AI Notes

Rafael Blecher - z3380867

June 18, 2014

Contents

Part I

Week 1

Chapter 1

Environment Types

An *agent* is a function from *percept sequences* to actions. Ideally, an agent is rational, and picks actions which maximise the performance measure. This performance measure can be evaluated empirically, and sometimes analysed empirically.

1.1 PEAS: Performance, Environment, Actuators, Sensors

1.1.1 Performance

A function that measures the quality of the work done by the agent. Examples include Safe, Fast, Legal, Comfortable trip, Maximize profits

1.1.2 Environment

The environment that the agent operates in is generally described by the following properties:

Fully or partially observable

The environment is fully observable if the agent's sensors give it access to the complete state of the environment at each point in time. Note: the sensors only need to detect all RELEVANT information to the task. Relevance depends on the Performance Measure. Causes for partially observable environments are noise, inaccurate sensors, or missing sensors. Example of partially observable environments: a self-driving car cannot tell what other drivers are thinking.

Deterministic or stochastic or strategic

The environment is deterministic if the next state of the environment is completely determined by the current state and the action executed by the agent. We tend to define whether or not an environment is deterministic by considering the environment from the agent's point of view. Interesting to note is that if an environment is both deterministic and fully observable, the agent does not - in theory, at least - have to be concerned with uncertainty. If the environment

is deterministic except for the strategic actions of other agents, we say that the environment is strategic.

Episodic or sequential

Episodic environments divide the agent's experience into atomic episodes. No episode affects the state of another episode, and so every episode is a self-contained experience of [perceive, act]. An example of an episodic environment is checking for defective parts on as assembly line. The agent only considers the current part, and what the agent perceives, and its subsequent action, have no effect on past or future results.

In a sequential environment, the current action can affect all future decisions. Examples of sequential environments include playing chess, and self-driving cars. They are far more complicated than episodic environments because future moves must be considered before making a decision for the current state.

Discrete or continuous

Discrete/continuous can be understood in relation to state, time, and percepts and actions of the agent. Chess is a discrete-state environment, as there are a discrete number of states of a chess game. Chess also has a discrete set of percepts and actions. Taxi driving is both state- and time-continuous: the location of the taxi and other vehicles flow smoothly over time. Taxi-driving actions are also continuous (think of steering angles).

Sometimes it is difficult to ascertain whether or not an environment is discrete. Input from a digital camera is technically discrete: there are $n * m$ pixels, each of which can take any value in the RGB range 000000 to FFFFFF. On the other hand, the input depicts a real-life image, which forces pixels to be displayed with different intensities and hence feels distinctly continuous.

Single-agent or multi-agent

In a single-agent environment, the one agent does not have to account for the actions of other agents, and can concern itself only with how it interacts with the environment.

An environment is described as competitive if it is multi-agent and each agent is in competition with every other agent. The environment is described as cooperative if the agents are working together to achieve some goal. N.B.: competitive and cooperative are not mutually exclusive.

Static or Dynamic

An environment is described as static if it does not change while the agent is deliberating. Otherwise, the environment is dynamic.

Known or Unknown

The environment is *known* if the rules of the environment are known. Otherwise, the environment is unknown. Examples of rules are game rules, and physics/dynamics of the environment.

Simulated, or Situated

An environment is simulated if a separate program is used to simulate the environment, feed percepts to agents, evaluate performance, etc.
See Wikipedia: Artificial intelligence, situated approach for more.

Situated or Embodied? There is an important distinction to make between situatedness and embodiment.

Situatedness describes an agent being situated in a world - they are only concerned with the "here and now" of the world, rather than abstract descriptions. The state of the environment directly influences the behaviour of the agent system.

Embodiment describes the AI having a body (a robot), which experiences the world directly. The actions are part of a dynamic with the world, and an agent"s actions have immediate feedback on the robot"s own percepts.

An airline reservation system is situated but not embodied; it deals with requets and its responses change as the database changes.

A spray-painting robot is embodied but not situated; it does not perceive anything about the object presented to it, it just goes through a pre-programmed series of actions. The robot has physical extent and must correct for its interaction with gravity, etc.

