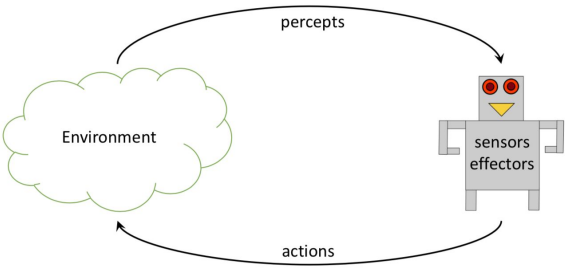
6CCS3AIN Artificial Intelligence

Lecture 1 Introduction to Artificial Intelligence

Intelligent Agents



Agent: a system that:

- Is **situated** in an environment
- Capable of **perceiving** its environment
- Capable of **acting** in its environment (which may change it)
- It has a **goal** it has to achieve.

i.e. Human agents, Software agents (Alexa, Siri), Robot agent

Ideal rational agent

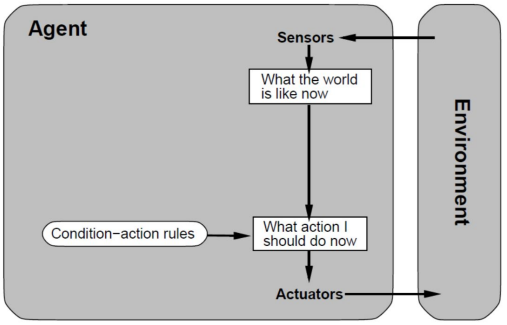
An agent that will act to maximise its expected performance measure, based on the information provided and that collected by the agents.

Being rational doesn't mean considering all decisions, some are very low probability (a door from an airplane falls and hits us). Thus a rational agent doesn't mean an *omniscient* nor *perfect* agent.

Simple Reflex Agent

maps each percept (or perception about the environment) to an action.

E.g. If temperature too hot, turn heating off.




```
function Simple-Reflex-Agent(percept) returns an action

    static: rules, a set of condition-action rules

    state ← Interpret-Input(percept)
    rule ← Rule-Match(state, rules)
    action ← rule.action
    return action
```

Environments

Environments may have the following properties:

- **Accessible:** the agent knows all the acuerate and up-to-date information about all relevant aspects of the environment.
- **Partially observable/accessible**: the agent doesn't have full information about the world.

- **Deterministic:** an environment in which any action has a **single** guaranteed effect, there is no uncertainty about the state that the environment will be in after the action. (*If you program pacman to go in that direction and it can go there, it will go)*

- **Non-deterministic:** *actions can have more than one effect.* When we cannot predict the following state of an action or the action fails to change the state for some reason.
    - In some environments we can quantify non-determinism with probabilities (i.e. there is a 0.2 chance that it will go to this state) then we call it **stochastic.**

- **Episodic:** performance of an agent is dependent on the number of discrete episodes, with no link between the performance between an agent and different episodes *(like independent events in probability)* aka one action doesn't depend on another. *(In pacman you could argue for both, you can play a game, lose it and the next game doesn't have an effect on the previous one).*
- **Non-Episodic (sequential):** what you do now affects what you might do in the future. *(Every time you make a move it's a different game, not the same).*

- **Static:** environment will only change if agent takes an action.
- **Dynamic:** an environment in which there are *other processes operating in it.* These may cause changes which are not dependent on the agent (i.e. raining, someone steps out the car, ghosts running around).

- **Discrete:** if an environment has a fixed finite number of actions and percepts in it (has a defined number of things it can sense or do [*ie go left, go right]*).
- **Continuous:** if it's not discrete (i.e turning a wheel, may be turned 90 degrees or 90.01212 degrees)

When an agent acts in an environment it generates a **run**, which is a sequence of interleaved **states** and **actions** i.e. s0, a1, s1, a2, s2, a3, s3….

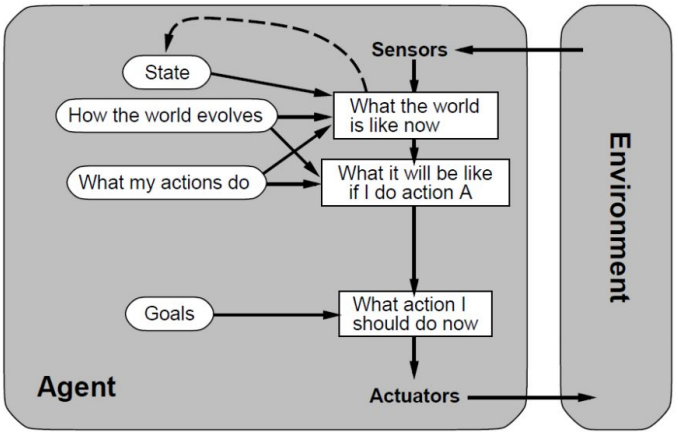
Evaluating simple agents

Say I see a car coming really fast but I turn right and then have to decide if I want to cross the road. A simple agent doesn't have memory (or state) to remember that there was a very fast car coming it only sees that the road is clear, but crossing is not a good idea. Thus we need more complex agents.

Agents with state and goals

New decisions are done taking into account information about the environment at the moment and the existing knowledge about the state.

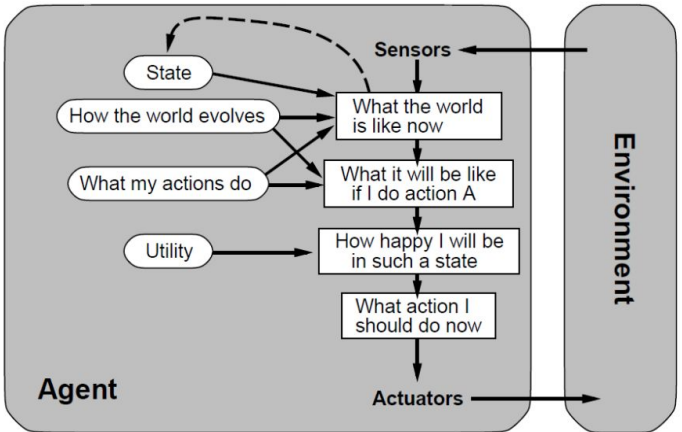
Goals may conflict or there might be very different ways of achieving goals (all of these have a different likelihood of being achieved) thus we need to classify which goals we will satisfy first and in what way.



Utility-based agents

Utility is a numerical value that tells us how "good" a state is.
If the environment is dynamic and non-deterministic we might not know what state we will end up at.

The **expected utility** is the probability that we will end up there * utility of that state



i.e.

A) 0.4 = probability of achieving goal, utility is 10, expected utility = 4
B) 0.2 = probability of achieving goal, utility is 100, expected utility = 20

An **optimal agent** will maximise the sum of the expected utilities of all the possible resulting states.

Lecture 2 Probabilistic Reasoning I

Partial Observability and Uncertainty

Full observability is not true in the real world.
Partial observability -> uncertainty. Uncertainty may raise because of sensor noise, limited computational complexity (you can't process all the data). To accommodate the uncertainty we use probability.

Example:

- Partial observability: road state or other driver plans.
- Noisey sensors: google map traffic updates.
- Uncertainty in action: flat tyre
- Immense complexity of modelling and predicting traffic (too much for my brain).

Probability

Probabilistic assertions summarize effects of

- **Laziness**: failure to enumerate exceptions, qualifications, etc
- **Ignorance**: lack of relevant facts, initial conditions, etc

Subjective or **Bayesian** probability: probability of things given what we know about the world.

Probability of leaving to the airport 25 minutes before **given** that there are no reported accidents then the probability is 0.6. Adding info like at 5 am has an impact on probability.

- Probabilities relate propositions to one's own state of knowledge

$$P(A_{25}|\text{no reported accidents}) = 0.06$$

- Probabilities of propositions change with new evidence:

$$P(A_{25}|\text{no reported accidents, 5 a.m.}) = 0.15$$

*Figuring out the probability might not give the full picture. You might not want to leave 4 days even if it guarantees that you will get the plane in time. We introduce **utility theory** to calculate this.*

Basics

- Begin with a set Ω—the sample space.
- This is all the possible things that could happen.
  - 6 possible rolls of a die.
- $\omega \in \Omega$ is a sample point, possible world, atomic event.

Omega represents an element in the sample space (e.g. drawing a 2 on a die)

- A probability space or probability model is a sample space with an assignment $P(\omega)$ for every $\omega \in \Omega$ such that:

$$0 \leqslant P(\omega) \leqslant 1$$
$$\sum_{\omega} P(\omega) = 1$$

- For a typical die:
  $P(1) = P(2) = P(3) = P(4) = P(5) = P(6) = 1/6$.

The probability of a number must be between 0 and 1 and the sum of the probabilities must be 1.

- An event A is any subset of Ω

$$P(A) = \sum_{\{\omega \in A\}} P(\omega)$$

- Again, for a regular die:
  $P(\text{die roll} < 4) = P(1) + P(2) + P(3) = 1/6 + 1/6 + 1/6 = 1/2$

An event is a number of samples.

- A random variable is a function from sample points to some range.
  - $raining(London) = true$.
  - $temperature(S7.14) = 27$
- P induces a probability distribution for any r.v. X:

$$P(X = x_i) = \sum_{\{\omega : X(\omega) = x_i\}} P(\omega)$$

The probability distribution is the set of probabilities of all the different values a random variable can have.

- Continuing our example:

$$\begin{aligned} P(Odd = true) &= P(1) + P(3) + P(5) \\ &= 1/6 + 1/6 + 1/6 \\ &= 1/2 \end{aligned}$$

Propositions

Propositions map to a sample space in the world. Probability of events is given by:

$$P(a \vee b) = P(a) + P(b) - P(a \wedge b)$$

Syntax for propositions

Propositional or Boolean random variables

- Variable with **capital letter** is a variable for which we do not know the value yet.
- When a variable is true it will be written in lowercase, false ones will have a line over or a negation symbol.
    - Cavity - variable which may have a range of values {true, false}

  We write:
    - cavity if cavity = true.
    - ¬cavity if cavity = false

Discrete random variables (discrete = an established range of values)

- Weather is one of <sunny, rain, cloudy, snow>
- Weather = rain is a proposition (its mapping it to the sample space)
  Values must be exhaustive and mutually exclusive.

Continuous random variables (bounded or unbounded)

- Temp = 21.6 also allows for Temp < 22.0

Prior or unconditional values of propositions

- P(Cavity = true) = 0.1 and P(Weather = sunny) = 0.72
- correspond to belief before (prior) to arrival of any (new) evidence.

Probability distribution gives values for all possible assignments:

- **P**(Weather) = < 0.72, 0.1, 0.08, 0.1 >
  Distribution is normalized, i.e., sums to 1

NOTE: *Probability with a bold capital* **P**, means that it wants a range of probabilities. Note that this must be formalized and must add up to 1. Example bellow:

$$
\begin{aligned}
\mathbf{P}(Toothache) &= \langle P(toothache), P(\neg toothache) \rangle \\
P(toothache) &= P(cavity \wedge catch \wedge toothache) + P(cavity \wedge \neg catch \wedge toothache) \\
&\quad + P(\neg cavity \wedge catch \wedge toothache) + P(\neg cavity \wedge \neg catch \wedge toothache) \\
&= 0.108 + 0.012 + 0.016 + 0.064 \\
&= 0.2 \\
\mathbf{P}(Toothache) &= \langle 0.2, 0.8 \rangle
\end{aligned}
$$

The second probability is computed either in the same way as the first, or by knowing that the two sum to 1.

Joint Probability Distribution for a set of random variables gives the probability of every atomic event (every sample point). We can answer every question about a domain by combining different sample points of the table.

$\mathbf{P}(Weather, Cavity)$ = a 4×2 matrix of values

$Weather =$	*sunny*	*rain*	*cloudy*	*snow*
$Cavity = true$	0.144	0.02	0.016	0.02
$Cavity = false$	0.576	0.08	0.064	0.08

Inference by Enumeration

Choose the ones that satisfy your proposition and add them up.

	toothache		¬ *toothache*	
	catch	¬ *catch*	*catch*	¬ *catch*
cavity	**.108**	**.012**	**.072**	**.008**
¬ *cavity*	**.016**	**.064**	**.144**	**.576**

- For any proposition ϕ, sum the atomic events where it is true:

$$P(\phi) = \sum_{\omega:\omega \models \phi} P(\omega)$$

$$\begin{aligned} P(toothache) &= 0.108 + 0.012 + 0.016 + 0.064 \\ &= 0.2 \end{aligned}$$

Conditional Probability

Conditional or posterior probabilities

$P(cavity|toothache) = 0.8$

given that *toothache* is all I know

NOT "if *toothache* then 80% chance of *cavity*"

Notation for conditional distributions:

$\mathbf{P}(Cavity|Toothache)$

A 2-element vector of 2-element vectors.

The distribution would be: cavity given toothache, cavity given not toothache, not cavity given toothache and not cavity given not toothache.

If the proposition is given in the condition:

$P(cavity|toothache, cavity) = 1$

Some conditions may have no impact on the rest of the probability since it might be irrelevant. We infer this by understanding how the world works.

$P(cavity|toothache, soxWin) = P(cavity|toothache) = 0.8$

Conditional Probability Formula

$$P(a|b) = \frac{P(a \wedge b)}{P(b)} \text{ if } P(b) \neq 0$$

Tip: the condition is the one that's on the bottom part.

Product Rule Probability Formula

Product rule gives an alternative formulation:

$$P(a \wedge b) = P(a|b)P(b) = P(b|a)P(a)$$

Tip: we always multiply by the condition, no matter the order of the conjunction.

We can talk about the product rule holding for an entire distribution. We can compute it in the following way:

- A general version holds for whole distributions,

$$\mathbf{P}(Weather, Cavity) = \mathbf{P}(Weather|Cavity)\mathbf{P}(Cavity)$$

(View as a 4×2 set of equations, not matrix multiplication)

Chain Rule Probability Formula

Applying the product rule multiple times (normally for equations with more than two literals).

- Chain rule is derived by successive application of product rule:

$$\begin{aligned}
\mathbf{P}(X_1, \ldots, X_n) &= \mathbf{P}(X_1, \ldots, X_{n-1})\mathbf{P}(X_n|X_1, \ldots, X_{n-1}) \\
&= \mathbf{P}(X_1, \ldots, X_{n-2})\ \mathbf{P}(X_{n-1}|X_1, \ldots, X_{n-2}) \\
&\qquad \mathbf{P}(X_n|X_1, \ldots, X_{n-1}) \\
&= \ldots \\
&= \prod_{i=1}^{n} \mathbf{P}(X_i|X_1, \ldots, X_{i-1})
\end{aligned}$$

The probability distribution of the first n-1 random variables (r.v.) (then we will continue to expand this) multiplied by the probability of what we are looking for (the last rv) given the other r.vs.

This is a simple recursive calculation to calculate the probability distribution.

- Or, in terms of a more concrete example:

$$\begin{aligned} P(a,b,c) &= P(a,b)P(c|b,a) \\ &= P(a)P(b|a)P(c|b,a) \end{aligned}$$

Probability of (a,b) which we will further expand. Times the probability of c given b,a.

Inference by enumeration

We can also use inference by enumerations with conditional probability.

- Can also compute conditional probabilities:

$$\begin{aligned} P(\neg cavity|toothache) &= \frac{P(\neg cavity \wedge toothache)}{P(toothache)} \\ &= \frac{0.016 + 0.064}{0.108 + 0.012 + 0.016 + 0.064} \\ &= 0.4 \end{aligned}$$

	toothache		¬ *toothache*	
	catch	¬ *catch*	*catch*	¬ *catch*
cavity	**.108**	**.012**	**.072**	**.008**
¬ *cavity*	**.016**	**.064**	**.144**	**.576**

Tip: look at the denominator of the function. This denominator is simply normalising everything, dividing everything by the same thing. We can use this to calculate the normalised values without knowing P(toothache).

Normalization

	toothache		¬ *toothache*	
	catch	¬ *catch*	*catch*	¬ *catch*
cavity	**.108**	**.012**	**.072**	**.008**
¬ *cavity*	**.016**	**.064**	**.144**	**.576**

- Denominator can be viewed as a normalization constant α

$$\begin{aligned}\mathbf{P}(Cavity|toothache) &= \alpha\,\mathbf{P}(Cavity, toothache)\\ &= \alpha\,[\mathbf{P}(Cavity, toothache, catch) + \mathbf{P}(Cavity, toothache, \neg catch)]\\ &= \alpha\,[\langle 0.108, 0.016\rangle + \langle 0.012, 0.064\rangle]\\ &= \alpha\,\langle 0.12, 0.08\rangle = \langle 0.6, 0.4\rangle\end{aligned}$$

We don't have tom compute the probability of P(toothache) since we know that we are going to get the same normalised outcome.

Inference by Enumeration difficulties

If we have all the probability distributions, we can calculate the probability of whatever we want but we have some problems:

- Worst-case time complexity O(d^n) where d is the largest arity.
- Space complexity O(d^n) to store the joint distribution.
- How to find all the values for the probability table???

To be able to compute efficiency probabilistic computations we need:

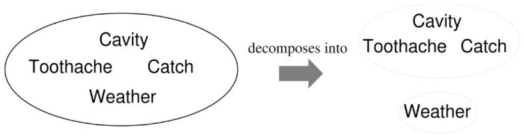
1. **Conditional independence**
2. **Bayes rule**

Independence

A and *B* are independent iff

$$\begin{aligned}\mathbf{P}(A|B) &= \mathbf{P}(A), \text{or}\\ \mathbf{P}(B|A) &= \mathbf{P}(B), \text{or}\\ \mathbf{P}(A, B) &= \mathbf{P}(A)\mathbf{P}(B)\end{aligned}$$

Independence is important since recognizing it makes the sample space much smaller.



- $\mathbf{P}(Toothache, Catch, Cavity, Weather)$
  $= \mathbf{P}(Toothache, Catch, Cavity)\mathbf{P}(Weather)$
- 31 values reduced to 10.
- For n independent biased coins, $2^n \rightarrow n$

In this example recognizing that weather is not relevant helps us.
Note: it's 31 because you can infer the 32nd since they all have to add up to 1. Thus you only have to compute 31.

Conditional Independence

Absolute independence is powerful but extremely rare thus we use conditional independence to make our probability tables smaller.

Example: increasing computational efficiency

- $\mathbf{P}(Toothache, Cavity, Catch)$ has:
- Three binary variables.
- Thus 2^3 entries in the joint probability table.
- But these sum to 1.
- So $2^3 - 1$ independent entries
- That's 7 independent entries
- If I have a cavity, the probability that the probe catches in it doesn't depend on whether I have a toothache:

$$P(catch|toothache, cavity) = P(catch|cavity) \quad (1)$$

- The same independence holds if I haven't got a cavity:

$$P(catch|toothache, \neg cavity) = P(catch|\neg cavity) \quad (2)$$

- *Catch* is conditionally independent of *Toothache* given *Cavity*

$$\mathbf{P}(Catch|Toothache, Cavity) = \mathbf{P}(Catch|Cavity)$$

We have now reduced the number of entries.

- Equivalent statements:

$$\begin{aligned}\mathbf{P}(Toothache|Catch, Cavity) &= \mathbf{P}(Toothache|Cavity)\\ \mathbf{P}(Toothache, Catch|Cavity) &= \mathbf{P}(Toothache|Cavity)\\ &\quad \times \mathbf{P}(Catch|Cavity)\end{aligned}$$

All the above are equivalent, thus we don't need to compute them.

- Write out full joint distribution using chain rule:

$$\begin{aligned}&\mathbf{P}(Toothache, Catch, Cavity)\\&= \mathbf{P}(Toothache|Catch, Cavity)\mathbf{P}(Catch, Cavity)\\&= \mathbf{P}(Toothache|Catch, Cavity)\mathbf{P}(Catch|Cavity)\mathbf{P}(Cavity)\\&= \mathbf{P}(Toothache|Cavity)\mathbf{P}(Catch|Cavity)\mathbf{P}(Cavity)\end{aligned}$$

- 2 + 2 + 1 = 5 independent numbers
- Equations 1 and 2 remove 2.

From needing 31 we now now 5, the drop of numbers would be more significant if there were more variables and more values for each variable.

Conclusion

In most cases conditional independence reduces the size of the representation of the joint distribution from exponential in n to (close to) linear in n thus more computationally efficient.

Bayes' Rule

Is a rewriting of the product rule

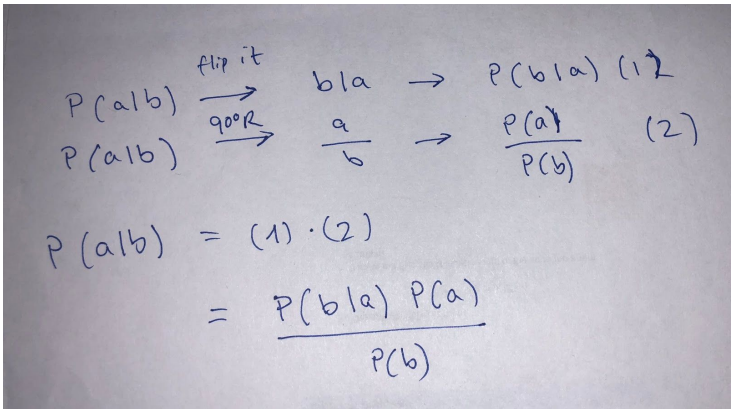
$$\text{Product rule } P(a \wedge b) = P(a|b)P(b) = P(b|a)P(a)$$

$$\Rightarrow \text{ Bayes' rule} P(a|b) = \frac{P(b|a)P(a)}{P(b)}$$

or in distribution form

$$\mathbf{P}(Y|X) = \frac{\mathbf{P}(X|Y)\mathbf{P}(Y)}{\mathbf{P}(X)} = \alpha\mathbf{P}(X|Y)\mathbf{P}(Y)$$

Tip to remember: *in Bayes we take our condition and flip it, then we multiply it by the original condition moving it 90 degrees to the right.*



- Probability that some hypothesis is true, given that some event has occured.

$$\mathbf{P}(H|E) = \frac{\mathbf{P}(E|H)\mathbf{P}(H)}{\mathbf{P}(E)}$$

- Now, since:

$$P(e) = P(e|h)P(h) + P(e|\neg h)P(\neg h)$$

we can rewrite this in a form in which it is commonly applied:

$$P(h|e) = \frac{P(e|h)P(h)}{P(e|h)P(h) + P(e|\neg h)(1 - P(h))}$$

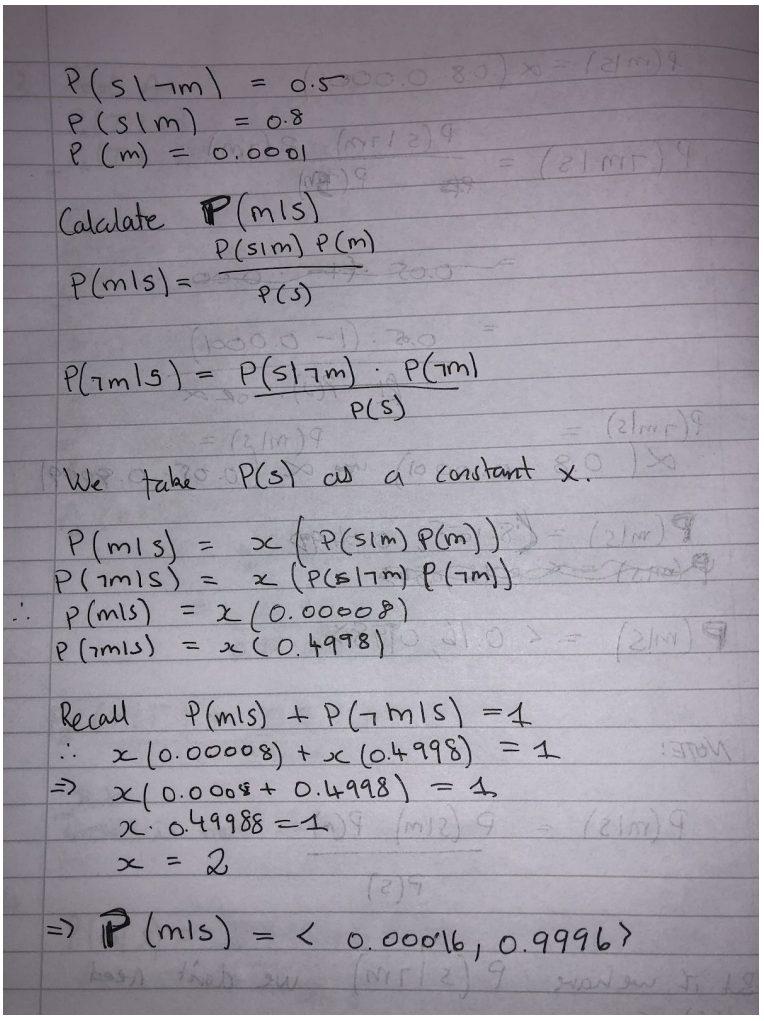
where variables are binary valued.

