The University of Melbourne

COMP30024 Artificial Intelligence

Notes Matt Farrugia 2016

[TOC]

Artificial Intelligence

Defining AI

Four broad approaches to defining the goals of AI

- Acting Humanly
  - All about designing agents who will be able to act as humans do/would
  - Requires
    - natural language processing
    - knowledge representation
    - automated reasoning
    - machine learning
    - computer vision
    - robotics
  - Might be tested via the turing test or total turing test
- Thinking Humanly
  - Replicating human thought patterns in agents
    - For example Newell and Simon's GPS using the same reasoning steps as a human when solving problems
  - Brings up the question of how humans actually think!
    - See this through introspection
    - or observation (experimentation)
    - or even brain imaging
  - See cognitive science
- Thinking Rationally
  - Rather than thinking like a human would, thinking with perfect rationality, with irrefutable reasoning processes
    - For example following Aristotle's syllogisms or formal logic
  - However it's tricky to formulate problems in this manner, in particular problems involving uncertainty
  - And the computational complexity explosion is also a big obstacle (in any approach but appeared first in this one)

- Acting Rationally
  - Acting to achieve the best expected outcome according to some performance measure
  - Not concerned with doing so in a 'human-like' manner, in fact humans are not always rational anyway
  - More general than thinking rationally, in fact thinking rationally is just one method for acting rationally
  - More solid mathematics behind it than the human approaches because rationality is easy to define, compared to human thought or reasoning which we don't actually fully understand

The Turing Test

Provides an operational definition for intelligence (at least from an 'acting humanly' perspective):

To pass the test a computer must respond to questions posed by a human interrogator in a way such that the interrogator cannot tell whether the responses come from a human or from a computer.

Note that it's a biased test - a human would not be able to imitate a computer because of things like processing speed and stuff like that

The State of the Art

Things we can do now

- Automatic Driving
  - Google's self-driving car
- Speech recognition
  - For example Siri and Automated phone conversations
- Autonomous planning and scheduling
  - NASA's Remote Agent planning space station repairs and dealing with problems based on goals specified from Earth
  - Followed up by daily planning systems on Mars rovers
- Game playing
  - IBM's Deep Blue beating chess champions
  - Google's AlphaGo beating Go champion 4-1
- Spam classification
  - Most email systems use advanced learning algorithms to keep up with spam email

- Robotics
  - iRobot's Roomba cleaning homes
- Machine Translation
  - Computer Scientists (who do not speak arabic) using statistical modelling to translate news from arabic to english
- Theorem Proving
  - This one was solved ages ago, but Newell and Simon's Logic Theorist proved lots of geometry theorems and even found a nicer proof for one of them than mathematicians had found so far

Agents

Defining Agents

Agents are systems that interact with an environment, they receive *percepts* from their environment through their sensors, and respond with *actions* which influence the environment through their actuators.

An agent's behaviour is fully (and externally) specified by its *Agent Function* — an abstract mathematical idea that represents a mapping from 'percept sequences' to actions; which action will the agent perform at this instant given the sequence of percepts from its first to its most recent? This abstract function takes the form of an incredibly large (probably infinite) table of percept sequences and their corresponding actions. It's probably not a good idea to try and use this table as a means for implementing an agent. Instead, we need to come up with an *Agent Program*; a finite set of instructions for responding to current percepts. This program will give the illusion of our large/infinite table.

Rational Agents and Performance Measures

Within this framework it's now possible to decide whether or not an agent is *rational*. A *rational agent* acts in whatever way will maximise the expected value of its (eventual?) *performance measure* based on the current percept sequence.

Whether or not an agent is rational will therefore depend on this performance measure. A performance measure is an external measurement of the 'desirability' of the sequence of states of the environment. It's important that this performance measure is external to the agent, so that the agent cannot delude itself into considering its performance perfect. **Further, it's generally better to design performance measures according to what you actually want, rather than how you think the agent should behave.** An agent can otherwise 'game' the performance measure, without actually delivering desirable outcomes.

As an example, consider a soccer playing agent. One possible performance measure may be to measure the number of goals scored, the number of successful passes, the percent possession of the team, the following of some established team strategy, the avoidance of fouls, or some combination of these measures. However, it's easy to miss something; like the fact that you don't want the agents to pass the ball around without trying to *actually win the game* just to maximise passes and possession, or that you don't want the agents to let the other team score goals just so that they can get a free kick from the center of the field. In fact, if both teams were agents with this same simple 'goals scored' performance measure, it may be considered rational for them to co-operatively take turns in scoring one goal at a time without trying to stop each other, so that by the end of the game you have a scoreboard of 60-61 and both teams think they have done extraordinarily well, when what you really wanted was a competition. Following a team strategy may similarly result in undesirable behaviour if the strategy is proving ineffective, in which case what you really want is the agents to adapt and respond to that.

Maybe a better performance measure would be 'games won'. This leaves the complex decision making to the agent; if acting rationally, the agent would therefore try to maximise goals scored by their team and minimise those scored by the other team. Passing, competitively maximising possession and following a team strategy are legitimate rational means towards winning the game, so these are automatically considered favourable actions. Avoiding fouls is a legitimate way to maximise your team's performance because for one thing, it prevents the other team getting free kicks, and for another, it allows you to continue playing and working with your team towards winning the game.

Other factors could be included other than games won, such as 'audience enjoyment' or 'overall happiness' (since it's not always all about winning!), but it's advantageous to specify your goals rather than broad suggestions for how to achieve them.

Rationality is not the same as *omniscience* — to be considered perfect or omniscient an agent would have to act based on the actual future outcome of its actions, and all future percepts. However it's often impossible to know these in advance and so omniscience is unattainable. Rationality only requires an agent select the action *most likely* to result in the highest value of the performance measure in the future, based only on the past and the present and any inbuilt knowledge the agent has of the possible consequences of its actions. It's also useful to minimise the inbuilt knowledge installed in the agent at the beginning, and it's better that the agent can learn this on its own in an *autonomous* manner.

As an example, consider humans, provided with a set of instincts and reflexes (to breath, heartbeat, to recoil from a hot stove) enough to ensure survival long enough to learn how to operate in a rational manner.

Environments

An agent's task environments can be specified by four things

- The performance measure (P) — with which the agent's rationality will be judged.
- The environment itself (E) — with which the agent will interact.
- The agent's actuators (A) — with which the agent will influence its environment.
- The agent's sensors (S) — with which the agent will perceive its environment.

There are lots of examples of these classifications in the textbook on page 42 (3ED).

From there, we can classify an environment based on a its properties

---

- **Fully Observable**
  - the agent's sensors give it access to the complete state of the environment at any time
  - or, if they give access to all information relevant in making rational decisions under this performance measure, it's *effectively* fully observable
- **Partially Observable**
  - the agent's sensors give it access to only some of the state of the environment, for example, because it can only see parts of the physical environment immediately nearby, *or because it cannot see what other characters in the environment are thinking*
- **Unobservable**
  - the agent cannot perceive any of its environment, as it has no effective sensors

---

- **Single agent**

  - the agent is the only one interacting with its environment, or at least the only one considered an agent

- **Multi agent**
  - or, are there other agents acting in the same environment
  - The choice of whether or not the consider another entity as an agent will depend on whether it can be viewed as *acting to maximise some performance measure which depends on our agent's actions*
    * this leads to further distinctions between *competitive* and *cooperative* multi-agent environment depending on how each agent's performance affects the others'.