The Real World

Using these definitions, we describe our world as partially observable, stochastic, sequential, continuous, multi-agent, dynamic, unknown, and situated.

1.1.3 Actuators

Actuators are the set of devices that the agent can use to perform actions.

1.1.4 Sensors

Sensors allow the agent to collect the percept sequence that will be used to decide on the next action.

1.1.5 Applying PEAS to Wumpus World

Performance Measure

Return with Gold +1000, Death -1000, -1 per step, -10 for using the arrow

Actuators

Left, right, forward, grab, shoot

Sensors

Breeze, Glitter, Stench

1.1.6 Applying PEAS to a self-driving car

Performance measure

safety, reach destination, maximise profits, obey laws, passenger comfort

Environment

city streets, freeways, traffic, pedestrians, weather, customers

Actuators

steer, accelerate, brake, horn, speak/display

Sensors

video, accelerometers, guages, engine sensors, keyboard, GPS

Part II

Week 2

Chapter 2

Agent Types

2.1 Reactive Agent

Reactive agents are quite simple; steps are repeated in a very simple procedure:

1. Perceive environment
2. Act

Reactive agents choose the next action to perform based only on what the currently perceive. The action is based on a simple set of rules (called a *policy*) which are easy to apply.

Reactive agents are sometimes called "simple reflex agents", despite the fact that they can actually perform fairly sophisticated tasks (for example, simulated hockey can be performed by a reactive agent.)

2.1.1 Limitations of Reactive Agents

A reactive agent only knows about what it currently perceives and in many cases, you need to remember previous percepts to make intelligent decisions. Reactive agents may also perform the same action over and over, if what they perceive never changes.

2.2 Model-Based Agent

A model-based agent is slightly more complicated than a reactive agent. A model-based agent includes a world model, which represents the important parts of the environment. The agent adjusts its world model based on its percepts, and the agent's actions are informed by both the world model and its current percept.

2.2.1 Advantages of Model-Based Agents

A model-based agent can remember past percepts.

2.2.2 Limitations of Model-Based Agents

Sometimes we need to plan several steps into the future. Hence, a model-based agent will perform badly in the following cases:

Searching several moves ahead Chess, Rubik's Cube

Complex tasks with many steps Cooking a meal, assembling a watch

Logical reasoning to achieve a goal travel to New York

2.3 Planning Agent

A planning agent adds a planning step to a model-based agent. The world model can be a transition table, a dynamical system, a parametric model, a knowledge base, etc.

The planning step can be a state-based search, a simulation, logical inference, or a set of goals. The action of the planning agent is informed by the planning step.

A planning agent has the capacity to reason about future states. However, sometimes agents appear to be planning but are really just reactive, applying rules which are hard-coded or learned. These agents appear intelligent, but are not flexible in adapting to new situations.

2.4 Learning Agent

A learning agent adds a Learning step to each part of the planning agent. Statistical learning is applied to Perception, reinforcement learinng is applied to the Action, Bayesian learning is applied to the world model, and inference learning is applied to the planning step.

Learning is a set of techniques for improving the existing modules in the planning agent, rather than being a different module in the agent.

Learning is necessary because it may be difficult for a human to design all aspects of a system by hand, and because the agent may have to adapt to new situations without being re-programmed by a human.

Application of learned behaviour is distinct from the learning process. For example, the policy for a simulated hockey player took sevel days of computation to derive, but after that process, can be applied in real-time.

Chapter 3

Solving Problems by Searching

3.1 Motivation

Reactive and model-based agents choose their actions based on what they currently perceive, or what the have perceived in the recent past.

A planning agent can use search techniques to plan several steps ahead in order to achieve its goal(s).

There are two classes of search strategies:

1. Uninformed search strategies can only distinguish goal states from non-goal states
2. Informed search strategies use heuristics to try to get closer to the goal

3.2 Romania Example

Now, let's apply the search strategy to the example covered in lectures; the map of Romania. We are touring Romania, and we are currently in Arad. Our flight leaves tomorrow from Bucharest.

Formulate goal be in Bucharest on time.