The output of the equation depends on the probability of the hypothesis. If it is high it will yield a higher result.

When you re-apply bayes' theorem multiple times in a row, we use the result of the previous output for our probability of the hypotheses P(h). Thus when things occur multiple times the probability of the hypothesis increases.

Calculating two values without the probability of a single event

In this example we take P(s) as a constant since we know that the two probabilities must sum to 1.



Lecture 3 Probabilistic Reasoning II

Conditional independence: the fact that some values don't affect other values. Ie the probability of catch given that you have a cavity is not affected by the fact you have a toothache.

Bayes' Rule & Conditional Independence

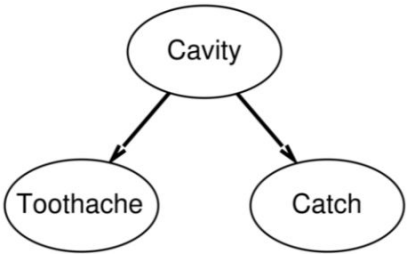
- So, in our running dentist example

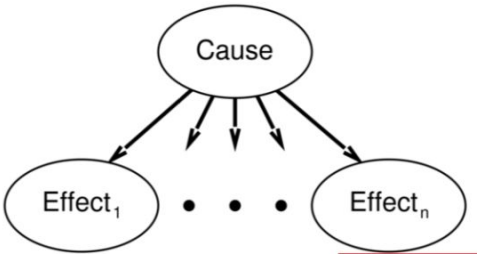
$$\begin{aligned}&\mathbf{P}(Cavity|toothache \wedge catch)\\ &= \alpha\,\mathbf{P}(toothache \wedge catch|Cavity)\mathbf{P}(Cavity)\\ &= \alpha\,\mathbf{P}(toothache|Cavity)\mathbf{P}(catch|Cavity)\mathbf{P}(Cavity)\\ &= \alpha\,\mathbf{P}(Cavity)\mathbf{P}(toothache|Cavity)\mathbf{P}(catch|Cavity)\end{aligned}$$

- This is an example of a naive Bayes model:

$$\mathbf{P}(Cause, Effect_1, \ldots, Effect_n) = \mathbf{P}(Cause)\prod_i \mathbf{P}(Effect_i|Cause)$$

- Visualise as:





With this method, the number of variables is linear (not 2^n).

Bayesian Networks can be represented with a graph in which:

- a set of nodes, one per variable
- a directed, acyclic graph (link ≈ "directly influences") a conditional distribution for each node given its parents

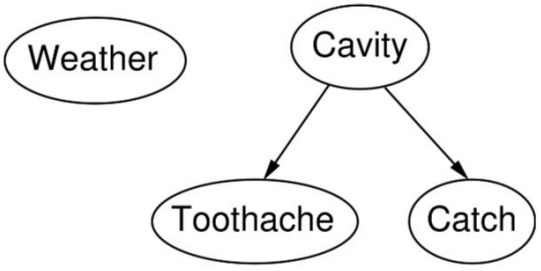
$$\mathbf{P}(X_i|Parents(X_i))$$

Graphs must be **acyclic** meaning they cannot have loops in it.

We can show conditional distributed in a **Conditional Probability Table (CPT)** which gives the distribution of a node given its parent nodes.

Showing conditional independence in a graph

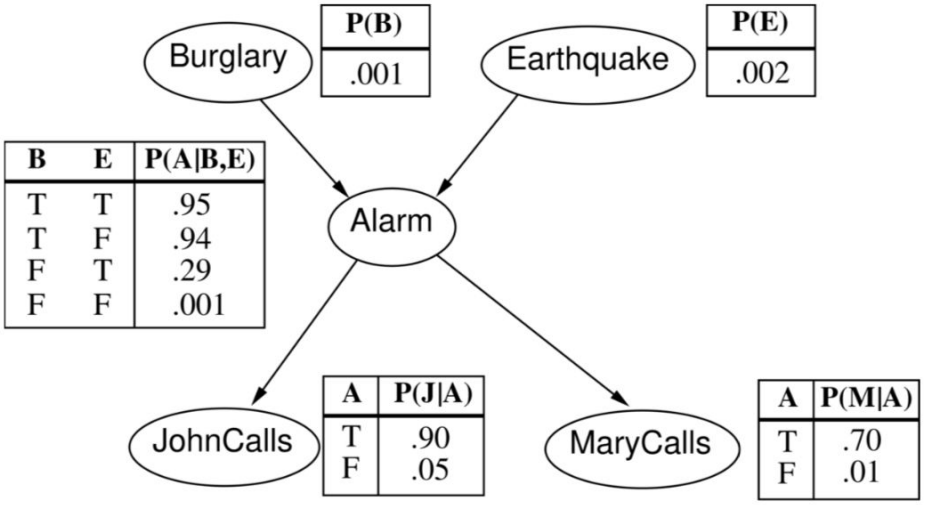
- Topology of network encodes conditional independence assertions:



- *Weather* is independent of the other variables
- *Toothache* and *Catch* are conditionally independent given *Cavity*

Example of a Bayesian network

I'm at work, neighbor John calls to say my alarm is ringing, but neighbor Mary doesn't call. Sometimes it's set off by minor earthquakes. Is there a burglar?



Probability is dependent of the conditional probabilities of their parents. Hence, we have to take B and E in consideration when calculating the probability of an alarm.

This conditional probability table (CPT) seems to be missing some values since it only has two values rather than four to specify the relation between J and A. This is because that info is not relevant given the example.

- The table tells us that:

$$P(J = T|A = T) = 0.9$$

which means:

$$P(J = F|A = T) = 0.1$$

because $P(J = T|A = T) + P(J = F|A = T) = 1$

Or, writing the values of J and A the other way:

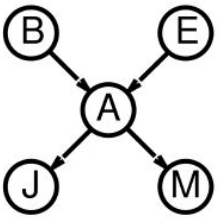
$$P(j|a) = 0.9$$
$$P(\neg j|a) = 0.1$$

because $P(j|a) + P(j|\neg a) = 1$

Global Semantics

The joint probability of all variables is given by the product of the variable given its parents.

$$P(x_1, \ldots, x_n) = \prod_{i=1}^{n} P(x_i|parents(X_i))$$



The above syntax which looks like a Stool, means the multiplication of, just like sigma but with multiplication.

- $P(j \wedge m \wedge a \wedge \neg b \wedge \neg e)$

To develop the probability of the above, we find the probability of each parameter given their parents. *E.g. for **a**, we would find the P(**a** | notB, notE) since we see that b and e are not true given the next to parameters after e. We have to look at the diagram above to the right to know which ones to choose.*

$$
\begin{aligned}
&= P(j|a)P(m|a)P(a|\neg b, \neg e)P(\neg b)P(\neg e) \\
&= 0.9 \times 0.7 \times 0.001 \times 0.999 \times 0.998 \\
&\approx 0.00063
\end{aligned}
$$

Using this method is much more efficient than computing the whole CPT. Since we specify the probability for each node, the probability of not that node comes for free so it's more computationally efficient.

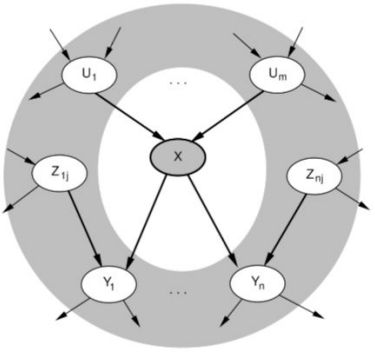
Local Semantics

Each node is conditionally independent of its non descendants (non-descendants are those which aren't children or children's children) given its parents. So you should be able to do calculations locally (of the current node and its parents) and it will give you the same result as doing it globally for all nodes.

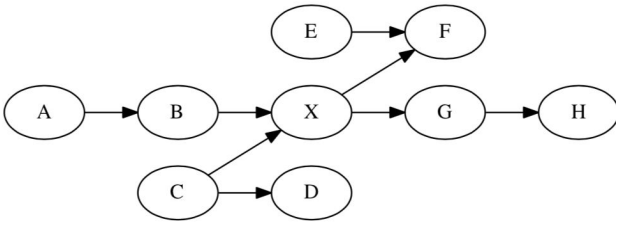
Markov Blanket

Each node is conditionally independent of all others given its **Markov blanket:** parents + children + children's parents

In the case to the right: all Z, Y and U.



The Markov blanket is: B,C, E, F, G

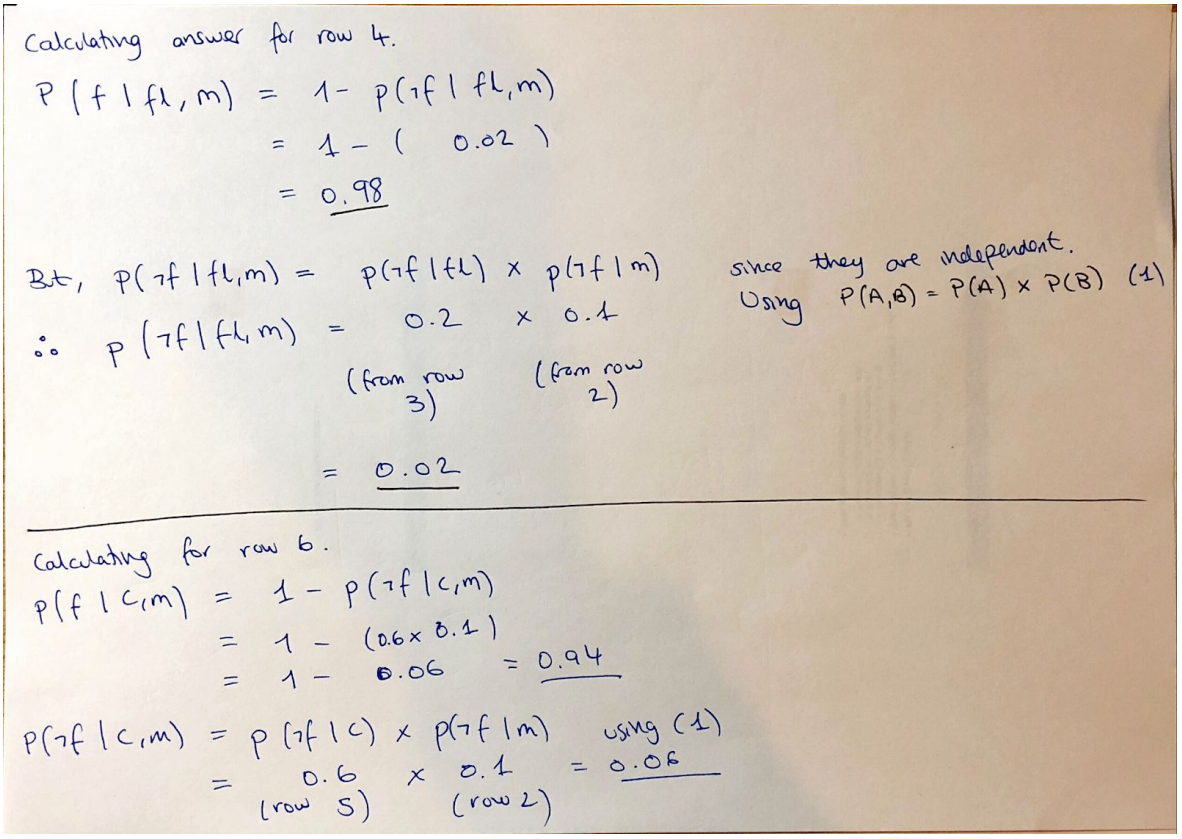


We still have the issue that CPT grows exponentially with the number of parents. To reduce this nodes might have a deterministic case or we might use a noisy or.

Compact conditional distribution (Noisy-OR)

Is a technique to infer probabilities. Like so:

Cold	*Flu*	*Malaria*	*P*(*Fever*)	*P*(¬*Fever*)
F	F	F	**0.0**	1.0
F	F	T	0.9	**0.1**
F	T	F	0.8	**0.2**
F	T	T	0.98	$0.02 = 0.2 \times 0.1$
T	F	F	0.4	**0.6**
T	F	T	0.94	$0.06 = 0.6 \times 0.1$
T	T	F	0.88	$0.12 = 0.6 \times 0.2$
T	T	T	0.988	$0.012 = 0.6 \times 0.2 \times 0.1$



This method is much more computationally efficient since we only need one value for each parent (**linear**). Using only 0.6, 0.1 and 0.2 we can deduce all the table.
Note: if we know all the causes of the effect, we know that the probability of an effect given the causes NOT happening is 0. P(f | ¬i , ¬l) = 0 given that i and l are the only causes and that they are ***non-interacting causes***

Inference Tasks

What things we can infer using a Bayesian network. We will focus on simple queries for now.

Simple queries: compute posterior marginal $\mathbf{P}(X_i|\mathbf{E}=\mathbf{e})$

$P(NoGas|Gauge=empty, Lights=on, Starts=false)$

Conjunctive queries

$\mathbf{P}(X_i, X_j|\mathbf{E}=\mathbf{e}) = \mathbf{P}(X_i|\mathbf{E}=\mathbf{e})\mathbf{P}(X_j|X_i, \mathbf{E}=\mathbf{e})$

Optimal decisions: decision networks include utility information; probabilistic inference required for $P(outcome|action, evidence)$

Value of information: which evidence to seek next?

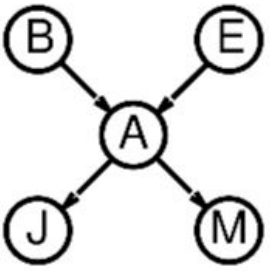
Sensitivity analysis: which probability values are most critical?

Explanation: why do I need a new starter motor?

Inference by Enumeration

- Simple query on the burglary network.

$$\mathbf{P}(B|j,m) = \frac{\mathbf{P}(B,j,m)}{P(j,m)}$$
$$= \alpha\mathbf{P}(B,j,m)$$
$$= \alpha \sum_e \sum_a \mathbf{P}(B,e,a,j,m)$$



In this case we have decided to sum out e and a because since we want the prob distribution we have to take into account all the values for the distribution. We expand the formula just as we did before multiplying all parameters like so: p(parameter | parents). Note that B and E don't have parents thus we can just say p(B) and p(e).

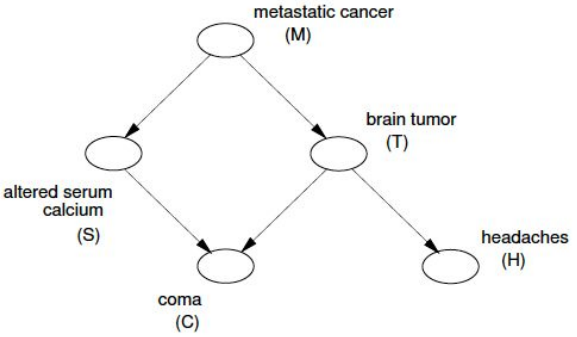
Tip: when we have variables that aren't part of the query we have to look at all the values and them add them up.

Rule for number of probabilities needed for a network: if no arrows in 1, else 2^number of arrows. All the nodes summed -1. ***If there is no network then we need 2^nodes - 1.***

- Rewrite full joint entries taking network into account:

$$\mathbf{P}(B|j,m) = \alpha \sum_e \sum_a \mathbf{P}(B)P(e)\mathbf{P}(a|B,e)P(j|a)P(m|a)$$
$$= \alpha \mathbf{P}(B) \sum_e P(e) \sum_a \mathbf{P}(a|B,e)P(j|a)P(m|a)$$

Full Example



$P(M)$
0.1

M	$P(S \mid M)$
T	0.8
F	0.2

M	$P(T \mid M)$
T	0.7
F	0.1

S	T	$P(C \mid S,T)$
T	T	0.95
T	F	0.85
F	T	0.85
F	F	0.01

T	$P(H \mid T)$
T	0.9
F	0.7

Figure 1: The example Bayesian network.

Calculate **P**(M | h,s)
Note: we want to find the distribution of M (thus we will need the scenario where m is true and when its false)
Out evidence variables are c and t. We need to account for all combination of these.

$$\mathbf{P}(M|h,s) = \frac{\mathbf{P}(M,h,s)}{P(h,s)}$$
$$= \alpha \mathbf{P}(M,h,s)$$
$$= \alpha \sum_t \sum_c /pv(M,h,s,t,c)$$

Factorising as in Q4, we then have:

$$\mathbf{P}(M|h,s) = \alpha \sum_t \sum_c P(h|t).P(c|s,t).P(t|M).P(s|M).\mathbf{P}(M)$$
$$= \alpha \mathbf{P}(M).P(s|M) \sum_t \sum_c P(h|t).P(c|s,t).P(t|M)$$
$$= \alpha \Big\langle P(m).P(s|m) \sum_t \sum_c P(h|t).P(c|s,t).P(t|m),$$
$$P(\neg m).P(s|\neg m) \sum_t \sum_c P(h|t).P(c|s,t).P(t|\neg m) \Big\rangle$$

Let's say that:

$$\begin{aligned} pm &= P(m).P(s|m)\sum_t\sum_c P(h|t).P(c|s,t).P(t|m) \\ &= P(m).P(s|m)\,(P(h|t).P(t|m).P(c|s,t) + P(h|t).P(t|m).P(\neg c|s,t) \\ &\qquad\qquad P(h|\neg t).P(\neg t|m).P(c|s,\neg t) + P(h|\neg t).P(\neg t|m).P(\neg c|s,\neg t)) \\ &= 0.1 \times 0.8(0.9 \times 0.7 \times 0.95 + 0.9 \times 0.7 \times 0.05 + 0.7 \times 0.3 \times 0.85 + 0.7 \times 0.3 \times 0.15) \\ &= 0.0672 \end{aligned}$$

Similarly, let:

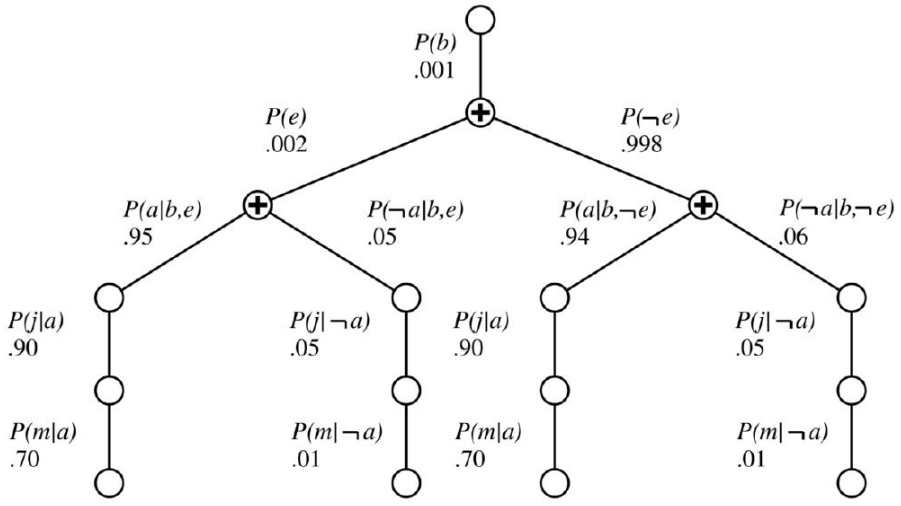
$$\begin{aligned} pm' &= P(\neg m).P(s|\neg m)\sum_t\sum_c P(h|t).P(c|s,t).P(t|\neg m) \\ &= 0.9 \times 0.2(0.9 \times 0.1 \times 0.95 + 0.9 \times 0.1 \times 0.05 + 0.7 \times 0.9 \times 0.85 + 0.7 \times 0.9 \times 0.15) \\ &= 0.1296 \end{aligned}$$

Now:

$$\begin{aligned} \mathbf{P}(M|h,s) &= \alpha\langle pm, pm'\rangle \\ &= \alpha\langle 0.0672, 0.1296\rangle \\ &= \langle 0.34, 0.66\rangle \end{aligned}$$

Evaluation Tree

Inference by enumeration works but not very efficient as we can see the bottom parts of the evaluation tree are the same and those are things we are computing.



$$\mathbf{P}(B|j,m) = \alpha\mathbf{P}(B)\ \textstyle\sum_e\ P(e)\ \sum_a\ \mathbf{P}(a|B,e)P(j|a)P(m|a)$$

Inefficient: computes $P(j|a)P(m|a)$ for each value of e

It can be improved by

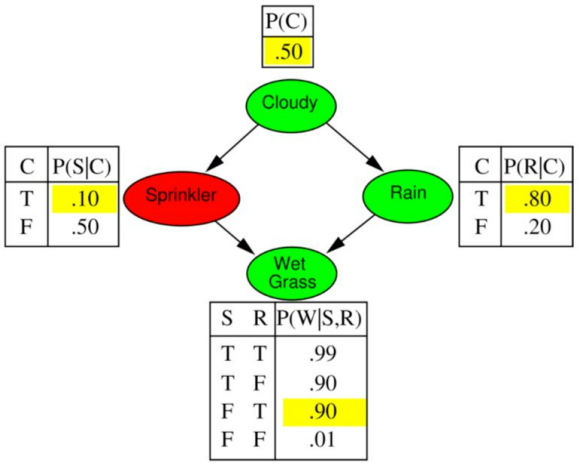
- **Variable elimination** evaluates tree bottom up, remembering intermediate values

- **Clustering algorithms** can be fore efficient, they group variables together strategically

Inference by Stochastic Simulation

Prior sampling

Given the probability of each event we make a run through the network. Say this:



WetGrass = true

This yields [true,false,true,true]. We can run the whole system again and see what other output we get. The proportion that we get the exact same run as above out of the whole sample set will be:

$$\mathbf{P}(c, \neg s, r, w)$$

The more runs, the more accurate the probability.

To replicate this we need a random number generator. To determine if in our run cloudy is true or not we will say: it will only be true is my number is under 50% of the range of values. For a range [0,1] if we get 0.1 we will say its true, else we will say is false. Moving on to the next one (sprinkler) given that cloudy is true we will have to get a number between 0 and 0.10 to say that sprinkler is true and so on.

This is inefficient because it means we have to calculate a lot of joint probabilities and then sum out the ones you don't want to find the probability of your event. Rejection sampling does it better.

Rejection Sampling

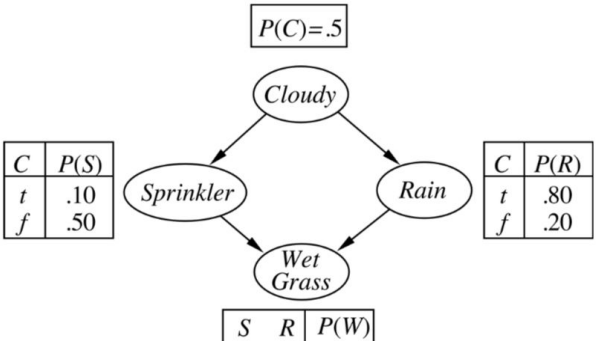
We stop the computation as long as we realize that this sample is not important.
This is better than prior sampling but if the condition we are probing is at the bottom, we have to go down to the end to see that our sample is no good. This is more efficient than prior sampling but for unlikely events we may have to wait a long time to get enough matching samples.

Likelihood Weighting

This is a version of importance sampling in which we fix the evidence variable to be true and then sample for that. We set things we want to be true to be true and then we compute how reliable this is. We will then put a weight on that sample to tell us how likely that outcome is.

Note an evidence variable is one in which we don't know if it is true or false in the query.

- Consider we have the following network:



- We want $\mathbf{P}(Rain|Cloudy = true, WetGrass = true)$
- We pick a variable ordering, say:

  $Cloudy, Sprinkler, Rain, WetGrass.$

  as before.
- Set the weight $w = 1$ and we start.
- Deal with each variable in order.

