Artificial Intelligence I

Dr Sean Holden

Computer Laboratory, Room FC06

Telephone extension 63725

Email: sbh11@cl.cam.ac.uk

www.cl.cam.ac.uk/~sbh11/

Introduction: what's AI for?

What is the purpose of Artificial Intelligence (AI)? If you're a *philosopher* or a *psychologist* then perhaps it's:

- To *understand intelligence*.
- To understand *ourselves*.

Philosophers have worked on this for at least 2000 years. They've also wondered about:

- *Can* we do AI? *Should* we do AI?
- Is AI *impossible*? (Note: I didn't write *possible* here, for a good reason...)

Despite 2000 years of work, there's essentially *diddly-squat* in the way of results.

Introduction: what's AI for?

Luckily, we were sensible enough not to pursue degrees in philosophy—we're scientists/engineers, so while we might have *some* interest in such pursuits, our perspective is different:

- Brains are small (true) and apparently slow (not quite so clear-cut), but incredibly good at some tasks—we want to understand a specific form of *computation*.
- It would be nice to be able to *construct* intelligent systems.
- It is also nice to *make and sell cool stuff*.

This view *seems to be the more successful...*

AI is entering our lives almost without us being aware of it.

Introduction: now is a fantastic time to investigate AI

In many ways this is a young field, having only really got under way in 1956 with the *Dartmouth Conference*.

`www-formal.stanford.edu/jmc/history/dartmouth/dartmouth.html`

- This means we can actually *do* things. It's as if we were physicists before anyone thought about atoms, or gravity, or. . . .
- Also, we know what we're trying to do is *possible*. (Unless we think humans don't exist. *NOW STEP AWAY FROM THE PHILOSOPHY* before *SOMEONE GETS HURT!!!!*)

Perhaps I'm being too hard on them; there was some good groundwork: *Socrates* wanted an algorithm for *"piety"*, leading to *Syllogisms*. Ramon Lull's *concept wheels* and other attempts at mechanical calculators. Rene Descartes' *Dualism* and the idea of mind as a *physical system*. Wilhelm Leibnitz's opposing position of *Materialism*. (The intermediate position: mind is *physical* but *unknowable*.) The origin of *knowledge*: Francis Bacon's *Empiricism*, John Locke: *"Nothing is in the understanding, which was not first in the senses"*. David Hume: we obtain rules by repeated exposure: *Induction*. Further developed by Bertrand Russel and in the *Confirmation Theory* of Carnap and Hempel.

More recently: the connection between *knowledge* and *action*? How are actions *justified*? If to achieve the end you need to achieve something intermediate, consider how to achieve that, and so on. This approach was implemented in Newell and Simon's 1957 *General Problem Solver (GPS)*.

Is AI possible?

Many philosophers are particularly keen to argue that AI is *impossible*? Why is this? We have:

- Perception (vision, speech processing...)
- Logical reasoning (prolog, expert systems, CYC...)
- Playing games (chess, backgammon, go...)
- Diagnosis of illness (in various contexts...)
- Theorem proving (Robbin's conjecture...)
- Literature and music (automated writing and composition...)
- And many more...

What's made the difference? In a nutshell: *we're the first lucky bunch to get our hands on computers*, and that allows us to *build things*.

The simple ability to *try things out* has led to huge advances in a relatively short time. *So:* don't believe the critics...

Further reading

Why do people dislike the idea that humanity might not be *special*.

An excellent article on why this view is much more problematic than it might seem is:

"Why people think computers can't," Marvin Minsky. AI Magazine, volume 3 number 4, 1982.

Aside: when something is understood it stops being AI

To have AI, you need a means of *implementing* the intelligence. Computers are (at present) the only devices in the race. (Although *quantum computation* is looking interesting...)

AI has had a major effect on computer science:

- Time sharing
- Interactive interpreters
- Linked lists
- Storage management
- Some fundamental ideas in object-oriented programming
- and so on...

When AI has a success, the ideas in question tend to *stop being called AI*.

Similarly: do you consider the fact that *your phone can do speech recognition* to be a form of AI?

The nature of the pursuit

What is AI? This is not necessarily a straightforward question.

It depends on who you ask...

We can find many definitions and a rough categorisation can be made depending on whether we are interested in:

- The way in which a system *acts* or the way in which it *thinks*.
- Whether we want it to do this in a *human* way or a *rational* way.

Here, the word *rational* has a special meaning: it means *doing the correct thing in given circumstances*.

Acting like a human

What is AI, version one: acting like a human

Alan Turing proposed what is now known as the *Turing Test*.

- A human judge is allowed to interact with an AI program via a terminal.
- This is the *only* method of interaction.
- If the judge can't decide whether the interaction is produced by a machine or another human then the program passes the test.

In the *unrestricted* Turing test the AI program may also have a camera attached, so that objects can be shown to it, and so on.

Acting like a human

The Turing test is informative, and (very!) hard to pass.

- It requires many abilities that seem necessary for AI, such as learning. *BUT*: a human child would probably not pass the test.
- Sometimes an AI system needs human-like acting abilities—for example *expert systems* often have to produce explanations—but *not always*.

See the *Loebner Prize in Artificial Intelligence*:

`www.loebner.net/Prizef/loebner-prize.html`

Thinking like a human

What is AI, version two: thinking like a human

There is always the possibility that a machine *acting* like a human does not actually *think*. The *cognitive modelling* approach to AI has tried to:

- Deduce *how humans think*—for example by *introspection* or *psychological experiments*.
- Copy the process by mimicking it within a program.

An early example of this approach is the *General Problem Solver* produced by Newell and Simon in 1957. They were concerned with whether or not the program reasoned in the same manner that a human did.

Computer Science + Psychology = *Cognitive Science*

Thinking rationally: the "laws of thought"

What is AI, version three: thinking rationally

The idea that intelligence reduces to *rational thinking* is a very old one, going at least as far back as Aristotle as we've already seen.

The general field of *logic* made major progress in the 19th and 20th centuries, allowing it to be applied to AI.

- We can *represent* and *reason* about many different things.
- The *logicist* approach to AI.

This is a very appealing idea. *However...*

Thinking rationally: the "laws of thought"

Unfortunately there are obstacles to any naive application of logic. It is hard to:

- Represent *commonsense knowledge*.
- Deal with *uncertainty*.
- Reason without being tripped up by *computational complexity*.

These will be recurring themes in this course, and in AI II.

Logic alone also falls short because:

- Sometimes it's necessary to act when there's *no* logical course of action.
- Sometimes inference is *unnecessary* (reflex actions).

Further reading

The *Fifth Generation Computer System* project has most certainly earned the badge of *"heroic failure"*.

It is an example of how much harder the logicist approach is than you might think:

"Overview of the Fifth Generation Computer Project," Tohru Moto-oka. ACM SIGARCH Computer Architecture News, volume 11, number 3, 1983.

Acting rationally

What is AI, version four: acting rationally

Basing AI on the idea of *acting rationally* means attempting to design systems that act to *achieve their goals* given their *beliefs*.

Thinking about this in engineering terms, it seems *almost inevitably* to lead us towards the usual subfields of AI. What might be needed?

- To make *good decisions* in many *different situations* we need to *represent* and *reason* with *knowledge*.
- We need to deal with *natural language*.
- We need to be able to *plan*.
- We need *vision*.
- We need *learning*.

And so on, so all the usual AI bases seem to be covered.

Acting rationally

The idea of *acting rationally* has several advantages:

- The concepts of *action*, *goal* and *belief* can be defined precisely making the field suitable for scientific study.

This is important: if we try to model AI systems on humans, we can't even propose *any* sensible definition of *what a belief or goal is*.

In addition, humans are a system that is still changing and adapted to a very specific environment.

Rational acting does not have these limitations.

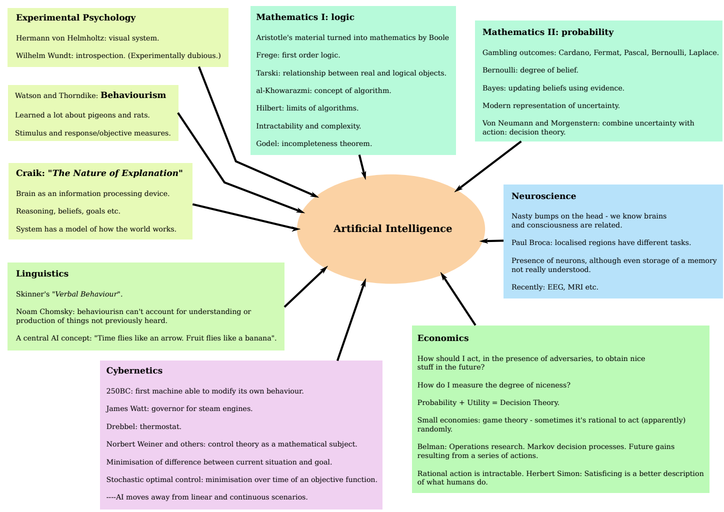
Acting rationally

Rational acting also seems to *include* two of the alternative approaches:

- All of the things needed to pass a Turing test seem necessary for rational acting, so this seems preferable to the *acting like a human* approach.
- The logicist approach can clearly form *part* of what's required to act rationally, so this seems preferable to the *thinking rationally* approach alone.

As a result, we will focus on the idea of designing systems that *act rationally*.

Other fields that have contributed to AI



What's in this course?

This course introduces some of the fundamental areas that make up AI:

- An outline of the background to the subject.
- An introduction to the idea of an *agent*.
- Solving problems in an intelligent way by *search*.
- Solving problems represented as *constraint satisfaction* problems.
- Playing *games*.
- *Knowledge representation, and reasoning*.
- *Planning*.
- *Learning* using *neural networks*.

Strictly speaking, AI I covers what is often referred to as *"Good Old-Fashioned AI"*. (Although "Old-Fashioned" is a misleading term.)

The nature of the subject changed a great deal when the importance of *uncertainty* became fully appreciated. AI II covers this more recent material.

What's *not* in this course?

- The classical AI programming languages *prolog* and *lisp*.
- A great deal of all the areas on the last slide!
- Perception: *vision*, *hearing* and *speech processing*, *touch* (force sensing, knowing where your limbs are, knowing when something is bad), *taste*, *smell*.
- Natural language processing.
- Acting on and in the world: *robotics* (effectors, locomotion, manipulation), *control engineering*, *mechanical engineering*, *navigation*.
- Areas such as *genetic algorithms/programming*, *swarm intelligence*, *artificial immune systems* and *fuzzy logic*, for reasons that I will expand upon during the lectures.
- *Uncertainty* and much further probabilistic material. (You'll have to wait until next year.)

Text book

The course is based on the relevant parts of:

Artificial Intelligence: A Modern Approach, Third Edition (2010). Stuart Russell and Peter Norvig, Prentice Hall International Editions.

NOTE: This is also the main recommended text for AI2.

Interesting things on the web

A few interesting web starting points:

The Honda Asimo robot: `world.honda.com/ASIMO`

AI at Nasa Ames: `www.nasa.gov/centers/ames/research/exploringtheuniverse/spiffy.html`

DARPA Grand Challenge: `http://www.darpagrandchallenge.com/`

2007 DARPA Urban Challenge: `cs.stanford.edu/group/roadrunner`

The Cyc project: `www.cyc.com`

Human-like robots: `www.ai.mit.edu/projects/humanoid-robotics-group`

Sony robots: `support.sony-europe.com/aibo`

NEC "PaPeRo": `www.nec.co.jp/products/robot/en`

Prerequisites

The prerequisites for the course are: first order logic, some algorithms and data structures, discrete and continuous mathematics, basic computational complexity.

DIRE WARNING:

In the lectures on *machine learning* I will be talking about *neural networks*.

This means you will need to be able to *differentiate* and also handle *vectors and matrices*.

If you've forgotten how to do this *you WILL get lost—I guarantee it!!!*

Prerequisites

Self test:

1. Let
$$f(x_1, \ldots, x_n) = \sum_{i=1}^{n} a_i x_i^2$$
where the a_i are constants. Can you compute $\partial f / \partial x_j$ where $1 \leq j \leq n$?
2. Let $f(x_1, \ldots, x_n)$ be a function. Now assume $x_i = g_i(y_1, \ldots, y_m)$ for each x_i and some collection of functions g_i. Assuming all requirements for differentiability and so on are met, can you write down an expression for $\partial f / \partial y_j$ where $1 \leq j \leq m$?

If the answer to either of these questions is "no" then it's time for some revision. (You have about three weeks notice, so I'll assume you know it!)

And finally...

There are some important points to be made regarding *computational complexity*.

First, you might well hear the term *AI-complete* being used a lot. What does it mean?

AI-complete: only solvable if you can solve AI in its entirety.

For example: high-quality automatic translation from one language to another.

To produce a genuinely good translation of *Moby Dick* from English to Cantonese is likely to be AI complete.

And finally...

More practically, you will often hear me make the claim that *everything that's at all interesting in AI is at least NP-complete*.

There are two ways to interpret this:

1. The wrong way: "It's all a waste of time.[1]" OK, so it's a partly understandable interpretation. *BUT* the fact that the travelling salesman problem is intractable *does not* mean there's no such thing as a satnav...
2. The right way: "It's an opportunity to design nice approximation algorithms." In reality, the algorithms that are *good in practice* are ones that try to *often* find a *good* but not necessarily *optimal* solution, in a *reasonable* amount of time.

[1] In essence, a comment on a course assessment a couple of years back to the effect of: "Why do you teach us this stuff if it's all futile?"

Artificial Intelligence I

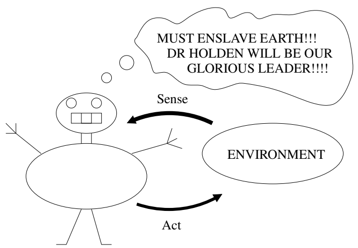
Dr Sean Holden

An introduction to *Agents*

Agents

There are many different definitions for the term *agent* within AI.

Allow me to introduce EVIL ROBOT.



We will use the following simple definition: *an agent is any device that can sense and act upon its environment*.

Agents

This definition can be very widely applied: to humans, robots, pieces of software, and so on.

We are taking quite an *applied* perspective. We want to *make things* rather than *copy humans*, so to be scientific there are some issues to be addressed:

- How can we judge an agent's performance?
- How can an agent's *environment* affect its design?
- Are there sensible ways in which to think about the *structure* of an agent?

Recall that we are interested in devices that *act rationally*, where 'rational' means doing the *correct thing* under *given circumstances*.

Reading: Russell and Norvig, chapter 2.

Measuring performance

How can we judge an agent's performance? Any measure of performance is likely to be *problem-specific*.

Example: For a chess playing agent, we might use its rating.

Example: For a mail-filtering agent, we might devise a measure of how well it blocks spam, but allows interesting email to be read.

Example: For a car driving agent the measure needs considerable sophistication: we need to take account of comfort, journey time, safety *etc*.

So: the choice of a performance measure is itself worthy of careful consideration.

Measuring performance

We're usually interested in *expected, long-term performance*.

- *Expected* performance because usually agents are not *omniscient*—they don't *infallibly* know the outcome of their actions.
- It is *rational* for you to enter this lecture theatre even if the roof falls in today.

An agent capable of detecting and protecting itself from a falling roof might be more *successful* than you, but *not* more *rational*.

- *Long-term performance* because it tends to lead to better approximations to what we'd consider rational behaviour.
- We probably don't want our car driving agent to be outstandingly smooth and safe for most of the time, but have episodes of *driving through the local orphanage at 150 mph*.

Environments

How can an agent's *environment* affect its design? *Example:* the environment for a *chess program* is vastly different to that for an *autonomous deep-space vehicle*. Some common attributes of an environment have a considerable influence on agent design.

- *Accessible/inaccessible:* do percepts tell you *everything* you need to know about the world?
- *Deterministic/non-deterministic:* does the future depend *predictably* on the present and your actions?
- *Episodic/non-episodic* is the agent run in independent episodes.
- *Static/dynamic:* can the world change while the agent is deciding what to do?
- *Discrete/continuous:* an environment is discrete if the sets of allowable percepts and actions are finite.

Environments

All of this assumes there is only one agent.

When multiple agents are involved we need to consider:

- Whether the situation is *competitive* or *cooperative*.
- Whether *communication* required?

An example of multiple agents:

news.bbc.co.uk/1/hi/technology/3486335.stm

Basic structures for intelligent agents

Are there sensible ways in which to think about the *structure* of an agent? Again, this is likely to be *problem-specific*, although perhaps to a lesser extent.

So far, an agent is based on percepts, actions and goals.

Example: Aircraft piloting agent.

Percepts: sensor information regarding height, speed, engines *etc*, audio and video inputs, and so on.

Actions: manipulation of the aircraft's controls.

Also, perhaps talking to the passengers *etc*.

Goals: get to the necessary destination as quickly as possible with minimal use of fuel, without crashing *etc*.

Programming agents

A basic agent can be thought of as working on a straightforward underlying process:

- *Gather perceptions*.
- Update *working memory* to take account of them.
- On the basis of what's in the working memory, *choose an action* to perform.
- *Update* the working memory to take account of this action.
- *Do* the chosen action.

Obviously, this hides a great deal of complexity.

Also, it ignores subtleties such as the fact that a percept might arrive while an action is being chosen.

Programming agents

We'll initially look at two hopelessly limited approaches, because they do suggest a couple of important points.

Hopelessly limited approach number 1: use a table to map percept sequences to actions. This can quickly be rejected.

- The table will be *huge* for any problem of interest. About 35^{100} entries for a chess player.
- We don't usually know how to fill the table.
- Even if we allow table entries to be *learned* it will take too long.
- The system would have no *autonomy*.

We can attempt to overcome these problems by allowing agents to *reason*.

Autonomy is an interesting issue though...

Autonomy

If an agent's behaviour depends in some manner on its *own experience of the world* via its percept sequence, we say it is *autonomous*.

- An agent using only built-in knowledge would seem not to be successful at AI in any meaningful sense: its behaviour is predefined by its designer.
- On the other hand *some* built-in knowledge seems essential, even to humans.

Not all animals are entirely autonomous.

For example: dung beetles.

Reflex agents

Hopelessly limited approach number 2: try *extracting* pertinent information and using *rules* based on this.

Condition-action rules: `if` a certain *state* is observed `then` perform some *action*

Some points immediately present themselves regarding *why* reflex agents are unsatisfactory:

- We can't always decide what to do based on the *current percept*.
- However storing *all* past percepts might be undesirable (for example requiring too much memory) or just unnecessary.
- Reflex agents don't maintain a description of the *state of their environment*...
- ...however this seems necessary for any meaningful AI. (Consider automating the task of driving.)

This is all the more important as usually percepts don't tell you *everything about the state*.

Keeping track of the environment

It seems reasonable that an agent should maintain:

- A *description of the current state of its environment*.
- Knowledge of how the environment *changes independently of the agent*.
- Knowledge of how the agent's *actions affect its environment*.

This requires us to do *knowledge representation* and *reasoning*.

Goal-based agents

It seems reasonable that an agent should choose a rational course of action depending on its *goal*.

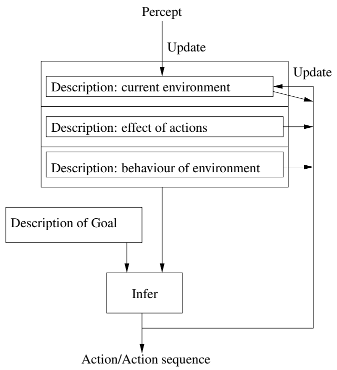
- If an agent has knowledge of how its actions affect the environment, then it has a basis for choosing actions to achieve goals.
- To obtain a *sequence* of actions we need to be able to *search* and to *plan*.

This is *fundamentally different* from a reflex agent.

For example: by changing the goal you can change the entire behaviour.

Goal-based agents

We now have a basic design that looks something like this:



Utility-based agents

Introducing goals is still not the end of the story.

There may be *many* sequences of actions that lead to a given goal, and *some may be preferable to others*.

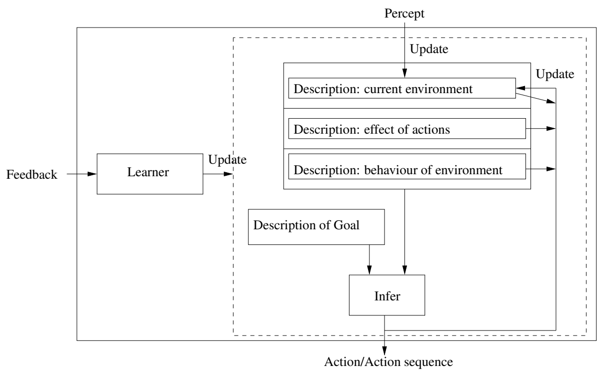
A *utility function* maps a state to a number representing the desirability of that state.

- We can trade-off *conflicting goals*, for example speed and safety.
- If an agent has several goals and is not certain of achieving any of them, then it can trade-off likelihood of reaching a goal against the desirability of getting there.

Maximising expected utility over time forms a fundamental model for the design of agents. However we don't get as far as that until AI II.

Learning agents

It seems reasonable that an agent should *learn from experience*.



Learning agents

This requires two additions:

- The learner needs some form of *feedback* on the agent's performance. This can come in several different forms.
- In general, we also need a means of *generating new behaviour* in order to find out about the world.

This in turn implies a trade-off: should the agent spend time *exploiting* what it's learned so far, or *exploring* the environment on the basis that it might learn something really useful?

What have we learned? (No pun intended...)

The *crucial* things that should be taken away from this lecture are:

- The nature of an agent depends on its *environment* and *performance measure*.
- We're usually interested in *expected, long-term performance*.
- Autonomy requires that an agent in some way behaves *depending on its experience of the world*.
- There is a *natural basic structure* on which agent design can be based.
- Consideration of that structure leads naturally to the basic areas covered in this course.

Those basic areas are: *knowledge representation and reasoning, search, planning and learning*. Oh, and finally, we've learned NOT TO MESS WITH EVIL ROBOT... he's a VERY BAD ROBOT!

Artificial Intelligence I

Dr Sean Holden

Notes on *problem solving by search*

Problem solving by search

We begin with what is perhaps the simplest collection of AI techniques: those allowing an *agent* existing within an *environment* to *search* for a *sequence of actions* that *achieves a goal*.

The algorithms can, crudely, be divided into two kinds: *uninformed* and *informed*.

Not surprisingly, the latter are more effective and so we'll look at those in more detail.

Reading: Russell and Norvig, chapters 3 and 4.

Problem solving by search

As with any area of computer science, some degree of *abstraction* is necessary when designing AI algorithms.

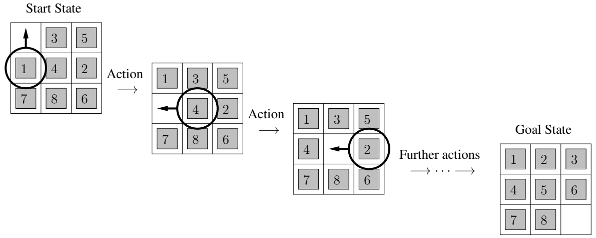
Search algorithms apply to a particularly simple class of problems—we need to identify:

- *An initial state*: what is the agent's situation to start with?
- *A set of actions*: these are modelled by specifying what state will result on performing any available action from any known state.
- *A goal test*: we can tell whether or not the state we're in corresponds to a goal.

Note that the goal may be described by a property rather than an explicit state or set of states, for example *checkmate*.

Problem solving by search

A simple example: *the 8-puzzle*.



(A good way of keeping kids quiet...)

Problem solving by search

Start state: a randomly-selected configuration of the numbers 1 to 8 arranged on a 3×3 square grid, with one square empty.

