic:***

$$h : S \rightarrow \mathbb{R} \quad (\mathbb{R} : \text{the set of real numbers})$$

We use heuristics all the time in everyday life.

e.g., Suppose you are looking for W.H. Smiths in a strange town.

A useful heuristic is that it is probably pretty close to Marks and Spencers.

An evaluation function doesn't have to be perfect to be useful.

# Heuristic Search Strategies

Heuristic search strategies are those that use a heuristic evaluation function in determining **which node to expand next**. (a key question to be answered by any search strategy)

Much of AI is concerned with heuristics

They make problems with exponential time complexity tractable, by reducing search space or avoiding searching part of state space where there is surely no goal state.

— Benefit from using heuristics

# Route Finding Problems

## SatNav

- Use GPS to determine where you are now.
- Use heuristic search to find best route to where you want to be.

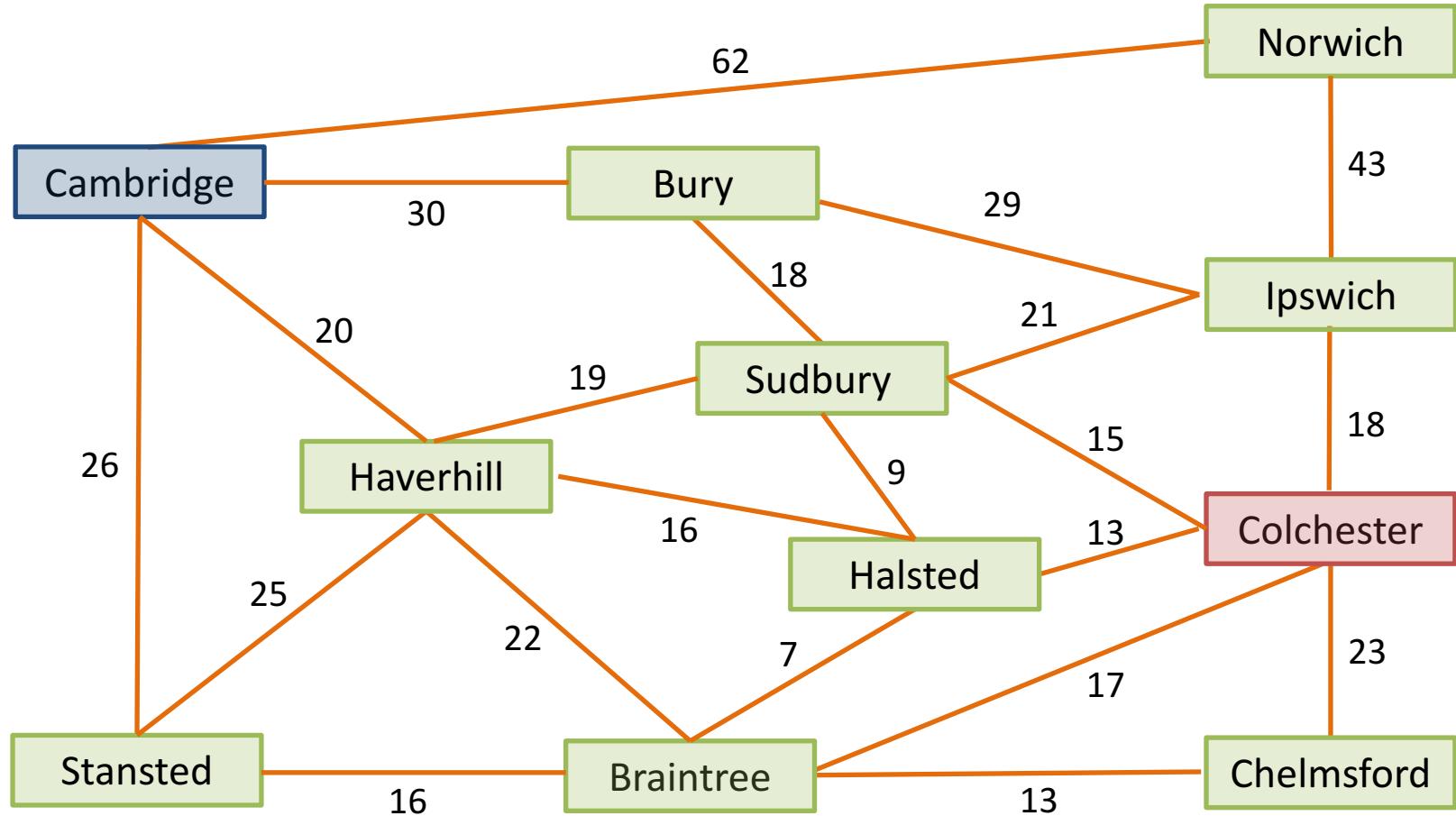
Given a starting point and a destination

Return the ‘best’ route

## An Example:

Suppose we want to drive from Colchester to Cambridge

# Major Roads in East Anglia (as a state space)



# Heuristic Distance Estimates

Straight line distances of towns from Cambridge

Town	Direct Distance
Braintree	33
Bury	26
Cambridge	0
Chelmsford	36
Colchester	40
Halsted	29
Haverhill	16
Ipswich	45
Norwich	59
Stansted	24
Sudbury	28

$$h : S \rightarrow \mathbb{R}$$

(a lookup table defining an evaluation function)

# Greedy Search

Greedy search is the simplest and most obvious form of heuristic search:

It always chooses the node that is closest to a goal state in terms of the heuristic, regardless of the distance (cost) between the initial state (root) and the current state (node).

# Greedy Search (2)

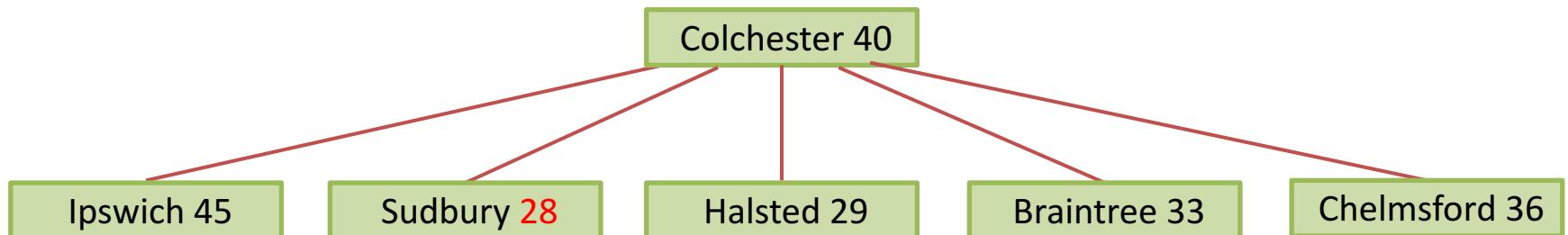
Applying this to the route finding problem:

Clearly the root node is Colchester:

Colchester 40

For convenience, the distance heuristic is indicated in the node.

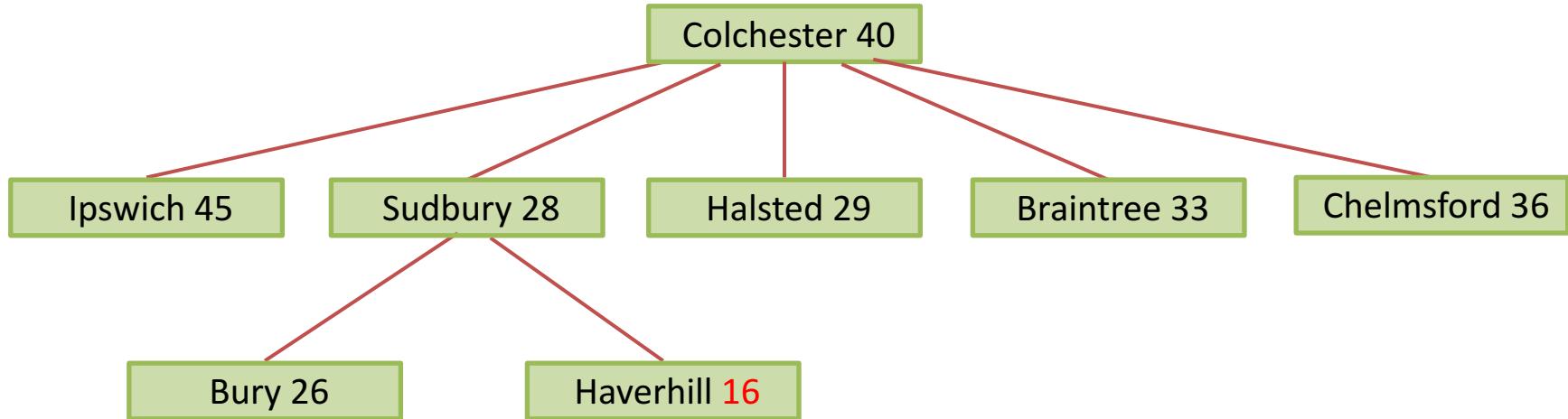
Expanding this gives:



The heuristic indicates that Sudbury is closest to the goal among the 5 unexpanded nodes, so this node is expanded next:

# Greedy Search (3)

Expanding Sudbury:

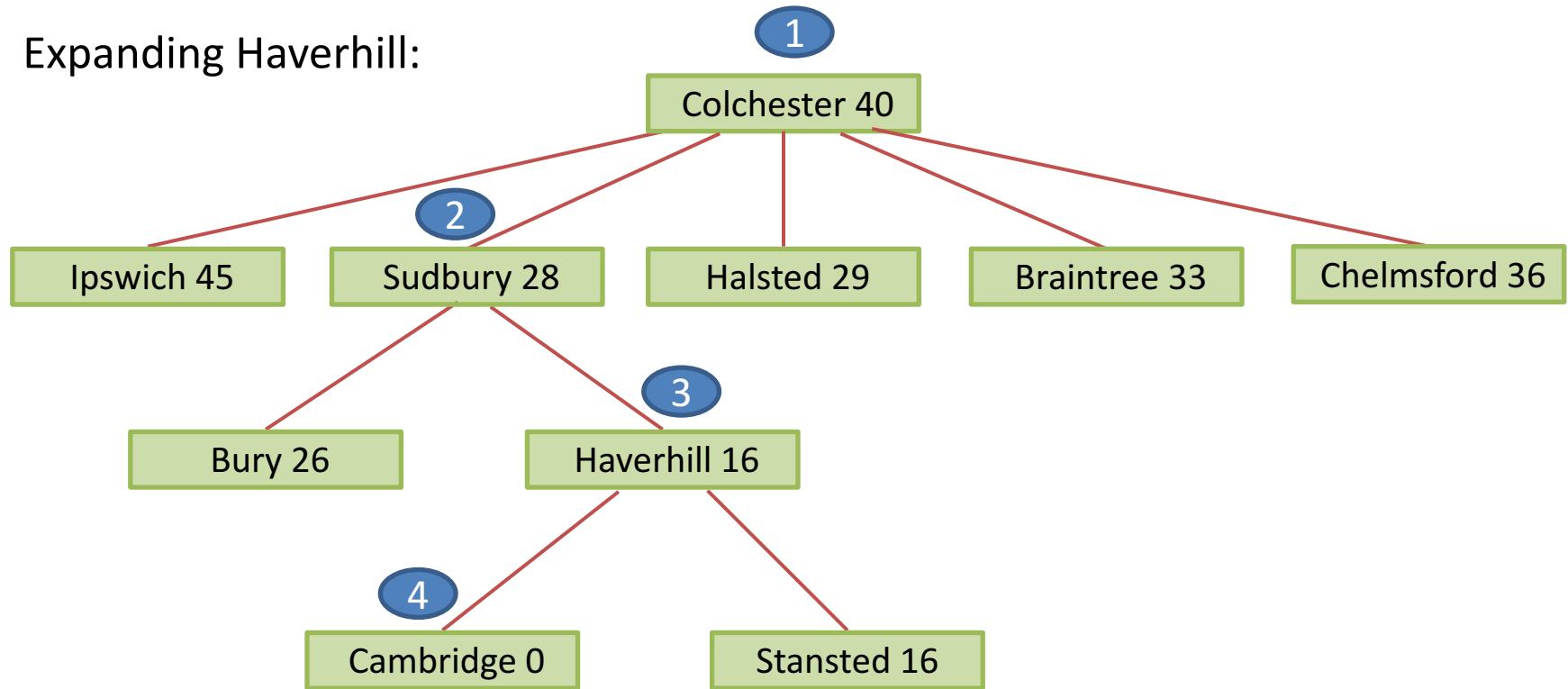


New nodes for towns that have already been reached have not been added.

At this point, the heuristic indicates that Haverhill is closest to the goal among the **6** unexpanded nodes (They are at different depth), so this node is chosen for expansion:

# Greedy Search (4)

Expanding Haverhill:



And the problem is solved: Colchester-Sudbury-Haverhill-Cambridge  
(4 nodes have been selected for expansion.)

## Greedy Search (5)

Two features of greedy search should be noted:

- The solver moves fairly directly toward a solution  
Nodes such as Norwich are never reached.  
Nodes such as Ipswich and Chelmsford are never expanded.  
So the search time is kept low.
- The solution is quite good, but **not optimal**.

