Lecture notes from the online Artificial Intelligence course tought by Peter Norvig and Sebatian Thrun fall 2011

Bjørn Remseth
rmz@rmz.no

January 13, 2012

Contents

.

Introduction

These are my notes from the course "Artificial Intelligence" taught by Peter Norvig and Sebastian Thrun, fall 2011.

Unfortunately I lost my notes for module one and two in a freakish accident involving emacs, to goats and a midget, so these notes start from module three.

I usually watched the videos while typing notes in LaTeX. I have experimented with various note-taking techniques including free text, mindmaps and handwritten notes, but I ended up using LaTeX, since it's not too hard, it gives great readability for the math that inevitably pops up in the things I like to take notes about, and it's easy to include various types of graphics. The graphics in this video is exclusively screenshots copied directly out of the videos, and to a large extent, but not completely, the text is based on Thrun and Norvig's narrative. I haven't been very creative, that wasn't my purpose. I did take more screenshots than are actually available in this text. Some of them are indicated in figures stating that a screenshot is missing. I may or may not get back to putting these missing screenshots back in, but for now the are just not there. Deal with it :-)

A word of warning: These are just my notes. They should't be interpreted as anything else. I take notes as an aid for myself. When I take notes I find myself spending more time with the subject at hand, and that alone lets me remember it better. I can also refer to the notes, and since I've written them myself, I usually find them quite useful. I state this clearly since the use of LaTeX will give some typographical cues that may lead the unwary reader to believe that this is a textbook or something more ambitious. It's not. This is a learning tool for me. If anyone else reads this and find it useful, that's nice. I'm happy for you, but I didn't have that, or you in mind when writing this. That said, if you have any suggestions to make the text or presentation better, please let me know. My email address is la3lma@gmail.com.

Source code for this document can be found at github https://github.com/la3lma/aiclassnotes, a fully processed pdf file can be found at http://dl.dropbox.com/u/187726/ai-course-notes.pdf.

1. Module three: Probability in AI

Rmz: *It is a really good idea to go over those initial modules again before the final exam. There will be questions, and it needs to be grokked.*

1.0.1. Introduction

Involved material. Bayes networks.

Why won't your car start?. Perhaps a flat battery. Perhaps the battery is dead, or it's not charging, which in turn be caused by the fanbelt or the alternator being broken. These set of relationships is a called a bayes network.

We can introduce things to inspect which can be used (lights, oi, gas, dipstick) etc. Causes affect measurements.

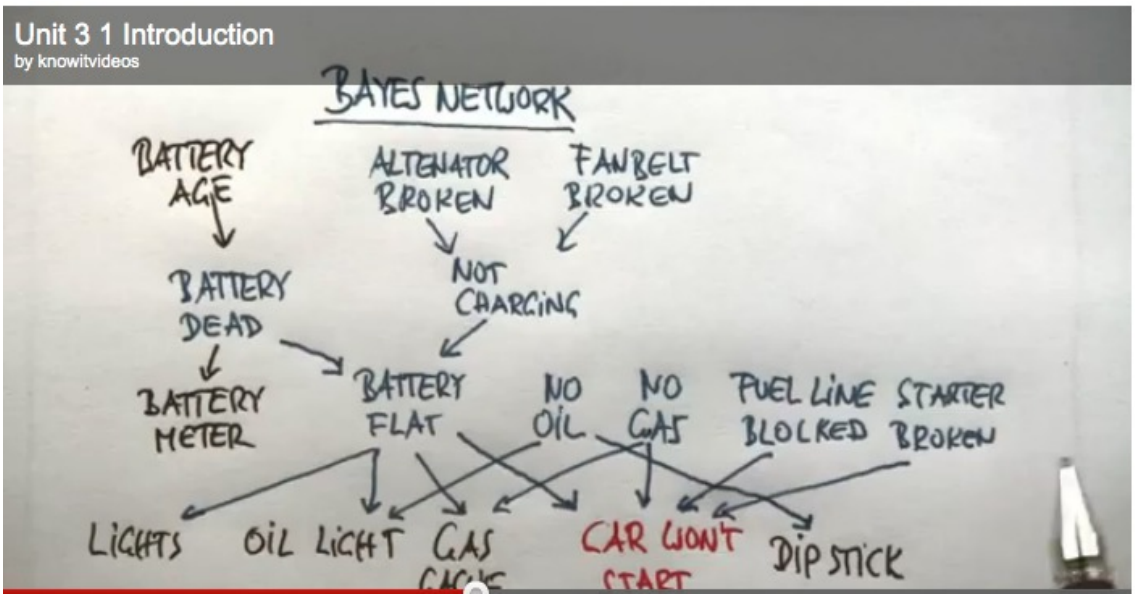


Fig. 1.1. A Bayes network from the lectures

Things we cannot measure immediately and measurements we can observe. The bayes network helps us to reason with hidden causes based on observable events.

A bayes network consists of nodes and arcs. The non-observable nodes are called random (or stochastic) variables. The child of a node is infered in a probabilistic way.

The graph structure and associated probability structure the network is a compact representation of a very large joint probability distribution.

With the network in place, we can compute probabilities of the random variables. We'll talk about how to construct bayes network and to use them to reason about non-observable variables

Assumptions:

- Binary events
- Probability
- Simple bayes networks
- Conditional independence
- Bayes neworks
- D-separation
- Paramemeter coount
- later: Inferences based on bayes networks.

B-networks is really imporant, used in diagnostics, predictions, machine learning. Finance, google, robotics. Also used as basis for *particle filters*, *hidden markov models*, *MDP*s and *POMDP*s, *Kalman filters* and others.

1.0.2. Probabilities

Used to express uncertainty. Given a coin with probability for coming u heads is $P(H) = 1/2$ the probaility of getting tails is $1/2$. The sum of the probabilities is always one.

Complementary probability (1-p).

1.0.3. Independence:

$$X \perp Y : P(X)P(Y) = P(X, Y)$$

where $P(X)$ and $P(Y)$ are called *marginals*.

1.0.4. Dependence:

Dependence:

Notation

$$P(X_1) = 1/2$$

For one coin we have $H : P(X_2 = H|X_1 = H) = 0.9$ for another coin we have $H : P(X_2 = T|X_1 = T) = 0.8$

The probability of the second coin being heads is 0.55. The calculation is $0.5 * 0.2 + 0.5 * 0.9 = 0.55$, and amazingly that took me several minutes to figure out, and I needed to draw a tree and do the sums to get the answer right. It's obvious I need this refresher course :-)

Lessons

The probability of some variable

$$P(Y) = \sum_i P(Y|X = i)P(X = i)$$

this is called the *total probability*. The negation of a probabilityu

$$P(\neg X|Y) = 1 - P(X|Y)$$

Cancer

One percent cancer in the population. Probability of a positive test if you have the cancer $P(+|C) = 0.9$. The probability of a negative result if you don't have the cancer, $P(+|C) = 0.1$.

The probability of the test coming out positive if you don't have the cancer is $P(+|\neg C) = 0.2$, the probability of the test being negative if you don't have the cancer is $P(-|\neg C) = 0.8$.

Join probabilities (probability of having a positive test with cancer and negative without (etc.)).

The Bayes Rule.

Thomas Bayes, presbytarian minister. Bayes Theorem is:

$$P(A|B) = \frac{P(A|B) \cdot P(A)}{P(B)}$$

The terms are: The *likelyhood* $= P(B|A)$, the *prior likelyhood* $= P(A)$, and the *marginal likelyhood* $= P(B)$. *the posterior* $= P(A|B)$.

The interesting thing is the way that the probabilities are reverted. We don't really care about the evidedence, we care if we have cancer or not.

The *marginal likelyhood* is in turn denoted as the *total probability* of B:

$$P(B) = \sum_a P(B|A = a)P(A = a)$$

In the cancer case we are interested in:

$$\begin{aligned} P(C|+) &= \frac{P(+|C) \cdot P(C)}{P(+)} \\ &= \frac{0.9 \cdot 0.01}{(0.9 \cdot 0.01) + (0.2 \cdot 0.99)} \\ &= \frac{0.009}{0.009 + 0.198} \\ &= 0.0434 \end{aligned}$$

The Bayes Rule can be drawn graphically. Two variables. B observable, A not. A causes B to be positive with some probability $P(B|A)$. What we care about is *diagnostic reasoning* which is the inverse of the *causal reasoning*. The graphical representation is a graph with an arrow going from A to B.

To fully specify a Bayes network with one observable and one non-observable, we need three parameters: The observed probability (of the observable), the conditional probability of the non-observable being true in the case of the observable being true, and the probability of the non-observable being true when the observable is false.

Computing bayes rule

Given the rule

$$P(A|B) = \frac{P(A|B) \cdot P(A)}{P(B)}$$

the denominator is relatively easy to calculate since it's just a product. The total probability (the divisor) is really hard to compute since it is a sum over potentially many terms. However, the denominator isn't dependent on the thing we're trying to find the probability for. If for instance we are interested in

$$P(\neg A|B) = \frac{P(\neg A|B) \cdot P(\neg A)}{P(B)}$$

we use the same denominator. We also know that $P(A|B) + A(\neg A|B) = 1$ since these are *compementary events*. This allows us to compute the bayes rule very differently by ignoring the normalizer.

First we calculate *pseudoprobabilities*

$$\begin{aligned} P'(A|B) &= P(B|A)P(A) \\ P'(\neg A|B) &= P(B|\neg A)P(\neg A) \end{aligned}$$

then we normalize by multiplying the pseudoprobabilities with a *normalizer* (eta)

$\eta = (P'(A|B) + P'(\neg A|B))^{-1}$

We defer calculating the actual probabilities:

$$\begin{aligned} P(A|B) &= \eta P'(A|B) \\ P(\neg A|B) &= \eta P'(\neg A|B) \end{aligned}$$

Two cancer

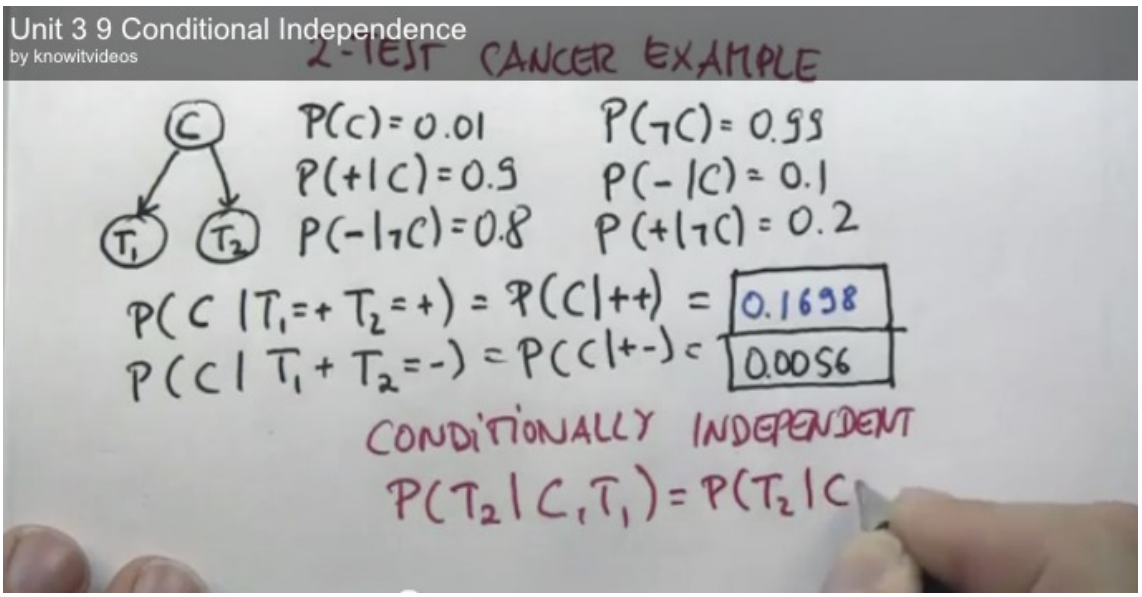


Fig. 1.2. A Bayes network from the lectures

1.0.5. Conditional Independence

In the two-cancer example (to be scanned and/or type) we assumed conditional independence:

$$P(T_2|C_1T_1) = P(T_2|C)$$

Conditional independence is a really big thing in bayes networks. The thing is that if an arrow goes out of something, that will make variables in the receiving end dependent. It's even better, since it's a model of the causality behind the dependence. Snippets

Absolute indepence does not give conditional independence, conditional independce does not imply absolute independence.

Differen type of bayes network. Two hidden causes that are confounded into a single state. Happyness may be caused by sunnyness, a raise.

The "explaining away" method. If there are multipe causes for happiness. If we can observe one of them, that makes the other less likely, and it is "explained away".

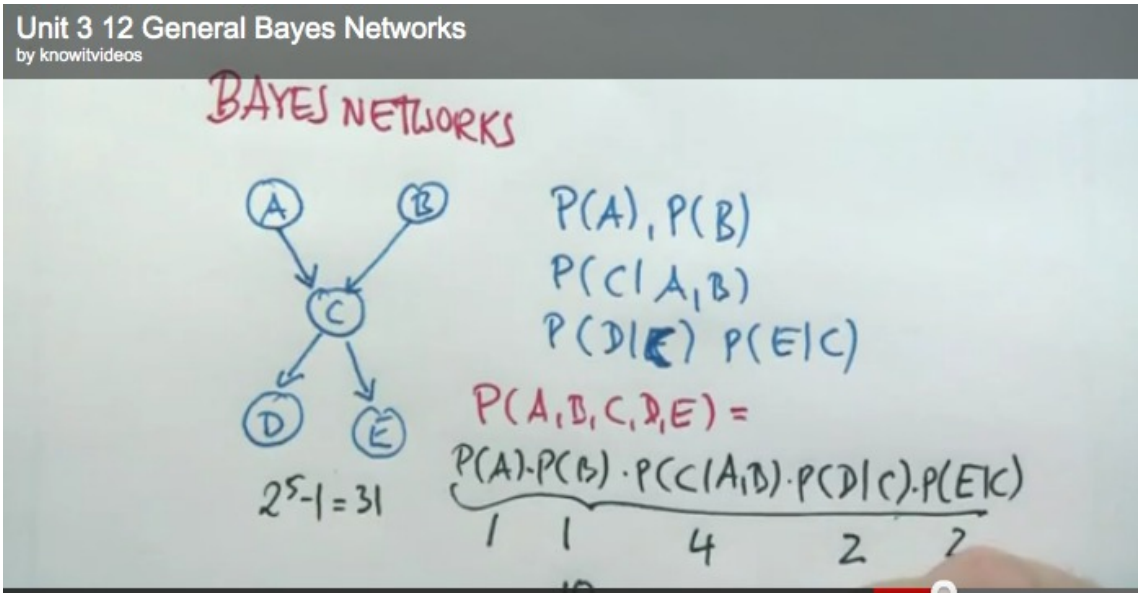


Fig. 1.3. Compact bayes

Bayes networks defines probabiliteis over graphs of random variables. Instead of enumerating all dependenies, the network is defined by distributions of probabilities dependent on the incoming arcs. The joint distribution of probability of the network is defined as the product of all the product of all the nodes (dependent of the nodes). This is a very compact definition. It doesn't have to look at the set of all subsets of dependent variables. In this case only ten parameters instead of 31. This compactness is the reason bayes networks are used in so many situations.

Unstructured joint representations sucks compared with bayes representation.

1.0.6. Parameter count

A bayes network is defined as an equation like this one.

$$P(A,B,C) = P(C|B,C) * P(A) * P(B) = \prod_i P_i(c_i|c_1 \cdots c_{n(i)})$$

The number of parameters that are necessary to specify this network is $\sum_i 2^{n(i)}$, so in the case above the count would be $1 + 1 + 4 = 6$.

1.0.7. D-separation

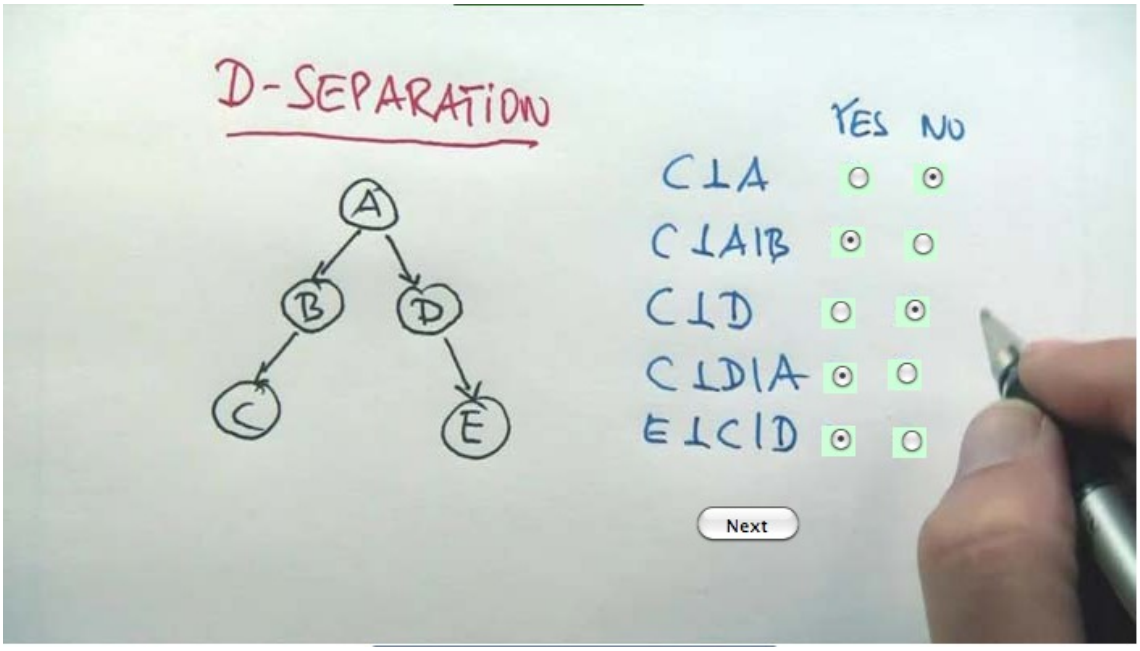


Fig. 1.4. D separation

Two variables are independent if they are independent of Any two variables independent if they are linked by just unknown variables. Anything downstreaam

Reachability = D-separation

Active triplets: render triplets dependent.

Inactive triplets: render triplets independent.

We'vw learned about graph structure, compact representation, conditional indepence and some applications.

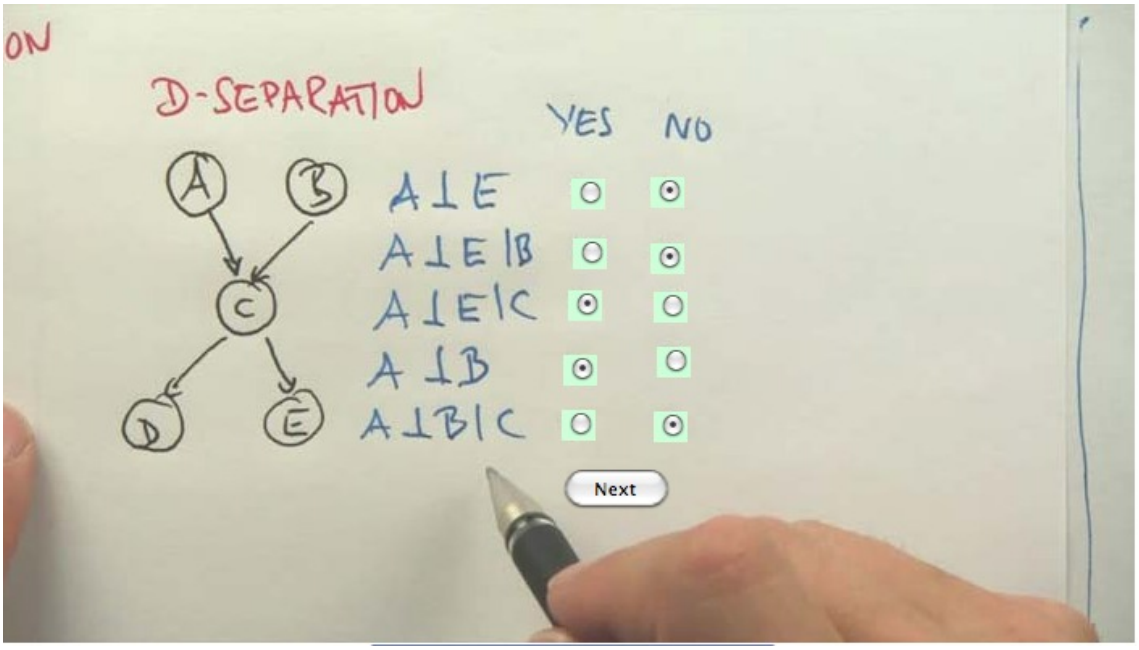


Fig. 1.5. More D separation

1.1. Probabilistic inference

Probability theory and bayes net and independence. In this unit we will look at probabilistic inference.

In probabilisticinference we don't talk about *input variables* and *output values*, but evidence and query. The *evidence* is the stuff we know something about, the *query* are the things we want to find out about. Anything that is neither evicence nor query is a *hidden variable*. In probabilstic inference, the output isn't a single number, but a *probability distribution*. The answer will be a *complete probability distribution* over the query variables. We call this the *posterior distribution*.

Some notation:

The Qs are the query variables, the Es are the evidence variables. The computation we want to come up with is:

$$P(Q_1, Q_2, \ldots | E_1 = e_1, E_2 = 2_2, \ldots)$$

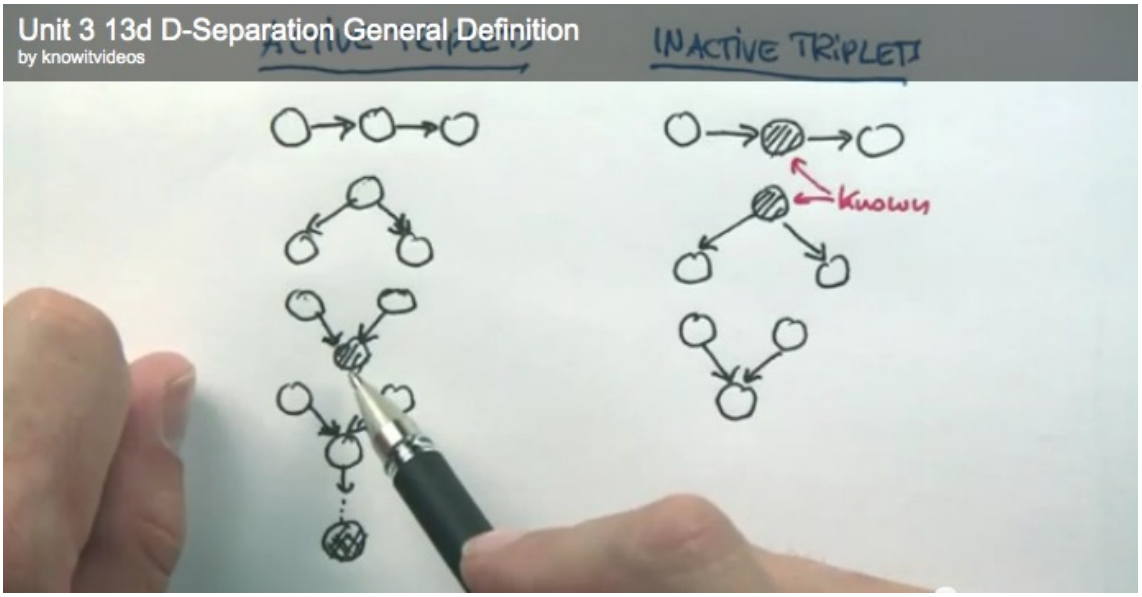


Fig. 1.6. Reachability

Another question we can ask: What is the most probable distribution. The notation for that is:

$$\text{argmax}_q \, P(Q_1 = 1_1, Q_2 = 1_2, \ldots | E_1 = e_1, E_2 = e_2, \ldots)$$

Find the combination of q values that are "maxible" given the evidence. That is the set of qs having the highest probability of being true.

In an ordinary computer language computation goes only one way, from input to output. Bayes nets are not restricted to going in only oen direction. We can go in the causal directing, using the evidence as input and using the query values at the bottom to figure out what's the most likely situation given some evidence.

However we can reverse the *causal flow*. We can actually mix the variables in any way we want.

inference on bayes networks: Enumeration

Goes through all the combinations, adds them up and comes up with an answer:

- *State the problem:* "What is the probability that the burglar alarm went of given that john called and Mary called": $P(+b| + j, +m)$.

- We will be using a definition of conditional probability that goes like this:

$$P(Q|E) = \frac{P(Q, E)}{P(E)}$$

- The query we wish to get is the *joint probability distribution* divided by the *conditionalized variables*.

$$P(+b|, +j, +m) = \frac{P(+b, +j, +m)}{P(+j, +m)}$$

  Reading this equality from left to right, it can be interpreted as rewriting the conditioned probability into a fraction of unconditioned probability.

We're using a notation here:

$$P(E = \text{true}) \equiv P(+e) \equiv 1 - P(\neg e) \equiv P(e)$$

The latter notation may be confusing, since it's unclear if e is a variable.

First take a conditional probability, and then rewrite it as an unconditional probability. Now we enumerate all the atomic probabilities and calculate the sum of products.

We'll take a look at $P(+b, +j, +m)$. The probability of these three terms can be expressed like this.

$$P(+b, +j, +m) = \sum_e \sum_a P(+b, +j, +m)$$

We get the answer by summing over all the possibilities of e and a being true and false, four terms in total. To get the values of these atomic events, we need to rewrite the term $P(+b, +j, +m)$ to fit the conditional probability tables that we have associated with he bayes network.

$$\begin{aligned} P(+b, +j, +m) &= \textstyle\sum_e \sum_a P(+b, +j, +m) \\ &= \textstyle\sum_e \sum_a P(+b)P(e)P(a| + b, e)P(+j|a)P(+m, a) \\ &= f(+e, +a) + f(+e, \neg a) + f(\neg e, +a) + f(\neg e, \neg a) \end{aligned}$$

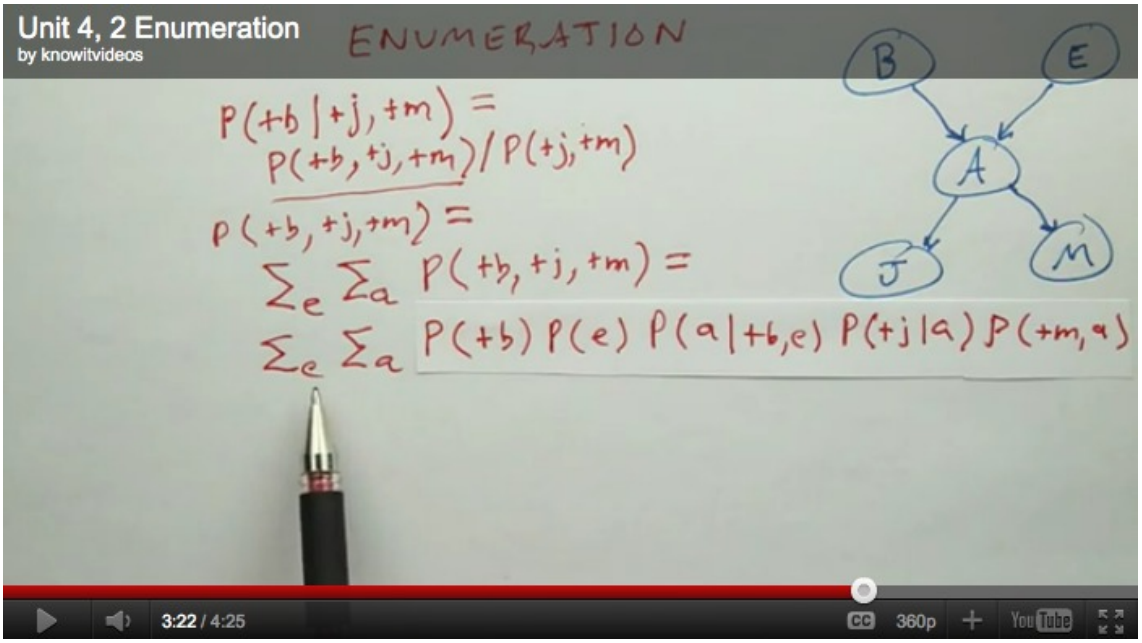


Fig. 1.7. Enumeration

The numbers to fill in we get from conditional probability tables

For simple networks enumeration is simple to do. For five hidden variables there will only be 32 terms to consider. If there are several tens of hidden variables there will be billions or quadrillion rows. That is not just practical.

Pulling out terms

The first optimization we'll consider is *pulling out terms*. If we start with:

$$\sum_e \sum_a P(+b)P(e)P(a|+b,e)P(+j|a)P(+m,a)$$

The probability of $P(+b)$ will be the same all the time, so we can pull it outside the summation. That's a little bit less work to do. We can also move the term $P(e)$ can be moved in front of the second summation:

$$P(+b)\sum_e P(e)\sum_a P(a|+b,e)P(+j|a)P(+m,a)$$

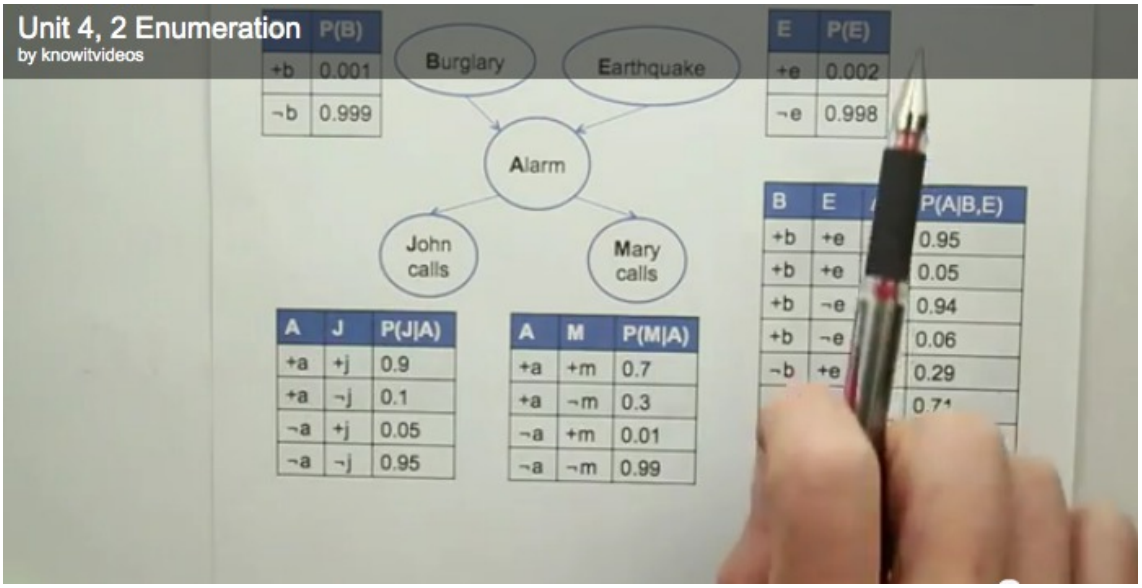


Fig. 1.8. Enumeration tables

This is good since it means that each row takes less work, but it's still a large number of rows.

Maximize independence

For instance a network that is a linear string can have inference done in time that is proportional to the number $O(n$, whereas a network that is a *complete network*, i.e. a network where all nodes are connected, could take time proportional to $O(2^n)$ for boolean variables. In the alarm network we took care that we had all the dependence relations represented in the network. However, we could do it differently.

The moral is is that bayes networks are written the most compactly when they are written in that *causal direction*, when the networks flows from causes to effects. In the equation

$$P(+b)\sum_{e} P(e)\sum_{a} P(a|+b,e)P(+j|a)P(+m,a)$$

we sum up everything. That's slow since we end up repeating a lot of work. We'll now look at another technique called *variable elimination* which in many network

operates much faster. It's still a hard problem (*NP-hard*) to inference on Bayes networks in general, but variable elimination works faster in most practical cases. It requires an algebra for addressing elements in *multi dimensional arrays* that comes out of the P terms in the expression above.

We look at a network with the variables R (raining), T (Traffic). T is dependent on R. L (late for next appointment). L is dependent on T. For a simple network like this enumeration will work fine, but we can also use eleminitation.

It gives us a way to combine parts of the network into smaller parts, en then enumerate over those smaller parts and then continue combining. So we start with a big network we elminiate som of the variables, then compute by *marginalizing out* and then we have a smaller network to deal with.

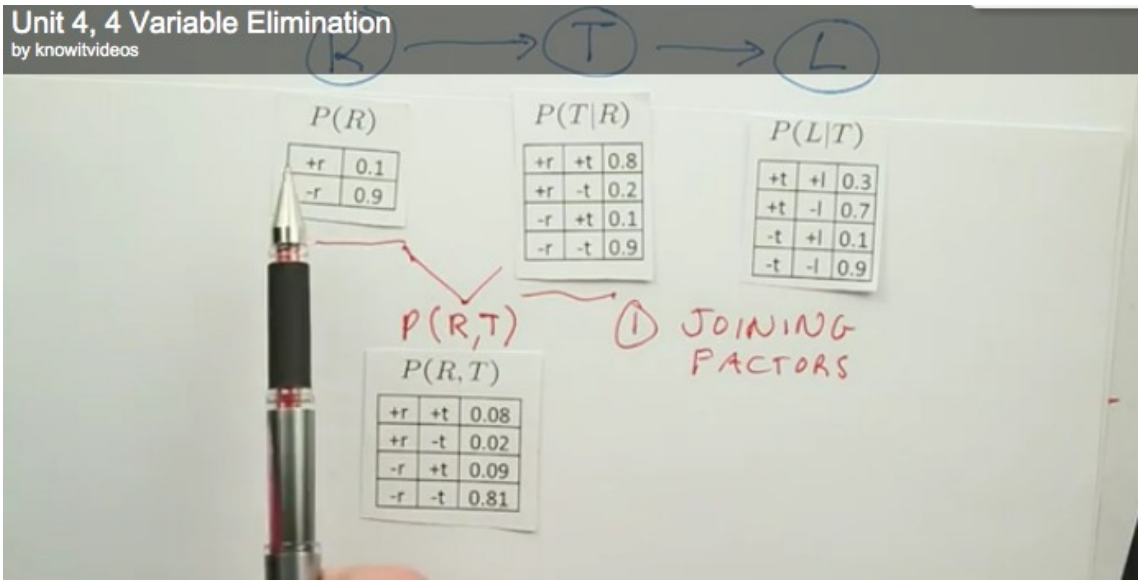


Fig. 1.9. Variable elimination

1. *Joining factors:* We chose two or more of the factors, and join the together to a new factor. $P(R,T)$ could be one such factor. It's just a "join"operator over the relevant tables where the probabilities are multiplied. I'm sure an sql statement can be concocted to calculate this :-)

2. *summing out or marginalizing:* Continue the process to remove factors, so in the end you end up with a much smaller number of factors.

If we make a god choice of the orde of eliminations to do it can be very much more efficient than just enumeration.

1.1.1. Approximate inference sampling

Repeat experiments to sample input. Repeat until we can estimate the joint probability. Large numbers of samples are necessary to get true to the true distribution.

Sampling has an advantage over inference in that it gives us a procedure to get to an approximation to the joint distribution, whereas the exact computation may be very complex.

Sampling has another advantage: If we don't know what the probability distributions are, we can still proceed with sampling, but we couldn't do that with inference.

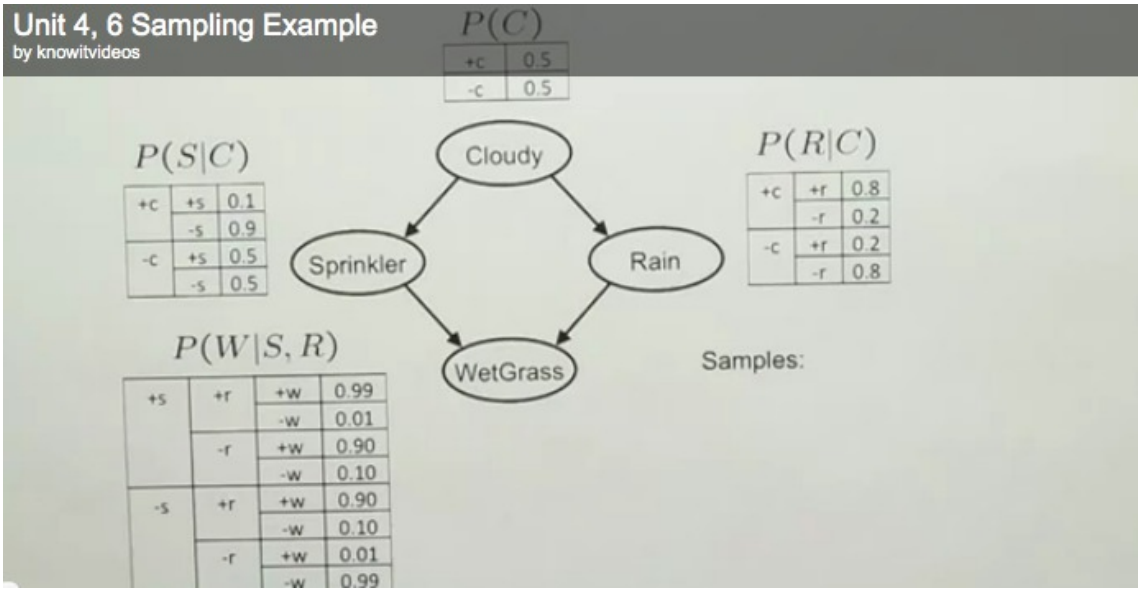


Fig. 1.10. Sampling example

Four variables, all boolean. Cloudy, Sprinkler, Rain, WetGrass. To sample we use the probability tables and traverses the graph and assigns values randomly (according to the distributions) and then uses those values to build up an image of the total probability.

The probability of sampling a particular value depends on the parent. In the limit the count of the counted probability will approach the true probability. With an infinite number of samplings we would know the result :-) The sampling method

is *consistent*. We can use this sampling method to compute the complete joint probability distribution, or we can use it to compute values for an individual variable.

Now what if we wish to compute a conditioal probability? To do that we need to to start doing the same thing (generating samples), but then reject the samples that don't match the condition we are looking for. This technique is called *rejection sampling* since we reject the samples that don't have the properties we are interested in and keep those that do.

This procedure would also be consistent.

Tere is a problem with rejection sampling: If the evidence is unlikely, you end up rejecting a lot of samples. Let's consider the alarm network again. If we're interested in calculating the probability $B|a+)$ i.e. the probability of a burglary given that the alarm has gone off. The problem is that burglaries are very infrequent, so we will end up rejecting a lot of samples.

To counter this we introduce a new method called *likelyhood weighting* that generates samples that can be used all the time. We fix the conditional variables (setting $a+$ in this case), and then sample the rest of the variables.

We get samples that we want, but the result we get is *inconsistent*. We can fix that by assigning a probability to each sample and weighting them correctly.

In *likelyhood weighting* we sample like before but add a probabilistic weight to each sample. Each time we are forced to make a choice, we must multiply a probability for the value we choose. With weighting likelyhood weighting is consistent.

Likelyhood weighting is a great technique but it doesn't solve all problems. There are cases where we fix some variables, but not all, and that makes variables that the fixed variables are dependent on likely to make a lot of values with very low probabilities. It's consistent, but not efficient.

Gibbs sampling takes all the evidence, not just the upstream evidence into account. It uses a technique called *markov chain monte carlo* or *mcmc*. We resample just one variable at a time, conditioned on all the others: Assume we have a set of variables and initialize them to random values keeping the evidence values fixed. At each iteration through the loop we select just one non-evidence variable and resample that based on all the other variables. That will give us another variable. Repeat.

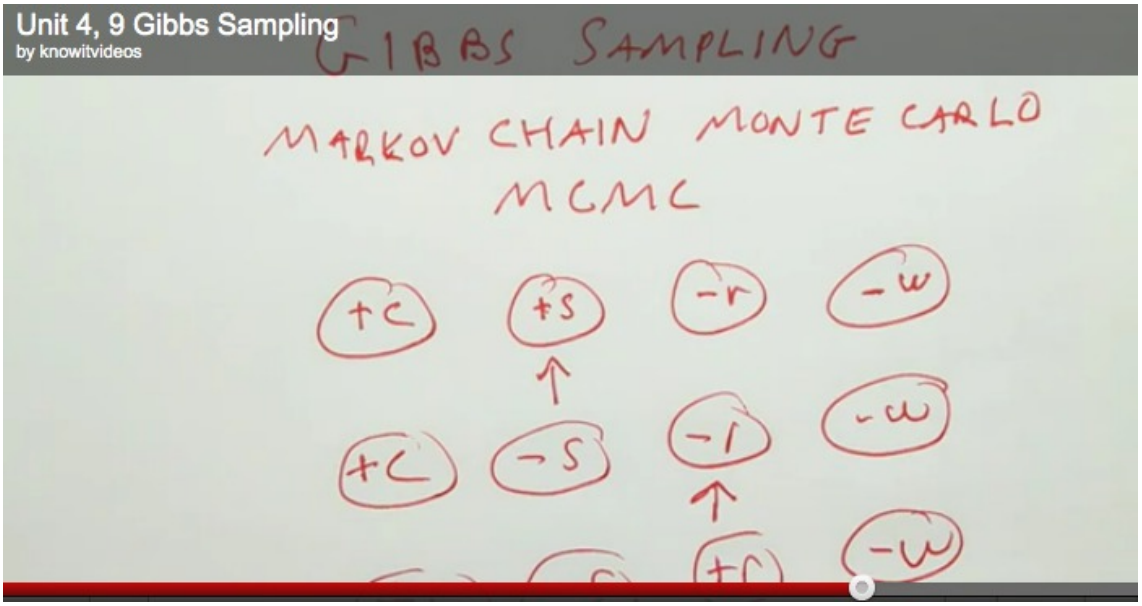


Fig. 1.11. Gibbs sampling

We end up walking around in the space. In mcmc all the samples are dependent on each other, in fact adjacent samples are very similar. However, the technique is still consistent.

The monty hall problem Three doors, one with an expensive sports car, the other two contains a goat. If you choose door number one, the host will now open a door containing a goat. You can now stick with your choice or switch.

2. Machine learning

The world is data rich. Chemihal, financial pharmaceutical databases etc. To make sense of it macine learning is really important to make sense of it.

So ar we've talked about bayes networks that are known. In machine learning addreses how to learn models from data.

2.1. Supervised learning

Fig. 2.1. Stanley, the winning robot in the DARPA grand challenge.

Thrun shows off the Stanley car that uses a laser that builds a 3d model of the terrain ahead. It tries to drive on flat ground. The laser only looks about 25

meters ahead. The camera image sees further, but the robot uses machine learning to extrapolate the knowledge about where drivable road is loacated out to two hundred meters, and that was crucial in rder to get the robot to drive fast.

- what? Parameters, structure and hidden concepts (some hidden rules may exist in the data).
- What from? Some sort of target information. In supervised learning we have specific target labels rules) .
- Unsupervised learning? Target labels are missing.
- Reinforcement learning. Receiving feedback from the environment
- what for? Prediction, diagnosis? Summarization (gisting) ..
- how? Passive (if the agent is just an observer), otherwise active Sometimes online (when data is generated) or offline.
- Classification (fixed no of classes) v.s. regression (continous)
- Details? Generative v.s. discriminative. Generative tries to describe stuffs as general as ppossible, dicriminitave attempts to find things as specific as possible.

In supervised learning we have a bunch of features and a target:

$$x_1, x_2, \ldots, x_n \rightarrow y$$

The learning algorithm gets a learning set and generates a model. Features, behavior etc. The subject is to identify the function f.

When selecting models, *occam's razor* is good: Everything else being equal choose the less complex hypothesis. In reality there is a tradeoff between fit and low complexity. *bias variance methods.*

In practice it's a good idea to push back on complexity to avoid overfitting and generative errors. *Overfitting* is a major source of poor performance for machine learning.

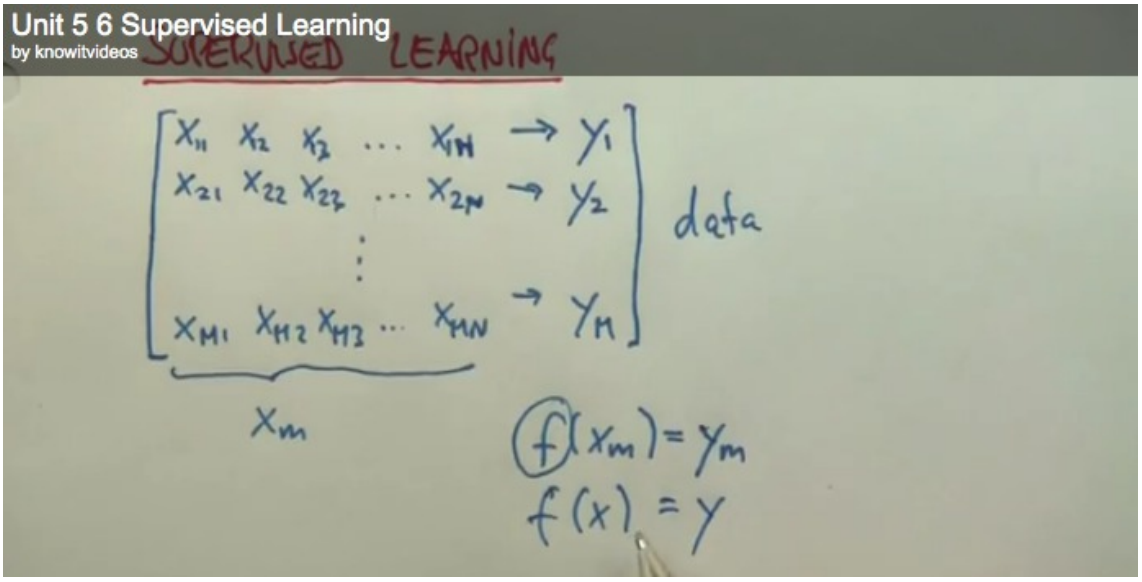


Fig. 2.2. Supervised learning

2.1.1. Spam detection using bayes networks

We wish to categorize into spam/ham. to get things usable for emails we need a representation. We use the *bag of words* represention. It is a word frequencey vector. The vector is obvlivious to the location of the word in the input.

In the ham/spam examle, this the vocabulary

the size is 14. The probability that is 3/8 that it's spam

2.1.2. Maximum likelyhood

Since I'm not really conversant on maximum likelyhood issues, I'll copy a bit in detail, since that is something that frequently helps me transfer the subject matter into my mind :-)

We assume that there are two possible outcomes of a process, "S" and "H", and then define $P(S) = \pi$ and conversely $P(H) = 1 - \pi$. Rewriting this yet once more gives:

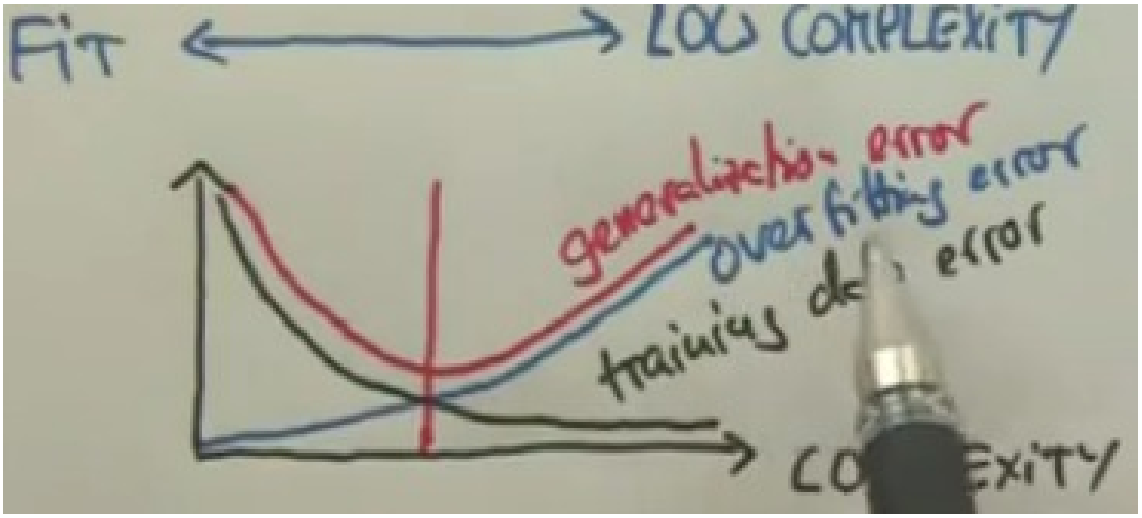


Fig. 2.3. Error classes

$$p(y_i) = \begin{cases} \pi & \text{if } y_i = S \\ 1-\pi & \text{if } y_i = H \end{cases}$$

Now comes a bit that I don't understand, since he says:

$$p(y_i) = \pi^{yi} \cdot (1-\pi)^{1-y_i}$$

Less mysterious, since it is a a consequence of the above:

$$p(\text{data}) = \pi_{i=1}^{n} p(y_i) = \pi^{\text{count}_{(y_i=1)}} \cdot (1-\pi)^{\text{count}_{(y_i=1)}}$$

$$\begin{array}{lcll} P(\text{secret}|\text{spam}) & = & 3/9 & = & 1/3 \\ P(\text{secret}|\text{ham}) & = & 1/15 & & \end{array}$$

2.1.3. Relationship t Bayes Networks

We are making a maximum likelyhood etimator modelled as an bayesian estimator and using machine learning to finding the network.

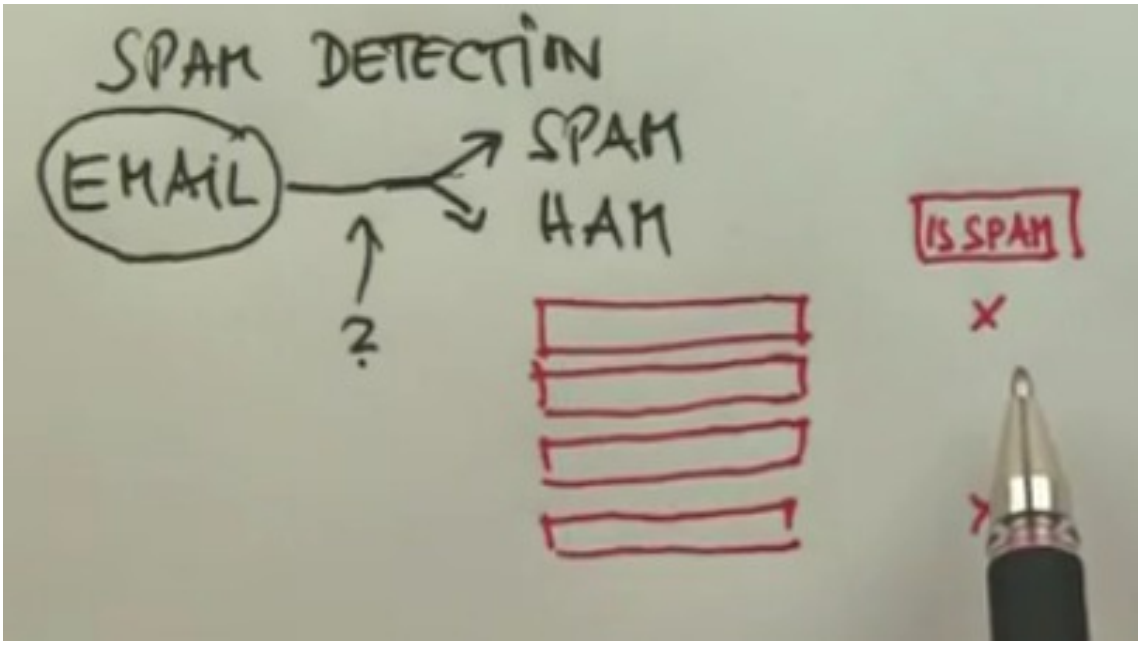


Fig. 2.4. Detecting spam in email

Given a twelve word vocablary, we need 23 parameters. P(spam), and two for each word (one for ham, and one for spam). Now, there are 12 words, so that makes 12 parameters for ham probability, but since these add up to one, we only need eleven :-) The same thing applies to spam, so that is 11 + 11 + 1 = 23.

So, what i the probability of classifying "sports" as spam? Well, we have to bring out Bayes:

```
\[
P(spam|''sports'') =
    \frac{p("sports"|spam) \cdot p(spam)}{P(spam)}
=
    \frac{p("sports"|spam) \cdot p(spam)}{P(m|spam)P(spam) + P(m|ham)P(ham)}
\]
```

1/9*3/8 / (1/3 * 3/8 + 1/3 * 5/8) = (3/72)/ (18/72) = 3/18 = 1/6 = 0.667 The answer is 3/18

```
click
costs
event
is
link
money
offer
play
secret
sport
sports
today
went
```

Fig. 2.5. The spam/ham vocabulary

Laplace smoothing

One way of fixing the overfitting is *laplace smoothing*.

$$\begin{array}{lrcl} \text{ML} & p(x) & = & \frac{\text{count}(x)}{N} \\ \text{LS}(k) & p(x) & = & \frac{\text{count}(x)+k}{N+k|x|} \end{array}$$

ML = maximum likelyhood. LS = Lapace smoothing

where count(x) is the number of occurrences of this value of the variable x. $|x|$ is the number of values that the variable x can take on. k is a smoothing parameter. And N is the total number of occurrences of x (the variable, not the value) in the sample size.

This is equivalent of assuming that we have of fake count k to he count, and then adding k to every single class we are estimating over.

if k=1. one message with one spam. For the laplace smoothed thing we get these nmbers

Summary naive bayes

We've got features and labels (e.g. spam/ham). We used ML and laplacian smoother to make a bayes network. This is called a *generative model* in that the

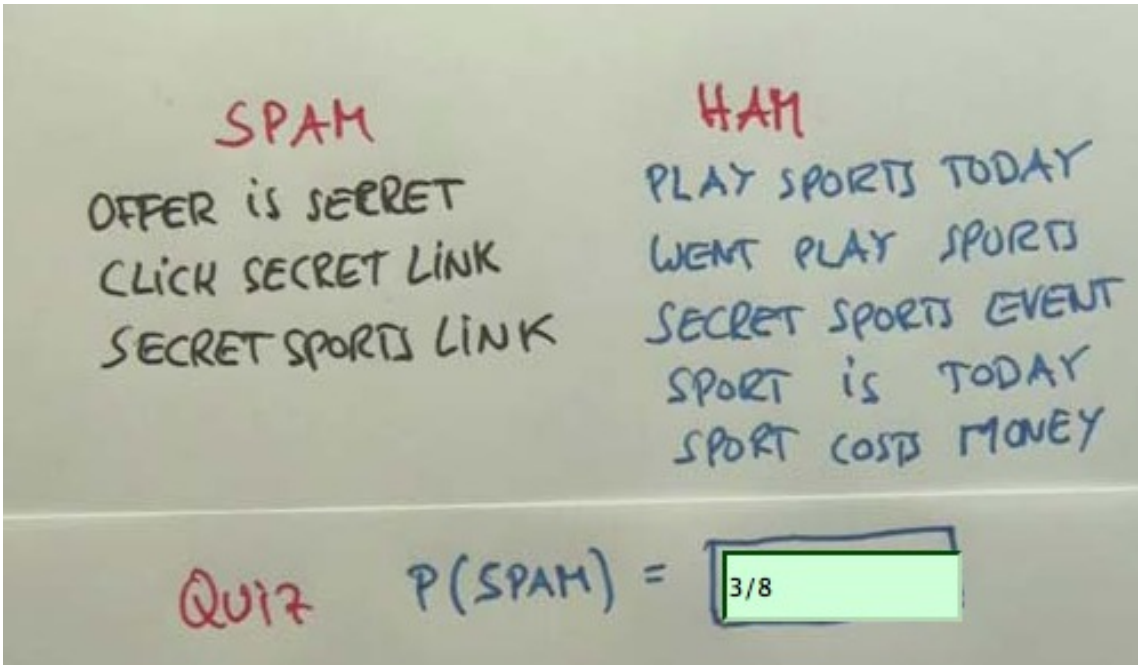


Fig. 2.6. Calculating the probability of spam

conditioned probabilities are all aiming to maximize the predictability of individual features as if those features describe the physical world. We use the *bag of words* model.

This is a very powerful method for finding spam. It's unfortunately not good enough since spammers know how to circumvent naive bayes models.

An advanced spam filter:

- Known spamming ip
- Have you mailed the person before?
- Have 1000 other people received the same message?
- Is the email header consistent?
- Is the email all caps?
- Do inline URLs point to the points they say they point to

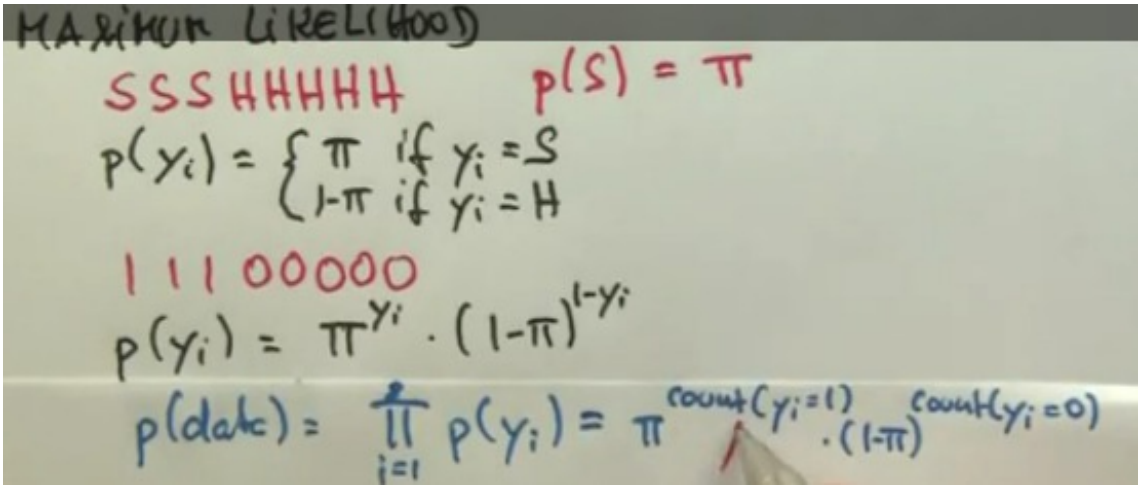


Fig. 2.7. First maximum likelyhood slide

- Are you addressed by name?

All of these features can be used by a naive bayes filter.

Hand writen digit recognition

Handwritten zipcodes. The machine learning problem is to take a handwritten symbol and find the matching correct number. The input vector for the naive bayes classifier could be pixel values in a 16x16 grid. Given sufficient many training examples we could hope to recognize letters. However, it is not sufficient *shift invariant*. There are many different solution , but one could be smoothing. We can convolve the input to "smear" the input into neighbouring pixels with a gaussian variable, and we might get a better result.