Goal state: the numbers in ascending order with the bottom right square empty.

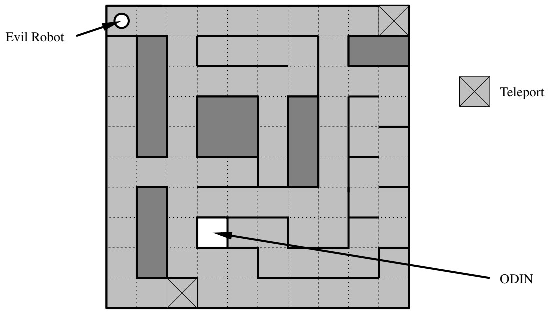
Actions: `left`, `right`, `up`, `down`. We can move any square adjacent to the empty square into the empty square. (It's not always possible to choose from all four actions.)

Path cost: 1 per move.

The 8-puzzle is very simple. However general sliding block puzzles are a good test case. The general problem is NP-complete. The 5×5 version has about 10^{25} states, and a random instance is in fact quite a challenge.

Problem solving by basic search

EVIL ROBOT has found himself in an unfamiliar building:



He wants the *ODIN (Oblivion Device of Indescribable Nastiness)*.

Problem solving by search

Start state: EVIL ROBOT is in the top left corner.

Goal state: EVIL ROBOT is in the area containing the ODIN.

Actions: `left`, `right`, `up`, `down`. We can move as long as there's no wall in the way. (Again, it's not always possible to choose from all four actions.)

Path cost: 1 per move. If you step on a teleport then you move to the other one with a cost of 0.

Problems of this kind are very simple, but a surprisingly large number of applications have appeared:

- Route-finding/tour-finding.
- Layout of VLSI systems.
- Navigation systems for robots.
- Sequencing for automatic assembly.
- Searching the internet.
- Design of proteins.

and many others...

Problems of this kind continue to form an active research area.

It's worth emphasising that a lot of abstraction has taken place here:

- Can the agent know it's current state in full?
- Can the agent know the outcome of its actions in full?

Single-state problems: the state is always known precisely, as is the effect of any action. There is therefore a single outcome state.

Multiple-state problems: The effects of actions are known, but the state can not reliably be inferred, or the state is known but not the effects of the actions.

Both single and multiple state problems can be handled using these search techniques. In the latter, we must reason about the set of states that we could be in:

- In this case we have an initial *set* of states.
- Each action leads to a further *set* of states.
- The goal is a set of states *all* of which are valid goals.

Contingency problems

In some situations it is necessary to perform sensing *while* the actions are being carried out in order to guarantee reaching a goal.

(It's good to keep your eyes open while you cross the road!)

This kind of problem requires *planning* and will be dealt with later.

Sometimes it is actively beneficial to act and see what happens, rather than to try to consider all possibilities in advance in order to obtain a perfect plan.

Exploration problems

Sometimes you have *no* knowledge of the effect that your actions have on the environment.

Babies in particular have this experience.

This means you need to experiment to find out what happens when you act.

This kind of problem requires *reinforcement learning* for a solution. We will not cover reinforcement learning in this course. (Although it is in AI II.)

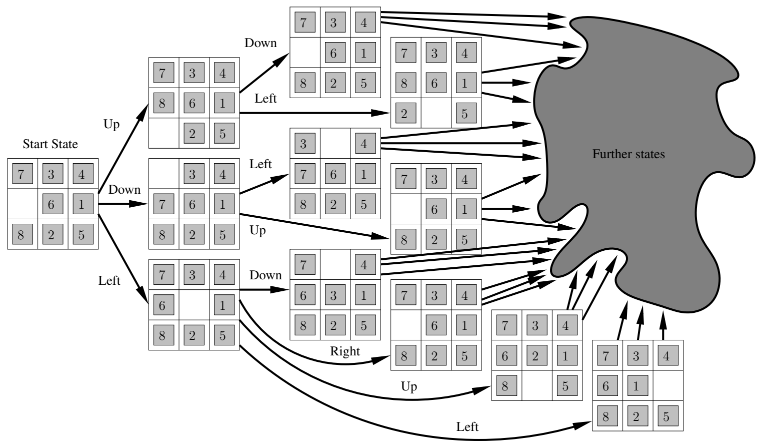
Search trees

The basic idea should be familiar from your *Algorithms I* course, and also from *Foundations of Computer Science*.

- We build a *tree* with the start state as root node.
- A node is *expanded* by applying actions to it to generate new states.
- A *path* is a *sequence of actions* that lead from state to state.
- The aim is to find a *goal state* within the tree.
- A *solution* is a path beginning with the initial state and ending in a goal state.

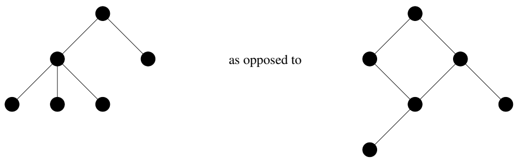
We may also be interested in the *path cost* as some solutions might be better than others.

Path cost will be denoted by p.



Search trees versus search graphs

We need to make an important distinction between *search trees* and *search graphs*. For the time being we assume that it's a *tree* as opposed to a *graph* that we're dealing with.



(There is a good reason for this, which we'll get to in a moment...)

In a *tree* only *one path* can lead to a given state. In a *graph* a *state* can be reached via possibly *multiple paths*.

Search trees

Basic approach:

- Test the root to see if it is a goal.
- If not then *expand* it by generating all possible successor states according to the available actions.
- If there is only one outcome state then move to it. Otherwise choose one of the outcomes and expand it.
- The way in which this choice is made defines a *search strategy*.
- Repeat until you find a goal.

The collection of states generated but not yet expanded is called the *fringe* or *frontier* and is generally stored as a *queue*.

The basic tree-search algorithm

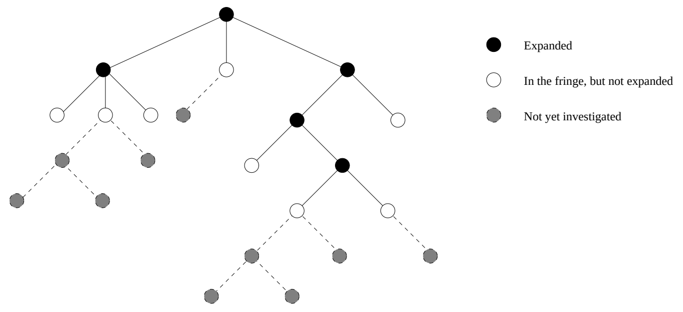
In pseudo-code, the algorithm looks like this:

```
function treeSearch {
  fringe = queue containing only the start state;
  while() {
    if (empty(fringe))
      return fail;
    node = head(fringe);
    if (goal(node))
      return solution(node);
    fringe = insert(expand(node), fringe);
  }
}
```

The *search strategy* is set by using a *priority queue*.

The definition of *priority* then sets the way in which the tree is searched.

The basic tree-search algorithm



The basic tree-search algorithm

We can immediately define some familiar tree search algorithms:

- New nodes are added to the *head of the queue*. This is *depth-first search*.
- New nodes are added to the *tail of the queue*. This is *breadth-first search*.

We will not dwell on these, as they are both *completely hopeless* in practice.

Why is that?

The performance of search techniques

How might we judge the performance of a search technique?

We are interested in:

- Whether a solution is found.
- Whether the solution found is a good one in terms of path cost.
- The cost of the search in terms of time and memory.

So

$$\text{the total cost} = \text{path cost} + \text{search cost}$$

If a problem is highly complex it may be worth settling for a *sub-optimal solution* obtained in a *short time*.

We are also interested in:

Completeness: does the strategy *guarantee* a solution is found?

Optimality: does the strategy guarantee that the *best* solution is found?

Once we start to consider these, things get a lot more interesting...

Breadth-first search

Why is breadth-first search hopeless?

- The procedure is *complete*: it is guaranteed to find a solution if one exists.
- The procedure is *optimal* if the path cost is a non-decreasing function of node-depth.
- The procedure has *exponential complexity for both memory and time*. A branching factor b requires

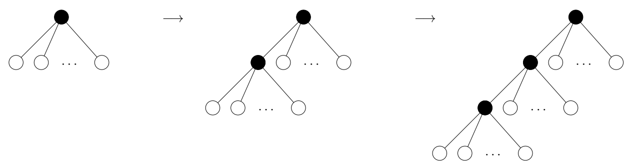
$$1 + b + b^2 + b^3 + \cdots + b^d = \frac{b^{d+1} - 1}{b - 1}$$

nodes if the shortest path has depth d.

In practice it is the *memory* requirement that is problematic.

Depth-first search

With depth-first search: for a given branching factor b and depth d the memory requirement is $O(bd)$.



This is because we need to store *nodes on the current path* and *the other unexpanded nodes*.

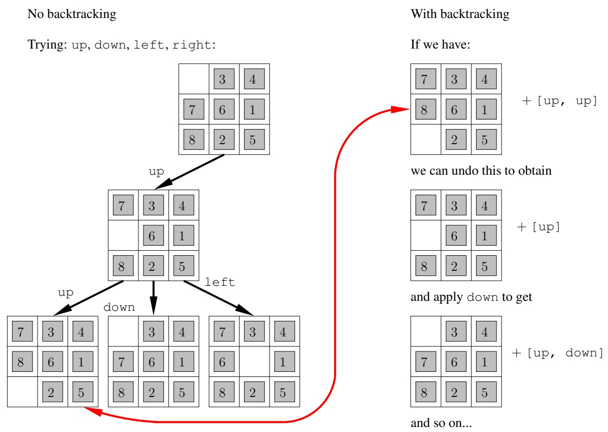
The time complexity is $O(b^d)$. Despite this, if there are *many solutions* we stand a chance of finding one quickly, compared with breadth-first search.

Backtracking search

We can sometimes improve on depth-first search by using *backtracking search*.

- If each node knows how to *generate the next possibility* then memory is improved to $O(d)$.
- Even better, if we can work by *making modifications* to a *state description* then the memory requirement is:
  - One full state description, plus...
  - ... $O(d)$ actions (in order to be able to *undo* actions).

How does this work?



Depth-first, depth-limited, and iterative deepening search

Depth-first search is clearly dangerous if the tree is *very deep or infinite*.

Depth-limited search simply imposes a limit on depth. For example if we're searching for a route on a map with n cities we know that the maximum depth will be n. However:

- We still risk finding a suboptimal solution.
- The procedure becomes problematic if we impose a depth limit that is too small.

Usually we do not know a reasonable depth limit in advance.

Iterative deepening search repeatedly runs depth-limited search for increasing depth limits $0, 1, 2, \ldots$

Iterative deepening search

Iterative deepening search:

- Essentially combines the advantages of depth-first and breadth-first search.
- It is complete and optimal.
- It has a memory requirement similar to that of depth-first search.

Importantly, the fact that you're repeating a search process several times is less significant than it might seem.

It's *still* not a good practical method, but it does point us in the direction of one...

Iterative deepening search

Iterative deepening depends on the fact that *the vast majority of the nodes in a tree are in the bottom level*:

- In a tree with branching factor b and depth d the number of nodes is
$$f_1(b, d) = 1 + b + b^2 + b^3 + \cdots + b^d = \frac{b^{d+1} - 1}{b - 1}$$
- A complete iterative deepening search of this tree generates the final layer once, the penultimate layer twice, and so on down to the root, which is generated $d + 1$ times. The total number of nodes generated is therefore
$$f_2(b, d) = (d + 1) + db + (d - 1)b^2 + (d - 2)b^3 + \cdots + 2b^{d-1} + b^d$$

Iterative deepening search

Example:

- For $b = 20$ and $d = 5$ we have
$$f_1(b, d) = 3,368,421$$
$$f_2(b, d) = 3,545,706$$
which represents a 5 percent increase with iterative deepening search.
- The overhead gets *smaller* as b increases. However the time complexity is still exponential.

Iterative deepening search

Further insight can be gained if we note that

$$f_2(b,d) = f_1(b,0) + f_1(b,1) + \cdots + f_1(b,d)$$

as we generate the root, then the tree to depth 1, and so on. Thus

$$\begin{aligned} f_2(b,d) &= \sum_{i=0}^{d} f_1(b,i) = \sum_{i=0}^{d} \frac{b^{i+1}-1}{b-1} \\ &= \frac{1}{b-1}\sum_{i=0}^{d} b^{i+1} - 1 = \frac{1}{b-1}\left[\left(\sum_{i=0}^{d} b^{i+1}\right) - (d+1)\right] \end{aligned}$$

Noting that

$$bf_1(b,d) = b + b^2 + \cdots + b^{d+1} = \sum_{i=0}^{d} b^{i+1}$$

we have

$$f_2(b,d) = \frac{b}{b-1} f_1(b,d) - \frac{d+1}{b-1}$$

so $f_2(b,d)$ is about equal to $f_1(b,d)$ for large b.

Bidirectional search

In some problems we can simultaneously search:

forward from the *start* state

backward from the *goal* state

until the searches meet.

This is potentially a very good idea:

- If the search methods have complexity $O(b^d)$ then...
- ...we are converting this to $O(2b^{d/2}) = O(b^{d/2})$.

(Here, we are assuming the branching factor is b in both directions.)

Bidirectional search - beware!

- It is not always possible to generate efficiently *predecessors* as well as successors.
- If we only have the *description* of a goal, not an *explicit goal*, then generating predecessors can be hard. (For example, consider the concept of *checkmate*.)
- We need a way of checking whether or not a node appears in the *other search*...
- ... and the figure of $O(b^{d/2})$ hides the assumption that we can do *constant time* checking for intersection of the frontiers. (This may be possible using a hash table).
- We need to decide what kind of search to use in each half. For example, would *depth-first search* be sensible? Possibly not...
- ...to guarantee that the searches meet, we need to store all the nodes of at least one of the searches. Consequently the memory requirement is $O(b^{d/2})$.

Uniform-cost search

Breadth-first search finds the *shallowest* solution, but this is not necessarily the *best* one.

Uniform-cost search is a variant. It uses the *path cost* $p(n)$ as the priority for the priority queue.

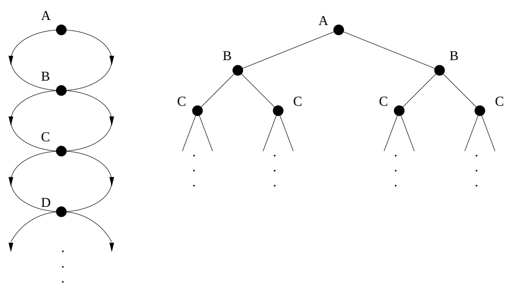
Thus, the paths that are apparently best are explored first, and the best solution will always be found if

$$\forall n\, (\forall n' \in \text{successors}(n)\; .\; p(n') \geq p(n))$$

Although this is still not a good practical algorithm, it does point the way forward to *informed search*...

Repeated states

With many problems it is easy to waste time by expanding nodes that have appeared elsewhere in the tree. For example:



The sliding blocks puzzle for example suffers this way.

Repeated states

For example, in a problem such as finding a route in a map, where all of the operators are *reversible*, this is inevitable.

There are three basic ways to avoid this, depending on how you trade off effectiveness against overhead.

- Never return to *the state you came from*.
- Avoid cycles: never proceed to *a state identical to one of your ancestors*.
- Do not expand *any state that has previously appeared*.

Graph search is a standard approach to dealing with the situation. It uses the last of these possibilities.

Graph search

In pseudocode:

```
function graphSearch() {
  closed = {};
  fringe = queue containing only the start state;
  while () {
    if (empty(fringe))
      return fail;
    node = head(fringe);
    if goal(node)
      return solution(node);
    if (node not a member of closed) {
      closed = closed + node;
      fringe = insert(expand(node), fringe); // See note...
    }
  }
}
```

Note: if `node` is in `closed` then it must already have been expanded.

Graph search

There are several points to note regarding graph search:

1. The *closed list* contains all the expanded nodes.
2. The closed list can be implemented using a hash table.
3. Both worst case time and space are now proportional to the size of the state space.
4. *Memory:* depth first and iterative deepening search are no longer linear space as we need to store the closed list.
5. *Optimality:* when a repeat is found we are discarding the new possibility even if it is better than the first one.
   - This never happens for uniform-cost or breadth-first search with constant step costs, so these remain optimal.
   - Iterative deepening search needs to check which solution is better and if necessary modify path costs and depths for descendants of the repeated state.

Search trees

Everything we've seen so far is an example of *uninformed* or *blind* search—we only distinguish goal states from non-goal states.

(Uniform cost search is a slight anomaly as it uses the path cost as a guide.)

To perform well in practice we need to employ *informed* or *heuristic* search.

This involves exploiting knowledge of the *distance between the current state and a goal*.

Problem solving by informed search

Basic search methods make limited use of any *problem-specific knowledge* we might have.

- We have already seen the concept of *path cost* $p(n)$

$$p(n) = \text{cost of path (sequence of actions) from the start state to } n$$

- We can now introduce an *evaluation function*. This is a function that attempts to measure the *desirability of each node*.

The evaluation function will clearly not be perfect. (If it is, there is no need to search.)

Best-first search simply expands nodes using the ordering given by the evaluation function.

Greedy search

We've already seen *path cost* used for this purpose.

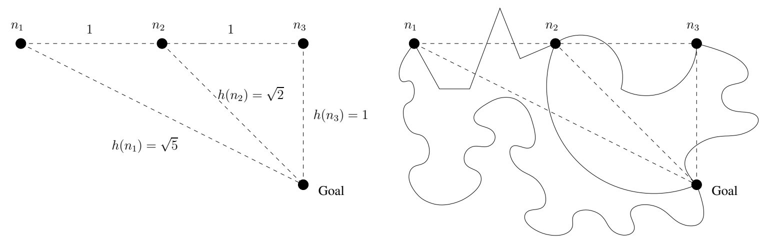
- This is misguided as path cost is not in general *directed* in any sense *toward the goal*.
- A *heuristic function*, usually denoted $h(n)$ is one that *estimates* the cost of the best path from any node n to a goal.
- If n is a goal then $h(n) = 0$.

Using a heuristic function along with best-first search gives us the *greedy search* algorithm.

Example: route-finding

Example: for route finding a reasonable heuristic function is

$$h(n) = \text{straight line distance from } n \text{ to the nearest goal}$$



Accuracy here obviously depends on what the roads are really like.

Example: route-finding

Greedy search suffers from some problems:

- Its time complexity is $O(b^d)$.
- Its space-complexity is $O(b^d)$.
- It is not optimal or complete.

BUT: greedy search *can* be effective, provided we have a good $h(n)$.

Wouldn't it be nice if we could improve it to make it optimal and complete?

$A^\star$ search

Well, we can.

$A^\star$ *search* combines the good points of:

- Greedy search—by making use of $h(n)$.
- Uniform-cost search—by being optimal and complete.

It does this in a very simple manner: it uses path cost $p(n)$ and also the heuristic function $h(n)$ by forming

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

where

$$p(n) = \text{cost of path } \textit{to } n$$

and

$$h(n) = \text{estimated cost of best path } \textit{from } n$$

So: $f(n)$ is the estimated cost of a path *through* n.

$A^\star$ search

$A^\star$ search:

- A best-first search using $f(n)$.
- It is both complete and optimal...
- ...provided that h obeys some simple conditions.

Definition: an *admissible heuristic* $h(n)$ is one that *never overestimates* the cost of the best path from n to a goal. So if $h'(n)$ denotes the *actual* distance from n to the goal we have

$$\forall n. h(n) \leq h'(n).$$

If $h(n)$ is admissible then *tree-search* $A^\star$ is optimal.

$A^\star$ tree-search is optimal for admissible $h(n)$

To see that $A^\star$ search is optimal we reason as follows.

Let Goal_{opt} be an optimal goal state with

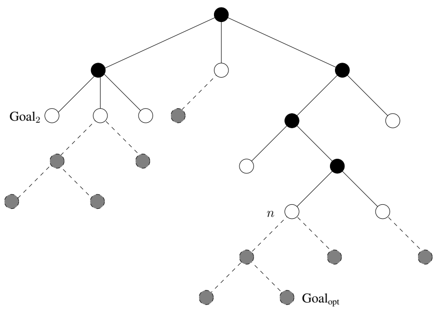
$$f(\text{Goal}_{\text{opt}}) = p(\text{Goal}_{\text{opt}}) = f_{\text{opt}}$$

(because $h(\text{Goal}_{\text{opt}}) = 0$). Let Goal_2 be a suboptimal goal state with

$$f(\text{Goal}_2) = p(\text{Goal}_2) = f_2 > f_{\text{opt}}$$

We need to demonstrate that the search can never select Goal_2.

$A^\star$ tree-search is optimal for admissible $h(n)$



At some point Goal$_2$ is in the fringe.

Can it be selected before n?

$A^\star$ tree-search is optimal for admissible $h(n)$

Let n be a leaf node in the fringe on an optimal path to Goal$_{\text{opt}}$. So

$$f_{\text{opt}} \geq p(n) + h(n) = f(n)$$

because h is admissible.

Now say Goal$_2$ is chosen for expansion *before* n. This means that

$$f(n) \geq f_2$$

so we've established that

$$f_{\text{opt}} \geq f_2 = p(\text{Goal}_2).$$

But this means that Goal$_{\text{opt}}$ is not optimal: a contradiction.

$A^\star$ graph search

Of course, we will generally be dealing with *graph search*.

Unfortunately the proof breaks in this case.

- Graph search can *discard an optimal* route if that route is not the first one generated.
- We could keep *only the least expensive path*. This means updating, which is extra work, not to mention messy, but sufficient to insure optimality.
- Alternatively, we can impose a further condition on $h(n)$ which *forces the best path to a repeated state to be generated first*.

The required condition is called *monotonicity*. As

$$\text{monotonicity} \longrightarrow \text{admissibility}$$

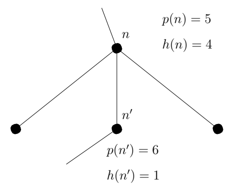
this is an important property.

Monotonicity

Assume h is admissible. Remember that $f(n) = p(n) + h(n)$ so if n' follows n

$$p(n') \geq p(n)$$

and we expect that $h(n') \leq h(n)$ although this does not have to be the case.



Here $f(n) = 9$ and $f(n') = 7$ so $f(n') < f(n)$.

Monotonicity

Monotonicity:

- If it is always the case that $f(n') \geq f(n)$ then $h(n)$ is called *monotonic*.
- $h(n)$ is monotonic if and only if it obeys the *triangle inequality*.

$$h(n) \leq \text{cost}(n \xrightarrow{a} n') + h(n')$$

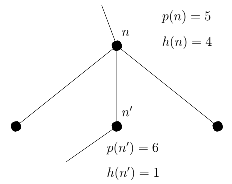
If $h(n)$ is *not* monotonic we can make a simple alteration and use

$$f(n') = \max\{f(n), p(n') + h(n')\}$$

This is called the *pathmax* equation.

The pathmax equation

Why does the pathmax equation make sense?



The fact that $f(n) = 9$ tells us the cost of a path through n is *at least* 9 (because $h(n)$ is admissible).

But n' is *on a path through* n. So to say that $f(n') = 7$ makes no sense.

$A^\star$ graph search is optimal for monotonic heuristics

$A^\star$ graph search is optimal for monotonic heuristics.

The crucial fact from which optimality follows is that if $h(n)$ is monotonic then the values of $f(n)$ along any path are non-decreasing.

