Introduction: what are our aims?

Artificial Intelligence (AI) is currently at the top of its periodic hype-cycle. Much of this has been driven by philosophers and people with something to sell.

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? What are the ethical implications?
- Is AI impossible? (Note: I didn’t write possible here, for a good reason...)

Despite 2000 years of work by philosophers, there’s essentially nothing in the way of results.

Reading: AIMA chapters 1, 2, 26 and 27.

Introduction: what are our aims?

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.

Historically speaking, this view seems to be the more successful...

AI has been entering our lives for decades, almost without us being aware of it. But be careful: brains are much more complex than you think.
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 Stoicism; 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 Russell 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).

What has been achieved?

Artificial Intelligence (AI) is currently at the top of its periodic hype-cycle.

As a result, it’s important to maintain some sense of perspective.

Notable successes:
- Perception: vision, speech processing, inference of emotion from video, scene labelling, touch sensing, artificial noses...
- Logical reasoning: prolog, expert systems, CYC, Bayesian reasoning, Watson...
- Playing games: chess, backgammon, go, robot football...
- Diagnosis of illness in various contexts...
- Theorem proving: Robbin’s conjecture, formalization of the Kepler conjecture...
- Literature and music: automated writing and composition...

And many more... (most of which don’t include the word ‘DEEP!’)

Aside: when something is understood it stops being AI

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.

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.

The Turing test is informative, and (very!) hard to pass. (See the Loebner Prize...)

• 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.

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

What is AI, version three: thinking rationally and the “laws of thought”

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, but there are obstacles. It is hard to:

• Represent commonsense knowledge.
• Deal with uncertainty.
• Reason without being tripped up by computational complexity.
• Sometimes it’s necessary to act when there’s no logical course of action.
• Sometimes inference is unnecessary (reflex actions).

These will be recurring themes in this course, and in Machine Learning and Bayesian Inference next year.
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?
- 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.
- 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.**

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 when the importance of **uncertainty** was fully appreciated. **Machine Learning and Bayesian Inference** 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.)

Other fields that have contributed to AI
Introductory reading that isn’t nonsense

• Francis Crick, “The recent excitement about neural networks”, Nature (1989) is still entirely relevant:
  http://www.nature.com/nature/journal/v337/n6203/abs/337129a0.html
• The Loebner Prize in Artificial Intelligence:
  www.aisb.org.uk/events/loebner-prize
  provides a good illustration of how far we are from passing the Turing test.
• Marvin Minsky, “Why people think computers can’t”, AI Magazine (1982) is an excellent response to nay-saying philosophers.
• Watson: researcher.watson.ibm.com/researcher/view_group_pubs.php?grp=2099
• Go: www.nature.com/nature/journal/v529/n7587/full/nature16961.html
• The Cyc project: www.cyc.com
• 2007 DARPA Urban Challenge: cs.stanford.edu/group/roadrunner
• AI at Nasa Ames:
  www.nasa.gov/centers/ames/research/exploringtheuniverse/spiffy.html

Introductory reading that isn’t nonsense

• AI in the UK: ready, willing and able?
  House of Lords, Select Committee on Artificial Intelligence
  https://publications.parliament.uk/pa/ld201719/ldselect/ldai/100/100.pdf
• Machine learning: the power and promise of computers that learn by example
  The Royal Society
  https://royalsociety.org/topics-policy/projects/machine-learning/
• Building machines that learn and think like people

Text book

The course is based on the relevant parts of:


For more depth on specific areas see:


Prerequisites

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

DIRE WARNING:

No doubt you want to know something about machine learning, given the recent peek in interest.
In the lectures on machine learning I will be talking about neural networks.
I will introduce the backpropagation algorithm, which is the foundation for both classical neural networks and the more fashionable deep learning methods.
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 \( \frac{\partial f}{\partial x_j} \) 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 \( \frac{\partial f}{\partial y_j} \) 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.

---

**Agents**

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

Allow me to introduce *EVIL ROBOT.*

![Agent Diagram]

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.

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

**Examples:** chess player (rating), mail-filter (various measures), self-driving car (comfort, journey time, safety and so on).

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.
- 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.
- For multiple agents: whether the situation is competitive or cooperative, and whether communication is required.

### Programming agents
A basic agent can be thought of as working on a straightforward underlying process. To achieve some goal:

- 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

There are (at least) two obvious but very limited approaches to this. While limited, they suggest a couple of important points.

**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; its behaviour should depend on its own experience of the world via the percept sequence.

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

Where autonomy is concerned, note that even humans have some built-in knowledge, and some animals, such as dung beetles are not fully autonomous.

Reflex agents

**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, and having a goal

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.

It also 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.

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.

Learning agents

It seems reasonable that an agent should learn from experience.

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?

Artificial Intelligence I

Problem solving by search

Reading: AIMA chapters 3 and 4.
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.

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.

A simple example: the 8-puzzle.

**Start state:** a randomly-selected configuration of the numbers 1 to 8 arranged on a $3 \times 3$ 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 $5 \times 5$ version has about $10^{25}$ states, and a random instance is in fact quite a challenge.

From the pre-PC dark ages. Christmas was grim...

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 its 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.
Contingency problems: In some situations it is necessary to perform sensing while the actions are being carried out. This kind of problem requires planning and will be dealt with later.
Exploration problems: Sometimes you have no knowledge of the effect that your actions have on the environment. This means you need to experiment to find out what happens when you act. This kind of problem requires reinforcement learning for a solution.

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 $p$. 
The basic tree-search algorithm

In pseudo-code, the algorithm looks like this:

```python
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.

Search trees versus search graphs

We need to make an important distinction between search trees and search graphs.

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 versus search graphs

Graphs can lead to problems:

The sliding blocks puzzle for example suffers this way.

Graph search

In pseudocode:

```plaintext
function graphSearch() {
    closed = {};
    fringe = queue containing only the start state;
    while (! empty(fringe)) {
        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. We may need to check which solution is better and if necessary modify path costs and depths for descendants of the repeated state.

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...
Basic search algorithms

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 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.

In practice it is the memory requirement that is problematic.

Basic search methods

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 $O(bd)$.

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, ...

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...

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.)

In practice however it can be problematic. See the problem sheet...

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

$$\forall n \left( \forall n' \in \text{successors}(n) : p(n') \geq p(n) \right)$$

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

So far: 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 get usable algorithms we need to exploit knowledge of the distance between the current state and a goal. This is problem-specific knowledge.

- 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.)

Greedy search

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

We’ve already seen path cost used for this purpose. (Uniform-cost search.)

- 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 $h(n) = 0$.

Note again that this is a problem-dependent measure.

We are required either to design it using our knowledge of the problem, or by some other means.

Example: route-finding

Example: for route finding a reasonable heuristic function is

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

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

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* search
Well, we can.
A* 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

$p(n) =$ cost of path to $n$

and $h(n) =$ estimated cost of best path from $n$

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

A* tree-search is optimal for admissible $h(n)$

To see that tree-search $A^*$ is optimal we reason as follows. Let Goal$_{opt}$ be an optimal goal state with $f(Goal_{opt}) = p(Goal_{opt}) = f_{opt}$ (because $h(Goal_{opt}) = 0$).

Let $n$ be a leaf node in the fringe on an optimal path to Goal$_{opt}$. So $f_{opt} \geq p(n) + h(n) = f(n)$ 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 $f_{opt} \geq f_2 = p(Goal_2)$.

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

A* search
A* 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 $A^*$ is optimal.

