

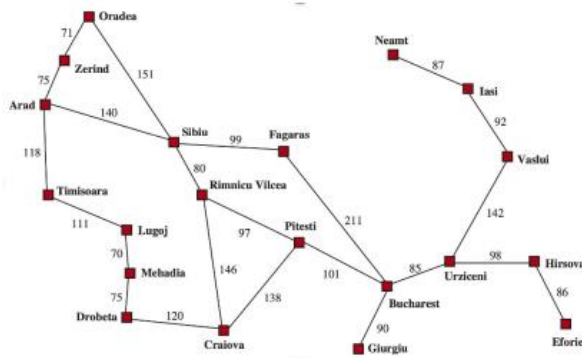
# Artificial Intelligence

- **Rational Agent:** Something that acts rationally.
- **Acting Rationally:** Acting in a way to achieve the best outcome
- **State/Problem Space:** A discrete space comprised of all states and actions between them which represents the search problem.

## Search Problems

A search problem is a situation where an agent chooses a series of **actions** to take it from an **initial state** to a specified **goal state**. You can identify solutions and sometimes **optimal** solutions!

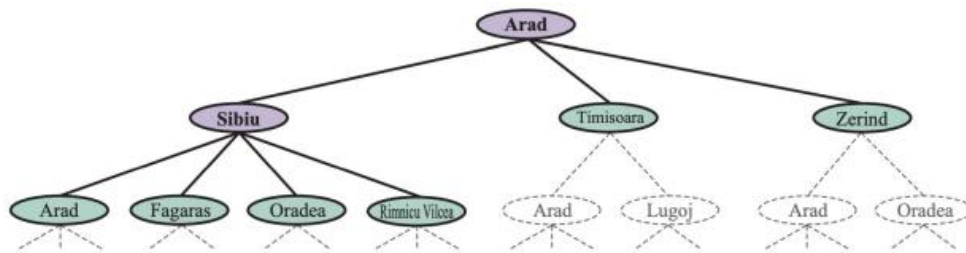
## Representation



## Graphs

A collection of **vertices** and **edges**. They can be **weighted** or unweighted and **directed** or undirected.

## Search Trees



We use a **search tree** to represent a search in a state space.

- Every node except the root node has one parent node.
- No node can be its own ancestor so no loops.
- The children of each node are the states accessible in one action.

Search trees are built from state space graphs. They contain two sets:

- Explored states.
- Frontier: states which can be discovered using only one action.

## Comparing Algorithms

- **Complete:** Guaranteed to find a solution if one exists.
- **Optimal:** Guaranteed to find an optimal solution if one exists.
- **Time Complexity:** Measured in terms of the number of nodes generated during the search.
- **Space Complexity:** Measured in terms of the maximum number of nodes stored in memory.

## Specifying Time & Space Complexity

The following is used to express time and space complexity:

**Branching Factor ( $b$ ):**

Number of children which every node has.

**Depth of the Shallowest**

**Goal Node ( $d$ ):** Depth of the optimal solution.

**Maximal Length of any**

**Path ( $m$ ):** Longest distance from the root to any path.

## Uninformed Search Algorithms

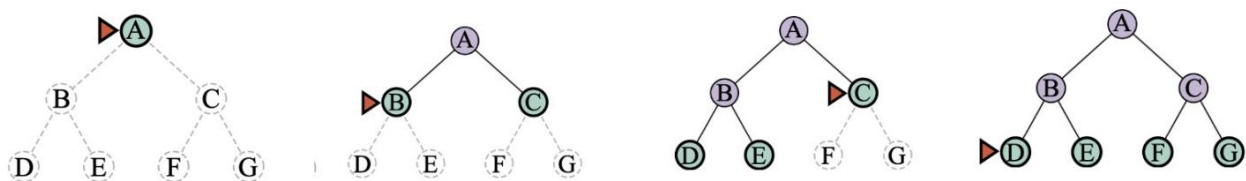
Uninformed search algorithms explore a problem space **without any knowledge of the problem** other than the **initial state** and **possible actions**.

### Breadth-first Search

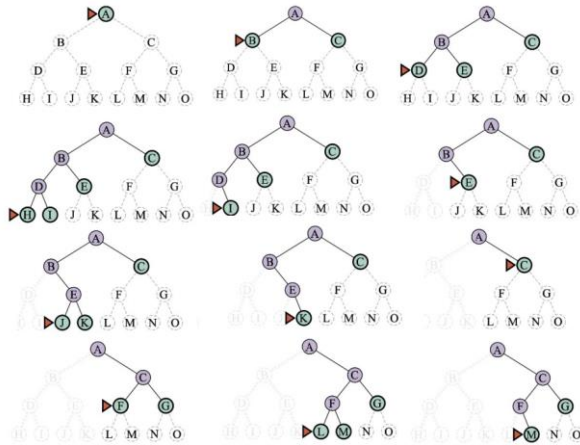
Always explores the **shallowest state in the frontier**. If there are multiple at the same level, it chooses the first one.

#### Properties

- **Solution:** Complete & optimal.
- **Time Complexity:**  $O(b^d)$
- **Space Complexity:**  $O(b^d)$



## Depth-first Search

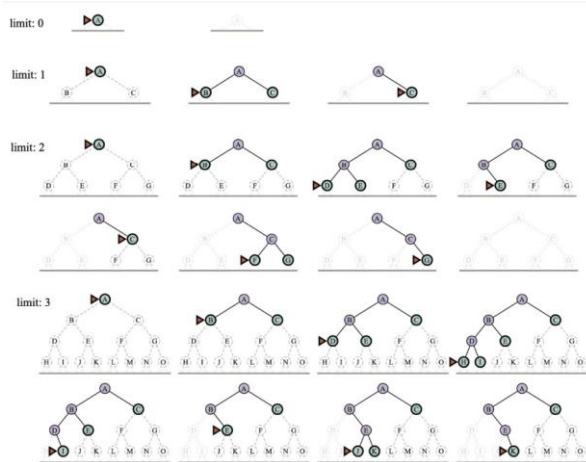


Always explores the **deepest state in the frontier**. If there are multiple at the same level, it chooses the first one.