- **Deterministic**
  - the next state of the environment is completely determined by the current state and the action executed by the agent (apart from uncertainty that arises from the actions of other agents)

- **Stochastic**
  - the next state of the environment is one of a set of possible states, each with their own probabilities
  - in a lot of practical situations, due to complicated (but technically deterministic) state transitions or partial observability, environments are *effectively* stochastic (for example, driving a car)

- **Non-deterministic**
  - the next state is one of a set of possible states, but without worrying about probability
    * useful if you're trying to maximise the outcome for all possible outcomes rather than just the expected outcome

Note: an *uncertain* environment is one that is not both fully observable and deterministic, so partially observable or stochastic

---

- **Episodic**
  - the agent's experience is divided into atomic episodes, each of which involve a percept and a single action, and *the outcome of future episodes does not depend on previous ones*

- **Sequential**

  - the response the the current percept could effect future decisions
  - it's also worth noting that task environments can be episodic at higher levels, for example the moves in a game of chess have consequences for the rest of that game however if your environment is a chess *tournament* composed of many games then perhaps the individual games can be considered episodes

---

- **Static**
  - the environment will not change until the agent responds to the last percept, which is useful because it means the agent does not have to continuously perceive its environment
- **Dynamic**
  - the environment may change while the agent is deciding on how to respond to a percept, and therefore the act of not responding before the environment changes is considered an action; 'doing nothing'
- **Semi Dynamic**
  - the environment does not change with the passage of time but the agent's performance score does, for example timed tasks or games like chess with a clock (assuming the clock is not considered part of the environment, which actually might be useful because it can effect how long to spend thinking about things like for when there's not much time left on the clock)

- **Discrete**
  - the passage of time, the environment states, and the percept inputs are quantized
  - note that technically visual input is discrete but is usually treated as continuous
- **Continuous**
  - the passage of time, the environment states, and the percept inputs range over continuous values

---

- **Known**

  - the agent knows how its actions will (or could) influence the environment

- **Unknown**
  - the agent does not know how its actions will (or could) influence the environment note that this last one would kind of be a property of the agent rather than the environment (unless I suppose we had an agent)

Agent Types

Within this task environment, we are tasked with creating an agent program that takes percepts from the sensors and returns actions to be carried out by the actuators. The collection of sensors, actuators, and hardware running this agent program is referred to as the agent's *architecture*.

Within this framework for agent program there are many possible agent designs you can try as you aim for a rational agent for your task environment.

Table-Driven Agents

Perhaps the most naive type of agent is a table-driven agent, a direct implementation of the agent function.

```
persistent: percepts --- a sequence, initially empty
            table --- a table of actions indexed by percept sequences, initially fully speci

function: TABLE-DRIVEN-AGENT(percept) returns and action
    append percept to end of percepts
    action <-- LOOKUP(percepts, table)
    return action
```

Unfortunately, such an agent is impossibly impractical due to the immense size of the required table.

Despite this, it's effective, and now we can try to come up with better ways of achieving the same thing!

Simple Reflex Agents

The simplest kind of agent. Selects an action purely based on the incoming percepts, based on a set of **condition-action-rules** for how to respond to such states.

```
persistent:
    rules --- a set of condition-action rules

function: SIMPLE-REFLEX-AGENT(percept) returns action
    state <-- INTERPRET-INPUT(percept)
    action <-- GET-RESPONSE(sate, rules)
    return action
```

`INTERPRET-INPUT` takes a percept and generates an abstract representation of the current state of the environment. `GET-RESPONSE` looks up this state/condition in the rules table and returns the specified response, the action which should be returned.

Model-Based Reflex Agents

Simple Reflex Agents are good because of how simple they are, however they cannot be expected to perform rationally when the correct rational action depends on more than the most recent percept.

In these situations, an agent must keep track of at least some information about the state of the environment and its percept history, and use this to determine the action to take. Without going all the way to storing a complete percept sequence like in the table-driven agent, we can store relevant information about the environment, its history and the states we cannot observe right now in our own abstract internal representation. To keep track of the parts of the environment that we have not observed for some time, we will also need knowledge of how those parts of the environment may evolve according to our actions (a *model*).

With this information, we can use the current percept and our model to update our internal representation of the state of the environment, and then select an action based on this more complete information.

```
persistent:
    rules --- a set of condition-action rules
    model --- a description of how the next state depends on the current state and previous
    state --- abstract representation of the state of the environment
    action --- the previous action (initially none)

function: MODEL-BASED-REFLEX-AGENT(percept) returns action
    state <-- UPDATE-STATE(state, action, percept, model)
    action <-- GET-RESPONSE(sate, rules)
    return action
```

Goal-Based Agents

Reflex agents do not consider the consequences of their actions, they require a specific set of condition-action rules that lead to rational decision-making in

relation to a specific performance measure. A more general approach is to give the agent a *goal*, allowing it to consider its possible responses and make a choice between them based on how they may influence the performance measure.

In this situation, an agent program is decoupled from the goal it is built to achieve, and different goals can be specified without having to change the structure of the agent program (as opposed to having to come up with a set of condition-action rules for every new goal). The trade-off for this flexibility is speed — the time an agent spends deliberating between possible actions can be costly in dynamic environments.

The process of making decisions can also become rather complicated, and will often require consideration of long sequences of future actions and their consequences through search and planning algorithms.

Utility-Based Agents

When there are multiple ways to achieve a goal, each resulting in a different level of measured performance, we need some way to decide between them. A more thoughtful agent may quantify exactly how 'happy' each sequence makes it, and act to maximise this measure. This also deals with situations where goals are in competition. The agent's measure of happiness can specify the appropriate trade-off. Moreover, this measure of happiness can take into account uncertainty related to future environment states in an uncertain environment.

A *utility-based* agent goes one step further than a goal-based agent and does just this; rather than deciding on actions based on their leading to some 'goal', it will evaluate action sequences in terms of some internal measure of *utility* (happiness). A good measure of utility will reflect the external performance measure, and lead an agent to make decisions that maximise its expected performance measure, resulting in rational behaviour.

General Learning Agents

These more complex agents require complicated programs to make rational decisions. For complicated environments the task of creating such agents manually can be incomprehensible.

Instead, one can attempt to make a *learning agent* which can learn how to be rational over time, by experimentation and by modifying its own behaviour in response to certain actions. This is analogous to how human children are armed with enough intelligence to learn and become better at approximating rational agents.

There's more on learning agents in the book (page 55 and part V (3ED)) but I don't think we're spending much time on them this semester.

Search

Problem Formulation

A search strategy is undertaken by a *problem-solving agent* (goal-based agent working with atomic state representations) to find a sequence of state transitions from some start state to some goal-state. Any agent may attempt to solve a search problem as a means of achieving some temporary goal on its way to maximising its performance measure.

The formulation of a search problem has 4 components

- The *initial state*, where the agent starts
- A description of the applicable *actions* from any state, and a *transition model*, which is a description of what state each possible action takes us to from a given state (these reachable states are called the *successors* of a state)
- A *goal test*, for checking whether or not a given state is considered a goal state (can be *explicit* ~(`x==a`) or *implicit* ~(`a<x<b`))
- A *path cost function* that assigns a numerical cost to each path (a path is simply a sequence of states connected by actions)

A solution to such a search problem is a path that begins with the start state and ends with a goal state. We can use its path cost to determine whether or not a particular solution is *optimal*, meaning it has the lowest possible path cost of all solutions.

In deciding on the formulation of a search problem, we undergo a process of *abstraction* — removing unnecessary details. It's important to remove unnecessary details so that the state space of our search problem is not cluttered. An abstraction is *valid* if it's possible to convert any abstract solution into a detailed solution, and it's *useful* if doing so is not harder than solving the original detailed problem in the first place.

Search

The whole idea of search is that it's an offline *simulation* of an exploration of the state space, so that we can foresee what is ahead and choose the best path. The exploration constructs a *search tree* made of *nodes* which correspond to a particular state in the state space of the problem, and by *expanding* these nodes we generate child nodes which contain successor states.