Note that the order depends on the topology of the network (cloudy is first because it has no parents), order is important.

  - *Cloudy* is true, so:

$$w \leftarrow w \times P(Cloudy = true)$$
$$w \leftarrow 0.5$$

  - $Cloudy = true$, $Sprinkler =?$, $Rain =?$, $WetGrass =?$.
  - $w = 0.5$

Sprinkler is not an evidence variable thus we have to sample it.

  - *Sprinkler* is not an evidence variable, so we don't know whether it is true or false.
  - Sample a value just as we did for prior sampling:

$$\mathbf{P}(Sprinkler|Cloudy = true) = \langle 0.1, 0.9 \rangle$$

  - Let's assume this returns *false*.
  - w remains the same.
  - $Cloudy = true$, $Sprinkler = false$, $Rain =?$, $WetGrass =?$.
  - $w = 0.5$

Same for rain, we assume it is true, w remains the same. *(Remember that we test the probability of a parameter given its parents!)*

- *WetGrass* is an evidence variable with value *true*, so we set:

$$w \leftarrow w \times P(WetGrass = true | Sprinkler = false, Rain = true)$$
$$w \leftarrow 0.45$$

- $Cloudy = true$, $Sprinkler = false$, $Rain = true$, $WetGrass = true$.
- $w = 0.45$

The P(WetGr…) just comes from the CPT and it's 0.9. Therefore 0.5 * 0.9 = 0.45

- So we end with the event $[true, false, true, true]$ and weight 0.45.
- To find a probability we tally up all the relevant events, weighted with their weights.
- The one we just calculated would tally under

$$Rain = true$$

- As before, more samples means more accuracy.

Gibbs Sampling

Part of the Markov Chain Monte Carlo (MCM) algorithms. They generate samples by making random change to the previous sample.

- Gibbs sampling for Bayesian networks starts with an arbitrary state.
- So pick state with evidence variables fixed at observed values (If we know $Cloudy = true$, we pick that.)
- Generate next state by randomly sampling from a non-evidence variable.
- Do this sampling conditional on the current values of the Markov blanket.

Before we walked through the network in a specific order so we could look at the CPT in this case we look at its Markov Blanket.

- "The algorithm therefore wanders randomly around the state space . . . flipping one variable at a time, but keeping the evidence variables fixed".

You then count up how many times you were at each state and then you have the probability.

Example

- Consider the query:

$$\mathbf{P}(Cloudy | Sprinker = true, WetGrass = true)$$

- The evidence variables are fixed to their observed values.
- The non-evidence variables are initialised randomly.

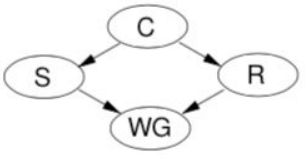
$$Cloudy = true$$
$$Rain = false$$

- State is thus:

$$[Cloudy = true, Sprinkler = true,$$
$$Rain = false, WetGrass = true].$$

To get values for Sprinkler and WetGrass we can just sample (50% true or false)

- First we sample *Cloudy* given the current state of its Markov blanket.
- Markov blanket is *Sprinkler* and *Rain*.
- So, sample from:



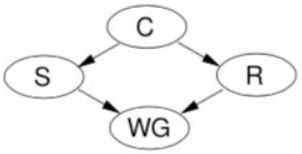
$$\mathbf{P}(Cloudy | Sprinkler = true, Rain = false)$$

- Suppose we get $Cloudy = false$, then new state is:

$$[Cloudy = false, Sprinkler = true,$$
$$Rain = false, WetGrass = true].$$

Markov Blanket = children + parents + children's other parents.

- Next we sample *Rain* given the current state of its Markov blanket.
- Marko*Cloudy*, *Sprinkler* and *WetGrass*.
- So, sample from:



$$\mathbf{P}(Rain|Cloudy = false, Sprinkler = true, WetGrass = false)$$

- Suppose we get $Rain = true$, then new state is:

$$[Cloudy = false, Sprinkler = true, Rain = true, WetGrass = true].$$

- Each state visited during this process contributes to our estimate for:

$$\mathbf{P}(Cloudy|Sprinkler = true, WetGrass = true)$$

- Say the process visits 80 states.
- In 20, $Cloudy = true$
- In 60, $Cloudy = false$
- Then

$$\begin{aligned}\mathbf{P}(Cloudy|Sprinkler = true, WetGrass = true) &= Normalize(\langle 20, 60\rangle) \\ &= \langle 0.25, 0.75\rangle\end{aligned}$$

- All of this begs the question:

  "How do we sample a variable given the state of its Markov blanket?"
- For a value x of a variable X:

$$\mathbf{P}(X|mb(X)) = \alpha\mathbf{P}(X|parents(X)) \prod_{Y \in Children(X)} \mathbf{P}(Y|parents(Y))$$

  where $mb(X)$ is the Markov blanket of X.
- Given $\mathbf{P}(X|mb(X))$, we can sample from it just as we have before.

The conditional probability of the thing you care about given its parents times each of its children given its parents.

Lecture 4 Sequential Decision Making

Analyzing Gambles

To analyze which of two betting games are better we need to analyze the **expected value** of the bet.

We roll a die and if the number is odd you pay 2£, if an even number appears you win 3£. Is it a good game?

We do this in terms of a random variable, which we will call X.

X can take two values:

3 if the die rolls odd
-2 if the die rolls even

And we can also calculate the probability of these two values

$\Pr(X = 3) = 0.5$
$\Pr(X = -2) = 0.5$

The **EV** is the sum of (all the outcomes times the probability of the outcome)

E(X) = prob of value * value

$$E(X) = \sum_k k \Pr(X = k)$$

where the summation is over all values of k for which $\Pr(X = k) \neq 0$.

Here the expected value is:

$$E(X) = 0.5 \times 3 + 0.5 \times -2$$

Thus the expected value of X is £0.5, and we take this to be the value of the bet.

How to decide what to do

The expected value of a “bet” is the sum of the all the (resulting values * prob of that value)

E(X) = R1 * P1 + R2 * P2 + Rn* Pn…
E(X) is not the value you are going to get, but the average value if you were to run the same bet again and again. There is a distribution of wins and losses but in the long term you will make 0.5£ per run with the game above.

How agents decide what to do

- Consider an agent with a set of possible actions A.
- Each $a \in A$ has a set of possible outcomes s_a.

A rational agent should choose an action that maximises the agent’s utility.

$$a^* = arg \max_{a \in A} u(s_a)$$

Look at the maximum over all the actions and pick the a that makes that maximum.

a*: means the action with the highest utility.
Arg max = returns the argument that yields the maximum value not just the maximum value.

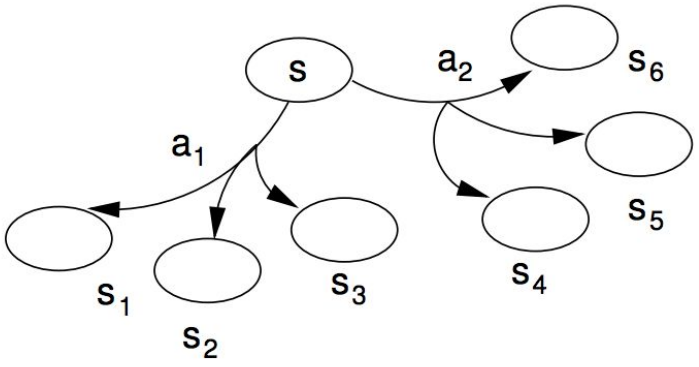
The problem is that in a realistic situation we don't know which s(a) will result from a given action (non-deterministic world) so we can only calculate the expected utility.

To calculate the expected utility, we will have to think of all the states that action might lead us to, calculate the expected utility of those and add them together and pick the best.

- In other words, for each action a with a set of outcomes s_a, the agent should calculate:

$$E(u(a)) = \sum_{s' \in s_a} u(s').\Pr(s_a = s')$$

and pick the best.



We calculate: how likely is it that we get to S1 * the utility of S1 plus the same for S2, S3 and so on. We do this for a1 and for a2 and then see which one gives you a better utility.

Other notions of "rational"

There are other criteria for decision-making than maximising expected utility (they represent other mindsets):

Maximin: look at the option which has the least-bad worst outcome (risk adverse)

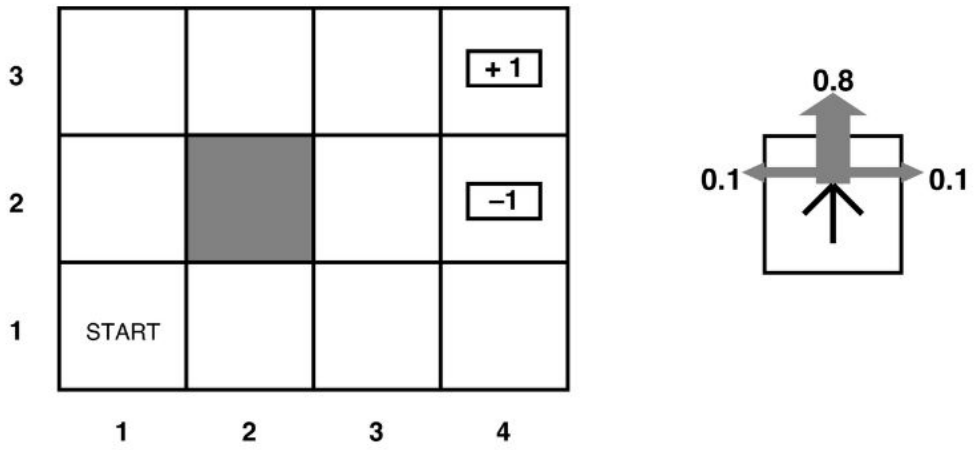
$$a^* = \arg\max_{a \in A} \{\min_{s' \in s_a} u(s')\}$$

Maximax: ignore possible bad outcomes and just focus on the best outcome of each action (risk seeking)

$$a^* = \arg\max_{a \in A} \{\max_{s' \in s_a} u(s')\}$$

Though the best outcome might not be the best decision in the long run. Maximising utility helps us for one decision but may hurt in the long run. *Sometimes we have to tell our robot to eat broccoli not only pie.*

Example



- The agent has to pick a sequence of actions.

$$A(s) = \{Up, Down, Left, Right\}$$

for all states s.

Top right corner is the food, he wins below there a ghost. The diagram one the right shows that given an a direction there is a 0.8 chance it will follow that, a 0.1 that it will go to the right and a 0.1 that it will go to the left (stochastic actions).

- So $Up, Up, Right, Right, Right$ succeeds with probability

$$0.8^5 = 0.32768$$

Transition Model

Model to describe actions taken by an agent.

- Since the actions are stochastic, the model looks like:

$$P(s'|s, a)$$

where a is the action that takes the agent from s to s'.

- Transitions are assumed to be (first order) Markovian.
- They only depend on the current and next states.

The problem must also include the utility function. This should be defined over sequences of states or runs, it is assumed that at each state the agent receives a reward R(s) which might be positive or negative.

We can think of each action as giving a reward or having a cost.

Markov Decision Process (MDP)

This agent faces a Markov decision process:

- Mathematically we have
  - a set of states $s \in S$ with an initial state s_0.
  - A set of actions $A(s)$ in each state.
  - A transition model $P(s'|s, a)$; and
  - A reward function $R(s)$.
- Captures any fully observable non-deterministic environment with a Markovian transition model and additive rewards.

We will use this for any **observable** (we know the state utility of each action at all times) **non-deterministic** (many fixed states to one action).

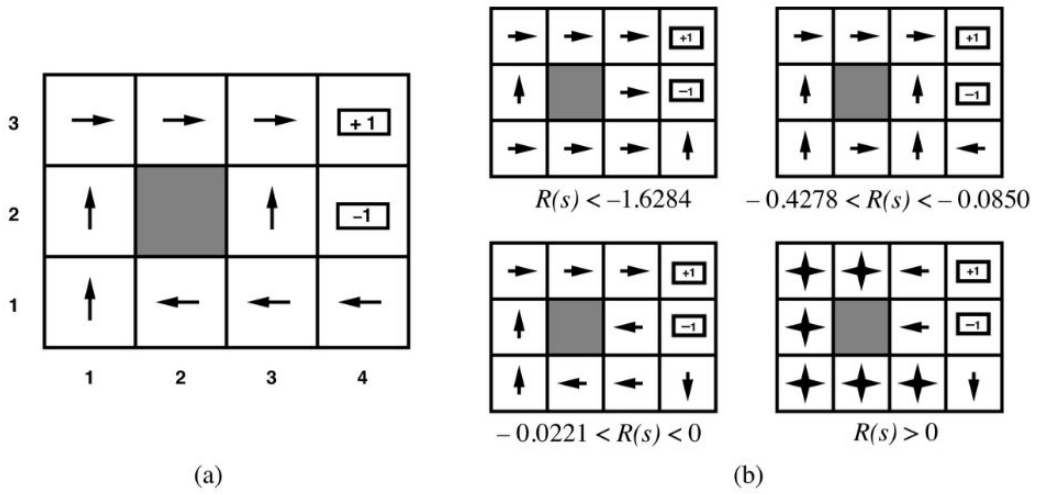
- A solution is a policy, which we write as π.
- This is a choice of action for every state.
  - that way if we get off track, we still know what to do.
- In any state s, $\pi(s)$ identifies what action to take.

A policy is like a book that states if you're in this state do this action, for every state. You solve the markov decision process that compiles a policy, you give that to the agent and he follows that. This is useful since if one of the actions goes wrong and ends up in another place, the agent still knows what to do.

- We want the **optimum** policy, not that since actions are stochastics, they won't give the same reward every time (as not the same thing will happen every time) thus we will take into account) expected value.

- The optimum policy π^* is the policy with the highest expected value.
- At every stage the agent should do $\pi^*(s)$.

Example of a policy



(a) Optimal policy for the original problem.
(b) Optimal policies for different values of $R(s)$.

Every state has an action in the policy so that if we don't end up in the state that we expect, we know what to do next.

Top left: the reward for each state is so bad that the agent wants to leave the world as quick as possible (since even -1 is better than -1.6).
Bot right: agent wants to survive since the reward is always good and finishing may not give him a better outcome.

Calculating Utilities

We need to compute the utility for a series of states. It is easier to calculate the utility for an infinite game and apply it to finite games.

- It is important to consider if utilities are stationary or non-stationary. E.g. you always want to pass the module rather than failing it. You ALWAYS want to pass it, whether there is an exam today or class or not.
- You might not only want to eat pizza thus they might not be stationary.

Stationary Utilities

- With stationary utilities, there are two ways to establish $U_r([s_0, s_1, \ldots, s_n])$ from $R(s)$.
- Additive rewards:

$$U_r([s_0, s_1, \ldots, s_n]) = R(s_0) + R(s_1) + \ldots + R(s_n)$$

as above.

- Discounted rewards:

$$U_r([s_0, s_1, \ldots, s_n]) = R(s_0) + \gamma R(s_1) + \ldots + \gamma^n R(s_n)$$

where the discount factor γ is a number between 0 and 1.

- The discount factor models the preference of the agent for current over future rewards.

With discounted rewards, you get the full reward of the first state but then you get less and less of the coming rewards. The idea is that you would rather have a reward now than later.

Note that with infinite additive undiscounted rewards we cannot estimate the utility of the possible since addition numbers infinitely gives us **infinity** and comparing infinities is really complex. We can fix this by using:

- Proper policies: policies end up in a terminal state (thus finite expected utility (EU))
- Average reward: calculate average reward per time step (this will normally be finite)
- Discounted rewards (most common) as the result over an infinite sequence is finite

Discounted Rewards

We compare policies by comparing their EU.

- The expected utility of executing π starting in s is given by:

$$U^{\pi}(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(S_t)\right]$$

where S_t is the state the agent gets to at time t.

Utility of a policy starting at state s = the sum of the discounted rewards in each state.

St (or state at a particular point in time) is a random variable, we can calculate the probability of all its values by thinking of all the possible runs which end up at there after t steps. *(this is complicated and we don't want to do it but conceptually we can do this).*

The optimal policy is:

$$\pi^* = \arg\max_{\pi} U^{\pi}(s)$$

We think of all the policies, we find the one with the max EV and we use that. This is a brute force way of doing it.

- Since we are finding the optimal policy, it doesn't matter at what state we start. Since eventually we will look at all, the optimal policy is the optimal policy.

Example

3	0.812	0.868	0.918	+1
2	0.762		0.660	−1
1	0.705	0.655	0.611	0.388
	1	2	3	4

- Here we have the values of states if the agent executes an optimal policy

  $U^{\pi^*}(s)$

- What should the agent do if it is in (3, 1)?

If we have the values of the states under optimal policy, we can find the optimal policy.

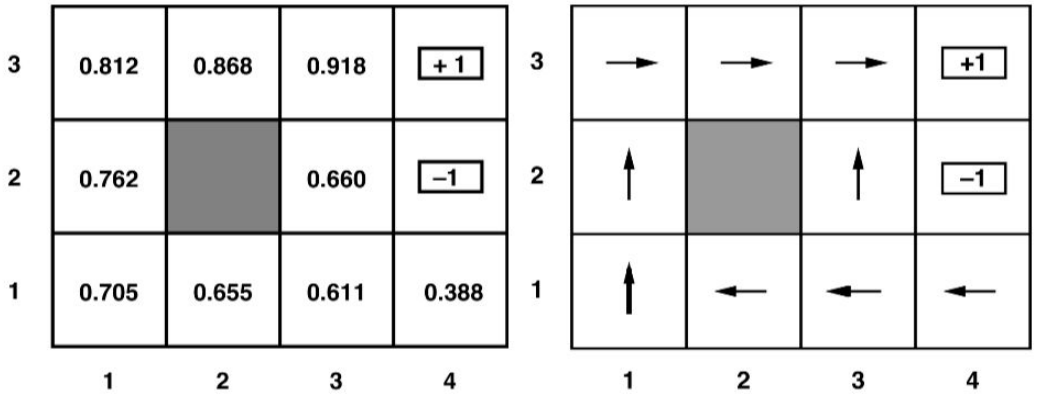
Going left is the best choice, since going up is to risky since in the next step there is a high chance that we will go to -1. You can calculate that by:

- If we have these values, the agent has a simple decision process
- It just picks the action *a* that maximises the expected utility of the next state:

$$\pi^*(s) = \arg\max_{a \in A(s)} \sum_{s'} P(s'|s,a) U^{\pi^*}(s')$$

- Only have to consider the next step.
- The big question is how to compute $U^{\pi^*}(s)$.

The optimal policy is the table to the right below.



- Note that this is specific to the value of the reward $R(s)$ for non-terminal states — different rewards will give different values and policies.

To get the values for each state we can use the Bellman Equation.

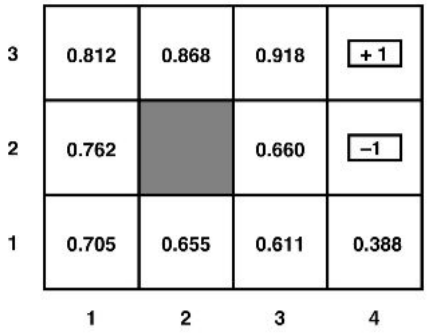
Bellman Equation

$$U(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} \Pr(s'|s,a) U(s')$$

- γ is a discount factor.

Utility of state = reward for the state + discounted value of the expected utility of the best action in that state.

Example



$$\begin{aligned} U(1,1) = {} & -0.04 + \\ & \gamma \, max[0.8U(1,2) + 0.1U(2,1) + 0.1U(1,1), && (Up) \\ & \quad 0.9U(1,1) + 0.1U(1,2), && (Left) \\ & \quad 0.9U(1,1) + 0.1U(2,1), && (Down) \\ & \quad 0.8U(2,1) + 0.1U(1,2) + 0.1U(1,1)] && (Right) \end{aligned}$$

We have to consider all the actions possible and calculate all the possibilities.

Note that *max is non linear thus we might want to find a more efficient solution.* We can use a method called value iteration to get the utility of each state.

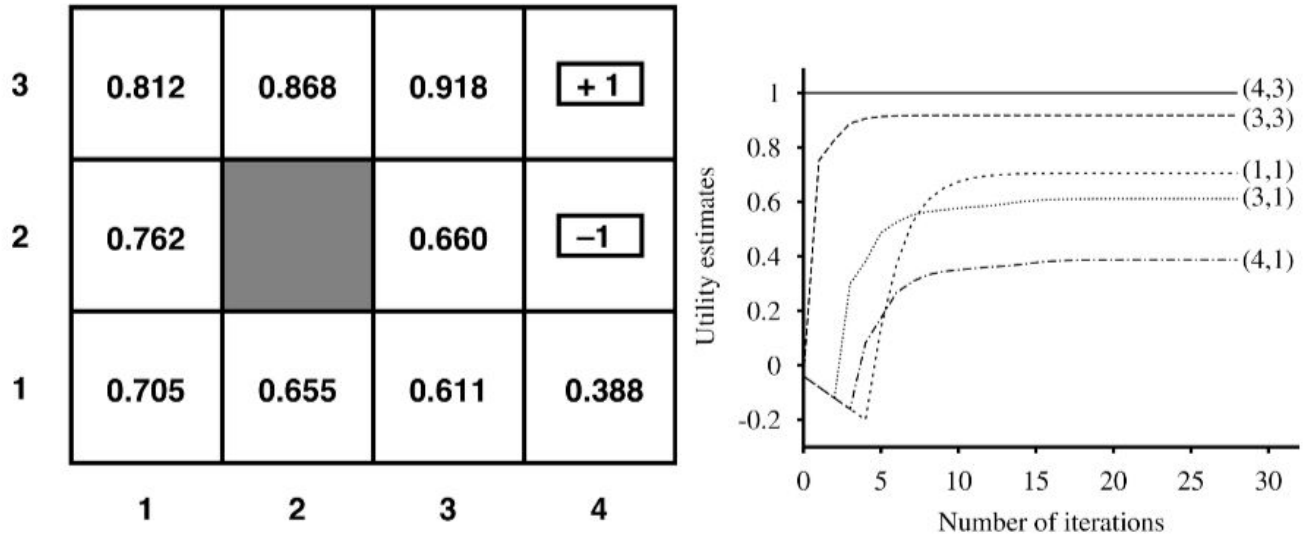
Value iteration

- Start with arbitrary values for states and apply the Bellman update:

$$U_{i+1}(s) \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s'|s,a) U_i(s')$$

  simultaneously to all the states.
- Continue until the values of states do not change.
- After an infinite number of applications, the values are guaranteed to converge on the optimal values.

When the value no longer changes that is the most optimal value (we can do this until a certain degree of accuracy, this way we can make it linear).



This is what occurs when we run the equation multiple times, values will eventually tend towards their optimal values.

The above equation takes into account that the values get a reward, we can also think of this as a cost of moving to the next state. We use the following equation for cost:

$$R(s) = -c(s,a)$$

where s is the action used.

- Bellman becomes:

$$U_{i+1}(s) \leftarrow \gamma \max_{a \in A(s)} (\sum_{s'} P(s'|s,a) U_i(s')) - c(s,a)$$

The value can be dependent on the state.

Policy Iteration

You directly compare policies to each other and using the best. We start with a policy, figure out its utility, improve it and repeat. We need the policy value to improve it.

Policy Improvement

For each state we calculate the best action given the values.

Calculate a new policy π_{i+1} by applying:

$$\pi_{i+1}(s) = \arg \max_{a \in A(s)} \sum_{s'} P(s'|s,a) U_i(s')$$

For each state we do a one-step lookahead.

A simple decision.

Use the values established by policy evaluation.

We figure out the best next action and make it the next action in the policy.

Policy Evaluation

We can calculate the utility by applying the action and seeing what happens.

- How do we calculate the utility of each step given the policy π_i?
- Turns out not to be so hard.
- Given a policy, the choice of action in a given state is fixed (that is what a policy tells us) so:

$$U_i(s) = R(s) + \gamma \sum_{s'} P(s'|s, \pi_i(s)) U_i(s')$$

- Again there are lots of simultaneous equations, but now they are linear (no max) and so standard linear algebra solutions will work.

Conclusion

- Put these together to get:
- Starting from some initial policy π_0 we do:
  1. Policy evaluation
     Compute:

$$U_i(s) = R(s) + \gamma \sum_{s'} P(s'|s, \pi_i(s)) U_i(s')$$

     for every state.
  2. Policy improvement
     Calculate a new policy π_{i+1} by applying:

$$\pi_{i+1}(s) = \arg \max_{a \in A(s)} \sum_{s'} P(s'|s, a) U_i(s')$$

     for every state s.