However, Bayesian classification is not really a good choice for this task. The indipendent variable assumption for each pixel is too strong an assumption in this case, but it's fun :-)

Overfitting prevention

Occam's razor tells us that there is a tradeoff between precision and smoothnes. The "k" parameter i Laplacian smoothing is that sort of thing. However, it gives us a new question: How to choose the "k".

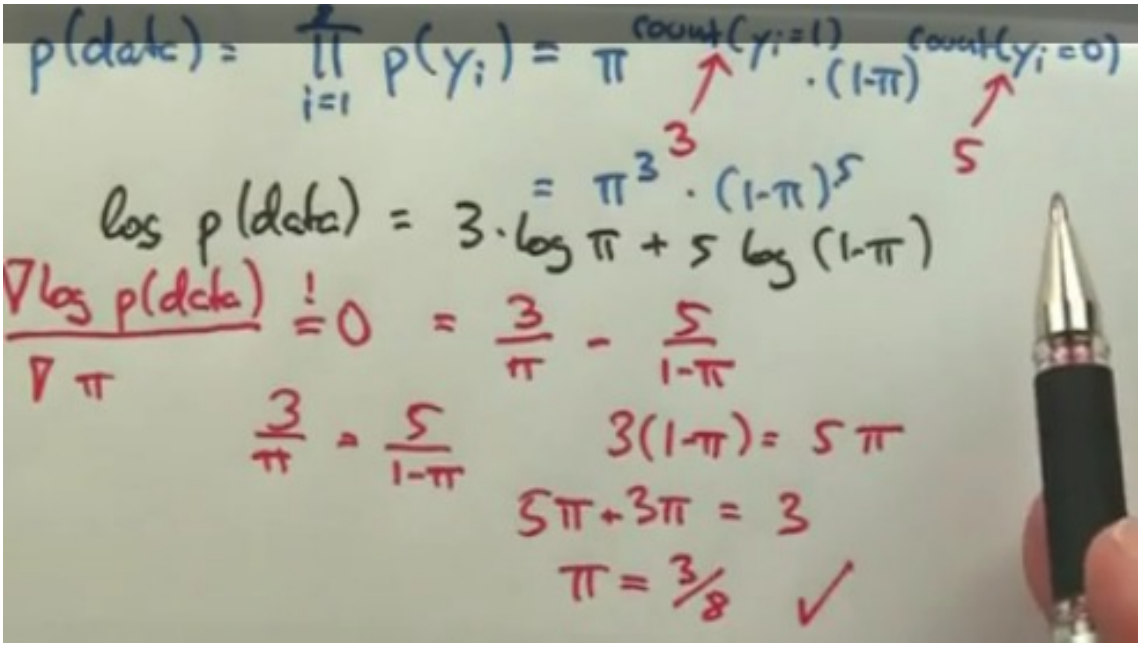


Fig. 2.8. Second maximum likelyhood slide

One method that can help is *cross validation*. The idea is to take your training data and divide it into three buckets: Train, Cross validate and Test. A typical percent-wise partitioning will be 80/10/10.

First train to find all the parameters. Then for the cross validation data test how well the classifier works. In essence perform an optimization of the classifier where the cost function is the match on the cross validation data, and use this to optimize the smoothing parameters. Finally, and only once do we validate the result using the test data, and it's only the performance on the test data we report.

It's really important to use different test and crossvalidation sets different, otherwise we are prone to overfitting. This model is used by pretty much used by everyone in machine learning. Often one mixes the training and crossvalidation sets in different ways. One common way is to use ten mixes, called "tenfold cross validation" (and run the model ten times).

Supervised learning, regression

We have classification and regression problems. Regression problems are fundamentally different than classifications. Bayes networks only predicts discrete classes so

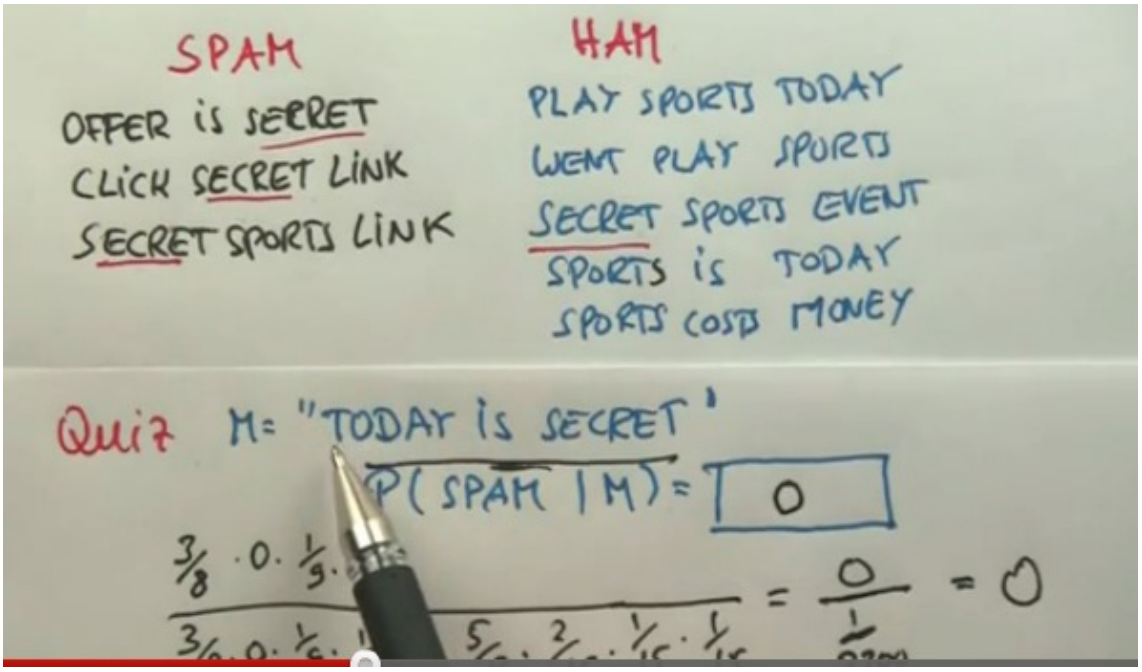


Fig. 2.9. Overfitting using a naive Bayes classifier

it's not useful for regression.

In regression our job is to fit a curve to predict the line.

In all regression we have a set of data points that maps to a continous variable, and we look for a function f that matches the vector x into y.

We are trying to minimize the loss function. In this case the loss function is the sum of all the errors (squared).

This system can be solved in closed form using the *normal equations*.

To find the minimum we take the derivative of the quadratic loss function

When calculating these things, I fin that it's a good idea to calculate all the sums in advance in a table, and then just plug them in. When using a computer it's of course even simpler, but we're not doing that now.

Linear regression works very well when the data is linar, but not otherwise :-)

Outliers are handled very badly. Also when x goes to infinity, your y also does that. That is bad.

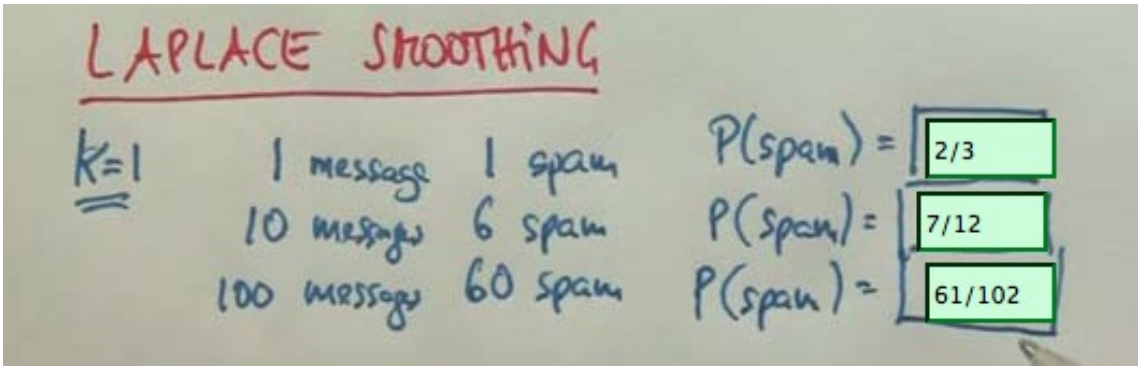


Fig. 2.10. An example of using Laplace smoothing

Logistic regression is an alternative.

$$z = \frac{1}{1 + e^{f(x)}}$$

Regularization

We assume that loss is a combination of loss from data and loss from parameters:

$$\begin{array}{rcccc} \text{Loss} & = & \text{Loss}(data) & + & \text{Loss}(parameters) \\ & & \sum_j \left(y_j - w_i x_j - w_0\right)^2 & + & \sum_i \left(w_i\right)^p \end{array}$$

The *parameter loss* is just some function that penalizes the parameters becoming large, a potential like shown above (and it's usually one or two)

2.1.4. Minimizing more complicated loss functions

In general there are no closed form solutions, so we have to revert to iterative methods, often using gradient descent.

Many books on optimization can tell you how to avoid local minima, but that isn't part of this course.

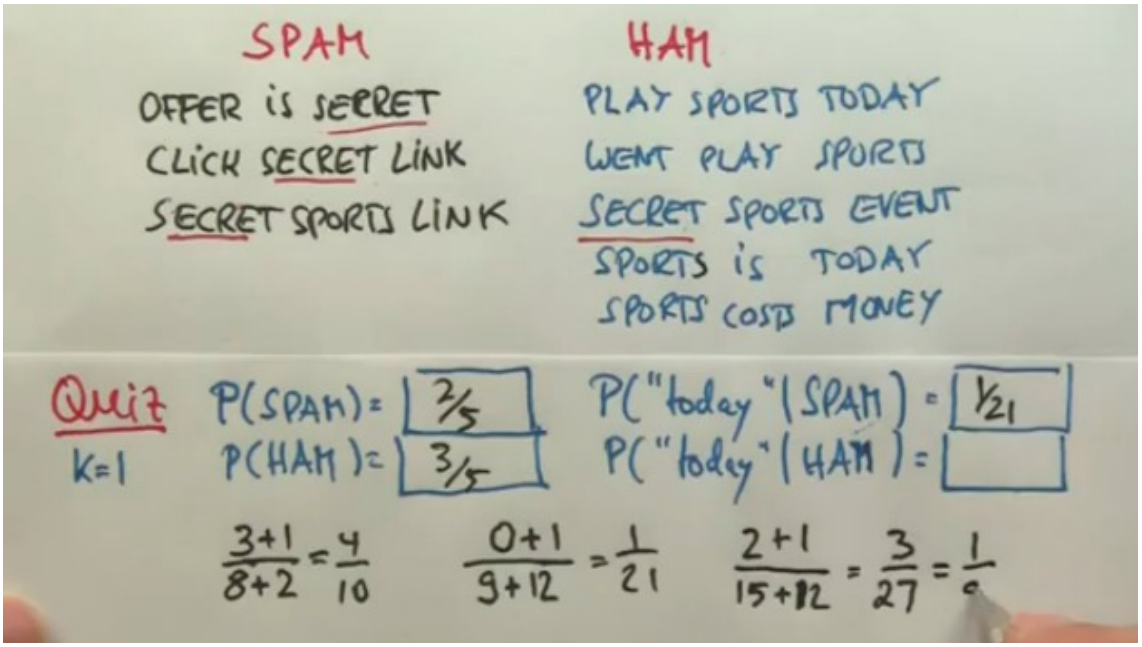


Fig. 2.11. Using Laplace smooting in a spam detector

2.1.5. The perceptron algorithm

A simple algorithm invented in the forties. The perceptron finds a linear separator. We can create a linear separation classification algorithm:

The altorithm only converges if the data is separabel, but then it converges to a linear separator

- Start with a random guess for w_1, w_0. It's usually inaccurate.
- Update the new weight $m_i^m \leftarrow w_i^{m-1} + \alpha\left(y_j - f(x_j)\right)$ The learning is guided by the difference between the wanted result. inaccurate. It's an online rule, it can be iterated many times.

Use the error to move in the direction that minimizes the error. A question is determine which of the many possible linear separators should be preferred.

K-nearest neigbours

The final method is a non-parametric method. In parametric learning the number of parameters is independent of the learning set. In parametric methods we kind of

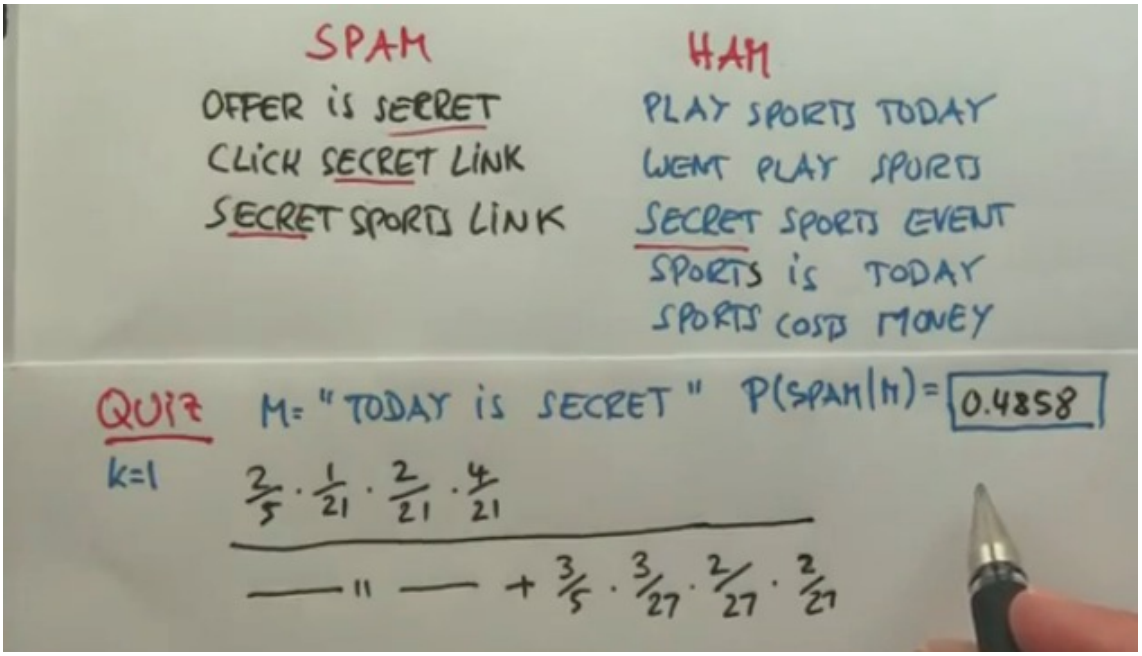


Fig. 2.12. Calculating even more spam probabilities using Laplace smoothing

drag the entire learning set (or a subset of it) along into the model we are creating. For the Bayes examples since the dictionary was variable, so that that wasn't exactly right, but for any fixed dictionary the number of the parameter number is independent the training set size.

1-nearest neigbour is a very simple method. Given a set of data points search for the nearest point in the euclidian space and copy its label. The *k-nearest neigbour* is also blatantly simple. In the learning step, just memorize all the data. When an input data arrives do, you find the k nearest neigbours, and you return the majority class label. It's actually brutally simple. The trick is of course to pick the right "k". K is a smoothing parameter (like the lambda in the laplace smoothing). Higher k gives more smoothing. The cleaner the decision boundary will be, but the more outliers there will be. k is a *regularizer*. Like for the laplace smoother we can use cross validation to find an optimal value of it.

KNN has two main problems, the first is very large data sets, and the other is very large feature spaces. Fortunately there are methods of searching very quickly in tres so the search "*kdot trees*". Large feature spaces is actually the bigger problem. The tree methods become brittle for higher dimensions.

The edge length goes up really fast. All points ends up being very far away. For

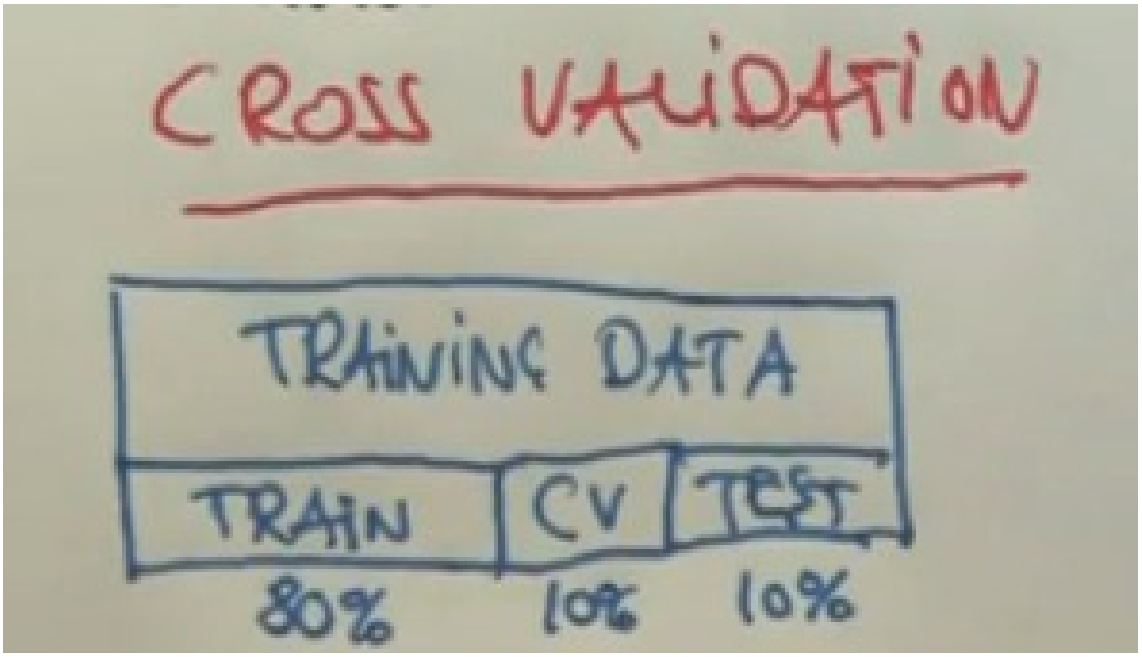


Fig. 2.13. Cross validation

few dimensions like three or four it works well, for twenty or hundred it doesn't.

2.2. Unsupervised learning

Here we just have data, so our task is to find structure in the data.

IID Identically distributed and Independently drawn from the same distribution. We wish to perform a *density estimation* to find the distribution that produced the data.

Methods used are *clustering* and *dimensionality reduction*. Unsupervised learning can be applied to find structure in data. One fascinating application is *blind signal separation*. How to recover two speaker output from a single microphone. It's possible, but it doesn't require target signals. It can be construed by using *factor analysis*.

Unsolved problem from Streetview. The objects that can be seen is similar in many of the images. Can one take the streetview dataset to discover concepts such

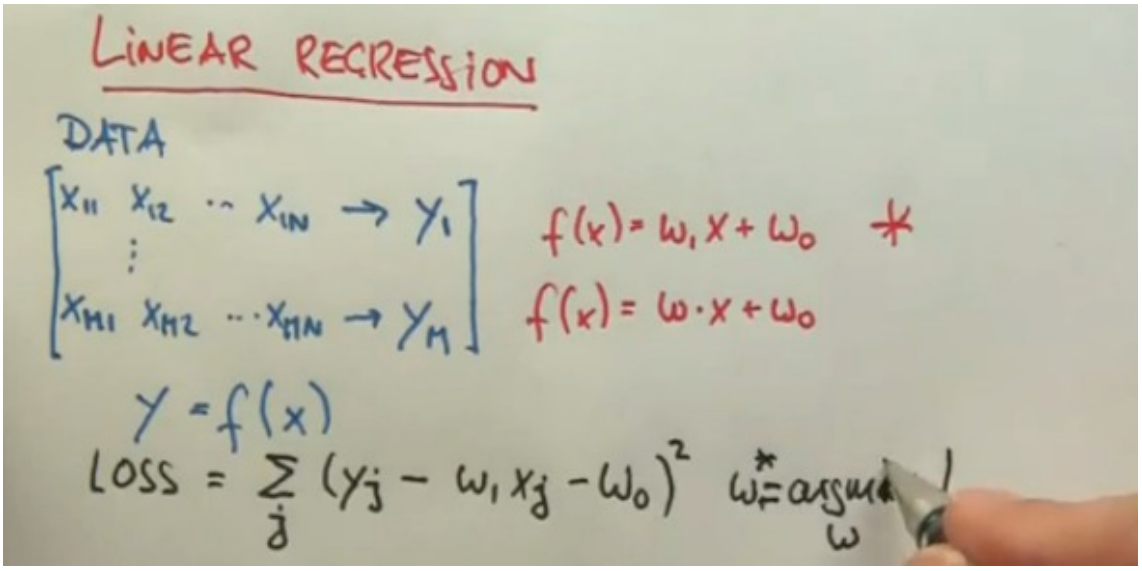


Fig. 2.14. Linear regression

as trees, stopsigns, pedestrians etc. Humans can learn without target sets, can machines?

Clustering: The most basic forms of unsupervised learning *k-mans* and *expectation maximization*.

k-means

- Choose to cluster centers in the space and do so at random.
- The clusters will now divide the sample space in two by a line being orthogonal to the vector between the two cluster centers (a *voronoi graph* based on the two centers).
- Now, find the optimal cluster centers for the two points. Minimize the joint squared distance from the center to all the data points in the cluster.
- Now iterate. The cluster centers have moved, so the voronoi diagram is different, that means that the clustering will be different. Repeat until convergence.

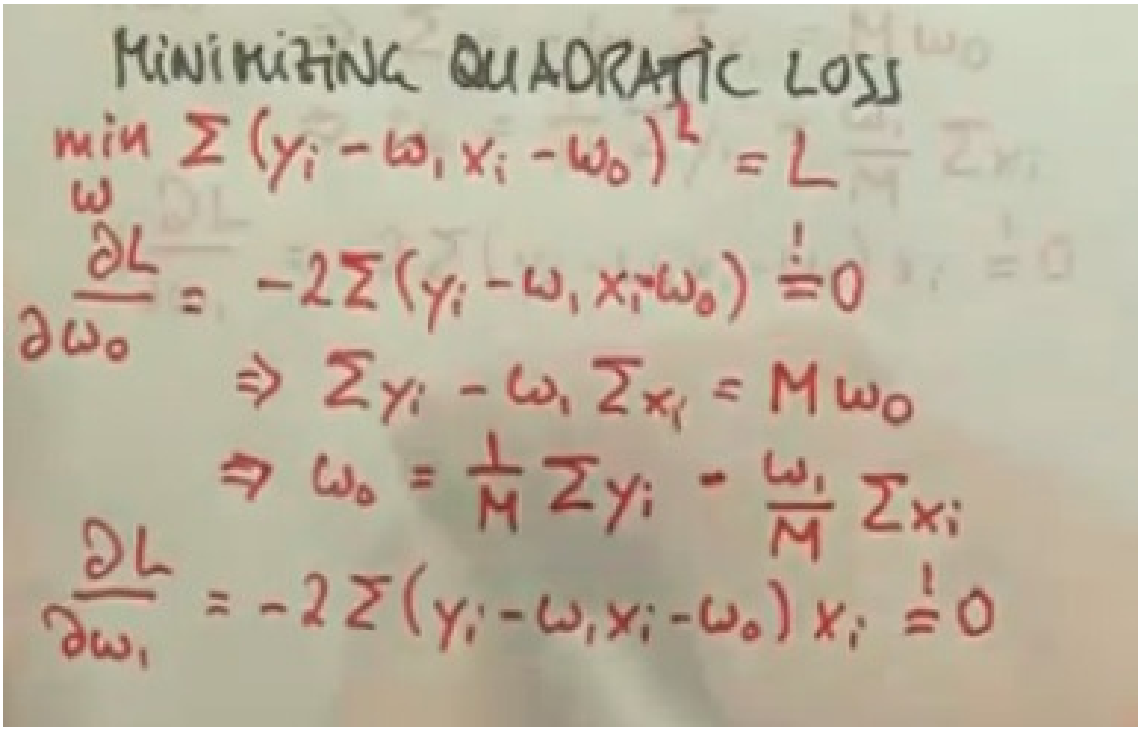


Fig. 2.15. Minimizing quadratic loss

The algorithm is know to converge. However the general clustering algorithm is know to be *NP-complete*.

Problems with k-means: First we need to know k. Then there are the local minima we can fall into. There is a general problem with high dimentionality and there is a lack of mathematematical basis for it.

2.2.1. Expectation maximization

Probabilistic generalization of k-means.. Uses actual probability distribution. It has a probabilistic basis, more general than k-means.

Gaussians

Continous distribution . Mean is μ the normal distribution σ (where the second derivative changes sign :-) The density is given by:

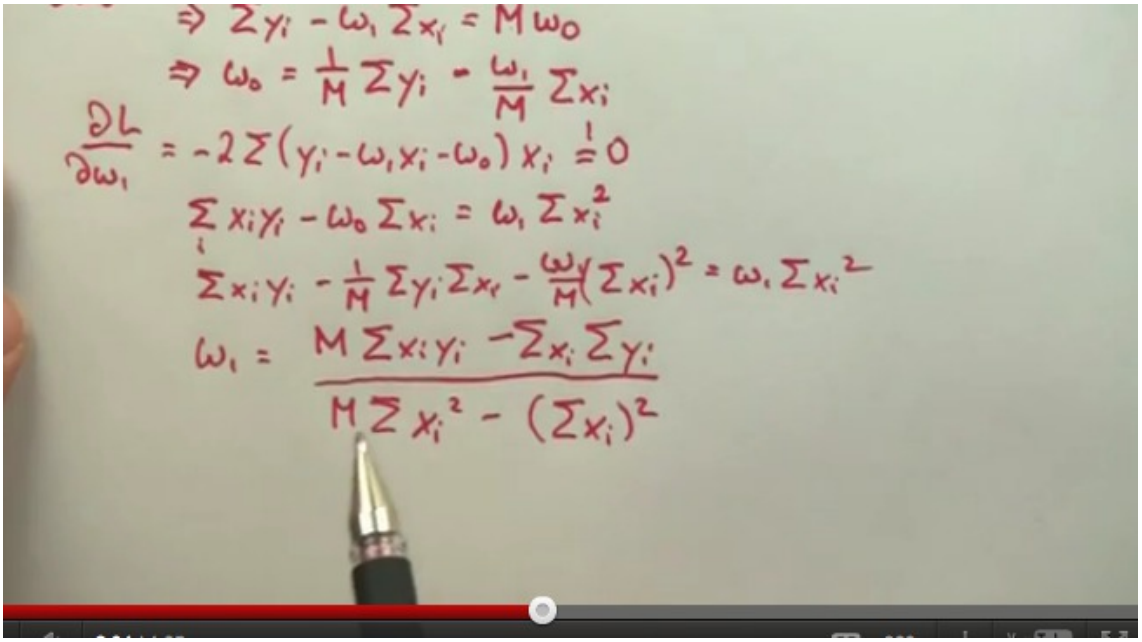


Fig. 2.16. Hand calculating the parameters of linear regression

$$f(x) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{1}{2}\frac{(x-\mu)^2}{\sigma^2}\right)$$

This distribution has the nice property (like all continous probability distributions) that:

$$\int_{-\infty}^{\infty} f(x)dx = 1$$

Goes to zero exponentially fast :-) Very nice.

For every we can now assign a density value. For any interval we can now generate probabilities by calculating integrals.

The multivariate gaussian has multipel input variables. Often they are drawn by level sets. The center has the highest probability.

The formula is;

$$(2\pi)^{-\mu/2} \left|\sum\right|^{-\frac{1}{2}} \exp\left(-\frac{1}{2}(x-\mu)^T \sigma^{-1}(x-\mu)\right)$$

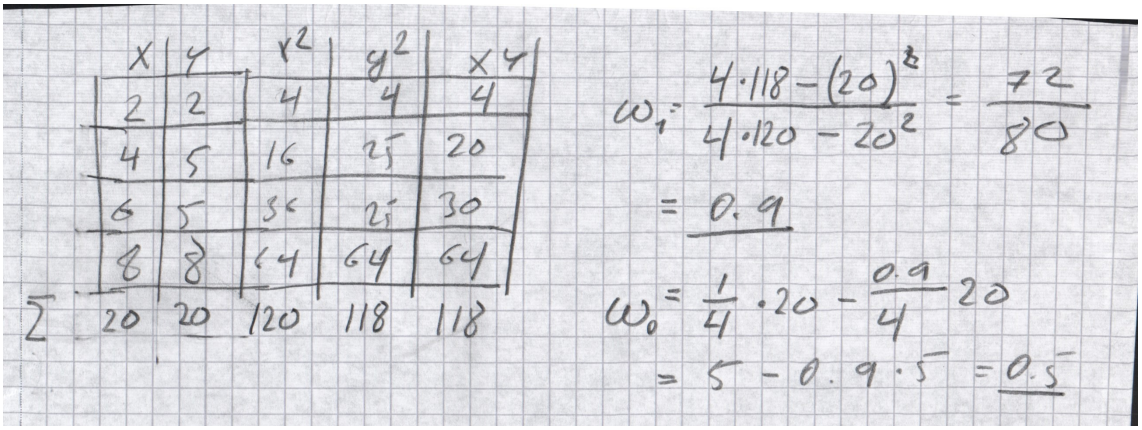


Fig. 2.17. An actual calculation of a set of linear regression parameters

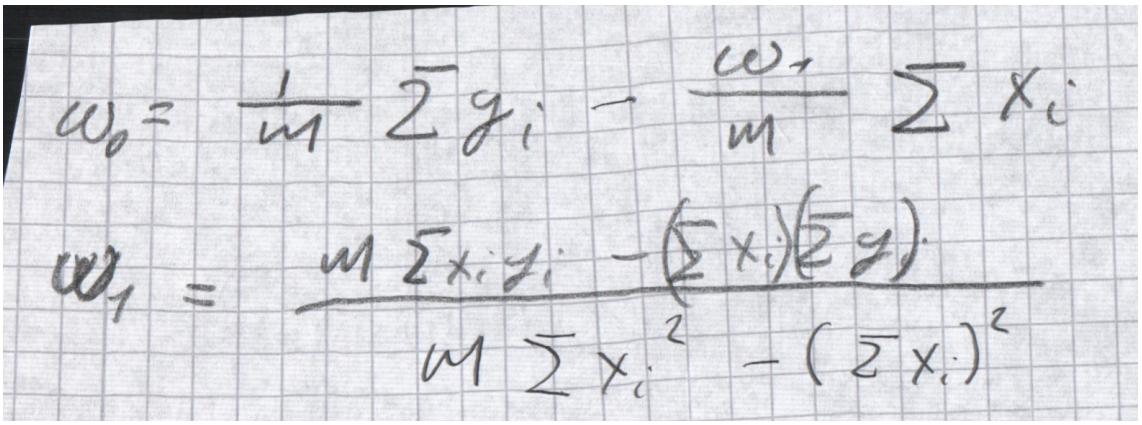


Fig. 2.18. The linear regression formulas in my beautiful handwriting

σ is a *covariance matrice*. x is a *probe point* and μ is a *mean vector*. This can be found in any textbook about *multivariate distributions*. It's very simillar: Quadratic function, exponential, normalization etc.

Gaussian Learning

In the one dimensional case the *Gaussian* is parameterized by the *mean* *mu* and the *variation* σ^2, so we have:

\[

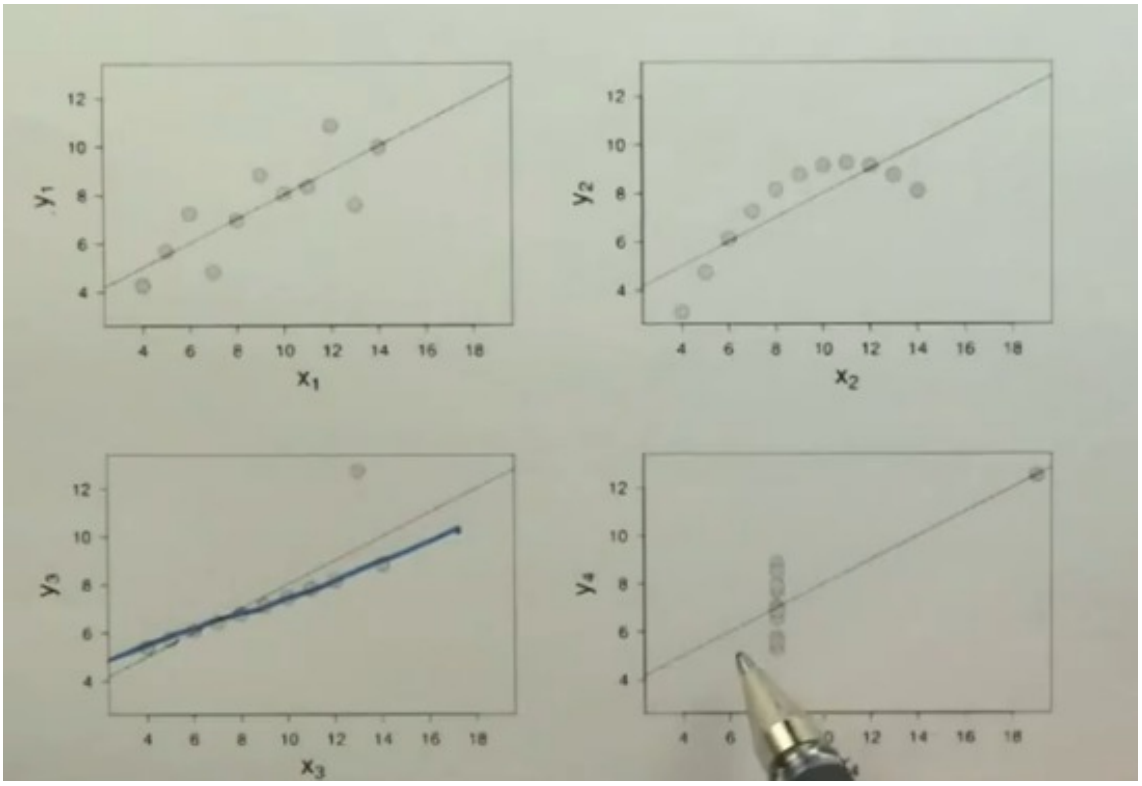


Fig. 2.19. Some curves that are not very well approximated by linear regression

```
    f(x | \mu, \sigma^2) =     \frac{1}{\sqrt{2\pi}\exp{\parens{-\frac{1}{2}\, \fra
\]
```

Fitting data, if we assume that the data is one dimension. There are really easy formaula for fitting gaussians to data:

$$\begin{aligned} \mu &= \frac{1}{M}\sum_{j=1}^{M} x_j \\ \sigma^2 &= = \frac{1}{M}\sum_{j=1}^{M}(x_j-\mu)^2 \end{aligned}$$

The mean and the sum of the squared deviations from the mean.

For multiple data points we have for data $x_1, \ldots, x_M$

$$\begin{aligned} p(x_1, \ldots, x_M|\mu, \sigma) &= \prod_i f(x_i|\mu, \sigma) \\ &= \left(\frac{1}{2\pi\sigma^2}^{M/2}\right) \exp\left(-\frac{\sum_i (x_i-\mu)^2}{2\sigma^2}\right) \end{aligned}$$

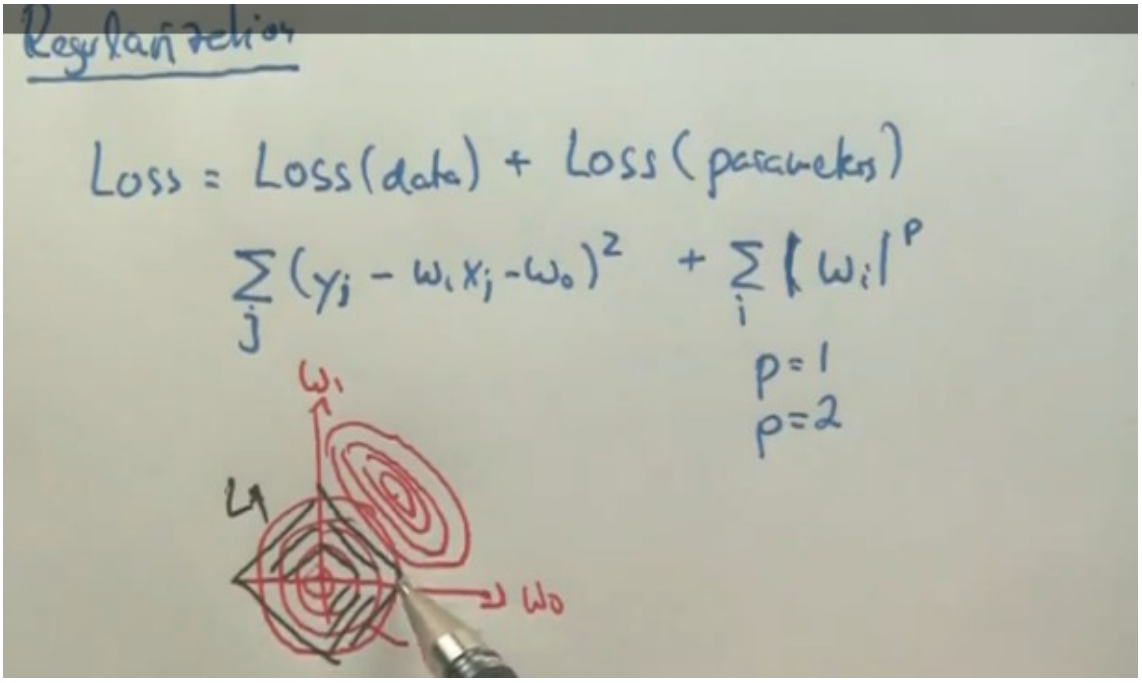


Fig. 2.20. Regularization imposes a cost for introducing parameters. This is good since it reduces the risk of overfitting.

The best choice for a parametrization is the one that maximizes the expression above. That gives us the *maximum likelyhood estimator*. We now apply a trick, which is to maximize the log of the function above:

$$M/2 \log \frac{1}{2\pi\sigma^2} - \frac{1}{2\sigma^2} \sum_{i=1}^{M} (x_i - \mu)^2$$

The maximum is found by setting the partials to zero:

$$0 = \frac{\partial \log f}{\partial \mu} = \frac{1}{\sigma^2} \sum (x_i - \mu)$$

This is equivalent to $\mu = \frac{1}{M} \sum_{i=1}^{M} x_i$, so we have shown that the mean is the maximum likelyhood estimator for μ in a gaussian distribution :-)

We do the same thing for the variance:

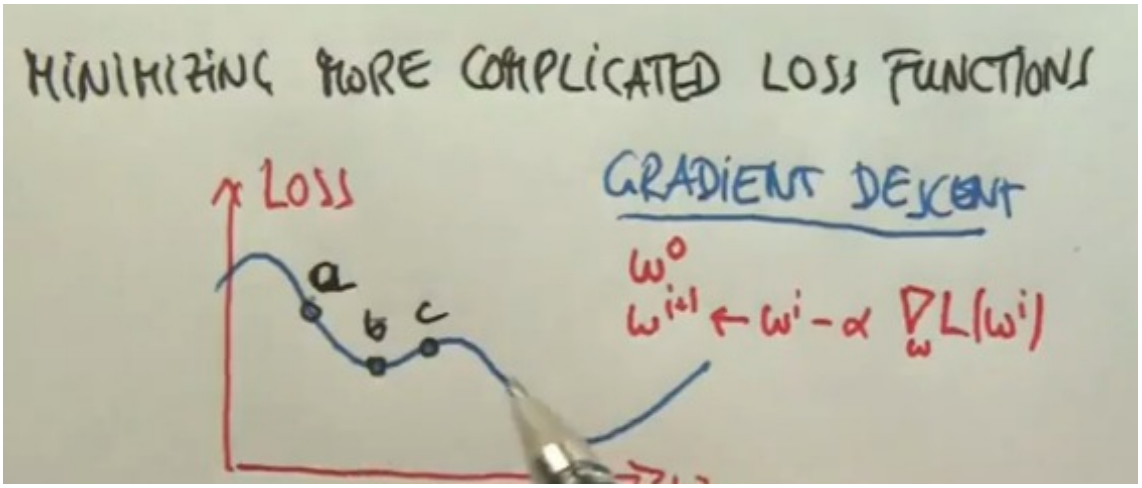


Fig. 2.21. Gradient descent

$$\frac{\partial \log f}{\partial \sigma} = -\frac{M}{\sigma} + \frac{1}{\sigma^3} \sum_{i=1}^{M} (X_i - \mu)^2 = 0$$

which works out to $\sigma^2 = \frac{1}{M} \sum_{i=1}^{M} (x_i - \mu)^2$

Rmz: *Verify these steps sometime*

2.2.2. Expectation maximization

After this detour into Gaussians we are ready to continue with to *expectation maximization* as a generalization of *k-means*.

Stat the same way. Two randomly chosen cluster centers. But instead of making a *hard correspondence* we make a *soft correspondence*. A data point atracted to a cluster according the the *posterior likelyhood*. The adjustment step now corresponds to all data points, not just the nearest ones. As a result the cluster center tend not to move so far way. The cluster centers converge to the same locations, but the correspondences are all alive. There is not a zero-one correspondence.

Each data point is generated from a *mixture*. There are K clusters corresponding to a class. We draw one at random

$$p(x) = \sum_{i=1}^{M} p(c = i) \cdot P(x|c = c = i)$$

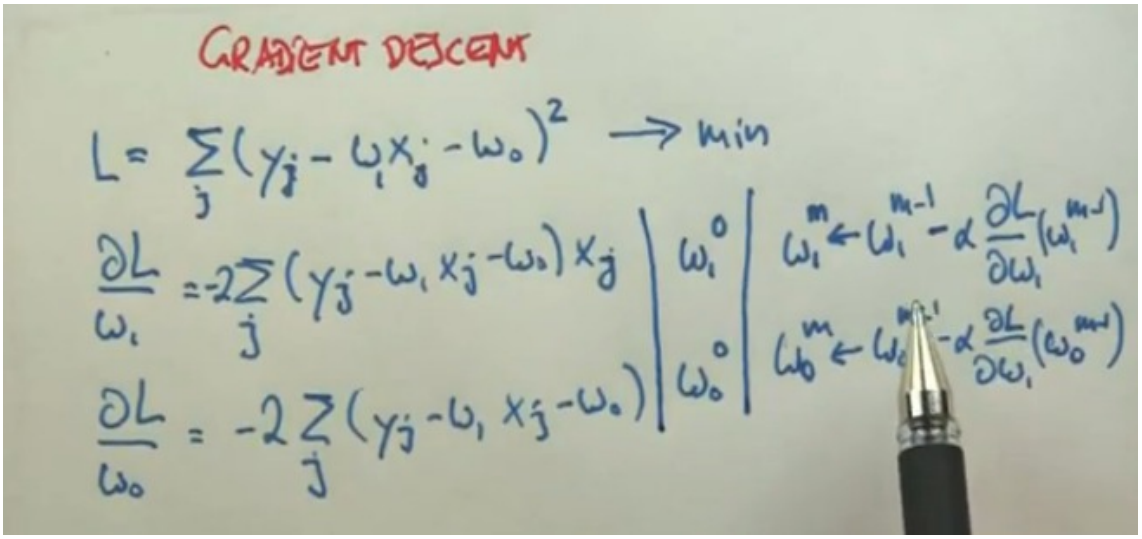


Fig. 2.22. The Gradient descent algorithm

Each cluster center. has a multidimensional gaussian attached. The unknowns are the probabilities for each cluster center π_i (the $p(c = i)$ s and the μ_i and in the general case $\mu_i \sum_i$ for each individual Gaussian.

The algorithm: We assume that we know the π_i and σ_is. The E-step of the algorithm is:

$$e_{i,j} = \pi_i \cdot (2\pi)^{\mu/2} \left| \sum \right|^{-i} \exp\left(\frac{1}{2}\right) (x_j - \mu_i)^T \sum^{-1} (x_j - \mu_i)$$

$e_{i,j}$ the probability tha the j-th data point belongs to cluster center i.

In the M-step we figure out what these probabilities should have been.

$$\begin{array}{lll} \pi_i & \leftarrow & \sum_j e_{ij}/M \\ \mu_i & \leftarrow & \sum_j e_{ij} x_j / \sum_j e_{ij} \\ \sum_i & \rightarrow & \sum_j e_{ij} (x_j - \mu_i)^T (x_j - \mu_i) / \sum_j e_{ij} \end{array}$$

These are the same calculations that we used before to find aussians, but they are weighted.

But how many clusters? In reality we don't know that and will have to find out. The justification test is based of a test based on the negative log likelyhood and a penalty for each cluster. In particular you want to minimize this function

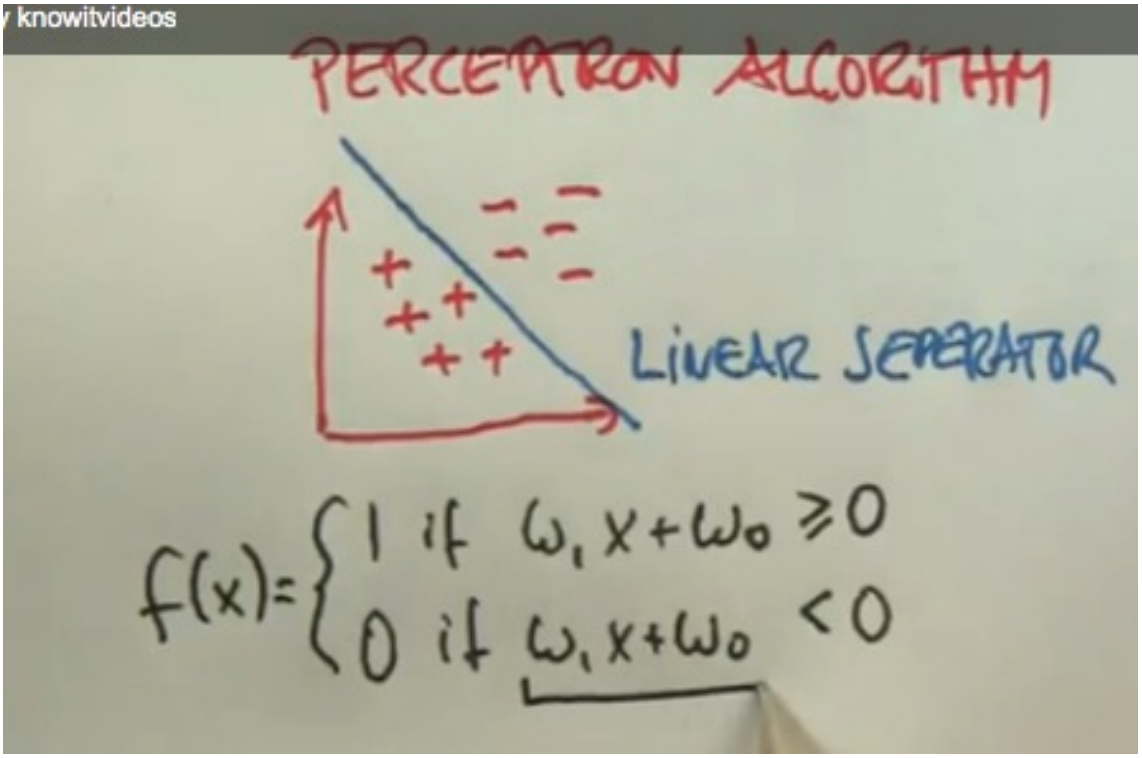


Fig. 2.23. The perceptron algorithm

```
\[
  -\sum_j\log p\parens{x_j|\sigma_\sum_i k} + \mbox{cost}\cdot k
\]
```

We maxzimize the posterior probability of data. The logarithm is a monotonic function and the negative term in front is to make the maximization problem into a minimization problem.

We have a constant cost per cluster. The typical case is to guess an initial k, run the EM; remove unecessary clusters, create some new random clsuters at random and run the algorithm again.

To some extent this algorithm also avoids local minimum problems. By restarting a few times with random clusters this tend to give much better results and is highly recommended for any implementation.

We have learned about K-means and expectation maximization. Both need to know the number or clusters.

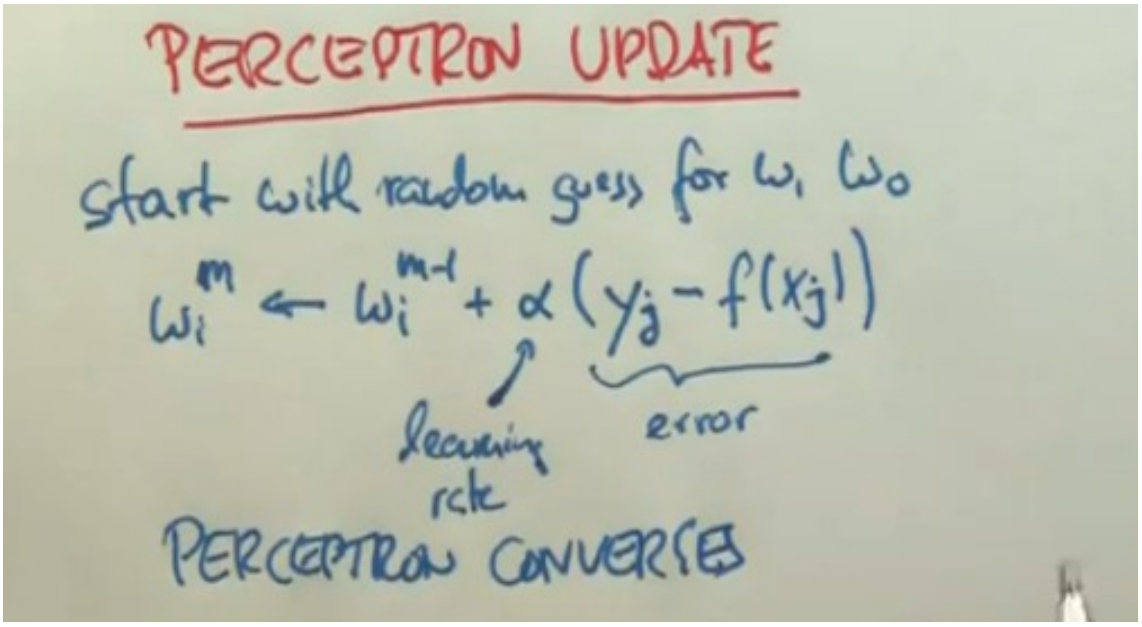


Fig. 2.24. Perceptron updates

2.2.3. Dimensionality reduction

We will talk about linar dimensionality reduction. We wish to find a linear subspace on which to project the data.

The algorithm is

- Calculate the *Gaussian*
- Calculate the *eigenvectors* and *eigenvalues* of the covariance matrix of the Gaussian.
- Pick eigenvectors with maximum eigenvalues
- Project data onto your chosen eigenvectors

This is standard statistics material and you can find it in many linear algebra class.

Dimensional reduction may look a bit weird in two dimensions, but for high dimensions it's not weird at all, just look at the eigenfaces (fig.):

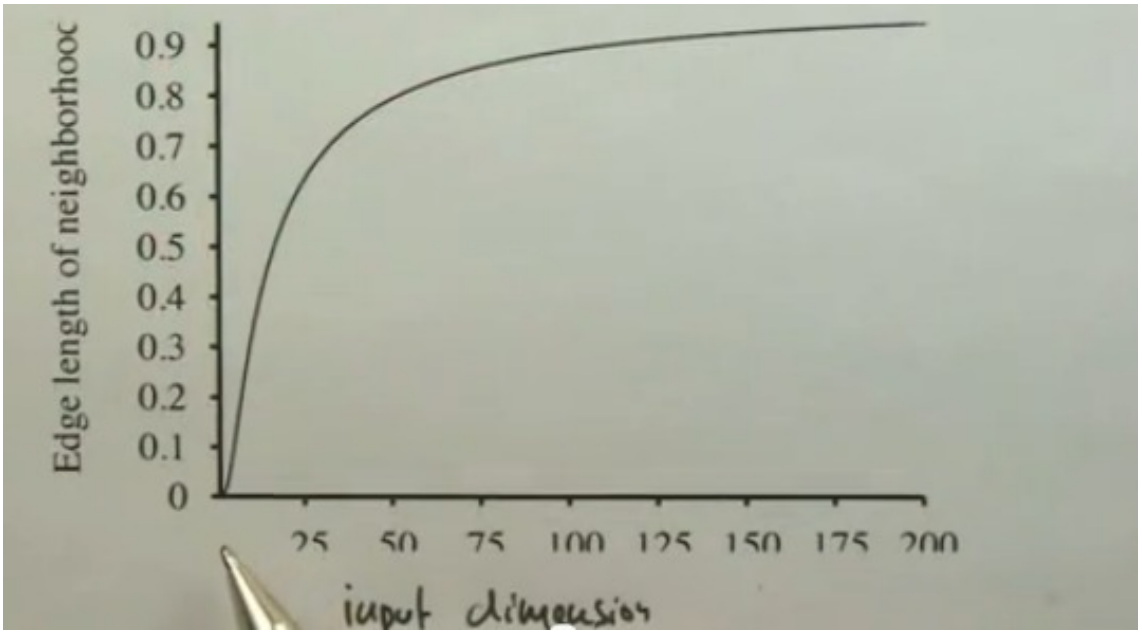


Fig. 2.25. The graph length as a function of the dimensions in the dat set. Conclusion: K-means don't work that well for higher order data.

There is a smallish subspace of pictures that represents faces. This is the *principal component analysis* (*PCA* thing.

In figure ?? we see first (upper left) the average face, then the eigenfaces associated with the image database. The faces are then projected down on the space of eigenfaces. Usually a dozen or so eigenfaces suffice, and if the pictures are 2500 pixels each, we have just mapped a 1500 dimensional space down to a dozen (twelve) or so dimensional space.

Thrun has used eigenvector methods in his own research. He scanned people (thin/fat, tall, etc.) in 3D and to determine if there is a latent lower dimensional space that is sufficient to determine the different physiques and postures people can assume.

It turns out that that this is possible. Only three dimensions suffice to work on physique. Our surfaces are represented by tens of thousands of data points. The method is called "scape"

Use surface meshes. Uses sampels of pose variations from each subject. The model can both represent articulated motion and non-rigid muscle deformation motion

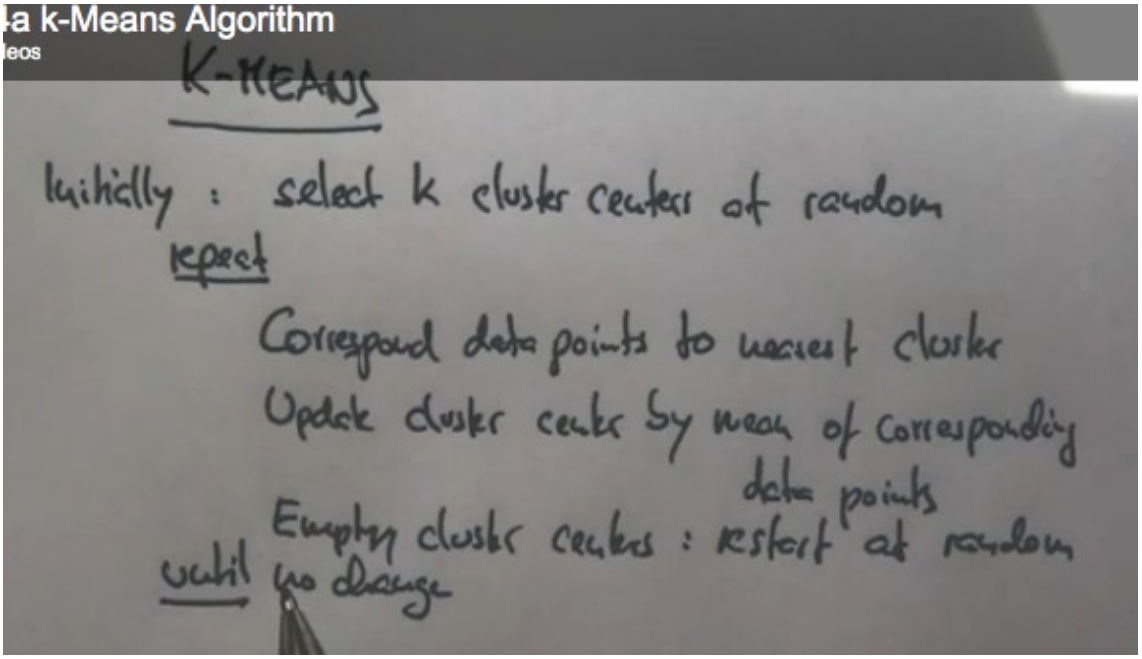


Fig. 2.26. The K-means algorithm in a nutshell

(that is really cool btw). The model can also represent a wide variety of body shapes. Both shape and pose can be varied simultaneousy Can be used for shape completion to estimate the entire shape. E.g. if we only have a scan of the back of a person, we can estimate the front half.

Mesh completion is possible even when neither the person nor the pose exists in the original training set.

Shape completion can be used to complete time series. Starting from motion capture markers, the movement of a "puppet" can be synthesized.

The trick has been to define picewise linear og even nonlinear patches that the data is matched to. This is not dissimilar to K-nearest neigbour methods, but common methods used today are *local linear embedding* and *iso map* methods. There is tons of info on these methods on the world wide web.

2.2.4. Spectral clustering

Both EM and K-means will fail very badly to cluster data sets like the one in fig **??**. The difference is the affinity, or the proximity that affects the clustering, not

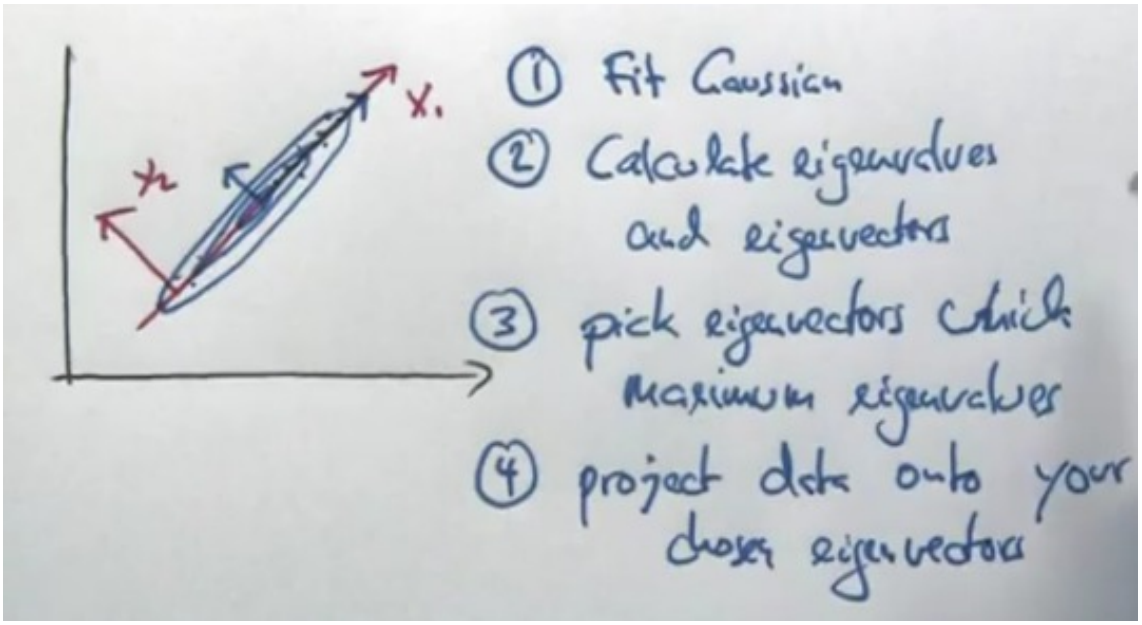


Fig. 2.27. Fitting a multidimensional Gaussian to a dataset

the distance between points. *Spectral clustering* is a method that hellp us.