### Properties

- **Solution:** Complete.
- **Time Complexity:**  $O(b^m)$
- **Space Complexity:**  $O(bm)$

## Iterative Deepening Search



A depth-first search, up to a certain depth ( $d$ ).

### Properties

- **Solution:** Complete & optimal.
- **Time Complexity:**  $O(b^d)$
- **Space Complexity:**  $O(bd)$

## Informed Search Algorithms

In addition to the initial state and possible actions, informed search algorithms **contain information about the goal state**. They use an **evaluation function (lookup table)** to provide value to each node which helps to select the node to explore next.

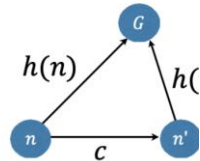
Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

A **heuristic function** is a type of evaluation function which **estimates the cost of the best path from the current node to the goal**. In the following example, the heuristic function is the straight-line distance from a given node to the goal.

### Properties of Heuristic Functions

#### Consistency

$$h(n) \leq c + h(n')$$



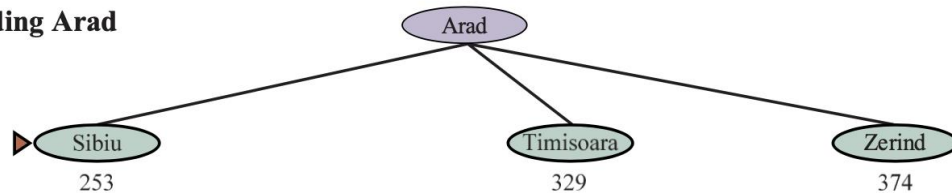
- **Admissible:** Doesn't overestimate the cost to reach the goal state.
- **Consistent:** It satisfies the triangle inequality. All consistent heuristics are admissible.

## Greedy Search

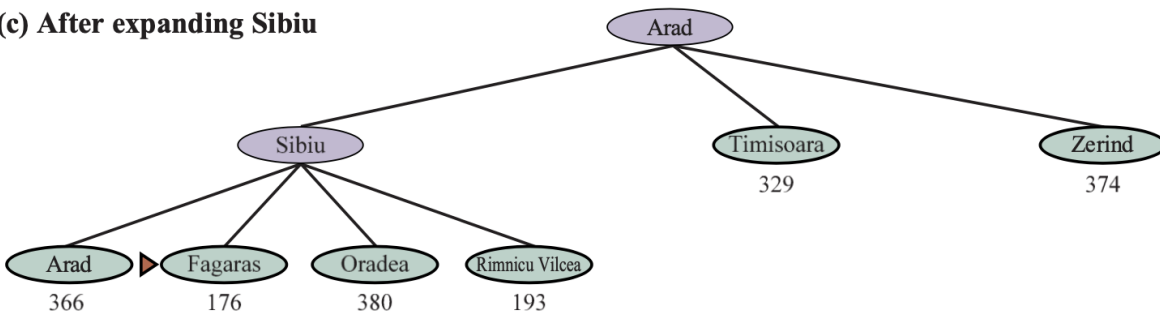
A **complete** algorithm, meaning it always finds a result. Starting with the root node:

1. Add the child nodes to the frontier.
2. Explore the node that has the lowest value in the evaluation function.
3. Repeat.

#### (b) After expanding Arad



#### (c) After expanding Sibiu



## A\* Search

A\* search uses a new evaluation function:

$$f(n) = g(n) + h(n)$$

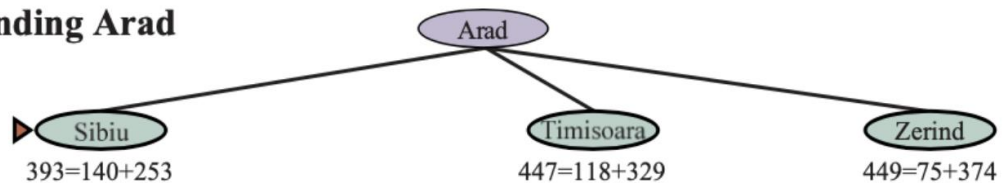
Where  $n$  is the current node,  $g(n)$  is the distance from the initial node to  $n$  and  $h(n)$  is the straight-line distance from the current node to the goal.

- Takes slightly longer than a Greedy search.
- **Optimal with an admissible heuristic.**
- **Optimally efficient with a consistent heuristic**, no algorithm with the same info can perform better.

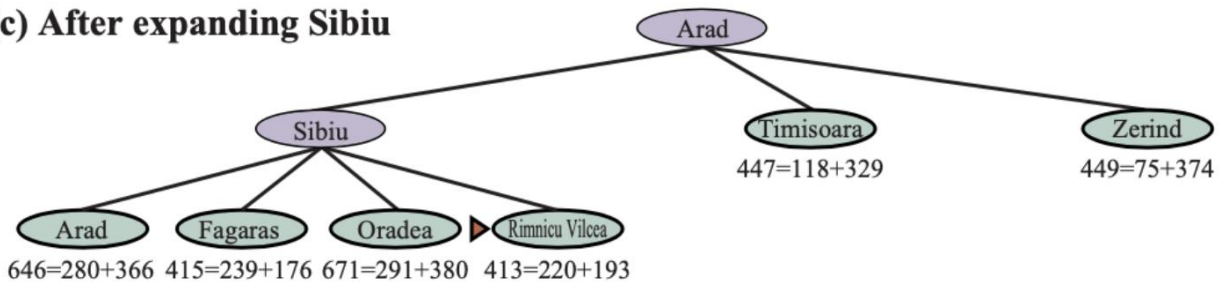
Starting with the root node:

1. Add the child nodes to the frontier.
2. Explore the node that has the lowest value in the evaluation function.
3. Repeat.