  Until convergence.

- The iteration will terminate when there is no improvement in utility from one iteration to the next.
- At this point the utility Ui is a fixed point of the Bellman update and so πi must be optimal.

Problems

There is a problem with the policy evaluation stage of the policy iteration approach.

- If we have n states, we have n linear equations with n unknowns in the evaluation stage.

- The Solution is in O(n^3)
- For large n, can be a problem.
- So, we settle for an approximate solution.

E.g. in pacman the world is changing so the rewards of states change, thus we cannot just solve the world once, we have to solve it every time, thus it must be efficient.

Approximate policy evaluation

To get an estimate of the utility of a given state under a policy, we can run value iteration with a fixed policy. We just run value iteration for every state a number of times (it is not guaranteed to converge) and then you use that to estimate the value. Choosing the number of times right, it is much cheaper than O(n^3). It's approximate but it works.

Repeat:

$$U_{i+1}(s) \leftarrow R(s) + \gamma \sum_{s'} P(s'|s, \pi_i(s)) U_i(s')$$

a fixed number of times.

Modified Policy Iteration

- Starting from some initial policy π_0 we do:
  1. Approximate policy evaluation
     Repeat

     $$U_{i+1}(s) \leftarrow R(s) + \gamma \sum_{s'} P(s'|s, \pi_i(s)) U_i(s')$$

     a fixed number of times.
  2. Policy improvement

     $$\pi_{i+1}(s) = \arg \max_{a \in A(s)} \sum_{s'} P(s'|s, a) U_i(s')$$

     for every state s.

  Until convergance
- Often more efficient than policy iteration or value iteration.

Solving MDPs

- Value iteration - exact solution
- Policy Iteration - exact
- Modified policy iteration - approximate but much faster

Which one to use depends on the problem.

Limitations of MDPs (Markov Decision Processes)

- Environments must be fully observable.
- Assume agent always knows what state it is.
- The optimal policy only depends on the current state (in the real world we can only guess the current state).
- POMDPs extend the model to deal with partial observability.

Partial Observability Markov Decision Processes (POMDP)

It adds the sensor model to the MDP model,

$$P(e|s)$$

probability of perceiving e in state s.

Rather than being in a state it has a probability it is in each state (probability distribution over the states).

The outcome is something like: *:"if you think you are here do this"*

In a POMDP we have continuous states (probability for every action) - an exact solution difficult to achieve for simple problems. We just try to estimate it.

Lecture 5 Game Theory

Strategic reasoning is making a decision in the presence of others.
Game Theory: a framework for analysing interactions between a set of agents. It describes its agent's preference in terms of their utility (all agents want to maximise utility).

- Given a range of **solution strategies** we can make predictions about how agents will/should interact.

Payoff Matrices

We can characterise the "choose side" scenario in a payoff matrix

		j left	*j* right
i	left	1, 1	0, 0
	right	0, 0	1, 1

Agent *i* is the row player
gets the lower reward in a cell.

Agent *j* is the column player
gets the upper reward in a cell.

Really there are two matrices here, one (A) that describes payoff to i and B which describes payoff to j. Sometimes we will write the description madtrix as (A,B) due to this.

An outcome is what we get when we combine the actions of all the players.

An outcome corresponds to an element of the payoff matrix

		j left	*j* right
i	up	1, 1	0, 0
	down	0, 0	1, 1

We identify outcomes by the moves the players make:

(*what i plays*, *what j plays*)

Thus (*up*, *right*) identifies the outcome in which *i* plays *up* and *j* plays *right*

Given a particular scenario an agent will play, but they might take into account what the other person might play (and so on)

- Dominant Strategy

- Nash Equilibrium Strategy
- Pareto Optimal Strategies
- Strategies that maximise social welfare

Dominant Strategies

Given a strategym we say **s1 dominates s2** if every possible outcome played by i playing **s1** is better than any outcome of playing **s2** by i. If one strategy is always better than the other in the same situation it is the **dominant**.

Thus in this game:

		j D	*j* C
i	D	1 1	4 1
	C	1 4	4 4

C dominates *D* for both players.

- **S1 strongly dominates s2** if u(s1) > u(s2) for all outcomes.
- **S1 weakly dominates s2** if u(s1) >= u(s2) for all outcomes.

A rational agent will never play a dominated strategy, thus we can ignore dominated strategies.

Example

	L	C	R
U	1 3	1 0	0 0
M	1 1	1 1	0 5
D	1 0	1 4	0 0

In this case strategy R is dominated by L thus we can remove it.

Nash Equilibrium

Two strategies s1 and s2 are in nash equilibrium (NE) if :

- Assuming that i plays s1, agent j can do no better than play s2 and
- Assuming that j plays s2, agent i can do no better than play s1.

No one is better off by doing anything else. No incentive do to something else.

Let's consider the payoff matrix for the grade game:

		j = Y	j = X
i	Y	2, 2	1, 4
	X	4, 1	3, 3

Here the Nash equilibrium is (Y, Y).

If i assumes that j is playing Y, then i's best response is to play Y.

Same for j.

We obtain **nash equilibrium** if two strategies are best responses to each other. If an agent would be better off playing some other strategy then we are **not** in nash equilibrium.

More formally:

A pair of strategies (i^*, j^*) is a Nash equilibrium solution to the game (A, B) if:

$$\forall i,\ a_{i^*,j^*} \geqslant a_{i,j^*}$$
$$\forall j,\ b_{i^*,j^*} \geqslant b_{i^*,j}$$

That is, (i^*, j^*) is a Nash equilibrium if:

- If j plays j^*, then i^* gives the best outcome for i.
- If i plays i^*, then j^* gives the best outcome for i.

Unfortunately:

- Not every interaction scenario has a pure NE.
- Some interaction scenarios have more than one NE.

This game has two pure strategy NEs, (C, C) and (D, D):

		j = D	j = C
i	D	5 3	1 2
	C	0 2	3 3

In both cases, a single agent can't unilaterally improve its payoff.

Unilaterally means that they will both have to agree to maximise output.

Pareto Optimality (Pareto efficient)

Ask the cell: can we improve an output without making anyone worse-off? If not, it's pareto optimal.
An outcome is **Pareto Optimal** if there is no other outcome that makes one agent **better off** without making another agent **worse off**.

- An outcome w is **not** pareto optimal if there is another outcome w' that makes everyone as happy, if not happier than w.

Find the outcome in which everyone is as happy as possible, even though you might not be better off. I.e. Let your friend eat the rest of your food because he's hungry and you are full.

This game has one Pareto efficient outcome, (D, D).

		j = D	j = C
i	D	5 3	1 2
	C	0 2	0 1

There is no solution in which either agent does better.

This next game has two Pareto efficient outcomes, (C, D) and (D, C).

i \ j	D	C
D	1, 3	4, 1
C	1, 4	1, 1

Note that Pareto efficiency doesn't necessarily mean *fair*.

Just that you can't move away and make one agent better off without making the other worse off.

- Pareto optimality is a rather weak concept. If I have a pile of money and we divide it into two people the pareto optimal point is to divide it anywhere. Give you 700 and 300 to me. If we change that in any way we are making one better off and the other worse off.

Social Welfare

The social welfare of an outcome w is the sum of the utilities that each agent gets form w.

$$\sum_{i \in Ag} u_i(\omega)$$

- This solution may be appropriate if the whole system (all agents) has a single owner. Then the overall benefit of the system is important, not individuals. I.e. how I split my money into different bank accounts.

In the following example (C,C) maximises social welfare

i \ j	D	C
D	2, 2	1, 1
C	3, 3	4, 4

i \ j	D	C
D	2, 2	1, 1
C	3, 3	9, 0

Here welfare is shared
But it is maximised for both.

here it gives it all to one player

Normal Form Games

A normal form game for n-people is a tuple (N,A,u) where:

- A fine set of players
- A finite set of actions
- There's a utility function that assigns a value to each action.

Formally:

- N is a finite set of players.
- $A = A_1 \times \ldots \times A_n$ where A_i is a finite set of actions available to i.
  Each $a = (a_1, \ldots, a_n) \in A$ is an action profile.
- $u = (u_1, \ldots, u_n)$ where $u_i : A \mapsto \Re$ is a real-valued utility function for i.
  - Represented by a n-dimensional matrix (the one for 2 is what we have seen until now)

Agents decide what to do, they use a strategy. Combined with what other agents do this determines a payoff.

- Agents have a **strategy set**: the set of available choices (which can be just actions or pure strategies or mixed strategies).

Common Payoff Game

Here is the payoff matrix from the "choose which side" (of the road) game:

		j: left	j: right
i	left	1, 1	0, 0
	right	0, 0	1, 1

This may also be called a **common payoff game** since in all outcomes, players get the same payoff. Formally:

Any game with $u_i(a) = u_j(a)$ for all $a \in A_i \times A_j$ is a common payoff game.

A **coordination game** is one in which agents choose to do the same action. An **uncoordination** game is one in which agents try to do the opposite of each other:

	left	right
left	0 0	1 1
right	1 1	0 0

Constant Sum Games

Any game in which all the outcomes add up to the same thing.

Matching pennies

	heads	tails
heads	-1 1	1 -1
tails	1 -1	-1 1

Any game with $u_i(a) + u_j(a) = c$ for all $a \in A_i \times A_j$ is a constant sum game.

Zero-sum Games

A game in which all utilities sum zero.

$$u_1(a_i) + u_j(\omega) = 0 \quad \text{for all } a \in A_i \times A_j$$

- Where preferences of agents are diametrically opposed, we have **strictly competitive** scenarios.
- Zero sum implies **strictly competitive.**

- In real life, zero-sum games are rare, normally there is some option that is somewhat better.

Rock Paper Scissors

A constant/zero sum game.

		j: rock	j: paper	j: scissors
i	rock	0 0	1 -1	-1 1
	paper	-1 1	0 0	1 -1
	scissors	1 -1	-1 1	0 0

The optimal strategy is to play a **mixed strategy**, change strategy all the time. (In this case is to play at random).

- You'd sometimes play rock, paper and sometimes scissors. Since a fixed/pure strategy is easy for an adaptive player to beat. (i.e if you always play rock then the other player will see and beat you)

A **mixed strategy** is just a probability across a set of pure strategies.

So, for a game where agent i has two actions a_1 and a_2, a mixed strategy for i is a probability distirbution:

$$MS_i = \{P(a_1), P(a_2)\}$$

Given this mixed strategy, when i comes to play, they pick action a_1 with probability $P(a_1)$ and a_2 with probability $P(a_2)$.

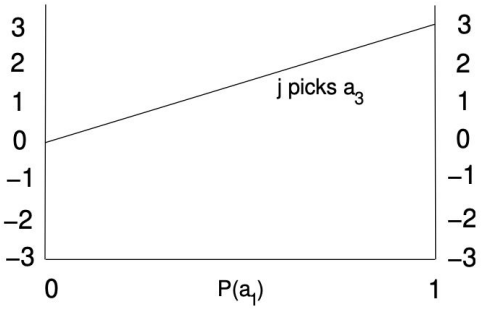
Let's consider the payoff matrix:

		j: a_3	j: a_4
i	a_1	-3 3	1 -1
	a_2	0 0	-1 1

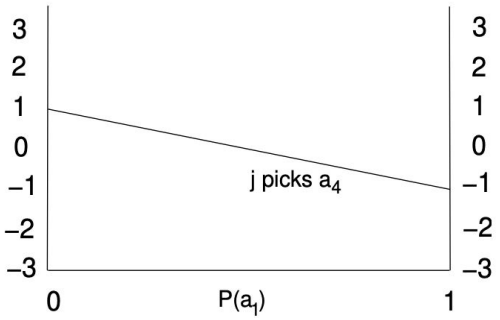
We want to compute mixed strategies to be used by the players.

That means decide $P(a_1)$ and $P(a_2)$ etc.

For i, we would analyse the game like so:



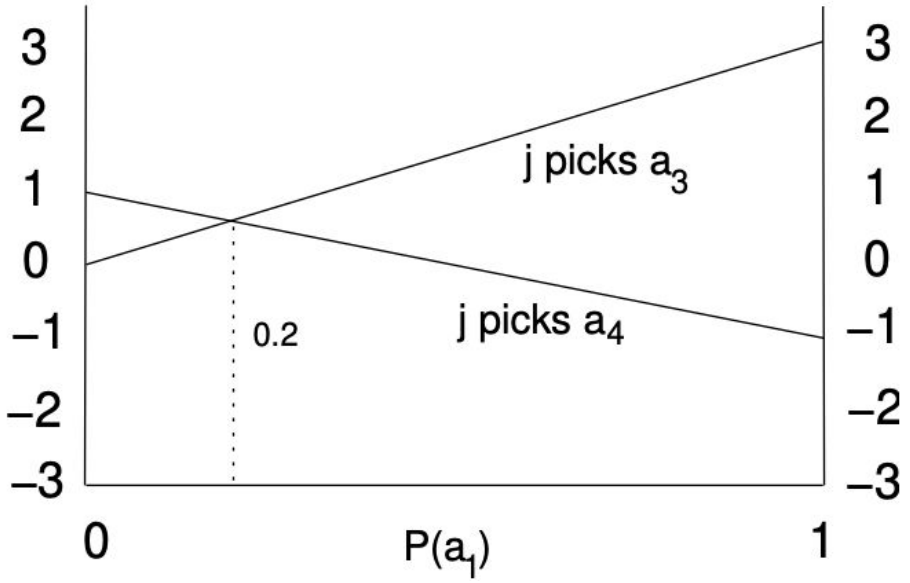
If *j* picks a_3, *i*'s payoff will be 3 or 0 depending on whether *i* picks a_1 or a_2.



If *j* picks a_4, *i*'s payoff will be −1 or 1 depending on whether *i* picks a_1 or a_2.

As we vary the probability of getting a1 from 0 to 1 then our expected payoff given that j picks a3 will be on that line.

If we plot those lines together, the intersection is where *i* has the same payoff no matter what j does.



This is a rational choice of mixed strategy

General Sum Games

- aka Battle of the Sexes

	this	that
this	1 2	0 0
that	0 0	2 1

We like doing stuff together so if we are doing the same thing we are all kind of happy but if I'm doing something I also enjoy then I am better off.

Alex Franch Tapia
Manuel Martinez Chamorro

Negotiation: yields a better outcome than if we just decide what to do.

Every game has a mixed strategy Nash Equilibrium!

- For a game with payoff matrices A (to i) and B (to j), a mixed strategy (x^*, y^*) is a Nash equilibrium solution if:

$$\forall x, x^* A y^{*T} \geqslant x A y^{*T}$$
$$\forall y, x^* B y^{*T} \geqslant x^* B y^T$$

- In other words, x^* gives a higher expected value to i than any other strategy when j plays y^*.
- Similarly, y^* gives a higher expected value to j than any other strategy when i plays x^*.

Note: x* y* are both vectors of probabilities and A is a matrix. To make the multiplication come out right we have to multiply x* by the matrix by the transpose of y.

But this doesn't solve the question of: which nash equilibrium we should play.

The Prisoner's Dilemma

Occurs when two would be better of if they collaborate than if they don't but they don't because only dependent on them they are better off not collaborating.

Payoff matrix for prisoner's dilemma:

		j defect	*j* coop
i	defect	2 2	1 4
	coop	4 1	3 3

This models two criminals that are said that if one confesses and the other doesn't one will go free and the other will stay but that if they both confess they'll get less time in Jail.

"defect" = confess
Numbers are payoffs to players, not years in jail.
What should each agent do?

The individually rational action is defect.
This guarantees a payoff of no worse than 2, whereas cooperating guarantees a payoff of at most 1.

So defection is the best response to all possible strategies: both agents defect, and get payoff = 2.

But intuition says this is not the best outcome:
Surely they should both cooperate and each get payoff of 3!

Solutions for Prisoner's Dilemma

- (D,D) is the only nash equilibrium
- All outcomes except (D,D) are Pareto optimal.
- (C,C) maximises social welfare

There is a paradox:

- The fact that the Nash Equilibrium is not the co-operative solution is problematic.
- This apparent paradox is the fundamental problem of multi-agent interactions.
- It appears to imply that cooperation will not occur in societies of self-interested agents.

Lecture 6 Probabilistic Reasoning Over Time

Our robots depend on sensors to understand the outside world, just as we depend on our senses. In this lecture we look at how to represent the probability of events given a transition model and a sensor model and how we can infer probabilities of these given the world we are in. We will determine the probability of rain given that we have been seeing umbrellas in the past couple of days.

States and observations

States refer to the states of the actors in our world and **observations** are things that we see about the world. They are a means to understand the world and represent it. Between different observations which update us on the state of the world time passes, called **time slices** (just like a frame in a video). Each slice contains some variables:

- The set $\mathbf{X_t}$, which we can't observe, we care about and are trying to measure
- And the set $\mathbf{E_t}$, observations we can make.

Capital letters are for variables, small case represent values at a particular point in time.

I.e. You live in a room with no windows, you want to know if its raining, what you can know is whether people coming from the outside are carrying an umbrella
Each day is one value of t
In the example:

- **Et** is Ut - is the person carrying an umbrella?

- **Xt** is Rt - is it raining?

Sequence starts at t = 0 the day before we start observing things, first piece of evidence comes at t = 1.
So the umbrella world is:
R0, R1, R2 ...
U1, U2, U3 …
Note: a : b means the sequence of integers from a to b, so that $U_{2:4}$ is the sequence U2, U3, U4

Transition and sensor models

We have to talk about how the variables in our time slice are related to each other and how one time slice is related to the next time slice. If they are not linked to each other there would be no evolution over time which is what we are trying to capture. To do so we must understand:

- How the world evolves -> **Transition model**
- What evidence tells us -> **Sensor model**

The transition model

Explains how the world changes

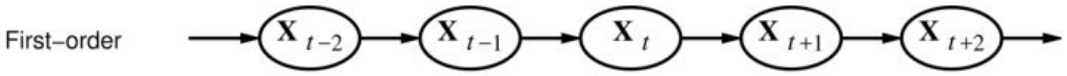
$$\mathbf{P}(\mathbf{X}_t|\mathbf{X}_{0:t-1})$$

What the probability is that is raining today given the weather every previous day for as long as records have existed.

Markov Assumption

There are a lot of things that we have to record to follow these models but we can use a First order Markov assumption to simplify this.

We commonly assume a **Markov assumption** where the current state depends only on the previous state



Where:

$$\mathbf{P}(\mathbf{X}_t|\mathbf{X}_{0:t-1}) = \mathbf{P}(\mathbf{X}_t|\mathbf{X}_{t-1})$$

Fro a second order Markov Process:

$$\mathbf{P}(\mathbf{X}_t|\mathbf{X}_{0:t-1}) = \mathbf{P}(\mathbf{X}_t|\mathbf{X}_{t-2}, \mathbf{X}_{t-1})$$

We can still have an infinite set of conditional probabilities:

$$\mathbf{P}(\mathbf{X}_1|\mathbf{X}_0), \mathbf{P}(\mathbf{X}_2|\mathbf{X}_1), \mathbf{P}(\mathbf{X}_3|\mathbf{X}_2) \ldots$$

To fix this, we assume that the process is stationary, the model doesn't change but the states can and we can use a general formula:

$$\mathbf{P}(\mathbf{X}_t|\mathbf{X}_{t-1})$$

The sensor model

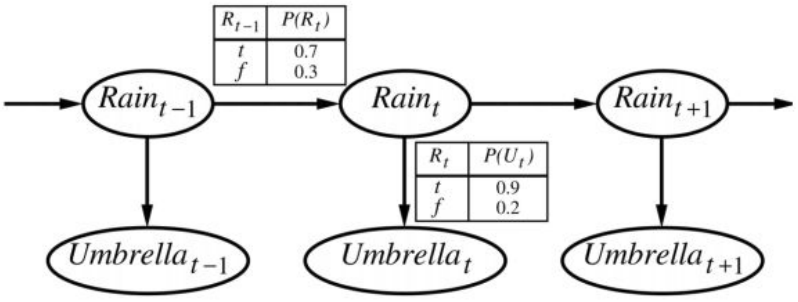
Tells us how what we can see is related to what we try to measure. The evidence variables **Et** could depend on lots of previous variables, we will assume the state is constructed in a way in which it only depends on the current state.

Markov assumption for the sensor model:

$$\mathbf{P}(\mathbf{E}_t|\mathbf{X}_{0:t-1}, \mathbf{E}_{0:t-1}) = \mathbf{P}(\mathbf{E}_t|\mathbf{X}_t)$$

The transition and sensor model for the umbrella world

Given that we have rain affect if we see an umbrella or not at different points in time created a model like so:



But we need to know how things started:

$$\mathbf{P}(X_0)$$

In the example it would be P(R0)

Therefore to conduct the **joint** probability of all time slices. That given the information we have what we say is true is true is:

$$\mathbf{P}(\mathbf{X}_{0:t}, \mathbf{E}_{1:t}) = \mathbf{P}(\mathbf{X}_0) \prod_{i=1}^{t} \mathbf{P}(\mathbf{X}_i|\mathbf{X}_{i-1})\mathbf{P}(\mathbf{E}_i|\mathbf{X}_i)$$

Essentially the probability of X0, X1, X2…

As we know from Lecture 3, the probability over a Bayesian Network is:

$$P(x_1, \ldots, x_n) = \prod_{i=1}^{n} P(x_i|parents(X_i))$$

The only problem with this, is that computationally this is very complex, we will talk about algorithms that make this easier.

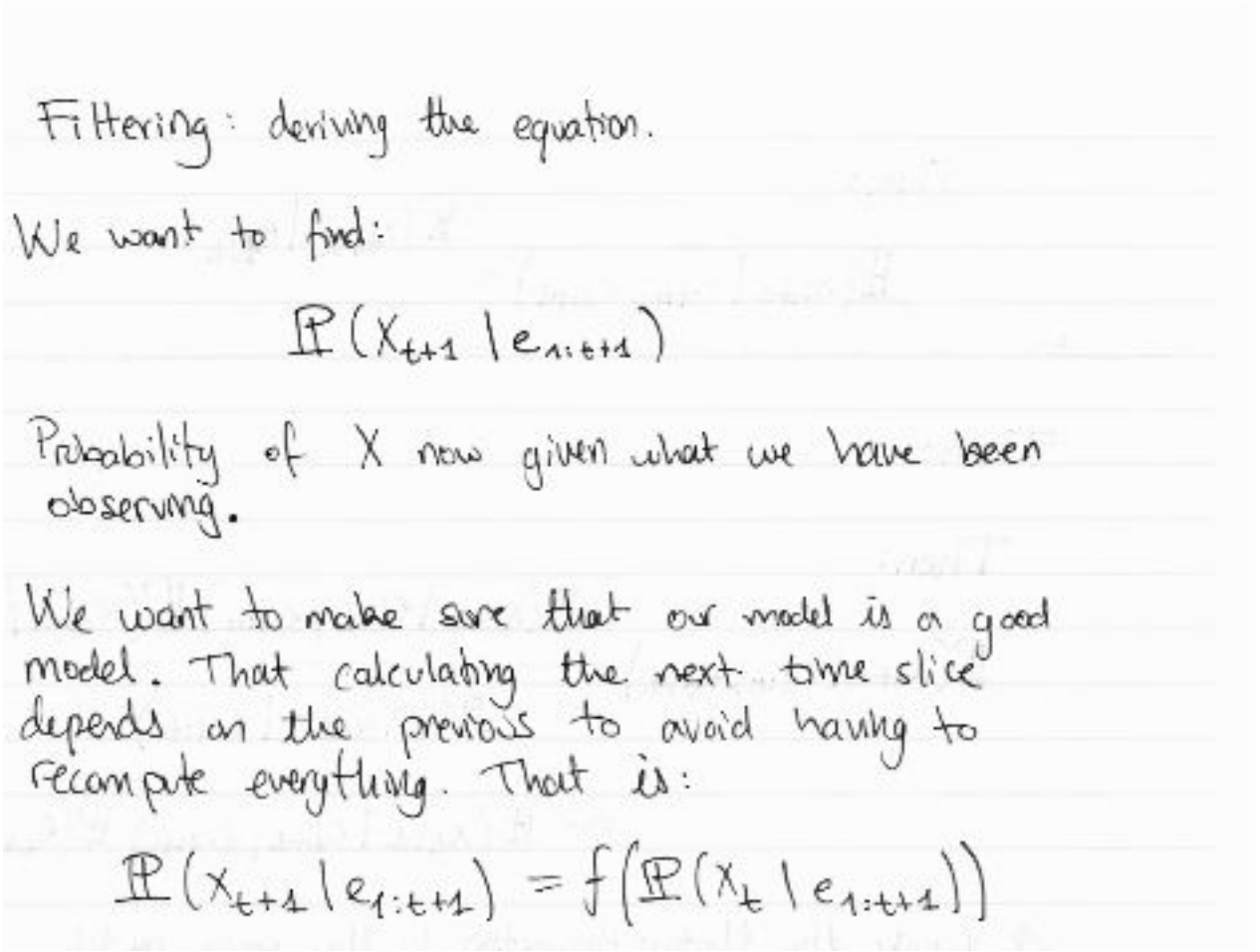
Each time slice is a Bayesian network and the whole thing is called a **Dynamic Bayesian Network**.