A* tree-search is optimal for admissible $h(n)$

Let $n$ be a leaf node in the fringe on an optimal path to Goal$_{opt}$. So $f_{opt} \geq p(n) + h(n) = f(n)$ 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 $f_{opt} \geq f_2 = p(Goal_2)$.

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

\[ \text{monotonicity} \rightarrow \text{admissibility} \]

this is an important property.

---

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.

\[ h(n) \leq \text{cost}(n \rightarrow n') + h(n') \]

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* graph search is optimal for monotonic heuristics

A* 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

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

Thus

\[ f(n') = p(n') + h(n') \]
\[ = p(n) + \text{cost}(n \xrightarrow{a} n') + h(n') \]
\[ \geq p(n) + h(n) \]
\[ = f(n) \]

where the inequality follows from equation 1.

We therefore have the following situation:

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

A* search is complete

A* 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* 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* 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* search also stores all the nodes it generates, once again it is generally memory that becomes a problem before time.
How might we improve the way in which $A^*$ 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.

```c
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;
    }
}
```

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.

```c
(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);
}
```

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.

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* search

And again:

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

Properties of IDA*:
• It is complete and optimal under the same conditions as A*.
• 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 $\epsilon$ at each stage, guaranteeing a solution at most $\epsilon$ 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.

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

```java
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:

Function call number 2:

Now perform the recursive function call (result2, f′) = RBFS(best2, 5) so f(best2) takes the returned value f′.

Function call number 3:

Now f(best3) > fLimit3 so the function call returns (fail, 10) into (result3, f′) and f(best2) = 10.

The while loop for function call 2 now repeats:

Now f(best2) > fLimit2 so the function call returns (fail, 9) into (result2, f′) and f(best1) = 9.
Recursive best-first search (RBFS): an example

The while loop for function call now repeats:

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

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 \textit{goal} that we’re interested in. The \textit{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 \textit{the best}.
- This is also known as optimisation.

This leads to the remarkably simple concept of \textit{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:

\[
f(n) = 52
\]

\[
f(n) = 24
\]

\[
f(n) = 1
\]

\[
f(n) = 24
\]

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

Find an arrangement of \( m \) queens on an \( m \) by \( m \) 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 \( m \). (Imagine \( m \simeq 1,000,000 \).)

The \( n \)-queens problem

You may be familiar with the \( n \)-queens problem.

Find an arrangement of \( m \) queens on an \( m \) by \( m \) 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 \( m \). (Imagine \( m \simeq 1,000,000 \).)

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.)

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

\(^2\)Note that we actually want to minimize \( f \) here. This is equivalent to maximizing \(-f\), and I will generally use whichever seems more appropriate.
Hill-climbing search

Hill-climbing search is remarkably simple:

Generate a start state n.
while () {
  Generate the N neighbours \{n_1, \ldots, n_N\} of n;
  if (max\{f(n_i)\} \leq 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

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

\[
Pr(n'|n) = \begin{cases} 0 & \text{if } n' \in N^-(n) \\ \frac{1}{2} (f(n') - f(n)) & \text{otherwise.} \end{cases}
\]

- **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.

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.
Gradient ascent and related methods

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

Typically, we have a function $f(x) : \mathbb{R}^n \rightarrow \mathbb{R}$ and we want to find $x_{\text{opt}} = \arg\max_x f(x)$.

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

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 $\epsilon$ can lead to a different problem:

We can easily jump too far...

Artificial Intelligence I

Games (adversarial search)

Reading: AIMA chapter 5.

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.

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.

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.

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

- **Line search**: increase $\epsilon$ until $f$ decreases and maximise in the resulting interval. Then choose a new direction to move in. Conjugate gradients, the Fletcher-Reeves and Polak-Ribiere methods etc.
- **Use $H$ 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

Games (adversarial search)
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.

And chess isn’t even very hard: Go is much harder...

Perfect decisions in a two-person game

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

- We’ll use \textit{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—ЕВИЛ РОБОТ 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 \textit{initial state}.

\begin{center}
\begin{tabular}{|c|c|c|}
\hline
0 & 0 & 0 \\
\hline
0 & 0 & 0 \\
\hline
\end{tabular}
\end{center}

Max to move

- There is a set of \textit{operators}. Here, Max can place a cross in any empty square, or Min a nought.
- There is a \textit{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 \textit{utility} or \textit{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 \textit{construct a tree} to represent a game. From the initial state Max can make nine possible moves:

\begin{center}
\begin{tabular}{|c|c|c|}
\hline
0 & 0 & 0 \\
\hline
0 & 0 & 0 \\
\hline
\end{tabular}
\end{center}

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.

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

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

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

Moving one further step up the tree:

4 5 2 20 20 15 6 7 1 4 10 9 5 8 5 4
1 42
6
6

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

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.

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

\[ \text{eval}(\text{position}) = \sum_{\text{black’s pieces}} \text{value of } p_i - \sum_{\text{white’s pieces}} \text{value of } q_i \]

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

- 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.

So in fact this seems highly naive...

\[ \alpha - \beta \text{ pruning} \]

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 5 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.

\[ \alpha - \beta \text{ pruning} \]

Returning for a moment to the earlier, simplified example:

The search is depth-first and left to right.
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.

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 $\alpha$ and $\beta$ where

$\alpha =$ the highest utility seen so far on the path for Max

$\beta =$ the lowest utility seen so far on the path for Min

Assume Max begins. Initial values for $\alpha$ and $\beta$ are

$\alpha = -\infty$

and

$\beta = +\infty$.

So: we start with the function call

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

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($\alpha, \beta, n'$));
        if(value > $\beta$) return value;
        if(value > $\alpha$) $\alpha$ = value;
    }
    return value;
}
```
**α−β pruning in general**

The function `opponent` is exactly analogous:

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

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

---

**How effective is α−β 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 α−β 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!

---

**α−β pruning in general**

Applying this to the earlier example and keeping track of the values for $\alpha$ and $\beta$

You should obtain:

```
Return 1
Return 2
Return 3

- $\alpha = -\infty$
- $\beta = +\infty$
```

```
4 5 2 20 20 15 6 7 1
```

```
Return 2
```

```
6
```

```
2
```

```
\alpha = -\infty
\beta = +\infty
```

**130**

---

**How effective is α−β pruning?**

If moves are arranged at random then α−β 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.

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. 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.

Reading: AIMA chapter 6.

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—and of the application of CSPs—when we discuss planning.

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.

We’ll see an example of this—and of the application of CSPs—when we discuss planning.
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.

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

Backtracking search now takes on a very simple form: search depth-first, assigning a single variable at a time, and backtrack if no valid assignment is available.

Using the graph colouring example, the search now looks something like this...

Rather than using problem-specific heuristics to try to improve searching, we can now explore heuristics applicable to general CSPs.
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?
• Can we try to force the search in a successful direction (remember the use of heuristics)?
• Can we try to force failures/backtracks to occur quickly?

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.

Start with 5, 3 or 7.

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

Assigning such variables first is called the minimum remaining values (MRV) heuristic.
(Alternatively, the most constrained variable or fail first heuristic.)

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.

We can visualise this process as follows:

![Diagram showing forward checking]

At the fourth step, 7 has no possible assignments left. However, we could have detected a problem a little earlier... by looking at step three.

**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?

The AC-3 algorithm

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

Complexity:

- A binary CSP with \( n \) variables can have \( O(n^2) \) directional constraints \( i \rightarrow j \).
- Any \( i \rightarrow j \) 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 \( O(n^2d^3) \).

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 \( k \)th 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.