### (b) After expanding Arad



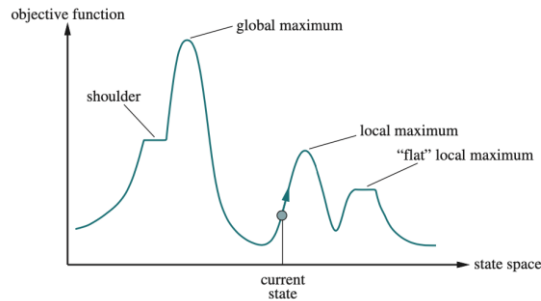
### (c) After expanding Sibiu



## Local Search

Local search algorithms allow us to find a **sub-optimal solution** while maintaining a **good space complexity**.

## Hill Climbing Algorithms



**Navigates to the neighbouring state which leads to the solution.** In the context of hill climbing, this would be navigating to the state which is higher up.

Stochastic hill climbing and random-restart hill climbing can help **avoid getting stuck at local maxima**.

## Other Local Search Algorithms

- **Simulated Annealing:** Takes inspiration from a technique used in metallurgy.
- **Genetic algorithms:** Takes inspiration from natural selection

## Constraint Satisfaction

A search problem can be solved using constraint satisfaction if it has the following attributes:

- A set of variables  $X = \{x_1, x_2, x_3, \dots, x_n\}$
- A set of values these variables can take  $D = \{d_1, d_2, d_3, \dots, d_n\}$
- A set of constraints  $C$  detailing rules for the values the variables can collectively take.



$X = \{WA, NT, Q, SA, NSW, V, T\}$

$D_i = \{R, G, B, Y\}$

$C = \text{"No adjacent countries can have the same colour"}$

## Solving C.S. Problems

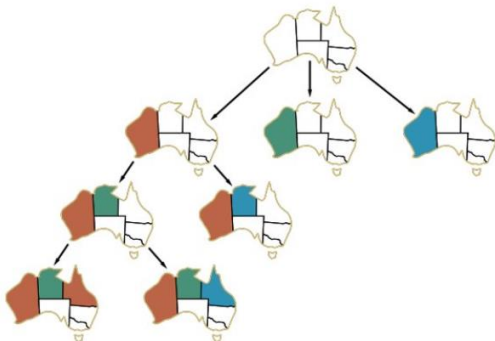
We translate it into a search problem:

- **State:** An assignment of values to variables.
- **Action:** Assigning a state to a variable.
- **Solution:** A state which is complete and consistent.

**Complete:** Assigns all values to all variables. A state which isn't complete is **partially complete**.

**Consistent:** Doesn't break the constraints.

Several algorithms can be used to solve constraint satisfaction problems.



## Backtracking Search

Combines a depth-first search with **constraint propagation**. Only the children which are consistent with the constraints are investigated, reducing the frontier which reduces time and space complexity.

## Heuristics for Backtracking

A **heuristic function** is a type of evaluation function which **estimates the cost of the best path from the current node to the goal**.

- **Minimum Remaining Values (MRV):** Expand the variable with the least remaining values. This keeps the branching factor down to a minimum, decreasing time and space complexity.
- **Least Constraining Value (LCV):** The value of this variable should be the one which constrains the fewest other values because this increases the probability of finding a solution down a given branch.

## Min-Conflicts

If a constraint satisfaction problem has multiple solutions, this algorithm can find the closest local maximum using local search.

1. Starts with a random state.
2. Reassigns variables to minimise the number of conflicts until a solution is found.



## Adversarial Search

Searches with the following properties can be solved using an adversarial search algorithm:

- An initial state.
- Players take turns taking actions.
- A terminal test is used to see if the game is over.
- The utilities of the game over states determine the optimality.

**Zero-Sum Game:** A positive outcome for one player causes a negative outcome for another player.

## Utility Functions

Assign values to final states. The scores they output need to be **zero-sum**.

*e.g. identifying the score at the end of a noughts-and-crosses game from 1, -1 and 0.*

## Minimax

In a two-player, zero sum game we can assume:

- One player is trying to minimise the utility function.
- One player is trying to maximise the utility function.
- Both players always play their **optimal action**.

Minimax algorithms allow a player to find the **best possible action at each stage of a game**.

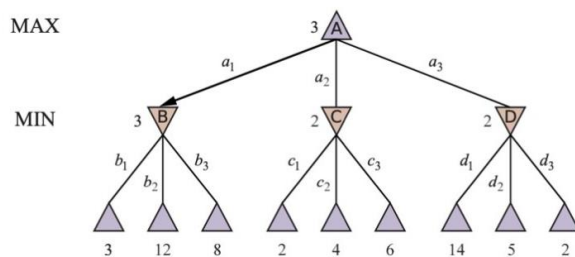
## Properties

- Will always find a solution (complete) if the game tree is finite.
- Uses depth-first search has these time and space complexity values.

## Diagrams

Help us to find **the value the utility function will produce if both players play optimally**.

- Each row is either the maximiser or minimiser.
- Labelling every node has **exponential complexity**.



Starting at the bottom:

1. Determine which node the parent would pick.
2. Assign this value to the parent.
3. Repeat for all branches at that level.