Note: nodes form the tree and contain states while states form the state space and do not concern themselves with search stuff like path cost or depth - in fact states may be referred to by multiple nodes in the search tree (corresponding to their occurrence in different paths)

Here's some rough pseudo-code:

```
Search:
    nodes = an empty queue
    add initial state node to queue
    while queue has nodes:
        if next node in queue is goal:
            return this node
        else:
            expand this node and add its children to the queue
```

Different search strategies mostly use this structure, with the queue's semantics (FIFO, FILO, something else?) being the main source of difference in strategy.

This 'tree search' approach (where we'd discard nodes after their expansion and only keep the frontier of the search) works but may add to the (time) complexity of the algorithm if we end up regenerating nodes unnecessarily. In contrast a 'graph search' approach has us only consider each node once by keeping track of the expanded set as well as the frontier and performing a lookup when you generate new nodes (which may increase space complexity but improve time complexity). In finite graphs, graph search's complexity is bounded also by the size of the graph.

To evaluate a search strategy, we can consider its time and space complexity, usually measured in terms of + b — the branching factor of the problem + d — the depth of the shallowest goal node in the complete search tree + m — the depth of the deepest node in the complete search tree (which may be infinite if repetition starts happening!)

We're also concerned with *completeness* (will we find a solution if one exists?) and *optimality* (will the solution we find be the least-cost solution?).

Uninformed Search Strategies

With no extra information about the state space than the current state and its successors, we can only conduct search in a blind manner; generating successors and exploring them in some order. Search strategies that do so are called *uninformed* search strategies.

Breadth First Search (BFS)

Expands nodes in FIFO order (using a regular FIFO queue).

BFS is complete (*if b is finite*, this caveat applies to all the complete searches, actually) as it tests/expands all nodes at any depth and then proceeds to test/expand those at a shallower depth. It not guaranteed to produce an optimal solution, unless all transitions incur a uniform cost.

It's runtime will look like 1 + b + bˆ2 + bˆ3 + ... + bˆd = O(bˆd)*

As we have to store the entire frontier (up to bˆd nodes) in a queue at one time, the space complexity is also O(bˆd)*.

*This analysis depends on the strategy of goal-testing nodes as they are generated, rather than as they are expanded. Since BFS expands all nodes at a layer (generating nodes at the next layer) before expanding any nodes at the next layer (generating nodes at the layer after that) this doesn't effect the optimality analysis, shallower nodes are still tested first. All we need is a special case for testing the initial state before we begin expanding.

Uniform Cost Search (UCS)

Expands nodes in lowest-path-cost-first order (using a min-priority queue). (note not quite the same as lowest-transition-cost-first). Basically Dijkstra's algorithm but only searching from a single source to a single goal (the closest goal state).

UCS is optimal and complete assuming transition costs are positive (zero's could cause infinite search loops if we're conducting a tree search and negatives are a no-go for Dijkstra's algorithm, they could cause these loops and also they undermine its optimality).

UCS has a runtime and memory requirements proportional to the number of nodes with path-cost <= the cost of the optimal solution. Let C* represent the cost of the optimal solution, and assume each edge has cost at least ϵ. Then, the algorithm's worst case time and space complexity is in $O(b^{1+C*/\epsilon})$ (floor division).

Depth First Search (DFS)

Expands nodes in FILO order (using a regular stack).

DFS is ONLY complete if the search space is finite and acyclic (a finite tree or DAG), otherwise it can go off on an infinite journey into the depths and never return to find a solution. What a tragedy! The other fix to this is using graph-search again, storing expanded nodes' states so that they are not generated a second time. As long as the search space is finite, this will give us completeness. However, even when it's complete, it's not going to be optimal, it could always miss the best path.

DFS's time complexity is O(1+b+$b^{2+\cdots+b}$m) = O(bˆm), which is bad of course especially if m >> d (and horrible if m is infinite!). Its redeeming quality is its O(m*b) space complexity, as all it needs to store is the current path (depth at most m) and* stubs* of the b alternate paths expanded at each node along the current paths so that it can come back and follow them instead.

Depth-limited Search (DFS(l))

DFS, but with a 'depth-limit' of l which just means that any nodes at depth l are not expanded.

DFS(l) is not complete because if $l < d$ then goal nodes are never expanded. Once again, even if it does find a solution with $d <= l$ it's not guaranteed to be an optimal solution by any means, for the same reasons as for DFS.

However, it does have a fixed maximum runtime of O(bˆl) (and space complexity of O(l*b*)), *which is a little bit redeeming, especially if you happen to have some idea about* d* and can choose a good *l*. Also, it's super useful in the following strategy...

Iterative Deepening Search (IDS)

Perform DFS(l) with iteratively-increasing depth limits! This has the effect of searching the tree in a breadth-first-depth-first manner! If that makes sense. Here's pseudo-code:

```
l = 0
while DFS(l) doesn't find a solution:
    l++
return solution
```

This is complete for the same reasons as BFS is complete, it will expand all nodes at depth n before expanding any at depth n+1 (on the search with l = n).

It's also optimal under the same conditions as BFS, when transition-costs are uniform.

You may think that performing DFS(l) so many times is inefficient, but not asymptotically (it turns out)! The runtime complexity looks like:

$$(d+1) * b^0 + (d) * b^1 + (d-1) * b^2 + ... + b^d$$

which, it turns out, is in O(bˆd) (even though it's larger than BFS's bˆd, it's all a matter of lower-order terms).

Likewise, space complexity is *great* because it has the benefits of being a depth-first searching strategy, in total O(d*b).

Bidirectional Search

This search strategy actually performs two searches in parallel: one from the start node and one from the goal node. In theory, if these meet in the middle it will involve two searches of time O(bˆ(d/2)) which is way better than O(bˆd)!

However, there are some issue to consider. Firstly, there may be multiple goal nodes. This is easily circumvented by creating a new pseudo-goal node whose predecessor is all of these goal nodes, if you know where they all are in advance. However, that may not be the case. Further, how do you generate predecessor states for the backwards search? This may be awkward and may involve a much higher branching factor than generating successor states in some search problems. Finally, the issue of 'meeting in the middle' is easier said than done. I think at least one of the searches needs to proceed with a frontier with no holes in it (so not DFS or IDS) if we're not storing an explored set, otherwise, the two searches may miss each other. Then there are other optimality considerations; are you happy with the *first* instance of a collision between the searches? or may this strategy miss something? I suppose if you're using two optimal searches, then this might be all right.

From the slides, the analysis is fairly straight forward; complete and optimal* (depending on strategy), with time/space complexity of O(bˆ(d/2))

Informed / Best-First Search Strategies

A step above the 'uninformed' search strategies which follow some variation of a simple queuing mechanism is to 'evaluate' each node we encounter using some *evaluation function*, f(n), and to use a priority queue that allows us to pick the 'best' looking node to expand next (minimal value of f(n)). Coming up with a good evaluation function requires knowing about the search space and so these search strategies are called 'informed' search strategies.

UCS can be thought of as a simple best-first search strategy where f(n) is just g(n) - the path cost from the source node to the node n.

Heuristics

By using a more thoughtful choice of f(n), we can achieve better performance than UCS. For example, we could introduce a heuristic function *h(n)*, which estimates the best cost of a goal node from a search node *n*. Using a heuristic function as part an evaluation function can help to steer the search in the 'right direction', helping it to avoid expanding nodes that are unlikely to lead to a solution. The choice of a good heuristic is important here, of course.

Using any old heuristic can get us into trouble. For example, if the heuristic assigns a high cost to a node on a path that is actually a good path (perhaps even an optimal one), the algorithm following a heuristic will neglect this path. For this reason, it's important to spend time making sure your heuristic is 'good'.

For a heuristic to be considered 'good', it should first be *admissible*, which is a mathematical property that requires that for all nodes n,

$$h(n) \leq h*(n)$$

where h*(n) is the true best path-cost from n to the goal. This is saying that we don't want to* overestimates* the path-cost. This will help to prevent our algorithm from discounting optimal solutions.