Assume we move from n to n' using action a. Then

$$\forall a \;.\; p(n') = p(n) + \text{cost}(n \xrightarrow{a} n')$$

and using the triangle inequality

$$h(n) \leq \text{cost}(n \xrightarrow{a} n') + h(n') \quad (1)$$

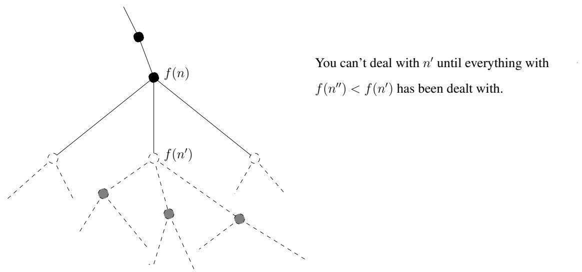
Thus

$$\begin{aligned} f(n') &= p(n') + h(n') \\ &= p(n) + \text{cost}(n \xrightarrow{a} n') + h(n') \\ &\geq p(n) + h(n) \\ &= f(n) \end{aligned}$$

where the inequality follows from equation 1.

$A^\star$ graph search is optimal for monotonic heuristics

We therefore have the following situation:



Consequently everything with $f(n'') < f_{\text{opt}}$ gets explored. Then one or more things with f_{opt} get found (not necessarily all goals).

$A^\star$ search is complete

$A^\star$ search is complete provided:

1. The graph has finite branching factor.
2. There is a finite, positive constant c such that each operator has cost at least c.

Why is this? The search expands nodes according to increasing $f(n)$. So: the only way it can fail to find a goal is if there are infinitely many nodes with $f(n) < f(\text{Goal})$.

There are two ways this can happen:

1. There is a node with an infinite number of descendants.
2. There is a path with an infinite number of nodes but a finite path cost.

Complexity

- $A^\star$ search has a further desirable property: it is *optimally efficient*.
- This means that no other optimal algorithm that works by constructing paths from the root can guarantee to examine fewer nodes.
- BUT: despite its good properties we're not done yet...
- ...$A^\star$ search unfortunately still has exponential time complexity in most cases unless $h(n)$ satisfies a very stringent condition that is generally unrealistic:

$$|h(n) - h'(n)| \leq O(\log h'(n))$$

  where $h'(n)$ denotes the *real* cost from n to the goal.
- As $A^\star$ search also stores all the nodes it generates, once again it is generally *memory that becomes a problem before time*.

IDA* - iterative deepening $A^\star$ search

How might we improve the way in which $A^\star$ search uses memory?

- Iterative deepening search used depth-first search with a limit on depth that is gradually increased.
- *IDA** does the same thing *with a limit on f cost*.

```
ActionSequence ida() {
   root = root node for problem;
   float fLimit = f(root);
   while() {
      (sequence, fLimit) = contour(root,fLimit,emptySequence);
      if (sequence != emptySequence)
        return sequence;
      if (fLimit == infinity)
        return emptySequence;
   }
}
```

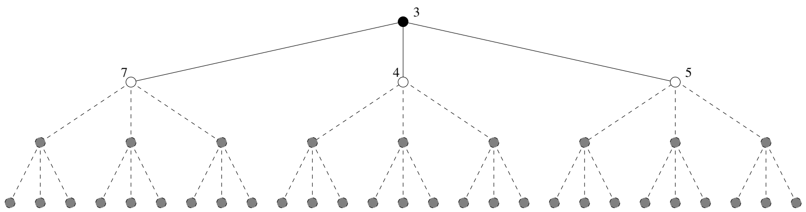

IDA* - iterative deepening $A^\star$ search

The function `contour` searches from a given node, *as far as the specified f limit*. It returns either a solution, or the *next biggest* value of f to try.

```
(ActionSequence,float) contour(Node node, float fLimit, ActionSequence s) {
   float nextF = infinity;
   if (f(node) > fLimit)
      return (emptySequence,f(node));
   ActionSequence s' = addToSequence(node,s);
   if (goalTest(node))
      return (s',fLimit);
   for (each successor n' of node) {
      (sequence,newF) = contour(n',fLimit,s');
      if (sequence != emptySequence)
         return (sequence,fLimit);
      nextF = minimum(nextF,newF);
   }
   return (emptySequence,nextF);
}
```

IDA* - iterative deepening $A^\star$ search

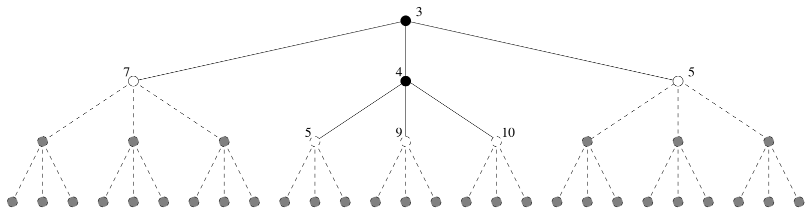
This is a little tricky to unravel, so here is an example:



Initially, the algorithm looks ahead and finds the *smallest* f cost that is *greater than* its current f cost limit. The new limit is 4.

IDA* - iterative deepening $A^\star$ search

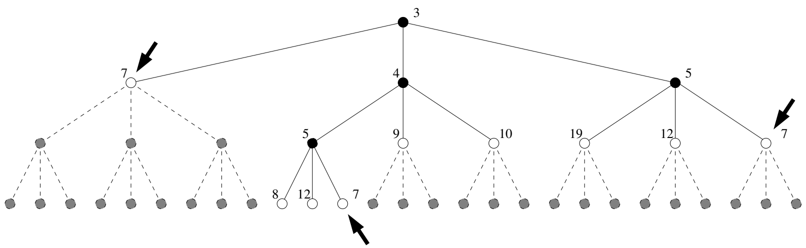
It now does the same again:



Anything with f cost *at most* equal to the current limit gets explored, and the algorithm keeps track of the *smallest* f cost that is *greater than* its current limit. The new limit is 5.

IDA* - iterative deepening $A^\star$ search

And again:



The new limit is 7, so at the next iteration the three arrowed nodes will be explored.

IDA* - iterative deepening $A^\star$ search

Properties of IDA*:

- It is complete and optimal under the same conditions as $A^\star$.
- It is often good if we have step costs equal to 1.
- It does not require us to maintain a sorted queue of nodes.
- It only requires *space proportional to the longest path*.
- The time taken depends on the number of values h can take.

If h takes enough values to be problematic we can increase f by a fixed ϵ at each stage, guaranteeing a solution at most ϵ worse than the optimum.

Recursive best-first search (RBFS)

Another method by which we can attempt to overcome memory limitations is the *Recursive best-first search (RBFS)*.

Idea: try to do a best-first search, but only use *linear space* by doing a depth-first search with a few modifications:

1. We remember the $f(n')$ for the best alternative node n' we've seen so far on the way to the node n we're currently considering.
2. If n has $f(n) > f(n')$:
   - We go back and explore the best alternative...
   - ...and as we retrace our steps we replace the f cost of every node we've seen in the current path with $f(n)$.

The replacement of f values as we retrace our steps provides a means of remembering how good a discarded path might be, so that we can easily return to it later.

Recursive best-first search (RBFS)

Note: for simplicity a parameter for the path has been omitted.

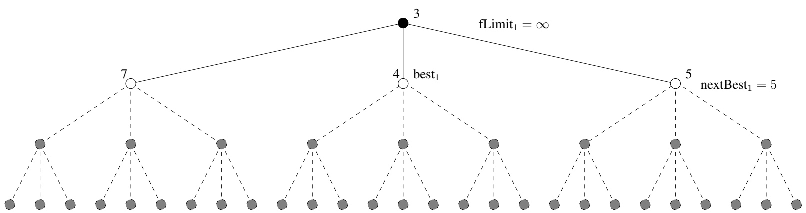
```
function RBFS(Node n, Float fLimit) {
  if (goaltest(n))
    return n;
  if (n has no successors)
    return (fail, infinity);
  for (each successor n' of n)
    f(n') = maximum(f(n'), f(n));
  while() {
    best = successor of n that has the smallest f(n');
    if (f(best) > fLimit)
      return (fail, f(best));
    nextBest = second smallest f(n') value for successors of n;
    (result, f') = RBFS(best, minimum(fLimit, nextBest));
    f(best) = f';
    if (result != fail)
      return result;
  }
}
```

IMPORTANT: `f(best)` is *modified* when `RBFS` produces a result.

Recursive best-first search (RBFS): an example

This function is called using `RBFS(startState, infinity)` to begin the process.

Function call number 1:

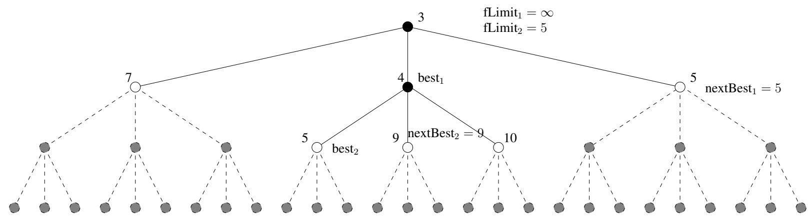


Now perform the recursive function call $(\text{result}_2, f') = \text{RBFS}(\text{best}_1, 5)$

so $f(\text{best}_1)$ takes the returned value f'

Recursive best-first search (RBFS): an example

Function call number 2:

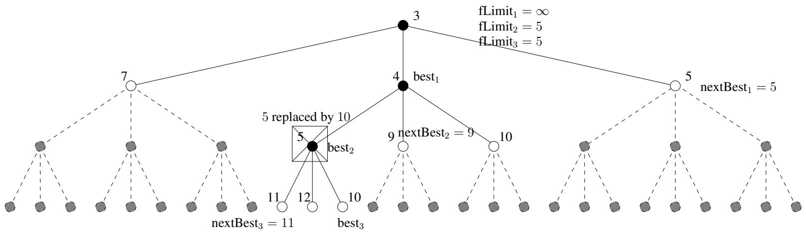


Now perform the recursive function call $(\text{result}_3, f') = \text{RBFS}(\text{best}_2, 5)$

so $f(\text{best}_2)$ takes the returned value f'

Recursive best-first search (RBFS): an example

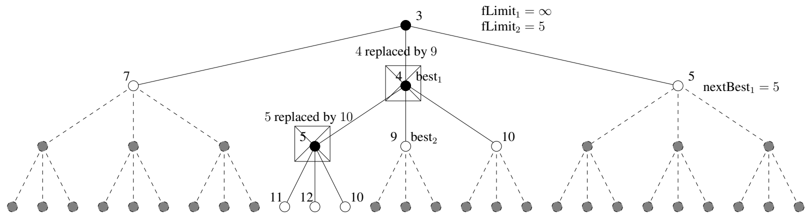
Function call number 3:



Now $f(\text{best}_3) > \text{fLimit}_3$ so the function call returns $(\text{fail}, 10)$ into (result_3, f') and $f(\text{best}_2) = 10$.

Recursive best-first search (RBFS): an example

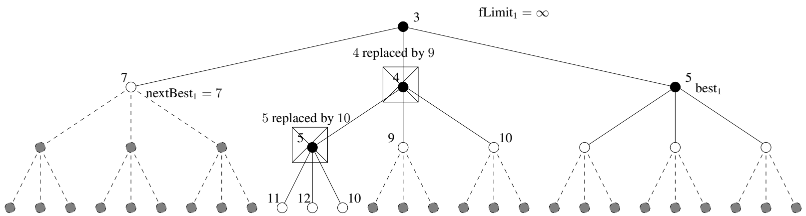
The while loop for function call 2 now repeats:



Now $f(\text{best}_2) > \text{fLimit}_2$ so the function call returns $(\text{fail}, 9)$ into (result_2, f') and $f(\text{best}_1) = 9$.

Recursive best-first search (RBFS): an example

The while loop for function call 1 now repeats:



We do a further function call to expand the new best node, and so on...

Recursive best-first search (RBFS)

Some nice properties:

- If h is admissible then RBFS is optimal.
- Memory requirement is $O(bd)$
- Generally more efficient than IDA*.

And some less nice ones:

- Time complexity is hard to analyse, but can be exponential.
- Can spend a lot of time *re-generating nodes*.

Other methods for getting around the memory problem

To some extent IDA* and RBFS throw the baby out with the bathwater.

- They limit memory too harshly, so...
- ...we can try to use *all available memory*.

MA* and SMA* will not be covered in this course...

Local search

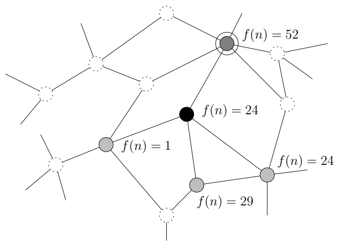
Sometimes, it's only the *goal* that we're interested in. The *path* needed to get there is irrelevant.

- For example: VLSI layout, factory design, vehicle routing, automatic programming...
- We are now simply searching for a node that is in some sense *the best*.
- This is also known as *optimisation*.

This leads to the remarkably simple concept of *local search*.

Local search

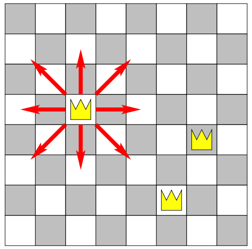
Instead of trying to find a path from start state to goal, we explore the *local area* of the graph, meaning those nodes one edge away from the one we're at.



We assume that we have a function $f(n)$ such that $f(n') > f(n)$ indicates n' is preferable to n.

The n-queens problem

You may be familiar with the *n-queens problem*.



Find an arrangement of n queens on an n by n board such that no queen is attacking another.

In the Prolog course you may have been tempted to generate permutations of row numbers and test for attacks.

This is a *hopeless strategy* for large n. (Imagine $n \simeq 1,000,000$.)

The n-queens problem

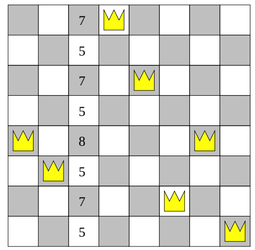
We might however consider the following:

- A state (node) n for an m by m board is a sequence of m numbers drawn from the set $\{1, \ldots, m\}$, possibly including repeats.
- We move from one node to another by moving a *single queen* to *any* alternative row.
- We define $f(n)$ to be the number of pairs of queens attacking one-another in the new position[2]. (Regardless of whether or not the attack is direct.)

[2] Note that we actually want to *minimize* f here. This is equivalent to maximizing $-f$, and I will generally use whichever seems more appropriate.

The n-queens problem

Here, $n = \{4, 3, ?, 8, 6, 2, 4, 1\}$ and the f values for the undecided queen are shown.



As we can choose which queen to move, each node in fact has 56 neighbours in the graph.

Hill-climbing search

Hill-climbing search is remarkably simple:

```
Generate a start state n.
while () {
  Generate the N neighbours {n_1,...,n_N} of n;
  if (max(f(n_i)) <= f(n)) return n;
  n = n_i maximizing f(n_i);
}
```

In fact, that looks so simple that it's amazing the algorithm is at all useful.

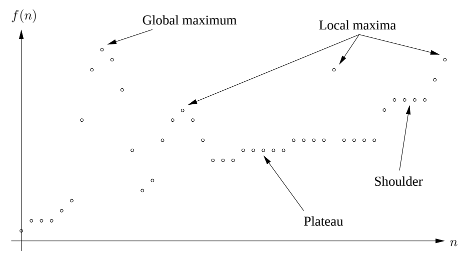
In this version we stop when we get to a node with no better neighbour. We might alternatively allow *sideways moves* by changing the stopping condition:

```
if (max(f(n_i)) < f(n)) return n;
```

Why would we consider doing this?

Hill-climbing search: the reality

In reality, nature has a number of ways of shaping f to complicate the search process.



Sideways moves allow us to move across *plateaus* and *shoulders*.

However, should we ever find a *local maximum* then we'll return it: we won't keep searching to find a *global maximum*.

Hill-climbing search: the reality

Of course, the fact that we're dealing with a *general graph* means we need to think of something like the preceding figure, but in a *very large number of dimensions*, and this makes the problem *much harder*.

There is a body of techniques for trying to overcome such problems. For example:

- *Stochastic hill-climbing:* Choose a neighbour at random, perhaps with a probability depending on its f value. For example: let $N(n)$ denote the neighbours of n. Define

$$N^{+}(n) = \{n' \in N(n) | f(n') \geq f(n)\}$$
$$N^{-}(n) = \{n' \in N(n) | f(n') < f(n)\}.$$

  Then

$$\Pr(n') = \begin{cases} 0 & \text{if } n' \in N^{-}(n) \\ \frac{1}{Z}(f(n') - f(n)) & \text{otherwise.} \end{cases}$$

Hill-climbing search: the reality

- *First choice:* Generate neighbours at random. Select the first one that is better than the current one. (Particularly good if nodes have *many neighbours*.)
- *Random restarts:* Run a procedure k times with a limit on the time allowed for each run.

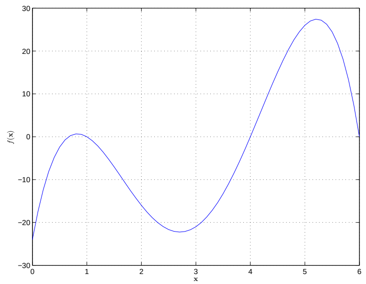
  Note: generating a start state at random may itself not be straightforward.
- *Simulated annealing:* Similar to stochastic hill-climbing, but start with lots of random variation and *reduce it over time*.

  Note: in some cases this is *provably* an effective procedure, although the time taken may be excessive if we want the proof to hold.
- *Beam search:* Maintain k nodes at any given time. At each search step, find the successors of each, and retain the best k from *all* the successors.

  Note: this is *not* the same as random restarts.

Gradient ascent and related methods

For some problems[3]—we do not have a search graph, but a *continuous search space*.



Typically, we have a function $f(\mathbf{x}) : \mathbb{R}^n \to \mathbb{R}$ and we want to find

$$\mathbf{x}_{\text{opt}} = \underset{\mathbf{x}}{\operatorname{argmax}}\, f(\mathbf{x})$$

[3]For the purposes of this course, the *training of neural networks* is a notable example.

Gradient ascent and related methods

In a single dimension we can clearly try to solve

$$\frac{df(x)}{dx} = 0$$

to find the *stationary points*, and use

$$\frac{d^2 f(x)}{dx^2}$$

to find a global *maximum*. In *multiple dimensions* the equivalent is to solve

$$\nabla f(\mathbf{x}) = \frac{\partial f(\mathbf{x})}{\partial \mathbf{x}} = \mathbf{0}$$

where

$$\frac{\partial f(\mathbf{x})}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial f(\mathbf{x})}{\partial x_1} & \frac{\partial f(\mathbf{x})}{\partial x_2} & \cdots & \frac{\partial f(\mathbf{x})}{\partial x_n} \end{bmatrix}.$$

and the equivalent of the second derivative is the *Hessian* matrix

$$\mathbf{H} = \begin{bmatrix} \frac{\partial f^2(\mathbf{x})}{\partial x_1^2} & \frac{\partial f^2(\mathbf{x})}{\partial x_1 \partial x_2} & \cdots & \frac{\partial f^2(\mathbf{x})}{\partial x_1 \partial x_n} \\ \frac{\partial f^2(\mathbf{x})}{\partial x_2 \partial x_1} & \frac{\partial f^2(\mathbf{x})}{\partial x_2^2} & \cdots & \frac{\partial f^2(\mathbf{x})}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f^2(\mathbf{x})}{\partial x_n \partial x_1} & \frac{\partial f^2(\mathbf{x})}{\partial x_n \partial x_2} & \cdots & \frac{\partial f^2(\mathbf{x})}{\partial x_n^2} \end{bmatrix}.$$

Gradient ascent and related methods

However this approach is usually *not analytically tractable* regardless of dimensionality.

The simplest way around this is to employ *gradient ascent*:

- Start with a randomly chosen point $\mathbf{x}_0$.
- Using a small *step size* ϵ, iterate using the equation

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \epsilon \nabla f(\mathbf{x}_i).$$

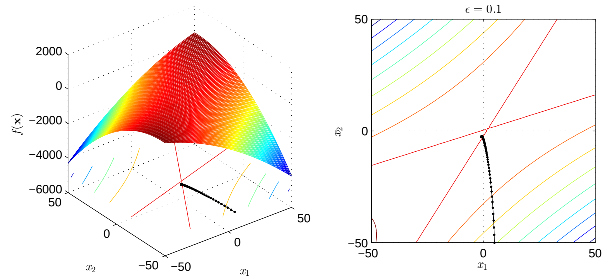
This can be understood as follows:

- At the current point $\mathbf{x}_i$ the gradient $\nabla f(\mathbf{x}_i)$ tells us the *direction* and *magnitude* of the slope at $\mathbf{x}_i$.
- Adding $\epsilon \nabla f(\mathbf{x}_i)$ therefore moves us a *small distance upward*.

This is perhaps more easily seen graphically...

Gradient ascent and related methods

Here we have a simple *parabolic surface*:

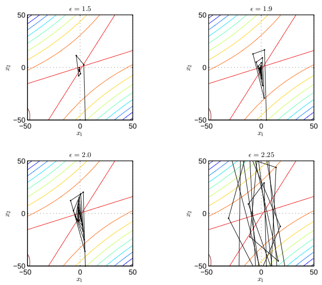


With $\epsilon = 0.1$ the procedure is clearly effective at finding the maximum.

Note however that *the steps are small*, and in a more realistic problem *it might take some time...*

Gradient ascent and related methods

Simply increasing the step size ϵ can lead to a different problem:



We can easily jump too far...

Gradient ascent and related methods

There is a large collection of more sophisticated methods. For example:

- *Line search:* increase ϵ until f *increases* and minimise in the resulting interval. Then choose a new direction to move in. *Conjugate gradients*, the *Fletcher-Reeves* and *Polak-Ribiere* methods etc.
- Use $\mathbf{H}$ to exploit knowledge of the local shape of f. For example the *Newton-Raphson* and *Broyden-Fletcher-Goldfarb-Shanno (BFGS)* methods etc.

Artificial Intelligence I

Dr Sean Holden

Notes on *games (adversarial search)*

Solving problems by search: playing games

How might an agent act when *the outcomes of its actions are not known* because an *adversary is trying to hinder it*?

- This is essentially a more realistic kind of search problem because we do not know the exact outcome of an action.
- This is a common situation when *playing games*: in chess, draughts, and so on an opponent *responds* to our moves.
- We don't know what their response will be, and so the outcome of our moves is not clear.

Game playing has been of interest in AI because it provides an *idealisation* of a world in which two agents act to *reduce* each other's well-being.

Playing games: search against an adversary

Despite the fact that games are an idealisation, game playing can be an excellent source of hard problems. For instance with chess:

- The average branching factor is roughly 35.
- Games can reach 50 moves per player.
- So a rough calculation gives the search tree 35^{100} nodes.
- Even if only different, legal positions are considered it's about 10^{40}.

So: in addition to the uncertainty due to the opponent:

- We can't make a complete search to find the best move...
- ... so we have to act even though we're not sure about the best thing to do.

Playing games: search against an adversary

And chess isn't even very hard:

- *Go* is *much* harder than chess.
- The branching factor is about 360.

Until very recently it has resisted all attempts to produce a good AI player.

See:

`senseis.xmp.net/?MoGo`

and others.

Playing games: search against an adversary

It seems that games are a step closer to the complexities inherent in the world around us than are the standard search problems considered so far.

The study of games has led to some of the most celebrated applications and techniques in AI.

We now look at:

- How game-playing can be modelled as *search*.
- The *minimax algorithm* for game-playing.
- Some problems inherent in the use of minimax.
- The concept of $\alpha - \beta$ *pruning*.

Reading: Russell and Norvig chapter 6.

Perfect decisions in a two-person game

Say we have two players. Traditionally, they are called *Max* and *Min* for reasons that will become clear.