Route found:

**Colchester-Sudbury-Haverhill-Cambridge**

Length 54 miles

Shortest route is only 49 miles (Colchester-Halsted-Haverhill-Cambridge)

# Pseudo Code for Greedy Search

```
Create an empty list UnexpNodes;  
Add initial node to UnexpNodes;  
WHILE (UnexpNodes not empty)  
    N = The item n in UnexpNodes, whose heuristic  
        value h(n) is smallest  
    Remove N from UnexpNodes  
    IF N is goal  
        THEN Produce Solution and Return(Success);  
    Expand N to produce list of successors of N;  
    Add successors to tail of UnexpNodes; 1)  
Return(Failure)
```

Note 1: If a second route is found to a node that has already been reached before, it is not necessary to add it to the unexpanded nodes list.

## A\* Search

Why does the greedy search fail to find an optimal solution?

Because it takes no account of the **cost of reaching a node from the root**, and makes decisions purely on the basis of the estimated **cost to go from the current node to the goal**.

But we already have uniform cost search that takes account of cost of reaching a node from the root.

Why not combine greedy search and uniform cost search?

If we do this, we have **A\* Search**.

## A\* Search (2)

Suppose:

$g(n)$  is actual **cost of getting to node  $n$  from initial state (root)**.

$h(n)$  is heuristic estimate of **cost of getting to goal from node  $n$** .

Then we have three alternative search strategies:

***Uniform Cost:***

Expand the node with smallest  $g(n)$

***Greedy:***

Expand the node with smallest  $h(n)$

***A\*:***

Expand the node with smallest  $g(n) + h(n)$

# A\* Search (3)

Applying A\* Search to the route finding problem.

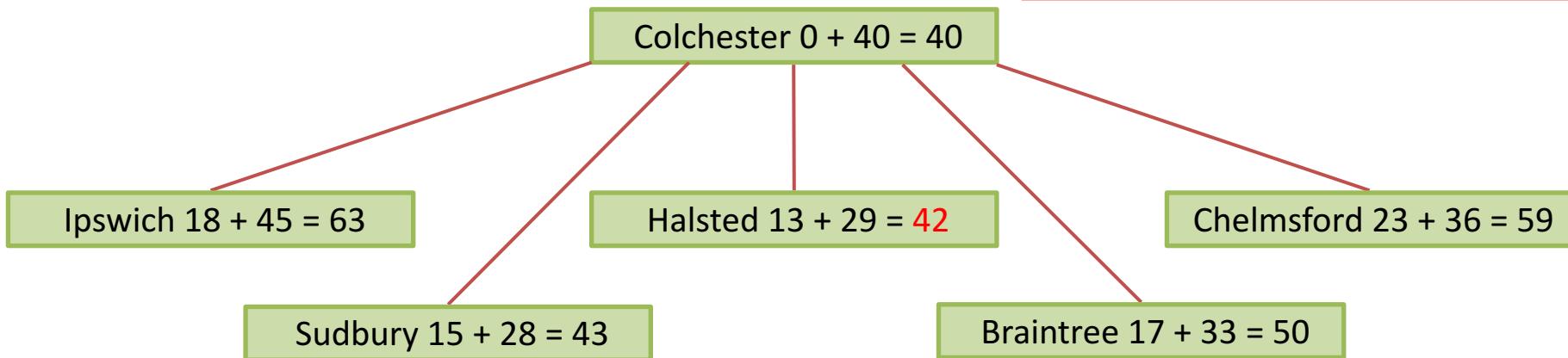
As before, Colchester is the root node:

$$\text{Colchester } 0 + 40 = 40$$

$$g(n)+h(n)$$

The value of  $g(n)$  is from the map;  
The value of  $h(n)$  is from the table.

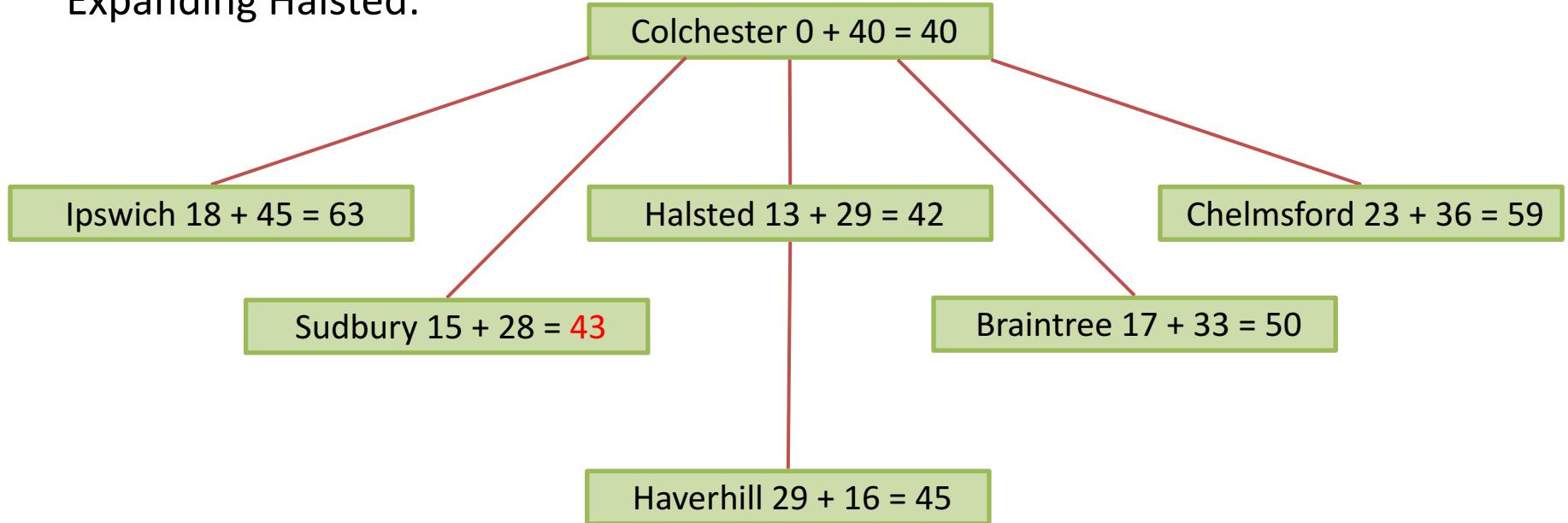
Expanding this gives:



The lowest value of  $g(n) + h(n)$  is at Halsted so this node is expanded next:

## A\* Search (4)

Expanding Halsted:

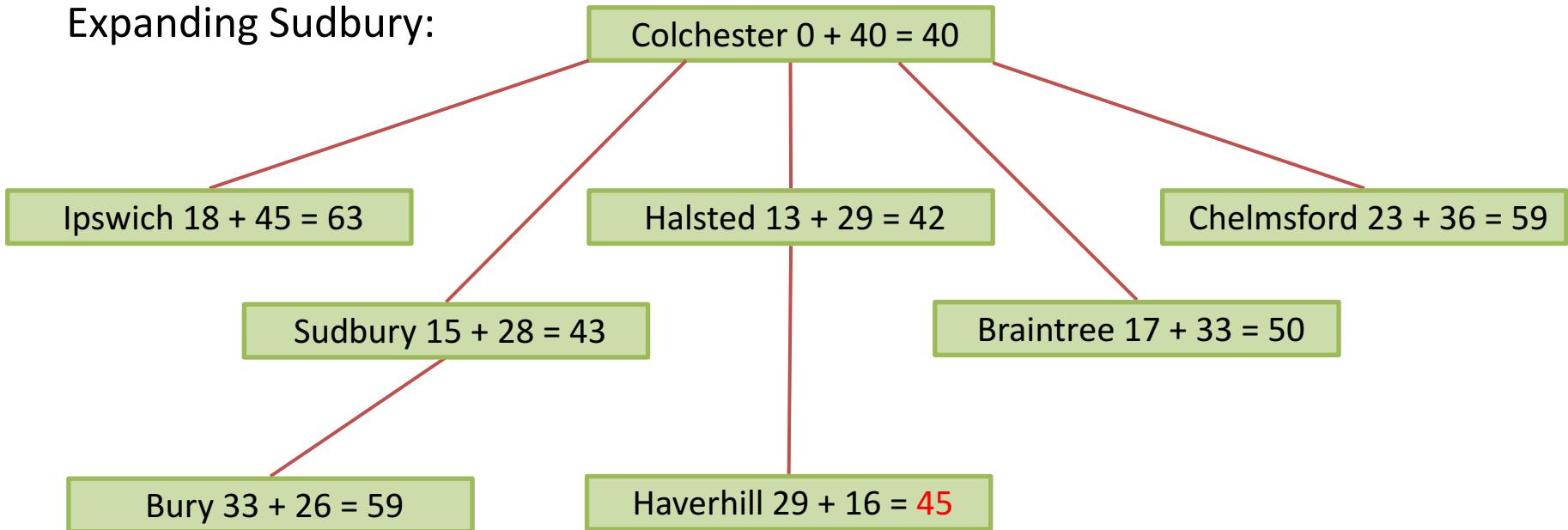


This time we have not included the new nodes for places already reached (e.g. Sudbury and Braintree) only because the new paths to them have higher cost than those already found.

The lowest value of  $g(n) + h(n)$  (among the 5 unexpanded nodes) is now at Sudbury, so this node is expanded next:

# A\* Search (5)

Expanding Sudbury:

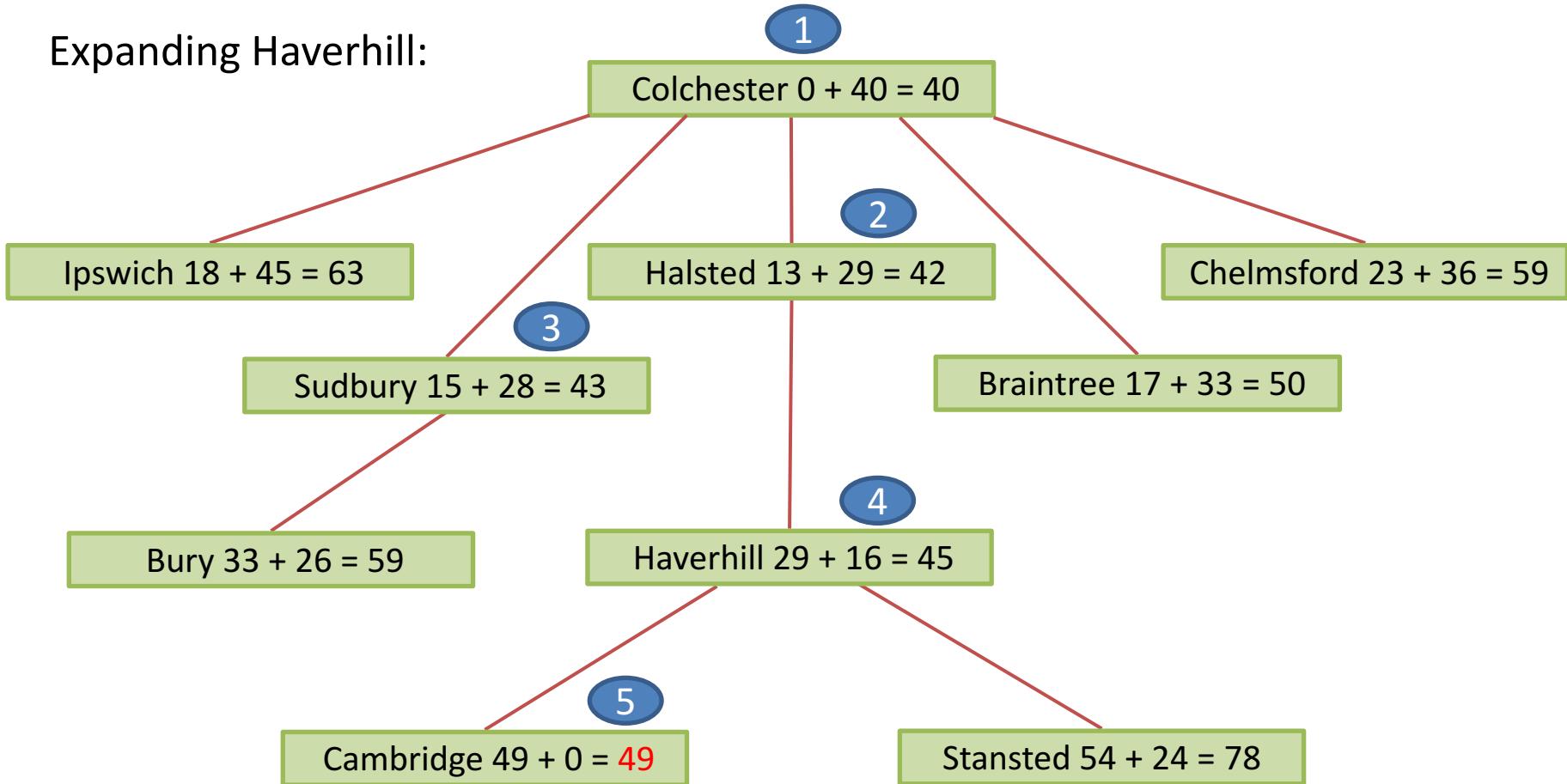


Note that we do not include the new path to Haverhill because it is longer (34) than the one already discovered (29).

Now the lowest value of  $g(n) + h(n)$  is at Haverhill, so this node is expanded next: (5 nodes available for expansion: some are at depth 1 and some at depth 2)

# A\* Search (6)

Expanding Haverhill:



Finally the node for Cambridge itself is selected for expansion (lowest value among 6 unexpanded nodes). Thus, A\* search stops.  
(5 nodes have been selected for expansion.)

## A\* Search (7)

We have found a solution.

**Colchester-Halsted-Haverhill-Cambridge**

Length 49 miles

And this time it is an optimal solution.