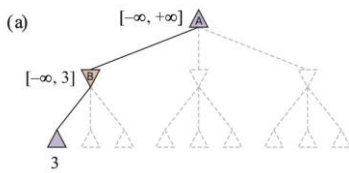
4. Move up to the next layer and repeat.

## Alpha-beta Pruning

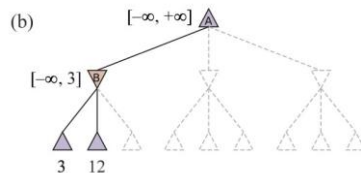
A variant of minimax which **uses a depth-first** to decrease complexity. Upper and lower bounds are assigned to each node which allows us to prune the branches which don't need exploring.

### Example

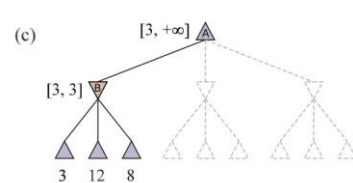
1.



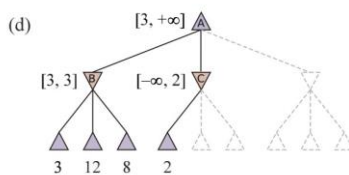
2.



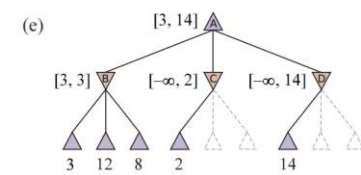
3.



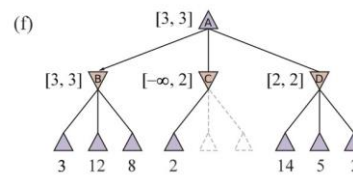
4.



5.



6.



### Evaluation Functions
















Sometimes even alpha-beta pruning isn't efficient enough. Evaluation functions help to **improve efficiency by giving states to values without calculating the full tree.**

*e.g. storing previously solved positions in a table for reference*

# Propositional Logic

Propositional logic can be used in problems where **sensory information can be used to find a solution**. You can form **atomic sentences**, which are observations about the state, and use these to form more complex sentences to reason about the world. These sentences can either be true or false.

- **Model:** A full assignment of truth values to atomic sentences.

4				
3		  		
2				
1				
	1	2	3	4

## The Wumpus World

In the Wumpus World, an agent can move around a grid, fire their bow or pick up gold. The initial state is in one corner and the gold, the goal state, is in a random square. The agent loses if they enter a square with a Wumpus or a pit. There is a stench in all squares adjacent to the Wumpus, a breeze adjacent to the pit, and the gold glitters.

## Atomic Sentences



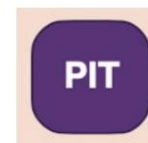
$P_X$  = "There is a stench in square X"



$Q_X$  = "There is a breeze in square X"



$R_X$  = "There is a Wumpus in square X"



$S_X$  = "There is a pit in square X"

## Complex Sentence Examples

"There is **not** a Wumpus in square X"



$P_X \wedge Q_X$  = "There is a breeze **and** a stench in square X"

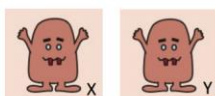


$S_X \rightarrow Q_Y$  =

"If there is a pit in square X **then** there is a breeze in square Y"



$R_X \vee R_Y$  = "There is a Wumpus in square X **or** a Wumpus in square Y"



$R_X \leftrightarrow R_Y$  =

"There is a stench in square X **if and only if** there is a Wumpus in square Y"



You can use propositional logic to describe the rules of the Wumpus world.

e.g.  $P = R_X \Rightarrow (P_Y \wedge P_Z)$  i.e.  $P$  = "If there is a Wumpus in square X, then there is a stench in square Y and square Z".

# Entailment & Knowledge Bases

- $\alpha \models \beta$  means that every model where  $\alpha$  is true,  $\beta$  is also true. Basically,  $\alpha$  entails  $\beta$
- We summarise everything we know about the world in a set of sentences and sort it in the **knowledge base (KB)**.
- We write  $KB \models \alpha$  if  $\alpha$  is always true when every sentence in the knowledge base is true.

## Model Checking

Allows us to check that  $KB \models \alpha$ . We simply check that alpha is true in all possible world where the sentences of KB are true. It has **exponential complexity** because there are  $2^n$  models for every  $n$  sentences.

For example, given this information about the world, we can use our knowledge base to work out where the pits can be (the worlds highlighted in red are the only ones consistent with the rules). We are only concerned about where the pits are, not the breezes.



## Soundness & Completeness

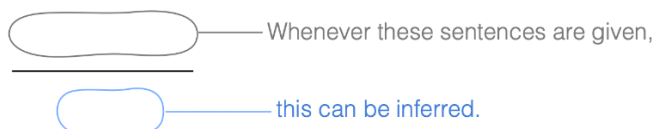
If we can derive  $\alpha$  from  $KB$  given algorithm  $i$ , we write  $KB \vdash_i \alpha$ .

- An algorithm is **sound** if  $KB \vdash_i \alpha$  means that  $KB \models \alpha$ .
- An algorithm is **complete** if  $KB \models \alpha$  means that  $KB \vdash_i \alpha$ .

Model checking is **sound and complete**.

# Proof Systems

An faster method of using observations about a problem to get new information about its state. A proof system is a **set of inference rules**.



$$\frac{\alpha \Rightarrow \beta, \alpha}{\beta}$$

This is **Modus Ponens** – a very important inference rule.

$$\frac{\beta}{\alpha \wedge \beta}$$

Basically, if  $\alpha$  implies  $\beta$  and  $\alpha$  is true, then  $\beta$  must also be true.

Another rule is that any part of an and statement can be isolated.

$$\frac{\alpha}{\neg \neg \alpha}$$

Any **logical equivalence** can be expressed as an inference rule.