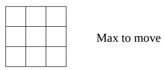
- We'll use *noughts and crosses* as an initial example.
- Max moves first.
- The players alternate until the game ends.
- At the end of the game, prizes are awarded. (Or punishments administered—EVIL ROBOT is starting up his favourite chainsaw...)

This is exactly the same game format as chess, Go, draughts and so on.

Perfect decisions in a two-person game

Games like this can be modelled as search problems as follows:

- There is an *initial state*.

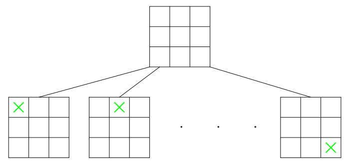


- There is a set of *operators*. Here, Max can place a cross in any empty square, or Min a nought.
- There is a *terminal test*. Here, the game ends when three noughts or three crosses are in a row, or there are no unused spaces.
- There is a *utility* or *payoff* function. This tells us, numerically, what the outcome of the game is.

This is enough to model the entire game.

Perfect decisions in a two-person game

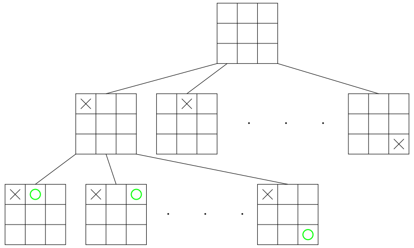
We can *construct a tree* to represent a game. From the initial state Max can make nine possible moves:



Then it's Min's turn...

Perfect decisions in a two-person game

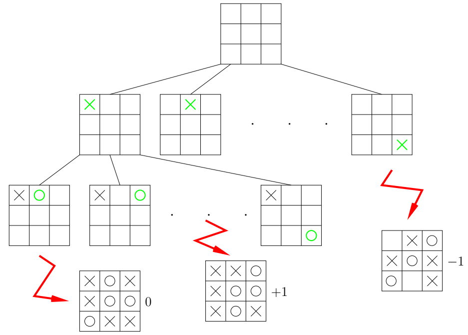
For each of Max's opening moves Min has eight replies:



And so on...

This can be continued to represent *all* possibilities for the game.

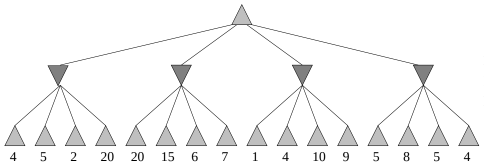
Perfect decisions in a two-person game



At the leaves a player has won or there are no spaces. Leaves are *labelled* using the utility function.

Perfect decisions in a two-person game

How can Max use this tree to decide on a move? Consider a much simpler tree:



Labels on the leaves denote utility.
High values are preferred by Max.
Low values are preferred by Min.

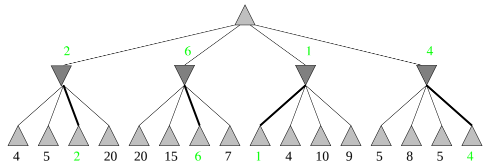
If Max is rational he will play to reach a position with the *biggest utility possible*

But if Min is rational she will play to *minimise* the utility available to Max.

The minimax algorithm

There are two moves: Max then Min. Game theorists would call this one move, or two *ply* deep.

The *minimax algorithm* allows us to infer the best move that the current player can make, given the utility function, by working backward from the leaves.



As Min plays the last move, she *minimises* the utility available to Max.

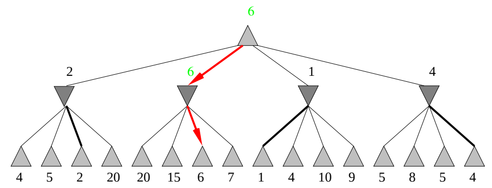
The minimax algorithm

Min takes the final move:

- If Min is in game position 1, her best choice is move 3. So from Max's point of view this node has a utility of 2.
- If Min is in game position 2, her best choice is move 3. So from Max's point of view this node has a utility of 6.
- If Min is in game position 3, her best choice is move 1. So from Max's point of view this node has a utility of 1.
- If Min is in game position 4, her best choice is move 4. So from Max's point of view this node has a utility of 4.

The minimax algorithm

Moving one further step up the tree:



We can see that Max's best opening move is move 2, as this leads to the node with highest utility.

The minimax algorithm

In general:

- Generate the complete tree and label the leaves according to the utility function.
- Working from the leaves of the tree upward, label the nodes depending on whether Max or Min is to move.
- If *Min* is to move label the current node with the *minimum* utility of any descendant.
- If *Max* is to move label the current node with the *maximum* utility of any descendant.

If the game is p ply and at each point there are q available moves then this process has (surprise, surprise) $O(q^p)$ time complexity and space complexity linear in p and q.

Making imperfect decisions

We need to avoid searching all the way to the end of the tree. *So:*

- We generate only part of the tree: instead of testing whether a node is a leaf we introduce a *cut-off* test telling us when to stop.
- Instead of a utility function we introduce an *evaluation function* for the evaluation of positions for an incomplete game.

The evaluation function attempts to measure the expected utility of the current game position.

Making imperfect decisions

How can this be justified?

- This is a strategy that humans clearly sometimes make use of.
- For example, when using the concept of *material value* in chess.
- The effectiveness of the evaluation function is *critical*...
- ... but it must be computable in a reasonable time.
- (In principle it could just be done using minimax.)

The importance of the evaluation function can not be understated—it is probably the most important part of the design.

The evaluation function

Designing a good evaluation function can be extremely tricky:

- Let's say we want to design one for chess by giving each piece its material value: pawn = 1, knight/bishop = 3, rook = 5 and so on.
- Define the evaluation of a position to be the difference between the material value of black's and white's pieces

$$\text{eval}(\text{position}) = \sum_{\text{black's pieces } p_i} \text{value of } p_i - \sum_{\text{white's pieces } q_i} \text{value of } q_i$$

This seems like a reasonable first attempt. Why might it go wrong?

The evaluation function

Consider what happens at the start of a game:

- Until the first capture the evaluation function gives 0, so in fact we have a *category* containing many different game positions with equal estimated utility.
- For example, all positions where white is one pawn ahead.
- The evaluation function for such a category should perhaps represent the probability that a position chosen at random from it leads to a win.

So in fact this seems highly naive...

The evaluation function

Ideally, we should consider *individual positions*.

If on the basis of past experience a position has 50% chance of winning, 10% chance of losing and 40% chance of reaching a draw, we might give it an evaluation of

$$\text{eval}(\text{position}) = (0.5 \times 1) + (0.1 \times -1) + (0.4 \times 0) = 0.4.$$

Extending this to the evaluation of categories, we should then weight the positions in the category according to their likelihood of occurring.

Of course, we *don't know* what any of these likelihoods are...

The evaluation function

Using material value can be thought of as giving us a *weighted linear evaluation function*

$$\text{eval}(\text{position}) = \sum_{i=1}^{n} w_i f_i$$

where the w_i are *weights* and the f_i represent *features* of the position. In this example

$$f_i = \text{value of the } i\text{th piece}$$

$$w_i = \text{number of } i\text{th pieces on the board}$$

where black and white pieces are regarded as different and the f_i are positive for one and negative for the other.

The evaluation function

Evaluation functions of this type are very common in game playing.

There is no systematic method for their design.

Weights can be chosen by allowing the game to play itself and using *learning* techniques to adjust the weights to improve performance.

By using more carefully crafted features we can give *different evaluations* to *individual positions*.

$\alpha - \beta$ pruning

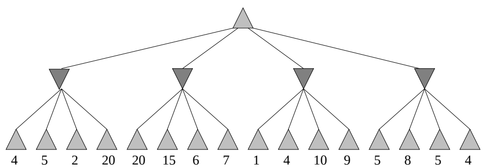
Even with a good evaluation function and cut-off test, the time complexity of the minimax algorithm makes it impossible to write a good chess program without some further improvement.

- Assuming we have 150 seconds to make each move, for chess we would be limited to a search of about 3 to 4 ply whereas...
- ...even an average human player can manage 6 to 8.

Luckily, it is possible to prune the search tree *without affecting the outcome* and *without having to examine all of it*.

$\alpha - \beta$ pruning

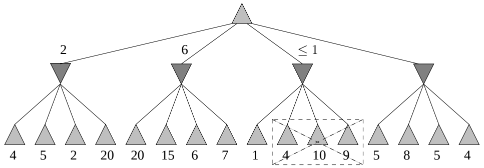
Returning for a moment to the earlier, simplified example:



The search is depth-first and left to right.

$\alpha - \beta$ pruning

The search continues as previously for the first 8 leaves.

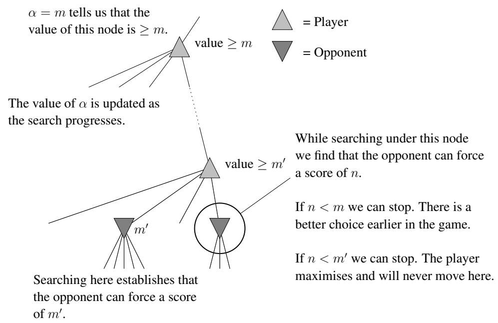


Then we note: if *Max* plays move 3 then *Min* can reach a leaf with utility at most 1.

So: *we don't need to search any further under Max's opening move* 3. This is because the search has *already established* that *Max* can do better by making opening move 2.

$\alpha - \beta$ pruning in general

Remember that this search is *depth-first*. We're only going to use knowledge of *nodes on the current path*.



So: once you've established that n is sufficiently small, you don't need to explore any more of the corresponding node's children.

$\alpha - \beta$ pruning in general

The situation is exactly analogous if we *swap player and opponent* in the previous diagram.

The search is depth-first, so we're only ever looking at *one path through the tree*.

We need to keep track of the values α and β where

$$\alpha = \text{the } \textit{highest} \text{ utility seen so far on the path for } \textit{Max}$$

$$\beta = \text{the } \textit{lowest} \text{ utility seen so far on the path for } \textit{Min}$$

Assume *Max begins*. Initial values for α and β are

$$\alpha = -\infty$$

and

$$\beta = +\infty.$$

$\alpha - \beta$ pruning in general

So: we start with the function call

$$\texttt{player}(-\infty, +\infty, \texttt{root})$$

The following function implements the procedure suggested by the previous diagram:

player$(\alpha, \beta, n)\{$
  if$(n$ is at the cut-off point $)$ return evaluation(n);
  value $= -\infty$;
  for(each successor n' of $n)\{$
    value $=$ max(value, opponent(α, β, n'));
    if(value $> \beta$) return value;
    if(value $> \alpha$) $\alpha =$ value;
  $\}$
  return value;
$\}$

$\alpha - \beta$ pruning in general

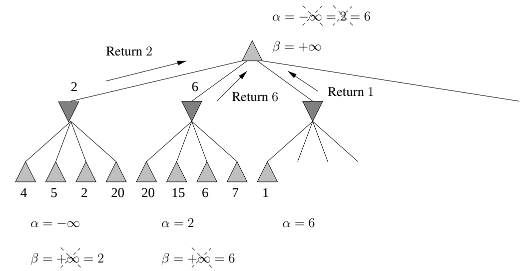
The function `opponent` is exactly analogous:

opponent$(\alpha, \beta, n)\{$
    if$(n$ is at the cut-off point) return evaluation(n);
    value $= +\infty$;
    for(each successor n' of n)$\{$
        value $=$ min(value, player(α, β, n'));
        if(value $< \alpha$) return value;
        if(value $< \beta$) $\beta =$ value;
    $\}$
    return value;
$\}$

Note: the semantics here is that parameters are passed to functions *by value*.

$\alpha - \beta$ pruning in general

Applying this to the earlier example and keeping track of the values for α and β you should obtain:



How effective is $\alpha - \beta$ pruning?

(Warning: the theoretical results that follow are somewhat idealised.)

A quick inspection should convince you that the *order* in which moves are arranged in the tree is critical.

So, it seems sensible to try good moves first:

- If you were to have a perfect move-ordering technique then $\alpha - \beta$ pruning would be $O(q^{p/2})$ as opposed to $O(q^p)$.
- so the branching factor would effectively be $\sqrt{q}$ instead of q.
- We would therefore expect to be able to search ahead *twice as many moves as before*.

However, this is not realistic: if you had such an ordering technique you'd be able to play perfect games!

How effective is $\alpha - \beta$ pruning?

If moves are arranged at random then $\alpha - \beta$ pruning is:

- $O((q/\log q)^p)$ asymptotically when $q > 1000$ or...
- ...about $O(q^{3p/4})$ for reasonable values of q.

In practice simple ordering techniques can get close to the best case. For example, if we try captures, then threats, then moves forward *etc*.

Alternatively, we can implement an iterative deepening approach and use the order obtained at one iteration to drive the next.

A further optimisation: the transposition table

Finally, note that many games correspond to *graphs* rather than *trees* because the same state can be arrived at in different ways.

- This is essentially the same effect we saw in heuristic search: recall *graph search* versus *tree search*.
- It can be addressed in a similar way: store a state with its evaluation in a hash table—generally called a *transposition table*—the first time it is seen.

The transposition table is essentially equivalent to the *closed list* introduced as part of graph search.

This can vastly increase the effectiveness of the search process, because we don't have to evaluate a single state multiple times.

Artificial Intelligence I

Dr Sean Holden

Notes on *constraint satisfaction problems (CSPs)*

Constraint satisfaction problems (CSPs)

The search scenarios examined so far seem in some ways unsatisfactory.

- States were represented using an *arbitrary* and *problem-specific* data structure.
- Heuristics were also *problem-specific*.
- It would be nice to be able to *transform* general search problems into a *standard format*.

CSPs *standardise* the manner in which states and goal tests are represented...

Constraint satisfaction problems (CSPs)

By standardising like this we benefit in several ways:

- We can devise *general purpose* algorithms and heuristics.
- We can look at general methods for exploring the *structure* of the problem.
- Consequently it is possible to introduce techniques for *decomposing* problems.
- We can try to understand the relationship between the *structure* of a problem and the *difficulty of solving it*.

Note: another method of interest in AI that allows us to do similar things involves transforming to a *propositional satisfiability* problem. We'll see an example of this in AI II.

Introduction to constraint satisfaction problems

We now return to the idea of problem solving by search and examine it from this new perspective.

Aims:

- To introduce the idea of a constraint satisfaction problem (CSP) as a general means of representing and solving problems by search.
- To look at a *backtracking algorithm* for solving CSPs.
- To look at some *general heuristics* for solving CSPs.
- To look at *more intelligent ways of backtracking*.

Reading: Russell and Norvig, chapter 5.

Constraint satisfaction problems

We have:

- A set of n *variables* $V_1, V_2, \ldots, V_n$.
- For each V_i a *domain* D_i specifying the values that V_i can take.
- A set of m *constraints* $C_1, C_2, \ldots, C_m$.

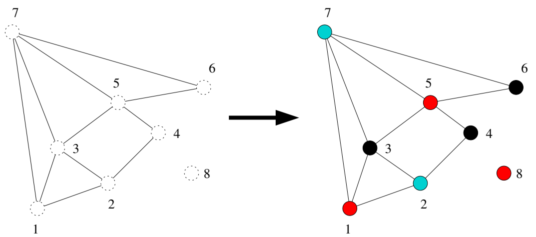
Each constraint C_i involves a set of variables and specifies an *allowable collection of values*.

- A *state* is an assignment of specific values to some or all of the variables.
- An assignment is *consistent* if it violates no constraints.
- An assignment is *complete* if it gives a value to every variable.

A *solution* is a consistent and complete assignment.

Example

We will use the problem of *colouring the nodes of a graph* as a running example.



Each node corresponds to a *variable*. We have three colours and directly connected nodes should have different colours.

Example

This translates easily to a CSP formulation:

- The variables are the nodes

$$V_i = \text{node } i$$

- The domain for each variable contains the values black, red and cyan

$$D_i = \{B, R, C\}$$

- The constraints enforce the idea that directly connected nodes must have different colours. For example, for variables V_1 and V_2 the constraints specify

$$(B, R), (B, C), (R, B), (R, C), (C, B), (C, R)$$

- Variable V_8 is unconstrained.

Different kinds of CSP

This is an example of the simplest kind of CSP: it is *discrete* with *finite domains*. We will concentrate on these.

We will also concentrate on *binary constraints*; that is, constraints between *pairs of variables*.

- Constraints on single variables—*unary constraints*—can be handled by adjusting the variable's domain. For example, if we don't want V_i to be *red*, then we just remove that possibility from D_i.
- *Higher-order constraints* applying to three or more variables can certainly be considered, but...
- ...when dealing with finite domains they can always be converted to sets of binary constraints by introducing extra *auxiliary variables*.

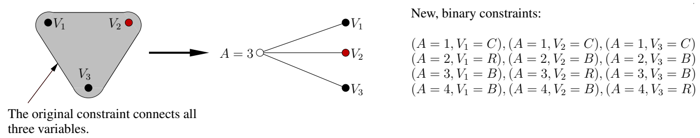
How does that work?

Auxiliary variables

Example: three variables each with domain $\{B, R, C\}$.

A single constraint

$$(C, C, C), (R, B, B), (B, R, B), (B, B, R)$$



Introducing auxiliary variable A with domain $\{1, 2, 3, 4\}$ allows us to convert this to a set of binary constraints.

Backtracking search

Consider what happens if we try to solve a CSP using a simple technique such as *breadth-first search*.

The branching factor is nd at the first step, for n variables each with d possible values.

$$\left.\begin{array}{ll}\text{Step 2:} & (n-1)d \\ \text{Step 3:} & (n-2)d \\ & \vdots \\ \text{Step } n\text{:} & d\end{array}\right\} \begin{aligned}\text{Number of leaves} &= nd \times (n-1)d \times \cdots \times 1 \\ &= n!d^n\end{aligned}$$

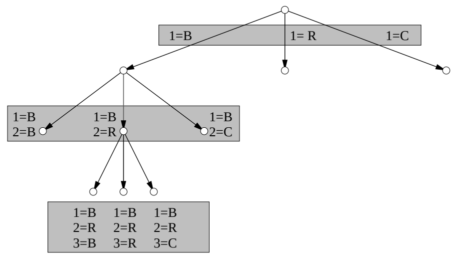
BUT: only d^n assignments are possible.

The order of assignment doesn't matter, and we should assign to one variable at a time.

Backtracking search

Using the graph colouring example:

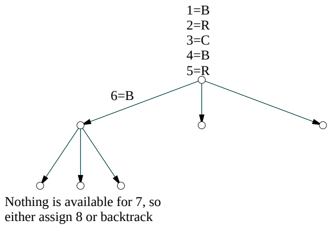
The search now looks something like this...

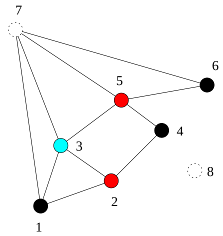


...and new possibilities appear.

Backtracking search

Backtracking search searches depth-first, assigning a single variable at a time, and backtracking if no valid assignment is available.





Rather than using problem-specific heuristics to try to improve searching, we can now explore heuristics applicable to *general* CSPs.

Backtracking search

```
Result backTrack(problem) {
  return bt ([], problem);
}

Result bt(assignmentList, problem) {
  if (assignmentList is complete)
    return assignmentList;
  nextVar = getNextVar(assignmentList, problem);
  for (all v in orderVariables(nextVar, assignmentList, problem)) {
    if (v is consistent with assignmentList) {
      add "nextVar = v" to assignmentList;
      solution = bt(assignmentList, problem);
      if (solution is not "fail")
        return solution;
      remove "nextVar = v" from assignmentList;
    }
  }
  return "fail";
}
```


Backtracking search: possible heuristics

There are several points we can examine in an attempt to obtain general CSP-based heuristics:

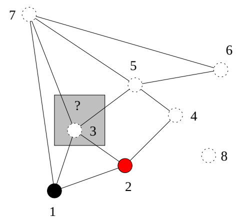
- In what order should we try to *assign variables*?
- In what order should we try to *assign possible values* to a variable?

Or being a little more subtle:

- What effect might the values assigned so far have on later attempted assignments?
- When forced to backtrack, is it possible to avoid the same failure later on?

Heuristics I: Choosing the order of variable assignments and values

Say we have $1 = B$ and $2 = R$



At this point there is *only one possible assignment* for 3, whereas the others have more flexibility.

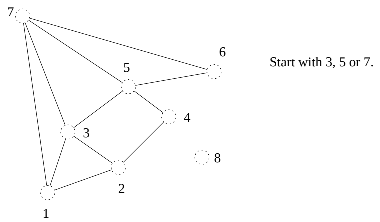
Assigning such variables *first* is called the *minimum remaining values (MRV)* heuristic.

(Alternatively, the *most constrained variable* or *fail first* heuristic.)

Heuristics I: Choosing the order of variable assignments and values

How do we choose a variable to begin with?

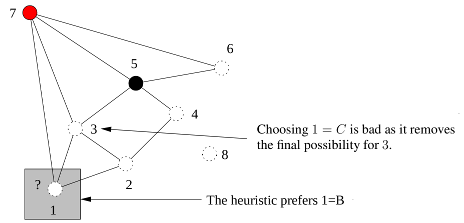
The *degree heuristic* chooses the variable involved in the most constraints on as yet unassigned variables.



MRV is usually better but the degree heuristic is a good tie breaker.

Heuristics I: Choosing the order of variable assignments and values

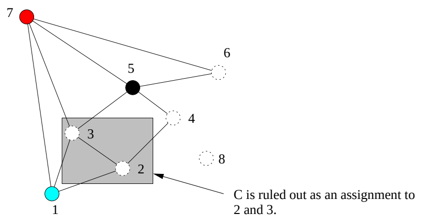
Once a variable is chosen, in *what order should values be assigned*?



The *least constraining value* heuristic chooses first the value that leaves the maximum possible freedom in choosing assignments for the variable's neighbours.

Heuristics II: forward checking and constraint propagation

Continuing the previous slide's progress, now add $1 = C$.



Each time we assign a value to a variable, it makes sense to delete that value from the collection of *possible assignments to its neighbours*.

This is called *forward checking*. It works nicely in conjunction with MRV.

Heuristics II: forward checking and constraint propagation

We can visualise this process as follows:

	1	2	3	4	5	6	7	8
Start	BRC	BRC	BRC	BRC	BRC	BRC	BRC	BRC
$2 = B$	RC	$= B$	RC	RC	BRC	BRC	BRC	BRC
$3 = R$	C	$= B$	$= R$	RC	BC	BRC	BC	BRC
$6 = B$	C	$= B$	$= R$	RC	C	$= B$	C	BRC
$5 = C$	C	$= B$	$= R$	R	$= C$	$= B$	!	BRC

At the fourth step 7 has *no possible assignments left*.

However, we could have detected a problem a little earlier...

Heuristics II: forward checking and constraint propagation

...by looking at step three.

	1	2	3	4	5	6	7	8
Start	BRC	BRC	BRC	BRC	BRC	BRC	BRC	BRC
$2 = B$	RC	$= B$	RC	RC	BRC	BRC	BRC	BRC
$3 = R$	C	$= B$	$= R$	RC	BC	BRC	BC	BRC
$6 = B$	C	$= B$	$= R$	RC	C	$= B$	C	BRC
$5 = C$	C	$= B$	$= R$	R	$= C$	$= B$	!	BRC

- At step three, 5 can be C only and 7 can be C only.
- But 5 and 7 are connected.
- So we can't progress, but this hasn't been detected.
- Ideally we want to do *constraint propagation*.

Trade-off: time to do the search, against time to explore constraints.

Constraint propagation

Arc consistency:

Consider a constraint as being *directed*. For example $4 \rightarrow 5$.