Data is stuffed into an affinity matrix where everything is mapped to the "affinity" to the other points. Quadratic distance is one option.

The *affinity matrix* is a a *rank deficient matrix*. The vectors correspoinding to items in the same clusters are very similar to each other, but not to the vectors representing elements in the other clusters. This kind of situation is *easily* addressed by *principal component analysis* (PCA). We just pop the affinity matrix in and get a set of eigenvectors that represents the clusters, project each individual point to the cluster with the eigenvector it is less distanced from and we're done. Mind you, we should only use the eigenvectors with large eigenvalues since the eigenvectors with low eigenvalues represents noise (most likely).

The dimensionality is the number of large eigenvalues. This is called *affinity based clustering* or *spectral clustering*.

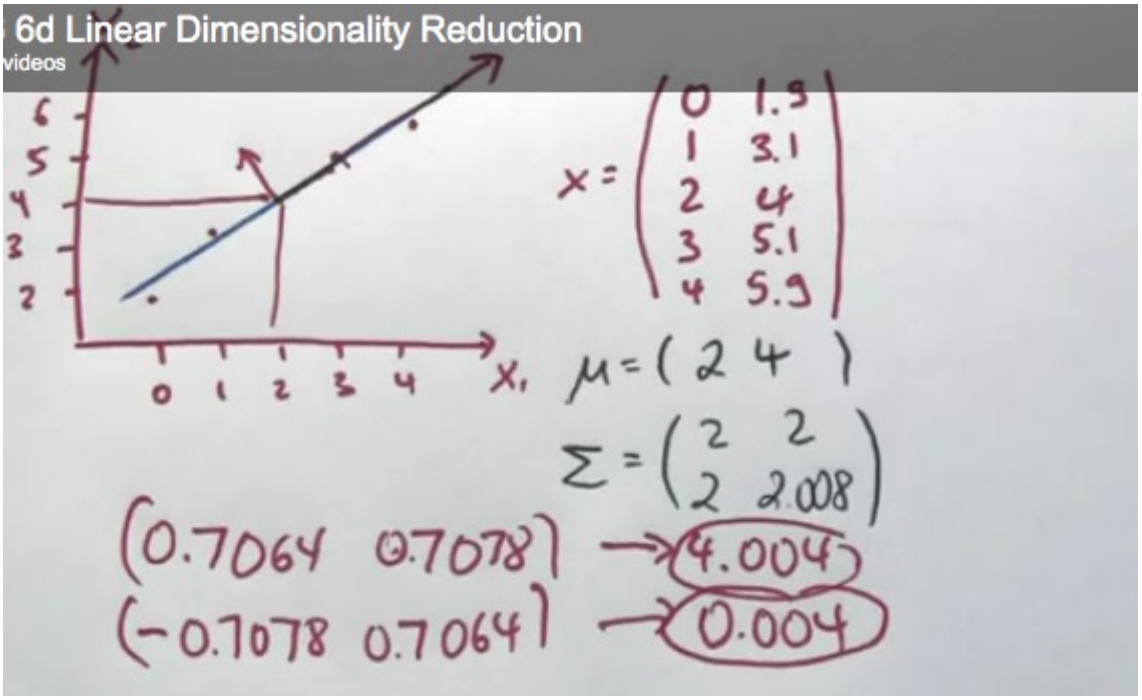


Fig. 2.28. Picking out the eigenvectors of a dataset

2.3. Supervised v.s. unsupervised learning

Unsupervised learning paradigm less explored than supervised learnning. Getting data is easy, but getting labels is hard. It's one of the most interesting research problesems.

In between there are *semi-supervised* or *self supervised* learning methods, and they use elements of both supervised and unsupervised methods. The robot Stanley (fig ?? on page 22) used its own sensors to generate labels on the fly.

Fig. 2.29. A slide created by Santioago Serrano from MIT containing a set of faces, used to detect "eigenfaces" that describe a low(ish) dimensional structure spanning this set of faces

Fig. 2.30. Eigenenfaces based on the dataset in **??**

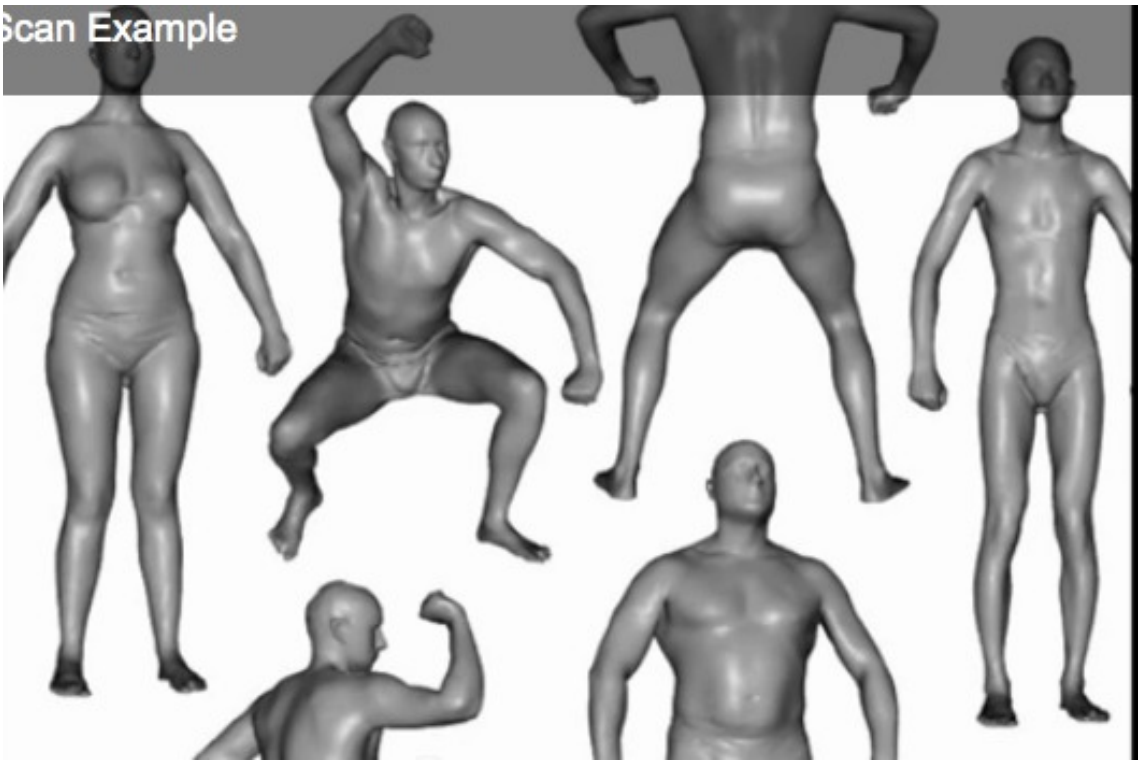


Fig. 2.31. An example of body postures scanned with a laser scanner. These bodies and postures can be modelled as a surprisingly low dimensiona vector.

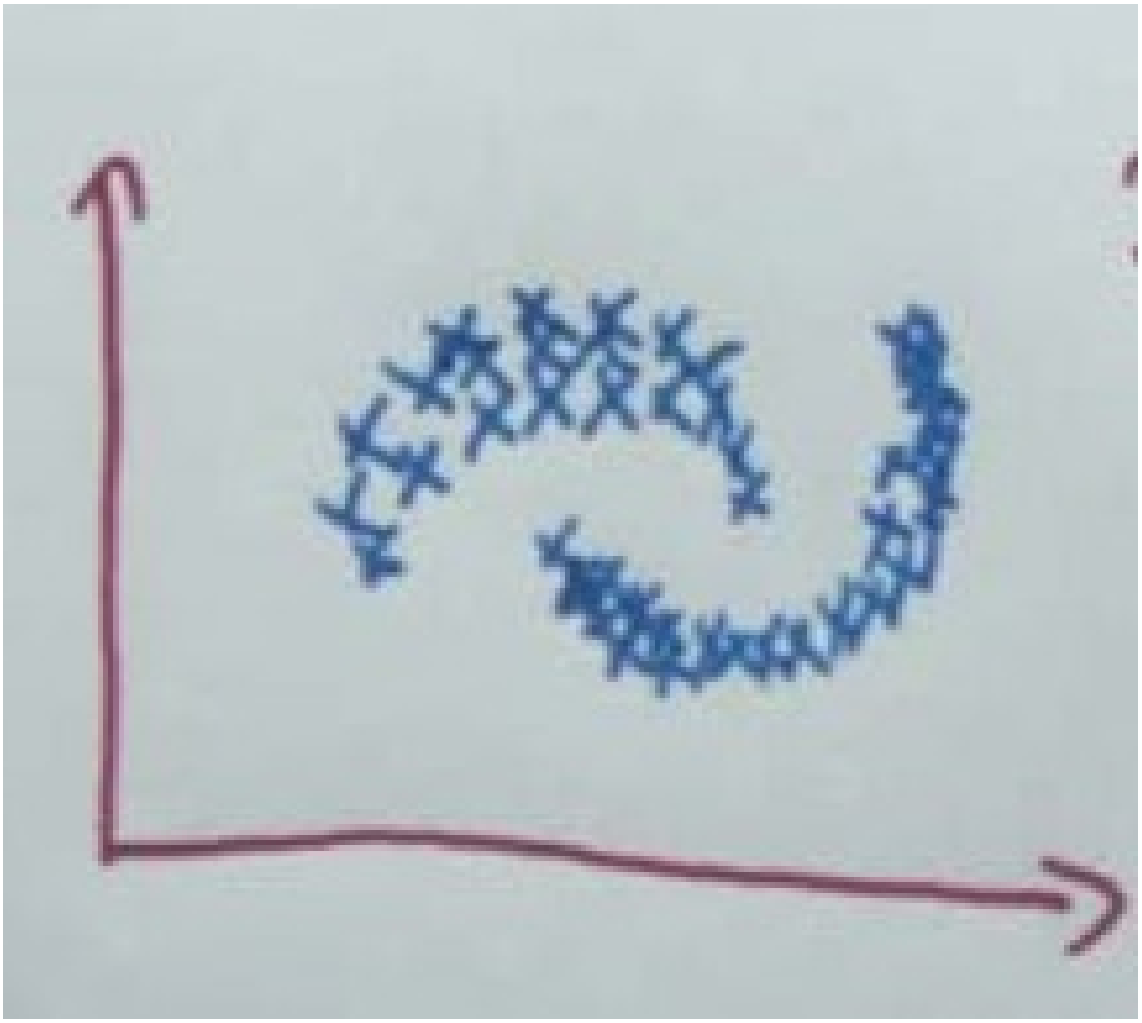


Fig. 2.32. Banana clusters. Not ideal for the K-Means cluster algoritm. Affinity clustering works much better

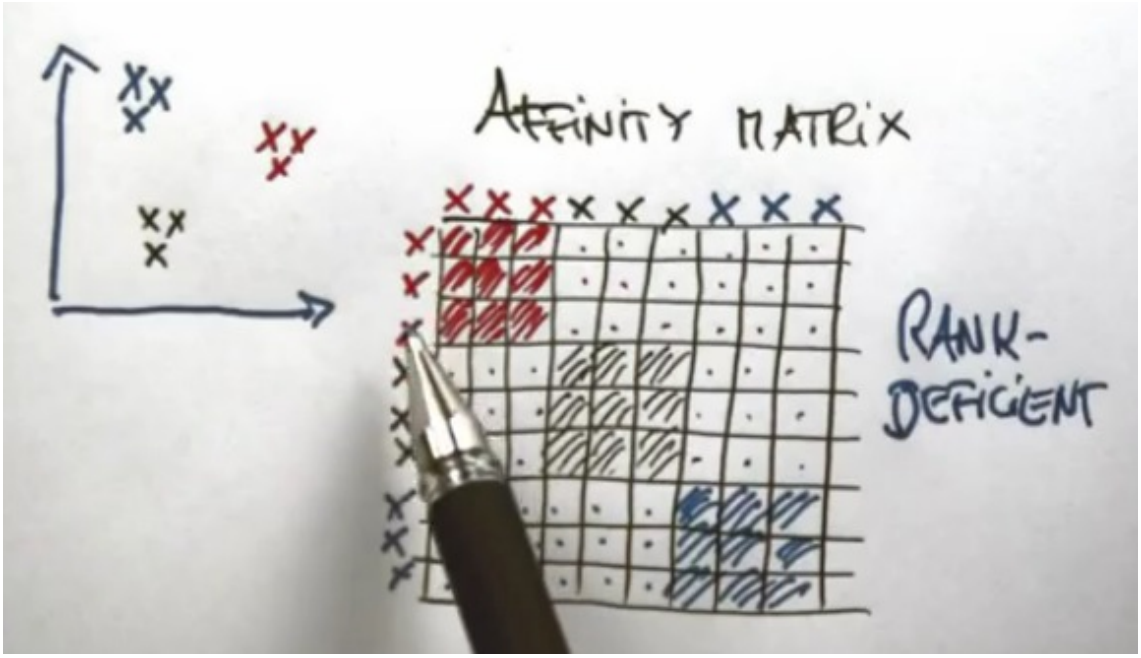


Fig. 2.33. An affinity matrix

Fig. 2.34. The principal component analysis (PCA) algorithm does its magic

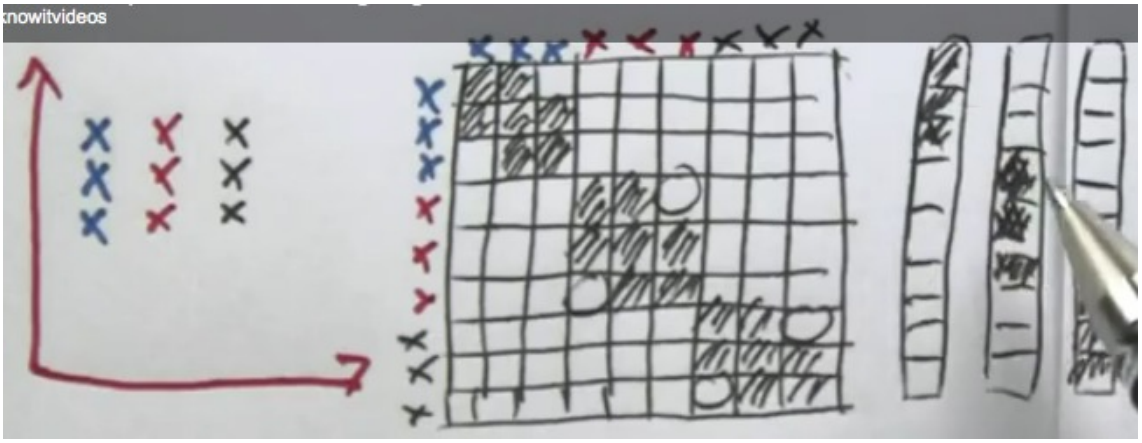


Fig. 2.35. Spectral clustering. Clustering based on closeness to an eigenvector.

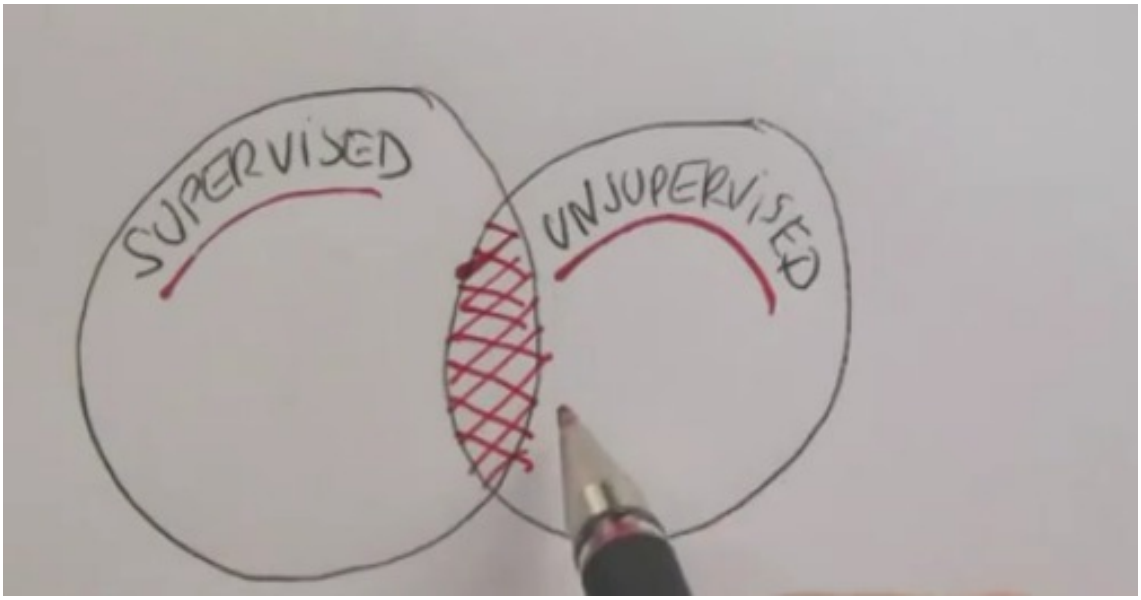


Fig. 2.36. Venn diagram of supervised and unsupervised learning algorithms

3. Representation with logic

AI push agiainst complexity in thrree direction. Agents, environments, and also representations.

In this section we'll see how logic can be used to model the world.

3.0.1. Propositional logic

We'll start with an exmple: We have a world with a set of propositons:

- B: burglary
- E: earthquake
- A: The alarm going off
- M: Mary calling
- J: John calling.

Tese proposition can be either true or false. Oure beliefs in these statements are either true or false, or unknown. We can combine them using logical operators:

$$(W \vee B) \implies A$$

meaning: "Earthquake or burglary implies alarm". We can also write "alarm implies either john or mary calls" as $A \Rightarrow (A \wedge N$,

Biconditional: $\Leftrightarrow$ means that implications flows in both directions.

Negation: $J \Rightarrow \neg M$.

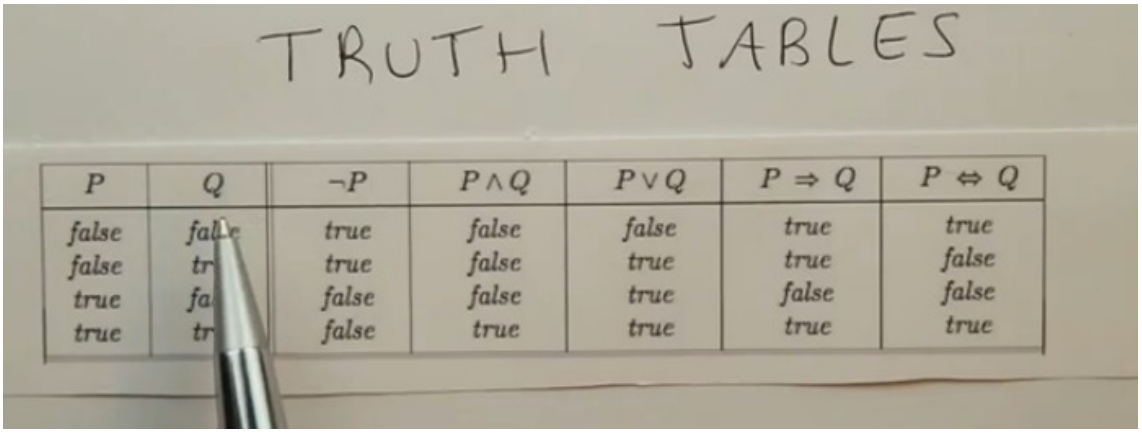


Fig. 3.1. A truth table

A propositional value is true or false with respect to a *model*, and we can determine truth bi evaluating if the statement is or isn't true as dictated by the model.

One method is to construct truth tables (as in fig ??). Lesson from rmz: Don't try to calculate too large truth tables by hand. It always ends up in tears :-) Write a small program, it's in general both simpler and safer :-) (the exception is when some clever trick can be applied by hand, so it's alway smart to try to be clever, but then to give up fairly quickly and use brute force).

A *valid sentence* is true in any model. A *satisfiable sentence* is true in some models, but not all.

Propositional logic is av very powerful tool for what it does. However, no ability to handle uncertainty, can't talk about events that are true or false in the world. Neither we can't talk about relations between objects. Finally there are no shortcuts to talk about many thing (e.g. "every location is free of dirt"), we need a sentence for every possibility

3.0.2. First order logic

We'll talk about FOL and propositional logic and probability theory wrt what they say about the world (the *ontological committment*), what types of beliefs agents can have based on these logics (the *epistomological commitment*).

In FOL we have relations between objects in the world, objects and functions on those objects. Our beliefs can be true, false or unknown.

In PL we have facts, but they can be only true or false.

In probability theory we have the same facts as in propositional logic, but the beliefs can be in ranges from zero to one.

Another way to look at representations is to break them into *atomic representation* and *factored representation*. Atomic has no components. They are either identical or not (as used in the search problems we looked at). A factored representation has multiple facets that can be other types than boolean. The most complex representation is called *structured representation*. Can include variables, relations between objects, complex representations and relations. This latest is what we see in programming languages and databases ("structured query languages") and that's a more powerful representation, and that's what we get in first order logic.

Like in propositional model we start with a *model* that assigns a value to each propositional value, e.g.:

$$\{P : T, Q : T\}$$

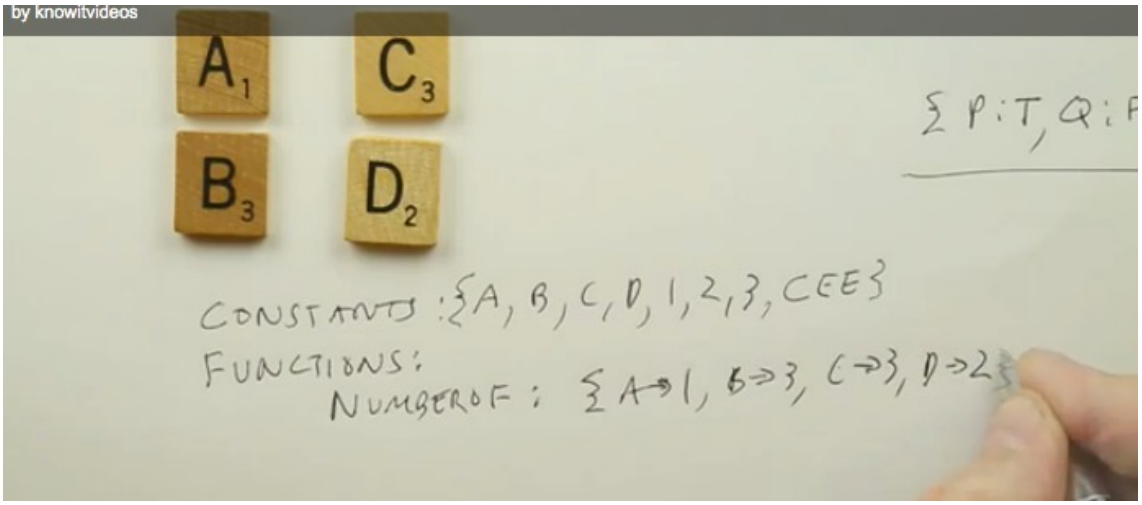


Fig. 3.2. First order logic model

In first order models we have more complex models. We can have a set of objects, a *set of constants*, e.g. A, B, C, D , 1,2,3. But there is no need to have one to one correspondence referring to the same objects. There can also be objects without any names. We also have a set of *functions* that maps from objects to objects, e.g. the numberof function:

$$\text{numberof} : \{A \mapsto 1, B \mapsto 3, C \mapsto 3, D \mapsto 2 \ldots\}$$

In addition to functions we can have *relations*, e.g. "above" etc. Relations can be n-ary (unary, binary etc.) that are (can be) represented by tuples. There may be 0-ary relations, like "rainy"

Rmz: *Didn't understand that*

The syntax of first order logic consists of sentences that statements about relations (e.g. "vowel(A)"). There is always equivalence (=), in addition there are the elements from propositional logic $\wedge, \vee, \neg, \Rightarrow, \Leftrightarrow, ()$.

The terms can be constants (uppercase) or variables (lowercase) or function invocations.

When the quantifier is omitted we can assume a forall quantifier that is implicit. Usually the forall is used with a sentence that qualifies a statement, e.g. "for all object that are vowels, there will be only one of them", since the universe of all objects is so large that we usually don't want to say much about it.

First order logic means that the relations is about objects not relations. *Higher order* objects talk about relationships between reations. That means that sentences in higher order logic is in general invalid in first order logic.

When you define something, you usually want a definition with a bidirecitonal implication.

4. Planning

Planning is in some sense the core of AI. A* and similar algorithms for problem solving search in a state space are a great planning tools, but they only work when the environment is deterministic and fully observable. In this unit we will se how to relax those constraints.

Fig. 4.1. A blindfolded navigator will tend to follow a mostly random walk.

Looking at the Romania problem again. The A* does all the planning ahead and then executes. In real life this doesn't work very well. People are incapable of going in straight lines without external references. The yellow hiker could see shadows (it was sunny) and that helped to make a straight line. We need some feedback from the environment. We need to interleave planning and execution.

In stochastic environments we have to be able to deal with *contingencies*. *Multi agent environments* means that we must react to the other agents. *Partial observability* also introduces problems. An example is a roadsign that may or may not indicate that a road is closed, but we can't observe it until we're near the sign.

In addition to properties with the environment, we may have lack of knowledge. The map may be incomplete or missing. Often we need to deal with *plans that are hierarchical*. Even for a perfectly planned roadtrip we will still have to do some micro-planning on the way, steering left, right, moving the pedals up and down to accomodate curves and things that are not on the map and that we couldn't plan for even if we don't ever deviate from the actually planned path.

Most of this can be changed by changing our point of view from planning with *world states* to instead plan from *belief states*.

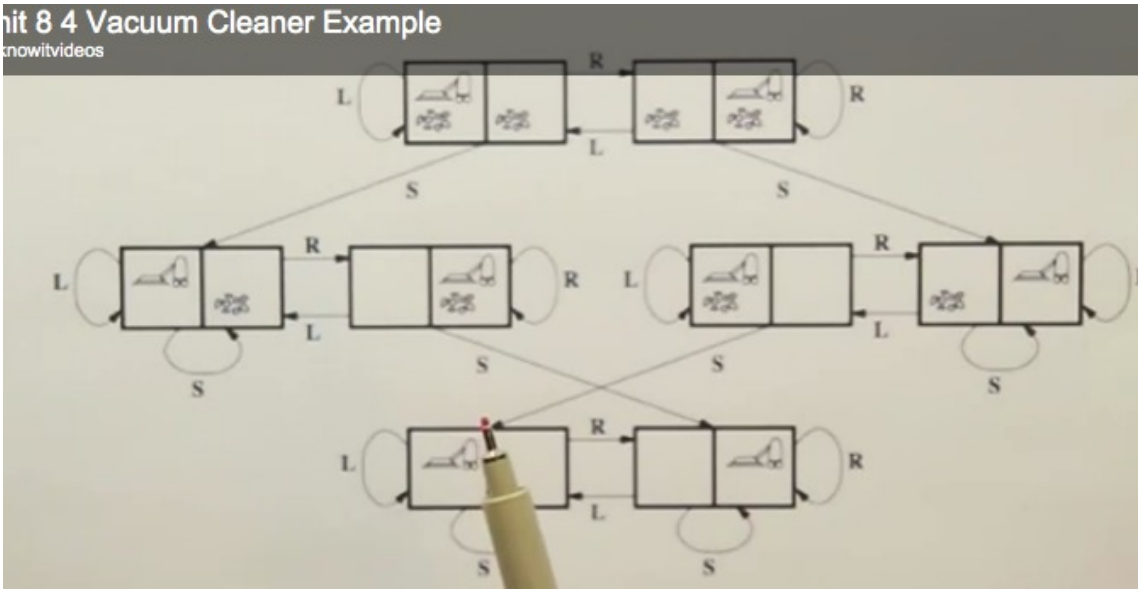


Fig. 4.2. Vacum world

We're looking at the vacum world with eight states (dirt/clean, left right). What if the sensors of the cleaner breaks down? We don't which state we're in. We must assume we are in any of the possible worlds. However, we can then search in the state of belief states.

Sensorless planning in a deterministic world

If we execute action we can get knowledge about the world without sensors. For instance if we move right we know we are in the right square: Either we were in the left space and moved right, or we were in the right space and bumped against

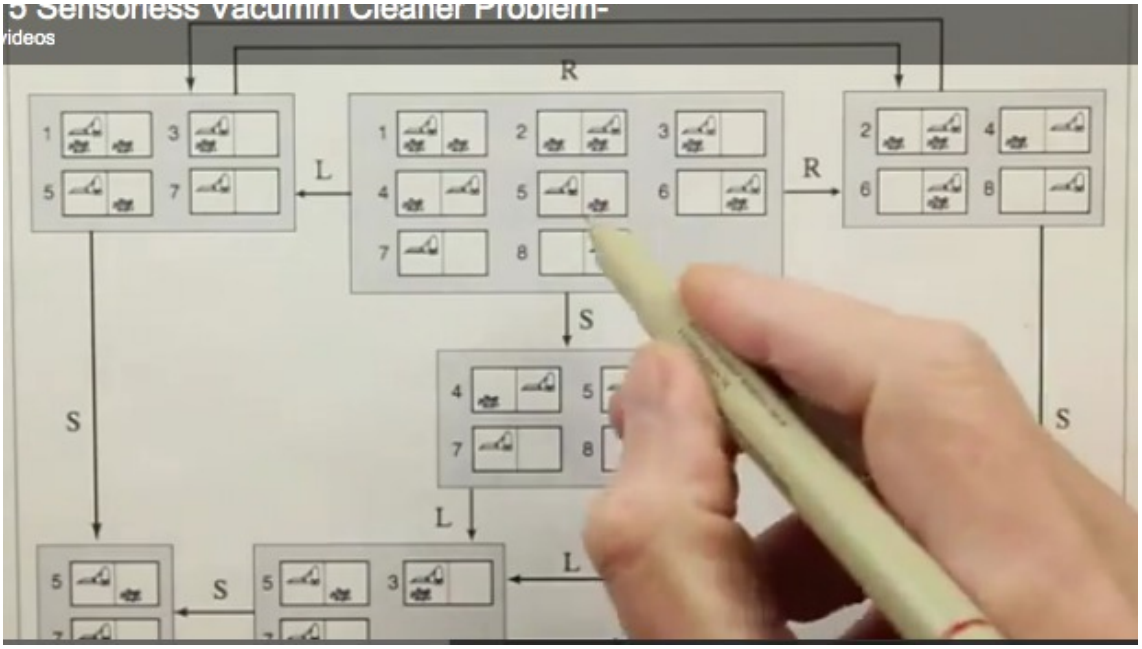


Fig. 4.3. Belief states in the vacum world

the wall and stayed there. We know know more about the world, we only have four possible worlds even if we haven't observed anything.

Another interesting thing is that unlike in the world state, going left and going right are not inverses of each other. If we go right, then left, we know we are in the left square and that we came there from being in the right square. This means that it's possible to formulate a plan to reach a goal without *ever* observing the world. Such plans are called *conformant plans*. For example, if our goal is to be in a clean state, all we have to do is suck.

Partial sensing in a deterministic world

The cleaner can see if it's dirt in its current location. In a deterministic world state the size of the belief state will stay the same or decrease. Observations works the oter way, we are taking the current belief state and partition it up. Observation of alone can't introduce new states, but it can make us less confused than we were before the observation.

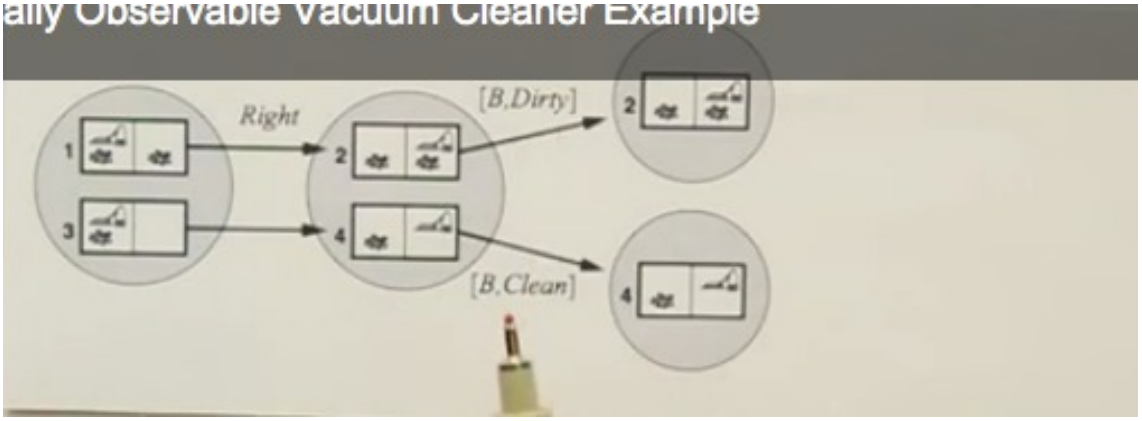


Fig. 4.4. A partially observable vacum cleaner's world

Stochastic environment

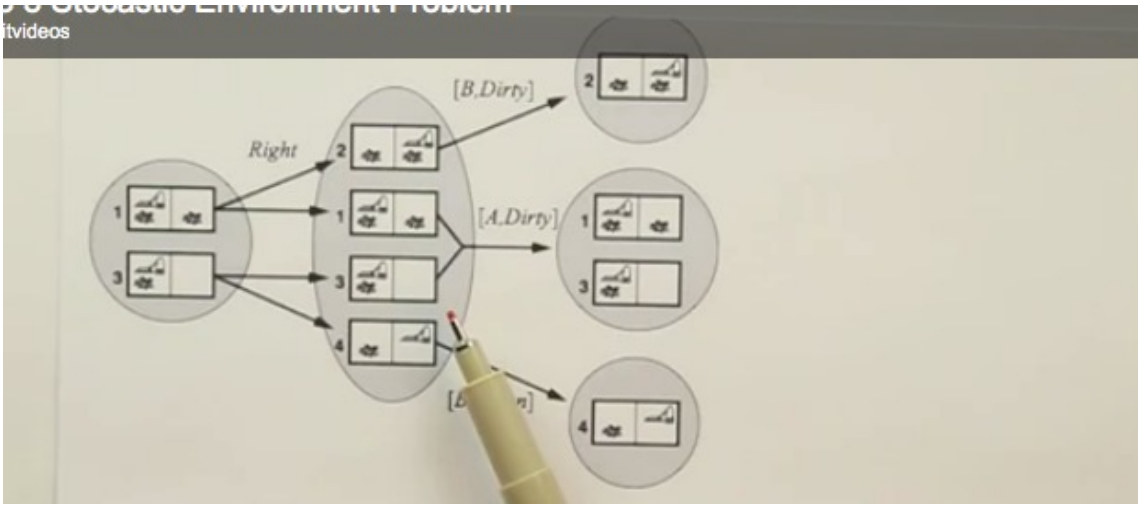


Fig. 4.5. A stochastic environment: Actions may or may not do what you think they do.

Sometimes the wheels slip so sometimes you stay in the same location and sometimes you move when you tell the robot to move. We are assuming that the suck action works perfectly. Often the state space expands. The action will increase the uncertainty since we don't know what the result of the action is. That's what *stochastic* means. However, observation still partitions the statespace.

Notation for writing plans, use tree structure not sequence of action. Conditionals.

In general stochastic plans work in the limit.

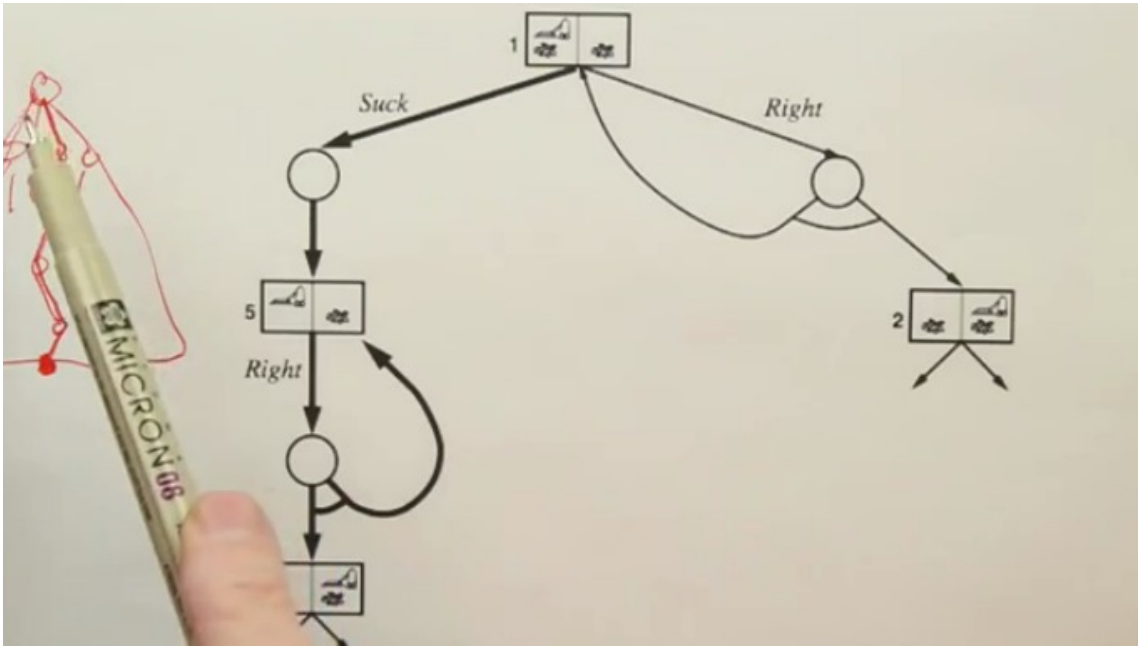


Fig. 4.6. A plan with alternatives notated as branches

However plans may be found by search just as in problem solving. In the notation branches with connecting lines between the outgoing edges are branches in the *plan*, not in the search space. We then search until we have a plan that reach the goal.

In an unbounded solution, what do we need to ensure success in the limit? Every leaf need to be a goal. To have bounded solution we can't have any loops in the plan.

Some people like trees, other like math notation. For deterministic states things are simple expression. For stochastic situations the beliefe states are bigger, but the

Tracking the predict/update cycle

This gives us a calculus of belief states. However, the belief states can get large. There may be succinct representations than just listing them all. For instance, we could have a algebra saying "vacum is on the right". that way we can make small descriptions.

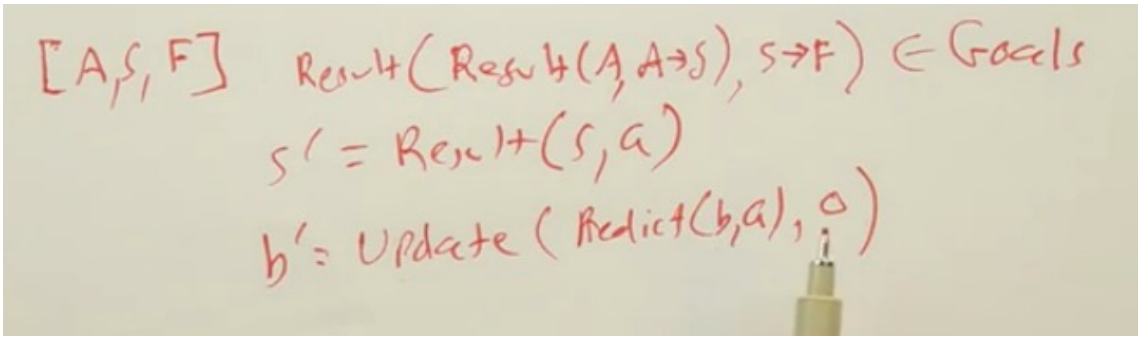


Fig. 4.7. A mathematical notation for planning

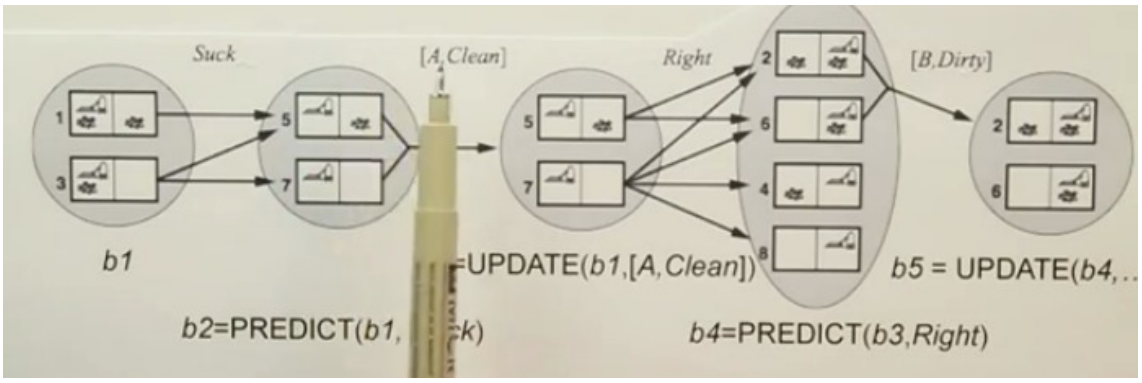


Fig. 4.8. Predicting updates (whatever that means)

There is a notation called *classical planning* that is both a language and a method for describing states. The state space consists of k boolean variables so here are 2^k states in that statespace. For the two-location vacum world there would be three bolean variables. A world set consist of a concrete assignment to all the variables. A belief state has to be a complete assignment in classical planning, however we can also extend classical planning to have partial assignments (e.g. "vacum in A true" and the rest unknown). We can even have a belief state that is an arbitrary formula in bolean algebra.

In classical planning actions are represented by *action schema*. They are called *schema* since there are many possible actions that are similar to each other.

A representation for a plane going from somewhere to somewhere else:

```
Action(Fly(p, x, y) -- plane p from x to y
```

```
precond: plane(p) AND Airport(x) AND
         airport(y) AND
         At(p,x)
effect:  Not At(p,x)) AND at(p, y))
```

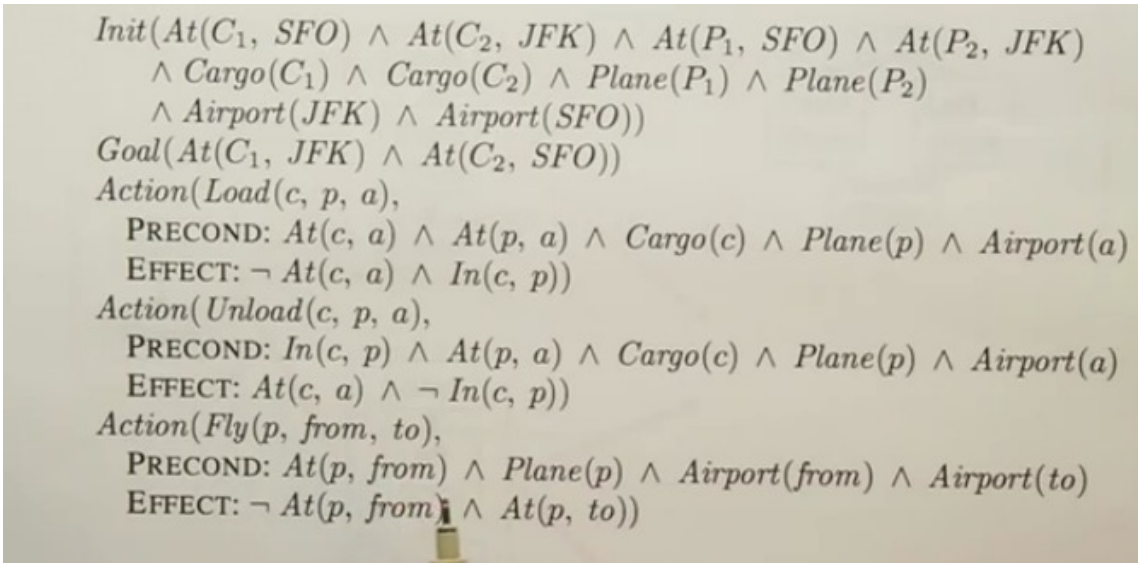


Fig. 4.9. Classical planning

Now, this looks like first order logic, but this is a completely propositional world. We can apply the schema to specific world states.

In figure ?? we see a somewhat more elaborate schema for cargo transports.

4.0.3. Planning

Now we know how to represent states, but how do we do planning? Well, one way is just to do it like we did in problem solving with search. We start in one state, make a graph and search in the graph. This is forward or progressive state space search.

However, since we have this representation we can also use backward or regression search. We can start with the goal state. In problem solving we had the option of searching backwards, but in this situation we have a wide range of real goal states.

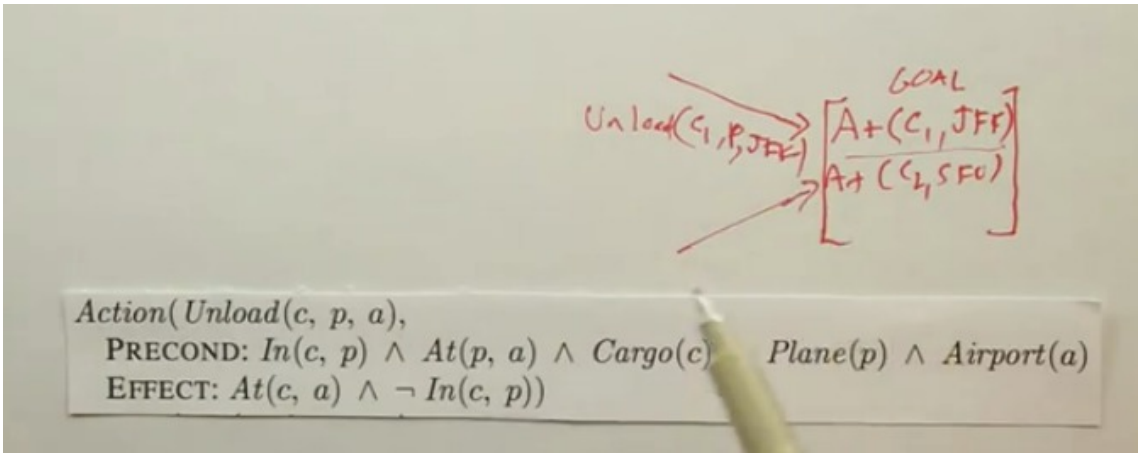


Fig. 4.10. Regression search

What we can do is to look at the possible actions that can result in the desirable goals.

Looking at the action schema we find a matching goal. Backwards search makes sense when we're buying a book :-)

There is another option we have, and that is to *search through the space of plans* insetead of searching through the space of states. We start with a start and end state. That isn't a good plan, so we edit it by adding one of the operators. Add some more elements.

In the eighties this was a popular way of doing things, but today the most popular way of doing things is through forward search. The advantage of forward search is that we can use heuristics, and since forward search deals with concrete planned states it seems easier to come up with good heuristics.

Looking at the for-by-four sliding puzzle. The action is:

```
Action(Slide(t, a, b)
    pre: on(t, a) AND tile (t) AND blank(b)
         AND adjacent(a,b)
 effect: on(t,b) AND blank(A) AND not( on(t,a))
         AND not(blank(b))
```

With this kind of formal representation we can automatically come up heuristics through relaxed representations by throwing out some of the prerequisites. If we

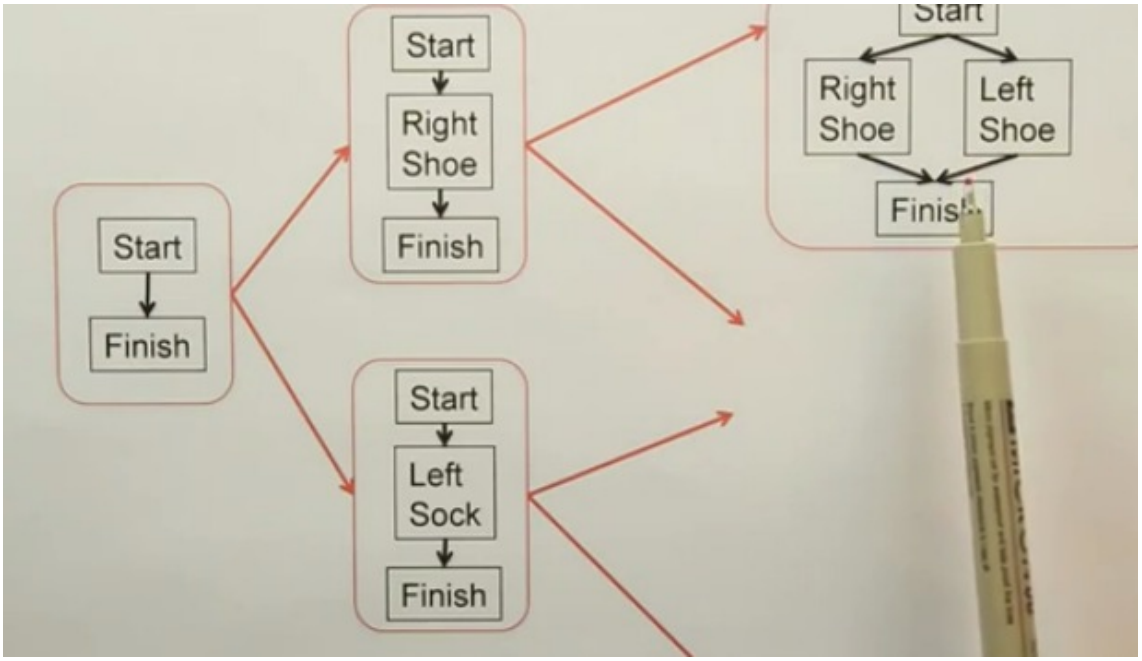


Fig. 4.11. Plan state search

cross out the heuristic that the tile has to be blank we get the "*manhattan distance*", if we throw out the requirement that the tiles has to be adjacent, then then we have *the number of misplaced tiles heuristic*. Many others are possible, like *ignore negative effects* (take away the "not blank b" requirement of the effect"). Programs can perform these editings and thus come up with heuristics programmatically instead of having humans have to come up with heuristics.

4.0.4. Situation calculus

Suppose we have a goal of moving all the cargo from airport A to airport B. We can represent this in first order logic. This is regular first order logic with some convensions. Actions are represented as objects. Then we have situations that are also objects in the logic, but they don't represent states but paths. If we arrive at the same state through two different actions, those would be considered different in situational calculus. There is an initial situation, called S_0. We have a function on situations called "result", so that $S' = \text{reslult}(s, a)$.

Instead of describing the applicable actions, situation calculus instead talks about the actions that are possible in a state. There is a predicate where some precondition

of state s says it is possible to do something. These rules are called *possibility axioms* for actions. Here is one such axiom for the fly action:

```
plane(p,s) AND airport(x,s) AND airport(y,s) at(p,x,s)
 =>
 possible(fly(p,x,y),s)
```

There is a convention in situational calculus that predicates that can change depending on the situation are calle *fluents*. The convension is that they refer to a situation, and that the parameter is the last in the list of parameters.

In classical planning we had action schema and described what changed one step at a time. In situation calculus it turns out it's easier to do it the other way around, it's easier to make one axiom for each fluent that can change.

We use a convention called *successor state axioms*: In general a successor has a precondition like this:

$$\forall a, s \, \text{poss}a, s \Rightarrow (\text{fluent is true} \Leftrightarrow a \, \text{made it true} \vee a \, \text{didn't undo it})$$

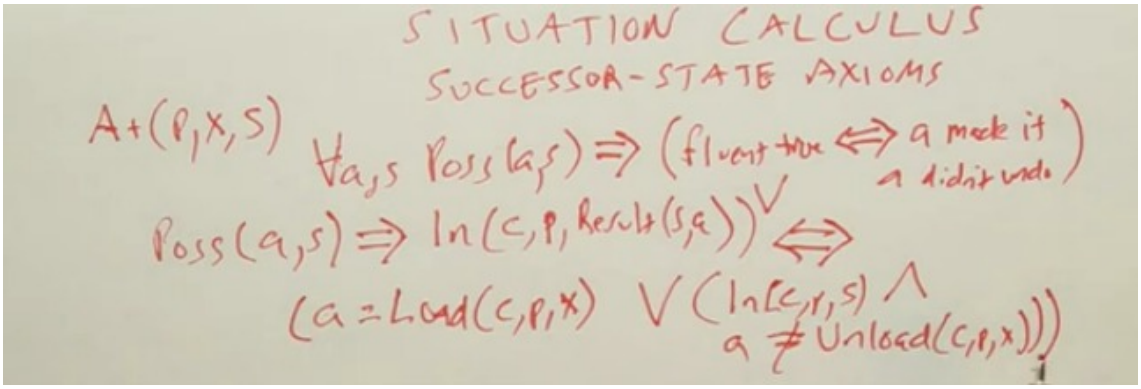


Fig. 4.12. succstateax

5. Planning under uncertainty

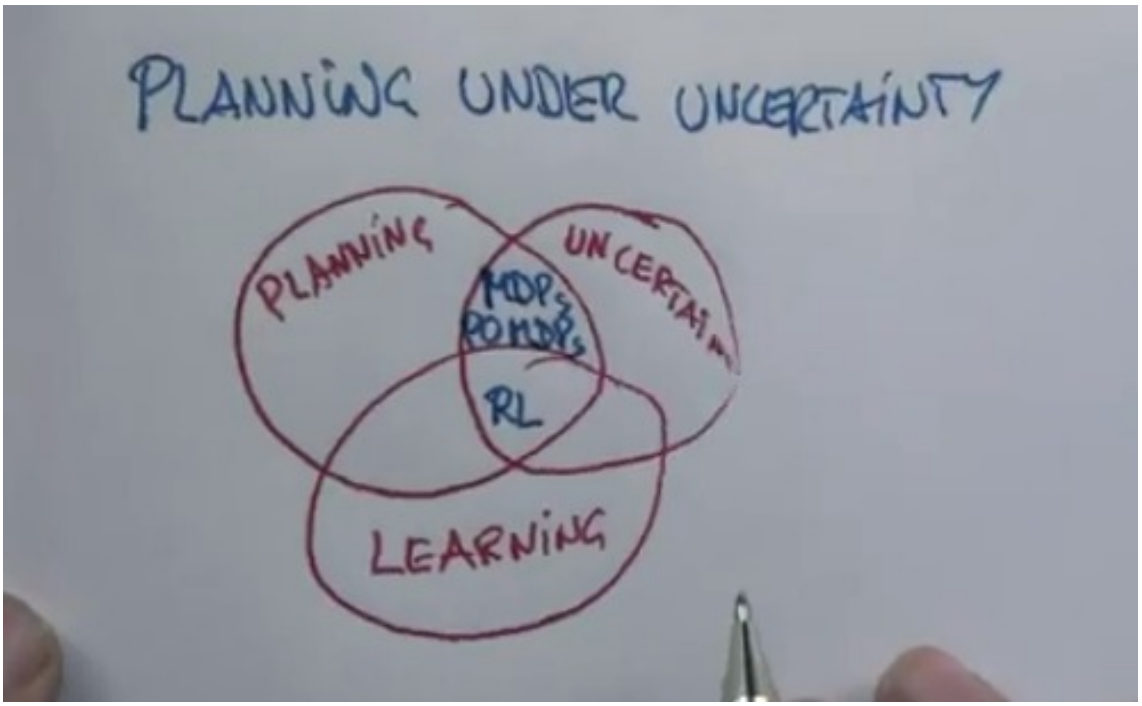


Fig. 5.1. A venn diagram depicting the various types of planning we can use when faced with different types of uncertainty

This puts togheter both planning and uncertain. This is really important because the world is full of uncertainty. We will learn techniques to plan robot actions under uncertainty.

We'll learn about *MDP*s (*Markov Decision Processes*) and *partially Observable Markov processes POMDP*.

Remember the agent tasks that could be classified according to this schema:

	Deterministic	**Stochastic**
Fully observable	A*, Depth first, Breadth first	MDP
Partialy observable		POMDP

Stochastic = the outcome is nondeterministic.

5.0.5. Markov decision process (MDP)

We can think of a graph describing a state machine. It becomes *markov* if the choice of action becomes somewhat random.

A markov decision process is a set of states $s_1, \ldots, s_N$, activities $a_1, \ldots, a_k$ and astate transition matrix:

$$T(s, a, s') = P(s'|a, s)$$

"P(s'—...)" is the the posterior probability of state s' given state a ans action s.

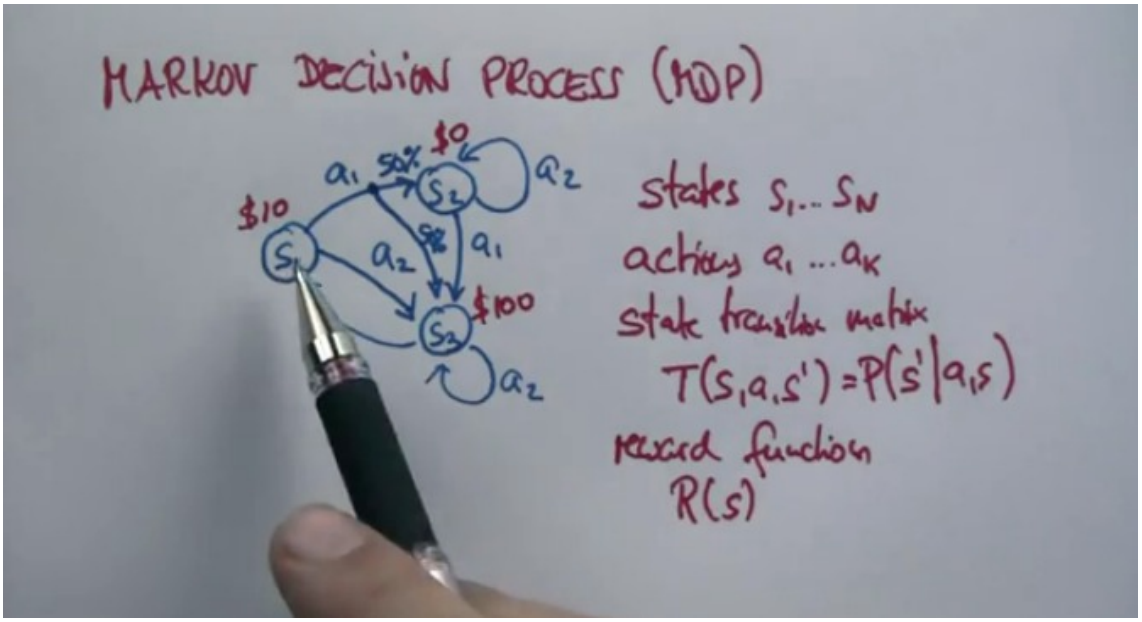


Fig. 5.2. A markov decision process with a reward function

Fig. 5.3. A robot made by Thrun for the Smithsonian institution

We also attach a "reward function" to the state to determine which states are good and which that are not. The planning problem is now to maximize the reward.

