Introduction: what are our aims?

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

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.
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…

Al 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 Al

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

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

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

Perhaps I’m being too hard on them; there was some good groundwork: Socrates wanted an algorithm for “piety”, leading to Syllogisms. Ramon Lull’s concept wheels and other attempts at mechanical calculators. Rene Descartes’ Dualism and the idea of mind as a physical system. Wilhelm Leibnitz’s opposing position of Materialism. (The intermediate position: mind is physical but unknowable.) The origin of knowledge: Francis Bacon’s Empiricism, John Locke: “Nothing 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’!)

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.

There are equally many areas in which we currently can’t do things very well:

“Sleep that knits up the ragged sleeve of care”

is a line from Shakespeare’s Macbeth.

On the other hand…

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

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.

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, this course 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.)
Introductory reading that isn’t nonsense

- Francis Crick, “The recent excitement about neural networks”, Nature (1989) is still entirely relevant:
  www.nature.com/nature/journal/v337/n6203/abs/337129a0.html
- The Loebner Prize in Artificial Intelligence:
  aisb.org.uk/aisb-events/
  provides a good illustration of how far we are from passing the Turing test.
  http://web.media.mit.edu/~minsky/
- Go:
  www.nature.com/nature/journal/v529/n7587/full/nature16961.html
- The Cyc project:
  www.cyc.com
- AI at Nasa Ames:
  www.nasa.gov/centers/ames/research/areas-of-ames-ingenuity-autonomy-and-robotics

Text book

The course is based on the relevant parts of:


and an alternative source is:


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 \( \partial f / \partial x_j \) where \( 1 \leq j \leq n \)?

2. Let \( f(x_1, \ldots, x_n) \) be a function. Now assume \( x_i = g_i(y_1, \ldots, y_m) \) for each \( x_i \) and some collection of functions \( g_i \). Assuming all requirements for differentiability and so on are met, can you write down an expression for \( \partial f / \partial y_j \) where \( 1 \leq j \leq m \)?

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

And finally…

There are some important points to be made regarding computational complexity. First, you might well hear the term AI-complete being used a lot. What does it mean?

AI-complete: only solvable if you can solve AI in its entirety. For example: high-quality automatic translation from one language to another. To produce a genuinely good translation of Moby Dick from English to Cantonese is likely to be AI-complete.

More practically, you will often hear me make the claim that everything that’s at all interesting in AI is at least NP-complete. There are two ways to interpret this:

1. The wrong way: “It’s all a waste of time.” OK, so it’s a partly understandable interpretation. BUT the fact that Boolean satisfiability is intractable does not mean we can’t solve large instances in practice…

2. The right way: “It’s an opportunity to design nice approximation algorithms.” In reality, the algorithms that are good in practice are ones that try to often find a good but not necessarily optimal solution, in a reasonable amount of time and memory.

Agents

There are many different definitions for the term agent within AI. Allow me to introduce EVIL ROBOT.

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

\[ \text{MUST ENSLAVE EARTH!!! Dr Holden will be our GLORIOUS LEADER!!!} \]

\[ \text{Environment} \]

\[ \text{Sensor} \rightarrow \text{Act} \]

---

*In essence, a comment on a course assessment a couple of years back to the effect of: “Why do you teach us this stuff if it’s all futile?”*
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:
1. How can we judge an agent’s performance?
2. How can an agent’s environment affect its design?
3. 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
Item 1: How can we judge an agent’s performance?
• Any measure of performance is likely to be problem-specific.
  – Even a simple email filter is an agent—it can sense and act. Here the performance measure is straightforward.
  – For a self-driving car, it is more complicated!
• We’re usually interested in expected, long-term performance.
  – Expected performance because usually agents are not omniscient—they don’t infallibly know the outcome of their actions.
    (It is rational for you to enter this lecture theatre even if the roof falls in today. An agent capable of detecting and protecting itself from a falling roof might be more successful than you, but not more rational.
  – Long-term performance because it tends to lead to better approximations to what we’d consider rational behaviour.

Environments
Item 2: How can an agent’s environment affect its design?
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
Item 3: Are there sensible ways in which to think about the structure of an agent?
A basic agent can be thought of as working according to 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:
• A percept might arrive while an action is being chosen.
• The world may change while an action is being chosen.
• Actions may affect the world in unexpected ways.
• We might have multiple goals, which interact with each other.
• And so on...
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.
- We might need to trade-off conflicting goals, for example speed and safety.
- An agent may have several goals, but not be certain of achieving any of them. Can it trade-off the likelihood of reaching a goal against the desirability of getting there?