In general, say we have a constraint $i \rightarrow j$ and currently the domain of i is D_i and the domain of j is D_j.

$i \rightarrow j$ is *consistent* if

$$\forall d \in D_i, \exists d' \in D_j \text{ such that } i \rightarrow j \text{ is valid}$$

Constraint propagation

Example:

In step three of the table, $D_4 = \{R, C\}$ and $D_5 = \{C\}$.

- $5 \rightarrow 4$ in step three of the table *is consistent*.
- $4 \rightarrow 5$ in step three of the table *is not consistent*.

$4 \rightarrow 5$ can be made consistent by deleting C from D_4.

Or in other words, regardless of what you assign to i you'll be able to find something valid to assign to j.

Enforcing arc consistency

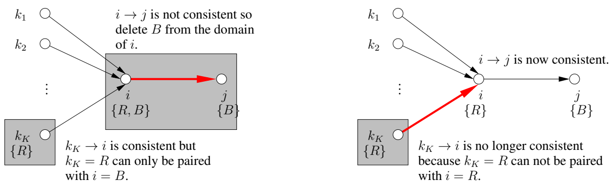
We can enforce arc consistency each time a variable i is assigned.

- We need to maintain a *collection of arcs to be checked*.
- Each time we alter a domain, we may have to include further arcs in the collection.

This is because if $i \rightarrow j$ is inconsistent resulting in a deletion from D_i we may as a consequence make some arc $k \rightarrow i$ inconsistent.

Why is this?

Enforcing arc consistency



- $i \to j$ inconsistent means removing a value from D_i.
- $\exists d \in D_i$ such that there is no valid $d' \in D_j$ *so delete* $d \in D_i$.

However some $d'' \in D_k$ may only have been pairable with d.

We need to continue until all consequences are taken care of.

The AC-3 algorithm

```
NewDomains AC-3 (problem) {
  Queue toCheck = all arcs i->j;
  while (toCheck is not empty) {
    i->j = next(toCheck);
    if (removeInconsistencies(Di,Dj)) {
      for (each k that is a neighbour of i)
        add k->i to toCheck;
    }
  }
}

Bool removeInconsistencies (domain1, domain2) {
  Bool result = false;
  for (each d in domain1) {
    if (no d' in domain2 valid with d) {
      remove d from domain1;
      result = true;
    }
  }
  return result;
}
```


Enforcing arc consistency

Complexity:

- A binary CSP with n variables can have $O(n^2)$ directional constraints $i \to j$.
- Any $i \to j$ can be considered at most d times where $d = \max_k |D_k|$ because only d things can be removed from D_i.
- Checking any single arc for consistency can be done in $O(d^2)$.

So the complexity is $O(n^2 d^3)$.

Note: this setup includes 3SAT.

Consequence: we can't check for consistency in polynomial time, which suggests this doesn't guarantee to find all inconsistencies.

A more powerful form of consistency

We can define a stronger notion of consistency as follows:

- *Given:* any $k - 1$ variables and any consistent assignment to these.
- *Then:* We can find a consistent assignment to any kth variable.

This is known as *k-consistency*.

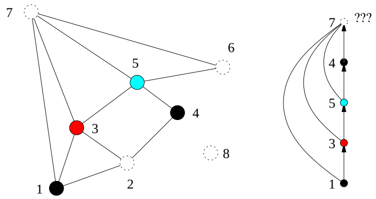
Strong k-consistency requires the we be k-consistent, $k - 1$-consistent *etc* as far down as 1-consistent.

If we can demonstrate strong n-consistency (where as usual n is the number of variables) then an assignment can be found in $O(nd)$.

Unfortunately, demonstrating strong n-consistency will be *worst-case exponential*.

Backjumping

The basic backtracking algorithm backtracks to the *most recent assignment*. This is known as *chronological backtracking*. It is not always the best policy:



Say we've assigned $1 = B$, $3 = R$, $5 = C$ and $4 = B$ and now we want to assign something to 7. This isn't possible so we backtrack, however re-assigning 4 clearly doesn't help.

Backjumping

With some careful bookkeeping it is often possible to *jump back multiple levels* without sacrificing the ability to find a solution.

We need some definitions:

- When we set a variable V_i to some value $d \in D_i$ we refer to this as the *assignment* $A_i = (V_i \leftarrow d)$.
- A *partial instantiation* $I_k = \{A_1, A_2, \ldots, A_k\}$ is a *consistent* set of assignments to the first k variables...
- ... where *consistent* means that no constraints are violated.

Henceforth we shall assume that variables are assigned in the order $V_1, V_2, \ldots, V_n$ when formally presenting algorithms.

Gaschnig's algorithm

Gaschnig's algorithm works as follows. Say we have a partial instantiation I_k:

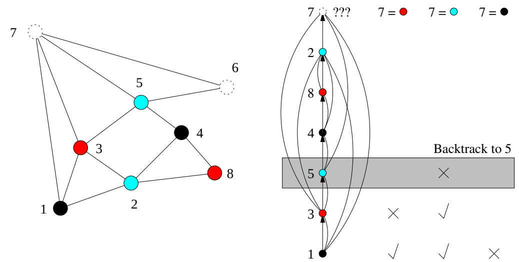
- When choosing a value for V_{k+1} we need to check that any candidate value $d \in D_{k+1}$, is consistent with I_k.
- When testing potential values for d, we will generally discard one or more possibilities, because they conflict with some member of I_k
- We keep track of the *most recent assignment* A_j for which this has happened.

Finally, if *no* value for V_{k+1} is consistent with I_k then we backtrack to V_j.

If there are no possible values left to try for V_j then we backtrack *chronologically*.

Gaschnig's algorithm

Example:



If there's no value left to try for 5 then backtrack to 3 and so on.

Graph-based backjumping

This allows us to jump back multiple levels *when we initially detect a conflict*.

Can we do better than chronological backtracking *thereafter*?

Some more definitions:

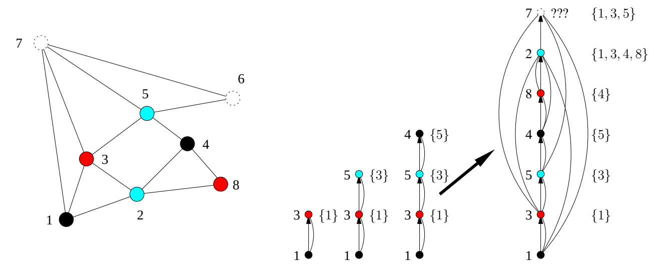
- We assume an ordering $V_1, V_2, \ldots, V_n$ for the variables.
- Given $V' = \{V_1, V_2, \ldots, V_k\}$ where $k < n$ the *ancestors* of V_{k+1} are the members of V' connected to V_{k+1} by a constraint.
- The *parent* $P(V)$ of V_{k+1} is its most recent ancestor.

The ancestors for each variable can be accumulated as assignments are made.

Graph-based backjumping backtracks to the *parent* of V_{k+1}.

Note: Gaschnig's algorithm uses *assignments* whereas graph-based backjumping uses *constraints*.

Graph-based backjumping



At this point, backjump to the *parent* for 7, which is 5.

Backjumping and forward checking

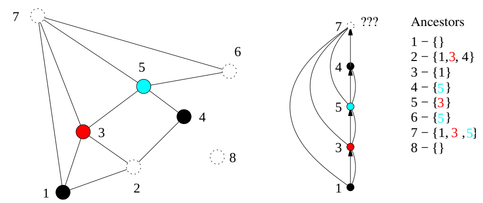
If we use *forward checking*: say we're assigning to V_{k+1} by making $V_{k+1} = d$:

- Forward checking removes d from the D_i of *all* V_i connected to V_{k+1} by a constraint.
- When doing graph-based backjumping, we'd also add V_{k+1} to the ancestors of V_i.

In fact, use of forward checking can make some forms of backjumping *redundant*.

Note: there are in fact many ways of combining *constraint propagation* with *backjumping*, and we will not explore them in further detail here.

Backjumping and forward checking

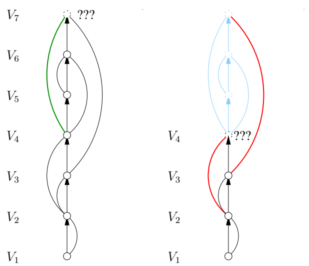


	1	2	3	4	5	6	7	8
Start	BRC	BRC	BRC	BRC	BRC	BRC	BRC	BRC
$1 = B$	$= B$	RC	RC	BRC	BRC	BRC	RC	BRC
$3 = R$	$= B$	C	$= R$	BRC	BC	BRC	C	BRC
$5 = C$	$= B$	C	$= R$	BR	$= C$	BR	!	BRC
$4 = B$	$= B$	C	$= R$	BR	$= C$	BR	!	BRC

Forward checking finds the problem *before backtracking does*.

Graph-based backjumping

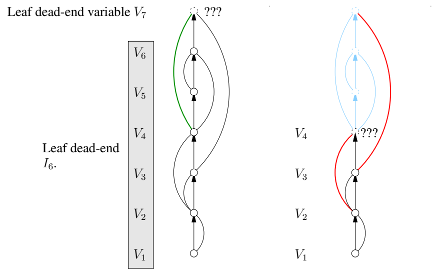
We're not quite done yet though. What happens when *there are no assignments left for the parent we just backjumped to*?



Backjumping from V_7 to V_4 is fine. However we shouldn't then just backjump to V_2, because changing V_3 could fix the problem at V_7.

Graph-based backjumping

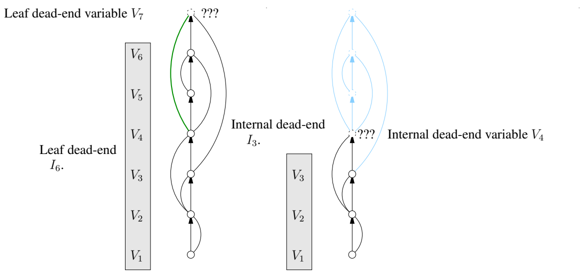
To describe an algorithm in this case is a little involved.



Given an instantiation I_k and V_{k+1}, if there is no consistent $d \in D_{k+1}$ we call I_k a *leaf dead-end* and V_{k+1} a *leaf dead-end variable*.

Graph-based backjumping

Also



If V_i was backtracked to from a later leaf dead-end and there are no more values to try for V_i then we refer to it as an *internal dead-end variable* and call I_{i-1} an *internal dead-end*.

Graph-based backjumping

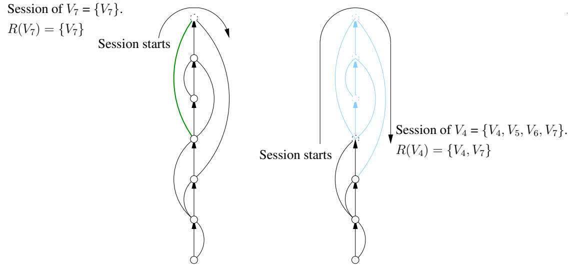
To keep track of exactly where to jump to we also need the definitions:

- The *session* of a variable V begins when the search algorithm visits it and ends when it backtracks through it to an earlier variable.
- The *current session* of a variable V is the set of all variables visiting during its session.
- In particular, the current session for any V contains V.
- The *relevant dead-ends for the current session* $R(V)$ for a variable V are:
  1. $R(V)$ is initialized to $\{V\}$ when V is first visited.
  2. If V is a leaf dead-end variable then $R(V) = \{V\}$.
  3. If V was backtracked to from a dead-end V' then $R(V) = R(V) \cup R(V')$.

And we're not done yet...

Graph-based backjumping

Example:



As expected, the relevant dead-ends for V_4 are $\{V_4\}$ and $\{V_7\}$.

Graph-based backjumping

One more bunch of definitions before the pain stops. Say V_k is a dead-end:

- The *induced ancestors* $\text{ind}(V_k)$ of V_k are defined as

$$\text{ind}(V_k) = \{V_1, V_2, \ldots, V_{k-1}\} \cap \left(\bigcup_{V \in R(V_k)} \text{ancestors}(V) \right)$$

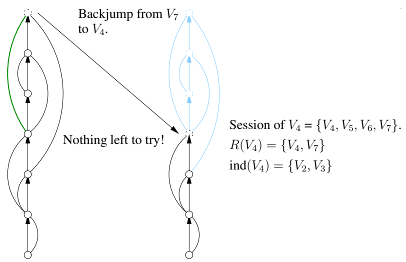
- The *culprit* for V_k is the most recent $V' \in \text{ind}(V_k)$.

Note that these definitions depend on $R(V_k)$.

FINALLY: graph-based backjumping *backjumps to the culprit*.

Graph-based backjumping

Example:



As expected, we back jump to V_3 instead of V_2. Hooray!

Conflict-directed backjumping

Gaschnig's algorithm and graph-based backjumping can be *combined* to produce *conflict-directed backjumping*.

We will not explore conflict-directed backjumping in this course.

For considerable further detail on algorithms for CSPs see:

"Constraint Processing," Rina Dechter. Morgan Kaufmann, 2003.

Varieties of CSP

We have only looked at *discrete* CSPs with *finite domains*. These are the simplest. We could also consider:

1. Discrete CSPs with *infinite domains*:
   - We need a *constraint language*. For example
   $$V_3 \leq V_{10} + 5$$
   - Algorithms are available for integer variables and linear constraints.
   - There is *no algorithm* for integer variables and nonlinear constraints.
2. Continuous domains—using linear constraints defining convex regions we have *linear programming*. This is solvable in polynomial time in n.
3. We can introduce *preference constraints* in addition to *absolute constraints*, and in some cases an *objective function*.

Artificial Intelligence I

Dr Sean Holden

Notes on *knowledge representation and reasoning using first-order logic (FOL)*

Knowledge representation and reasoning using FOL

We now look at how an agent might *represent* knowledge about its environment using first order logic (FOL), and *reason* with this knowledge to achieve its goals.

Aims:

- To show how FOL can be used to *represent knowledge* about an environment in the form of both *background knowledge* and *knowledge derived from percepts*.
- To show how this knowledge can be used to *derive non-perceived knowledge* about the environment using a *theorem prover*.
- To introduce the *situation calculus* and demonstrate its application in a simple environment as a means by which an agent can work out what to do next.

Interesting reading

Reading: Russell and Norvig, chapters 7 to 10.

Knowledge representation based on logic is a vast subject and can't be covered in full in the lectures.

In particular:

- Techniques for representing *further kinds of knowledge*.
- Techniques for moving beyond the idea of a *situation*.
- Reasoning systems based on *categories*.
- Reasoning systems using *default information*.
- *Truth maintenance systems*.

Happy reading :-)

Knowledge representation and reasoning

Earlier in the course we looked at what an *agent* should be able to do.

It seems that all of us—and all intelligent agents—should use *logical reasoning* to help us interact successfully with the world.

Any intelligent agent should:

- Possess *knowledge* about the *environment* and about *how its actions affect the environment*.
- Use some form of *logical reasoning* to *maintain* its knowledge as *percepts* arrive.
- Use some form of *logical reasoning* to *deduce actions* to perform in order to achieve *goals*.

Knowledge representation and reasoning

This raises some important questions:

- How do we describe the current state of the world?
- How do we infer from our percepts, knowledge of unseen parts of the world?
- How does the world change as time passes?
- How does the world stay the same as time passes? (The *frame problem*.)
- How do we know the effects of our actions? (The *qualification* and *ramification problems*.)

We'll now look at one way of answering some of these questions.

Logic for knowledge representation

FOL (arguably?) seems to provide a good way in which to represent the required kinds of knowledge:

- It is *expressive*—anything you can program can be expressed.
- It is *concise*.
- It is *unambiguous*
- It can be adapted to *different contexts*.
- It has an *inference procedure*, although a semidecidable one.

In addition is has a well-defined *syntax* and *semantics*.

Logic for knowledge representation

Problem: it's quite easy to talk about things like *set theory* using FOL. For example, we can easily write axioms like

$$\forall S \;.\; \forall S' \;.\; ((\forall x \;.\; (x \in S \Leftrightarrow x \in S')) \Rightarrow S = S')$$

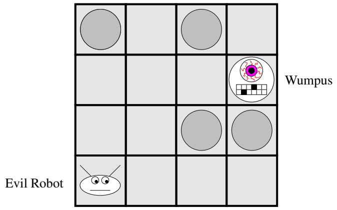
But how would we go about representing the proposition that *if you have a bucket of water and throw it at your friend they will get wet, have a bump on their head from being hit by a bucket, and the bucket will now be empty and dented*?

More importantly, how could this be represented within a wider framework for reasoning about the world?

It's time to introduce my friend, *The Wumpus*...

Wumpus world

As a simple test scenario for a knowledge-based agent we will make use of the *Wumpus World*.



The Wumpus World is a 4 by 4 grid-based cave.

EVIL ROBOT wants to enter the cave, find some gold, and get out again unscathed.

Wumpus world

The rules of *Wumpus World*:

- Unfortunately the cave contains a number of pits, which EVIL ROBOT can fall into. Eventually his batteries will fail, and that's the end of him.
- The cave also contains the Wumpus, who is armed with state of the art *Evil Robot Obliteration Technology*.
- The Wumpus itself knows where the pits are and never falls into one.

Wumpus world

EVIL ROBOT can move around the cave at will and can perceive the following:

- In a position adjacent to the Wumpus, a stench is perceived. (Wumpuses are famed for their *lack of personal hygiene*.)
- In a position adjacent to a pit, a *breeze* is perceived.
- In the position where the gold is, a glitter is perceived.
- On trying to move into a wall, a *bump* is perceived.
- On killing the Wumpus a *scream* is perceived.

In addition, EVIL ROBOT has a single arrow, with which to try to kill the Wumpus.

"Adjacent" in the following does *not* include diagonals.

Wumpus world

So we have:

Percepts: `stench`, `breeze`, `glitter`, `bump`, `scream`.

Actions: `forward`, `turnLeft`, `turnRight`, `grab`, `release`, `shoot`, `climb`.

Of course, our aim now is *not* just to design an agent that can perform well in a single cave layout.

We want to design an agent that can *usually* perform well *regardless* of the layout of the cave.

Some nomenclature

The choice of knowledge representation language tends to lead to two important commitments:

- *Ontological commitments*: what does the world consist of?
- *Epistemological commitments*: what are the allowable states of knowledge?

Propositional logic is useful for introducing some fundamental ideas, but its ontological commitment—that the world consists of facts—sometimes makes it too limited for further use.

FOL has a different ontological commitment—the world consists of *facts*, *objects* and *relations*.

Logic for knowledge representation

The fundamental aim is to construct a *knowledge base* KB containing a *collection of statements* about the world—expressed in FOL—such that *useful things can be derived* from it.

Our central aim is to generate sentences that are *true*, if *the sentences in the* KB *are true*.

This process is based on concepts familiar from your introductory logic courses:

- Entailment: $\texttt{KB} \models \alpha$ means that the KB entails α.
- Proof: $\texttt{KB} \vdash_i \alpha$ means that α is derived from the KB using i. If i is *sound* then we have a *proof*.
- i is *sound* if it can generate only entailed α.
- i is *complete* if it can find a proof for *any* entailed α.

Example: Prolog

You have by now learned a little about programming in *Prolog*. For example:

```
concat([],L,L).
concat([H|T],L,[H|L2]) :- concat(T,L,L2).
```

is a program to concatenate two lists. The query

```
concat([1,2,3],[4,5],X).
```

results in

```
X = [1, 2, 3, 4, 5].
```

What's happening here? Well, Prolog is just a *more limited form of FOL* so...

Example: Prolog

... we are in fact doing inference from a KB:

- The Prolog programme itself is the KB. It expresses some *knowledge about lists*.
- The query is expressed in such a way as to *derive some new knowledge*.

How does this relate to full FOL? First of all the list notation is nothing but *syntactic sugar*. It can be removed: we define a constant called `empty` and a function called `cons`.

Now `[1,2,3]` just means `cons(1, cons(2, cons(3, empty))))` which is a term in FOL.

I will assume the use of the syntactic sugar for lists from now on.

Prolog and FOL

The program when expressed in FOL, says

$$\forall x\,.\,\texttt{concat}(\texttt{empty}, x, x) \wedge$$
$$\forall h, t, l_1, l_2\,.\,\texttt{concat}(t, l_1, l_2) \Longrightarrow \texttt{concat}(\texttt{cons}(h, t), l_1, \texttt{cons}(h, l_2))$$

The rule is simple—given a Prolog program:

- *Universally quantify all the unbound variables in each line of the program* and ...
- *... form the conjunction of the results*.

If the universally quantified lines are $L_1, L_2, \ldots, L_n$ then the Prolog programme corresponds to the KB

$$\texttt{KB} = L_1 \wedge L_2 \wedge \cdots \wedge L_n$$

Now, what does the query mean?

Prolog and FOL

When you give the query

```
concat([1,2,3],[4,5],X).
```

to Prolog it responds by *trying to prove* the following statement

$$\texttt{KB} \Longrightarrow \exists x\,.\,\texttt{concat}([1,2,3],[4,5],x)$$

So: it tries to prove that the KB *implies the query*, and variables in the query are existentially quantified.

When a proof is found, it supplies a *value for x* that *makes the inference true*.

Prolog and FOL

Prolog differs from FOL in that, amongst other things:

- It restricts you to using *Horn clauses*.
- Its inference procedure is not a *full-blown proof procedure*.
- It does not deal with *negation* correctly.

However *the central idea also works for full-blown theorem provers*.

If you want to experiment, you can obtain *Prover9* from

http://www.cs.unm.edu/∼mccune/mace4/

We'll see a brief example now, and a more extensive example of its use later, time permitting...

Prolog and FOL

Expressed in Prover9, the above Prolog program and query look like this:

```
set(prolog_style_variables).

% This is the translated Prolog program for list concatenation.
% Prover9 has its own syntactic sugar for lists.

formulas(assumptions).
  concat([], L, L).
  concat(T, L, L2) -> concat([H:T], L, [H:L2]).
end_of_list.

% This is the query.

formulas(goals).
  exists X concat([1, 2, 3], [4, 5], X).
end_of_list.
```

Note: it is assumed that *unbound variables are universally quantified*.

Prolog and FOL

You can try to infer a proof using

```
prover9 -f file.in
```

and the result is (in addition to a lot of other information):

```
1 concat(T,L,L2) -> concat([H:T],L,[H:L2]) # label(non_clause).  [assumption].
2 (exists X concat([1,2,3],[4,5],X)) # label(non_clause) # label(goal).  [goal].
3 concat([],A,A).  [assumption].
4 -concat(A,B,C) | concat([D:A],B,[D:C]).  [clausify(1)].
5 -concat([1,2,3],[4,5],A).  [deny(2)].
6 concat([A],B,[A:B]).  [ur(4,a,3,a)].
7 -concat([2,3],[4,5],A).  [resolve(5,a,4,b)].
8 concat([A,B],C,[A,B:C]).  [ur(4,a,6,a)].
9 $F.  [resolve(8,a,7,a)].
```

This shows that a proof is found but doesn't explicitly give a value for `X`—we'll see how to extract that later...

The fundamental idea

So the *basic idea* is: build a `KB` that encodes *knowledge about the world*, the *effects of actions* and so on.

The `KB` is a conjunction of pieces of knowledge, such that:

- A query regarding what our agent should do *can be posed in the form*

  $\exists$`actionList.Goal(... actionList ...)`

- Proving that

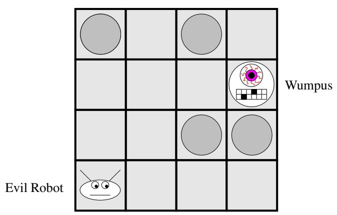
  `KB` $\Longrightarrow$ $\exists$`actionList.Goal(... actionList ...)`

  instantiates `actionList` to an *actual list of actions* that will achieve a goal represented by the `Goal` predicate.