Another nice property for your heuristic to have is *consistency*. For a heuristic to be considered *consistent*, h(n) must always be at most equal to h(m) + C(n, m) where m is a the successor node of n and C(n, m) is the step cost from n to m.

$$h(n) \leq h(m) + C(n, m)$$

That is, the heuristic is not only never an overestimate, with respect to the actual cost, but it's actually also never an overestimate with respect to its own estimates, further down a path! Another fun fact; consistency implies admissibility! Isn't that nice?

Another term that comes into evaluating heuristics is *dominance*. A heuristic dominates another heuristic if it gives a larger (or equal) value for all nodes in the state space. If it does so while also remaining admissible (and potentially also consistent) then it's going to help out algorithms perform better, as it gives a closer estimate of the true cost.

The best, most informed/dominant heuristic is the exact, true best path-cost function h*(n), however that value is often part of what we are trying to find out! There's no use solving a search problem during our search problem to find the value of h* for a node. However, we can often get a great heuristic from solving a *simpler* problem; one that is computationally trivial compared to the original search. This can commonly be found by *removing constraints from the original problem.*

For example, in a Euclidean navigational search problem, perhaps we must avoid obstacles and can only move in the horizontal and vertical dimensions. If this was the case, perhaps we could find a heuristic by removing the obstacles and the constraints on movement, and just use the straight-line, Euclidean distance from our position to the goal as an estimate of the true cost. Since there *are* obstacles and movement constraints, this is never going to be an underestimate. Further, we could construct an even better heuristic by only removing the obstacle constraint, in which case the *Manhattan distance* (horizontal distance + vertical distance) would do fine. This is bound to be greater than or equal to the Euclidean distance, and it's still bound to be an underestimate on the true cost, so it would be a better (dominant) heuristic in a situation with these movement constraints.

Greedy Search

One informed search strategy is then to use f(n) = h(n), meaning will expand the node that has the lowest h(n) value at each stage. This seems like a decent idea, as this represents choosing the node that is closest to our goal.

This 'greedy' search, however, runs into issues in certain situations. It's possible that expanding according to a heuristic can get you into an infinite loop if you're stuck moving back and forth at a dead end facing the solution, incapable of moving further way in order to find a path that actually leads there. For this reason, Greedy search is not complete. It's possible to avoid infinite loops by performing a graph search instead, though this does not give us completeness in infinite state spaces!

Greedy search is also not optimal. It's entirely possible that the path that is always taking you closer to the solution is not actually the shortest path to take. There may be a shorter overall path that Greedy search does not consider.

In terms of space and time complexity, since we need to keep the entire frontier with us, we're looking at O(bˆm) for both in the worst case. This of course depends on the quality of the heuristic, as a good heuristic will allow us to avoid expanding many of those paths and instead head straight for the solution.

A-Star Search

A* (A-Star) search is a version of best-first searching that uses a combination of a heuristic function h(n) and also the path-cost function g(n). In A, *f(n) is simply g(n) + h(n). That way, f(n) is a half-estimate of the total cost of a path, when n is somewhere along a partly-formed path. The g(n) part is accurate, but since we're using a heuristic it's possible that h(n) is not exactly right - the true cost would be g(n) + h*(n).

Including g(n) *and* h(n) in our evaluation function combines the best of UCS and Greedy Search. A* is hesitant to follow paths that take us off in the wrong direction (due to h(n)) and it also doesn't relentlessly follow paths that take it close to the solution while there is another way that could also be investigated, and it won't get stuck like Greedy search. In fact, A* is complete as long as there are finitely-many nodes with f(n) less than f(G) where G is the goal node along the optimal path.

A* is also optimal, *with a good heuristic*. Specifically, the tree-search version of A* is optimal as long as h(n) is *admissible*, and the graph-search version has the stricter requirement that h(n) be *consistent*. In addition, it's also *optimally efficient* for a given good heuristic. That means that any other algorithm viewing less nodes for a certain problem runs the risk of missing the optimal solution.

In terms of its time and space complexity, however, it's not perfect. A* has time complexity $O((b^{\epsilon})^{d})$, where ϵ is the heuristic's *relative error*, or (h* - h) / h. *There may also be exponentially many nodes with f(n) less than the optimal cost of a goal node, which means the search will perform poorly in these cases. Further, by keeping all nodes in memory, A* has poor space complexity too (and often runs out of memory before it runs out of time).

Extra Cool Evaluation f(n)'s

Other evaluation functions can make interesting search algorithms. With f(n) = -h(n), the search algorithm will run away from the solution (oh no!). With f(n) = -g(n), the algorithm will run away from the start node (kind of imitating DFS for uniform step cost problems). With f(n) = g(n), we'll stay as close to the start as possible (which is actually UCS, and will imitate BFS on uniform step-cost problems).

Local Search

Sometimes, instead of searching for a path from an initial state to goal state of known criteria, we're searching a state space for an unknown solution, and the final state itself is the solution to our problem (the path to the solution is irrelevant). The state space may be infinite, or continuous, and the goal may be 'optimisation' rather than a specific test. Problem solving by systematic tree / graph search does not suit these conditions. In these cases, a more appropriate strategy is to conduct a local exploration from a start state, tracking only the current state and not the cumulative path.

Hill Climbing / Gradient Ascent

The gradient ascent algorithm's operation is simple; consider all neighbour states, and move to the one with the highest (or lowest, as appropriate) value according to some evaluation. Repeat this process until a a local maximum is reached - no neighbours are 'better' than the current state. The state space we are searching is the collection of all complete states. In an optimisation situation (such as TSP) the value of a state is would be the optimisation criteria (the cost of a path, which we would try to minimise by changing the path). In a constraint satisfaction problem (e.g. n-queens), the value may be the number of violated constraints (the number of 'queens attacking queens', for example, which we would again try to minimise).

This greedy local search is analogous to 'climbing a hill' in a state space where the value of a state is represented as its height. The algorithm takes a step upwards until it is no longer possible to do so. It is not able to look ahead, however, and see further than the options directly in front of it. Neither can it watch where it came from. Somewhat analogous to climbing a hill *in the fog*.

This algorithm performs well in some search spaces. But hill climbing in the fog can get you into trouble. One major issue is that a 'local maximum' is not necessarily the global maximum. By terminating whenever we see no better option around us, we risk passing up the optimal solution. It's not even necessarily a maximum, by our definition, but may be part of a plateau, where all neighbours have the same value. This plateau may be at the top of a hill, or it may be part of a *shoulder*, where progress in one direction is possible. However, we have no way to know in which direction since we are steering based on the direction of steepest ascent!

Some variations of the basic hill climbing approach can improve the algorithm's performance.

- Stochastic Hill Climbing — choosing a random move from among those that steer us uphill, not necessarily always going from the best move. May take longer to converge on solutions but in some state spaces, it finds solutions better than an approach of pure greed.
- First-choice Hill Climbing — choosing the first uphill state generated. This can be necessary in state spaces where there are an incredibly large number of neighbour states.
- Random-restart Hill Climbing — responding to convergence by restarting in a randomised initial state (which is sometimes easier said than done) in an attempt to eventually end up in a state that finds a *great* solution, even if individual searches don't turn out too great.
- Simulated Annealing — a combination of pure 'hill-climbing' (never make a downhill move) and pure 'random walking' (make random moves) can be shown to improve performance, borrowing a concept from metallurgy (annealing, which tempers metals by rapidly heating and then slowly cooling them, allowing their atoms to reach a low-energy crystalline state). In our case, we wish to find a low-cost state, which may be a minima or maxima depending on our perspective. Either way, the idea of simulated annealing is to allow ourselves to make downhill moves while the state space is hot, with decreasing probability as things start to cool down. This should stop us from quickly getting stuck on a local maximum, long enough to let us try to find a better hill to climb. With an appropriate 'cooling' schedule, simulated annealing can yield optimal results with high probability.