Specify task

- states: various cities
- operators or actions (transitions between states): drive between cities

Find solution (action sequences) sequence of cities to drive through

Execute drive through all the cities given by the solution.

3.2.1 Single-State Task Specification

A task is specified by states and actions.

initial state At Arad

state space other cities

actions or operators (or successor function S(x)) Arad $\rightarrow$ Zerind

goal test check if a state is a goal state. Can be explicit ("at Bucharest") or implicit ($NoDirt(x)$)

path cost sum of distances from Arad to Bucharest

total cost = search cost + path cost = offline cost + online cost

A *solution* is a state-action sequence (initial to goal state)

3.2.2 Choosing states and actions

The real world is complex; the state space must be abstracted for problem-solving.

An abstract state is a set of real states.

An abstract action is a complex combination of real actions. For example, "Arad $\rightarrow$ Zerind" describes a complex set of possible routes, detours, rest stops, etc. For guaranteed realisability, any real state must get to some real state (for example, Arad must get to Zerind).

An abstract solution is a set of real paths that are solutions in the real world.

3.2.3 Problem types

Toy problems have a concise, exact description

Real-world problems don't have a single agreed description.

3.2.4 Applying task specifications

The 8-Puzzle

states integer locations of tiles (ignore intermediate positions)

operators move blank left, right, up, down (ignore unjamming, etc.)

goal test = goal state (given)

path cost 1 per move

Rubik's Cube

TODO NOTHING HERE. Any takers?

3.3 Path Search Algorithms

A *search* finds state-action sequences that lead to desirable states. Search is a function with prototype

`solutionType` *search*(*task*)

The basic idea is that a search is an offline, simulated exploration of the state space by generating successors of already-explored states (by "expanding" them).

To generate an action sequence:

- Start with the initial state
- Test if it is a goal state
- Expand one of the states
- If there are multiple possibilities, make a choice
- Procedure: choosing, testing, and expanding until a solution is found or there are no more states to expand

Generating an action sequence can be thought of as building a *search tree*. A search tree is superimposed over the state space. The root of the tree is the initial state, and the leaves of the tree are states that are not expanded, or that generated no new nodes. The state space is not the same as the search tree; in the Romania example there are only 20 states (20 cities) but there are infinitely many paths.

A node data structure is composed of five components:

1. Corresponding state
2. Parent node: the node which generated the current node
3. Operator that was applied to generate the current node
4. Depth: the number of nodes from the root to the current node
5. Path cost

Nodes are distinct from states: a state is [a representation of] a physical configuration. A node is a data structure that makes up part of a search tree. States do not have parents, children, depth, or path cost.
Note: multiple nodes can contain the same state.

3.3.1 Data Structures for Search Trees

The *frontier* is the collection of nodes that have not yet been expanded. It can be implemented as a priority queue with the following operations:

Make-Queue(Items) creates queue with given items

Boolean Empty(Queue) returns TRUE if no items in queue

Remove-Front(Queue) removes the item at the front of the queue and returns it

Queueing-Fn(Items, Queue) inserts new items into the queue.

3.3.2 Search Strategies

A strategy is defined by picking the order of node expansion. Strategies are evaluated along the following dimensions:

completeness does it always find a solution if one exists?

time complexity number of nodes generated/expanded

space complexity maximum number of nodes in memory

optimality does it always find a least-cost solution?

Time and space complexity are measured in terms of

b maximum branching factor of the search tree

d depth of the least-cost solution

m maximum depth of the state space (may be ∞)

3.3.3 Approaches to comparing algorithms

Benchmarking

Run two algorithms on a computer and measure speed. This is not a great way of comparing algorithms, as it depends on implementation, compiler, computer, data, network, ...

Analysis of algorithms

We tend to use Big-O notation to describe algorithms. $\mathrm{T}(n) \in \mathrm{O}(f(n))$ means $\exists k > 0 \exists n_0 : \forall n > n_0 \mathrm{T}(n) \leq kf(n)$ Here, n is the input size and $\mathrm{T}(n)$ is the total number of steps in the algorithm. In words, this means that $f(n)$ describes the limiting behaviour of $\mathrm{T}(n)$ when n tends towards some value, or infinity. For more, see Wikipedia: Big O notation.