We sometimes use the notation `ask` and `tell` to refer to *querying* and *adding to the* `KB`.

Using FOL in AI: the triumphant return of the Wumpus

We want to be able to *speculate* about the past and about *possible futures*. So:



- We include *situations* in the logical language used by our `KB`.
- We include *axioms* in our `KB` that relate to situations.

This gives rise to *situation calculus*.

Situation calculus

In *situation calculus*:

- The world consists of sequences of *situations*.
- Over time, an agent moves from one situation to another.
- Situations are changed as a result of *actions*.

In Wumpus World the actions are: `forward`, `shoot`, `grab`, `climb`, `release`, `turnRight`, `turnLeft`.

- A *situation argument* is added to items that can change over time. For example

  $$\text{At}(\text{location}, s)$$

  Items that can change over time are called *fluents*.

- A situation argument is not needed for things that don't change. These are sometimes referred to as *eternal* or *atemporal*.

Representing change as a result of actions

Situation calculus uses a function

$$\text{result}(\texttt{action}, s)$$

to denote the *new* situation arising as a result of performing the specified action in the specified situation.

$$\begin{aligned}
&\text{result}(\texttt{grab}, s_0) = s_1 \\
&\text{result}(\texttt{turnLeft}, s_1) = s_2 \\
&\text{result}(\texttt{shoot}, s_2) = s_3 \\
&\text{result}(\texttt{forward}, s_3) = s_4 \\
&\quad\vdots
\end{aligned}$$

Axioms I: possibility axioms

The first kind of axiom we need in a KB specifies *when particular actions are possible*.

We introduce a predicate

$$\text{Poss}(\texttt{action}, s)$$

denoting that an action can be performed in situation s.

We then need a *possibility axiom* for each action. For example:

$$\text{At}(l, s) \wedge \text{Available}(\texttt{gold}, l, s) \Longrightarrow \text{Poss}(\texttt{grab}, s)$$

Remember that *unbound variables are universally quantified*.

Axioms II: effect axioms

Given that an action results in a new situation, we can introduce *effect axioms* to specify the properties of the new situation.

For example, to keep track of whether EVIL ROBOT has the gold we need *effect axioms* to describe the effect of picking it up:

$$\text{Poss}(\texttt{grab}, s) \Longrightarrow \text{Have}(\texttt{gold}, \text{result}(\texttt{grab}, s))$$

Effect axioms describe the way in which the world *changes*.

We would probably also include

$$\neg\text{Have}(\texttt{gold}, s_0)$$

in the KB, where s_0 is the *starting state*.

Important: we are describing *what is true* in the *situation that results* from *performing an action* in a *given situation*.

Axioms III: frame axioms

We need *frame axioms* to describe *the way in which the world stays the same*.

Example:

$$\begin{aligned}
&\text{Have}(o, s) \wedge \\
&\quad \neg(a = \texttt{release} \wedge o = \texttt{gold}) \ \wedge\ \neg(a = \texttt{shoot} \wedge o = \texttt{arrow}) \\
&\quad \Longrightarrow \text{Have}(o, \text{result}(a, s))
\end{aligned}$$

describes the effect of *having something and not discarding it*.

In a more general setting such an axiom might well look different. For example

$$\begin{aligned}
&\neg\text{Have}(o, s) \wedge \\
&\quad (a \neq \texttt{grab}(o) \ \vee\ \neg(\text{Available}(o, s) \ \wedge\ \text{Portable}(o))) \\
&\quad \Longrightarrow \neg\texttt{Have}(o, \text{result}(a, s))
\end{aligned}$$

describes the effect of *not having something and not picking it up*.

The frame problem

The *frame problem* has historically been a major issue.

Representational frame problem: a large number of frame axioms are required to represent the many things in the world which will not change as the result of an action.

We will see how to solve this in a moment.

Inferential frame problem: when reasoning about a sequence of situations, all the unchanged properties still need to be carried through all the steps.

This can be alleviated using *planning systems* that allow us to reason efficiently when actions change only a small part of the world. There are also other remedies, which we will not cover.

Successor-state axioms

Effect axioms and frame axioms can be combined into *successor-state axioms*.

One is needed for each predicate that can change over time.

Action $\mathtt{a}$ is possible $\Longrightarrow$
(true in new situation $\iff$
(you did something to make it true $\vee$
it was already true and you didn't make it false))

For example

$$\begin{aligned}&\text{Poss}(a, s) \Longrightarrow\\&(\text{Have}(o, \text{result}(a, s)) \iff ((a = \texttt{grab} \wedge \text{Available(o,s)}) \vee\\&(\text{Have}(o, s) \wedge \neg(a = \texttt{release} \wedge o = \mathtt{gold}) \wedge\\&\neg(a = \texttt{shoot} \wedge o = \mathtt{arrow}))))\end{aligned}$$

Knowing where you are

If s_0 is the initial situation we know that

$$\text{At}((1, 1), s_0)$$

I am *assuming* that we've added axioms allowing us to deal with the numbers 0 to 5 and pairs of such numbers. *(Exercise: do this.)*

We need to keep track of what way we're facing. Say north is 0, south is 2, east is 1 and west is 3.

$$\text{facing}(s_0) = 0$$

We need to know how motion affects location

$$\text{forwardResult}((x, y), \mathtt{north}) = (x, y + 1)$$

$$\text{forwardResult}((x, y), \mathtt{east}) = (x + 1, y)$$

$$\vdots$$

and

$$\text{At}(l, s) \Longrightarrow \text{goForward}(s) = \text{forwardResult}(l, \text{facing}(s))$$

Knowing where you are

The concept of adjacency is very important in the Wumpus world

$$\text{Adjacent}(l_1, l_2) \iff \exists d \ \text{forwardResult}(l_1, d) = l_2$$

We also know that the cave is 4 by 4 and surrounded by walls

$$\text{WallHere}((x, y)) \iff (x = 0 \vee y = 0 \vee x = 5 \vee y = 5)$$

It is only possible to change location by moving, and this only works if you're not facing a wall. So...

...we need a successor-state axiom:

$$\begin{aligned}&\text{Poss}(a, s) \Longrightarrow\\&\text{At}(l, \text{result}(a, s)) \iff (l = \text{goForward}(s)\\&\wedge a = \texttt{forward}\\&\wedge \neg\text{WallHere}(l))\\&\vee (\text{At}(l, s) \wedge a \neq \texttt{forward})\end{aligned}$$

Knowing where you are

It is only possible to change orientation by turning. Again, we need a successor-state axiom

$$
\begin{aligned}
&\text{Poss}(a, s) \Longrightarrow \\
&\quad \text{facing}(\text{result}(a, s)) = d \iff \\
&\qquad (a = \texttt{turnRight} \wedge d = \text{mod}(\text{facing}(s) + 1, 4)) \\
&\qquad \vee (a = \texttt{turnLeft} \wedge d = \text{mod}(\text{facing}(s) - 1, 4)) \\
&\qquad \vee (\text{facing}(s) = d \wedge a \neq \texttt{turnRight} \wedge a \neq \texttt{turnLeft})
\end{aligned}
$$

and so on...

The qualification and ramification problems

Qualification problem: we are in general never completely certain what conditions are required for an action to be effective.

Consider for example turning the key to start your car.

This will lead to problems if important conditions are omitted from axioms.

Ramification problem: actions tend to have implicit consequences that are large in number.

For example, if I pick up a sandwich in a dodgy sandwich shop, I will also be picking up all the bugs that live in it. I don't want to model this explicitly.

Solving the ramification problem

The ramification problem can be solved by *modifying successor-state axioms*.

For example:

$$
\begin{aligned}
&\text{Poss}(a, s) \Longrightarrow \\
&\quad (\text{At}(o, l, \text{result}(a, s)) \iff \\
&\qquad (a = \text{go}(l', l) \wedge \\
&\qquad\quad [o = \texttt{robot} \vee \text{Has}(\texttt{robot}, o, s)]) \vee \\
&\qquad (\text{At}(o, l, s) \wedge \\
&\qquad\quad [\neg \exists l'' \,.\, a = \text{go}(l, l'') \wedge l \neq l'' \wedge \\
&\qquad\quad \{o = \text{robot} \vee \text{Has}(\text{robot}, o, s)\}]))
\end{aligned}
$$

describes the fact that anything EVIL ROBOT is carrying moves around with him.

Deducing properties of the world: causal rules

If you know where you are, then you can think about *places* rather than just *situations*.

Synchronic rules relate properties shared by a single state of the world.

There are two kinds: *causal* and *diagnostic*.

Causal rules: some properties of the world will produce percepts.

$$\text{WumpusAt}(l_1) \wedge \text{Adjacent}(l_1, l_2) \Longrightarrow \text{StenchAt}(l_2)$$

$$\text{PitAt}(l_1) \wedge \text{Adjacent}(l_1, l_2) \Longrightarrow \text{BreezeAt}(l_2)$$

Systems reasoning with such rules are known as *model-based* reasoning systems.

Deducing properties of the world: diagnostic rules

Diagnostic rules: infer properties of the world from percepts.

For example:

$$\text{At}(l, s) \wedge \text{Breeze}(s) \Longrightarrow \text{BreezeAt}(l)$$
$$\text{At}(l, s) \wedge \texttt{Stench}(s) \Longrightarrow \texttt{StenchAt}(l)$$

These may not be very strong.

The difference between model-based and diagnostic reasoning can be important. For example, medical diagnosis can be done based on symptoms or based on a model of disease.

General axioms for situations and objects

Note: in FOL, if we have two constants `robot` and `gold` then an interpretation is free to assign them to be the same thing.

This is not something we want to allow.

Unique names axioms state that each pair of distinct items in our model of the world must be different

$$\begin{aligned} \texttt{robot} &\neq \texttt{gold} \\ \texttt{robot} &\neq \texttt{arrow} \\ \texttt{robot} &\neq \texttt{wumpus} \\ &\vdots \\ \texttt{wumpus} &\neq \texttt{gold} \\ &\vdots \end{aligned}$$

General axioms for situations and objects

Unique actions axioms state that actions must share this property, so for each pair of actions

$$\begin{aligned} \texttt{go}(l, l') &\neq \texttt{grab} \\ \texttt{go}(l, l') &\neq \texttt{drop}(o) \\ &\vdots \\ \texttt{drop}(o) &\neq \texttt{shoot} \\ &\vdots \end{aligned}$$

and in addition we need to define equality for actions, so for each action

$$\begin{aligned} \texttt{go}(l, l') = \texttt{go}(l'', l''') &\iff l = l'' \wedge l' = l''' \\ \texttt{drop}(o) = \texttt{drop}(o') &\iff o = o' \\ &\vdots \end{aligned}$$

General axioms for situations and objects

The situations are *ordered* so

$$s_0 \neq \text{result}(a, s)$$

and situations are *distinct* so

$$\text{result}(a, s) = \text{result}(a', s') \iff a = a' \wedge s = s'$$

Strictly speaking we should be using a *many-sorted* version of FOL.

In such a system variables can be divided into *sorts* which are implicitly separate from one another.

The start state

Finally, we're going to need to specify *what's true in the start state*.

For example

$$\text{At}(\texttt{robot}, [1,1], s_0)$$
$$\text{At}(\texttt{wumpus}, [3,4], s_0)$$
$$\text{Has}(\texttt{robot}, \texttt{arrow}, s_0)$$
$$\vdots$$

and so on.

Sequences of situations

We know that the function result tells us about the situation resulting from performing an action in an earlier situation.

How can this help us find *sequences of actions to get things done*?

Define

$$\text{Sequence}([], s, s') = s' = s$$
$$\text{Sequence}([a], s, s') = \text{Poss}(a, s) \land s' = \text{result}(a, s)$$
$$\text{Sequence}(a :: as, s, s') = \exists t \;.\; \text{Sequence}([a], s, t) \land \text{Sequence}(as, t, s')$$

To obtain a *sequence of actions that achieves* Goal(s) we can use the query

$$\exists a\ \exists s \;.\; \text{Sequence}(a, s_0, s) \land \text{Goal}(s)$$

Knowledge representation and reasoning

It should be clear that generating sequences of actions by inference in FOL is highly non-trivial.

Ideally we'd like to maintain an *expressive* language while *restricting* it enough to be able to do inference *efficiently*.

Further aims:

- To give a brief introduction to *semantic networks* and *frames* for knowledge representation.
- To see how *inheritance* can be applied as a reasoning method.
- To look at the use of *rules* for knowledge representation, along with *forward chaining* and *backward chaining* for reasoning.

Further reading: *The Essence of Artificial Intelligence*, Alison Cawsey. Prentice Hall, 1998.

Frames and semantic networks

Frames and semantic networks represent knowledge in the form of *classes of objects* and *relationships between them*:

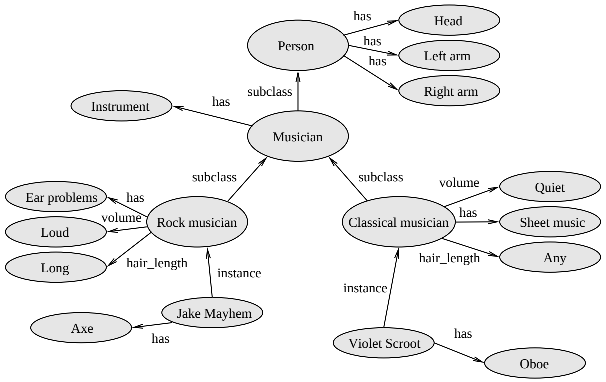
- The *subclass* and *instance* relationships are emphasised.
- We form *class hierarchies* in which *inheritance* is supported and provides the main *inference mechanism*.

As a result inference is quite limited.

We also need to be extremely careful about *semantics*.

The only major difference between the two ideas is *notational*.

Example of a semantic network



Frames

Frames once again support inheritance through the *subclass relationship*.

Rock musician

subclass: Musician
has: ear problems
hairlength: long
volume: loud

Musician

subclass: Person
has: instrument

`has`, `hairlength`, `volume` *etc* are *slots*.

`long`, `loud`, `instrument` *etc* are *slot values*.

These are a direct predecessor of *object-oriented programming languages*.

Defaults

Both approaches to knowledge representation are able to incorporate *defaults*:

Rock musician

subclass: Musician
has: ear problems
* hairlength: long
* volume: loud

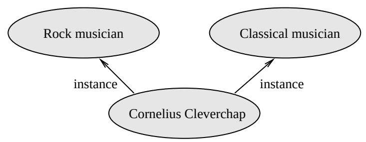
Dementia Evilperson

subclass: Rock musician
hairlength: short
image: gothic

Starred slots are *typical values* associated with subclasses and instances, but *can be overridden*.

Multiple inheritance

Both approaches can incorporate *multiple inheritance*, at a cost:



- What is `hairlength` for `Cornelius` if we're trying to use inheritance to establish it?
- This can be overcome initially by specifying which class is inherited from *in preference* when there's a conflict.
- But the problem is still not entirely solved—what if we want to prefer inheritance of some things from one class, but inheritance of others from a different one?

Other issues

- Slots and slot values can themselves be frames. For example `Dementia` may have an instrument slot with the value `Electric harp`, which itself may have properties described in a frame.
- Slots can have *specified attributes*. For example, we might specify that `instrument` can have multiple values, that each value can only be an instance of `Instrument`, that each value has a slot called `owned_by` and so on.
- Slots may contain arbitrary pieces of program. This is known as *procedural attachment*. The fragment might be executed to return the slot's value, or update the values in other slots *etc*.

Rule-based systems

A rule-based system requires three things:

1. A set of `if-then` *rules*. These denote specific pieces of knowledge about the world.

   They should be interpreted similarly to logical implication.

   Such rules denote *what to do* or *what can be inferred* under given circumstances.
2. A collection of *facts* denoting what the system regards as currently true about the world.
3. An interpreter able to apply the current rules in the light of the current facts.

Forward chaining

The first of two basic kinds of interpreter *begins with established facts and then applies rules to them.*

This is a *data-driven* process. It is appropriate if we know the *initial facts* but not the required conclusion.

Example: XCON—used for configuring VAX computers.

In addition:

- We maintain a *working memory*, typically of what has been inferred so far.
- Rules are often *condition-action rules*, where the right-hand side specifies an action such as adding or removing something from working memory, printing a message *etc*.
- In some cases actions might be entire program fragments.

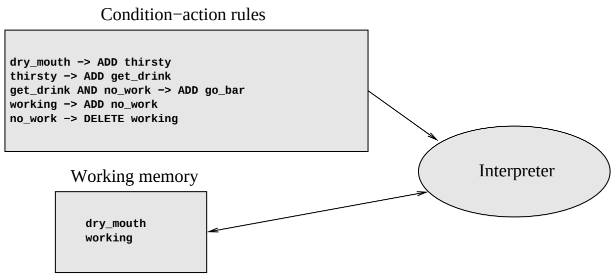
Forward chaining

The basic algorithm is:

1. Find all the rules that can fire, based on the current working memory.
2. Select a rule to fire. This requires a *conflict resolution strategy*.
3. Carry out the action specified, possibly updating the working memory.

Repeat this process until either *no rules can be used* or a *halt* appears in the working memory.

Example



Example

Progress is as follows:

1. The rule

   dry_mouth ⟹ ADD thirsty

   fires adding thirsty to working memory.
2. The rule

   thirsty ⟹ ADD get_drink

   fires adding get_drink to working memory.
3. The rule

   working ⟹ ADD no_work

   fires adding no_work to working memory.
4. The rule

   get_drink AND no_work ⟹ ADD go_bar

   fires, and we establish that it's time to go to the bar.

Conflict resolution

Clearly in any more realistic system we expect to have to deal with a scenario where *two or more rules can be fired at any one time*:

- Which rule we choose can clearly affect the outcome.
- We might also want to attempt to avoid inferring an abundance of useless information.

We therefore need a means of *resolving such conflicts*.

Conflict resolution

Common *conflict resolution strategies* are:

- Prefer rules involving more recently added facts.
- Prefer rules that are *more specific*. For example

  patient_coughing ⟹ ADD lung_problem

  is more general than

  patient_coughing AND patient_smoker ⟹ ADD lung_cancer.

  This allows us to define exceptions to general rules.
- Allow the designer of the rules to specify priorities.
- Fire all rules *simultaneously*—this essentially involves following all chains of inference at once.

Reason maintenance

Some systems will allow information to be removed from the working memory if it is no longer *justified*.

For example, we might find that

`patient_coughing`

and

`patient_smoker`

are in working memory, and hence fire

`patient_coughing` AND `patient_smoker` $\Longrightarrow$ ADD `lung_cancer`

but later infer something that causes `patient_coughing` to be *withdrawn* from working memory.

The justification for `lung_cancer` has been removed, and so it should perhaps be removed also.

Pattern matching

In general rules may be expressed in a slightly more flexible form involving *variables* which can work in conjunction with *pattern matching*.

For example the rule

`coughs`(X) AND `smoker`(X) $\Longrightarrow$ ADD `lung_cancer`(X)

contains the variable X.

If the working memory contains `coughs(neddy)` and `smoker(neddy)` then

$X =$ `neddy`

provides a match and

`lung_cancer(neddy)`

is added to the working memory.

Backward chaining

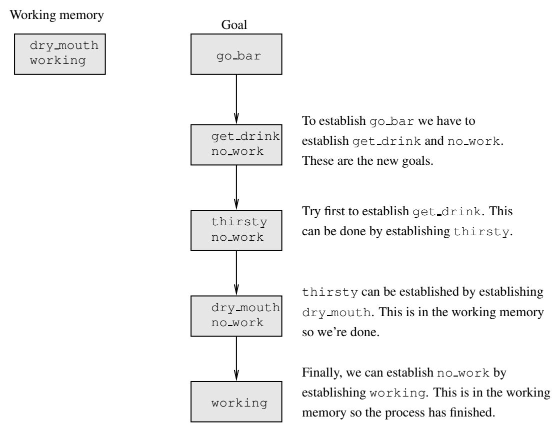
The second basic kind of interpreter begins with a *goal* and finds a rule that would achieve it.

It then works *backwards*, trying to achieve the resulting earlier goals in the succession of inferences.

Example: MYCIN—medical diagnosis with a small number of conditions.

This is a *goal-driven* process. If you want to *test a hypothesis* or you have some idea of a likely conclusion it can be more efficient than forward chaining.

Example



Example with backtracking

If at some point more than one rule has the required conclusion then we can *backtrack*.

Example: *Prolog* backtracks, and incorporates pattern matching. It orders attempts according to the order in which rules appear in the program.

Example: having added

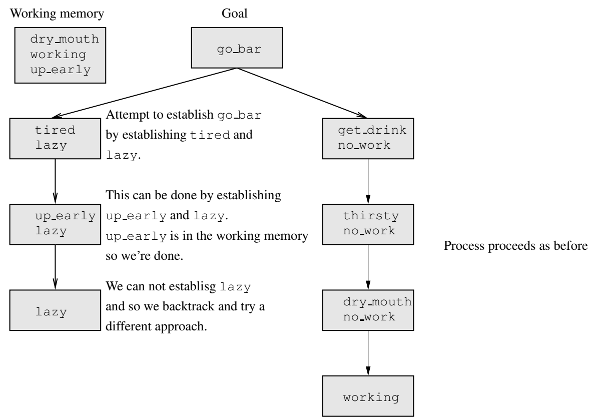
$$\texttt{up\_early} \Longrightarrow \text{ADD } \texttt{tired}$$

and

$$\texttt{tired} \text{ AND } \texttt{lazy} \Longrightarrow \text{ADD } \texttt{go\_bar}$$

to the rules, and `up_early` to the working memory:

Example with backtracking



Artificial Intelligence I

Dr Sean Holden

Notes on *planning*

Problem solving is different to planning

In *search problems* we:

- *Represent states*: and a state representation contains *everything* that's relevant about the environment.
- *Represent actions*: by describing a new state obtained from a current state.
- *Represent goals*: all we know is how to test a state either to see if it's a goal, or using a heuristic.
- *A sequence of actions is a 'plan'*: but we only consider *sequences of consecutive actions*.

Search algorithms are good for solving problems that fit this framework. However for more complex problems they may fail completely...

Problem solving is different to planning

Representing a problem such as: *'go out and buy some pies'* is hopeless:

- There are *too many possible actions* at each step.
- A heuristic can only help you rank states. In particular it does not help you *ignore* useless actions.
- We are forced to start at the initial state, but you have to work out *how to get the pies*—that is, go to town and buy them, get online and find a web site that sells pies *etc—before you can start to do it*.

Knowledge representation and reasoning might not help either: although we end up with a sequence of actions—a plan—there is so much flexibility that complexity might well become an issue.

Introduction to planning

We now look at how an agent might *construct a plan* enabling it to achieve a goal.

Aims:

- To look at how we might update our concept of *knowledge representation and reasoning* to apply more specifically to planning tasks.
- To look in detail at the basic *partial-order planning algorithm*.

Reading: Russell and Norvig, chapter 11.

Planning algorithms work differently

Difference 1:

- Planning algorithms use a *special purpose language*—often based on FOL or a subset— to represent states, goals, and actions.
- States and goals are described by sentences, as might be expected, but...
- ...actions are described by stating their *preconditions* and their *effects*.

So if you know the goal includes (maybe among other things)

$$\text{Have}(\texttt{pie})$$

and action $\text{Buy}(x)$ has an effect $\text{Have}(x)$ then you know that a plan *including*

$$\text{Buy}(\texttt{pie})$$

might be reasonable.