## Example

Given KB, prove  $\neg P_{1,2}$

$$\begin{aligned} B_{1,1} &\Leftrightarrow (P_{1,2} \vee P_{2,1}) \\ (B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1})) \wedge ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1}) \\ (P_{1,2} \vee P_{2,1}) &\Rightarrow B_{1,1} \\ \neg B_{1,1} &\Rightarrow \neg(P_{1,2} \vee P_{2,1}) \\ \neg(P_{1,2} \vee P_{2,1}) \\ \neg P_{1,2} \wedge \neg P_{2,1} \\ \neg P_{1,2} \end{aligned}$$

## Resolution Proofs

A popular type of proof system. It is **sound** and **complete**.

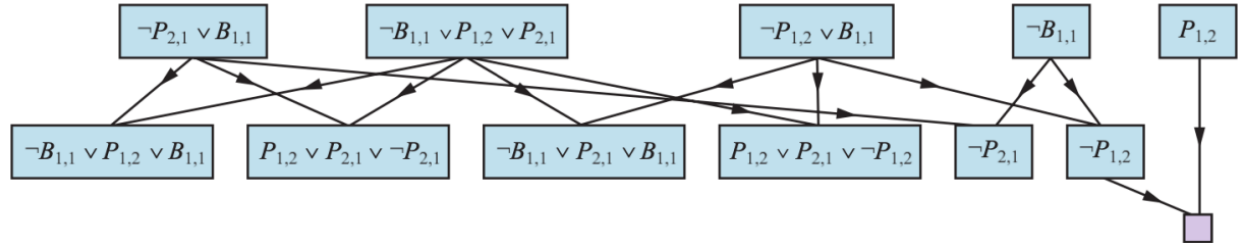
**Conjunctive Normal Form:** Conjunction of disjunction.

You can do a resolution proof using these steps:

1. Break all inference rules in the knowledge base into CNF.
2. If both a literal and its complement exist, you can cancel them out.
3. Repeat until you reach the desired conclusion.

### Example

To prove that  $KB \models \alpha$ , we need to show that  $(KB \wedge \neg\alpha)$  is impossible. So, we convert  $(KB \wedge \neg\alpha)$  into CNF and use the proof system until we reach a contradiction,  $(\neg KB \models \alpha)$  or we can't apply any more rules:



# Uncertainty

The language of **probability, probabilistic reasoning and inference** can be used to reason and act in uncertain situations.

## Terminology

- **Statistic:** A quantity that summarises some data.
- **Probability:** A measure of an uncertain event occurring.
- **Sample Space ( $\Omega$ ):** The set of all possible outcomes.
- **Event ( $\Phi$ ):** A subset of the sample space we're interested in. *E.g. landing a head when flipping a coin.*
- **Outcome ( $\omega$ ):** Mutually exclusive, exhaustive results of an uncertain event.
- **Realisation:** The outcome of a trial.
- **Random Variable:** A variable whose value is unknown.
- **Domain of a Random Variable:** The set of values a random variable can take.
- **Mutually Exclusive:** Cannot happen simultaneously.
- **Independent:** Events for which the outcomes don't impact one another.
- **Proposition:** A statement which is either true or false.
- **Unconditional/Prior Probability:** A probability in the absence of extra information.
- **Conditional/Posterior Probability:** A probability in the presence of extra information.

## Probability Distributions (PDFs)

The allocation of probability over all outcomes in a sample space. They tell us every outcome and their probabilities.

You can develop a model of the probability of each outcome in a sample space using tests, intuition or knowledge.

## The Mathematics

The probability of a given outcome  $\omega$  occurring is  $0 \leq p(\omega) \leq 1$ .

- $p(\omega) = 0$  - it won't happen.
- $p(\omega) = 1$  - it will happen.
- $\sum_{\omega \in \Omega} p(\omega) = 1$  - something will happen.
- $p(A, B) = p(A \cap B)$
- $p(e) = \sum_{\omega \in \Phi} p(\omega)$  - the probability of an event occurring is the **sum of the probabilities of each of the outcomes in the sample space where the event holds.**

# Calculating Probabilities

## OR / Union ( $\cup$ ) / Disjunction ( $\vee$ )

You can calculate the probability of one of multiple events occurring using the **additive rule**:

$$p(A \vee B) = p(A) + p(B) - p(A \wedge B)$$

## AND / Intersection ( $\cap$ ) / Conjunction ( $\wedge$ )

You can rearrange the **additive rule** to calculate the probability of a combinations of outcomes over multiple random variables.

$$p(A \wedge B) = p(A) + p(B) + p(A \vee B)$$

## Bayes' Theorem

$$p(A|B) = \frac{p(A \wedge B)}{p(B)} = \frac{p(B|A)p(A)}{p(B)}$$

## Product Rule

<i>Not Independent</i>	<i>Independent</i>
$p(X \wedge Y) = p(X Y) * p(Y)$	$P(X Y) = p(X) * p(Y)$

## Sum Rule / Marginalisation

We can calculate the probability of an event by summing all the probabilities of events where it occurs. We call this a **marginal probability**.

$$p(X) = \sum_y p(X \wedge Y) \quad \text{or} \quad p(X) = \sum_y p(X|Y) * p(Y)$$

# The Markov Decision Process

Enable sequential decision making under uncertainty.

- Outcomes are uncertain meaning they aren't guaranteed.
- Agents continuously interact with their environment.
- Rewards are received for discovering the outcome of a given action and state.