Note: alternatives to $\mathrm{T(n)} \in \mathrm{O}(f(n))$ are $\mathrm{T(n)} = \mathrm{O}(f(n))$ and T(n) is $\mathrm{O}(f(n))$.

Big O notation:

- is independent of the implementation, compiler, etc.
- provides asymptotic analysis; for large n, $\mathrm{O}(n)$ is better than $\mathrm{O}(n^2)$
- abstracts over contant factors. i.e. $\mathrm{T}(100n+1000)$ is better than $\mathrm{T}(n^2+1)$ only for $n > 110$.
- is a good compromise between precision and ease of analysis.

3.3.4 Uninformed Search Strategies

Uninformed (or "blind") search strategies use only the information available in the problem definition (can only distinguish a goal from a non-goal state). Examples include

- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative Deepening search

Strategies are distinguished by the order in which nodes are expanded.

Breadth-first search

All nodes are expanded at a given depth in the tree before any nodes at the next level are expanded.

Expand root first, then all nodes generated by the root, then all nodes generated by those nodes, ...

Expand shallowest unexpanded node

Implementation: put newly generated nodes at the end of the queue

Finds the shallowest goal first.

Breadth-first search is complete (if b is finite the shallowest goal is at fixed depth d and will be found before any deeper nodes are expanded).

Space complexity: $b + b^2 + b^3 + \ldots + b^d + (b^{d+1} - b) = O(b^{d+1})$

Time complexity: $O(b^{d+1})$

Is optimal if all actions have the same cost.

Space is the biggest problem with BFS; it grows exponentially with depth.

Uniform-Cost Search

Expand root node first, then expand least-cost unexpanded node first.

Implementation: enqueue nodes in order of increasing path cost.

Reduces to BFS when all actions have same cost.

Finds the cheapest goal provided the path cost is monotonically increasing along each path (i.e. no negative-cost paths)

Uniform-cost search is complete if b is finite and step cost $\geq \epsilon$ with $\epsilon > 0$

Time complexity: $O(b^{\lceil C*/\epsilon \rceil})$ where $C*$ = cost of optimal solution, and assume that every action costs at least ϵ

Space complexity: $O(b^{\lceil C*/\epsilon \rceil})$ ($b^{\lceil C*/\epsilon \rceil} = b^d$ if all step costs are equal)

Is optimal.

Depth-First Search

Expands one of the nodes at the deepest level of the tree

Implementation: insert newly generated nodes at the front of the queue. Can be alternatively implemented by recursive function calls.

Not complete: fails in infinite-depth spaces, spaces with loops. Can be modified to avoid repeated states along path $\Longrightarrow$ complete in finite spaces.

Time complexity: $O(b^m)$ (terrible if m is much larger than d but if solutions are dense, may be much faster than BFS)

Space complexity: $O(bm)$ (linear space!)

Not optimal, can find suboptimal solutions first.

Depth-Limited Search

Expands nodes like DFS but imposes a cutoff on the maximum depth of path.

Complete? Yes (no infinite loops)

Time complexity: $1 + b^1 + b^2 + \ldots + b^{l-1} + b^l = O(b^l)$ where l is the depth limit

Space complexity: $O(bl)$ linear space similar to DFS

Not optimal, like DFS.

Also comes with the problem of picking a good limit.

Iterative Deepening Search

Tries to combine the memory benefits of DFS and optimality/completeness of BFS by doing a series of depth-limited searches to depth 1, 2, 3, ...

Early states will be expanded multiple times, but that might not matter too much because most of the nodes are near the leaves.

Complete.

Time complexity: nodes at the bottom are expanded once, nodes at the next level are expanded twice, etc. $(d+1)b^0 + db^1 + (d-1)b^2 + ... + 2b^{d-1} + 1b^d = O(b^d)$. For example, given $b = 10$, $d = 5$:

depth-limited: $1 + 10 + 100 + 1000 + 10000 + 100000 = 111111$

iterative-deepening: $6 + 50 + 400 + 3000 + 20000 + 100000 = 123456$

only about 11% more nodes (for $b = 10$).