Planning algorithms work differently

Difference 2:

- Planners can add actions at *any relevant point at all between the start and the goal*, not just at the end of a sequence starting at the start state.
- This makes sense: I may determine that $\text{Have}(\texttt{carKeys})$ is a good state to be in without worrying about what happens before or after finding them.
- By making an important decision like requiring $\text{Have}(\texttt{carKeys})$ early on we may reduce branching and backtracking.
- State descriptions are not complete—$\text{Have}(\texttt{carKeys})$ describes a *class of states*—and this adds flexibility.

So: you have the potential to search both *forwards* and *backwards* within the same problem.

Planning algorithms work differently

Difference 3:

It is assumed that most elements of the environment are *independent of most other elements*.

- A goal including several requirements can be attacked with a divide-and-conquer approach.
- Each individual requirement can be fulfilled using a subplan...
- ...and the subplans then combined.

This works provided there is not significant interaction between the subplans.

Remember: the *frame problem*.

Running example: gorilla-based mischief

We will use the following simple example problem, which as based on a similar one due to Russell and Norvig.

The intrepid little scamps in the *Cambridge University Roof-Climbing Society* wish to attach an *inflatable gorilla* to the spire of a *Famous College*. To do this they need to leave home and obtain:

- *An inflatable gorilla*: these can be purchased from all good joke shops.
- *Some rope*: available from a hardware store.
- *A first-aid kit*: also available from a hardware store.

They need to return home after they've finished their shopping.

How do they go about planning their *jolly escapade*?

The STRIPS language

STRIPS: *"Stanford Research Institute Problem Solver"* (1970).

States: are *conjunctions* of *ground literals*. They must not include *function symbols*.

$$\begin{aligned}\text{At}(\texttt{home}) &\wedge \neg\text{Have}(\texttt{gorilla})\\ &\wedge \neg\text{Have}(\texttt{rope})\\ &\wedge \neg\text{Have}(\texttt{kit})\end{aligned}$$

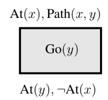
Goals: are *conjunctions* of *literals* where variables are assumed *existentially quantified*.

$$\text{At}(x) \wedge \text{Sells}(x, \texttt{gorilla})$$

A planner finds a sequence of actions that when performed makes the goal true. We are no longer employing a full theorem-prover.

The STRIPS language

STRIPS represents actions using *operators*. For example



$$\text{Op}(\text{Action: Go}(y), \text{Pre: At}(x) \wedge \text{Path}(x, y), \text{Effect: At}(y) \wedge \neg\text{At}(x))$$

All variables are implicitly universally quantified. An operator has:

- An *action description*: what the action does.
- A *precondition*: what must be true before the operator can be used. A *conjunction of positive literals*.
- An *effect*: what is true after the operator has been used. A *conjunction of literals*.

The space of plans

We now make a change in perspective—we search in *plan space*:

- Start with an *empty plan*.
- *Operate on it* to obtain new plans. Incomplete plans are called *partial plans*. *Refinement operators* add constraints to a partial plan. All other operators are called *modification operators*.
- Continue until we obtain a plan that solves the problem.

Operations on plans can be:

- *Adding a step*.
- *Instantiating a variable*.
- *Imposing an ordering* that places a step in front of another.
- and so on...

Representing a plan: partial order planners

When putting on your shoes and socks:

- It *does not matter* whether you deal with your left or right foot first.
- It *does matter* that you place a sock on *before* a shoe, for any given foot.

It makes sense in constructing a plan *not* to make any *commitment* to which side is done first *if you don't have to*.

Principle of least commitment: do not commit to any specific choices until you have to. This can be applied both to ordering and to instantiation of variables. A *partial order planner* allows plans to specify that some steps must come before others but others have no ordering. A *linearisation* of such a plan imposes a specific sequence on the actions therein.

Representing a plan: partial order planners

A plan consists of:

1. A set $\{S_1, S_2, \ldots, S_n\}$ of *steps*. Each of these is one of the available *operators*.
2. A set of *ordering constraints*. An ordering constraint $S_i < S_j$ denotes the fact that step S_i must happen before step S_j. $S_i < S_j < S_k$ and so on has the obvious meaning. $S_i < S_j$ does *not* mean that S_i must *immediately* precede S_j.
3. A set of variable bindings $v = x$ where v is a variable and x is either a variable or a constant.
4. A set of *causal links* or *protection intervals* $S_i \xrightarrow{c} S_j$. This denotes the fact that the purpose of S_i is to achieve the precondition c for S_j.

A causal link is *always* paired with an equivalent ordering constraint.

Representing a plan: partial order planners

The *initial plan* has:

- Two steps, called Start and Finish.
- a single ordering constraint Start $<$ Finish.
- No *variable bindings*.
- No *causal links*.

In addition to this:

- The step Start has no preconditions, and its effect is the start state for the problem.
- The step Finish has no effect, and its precondition is the goal.
- Neither Start or Finish has an associated action.

We now need to consider what constitutes a *solution*...

Solutions to planning problems

A solution to a planning problem is any *complete* and *consistent* partially ordered plan.

Complete: each precondition of each step is *achieved* by another step in the solution.

A precondition c for S is achieved by a step S' if:

1. The precondition is an effect of the step

   $$S' < S \text{ and } c \in \text{Effects}(S')$$

   and...

2. ... there is *no other* step that *could* cancel the precondition. That is, no S'' exists where:
   - The existing ordering constraints allow S'' to occur *after* S' but *before* S.
   - $\neg c \in \text{Effects}(S'')$.

Solutions to planning problems

Consistent: no contradictions exist in the binding constraints or in the proposed ordering. That is:

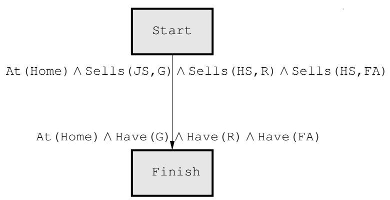
1. For binding constraints, we never have $v = X$ and $v = Y$ for distinct constants X and Y.
2. For the ordering, we never have $S < S'$ and $S' < S$.

Returning to the roof-climber's shopping expedition, here is the basic approach:

- Begin with only the `Start` and `Finish` steps in the plan.
- At each stage add a new step.
- Always add a new step such that a *currently non-achieved precondition is achieved*.
- Backtrack when necessary.

An example of partial-order planning

Here is the *initial plan*:



Thin arrows denote ordering.

An example of partial-order planning

There are *two actions available*:

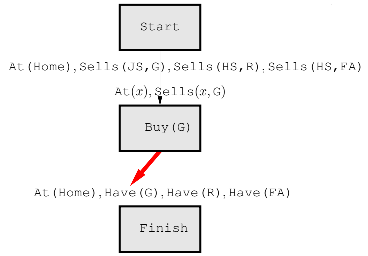
A planner might begin, for example, by adding a Buy(G) action in order to achieve the Have(G) precondition of Finish.

Note: the following order of events is by no means the only one available to a planner.

It has been chosen for illustrative purposes.

An example of partial-order planning

Incorporating the suggested step into the plan:

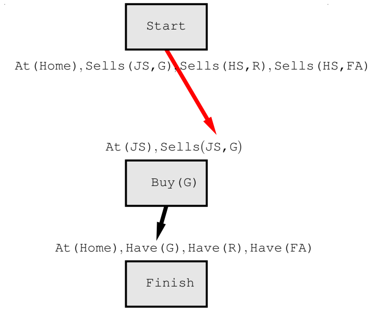


Thick arrows denote causal links. They always have a thin arrow underneath.

Here the new `Buy` step achieves the Have(G) precondition of Finish.

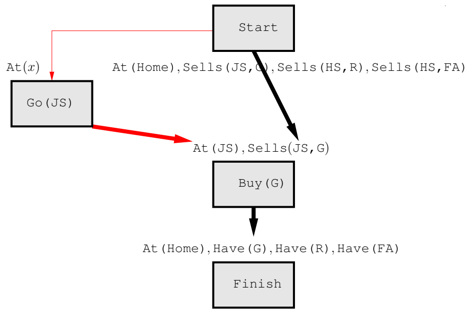
An example of partial-order planning

The planner can now introduce a second causal link from Start to achieve the Sells(x, G) precondition of Buy(G).



An example of partial-order planning

The planner's next obvious move is to introduce a Go step to achieve the At(JS) precondition of Buy(G).



And we continue...

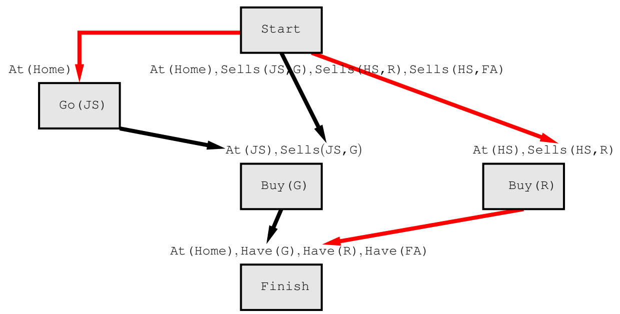
An example of partial-order planning

Initially the planner can continue quite easily in this manner:

- Add a causal link from Start to Go(JS) to achieve the At(x) precondition.
- Add the step Buy(R) with an associated causal link to the Have(R) precondition of Finish.
- Add a causal link from Start to Buy(R) to achieve the Sells(HS, R) precondition.

But then things get more interesting...

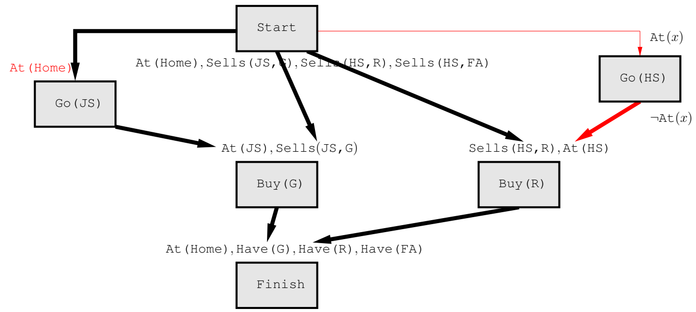
An example of partial-order planning



At this point it starts to get tricky...

The At(HS) precondition in Buy(R) is not achieved.

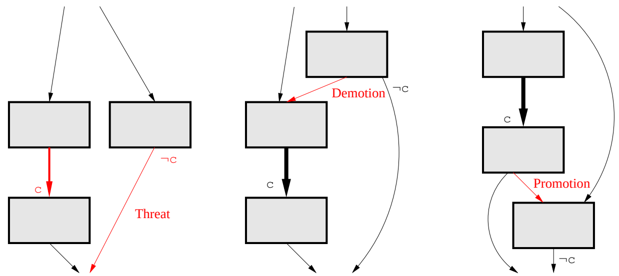
An example of partial-order planning



The At(HS) precondition is easy to achieve. *But if we introduce a causal link from* Start *to* Go(HS) *then we risk invalidating the precondition for* Go(JS).

An example of partial-order planning

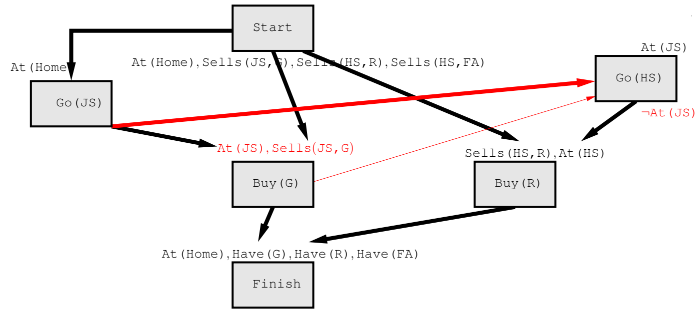
A step that might invalidate (sometimes the word *clobber* is employed) a previously achieved precondition is called a *threat*.



A planner can try to fix a threat by introducing an ordering constraint.

An example of partial-order planning

The planner could backtrack and try to achieve the At(x) precondition using the existing Go(JS) step.



This involves a threat, but one that can be fixed using promotion.

The algorithm

Simplifying slightly to the case where there are *no variables*.

Say we have a partially completed plan and a set of the preconditions that have yet to be achieved.

- Select a precondition p that has not yet been achieved and is associated with an action B.
- At each stage *the partially complete plan is expanded into a new collection of plans*.
- To expand a plan, we can try to achieve p *either* by using an action that's already in the plan or by adding a new action to the plan. In either case, call the action A.

We then try to construct consistent plans where A achieves p.

The algorithm

This works as follows:

- For *each possible way of achieving* p:
  - Add Start $< A$, $A <$ Finish, $A < B$ and the causal link $A \xrightarrow{p} B$ to the plan.
  - If the resulting plan is consistent we're done, otherwise *generate all possible ways of removing inconsistencies* by promotion or demotion and *keep any resulting consistent plans*.

At this stage:

- If you have *no further preconditions that haven't been achieved* then *any plan obtained is valid*.

The algorithm

But how do we try to *enforce consistency*?

When you attempt to achieve p using A:

- Find all the existing causal links $A' \xrightarrow{\neg p} B'$ that are *clobbered* by A.
- For each of those you can try adding $A < A'$ or $B' < A$ to the plan.
- Find all existing actions C in the plan that clobber the *new* causal link $A \xrightarrow{p} B$.
- For each of those you can try adding $C < A$ or $B < C$ to the plan.
- Generate *every possible combination* in this way and retain any consistent plans that result.

Possible threats

What about dealing with *variables*?

If at any stage an effect $\neg$At(x) appears, is it a threat to At$(\texttt{JS})$?

Such an occurrence is called a *possible threat* and we can deal with it by introducing *inequality constraints*: in this case $x \neq \texttt{JS}$.

- Each partially complete plan now has a set I of inequality constraints associated with it.
- An inequality constraint has the form $v \neq X$ where v is a variable and X is a variable or a constant.
- Whenever we try to make a substitution we check I to make sure we won't introduce a conflict.

If we *would* introduce a conflict then we discard the partially completed plan as inconsistent.

Artificial Intelligence I

Dr Sean Holden

Notes on *machine learning using neural networks*

Did you heed the DIRE WARNING?

At the beginning of the course I suggested making sure you can answer the following two questions:

1. Let
$$f(x_1, \dots, x_n) = \sum_{i=1}^{n} a_i x_i^2$$
where the a_i are constants. Compute $\partial f / \partial x_j$ where $1 \leq j \leq n$?
Answer: As
$$f(x_1, \dots, x_n) = a_1 x_1^2 + \cdots + a_j x_j^2 + \cdots + a_n x_n^2$$
only one term in the sum depends on x_j, so all the other terms differentiate to give 0 and
$$\frac{\partial f}{\partial x_j} = 2 a_j x_j$$

Did you heed the DIRE WARNING?

2. Let $f(x_1, \dots, x_n)$ be a function. Now assume $x_i = g_i(y_1, \dots, y_m)$ for each x_i and some collection of functions g_i. Assuming all requirements for differentiability and so on are met, can you write down an expression for $\partial f / \partial y_j$ where $1 \leq j \leq m$?
Answer: this is just the *chain rule* for partial differentiation
$$\frac{\partial f}{\partial y_j} = \sum_{i=1}^{n} \frac{\partial f}{\partial g_i} \frac{\partial g_i}{\partial y_j}$$

Supervised learning with neural networks

We now look at how an agent might *learn* to solve a general problem by seeing *examples*.

Aims:

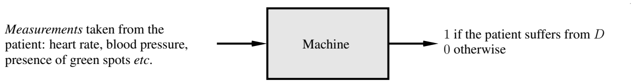
- To present an outline of *supervised learning* as part of AI.
- To introduce much of the notation and terminology used.
- To introduce the classical *perceptron*.
- To introduce *multilayer perceptrons* and the *backpropagation algorithm* for training them.

Reading: Russell and Norvig chapter 20.

An example

A common source of problems in AI is *medical diagnosis*.

Imagine that we want to automate the diagnosis of an Embarrassing Disease (call it D) by constructing a machine:



Could we do this by *explicitly writing a program* that examines the measurements and outputs a diagnosis?

Experience suggests that this is unlikely.

An example, continued...

An alternative approach: each collection of measurements can be written as a vector,

$$\mathbf{x}^T = (\ x_1 \ x_2 \ \cdots \ x_n\ )$$

where,

$$\begin{aligned} x_1 &= \text{heart rate} \\ x_2 &= \text{blood pressure} \\ x_3 &= 1 \text{ if the patient has green spots} \\ &\quad\ 0 \text{ otherwise} \\ &\vdots \\ &\quad \text{and so on} \end{aligned}$$

(*Note*: it's a common convention that vectors are *column vectors* by default. This is why the above is written as a *transpose*.)

An example, continued...

A vector of this kind contains all the measurements for a single patient and is called a *feature vector* or *instance*.

The measurements are *attributes* or *features*.

Attributes or features generally appear as one of three basic types:

- *Continuous*: $x_i \in [x_{\text{min}}, x_{\text{max}}]$ where $x_{\text{min}}, x_{\text{max}} \in \mathbb{R}$.
- *Binary*: $x_i \in \{0, 1\}$ or $x_i \in \{-1, +1\}$.
- *Discrete*: x_i can take one of a finite number of values, say $x_i \in \{X_1, \ldots, X_p\}$.

An example, continued...

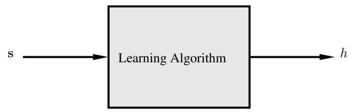
Now imagine that we have a large collection of patient histories (m in total) and for each of these we know whether or not the patient suffered from D.

- The ith patient history gives us an instance $\mathbf{x}_i$.
- This can be paired with a single bit—0 or 1—denoting whether or not the ith patient suffers from D. The resulting pair is called an *example* or a *labelled example*.
- Collecting all the examples together we obtain a *training sequence*

$$\mathbf{s} = ((\mathbf{x}_1, 0), (\mathbf{x}_2, 1), \ldots, (\mathbf{x}_m, 0))$$

An example, continued...

In supervised machine learning we aim to design a *learning algorithm* which takes $\mathbf{s}$ and produces a *hypothesis* h.



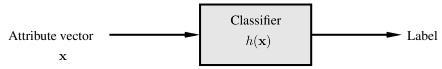
Intuitively, a hypothesis is something that lets us diagnose *new* patients.

This is *IMPORTANT*: we want to diagnose patients that *the system has never seen*.

The ability to do this successfully is called *generalisation*.

An example, continued...

In fact, a hypothesis is just a *function* that maps *instances* to *labels*.



As h is a *function* it assigns a label to *any* $\mathbf{x}$ and *not just the ones that were in the training sequence*.

What we mean by a *label* here depends on whether we're doing *classification* or *regression*.

Supervised learning: classification

In *classification* we're assigning $\mathbf{x}$ to one of a set $\{\omega_1, \dots, \omega_c\}$ of c *classes*.

For example, if $\mathbf{x}$ contains measurements taken from a patient then there might be three classes:

$$\begin{aligned}\omega_1 &= \text{patient has disease}\\ \omega_2 &= \text{patient doesn't have disease}\\ \omega_3 &= \text{don't ask me buddy, I'm just a computer!}\end{aligned}$$

The *binary* case above also fits into this framework, and we'll often specialise to the case of two classes, denoted C_1 and C_2.

Supervised learning: regression

In *regression* we're assigning $\mathbf{x}$ to a *real number* $h(\mathbf{x}) \in \mathbb{R}$.

For example, if $\mathbf{x}$ contains measurements taken regarding today's weather then we might have

$$h(\mathbf{x}) = \text{estimate of amount of rainfall expected tomorrow}$$

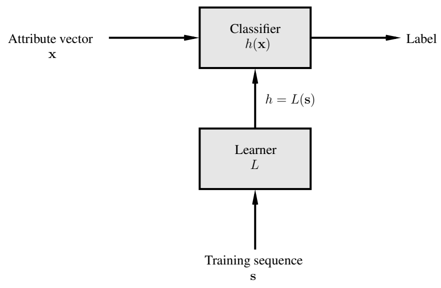
For the two-class classification problem we will also refer to a situation somewhat between the two, where

$$h(\mathbf{x}) = \Pr(\mathbf{x} \text{ is in } C_1)$$

and so we would typically assign $\mathbf{x}$ to class C_1 if $h(\mathbf{x}) > 1/2$.

Summary

We don't want to design h explicitly.



So we use a *learner* L to infer it on the basis of a sequence s of *training examples*.

Neural networks

There is generally a set $\mathcal{H}$ of hypotheses from which L is allowed to select h

$$L(\mathbf{s}) = h \in \mathcal{H}$$

$\mathcal{H}$ is called the *hypothesis space*.

The learner can output a hypothesis explicitly or—as in the case of a *neural network*—it can output a vector

$$\mathbf{w}^T = \left(w_1 \ w_2 \ \cdots \ w_W \right)$$

of *weights* which in turn specify h

$$h(\mathbf{x}) = f(\mathbf{w}; \mathbf{x})$$

where $\mathbf{w} = L(\mathbf{s})$.

Types of learning

The form of machine learning described is called *supervised learning*.

This introduction will concentrate on this kind of learning. In particular, the literature also discusses:

1. *Unsupervised learning*.
2. Learning using *membership queries* and *equivalence queries*.
3. *Reinforcement learning*.

Some of this further material will be covered in AI 2.

Some further examples

- *Speech recognition*.
- Deciding *whether or not to give credit*.
- Detecting *credit card fraud*.
- Deciding whether to *buy or sell a stock option*.
- Deciding whether a *tumour is benign*.
- *Data mining*: extracting interesting but hidden knowledge from existing, large databases. For example, databases containing *financial transactions* or *loan applications*.
- Deciding whether *driving conditions are dangerous*.
- *Automatic driving*. (See Pomerleau, 1989, in which a car is driven for 90 miles at 70 miles per hour, on a public road with other cars present, but with no assistance from humans.)

This is very similar to curve fitting

This process is in fact very similar to *curve fitting*.

Think of the process as follows:

- Nature picks an $h' \in \mathcal{H}$ but doesn't reveal it to us.
- Nature then shows us a training sequence $\mathbf{s}$ where each $\mathbf{x}_i$ is labelled as $h'(\mathbf{x}_i) + \epsilon_i$ where ϵ_i is noise of some kind.

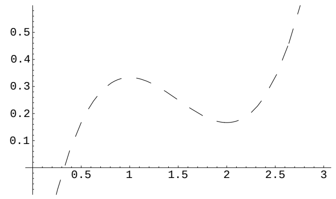
Our job is to try to infer what h' is *on the basis of* $\mathbf{s}$ *only*.

This is easy to visualise in one dimension: *it's just fitting a curve to some points*.

Curve fitting

Example: if $\mathcal{H}$ is the set of all polynomials of degree 3 then nature might pick

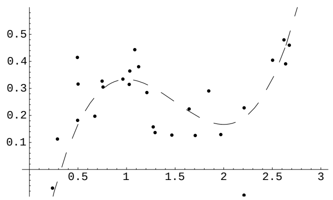
$$h'(x) = \frac{1}{3}x^3 - \frac{3}{2}x^2 + 2x - \frac{1}{2}$$



The line is dashed to emphasise the fact that *we don't get to see it*.

Curve fitting

We can now use h' to obtain a training sequence $\mathbf{s}$ in the manner suggested..



Here we have,

$$\mathbf{s}^T = ((x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m))$$

where each x_i and y_i is a real number.