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

Maximising expected utility over time forms a fundamental model for the design of agents.

Unfortunately, there is insufficient time in this course to properly explore agents based on utility.

Learning agents

It seems reasonable that an agent should learn from experience:

What might this entail?
Learning agents

Learning mainly requires two additions:

1. The learner needs some form of feedback on the agent’s performance. This can come in several different forms.
2. The learner needs a means of generating new behaviour in order to find out about the world.

The second point leads to an important trade-off:

1. Should the agent spend time exploiting what it’s learned so far, if it’s achieving a level of success, or...
2. …should the agent try new things, exploring the environment on the basis that it might learn something really useful even if it performs worse in the short term?

Artificial Intelligence

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 \( s_0 \) from a set \( S \) of possible states.
  This models the agent’s situation before anything else happens.
- A set of actions, denoted \( A \).
  These are modelled by specifying what state will result on performing any available action in any state.
  We can model this using a function \( \text{action} : A \times S \rightarrow S \): if the agent is in state \( s \) and performs action \( a \) then its new state is \( \text{action}(a, s) \).
- A goal test: we can tell whether or not the state we’re in corresponds to a goal.
  We can model this using a function \( \text{goal} : S \rightarrow \{ \text{true}, \text{false} \} \).

Reading: AIMA chapters 3 and 4.
Problem solving by search

You have already seen problems like this...

- *Algorithms*: talks about searching in graphs. It also covers depth-first and breadth-first search, from a more formal perspective.

This is all important stuff, but there’s a problem: *none of these methods works in practice for typical AI problems!*

Essentially, the problem is that they are too naïve in the way that they choose a state to explore at each step.

I’m going to assume that you know this material and move on...

A simple example: the 8-puzzle.

Here we have:

- **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.

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.
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.
- In a graph we may also encounter cycles.

The sliding blocks puzzle for example suffers this way. So: we start by assuming the search is taking place on a tree.

The basic tree-search algorithm

We need to define one more function: expand takes any state s. It applies all actions that can be applied in s and returns the set of the resulting states:

$$\text{expand}(s) = \{ s' | s' = \text{action}(a, s) \text{ where } a \text{ is an action possible in } s \}.$$ 

The algorithm for searching in a tree then looks like this:

```
1 fringe = [s_0];
2 while true do
  3   if fringe empty() then
  4     return NONE;
  5   s = fringe remove();
  6   if goal(s) then
  7     return SOME(s);
  8     fringe addAll(expand(s));
```

The search strategy is set by using a priority queue to implement the fringe. The definition of priority then sets the way in which the tree is searched.

The entire tree to depth d can have $$\sum_{i=0}^{d} b^i = \frac{b^{d+1} - 1}{b-1}$$ states.

At each iteration, one node from the fringe is expanded. In general, if the branching factor is b then the layer at depth d can have $$b^d$$ states.
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.

And we are 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 still \( O(b^d) \) (if you know you only have to go to depth \( d \)).

The search is no longer optimal, and may not be complete.

Iterative-deepening combines the two, but we can do better.

Uniform-cost search

How might we change tree search to try to get to an optimal solution while limiting the time and memory needed?

The key point: so far we only distinguish goal states from non-goal states!

None of the searches you’ve seen so far tries to prioritize the exploration of good states!!

What is a good state?

- Well, at any point in the search we can work out the path cost \( p(s) \) of whatever state \( s \) we’ve got to.
- How about using the \( p(s) \) as the priority for the priority queue?

This is called Uniform-Cost Search.

In practice it doesn’t work very well: we need something more subtle.

But it does suggest the idea of an evaluation function: a function that attempts to measure the desirability of each state.
Heuristics

Why is path cost not a good evaluation function? It is not directed in any sense toward the goal.

A heuristic function, usually denoted $h(s)$, is one that estimates the cost of the best path from any state $s$ to a goal. If $s$ is a goal then $h(s) = 0$.

This is a problem-dependent measure. We are required either to design it using our knowledge of the problem, or by some other means.

The last point is critical: AI is a long way from being independent of human ingenuity.

Example: route-finding

Example: for route-finding a reasonable heuristic function is $h(s) =$ straight line distance from $s$ to the nearest goal.

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

Can we use $h(s)$ in choosing a state to explore? If it’s really good it can work well, but we can still do better!

$A^*$ search

$A^*$ search is the classical AI-oriented search algorithm.

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

- Using $p(s)$ to know how far we’ve come.
- Using $h(s)$ to estimate how far we have to go.