## The Markov Property

Systems with the Markov property can be modelled using the Markov Decision Process. The Markov property is that **actions only depend on the current state**. i.e.

$$p(s_{t+1} | s_t \wedge a_t)$$



## Properties

An MDP consists of:

- A set of **states**:  $\{s_0, s_1, \dots\}$
- A set of **actions**  $A(s)$  for each state  $s$ :  $\{a_{s_1}, a_{s_2}, \dots\}$
- A **transition model**: a set of transition probabilities from each state to another:

$$p(s_{t+1} | s_t \wedge a_t)$$

- A **reward** for each transition:  $r(s_{t+1}, s_t, a_t)$



## Transition Probabilities

These specify the probability of arriving at state  $s_{t+1}$  from state  $s_t$  given action  $a_t$ .

- For each state and action, the probabilities of the resulting states should total to 1.

## The Discounted Return Function

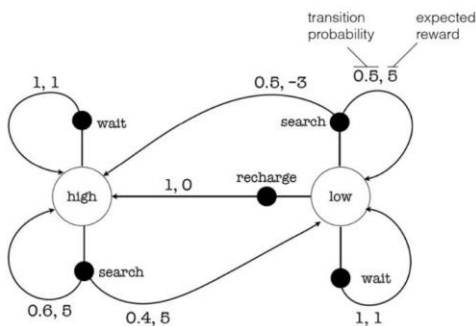
The objective is to maximise the long-term reward. We can evaluate the long-term rewards for a sequence of states in an MDP using a type of objective function called a **discounted return function** for a given **discount rate**  $0 \leq \gamma \leq 1$ .

$$G_t = \sum_{k=0}^{\infty} \gamma^k \cdot r_{t+k+1} + \gamma \cdot r_{t+2} + \gamma^2 \cdot r_{t+3} + \gamma^3 \cdot r_{t+4} + \dots$$

The discount rate specifies how much **value we place on future rewards**.

- $\gamma = 1$  means future rewards have the same value as present rewards.
- $\gamma = 0$  means future rewards have no value.

The lower the discount rate, the earlier the decision-maker should take actions.



## Transition Graphs

We can use a **transitional graph** to model a **Markov decision property**.

- **States:**  $S = \{low, high\}$
- **Actions:**  $A(s), s \in S$ 
  - $A(low) = search, wait, recharge$
  - $A(high) = search, wait$
- **Transition Model:**  $p(s' | s, a)$
- **Reward Function:**  $r(s, a, s')$

## Expected Value

The expected value of a random variable can be calculated using the following formula:

$$E[X] = \sum_x p(X = x) \cdot x$$
$$E[X + Y] = E[X] + E[Y]$$

## Policies for MDPs

$\pi(a|s)$  Policies specify the probabilities of taking actions from a state.

Stochastic Policy: Selects multiple possible actions for each state.			
$\pi_1$	$a_1$	$a_2$	$a_3$
$s_1$	0.25	0.25	0.5
$s_2$	0	0.8	0.2

Stochastic Policy: Selects multiple possible actions for each state.			
$\pi_2$	$a_1$	$a_2$	$a_3$
$s_1$	0	0	1
$s_2$	0	1	0

Deterministic policies don't fully determine the path through an MDP, since actions still have probabilistic outcomes.

- **Optimal Policy:** A policy which maximises the objective function. An MDP will always have at least one optimal policy.
- **Value Functions:** Help us to find the optimal policy.

## The State-Value Function

State-value functions calculate the expected **value of a state under a certain policy**.

$$V^\pi(s) = E_\pi[G_t | s_t = s] = E_\pi[r_{t+1} + \gamma G_{t+1} | s_t = s]$$

## Comparing Policies

A policy  $\pi$  is better than another if  $\forall s \in S (V^\pi(s) \geq V^{\pi'}(s))$ . The optimal policy is denoted by  $V^*$ .

- Any policy which is greedy, meaning that it always takes the action which gives the best possible expected result given its current state, is optimal.
- Once  $V^*$  has been calculated, we can identify the best possible action for each state meaning we can **always develop a deterministic optimal policy**.

## The Bellman Equation

A recursive equation which represents **the value of a state** in terms of the values of successor states, plus any rewards gained between the two.

This formula forms the basis for many different approaches that try to find the optimal policy to a given problem.

$$V^{\pi}(s) = \sum_a \pi(a|s) \sum_{r,s'} p(r, s'|a, s) * (r + \gamma V^{\pi}(s'))$$

*Read the example on Moodle.*

## Expected Value of an Action

You can calculate the expected value of an action under the policy  $\pi$  using the formula:

$$\sum_a \pi(a|s) \sum_{r,s'} p(r, s'|a, s) * r$$

## The Optimality Equation

An optimal policy has one action with a probability of 1 and all others are 0. This action is the optimal choice, the action with the highest expected value.

Hence, you can find the optimal value of a state by picking the action with the highest expected value:

$$V^*(s) = \max(a) \sum_{r,s'} p(r, s'|a, s) * (r + \gamma V^*(s'))$$

## Value Iteration

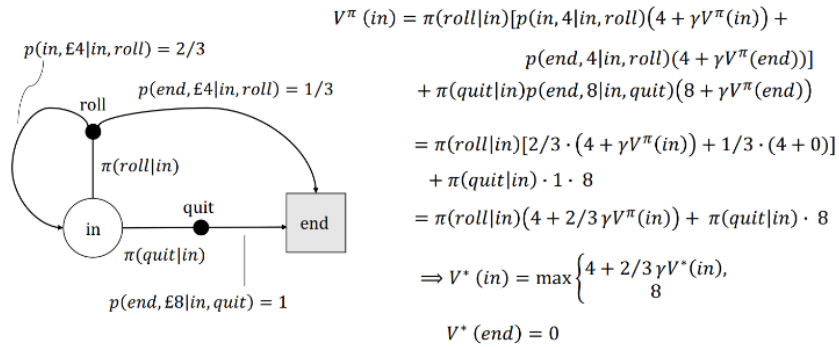
Helps us to find the optimal action for a given state.