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 \( V_j \). More formally: if \( I_k \) conflicts with \( V_{k+1} \) we backtrack to \( V_j \) where

\[
j = \min\{ j \leq i | I_j \text{ conflicts with } V_{k+1} \}.
\]

If there are no possible values left to try for \( V_j \) then we backtrack **chronologically**.
Gaschnig’s algorithm

Example:

Backtrack to 5

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.

---

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

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.

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.

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

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...

---

**Example:**

Session of $V_4 = \{V_4, V_5, V_6, V_7\}$.

- Backjump from $V_7$ to $V_4$.
- Nothing left to try!
  
  $R(V_4) = \{V_4, V_7\}$

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

---

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

- 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.**
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
     \[ V_4 \leq V_{10} + 5 \]
   - 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.

Knowledge representation and reasoning

We now look at how an agent might represent knowledge about its environment, and reason with this knowledge to achieve its goals.

Initially we’ll represent and reason using first order logic (FOL). 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.

Using FOL in all its glory can be problematic.

Later we’ll look at how some of the problems can be addressed using semantic networks, frames, inheritance and rules.

Reading: AIMA, chapters 7 to 10.
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.

FOL (arguably?) seems to provide a good way in which to represent the required kinds of knowledge: it is expressive, concise, unambiguous, it can be adapted to different contexts, and it has an inference procedure, although a semidecidable one.

In addition, it 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

\[
\forall S . \forall S' . ((\forall x . (x \in S \iff x \in S')) \Rightarrow S = S')
\]

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**

**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.

---

**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: **KB ||= \( \alpha \)** means that the **KB** entails \( \alpha \).
- Proof: **KB \vdash \_i \alpha** means that \( \alpha \) 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 \( \alpha \).
- \( \_i \) is complete if it can find a proof for any entailed \( \alpha \).

---

**Example: Prolog**

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

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

is a program to concatenate two lists. The query

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

results in

```prolog
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

\[
\forall x. \text{concat}(\text{empty}, x, x) \land \\
\forall h, t, l_1, l_2. \text{concat}(t, l_1, l_2) \implies \text{concat}(\text{cons}(h, t), l_1, \text{cons}(h, l_2))
\]

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 \(L_1, L_2, ..., L_n\) then the Prolog programme corresponds to the KB

\[
\text{KB} = L_1 \land L_2 \land \cdots \land L_n
\]

Now, what does the query mean?

Prolog and FOL

When you give the query

\[
\text{concat}([1, 2, 3], [4, 5], X).
\]

to Prolog it responds by trying to prove the following statement

\[
\text{KB} \implies \exists x. \text{concat}([1, 2, 3], [4, 5], x)
\]

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

https://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.

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]). [assumption].
2 (exists X concat([1,2,3],[4,5],X)) [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([1,3],[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  
  `∃actionList.Goal(... actionList ...)`

- Proving that  
  `KB ⇒ ∃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

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

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

\[
\begin{align*}
\text{result}(\text{grab}, s_0) &= s_1 \\
\text{result}(\text{turnLeft}, s_1) &= s_2 \\
\text{result}(\text{shoot}, s_2) &= s_3 \\
\text{result}(\text{forward}, s_3) &= s_4 \\
&\text{etc.}
\end{align*}
\]

Axioms I: possibility axioms

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

We introduce a predicate \( \text{Poss}(\text{action}, s) \)
denoting that an action can be performed in situation \( s \).

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

\( \text{At}(l, s) \land \text{Available}(\text{gold}, l, s) \Rightarrow \text{Poss}(\text{grab}, s) \)

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:

\( \text{Poss}(\text{grab}, s) \Rightarrow \text{Have}(\text{gold}, \text{result}(\text{grab}, s)) \)

Effect axioms describe the way in which the world changes.

We would probably also include

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

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:

```latex
\text{Have}(o,s) \wedge \\
\neg(a = \text{release} \wedge o = \text{gold}) \wedge \neg(a = \text{shoot} \wedge o = \text{arrow}) \\
\implies \text{Have}(o,\text{result}(a,s))
```

describes the effect of having something and not discarding it.

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

```latex
\neg \text{Have}(o,s) \wedge \\
(a \neq \text{grab}(o) \vee \neg(\text{Available}(a,s) \wedge \text{Portable}(o))) \\
\implies \neg \text{Have}(o,\text{result}(a,s))
```

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 $a$ is possible $\implies$

- (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