**It is important that the search waits until the goal node is selected for expansion before declaring a solution has been found.**

Otherwise the solution may not be optimal. We will give examples to show this in the class.

# Pseudo Code for A\* Search

```
Create an empty list UnexpNodes;  
Add initial node to UnexpNodes;  
WHILE (UnexpNodes not empty)  
    N = The item n in UnexpNodes with g(n)+h(n) being smallest  
    Remove N from UnexpNodes  
    IF N is goal  
        THEN Produce Solution and Return(Success);  
    Expand N to produce list of successors of N;  
    IF cheaper routes to any node in UnexpNodes have been found  
    THEN replace their costlier parent nodes with the cheaper  
    ones;  
    Add successors to tail of UnexpNodes;  
Return(Failure);
```

**h (n)** is a function returning a heuristic estimate of the cost of reaching the goal from node n.

**g (n)** is a function returning the actual cost of reaching node n from the initial state (root).

# Properties of A\* Search

The behaviour of A\* Search depends on the heuristic used. In practice, admissible heuristics are often used.

An **admissible** heuristic is one that **never overestimates** the true cost of reaching the goal (e.g., direct distance is admissible).

It can be proved that, ***provided an admissible heuristic is used,*** A\* search is Complete and Optimal

It is desirable that A\* Search adopts admissible heuristics. However, this may not be guaranteed, and thus A\* search could be non-optimal in some applications.

# Properties of A\* Search (2)

## Time and Space Complexity

The ***time*** taken by A\* search depends on how accurate the heuristic is.

In the best case, with a perfect heuristic, the search goes straight to the solution.

With an uninformative heuristic (e.g.,  $h = 0$  for all nodes) the method reduces to uniform cost search.

With a misleading heuristic it can be worse.

The ***space*** required grows exponentially with solution path length unless the heuristic is very accurate. This is the main limitation on the use of A\* search.

**The main advantage of A\* search lies in its possible low time complexity (depending on accurate heuristic).**

# Some Comparative Results

**Breadth First:** Colchester, Braintree, Haverhill, Cambridge.

→ **59 miles**      11 nodes expanded

**Depth First:** Colchester, Ipswich, Norwich, Cambridge.

123 miles      5 nodes expanded

**Uniform Cost:** Colchester, Halsted, Haverhill, Cambridge.

**49 miles**      10 nodes expanded ←

**Greedy:** Colchester, Sudbury, Haverhill, Cambridge.

54 miles      **4 nodes expanded**

**A\*:** Colchester, Halsted, Haverhill, Cambridge.

**49 miles**      **5 nodes expanded**

N.B. Results for depth first are extremely sensitive to order in which roads are considered. Uniform cost search took much more time to find the same route as A\*.

# Hill Climbing (for self study)

All the search strategies we have examined before have assumed that we could explore several possibilities before choosing what to do.

- One option didn't seem to be leading us to a solution, so we went back and tried an alternative. (e.g., depth first)
- We always considered alternatives before pursuing one of them further. (e.g., breadth first)

This is often unrealistic.

- You often don't know the transitions available until you actually reach a node.
- The number of transitions may be so large that it would be impossible to consider all the transitions from a particular node.

# Re-visiting the Colchester-Cambridge problem

If you didn't have a map (no state transitions defined) or a table of town-town distances (no quantitative heuristic), then you couldn't search for a route in advance.

What could you do?

*Pick the road that seems to be heading in the right direction (we may call it qualitative heuristic).*

*When you get to the next town, repeat.*

(Of course this requires a rough knowledge of the spatial arrangement of the towns)

This is a simple example of what is called ***hill climbing***.

# Hill Climbing (3)

Why is it called hill climbing?

Suppose you want to climb a mountain on a misty day.

How could you select a route?

**REPEAT**

**Select direction that goes most steeply uphill.**

**Advance 5 metres (5 is just an example)**

**UNTIL All directions lead downhill (at a peak).**

Will this always work?

It will always get you to a summit within 5 metres

**But** it may not be the highest point of the mountain.

Such a subsidiary peak is called a ***local maximum***.

A local maximum is higher than its immediate surroundings but may not be the ***global maximum***.

# Hill Climbing (4)

## Hill Climbing

- Finds local maxima.
- May not find the global maximum
- Is a heuristic search technique
- Explores a single route through state space.
- Is the only practical approach in some real problems.

## Hill Climbing and Greedy Search:

Is greedy search an example of hill climbing?

Not quite:

- Greedy search allows back tracking.
- Hill climbing does not.

# Summary

*How to obtain  
reliable and  
accurate heuristics?*

## Greedy Search

Expand node with smallest  $h(n)$

Quick but not optimal

## A\* Search

Expand node with smallest  $g(n)+h(n)$

Optimal (if the heuristic is admissible)

Efficient with a good heuristic

## Hill Climbing (for self study)

Only practical approach in some real problems

May not find the global maximum

# CE213 Artificial Intelligence – Lecture 7

## Knowledge Representation & Expert Systems

What problems cannot be solved by state space search?

How important is domain knowledge in AI?

How to represent knowledge?

How to make use of knowledge in decision making?

# A Brief Review of State Space Search

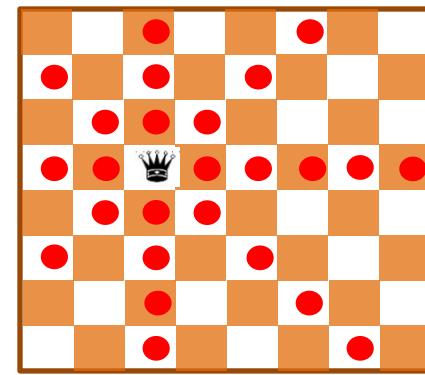
- Basic elements for problem solving by state space search:  
State space, Initial state, Goal state, Operators, Transition functions;  
Search strategies (not necessarily part of state space representation).
- Key issues in search strategies:  
How to select nodes for expansion during search tree construction?  
(or how to evaluate nodes/states?)  
How to evaluate search strategies?  
e.g., uniform cost search vs. A\* search (in terms of 4 criteria)  
Search strategies for game playing: minimax, alpha-beta pruning, MCTS.
- What problems can or cannot be solved by state space search?  
Whether solutions to problems can or cannot be represented as a sequence of operations/actions/moves.  
Whether efficient search strategies are available for very complex problems.

## Examples of problems that cannot be solved or are hard to be solved by State Space Search

➤ How can you solve problems like face recognition, medical diagnosis, financial market forecasting by state space search?

➤ How about the 8-queens problem?

How to place 8 queens on an 8x8 chessboard in such a way that no queen will be attacked by any other queens.



This is a puzzle problem and can be easily represented as a state space, but the size of this state space is huge and there is no good heuristic for evaluating states.

# Is it possible to solve the 8-queens problem by state space search?

- If we look for a satisfactory configuration of 8 queens on a chessboard, but not a sequence of moves:

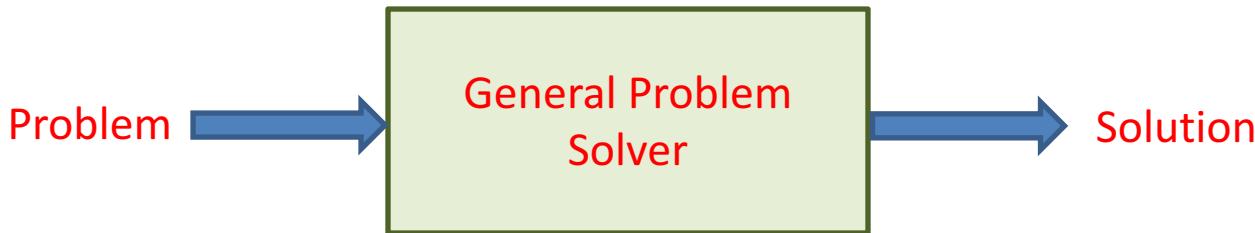
*There would be  $64 \times 63 \times 62 \times 61 \times 60 \times 59 \times 58 \times 57 \approx 2 \times 10^{14}$  possible configurations. Checking whether all these possible configurations satisfy the requirement of the 8-queens problem may take several years by a computer.*

- We could formalise the 8-queens problem for state space search:
  - Approach 1: Use a random 8 queens configuration as initial state, and changing a queen's position as operator.
  - Approach 2: Use an empty chessboard as initial state, and adding a queen to or removing a queen from the chessboard as operator.
- What search strategy would be suitable for this problem?
- Other approaches?

# Pre-1970 AI: Heuristic Search Era

Two AI techniques:

1. Universal Problem Solving Techniques: '**Generate and Evaluate**'  
e.g., GPS (general problem solver) based on search strategies



2. Universal Learning Techniques  
e.g., Perceptrons

Both are characterised by absence of *a priori* knowledge of problem domain (There could be no knowledge available or it is difficult to use knowledge in some problems, e.g., environment understanding).

# Post-1970 AI: Knowledge Based Systems Era

- The central role of ***knowledge*** in all types of intelligent activity was well recognised.
- The research emphasis of AI was shifted to study of **techniques for representing and using knowledge**.
- As a consequence, there was a 10 year eclipse of research on machine learning (This was also due to the book *Perceptrons* by Marvin Minskey, published in 1969).

**knowledge vs. heuristic ?**

# The First Influential Expert System

DENDRAL (<https://en.wikipedia.org/wiki/Dendral>): 1965-1972, Stanford Uni.

It was the first serious attempt to produce a system as good as a human expert in a real world problem domain (*It used a 'generate and evaluate' approach, but was different from state space search – no operators are available for state transition and thus solutions cannot be a sequence of operations*)

**DENDRAL's Task:**

To determine the **structure** of a chemical compound from **mass spectrogram** data. (There could be a very large number of possible structures)

Basic idea of mass spectrogram:

Smash the molecules up and get a histogram of fragment masses.

Chemical formula is known.

Atomic weights are known.

All this **information** can be used to deduce how a molecule is put together.

# The First Influential Expert System (2)

**Why is it a difficult task? – Huge search space**

Because atoms may be joined in many different configurations.

e.g., An organic compound with only 20 atoms could have  $10^4$  isomers.

**Naive solution: Unconstrained “Generate and Evaluate”**

Generate all possible structures (isomers).

For each, check if it is consistent with spectral data (and any other known constraints).

Computationally expensive!

# The First Influential Expert System (3)

**Better solution: Constrained “Generate and Evaluate”**

Use ***knowledge about the problem domain*** to constrain the generation of candidate solutions.

Knowledge might include:

Substructures that molecule is known to contain.

Substructures that molecule is known not to contain.

Which bonds are more likely to break.

Result is ***far fewer candidate solutions will be generated.***

== Benefit of using knowledge for heuristic search

# Revisit the 8-Queens Problem

**Naïve “Generate and Evaluate”:**

Generate every possible configuration of 8 queens on a chessboard.

For each, check if it satisfies requirement.

How many such configurations are there?

$$64 \times 63 \times 62 \times 61 \times 60 \times 59 \times 58 \times 57 \approx 2 \times 10^{14}$$

A program that can check a million configurations per second would take about  $2 \times 10^8$  seconds (5.6 years) to execute.

## 8-Queens Problem: Adding a Constraint

Only one queen is allowed in each row.

How many configurations are now possible?

$$8^8 \approx 2 \times 10^7$$

Thus by applying one constraint, number of configurations to be tested is reduced by a factor of  $10^7$ .

Execution time is reduced to about 20 seconds.

# 8-Queens Problem: Further Constraints

Further constraints are possible:

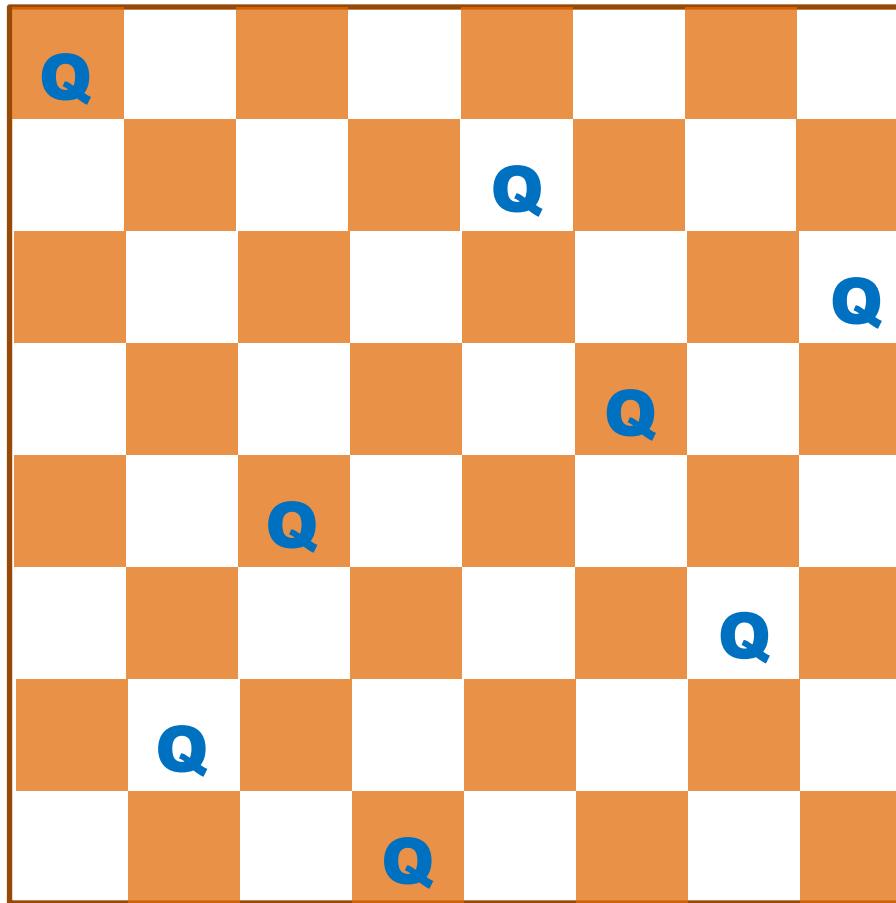
Only one queen is allowed in each column.