Inference Tasks

What can we do with this model?

- **Filtering:** there's a process evolving over time and you care about the state of the world at that particular point in time. You've been observing stuff, you are now here and you're like what's my world like. **P**(X_t | $e_{1:t}$)
    - Determining where a hurricane is in a future point in time.
- **Prediction:** what is the world going to look like in the future. Like filtering but without the evidence for it. **P**(X_{t+k} | $e_{1:t}$) for k > 0
    - Predicting where the hurricane will be in future points of time.
- **Smoothing:** you are interested in how the world was back at an earlier point and you look at that. It turns out that later on you might revise what happened earlier and since you have collected more evidence you might be able to be more accurate. Better estimating past terms can make you learn better. **P**(X_k | $e_{1:t}$) for 0 <= k < t
    - Going back and re-adjusting where it actually when throughout that time given the new evidence.
- **Most likely explanation:** trying to figure out what was the most likely value of the Xs at a point in time. Once you get to the end of a phrase, you might want to change what that meant but some things you say have more probability of being said than others.

#heading=h.#heading=h.Filtering



In terms of a robot, we want to calculate where we will be by using where we are now, not having to recompute everything. Therefore we want to find the "function".

$$\mathbb{P}(X_{t+1} \mid e_{1:t+1}) = ?$$

1. Break up the evidence

$$\mathbb{P}(X_{t+1} \mid e_{1:t+1}) = \mathbb{P}(X_{t+1} \mid e_{1:t}, e_{t+1})$$

2. Apply Baye's law

$$(1)\ P(A|C) = \frac{P(C|A)\,P(A)}{P(B)} \quad \& \quad (2)\ P(A|B,C) = \frac{P(C|A,B)\,P(A|B)}{P(C|B)}$$

If for (2): $A = e_{t+1}$ $B = e_{1:t}$ $C = X_{t+1}$

Then:

$$\mathbb{P}(X_{t+1} \mid e_{1:t}, e_{t+1}) = \frac{\mathbb{P}(X_{t+1} \mid e_{t+1}, e_{1:t})\,\mathbb{P}(e_{t+1} \mid e_{1:t})}{\mathbb{P}(X_{t+1} \mid e_{1:t})}$$

$$= \alpha\, \mathbb{P}(X_{t+1} \mid e_{t+1}, e_{1:t})\, \mathbb{P}(e_{t+1} \mid e_{1:t})$$

3. Apply the Markov Assumption to the sensor model.

$$\mathbb{P}(X_{t+1} \mid e_{1:t}, e_{t+1}) = \alpha\, \mathbb{P}(X_{t+1} \mid e_{t+1})\, \mathbb{P}(e_{t+1} \mid e_{1:t})$$

4. Apply more probability theory.

$$(3)\ P(A|C) = \sum_B P(A|B,C)\,P(B|C)$$

By $\sum_S$ we can eliminate it or add it to our eq.

$$\mathbb{P}(X_{t+1} \mid e_{1:t}, e_{t+1}) = \alpha\, \mathbb{P}(X_{t+1} \mid e_{t+1}) \sum_{x_t} P(X_{t+1} \mid x_t, e_{1:t})\, P(x_t \mid e_{1:t})$$

5. Apply markov assumption again. Since everything affecting X at t+1 also affects X at time t. Then.

$$P(x_{t+1} \mid x_t, e_{1:t}) = P(x_{t+1} \mid x_t)$$

Therefore:

$$\mathbb{P}(x_{t+1} \mid e_{1:t}, e_{t+1}) = \mathbb{P}(e_{t+1} \mid x_{t+1}) \sum_{x_t} P(x_{t+1} \mid x_t) P(x_t \mid e_{1:t})$$

In words:
P of rain on a particular day, given all that we have observed is whatever the probability was the day before times how much "rain on the previous day affects rain today" times how seeing an umbrella today affects the probability of rain today.

With this model, we only need the value for day zero to calculate the next and so on and so forth.

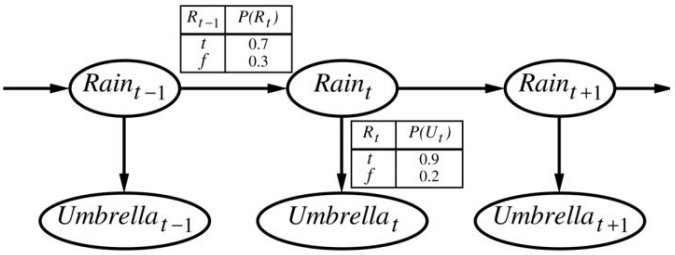
Note the final eq after therefore is missing an alpha after the equals sign. Note2 P after Sigma is a distribution P.

In a prettier way:

$$\begin{aligned} \mathbf{f}_{1:t} &= \mathbf{P}(\mathbf{X}_t \mid \mathbf{e}_{1:t}) \\ &= \alpha \mathbf{P}(\mathbf{e}_t \mid \mathbf{X}_t) \sum_{\mathbf{x}_t} \mathbf{P}(\mathbf{X}_t \mid \mathbf{x}_{t-1}) P(\mathbf{x}_{t-1} \mid \mathbf{e}_{1:t-1}) \end{aligned} \qquad (1)$$

So probability of rain on t+1 = Sensor mode * Transition model * probability of rain on previous day

Filtering: Umbrella Example



- The prior for day 0, $P(R_0)$, is $\langle 0.5, 0.5 \rangle$.
- We can first *predict* whether it will rain on day 1 given what we already know:

$$\begin{aligned} \mathbf{P}(R_1) &= \sum_{r_0} \mathbf{P}(R_1|r_0)P(r_0) \\ &= \langle 0.7, 0.3 \rangle \times 0.5 + \langle 0.3, 0.7 \rangle \times 0.5 \\ &= \langle 0.5, 0.5 \rangle \end{aligned}$$

Note: <0.7,0.3> is <P(R_1 | r_0), P(R_1 | ¬r_0)>
We are not really sure if it will rain on day one since there is no evidence to help us.

We can filter the probability of rain on day one, given that new evidence has arrived. We have seen an umbrella.

- However, we have observed the umbrella, so that $U_1 = true$, and we can update using the sensor model:

$$\begin{aligned} \mathbf{P}(R_1|u_1) &= \alpha \mathbf{P}(u_1|R_1)\mathbf{P}(R_1) \\ &= \alpha \langle 0.9, 0.2 \rangle \langle 0.5, 0.5 \rangle \\ &= \alpha \langle 0.45, 0.1 \rangle \\ &\approx \langle 0.818, 0.182 \rangle \end{aligned}$$

So, since umbrella is strong evidence for rain, the probability of rain is much higher once we take the observation into account.

Now although no new evidence has arrived, we can predict the probability for day two:

- We can then carry out the same computation for day 2, first *predict*ing whether it will rain given what we already know:

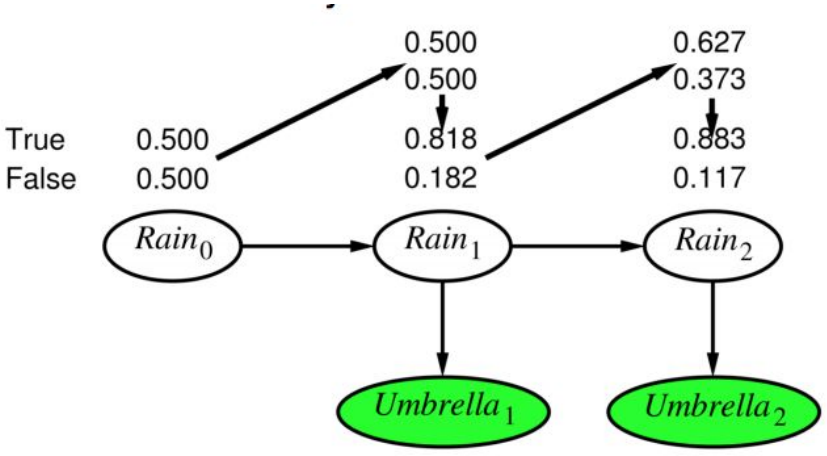
$$\begin{aligned}\mathbf{P}(R_2|u_1) &= \sum_{r_1}\mathbf{P}(R_2|r_1)P(r_1|u_1)\\ &= \langle 0.7, 0.3\rangle \times 0.818 + \langle 0.3, 0.7\rangle \times 0.182\\ &\approx \langle 0.627, 0.373\rangle\end{aligned}$$

- So even without evidence of rain on the second day there is a higher probability of rain than the prior because rain tends to follow rain.

When new evidence arrives. We can update our function.

- Then we can repeat the evidence update, u_2 ($U_2 = true$), so:

$$\begin{aligned}\mathbf{P}(R_2|u_1, u_2) &= \alpha\mathbf{P}(u_2|R_2)\mathbf{P}(R_2|u_1)\\ &= \alpha\langle 0.9, 0.2\rangle\langle 0.627, 0.373\rangle\\ &= \alpha\langle 0.565, 0.075\rangle\\ &\approx \langle 0.883, 0.117\rangle\end{aligned}$$



The reason why we have a higher probability on day 2 than the probability of rain on day 0 is because rain persits from day to day, making it more likely it will rain.

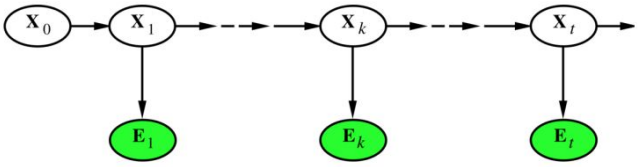
Prediction

Prediction is filtering without new evidence (no sensor model). Is what we do when we go from one day to the other (RHS of eq).

$$\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{e}_{1:t+1}) = \sum_{\mathbf{x}_t} \mathbf{P}(\mathbf{X}_{t+1}|\mathbf{x}_t, \mathbf{e}_{1:t}) P(\mathbf{x}_t|\mathbf{e}_{1:t})$$

Smoothing

Smoothing is computing the distribution over the past states given evidence up to the present.



- Want the probability over all states k, $0 \leqslant k < t$.

We want to know the probability of states before where w are. Between 0 and t. We are at time *t* we want the probability at time *k*.

We break the computation into two pieces, evidence from 0 to K and evidence from K + 1 to t
Proceeding as before

$$\begin{aligned}
\mathbf{P}(\mathbf{X}_k|\mathbf{e}_{1:t}) &= \mathbf{P}(\mathbf{X}_k|\mathbf{e}_{1:k}, \mathbf{e}_{k+1:t}) \\
&= \alpha \mathbf{P}(\mathbf{X}_k|\mathbf{e}_{1:k}) \mathbf{P}(\mathbf{e}_{k+1:t}|\mathbf{X}_k, \mathbf{e}_{1:k}) \quad \text{Bayes' Rule} \\
&= \alpha \mathbf{P}(\mathbf{X}_k|\mathbf{e}_{1:k}) \mathbf{P}(\mathbf{e}_{k+1:t}|\mathbf{X}_k) \quad \text{Cond. Ind.} \\
&= \alpha \mathbf{f}_{1:k} \mathbf{b}_{k+1:t} \quad (2)
\end{aligned}$$

Where **f** is a “forward” message compiled like the filtering case and **b** is a backwards message\

- To compute the backwards message we condition on $\mathbf{X}_{k+1}$:

$$\begin{aligned}
\mathbf{b}_{k+1:t} &= \mathbf{P}(\mathbf{e}_{k+1:t}|\mathbf{X}_k) \\
&= \sum_{\mathbf{x}_{k+1}} \mathbf{P}(\mathbf{e}_{k+1:t}|\mathbf{X}_k, \mathbf{x}_{k+1}) \mathbf{P}(\mathbf{x}_{k+1}|\mathbf{X}_k)
\end{aligned}$$

- Then apply conditional independence:

$$\mathbf{b}_{k+1:t} = \sum_{\mathbf{x}_{k+1}} P(\mathbf{e}_{k+1:t}|\mathbf{x}_{k+1}) \mathbf{P}(\mathbf{x}_{k+1}|\mathbf{X}_k)$$

- Rewriting $\mathbf{e}_{k+1:t}$:

$$\mathbf{b}_{k+1:t} = \sum_{\mathbf{x}_{k+1}} P(\mathbf{e}_{k+1}, \mathbf{e}_{k+2:t}|\mathbf{x}_{k+1}) \mathbf{P}(\mathbf{x}_{k+1}|\mathbf{X}_k)$$

- Applying conditional independence again:

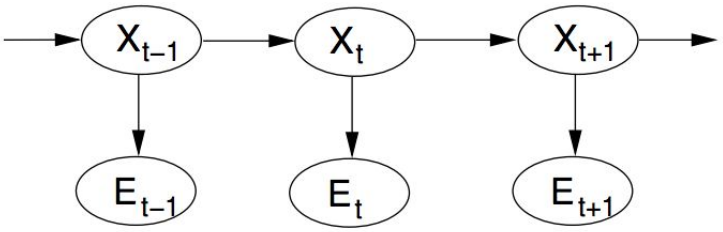
$$\mathbf{b}_{k+1:t} = P(\mathbf{e}_{k+1}|\mathbf{x}_{k+1})P(\mathbf{e}_{k+2:t}|\mathbf{x}_{k+1})\mathbf{P}(\mathbf{x}_{k+1}|\mathbf{X}_k) \quad (3)$$

- The first and third terms on the right hand side come from the model, the second term is the bit we compute recursively.
- Remember that $\mathbf{b}_{k+1:t}$ is shorthand for $\mathbf{P}(\mathbf{e}_{k+1:t}|\mathbf{X}_k)$
- So we have $\mathbf{b}_{k+1:t}$ computed from $\mathbf{b}_{k+2:t}$

<< Continue Smoothing>>

Hidden Markov Models

There are some other frameworks we can use to compute the equations before in an easier way.



- HMMs have **Xt** as a single discrete variable
- Usually **Et** is also discrete
- Domain of **Xt** is {1,..., S}

Similar umbrella and rain but with a fixed length of time.

Due to these models being fixed, we can create simple transitions.

- Transition matrix $\mathbf{T}_{ij} = P(X_t = j|X_{t-1} = i)$
  - e.g., $\begin{pmatrix} 0.7 & 0.3 \\ 0.3 & 0.7 \end{pmatrix}$
- Sensor matrix $\mathbf{O}_t$ for each time step, diagonal elements $P(e_t|X_t = i)$
  - $\mathbf{O} = \begin{pmatrix} 0.9 & 0 \\ 0 & 0.2 \end{pmatrix}$

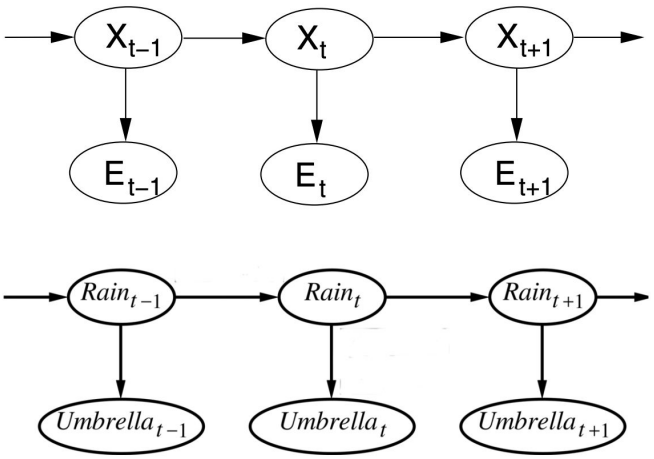
And the forwards and backward equations are simple vector updates.

- Forward and backward messages as column vectors:

$$\mathbf{f}_{1:t+1} = \alpha\mathbf{O}_{t+1}\mathbf{T}^\top\mathbf{f}_{1:t}$$
$$\mathbf{b}_{k+1:t} = \mathbf{T}\mathbf{O}_{k+1}\mathbf{b}_{k+2:t}$$

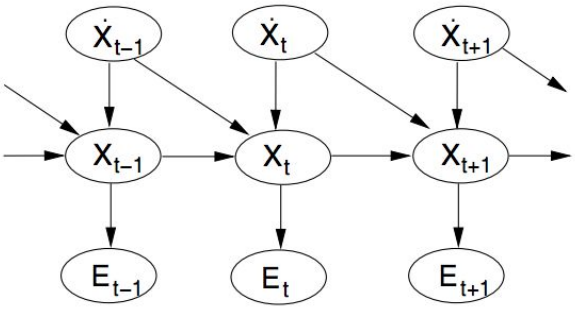
- Forward-backward algorithm needs time $O(S^2t)$ and space $O(St)$

Restricting our Markov Model to be a Hidden one makes it very efficient.



Kalman Filters

A way to do things like tracking. (Below is tracking in one dimension)

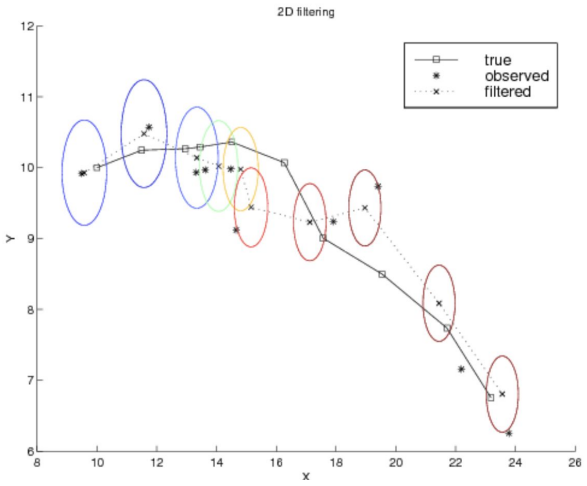


The things that affect the value at time X^t, is where it was at a previous point in time, but also what is velocity was at its previous point in time and what its velocity is now and what it is now affects what it is in the next step.

What makes Kalman Filter widely used and complicated is the fact what you use continuous variables and you allow the probabilities of them to be Gaussian distribution (normal distributions).
Used for:

- Robot tracking
- Airplanes, ecosystems, economies - anything when the current state affects the next and the next affects the next.

Output of Kalman filters



Sometimes your observations is far from the wrong thing, the advantage is that the real value is closer to the center of the oval than the one observed thanks to it remembering where the thing was at a previous point in time and takes that into account in the calculation.

DBNs

Using a HMM on a world described by 20 binary state variables. X is a cross product of 20 and E bears on 20 variables essentially giving a model with 20^{20} x 20^{20} = 10^{12} parameters. Which is a lot

But the good thing about using a Bayesian Network, is that we can make our time slices as complicated as we want which cut down the computation.

- 20 state variables, three parents each
- $20 \times 2^3 = 160$ parameters
- A lot better than 10^{12}

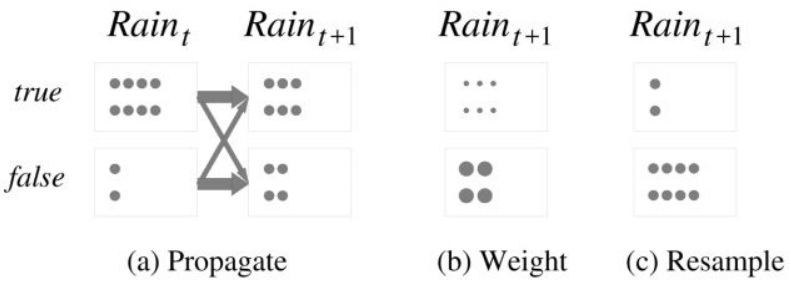
We can calculate inference in DBNs by **unrolling** DBN to create static Bayesian Networks and use the standard approaches. But this may not be worth it given that inference by enumeration has space and time complexity problems and likelihood waiting may need many examples or have large errors.

Particle Filters

Is a better technique for approximate solution of a DBN. It is related to likelihood sampling but instead of going through the whole network, hat particle filtering is closer to the update we do with the forward backward calculation rather than.

You have a bunch of samples but the samples represent a variable in a particular point in time.

Example: we have sames that say that it's raining at time t and others which say its not. For each of those we apply the transition model to know how likely it is that it is raining at time t+1. Then that gives us some samples of that being true and false that it is raining at time t and t+1 (a), we then look at the evidence, we look if there's an umbrella or not and that allows us to put a number on those samples (weight it) (b) and given those weights we pick true and false from the population proportional to the weight of the sample and that gives us a new set of samples (c).



This solution is linear in time/space in the number of particles. You pick how many particles you want to have, your updated are then linear in that number and your error is proportional to that size. If you want less error you can then use more particles/samples. You can play around with how accurate you want to be and how many particles you need.

Note: the samples are called particles

This is used for:

- Figuring out where robots are - good because the total error remains the same over time, easy to account for.

Applying the particle filter

- We approximate $P(x_t)$ by a set of samples:

$$P(x_t) \approx \{x_t^{(i)}, w_t^{(i)}\}_{i=1,\ldots,m}$$

- Each $x_t^{(i)}$ is a possible value of x, and each $w_t^{(i)}$ is the probability of that value (also called an importance factor).
- Initially we have a set of samples (typically uniform) that give us $P(x_0)$.
- Then we update with the following algorithm.

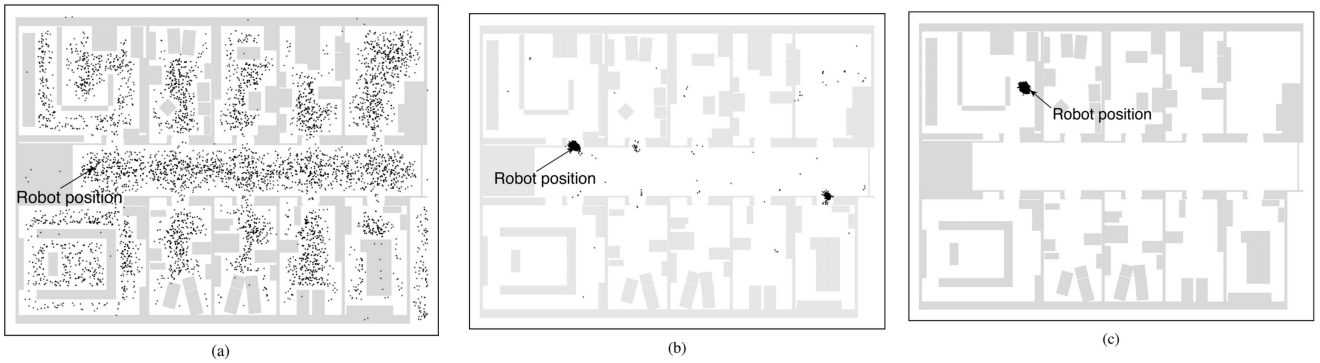
$x_{t+1} = \varnothing$

for $j = 1$ to m
  // apply the transition model
  generate a new sample $x_{t+1}^{(j)}$ from $x_t^{(j)}$, a_t and $\Pr(x_{t+1} \mid x_t, a_t)$
  // apply the sensor model
  compute the weight $w_{t+1}^{(j)} = \Pr(e_{t+1} \mid x_{t+1})$

// pick points randomly but biased by their weight
for $j = 1$ to m
  pick a random $x_{t+1}^{(i)}$ from x_{t+1} according to $w_{t+1}^{(1)}, \ldots, w_{t+1}^{(m)}$

normalize w_{t+1} in x_{t+1}
return x_{t+1}

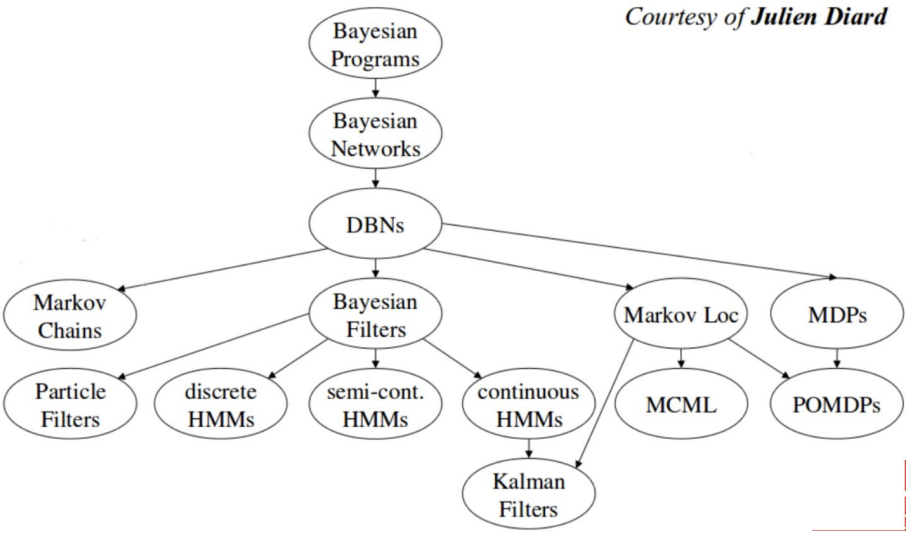
The values converge on the real value.