```latex
\text{Poss}(a,s) \implies \\
(\text{Have}(a,\text{result}(a,s)) \iff ((a = \text{grab} \wedge \text{Available}(a,s)) \vee \\
(\text{Have}(a,s) \wedge \neg(a = \text{release} \wedge o = \text{gold}) \wedge \\
\neg(a = \text{shoot} \wedge o = \text{arrow}))))
```

---

**Knowing where you are, and so on...**

We now have considerable flexibility in adding further rules:

- If $s_0$ is the initial situation we know that $\text{At}((1,1), s_0)$.
- 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. We might assume facing($s_0$) = 0.
- We need to know how motion affects location

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

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

  and so on.
- The concept of adjacency is very important in the Wumpus world

  $\text{Adjacent}(t_1, t_2) \iff \exists d \text{ forwardResult}(t_1, d) = t_2$

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

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

\[
\text{Poss}(a, s) \implies (\text{At}(o, l, \text{result}(a, s)) \iff (a = \text{go}(l', l) \land \{o = \text{robot} \lor \text{Has(robot, o, s)}\}) \lor \\
(\text{At}(o, l, s) \land \{\neg \exists l''. a = \text{go}(l, l'') \land l \neq l'' \land \{o = \text{robot} \lor \text{Has(robot, o, s)}\}\})))
\]

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

Deducing properties of the world: causal and diagnostic 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. For example:

\[
\begin{align*}
\text{WumpusAt}(l_1) \land \text{Adjacent}(l_1, l_2) & \implies \text{StenchAt}(l_2) \\
\text{PitAt}(l_1) \land \text{Adjacent}(l_1, l_2) & \implies \text{BreezeAt}(l_2)
\end{align*}
\]

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

Diagnostic rules: infer properties of the world from percepts. For example:

\[
\begin{align*}
\text{At}(l, s) \land \text{Breeze}(s) & \implies \text{BreezeAt}(l) \\
\text{At}(l, s) \land \text{Stench}(s) & \implies \text{StenchAt}(l)
\end{align*}
\]

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

\[
\begin{align*}
\text{robot} \neq \text{gold} \\
\text{robot} \neq \text{arrow} \\
\text{robot} \neq \text{wumpus}
\end{align*}
\]

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

\[
\begin{align*}
\text{go}(l, l') \neq \text{grab} \\
\text{go}(l, l') \neq \text{drop}(o)
\end{align*}
\]

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

\[
\begin{align*}
\text{go}(l, l') = \text{go}(l'', l''') & \iff l = l'' \land l' = l'''
\end{align*}
\]

\[
\begin{align*}
\text{drop}(o) = \text{drop}(o') & \iff o = o'
\end{align*}
\]
General axioms for situations and objects

The situations are *ordered* so

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

and situations are *distinct* so

\[ \text{result}(a, s) = \text{result}(a', s') \iff a = a' \land s = s' \]

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.

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

For example

\[
\begin{align*}
\text{At}(\text{robot}, [1, 1], s_0) \\
\text{At}(\text{wumpus}, [3, 4], s_0) \\
\text{Has}(\text{robot}, \text{arrow}, s_0)
\end{align*}
\]

and so on.

## Sequences of situations

We know that the function \( \text{result} \) 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

\[
\begin{align*}
\text{Sequence}([], s, s') &= s' = s \\
\text{Sequence}([a], s, s') &= \text{Poss}(a, s) \land s' = \text{result}(a, s) \\
\text{Sequence}(a :: as, s, s') &= \exists t. \text{Sequence}(a, s, t) \land \text{Sequence}(as, t, s')
\end{align*}
\]

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

\[
\exists a \exists s. \text{Sequence}(a, s_0, s) \land \text{Goal}(s)
\]

## Interesting reading

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

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.

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.

Frames once again support inheritance through the subclass relationship.

Example of a semantic network

Both approaches to knowledge representation are able to incorporate defaults:

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

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

has, hairlength, volume etc are slots.
long, loud, instrument etc are slot values.
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**

Progress is as follows:

1. The rule  
   \[\text{dry_mouth} \implies \text{ADD thirsty}\]
   fires adding \text{thirsty} to working memory.

2. The rule  
   \[\text{thirsty} \implies \text{ADD get_drink}\]
   fires adding \text{get_drink} to working memory.

3. The rule  
   \[\text{working} \implies \text{ADD no_work}\]
   fires adding \text{no_work} to working memory.

4. The rule  
   \[\text{get_drink AND no_work} \implies \text{ADD go_bar}\]
   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. Common conflict resolution strategies are:

- Prefer rules involving more recently added facts.
- Prefer rules that are more specific. For example  
  \[\text{patient.coughing} \implies \text{ADD lung_problem}\]
  is more general than  
  \[\text{patient.coughing AND patient_smoker} \implies \text{ADD lung_cancer}\].
- 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

\[
\text{patient\_coughing}
\]

and

\[
\text{patient\_smoker}
\]

are in working memory, and hence fire

\[
\text{patient\_coughing AND patient\_smoker} \implies \text{ADD lung\_cancer}
\]

but later infer something that causes \(\text{patient\_coughing}\) to be withdrawn from working memory.

The justification for \(\text{lung\_cancer}\) 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 \textit{variables} which can work in conjunction with \textit{pattern matching}.

For example the rule

\[
\text{coughs}(X) \text{ AND smoker}(X) \implies \text{ADD lung\_cancer}(X)
\]

contains the variable \(X\).

If the working memory contains \(\text{coughs}(\text{neddy})\) and \(\text{smoker}(\text{neddy})\) then

\[
X = \text{neddy}
\]

provides a match and \(\text{lung\_cancer}(\text{neddy})\) is added to the working memory.

---

**Backward chaining**

The second basic kind of interpreter begins with a \textit{goal} and finds a rule that would achieve it.

It then works \textit{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 \textit{goal-driven} process. If you want to \textit{test a hypothesis} or you have some idea of a likely conclusion it can be more efficient than forward chaining.

---

**Example**

To establish \(\text{get\_drink}\) we have to establish \(\text{get\_drink} \text{ and no\_work}\). These are the new goals.

Try first to establish \(\text{get\_drink}\). This can be done by establishing \(\text{thirsty}\).

\text{thirsty} can be established by establishing \(\text{dry\_mouth}\). This is in the working memory so we’re done.

Finally, we can establish \(\text{no\_work}\) by establishing \(\text{working}\). This is in the working memory so the process has finished.
Example with backtracking

If at some point more than one rule has the required conclusion then we can back-track.

Example: Prolog backtracks, and incorporates pattern matching. It orders attempts according to the order in which rules appear in the program.

Example: having added
\[ \text{up\_early} \implies \text{ADD tired} \]
and
\[ \text{tired AND lazy} \implies \text{ADD go\_bar} \]
to the rules, and \text{up\_early} to the working memory:

\[ \text{Problem solving is different to planning} \]

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.

Our aim now is to look at how an agent might construct a plan enabling it to achieve a goal.

- We look at how we might update our concept of knowledge representation and reasoning to apply more specifically to planning tasks.
- We look in detail at the partial-order planning algorithm.

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)

\[
\text{Have(pie)}
\]

and action \( \text{Buy(x)} \) has an effect \( \text{Have(x)} \) then you know that a plan including \( \text{Buy(pie)} \)

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 \( \text{Have(carKeys)} \) is a good state to be in without worrying about what happens before or after finding them.
- By making an important decision like requiring \( \text{Have(carKeys)} \) early on we may reduce branching and backtracking.
- State descriptions are not complete—\( \text{Have(carKeys)} \) 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


States: are conjunctions of ground literals. They must not include function symbols.

\[
\text{At(home)} \land \neg \text{Have(gorilla)} \\