We see a lot of interestin robotd.

The environments are stochastic and the planning problems are very challenging

5.0.6. The gridworld

Two goal state (absorbing states). When the wall is hit the outcome is stochastic, in ten percent of the cases it will simply stay put, but with a ten percent probability it will bounce left, and with ten percent probability it will bounce right.

Conventional planning is inadequte here. Instead we use a a *policy*, that assigns an action to every state.

The planning problem thefore becomes one of finding the optimal policy.

Fig. 5.4. A robot made by Thrun for exploring mines in Pennsylvania

Stochastic environents conventional planning

We would create a tree depending on the directions we go. The tree would get a very high branching factor.

Another problem with the search paradigm is that the tree may become very deep. Essentially "infinite loop".

Yet one problem is that many states are visited more than once.

A policy method overcomes all of this.

5.0.7. Policy

The optimal diretion in the gridworld is a bit nontrivial, since it's in general more important to avoid a disastrous outcome than to have a quick path to the goal (due to nondeterminism).

We have a reward function over all states, in the gridworld we give +/- 100 for the goal states, but perhaps -3 for all other states to ensure that we don't end up making very long paths.

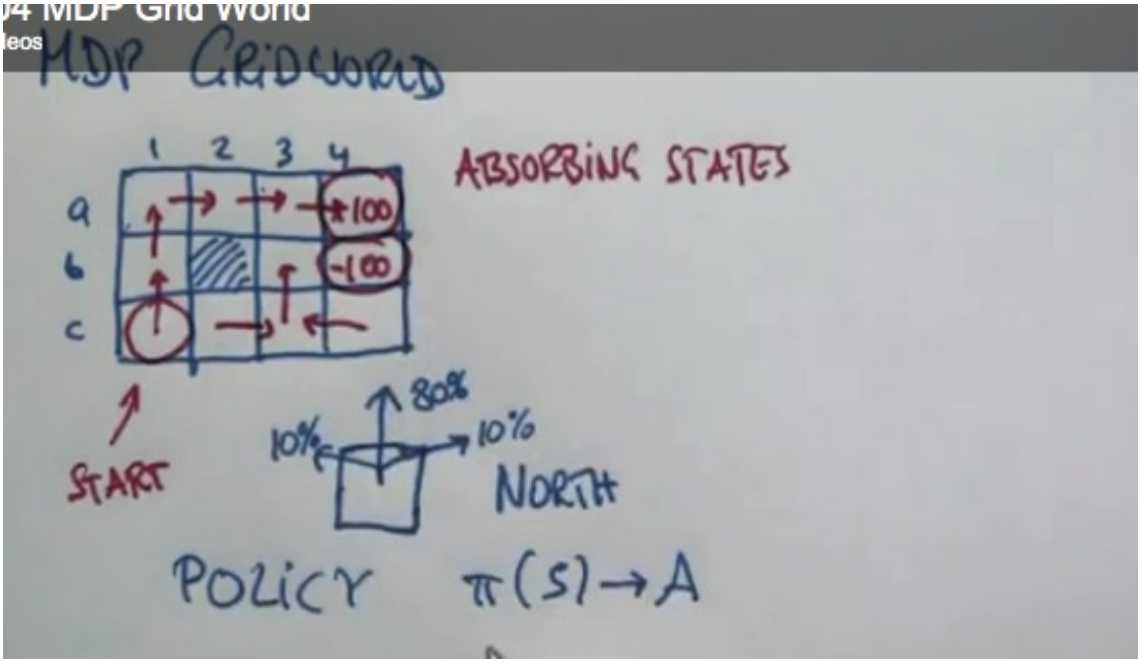


Fig. 5.5. A simple world for a robot to navigate in

We can now state the objective of the MDP:

$$E(\sum_{t=0}^{\infty} R_t) \rightarrow \max$$

What we want to do is to find the policy that maximizes the reward. Sometimes people puts a *discount factor* (γ^t) into the sum of the equation that decays future rewards relative to immediate rewards. This is kind of an incentive to get to the goal as fast as possible and is an alternative to the penalty per step indicated above.

The nice mathematical thing about discount factor is that it keeps the expectation bounded. The expectation in the equation above will always be less than $\leq \frac{1}{1-\gamma}|R_{max}|$.

The definition of the *future sum of discounted rewards* as defined above defines a *value function*:

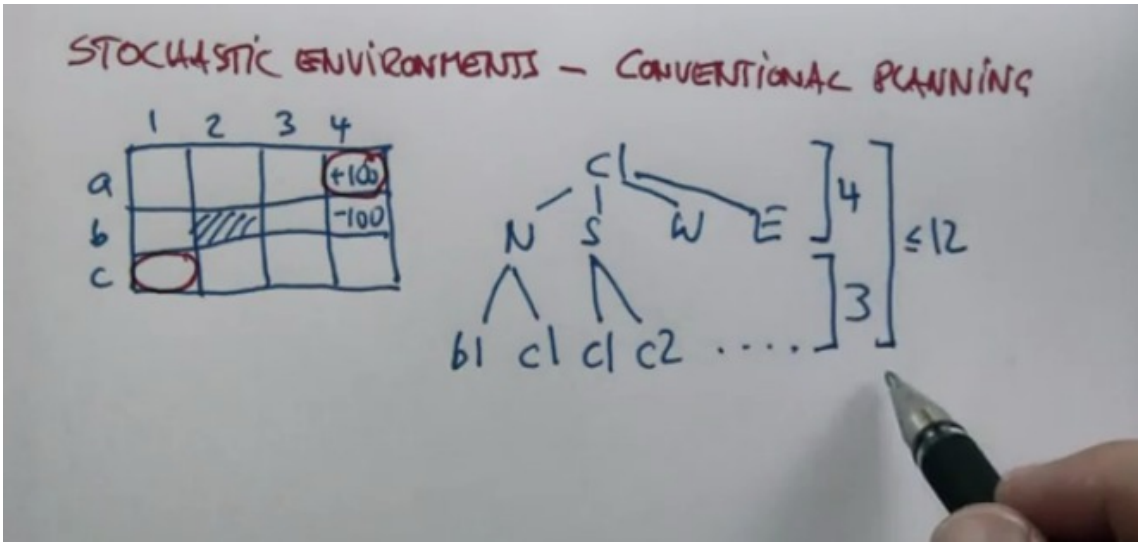


Fig. 5.6. Stochastic conventional planning

$$V^{\pi}(s) = E_{\pi}(\sum_{t=0}^{\infty} \gamma^t \, R_t | s_0 = s)$$

This is the expected sum of discounted future rewards provided that we start in state s and execute policy π.

We can now compute the average reward that will be received given any choice. There is a well defined expectation and we are going to use them a lot. Planning not only depends on calculating these values a lot, it will also turn out that we will be able to find better policies as a result of this work too.

The value function is a potential function that leads from the goals to all the other states so that *hillclimbing* leads to finding the shortest path .

The algorithm is a recursive algorithm that converges to the best path by following the gradient.

Rmz: *This must mean that the value function is convex, but I don't yet see how that can generally be the case, at least for interesting topologies of action sequences, so there must be something that is left out in the description so far*

5.0.8. Value iteration

This is a *truely magical algorithm* according to Thrun. It will recursively calculate the value function to find the *optimal value function* an from that we can derive an optimal policy.

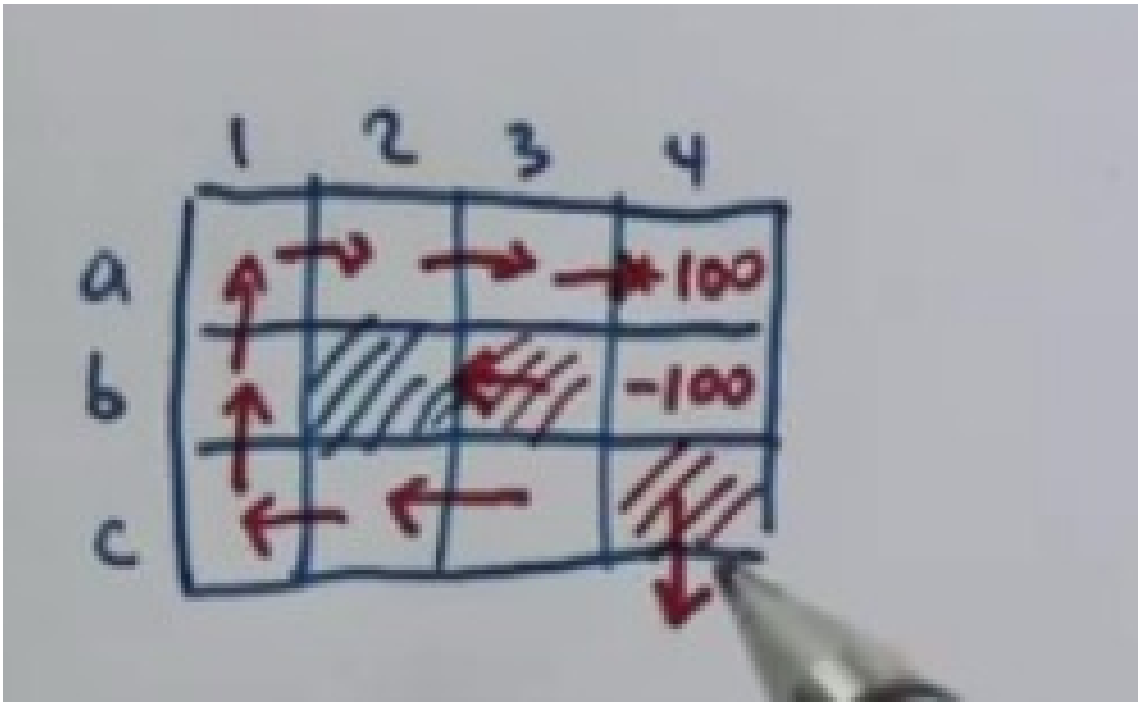


Fig. 5.7. Finding an optimal path in the gridworld

Start with a value function with a value zero everywere, then diffuse the value from the positive absorbing state throughout the statespace.

A recursive function to calculate the value:

$$V(s) \leftarrow (\max_a \gamma \sum_{s'} P(s'|s,a)V(s')) + R(s)$$

We compute a value recursively by taking the max of all the possible successor states. This equation is called *back-up*. In terminal states we just assign $R(s)$. The process described over converges, and when it does, the we just replace the $\rightarrow$ with an "=", and when the equality holds we have what is called a *bellman equality* or a *bellman equation*.

The optimal policy is now define:

$$\pi(s) = \text{argmax}_a \sum_{s'} P(s'|s,a)V(s')$$

Once the values has been backed up, this is the way to find the optimal thing to do.

Markov decision processes are fully observable but have stochastic effects. We seek to maximise a reward function, the effective is to optize the effective amortized reward. The solution was to use value iteration where we are using a value.

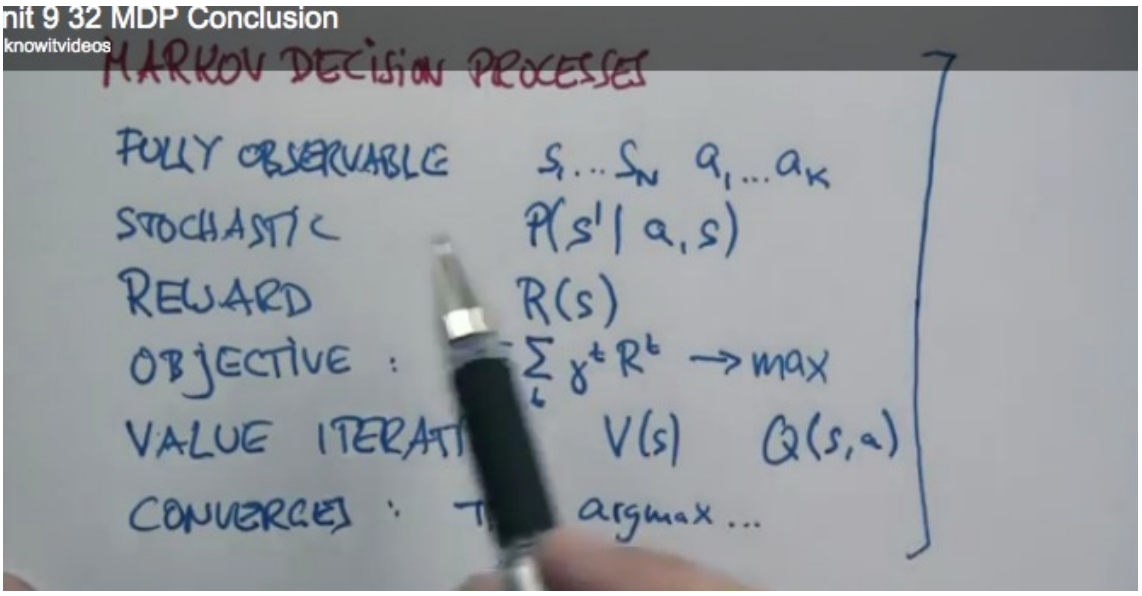


Fig. 5.8. Notation and terms used when describing Markov decision processes

Fully observable:	$s_1, \ldots, s_N$ $a_1, \ldots, a_K$
Stochastic:	$P(s'\|a, s)$
Reward:	$R(s)$
Objective:	$E\sum_t \gamma^t R^t \rightarrow \max$
Value iteration:	$V(s)$ Sometimes pairs $Q(s, a)$
Converges:	$\pi_{\text{policy}} = \text{argmax}$

The key takeaway is that we make entire field of decisions, a so called *policy* that gives optimal decisions for every state.

We'll now get back to partial observability. We'll not in this class go into all of the advanced technicques that has been used in real cases (like the car etc.)

	Deterministic	**Stochastic**
hline Fully observable	A*, Depth first, Breadth first	MDP
Partialy observable		POMDP

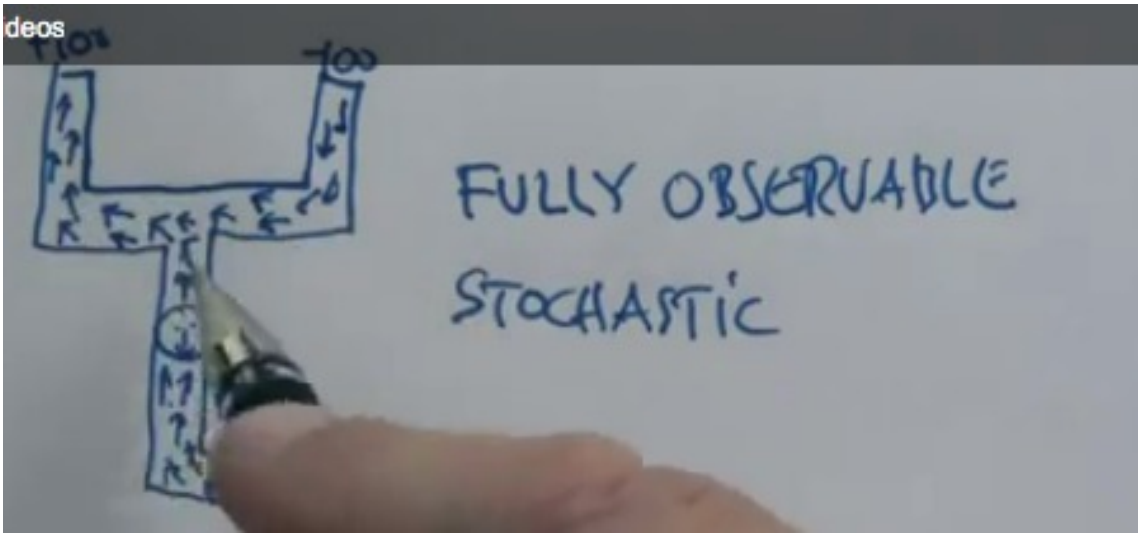


Fig. 5.9. Gradient using a partialy observable Markov decision process (POMDP)

In MDP we also need to think about tasks with information gathering:

Now, what if we don't know where the +100 exit is. Instead, we need to look for a sign, and then act on that.

In the ?? case the optimal choice is to go south, just to gather information. The question becomes: How can we formulate the question the agent must solve in order for the detour to the south to be the optimal path?

One solution that doesn't work is to work out both of the possibilities and then take the average. It simply doesn't work.

A solution that works is to use information space - belief space. The trick is to assume that the transition in belief space, when reading the sign, is stochastic

The MDP trick in the new belief spce and create gradients from both of the possible belief states, but let that flow flow through the state where the sign is read. This means that we can in fact use value iteration (MDP style) in this new space to find a solution to compliated partial observable stochastic problem.

This style planning technique is very useful in artificial intelligence.

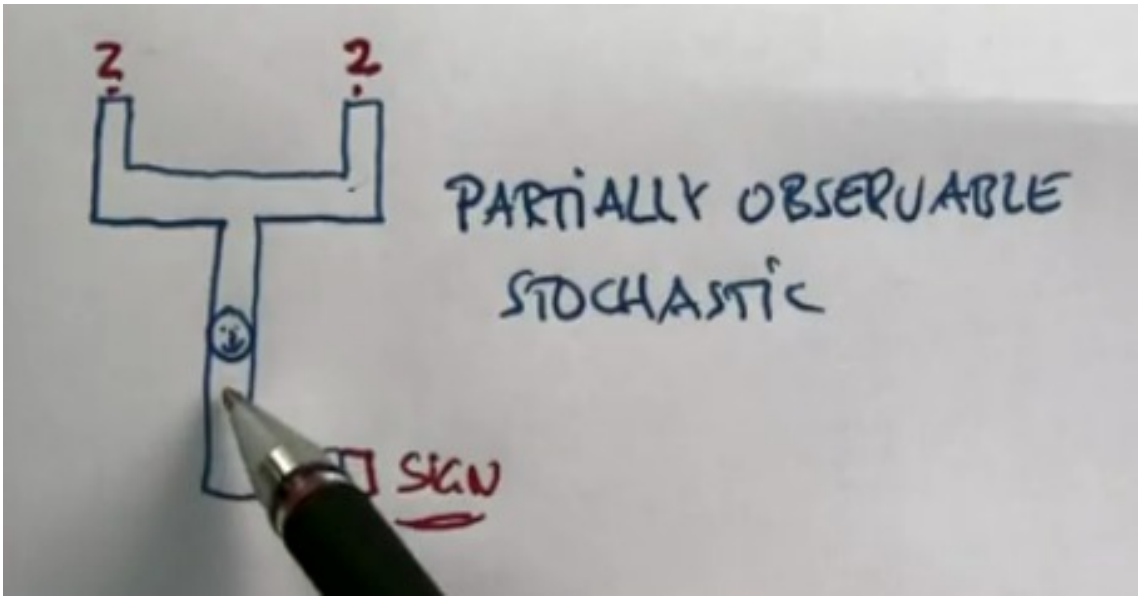


Fig. 5.10. In this "signed Y"problem the agent don't know where the target is, but can gain knowledge about that by reading what's written on a sign at the bottom right. This environment is stochastic but it's also partially observable.

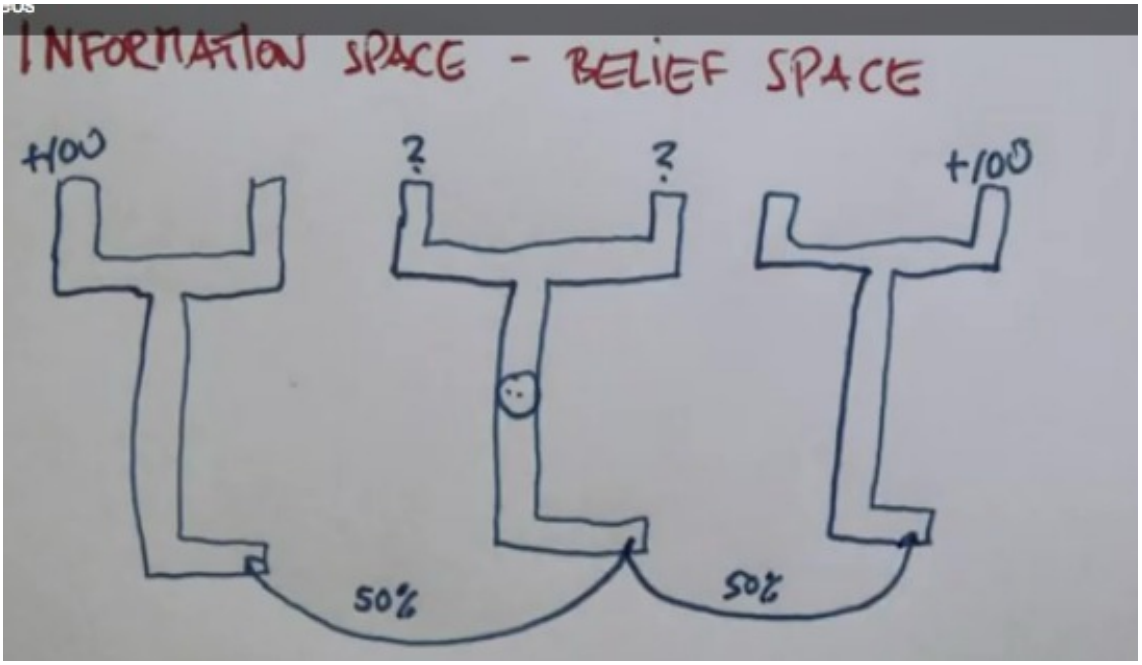


Fig. 5.11. The belief space of the signed Y problem

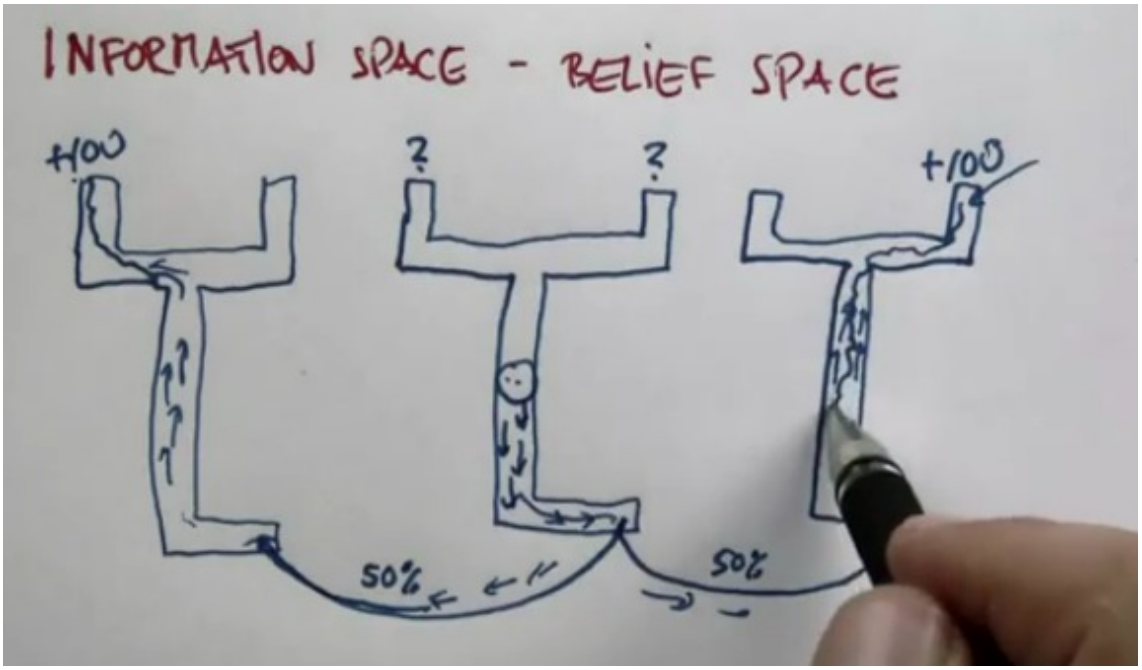


Fig. 5.12. The MDP trick of assigning a gradient to the information space according to the estimated utility of a position in the space, and then let the agent follow a simple gradient-descent behavior.

6. Reinforcement learning

We will learn how an agent can learn how to find an optimal policy even if he doesn't know anything about the rewards when he starts out. This is in contrast to the previous section where we assumed that there were known states that were good or bad to be in.

Backgammon was a problem Gary Tisarov at IBM worked on in the nineties. He tried to use supervised learning with experts classifying states. But this was tedious and he had limited success. Generalizing from this small number of state didn't work to well. The next experiment let one version of the program play against another. The winner got a positive reward and the loser a negative reward. He was able to arrive at a function with no input from human players, but was still able to perform at a level of the very best players in the world. This took about two hundred thousand games. This may sound like a lot but it really just covers about a trillionth of the statespace of backgammon.

Fig. 6.1. Andrew Ng's helicopter flying upside down :-)

Another example is Ng's helicopter, he used reinforcement learning to let the helicopter use just a few hours of experts and then make programs that got really good through reinforcement learning.

The three main forms of learning:

- Supervised. Datapoints with a classification telling us what is right and what is wrong.
- Unsupervised. Just points, and we try to find patterns (clusters, probability distributions).
- Reinforcement learning: Sequance of action and state transitions, and at some points some rewards associated with some states (just scalar numbers). What we try to learn here is the optimal policy: What's the right thing to do in any given state.

MDP review. Aan mdp is a set of states $s \in S$, a set of actions $a \in \text{Actions}(s)$, a start state s_0 and a transition function to give tell os how the world evolves when do somethin: $P(s'|s,a)$, that is the probability that we end up in state s' when we are in state s and apply action a. The transition function is denoted like this: $T(s,a,s')$. In addition we need a reward function: $R(s,a,s')$, or sometimes just $R(s'$.

To solve an mdp we find a policy , an optimal policy, that optimize the discounted total reward:

$$\sum_t \gamma^t R(s_t, \pi(s_t), s_{t+1})$$

The factor γ is just to make sure that things cost more if they are further out in time (to avoid long paths). (future rewards count less than rewards near in time).

The utility of any state is $U(s) = \text{argmax}_a \sum_{s'} P(s'|s,a)U(s')$. Look at all the possible actions, choose the best one .-)

Here is where reinforcement function comes in. What if we don't know R and don't know P? Then we can't solve the Markov decision problem, because we don't know what we need.

we can learn r and p by interacting with the world, or we can learn substitues so we don't have to compute with r and p.

We have several choices:

agent	know	learn	use
Utility-based agent	P	R and U	U
Q-learning agent		Q(s,a)	Q
Reflex-agent		$\pi(s)$	π

Q is a type of utility but rather being an utility over states, it is an utility over state/action pairs, and that tells us for any given state/action, what is the utility of the result, without using the utilities directly. We can then use Q directly.

The reflex agent is pure stimulus/response. No need to model the world.

6.0.9. Passive reinforcement learning

The agent has a fixed policy and executes that but learns about the reward (and possibly transition) while executing that policy. E.g. on a ship, the captain has a policy for moving around but it's your job to create the reward funciton.

The alternative is *active reinforcement learning*, and that is where we change the behavior. This is good because we can both explore, and to cash in early on our learning.

6.0.10. Temporal diference learning

We move between two states and learn the difference, then once we learn something (e.g. that a state has a +1 reward), we propagate that learning back into the states we have traversed to get there, so they get somewhat good if the target state was really good, back to the start state. The inner loop of the algorithm is:

if s' is new **then** $U[s'] \rightarrow r'$
if s is not null **then**
  increment $Ns[s]$
  $U[s] \rightarrow U[s] + \alpha(N_s[s])(r + \gamma U[s'] - U[s])$

New states get default reward (e.g. zero by default). The difference $U[s'] - U[s])$ is the difference in utility of states, the α is the *learning rate*. The propagation goes relatively slowly, since we only update when we get a state with a nonzero value, which will probably be in the next iteration.

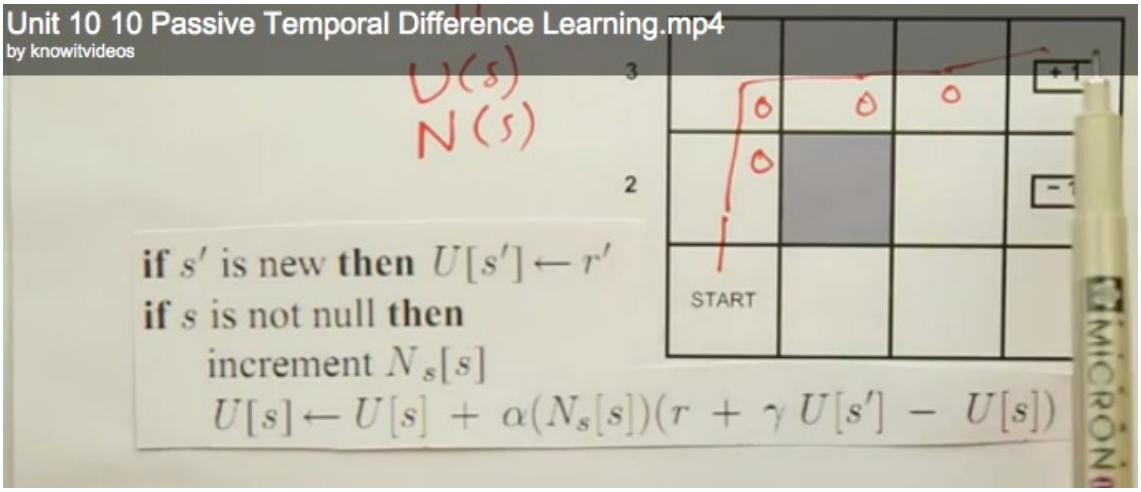


Fig. 6.2. Passive difference learning

Eventually this will propagate to the correct utilty for the policy.

This strategy takes a while to converge to real values.

There are multiple problems with this passive approach:

- It takes a long time to converge.
- We are limited byt he policy choices.
- There may be states we haven't visited and thus don't know anything about (a function of the policy).
- We may must get poor estimates due to few visits.

6.0.11. Active reinforcement learning

Greedy reinforcement learner works the same way as the *passive reinforcement learner* but after every few utility updates, we recompute the new optimal policy so we throw away the old p_1 and replace it with a new p_2 which is the result of solving the MDP based on our new estimates of the utility. We continue learning with the new policy. If the intial policy was flawed, we will tend to move away from it. However, we're not guaranteed to find the *optimal policy*. Since it is greedy it will never deviate from something it things is ok. Particular in stochastic environments

that is bad since randomness may cause us to do stupid things. To get out of this rut we need to explore suboptimal policies for a while just to make sure that we explore the utility space a bit more thoroghly before deciding on the policy to generate.

This means that we will find a tradeoff between exploration and execution of the currently know best policy. One way to do that is to sometimes do something randomly. This actually works, but it's slow to converge. We need to think a bit more about this whole *exploration v.s. exploitation* thing to be effective.

In greedy polcy we are keeping track both of the utility and the policy as well as the number of times we've visited each state. There are several reasons why things can go wrong:

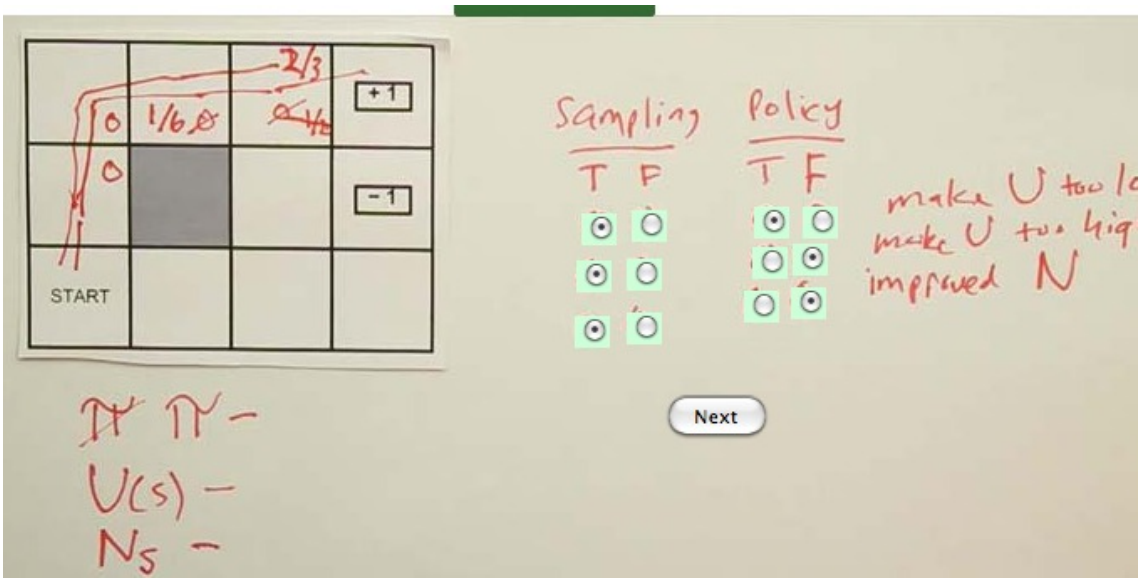


Fig. 6.3. Active error sources

- *Sampling errors:* The number of samples for a state is too low.
- *Utility errors:*We can get a bad utility since the policy is off. We're underestimating the utility.

What fig ?? suggests is a design for an agent that is more proactive in exploring the world when it is uncertain and will fall back to exploiting the policy it has when it is more certain of the world.

One way of encoding this is to let the utility of a state be some large value $U(s) = +R$ when $N_s < e$, or some threshold e. When we've visited the state e times we revert to the learned utility. When we encunter a new state we will explore it, and when we have a good estimate of its utility we will use that utility estimate instead.

Exploratory learning usually does much better than the greedy learning. It both converges faster and learns better.

Rmz: *What kind of tasks in the domain we are working can we formulate using policies, and hence make accesible to MDP-like learning algorithms?*

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

If we have all the probabilities we can apply the policy, but if we do't then we can't calculate, and hence not apply the policy since we don't know the probabilities.

6.0.12. Q-learning

In Q-learning we don't need the transition model. Instead we learn a direct mapping from states and actions to utilities.

$$\pi^* = \text{argmax}_{a \in A(s)} \sum_{s'} Q(s,a)$$

All we have to do is to take the maximum over all possible actions.

In figure **??** we have utilities for each of the possible actions associated with each of the possible states, nswe, hence four different values per state. All Q-utilities start being zero, and then we have an update formula that is very similar to the one we have already seen:

$$Q(sa) \rightarrow Q(s,a) + \alpha \left(R(s) + \gamma W(s',a') - Q(s,a) \right)$$

It has a learning rate α, and a discount factor γ

For really large statespaces the approaches described above will not be directly applicable. The space is just to large. The packman-states in figure ?? illustrates this. The two states are bad, but they are not similar. One thing that would be

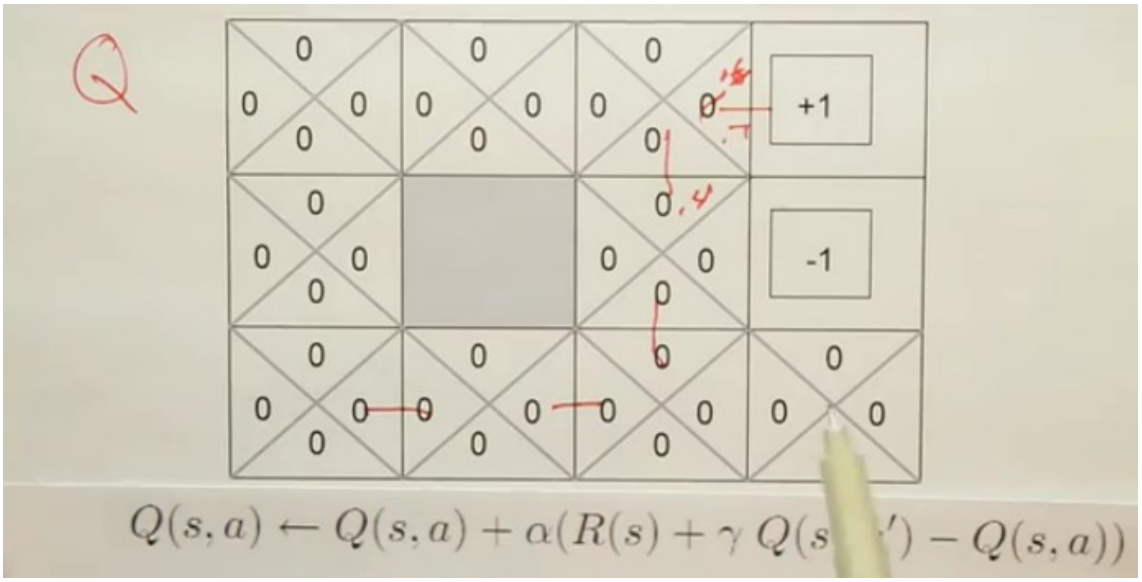


Fig. 6.4. A Q-learning table

nice would be to be able to learn from both these situations and in some sense treat them as the same thing.

Just as we did in supervised machine learning we an use a similar reprsentation by representing as state by a colletion of impportant features, such as $s = \{f_1, f_2, \ldots\}$. The features don't have to be the exact positions on the board, they can instead be tings like "distance to the nearest ghost" or the squared distanance, or the inverse squared distance, the number of ghost remaining and so on.

We can then let the Q-value be defined by:

$$Q(, a) = \sum_i w_i \cdot f_i$$

The task then is to learn values of these weights. How important are each feature. This is good since it means that similar states have the same values, but bad if similar states have different values.

The great thing is that we can make a small modification to the Q-learning algorithm. Instead of just updating $Q(s, a) \rightarrow \ldots$ we can update the weights $w_i \rightarrow \ldots$ as we update the Q-values. This is just the same type of thing we used when did supervised learning. It is as we bring our own supervision to reinforcement learning.

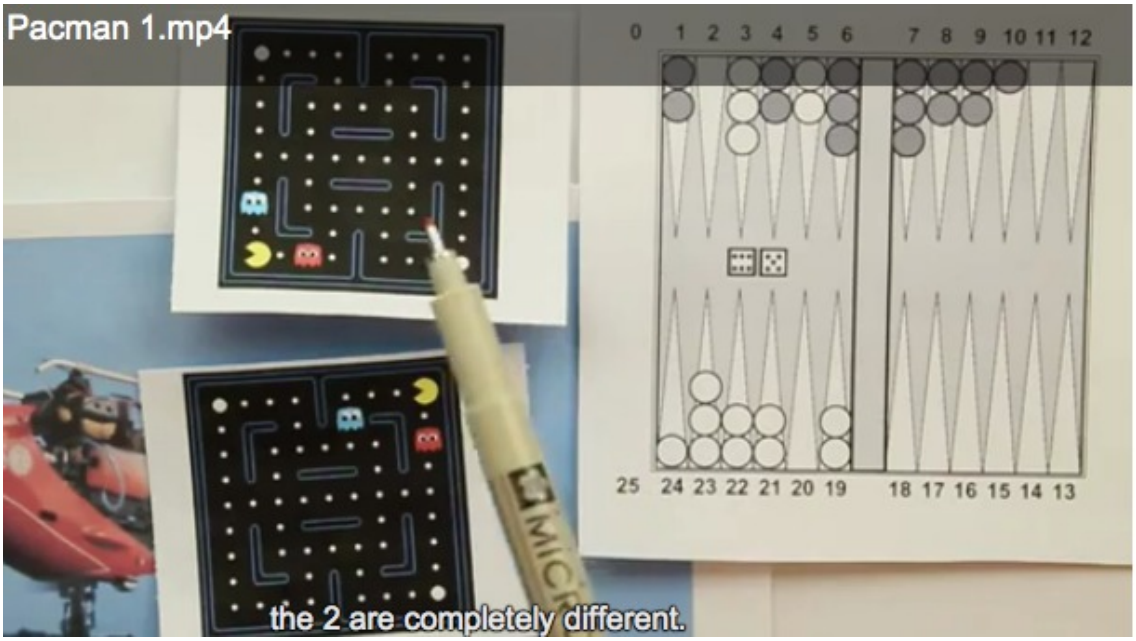


Fig. 6.5. Packman state

6.1. Summary

If we have an MDP we can calculate the optimal policy. If we don't know what the MDP is, we can estimate it and then solve it. Or we can update the Q-values.

Reinforcement learning is one of the most interesting parts of AI. NGs helicopters or the Backgammon player are instances of this. There is probably a lot more to learn.

7. Hidden Markov models (HMMs) and filters

Hidden Markov models and filters is used in just about every interesting robotic system Thrun builds. Not so strange perhaps since his "job talk" that got him into Stanford as a professor was about this topic :-) The techniques are applicable to finance, medicine, robiotics, weather prediction, time series prediction, speech, language techinques and many other.

A Hidden markov model (hmm) is used to analyze or predict time series. Time series with noise sensors this is the technique of choice. The basic model is a *Bayes network* where the next state only depends on the previous state:

$$S_1 \to S_2 \to \ldots \to S_N$$

Each state also emits a sequence of measurements

$$Z_1 \to Z_2 \to \ldots \to Z_N$$

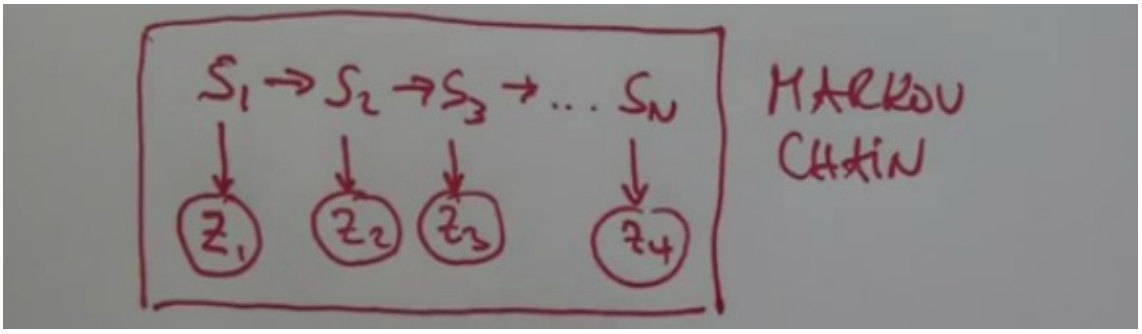


Fig. 7.1. A markov chain where the state changes and measurements are made.

and it is this type of meaurements that are used in hidden markov models and various probabilistic filters such as *Kalman filters*, particle filters and many others.

In a *markov chain* the state only depends on its immediate predecessor. What makes the model *hidden* is the fact that we don't observe the markov process itself, but rather the measurement series.

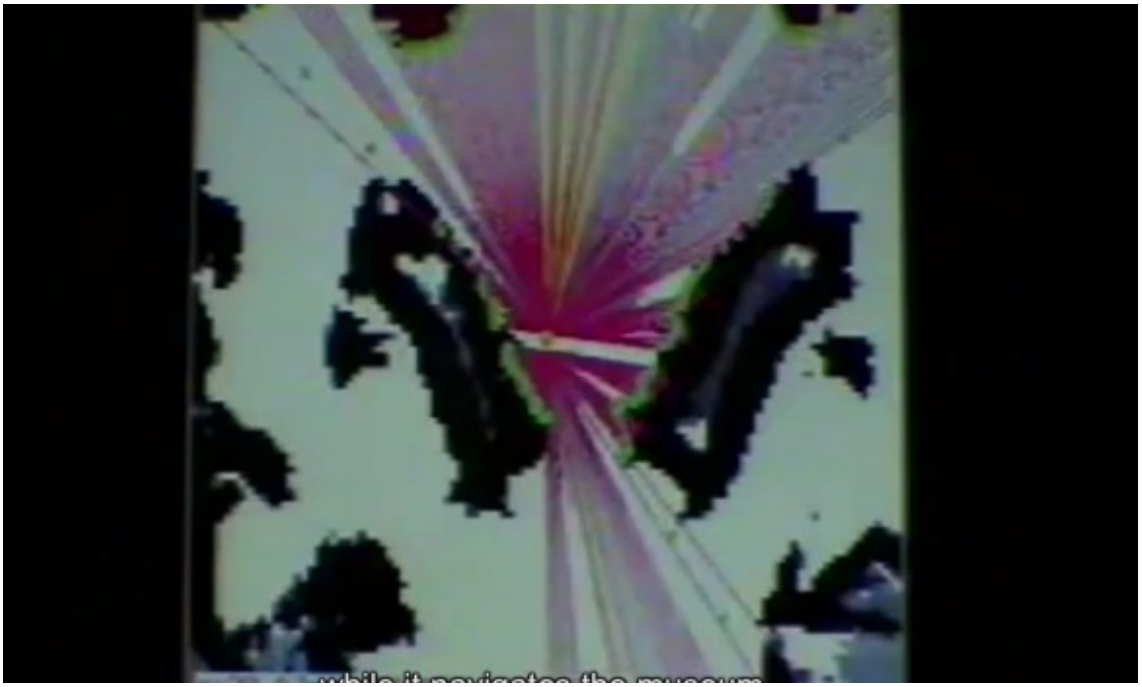


Fig. 7.2. Using a markov process to estimate the position of a robot roving around in a museum using ultrasonic rangefinders.

We see the tour guide robot from the national science museum. The robot needs to figure out where it is. This is hard because the robot doesn't have a sensor that tells it where it is. Instead it has a bunch of rangefinders that figures out how far away objects are. It also has maps of the environment and can use this to infer where it is. The problem of figuring out where the robot is the problem of *filtering*, and the underlying model is

The second problem is the tunnel-explorer. It basically has the same problem except that it doesn't have a map, so it uses a particle-filter applied to robotic mapping. The robot transcends into the tunnel it builds many possible paths ("particles" or hypothesis). When the robot gets out, it is able to select which one of the particles is the real one and is then capable of building a coherent map.

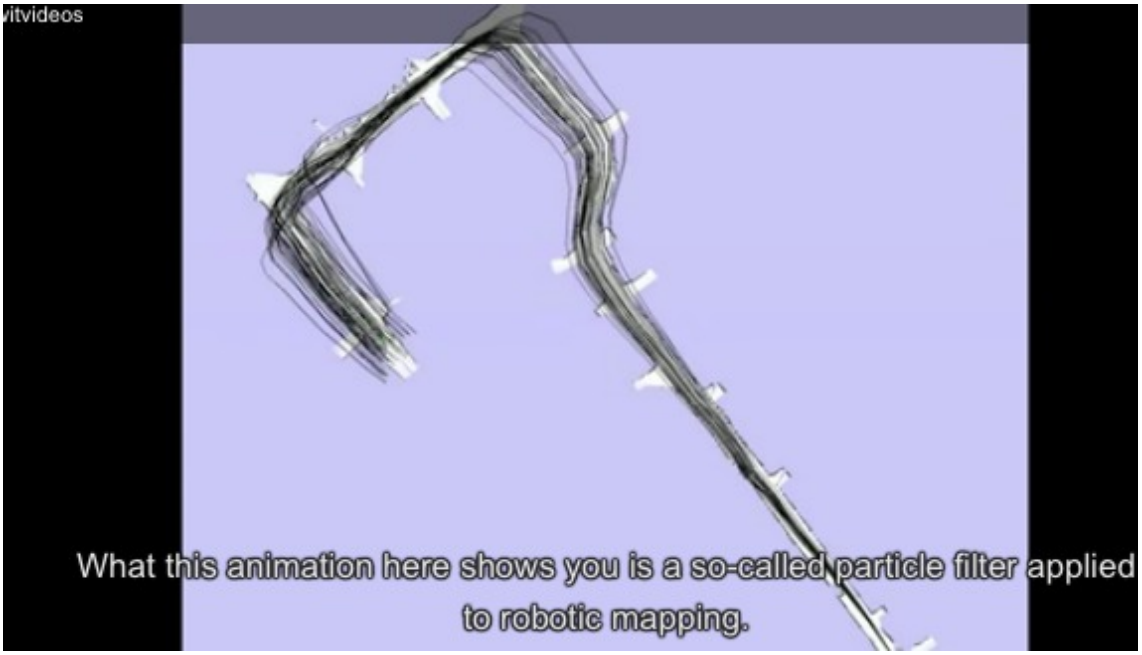


Fig. 7.3. A mapper based on particle filters

A final example is speech recognition (Simon Arnfield). We see a bunch of oscillation. Over time that is the speech signal. This signal is transformed back into letter, and it is not an easy task. Different speakers speak differently and there may be background noise. Today's best recognizers uses HMMs.

7.1. Markov chains

If we assume there are two states, rainy or sunny. The time evolution follows the distribution in figure ??.

All markov chains settle to a stationary distribution (or a limit cycle in some cases that will not be discussed ehre). The key to solving this is to assume that

$$\begin{aligned} P(A_t) &= P(A_{t+1}) \\ & P(A_t|A_{t-1})P(A_{t-1}) + P(A_t|B_{t+1}) \cdot P(B_{t-1}) \end{aligned}$$

The latter is just the total probability applied to this case to find the stationary

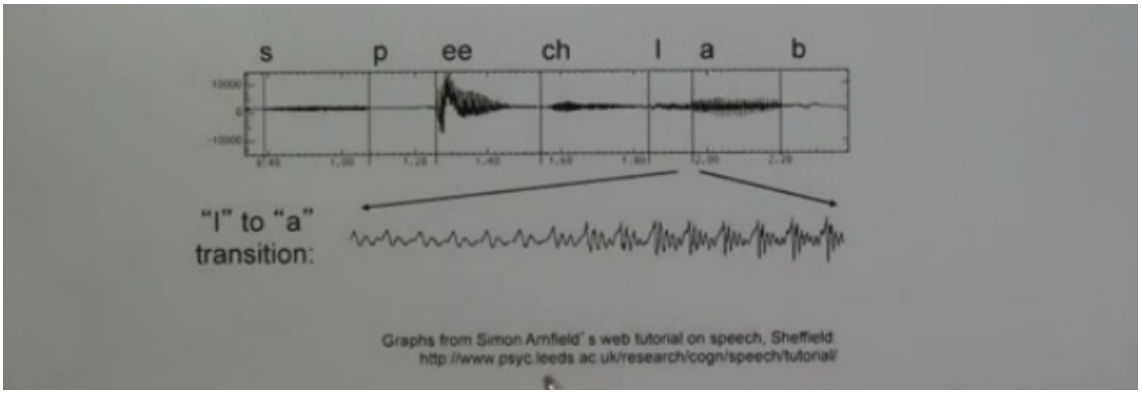


Fig. 7.4. A speech recognition system using a hidden markov model

distribution wil lhave at A at 2/3 and at 1/3. The trick is to use the first equality to solve the recursive equation.

The cool thing here is that we didn't even need the initial distribution to get the final result.

Rmz: *This simply must have something todo with eignenvectors, but what?*

Markov chains with this property are called *ergodic*, but that's a word that can safely be forgotten :-) This means that the markov chain *mixes*, which means that the distribution fades over time until disappars in the end. The speed at which it is lost is called the *mixing speed*.

If you see a sequence of states, these can be used to estimate the distribution of probabilities in the markov model. We can use *maximum likelyhood* estimation to figure out these probabilities, or we can use *Laplace smoothing*.

If we use a maximum likelyhood estimation we can do a calculation like the one in figure ??.

One of the oddities is that there may be overfitting. One thing we can do is to use laplacian smoothing.

When calculating laplacian probabilities, the method is this:

- First set up the maximum likelyhood probability as a fraction.
- Then add the correctives, one pseudo observation (for k = 1) to the numerator, and a class count corrective (multiplied by k) to the denominator to form the laplacian smoothed probability.

Laplacian smoothing is applicable to many types

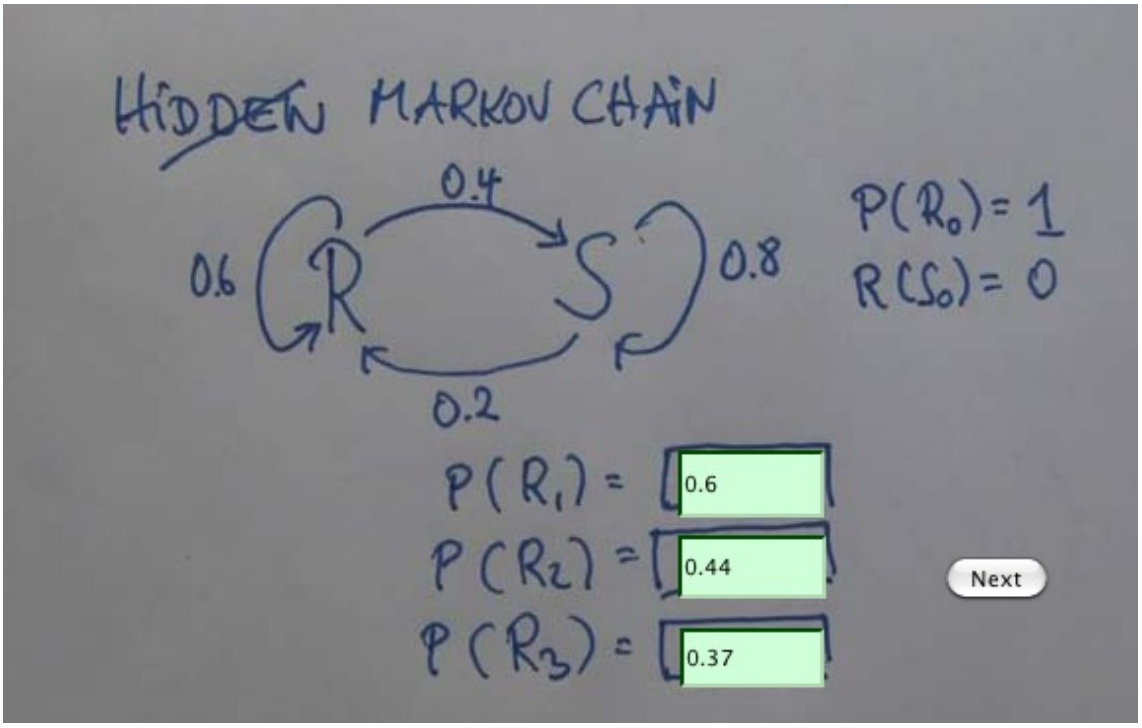


Fig. 7.5. A markov process for predicting the weather. If it rains today, it's likely to rain tomorrow (.6), and if it's sunny today it's likely sunny tomorrow (.8)

7.1.1. Hidden markov models

Let's consider the rainy day model again. We can't observe the weather, but we can observe your mood. The rain can make you happy or grumpy, and sunny weather can make you happy or grumpy (with some probability). We can answer questions like this based on a combination of the bayes rule and the markov transition rules.

We can use these techniques both for prediction and for state estimation (fancy word for estimating an internal state given measurements).

HMM and robot localization

The robot knows where north is, can sense if there is a wall in the adjancent cell. Initially the robot has no clue where it is, so it has to solve a *global localization problem*. After getting more measurements, the posterior probabilities that are

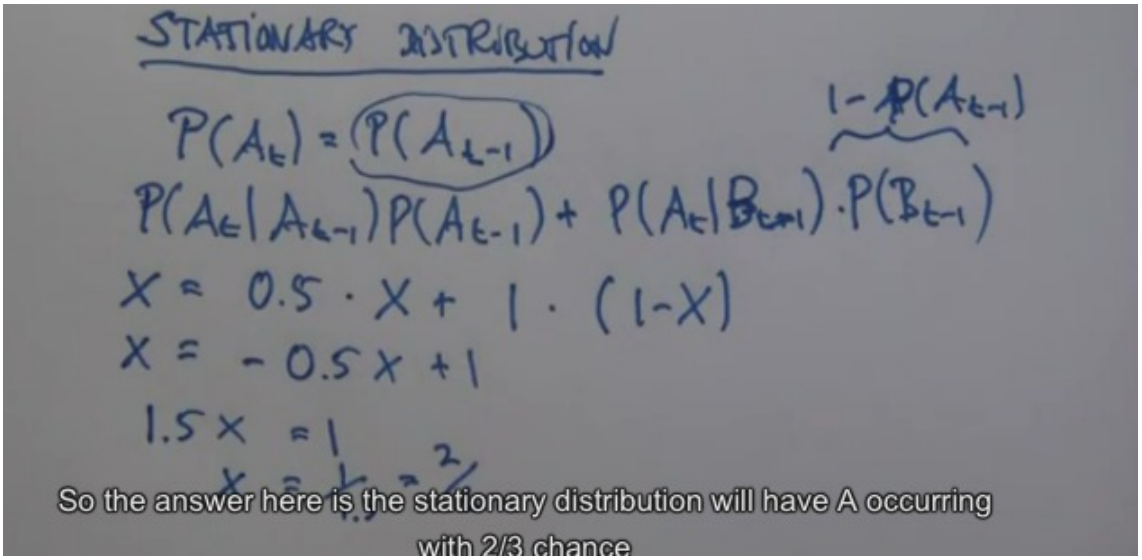


Fig. 7.6. The stationary distribution of a Markov process is the probabiities as they will be after a large number of observations. The trick is to assume that the probability in the next step will be equal the probability in the current step, use this to get a system of equations that can then be solved.

consistent with the measurements will increase, and the probabilities of states inconsistent with the measurement will decrease. Sensor errors can happen and that is factored into the model. After having passed an *distinguishing state* the robot has pretty much narrowed down its state estimation.

7.1.2. HMM Equations.

The HMMs are specified by equations for the hidden states and observations:

$$\begin{array}{ccccccc} x_1 & \rightarrow & x_2 & \rightarrow & x_4 & \rightarrow & \dots \\ \downarrow & & \downarrow & & \downarrow & & \downarrow \\ z_1 & & z_2 & & z_4 & & \dots \end{array}$$

This is in fact a Bayes network

Using the concept of *d-separation* we can see that if x_2 and z_2 is the *present*, then the entire past, the future, and the present measurement are all conditionally idependent give x_2.