(b) depends on what the sensors see it could be either of the two points, as we continue to iterate we arrive to (c).

General Classification of Dynamic Probabilistic Models

A general classification of dynamic probabilistic models



Lecture 7 Argumentation

Argument: a statement, reason, or fact for or against the a point.

Argumentation Theory

Argumentation theory allows us to deal with incomplete, uncertain, inconsistent knowledge - typical of the real world.

We need machines to be able to explain why they are doing what they are doing. Therefore as AI is deployed, it is important that humans can understand why humans are doing what they are doing.

Argumentation theory is concerned with knowing which arguments we should accept (**acceptability**).

For example

a1 We should go to the park for a picnic, that's cheap and I don't have much money at the moment.

a2 But I'm cold today, we should go to the café instead.

a3 You won't be cold this afternoon, the forecast says it's going to be really hot.

We have these arguments, which one we decide to do?

Arguments can **attack** one another. We cannot do both a1 and a2 thus they are conflicting.

Approaches to argumentation

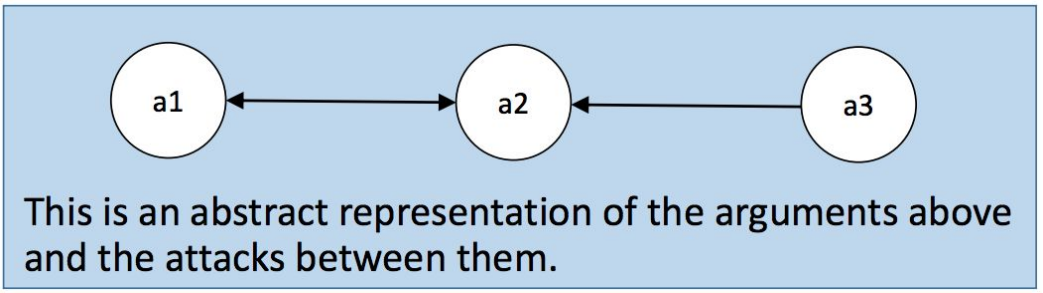
Argumentation can be studied as:

- A **dialogue** mechanism
- Considering the **structure** of arguments and how this affect the attack relation

- From an **abstract** point of view. We assume that an attack relation is given and we want to reason about what it is reasonable to conclude

Abstract Argumentation

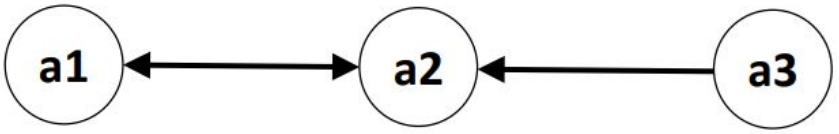
Disregards the internal structure of arguments and focuses on acceptability conditions that allow certain sets of arguments to co-exist in a rational manner.



Abstract argumentation framework: is a tuple <S, R> where S is a set of **arguments** and $R \subseteq S \times S$ is an **attack relation.**

For $a, b \in S$, $(a, b) \in R$ means that argument a attacks argument b.

The abstract argumentation framework $\langle S, R \rangle$ where $S = \{a1, a2, a3\}$ and $R = \{(a1, a2), (a2, a1), (a3, a2)\}$ can be represented as:



Argumentation semantics

An argumentation framework can be given semantics in different ways.

- Via **extensions** (subsets of S with special properties)
- Via **labels** (labelling functions on S with special properties)
- Via **equations** (solutions to a system of equations describing the interactions in the argument framework)

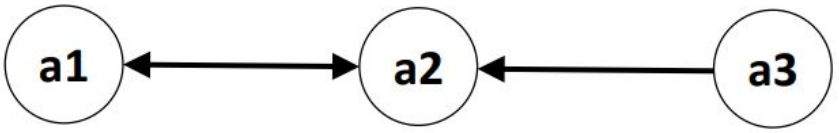
Extension-Based Semantics

An extension is a set of arguments that are jointly “acceptable”, arguments are justified according to their statuses in these extensions.

Conflict-free sets

A set $T \subseteq S$ is conflict-free if and only if there are no arguments in T that **attack each other**.

For all $a, b \in T, a, b \notin R$.



The conflict free subsets of S are: {}, {a1} , {a2} , {a3} , {a1, a3} .

This is important because it is not rational to accept to arguments that contradict each other.

TIP: Remember considering empty sets, sets with one element as well.
TIP2: Remember that if something attacks itself, it's not conflict free.

To make sure we are not skipping any relations, we must ask the question what does "an" attack? And what attacks an for every element.

Argument defence

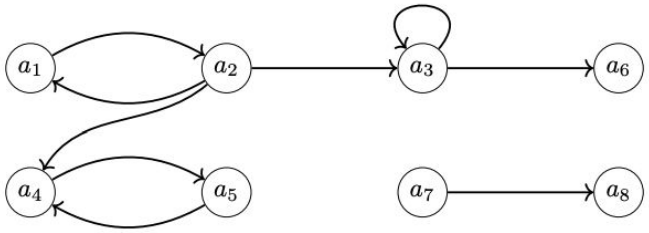
A set that defends an argument by attacking every single thing that attacks the argument.
A set $T \subseteq S$ defends an argument $x \in S$ if and only if for every $y \in S$ such that y attacks x there is an element $z \in T$ such that z attacks y.

In our example, we can see that a1 is attacked by a2 and only a2, but a3 attacks a2. Therefore a3 attacks everything that is attacking a1, therefore a3 defends a1. Additionally, nothing is attacking a3, therefore the empty set defends a3.

Therefore: {a3} defends a1 and {} defends a3.

RULE: If there is an argument X that is not attacked by anything**, then all sets defend X.**

- think about the example "all the elephants in this room are pink". To check whether this is true, you need to check all the elephants in the room to see if they are pink. If there are no elephants in the room then the statement is true. Same thing holds above, if there are no arguments Y that attack X, then it will always be true that for all arguments Y that attack X, there is some argument in S that attacks Y.



- The example is a7.
- Since all complete extensions must contain a7, in this case the {} would not be a complete extension.

Admissible sets

"*Conflict-free set that defends all its members"*
Informally: A set is admissible if the set is conflict free and every element in our set and every element is defended by the people in our set. *The set defends all of its members.*

Formally: A set $A \subseteq S$ is admissible if and only if A is conflict-free and A defends each argument that is a member of A.

- Think of it as a football team, we are all part of a team so there is no conflict and everyone will defend the position of another if they go out of position, so everyone is defended by everyone in our team.

In our example: the admissible sets are: {} , {a1} , {a3} , {a1, a3}

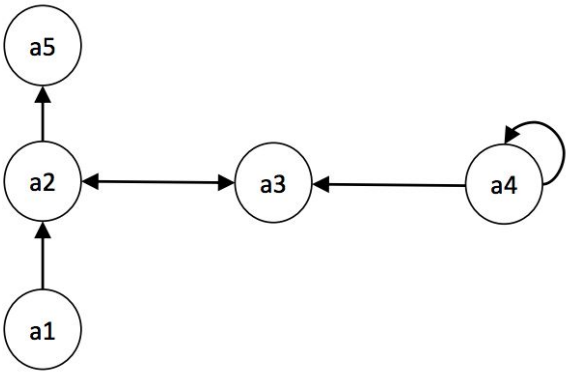
- {} - nothing to defend and no elements so no conflicts
- {a1} - the only thing that attacks a1 is a2 and a1 attacks it so it's defending itself and cannot have conflict since there is only 1 element.
- {a1, a3} - nothing attacks a3 so nothing has to defend it, a1 is attacked by a2 and a3 attacks a2 and a1 and a3 don't attack each other.

How can an argument attacks itself? *Think of saying "there is no truth", if there is no truth how can I know this is true?*

Complete extensions

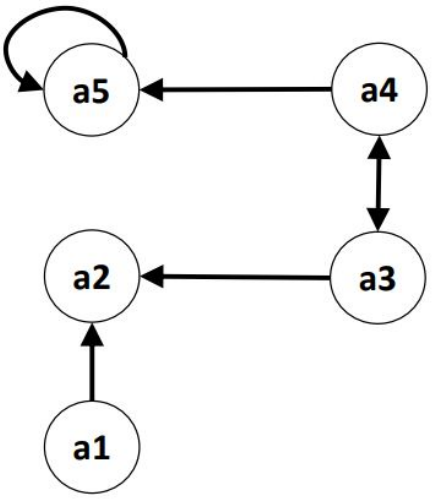
Complete semantics: aims to include in a set all arguments that the set can defend.

Complete extension: admissible set that includes **all** arguments the set can defend.
A set is a complete extension if the set is conflict free and every element in our set and every element is defended by the people in our set.
*Question we can make to know if it is an empty set : "**Does it contain everything that it defends?**"*



- E = {a1, a5} is a complete extension, since it is admissible and it defends both a1 and a5. It does not defend anything which is not in the set, thus it is a **complete extension**.
- *There are no arguments that the set defends that are in the set.* The set {a1} is not a complete extension, since it does not include a5, which it defends.
- Note that a3 is not defended by E, since there is no argument in E that attacks a4, and a4 cannot be part of a conflict-free set.

Example



Admissible subsets:

$\{\}, \{a1\}, \{a3\}, \{a4\}, \{a1, a3\}, \{a1, a4\}$

Complete extensions:

$\{a1\}, \{a1, a3\}, \{a1, a4\}$

Explanation:

Admissible	Complete Extension
{} - conflict free and has no members it needs to def	{} - it defends a1 since “nothing” attacks a1 and it doesn’t contain it.
{a1} - conflict free and nothing attacks it	{a1} - it does not defend anything is a **CE**
{a3} - conflict free and a4 attacks it but it defends itself.	{a3} - defends a5, but doesn’t include it
{a4} - conflict free and a3 attacks it but it defends itself.	{a4} - it does not defend a2 because it does not attack a1 although it attacks a3, but it defends a1 because nothing attacks a1 but it doesn’t include it.
{a1, a3} - conflict free, nothing attacks a1 and a4 attacks a3 but it defends itself	{a1, a3} - is a **CE**
{a1, a4} - conflict free, nothing attacks a1 and a3 attacks a4 but it defends itself	{a1, a4} - is a **CE**

The empty set is a complete extension if there is no arguments that are not attacked by anything. Because if every argument has at least one attacker, the empty set cannot defend any of those arguments.

Maximal and minimal subset

Maximal

A member of a collection of sets is said to be maximal if it cannot be expanded to another member by addition of any element. We must arrive to a point where we have no strict supersets.

Take $C \subseteq 2^S$, a maximal subset of S in C is a set $T \in C$ such that no other set in C strictly includes T.

Formally, T is a maximal subset of S in C if and only if $\nexists T' \in C$ such that $T \subsetneq T'$.
I.e.
Take S = a{1, a2, a3} and let C = {{a1} , {a2} , {a1, a2} , {a2, a3}} .
The maximal subsets of C in S are {a1, a2} and {a2, a3}.

Question we can ask: ***can we add some element in the set o the current element to create an element in the set? If so that is an element of our maximal subset.***

Minimal

The same as above but the opposite, the minimum representation of all the elements in the set. We must arrive to a point where we have no strict subsets.

The minimal subsets of C in S are {a1} , {a2}. *If empty set then they would not be minimal subsets.*

Rule: if the empty set is something we are considering, it will always be the minimal subset.

Question we can ask: ***can we take away something from the element that will yield some other element in the set?***

ULTIMATE TIP: *ALWAYS ASK THE QUESTIONS, WHAT DOES NOTHING ATTACK? And WHAT ATTACKS NOTHING, MAKE A NOTE ABOUT IT or even draw it in your diagram.*

Because we ask the questions:

- what attacks a_1? Nothing "{}"
- what does a_4 attack? Nothing "{}"

* Remember that trivially {} attacks everything, this is why it is never present in any other set than by itself in the conflict free sets.

  ie {{}, a_2} is not conflict free bc {} trivially attacks everything.

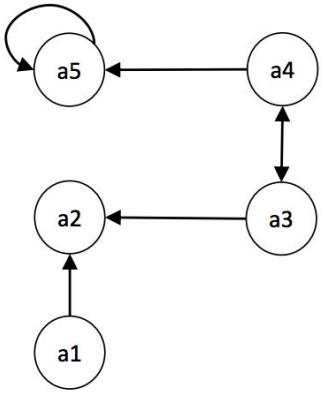
  If we include all of those, the graph gets too complicated.

Grounded Semantics

The idea that we only accept what is not controversial.

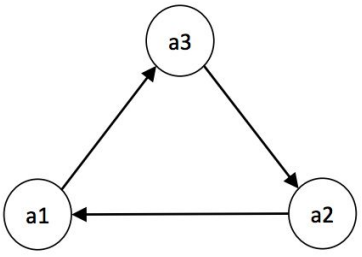
Grounded Extension

Grounded extension: is the minimal subset of the set of complete extensions.



Example
Complete extensions: {$a1$} , {$a1$, $a3$} , {$a1$, $a4$}
Grounded extension: {$a1$}

There is always exactly **one** grounded extension. It can be empty.



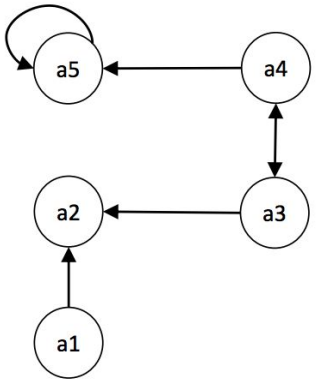
Example.
Conflict-free subsets: {}, {$a1$} , {$a2$} and {$a3$} .
Admissible subsets: {}
Complete extensions: {}
Grounded extension: {}

Preferred Semantics

Preferred semantics try to maximise the acceptance of arguments. We can think of it as it “accepts as much as you can defend”

Preferred Extension

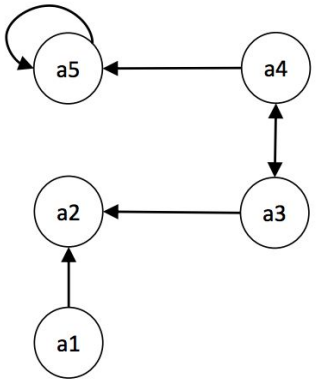
Prefered extension: is the maximal subset of the set of complete extensions.



Example.
Complete extensions: {$a1$} , {$a1$, $a3$} ,{ $a1$, $a4$} .
Preferred extensions: {$a1$, $a3$} , {$a1$, $a4$} .

Stable Extension

Preferred extension that attacks everything that is not in the extension.
Formally, a preferred extension E such that for all $y \in S \setminus E$ there exists $x \in E$ such that $x, y \in R$ (in other words, for every argument y that isn't part of E, there is an argument in E that attacks y).
It doesn't always exist



Example.
Preferred extensions: {a1, a3} , {a1, a4} .
Stable extensions: {a1, a4}.

A1 attacks a2 and a4 attacks a5 and a3 therefore it attacks everything in S which is not in the preferred extension.

Credulous Acceptance

An argument is **credulously accepted** by an argumentation framework under a particular semantics if and only if it is part of at least one of the extensions generated by those semantics

I.e. an argument is credulously accepted under the **complete** semantics if and only if it is part of at least one **complete** extension.

Skeptical Acceptance

An argument is **skeptically accepted** by an argumentation framework under a particular semantics if and only if it is part of all of the extensions generated by those semantics

I.e an argument is skeptically accepted under the **complete** semantics if and only if it is part of all of the **complete** extensions.

- Credulous - at least one
- Skeptical - all

Summary

- Conflict Free - no arrows between the elements.
- Accessible - CF and everything in the set is defended.
- Complete Extension - accessible and it includes everything that it defends
- Grounded Extension - minimal subset of CE - exactly one
- Preferred Extension - maximal subset of CE
- Stable Extension - preferred extension that attacks everything that is not in it

Lecture 8 Argumentation II

Problems to do with computation of semantics of argumentation frameworks can be divided into enumeration (ie finding complete sets) or decision (ie given arguments decide the best outcome). The approaches we use to do such things are either **reduction-based** or **direct** approaches

Reduction-based approaches translate our arguments and then use some highly research and developed frameworks to solve it, then they are translated again.

Direct approaches don't undergo translation, rather they solve the argument directly.

Equational Approach for the Complete Semantics

The equational approach **translates the argumentation problem to a set of equations such that solutions** to those equations directly represent solutions to the original argumentation problem.

Consider a function f that maps each argument to a number between 0,1 and a set of equations:

$$f(a_1) = 1 - \max(f(x_1^1), \ldots, f(x_{n_1}^1))$$
$$\vdots$$
$$f(a_m) = 1 - \max(f(x_1^m), \ldots, f(x_{n_m}^m))$$

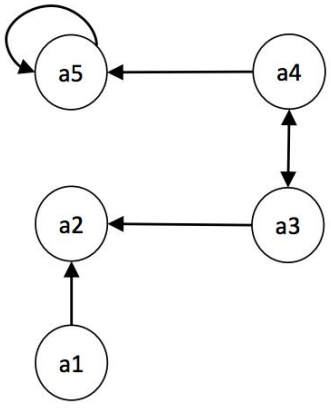
where $\{x_1^i, \ldots, x_{n_i}^i\}$ are all the arguments that attack a_i.

The function of an argument a_m is 1 - max(its attackers)

Solutions to this system will assign (with the function f) a number to each variable a_i (which corresponds to an argument).

For each solution, the arguments that its function equal **1** can be joined to create a complete extension.

Example



Equations

$f(a1) = 1 - \max\{\} = 1$

$f(a2) = 1 - \max\{f(a1), f(a3)\} = 0$

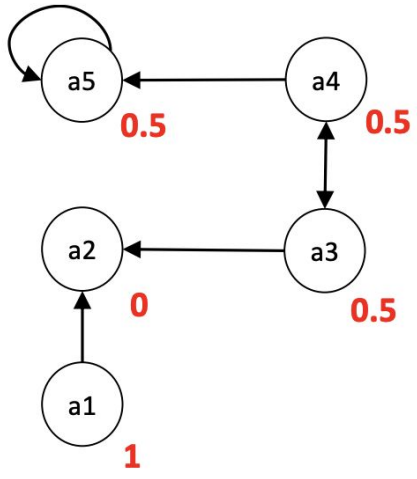
$f(a3) = 1 - \max\{f(a4)\} = 0.5$

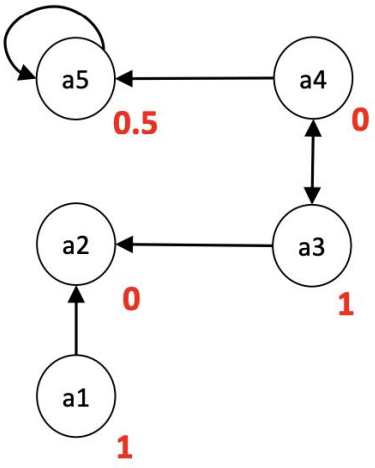
$f(a4) = 1 - \max\{f(a3)\} = 0.5$

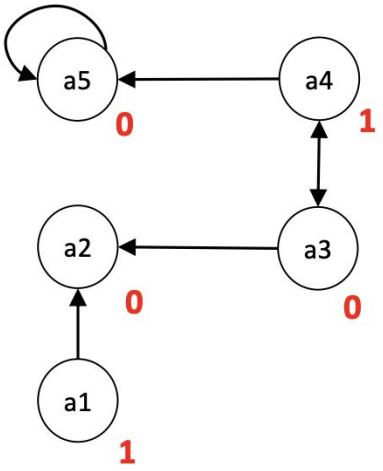
$f(a5) = 1 - \max\{f(a4), f(a5)\} = 0.5$

We are assuming the values for a4 and a3 such that f(a3) + f(a4) = 1

*We can assign any numbers we want but the equations **must hold**.*



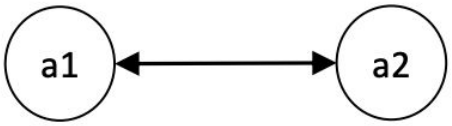




As we can see we can give each argument different values and they all give us valid equations. But each solution yields different complete extensions:

{a1} {a1}, {a1,a3} {a1}, {a1,a3}, {a1,a4}

Example with no Complete Extensions



Equations

$$f(a1) = 1 - \max\{f(a2)\}$$
$$f(a2) = 1 - \max\{f(a1)\}$$

There are no complete extensions here if we do not give a1 or a2 a value of 1,0 or vice versa

Or we could have any solution that satisfies the following:

$$f(a1) = x$$
$$f(a2) = 1 - x$$

where $0 < x < 1$

With this strategy we have explored an **enumeration approach** in which we want to find out which are the complete extensions.

Argument Game Approach

In this case we look at different arguments and we look to make a decision. We look at whether a given argument is part of an extensions. We'll look at an approach for the grounded semantics.

- **Argument game** between **proponent** P and **opponent** O
- Proponent starts with argument in question
- Each party replies with suitable attacker (following the rules)
- We have some winning criterion
- If P can always win then the argument is in the extension:
    - Means that there is a way P can play such that no matter what O does P will win, P has a winning strategy.