\land \neg \text{Have(rope)} \\
\land \neg \text{Have(kit)}
\]

Goals: are conjunctions of literals where variables are assumed existentially quantified.

\[
\text{At(x)} \land \text{Sells(x, gorilla)}
\]

A planner finds a sequence of actions that when performed makes the goal true. We are no longer employing a full theorem-prover.

The 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.

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 \( S_i \rightarrow S_j \). 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.

The initial plan has:
- Two steps, called Start and Finish.
- a single ordering constraint \( \text{Start} < \text{Finish} \).
- 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 \( S' < S \) 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 \( S \).
   - \( c \in \text{Effects}(S'') \).
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:

- **Start**
  - \( \text{At(Home)} \wedge \text{Have(G)} \wedge \text{Have(R)} \wedge \text{Have(FA)} \)
- **Finish**
  - \( \text{At(Home)} \wedge \text{Sells(JS,G)} \wedge \text{Sells(HS,R)} \wedge \text{Sells(HS,FA)} \)

Thin arrows denote ordering.

Incorporating the suggested step into the plan:

- **Start**
  - \( \text{At(Home)} \wedge \text{Have(G)} \wedge \text{Have(R)} \wedge \text{Have(FA)} \)
- **Finish**
  - \( \text{At(Home)} \wedge \text{Sells(JS,G)} \wedge \text{Sells(HS,R)} \wedge \text{Sells(HS,FA)} \)

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:

- **Start**
  - \( \text{At(Home)} \wedge \text{Sells(JS,G)} \wedge \text{Sells(HS,R)} \wedge \text{Sells(HS,FA)} \)
  - \( \text{At(Home)} \wedge \text{Have(G)} \wedge \text{Have(FA)} \)
- **Finish**
  - \( \text{At(Home)} \wedge \text{Have(G)} \wedge \text{Have(FA)} \)

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 \text{Sells}(x, G) precondition of Buy(G).

![Diagram showing causal links and states]

The planner’s next obvious move is to introduce a Go step to achieve the \text{At}(JS) precondition of Buy(G).

![Diagram showing Go step and states]

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 \text{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 \text{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 \text{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).

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.

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 $A \rightarrow B$ 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.

Possible threats

What about dealing with variables?

If at any stage an effect $\neg At(x)$ appears, is it a threat to $At(JS)$?

Such an occurrence is called a possible threat and we can deal with it by introducing inequality constraints: in this case $x \neq JS$.

• 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.

Planning II

Unsurprisingly, this process can become complex.

How might we improve matters?

One way would be to introduce heuristics. We now consider:

• The way in which basic heuristics might be defined for use in planning problems.

• The construction of planning graphs and their use in obtaining more sensible heuristics.

• Planning graphs as the basis of the GraphPlan algorithm.

Another is to translate into the language of a general-purpose algorithm exploiting its own heuristics. We now consider:

• Planning using propositional logic.

• Planning using constraint satisfaction.
An example of partial-order planning

We left our example problem here:
The planner could backtrack and try to achieve the \( \text{At}(x) \) precondition using the existing \( \text{Go}(\text{JS}) \) step.

This involves a threat, but one that can be fixed using promotion.

Using heuristics in planning

We found in looking at search problems that heuristics were a helpful thing to have.

Note that now there is no simple representation of a state, and consequently it is harder to measure the distance to a goal.

Defining heuristics for planning is therefore more difficult than it was for search problems. Simple possibilities:

\[ h = \text{number of unsatisfied preconditions} \]

or

\[ h = \text{number of unsatisfied preconditions} - \text{number satisfied by the start state} \]

These can lead to underestimates or overestimates:

- Underestimates if actions can affect one another in undesirable ways.
- Overestimates if actions achieve many preconditions.

Planning Graphs apply when it is possible to work entirely using propositional representations of plans. Luckily, STRIPS can always be propositionalized...

For example: the triumphant return of the gorilla-purchasing roof-climbers...
Planning graphs

A planning graph is constructed in levels:

- **Level 0** corresponds to the start state.
- At each level we keep approximate track of all things that could be true at the corresponding time.
- At each level we keep approximate track of what actions could be applicable at the corresponding time.

The approximation is due to the fact that not all conflicts between actions are tracked. So:

- The graph can underestimate how long it might take for a particular proposition to appear, and therefore . . .
- . . . a heuristic can be extracted.

Planning graphs: a simple example

Our intrepid student adventurers will of course need to inflate their gorilla before attaching it to a distinguished roof. It has to be purchased before it can be inflated.

**Start state:** Empty.

We assume that anything not mentioned in a state is false. So the state is actually

\[-\text{Have}(\text{Gorilla}) \text{ and } \neg\text{Inflated}(\text{Gorilla})\]

**Actions:**

- \(-\text{Have}(\text{Gorilla})\)
- \(\text{Have}(\text{Gorilla})\)
- \(\text{Buy}(\text{Gorilla})\)
- \(\text{Inflate}(\text{Gorilla})\)

**Goal:** \(\text{Have}(\text{Gorilla}) \text{ and } \text{Inflated}(\text{Gorilla})\).

Mutex links

We also record, using mutual exclusion (mutex) links which pairs of actions could not occur together.

**Mutex links 1:** Effects are inconsistent.

The effect of one action negates the effect of another.
**Mutex links**

*Mutex links 2*: The actions interfere.

\[
\neg I(G) \quad I(G)
\]

The effect of an action negates the precondition of another.

---

**Mutex links**

*Mutex links 3*: Competing for preconditions.

\[
\neg H(G) \quad H(G)
\]

The precondition for an action is mutually exclusive with the precondition for another. (See next slide!)

---

**Mutex links**

A state level \( S_i \) contains all propositions that could be true, given the possible preceding actions.

We also use mutex links to record pairs that can not be true simultaneously:

*Possibility 1*: pair consists of a proposition and its negation.

\[
\neg H(G) \quad H(G)
\]

---

**Mutex links**

*Possibility 2*: all pairs of actions that could achieve the pair of propositions are mutex.

\[
A_1 \quad S_2
\]

The construction of a planning graph is continued until two identical levels are obtained.
Obtaining heuristics from a planning graph

To estimate the cost of reaching a single proposition:

- Any proposition not appearing in the final level has infinite cost and can never be reached.
- The level cost of a proposition is the level at which it first appears but this may be inaccurate as several actions can apply at each level and this cost does not count the number of actions. (It is however admissible.)
- A serial planning graph includes mutex links between all pairs of actions except persistence actions.

*Level cost in serial planning graphs* can be quite a good measurement.

Obtaining heuristics from a planning graph

How about estimating the cost to achieve a collection of propositions?

- **Max-level**: use the maximum level in the graph of any proposition in the set. Admissible but can be inaccurate.
- **Level-sum**: use the sum of the levels of the propositions. Inadmissible but sometimes quite accurate if goals tend to be decomposable.
- **Set-level**: use the level at which all propositions appear with none being mutex. Can be accurate if goals tend not to be decomposable.

Other points about planning graphs

A planning graph guarantees that:

1. If a proposition appears at some level, there may be a way of achieving it.
2. If a proposition does not appear, it can not be achieved.

The first point here is a loose guarantee because only *pairs* of items are linked by mutex links.

Looking at larger collections can strengthen the guarantee, but in practice the gains are outweighed by the increased computation.