Adversarial Search Strategies

In a competitive multi-agent environment (such as many turn-based two-player games) our agent has to deal with the unpredictability of an *opponent* or *adversary* making decisions, usually with the ultimate goal of trying to minimise our performance (in order to maximise its own). It's not enough to come up with a single search path from the current state of the game to a 'winning' or favourable outcome, as we cannot know in advance what choices our opponent will make. We can formulate a game like this as an *adversarial search problem*, which comprises + an *initial state*, representing the beginning of a game. + a set of allowable *actions* for each state, and a *transition model* defining the resulting states from each of those actions. + a *terminal test* which tells us if the game is over, and what the outcome is. + a *utility function* that gives us a numerical outcome of a finished game. In a 'zero sum' game, the utility value from the perspectives of each player sums to a constant (let's say zero because that makes sense) no matter the outcome. This means that in a 2-player game of this sort, the opponents utility value is the negative of our utility value.

It's often also important to keep track of whose 'turn' it is, but this can be included in the state itself I suppose.

Another point of terminology: a 'move' (or 'play'?) represents an action from each player, each of which is a 'ply' (or half-move).

Perfect Play and Minimax

One strategy under these conditions is to make choices so that we maximise the utility of the worst possible outcome that our opponent can force on us. This maximisation needs to take into account all of the possible moves that our opponent could possible make, and all of the responses it could possible have to all of our subsequent moves, and so on. This strategy is called 'perfect play', as it leads to outcomes at least as good as any other strategies when playing an infallible opponent (though other strategies can perform better against less-than-infallible opponents).

An algorithm for this strategy is the Minimax algorithm. Minimax considers a game search as a search tree comprising nodes alternating between 'min' nodes and 'max' nodes as we move down the tree. It then works up from the trees terminal states, assigning a 'minimax value' to each internal node according to the minimum / maximum value of each child node. This corresponds to the opponent (out to minimise our utility value in order to maximise their own) choosing the smallest choice, and at the layer above, we much choose the maximal move given that we know the opponent will be minimising while it's their turn.

Here's some pseudo-code:

```
minimax(state, maximising):
    if this state is a terminal state:
        return the utility value of this state
    else if maximising:
        return action, value such that minimax(result of action, not maximising) is a maximu
    else:
        return action, value such that minimax(result of action, maximising) is a minimum
```

In terms of actually implementing it, there are a few extra for loops and if statements to implement finding the maximum/minimum values and a few details involved with returning actions and values at the top level, but the logic is there!

Going back to our search algorithm evaluation criteria, Minimax is *complete* assuming the search tree is finite and *optimal* against an optimal opponent (as already mentioned). However, it's essentially unusable in most practical situations due to its complexity! It searches the entire game tree, with a time complexity of O(bˆm) and a space complexity of O(m*b) (as it can search in a depth-first manner, instead of expanding and evaluating the entire tree). In

most games that O(bˆm) is completely debilitating, unfortunately making this brute-force consider-all-possibilities approach useless.

Evaluation Functions and Cut-off Tests

Resource limits therefore inspire us to search a little harder for an algorithm that doesn't consider *absolutely everything* and search and entire tree. Well, if we had some way to reliably evaluate non-terminal states, I suppose we could cut off minimax before it got too deep, and perform a shallower search. This strategy is useful, but its success relies on having a great *evaluation function* for assigning a utility value to intermediate game states. Inherently, the true value of these intermediate states cannot be easily known, as it depends on how the rest of the game is played out.

However, with a good evaluation function (and a good *cut off test* for determining when to apply it) we are able to perform a *depth-limited* minimax search. This algorithm is then identical to regular minimax, except we use out cut off test instead of our terminal test and our evaluation function instead of our utility function.

Some basic evaluation function features are important. Firstly, it shouldn't assign higher values to states that result in a terminal state of lower utility. Otherwise, the agent following this evaluation function will end up steering the game towards a less favourable outcome even if it can search all the way to terminal states! Secondly, it should try to assign higher values to states to states that represent a higher 'chance of winning' given the uncertainty resulting from not actually searching all the way to the end of the game.

A typical evaluation function views a game state in terms of a set of *features* (such as the numbers of each type of piece available on the board in chess, as well as certain structural piece combinations) which it will combine, potentially using some linear combination function (or perhaps not, to capture complex non-independent relationships between features) where each feature is scaled by some 'weight' and summed to get a value for the state. The success of an agent's strategy is incredibly dependent on the accuracy of this evaluation function!

Care should also be taken in designing a cut off test. The simplest implementation of a depth-limiting test is susceptible to a few potential flaws including the instability (and potential inaccuracy) of the evaluation function at certain volatile points in a game (such as when pieces are under attack in chess, and the next few moves can change the board's features significantly) and also 'horizon' issues, where it appears strategic to delay an inevitable negative outcome in order to push it beyond the depth limit by playing (potentially damaging) 'stalling' moves first. The first problem of board volatility can be solved by using a *quiescence search* — continuing search temporarily beyond the cut off whenever the next few moves result in wild evaluation swings. The horizon effect can be mitigated by tracking 'excellent moves' (or 'dangerous moves'; excellent moves for the

opponent) and ensuring that they are considered and not cut off whenever an otherwise-cut-off state is reached where they are possible ('singular extension').

The problem still exists where we may run out of time searching with a given cut off, especially if we employ non-uniform cut off tests that may delay our searches. What happens if our time runs out and we have not considered every possible move because we searched too deep in some cases? To prevent this issue, we can take a leaf from the uninformed search book and use an *iteratively-deepening depth-limited minimax* so that we can always have a move in hand that considers things as many ply forward as the depth of the last search completed. We can then continue to search until our time runs out, and make the best move returned last. As for IDS, adding iterative deepening on an exponential search tree slows search by only a constant factor.

Pruning and Alpha-Beta

The previous section focuses on reducing the depth of our search at the cost of its optimality. *Pruning* is a technique for reducing the number of nodes evaluated *without* reducing the accuracy of a solution, by selectively refusing the evaluate particular move generations if they cannot possibly change the result of a search. Alpha-Beta pruning is an example of such an enhancement to minimax search, which in practice can cut a search l plies deep from O(bˆl) down into O(bˆ(l/2)) (hey, that lets us search *twice as far* in the same time!).

The general principle behind alpha-beta pruning is that if there is a node n at some depth and player p has the choice to make that move, but there exists another choice m along the path and p could get a better result by choosing m, then n will never be reached in actual play. That means, once we have found out enough about any node n to know that it can't be better than another previously-evaluated choice m (we can figure this out by examining some of n's descendants), we no longer need to continue evaluating n and can prune it and all of its unevaluated descendants from our search and return immediately from its evaluation!

Alpha-beta is implemented by keeping track of two values (named α and β) at each node along the current path. α represents the best (highest) score that the maximising player is guaranteed so far along this path. If a minimising node ever thinks it can do better than this by encountering a smaller value, then the maximising player will never let it get there! The minimising node can stop evaluating its children and return this min value. Likewise, β represents the best (lowest) score that the minimising node is guaranteed along the path so far. If a deeper maximising node ever thinks it has a chance to do better than this, the minimising node will never let it get there! α-β values are passed down with each recursive call, and can be thought of to start at -infinity and infinity respectively (though I prefer to initiate them as 'α' and 'β' initially to keep track of what they represent). The best way to understand alpha-beta pruning in action is to try your hand at an example, so see this awesome practice tool.

The ability for alpha-beta to successfully eliminate (prune) nodes from the search tree is dependent on the order in which nodes are evaluated. With the optimal ordering, we'd be able to reduce the branching factor to (it turns out) effectively sqrt(b), giving us the O(bˆ(l/2)) runtime we mentioned before.

Knowing an optimal move to expand first requires knowing the best move from state (which is the whole problem we're trying to solve), so it's impractical. However, we can use the result of past searches to inform the selection. This may be the search from the previous turn, or from the previous iteration of iterative deepening!. Trying these 'killer moves' first in practice leads to speed-ups approaching the optimal value (and more than pays for the constant factor increase from iterative deepening, plus you get the insurance policy of always having a move ready-to-go in case your time runs out!).