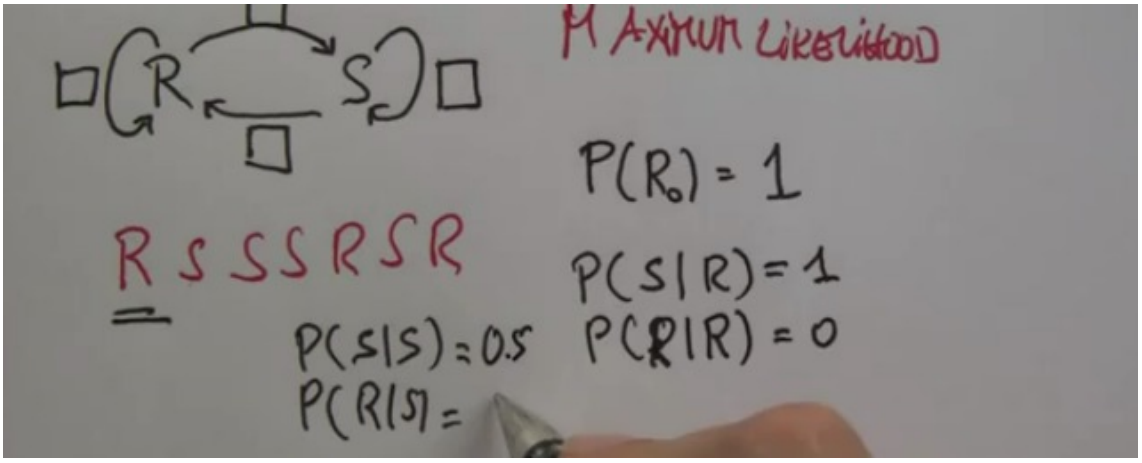


Fig. 7.7. Estimating transition probabilities using a maximum likelyhood classifier

This structure lets us do inference efficiently. Assume that we have a state x_1 and a measurement z_1 and we wish to find the probability of an internal state variable given a specific masurement: $P(x_1|z_1)$. We can set up an equation for this using Bayes rule:

$$P(x_1|z_1) = \frac{P(x_1|z_1)P(X_1)}{P(z_1)}$$

Since the *normalizer* $P(z_1)$ doesn't depend on the target variable x_1 it's common to use the proportionality sign

$$P(x_1|z_1) \propto P(x_1|z_1)P(X_1)$$

The product on the right of the proportionality sign is the *basic measurement update* of hidden markov models. The thing to remember is to normalize.

The other thing to remember is the *prediction equation* even though it doesn't usually have anything to do with predictions, but it comes from the fact that we wish to predict the distribution of x_2 given that we know the distribution of x_1:

$$P(x_2) = \sum_{x_i} P(x_1) \cdot P(x_2|x_1)$$

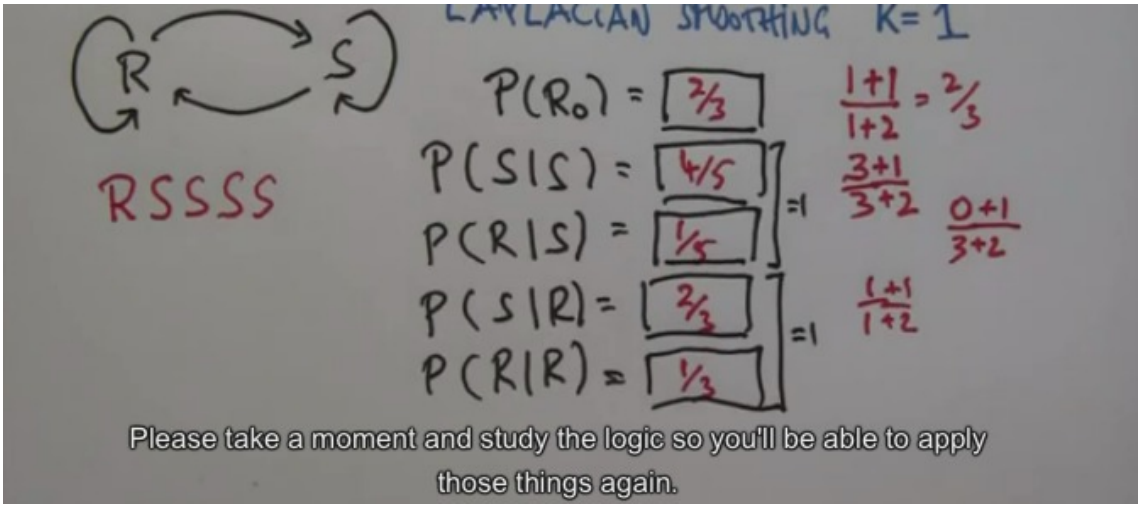


Fig. 7.8. Estimating transition probabilities using a Laplace smoothed probability

This is just the total probability of all the ways x_2 can have been produced, and that gives us the posterior probability of x_2.

Given the initial state distribution the measurement updates and the prediction equation, we know all there is to know about HMMs. :-)

7.1.3. localization example

Initially the robot doesn't know where it is. The location is represented as a histogram with uniform low values. The robot now senses the next door. Suddenly the probabilities of being near a door increases. The red graph is the probability of sensing a door, the probability is higher near a door. The green line below is the application: We multiply the prior with the measurement probability to obtain the posterior, and voila we have an updated probability of where the robot is. It's that simple.

The robot takes an action to the right. the probabilities are shifted a bit to the right (the convolution part) and smooth them out a little bit account for the control noise for the robot's actuators. The robot now senses a door again, and we use the current slightly bumpy probability measurement as the prior, apply the measurement update and get a much sharper probability of being in front of a door, and in fact of being in front of the right door.

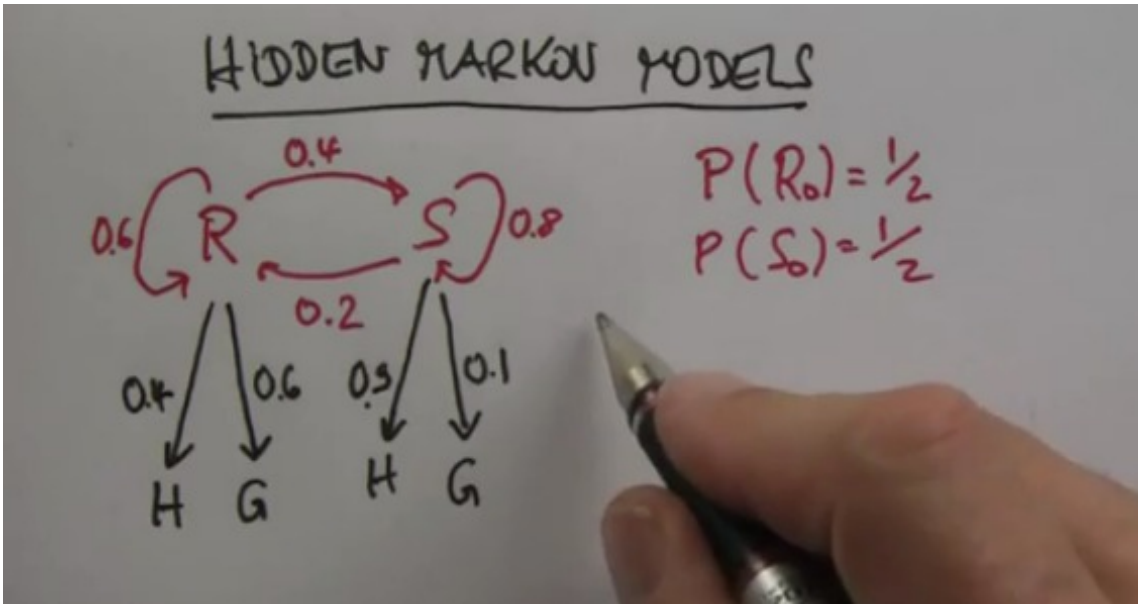


Fig. 7.9. The rainy / sunny / happy / grumpy - hidden markov model

This is really easy to implement this. Measurement is multiplications, and motions becomes essentially convolutions (shift with added noise).

7.1.4. Particle filter algorithm

Robot equipped with range sensor, task is to figure out where it is. The robot knows along the black line, but doesn't know where it is. The trick is how to represent the belief state. This isn't like the discrete example where we had sun or rain or a histogram approach where we cut the space into small bins. In particle systems the space is represented by collection of points or particles. Each of those small dots is a hypothesis of where the robot may be. It is in fact a location and a speed vector attached to that location. The set of all these vectors is the belief space. There are many many guesses, and the density of these guesses represents the posterior probability. Within a very short time the range sensors even if they are very noisy will converge into a very small set of possible locations.

Each particle is a state. The better the measurements fits with the predictions of the particle, the more likely it is that the particle survives. In fact, the survival rate is in proportion to the measurement probability. The measurement probability

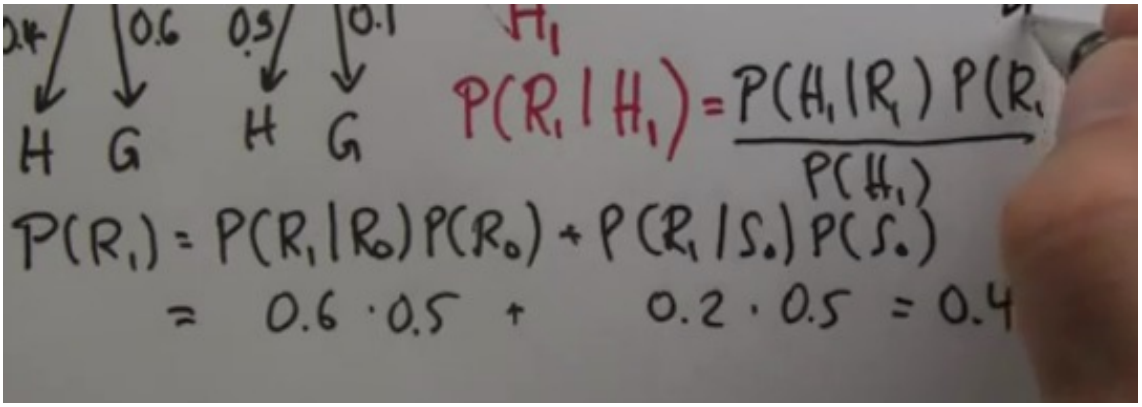


Fig. 7.10. Calculating probabilities in the rainy/sunny etc HMM.

is nothing but the concistency of the sonar range measurements with the location of the robot.

This algorithm is beautiful, and it can be imlemented in ten lines of code.

Initially particles are uniformly distributed. When a measurement is given (e.g. a door seen), the measurement probability is used to increase the particle importance weight. The robot moves, and it now does *resampling*. In essence this is to ensure that particles die out in proportion to their weight, so there will now be more particles where the weights where high, and fewer elsewhere. If the robot is moving, add some movement to the density field, shifting weights in proportion to the movement. Particles can be picked with replacements, single particles can be picked more than once. This is the *forward prediction step*. We now copy the particles verbatim, but attach new weights in proportion to the measurement probabilities. When we now resample, we will get a new density field for the particles.

Particle filters work in continous spaces, and what is often underappriciated, they use computational resources in proportion to how likely something is.

7.1.5. The particle filter algorithm

```
s: particle set with weights
v: control
z: measurement
{w} importance weight
```

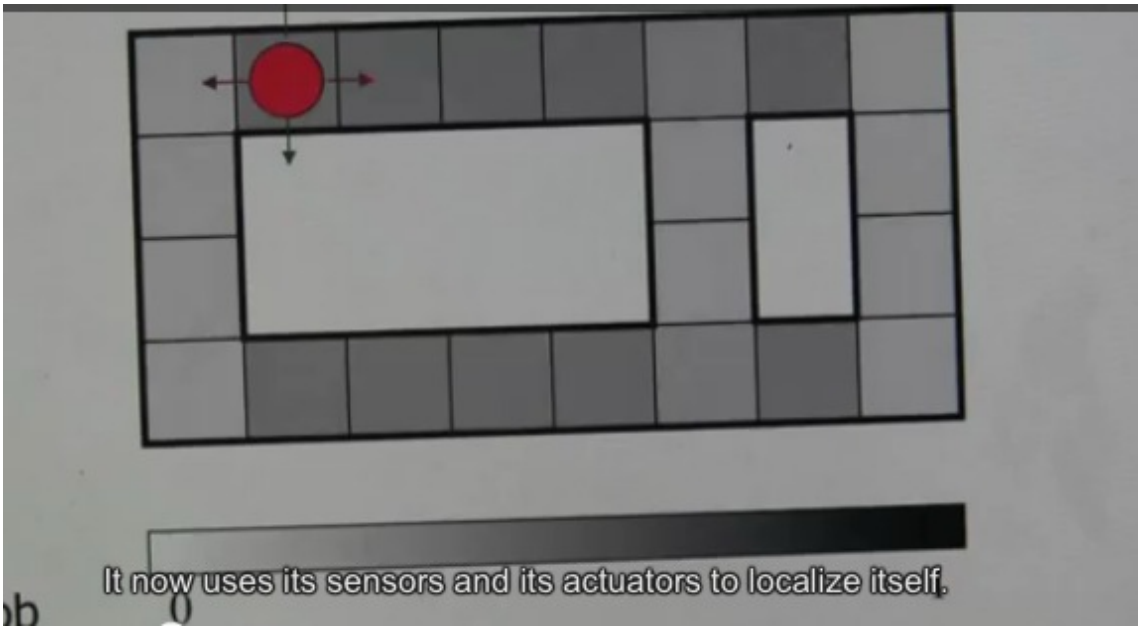


Fig. 7.11. A robot that can sense if there is a wall in the adjacent cells and can move about. It has a map, and from this it can determine where it is. Grayness indicates estimated probability of location.

```
particleFilter(s,v,z)
  S'  = Emptyset
  eta = 0
  // Resample
  // particles with un-normalized
  // weights
  for i = 1 ... n
     // Pick a particle at random, but in
     // accordance to the importance weight
     sample j ~ {w) with replacement
     x' ~  p(x' | v, s_j)
     w' =  p( z | x' ) // Use measurement probability
     S' = S' U {<x', w'>}
     eta += w'
  end
  // Weights now need to be normalized
  for i=1 ... n
     w_i = w_i / eta
```

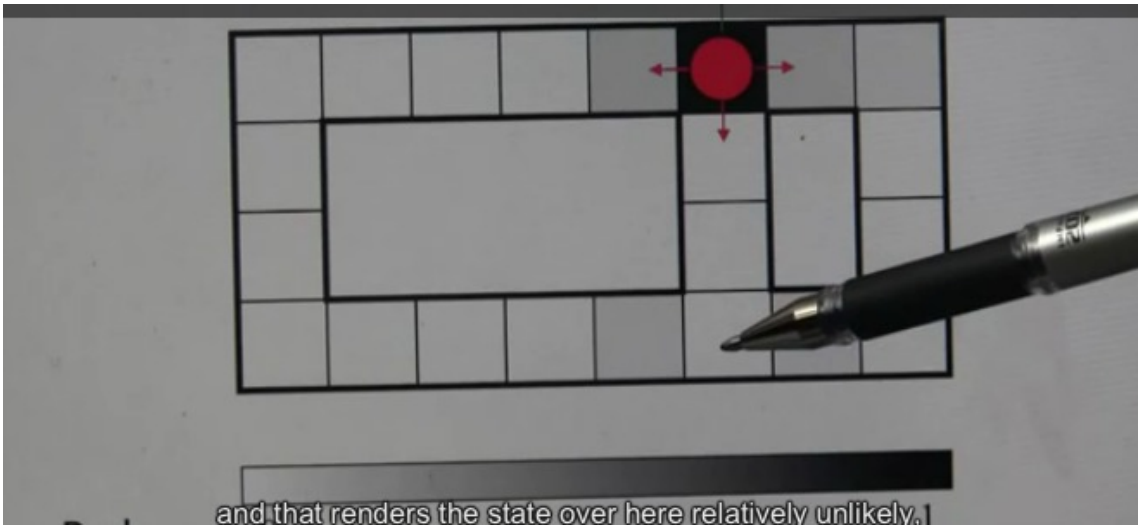


Fig. 7.12. After the robot has moved back and forth a bit, and after having seen a distinguishing state

```
end
```

7.1.6. Pro and cons

They are really simple to implement and work really well.

1. They don't work very well for high dimensional spaces. *Rao-Blackwellized* particle filters goes some steps in this direction anyhow :-)
2. They don't work well for degenerate conditions (one particle, two particles)
3. Don't work well if there is no noise in the measurement models or control models. You need to remix something.

Thrun's self-driving cars used particle filters for mapping and for a number of other tasks. The reason is that they are easy to implement, are computationally efficient and and they can deal with highly non-monotonic probability distributions with many peaks, and that's important because many other filters can't do that. Particle filters are often the method of choice for problems where the posterior is complex.

Particle filters is the most used algorithm in robotics. But are useful for any problem that involves time series, uncertainty and measurement.

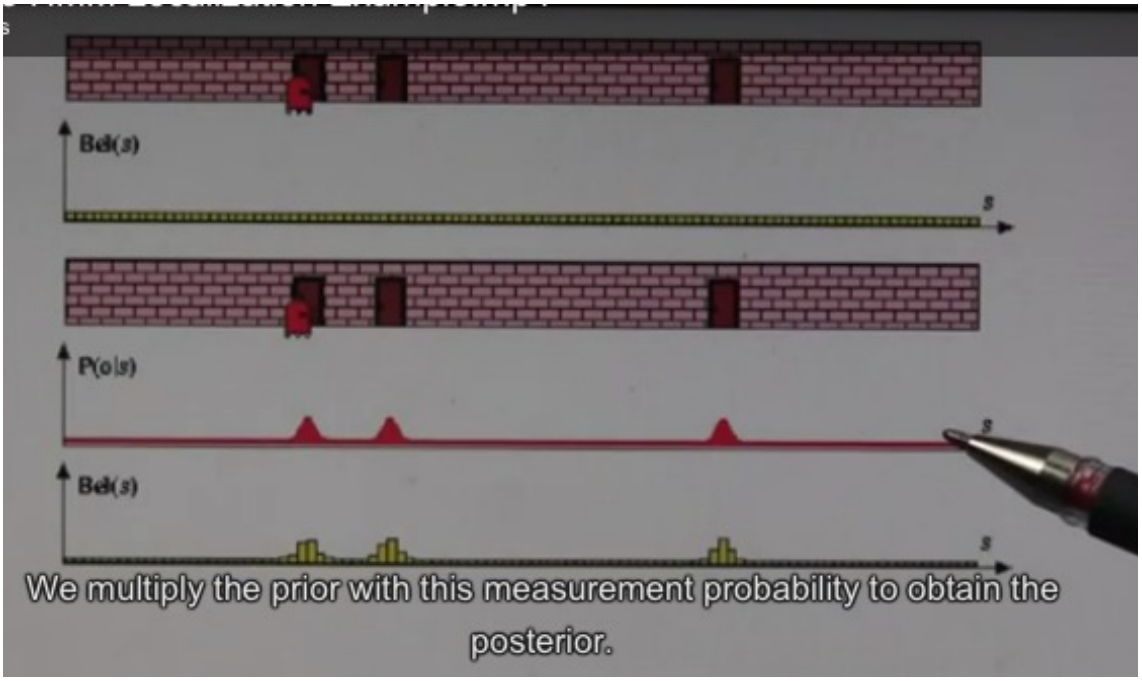


Fig. 7.13. Using a hidden markov model for localization: A partile filter.

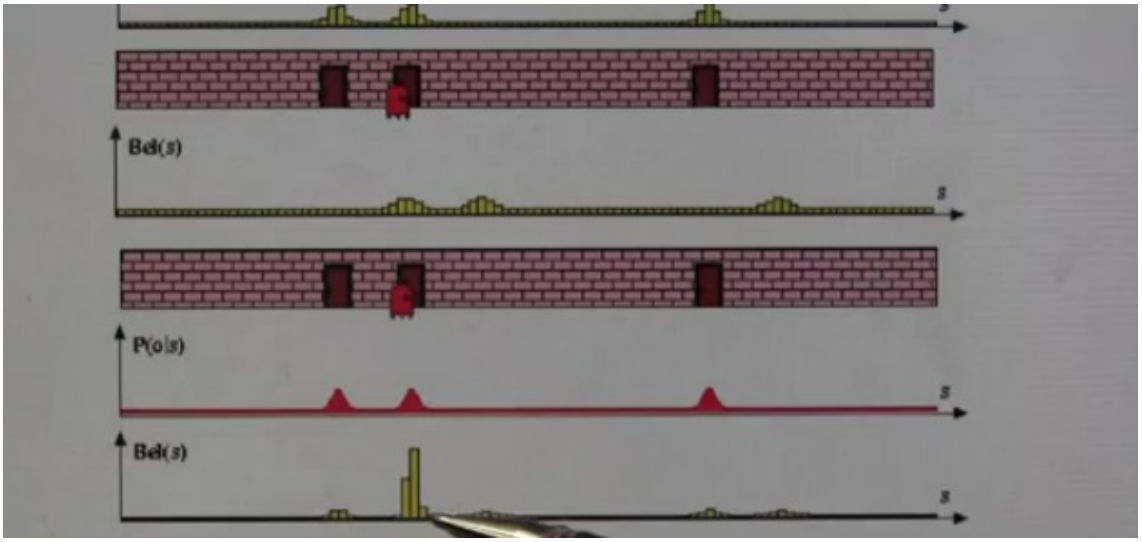


Fig. 7.14. Even more particle filter

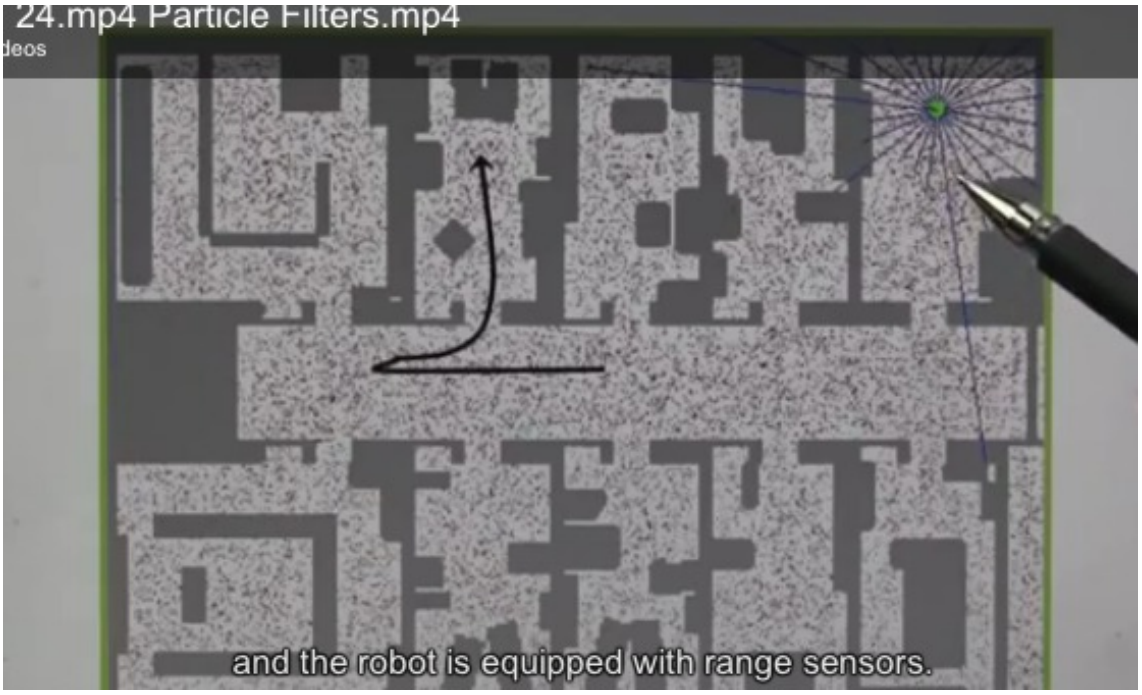


Fig. 7.15. Particle filter, initial state: A lot of particles randomly distributed both in space and direction.

Fig. 7.16. The particle filter from figure ?? after a few iterations. It's no longer very evenly distributed field of particles definite lumps representing beliefs with higher probabilities.

8. Games

Games are fun, they define a well-defined subset of the world that we an understand. They form a small scale problem of the problem of dealing with adversaries.

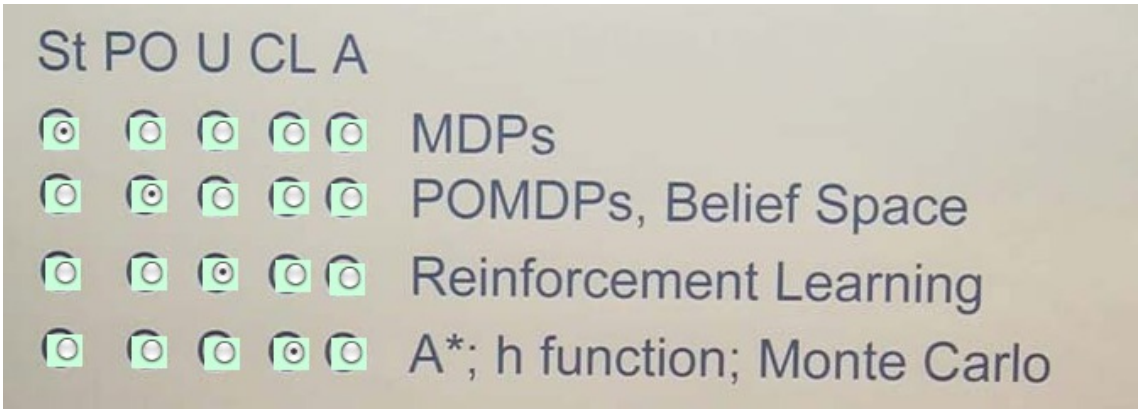


Fig. 8.1. A collection of tools for solving problems under varying degrees of noise, stochasticisty, adversity and observability, and situations where they are more or less applicable

In figure ?? we see how well various techniques are applicable to problems dealing with stochastic environments, partially observable environments, unknown environments, computational limitations and adversaries.

Wittgeinstein said that there is no set of necessary and sufficient conditions to define what games are, rather games have a set of features and some games share some of them, others other. At's an overlapping set, not a simple criterion.

Single player deterministic games are solved by searching through the statespace.

We have a state, a player (og players), and some function that gives us the possible actors for a player in a state a function that says if the state is a terminal, also we get a terminal utility function that gives us some utility for state for a player.

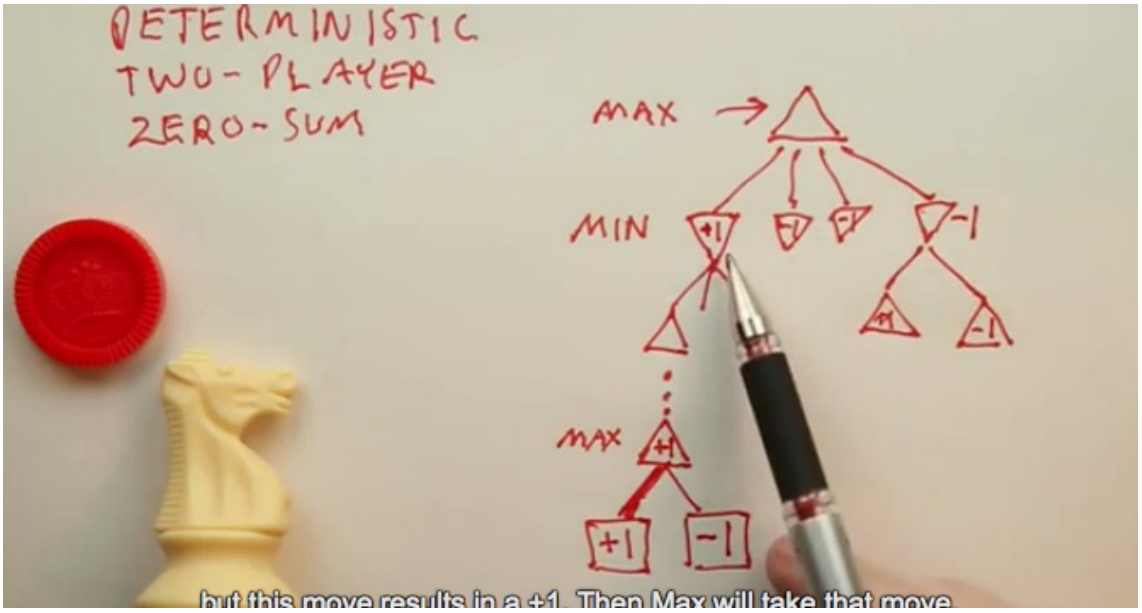


Fig. 8.2. The minimax algorithm will find optimal game trees, after some possibly very long time.

Let's consider chess or checkers. They are both determinstic, two-player zero sum games. Zero-sum means that the sum of the utilities for the players is zero.

We use a similar approach to solve this . We make a game tree, and alternatingly the players try to maximize or minimize the total utility for the player at the top of the choice tree (which is the same as maximizing utility for him/herself). The searh tree keeps going until some terminal states. If max is rational then max will then choose the terminal node with maximum utility.

The important thing is that we have taken the utility function that is only determined for the terminal states and the definitions of the available action and we have used this to determine the utility of every state in the tree including the initial state.

The time complexity for a game tree with depth m and branchout b is $O(b^m)$. The storage requirement is $b \cdot m$.

For chess with b=30 and m = 40, it will take about the lifetime of the universe to calculate all the possibilities. Transforming the tree into a graph, reducing the branching factor and the depth of the tree will all reduce the complexity.

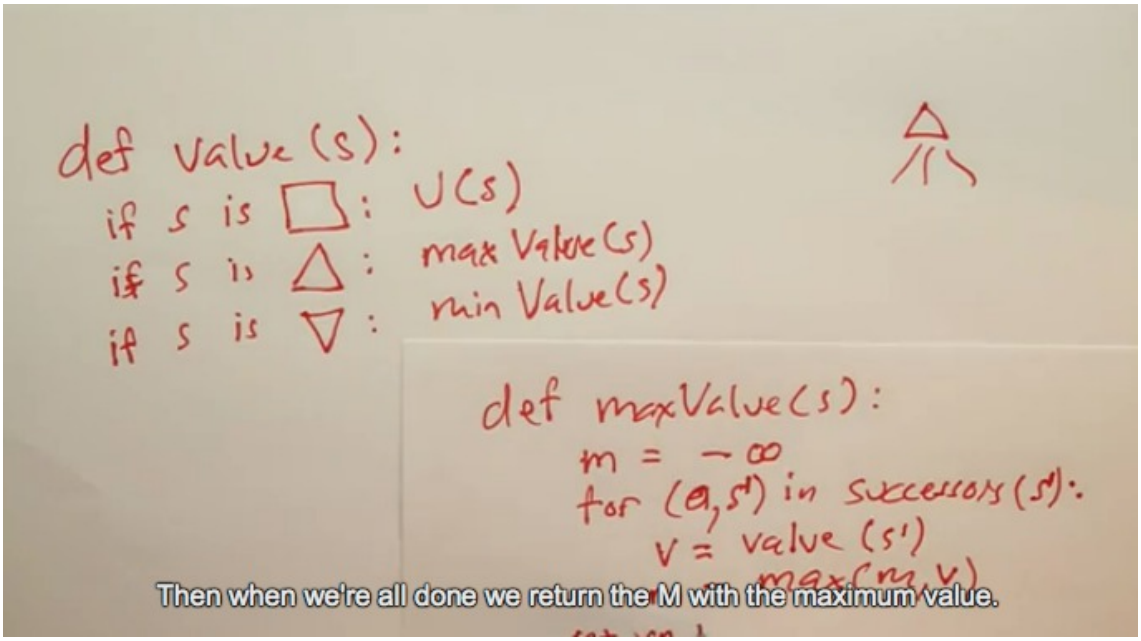


Fig. 8.3. The minimax algorthm (outlined)

Alpha beta pruning puts inequalities in the result nodes for the subtrees (e.g. "≤ 2", and that lets us prune away whole subtrees without having to evaluate them.

One way of reducing the tree size is to use an *evaluation function* that gives us an estimate of the utility of the mode. The function should return a higher value for stronger positions and lower values for weaker positions. We can find it using experience with the game. In chess it's traditional to say that a pawn is worth point, a knight three points a rook three points a bishop five points and the queen nine, and you can add up those points to get an evaluation function using a weighted sum for the pieces. Positive weights for your pieces and negative weights for the oppoents pieces. We've seen this idea in machine learning. We can make as many features as we like, e.g. "it's good to occupy the center" etc. We can then use machine learning to figure out an evaluation function. We then apply the evaluation function to each state at the cutoff point, and then back those estimates up as they were terminal values.

In alpha-beta pruning, α is the currently best know value for max, and β is the currently best know value for min. When we start, the best values known are plus and minus infinity.

Alpha-beta gets us from $O(b^m)$ to about $O(b^{m/2})$ if we do a good job, expanding

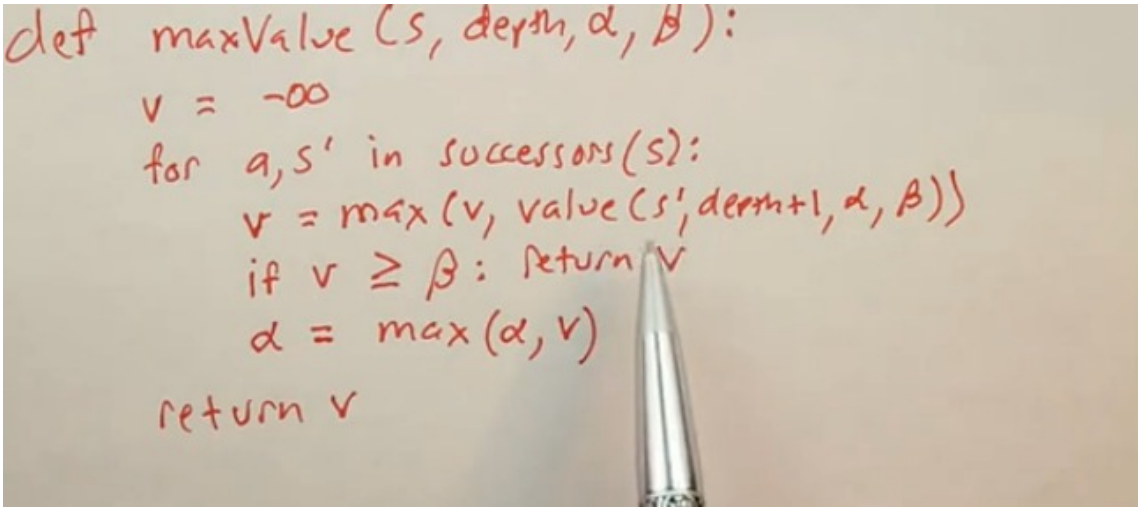


Fig. 8.4. The alpha beta pruning optimization can reduce the size of the min/max trees by a factor of two giving a very significant gain in efficiency.

the best nodes first getting to the cutoff points soon. This is perfect in the sense that the result is identical since we just don't do work we don't have to do.

Converting a tree to a graph can be efficient. In chess the use of memorized opening positions in *opening books* is an instance of this method. Similarly for closing books. In the mid-game we can do is to use the *killer move heuristic*: if there is one really good move in one part of a search tree, then try that move in sister branches for that tree.

Reducing the cutoff height. That is imperfect, and it can get us into trouble.

Chance is also an element of games. Dice or other sources of randomness makes it beneficial to figure out how to handle randomness :-) We can deal with stochastic functions by letting our value function include a chance element, then we use the expected value of the dice. The sequence of graph expansion will be, roll dice, let max choose, roll the dice (for all the possible values of the dice), use the expected value and then let min chose. Repeat.

Cutoff and evaluation functions are now the main problems less.

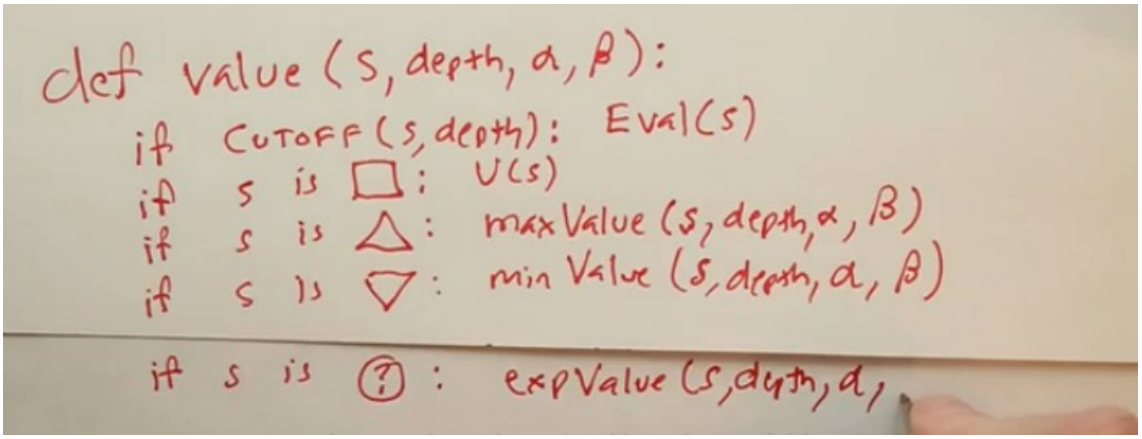


Fig. 8.5. When evaluating random moves in games, it's the expectation value of the move that counts.

9. Game theory

Game theory and AI has grown up together, taken different paths, but ar now merging back. Game theory focus on turn-based games where the opponent is adversarial. There are two problems: Agent design (find the optimal policy. The second is mechanism design: GIven the utility, how can we design a mechanism so that utility will be maximized in some way if the agents act rationally.

We wish to find the optimal policy when the optimal policy depends on the opponents policy. We'll start by looking at the prisoner's dilemma.

> Alice and Bob are both criminals taken at the same time. Both are are given an offer: If you testify against your cohort, you'll get a better deal. If both defect, they will both get a worse deal than if they didn't defect. If only one defects, the one that defected will get a better deal than without defection and a better deal than the other guy.

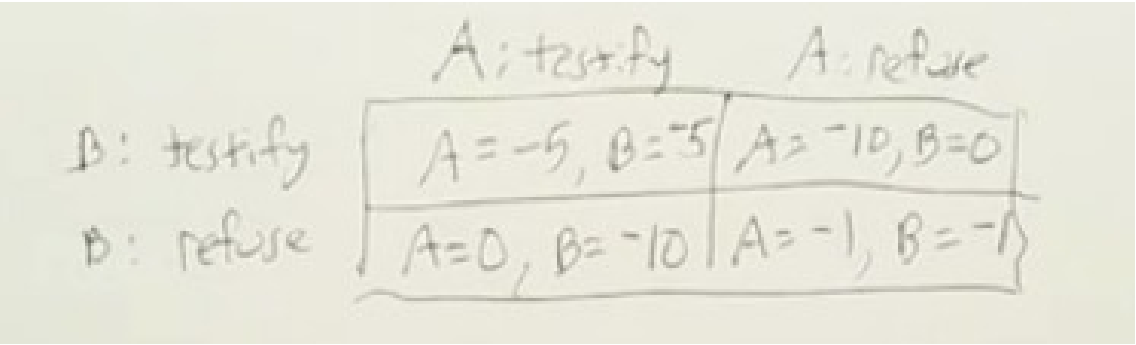


Fig. 9.1. Prisoner's dilemma

Alice an Bob understands what is going on, they are both rational. This isn't a zero-sum game.

A *dominant strategy* is one for which a player does better than any other strategy no matter what the other player does. So the question is, is there a dominant strategy in this game? And the answer is, both should testify (defect).

A *Pareto optimal* outome is an outcome such that there is no other outcome that all other players would prefer. In this case there is such an outcome, and that is that both of the players refuse to cooperate. In other words, the dominant strategy will never lead to the optimal outcome. How come? :-)

An *equilibrium* is such that no player can do better by switching to another strategy provided that all the other players stay the same. There is a famous proof (by John Nash) proving that every game has at least one equilibrium point. The question here is which, if any is the equilibrium for this game? And there is, its testify/testify.

If there is a dominant stratego or a pareto optimal solution to a game, it's easy to figure out an optimal strategy. The game *two finger morra* does not lend itself to that kind of easy solution. It's a betting game. Two playes "even" and "odd". The players show one or two fingers, and if the total number of fingers is even player wins that number of dollar from the odd player, if the number of fingers is odd then the odd player wins from the even player.

There is no single move (*pure strategy*) that is best, but there is a *mixed strategy* that is, and this uses the fact that there is a probability distribution over the moves.

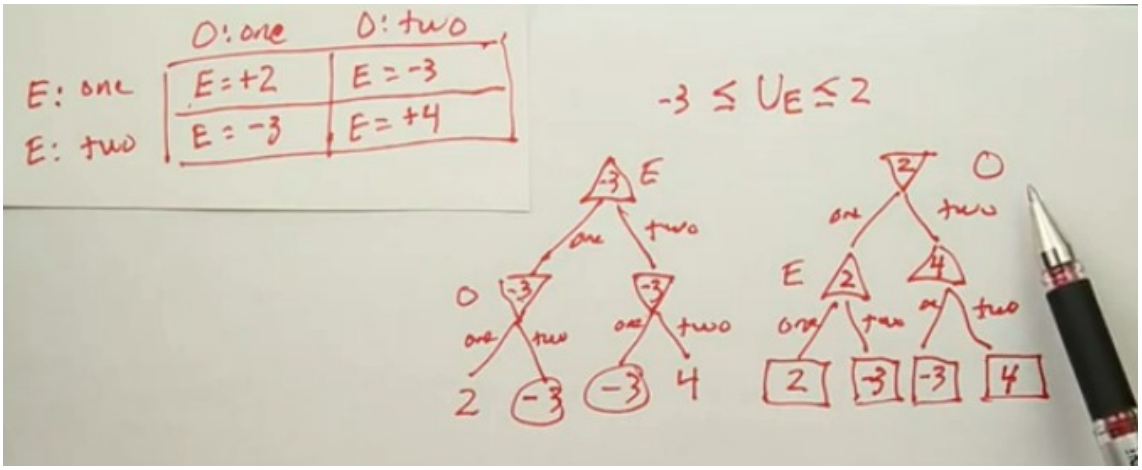


Fig. 9.2. The game "Two fingered morra"

A mixed strategy is to announce that the moves will be chosen with a probability distribution. This schema will give a parameterized outcome.

P should choose a value of p so that the two outcomes are equal!. In the example of ?? this would indicate p = q = 7/12.

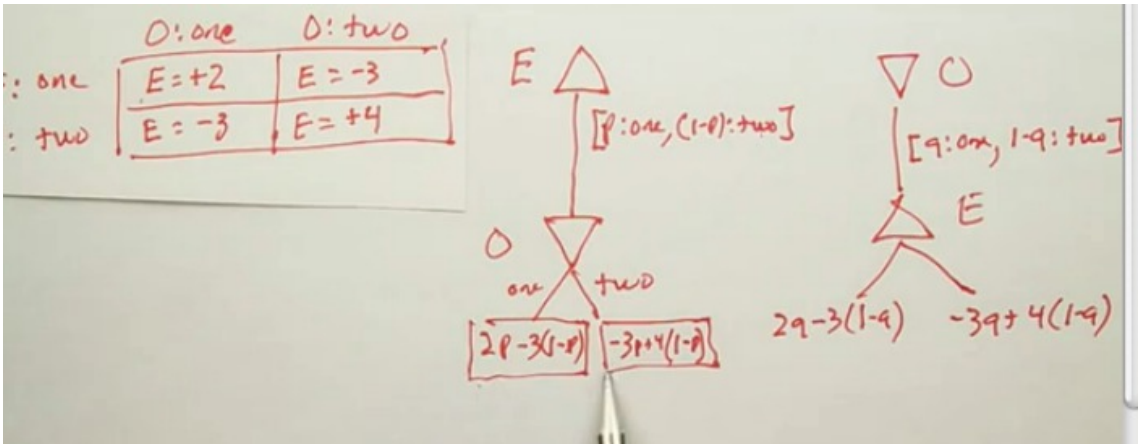


Fig. 9.3. The game "Mixed strategy Morra"

If we feed this back into the strategy of the game. The trick then is to show that the utility must between a couple of limits. Those limits are both 1/12, and that makes the game solved.

Mixed strategies gives us some curious strategies. Revealing our optimal strategy to our strategy is ok to reveal to our opponent, but the actual choices (choose a or b in any particular case), that is bad since our opponents will get an advantage over us.

There are games that you can do better in if your opponent don't believe you are rational.

When checking a strategy, always check if there are dominant strategies.

The geometric view in figure ?? indicatest that the two player's strategy end up being the intersection of the two possible set of strategies.

In ?? we use a representation called *the squential game format* that is particularly good at keeping track of the belief states for what the agents know and don't know. Each agent don't know in which node in the tre it is (the uncertainty denoted by the dotted line). The game tree can be solved using an approach that is not quite the max/min approach. One way to solve it is to convert this game in the *extensive form* into the matrix representation we hava already seen that is the *normal form* (see fig ??).

In general this approach lead to exponentially large tables, but for this game it is a tractable approach. There are two equilibria (denoted in boldface). For real poker

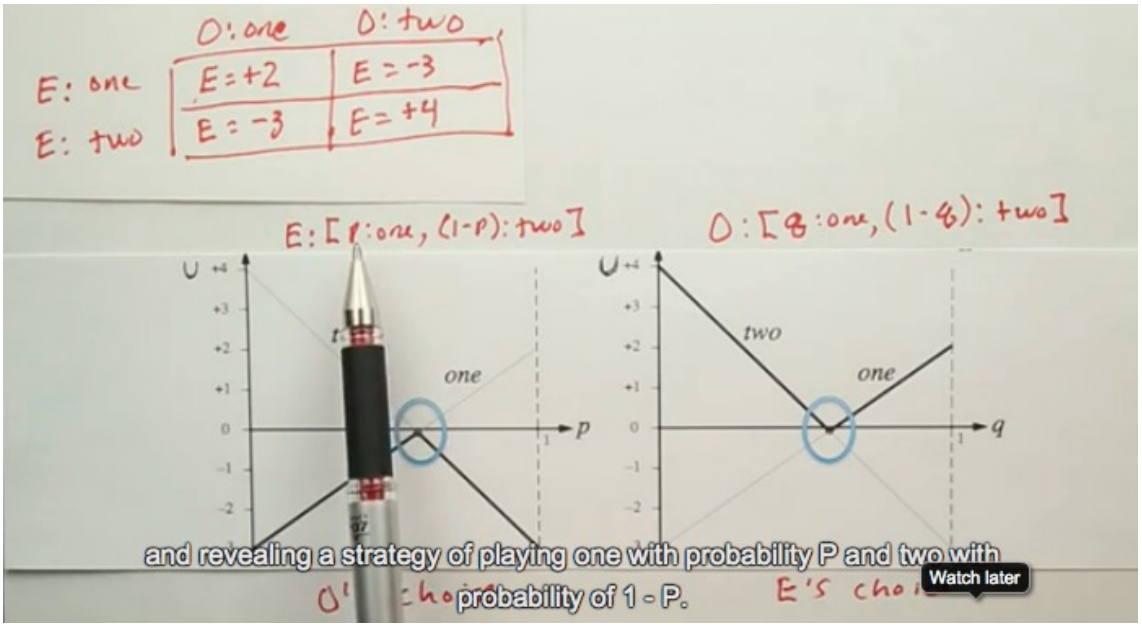


Fig. 9.4. The geometry of the strategies of two-fingered morra

the corresponding table would have about 10^{18} states, and it would be intractable, so we need some strategy to get down to a reasonable number of state.

One strategy abstraction: Treating multiple states as they were really one. One such abstraction is to treat all suits as the same. This means that if no player is trying to get a flush, we can treat all four aces as if they were identical.

Another thing we can do is to lump similar cards together, i.e. if we hold a pair of tens, we can consider the other players cards as being "greather than ten", "lower than ten" etc. Also we can think of bet sizes into small medium and large. Also instead of considering all the possible deals, we can consider subsets of them doing "*monte carlo*" simulations instead of calculating the entire game tree.

This apporach an handle quite a lot with partial observability, stochastic, sequential, dynamic and multiple adversarial agents. However, this method is not very good for unknown actions, and also game theory doesn't help us very much with continous games since we have this matrix form. Doesn't deal very well against an irrational opponent, and it doesn't deal with unknown utility.

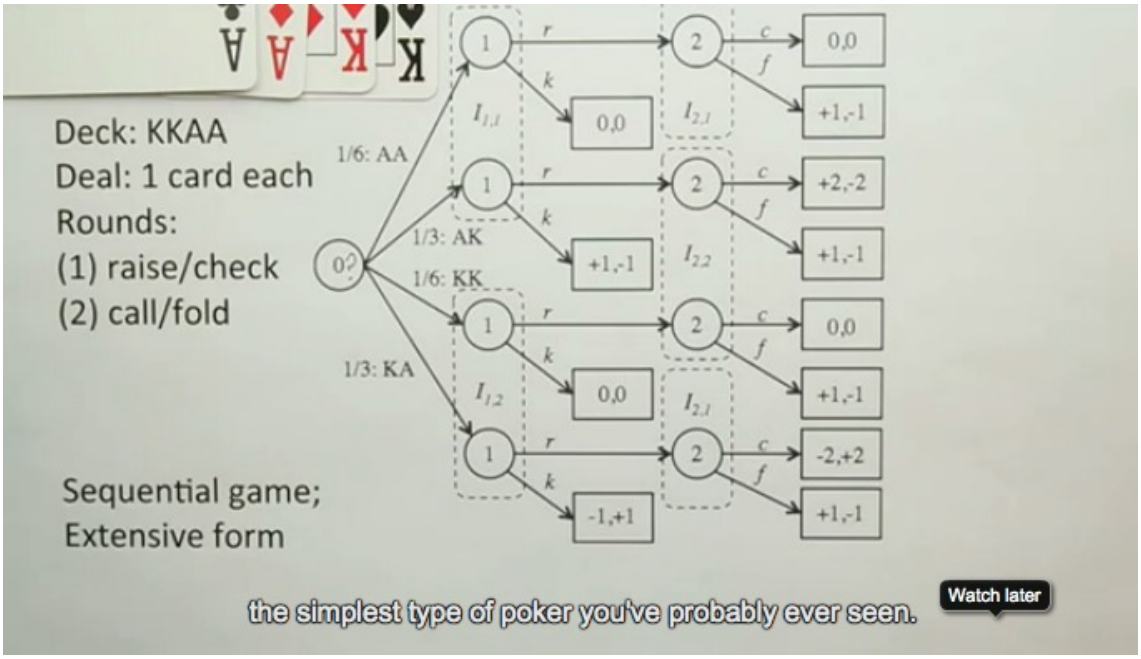


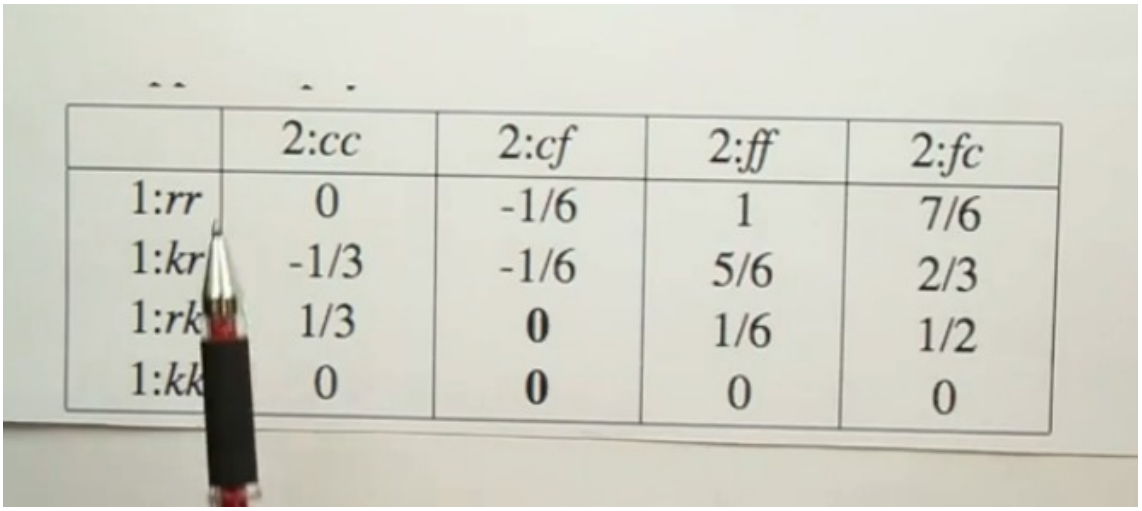
Fig. 9.5. (Very) simplified Poker

9.0.7. Dominating strategies

In figure ?? it at least for me wasn't at all obvious how to solve the thing, so this is a paraphrasing of the helpful explanation given by Norvig.

The game itself is between politicians and the feds. They can both contract or expand the economy or they can do nothing. All of these actions have different outcomes for the different players.

- This game has no dominating strategy.
- Instead of analyzing all strategies against each other (which is possible), we can instead look for dominated strategies.
- For the politicians strategy "pol:0" dominates strategy "pol:-" in the sense that all outcomes are strictly better for the politicians in the former than in the latter strategy.
- This simplifies since the feds can simply chose fed:- since that is better in all respects than any of the other strategies.



	2:*cc*	2:*cf*	2:*ff*	2:*fc*
1:*rr*	0	-1/6	1	7/6
1:*kr*	-1/3	-1/6	5/6	2/3
1:*rk*	1/3	**0**	1/6	1/2
1:*kk*	0	**0**	0	0

Fig. 9.6. Simple poker in "Normal form" notation

- This lead the politicians to optimze by chosing "pol:+", which then is the equilbrium.
- The utility for both parties are three, and this is in some sense the pareto pessimal value, so the equilibrium si not pareto optimal.

9.0.8. Mechanism design

Also called "game design". We want to design the rules of the game so that we get a high expected utility that runs the game, for the people who plays the game and the public at large.

The example that is highlighted is ?? the Google search engine considered as a game. Ads show up at the top, bottom or side of the page. The idea of mechanism design is to make it attractive for bidders and people who want to to respond to the ads. One feature you would want the auciton to have is to make it less work for the bidders if they have a dominant strategy. It's hard to do if you don't have a dominant strategy, so you want to put a dominant strategy into the auction game. An auction is *strategy proof* if you don't need to think about what all the other people ar doing, you only have to worry about your own strategy. We also call this *truth of revealing* or *incentive compatible*.

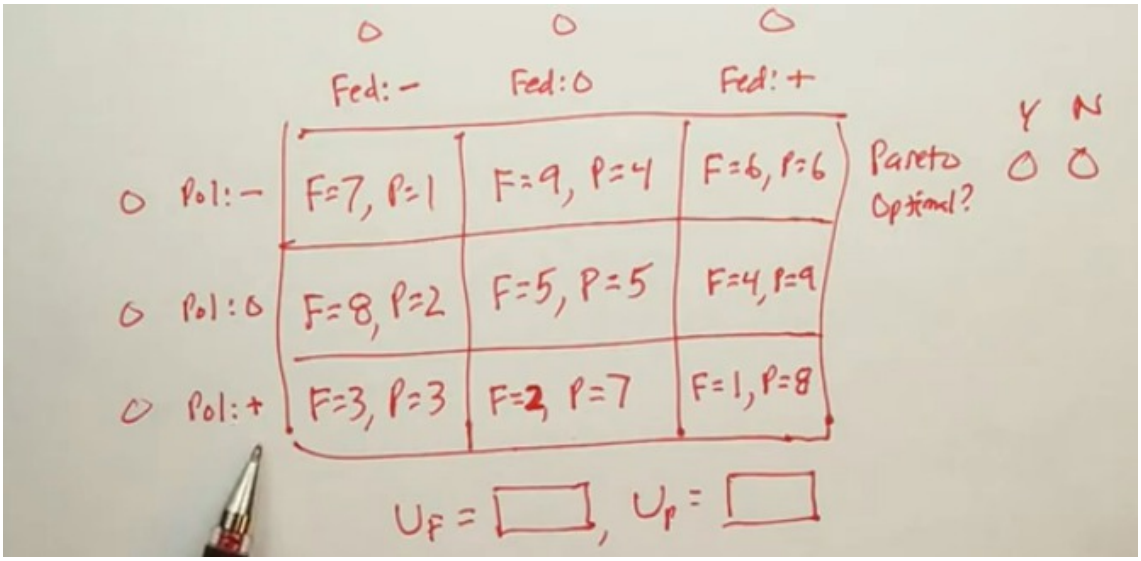


Fig. 9.7. Feds v.s. politicians, a game of chance and domination :-)

The second price auction

Bids come in whatever they want, whomever bids the highest wins, but the price they pay is the offer of the second highest bidder. A policy that is better than another strategy everywhere, then it is said to be *strictly dominating*, if it ties in some places, then it is called *weakly dominating*.

Rmz: *Need to check this again. I didn't understand the second price auciton thing!!*

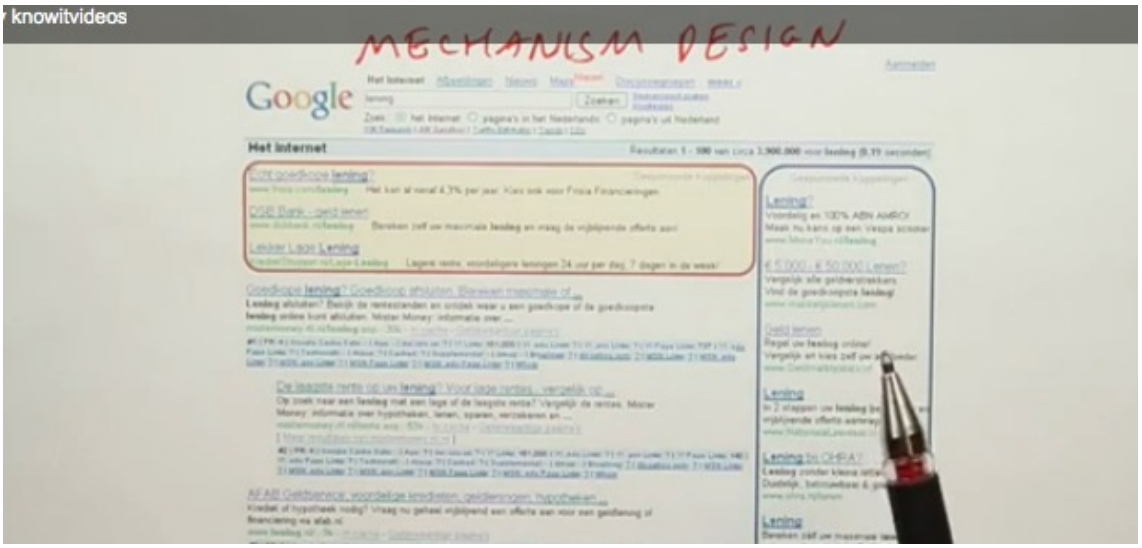


Fig. 9.8. Considering the google search engine as a game

10. Advanced planning

Last time we talked about planning we left out four important things: Time, reseources, active perception and hierarchical plans. We have a task network with dependencies.