Space complexity: $O(bd)$

Is optimal if step costs are identical.

Bidirectional Search

The idea of bidirectional search is to search both forward from the initial state and backward from the goal state, and stop when the two searches meet in the middle.

We need an efficient way to check if a new node already exists in the other half of the search. The complexity analysis assumes that this can be done in constant time, using a hash table.

Assuming branch factor b in both directions and that there is a solution at depth d. Then bidirectional search finds a solution in $O(2b^{d/2}) = O(b^{d/2})$ time steps.

However, there are problems with bidirectional search. Searching backwards means generating predecessors from the goal, which may be difficult. There can be several goals (e.g. checkmate positions in Chess). Space complexity is $O(b^{d/2})$ because the nodes of at least one half must be kept in memory.

Complexities of Uninformed Search Strategies

Search	BFS	Uniform-Cost Search	DFS	Depth-Limited Search	Iterative Deepening
Time complexity	$O(b^{d+1})$	$O(b^{\lceil C*/\epsilon \rceil})$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Space complexity	$O(b^{d+1})$	$O(b^{\lceil C*/\epsilon \rceil})$	$O(bm)$	$O(bl)$	$O(bd)$
Complete?	Yes[1]	Yes[2]	No	No	Yes[1]
Optimal?	Yes[3]	Yes	No	No	Yes[3]

3.3.5 Informed Search Strategies

Informed (or "heuristic") search strategies use task-specific knowledge. An example of task-specific knowledge: distance between two cities on the map of Romania. Informed search is more efficient than Uninformed search. Uninformed search systematically generates new states and tests them against the goal - informed search attempts to filter input, or order inputs in a way that the most likely candidate is tested before others.

[1] complete if b is finite
[2] complete if b is finite and step costs $\leq \epsilon$ with $\epsilon > 0$
[3] optimal if actions all have the same cost

Part III

Week 3

Chapter 4

Informed Search

4.1 Search Strategies

General search algorithm:

- Add initial state to queue
  - take node from front of queue
  - test if it is a goal state; if so, terminate
  - expand it (generate successor nodes and add them to the queue)

Search strategies are distinguished by the order in which new nodes are added to the queue of nodes awaiting expansion.

BFS and DFS treat all nodes equally; BFS adds new nodes to the end of the queue, DFS adds new nodes to the front of the queue.

Best First Search uses an evaluation function $f()$ to order the nodes in the queue. An example is Uniform Cost Search; $f(n) =$ cost $g(n)$ of path from root to node n.

Informed or heuristic search strategies incorporate into $f(n)$ an estimate of distance to goal. Greedy Search uses $f(n) =$ estimate $h(n)$ of cost from node n to goal. A* search uses $f(n) = g(n) + h(n)$.

4.2 Best-First Search

The Best-First Search family of algorithms have different evaluation functions $f(n)$. A key component of these algorithms is the heuristic function $h(n)$.

Heuristic function h: {Set of nodes} $\rightarrow R$. $h(n) =$ estimated cost of the cheapest path from the current node n to the goal node. Heuristic functions provide an estimate of solution cost.

4.2.1 Greedy Best-First Search

Greedy Best-First Search is a Best-First Search that selects the next node for expansion using the heuristic function as its evaluation function. i.e. $f(n) = h(n)$. $h(n) = 0 \iff n$ is a goal state. Greedy search minimises the estimated

cost to the goal; it expands whichever node n is estimated to be closest to the goal.

Greedy Best-First Search is not complete as it can get stuck in loops. It is complete is finite spaces with repeated-space checking.

Time complexity is $O(b^m)$ where m is the maximum depth in the search space.

Space complexity is $O(b^m)$ as all nodes are kept in memory.

Greedy search is not optimal. Greedy search has the same problems as DFS, but a good heuristic can reduce time and memory costs.

4.2.2 Straight Line Distance as a Heuristic

$h_{SLD}(n)$ = straight-line distance between n and the goal node. If you know map coordinates of n and the goal, you can find the straight line distance using $\sqrt{(n_x - goal_x)^2 + (n_y - goal_y)^2}$

4.2.3 A* Search

A* Search uses $f(n) = g(n) + h(n)$ where $g(n)$ is the cost from the initial node to node n, $h(n)$ is the estimated cost of the cheapest path from n to goal, and $f(n)$ is the estimated total cost of the cheapest solution through node n.