Only one queen is allowed in each diagonal.

Together these would constrain the generation of candidate solutions so much that only correct solutions are produced.

The execution time will be dominated by constraint application rather than checking whether candidate solutions satisfy problem requirements.

# 8-Queens Problem: A Solution



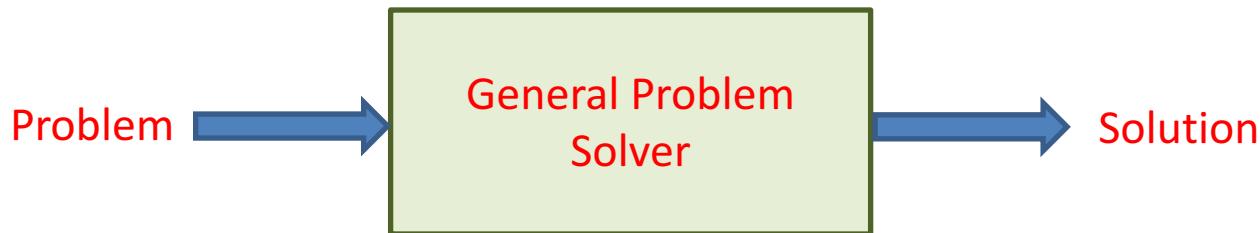
(There are 92 distinct solutions to the 8-queens problem.)

# Lessons from DENDRAL and 8-Queens Problem

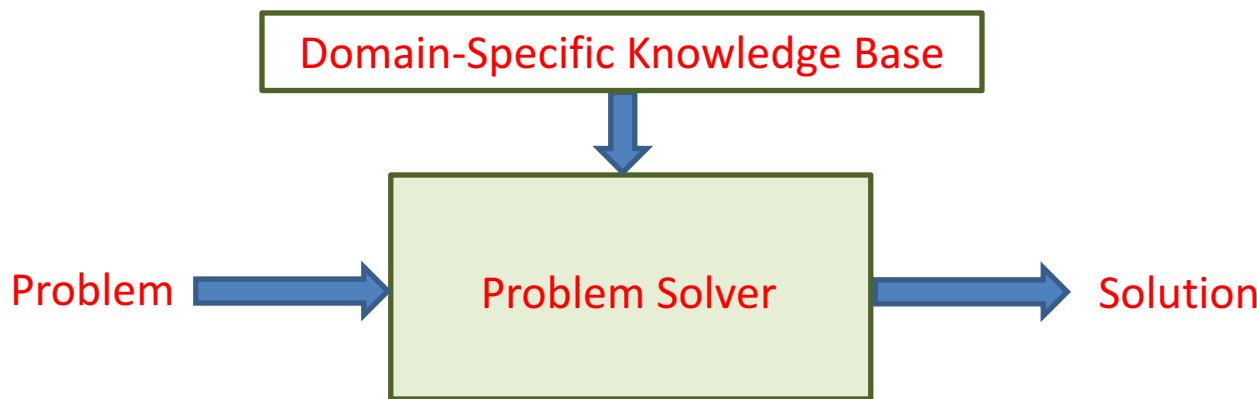
- Searching is easy but time consuming.
- The "intelligence" lies in **constraining the search** so the time taken is reasonable.
- Putting in the constraints is harder than programming the basic search.
- Constraints embody domain-specific knowledge in the problem solving process.

# Another Problem Solving Framework

All of the lessons suggest that we need to replace



with



And this suggests that we need a method which makes it easy to supply and use domain-specific knowledge.

# Production System

## Basic Idea (simple but powerful)

Codify knowledge as a set of condition (situation or state) - action rules:

e.g.,

IF IT IS RAINING THEN PUT UP YOUR UMBRELLA

(condition)

(action)

Such rules are called "*Production Rules*".

A set of production rules constitutes a main part of a "*Production System*" (the other main part is rule interpreters, the topic of next lecture).

In state space search, there are states and operations for state transition.  
In production systems, there are states and actions that can be state transition or others such as making a decision or drawing a conclusion.

## Production Systems (2)

A production rule is essentially an *ordered pair*.

The **condition component** is often called "antecedent", "left hand side" or simply "LHS".

The **action component** is often called "consequent", "right hand side" or simply "RHS".

Note the similarity to a logical implication:

antecedent  $\Rightarrow$  consequent  
(IF)                                  (THEN)

# Advantages of Production Systems

## Procedural Representation (easy to program)

The knowledge is encoded in a form that indicates *how it is to be used.*

Compare the production rule

IF it is raining THEN put up your umbrella  
with

An umbrella is a device to keep the rain off.

Both represent the same knowledge, but the production rule makes it easy to program.

The latter is an example of a ***declarative representation.***

# Advantages of Production Systems (2)

## Modularity

Each chunk of knowledge is encoded as a separate rule.

Thus it is comparatively **easy to add or remove** individual pieces of knowledge.

This is comparable to adding or removing individual statements from a program in Java, Python or C++.

Frequently the order of the rules has no significance.

Again, this is comparable to conventional languages.

# **Advantages of Production Systems (3)**

## **Explanation Facility**

For many production systems, it is easy to add a facility enabling the system to explain its reasoning to the user.

Similar to getting a statistical package to explain how it had calculated a result.

## **Similarity to Human Cognition - Naturalness**

It has been claimed that the way a production system operates has a closer resemblance to human cognition than the behaviour of other computer languages does.

Production systems also have some disadvantages - we will talk about these later (e.g., incompleteness, curse of dimensionality).

# Summary

**State space search can not solve all the problems**

**Knowledge is useful and necessary for problem solving**

DENDRAL as an example, the first influential expert system.

**Production system for knowledge representation**

Condition (situation or state) – action rules

Procedural representation

Knowledge represented in a form that indicates how it can be used

Modularity, Explanation facility, Similarity to human cognition

*Production system vs. state space representation?*

*How to use production rules to solve problems? – Expert systems*



# CE213 Artificial Intelligence – Lecture 8

## Production Rule Interpreters

How to search knowledge base for problem solving?

- How many rules are relevant to given situations or hypothesis?
- How to apply (combine) these relevant rules to make decisions?

Interpreters: Forward chaining vs. Backward chaining

# Core Components of a Production System

## 1. A set of production rules

Sometimes called the *rule base* (or even *knowledge base*)

## 2. A *production rule interpreter*

A program which selects and executes rules whose condition parts match the current situation.

## 3. An environment (may be embedded in rule set)

Condition parts of the rules must be propositions about some objects or entities.

Action parts of the rules must specify operations to be performed on some objects or entities.

*An environment consists of a set of objects or entities* that the system responds to and manipulates.

# Production Rule Interpreter

## What is a production rule interpreter?

An interpreter is a program that can read/select and apply the rules in a production system.

Such an interpreter will *systematically* access the rules in an attempt to establish what conclusions may be drawn.

There are several ways that such an interpreter could be constructed.

The two commonest types are called *forward chaining interpreter* and *backward chaining interpreter*.

# Information about Environment States

An environment consists of a set of objects or entities.

In all but the simplest systems, the states of an environment provide two types of information:

## *Dynamic*

The information that determines which rules can fire and is modified by the rules.

In many systems dynamic information is called ***working memory, which stores confirmed conditions and conclusions.***

## *Static*

Other information used by the rules in matching conditions and determining actions.

# Forward Chaining Rule Interpreter

## The Basic Idea and Steps:

For a simple forward chaining production system, the interpreter goes through ***match-execute cycles***:

### ***Match Step***

Scans through rules to find one whose **condition part** is matched by the current state (situation) of the environment (working memory).

### ***Execute Step***

Performs the operation specified by the **action part** of the rule that has been found to match.

This is sometimes called ***firing*** the rule.

Typically this will change the state of the environment.

Normally this match-execute cycle continues until either no rule matches or the system is explicitly stopped by the action part of a rule.

## A Toy Example: Mechanic

To demonstrate the operation of forward chaining we will use a toy expert system called Mechanic.

Mechanic's task is to determine why a car doesn't start and suggest a remedy.

To keep things really simple we will:

Consider just the first 5 rules of Mechanic (21 rules in total).  
And always choose the first rule we find that matches the current situation (simplest conflict resolution).

# Mechanic – The First Five Rules

Rule 1:

IF Lights work AND Starter will turn  
THEN CONCLUDE Battery OK

Can you identify the environment from these 5 rules?

Rule 2:

IF Lights do not work AND Starter will not turn  
THEN CONCLUDE Battery flat

Rule 3:

IF Battery flat  
THEN CONCLUDE Remedy is recharge battery

The remaining 16 rules are in the last 4 slides for this lecture.

Rule 4:

IF Lights work AND Starter will not turn  
THEN CONCLUDE Loose connection in starter circuit

Rule 5:

IF Loose connection in starter circuit  
THEN CONCLUDE Remedy is locate and repair loose connection

# Mechanic: Forward Chaining

The system begins the **first match-execute cycle** by considering Rule 1.

Rule 1:

IF *Lights work* AND *Starter will turn*  
THEN CONCLUDE Battery OK

To evaluate the condition part it must ask the user whether the lights work and whether the starter will turn.

Assume, as a result, it adds the following facts to its working memory:

*Lights work*

*Starter will not turn*

Rule 1 does not match, so the system considers Rule 2 next.

# Mechanic: Forward Chaining (2)

Rule 2:

IF Lights do not work AND Starter will not turn  
THEN CONCLUDE Battery flat

Because Lights work, Rule 2 fails to match. The system now considers Rule 3.

Rule 3:

IF Battery flat  
THEN CONCLUDE Remedy is recharge battery

There is no fact in working memory stating that the battery is flat, and this is not something that can be asked, so Rule 3 also fails.

## Mechanic: Forward Chaining (3)

So, the system next considers Rule 4

Rule 4:

IF Lights work AND Starter will not turn

THEN CONCLUDE Loose connection in starter circuit

The facts in working memory match the condition part of Rule 4, so this rule is selected to execute or fire.

The additional fact from the conclusion drawn by Rule 4

*Loose connection in starter circuit*

is added to working memory

This concludes the first match-execute cycle. Please note that in each match-execute cycle only one matched rule fires.

## Mechanic: Forward Chaining (4)

The system now begins the **second match-execute cycle**.

Rules 1 to 4 proceed as before (except that it is not necessary to ask the user what has already been asked):

Rules 1 to 3 will fail again.

Rule 4 will match again and would be able to fire.

It looks as though we are stuck in an infinite loop!

To avoid this, many forward chaining interpreters do not allow the same rule to fire twice as a result of matching the same facts.

This property is called ***refractoriness***.

We will assume that our forward chaining rule interpreter has this property, so Rule 4 will not be selected for firing again.

# Mechanic: Forward Chaining (5)

So, the system next considers Rule 5

Rule 5:

IF Loose connection in starter circuit

THEN CONCLUDE Remedy is locate and repair loose connection

The condition part of Rule 5 matches the fact in working memory, so this rule is selected to fire and the conclusion

*Remedy is locate and repair loose connection*

is added to working memory.

This completes the second match-execute cycle.

During the **third match-execute cycle**, the only rules that match the facts in working memory will be refractory.

Consequently, no rules will be selected for firing. The system will stop.

# Conflict Resolution

The need for conflict resolution arises whenever two or more rules match the current environment state at the same time.

Generally it is not possible for more than one action to be carried out, because they may be mutually contradictory.

e.g., a robot control system with two matched rules that suggest “Turn left” and “Turn right” at the same time.

Procedures used to select one rule from several candidates for firing are called ***conflict resolution strategies***.

# Conflict Resolution Strategies

## *First match found*

Widely used in early production systems. Rarely used now.

Convenient for human interpreter.

System's behaviour depends on the order in which rules are written down.

## *Random choice*

Major disadvantage - makes behaviour non-deterministic.

For some simulations this could be an advantage, but generally it is inconvenient.

# Conflict Resolution Strategies (2)

## *Specificity*

Chooses the rule which has the most restrictive condition part ( ... AND ... ).  
Makes it easy to handle exceptional cases.  
Widely used.

## *Recency*

Chooses the rule whose condition part has been satisfied by information most recently added to working memory.  
Tends to keep the system focused on same part of the problem until it has been resolved.  
Widely used.

## *Assigned Priorities*

In many systems, the programmer can assign a numerical priority to rules.  
In cases of conflict, the one with the highest priority is chosen.

# Features/Properties of Forward Chaining

- **Data Driven**

The system has no explicit goals. What it does is determined only by the current situation.

- **Rule Selection**

It selects rules with conditions matching the current situation.

Conflict resolution is used to choose which rule to fire if more than one matches.

- **Iterative**

Repeated execution of the match-execute cycle is used to choose and fire rules.

# Backward Chaining Rule Interpreter

## The Basic Idea and Steps:

1. Start with a conclusion hypothesis
2. Find **all** the rules whose RHSs draw conclusions about the hypothesis.
3. Determine whether the LHSs of those rules **match** the current situation.  
If it is not sure whether a rule matches or not, then make a subsidiary hypothesis about its LHS and go back to step 2 to determine whether the LHS is true. Otherwise go to step 4.
4. **Execute** the corresponding RHSs with matched LHSs, thus confirming or rejecting the hypothesis.