The GraphPlan algorithm goes beyond using the planning graph as a source of heuristics.

Start at level 0;
while (true) {
  if (all goal propositions appear in the current level
      AND no pair has a mutex link) {
    attempt to extract a plan;
    if (a solution is obtained)
      return the solution;
    else if (graph indicates there is no solution)
      return fail;
    expand the graph to the next level;
  }
  else
    expand the graph to the next level;
}

We extract a plan directly from the planning graph. Termination can be proved but will not be covered here.

Extracting a plan from the graph

Extraction of a plan can be formalised as a search problem.

States contain a level, and a collection of unsatisfied goal propositions.

Start state: the current final level of the graph, along with the relevant goal propositions.

Goal: a state at level $S_0$ containing the initial propositions.

Actions: For a state $S$ with level $S_i$, a valid action is to select any set $X$ of actions in $A_{i-1}$ such that:

1. no pair has a mutex link;
2. no pair of their preconditions has a mutex link;
3. the effects of the actions in $X$ achieve the propositions in $S$.

The effect of such an action is a state having level $S_{i-1}$, and containing the preconditions for the actions in $X$.

Each action has a cost of 1.
Heuristics for plan extraction

We can of course also apply heuristics to this part of the process. For example, when dealing with a set of propositions:

- Choose the proposition having maximum level cost first.
- For that proposition, attempt to achieve it using the action for which the maximum/sum level cost of its preconditions is minimum.

Planning III: planning using propositional logic

We’ve seen that plans might be extracted from a knowledge base via theorem proving, using first order logic (FOL) and situation calculus. BUT: this might be computationally infeasible for realistic problems. Sophisticated techniques are available for testing satisfiability in propositional logic, and these have also been applied to planning.

The basic idea is to attempt to find a model of a sentence having the form:

- description of start state
- ∧ descriptions of the possible actions
- ∧ description of goal

We attempt to construct this sentence such that:

- If $M$ is a model of the sentence then $M$ assigns $\top$ to a proposition if and only if it is in the plan.
- Any assignment denoting an incorrect plan will not be a model as the goal description will not be $\top$.
- The sentence is unsatisfiable if no plan exists.

Propositional logic for planning

Start state:

$$ S = \text{At}^0(a, \text{spire}) \land \text{At}^0(b, \text{ground})$$

$$ \land \neg \text{At}^0(a, \text{ground}) \land \neg \text{At}^0(b, \text{spire}) $$

Goal:

$$ G = \text{At}^i(a, \text{ground}) \land \text{At}^i(b, \text{spire}) $$

$$ \land \neg \text{At}^i(a, \text{spire}) \land \neg \text{At}^i(b, \text{ground}) $$

Actions:

- can be introduced using the equivalent of successor-state axioms

$$ \text{At}^1(a, \text{ground}) \leftrightarrow $$

$$ (\text{At}^0(a, \text{ground}) \land \neg \text{Move}^0(a, \text{ground, spire})) $$

$$ \lor (\text{At}^0(a, \text{spire}) \land \text{Move}^0(a, \text{spire, ground})) $$

Denote by $A$ the collection of all such axioms.
Propositional logic for planning

We will now find that $S \land A \land G$ has a model in which $Move^0(a, \text{spire}, \text{ground})$ and $Move^0(b, \text{ground}, \text{spire})$ are $\top$ while all remaining actions are $\bot$.

In more realistic planning problems we will clearly not know in advance at what time the goal might expect to be achieved.

We therefore:

• Loop through possible final times $T$.
• Generate a goal for time $T$ and actions up to time $T$.
• Try to find a model and extract a plan.
• Until a plan is obtained or we hit some maximum time.

Unfortunately there is a problem—we may, if considerable care is not applied, also be able to obtain less sensible plans.

In the current example

\[
\begin{align*}
Move^0(b, \text{ground}, \text{spire}) &= \top \\
Move^0(a, \text{spire}, \text{ground}) &= \top \\
Move^0(a, \text{ground}, \text{spire}) &= \top
\end{align*}
\]

is a model, because the successor-state axiom (2) does not in fact preclude the application of $Move^0(a, \text{ground}, \text{spire})$.

We need a **precondition axiom**

\[
Move^i(a, \text{ground}, \text{spire}) \rightarrow At^i(a, \text{ground})
\]

and so on.

Life becomes more complicated still if a third location is added: hospital.

\[
Move^0(a, \text{spire}, \text{ground}) \land Move^0(a, \text{spire}, \text{hospital})
\]

is perfectly valid and so we need to specify that he can’t move to two places simultaneously

\[
\neg (Move^i(a, \text{spire}, \text{ground}) \land Move^i(a, \text{spire}, \text{hospital}))
\]

\[
\neg (Move^i(a, \text{ground}, \text{spire}) \land Move^i(a, \text{ground}, \text{hospital}))
\]

and so on.

These are **action-exclusion** axioms.

Unfortunately they will tend to produce **totally-ordered** rather than **partially-ordered** plans.

Alternatively:

1. Prevent actions occurring together if one negates the effect or precondition of the other.
2. Or, specify that something can’t be in two places simultaneously

\[
\forall x, i, 11, 12 \; 11 \neq 12 \rightarrow \neg (At^i(x, 11) \land At^i(x, 12))
\]

This is an example of a **state constraint**.

Clearly this process can become very complex, but there are techniques to help deal with this.
Review of constraint satisfaction problems (CSPs)

Recall that in a CSP 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.

The state-variable representation

Another planning language: the state-variable representation.

Things of interest such as people, places, objects etc are divided into domains:

- \( D_1 = \{ \text{climber1, climber2} \} \)
- \( D_2 = \{ \text{home, jokeShop, hardwareStore, pavement, spire, hospital} \} \)
- \( D_3 = \{ \text{rope, inflatableGorilla} \} \)

Part of the specification of a planning problem involves stating which domain a particular item is in. For example

\[ D_1(\text{climber1}) \]

and so on.

Relations and functions have arguments chosen from unions of these domains.

\[ \text{above} \subseteq D_1^{\text{above}} \times D_2^{\text{above}} \]

is a relation. The \( D_i^{\text{above}} \) are unions of one or more \( D_i \).

Note: \( D \) is used for domains in the state-variable representation. \( D \) is used for domains in CSPs.

The state-variable representation

The relation \( \text{above} \) is in fact a rigid relation (RR), as it is unchanging: it does not depend upon state. (Remember fluents in situation calculus?)

Similarly, we have functions

\[ \text{at}(x_1, s) : D_1^{\text{at}} \times S \rightarrow D^{\text{at}}. \]

Here, \( \text{at}(x, s) \) is a state-variable. The domain \( D_1^{\text{at}} \) and range \( D^{\text{at}} \) are unions of one or more \( D \). In general these can have multiple parameters

\[ \text{at}(x_1, \ldots, x_n, s) : D_1^{\text{at}} \times \cdots \times D_n^{\text{at}} \times S \rightarrow D^{\text{at}}. \]

A state-variable denotes assertions such as

\[ \text{at}(\text{gorilla}, s) = \text{jokeShop} \]

where \( s \) denotes a state and the set \( S \) of all states will be defined later.

The state variable allows things such as locations to change—again, much like fluents in the situation calculus.

Variables appearing in relations and functions are considered to be typed.
The state-variable representation

Actions as usual, have a name, a set of preconditions and a set of effects.

- Names are unique, and followed by a list of variables involved in the action.
- Preconditions are expressions involving state variables and relations.
- Effects are assignments to state variables.

For example:

<table>
<thead>
<tr>
<th>buy(x, y, l)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preconditions</td>
</tr>
<tr>
<td>at(x, s) = l</td>
</tr>
<tr>
<td>sells(l, y)</td>
</tr>
<tr>
<td>has(y, s) = l</td>
</tr>
<tr>
<td>Effects</td>
</tr>
<tr>
<td>has(y, s) = x</td>
</tr>
</tbody>
</table>

Goals are sets of expressions involving state variables.

For example:

<table>
<thead>
<tr>
<th>Goal:</th>
</tr>
</thead>
<tbody>
<tr>
<td>at(climber, s) = home</td>
</tr>
<tr>
<td>has(rope, s) = climber</td>
</tr>
<tr>
<td>at(gorilla, s) = spire</td>
</tr>
</tbody>
</table>

From now on we will generally suppress the state s when writing state variables.