10.1. Time

The tasks has durations. The task is to figure out a schedule when each of the tasks start so that we can finish as soon as possible. Each task has a time called "ES", whichis the earliest possible start time, and another time called "LS" which is the latest possible start time, both of these constrained by minimizing the total time we have to complete the network.

We can find this using recursive formulas that can be solved by dynamic programming.

$$\begin{aligned} \mathrm{ES}(s) &= 0 \\ \mathrm{ES}(B) &= \max_{A \to B} \mathrm{ES}(A) + \mathrm{duration}(A) \\ \mathrm{LS}(f) &= \mathrm{ES}(f) \\ \mathrm{LS}(B) &= \min_{A \leftarrow B} \mathrm{LS}(B) - \mathrm{duration}(A) \end{aligned}$$

10.2. Resources

The assemblytask can't be solved because we are missing a nut, but we have to compute $4! \cdot 5!$ paths to discover that. The reason is that we would need to check all combinations of nuts and bolts during backtracking. The idea of resources is to let each of the nuts and bolts be less unique and then to extend the classical planning problem to handle this problem. We do this by ading a new type of statement stating that there are resources, and how many there are. The actions

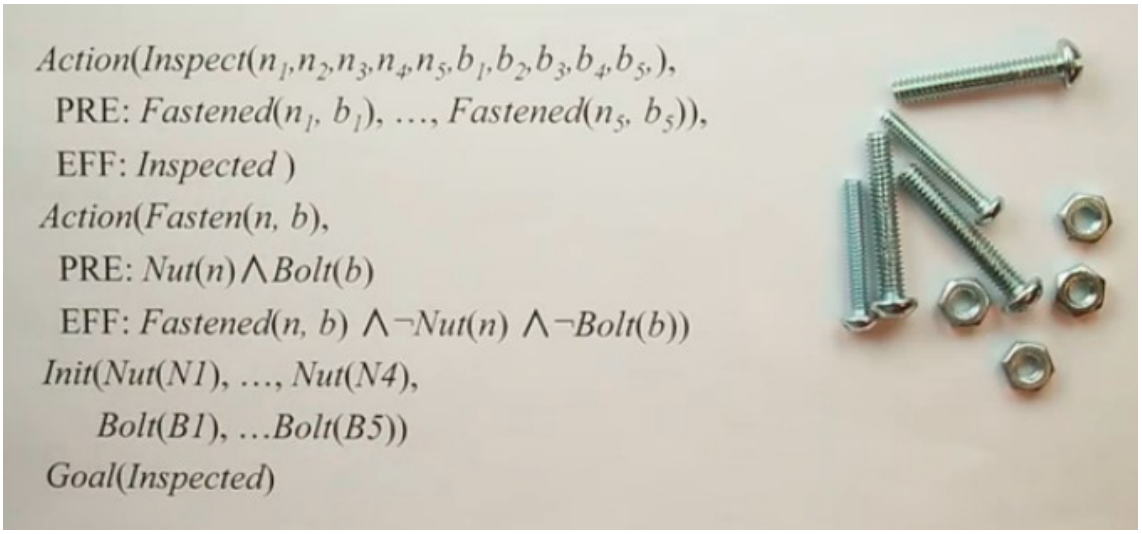


Fig. 10.1. An assembly task.

have "consume" clauses, and "use" clauses. The use clause uses some resource while the processing is being done, but then returns it to the pool afterwards. The consume clause removes a resource from the pool never to return it.

Keeping track of resources this way gets rid of that computational or exponential explosion by treating all of the resources identically.

10.2.1. Hiearchical planning

The idea is to reduce the abstraction gap. What does this mean? Well, we live about a billion seconds, each second perhaps a thousand muscles we can operate, maybe ten times per second, so we get something like 10^13 actions we can take during a lifetime (give or take a few orders of magnitudes). There is a big gap between that and about 10^4 which is the max number of items current planning algorithms can handle. Part of the problem there is such a gap is that it' really hard to work with all of the detailed muscle movements, we'd rather work with more abstract actions. So we'll introduce the notion of a *hierarchical task network* (*HTN*), so instead of talking about all the low level steps we can talk about higher order steps of which there is maybe a smaller number. This idea is called *refinement planning*. Here is how it works. In addition to regular actions we have *abstract actions* and various ways to map these into *concrete actions*.

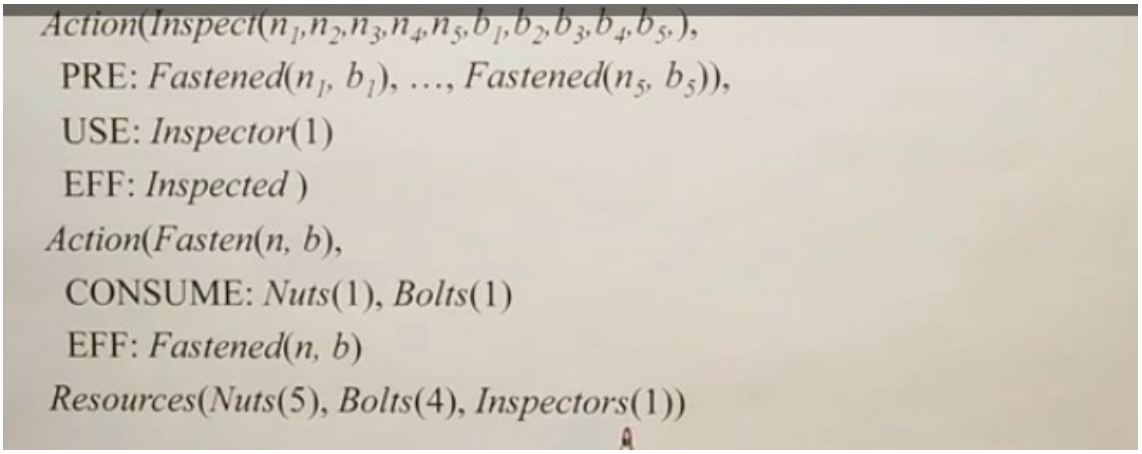


Fig. 10.2. An assembly task with resources

When do we know when we have a solution? We have a solution when at least one of the refinements of an abstract action reaches the goal.

In addition to do an and/or search we can solve an abstract planning problem without going down to the concrete steps. One way to do that is to use the concept of *reachable states*.

In figure ?? the dotted lines surround concrete actions and is in itself interpreted as an abstract action. This is like a belief state where we are in multiple states because we don't know which action was actually taken, but instead of being subject to a stochastic environment we are using abstractions to quantize over sets of states where we haven't chosen the refinement left. We can check if there is an intersection between the reachable state and the goal state, and if there is the plan is feasible. To find a plan that actually works we should the search backwards instead of forwars in order to have a smaller tree to search through.

Sometimes it's very hard to specify exactly which states are reachable, instead it is practical to work with sets of states that are *approximately reachable states*. We can then approximate the states with a lower bound and upper bounds of the states we might reach, but we're not entirely certain about all the combinations.

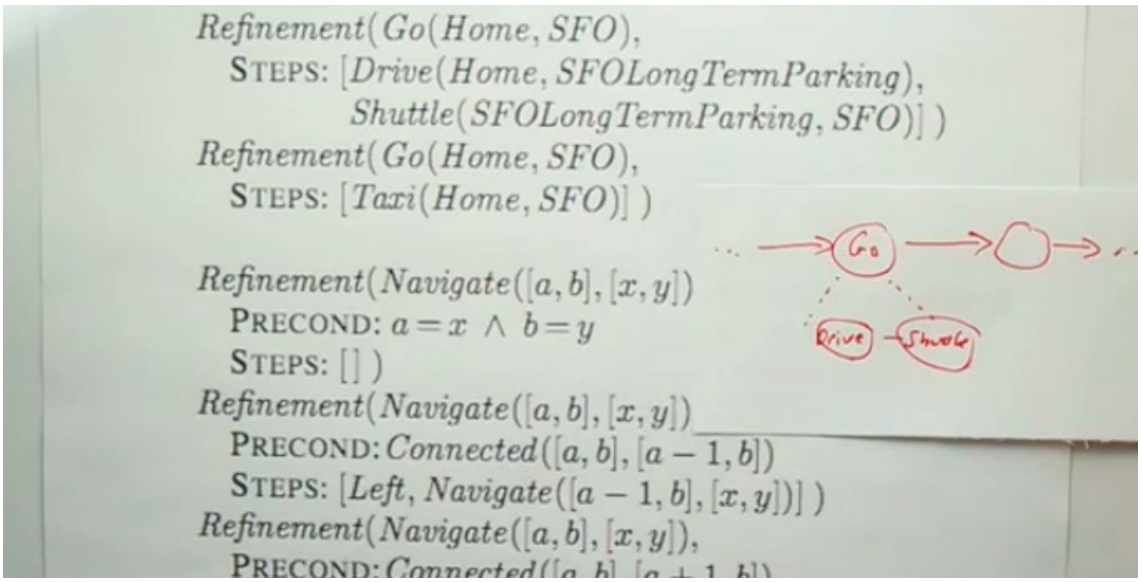


Fig. 10.3. Hierarchical planning, refining actions

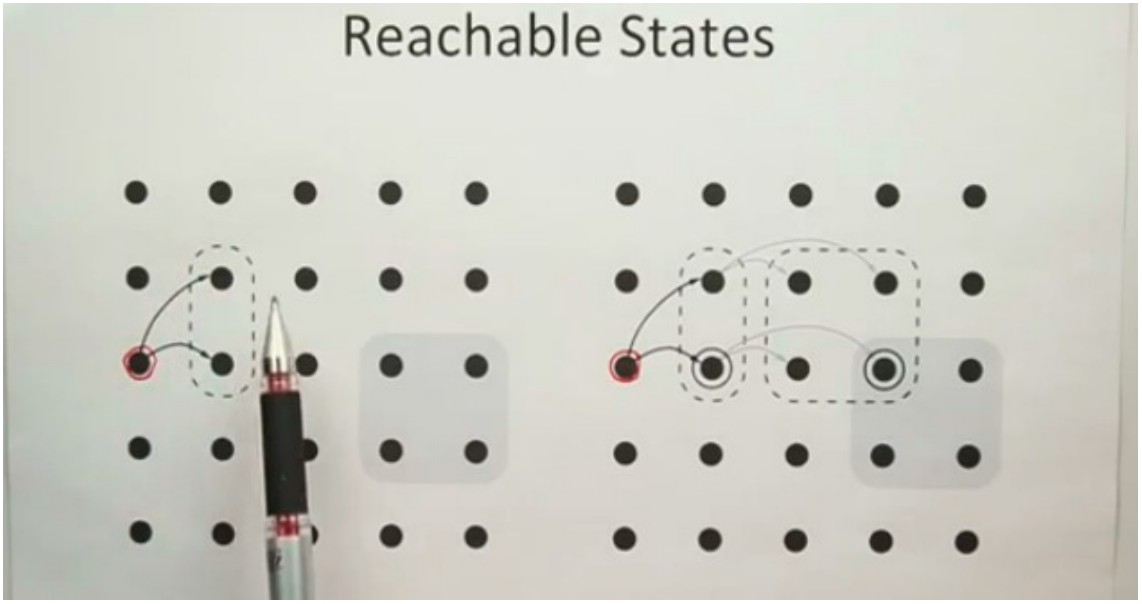


Fig. 10.4. Reachable states

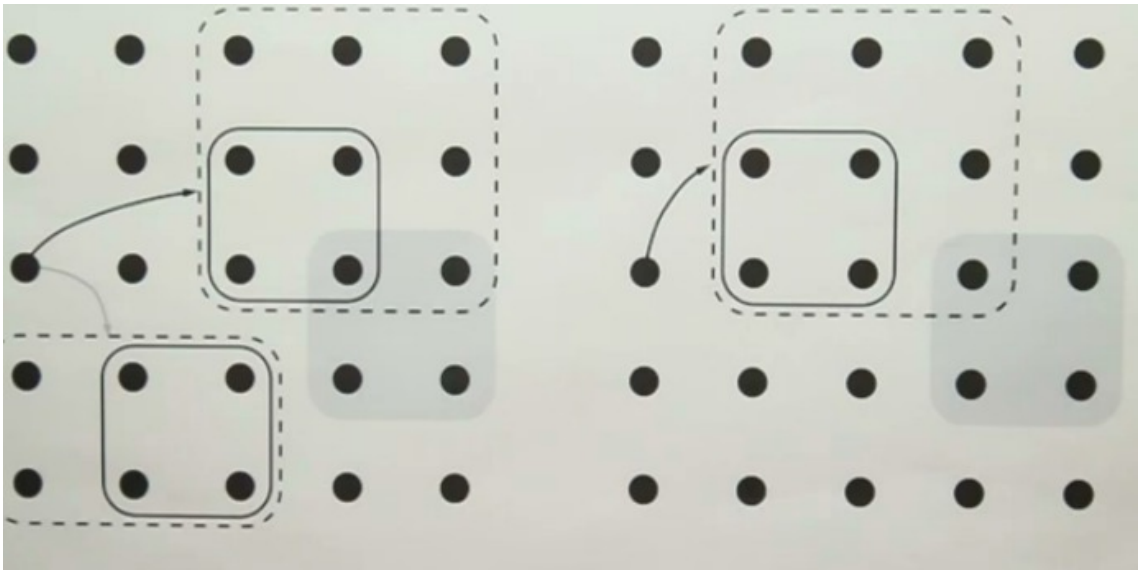


Fig. 10.5. Upper and lower bounded states

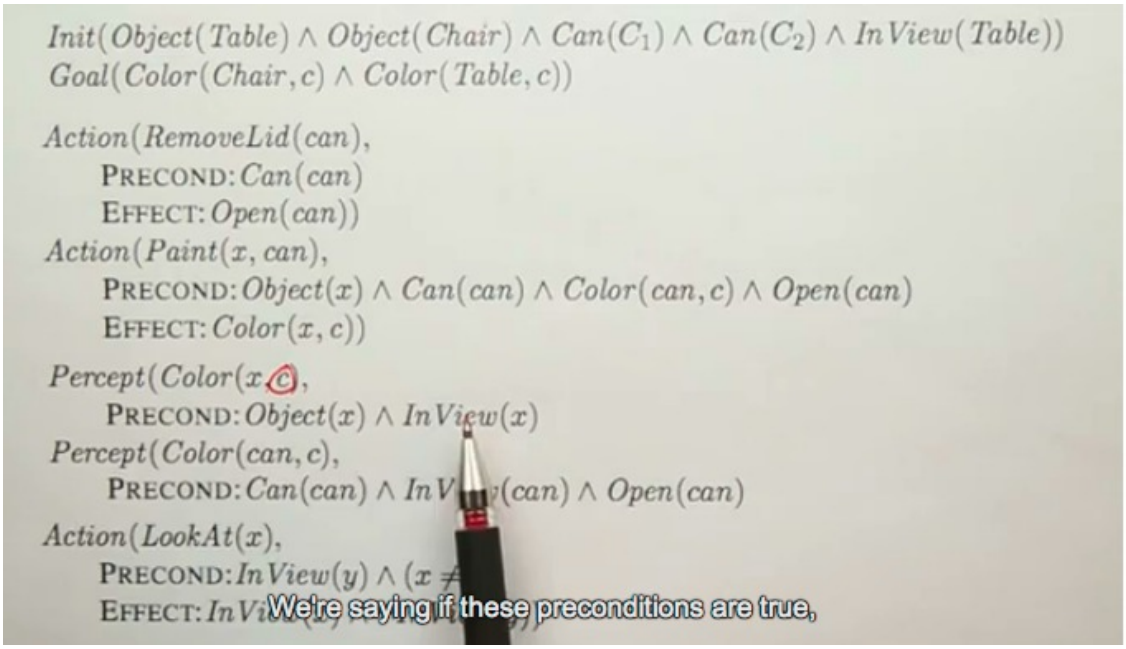


Fig. 10.6. Sensing and planning

11. Computer vision

Computer vision is the first application of AI that we will look at. Computer vision is the task as of making sense of computer imaging. It's basic stuff. We will do some things about classification and 3D reconstruction

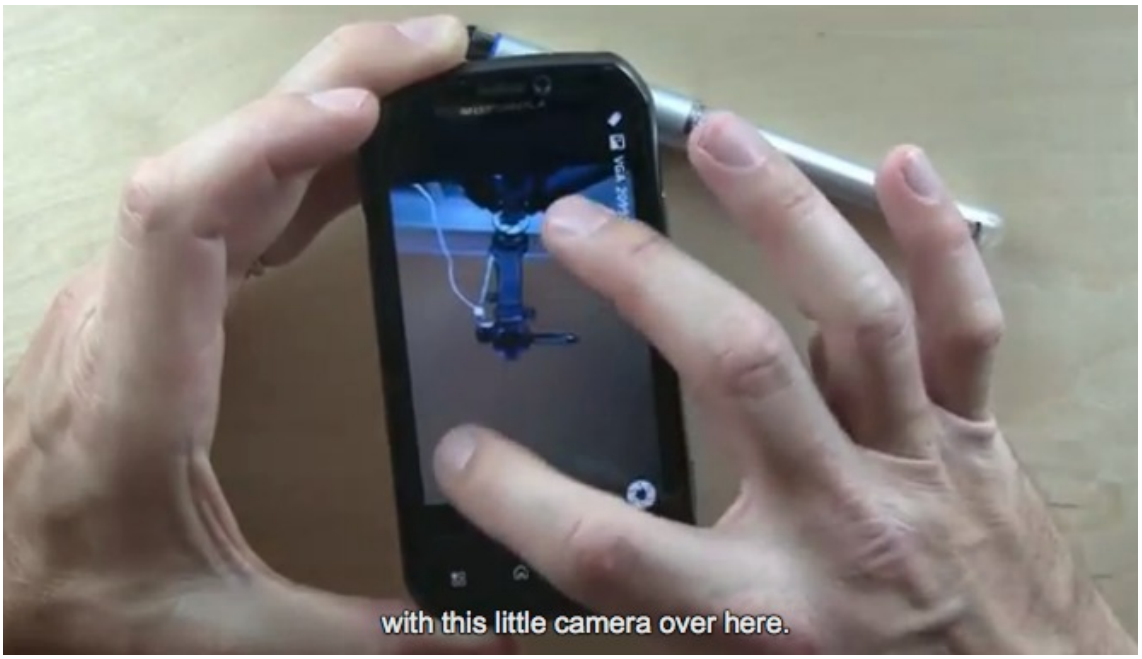


Fig. 11.1. The document camera used by Thrun imaged by his phone

The science of how images is captured is called *image formation*. The simplest camera is called a *pinhole camera*. There is some basic math. governing a pinhole camera.

Perspective projection means that the projective size of any object scales with distance. The size of the projected image is proportional to the size of the object being imaged, and inversely proportional to the object being imaged:

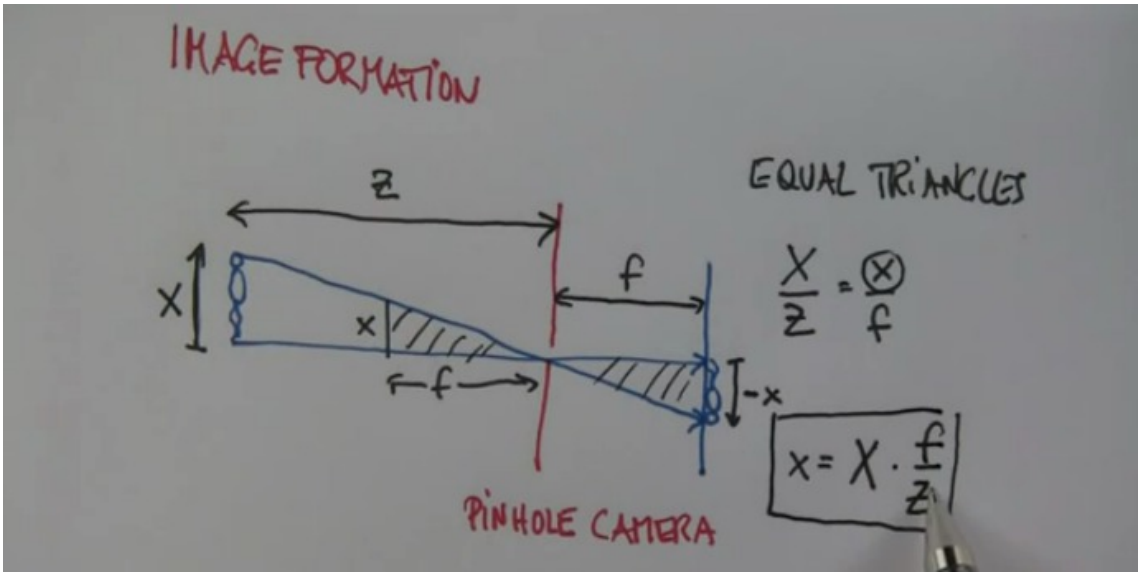


Fig. 11.2. A pinhole camera

$$x = X\frac{f}{Z}$$

Actual camera images have two dimensions, and the projection laws applies to both dimensions. A consequence of this is that parallell lines seems to converge in the distance, the point at which they meet called a *vanishing point*. In some cases (like in fig **??**) there are more than one vanishing point in a scene.

Lenses are more effective of collecting light than pinhole cameras. There is also a limit on how small the hole can be made, since at some small size (and smaller) something called *light diffraction* will start to blur the image. However, using a lens will require the lens to be focused. How to focus a lens is governed by the equation:

$$\frac{1}{f} = \frac{1}{Z} = \frac{1}{z}$$

This latter law isn't that important, but still it's there :-O

Fig. 11.3. Vanishing points are where parallell lines in the scene being imaged converge to a single point in the image.

11.1. Computer vision

Things we can do with computer vision is to:

- Classify objects.
- 3D reconstruction (with multiple images, or stereo cameras etc.)
- Motion analysis.

In object recognitions a key concept is called *invariance*: There is natural variations of the image that don't affect the nature of the object itself. We wish to be invariant in our software to to these natural variations. Some possible variances are:

- Scale (the thing becomes bigger or smaller)
- Illumination (the light source changes)
- Rotation (turn the thing around)
- Deformation (e.g. a rotor that rotates)

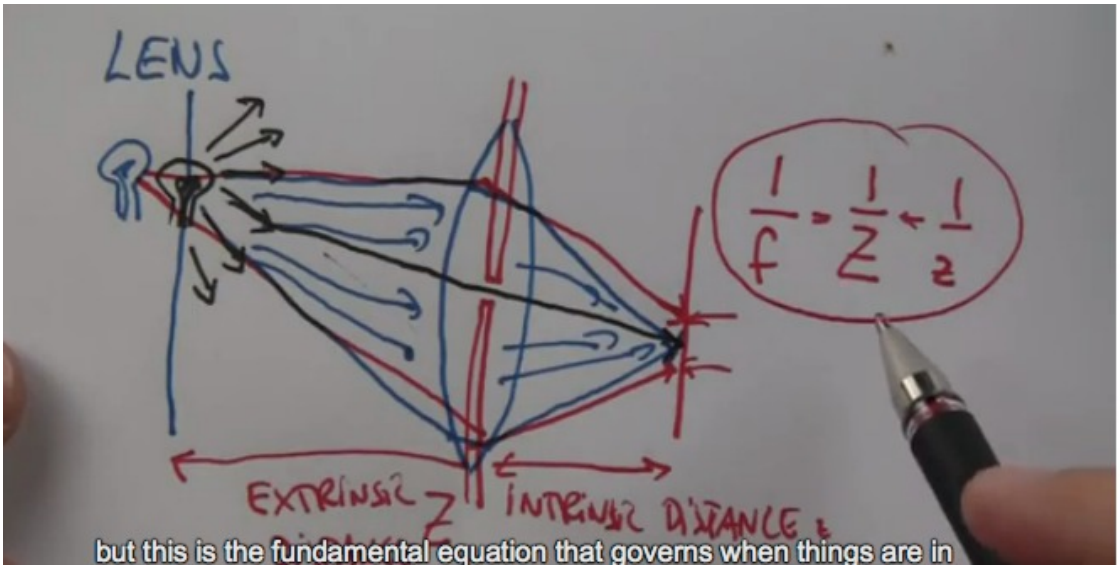


Fig. 11.4. A lens and some equations governing its behavior

- Occlusion (object behind other objects)
- View point (vantage point). The position from which one sees the object.

These and other invariances *really matter* when writing computer software. If we succeed in eliminating changes from the objects we look at we will have solved a major computer vision problem.

In computer vision we usually use grayscale representations since they in general are more robust with respect to lighting variations. A grayscale image is a matrix of numbers usually between 0 (black) and 255 (white).

One of the things we can do in computer vision is to *extract features*. We can do this by filtering the image matrix. These programs are called *feature detectors*.

Filters are in general formulated like this:

$$I \otimes g \mapsto I'$$

where I is an image, I' is the transformed image. g is called the *kernel*. In general what happens is that we use a *linear filter*:

Fig. 11.5. A grayscale image of a bridge in Amsterdam.

Fig. 11.6. The image in ?? after being processed by an edge detector.

$$I'(x,y) = \sum_{u,v} I(x-u, y-v) \cdot g(u,v)$$

This is of course a *convolution*. The kernel by any dimension (m, n).

Gradient images are combinations of horizontal and vertical edges.

The state of the art in edge detection is a *canny edge detector*. In addition to finding the gradient magnitude it traces areas and finds local maxima and tries to connect them in a way that there's always just a single edge. When multiple edges meet the canny detector has a hole. The detector is named after professor *J. C*

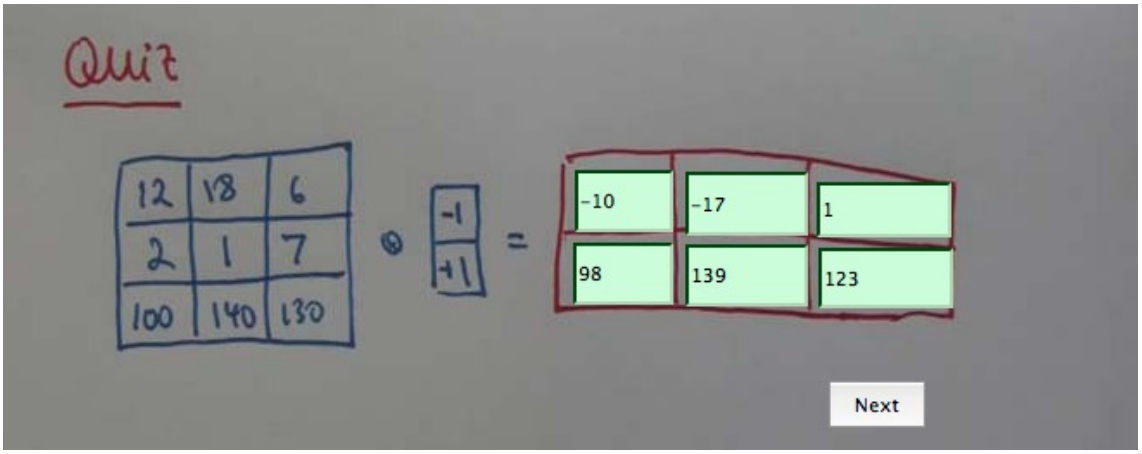


Fig. 11.7. An example of convolving a 3x3 matrix with with an 1x1 matrix

Canny at *U.C. Berkeley*, and he did most of the most impressive pieces of work on edge detection. There are also many commonly used masks, listed in figure ??.

Linear filters can also be applied using *gaussian kernels*. If you convolve an image using a Gaussian kernel, we get a blurred image. There are some reasons why we would want to blur an image:

- *Down-sampling*: It's better to blur by gaussian before down-sampling to avoid aliasing.
- *Noise reduction*: Reduce pixel-noise.

Kernels in linear filters are associative, so that if we have a convolution that is a combination of multiple filters, we can do:

$$I \otimes f \otimes g = I \otimes (f \otimes g)$$

So if f is a gradient kernel, and g is a Gaussian kernel, then $(f \otimes g)$ will be a *Gaussian gradient kernel*.

Sometime you want to find corners. Corners are *localizable*. The *harris corner detector* is a simple way to detect corners.

The trick used by the harris detector is to use an eigenvalue decomposition. If we get two large eigenvalues we have a corner (rotationally invariant). It is a very

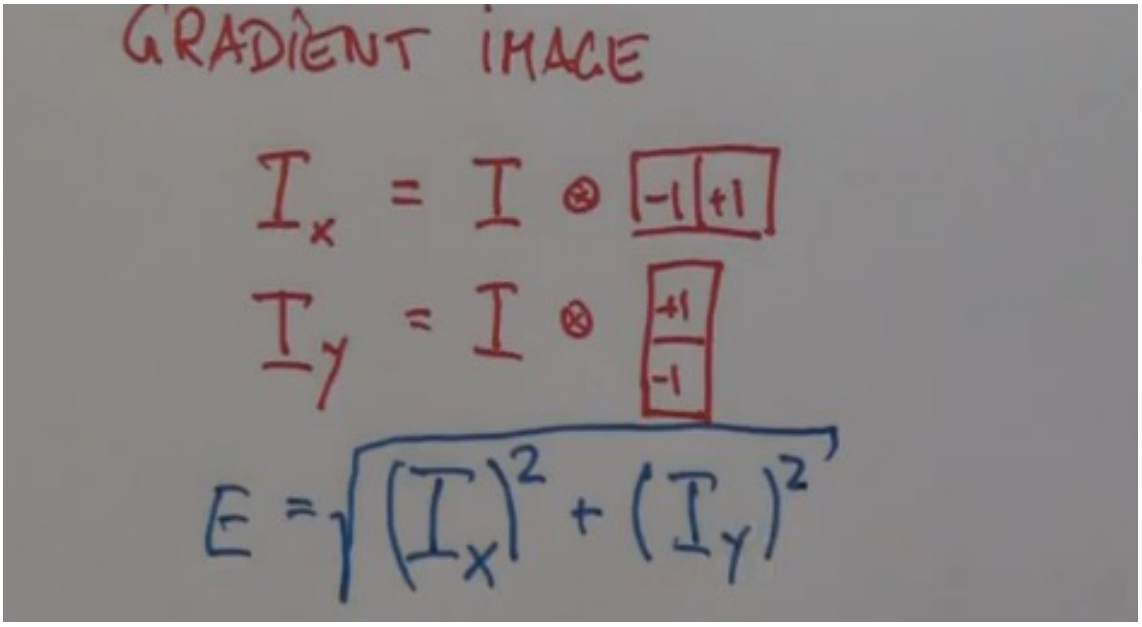


Fig. 11.8. Calculating the gradient of an image using two convolutions and a square root

efficient way to find stable features in high contrast images in a (quite) invariant way.

Modern feature detectors extend the Harris detector into much more advanced features that are localizable, has unique signatures. Some of these are *HOG Histogram of Oriented Gradients* and *SIFT Scale Invariant Feature Transform*. Thrun recommends using HOG or SIFT.

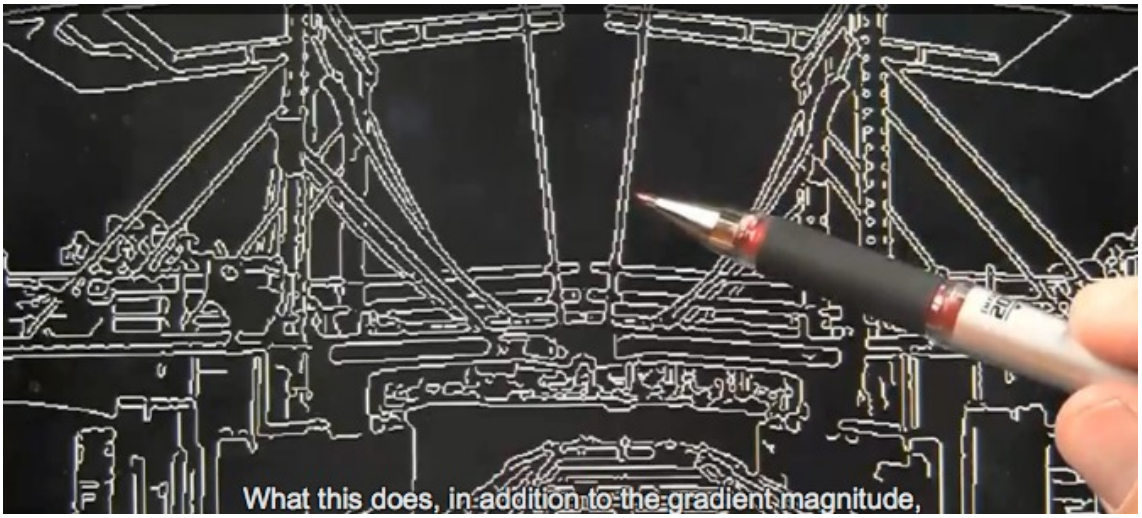


Fig. 11.9. The "Canny" edge detector. A really fancy edge detector

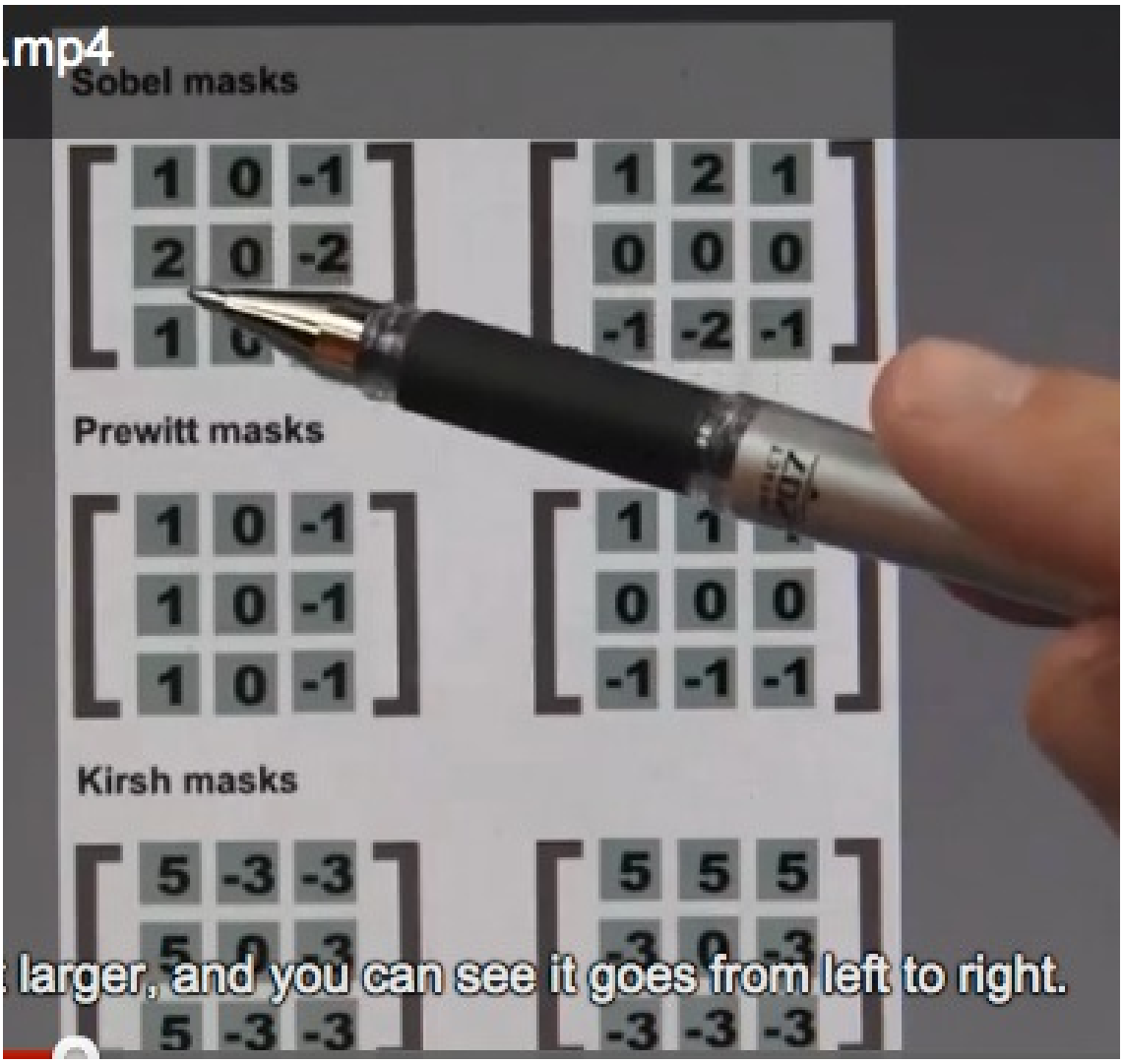


Fig. 11.10. A collection of masks (convolution kernels) that have traditionally been used to detect edges

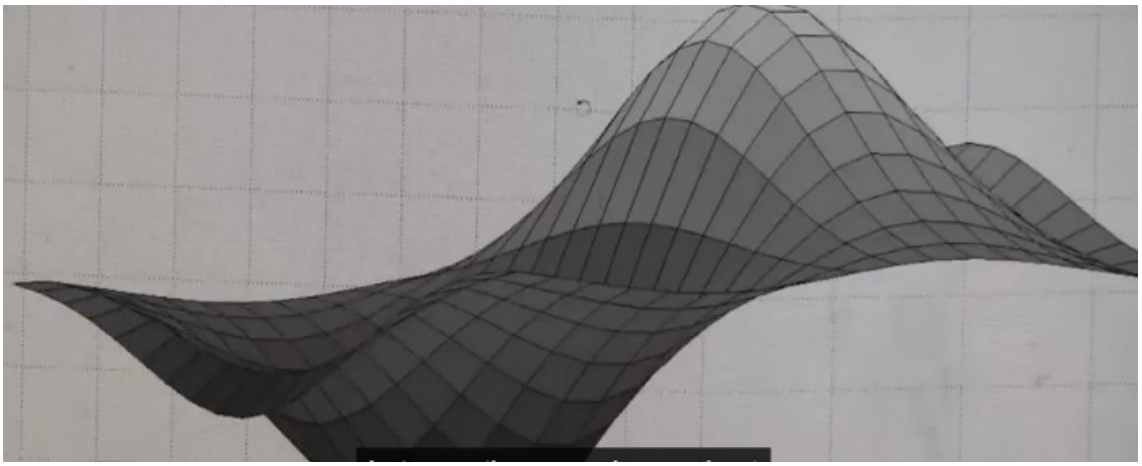


Fig. 11.11. A Gaussian gradient kernel

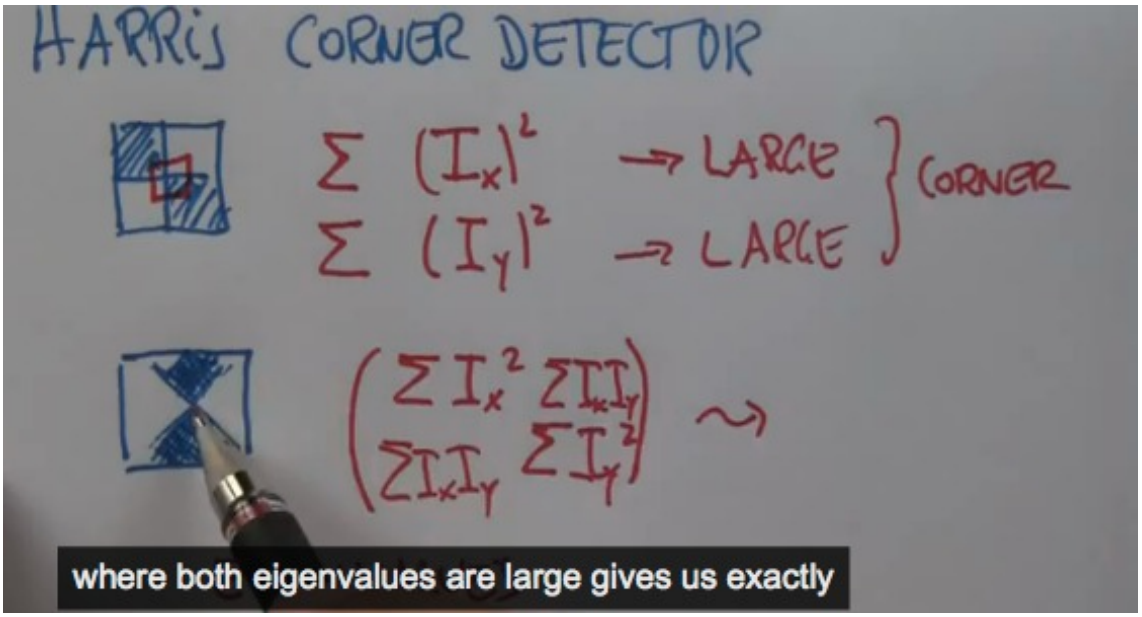


Fig. 11.12. The Harris corner detector using eigenvectors of subimages to find the directions to look for corners along. It's really good.

12. 3D vision

The idea is to make a representation of the 3D information from the 2D images. The range/depth/distance is perhaps the most important thing. The question is: Can we recover the full scene from a single or multiple images.

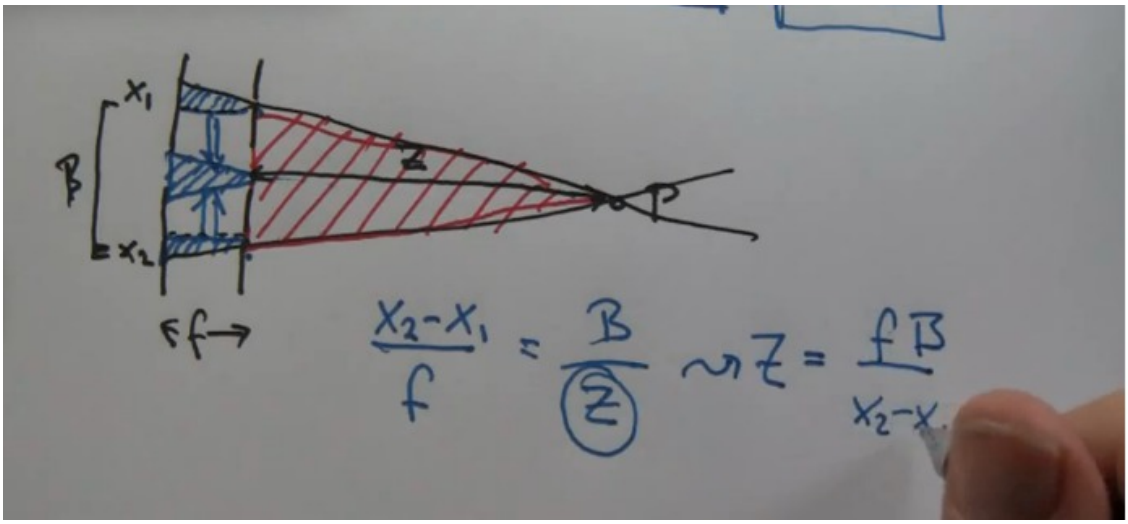


Fig. 12.1. The physics and math behind 3D stero vision

Assuming a pinhole stero-rig, and we wish to recover the depth z of a point p. There is a simple trick: The size of the red triangle in fig ?? is proportional to the combined small triangles "inside the camera" (the blue parts, moved together to form a single triangle). The things we know, the focal length and the baseline is called *intrinsics*, and what we wish to recover is the depth z.

12.0.1. Data association/correspondence.

Given a point in one image, where do you search for the same image in another image? If you correspond the wrong points, you will ger *phantom points*: Points

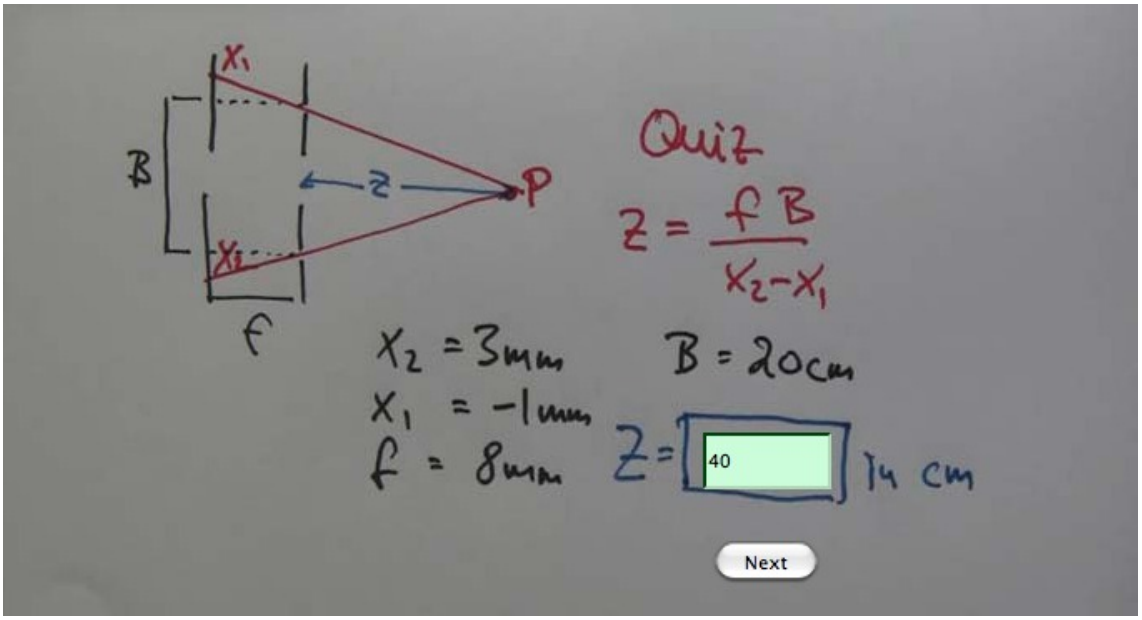


Fig. 12.2. A quiz about depth fields

you think are there but which are not there really there. Getting correspondences right is really important.

For stereo rigs the correspondence rigs will be along a line. How to find correspondences we can use various techniques, matching small image patches and matching features will both work. One can then use a sum of squared metric to find the displacement that minimizes that error (disparity) and use that as a match for the displacement.

The ssd method is very useful when searching for image templates, e.g. when searching ror alignments. Using a disparity map we can find which parts of the image is associated with large or small disparities. It is in fact possible to use these disparity maps to infer depth field information about the the images. High disparities indicate closeness and large disparities indicates larger distance. In some patches it is difficult to determine the disparity since there few features to use for detection. Disparity maps can be used in automatic driving (see fig **??**). It isn't very informative since there are not many features in the desert. However, where it does something it does a fine job.

Searching for correspondence means to search along a single scan line but it can also be maningful to look at the context in which a pixel appears in (color and/or

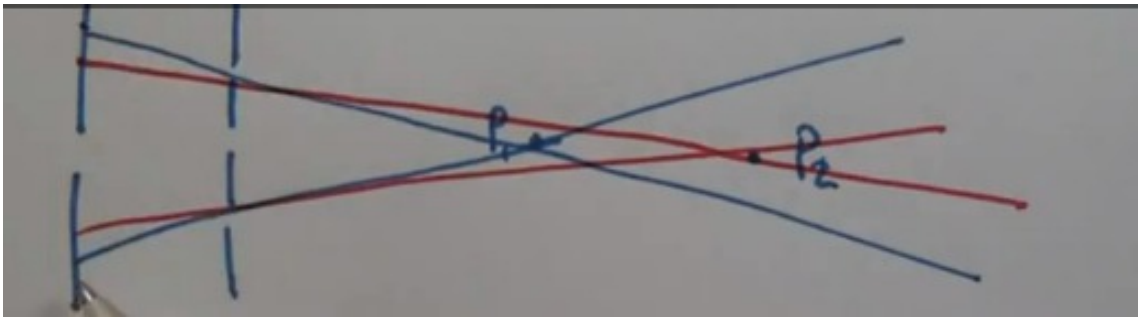


Fig. 12.3. Stero rigs can give rise to "phantom points". Points that appear to be identidal in the scene but really are just optical illusions.

grayscale). We do the shifting by minimzing a cost function consisting of two parts, color match and occlusion.

The tricky part is to compute the optimal aligment, usually it is done using dynamic programming. This gives an n squared algorithm which is reasonably fast considering the alternatives. The algorithm either goes right diagonal. See fig ??

We evaluate the value function (*Bellman function*) and trace the way the path propagates, and that gives us the best possible aligment for the scanlines. This method represents the state of the art in stereo vision.

There are a few things that doesn't work very well for dynamic programming. If you have a small object in front of a large object, the object will appear to the left in one camera and to the right in the other.

Big round near objects will not let the camera see the same point at the edge, and that can lead to problems.

A final problematic issue is reflective objects. The specular reflection shown in figure ?? will show up in quite different areas on the camera, and reconstruction will be tricky.

There are a few ways to improve stereo perception. In figure ?? a rig showing two cameras and a projector that can project various patterns onto a scene is depicted. These patterns can be used to reduce the ambiguity in the scenes (see ??). Disambiguation becomes easier. In ?? we see how Sebastian is imaged in the rig. Another solution is the microsoft kinect. It uses a camera system with a laser that adds texture to a scene. It's actually pretty good ;-)

Fig. 12.4. When looking for corresponding point between two images taken from a stereo rig, one must look a long a horizontal line in the image planes

There are a bunch of techniques for sensing range. Laser rangefinders use travel time for laser pulses. Laser rangefinders is used as an alternative to stereo vision because it gives extremely good 3d maps

12.1. Structure from Motion

The structure refer to the 3d world. Motion refers to the location of the camera. The idea was to move the camera around and then recover the 3d scene from many images.

Tomasi and Kanade in 92 used harris corner extraction and recovered the structure. Used PCA. Also used flight recorded stuff to produce elevation models of the terrain. Marc Pollifeys did some intersting work on buildings in his own hometown. Was able to make models from houses. Very impressive work. Did maps of entire cities. There are a bunch of occlusion maps, but that's ok. Everything can't be reconstructed.

The math behind SFM are involved (see ??). It uses the perspective model, assumes moving cameras. Contains three rotation matrices, offset matrices etc. The equation shows how the imaging happens from cameras that are translated and

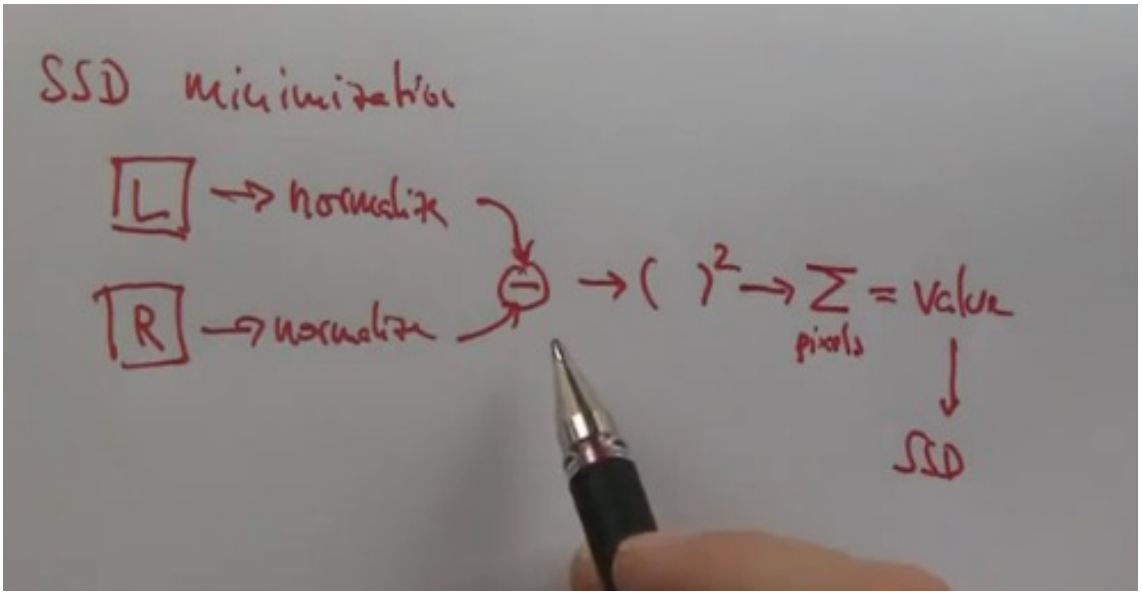


Fig. 12.5. This is a calculation used to find a disparity between images, the objective function will then be to minimize to total disparity, from that we get a disparity map as seen in figure ??.

rotated to any point in space. The problem of solving SFM can then be formulated as a minimzation problem (see **??**) that can then be solved using misc techniques *gradient descent*, *conjugate gradient*, *Gauss-Newton*, *Levenber Marquard* (a common method) also using *singular value decomposition (affine, orthographic)*. This is heavy stuff, and also really cool :-)

Even if you are perfectly able to recover all points in a camera/structure setup, there will still be seven variables you can't recover (fig ??). These are, position (3), direction (3) and scaling (1). Any one of these gives a full one-dimensional subspace that the solution is invariant with respect to.

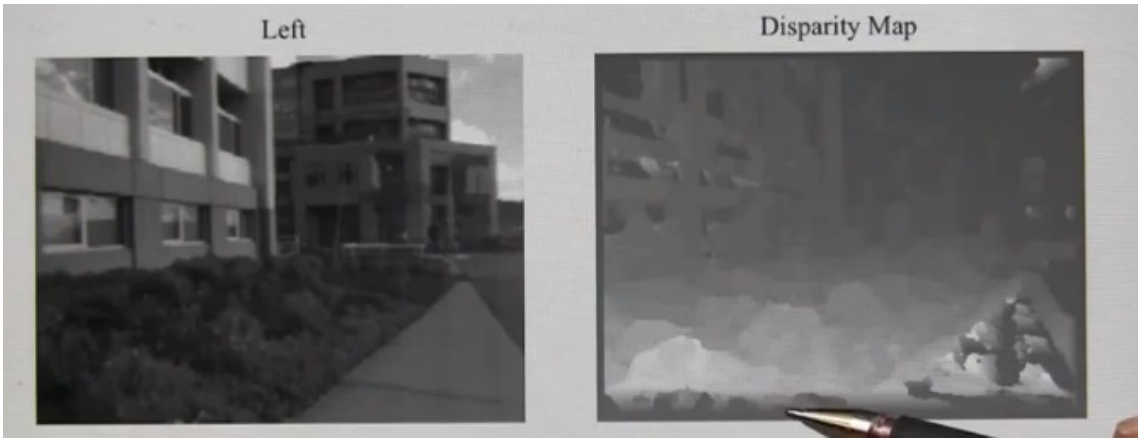


Fig. 12.6. The disparities between images in stero pairs of pictures can be summarized in a new image called a "disparity map". This map can then be used as an estimate for a depth field.

Fig. 12.7. The self-driving car "Stanley" that won the DARPA grand challenge uses disparity maps (among several other techiniques) to estimate the geometry of the road around the car.

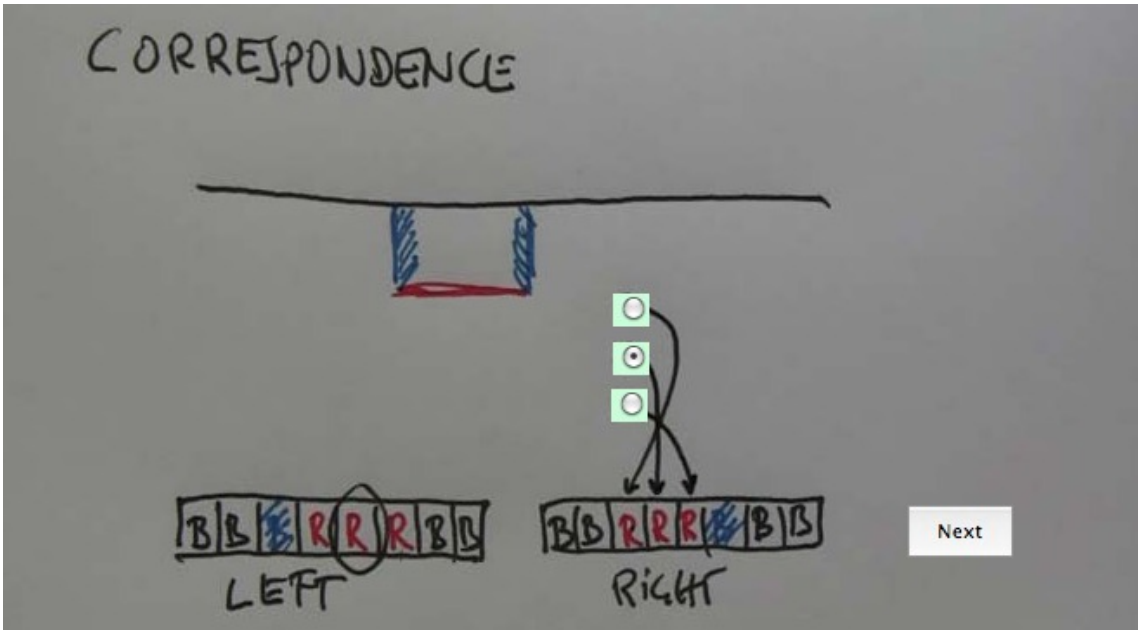


Fig. 12.8. Searching for correspondences can be a tricky task, since there can be both displacements and occlusions that come into play when generating the final result. The task of the correspondence map calculation is to balance the costs from these different sources of errors and then minimize the total cost.

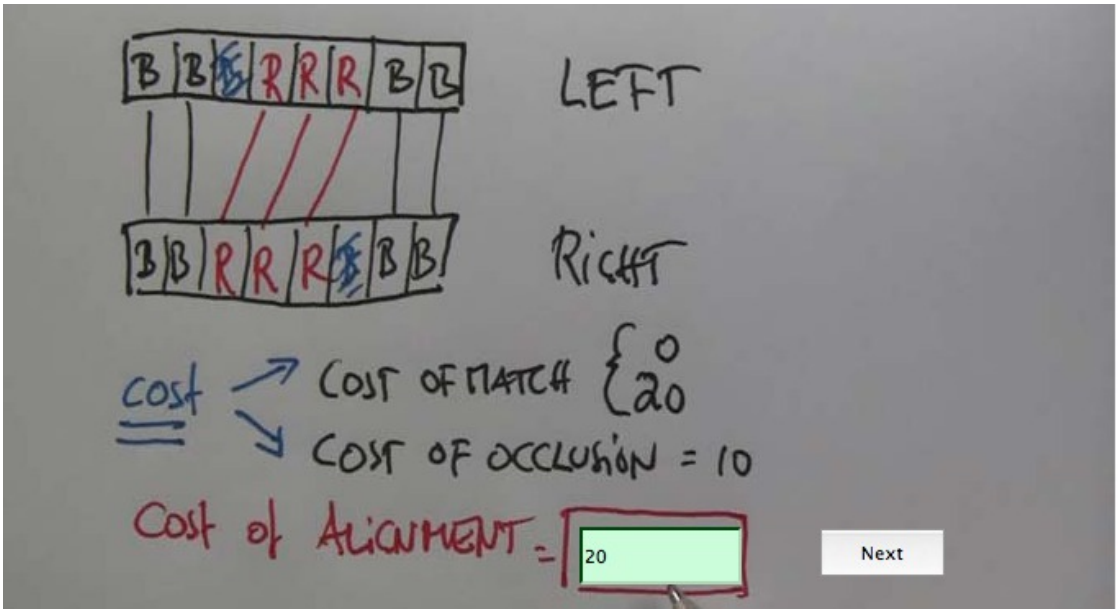


Fig. 12.9. An example of how the cost of alignment can be calculated.