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

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

By using this as a priority for exploring states we get a search algorithm that is optimal and complete under simple conditions, and can be vastly superior to the more naive approaches.

Definition: an admissible heuristic $h(s)$ is one that never overestimates the cost of the best path from $s$ to a goal.

So if $h'(s)$ denotes the actual distance from $s$ to the goal we have $\forall s, h(s) \leq h'(s)$.

If $h(s)$ is admissible then tree-search $A^*$ is optimal.
A* tree-search is optimal for admissible $h(s)$

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 $h(Goal_{opt}) = 0$).

As some point $Goal_{opt}$ is in the fringe. Can it be selected before $s$?

Let $Goal_2$ be a suboptimal goal state with $f(Goal_2) = p(Goal_2) = f_2 > f_{opt}$. We need to demonstrate that the search can never select $Goal_2$.

Graph search

To search in graphs we need a way to make sure no state gets visited more than once.

We need to add a closed list, and add a state to it when the state is first seen:

```
1 closed = [];
2 fringe = [s0];
3 while true do
4   if fringe empty() then
5     return NONE;
6   s = fringe remove();
7   if goal(s) then
8     return (SOME s);
9   if closed contains(s) then
10    closed.add(s);
11   fringe.addAll(expand(s));
```

There are several points to note regarding graph search:

1. The closed list contains all the expanded states.
2. The closed list can be implemented using a hash table. So the time taken to add or check membership can be manageable.
3. Both worst case time and space are now proportional to the size of the state space. (Which is BIG‼‼)
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.

Unfortunately last point breaks the proof…
Unfortunately last point breaks the proof...

- 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 \( h(s) \) which forces the best path to a repeated state to be generated *first*.

The required condition is called **monotonicity**. As

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

this is an important property.

**Monotonicity**

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

\[
h(s) \leq \text{cost}(a,s) + h(s')
\]

where \( a \) is the action moving us from \( s \) to \( s' \).

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

\[
f(s') = \max\{f(s), p(s') + h(s')\}
\]

This is called the *pathmax* equation.

The pathmax equation

Why does this make sense?

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

But \( s' \) is *on a path through* \( s \). So to say that \( f(s') = 7 \) makes no sense.
A* graph search is optimal for monotonic heuristics

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

We therefore have the following situation:

\[
\begin{align*}
\text{f}(s) &< \text{f}(s') \\
\text{f}(s''') &< \text{f}(s') \\
\text{f}(s''') &< \text{f}(\text{Goal})
\end{align*}
\]

Consequently everything with \( f(s'') < f_{\text{opt}} \) 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 action has cost at least \( c \).

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

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

We won’t be proving the following, but they are good things to know:

- 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 \( h(s) \) satisfies a very stringent condition that is generally unrealistic:

\[
|h(s) - h'(s)| \leq O(\log h'(s))
\]

where \( h'(s) \) denotes the real cost from \( s \) 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.

IDA’ - iterative deepening A* search

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.
IDA* - iterative deepening A* search

The function `contour` searches from a specified state \( s \) as far as a specified limit \( f \) limit on \( f \).

It returns either a path from \( s \) to a goal, or the next biggest value to try for the limit on \( f \).

```plaintext
1 function contour(s, fLimit, path)
2 nextF = ∞;
3 if (s) > fLimit then
4 return ([], f(s));
5 if goal(s) then
6 return (s :: path, fLimit);
7 for s' ∈ expand(s) do
8 (newPath, newF) = contour(s', fLimit, s :: path);
9 if newPath != then
10 return (newPath, newF);
11 nextF = min(nextF, newF);
12 return ([], nextF);
```

IDA* - iterative deepening A* search

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

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

```
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.

---

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 use \(f\), but only use linear space by doing a depth-first search with a few modifications:

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

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.

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 the limit on \(f\) by a fixed \(\epsilon\) at each stage, guaranteeing a solution at most \(\epsilon\) worse than the optimum.

---