We can essentially regard a state as just a statement of what values the state variables take at a given time.

Formally:

- For each state variable sv we can consider all ground instances such as—sv(climber, rope)—with arguments that are consistent with the rigid relations.

Define X to be the set of all such ground instances.

- A state s is then just a set
  
  \[ s = \{ (v = c) \mid v \in X \} \]

  where c is in the range of v.

This allows us to define the effect of an action:

A planning problem also needs a start state s0, which can be defined in this way.

Considering all the ground actions consistent with the rigid relations:

- An action is applicable in s if all expressions \( v = c \) appearing in the set of preconditions also appear in s.

Finally, there is a function \( \gamma \) that maps a state and an action to a new state

\[ \gamma(s, a) = s' \]

Specifically, we have

\[ \gamma(s, a) = \{ (v = c) \mid v \in X \} \]

where either \( c \) is specified in an effect of a, or otherwise \( v = c \) is a member of s.

Note: the definition of \( \gamma \) implicitly solves the frame problem.
The state-variable representation

A solution to a planning problem is a sequence \((a_0, a_1, \ldots, a_n)\) of actions such that...

- \(a_0\) is applicable in \(s_0\) and for each \(i\), \(a_i\) is applicable in \(s_i = \gamma(s_{i-1}, a_{i-1})\).
- For each goal \(g\) we have \(g \in \gamma(s_n, a_n)\).

What we need now is a method for transforming a problem described in this language into a CSP.

We’ll once again do this for a fixed upper limit \(T\) on the number of steps in the plan.

Converting to a CSP

**Step 1:** encode actions as CSP variables.
For each time step \(t\) where \(0 \leq t \leq T - 1\), the CSP has a variable \(a_t\) with domain \(D^{a_t} = \{a \mid a\text{ is the ground instance of an action}\} \cup \{\text{none}\}\).

**Example:** at some point in searching for a plan we might attempt to find the solution to the corresponding CSP involving 
\(a_5 = \text{attach}(\text{inflatableGorilla}, \text{spire})\)

**WARNING:** be careful in what follows to distinguish between state variables, actions etc in the planning problem and variables in the CSP.

**Step 2:** encode ground state variables as CSP variables, with a complete copy of all the state variables for each time step.
So, for each \(t\) where \(0 \leq t \leq T\) we have a CSP variable \(s^t\) with domain \(D^{s^t} = \text{range of the state variable.}\)

**Example:** at some point in searching for a plan we might attempt to find the solution to the corresponding CSP involving 
\(\text{location}^9(\text{climber1}) = \text{hospital}\).

**Step 3:** encode the preconditions for actions in the planning problem as constraints in the CSP problem.
For each time step \(t\) and for each ground action \(a(c_1, \ldots, c_n)\) with arguments consistent with the rigid relations in its preconditions:
For a precondition of the form \(s^t_i = v\) include constraint pairs
\((a_t = a(c_1, \ldots, c_n), s^t_i = v)\)

**Example:** consider the action \(\text{buy}(x, y, l)\) introduced above, and having the preconditions \(\text{at}(x) = l, \text{sells}(l, y)\) and \(\text{has}(y) = l\).
Assume \(\text{sells}(y, l)\) is only true for 
\(l = \text{jokeShop}\) and 
\(y = \text{inflatableGorilla}\)
so we only consider these values for \(l\) and \(y\). Then for each time step \(t\) we have the constraints...
Step 4: encode the effects of actions in the planning problem as constraints in the CSP problem.

For each time step $t$ and for each ground action $a(c_1, \ldots, c_n)$ with arguments consistent with the rigid relations in its preconditions:

For an effect of the form $sv_i = v$ include constraint pairs

$\langle \text{action}_t = a(c_1, \ldots, c_n),
sv_{t+1}^i = v \rangle$

Example: continuing with the previous example, we will include constraints

$\langle \text{action}_t = \text{buy}(\text{climber}_1, \text{inflatableGorilla}, \text{jokeShop}),
\text{has}_{t+1}(\text{inflatableGorilla}) = \text{climber}_1 \rangle$

$\langle \text{action}_t = \text{buy}(\text{climber}_2, \text{inflatableGorilla}, \text{jokeShop}),
\text{has}_{t+1}(\text{inflatableGorilla}) = \text{climber}_2 \rangle$

and so on...

Step 5: encode the frame axioms as constraints in the CSP problem.

An action must not change things not appearing in its effects. So:

For:

1. Each time step $t$.
2. Each ground action $a(c_1, \ldots, c_n)$ with arguments consistent with the rigid relations in its preconditions.
3. Each $sv_i$ that does not appear in the effects of $a$, and each $v \in D^{sv_i}$

include in the CSP the ternary constraint

$\langle \text{action}_t = a(c_1, \ldots, c_n),
sv_{t}^i = v, 
sv_{t+1}^i = v \rangle$.

Finally, having encoded a planning problem into a CSP, we solve the CSP.

The scheme has the following property:

A solution to the planning problem with at most $T$ steps exists if and only if there is a solution to the corresponding CSP.

Assume the CSP has a solution.

Then we can extract a plan simply by looking at the values assigned to the $\text{action}_t$ variables in the solution of the CSP.

It is also the case that:

There is a solution to the planning problem with at most $T$ steps if and only if there is a solution to the corresponding CSP from which the solution can be extracted in this way.

For a proof see:

Automated Planning: Theory and Practice
Artificial Intelligence I

Machine learning using neural networks

Reading: AIMA, chapter 20.

Supervised learning with neural networks

We now consider how an agent might learn to solve a general problem by seeing examples:

- I present an outline of supervised learning.
- I introduce the classical perceptron.
- I introduce multilayer perceptrons and the backpropagation algorithm for training them.

To begin, 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:

![Diagram of a machine inputting measurements and outputting a diagnosis]

Measurements taken from the patient: heart rate, blood pressure, presence of green spots etc.

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,

$$x^T = (x_1 \ x_2 \ \cdots \ x_n)$$

where,

- $x_1 = $ heart rate
- $x_2 = $ blood pressure
- $x_3 = 1$ if the patient has green spots
- $0$ otherwise

and so on

(\textit{Note}: it’s a common convention that vectors are \textit{column vectors} by default. This is why the above is written as a transpose.)

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

$$f(x_1, \ldots, x_n) = \sum_{i=1}^{n} a_i x_i^2$$

where the $a_i$ are constants. Compute $\partial f / \partial x_j$ where $1 \leq j \leq n$?

\textit{Answer}: As only one term in the sum depends on $x_j$, so all the other terms differentiate to give 0 and

$$\frac{\partial f}{\partial x_j} = 2a_j x_j$$

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 $1 \leq j \leq m$?

\textit{Answer}: this is just the chain rule for partial differentiation

$$\frac{\partial f}{\partial y_j} = \sum_{i=1}^{n} \frac{\partial f}{\partial g_i} \frac{\partial g_i}{\partial y_j}$$
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 \( x_i \in \{X_1, \ldots, X_p\} \).

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 \( i \)th patient history gives us an instance \( x_i \).
- This can be paired with a single bit—0 or 1—denoting whether or not the \( i \)th 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 \( s = ((x_1, 0), (x_2, 1), \ldots, (x_m, 0)) \).

In supervised machine learning we aim to design a learning algorithm which takes \( s \) 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**.

In fact, a hypothesis is just a function that maps instances to labels.

As \( h \) is a function it assigns a label to *any* \( x \) 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 and regression

In classification we’re assigning \( x \) to one of a set \( \{\omega_1, \ldots, \omega_c\} \) of \( c \) classes. For example, if \( x \) contains measurements taken from a patient then there might be three classes:

\[
\begin{align*}
\omega_1 &= \text{patient has disease} \\
\omega_2 &= \text{patient doesn’t have disease} \\
\omega_3 &= \text{don’t ask me buddy, I’m just a computer!}
\end{align*}
\]

The binary case above also fits into this framework, and we’ll often specialise to the case of two classes, denoted \( C_1 \) and \( C_2 \).

In regression we’re assigning \( x \) to a real number \( h(x) \in \mathbb{R} \). For example, if \( x \) contains measurements taken regarding today’s weather then we might have