1. Calculate the formula for each possible action resulting from a state.
2. Set  $V_k(s') = 0$ .
3. Calculate the proceeding value  $V_{k+1}(s)$  and repeat.
4. Once they have converged, select the action with the highest number. This is the **optimal action**.

## Example

The Bellman eq.

$$V^\pi(s) = \sum_a \pi(a|s) \sum_{r,s'} p(r,s'|a,s) \cdot (r + \gamma V^\pi(s')) \quad V(end) = 0$$



$$V^*(in) = \max \left\{ 4 + \frac{2}{3} \gamma V^*(in), 8 \right\} \quad \text{We want maximum total reward; } \gamma = 1$$

Solve analytically:

$$V^{\pi(roll|in)}(in) = 4 + 2/3 \cdot V^{\pi(roll|in)}(in) \Rightarrow V^{\pi(roll|in)}(in) = 12$$

$$\Rightarrow V^*(in) = \max\{12, 8\} = 12 \Rightarrow \pi^*(roll|in) = 1, \pi^*(quit|in) = 0$$

Solve by value iteration:

$$V_{k+1}(in) = \max\{4 + 2/3 \cdot V_k(in), 8\}$$

$$V_0(in) = 0$$

$$\Rightarrow 4 + 2/3 \cdot 0 = 4$$

$$\Rightarrow V_1(in) = \max\{4, 8\} = 8$$

$$\Rightarrow 4 + 2/3 \cdot 8 = 9.33$$

$$\Rightarrow V_2(in) = \max\{9.33, 8\} = 9.33$$

:

$$\Rightarrow 4 + 2/3 \cdot 11.991 = 11.994$$

$$\Rightarrow V_{17}(in) = \max\{11.994, 8\} = 11.994 \text{ Converged! } \pi^*(roll|in) = 1, \pi^*(quit|in) = 0$$

Try different  $\gamma$ !

# Machine Learning

The aim of machine learning is to find patterns which provide a general understanding of the phenomenon we wish to learn about.

**Machine Learning:** Any algorithm which improves its performance using data.

- **Quality** of data **is more important than quantity**.
- Data must be **representative** of the underlying phenomena being analysed.
- We are limited by our **model choice**.

There are three primary types of machine learning:

Supervised Learning	Unsupervised Learning	Reinforcement Learning
Runs an algorithm on a data set to produce a <b>model</b> which can predict a target variable.  This can be tested using existing data and fine-tuned to produce the same outcomes.	Like supervised learning but the data is <b>unlabelled</b> and <b>doesn't have a known result</b> .  Agents try to identify patterns in the data to identify similar data.	Learns through gathering data by <b>interacting with its environment</b> .  Its behaviour is tuned based on the outcome of these interactions to maximise a <b>performance metric</b> .

## Supervised Learning

Analysing the data set allows us to form hypotheses about the rules of a system which **enables us to identify patterns** and form a **model** which can **predict a target variable**.

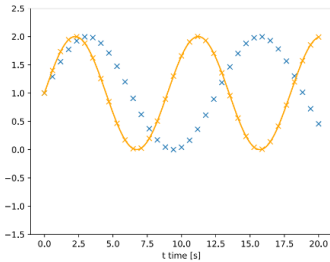
There are two types of supervised learning problems:

- **Classification:** The target is discrete. *E.g. spam and ham*
- **Regression:** The target is a real number. *E.g. weight*

The training data is a set of input and output pairs  $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$ . Keep in mind,  $x_i$  and  $y_i$  can be multidimensional. An unknown function exists linking the two:  $y = f(x)$ . Our aim is to discover a function  $h$  which best approximates  $f$ .

### Expressing Data

Input data is organised into a data matrix and output data is organised into a vector.



$$y = a_1 + \sin(a_2 \cdot t)$$

$$\Delta_n = y_n - (a_1 + \sin(a_2 \cdot t_n))$$

Minimise the sum of all  $\Delta_n^2$   
w.r.t.  $a_1$  and  $a_2$

## Modelling Graphically

Sometimes data can be modelled graphically which can help us to visualise our model as we tune it. Training a model could be taking an existing equation and **tuning it to fit the test data set** by fiddling with the parameters.

Our aim is to minimise the difference  $\Delta_n^2$  between the expected output and the actual output of the model for the values which we're aware of.

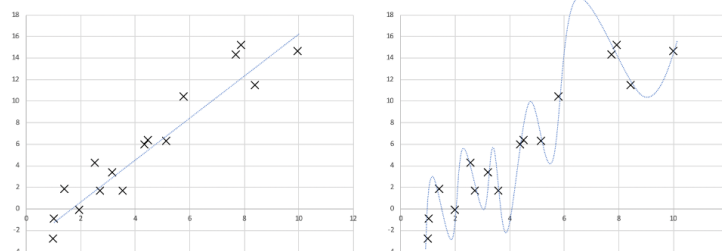
Using one function limits the number of parameters we can use. Combining functions allows us to produce stronger and more flexible models.

**Hyperparameter:** A parameter picked by hand before training begins. *E.g. the degree of a polynomial*

## Overfitting

Tuning the model too closely to the training data can result in **overfitting** which is when **increasing the complexity of the model gets better results on the training data but does so at the cost of generality.**

You can test if a model is suitable by using new observations, **not present in the training**, to see how often it produces the correct outputs.



## Nearest Neighbour

This algorithm calculates the Euclidean distance between a new point and makes a prediction based on the point which it is closest to.

We can visualise this information using a **decision boundary** which is the line of points which are exactly halfway between each class.