Step 3 may involve setting up subsidiary hypothesis to determine whether a LHS is true. This is the difficult part in backward chaining!

Backward chaining is **recursive**, whilst forward chaining is **iterative**.

# Mechanic: Backward Chaining

Consider again the Mechanic expert system made of just the first 5 rules.

The car won't start – our goal is to find out the reason and a remedy.

Therefore, the initial hypothesis is *Remedy is X*, where *X* could be anything.  
(Step 1)

Inspecting the RHSs of all 5 rules we find two rules that match the hypothesis:

Rule 3:

IF Battery flat

THEN CONCLUDE *Remedy is recharge battery*

Rule 5:

IF Loose connection in starter circuit

THEN CONCLUDE *Remedy is locate and repair loose connection*

(Step 2)

## Mechanic: Backward Chaining (2)

First, the system considers Rule 3. (Starting Step 3 ...)

Its LHS is *Battery Flat*.

But, we don't know if the battery is flat at the moment, so we are not sure if the LHS of this rule is satisfied.

We therefore set up a subsidiary hypothesis *Battery Flat*

Then we look for rules whose RHSs draw conclusions about the flatness of the battery.

*(Note we do not abandon this rule yet, as what would be done by forward chaining.)*

## Mechanic: Backward Chaining (3)

(Going back to Step 2 ...)

Among all 5 rules only Rule 2 has the RHS matching the subsidiary hypothesis.

Rule 2:

IF Lights do not work AND Starter will not turn  
THEN CONCLUDE **Battery flat**

(Step 3 again ...)

The LHS of Rule 2 is *Lights do not work* AND *Starter will not turn*.

The system itself doesn't know either of these facts.

These are something the system must ask the user, similar to what is done in forward chaining.

In general, the system will know which facts can be deduced using other rules and which are data the user must supply.

# Mechanic: Backward Chaining (4)

(Still in Step 3 ...)

Let us suppose that the user tells the system (similar as in forward chaining):

Lights work

Starter will not turn. (add them to working memory)

The LHS of Rule 2 is *Lights do not work* AND *Starter will not turn*.

So, the system is unable to conclude *Battery flat*.

The system has now exhausted all the rules it found when it set out to establish the subsidiary hypothesis *Battery flat*. It was doing this in an attempt to determine whether Rule 3 could fire, but the outcome is negative.

Rule 3:

IF Battery flat

THEN CONCLUDE Remedy is recharge battery

# Mechanic: Backward Chaining (5)

(Still is Step 3 ...)

The system therefore abandons Rule 3 and turns to Rule 5, the other rule it found when it set up the original hypothesis **Remedy is X**.

Rule 5:

IF Loose connection in starter circuit

THEN CONCLUDE **Remedy is locate and repair loose connection**

The LHS of Rule 5 is **Loose connection in starter circuit**, which we are not sure based on facts in working memory.

We therefore set up a subsidiary hypothesis **Loose connection in starter circuit**.

(Going back to Step 2 again ...)

The system finds one rule only, which draws the conclusion about this subsidiary hypothesis. That is,

Rule 4:

IF Lights work AND Starter will not turn

THEN CONCLUDE **Loose connection in starter circuit**

# Mechanic: Backward Chaining (6)

(Step 3 again ...)

The LHS of Rule 4 is matched by facts already established (Lights work AND Starter will not turn) , so the rule fires and draws the conclusion (add it to working memory):

Loose connection in starter circuit

(Step 4 ...)

This deduced fact matches the LHS of Rule 5, so it can fire and draw its conclusion

Remedy is locate and repair loose connection

This matches the original hypothesis (add it to working memory).

There are no more untried rules to consider (exhaustive backward chaining).

So, the consultation is complete. The rules that fire are in the following order:  
Rule 4, Rule 5.

# Features/Properties of Backward Chaining

- **Hypothesis Driven (good for diagnosis and classification problems)**  
At each stage the system has a goal of trying to establish the truth of some conclusion. May need ***less working memory*** than forward chaining.
- **Rule Selection**  
It selects rules with RHS matching the hypothesis, which will help determine if the hypothesis is true. All matched rules are considered.  
***Conflict resolution is not necessary.***
- **Recursive**  
While trying to establish the truth of a **hypothesis**, the system may find it needs to know the truth value of some other proposition.  
To do this, it will set up this other proposition as a **subsidiary hypothesis** and try to prove that it is true.

# **Summary**

## **Core Components of a Production System**

A set of production rules, Interpreter, Environment (working memory)

## **Forward Chaining Rule Interpreter**

Basic idea and steps, Conflict resolution, Refractoriness, Key features

## **Backward Chaining Rule Interpreter**

Basic idea and steps, Key features

## **Mechanic Expert System as an Example**

**Forward or backward, that is the question.**

# MECHANIC - A SIMPLE EXPERT SYSTEM

A simple set of informal rules that attempts to determine why a car will not start.

Rule 1:

IF Lights work AND Starter will turn  
THEN CONCLUDE Battery OK

Rule 2:

IF Lights do not work AND Starter will not turn  
THEN CONCLUDE Battery flat

Rule 3:

IF Battery flat  
THEN CONCLUDE Remedy is recharge battery

Rule 4:

IF Lights work AND Starter will not turn  
THEN CONCLUDE Loose connection in starter circuit

Rule 5:

IF Loose connection in starter circuit  
THEN CONCLUDE Remedy is locate and repair loose connection

Rule 6:

IF Starter will turn AND Engine will not fire  
THEN CONCLUDE Suspect ignition system fault  
AND CONCLUDE Suspect fuel supply fault

Rule 7:

IF Suspect ignition system fault  
AND Each spark plug fires correctly  
THEN CONCLUDE Ignition system OK

Rule 8:

IF Suspect ignition system fault  
AND Only one plug fails to fire  
THEN CONCLUDE Plug fault

Rule 9:

IF Plug fault  
THEN CONCLUDE Remedy is check HT cable from distributor and plug itself, replacing as necessary

Rule 10:

IF Suspect ignition system fault  
AND All plugs fail to fire  
THEN CONCLUDE Distributor fault

Rule 11:

IF Distributor fault  
THEN CONCLUDE Remedy is check LT cable to distributor and distributor itself, replacing as necessary

Rule 12:

IF Suspect fuel supply fault  
AND Fuel reaches cylinders  
THEN CONCLUDE Fuel supply OK

Rule 13:

IF Suspect fuel supply fault  
AND Fuel reaches carburettor  
THEN CONCLUDE Carburettor fault

Rule 14:

IF Carburettor fault

THEN CONCLUDE Remedy is repair or replace carburettor

Rule 15:

IF Suspect fuel supply fault

AND Fuel does not reach carburettor

THEN CONCLUDE Suspect empty petrol tank

AND CONCLUDE Suspect fuel pump failure

AND CONCLUDE Suspect fuel blockage

Rule 16:

IF Suspect empty petrol tank

AND Petrol gauge indicates no fuel

THEN CONCLUDE Remedy is fill tank with petrol

Rule 17:

If Suspect empty petrol tank

AND Petrol gauge indicates fuel in tank

THEN CONCLUDE Petrol in tank

Rule 18:

IF Suspect fuel pump AND Fuel pump operates  
THEN CONCLUDE Fuel pump OK

Rule 19:

IF Suspect fuel pump  
AND Fuel pump does not operate  
THEN CONCLUDE Remedy is repair or replace fuel pump

Rule 20:

IF Suspect fuel blockage  
AND Fuel pump OK  
AND Petrol in tank  
THEN CONCLUDE Remedy is clear blockage in fuel supply

Rule 21:

IF Battery OK  
AND Ignition system OK  
AND Fuel supply OK  
THEN CONCLUDE Remedy is call in a car expert!

# CE213 Artificial Intelligence – Lecture 9

## MYCIN and Reasoning with Uncertainty

(<https://en.wikipedia.org/wiki/Mycin>)

MYCIN is an expert system for identifying bacterial infection and appropriate antibiotic treatment, developed at Stanford University.

How many rules are there?

How to organise them?

How to search and interpret them?

***If rule matching is not certain, how to handle uncertainty?***

# MYCIN

A classic backward chaining (mainly) expert system developed in early 1970s by a research group led by *Edward Feigenbaum* at Stanford University.

The first large expert system to perform at the level of a human expert.

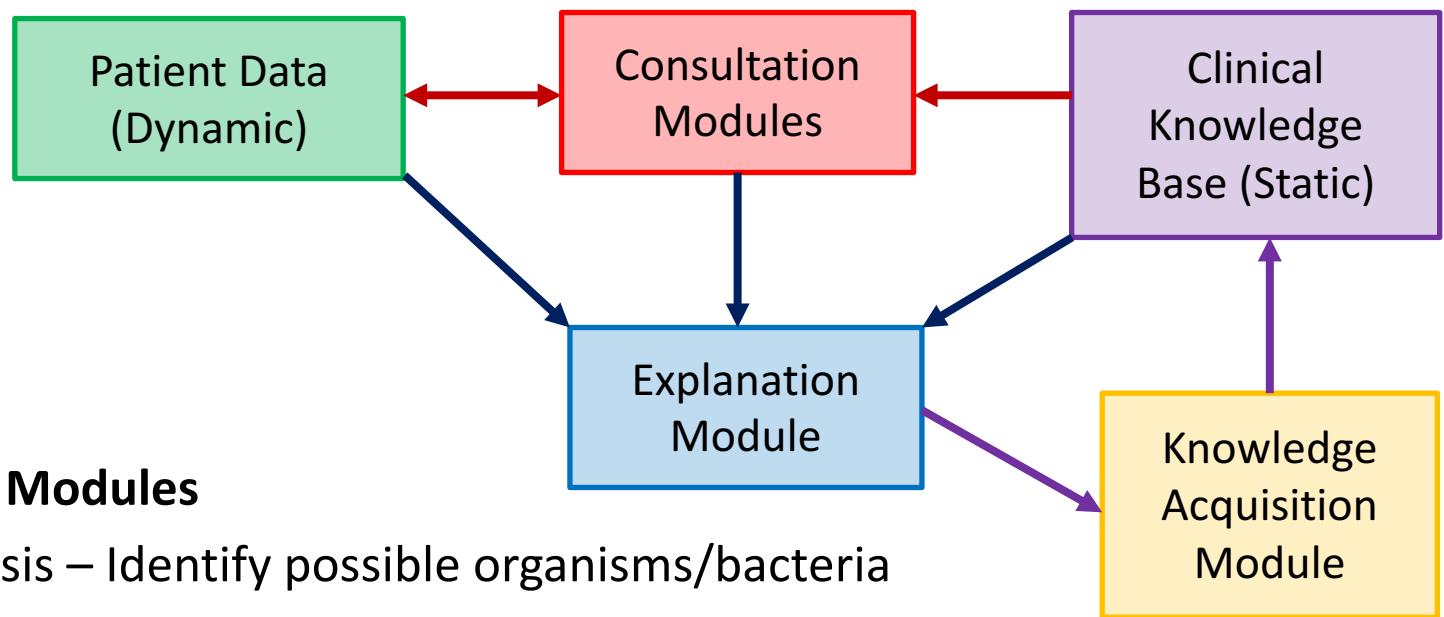
A widely adopted architecture for other expert systems.

## MYCIN's Task

Determine the most appropriate antibiotic treatment (therapy) for a patient suffering from a bacterial infection (diagnosis).

Although not a purely **diagnostic task**, identifying the bacteria responsible for the infection is the major part of MYCIN's task.

# MYCIN's Basic Architecture



## Consultation Modules

1. Diagnosis – Identify possible organisms/bacteria
2. Therapy – Select appropriate treatment

## Explanation Module

Provides an explanation for both the conclusions reached and the questions asked during a consultation.

## Knowledge Acquisition Module

Is used to build and modify rule base.

# MYCIN's Knowledge/Facts Representation

MYCIN represents most of the entities it reasons about as triplets:

$(Object, Attribute, Value)$

Each triplet, or an OAV, comprises an object, one of its attributes, and the current value of that attribute,

e.g., The fact that John is 50 could be represented by a triplet

$(John, Age, 50)$

This type of representation of facts has proved remarkably effective for most programming languages to handle.

# Attributes Used in MYCIN

Original version of MYCIN used **65 attributes**, which were divided into **6 groups** in terms of the type of objects they could be applied to:

<b><i>Attribute Groups</i></b>	<b><i>Example Attributes</i></b>
Cultures	Site where collected
Drugs	Duration of administration
Surgical Procedures	Body cavity opened
Organisms	Morphology (form and structure)
Patients	Age
Therapies	Dosage

## Attributes Used in MYCIN (2)

As part of its **static knowledge**, MYCIN stores various facts about each attribute, including:

- Range of possible values.

- How to ask the user for its value.

- Whether the value can be obtained from a lab test.

- Which rules refer to the attribute in their condition part.

- Which rules may draw conclusions about the attribute.

As knowledge representation, most production rules in MYCIN establish relationships among objects, attributes, and values.

Most of MYCIN's reasoning through rule interpretation attempts to **obtain or deduce the values of unknown attributes**.

# Production Rules in MYCIN

**Condition part:** in the form of ‘attribute of object is value’

Some relational predicates (e.g., =, >, etc.) can be used in condition expressions.

Compound conditions (e.g., conjunctions and disjunctions) can be formed using AND and OR.

**Consequent/action Part:** also in the form of ‘attribute of object is value’

It typically draws conclusions about the **value** of the **attribute** of an **object**. These conclusions usually take the form of evidence for or against a particular fact being true, with some **certainty**.