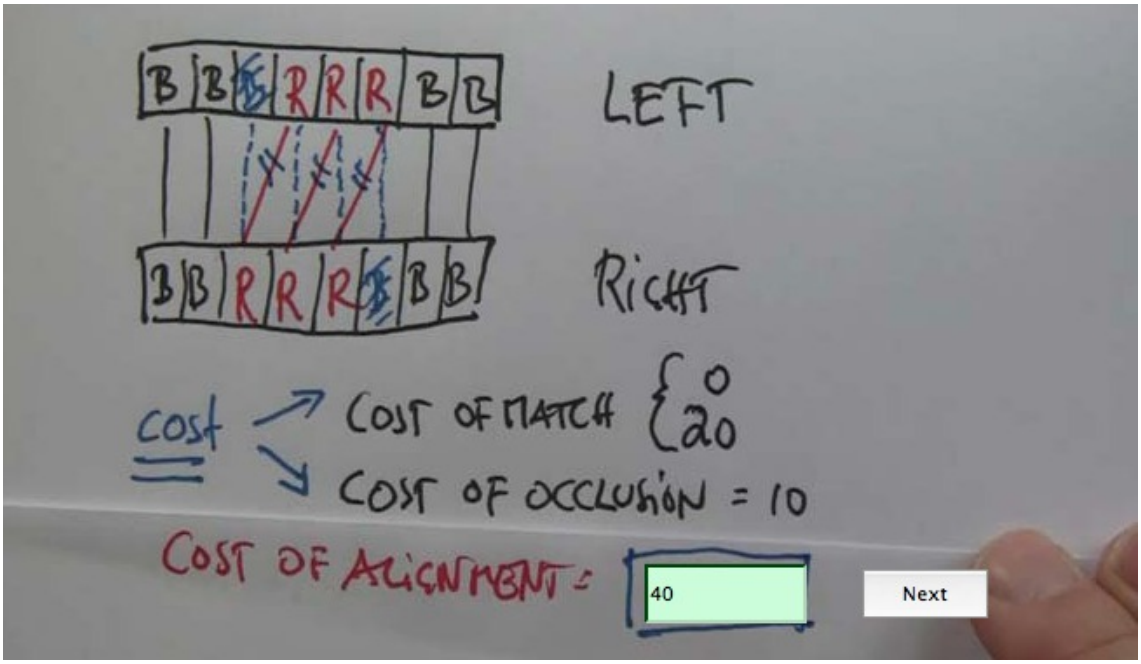


Fig. 12.10. Another example of how the cost of alignment can be calculated.

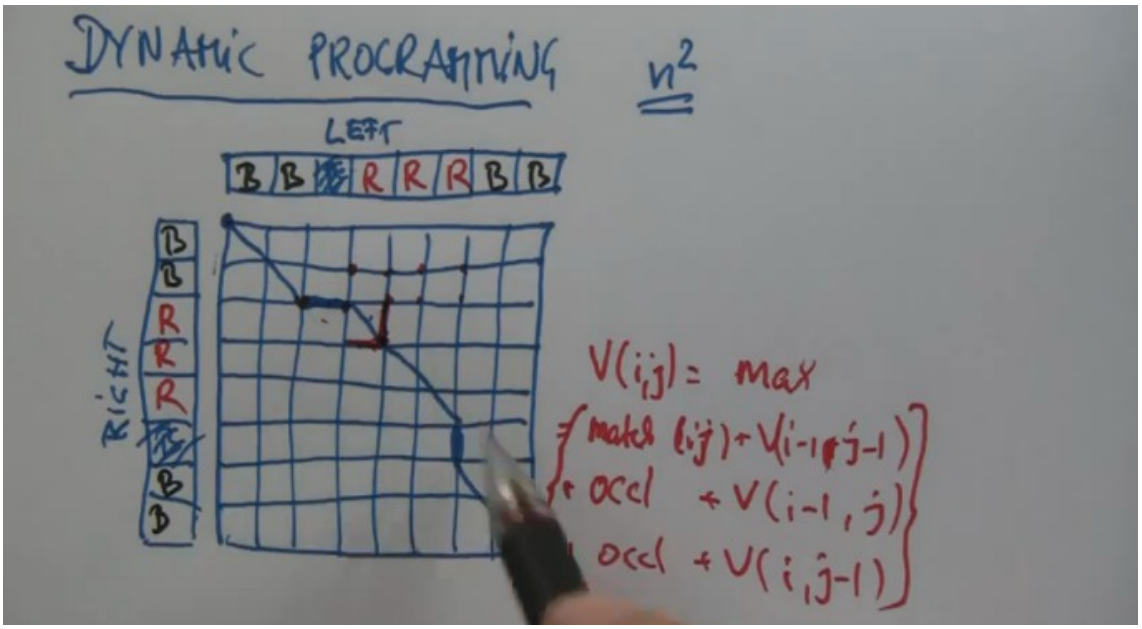


Fig. 12.11. Calculating correspondences interpreted as a dynamic programming problem

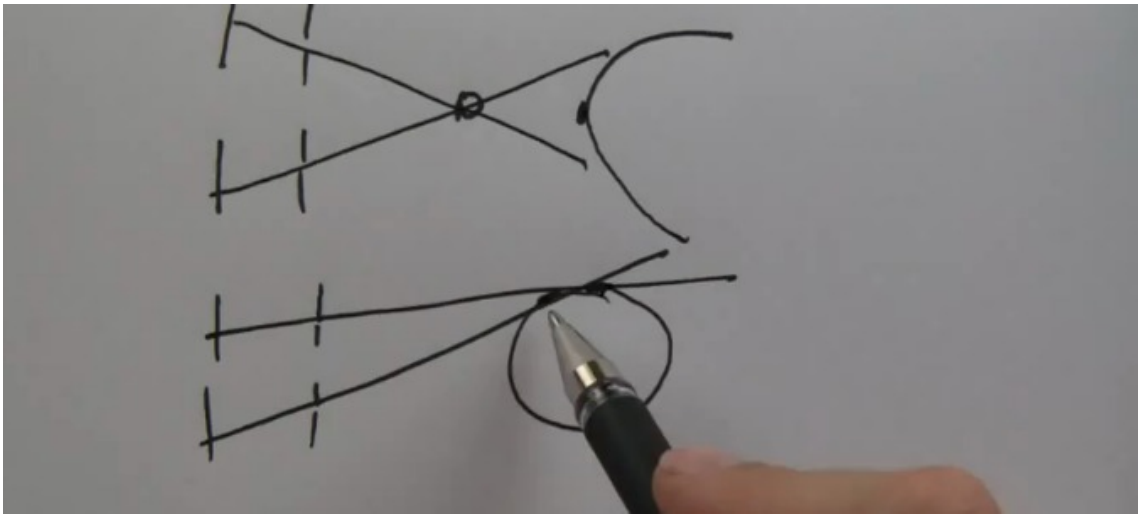


Fig. 12.12. Stereo vision is difficult when considering points on the edge of a large sphere close by, since the leftmost point of the sphere seen by the left imager is different than the leftmost point seen by the right imager, etc.

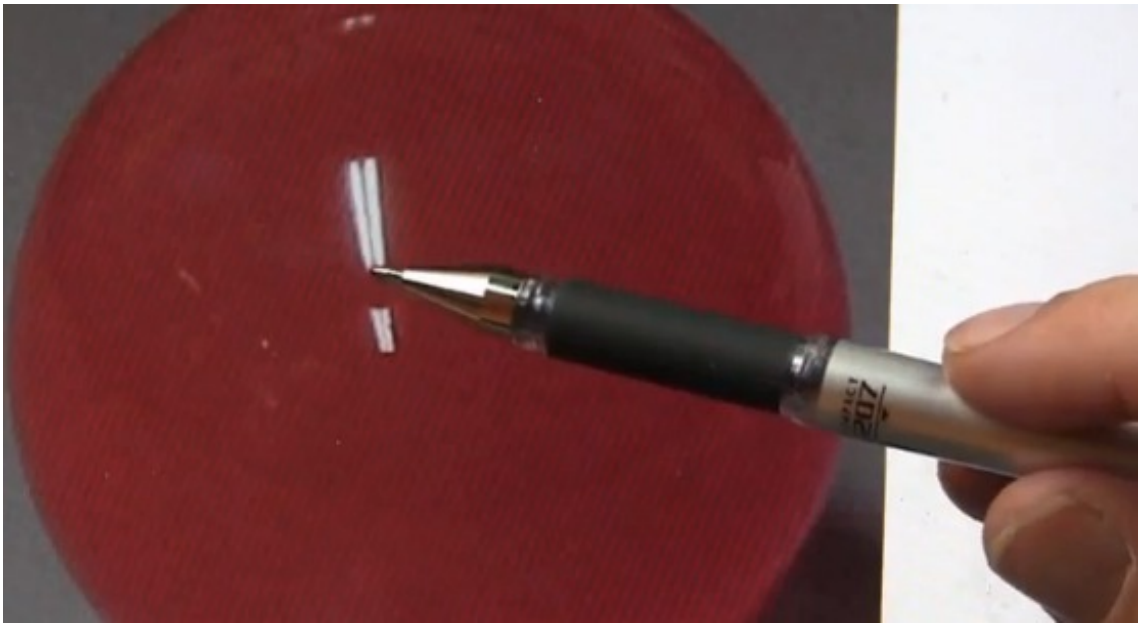


Fig. 12.13. Specular reflection is another source of confusion for 3D image interpretation

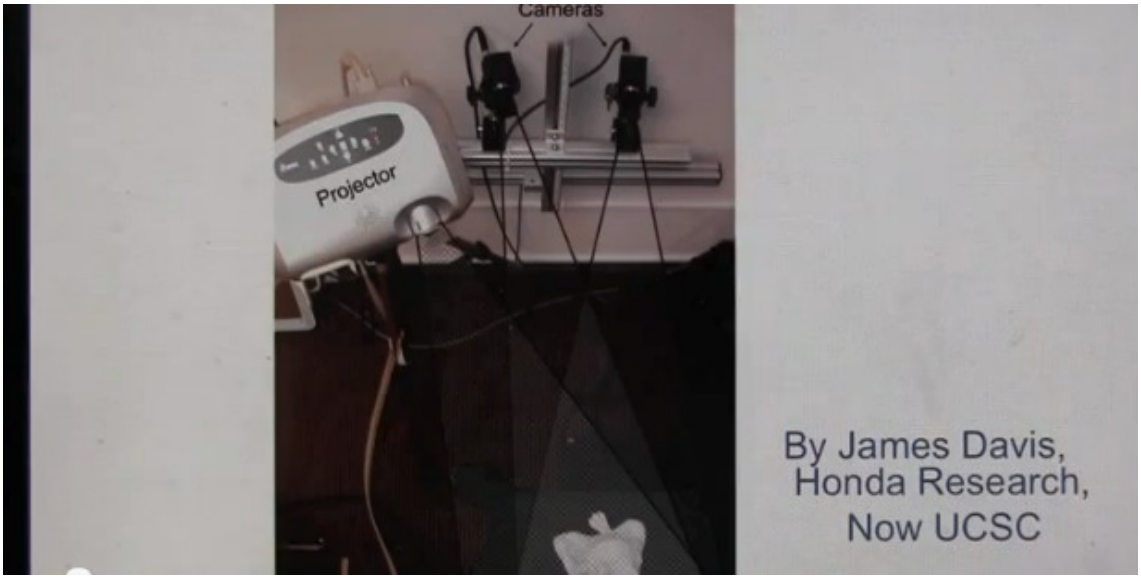


Fig. 12.14. A rig devised by James Davis at Honda research: It uses stereo cameras, but also a projector to project images to the object being scanned. The images are known, so they can be used to disambiguate the interpretation of what the cameras see.

Fig. 12.15. One pattern used by the Davis rig.

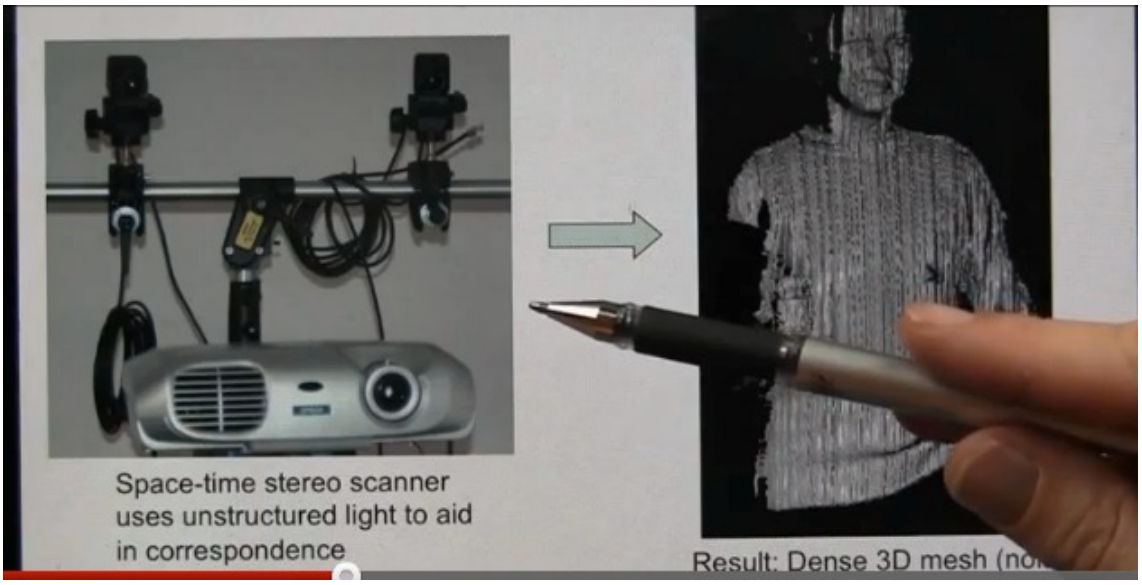


Fig. 12.16. Another pattern used by the Davis rig.

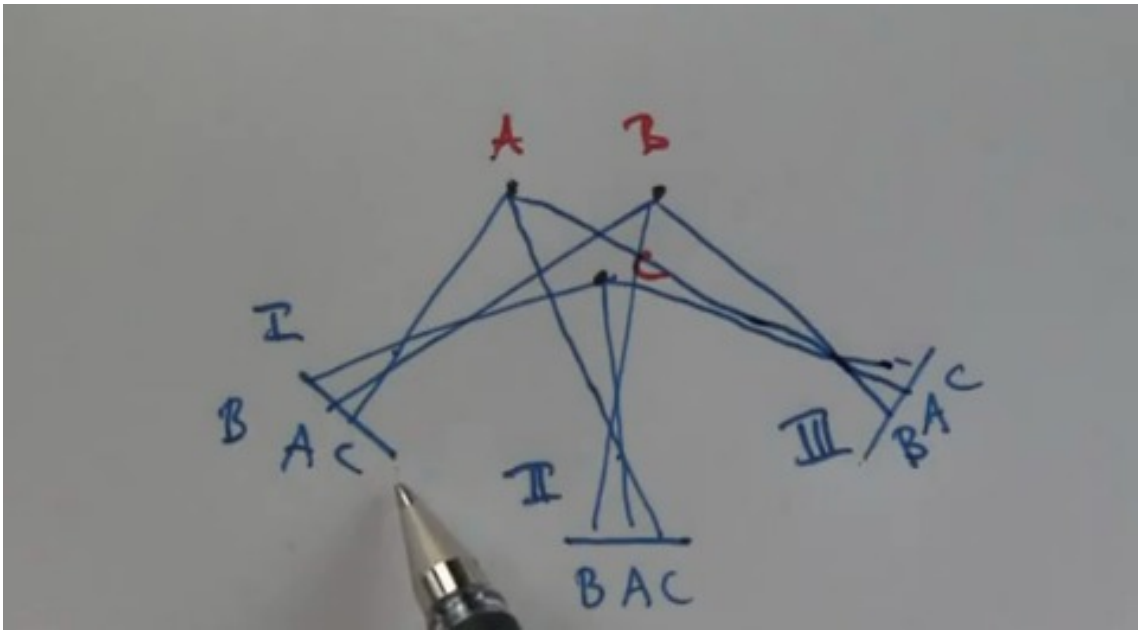


Fig. 12.17. "Structure from motion" refers to the process of combining multiple images from a single scene into a single 3D representation of the scene. The source of images can be either from the same time, but taken by multiple cameras, of from the same camera that has taken multiple images of the same scene.

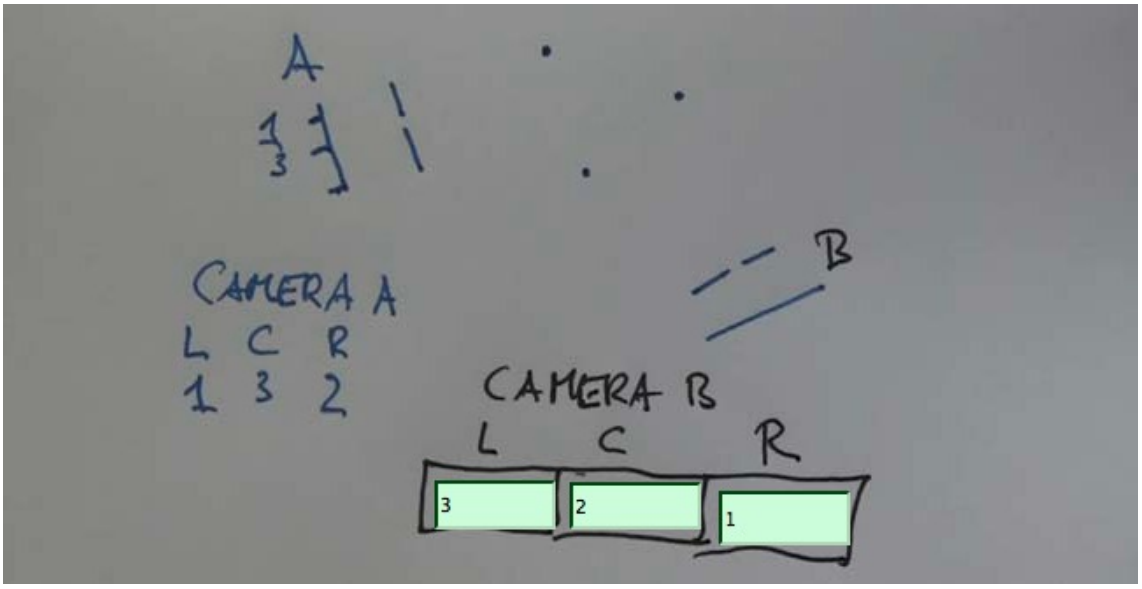


Fig. 12.18. Interpreting a scene

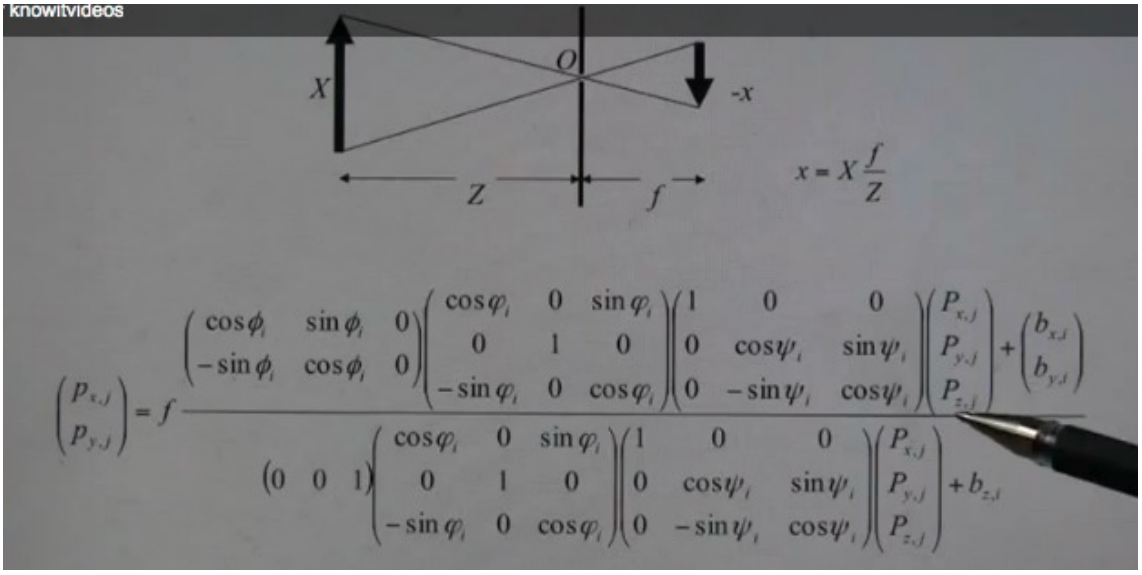


Fig. 12.19. A snapshot of an equation used in real SFM systems. The interpretation task is to minimize the error generated by fitting a model to the observed images through this equation.

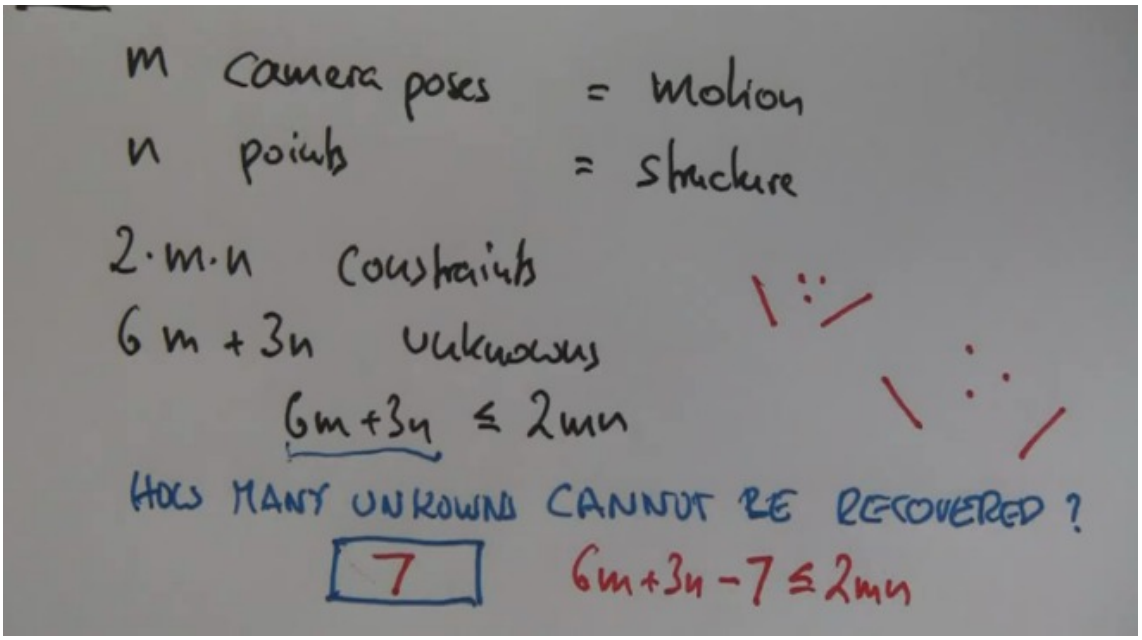


Fig. 12.20. A calculation of the number of variables necessary for structure from motion

13. Robots - self driving cars

Fig. 13.1. A car (not the winner) participating in the DARPA grand challenge competition

Self driving cars will to make cars safer and allow more people to use cars (elderly, disabled etc.). Thrun has worked most of his profesional life towards the goal of making self-driving cars. In this chapter we will learn a bit about how to make these self-driving cars :)

Thrun stated working on self driving cars in 2004 when cars were asked to drive through the mohave desert through 140 miles of desert. Really punishing terrain. Many people, including privates. Most participant failed. Some were very large. Most robots failed.

The first grand challenge was interesting. Nobody made it to the finish line. Some were large, some were small. No team made it further than five percent of the course. Some even went up in flames. The farthest one got was eight miles. In some sense this was a massive failure :-)

He started a class CS294 that became the Stanford racing team. difficult mountain terrain. After seven hours it finished the DARPA grand challenge. They were insanely proud.

Fig. 13.2. The winner, "Stanley" driving on a narrow mountain road during the DARPA grand challenge.

Later the team participated with *junior* in *DARPA urban challenge*. Junior used particle and histogram filters relative to a map of the environment. It was able to detect other cars and determine which size they were.

Google has a self driving car and drives as fast as a Prius can go.

13.1. Robotics

Robotics is the science of bridging the gap between sensor data and actions. In perception we get sensor data and use that to estimate some internal state. Usually the process is recursive (or iterative) called a *filter*. The *Kinematic state* is the orientation of the robot, but not really the speed at which it is moving. An idealized roomba will have three dimensions. The dynamic state contains all of the kinematic state pluss velocities. Thrun likes to have a forward speed and a yaw rate.

The car Thrun has made he can loate the car with about 10 cm resolution.

Fig. 13.3. "Junior", the second generation of self-driving car that participated in the DARPA urban challenge

Monte carlo localization. Each particle is a three dimensional vector (six dimensional representation). The particle filter has two steps, a prediction step and a XXX step. We assume a robot with two wheels with differential drive. It's about as complex as a car. The state is given by a vector $[a, y, \theta]^T$, and to predict the state we need to have a simple eqeuatio to describe what the vehicle does as time progresses a Δt. v is the velocity, and ω is the turning velocity (the differential of its wheels):

$$\begin{aligned} x' &= x + v \cdot \Delta t \cdot \cos\theta \\ y' &= x + v \cdot \Delta t \cdot \sin\theta \\ \theta' &= \theta + \omega \cdot \Delta t \end{aligned}$$

This is an approximation to the actual movement, but it is good enough for most practical applications in robotics.

In monte carlo localization we don't add exactly, we add noise. A single particle gives a set of particles. This exactly how the prediction step is done in the self-driving cars. We now have a set of prediction. The other important part of a particle filter is the *measurement step*. Usually we will let particles survive in proportion to the amount that they are consistent with measurements.

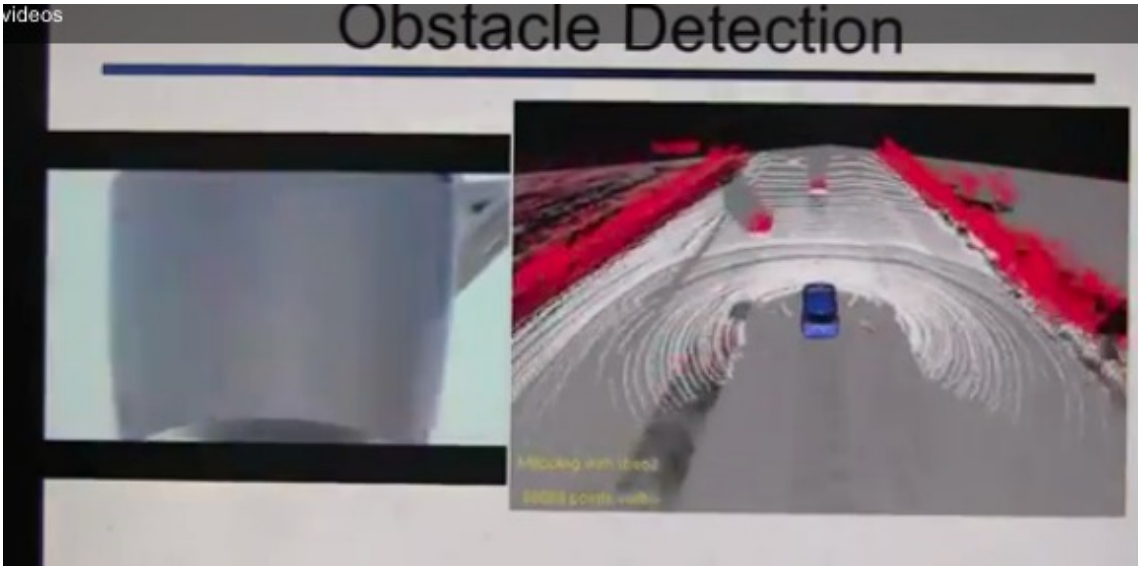


Fig. 13.4. One view of the world as seen through "Junior"'s sensors.

We translate the measurements to probabilities, and then we can do the filtering.

We then apply the resampling and typically draw the particles based on the normalized weights and repeat.

13.2. Planning

The planning problem is at multiple levels of abstraction. At street level, this is basically an mdp and/or search problem. We can use the usual MDP algorithm but we need to take the direction of the car into the equation. Red is high cost, green is low. This is value iteration appplied to the road graph.

Self driving cars use this a lot.

13.3. Robot path planning

Now the world is continous :-) The fundamental problem is that A^* is discrete and we need continous plans in the real world. So the problem is: Can we find a

Fig. 13.5. An illustration of the probabilistic localization system used by Junior. It uses both particle filters and histogram filters.

modification of A^* that gives us probably correct, continous plans? The key is to modify the state transition. By using an algorithm called "hybrid A^*", we can get correct solutions, but not necessarily the shortest ones. Every time we expand a grid cell we memorize the state of the vehicle (x', y', θ'). The state is creating the map as it moves. The plan is smoothed using a quadratic smoother.

Fig. 13.6. Google's self driving car (a Toyota Prius)

Fig. 13.7. A road scene as it is interpreted by a car. Notice the other vehicles (and a person) that is denoted by framed boxes.

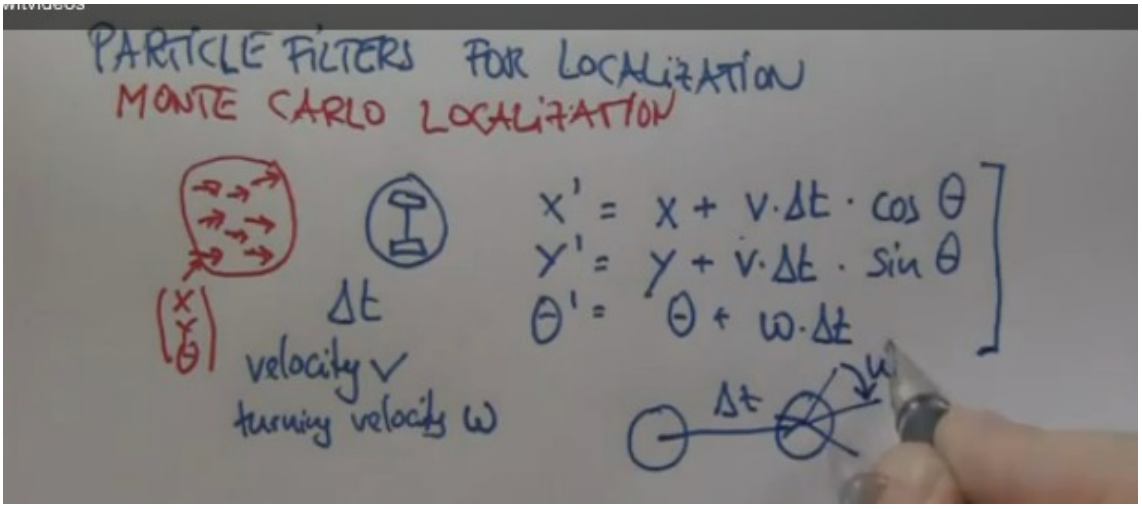


Fig. 13.8. localizationfilter

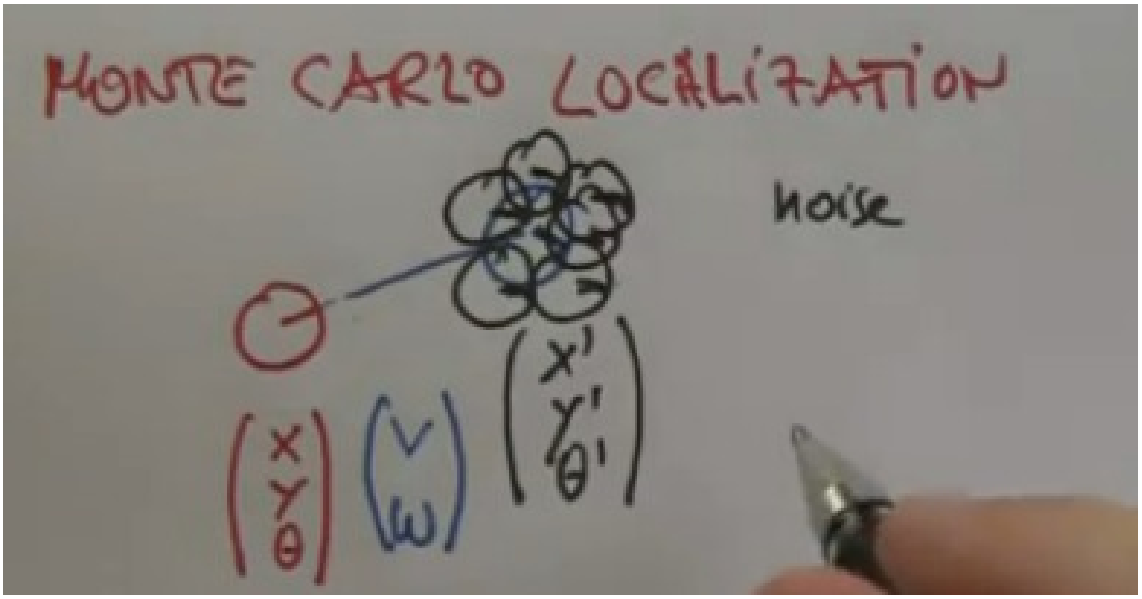


Fig. 13.9. Monte carlo prediction of location using a particle filter

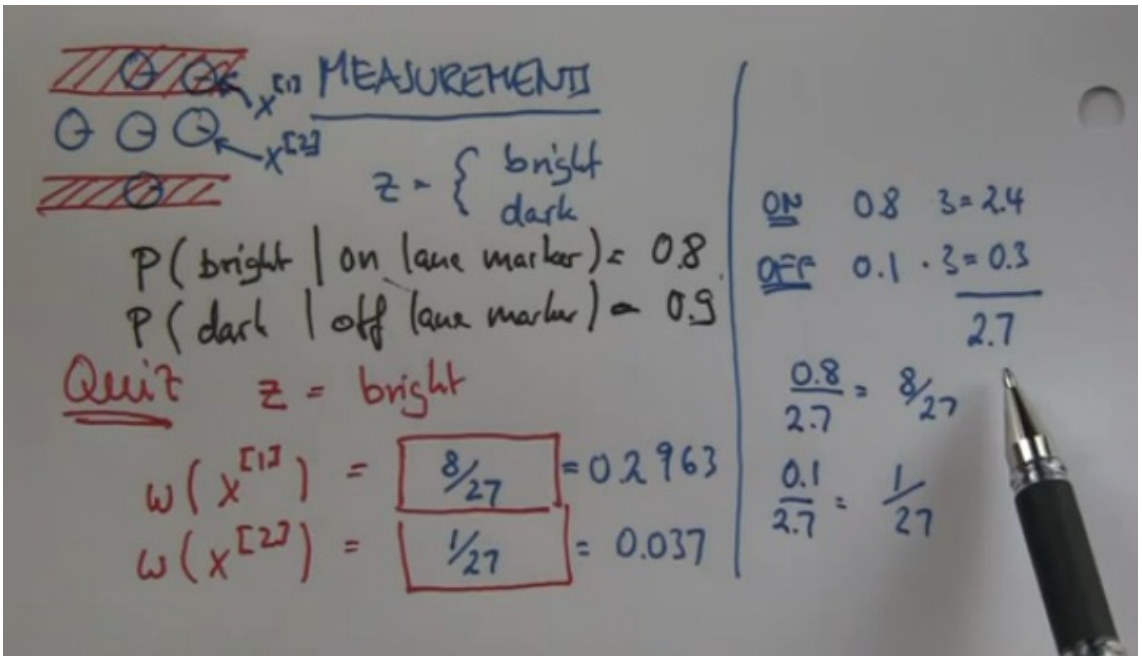


Fig. 13.10. Calculating the next location using a particle filter

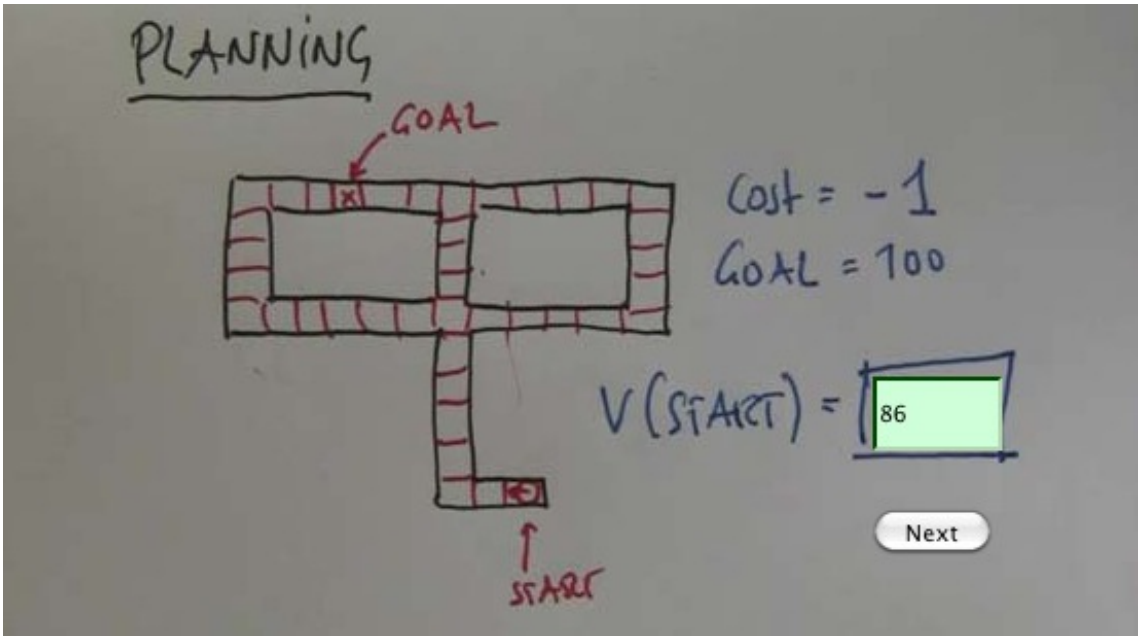


Fig. 13.11. Idealized planning of how to drive to some location

Fig. 13.12. Example of a high level plan for reaching a goal

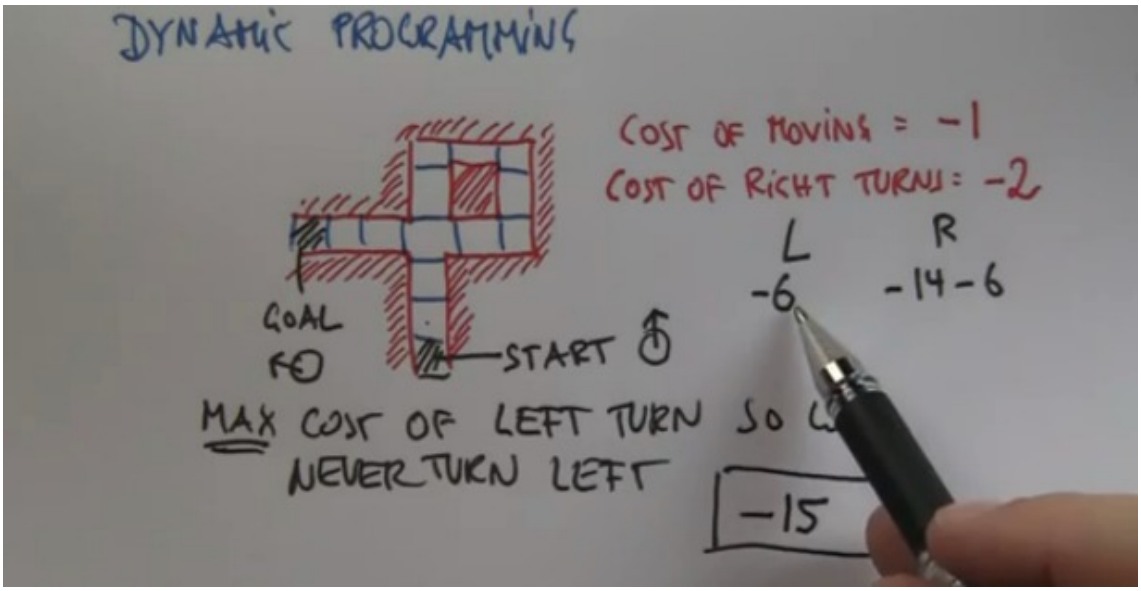


Fig. 13.13. Using dynamic planning in a gridworld type setting

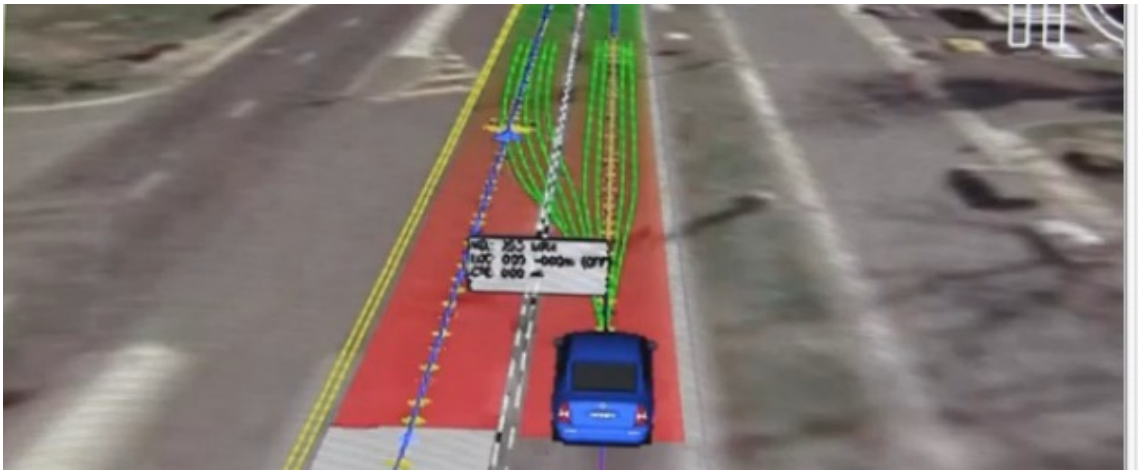


Fig. 13.14. Using dynamic programming to plan when to shift lanes

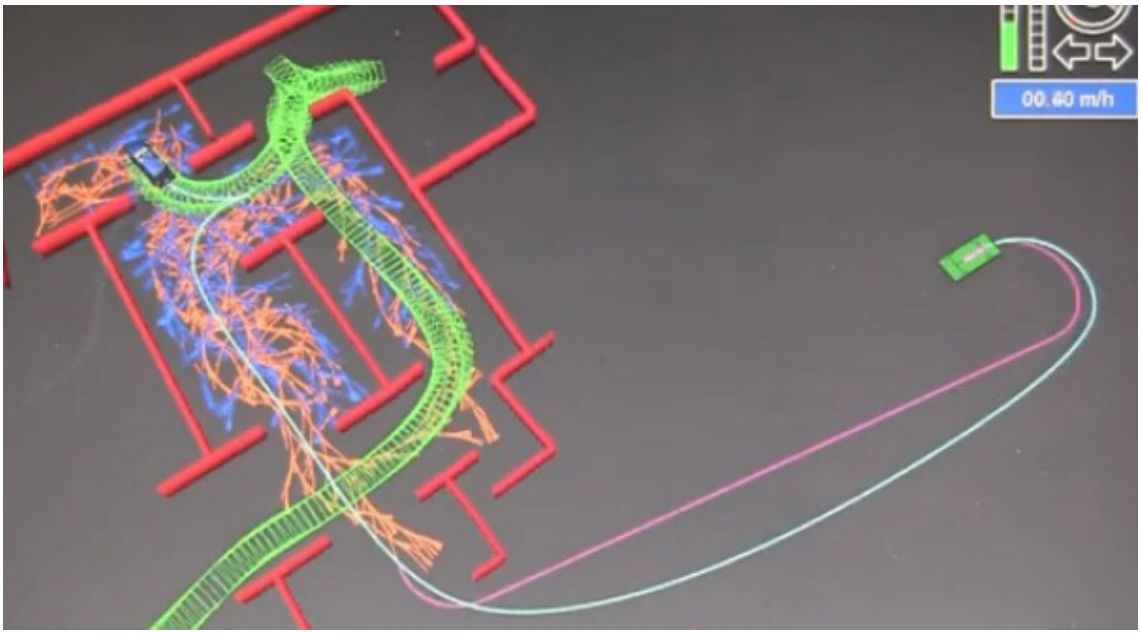


Fig. 13.15. Using the hybrid A^* algorithm to plan travels through a maze

Fig. 13.16. Navigating on a parking lot

14. Natural language processing

NLP is interesting philosophically since humans define themselves by our ability to speak with each other. It's something that sets us apart from all the other animals and all the other machines. Furthermore we would be like to talk to our computers since talking er natural. FInally it's in terms of learning: Lots of human knowledge is written down in form of text, not formal procedures. If we want our computers to be smart we need them to be able to read and understand that text.

14.1. Language models

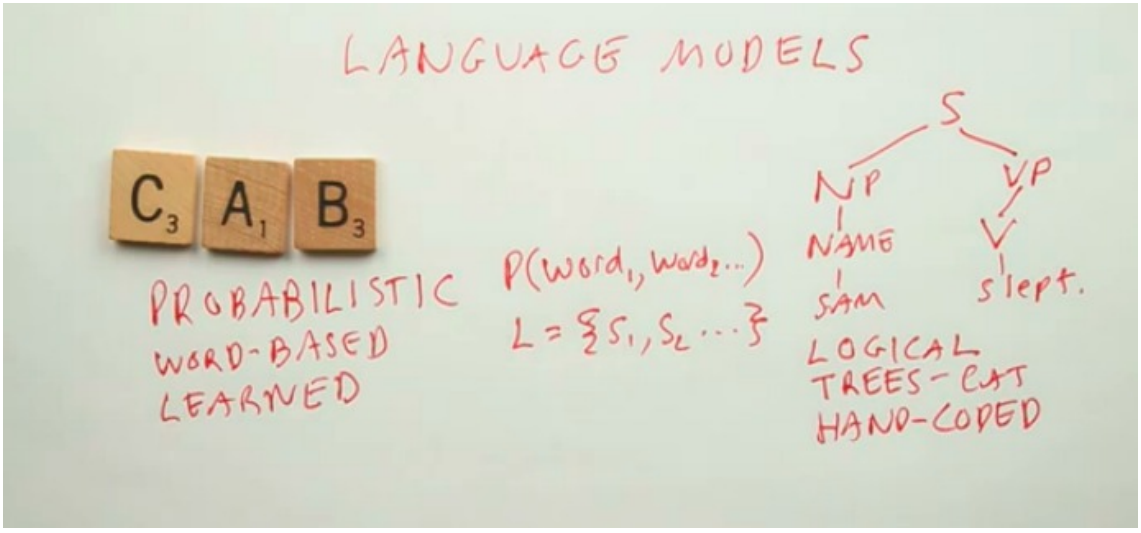


Fig. 14.1. Models for language: Probabilistic models v.s. syntactic trees.

Historically there has been two models. The first has to do with sequences of words, and it's the surface words themselves and it's probabilistic. We deal with the actual words we see. These models are primarily learned from data.

The other model deals with syntax trees. They tend to be logical rather than probabilistic. We have a set of sentences that defines the language. A boolean distinction, not probabilistic. It's based om trees and categories that don't actually occur in the surface form (e.g. an agent can't directly observe that the word "slept" is a verb). Primarily these types of methods has been hand coded. Instead we have had experts like linguists that has encoded these rules in code. These rules are not hard-cut, one can imagine having trees and probabilistic models of them, methods for learning trees etc.

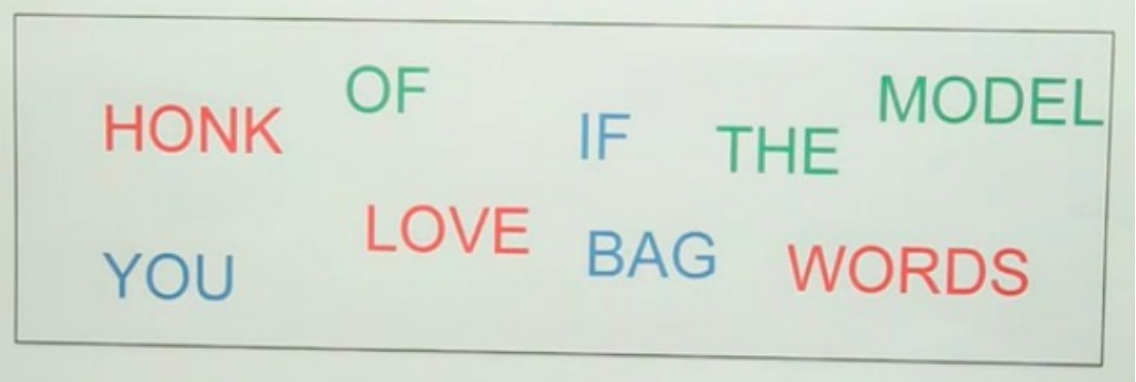


Fig. 14.2. A bumper sticker illustrating the "bag of words" model

The *bag of words* model is a probabilistic model where the words don't have sequence information. It's still useful for a lot of interesting probabilistic algorithms.

We can move on from the bag of words models into models where we do take sequences into account. One way of doing that is to have a sequence of words, introduce some notation for it, and desscribe the probability of that sentence as a conditional probability over the sequence.

$$\begin{aligned} P(w_1, w_2, \ldots, w_n) &= 6P(W_{1:n}) \\ &= \Pi_i P(w_i | w_{1:i-1}) \end{aligned}$$

If we use the *markov assumption* we make an assumption of only caring about going back k steps. Using this assumption we get

$$P(W_i | W_{i:i-1}) \approx P(W_i | W_{i-k:i-1})$$

Furthermore there is something called the *stationarity assumption* which is that $P(W_i|W_{i-1}) = P(w_j|w_{j-1})$

This gives us all the formalism we need to talk about these *probabilistic word sequence models*. In practice there are many different tricks. One of these tricks is *smoothing*. There will be a lot of counts that are small or zero, so using *laplace smoothing* or some other smothing technique is often useful. Also there are times when we wish to augment these models with other data than words, e.g. who the sender is, what time of day the message was made. There are cases where we wish to include context. We may wish to consider other things than words themselves. We may wish to use if the word "dog" is used as a noun in a sequence. That is not immediately observable, so it's a *hidden variable*. Also we may wish ti think at the phrase "New York City" as a single phrase rather than three different words. Also we may wish to go smaller than that and consider single letters. The type of model we chose depend of the application, but it is based on the idea of a probabilistic model over sequences, as described above.

Fig. 14.3. A plot showing the frequency of the search terms "ice cream" and "full moon" as seen by the Google search engine. We can see high regularity. The spikes for the ice cream search term was due to the introduction of the "ice cream sandwich" software release of the Android operating system.

By observing language use we can predict things. Norvik displays the pattern of use of the words "full mooon" and "ice cream" we can see that the search for "ice cream" has a 365 day cycle, and that the search for the term "full moon" has a

cycle corresponding to the lunar cycle. The blip on the right is caused by the fact that Google introduced a version of the *Android operating system* called *Ice Cream Sandwich* and that caused the blip to happen. That lasted a few days then it disappeared.

Some things we can do with language: Classification, clustering, input correlation, sentiment analysis, information retrieval, question answering, machine translation and speech recognition.

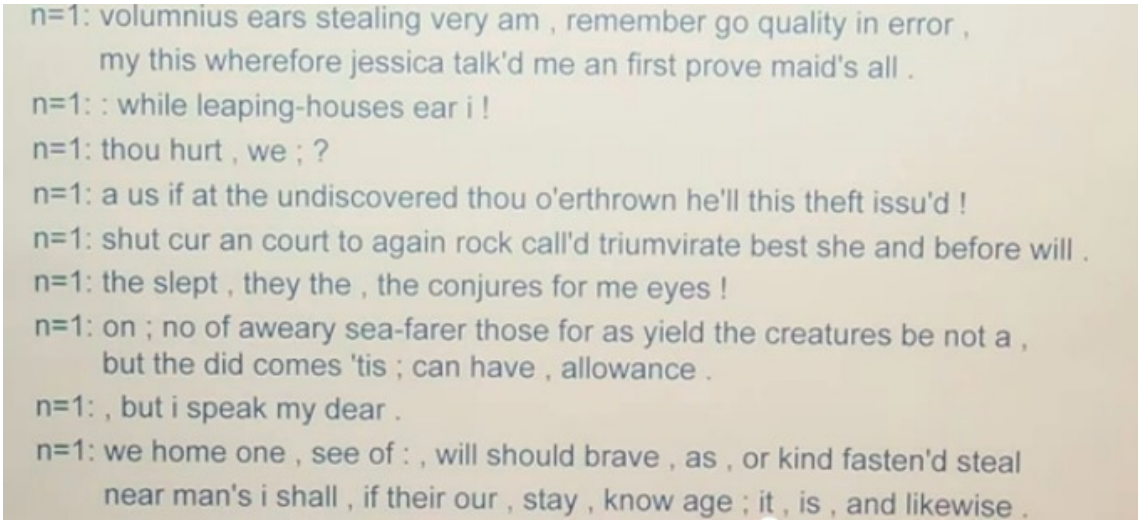


Fig. 14.4. Text generated based on N-grams for N=1 and a corpus of Shakespeare texts.

When considering ngram models, at n= 1 it's just a jumble of words. At n=2 the sentences make make locally sense, at n=3 the sentences even make a bit sense globally (see figs ??,??,??, ??). At n=4 w see even longer structures that makes sense. Some sentences that are not in Shakespeare but some that are there too.

Btw, in the ngramtest (fig ??) I only exchanged nr. 3 and nr. 7. A bit weird that I should do so well.

Single letter n-grams is actually quite useful to figure out which language a text originates in.

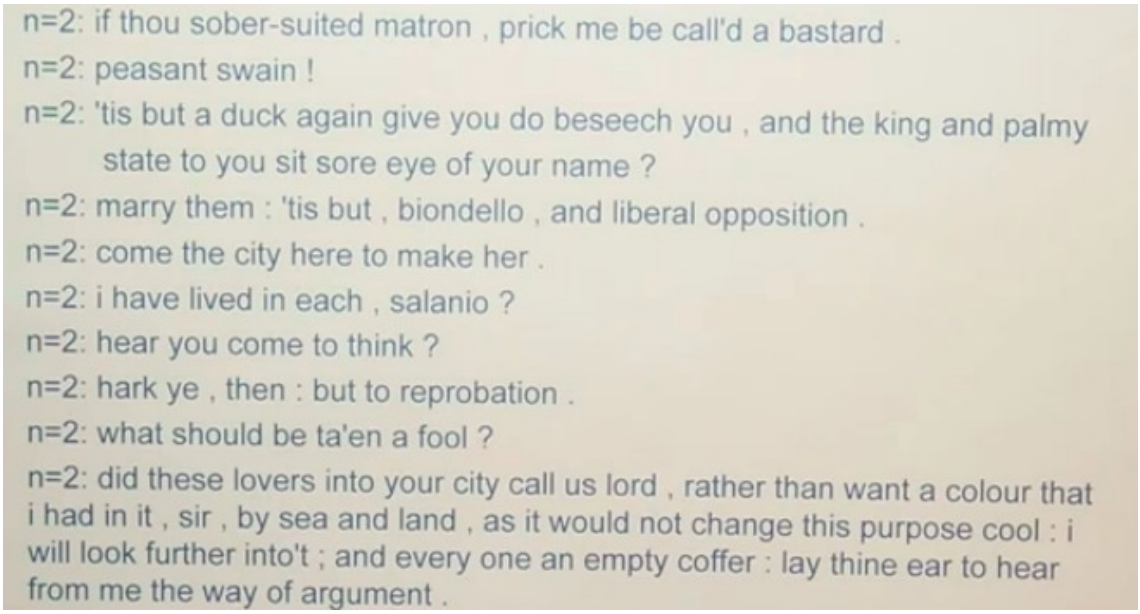


Fig. 14.5. Text generated based on N-grams for N=2 and a corpus of Shakespeare texts.

14.2. Classification into language classes

Classification into semantic classes. How to do that? We can look memorize common parts ("steve," "andy" for names, "new" and "san" for places, what's the last character, the last two characters etc.) we can then throw those into a classification algorithm :-)

One surprise is that the "gzip" command is a pretty good classifier. Compression and classification is actually pretty related fields.

14.3. Segmentation

In Chinese words don't have punctuations that separete them out. In english text we do. However in URL sequences and speech we certainly have that. Segmentation is an important problem. The best segmentation S^* is the one that maximizes the joint probability of

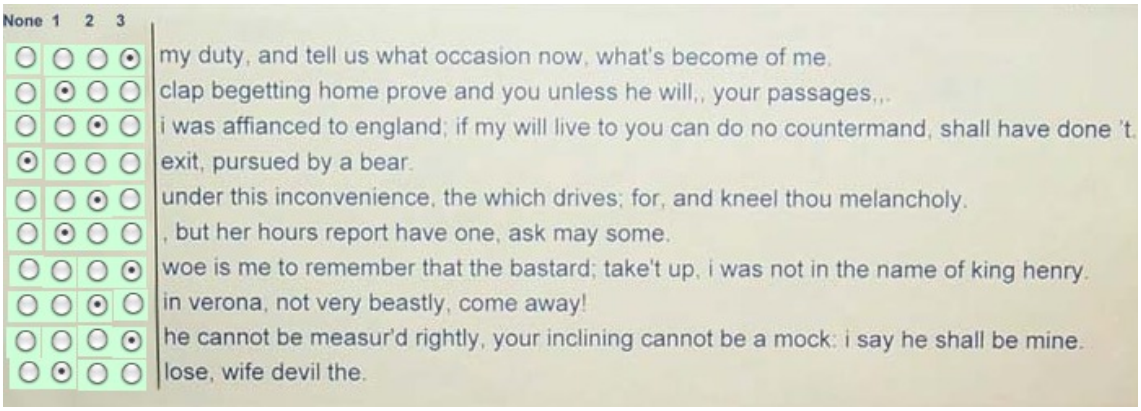


Fig. 14.6. A quiz where I was supposed to guess which N-gram model had generated the various texts. I only got two wrong, no. 3 and no. 7. Cool :-)

$$S^* = \max P(w_{1:n}) = \max \Pi_i p(w_i|w_{i:i-1})$$

We can approximate this by making the naive Bayes assumption and treat each word independently so we want to maximize the probability of each word regardless of which words came be for it or comes after it.

$$S^* \approx \max P(w_i)$$

That assumption is *wrong* but it will turn out to be good enough for learning. Now, there are a lot of possible segmentations. For a string of length n there are $2^{(n-1)}$ possible segmentations. That's a lot. We don't want to enumerate them all. That is one of the reason why making the naive Bayes assumption is so useful, since it means that we can consider each word one at a time.