Non-determinism and Expectiminimax

Deterministic games of perfect-information can be optimally approached with adversarial search, but can we come up with an optimal strategy for games that involve chance? Yes! In these games, we can consider random variables as a 'third player' in our game who does not make maximising or minimising moves, but *random* moves with appropriate probabilities. When approaching these nodes, the maximiser/minimiser can attempt to maximise/minimise the *expected* value of its actions. For this reason, search in these games is sometimes called 'expectiminimax' (even though personally I think 'minimax' was far enough).

Take care to note that pruning is not always possible at these expecti-nodes, if we can't make any assumptions about nodes we haven't evaluated yet. For all we know, they could be incredibly large (or incredibly negative) and sway the calculation of the expected value dramatically. The only case in which we can make assumptions and prune these nodes is when we know the bounds of the utility function, as in these cases we can calculate the 'minimum possible expected value' before evaluating all nodes by considering the case where all unevaluated nodes end up with their minimum value (and the same for the maximum case).

Also note that while in deterministic games the behaviour of an agent is independent of the evaluation function in response to *monotonic transformations* (where the *ordering* of the evaluations of each state is unchanged), only *positive linear transformations* (where the proportion of each pair of evaluations is unchanged) prevent expectiminimax evaluation functions from changing the result of a search.

Learning in Games

Game-playing agent programs require tuning to produce optimal performance. There are several factors where they can be tuned. For example, + tuning how the agent selects actions in an order that helps to maximise the benefits

of alpha-beta pruning + tuning how to decide what search depth should be allocated to different moves + possibly tuning the control of timing at different stages of a timed game + tuning the evaluation function, so that it closely approximates the true final utility from a non-terminal state Machine learning provides algorithms that perform this tuning, so that it does not have to be performed by hand.

Learning Paradigms

These aren't exactly comparable to each other, their grouping is not so helpful, but here are a bunch of terms floating around in machine learning reading! + **Supervised Learning** is learning from examples provided by a knowledgeable external supervisor. These examples take the form of inputs paired with desired outputs. The goal is to find a function that correctly follows training examples, with low error. + **Unsupervised Learning** is the task of inferring a function from *unlabelled* examples. Since these algorithms are not provided labels they must derive their own patterns and structure. + **Reinforcement Learning** has agents given a scalar reward for their actions and the corresponding state transitions. The goal of the agent is to collect as much reward as possible. From the notes, it seems that reinforcement learning may also be less *supervised*, with rewards delayed and the learning agent left to figure out what actions contributed positively/negatively to the overall reward. + **Decision Tree Learning** is another learning paradigm, it looks especially interesting due to its transparency/understandability!

Book Learning

Book learning is a form of *unsupervised reinforcement learning*, where an agent will keep a record of which moves work in certain situations, in an attempt to avoid *losing games multiple times in the same way.* Eventually, book learning would aim to record the best move to make in each game state. Due to the expansive number of possible states in most games, book learning is usually generally impractical.

However, it can be used in a limited sense in the opening stages of a game, where there are a reduced number of possible configurations. This also has implications for the learning process itself, as it can be difficult to deduce the effects of early moves on the end result of the game, how do we know which actions were responsible for the outcome (credit assignment)? The increased time between action and reward (delayed reinforcement) also slows down the learning process.

Gradient Descent Learning

Gradient descent is a *supervised learning* algorithm. It takes a set of training examples D = {d1, d2, d3, ...} where each training example consists of an input vector (x1, x2, ...) along with a desired output t. Gradient descent can be used to minimise the error between the outputs of a *weighted linear sum* over the input values, z, and the desired output.

It's common for an agent's evaluation function to be made of a weighted linear sum of *features* of a state, $z = V(s) = \Sigma w_i f_i(s)$. In this context, gradient descent can be used to tune the weights of the evaluation function, in an attempt to make it closely approximate the *true minimax utility* of the input state $t = U(s)$ (training examples can be found by conducting a minimax search before game play, when there's lots of time to do so).

Gradient descent works by minimising an 'error function' computed over all examples as, $E = 1/2\Sigma(t-z)^2$, by moving in the steepest downhill direction for each weight as given by the *weight update rule*, $w_i = w_i - \frac{\partial E}{\partial w_i}$ (or $\vec{w} = \vec{w} - \nabla E(\vec{w})$ for those who speak vector calculus). The weight update rule has the effect of moving the weight vector in the direction of steepest descent, by an amount proportional to its gradient's magnitude (so, the steeper we are descending, the further we will step!).

In practice, we'll need a way to actually compute this gradient! To do this, we can use the chain rule to find each dE/dwi. Further, we can apply this for each training example in the set independently. For each training example;

$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial z}(\frac{1}{2}(t-z)^2) \times \frac{\partial x}{\partial w_i} = (z-t)\frac{\partial z}{\partial w_i}$$

When each z is a weighted linear sum of features, $\frac{\partial z}{\partial w_i} = f_i(s)$. Therefore, the weight update rule simplifies to $w_i = w_i - (z-t)f_i(s)$ or, summing over all states s in the training set, $w_i = w_i - \Sigma_s(z(s) - t(s))f_i(s)$.

To control the behaviour of each step, we typically include an extra parameter, η, to scale the gradient used for updating. The formula becomes $\vec{w} = \vec{w} - \eta\nabla E(\vec{w})$. η is called the *learning rate* and a typical value might be 0.1. It's also possible to apply simulated annealing on the learning rate, reducing it towards 0 over a large series of examples, as much like hill climbing, gradient descent is susceptible to converging on local minima that are not actually global minima!

Temporal Difference Learning

Temporal difference learning is a *reinforcement learning* method. Unlike book learning, the agent's reward is not delayed until a terminal state is reached, Instead, subsequent predictions are used to determine an intermediate reward. While subsequent predictions may not correctly predict the eventual reward,

either, they are likely to represent a 'better guess' based on the availability of more information.

Temporal difference learning algorithms include a parameter λ (the 'trace decay' parameter) that determines the 'focus' of weight adjustment. Higher values lead to longer lasting 'traces'; a larger proportion of credit from a reward can be given to more distant states/actions. λ=0 causes weights to be adjusted so as to match predictions to subsequent predictions, and λ=1 causes weight adjustment towards the final true reward. Values between 0 and 1 interpolate between these two extremes. Over time, successive adjustments should cause the predictions to match the true reward, however higher values can help for situations where an initial evaluation function is not very reliable to begin with. It's possible that dynamically adjusting λ can also be beneficial.

In the context of game search, regular TD(λ) would look one ply ahead, choose the best move according to its evaluation function (based on its current weight parameters), and then execute that move, and repeat. At the end of the game, it would then apply the TD(λ) weight update rule to minimise the differences in reward predictions between search layers (on the grounds that the temporal difference of the theoretical 'correct' evaluation function are 0, and that's what we're trying to approximate).

However, in many games, looking one ply ahead does not provide enough information to make successful moves. One variant of TD(λ) addresses this by instead allowing the play to be guided by a deeper minimax search, but still applying the TD(λ) update to the sequence of game states occurring during the game. The only issue with this is that the deep evaluation functions may not always be a differentiable function of their weight parameters for all values of their weight parameters (though anomalies are rare in practice and can be dealt with).