# Production Rules in MYCIN (2)

**There are about 600 production rules in MYCIN.**

**An Example Rule:**

	(attribute)	(object)	(value)
IF		the stain of the current organism is gramneg	
AND		the morphology of the current organism is rod	
AND		the current patient is a compromised host	
THEN CONCLUDE		there is evidence with certainty 0.6 that the identity of the current organism is pseudomonas (a bacteria).	

**What are the objects, attributes, and values in this rule?**

# MYCIN's Rule Interpretation

The diagnosis module of MYCIN uses ***exhaustive backward chaining***.

By 'exhaustive' we mean that ***evidence from all matched rules***, for and against every possible hypothesis, will be accumulated before drawing any conclusions.

(no need of conflict resolution in exhaustive backward chaining)

The therapy module of MYCIN uses ***forward chaining***.

There may be many rules whose RHSs match a hypothesis about an antibiotic treatment. However, there may be one rule only whose LHS matches a bacteria infection. Therefore, forward chaining could be more efficient for the therapy module.

# Forward Chaining or Backward Chaining?

	Forward Chaining	Backward Chaining
Procedure	iterative	recursive
Need of conflict resolution	yes	no
Suitable problems or applications	where it is hard to make hypothesis or there may be too many possible hypotheses; or where there may be too many rules that match a specific hypothesis. e.g., prediction of the value of a continuous variable	where there are obvious hypotheses to make and the number of possible hypotheses is limited; or where there may be too many rules that match a specific situation but with different actions/conclusions. e.g., medical diagnosis, fault detection

# Uncertainty – A Key Issue in MYCIN's Rule Interpretation

In the medical domain, in which MYCIN operates, much of the information is uncertain.

- Facts may be believed with some degree of confidence rather than known for certain.
- Rules may lend support in favour of a hypothesis rather than confirm it definitely.

Ideally, manipulation of such uncertainties would be based on some rigorous foundations, such as **Bayesian methods** derived from probability theory.

In practice, it is not feasible to use such methods because:

- Information from medical experts is usually in vague terms like *fairly strong evidence*.
- Even if the experts could be persuaded to provide precise estimates of **conditional probabilities**, an enormous number of these would be needed to build a system using Bayesian methods.

# MYCIN's Uncertainty Representation

The creators of MYCIN (researchers at Stanford University) developed their own system for **representing and manipulating uncertainty**.

It is hard to justify, but its chief merits are that it is easy to use and it works well. (Compared to Bayesian methods, it is simple and effective.)

## Representing uncertain facts:

Degrees of belief in a fact or conclusion are represented by ***Certainty factors (CF)***, ranging from 0 (certainly false) to 1 (certainly true).

[N.B. In some applications, the range of certainty factors could be from  $-1$  (certainly false) through 0 (no belief either way) to  $+1$  (certainly true).]

# MYCIN's Uncertainty Representation (2)

## Drawing a conclusion through one rule with uncertainty:

To determine the certainty of a conclusion to be drawn from a rule, the certainty of the condition is multiplied by the certainty stated in the consequent part of the rule:

$$CF_{\text{conclusion}} = CF_{\text{condition}} \times CF_{\text{consequent}}$$

- The certainty of a *simple condition* is just the one stated in the condition part.
- The certainty of the ***conjunction*** (AND) of two or more conditions is the ***lowest*** certainty of the individual conditions:  $CF_{\text{condition}} = \min(CF_{\text{condition1}}, CF_{\text{condition2}})$
- The certainty of the ***disjunction*** (OR) of two or more conditions is the ***highest*** certainty of the individual conditions:  $CF_{\text{condition}} = \max(CF_{\text{condition1}}, CF_{\text{condition2}})$

# An Example of Rule Interpretation with Uncertainty

Consider the rule given earlier:

IF                   the stain of the current organism is gramneg  
AND                 the morphology of the current organism is rod  
AND                 the current patient is a compromised host  
THEN CONCLUDE     there is evidence with certainty 0.6  
                       that the identity of the current organism is  
                       pseudomonas (a type of bacteria).

Suppose the following facts had been established:

Stain of the current organism is gramneg with certainty 0.4  
Morphology of the current organism is rod with certainty 0.7  
Current patient is a compromised host with certainty 0.9

Then MYCIN will conclude the organism is pseudomonas with certainty

$$0.4 \times 0.6 = 0.24 \quad (\text{CF}_{\text{condition}} = 0.4, \text{CF}_{\text{consequent}} = 0.6)$$

In other words there is fairly weak evidence to support the hypothesis that the organism is pseudomonas.

# Combining Conclusions from Different Rules

If two or more rules draw conclusions about the same fact, then in exhaustive backward chaining they must be combined.

MYCIN uses the following formula to combine the certainties of conclusions drawn from two rules:

$$CF_{\text{combined}} = CF_{r1} + (1 - CF_{r1}) \times CF_{r2} = CF_{r2} + (1 - CF_{r2}) \times CF_{r1}$$

where  $0 \leq CF_{r1} \leq 1$  and  $0 \leq CF_{r2} \leq 1$ .

Note that the result of combination is greater than each of the individually concluded certainties but less than their sum, i.e.,

$$CF_{r1} \text{ or } CF_{r2} \leq CF_{\text{combined}} \leq CF_{r1} + CF_{r2}.$$

Also note also that no certainty produced by combining certainties, which are less than or equal to 1, could exceed 1, i.e.,  $0 \leq CF_{\text{combined}} \leq 1$ .

## Combining Conclusions from Different Rules (2)

If  $CF_{r1}$  and  $CF_{r2}$  are negative or they have different signs, different formula should be used.

If  $-1 \leq CF_{r1} \leq 0$  and  $-1 \leq CF_{r2} \leq 0$ ,

$$CF_{\text{combined}} = CF_{r1} + (1 + CF_{r1}) \times CF_{r2}$$

If  $CF_{r1}$  and  $CF_{r2}$  have different signs,

$$CF_{\text{combined}} = (CF_{r1} + CF_{r2}) / [1 - \min(|CF_{r1}|, |CF_{r2}|)]$$

In this module, we mainly consider situations where  $0 \leq CF_{r1} \leq 1$  and  $0 \leq CF_{r2} \leq 1$ .

**(This slide is for information only. It is optional for this module.)**

# Another Example of Rule Interpretation with Uncertainty

Suppose we have two rules:

- R1: If A AND B  
Then Conclude X with certainty 0.6
- R2: If C  
Then Conclude X with certainty 0.5

Suppose the system has already discovered

- A is satisfied with certainty 0.8
- B is satisfied with certainty 0.7
- C is satisfied with certainty 0.6

In attempting to establish X, the system will try both rules.

Note: A, B, C are conditions and X represents a conclusion. They are all in the form of '**attribute of object is value**'.

## Another Example of Rule Interpretation with Uncertainty (2)

Starting with R1:

Certainty of B is less than that of A.

Using the rule for conjunctive conditions, A AND B has a certainty of 0.7.

Hence X is concluded with certainty  $0.7 \times 0.6 = 0.42$

Then using R2:

Certainty of C is 0.6.

Hence X is concluded with certainty  $0.6 \times 0.5 = 0.3$

Thus we have two different values for the certainty of X and these must be combined.

## Another Example of Rule Interpretation with Uncertainty (3)

The formula is

$$CF_{\text{combined}} = CF_{r1} + (1 - CF_{r1}) \times CF_{r2}$$

Using the conclusion of R1 gives us  $CF_{r1} = 0.42$

The conclusion of R2 gives us  $CF_{r2} = 0.3$

So the certainty of X derived from both rules is

$$CF_{\text{combined}} = 0.42 + (1 - 0.42) \times 0.3 = 0.42 + 0.58 \times 0.3 = 0.42 + 0.174 = \mathbf{0.594}$$

Thus as a result of using both rules X now has a certainty of 0.594.

Exactly the same result would have been obtained if we use

$$CF_{\text{combined}} = CF_{r2} + (1 - CF_{r2}) \times CF_{r1} = \mathbf{0.594}$$

*In the next class, there will be an exercise of reasoning with uncertainty using exhaustive backward chaining, with a whole set of production rules.*

# Explanation Generation (mainly for self study)

MYCIN is able to answer two types of question:

*WHY*: Why the system asked the user a particular question.

*HOW*: How the system reached a particular conclusion.

As the consultation proceeds, MYCIN maintains a record of all the rule activations, indicating which rules were fired in an attempt to satisfy the conditions of which other rules (rule chaining).

This record will form a **tree of hypotheses and subsidiary hypotheses**.

- To answer a *WHY* question, the system must **look up** the tree from the current position to identify the hypotheses that led to the current question.
- To answer a *HOW* question, the system must **look down** the tree from the current position to identify the evidence that led to the current conclusion.

**(only possible with backward chaining)**

**(This slide is for information only. It is optional for this module.)**

# Explanation Generation (mainly for self study) (2)

Such explanation facilities are easy to implement and produce two benefits:

- They increase the user's confidence in the correctness of the conclusions.
- They are invaluable in debugging a system that draws erroneous conclusions.

**(This slide is for information only. It is optional for this module.)**

# Summary

**MYCIN's basic architecture**

**Knowledge/fact representation in MYCIN – OAV triplets**

**MYCIN's production rule format**

**MYCIN's production rule interpreters**

**MYCIN's system for representing and manipulating uncertainty**

**Reasoning with uncertainty with the MYCIN's system**

**Explanation generation in MYCIN**

# CE213 Artificial Intelligence – Lecture 10

## Introduction to Machine Learning

Problems in the AI approaches learnt so far:

- Inefficient search strategies
- Ineffective evaluation functions
- Ineffective representation of knowledge
- Lack of knowledge

Machine learning - A new approach to AI  
data-driven vs. knowledge-driven  
knowledge representation learning  
new methods for problem solving

# Why Machine Learning?

Dog or Cat?



Will the stock market go up or down?



Kaggle competition: Create a program to distinguish dogs from cats.

State of the art: a little kid may outperform the best computer (98.9%).

Can you use 'generate and evaluate' method for solving these problems?

Can you build expert systems for solving these problems?

**Key issues:** complex patterns, huge search space,  
inefficient search methods,  
ineffective knowledge representation,  
ineffective evaluation functions.

# What Is Machine Learning?

Machine learning is a branch of artificial intelligence focusing on **how to get computers to learn from data/experience**.

Very roughly:

**Programming** is telling a machine *what* to do and *how* to do it.

**Machine learning** is showing a machine *what* we want it to do and expecting it to figure out *how* to do it.

There could be more complicated situations:

Sometimes we cannot even show the machine what we want because we do not really know. A smart machine can even help in this situation through machine learning.

A short video introducing machine learning using layman's language:

[https://www.youtube.com/watch?v=f\\_uwKZIAeM0](https://www.youtube.com/watch?v=f_uwKZIAeM0)

# A More Specific Definition and Key Elements of Machine Learning

Machine learning can be defined as updating/optimising a model by a learning algorithm based on sample data and certain performance measure, so as to improve the performance of the model for a given task, so there must be

A ***task or problem*** and an associated ***performance measure***

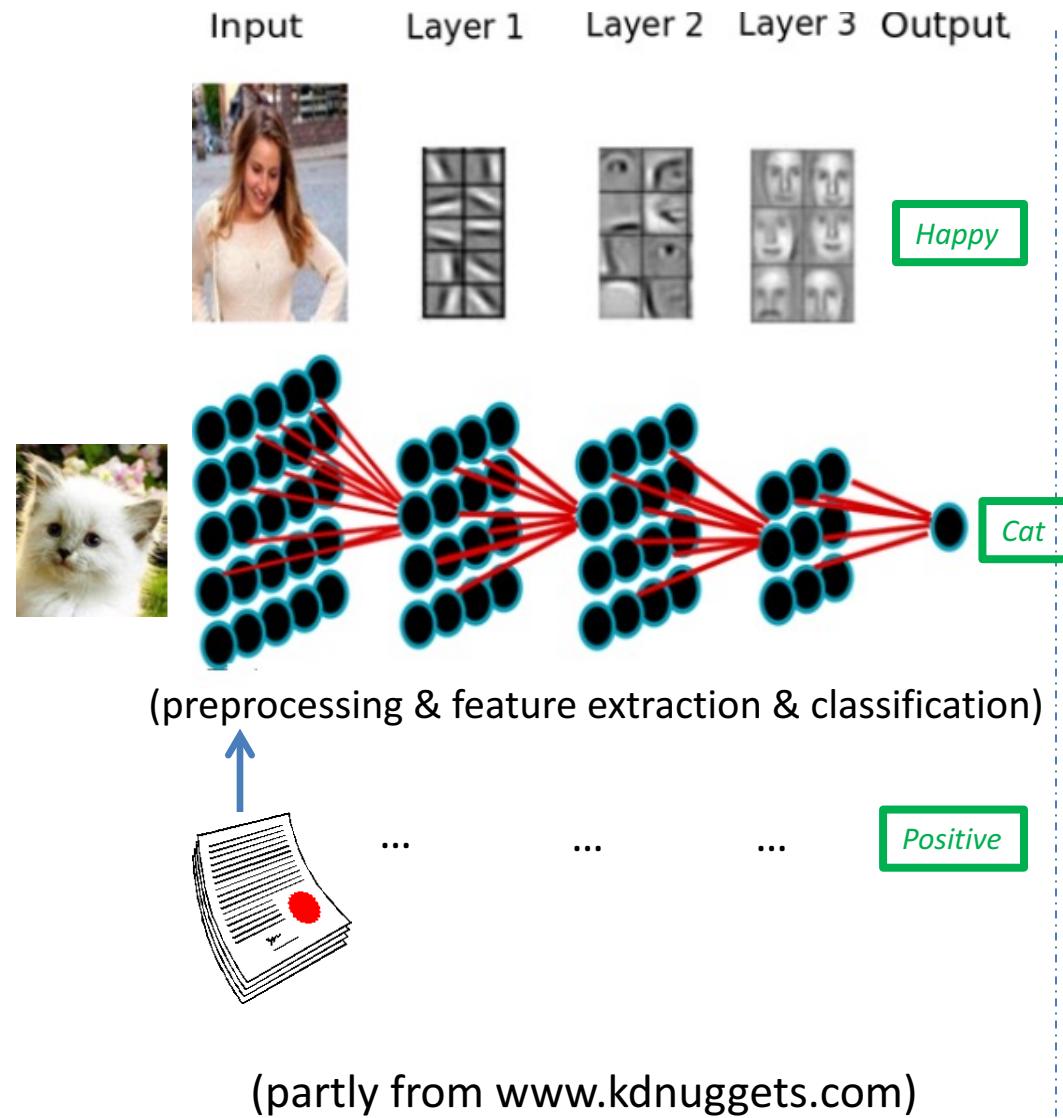
A ***learning environment*** or a ***set of sample data***