Curve fitting

We'll use a *learning algorithm* L that operates in a reasonable-looking way: it picks an $h \in \mathcal{H}$ minimising the following quantity,

$$E = \sum_{i=1}^{m} (h(x_i) - y_i)^2$$

In other words

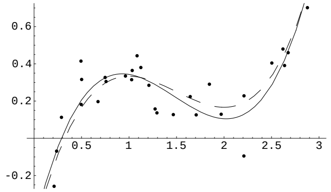
$$h = L(\mathbf{s}) = \operatorname*{argmin}_{h \in \mathcal{H}} \sum_{i=1}^{m} (h(x_i) - y_i)^2$$

Why is this sensible?

1. Each term in the sum is 0 if $h(x_i)$ is *exactly* y_i.
2. Each term *increases* as the difference between $h(x_i)$ and y_i increases.
3. We add the terms for all examples.

Curve fitting

If we pick h using this method then we get:



The chosen h is close to the target h', even though it was chosen *using only a small number of noisy examples*.

It is not quite identical to the target concept.

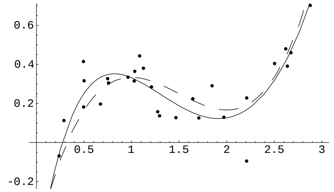
However if we were given a new point $\mathbf{x}'$ and asked to guess the value $h'(\mathbf{x}')$ then guessing $h(\mathbf{x}')$ might be expected to do quite well.

Curve fitting

Problem: we don't know *what $\mathcal{H}$ nature is using*. What if the one we choose doesn't match? We can make *our* $\mathcal{H}$ 'bigger' by defining it as

$$\mathcal{H} = \{h : h \text{ is a polynomial of degree at most } 5\}$$

If we use the same learning algorithm then we get:



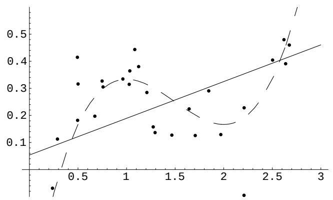
The result in this case is similar to the previous one: h is again quite close to h', but not quite identical.

Curve fitting

So what's the problem? Repeating the process with,

$$\mathcal{H} = \{h : h \text{ is a polynomial of degree at most } 1\}$$

gives the following:



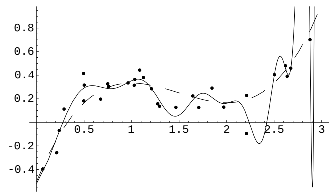
In effect, we have made *our* $\mathcal{H}$ too 'small'. It does not in fact contain any hypothesis similar to h'.

Curve fitting

So we have to make $\mathcal{H}$ huge, right? WRONG!!! With

$$\mathcal{H} = \{h : h \text{ is a polynomial of degree at most } 25\}$$

we get:



BEWARE!!! This is known as *overfitting*.

Curve fitting

An experiment to gain some further insight: using

$$h'(x) = \frac{1}{10}x^{10} - \frac{1}{12}x^8 + \frac{1}{15}x^6 + \frac{1}{3}x^3 - \frac{3}{2}x^2 + 2x - \frac{1}{2}.$$

as the unknown underlying function.

We can look at how *the degree of the polynomial the training algorithm can output affects the generalisation ability of the resulting* h.

We use the same training algorithm, and we train using

$$\mathcal{H} = \{h : h \text{ is a polynomial of degree at most } d\}$$

for values of d ranging from 1 to 30

Curve fitting

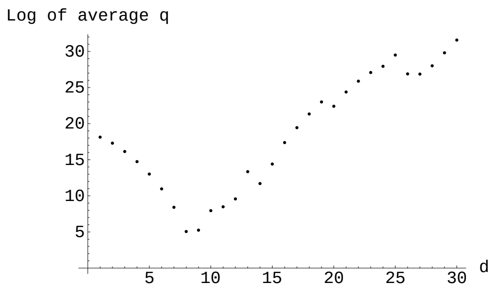
- Each time we obtain an h of a given degree—call it h_d—we assess its quality *using a further* 100 *inputs* $\mathbf{x}'_i$ *generated at random* and calculating

$$q(d) = \frac{1}{100}\sum_{i=1}^{100}(h'(\mathbf{x}'_i) - h_d(\mathbf{x}'_i))^2$$

- As the values $q(d)$ are found using inputs that are not necessarily included in the training sequence *they measure generalisation*.
- To smooth out the effects of the random selection of examples we repeat this process 100 times and average the values $q(d)$.

Curve fitting

Here is the result:



Clearly: we need to choose $\mathcal{H}$ sensibly if we want to obtain *good generalisation performance*.

The perceptron

The example just given illustrates much of what we want to do. However in practice we deal with *more than a single dimension*.

The simplest form of hypothesis used is the *linear discriminant*, also known as the *perceptron*. Here

$$h(\mathbf{w};\mathbf{x}) = \sigma\left(w_0 + \sum_{i=1}^{m} w_i x_i\right) = \sigma\left(w_0 + w_1x_1 + w_2x_2 + \cdots + w_nx_n\right)$$

So: we have a *linear function* modified by the *activation function* σ.

The perceptron's influence continues to be felt in the recent and ongoing development of *support vector machines*.

The perceptron activation function I

There are three standard forms for the activation function:

1. *Linear*: for *regression problems* we often use

$$\sigma(z) = z$$

2. *Step*: for *two-class classification problems* we often use

$$\sigma(z) = \begin{cases} C_1 & \text{if } z > 0 \\ C_2 & \text{otherwise.} \end{cases}$$

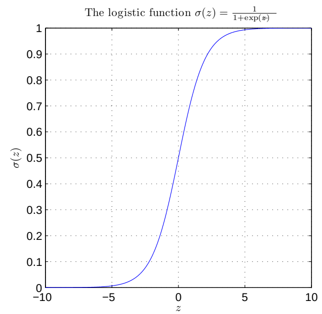
3. *Sigmoid/Logistic*: for *probabilistic classification* we often use

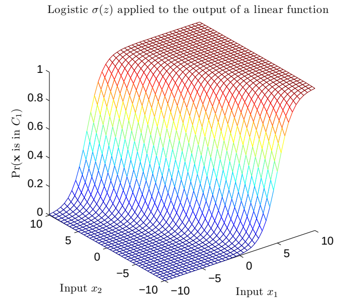
$$\Pr(\mathbf{x} \text{ is in } C_1) = \sigma(z) = \frac{1}{1 + \exp(-z)}.$$

The *step function* is important but the algorithms involved are somewhat different to those we'll be seeing. We won't consider it further.

The *sigmoid/logistic function* plays a major role in what follows.

The sigmoid/logistic function





Gradient descent

A method for *training a basic perceptron* works as follows. Assume we're dealing with a *regression problem* and using $\sigma(z) = z$.

We define a measure of *error* for a given collection of weights. For example

$$E(\mathbf{w}) = \sum_{i=1}^{m} (y_i - h(\mathbf{w}; \mathbf{x}_i))^2$$

Modifying our notation slightly so that

$$\mathbf{x}^T = \begin{pmatrix} 1 & x_1 & x_2 & \cdots & x_n \end{pmatrix}$$
$$\mathbf{w}^T = \begin{pmatrix} w_0 & w_1 & w_2 & \cdots & w_n \end{pmatrix}$$

lets us write

$$E(\mathbf{w}) = \sum_{i=1}^{m} (y_i - \mathbf{w}^T \mathbf{x}_i)^2$$

Gradient descent

We want to *minimise* $E(\mathbf{w})$.

One way to approach this is to start with a random $\mathbf{w}_0$ and update it as follows:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \left. \frac{\partial E(\mathbf{w})}{\partial \mathbf{w}} \right|_{\mathbf{w}_t}$$

where

$$\frac{\partial E(\mathbf{w})}{\partial \mathbf{w}} = \begin{pmatrix} \frac{\partial E(\mathbf{w})}{\partial w_0} & \frac{\partial E(\mathbf{w})}{\partial w_1} & \cdots & \frac{\partial E(\mathbf{w})}{\partial w_n} \end{pmatrix}^T$$

and η is some small positive number.

The vector

$$-\frac{\partial E(\mathbf{w})}{\partial \mathbf{w}}$$

tells us the *direction of the steepest decrease in* $E(\mathbf{w})$.

Gradient descent

With

$$E(\mathbf{w}) = \sum_{i=1}^{m} (y_i - \mathbf{w}^T \mathbf{x}_i)^2$$

we have

$$\begin{aligned}
\frac{\partial E(\mathbf{w})}{\partial w_j} &= \frac{\partial}{\partial w_j} \left(\sum_{i=1}^{m} (y_i - \mathbf{w}^T \mathbf{x}_i)^2 \right) \\
&= \sum_{i=1}^{m} \left(\frac{\partial}{\partial w_j} (y_i - \mathbf{w}^T \mathbf{x}_i)^2 \right) \\
&= \sum_{i=1}^{m} \left(2(y_i - \mathbf{w}^T \mathbf{x}_i) \frac{\partial}{\partial w_j} \left(-\mathbf{w}^T \mathbf{x}_i \right) \right) \\
&= -2 \sum_{i=1}^{m} \mathbf{x}_i^{(j)} \left(y_i - \mathbf{w}^T \mathbf{x}_i \right)
\end{aligned}$$

where $\mathbf{x}_i^{(j)}$ is the jth element of $\mathbf{x}_i$.

Gradient descent

The method therefore gives the algorithm

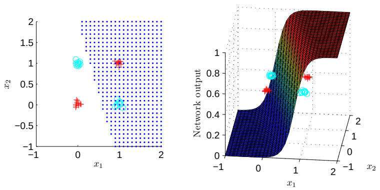
$$\mathbf{w}_{t+1} = \mathbf{w}_t + 2\eta \sum_{i=1}^{m} \left(y_i - \mathbf{w}_t^T \mathbf{x}_i \right) \mathbf{x}_i$$

Some things to note:

- In this case $E(\mathbf{w})$ is *parabolic* and has a *unique global minimum* and *no local minima* so this works well.
- *Gradient descent* in some form is a very common approach to this kind of problem.
- We can perform a similar calculation for *other activation functions* and for *other definitions for* $E(\mathbf{w})$.
- Such calculations lead to *different algorithms*.

Perceptrons aren't very powerful: the parity problem

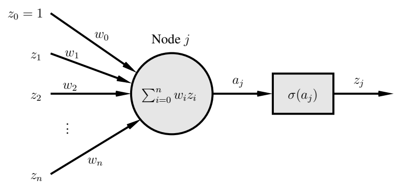
There are many problems a perceptron can't solve.



We need a network that computes *more interesting functions*.

The multilayer perceptron

Each *node* in the network is itself a perceptron:



- *Weights* w_i connect nodes together.
- a_j is the weighted sum or *activation* for node j.
- σ is the *activation function*.
- The *output* is $z_j = \sigma(a_j)$.

The multilayer perceptron

Reminder:

We'll continue to use the notation

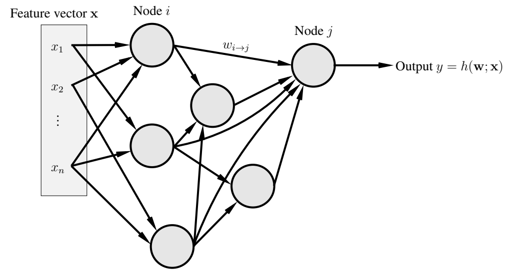
$$\mathbf{z}^T = (\ 1\ \ z_1\ \ z_2\ \ \cdots\ \ z_n\ )$$
$$\mathbf{w}^T = (\ w_0\ \ w_1\ \ w_2\ \ \cdots\ \ w_n\ )$$

So that

$$\sum_{i=0}^{n} w_i z_i = w_0 + \sum_{i=1}^{n} w_i z_i$$
$$= \mathbf{w}^T \mathbf{z}$$

The multilayer perceptron

In the general case we have a *completely unrestricted feedforward structure*:



Each node is a perceptron. *No specific layering* is assumed.

$w_{i\to j}$ connects node i to node j. w_0 for node j is denoted $w_{0\to j}$.

Backpropagation

As usual we have:

- Instances $\mathbf{x}^T = (x_1, \ldots, x_n)$.
- A training sequence $\mathbf{s} = ((\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m))$.

We also define a measure of training error

$$E(\mathbf{w}) = \text{measure of the error of the network on } \mathbf{s}$$

where $\mathbf{w}$ is the vector of *all the weights in the network*.

Our aim is to find a set of weights that *minimises* $E(\mathbf{w})$ using *gradient descent*.

Backpropagation: the general case

The *central task* is therefore to calculate

$$\frac{\partial E(\mathbf{w})}{\partial \mathbf{w}}$$

To do that we need to calculate the individual quantities

$$\frac{\partial E(\mathbf{w})}{\partial w_{i\to j}}$$

for *every weight* $w_{i\to j}$ *in the network*.

Often $E(\mathbf{w})$ is the sum of separate components, one for each example in $\mathbf{s}$

$$E(\mathbf{w}) = \sum_{p=1}^{m} E_p(\mathbf{w})$$

in which case

$$\frac{\partial E(\mathbf{w})}{\partial \mathbf{w}} = \sum_{p=1}^{m} \frac{\partial E_p(\mathbf{w})}{\partial \mathbf{w}}$$

We can therefore consider examples individually.

Backpropagation: the general case

Place example p at the input and calculate a_j and z_j for *all nodes* including the output y. This is *forward propagation*.

We have

$$\frac{\partial E_p(\mathbf{w})}{\partial w_{i \to j}} = \frac{\partial E_p(\mathbf{w})}{\partial a_j} \frac{\partial a_j}{\partial w_{i \to j}}$$

where $a_j = \sum_k w_{k \to j} z_k$.

Here the sum is over *all the nodes connected to node* j. As

$$\frac{\partial a_j}{\partial w_{i \to j}} = \frac{\partial}{\partial w_{i \to j}} \left(\sum_k w_{k \to j} z_k \right) = z_i$$

we can write

$$\frac{\partial E_p(\mathbf{w})}{\partial w_{i \to j}} = \delta_j z_i$$

where we've defined

$$\delta_j = \frac{\partial E_p(\mathbf{w})}{\partial a_j}$$

Backpropagation: the general case

So we now need to calculate the values for δ_j...

When j is the *output node*—that is, the one producing the output $y = h(\mathbf{w}; \mathbf{x}_p)$ of the network—this is easy as $z_j = y$ and

$$\begin{aligned}\delta_j &= \frac{\partial E_p(\mathbf{w})}{\partial a_j} \\ &= \frac{\partial E_p(\mathbf{w})}{\partial y} \frac{\partial y}{\partial a_j} \\ &= \frac{\partial E_p(\mathbf{w})}{\partial y} \sigma'(a_j)\end{aligned}$$

using the fact that $y = \sigma(a_j)$.

Backpropagation: the general case

The first term is in general easy to calculate for a given E as the error is generally just a measure of the distance between y and the label in the training sequence.

Example: when

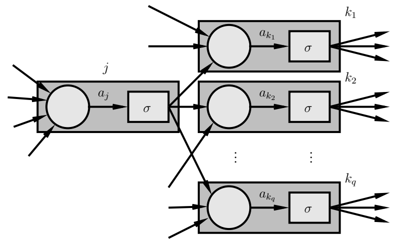
$$E_p(\mathbf{w}) = (y - y_p)^2$$

we have

$$\begin{aligned}\frac{\partial E_p(\mathbf{w})}{\partial y} &= 2(y - y_p) \\ &= 2(h(\mathbf{w}; \mathbf{x}_p) - y_p)\end{aligned}$$

Backpropagation: the general case

When j is *not an output node* we need something different:

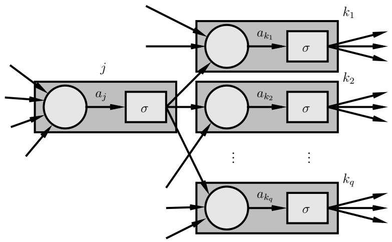


We're interested in

$$\delta_j = \frac{\partial E_p(\mathbf{w})}{\partial a_j}$$

Altering a_j can affect *several other nodes* $k_1, k_2, \ldots, k_q$ *each of which can in turn affect* $E_p(\mathbf{w})$.

Backpropagation: the general case

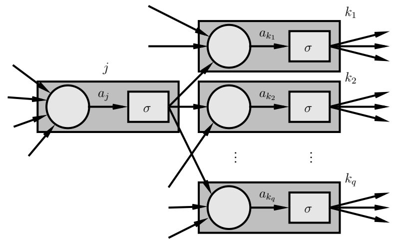


We have

$$\delta_j = \frac{\partial E_p(\mathbf{w})}{\partial a_j} = \sum_{k \in \{k_1, k_2, \ldots, k_q\}} \frac{\partial E_p(\mathbf{w})}{\partial a_k} \frac{\partial a_k}{\partial a_j} = \sum_{k \in \{k_1, k_2, \ldots, k_q\}} \delta_k \frac{\partial a_k}{\partial a_j}$$

where $k_1, k_2, \ldots, k_q$ are the nodes to which node j sends a connection.

Backpropagation: the general case

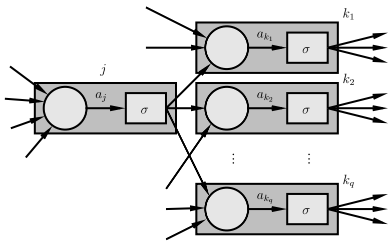


Because we know how to compute δ_j *for the output node* we can *work backwards* computing further δ values.

We will *always know all the values* δ_k *for nodes ahead of where we are*.

Hence the term *backpropagation*.

Backpropagation: the general case



$$\frac{\partial a_k}{\partial a_j} = \frac{\partial}{\partial a_j}\left(\sum_i w_{i \to k} \sigma(a_i)\right) = w_{j \to k} \sigma'(a_j)$$

and

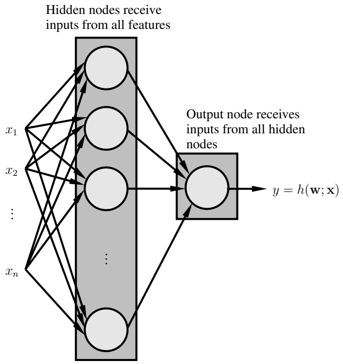
$$\delta_j = \sum_{k \in \{k_1, k_2, \ldots, k_q\}} \delta_k w_{j \to k} \sigma'(a_j) = \sigma'(a_j) \sum_{k \in \{k_1, k_2, \ldots, k_q\}} \delta_k w_{j \to k}$$

Backpropagation: the general case

Summary: to calculate $\frac{\partial E_p(\mathbf{w})}{\partial \mathbf{w}}$ for the pth pattern:

1. *Forward propagation*: apply $\mathbf{x}_p$ and calculate outputs *etc* for *all the nodes in the network*.
2. *Backpropagation 1*: for the *output* node
$$\frac{\partial E_p(\mathbf{w})}{\partial w_{i \to j}} = z_i \delta_j = z_i \sigma'(a_j) \frac{\partial E_p(\mathbf{w})}{\partial y}$$
where $y = h(\mathbf{w}; \mathbf{x}_p)$.
3. *Backpropagation 2*: For other nodes
$$\frac{\partial E_p(\mathbf{w})}{\partial w_{i \to j}} = z_i \sigma'(a_j) \sum_k \delta_k w_{j \to k}$$
where the δ_k were calculated at an earlier step.

Backpropagation: a specific example



For the output: $\sigma(a) = a$. For the hidden nodes $\sigma(a) = \frac{1}{1+\exp(-a)}$.

Backpropagation: a specific example

For the output: $\sigma(a) = a$ so $\sigma'(a) = 1$.

For the hidden nodes:

$$\sigma(a) = \frac{1}{1 + \exp(-a)}$$

so

$$\sigma'(a) = \sigma(a)\,[1 - \sigma(a)]$$

We'll continue using the same definition for the error

$$E(\mathbf{w}) = \sum_{p=1}^{m} (y_p - h(\mathbf{w}; \mathbf{x}_p))^2$$

$$E_p(\mathbf{w}) = (y_p - h(\mathbf{w}; \mathbf{x}_p))^2$$

Backpropagation: a specific example

For the output: the equation is

$$\frac{\partial E_p(\mathbf{w})}{\partial w_{i \to \text{output}}} = z_i \delta_{\text{output}} = z_i \sigma'(a_{\text{output}}) \frac{\partial E_p(\mathbf{w})}{\partial y}$$

where $y = h(\mathbf{w}; \mathbf{x}_p)$. So as

$$\begin{aligned} \frac{\partial E_p(\mathbf{w})}{\partial y} &= \frac{\partial}{\partial y}\left((y_p - y)^2\right) \\ &= 2(y - y_p) \\ &= 2\,[h(\mathbf{w}; \mathbf{x}_p) - y_p] \end{aligned}$$

and $\sigma'(a) = 1$ so

$$\delta_{\text{output}} = 2\,[h(\mathbf{w}; \mathbf{x}_p) - y_p]$$

and

$$\boxed{\frac{\partial E_p(\mathbf{w})}{\partial w_{i \to \text{output}}} = 2 z_i (h(\mathbf{w}; \mathbf{x}_p) - y_p)}$$

Backpropagation: a specific example

For the hidden nodes: the equation is

$$\frac{\partial E_p(\mathbf{w})}{\partial w_{i \to j}} = z_i \sigma'(a_j) \sum_k \delta_k w_{j \to k}$$

However *there is only one output* so

$$\frac{\partial E_p(\mathbf{w})}{\partial w_{i \to j}} = z_i \sigma(a_j)\,[1 - \sigma(a_j)]\,\delta_{\text{output}} w_{j \to \text{output}}$$

and we know that

$$\delta_{\text{output}} = 2\,[h(\mathbf{w}; \mathbf{x}_p) - y_p]$$

so

$$\boxed{\begin{aligned} \frac{\partial E_p(\mathbf{w})}{\partial w_{i \to j}} &= 2 z_i \sigma(a_j)\,[1 - \sigma(a_j)]\,[h(\mathbf{w}; \mathbf{x}_p) - y_p]\,w_{j \to \text{output}} \\ &= 2 x_i z_j (1 - z_j)\,[h(\mathbf{w}; \mathbf{x}_p) - y_p]\,w_{j \to \text{output}} \end{aligned}}$$

Putting it all together

We can then use the derivatives in one of two basic ways:

Batch: (as described previously)

$$\frac{\partial E(\mathbf{w})}{\partial \mathbf{w}} = \sum_{p=1}^{m} \frac{\partial E_p(\mathbf{w})}{\partial \mathbf{w}}$$

then

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \left. \frac{\partial E(\mathbf{w})}{\partial \mathbf{w}} \right|_{\mathbf{w}_t}$$

Sequential: using just one pattern at once

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \left. \frac{\partial E_p(\mathbf{w})}{\partial \mathbf{w}} \right|_{\mathbf{w}_t}$$

selecting patterns *in sequence or at random*.

Example: the parity problem revisited

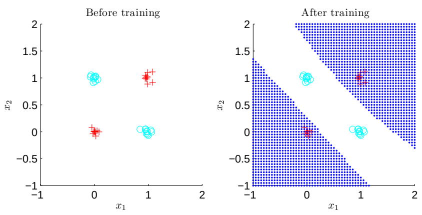
As an example we show the result of training a network with:

- Two inputs.
- One output.
- One hidden layer containing 5 units.
- $\eta = 0.01$.
- All other details as above.

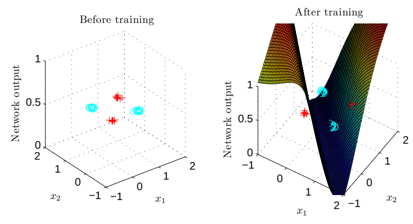
The problem is the parity problem. There are 40 noisy examples.

The sequential approach is used, with 1000 repetitions through the entire training sequence.

Example: the parity problem revisited



Example: the parity problem revisited



Example: the parity problem revisited