Another variant of TD(λ) called TDLeaf(λ) addresses the issue of shallow searching by applying temporal difference to the *leaves* chosen as part of a regular, deep minimax search, rather than the roots of those searches. TD(λ) is applied to the sequence of leaves chosen rather than to the sequence of game states actually encountered. To do this, we make use of the following definitions:

- $eval(s, \vec{w})$ is the evaluation function for state s with parameter vector $\vec{w}$
- $s_1, \ldots, s_N$ are the N states that occur during the game
- s_i^l is the best leaf state found using minimax search starting from s_i using a maximum depth l
- $r(s_N)$ is the eventual reward based on the outcome of a game (a function of the final state s_N); typically chosen to be discrete e.g. win = +1, tie = 0, lose = -1
- $r(s, w) = \tanh(eval(s, w)$ the evaluation of state s converted to a reward score (taking advantage of the properties of *tanh* to 'squash' evaluations into the range (-1, 1)) (note $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$)

With these definitions, we can derive the TDLeaf(λ) version of the TD(λ) update procedure:

- for each value of i in (1, 2, ..., N-1)
  - compute the temporal difference between successive states

$$d_i = r(s^l_{i+1}, \vec{w}) - r(s^l_i, \vec{w})$$

  - update each weight parameter w_j according to the weight update rule

$$w_j = w_j + \eta \Sigma_{i=1}^{N-1} \frac{\partial r(s^l_i, \vec{w})}{\partial w_j} [\Sigma_{m=1}^{N-1} \lambda^{m-1} d_m]$$

    - η is the learning rate, as for gradient descent search
    - λ is the trace decay, as described for TD(λ) above
    - l is the fixed search depth used for minimax search

Game Learning Environments

Examples for use in machine learning algorithms applied to games can come from many sources,

- Learning from labelled examples
  - Though this requires a large source of labelled examples, which can be hard to come by! Especially if you are trying to design an agent for a task where you do not know the best actions yourself.
- Learning by playing a skilled opponent
  - Can be very effective
- Learning by playing against random moves
- Learning by playing against oneself
  - Good for stochastic games but tends to lead to lots of similar games for deterministic game (note this is only a problem if these games aren't representative of realistic play) (in the paper the FICS-trained TDLeaf(λ) program smashed the self-play trained version 89 to 11 after the self-play version had twice as many training games)

Resources:

- Baxter, J; Tridgell, A; Weaver, L; 1998; TDLeaf(λ): Combining Temporal Difference Learning with Game-Tree Search

Constraint Satisfaction Problems

Constraint Satisfaction Problems (CSPs) involve thinking of states using a factored representation, and require searching through a state space to find a state that satisfies given constraints. A CSP has three components,

- X a set of variables $\{X_1, X_2, ..., X_n\}$
- D a set of domains, one for each variable $\{D_1, D_2, ..., D_n\}$, each of which is a set of values $D_i = \{v_1, v_2, ...v_m\}$ representing the possible values for X_i
- C a set of constraints, in the form $(variables, relation)$ where $variables$ is the set of variables which participate in the constraint and $relation$ is a relation defining allowable values for those variables

The goal of a CSP is then to find a *complete assignment* (all variables have an assigned value, c.f. *partial assignment*) which is also *consistent* (satisfies all constraints).

By using this formulation we gain access to a collection of *general-purpose algorithms* augmented by a myriad of *domain-inspecific heuristics* which can help us solve our CSP. The approaches we can use will further depend on the characteristics of our domains and constraints. + Domains can be *discrete + finite*, *discrete + infinite* or *continuous*. * Finite discrete domains impose a bound on the number of possible assignments ($\Pi_{D_i \in D}|D_i|$) and include examples like Boolean Satisfiability * Infinite discrete domains (such as the set of integers or strings) it becomes impossible to express constraints as lists of valid values, and so we need a constraint language to express relations - linear constraints are solvable, but non-linear constraints are undecidable * Continuous domains suffer the same issues, and with linear constraints, can be solved in polynomial time using *Linear Programming* + Constraints can be *unary*, *binary*, or *global/higher-order* * Unary constraints involve a single variable * Binary constraints involve two variables * Global/Higher-order constraints involve 3 or more variables, a commonly-occurring example is the 'Alldiff' constraint which says that none of the variables participating in the constraint can share a value. + *Soft* constraints are also possible, such as preferences for certain values over others. This gives rise to *Constrained optimisation problems*.

The constraints in a CSP form a *constraint graph* where constraints and variables are represented as nodes (we can use squares for constraints and circles for variables). Constraints are linked by edges to the variables they involve. This graph can be simplified where binary constraints relate two variables by removing the constraint node and showing only an edge between the two variables. It's also worth noting that any global constraints can be 'expanded' into an equivalent set of binary constraints, and so in theory every CSP could be expressed as a graph with only unary and binary constraints and variable nodes, and no constraint nodes.

Solving with Standard Search

A naive approach to solving a CSP involves formulating it as a search problem, where the initial state is an empty assignment of values to variables, the successor function involves assigning values to an unassigned variable (which does not cause a conflict with any previously assigned variable), and a goal test that simply checks if the current state is a complete assignment!

If the number of variables in the CSP is n, then + every solution is going to appear at search-depth n + there will be no states with $depth > n$ + the state-space is guaranteed to be acyclic (due to the successor function) Under these assumptions, DFS is complete (and 'optimal' doesn't really have a meaning for CSPs since pat cost is irrelevant), so we can use DFS to search for a solution!

At each layer of the search, the branching factor is determined by the remaining number of variables and the size of each of their domains. For simplicity, lets assume all variables have a domain of uniform size d. Then, at depth l, $b = (n - l)d$, and there are up to $\frac{n!}{(n-l)!} \times d^l$ nodes. At the nth layer, that's up to $n! \times d^n$ leaves, even though there are only d^n possible complete assignments!

Though complete, this search is going to be nightmarishly inefficient, and so we're going to need something better!

Solving with Backtracking Search

One option is to continue with a search formulation but to tweak it to take advantage of the fact that variable assignments are *commutative* (that is, assigning X=1 and then Y=2 is the same as first assigning Y=2 and then X=1). Our naive formulation ignores this, and its successor function generates all legal assignments for *all remaining variables* at each layer of the search.

Backtracking search avoids this by choosing a *single* variable to assign next at each search step. By *itself*, this reduces the worst case time complexity of the technique to $O(d^n)$ (under our previous definitions and assumptions).

```
function BACKTRACKING-SEARCH(csp):
    assignments <-- empty set
    return RECURSIVE-BACKTRACKING-SEARCH(csp, assignments)

function RECURSVE-BACKTRACKING-SEARCH(csp, assignments):

    if assignments is complete:
        return assignments

    next-variable <-- SELECT-UNASSIGNED-VARIABLE(csp, assignments)
    for each value v in ORDERED-DOMAIN(next-variable, csp, assignments):
        if v is consistent with assignment:
```

```
            add {next-variable = v} to assignment
            result <-- RECURSIVE-BACKTRACKING-SEARCH(csp, assignments)

            // can perform dead-end checking here

            if result != FAILURE:
                return result
            remove {next-variable = v}

    return FAILURE
```

Further to that, there are a host of different *domain-inspecific heuristics* which we can apply to the *choice of variable* (`SELECT-UNASSIGNED-VARIABLE`)and also the *choice of value* (`ORDERED-DOMAIN`) from that variable's domain. Moreover, we can do some extra housekeeping during our search to quit 'early' if we're on a search path that cannot lead us to a consistent, complete assignment (a 'dead-end').

Minimum Remaining Values (MRV) and Degree Heuristics

The minimum remaining values heuristic says that when choosing the next variable to expand on our backtracking search, we should choose the variable with the smallest number of legal values remaining in its domain.

Often (for example at the beginning of a search) there will be multiple variables with the same, minimum number of remaining values in their domains. In this situation, we can use the degree heuristic, which says to break ties among MRV variables by choosing the variable that imposes the most constraints on remaining, unassigned variables (the one with the highest degree in a binary constraint network, not counting constraints with assigned variables).

Together, these heuristics represents starting with the 'hardest' variables to satisfy. Dealing with these 'controversial' assignments first can help us to identify dead-ends as easily as possible, and streamline our search by not wasting time when we are eventually going to backtrack anyway!