```plaintext
1 function rbfs(s, fLimit)
  2 if goal(s) then
  3 return (SOME s, fLimit);
  4 if expand(s) = ∅ then
  5 return (NONE, ∞);
  6 for each s' ∈ expand(s) do
  7     f(s') = maximum(f(s'), f(s));
  8  while true do
  9      best = s' ∈ expand(s) with smallest f(s');
 10      if f(best) > fLimit then
 11         return (NONE, f(best));
 12      nextBest = s' ∈ expand(s) with second smallest f(s');
 13      (result, f') = rbfs(best, minimum(fLimit, f(nextBest)));
 14      f(best) = f';
 15      if result = NONE then
 16         return (result, f');
```

Recursive best-first search (RBFS): an example

This function is called using $rbfs(s_0, \infty)$ to begin the process.

Function call number 1:

![Diagram](https://via.placeholder.com/150)

Now perform the recursive function call $(\text{result}_1, f') = rbfs(\text{best}_1, 5)$ so $f(\text{best}_1)$ takes the returned value $f'$.

Function call number 2:

![Diagram](https://via.placeholder.com/150)

Now perform the recursive function call $(\text{result}_2, f') = rbfs(\text{best}_2, 5)$ so $f(\text{best}_2)$ takes the returned value $f'$.

Function call number 3:

![Diagram](https://via.placeholder.com/150)

Now $f(\text{best}_3) > f(\text{best}_1)$ so the function call returns $(\text{NONE}, 10)$ into $(\text{result}_3, f')$ and $f(\text{best}_2) = 10$.

The while loop for function call 2 now repeats:

![Diagram](https://via.placeholder.com/150)

Now $f(\text{best}_2) > f(\text{best}_1)$ so the function call returns $(\text{NONE}, 9)$ into $(\text{result}_2, f')$ and $f(\text{best}_1) = 9$. 
Recursive best-first search (RBFS): an example

The while loop for function call 1 now repeats:

```
nextBest = 7
```

5 replaced by 10
4 replaced by 9

best = 3

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$^\star$.

And some less nice ones:

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

To some extent IDA$^\star$ and RBFS throw the baby out with the bathwater.

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

MA$^\star$ and SMA$^\star$ will not be covered in this course...

Local search

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

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

This leads to the remarkably simple concept of local search.

Instead of trying to find a path from start state to goal, we explore the local area of the graph, meaning those states one edge away from the one we’re at:

We assume that we have a function $f(s)$ such that $f(s') > f(s)$ indicates $s'$ is preferable to $s$. 
The $m$-queens problem

You may be familiar with the $m$-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 $m \simeq 1,000,000$.)

---

Hill-climbing search

Hill-climbing search is remarkably simple:

1. Generate a start state $s$;
2. while true do
   3. Generate the neighbours $N = \{s_1, \ldots, s_p\}$ of $s$;
   4. $N_f = \{f(s) | s \in N\}$;
   5. if $\text{max } N_f \leq f(s)$ then
      6. return $s$;
   7. $s = s_i \in N$ with maximum $f(s_i)$;

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.
Hill-climbing search: the reality

We might alternatively allow *sideways moves* by changing the stopping condition:

1. if \( \max N f < f(s) \) then
2. return \( s \).

Why would we consider doing this?

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

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

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

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 \( N(s) \) denote the neighbours of \( s \). Define

\[
N^+(s) = \{ s' \in N(s) | f(s') \geq f(s) \}
\]

\[
N^-(s) = \{ s' \in N(s) | f(s') < f(s) \}.
\]

Then

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

- **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 \( k \) states at any given time. At each search step, find the successors of each, and retain the best \( k \) from all the successors.

  *Note*: this is *not* the same as random restarts.
Gradient ascent and related methods

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

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

$$x_{opt} = \arg \max_x f(x)$$

For the purposes of this course, the training of neural networks is a notable example.

In a single dimension we can clearly try to solve

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

to find the stationary points, and use

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

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

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

where

$$\frac{\partial f(x)}{\partial x} = \left[ \frac{\partial f(x)}{\partial x_1} \frac{\partial f(x)}{\partial x_2} \cdots \frac{\partial f(x)}{\partial x_n} \right] .$$

and the equivalent of the second derivative is the Hessian matrix

$$H = \left[ \begin{array}{ccc} \frac{\partial^2 f(x)}{\partial x_1^2} & \frac{\partial^2 f(x)}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f(x)}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f(x)}{\partial x_2 \partial x_1} & \frac{\partial^2 f(x)}{\partial x_2^2} & \cdots & \frac{\partial^2 f(x)}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f(x)}{\partial x_n \partial x_1} & \frac{\partial^2 f(x)}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f(x)}{\partial x_n^2} \end{array} \right] .$$

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

The simplest way around this is to employ gradient ascent:

- Start with a randomly chosen point $x_0$.
- Using a small step size $\epsilon$, iterate using the equation
  $$x_{i+1} = x_i + \epsilon \nabla f(x_i).$$

This can be understood as follows:

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

This is perhaps more easily seen graphically…

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...

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.