A **game tree** represents all the possible games that can be played according to the rules of the game

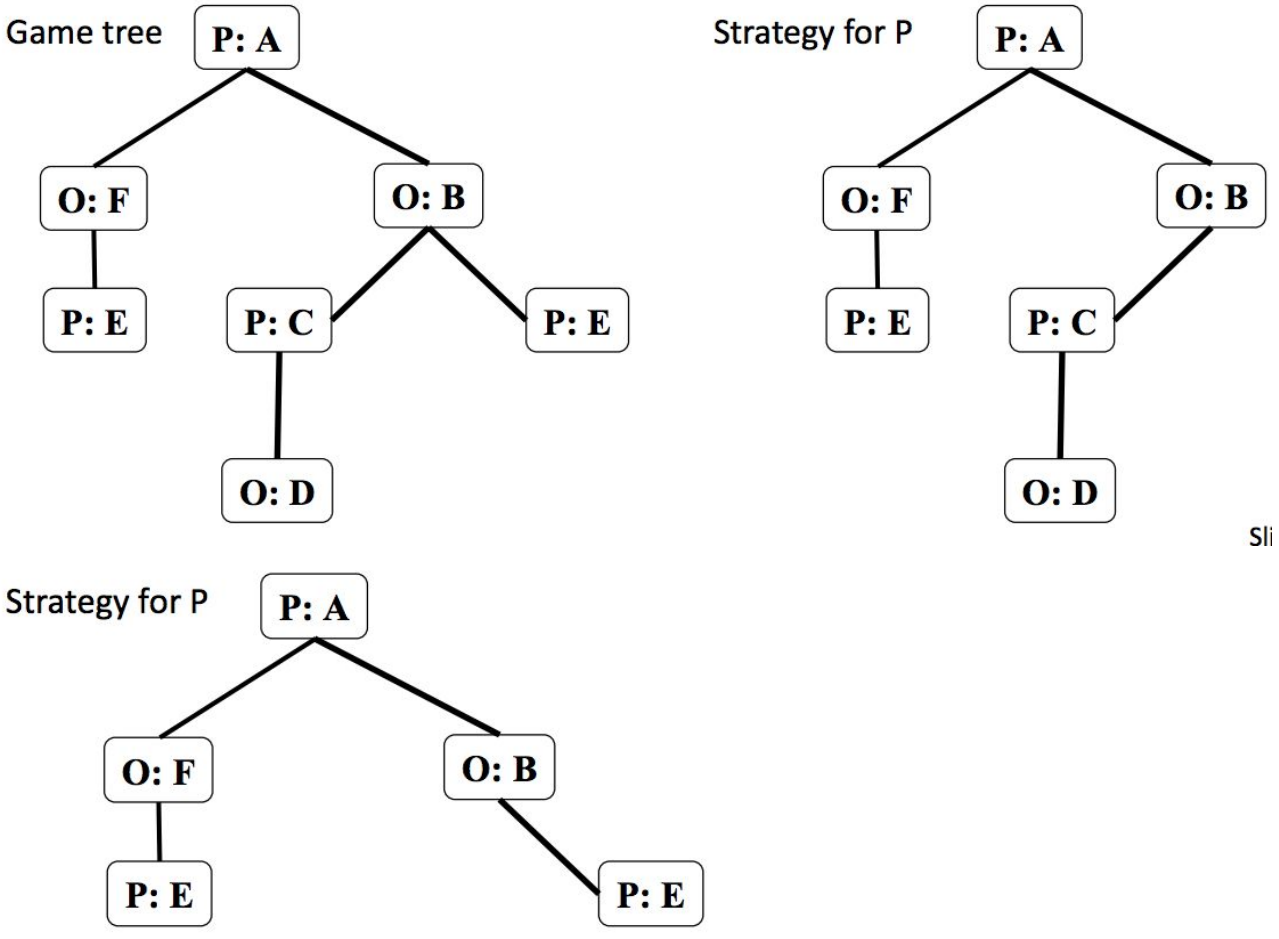
- Every branch is a **dispute** (sequence of allowable moves)
- The game tree contains all possible disputes

A **Strategy** for player p is a partial game tree that tells us everything the player p should do in every situation:

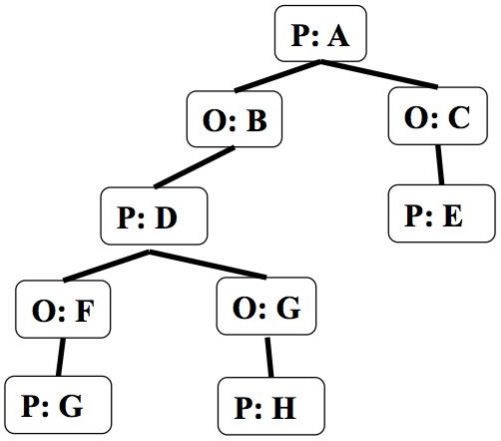
- Every branch is a dispute
- The tree only branches after moves by p
- The children of p's moves are all the legal moves by the other player

There are two strategies for P

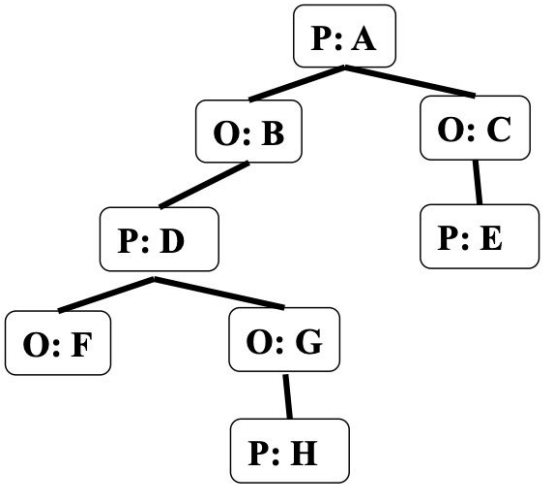
In order for something to be a strategy for P, everything must branch under P. If it's a strategy for O, everything must branch under O.



If P **can always win** , then the argument is in the extension. It means that there is a way **P** can play that no matter what **O** does, **P** always wins. P has a **winning strategy.**



This **is** an example of **a winning strategy** for the proponent. If the last argument is not one by our proponent then it is **not** a winning strategy since there is a possibility we could end up at a point where O wins.



This is not a winning strategy for P, given that P has no response if O plays F.

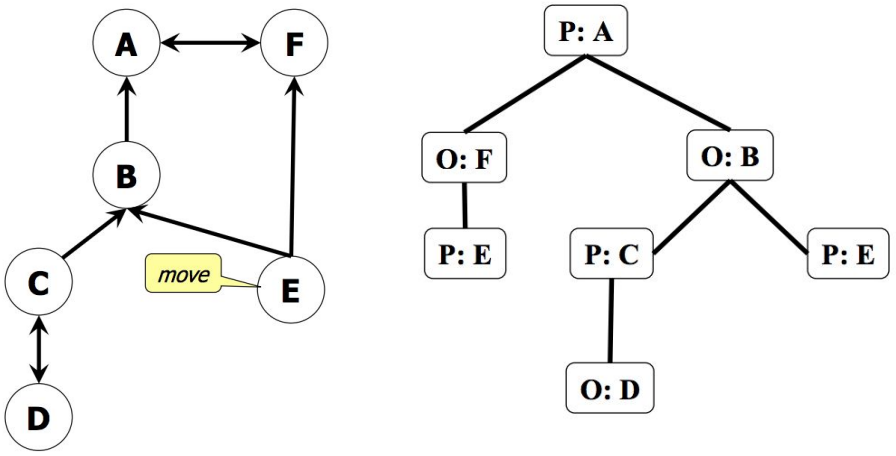
Argument game for grounded semantics (rules of the game)

- Each move must reply to the previous move (backtracking not allowed)
- Proponent can't repeat moves
- Proponent moves strict attackers (can't be bi-directional attack [double arrow])
- Opponent moves attackers (bi-directional attack ok [double arrows are ok])
- A player wins if and only if the player can't move

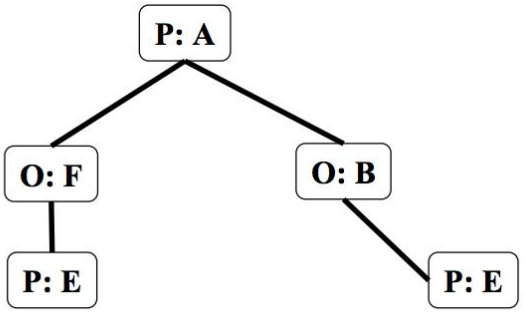
Given the rules, argument A (initial argument) is in the grounded extension if and only if the proponent has a winning strategy for the game that starts with the proponent moving A first.

Example

Game tree (on the right) captures all allowed disputes



This would be the **winning strategy for P**



If the proponent has a **winning strategy**, then the initial argument is in the complete extension.

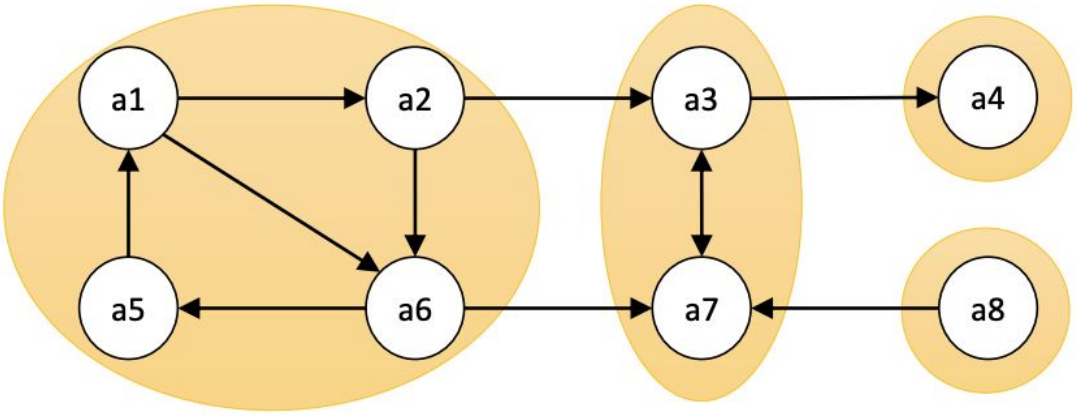
Decomposition Approach

This is another direct approach. The idea is to decompose an argumentation framework into a set of sub-frameworks and compute the semantics of each sub-framework locally.

We need to perform the decomposition so that the resulting sub-frameworks respect the dependency of the attack relation and form an acyclic graph.

We do this by collecting arguments together into **strongly connected components**

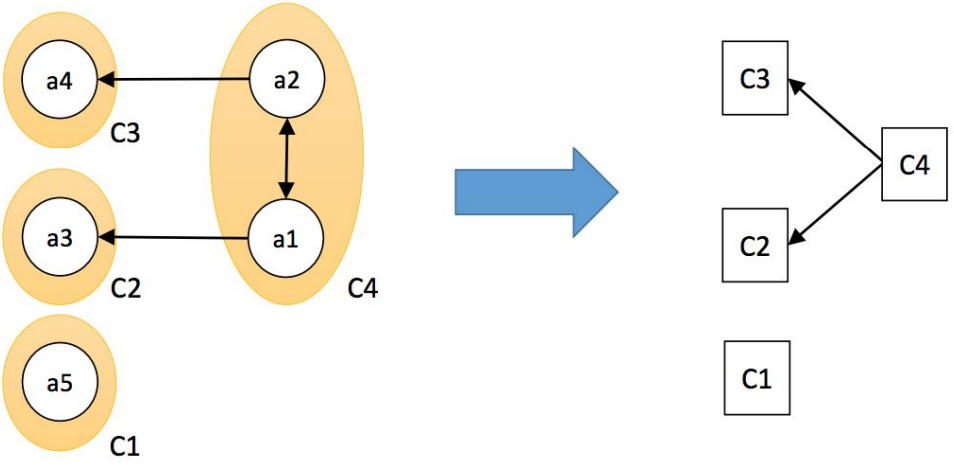
- Two arguments a and B are in the same strongly connected component if and only if there is a path from A to B and a path from B to A (using the attack relation)



Strongly connected components are: {a_1, a_2, a_5, a_6}, {a_3, a_7}, {a_4}, {a_8}

We have to make sure that if we go somewhere, we can get back to it for it to be a strongly connected component.

We can create a representation of our strongly connected components by naming each strongly connected component (C_i) and draw lines to see how these are connected, this is called a **condensation graph**. To get the **condensation graph** we take the strongly connected components (SCC) and add an edge from CX to CY if there are arguments $x \in$ CX and $y \in$ CY such that x attacks y.



This creates an acyclic graph. These edges represent the dependencies between these SCCs.

We want to group the SCCs in a way that their partial order is preserved, to do so, we assign them each a **level**.
Let parents(C_i) be the set of parents of the SCC C_i
E.g., parents(C4) = {}, parents(C3) = {C4}

A **trivial** SCC is an SCC that contains **exactly one argument** that doesn't **attack itself**, here, the trivial SCCs are C1, C2 and C3.

We can find the level of an **SCC C_i** recursively:

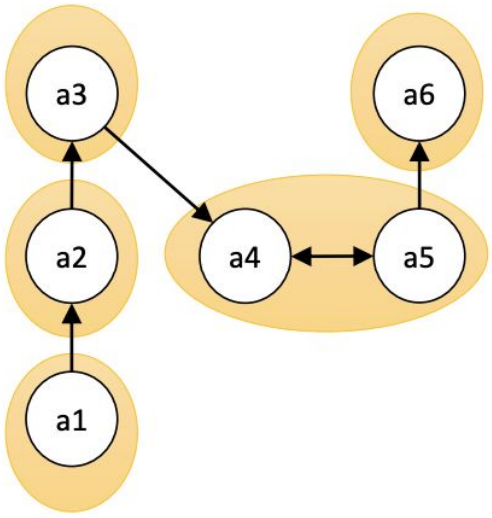
$$level(C) = \begin{cases} 0 & \text{if } parents(C) = \{\} \\ \max_{P \text{ in } parents(C)} \{level(P) + 1\} & \text{if C is non-trivial} \\ \max_{P,Q \text{ in } parents(C)} \{level(P) + 1 \mid P \text{ is non-trivial}\} \cup \{level(Q) \mid Q \text{ is trivial}\} & \text{if C is trivial} \end{cases}$$

In this case the level of C1 and C4 would be 0, and the level of C2 (trivial), C3 (trivial) would be 1.

In other words:

- If the SCC has no parents, then 0
- If the SCC is non-trivial then the maximum level of our parents + 1
- If the SCC is trivial then it is the maximum of (for it's non-trivial parents we take their level +1, for its trivial parents we take their level)

Example 2:



Exercise

Identify the level of each SCC.

$level(\{a1\}) = 0$

$level(\{a2\}) = 0$

$level(\{a3\}) = 0$

$level(\{a4,a5\}) = 1$

$level(\{a6\}) = 2$

$$level(C) = \begin{cases} 0 & \text{if } parents(C) = \{\} \\ \max_{P \text{ in } parents(C)} \{level(P) + 1\} & \text{if C is non-trivial} \\ \max_{P,Q \text{ in } parents(C)} \{level(P) + 1 \mid P \text{ is non-trivial}\} \cup \{level(Q) \mid Q \text{ is trivial}\} & \text{if C is trivial} \end{cases}$$

By computing their level we can establish an order depending on their level:

Layer 0: {a1}, {a2}, {a3}
Layer 1: {a4,a5}
Layer 2: {a6}

Looking at the Grounded Semantics with a Decomposition Approach

We can use this to find the **grounded extension**. Remember that the grounded extension aims to accept whatever it is uncontroversial.

- Starting from Layer 0, we see that {a3} depends on {a2} which depends on {a1}. Since a1 is not attacked **it is in the grounded extension.** Since a1 is in the GE, a2 cannot be since it must be conflict-free. But a3 is uncontroversial, since its attacker a2 is defeated by a1, which is uncontroversially in the grounded extension.

Partial GE = {a1,a3}

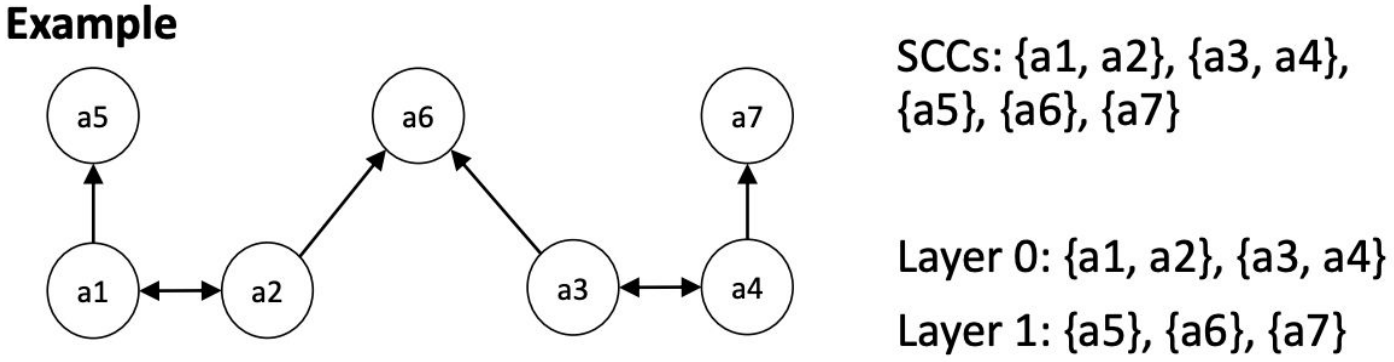
- Layer 1: we see that a4 depends on a3, but since a3 is in the GE we cannot add a4. Therefore it is uncontroversial to add a5 since its attacker is defeated by a3, which is in the extension.

Partial GE = {a1,a3, a5}

- Layer 2: a6 is dependent on a5, but a5 is already in our GE, therefore a6 cannot be (because it's not conflict-free).

Grounded Extension = {a1, a3, a5}

Looking at the Preferred Semantics with a Decomposition Approach



We can compute the semantics of each SCC within a layer independently, then we need to combine the results within a given layer, taking into account any dependencies from the layers below:

- {a5} depends on a1 from layer 0.
- {a6} depends on a2 and on a3 from layer 0.
- {a7} depends on a4 from layer 0.

Layer 0, SCC {a1, a2}, preferred extensions: {a1}, {a2}
Layer 0, SCC {a3, a4}, preferred extensions: {a3}, {a4}

Remember that the preferred extensions look to include everything that is possible but we cannot add {a1, a2} as that would not be conflict free, same for the second example.

We now must combine our answers with layer 0 to get our possible partial extensions:

{a1,a3} , {a1,a4}, {a2,a3}, {a2,a4}

Now we look at layer 1:

We take our partial extension and see if we can add to it: {a1, a3} -> we can't have a5 or a6 because they depends on a1 and a3 respectively. But a7 does not depend on anything in our PE. Therefore: {a1,a3,a7}

Partial Preferred Extension: {a1,a3,a7}, {a1,a4}, {a2,a3}, {a2,a4}

- For {a1,a4} -> we cannot have a5 or a7 (will not make it conflict free) therefore {a1, a4, a6}
- For {a2,a3} -> {a2, a3, a5, a7}
- For {a2, a4} -> {a2, a4, a5}

Preferred Extensions are: {a1, a3, a7}, {a1, a4, a6}, {a2, a3, a5, a7}, {a2, a4, a5}

Lecture 9 Machine Learning

There is no agreed definition, broadly speaking **how to use data on past situations to learn how to act in the future**.

There are 3 classes of machine learning:

Supervised learning

- You start with a collection of examples for which you know the answer
- Use these to be able to identify correct answers on new instances

Unsupervised learning

- Same set of instances but no correct answers available
- We want to Identify patterns/relations in the data

Reinforcement learning

- You try to learn by doing things and occasionally getting rewards
- Need to associate actions with the rewards they bring

Supervised Learning (Classification)

Given a training set $(x_1, y_1), (x_2, y_2), \ldots (x_N, y_N)$
Where $y_i = f(x_i)$

But nobody tells us what the function is, our job is to discover a function: $h(x) \approx f(x)$

X can be any value or set of values.

- When y is a member of a finite set this is **classification**
- When y is a number this is **regression** (determining the strength of the relationship)

Our Example X1 may have different attributes, which may be of different types:

- Boolean,
- Discrete,
- Continuous,
- …

Classification of examples is **positive** (T) or **negative** (F). The point is to learn a function that given examples we can determine their classification.

Example Choosing to Wait or Leave at a Restaurant

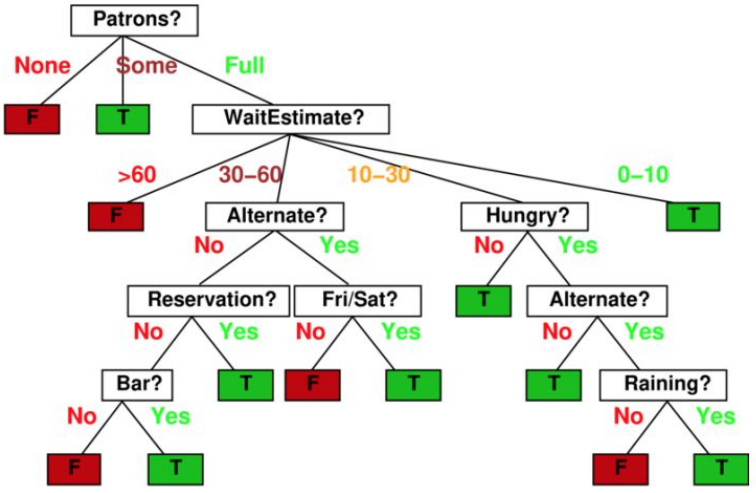
This is data from vists to a restaurant. For each visit we were offered the possibility to wait for a table:

Example	Attributes										Target
	Alt	*Bar*	*Fri*	*Hun*	*Pat*	*Price*	*Rain*	*Res*	*Type*	*Est*	*WillWait*
X_1	*T*	*F*	*F*	*T*	*Some*	*$$$*	*F*	*T*	*French*	*0–10*	*T*
X_2	*T*	*F*	*F*	*T*	*Full*	*$*	*F*	*F*	*Thai*	*30–60*	*F*
X_3	*F*	*T*	*F*	*F*	*Some*	*$*	*F*	*F*	*Burger*	*0–10*	*T*
X_4	*T*	*F*	*T*	*T*	*Full*	*$*	*F*	*F*	*Thai*	*10–30*	*T*
X_5	*T*	*F*	*T*	*F*	*Full*	*$$$*	*F*	*T*	*French*	*>60*	*F*
X_6	*F*	*T*	*F*	*T*	*Some*	*$$*	*T*	*T*	*Italian*	*0–10*	*T*
X_7	*F*	*T*	*F*	*F*	*None*	*$*	*T*	*F*	*Burger*	*0–10*	*F*
X_8	*F*	*F*	*F*	*T*	*Some*	*$$*	*T*	*T*	*Thai*	*0–10*	*T*
X_9	*F*	*T*	*T*	*F*	*Full*	*$*	*T*	*F*	*Burger*	*>60*	*F*
X_{10}	*T*	*T*	*T*	*T*	*Full*	*$$$*	*F*	*T*	*Italian*	*10–30*	*F*
X_{11}	*F*	*F*	*F*	*F*	*None*	*$*	*F*	*F*	*Thai*	*0–10*	*F*
X_{12}	*T*	*T*	*T*	*T*	*Full*	*$*	*F*	*F*	*Burger*	*30–60*	*T*

We have to find the function that given our attributes lead to true or false.

Below we see an example of the true function, given that someone has sat down and figured out which is the criterion with which they decide if they wait for a table or not. We will try to find an approximation of this function.

- Here is the “true” tree for deciding whether to wait:



You can make a tree based on the past experiences and perform a **table lookup** to see what will happen next,

But it probably won’t generalise to new examples unless there are very few actions that can be taken.

Decision trees are a good way to express these kinds of functions. An example is that it is easy to show for instance the xor property.

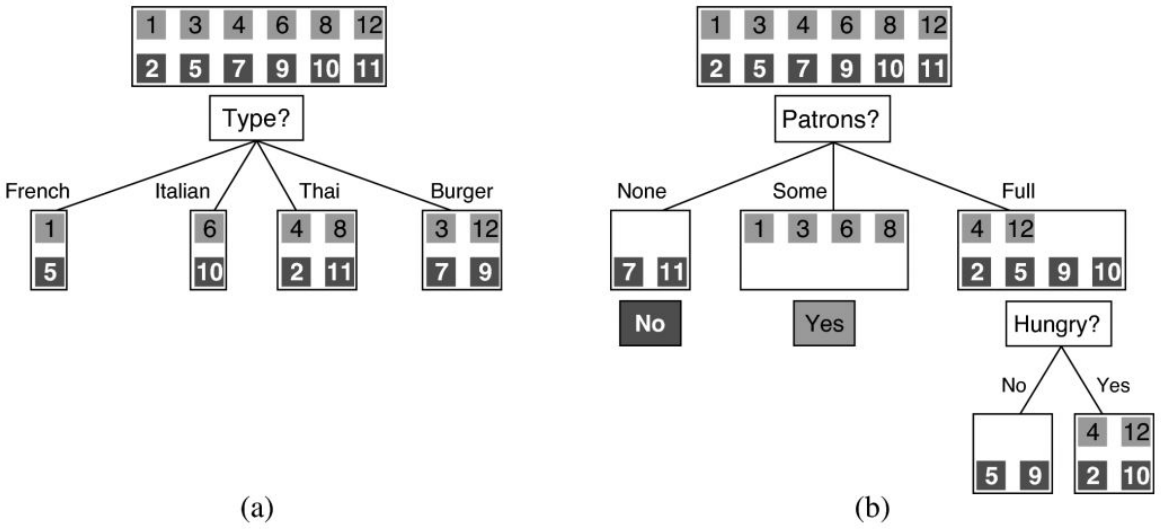
To find an answer (if we stay or don’t) we can do different things:

- We can look in the table (table lookup) which is efficient for examples with small number of attributes.
- The problem is that if we come up with something that was not on our original set of data, then we cannot describe it.