Least Constraining Value (LCV)

The least constraining value heuristic says that once we have selected a variable for assignment, we should choose to attempt to assign it values that rule out the fewest values in the domains of remaining, unassigned variables.

These values correspond to the assignments that are likely to be the most promising in terms of arriving at a satisfying the overall CSP. Once we have selected a variable for assignment, there's no point continuing to make the problem hard for ourselves by trying values that are *likely* to lead to dead ends. It's a much better idea to try the most promising assignments in the hope that they lead to a consistent, complete assignment.

Forward Checking

We can backtrack before we explicitly run into a dead-end in backtracking search by keeping track of the remaining legal values in the domains of our unassigned variables (we are probably already doing this to use MRV, in fact with MRV forward checking happens automatically anyway). If a variable ever runs out of remaining values, there's no point continuing to assign values until we select it, we might as well quit then and there!

Forward checking does this, backtracking early whenever it notices a partial assignment has rendered the problem unsolvable.

Constraint Propagation through Arc Consistency

Forward checking will not notice all inconsistencies as soon as they manifest, and it actually may be possible to recognise a dead-end *even earlier*! We can achieve this using *constraint propagation* (and we get a few other cool benefits, too). Constraint propagation is the idea of enforcing constraints locally and updating the legal domains of variables in response to these constraints. It can be applied repeatedly during search as selections are made, or it can be used as a 'preprocessing' step to simplify the search space before beginning a search.

One commonly used constraint propagation method is *arc consistency*. A variable X is *arc consistent* with respect to another variable Y if for every value in X's domain, there is at least one corresponding value in Y's domain that satisfies all binary constraints between the two variables. If this is not the case, then offending values can be *removed from X's domain*, without effecting the solution to the CSP (they cannot be part of any solutions that may exist, as there would be no way for assignments involving these values to satisfy all of the constraints!). A network of CSP variables is arc consistent if *each pair* of variables is arc consistent.

AC-3 is an algorithm for reducing a constraint network to an equivalent, arc consistent sub network. It basically repeatedly enforces arc-consistency between each pair of variables, making sure to note when changes to their domains demand that they must be checked again. If we cannot produce an arc consistent sub network, then the CSP is unsolvable and we can abandon the search. Here's some pseudo-code for AC-3,

```
function AC-3(csp):

    queue <-- a new queue with all the arcs in csp

    while queue is not empty:
        (A, B) <-- remove an arc from queue
        if REVISE(A, B):
            if A's domain is empty:
```

```
            return FAILURE // csp is inconsistent, no solution

        for each C in A's neighbours (except B)
            add arc (C, A) to queue

return SUCCESS // csp is now arc-consistent (with revised domains)

subroutine REVISE(A, B):
    revised <-- false
    for each value a in A's domain
        if no value b in B's domain allows (a, b) to satisfy the constraints between A a
            remove a from A's domain
            revised <-- true
    return revised
```

AC-3 runs in $O(cd^3)$ in the worse case, where c is the number of binary constraints in the CSP (c is O(nˆ2)$ if there are never multiple constraints between two variables). This is because each arc can only be inserted into the queue at most d times (as its second variable's domain has only that many values to potentially remove), for a total of cd insertions into the queue. Each arc in the queue can be checked for consistency in dˆ2 time, yielding the result of $O(cd^3)$ overall. Another version of the algorithm, AC-4, runs in worst-case $O(cd^2)$ time, however it performs slower on average cases.

Arc consistency can be generalised to handling constraints with more than two values (*generalised arc consistency*/*hyperarc consistency*), and also further generalised to all groups of k variables (k-*consistency*). For example, 1-consistency is 'node consistency' (each variable is consistent with its unary constraints), 2-consistency is arc consistency, 3-consistency is 'path consistency' with binary constraints, and so on. If n is the number of variables, then having n-consistency would actually tell us that there is a solution to the CSP! However, the general algorithm for showing that a constraint network is k-consistent requires time and space exponential in k! So, it's necessary to find an empirical balance between consistency-checking to potentially shrink the search space, and searching.

Solving with Local Search

Since the path we take during a CSP search is not important (we're only after the complete state, final assignment) we can use a complete-state formulation and *local search* techniques discussed earlier, beginning with any complete assignment and changing one variable assignment at a time to transition to different complete states, aiming to reach a consistent, complete state.

An appropriate heuristic is the 'min-conflicts' heuristic. This is a simple count of the number of constraints that are violated by a complete state. Our goal is a complete state with 0 violated constraints, and often we can achieve that

by 'hill-climbing' and taking steps to reduce the number of violated states and reach 0, a minimum.

Local search for CSPs is subject to the same issues as normal, namely local minima that are not global minima and areas in the search space that form a pateau. However, similar techniques can be applied to improve its performance.

Overall, local search performs very well, with high probability and very quickly in practice *except* in the case that |C|/|X| is near a 'critical ratio'. Nobody will tell me *what* that critical ratio is, but whatever.

Other enhancements to local search for CSPs involve forming a more informed heuristic by weighting differen constraints differently in the count, and also including strategies for navigating plateau regions of the state space.

Taking Advantage of Problem Structure

Sometimes we can take advantage of the *structure* of a constraint network and

For example, if the constrain network is comprised of multiple, discnnected subnetworks (groups of variables in which no variable is connected to any other) then these subnetworks actually form smaller, independent 'sub-problems'. There is no need to search for solutions as part of the same exponential search tree, and a huge complexity factor can be saved by solving them individually and then combining the independent solutions. For example, if a CSP with 80 variables, each with a domain of size 2, a CSP search has 2^{80} nodes in the worst case, which would take 4 *billion years* to compute at 10 million nodes per second. In contrast, if the CSP was found to actually consist of four independent subproblems each of 20 variables, then this would have 4×2^{20} nodes in the worse-case, which would take 0.4 *seconds* at the same rate.

Further, *tree-structured* constraint networks (with no constraint cycles) can be solved by a polynomial-time algorithm with worst-case $O(n \times d^2)$ time complexity. This algorithm has three steps. First, choose one variable / node to be the 'root', and linearlise the graph (this turns it into a topologically sorted DAG, by arbitrarily assigning a direction to each constraint, and takes $O(n)$ times). Next, make the network *directed arc consistent* (DAC): a new property in which each node in a topological sort of a constraint graph must be arc consistent with respect to all subsequent nodes. There are $n - 1$ arcs in an n-node tree, and resolving each arc takes $O(d^2)$ time, and so this takes the $O(n \times d^2)$ time. Finally, we can just walk down our DAC DAG and assign each variable any remaining legal value (which will always exist as it's DAC). The final step is also $O(n)$, and so overall this algorithm is $O(n \times d^2)$, as promised!

The advantages of tree-structured CSPs can even be utilised by CSPs with non-tree-structured constraint networks! As long as they are *nearly* tree structured, and a small set of nodes can be removed to leave behind a tree-structured network. The set of nodes which is removed (the *cutset*, size c) can have all assignments

enumerated (there are only d^c of them), and *conditioning* can be performed (pruning the domains of the neighbour variables in response to the assignment) and then the DAG+DAC algorithm can be attempted for each assignment, until a solution is found. This reduces the complexity to $O(d^c \times (n-c)d^2)$, which is great when c is small!

Summary

Recognising a CSP

Backtracking over naive DFS + Variable-ordering and value-selection heuristics + Forward Checking and Constraint Propagation (through Arc Consistency)

(Iterative) Local Search + min-conflicts heuristic (usually effective in practice!)

Analysing Problem Structure (independent sub problems, plus tree structure and cut sets)

Skills expected: + model a given problem as a CSP + demonstrate operation of CSP search algorithms + discuss and evaluate the properties of different CS techniques

Uncertainty

Chain rule

Probabilistic Reasoning

Bayes

Making Complex Decisions

Auctions

Robotics

IncreMENTAL bayes