If we consider the string as a concatenation of the first element "f" and the rest "r", the best possible segmentation is:

$$S^* = \text{argmax}_{s=f+r} p(f) \cdot P(S^*(r))$$

The argmax algorithm implemented in Python

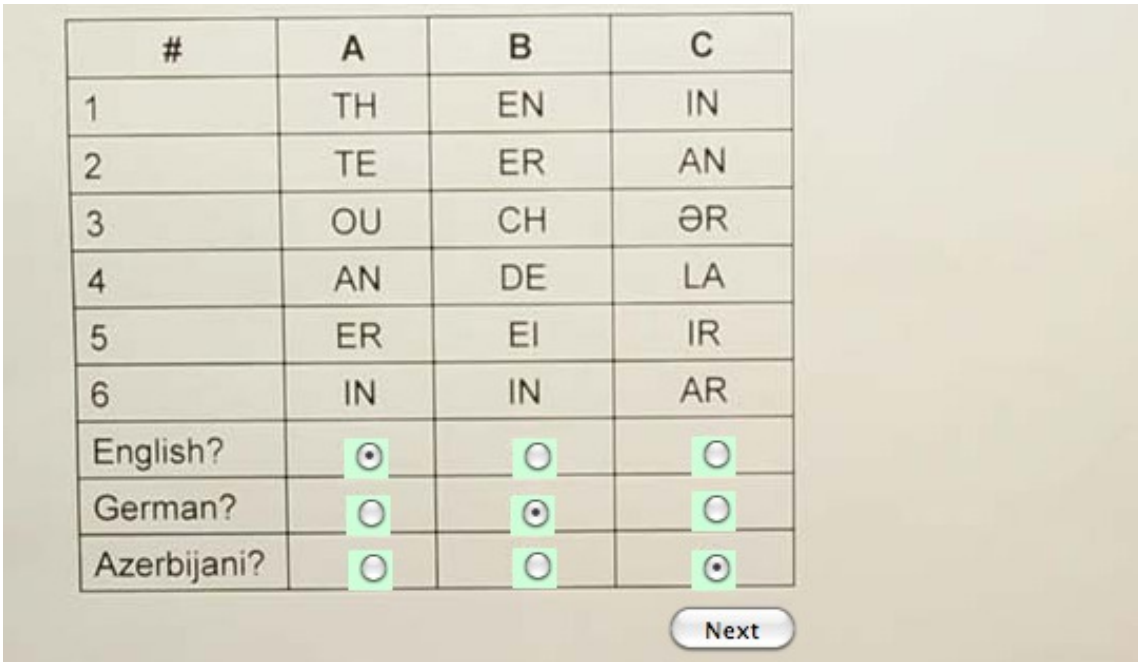


Fig. 14.7. Determining which language one sees based on individual character 2-grams

The segmentation of the rest is independent of the segmentation of the first word. This means that the Naive Bayes assumption both makes computation easier and makes learning easier, since it's easy to come up with unigram probabilities from a corpus of text. It's much harder to come up with n-gram probabilities since we have to make a lot more guesses and smoothing since we just won't have the counts for them.

How well does the segmentation algorithm work? In figure **??** there are some examples of the naive algorithm at work after being trained on a four million (or was it billion) words corpus. Some texts are not so bad, but others are actually not that good. Making a Markom assumption would improve the performance.

14.3.1. Spelling corrections

We are looking for the best possible correction, defined as:

$$\begin{aligned} c^* &= \text{argmax}_c P(c|w) \\ &= \text{argmax}_c p(w|c)p(c) \end{aligned}$$

People	Places	Drugs
Steve Jobs	San Francisco	Lipitor
Bill Gates	Palo Alto	Prevacid
Andy Grove	Stern Grove	Zoloft
Larry Page	San Mateo	Zocor
Andrew Ng	Santa Cruz	Plavix
Jennifer Widom	New York	Protonix
Daphne Koller	New Jersey	Celebrex
Noah Goodman	Jersey City	Zyrtec
Julie Zelinski	South San Francisco	Aggrenox

Fig. 14.8. Classifying texts in to word classes.

(in Bayes rule there is a correction factor, but that turns out to be equal for all corrections so it cancels out). The unigram statistics $p(c)$ we can get from the corpus. The other part, the probability that somebody typed the word "w" when they really ment "c" is harder, since we can't observe it directly. Perhaps we can look at lists of spelling correction data. That data is more difficult to come by. Bootstrapping it is hard. However there are sites that will give you thousands (not billions or trillions) of misspellings.

However, with few examples, we'll not get enough possible error words. Instead we can look at lettor-to-lettor errors. That means we can build up probability tables for edits (inserts, transpositions, deletions, additions) from words, and we can us these as inputs to our algorithms instead. This is called *edit distance*. We can then work with the most common edits and from that get conditional probabilities for the various misspellings. Just looking at unigram posibilities actually gives 80 percent accurate spelling correcter. With markov assumptions we can get up into the high ninties.

14.4. Software engineering

HTDIG is a great search engine and tries to figure out if words sound equal. (www.htdig.org) (looking at the htfuzzy). It's hard to maintain code that's specific

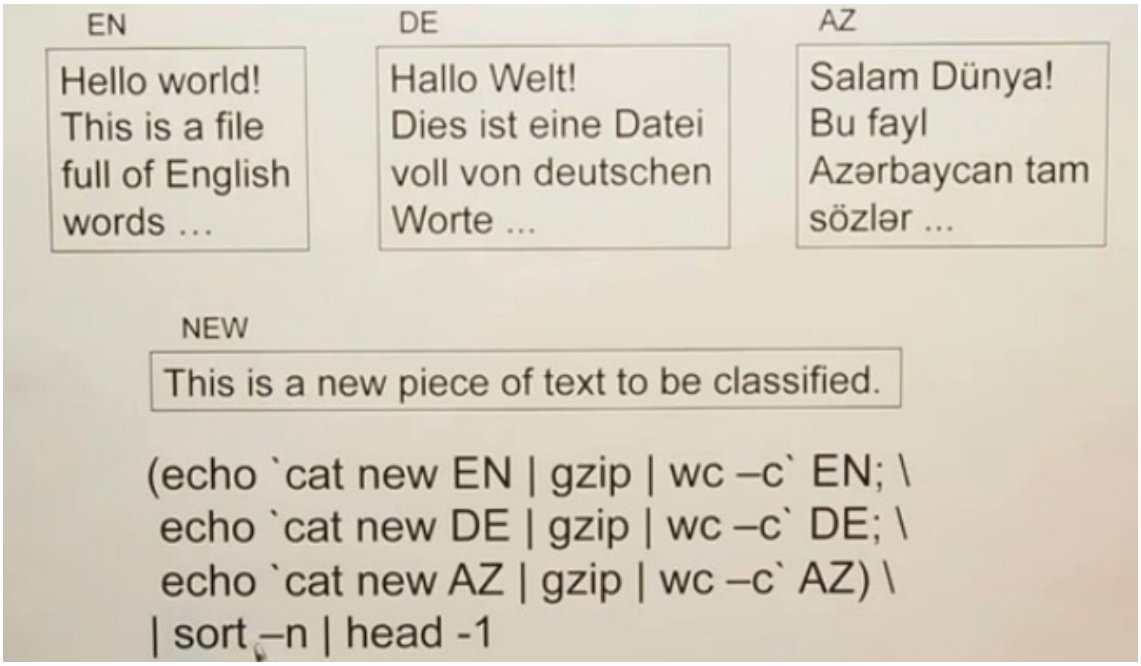


Fig. 14.9. Using the gzip compression program as a classifier algorithm

for the english language if you want to make things work for many languages. You may need to understand the pronounciation rules for all the langages. If you were working with a probabilistic model, you would only need a corpus of words and spelling errors (e.g. through spelling edits), it is easier software engineering. So you could say that machine learning over probabilistic models is the ultimate in agile programmin

Fig. 14.10. Segmenting characters into words is not very important in natural language processsing, but it certainly is importan in optica character recognition and in speech recognition.

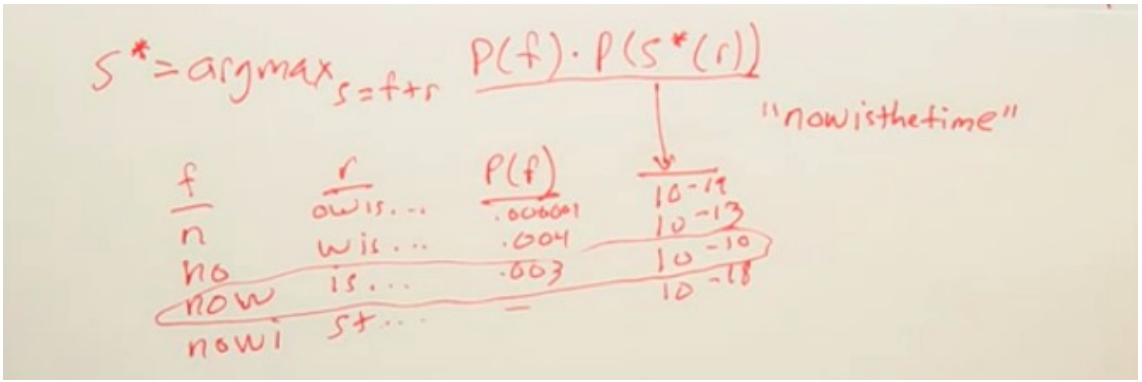
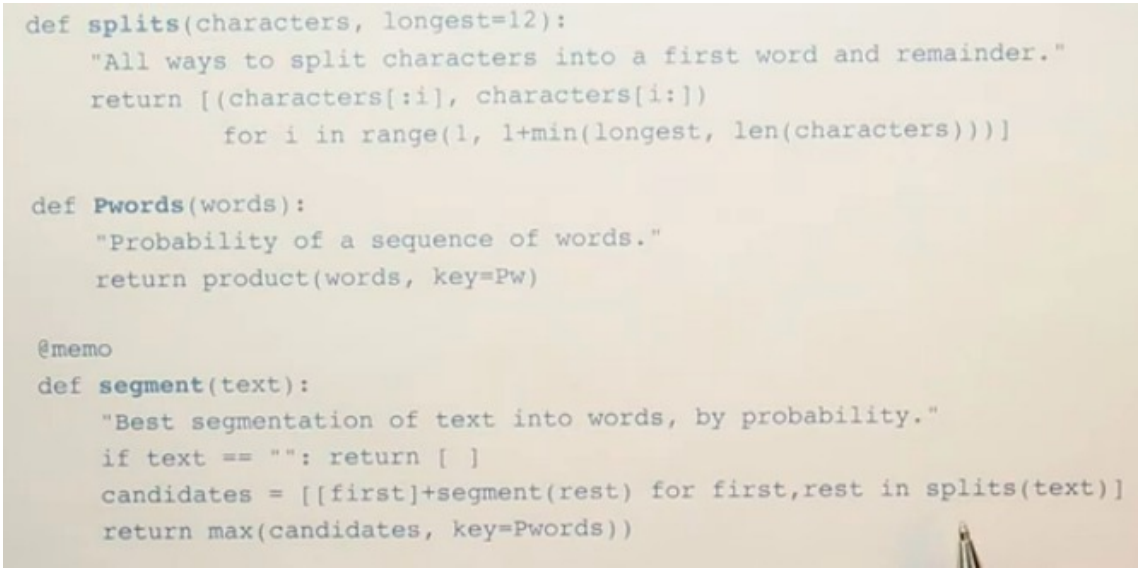


Fig. 14.11. The idea behind the argmax approach, is to interpret the sequence of characters as the word that gives the highest probability of the characters forming a word. Got that?



```
def splits(characters, longest=12):
    "All ways to split characters into a first word and remainder."
    return [(characters[:i], characters[i:])
            for i in range(1, 1+min(longest, len(characters)))]

def Pwords(words):
    "Probability of a sequence of words."
    return product(words, key=Pw)

@memo
def segment(text):
    "Best segmentation of text into words, by probability."
    if text == "": return [ ]
    candidates = [[first]+segment(rest) for first,rest in splits(text)]
    return max(candidates, key=Pwords))
```

Fig. 14.12. argmaxinpython

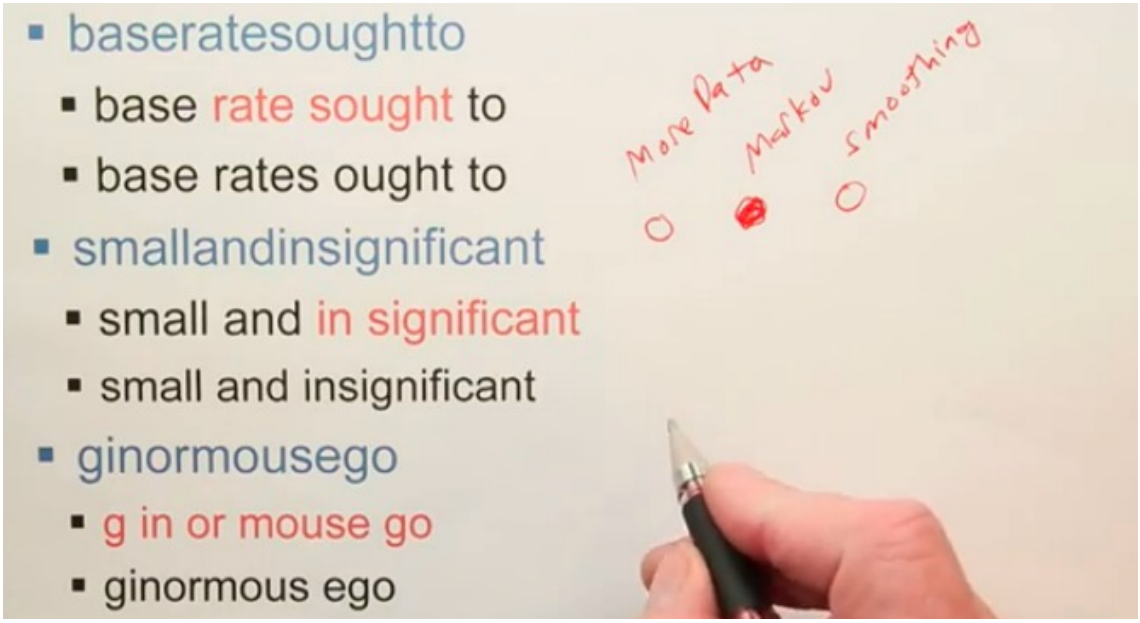


Fig. 14.13. Using a corpus of four million words, these are some segmentations that can be found

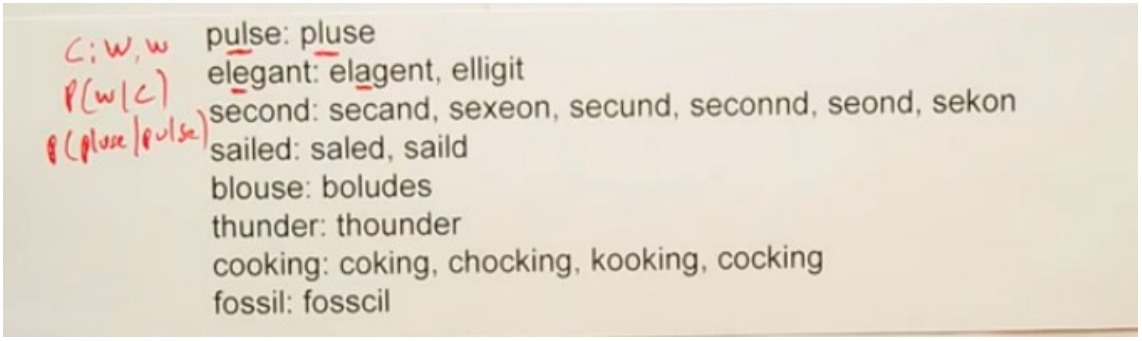


Fig. 14.14. One can measure the distance between words through "edit errors", how many transpositions, omissions, insertions etc. are necessary to transform one word into another. Words that are close in edit distance may be misspellings of each others.

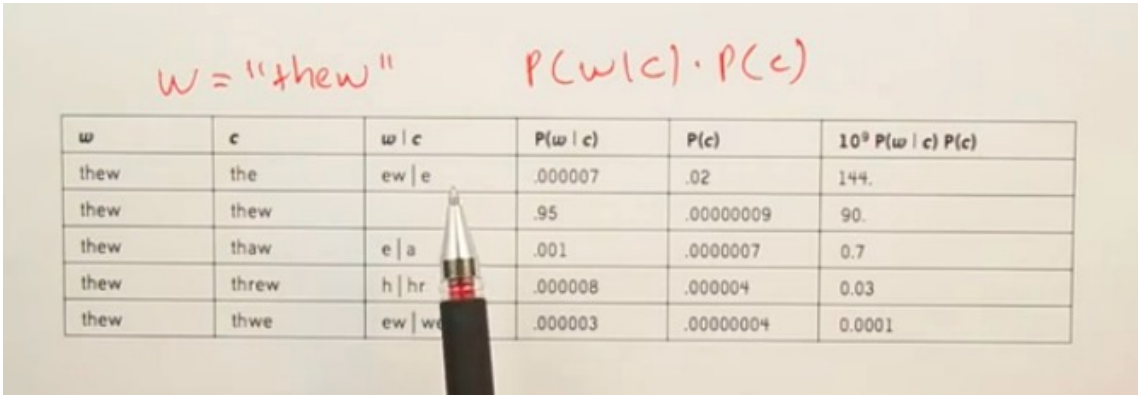


Fig. 14.15. A probility table of spellings and some misspellings. The frequencies are just made up by Norvig, but they are plausible.

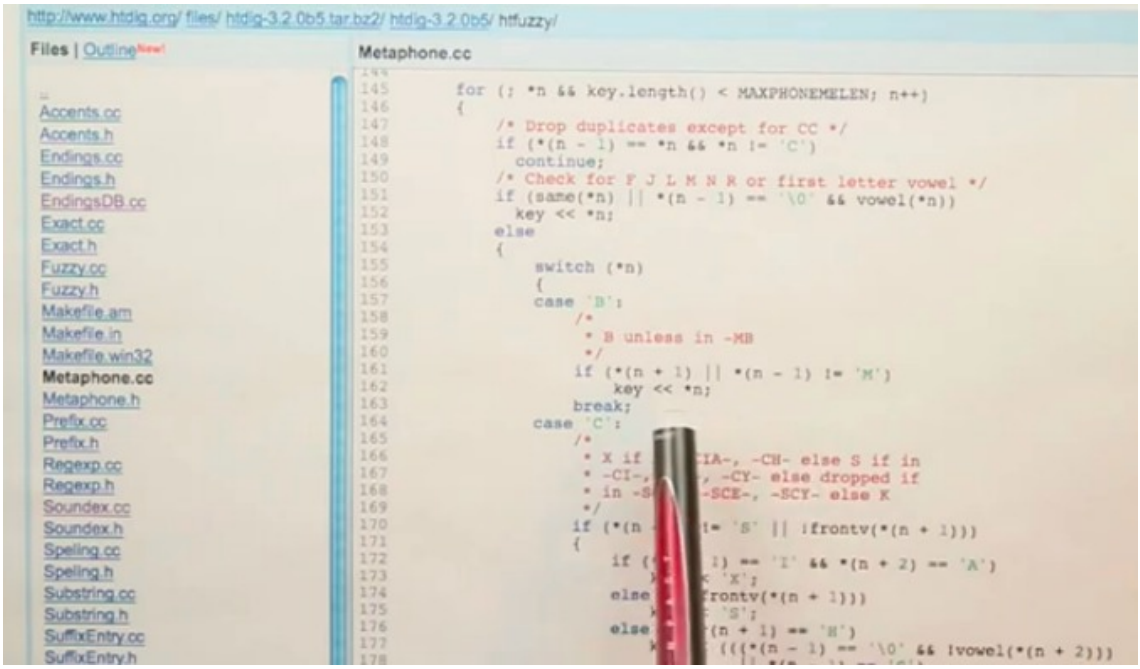


Fig. 14.16. HTDIG is a search engine that uses an algorithmic approach to finding misspellings and similar words. This is a snippet of its source code.

15. Natural language processing

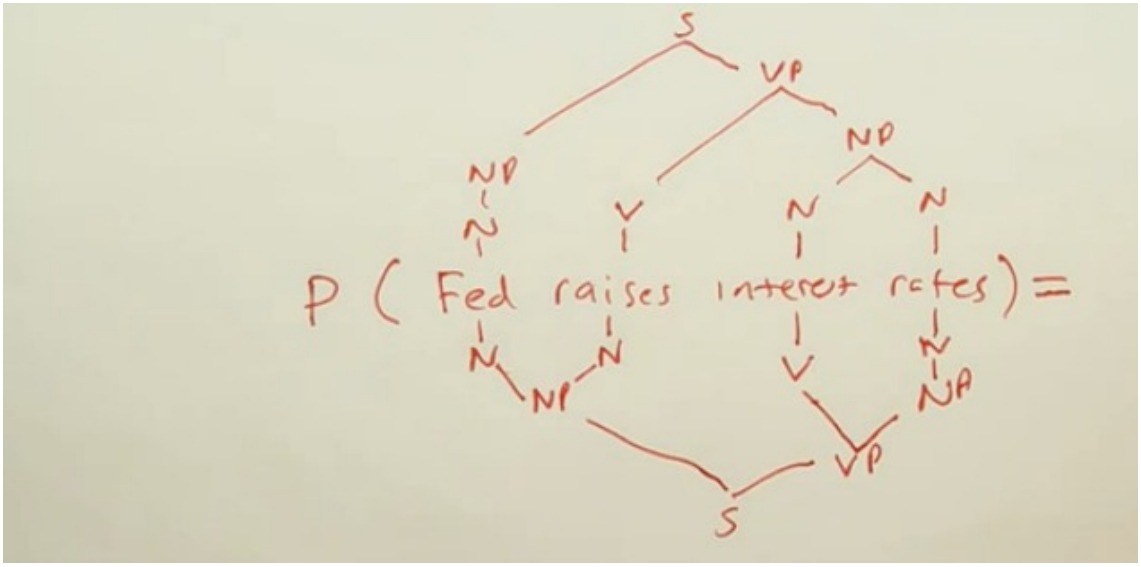


Fig. 15.1. A setence can be parsed using a grammar. For human languages, unfortunately, it is uncommon to have an unique parse for a sentence. The common case is some degree of ambiguity, hence reduction of ambiguity is a central part of any natural language parsing.

There is another way to parse sentences, and that is based on internal grammatical structure of the sentences. However, the sentence structure is not unique. There is in general no unique parse of a sentence

The sentences comes from *grammars*. Grammars are sets of rules that define sets of symbols that forms the language. The language is defined as all the legal sentences in the language.

There are terminal words, or composite parts of the grammar. The terminal words have no internal structure (as far as the grammar is concerned).

There are some serious approches to the grammar approach, some of which are:

- It's easy to omit good parses.
- It's easy to include bad parses.

Assigning probabilities to the trees and using word associations to help figure out which parse is right are both viable approaches. Making the grammar unambiguous is not.

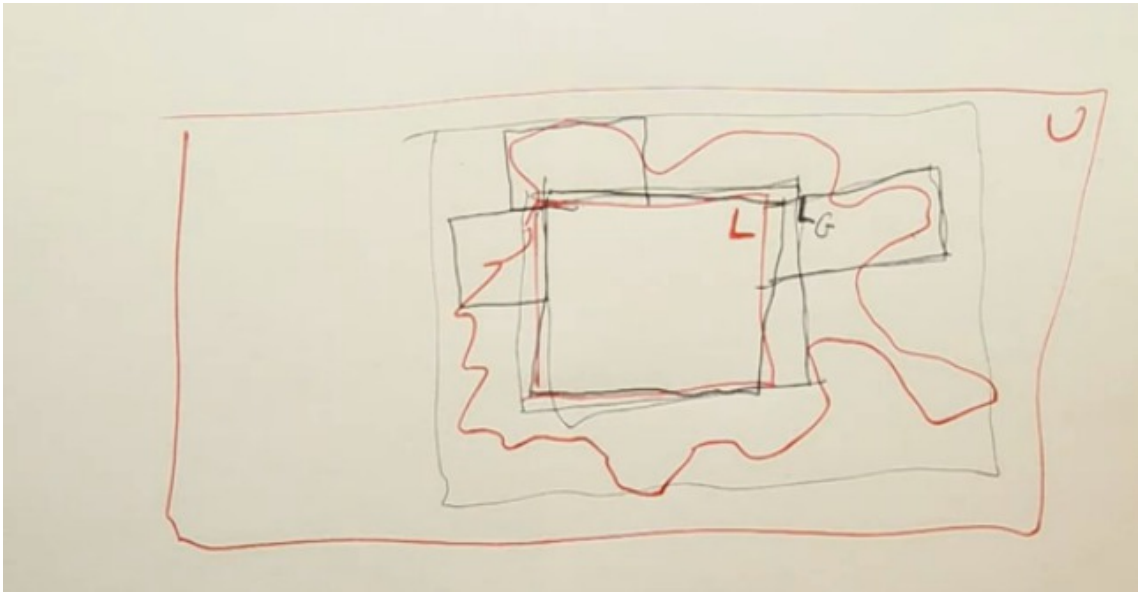


Fig. 15.2. The universe of all possible strings includes fare more strings than are valid sentences. The language as described by a simple grammar is rather small, denoted by a neat square near the center of the figure. Real human languages are messy, denoted by "ragged edges" and are not easily modelled by simple syntax models.

If the actual grammars were neat things it would be easy to write a good grammar. However that isn't true, since human languages are very messy. A grammar for a real human language is difficult to make.

15.1. Probabilistic Context Free Grammar

Writing simple context free grammars is hard, but there are some techniques that can help us. One such techinque is called *probabilistic context free grammar (Pfc)*.

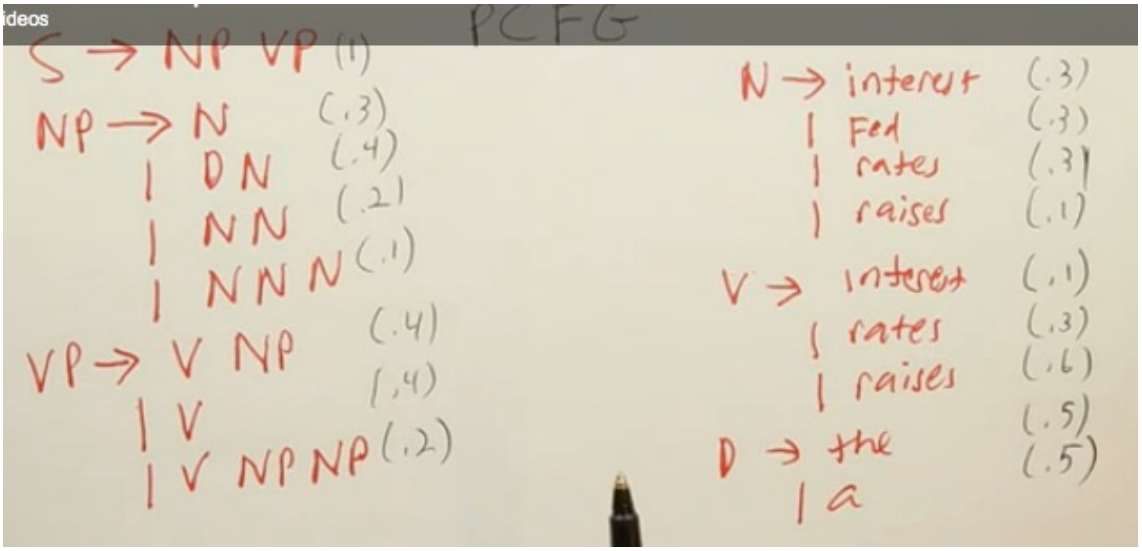


Fig. 15.3. A probabilistic context free grammar is just a context free grammar where each of the possible branches in a syntax production rule is assigned a probability, and where the sum of probabilities for a single rule sums up to one. Using argmax type argumentation, we can now disambiguate with equally legal but uniqually probable sentences

It's just to annotate the syntax rules in the grammar with probabilies like indicated in figure ??. The trick is that within every rule we add probabilities to each of the branches in the grammar, and the probabilities add up to one. The probability of the sentence as a whole is just the product of all the probabilities.

So where does the probabilities come from? Well, in the examples we've seen so far it's just Peter Norvig that has made up the numbers, but in the real world it's of course from real dta. Run parsers across huge swaths of sentences, parse them and figure out the probabilities fo the various parse rules being the correct ones. Use naturally occurring text. This is not necessarily very easy to do :-) We do need to parse the things, and check that the parses are actually the right thing. In the 1990's this was what people did in natural language processing. Manual markup. There are now corpuses that can be used. One such example is the *penn tree bank* from the *university of pennsylvania*, and it's a bank of trees :-)

As a field the AI people needed to decide on some standards for what good parse trees are, that is done, and now we can just look things up when creating probabilistic models.

The probabilities can then be used to resolve ambiguity. In the example ?? we are

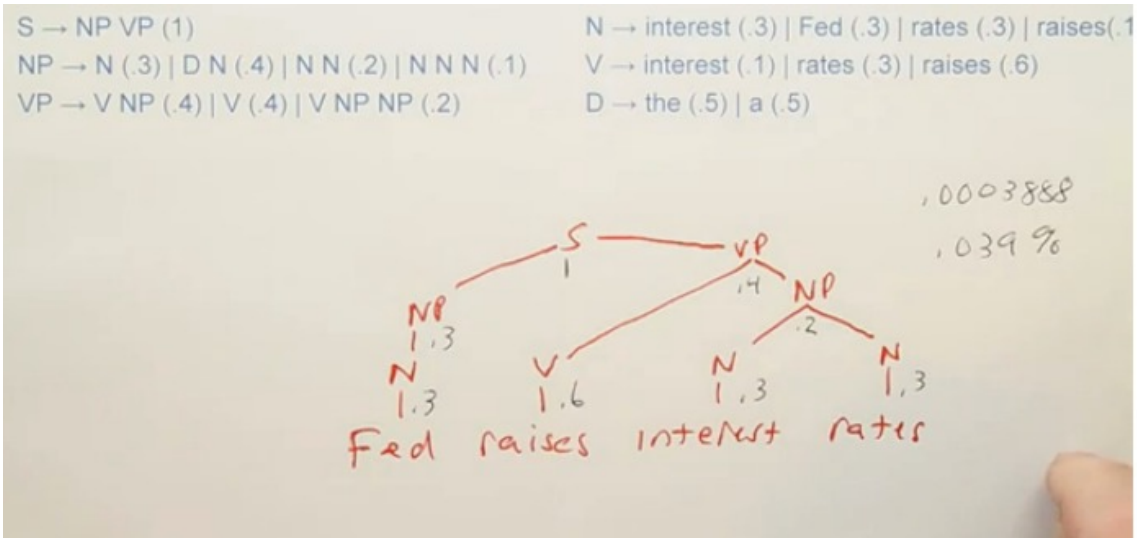


Fig. 15.4. A parse using the pcfg from figure **??**

interested in the conditional probability wether men or seeing are associated with teliscopes. When we do that we get *lexicalized probabilistic context free grammar*.

In lfpgs we condition the rules on specific words. Quake for instance is an *intransitive verb*, and that means that there are some probabilities that should be zero. However, in the real world that isn't always so. The dictionaries are too logical and too willing to give a clear cut language. The real world is messier. Various types of smoothings needs to be made. We do want to make the choices based on probabilities, and we get those probabilities from the treebanks.

We then use the real data as a basis for disambiguation.

15.2. How to parse

How do we parse? Well, we can use the search algorithms, either from the words, or from the sentences at the top. We can use both *top down* search or *bottom up* parsing. Try all the possibilities and backtrack when something goes wrong. Assigning probabilities can be done in pretty much the same way. A nice thing about this way of doing it is that making one choice in one part of the tree doesn't affect the other parts of the tree.

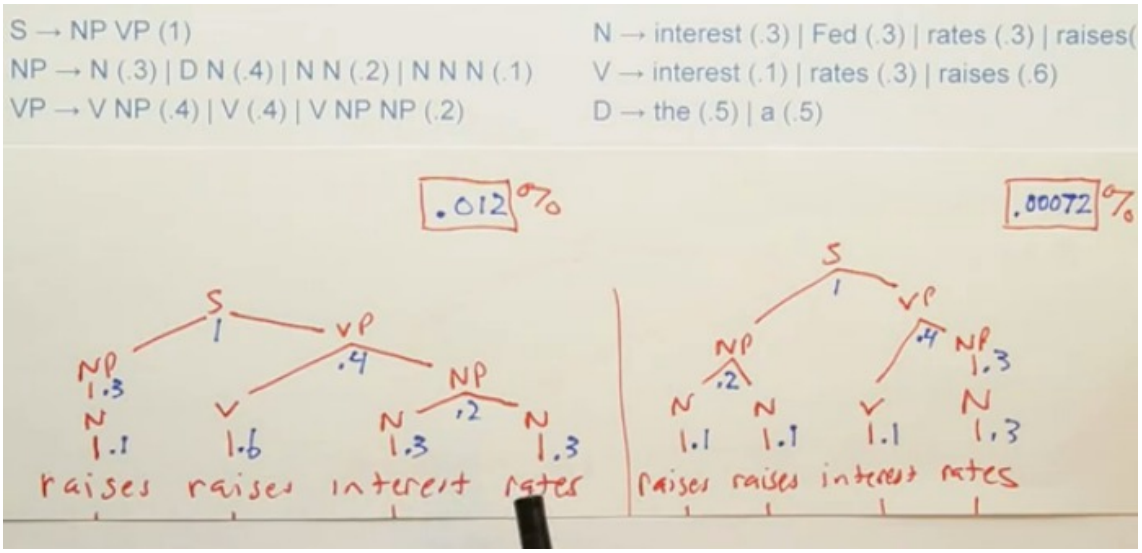


Fig. 15.5. Yet an example of probabilistic parsing.

15.3. Machine translation

We can look at word-by word translation, phrase level translations. We can consider multiple phrases at the same time, or we could go to the level of syntax. We can even go even higher and look at representations of semantics (meaning). We can in fact go up and down the pyramid depicted in fig. ?? called the *Vauquais* pyramid named after the linguist Bernard Vauquais and the translations can be done at any level or at multiple levels.

Modern translation systems do use multiple levels. In figure ?? we use three diffent levels: *Segmentation*, *translation* and *distortion* (partialy obscured by Norvig's pen in the illustration). The model translats from german to english. The segmentation is based on a database of phrases. We look for coherent phrases that occur frequently, and we get probabilities that things represents phrases, and we then get a high probability segmentation. Next is the translation step telling us which probability "Morgen" corresponds to "Tomorrow". Then there is distortion which tells us how much we should swap the phrases around. We just look at the beginning and endings of the phrases. We measure in the german, but get the indexes in the english. The distortion model is just a probability over numbers, are the things switched from the left or to the right, and is that shift probable or not. For pairs of languages where the things are not swapped very much there will be a high probability mass under zero distortion, and low probabilities elsewhere. For other

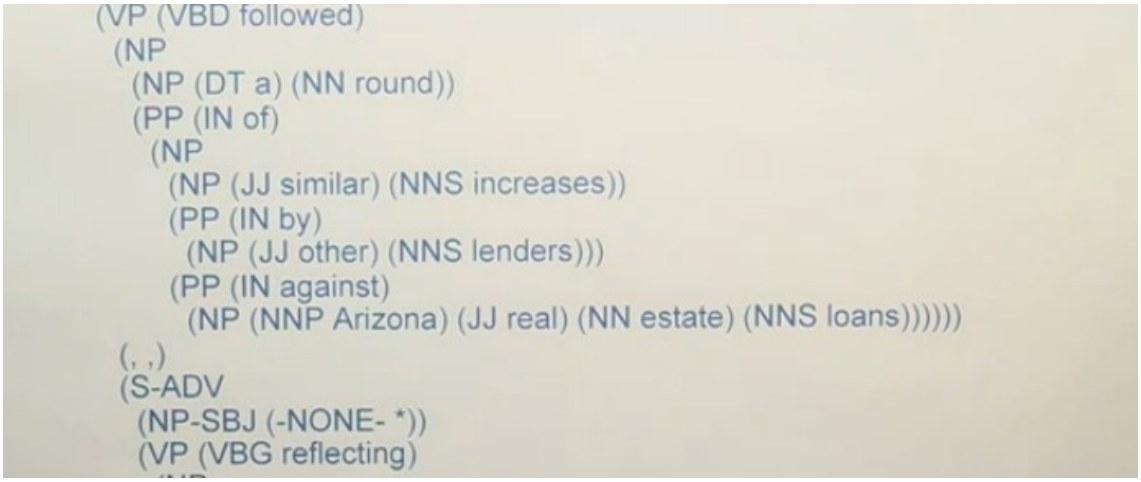


Fig. 15.6. An example of a parsed tree from the "penn tree bank" that consists of a myriad of parsed corpuses that can be used as a bass for statistical language analysis.

languages it is different.

This is a very simple model for translation. In a real translational model we would also have a probability of the final (in this case english) sentence. The problem of translation is then just a problem of search through all the possible segmentations, translations and distortions and probabilities of the final result being a valid sentence. Find the ones that gives the highest translation and pick that. The hard part is figuring out how to do this efficiently.

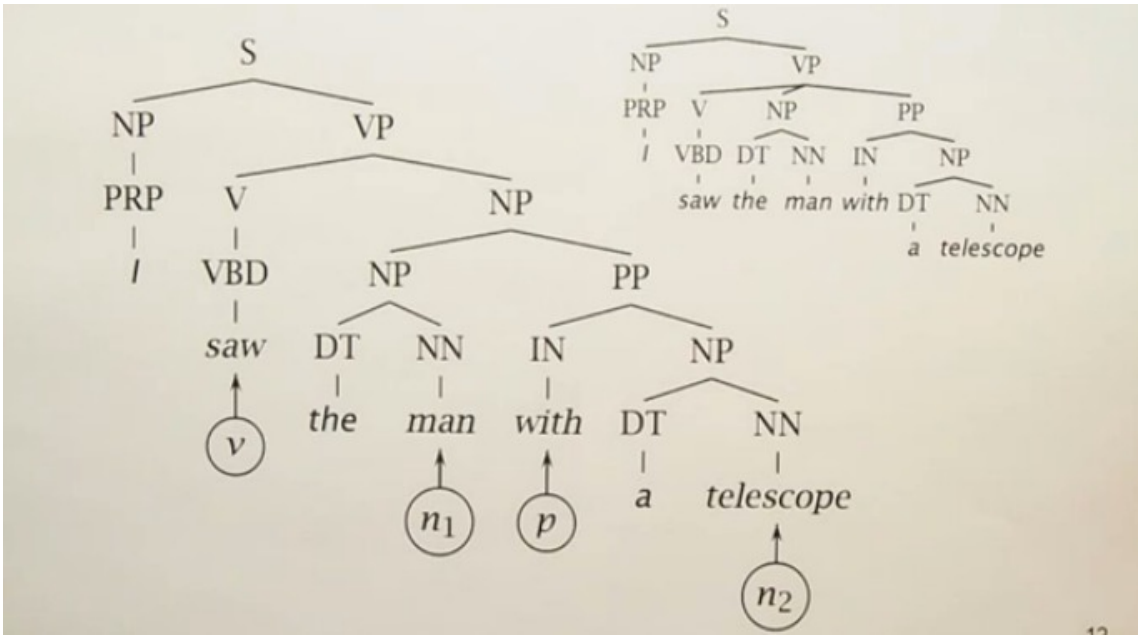


Fig. 15.7. Reduction of ambiguity in a parse

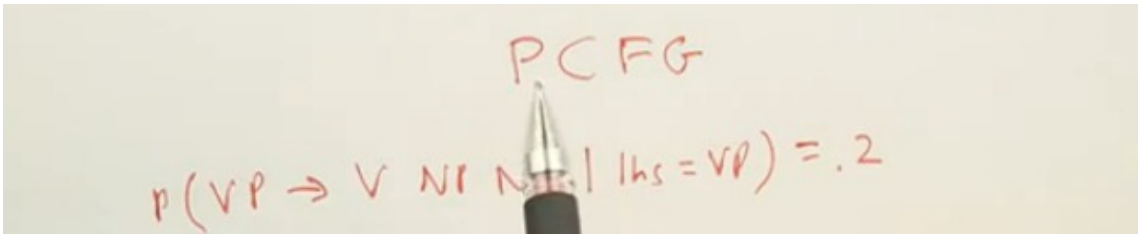


Fig. 15.8. Calculating probabilities in a sentence using probabilistic context free grammar.

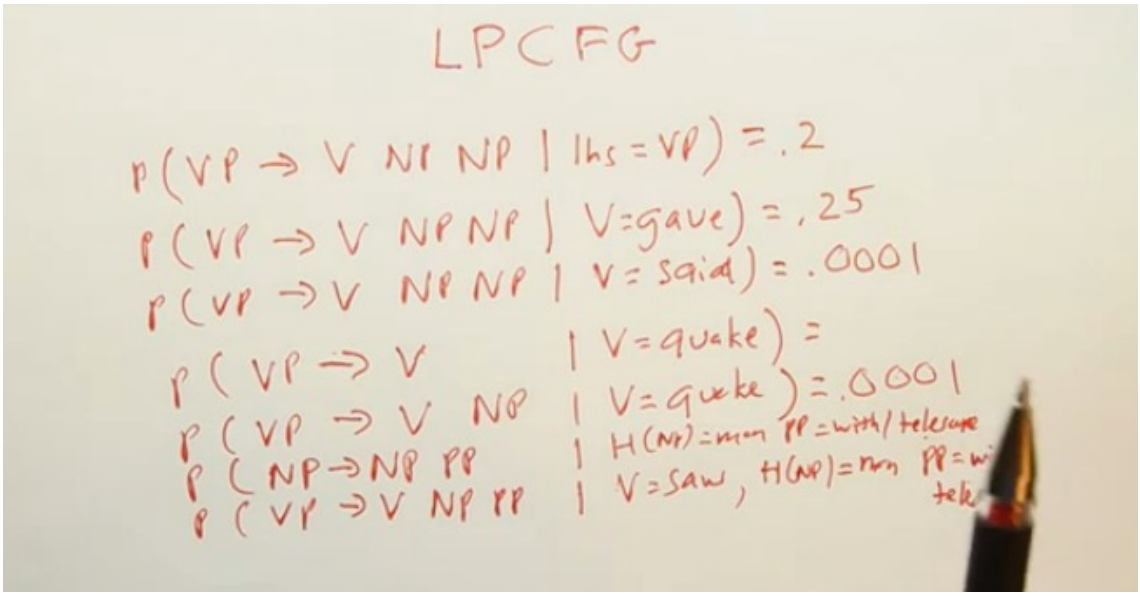


Fig. 15.9. A lexicalized probabilistic context free grammar (LPFG) will go one step further than the pfgs, and assign probabilities to individual words being associated with each other. For instance, how probable is it that a "telescope" is associated with a "a man" through which syntactic constructions? This knowledededeg can further help in disambiguation

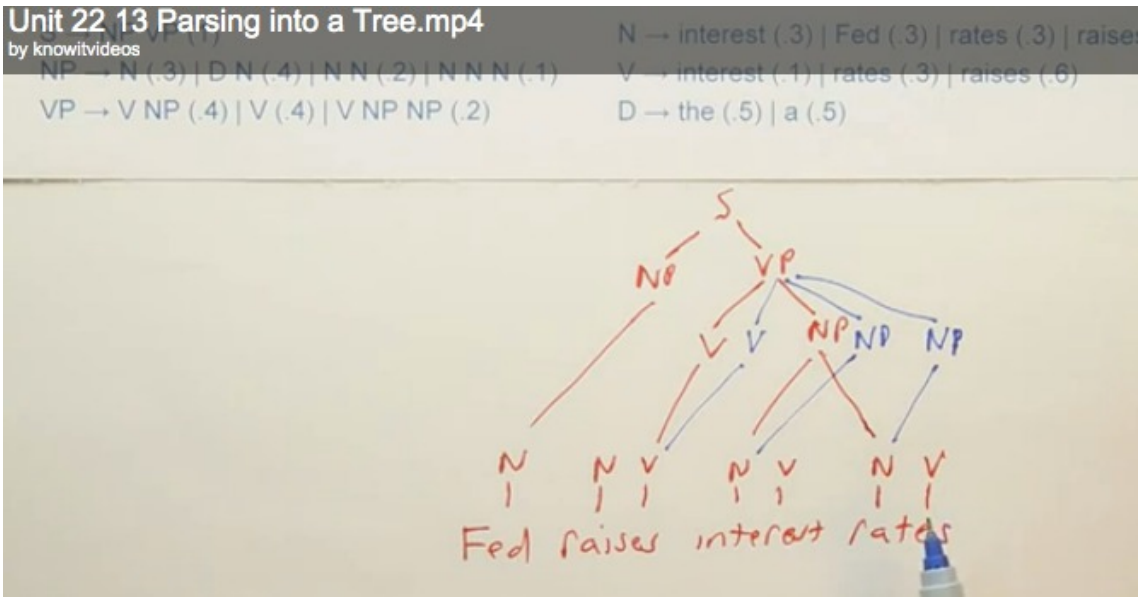


Fig. 15.10. parseexample

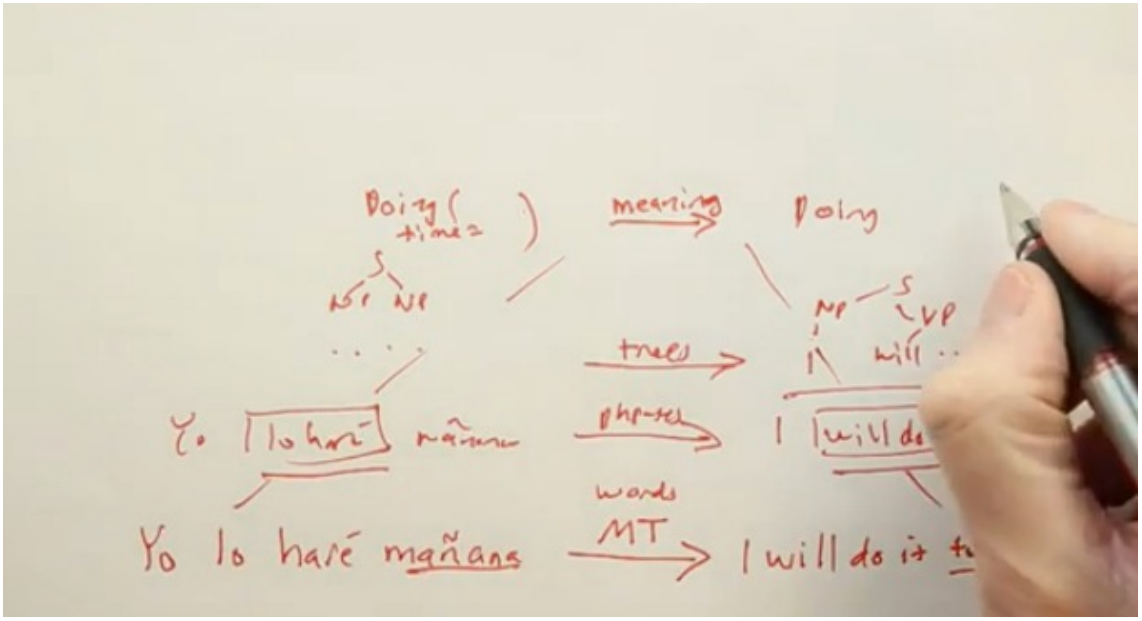


Fig. 15.11. machinetranslation

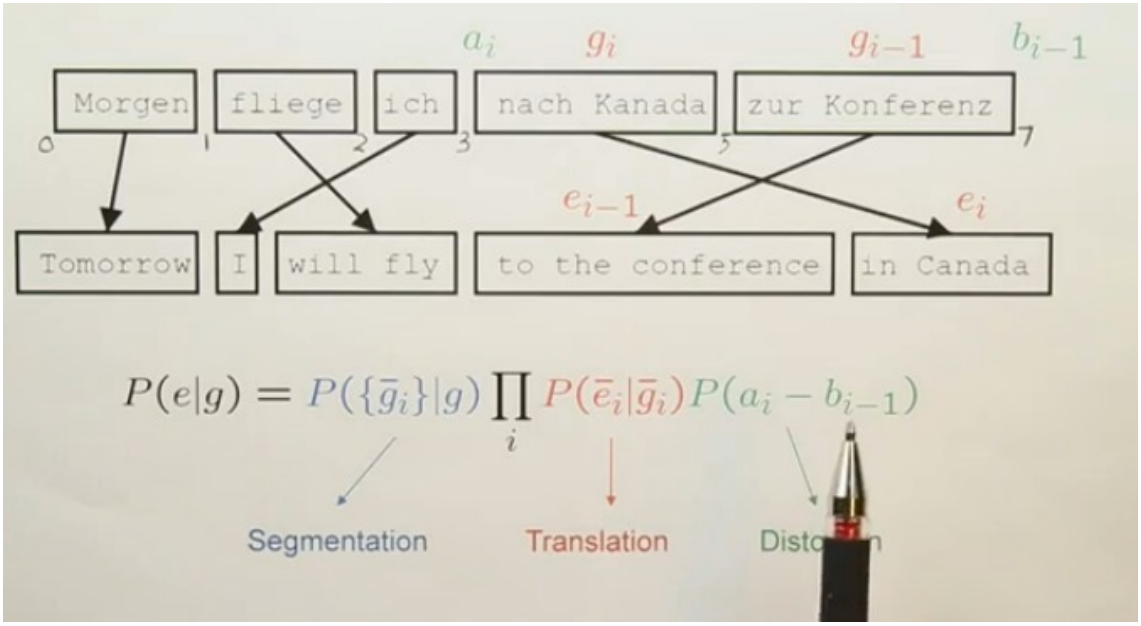


Fig. 15.12. Phrase based translation. The model gives probabilities for a sentence being a valid translation by looking at the probability for a legal segmentation, the probabilities of word for word translations and a probability for a "distortion", which in this case is just how far the phrase has to be moved to the left or right before it finds its proper location in the translated phrase.

A. Solving two by two game with an optimal mixed strategy

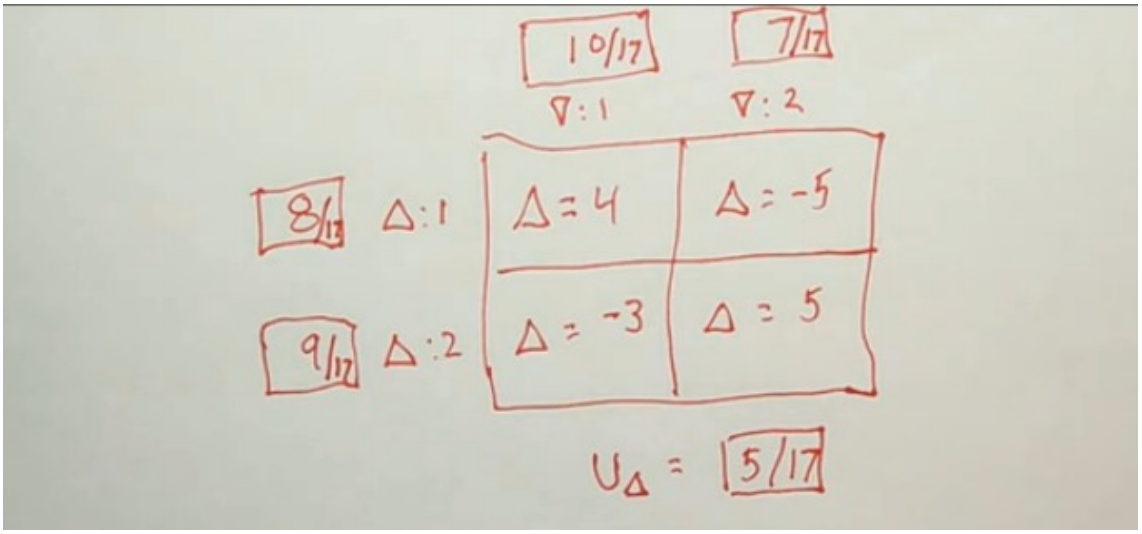


Fig. A.1. Question eight in homework six. Aka: "The persistent bugger"

There was one question, question eight in homework six that caused me an inordinate amunt of trouble. The problem was simple enough: Find the mixed strategy probabilities for two players based on a two-by-two standard representation for a game. The irritating thing was that I just couldn't get it right. With that in mind I set out to utterly demolish that type of problem once and for all, and this is the result of that demolishing process.

A.1. The problem

Given the game described by the standard matrix i figure ??, The game is a zero-sum game, and all the numbers in the matrix are the utilities for the maximizing

player. Since the game is a zero sum game the utility for the other player is minus minus the utility fot the maximizing player. The problem is to find the mixed-strategy probabilities for the two players and to determine the expected utility for the maximizing player:

	$\nabla : 1$	$\nabla : 2$
$\Delta : 1$	$\Delta = 4$	$\Delta = -5$
$\Delta : 2$	$\Delta = -3$	$\Delta = 5$

A.2. The solution

The procedure to solve this game is:

- Start by assuming that there exists an optimal mixed strategy for the maximizing player and that we are going to find it.
- Let p be the probability that max plays "1" first.
- To find the optimal p we must now assume that we don't want to give the minimizing player an advantage, so we must choose p so that whatever choice the minimizing player does, max will in the long run be neutral with respect to min's choices.
- The above condition is satisfied when this equation is satisified:

$$\begin{aligned} p \cdot (-5) + p \cdot 4 &= (1-p) \cdot (-3) \cdot p \cdot 5 \\ 17 \cdot p &= 8 \\ p &= \tfrac{8}{17} \\ (1-p) = &= \tfrac{9}{17} \end{aligned}$$

  Since the construction of this equation is what caused me the most problems, I'm here going to describe it in painstaking detail:

  - First assume that $p > 1 - p$, or equivalently $p > 0.5$. This means that the most probable move by max is to play "1".

  - Under this assumption, the minimizing player will pick the move that minimizes utility for max, and that is -5, the expected utility for the most probable move is thus $-5 \cdot p$. Since probabilities has to add up to one, we must now add the expected utility for the one remaining quadrant in the top row, and that gives us $(1 - p) \cdot 4$.

  - We then assume that $p < 0.5$, and do the same exercise for the case where we assume that max plays 2 with a better than even probability. This gives us the other side of the equation.

  - Finally we set both sides of the equation to be equal, and we find a situation where, seen from max's perspective perspective, it doesn't really matter what min chooses, since the expected payoff is invariant with respect to the choices made by min.

Rmz: *This is in fact handwaving. I need to grok this to a level of absolute certainty, and I don't do that now*

- We then do a similar exercise for the minimizing player, and come up with the equation:

$$q \cdot 4 + (1 - q) \cdot (-3) = (1 - q) \cdot 5 + q \cdot (-5)$$

  which resolves to $q = 10/17$. The only thing to keep in mind while solving this equation is that max is still a maximizing player and min is still minimizing, so this means that in the equation above (the "q") equation the oposition is trying to maximize the utility, while in the first example ("p"), the opposition is trying to minimize utility. Otherwise it's the same thing.

- to calculate the utility for the maximizing player, we sum the expectation values of all the quadrants.

$\frac{8}{17} \cdot 4$	$\frac{8}{17} \cdot -5$
$\frac{9}{17} \cdot -3$	$\frac{9}{17} \cdot 5$

When summing up all of these, the result is $\frac{10}{17}$ which is twice the utility we want, which is $\frac{5}{17}$. Why twice? Well, obviously because there are two rows, but I still need to grok this in its fullness before I can claim to understand how to solve this type of game.

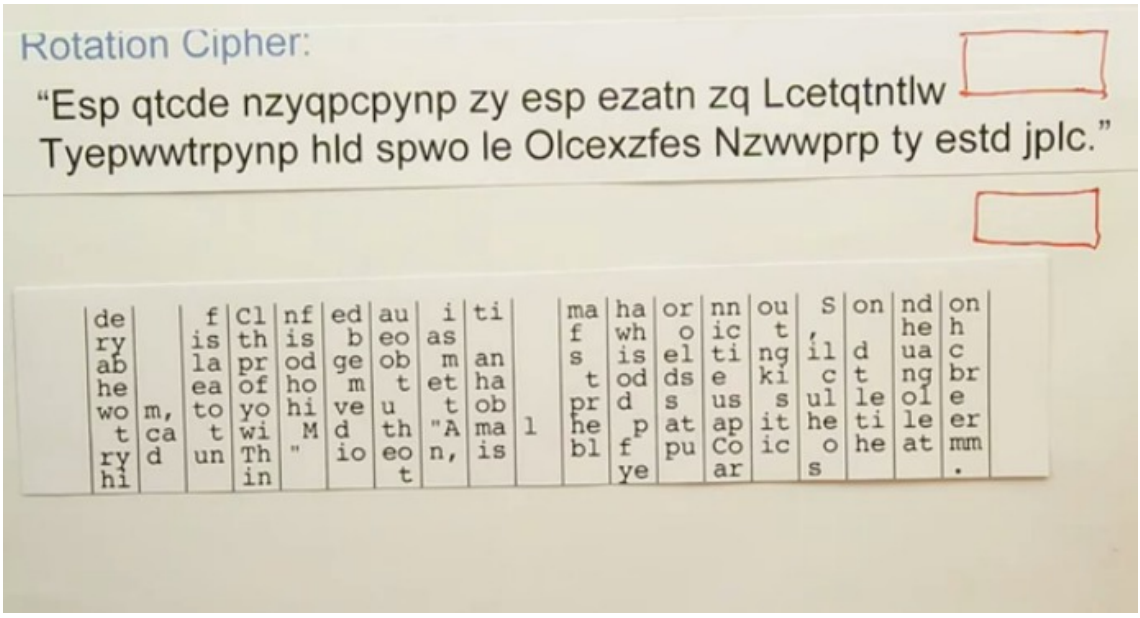


Fig. A.2. crypto challenge

A.3. The crypto challenge

```
"Esp qtcde nzyqpcpynp zy esp ezatn zq Lcetqtntlw Tyepwwtrpynp hld spwo le Olcexzfes
```

Index