A ***model***

A ***learning algorithm***

***Does a machine learner need a teacher? Where is the teacher?***

# How Does Machine Learning Work ?



Neural network model (one hidden layer) :

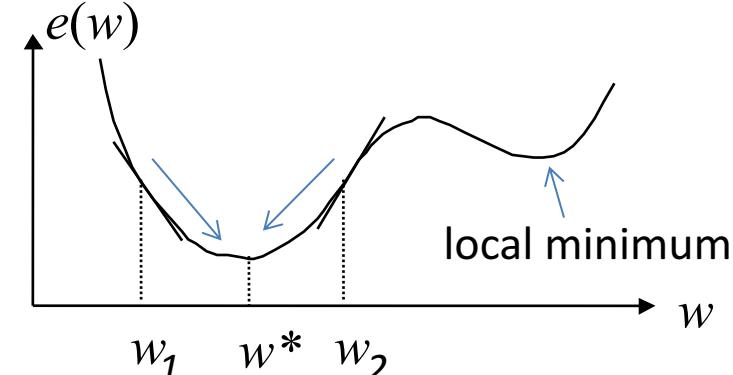
$$\begin{aligned}y_k &= f\left(\sum_{i=1}^{n_h} w_{ki}^o \cdot h_i - \theta_k^o\right) \\&= f\left(\sum_{i=1}^{n_h} w_{ki}^o \cdot f\left(\sum_{j=1}^n w_{ij}^h \cdot x_j - \theta_i^h\right) - \theta_k^o\right)\end{aligned}$$

Learning algorithm (gradient descent):

$$\Delta w = \alpha(z - y) \frac{dy}{dw} = -\alpha \frac{de}{dw}$$

$$e = \frac{1}{2}(z - y)^2$$

Error reduction by learning:



$$\left(\frac{de}{dw} < 0, \Delta w > 0\right) \quad \left(\frac{de}{dw} > 0, \Delta w < 0\right)$$

# Learning Viewed as Search for Optimal Models

A model for performing a task can be defined by its structure and parameters,

$$\text{e.g., } y = f(x, W)$$

$y$  – output,  $x$  – input,  $f$  – model structure,  $W$  – model parameters.

The process of learning can be viewed as **searching** the space of possible representations for a model (structure or/and parameters) that maximises the performance measure.

If we fix model structure  $f$ , then learning is to search for optimal model parameters  $W$  (example on previous slide). This is **parametric learning**.

In the lecture next week, the model involved is a decision tree that is not fixed, but constructed by learning. When the model structure  $f$  is not fixed, but updated/optimised by learning, it is **structural learning**.

# A Taxonomy of Learning Tasks

## Learning to classify

Given a set of training examples and their associated class labels,  
Learn to correctly predict the classification of unclassified examples.

e.g., a parent teaching a child to recognise animals by showing  
the child pictures of animals with associated names.

## Learning to predict numerical values (regression or approximation)

Given a set of training examples and associated numerical values,  
Learn to correctly predict the numerical value for other examples in which  
it is not known.

e.g., learning to predict tomorrow's temperature from recorded  
weather data or learning to evaluate game states.

Both of these are sometimes called *supervised learning*.

# A Taxonomy of Learning Tasks (2)

## Learning to form groups (clustering)

Given a set of unclassified/unlabelled examples,

Develop a “sensible” scheme for classifying them.

e.g., learning to group the cities around the world

Clustering is often called ***unsupervised learning***.

## Learning what to do next (*reinforcement learning*)

Given the experience of engaging actively in a task,

Learn to improve performance when engaged in similar tasks in future

e.g., learning to play game of Go, robot navigation

# **Topics on Machine Learning for CE213**

## **Decision Tree Induction (structural learning)**

A classical ‘learning to classify’ approach

## **Neural Networks (parametric and/or structural learning)**

McCulloch-Pitts neuron model

Multilayer neural networks and error back-propagation

Learning to predict or classify

## **Clustering (unsupervised learning)**

K-means, Agglomerative hierarchical clustering

## **Reinforcement Learning**

The Q learning algorithm

## **Genetic Algorithms**

# A Very Brief History of Machine Learning

## **1940s**

McCulloch-Pitts neuron model. Hebb's learning rule.

## ***Early 1950s***

Turing's rebutting of Lady Lovelace's Objection suggested that computers could learn for themselves.

## ***Middle 1950s – Middle 60s***

Some neural network learning systems were developed, including Perceptron learning rule, Delta rule. There was also some work on symbolic learning systems.

## ***Late 1960s***

Eclipse of neural networks as a result of wildly over-optimistic claims about their capabilities and criticisms from Prof. Marvin Minsky at MIT.

# Very Brief History of Machine Learning (2)

## ***Early 1970s – Middle 1970s***

Very little was done in machine learning – most researchers believed that they must first solve “the knowledge representation problem”.

## ***Late 1970s***

Machine learning was viewed as potential solution to the “knowledge bottleneck” in expert systems. Decision tree was proposed.

## ***Middle 1980s***

Renaissance of neural network approaches with many new models and learning algorithms, including generalised Delta rule, error backpropagation learning algorithm, multilayer perceptron.

## ***Late 1980s – Present***

Research in machine learning techniques has been massively expanded, including deep learning in the past decade.

It is machine learning that has made breakthroughs in AI in recent years!  
(AlphaGo, Face Recognition, Self-driving Car, Big Data Analysis, ...)

# Mathematical Preliminaries (mostly for self study)

There are a couple of branches of maths that occur repeatedly in machine learning:

## Probability

Is needed because machine learning algorithms usually draw statistical conclusions from evidence.

## Logarithm

Is fundamental to information theory that is widely used in machine learning – notably in **decision tree induction (information gain)**.

The next few slides provide a brief reminder of the fundamental ideas of these mathematical topics.

# Probability

**Probability is likelihood** measured on a scale from 0 to 1.

Suppose  $E$  is some event

$P(E) = 1$  means  $E$  certainly occurs/has occurred.

$P(E) = 0$  means  $E$  certainly does not occur/has not occurred

$P(E) = 0.5$  means it is equally likely that  $E$  does or does not occur.

If two events,  $X$  and  $Y$ , are **statistically independent** (i.e., neither helps you predict the other) then

$$P(X \wedge Y) = P(X) \times P(Y)$$

Conversely, if

$$P(X \wedge Y) \neq P(X) \times P(Y)$$

This indicates that the knowledge of one event can help you predict the other. This gives rise to the concept of conditional probability.

**$X \wedge Y$ : X and Y occur simultaneously.**

# Conditional Probability

The **conditional probability** of event  $X$  to occur, given that event  $Y$  has occurred, is defined as

$$P(X|Y) \equiv \frac{P(X \wedge Y)}{P(Y)}$$

**Do not confuse  $P(X|Y)$  with  $P(Y|X)$ .**

The probability that someone known to be an American is the US President is about  $3 \times 10^{-9}$ . ( $X$  – being the US President,  $Y$  – being an America (condition))

The probability that the US President is an American is 1.

Note that if  $X$  and  $Y$  are independent:

$$P(X|Y) = P(X)$$

which is, of course, another way of saying that knowing  $Y$  does not help you predict  $X$ .

# Bayes Theorem

There is a simple relationship between  $P(X|Y)$  and  $P(Y|X)$ :

$$P(X|Y) = P(Y|X) \times \frac{P(X)}{P(Y)}$$

This is known as **Bayes Theorem**.

It can also be expressed as

$$P(X \wedge Y) = P(X|Y) \times P(Y) = P(Y|X) \times P(X)$$

Bayes theorem forms the basis of an important branch of machine learning, such as naïve Bayes learning and Bayesian networks, but they will not be covered in this module.

# Logarithm

The **logarithm** of a number  $x$  is simply the power  $p$ , to which a fixed number  $b$  (the **base**) must be raised to produce the number.  $b^p = x \Leftrightarrow p = \log_b(x)$

## Some simple examples

$$8 = 2 \times 2 \times 2 = 2^3$$

So logarithm of 8 to base 2 is 3, which is written  $\log_2(8) = 3$

$$100000 = 10 \times 10 \times 10 \times 10 \times 10 = 10^5$$

So logarithm of 100000 to base 10 is 5, which is written  $\log_{10}(100000) = 5$

$$\log_2(32) = 5$$

$$\log_{10}(100) = 2$$

$$\log_2(2) = \log_{10}(10) = 1$$

$$\log_2(1) = \log_{10}(1) = \log_x(1) = 0$$

## Logarithm (2)

Logarithms to **base 2** are particularly important in computer science.

Here are few more examples:

$$\log_2(8) = \log_2(2^3) = 3$$

$$\log_2(0.5) = \log_2\left(\frac{1}{2}\right) = \log_2(2^{-1}) = -1$$

$$\log_2(0.25) = \log_2\left(\frac{1}{4}\right) = \log_2\left(\frac{1}{2^2}\right) = \log_2(2^{-2}) = -2$$

$$\log_2(0.125) = \log_2\left(\frac{1}{8}\right) = \log_2\left(\frac{1}{2^3}\right) = \log_2(2^{-3}) = -3$$

# Summary

- **Why Do We Need Machine Learning?**
- **What Is Machine Learning?**
- **Key Elements of Machine Learning**
- **How Does Machine Learning Work?**
- **Learning Viewed as Search for Optimal Models**
- **A Taxonomy of Learning Tasks**
  - Learning to classify, Learning to predict numerical values
  - Clustering, Reinforcement learning
- **Brief History of Machine Learning**
- **Mathematical Preliminaries (mostly for self study)**
  - Probability, Logarithms

# CE213 Artificial Intelligence – Lecture 11

## Decision Tree Induction: Part 1

What is induction?

What is a decision tree?

How to do decision tree induction?

(Information and information gain will be used for decision tree induction)

Is it structural learning or parametric learning?

# What is Induction?

**Induction** is learning (or drawing conclusions) by generalising from samples or experiences. It may be contrasted with **deduction**.

Consider the following proposition:

*The sum of any two odd numbers is even*

You could repeatedly add pairs of odd numbers and notice that the result is always even.

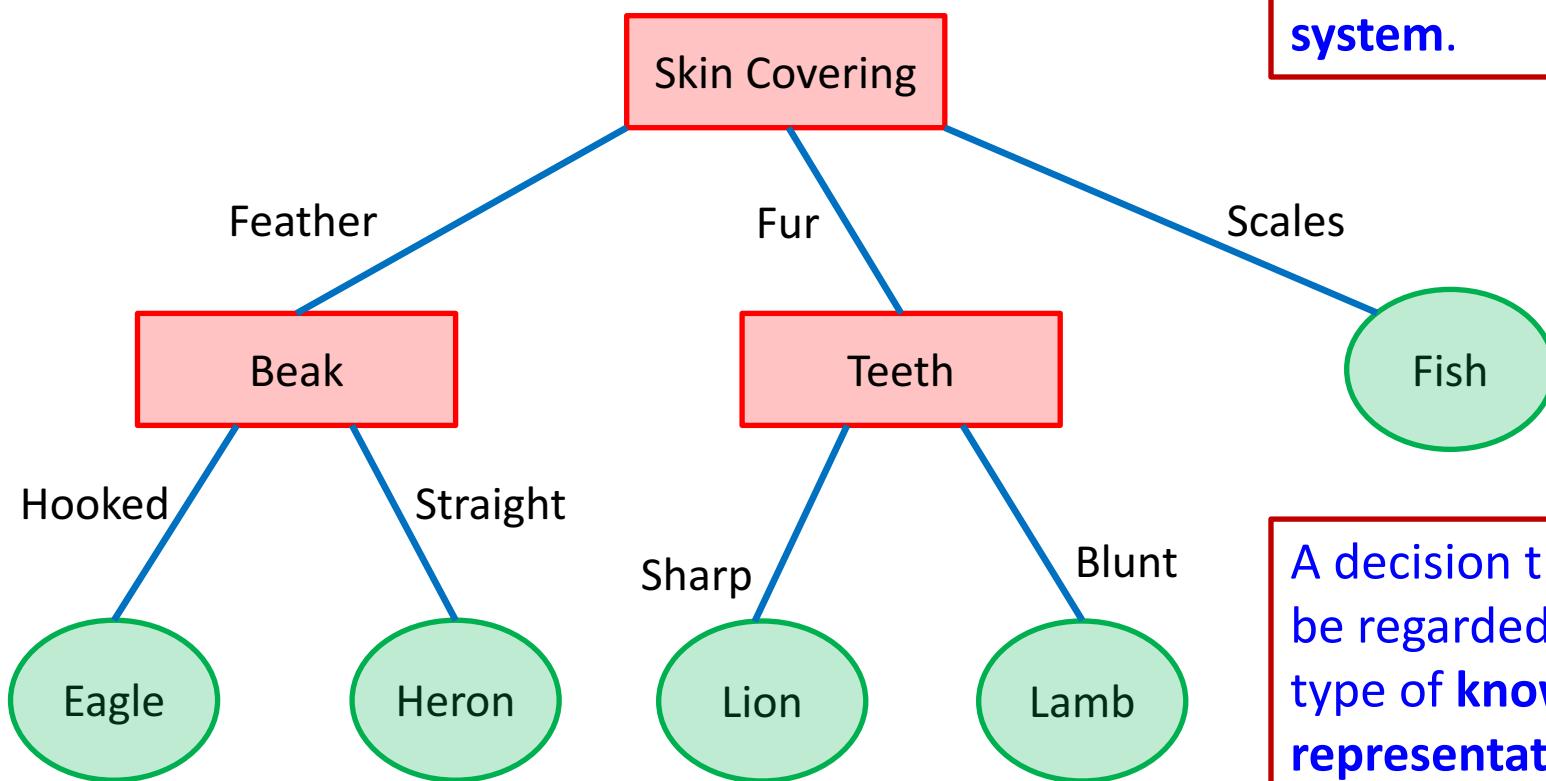
That is **induction**

Or you could construct a mathematical proof that the result will necessarily be even.