For a set of points,  $(x_1, x_2, \dots, x_n)$  and  $(y_1, y_2, \dots, y_n)$ , the Euclidean distance between them is:

$$\sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + \dots + (x_n - y_n)^2}$$

This algorithm **doesn't work very well with outliers**.

### K-Nearest Neighbour

Solves the outlier problem by taking the majority prediction of the  $k$  closest points.

### K-Fold Cross Validation

Recall that validating a model involves breaking the data set into training and testing data.

K-fold cross validation **splits the input data into  $k$  equal batches** and enables us to **analyse the effectiveness of models using all our data**.

$k$  rounds of training and testing are performed. In each,  $k - 1$  of the data batches are used to train the model and the remaining batch is used to test it. We take the average error rate across all  $k$  rounds.

Setting  $k = N$  means that all data except a single point is used to train the model. This is called **leave-one-out cross validation**.

Once this is complete, we can repeat training using **all the data**, giving the most accurate model possible.

## Unsupervised Learning

Develops a model and makes predictions using unlabelled input data. Unsupervised models use **clustering** for classification.

### K-Means Clustering

An **iterative** algorithm used to cluster data in unsupervised learning. To split  $n$  data points into  $k$  clusters, we follow these steps:

1. Pick  $k$  centroid points called  $c_1, c_2, \dots, c_k$ .
2. For each point in the data, find the closest centroid  $c_i$  and label this  $i$ .
3. Recalculate the centroid points to be the mean of the points with its respective label.
4. Repeat from step 2 until the labels stop changing.

Can easily be expanded to any number of dimensions to increase classification accuracy.

# Probabilistic Inference

**Probabilistic Inference:** Calculating conditional probabilities from observed evidence.

Supervised learning involves trying to find a function  $f(X) \rightarrow Y$ . We can express these using probabilities. For each possible label  $Y$ , we are looking for:

$$p(Y|X_1, X_2, \dots, X_n)$$

Using Bayes' theorem, we can express this as:

$$p(Y|X_1, X_2, \dots, X_n) = \frac{p(X_1, X_2, \dots, X_n|Y)p(Y)}{p(X_1, X_2, \dots, X_n)}$$

**Approximations** are a very useful tool when dealing with machine learning probabilities.

Making intelligent guesses requires selecting the action with the **highest utility**.

## Normalisation

We can ignore the denominator of this formula and instead multiply the result by a **normalisation constant**. When combined with marginalisation, this gives us the general inference procedure:

$$p(X|e) = \alpha * p(X, e) = \alpha * \sum_y p(X, e, y)$$

Where  $X$  is the query,  $e$  is the evidence and  $y$  is the unobserved variables.



# Naive Bayes Classifiers

## Conditional Independence

Bayes Classifiers rely on conditional independence. Three variables are conditional independent if, given  $X$ , we can write the following:

$$p(A, B, C, X) = p(A|X)p(B|X)p(C|X)p(X)$$

## The Assumption

All variables  $x_i$  are conditionally independent given some other variable  $Y$  i.e.

$$p(x_1, x_2, \dots, x_n|Y) = p(x_1|Y) * p(x_2|Y) * \dots * p(x_n|Y)$$

Each of  $x_i$  is a token. This means that we can convert Bayes Theorem into a new formula:

$$p(spam | w_1, w_2, \dots, w_n) = \frac{p(w_1 | spam) * p(w_2 | spam) * \dots * p(w_n | spam) * p(spam)}{p(w_1, w_2, \dots, w_n)}$$

This classifier is called naïve because the ordering of words doesn't matter!

## Ignoring the Denominator

The denominators will be the same for all classes. Hence, you can just compare the numerators. You don't need to know the proof for this!

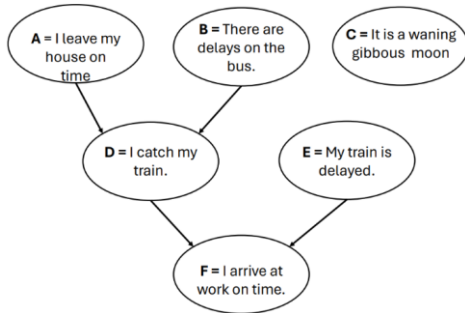
$$p(spam | w_1, w_2, \dots, w_n) = p(w_1 | spam) * \dots * p(w_n | spam) * p(spam)$$

## Laplace Smoothing

We may receive a token which hasn't been identified in the data set. We can't set this to zero because it would make the probability of any classification which contains it zero. Instead, we deal with it by applying **Laplace smoothing** to every class posterior which we calculate.

$$p(x_i | y) = \frac{Count(x_i, y) + 1}{Count(y) + |x_i|}$$

# Bayesian Belief Networks / Directed Acyclic Graphs (DAGs)



These graphs illustrate the dependence, independence and conditional independence relationships between variables.

- Every random variable is represented as a conditionally independent node unless indicated otherwise.
- Arrows indicate conditional dependence.
- Two variables not being connected means that they're independent.

## Factorising Joint Probabilities Using Graphs

We can simplify joint probabilities using the conditional dependence and independence information which DAGs tell us. For example, using Bayes Theorem we can get:

$$p(A, B, C, D, E, F) = p(A)p(B|A)p(C|A, B)p(D|A, B, C)p(E|A, B, C, D)p(F|A, B, C, D, E)$$

Which can be simplified using the information provided by the DAG:

$$p(A, B, C, D, E, F) = p(A)p(B)p(C)p(D|A, B, C)p(E)p(F|D, E)$$

## General Formula

$$p(x_1, \dots, x_n) = \prod_{i=1}^n p(x_i \mid \text{parents}(x_i))$$