\[
h(x) = \text{estimate of amount of rainfall expected tomorrow}
\]

For the two-class classification problem we will also refer to a situation somewhat between the two, where

\[
h(x) = \Pr(x \text{ is in } C_1)
\]

and so we would typically assign \( x \) to class \( C_1 \) if \( h(x) > 1/2 \).

Neural networks

There is generally a set \( \mathcal{H} \) of hypotheses from which \( L \) is allowed to select \( h \)

\[
L(s) = h \in \mathcal{H}
\]

\( \mathcal{H} \) 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

\[
w^T = (w_1, w_2, \ldots, w_W)
\]

of weights which in turn specify \( h \)

\[
h(x) = f(w; x)
\]

where \( w = L(s) \).

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.

Types of learning

The form of machine learning described is called supervised learning. The literature also discusses unsupervised learning, learning using membership queries and equivalence queries, and reinforcement learning. (More about some of this next year...)

For now, supervised learning has multiple applications:

- 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.
- 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 process is in fact very similar to *curve fitting*. Think of the process as follows:

- Nature picks an \( h' \in H \) but doesn’t reveal it to us.
- Nature then shows us a training sequence \( s \) where each \( x_i \) is labelled as \( h'(x_i) + \epsilon_i \) where \( \epsilon_i \) is noise of some kind.

Our job is to try to infer what \( h' \) is on the basis of \( s \) only. *Example*: if \( H \) is the set of all polynomials of degree 3 then nature might pick \( h'(x) = \frac{1}{2}x^3 - \frac{1}{2}x^2 + 2x - \frac{1}{2} \). The line is dashed to emphasise the fact that we don’t get to see it.

We’ll use a *learning algorithm* \( L \) that operates in a reasonable-looking way: it picks an \( h \in H \) minimising the following quantity,

\[
E = \sum_{i=1}^{m}(h(x_i) - y_i)^2
\]

In other words

\[
h = L(s) = \arg\min_{h \in H} \sum_{i=1}^{m}(h(x_i) - y_i)^2
\]

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.

We can now use \( h' \) to obtain a training sequence \( s \) in the manner suggested. Here we have,

\[
s = ((x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m))
\]

where each \( x_i \) and \( y_i \) is a real number.

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 \( x' \) and asked to guess the value \( h'(x') \) then guessing \( h(x') \) might be expected to do quite well.
Curve fitting

Problem: we don’t know what H nature is using. What if the one we choose doesn’t match? We can make our H ‘bigger’ by defining it as

\[ H = \{ h : h \text{ is a polynomial of degree at most } 5 \} \]

If we use the same learning algorithm then we get:

In effect, we have made our H too ‘small’. It does not in fact contain any hypothesis similar to \( h' \).

BEWARE!!! This is known as overfitting.

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, so

\[ x^T = (x_1 \ x_2 \ \cdots \ x_n) \]

The simplest form of hypothesis used is the linear discriminant, also known as the perceptron. Here

\[ h(\mathbf{w}; \mathbf{x}) = \sigma \left( w_0 + \sum_{i=1}^{n} w_i x_i \right) = \sigma \left( w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n \right). \]

So: we have a linear function modified by the activation function \( \sigma \).

The perceptron’s influence continues to be felt in the recent and ongoing development of support vector machines.
The perceptron activation function

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
   \[ \Pr(\text{x is in } C_1) = \sigma(z) = \frac{1}{1 + \exp(-z)}. \]

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.

---

**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
\[
E(w) = \sum_{i=1}^{m} (y_i - h(w; x_i))^2
\]

Modifying our notation slightly so that
\[
\mathbf{x}^T = (1 \ x_1 \ x_2 \ \cdots \ x_n) \\
\mathbf{w}^T = (w_0 \ w_1 \ w_2 \ \cdots \ w_n)
\]

lets us write
\[
E(w) = \sum_{i=1}^{m} (y_i - \mathbf{w}^T \mathbf{x}_i)^2
\]

We want to minimise \( E(w) \).

---

**The sigmoid/logistic function**

\[ \sigma(z) = \frac{1}{1 + \exp(-z)} \]
Gradient descent

With

\[ E(w) = \sum_{i=1}^{m} (y_i - w^T x_i)^2 \]

we have

\[
\frac{\partial E(w)}{\partial w_j} = \frac{\partial}{\partial w_j} \left( \sum_{i=1}^{m} (y_i - w^T x_i)^2 \right) \\
= \sum_{i=1}^{m} \left( 2(y_i - w^T x_i) \frac{\partial}{\partial w_j} (w^T x_i) \right) \\
= -2 \sum_{i=1}^{m} x_j^{(i)} (y_i - w^T x_i)
\]

where \( x_j^{(i)} \) is the \( j \)-th element of \( x_i \).

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:

\[ a_j z_j \sigma(a_j) \]

Weights \( w_j \) connect nodes together, and \( a_j \) is the weighted sum or activation for node \( j \). \( \sigma \) is the activation function and the output is \( z_j = \sigma(a_j) \).

Reminder: we’ll continue to use the notation

\[
\begin{align*}
\mathbf{z}^T &= (1 \ z_1 \ z_2 \ \cdots \ z_n) \\
\mathbf{w}^T &= (w_0 \ w_1 \ w_2 \ \cdots \ w_n)
\end{align*}
\]

So that

\[
\sum_{j=0}^{n} w_j z_j = w_0 + \sum_{j=1}^{n} w_j z_j = \mathbf{w}^T \mathbf{z}
\]
The multilayer perceptron

In the general case we have a completely unrestricted feedforward structure:

![Diagram of a multilayer perceptron]

Each node is a perceptron. No specific layering is assumed.
w_{i \rightarrow j} connects node i to node j. w_0 for node j is denoted w_0 \rightarrow j.

Backpropagation

As usual we have:

- Instances \( x^T = (x_1, \ldots, x_n) \).
- A training sequence \( s = ((x_1, y_1), \ldots, (x_m, y_m)) \).

We also define a measure of training error

\[ E(w) = \text{measure of the error of the network on } s \]

where \( w \) is the vector of all the weights in the network.

Our aim is to find a set of weights that minimises \( E(w) \) using gradient descent.

Backpropagation: the general case

The central task is therefore to calculate

\[ \frac{\partial E(w)}{\partial w} \]

To do that we need to calculate the individual quantities

\[ \frac{\partial E(w)}{\partial w_{i \rightarrow j}} \]

for every weight \( w_{i \rightarrow j} \) in the network.

Often \( E(w) \) is the sum of separate components, one for each example in \( s \)

\[ E(w) = \sum_{p=1}^{m} E_p(w) \]

in which case

\[ \frac{\partial E(w)}{\partial w} = \sum_{p=1}^{m} \frac{\partial E_p(w)}{\partial w} \]

We can therefore consider examples individually.
So we now need to calculate the values for $\delta_j$. When $j$ is the output node—that is, the one producing the output $y = h(w, x_p)$ of the network—this is easy as $z_j = y$ and

$$
\delta_j = \frac{\partial E_p(w)}{\partial a_j}
= \frac{\partial E_p(w)}{\partial y} \frac{\partial y}{\partial a_j}
= \frac{\partial E_p(w)}{\partial y} \sigma'(a_j)
$$

using the fact that $y = \sigma(a_j)$. 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

$$
E_p(w) = (y - y_p)^2
$$

we have

$$
\frac{\partial E_p(w)}{\partial y} = 2(y - y_p)
= 2(h(w; x_p) - y_p).
$$

We're interested in

$$
\delta_j = \frac{\partial E_p(w)}{\partial a_j}
$$

Because we know how to compute $\delta_j$ for the output node we can work backwards computing further $\delta$ values. We will always know all the values $\delta_k$ for nodes ahead of where we are. Hence the term backpropagation.
Backpropagation: the general case

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

$$E(w) = \sum_{p=1}^{m} (y_p - h(w; x_p))^2$$

$$E_p(w) = (y_p - h(w; x_p))^2$$
For the output: the equation is
\[
\frac{\partial E_p(w)}{\partial w_i \rightarrow \text{output}} = z_i \delta_{\text{output}} = z_i \sigma'(a_{\text{output}}) \frac{\partial E_p(w)}{\partial y}
\]
where \( y = h(w; x_p) \). So as \( \frac{\partial E_p(w)}{\partial y} = \frac{\partial}{\partial y} \left( (y_p - y)^2 \right) \)
\[
= 2(y - y_p)
\]
and \( \sigma'(a) = 1 \) so
\[
\delta_{\text{output}} = 2 \left[ h(w; x_p) - y_p \right]
\]
and
\[
\frac{\partial E_p(w)}{\partial w_i \rightarrow \text{output}} = 2z_i \left( h(w; x_p) - y_p \right)
\]

Putting it all together

We can then use the derivatives in one of two basic ways:

**Batch**: (as described previously)
\[
\frac{\partial E(w)}{\partial w} = \sum_{p=1}^{m} \frac{\partial E_p(w)}{\partial w}
\]
then
\[
w_{t+1} = w_t - \eta \frac{\partial E(w)}{\partial w}_{w_t}
\]

**Sequential**: using just one pattern at once
\[
w_{t+1} = w_t - \eta \frac{\partial E_p(w)}{\partial w}_{w_t}
\]
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.