That is **deduction**

# What is a Decision Tree?

Let's start with an example:  
a very simple decision tree (or subtree):



You may compare it  
with a **search tree**,  
or a **production  
system**.

A decision tree can  
be regarded as a  
**type of knowledge  
representation**.

# What is a Decision Tree? (2)

*It is a tree for making decisions based on attribute values*, in which:

- Each leaf node is associated with a **class or decision**.
- Each non-leaf node is associated with one of the **attributes** that objects possess.
- Each branch is associated with a particular **value** that the attribute of the connected parent node can take.

**So what is decision tree induction?**

It is a procedure that, based on a given set of training samples, attempts to build a decision tree capable of predicting the class of any new sample (generalising from training samples).



# The Basic Decision Tree Induction Procedure

```
METHOD buildDecTree(samples,atts)
Create node N if necessary; //starting as a node, ending as a tree
IF samples are all in same class
THEN RETURN N labelled with that class;
IF atts is empty
THEN RETURN N labelled with modal class 1;
bestAtt = chooseBestAtt(samples,atts);
label N with bestAtt;
FOR each value  $a_i$  of bestAtt
     $s_i$  = subset of samples with bestAtt =  $a_i$ ;
    IF  $s_i$  is not empty
        THEN
            newAtts = atts - bestAtt;
            subtree = buildDecTree( $s_i$ ,newAtts); //recursive
            attach subtree as child of N;
        ELSE
            Create leaf node L;
            Label L with modal class;
            attach L as child of N;
RETURN N;
```

Most important

{Note 1: Model class is the class of the group with the maximum number of samples or highest frequency}

Will come back  
to this pseudo  
code later with  
an example.

# What is a Training Sample Set?

## What is a training sample set?

It is a set of labelled samples, drawn from some population of possible samples.

The training set is almost always a very small fraction of the population.

## What is a sample?

Typically decision trees operate using samples that take the form of *feature or attribute vector*.

An **attribute vector** is simply a vector whose elements are the *values* taken by the *attributes* of objects in the sample set.

e.g., a heron might be represented as follows:

<i>Skin Covering</i>	<i>Beak</i>	<i>Teeth</i>	<i>Class</i>
Feather	Straight	None	Heron

# Choosing the Best Attribute

**What is “the best attribute”?**

Many possible definitions.

A reasonable answer:

The attribute that best discriminates the samples with respect to their classes.

**So what does “best discriminates” mean?**

Still many possible answers.

Many different criteria have been used.

The most popular is *information gain*.

We need to know how to calculate information in order to calculate information gain.

# Shannon's Information Formula

Given a situation in which there are N unknown outcomes:

How much information have you acquired once you know what the outcome is?

Let's begin by considering the simplest possible situation:

2 outcomes, each equally likely

e.g., A coin toss

We can **define** the amount of **information** you acquire when you learn the outcome of such an event as **1 bit**. ( 0 for one outcome and 1 for the other outcome)

## Shannon's Information Formula (2)

Now consider picking 1 card at random from a pack of 8.  
i.e., 8 equiprobable outcomes (8 different cards)

One way to make the random choice would be to toss a coin 3 times.

This would provide a binary number in the range 0~7  
The number could then be used to choose the card

So when you learn which of 8 equiprobable outcomes has occurred you would have acquired 3 bits of information.

## Shannon's Information Formula (3)

We can extend this to other situations with equiprobable outcomes:

Toss a coin	2 outcomes	1 bit of information
Pick 1 card from 8	8 outcomes	3 bits of information
Pick 1 card from 32	32 outcomes	5 bits of information
Pick 1 card from 128	128 outcomes	7 bits of information

Or in general, for a situation or event with N equiprobable outcomes:

$$\text{Information} = \log_2(N) \text{ bits}$$

Since the probability of each outcome  $p = 1/N$ , we can also express this as

$$\text{Information} = -\log_2(p) \text{ bits}$$

For example, being told the outcome when there are 5 equiprobable outcomes:

$$\text{Information} = \log_2(5) \text{ bits} = 2.322 \text{ bits}$$

# Shannon's Information Formula (4)

## Non-equiprobable outcomes:

Consider picking 1 card from a pack containing 127 red and 1 black.

There are 2 possible outcomes, red card or black card, but you would be almost certain that the result would be red.

Thus being told the outcome in this situation usually gives you less information than being told the outcome of an event with two equiprobable outcomes.

We need to modify the definition of information to reflect the fact that there is **less information to be gained when we already know that some outcomes are more likely than others.**

# Shannon's Information Formula (5)

Shannon proposed the following information formula:

$$\text{Information} = - \sum_{i=1}^N p_i \log_2(p_i) \text{ bits}$$

where

$N$  is the number of alternative outcomes

$p_i$  is the probability of the  $i$ th outcome

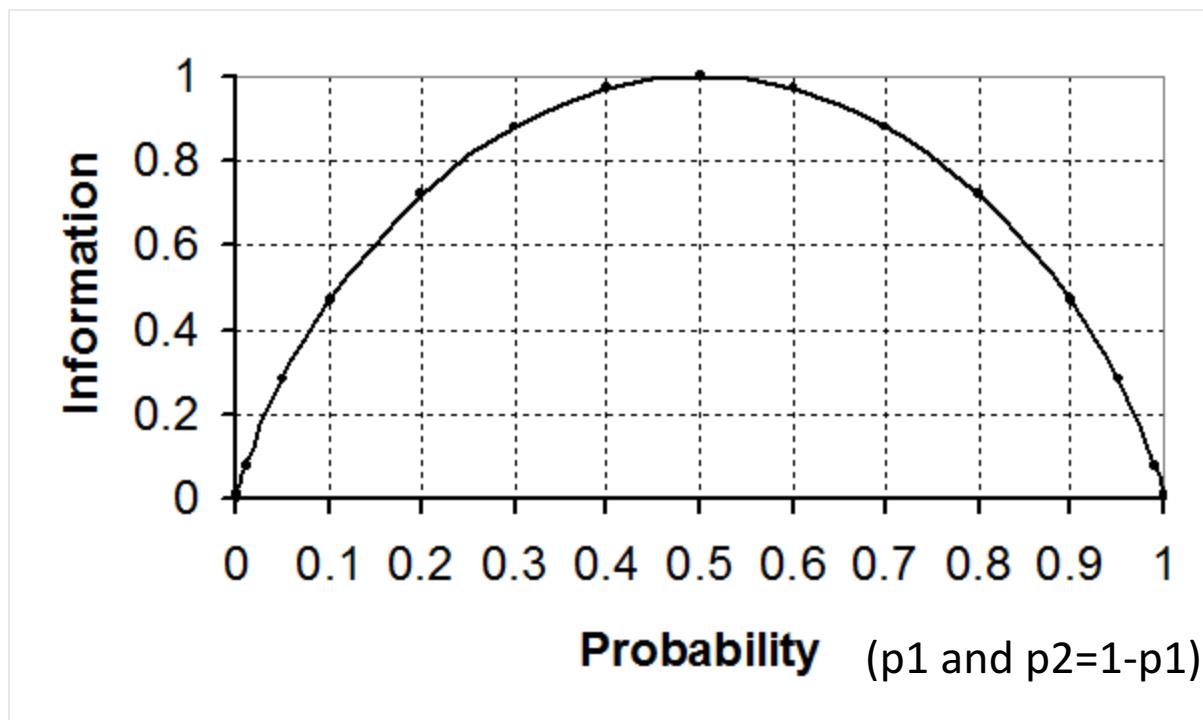
Notice that this formula reduces to  $-\log_2(p)$  when the outcomes are all equiprobable.

So Shannon's formula is a generalization of the formula we have already derived.

**In order to calculate the information of an event, you need to know the number of outcomes and the probability of each outcome. An event may be represented as a dataset, from which the probability of each outcome can be calculated.**

## Shannon's Information Formula (6)

If there are only **two outcomes**, we can use the following graph to show the change of Shannon's information with the probabilities of the 2 outcomes:



Information is also sometimes called ***uncertainty*** or ***entropy***.  
It is clear that higher information means higher uncertainty.

# Using Information Gain to Evaluate Attributes

*Information gain = Information before knowing attribute value  
– Information after knowing attribute value*

An example to show how knowing the value of an attribute affects the information or uncertainty about the class or outcome:

Suppose

You have a set of 100 samples (e.g., a set of colourful objects).

These samples fall in two classes,  $c_1$  and  $c_2$  (e.g., hot, cold):

70 samples are in  $c_1$  and 30 samples are in  $c_2$

How uncertain are you about the class that a sample belongs to?

$$\begin{aligned}\text{Information}_{\text{class}} &= -p(c_1) \times \log_2(p(c_1)) - p(c_2) \times \log_2(p(c_2)) \\ &= -0.7 \times \log_2(0.7) - 0.3 \times \log_2(0.3) \\ &= -0.7 \times (-0.51) - 0.3 \times (-1.74) = 0.88 \text{ bits}\end{aligned}$$

# Using Information Gain to Evaluate Attributes (2)

Now suppose *Colour* is one of the attributes with values *red* and *blue*.

The 100 samples are distributed as follows in terms of colour:

	<i>red</i>	<i>blue</i>
$c_1$	63	7
$c_2$	6	24

What is the information about the class of the samples whose *Colour* value is *red*?

There are 69 of them: 63 in  $c_1$  and 6 in  $c_2$ , which form a sample subset.

So for this *subset*,  $p(c_1) = 63/69 = 0.913$

and  $p(c_2) = 6/69 = 0.087$

Therefore

$$\begin{aligned}\text{Information\_class|colour\_red} &= -0.913 \times \log_2(0.913) - 0.087 \times \log_2(0.087) \\ &= 0.43 \text{ bits}\end{aligned}$$

# Using Information Gain to Evaluate Attributes (3)

Similarly, for the samples whose *Colour* value is *blue*:

There are 31 of them; 7 in  $c_1$  and 24 in  $c_2$ , which form another sample subset.

So for this *subset*,  $p(c_1) = 7/31 = 0.226$

and  $p(c_2) = 24/31 = 0.774$

Hence

$$\begin{aligned}\text{Information}_{\text{class}|\text{colour\_blue}} &= -0.226 \times \log_2(0.226) - 0.774 \times \log_2(0.774) \\ &= 0.77 \text{ bits}\end{aligned}$$

So

If we know the *Colour* is *red*, the remaining uncertainty is 0.43 bits

If we know the *Colour* is *blue*, the remaining uncertainty is 0.77 bits

How much information or uncertainty about the class will remain if we are told the *Colour*?

# Using Information Gain to Assess Attributes (4)

69% of samples are *red* and 31% of samples are *blue*.

So, if we are told the Colour:

69% of the time we will be told *red*

31% of the time we will be told *blue*

Hence, the average information about the class if we are told the value of the *Colour* attribute will be

$$\text{Information}_{\text{class}|\text{colour}} = 0.69 \times 0.43 + 0.31 \times 0.77 = 0.54 \text{ bits}$$

Compare this with the information about the class if we don't know the value of *Colour*, which we calculated earlier as 0.88 bits.

Therefore, the Colour attribute provides an ***information gain***:

$$\text{Information}_{\text{class}} - \text{Information}_{\text{class}|\text{colour}} = 0.88 - 0.54 = 0.34 \text{ bits}$$

You will learn how to use **information gain** to select best attributes and thus construct decision tree in the next lecture.

# Summary

## Decision Tree

Non-leaf nodes represent attributes;  
Leaf nodes represent classes;  
Branches represent attribute values.

## Decision Tree Induction

The Basic Decision Tree Induction Procedure (pseudo code)  
It is structural learning.  
It is recursive.

## Choosing the Best Attribute

### Shannon's Information Formula

Using **Information Gain** to Evaluate an Attribute

*(Example of decision tree induction using information gain: next lecture)*