What we can do is we can create a summary of everything that is able to answer all the questions. We can create a more compact decision tree which is consistent with the training examples

Idea: (recursively) chose the most significant attribute as root of (sub)tree.

The light colored squares are positive examples and the dark ones are negative examples.



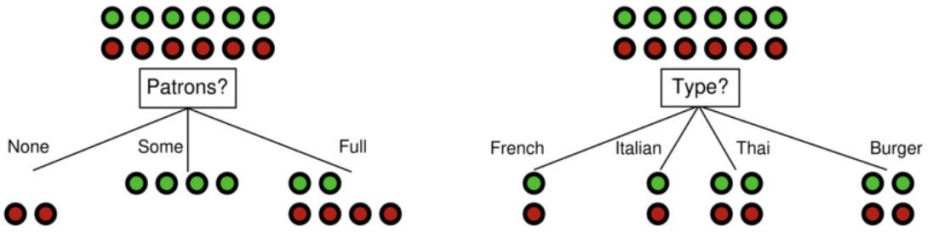
Attribute (a) Is not very significant because in each option there are the same number of T and F, in (b) on the other hand we have options in which we have only Fs and options with only Ts, giving **information** about the classification.

After picking an attribute 5 things can happen:

- All remaining examples are positive, in which case we are done
- All remaining examples are negative, then we are done too
- Some positive and some negative examples, then we choose another attribute for the subtree
- If there are no examples left then there are no examples that fit this case, which means we have no data on the specific case.
    - We return a best guess based on the parent node (**plurality classification**):
- If there are no more attributes but still have positive and negative examples
    - Can be due to noise (randomness in decision-making)
    - Can be due to unobserved attributes
    - We return **plurality classification**

Choosing and attribute

Idea: a good attribute splits the examples into subsets that are (ideally) all positive or all negative. If not the best attribute is the one that polarises this the most.



In this case patrons are the better decision, it gives more information about the classification.

Information

The more clueless I am about the answer initially, the more information is contained in the answer. *If you ask someone a question and already know the answer, then you have not discovered much information.*

- 1 bit of information is equivalent of the answer of a boolean when both possible options have the same probability of happening.

- Scale: 1 bit = answer to Boolean question with prior $\langle 0.5, 0.5 \rangle$
- Information in an answer when prior is $\langle P_1, \ldots, P_n \rangle$ is

$$H(\langle P_1, \ldots, P_n \rangle) = \sum_{i=1}^{n} -P_i \log_2 P_i$$

(also called entropy of the prior)

For a question with *n* answers.

To calculate $\log_2$ with our calculator we need to do (log in our calculator is really $\log_{10}$)

$$log_2(x) = \frac{\log_{10}(x)}{\log_{10}(2)}$$

If we have *p* positive and *n* negative examples in the root, then we need:

$$H\left(\left\langle \frac{p}{p+n}, \frac{n}{p+n} \right\rangle\right)$$

bits to classify a new example. Which tells us the proportion of positive to negative examples.

- For our 12 restaurants p = n = 6 so we need 1 bit.

We can also calculate the entropy for an attribute. An attribute splits the examples *E* into subsets E_i, each of which (we hope) needs less information to complete the classification

Each of these subsets is represented by a new branch

Let E_i have p_i positive and n_i negative examples.

$$H\left(\left\langle \frac{p_i}{p_i + n_i}, \frac{n_i}{p_i + n_i} \right\rangle\right)$$

bits needed to classify a new example

At some point we will want to calculate the value of an attribute which has branched out. We can calculate the best attribute by looking at its entropy and to calculate that, for each branch we calculate the entropy at that

point and then we multiply it by the proportion of the examples that are classified by that branch. We add that up for every branch.

$$\sum_i \frac{p_i + n_i}{p + n} H\left(\left\langle \frac{p_i}{p_i + n_i}, \frac{n_i}{p_i + n_i} \right\rangle\right)$$

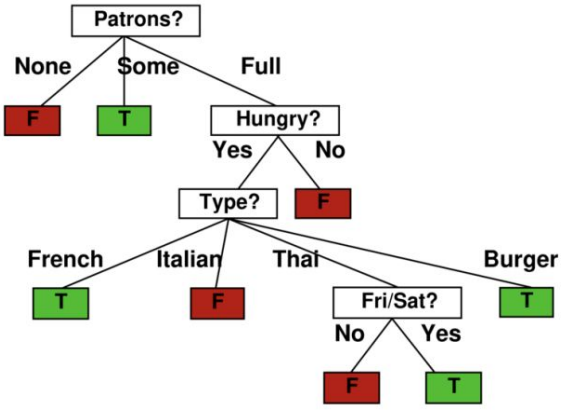
We can use this value to pick the best attribute.

I.e. For patrons the expected value is 0.459 bits, for Type its 1 bit -> so Patrons is better than Type.

In general, choose the attribute that minimizes the expected remaining information needed. You look at the value before, after and then you pick the one that gives you the biggest change.

Using our equation we can create a simpler tree:

Decision tree learned from the 12 examples:

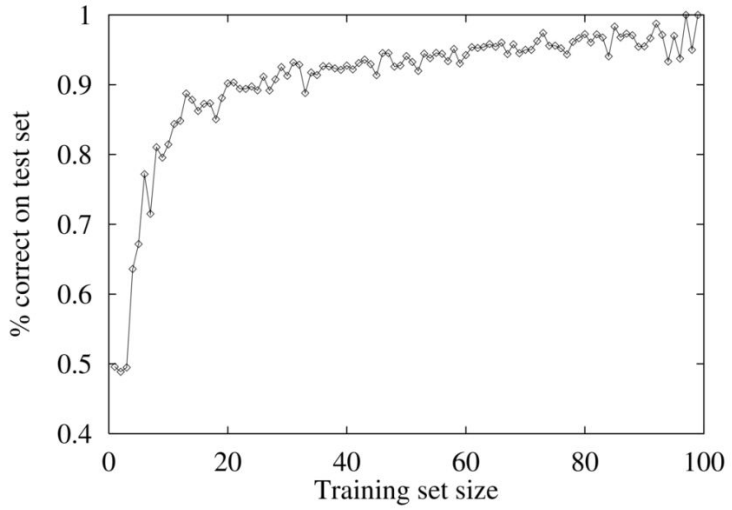


- This algorithm is a version of the **CART Algorithm.**
- If we use entropy to make the choice about what attribute to choose we get the **ID3 Algorithm**
- **C4.5 Algorithm** is similar, but adds the ability to handle unknown values, and does some post-processing to simplify the tree.

What we have learnt can be used to decide what to do, for instance you might use a classifier to control robots given it having wheels and sensors.

We have to make sure that our *h* is actually approximate to *f*, in other words, is my classifier a good classifier. The way in which we make sure is we give it a new set of **test samples** which has the same distribution over examples space as training set and we look at what percentage of the test sets are correct.

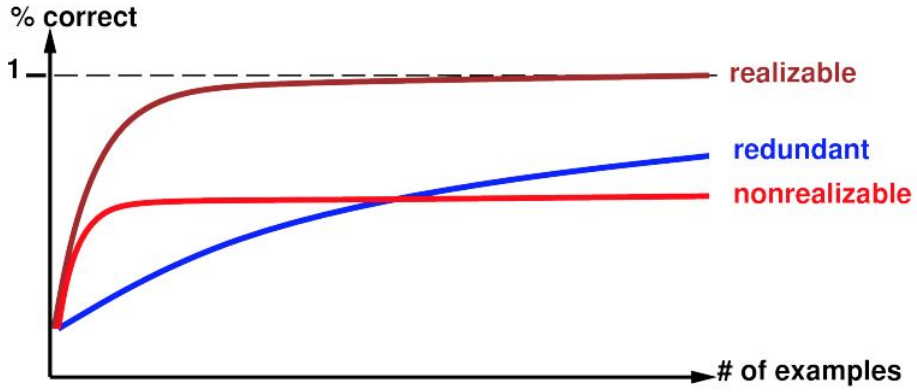
We get something like this:



We start at 0.5 since we are guessing at the start (for boolean values) and then, the larger the training set the more accurate.

Learning Curve

% correct on test set as a function of training set size



Learning curve depends on

- **Realisable (or learnable)** vs **non-realisable (you can't learn this thing)**, can be due to missing attributes or restricted hypothesis class
- **Redundant expressiveness**, e.g. loads of irrelevant attributes

Testing our model in this way is called **holdout cross-validation**

- It doesn't use all the data
- Depending on how we split our data, we can introduce bias to the results. Choosing a different set, we might get a better or worse classifier.
- Because you are taking some data away when you split, you might be creating a worse classifier.

A better option is **k-fold cross validation**

- Split data into k equal subsets. Learn on k-1 sets and test each result on the remainder (repeat k times)
- Average test score is a better estimate of the error rate than a single score
- Common values of k are 5 and 10, both giving error estimates that are very likely to be accurate

In the extreme case in which k = n, the number of data points we call is **Leave-one-out cross validation.**

Over-fitting: when a model is created which is too specialised for that data and it classifies the training data very well but the rest of data quite badly. This is one thing that cross-validation checks for.

Unsupervised Learning (Clustering)

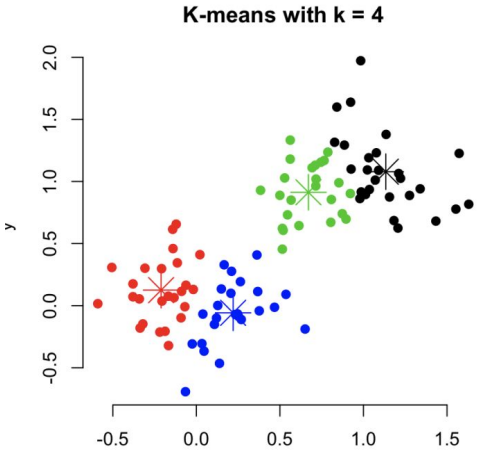
We have the same set of instances as supervised learning but we don't have the classification.

- Look for patterns in the data
- **Clustering** is breaking the data down into groups

Clustering

Given a set of examples X = {x_1, …, x_n }

- Identify K different groups.
- Each x_i should be in the group that it matches most closely
- Identify each group by its centre, mean, μ_j
- The clusters z^i are then made up of the points closest to the μ_j



K-Means

Is an algorithm to find the clusters in the data.

- Pick random values for μ_k.
- Then repeat:
  1. Assign each x_i to its closest cluster centre:

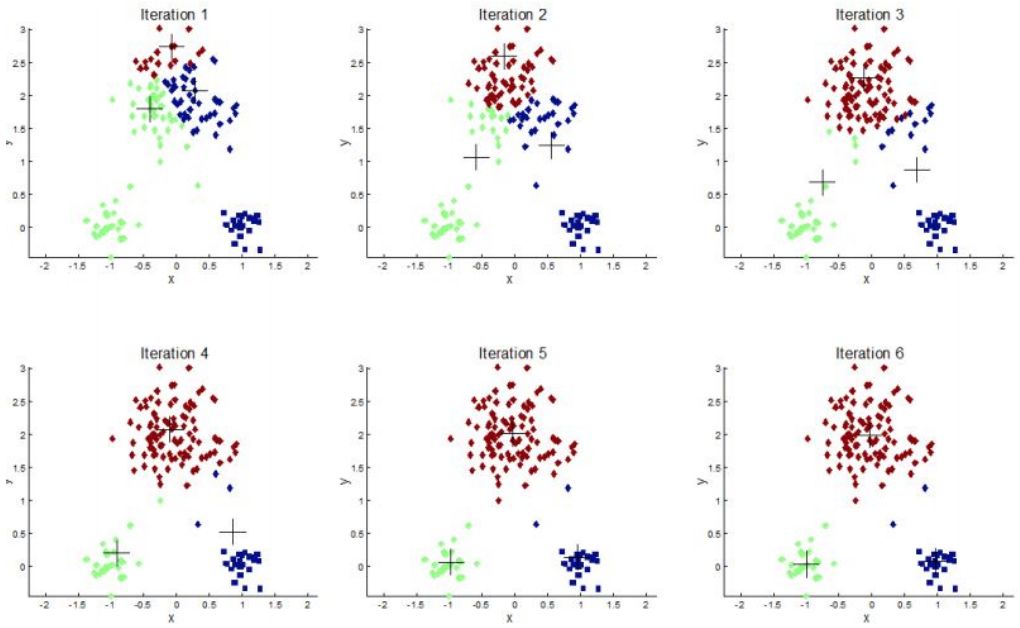
$$z^i = \arg\min |x_i - \mu_k|$$

  2. Update each cluster centre as the mean of the points assigned to that cluster:

$$\mu_k = \frac{1}{|\{j : z^j = k\}|} \sum_{i \in \{j : z^j = k\}} x_i$$

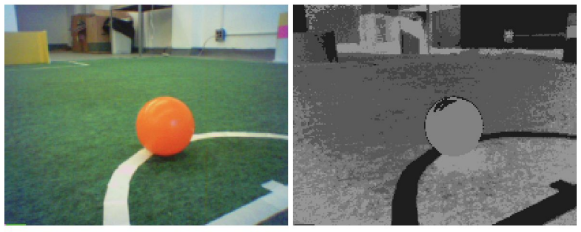
until converged.

We go through the data and we find the closest cluster center that that point is to, then we update the cluster center to be the average of the points in the cluster until we converge.



- There is no guarantee that clusters will converge
- Can get poor clusterings
- Re-run with random restart
- Or use kmeans++ to pick good initial cluster centers

Example:

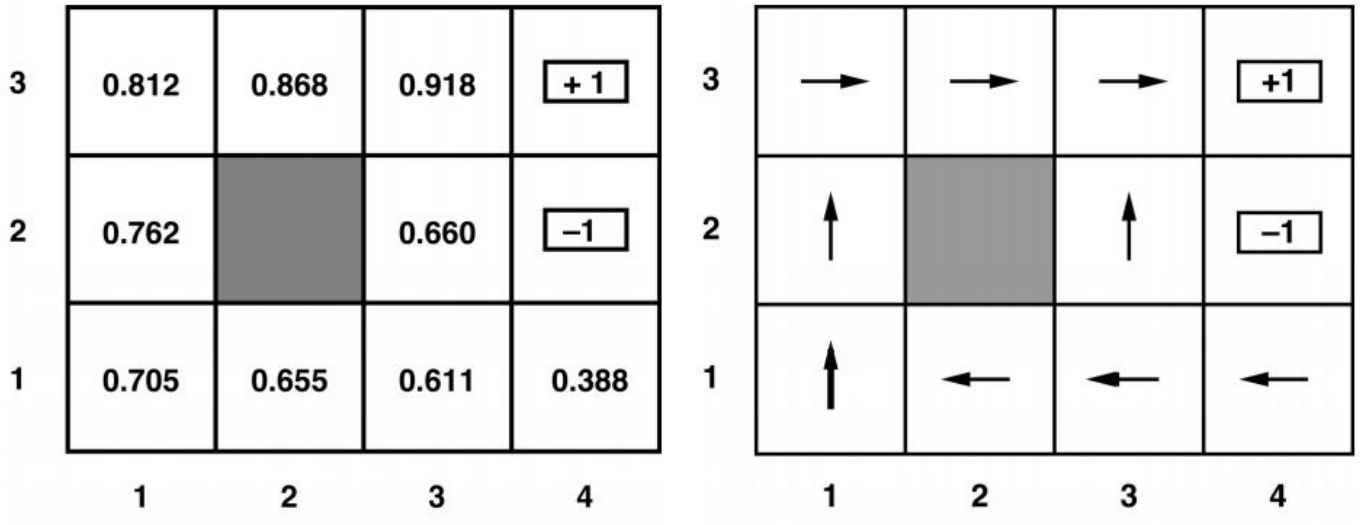


Cluster pixels depending on their RGB value allows us to spot the ball pretty well.

Reinforcement Learning

What happens when we don't have any examples? But we know when we succeed or fail.
This is the domain of **reinforcement learning** (**RL**)

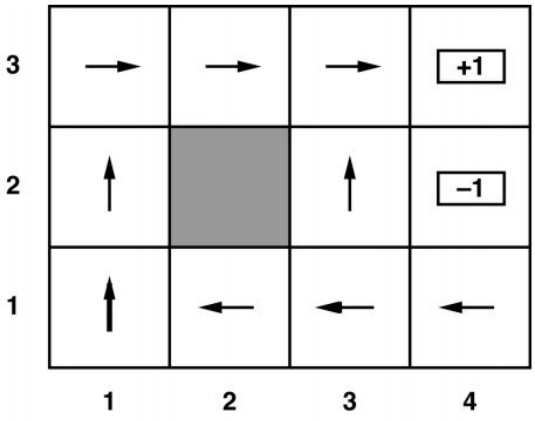
Since actions are non-deterministic, a natural framework to study this is Markov Decision Processes like this world bellow that we solved with an MDP



Now lets imagine our agent doesn't know the transition model P(s' | s,a) nor the reward function R(s)

It needs to **learn** the transition model and reward

Passive learning



To start off we need a policy, something that tells the agent what to do in each state.

Agent learns Uπ (s) by carrying out runs through the environment, following some policy π.

In **passive reinforcement learning** the agent's policy is fixed, he doesn't make a choice about how to act.

- The utility $U^{\pi}(s)$ of a state s under policy π is the expected sum of the (discounted) rewards obtained when following π.

$$U^{\pi}(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(S_t)\right]$$

  where S is the state reached at t from s when executing π.
- So if we run the policy for long enough, you will compute the utility of the states from the onward rewards.

To find the utility of a state we can do two things:

- Either think about all the possible things that may happen in the future and add those up

- Do a lot of runs and average the outcome

In reinforcement learning we do these runs, we add up the utility of each run and once we have done enough runs we have a good estimate of the utility of that path. This is called **Direct Utility Estimation.**

Direct utility estimation

We can estimate the utility of a state by the rewards generated along the run from the state, each run gives us one or more samples for the reward from a state.

- Given the run:

$$(1,1)_{-0.04} \rightarrow (1,2)_{-0.04} \rightarrow (1,3)_{-0.04} \rightarrow (1,2)_{-0.04} \rightarrow (1,3)_{-0.04} \rightarrow (2,3)_{-0.04} \rightarrow (3,3)_{-0.04} \rightarrow (4,3)_{+1}$$

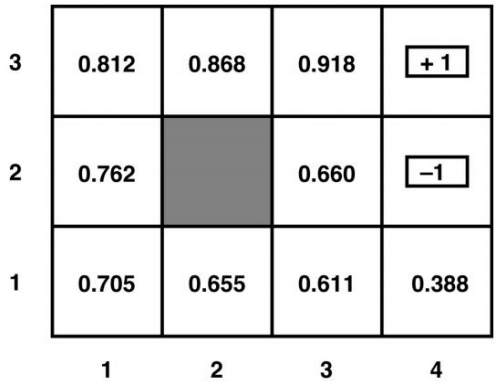
  a sample reward for $(1,1)$ from the run above is the sum of the rewards all the way to a goal state.
  - 0.72 in this case.
- The same run will produce two samples for $(1,2)$ and $(1,3)$.
  - 0.76 and 0.84
  - 0.8 and 0.88
- (Here we set the discount to 1).
- As the agent moves it can calculate a sample estimate of $P(s'|s,\pi(s))$
- Each time it moves it creates a new sample for one state.
- Given:

$$(1,1)_{-0.04} \rightarrow (1,2)_{-0.04} \rightarrow (1,3)_{-0.04} \rightarrow (1,2)_{-0.04} \rightarrow (1,3)_{-0.04} \rightarrow (2,3)_{-0.04} \rightarrow (3,3)_{-0.04} \rightarrow (4,3)_{+1}$$

  we get:

$$\begin{aligned} P((1,2)|(1,1), Up) &= 1 \\ P((1,2)|(1,3), Right) &= 0.5 \\ P((2,3)|(1,3), Right) &= 0.5 \end{aligned}$$

Over time, the agent builds up estimates a list of states s_i in which each state has a utility estimate associated with it U(s), an action associated with it π(s) and each state action pair has a probability distribution P(S' | s, π(s)) over the states S' that it gets to from s by doing π(s). In the end we get a solution:



and $P(s'|s, \pi(s))$, for every s, s' for the given $\pi(s)$.

Once we have our estimates, then we find what to do by doing the maximum expected utility at each step.

- Since we have a fixed policy, we might have not been everywhere. This means that estimates may not be the best. This means that the policy must vary if we want to learn the full space.
- Treats utilities of states as independent, but we know that they are connected by the Bellman Equation
- Ignoring the connection means the learning may converge slowly

Another approach to utility estimation is **adaptive dynamic programming** (still doing passive reinforcement learning)

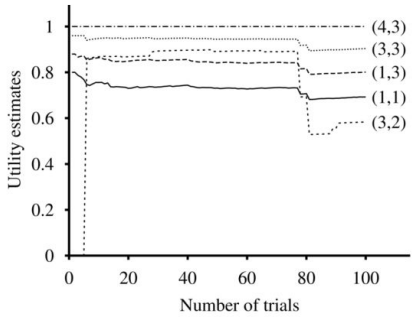
Adaptive Dynamic Programming

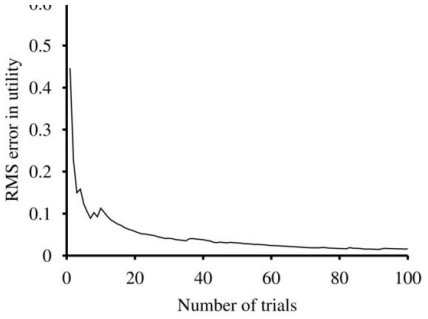
- Still passive learning so we have π.
- Since we are using the fixed policy version of the Bellman equation we don't have the max that makes the original hard to solve.
- We can just plug the results into an LP Solver
    - As discussed when talking about policy iteration
- Updates all the utilities of all the states where we have experienced the transitions
- Updated values are estimates and are no better than the estimated values of utility and probability

- Can also use value iteration to update the utilities we have for each state.
- Update using:

$$U_{i+1}(s) \leftarrow R(s) + \gamma \sum_{s'} P(s'|s, \pi(s)) U_i(s')$$

  to update utilities.
- Recall that we do this in modified policy iteration also.
- Back in Lecture 4 we called this "approximate policy evaluation".





After learning:

- To get the utilities the agent started with a fixed policy, so it always knew what action to take.
- It used this to get the utilities
- Having gotten the utilities, it could use them to choose actions
    - Just picks the action with the best expected utility in a given state
- However there is a problem with this, we might not have experienced the bad effects of an action

We must look at the trade-off between goals and exploration. It might be the case that we get somewhere quickly by running all the red lights but we have not yet experienced a crash, therefore it is difficult for us to know if we have a model which is good enough to let it work on its own.

Lecture 10 AI & Ethics

Future of Life Institute - Two main concerns

- **The AI is programmed to do something devastating**
    - E.g. autonomous weapons
- **The AI is programmed to do something beneficial, but it develops a destructive method for achieving its goal**
    - Is tasked with an ambitious geoengineering project, it might wreak havoc with our ecosystem as a side effect, and view human attempts to stop it as a threat to be met.

Asilomar AI Principles

Developed by participants at the Beneficial AI 2017 conference, organised by the Future of Life Institute

- **Safety -** All systems should be safe and secure throughout their operational lifetime, and verifiably so where applicable and possible
    - What do we mean by safe? How safe is safe?
- **Failure transparency** - if an AI system causes harm, it should be possible to ascertain why.
- **Judicial transparency** - any involvement by an autonomous system in judicial decision-making should be able to provide an explanation auditable by a competent human.
- **Responsibility:** Designers and builders of advanced AI systems are stakeholders must think about their use, misuse, and actions to see the safest way to develop the algorithms.

    - **EPSRC -** Anticipate: exploring the possible impacts of bringing in the technology, Reflect: on the purposes of the research, Engage: to bring in the wider stakeholders, Act: on these things.
    - **EU -** involve society in science and innovation. Connect R&I and society.
- **Value Alignment:** autonomous AI systems should align with human values. *We want to get to the airport as quickly as possible but we also want it to be have.*
- **Human Values:** AI systems should respect our human values.
- AI Systems should be capable of ethical reasoning and should be able to take into account input from humans
- **Shared Benefit:** AI technology should empower as many people as possible and contribute to economic prosperity.
    - We need to be careful we don't create biased AI. If we give AI biased data, we will get biased decision making.
- **Human Control**: Humans should choose how and whether to delegate decisions to AI systems, to accomplish human-chosen objectives