Greedy search minimises $h(n)$ (efficient but not optimal or complete), Uniform Cost Search minimises $g(n)$ (optimal and complete but not efficient). A* minimises $f(n) = g(n) + h(n)$ - the idea is to preserve the efficiency of Greedy without expanding paths that are already expensive. A* is both optimal and complete, provided that $h(n)$ is admissible (never overestimates the cost to reach the goal).

Specifically, a heuristic $h()$ is admissible if $\forall n, h(n) \leq h^*(n)$ where $h^*(n)$ is the true cost from n to the goal. If h is admissible then $f(n)$ never overestimates the actual cost of the best solution through n. $h_{SLD}()$ is admissible because the shortest path between two points is the straight line connecting them.

Proving Optimality of A* Search

Suppose a suboptimal goal node G_2 has been generated and is in the queue. Let n be the last unexpanded node on the shortest path to an optimal goal node G.

$$\begin{aligned} f(G_2) &= g(G_2) \text{ since } h(G_2) = 0 \\ &> g(G) \text{ since } G_2 \text{ is suboptimal} \\ &\geq f(n) \text{ since } h \text{ is admissible} \end{aligned}$$

Since $f(G_2) > f(n)$, A* will never select G_2 for expansion (although it may be generated and put into the queue). In words, A* will keep searching until there is no possibility of finding a shorter solution, even after finding a goal node.

A* is complete unless there are infinitely many nodes with $f \leq$ cost of solution. The time complexity is exponential in *relative error in (h * length of solution)*. All nodes are kept in memory, so space complexity is not great. A* is optimal, given that the heuristic is admissible.

4.2.4 Iterative Deepening A* Search

Iterative Deepening A* is a low-memory variant of A* which performs a series of DFSs, cutting off each search when the sum $f(n) = g(n) + h(n)$ exceeds some pre-defined threshold. The threshold is steadily increased with each successive search. Iterative Deepening A* is asymptotically as efficient as A* for domains where the number of states grows exponentially.

4.3 Heuristics

4.3.1 Dominance of Heuristics

If $h_2(n) \geq h_1(n)$ for all n (both admissible) then h_2 *dominates* h_1 and is better for search. The aim is therefore to make the heuristic $h()$ as large as possible without exceeding $h^*()$.

4.3.2 How to Find Heuristic Functions

Admissible heuristics can often be derived from the exact solution cost of a relaxed (simplified) version of the problem that has some constraints weakened or removed.

One can also use a *composite heuristic*; let $h_1, h_2, \ldots, h_m$ be admissible heuristics. The composite heuristic $h(n) = max(h_1(n), h_2(n), \ldots, h_m(n))$. h is trivially admissible, and more importantly, dominates $h_1, h_2, \ldots, h_m$.

Chapter 5

Reactive Agents

As mentioned in the beginning of chapter 2, reactive agents choose the next action based only on what they currently perceive, using a policy. They have no memory, and are unable to plan or logically reason. However, interesting behaviours can emerge from the simple rules in the policy.

5.1 History of Reactive Agents

5.1.1 Braitenberg Vehicles

Braitenberg showed how simple configurations of sensors and motors can lead to surprisingly sophisticated behaviour. The simplest Braitenberg vehicles have two wheels and two sensors. The sensors respond to light, and the response is inversely proportionate to distance from the light source.

The connections can be straight or crossed, and excitatory (+) or inhibitory. Combinations of these lead to four behaviours: hate, love, fear, and curiosity. Imagine a vehicle with a light source ahead of it to its right.

Hate: Crossed and excitatory

The vehicle will sense the light with its right-hand sensor first, causing its *left* wheel (the wires are crossed) to accelerate (excitatory). When the vehicle has turned so that the light source is directly ahead of it, the left and right sensors will be receiving the same input and the vehicle will travel in a straight line towards the light source.

The vehicle will travel faster and faster before ramming into the light source, and will then attempt to continue driving.

Love: Straight and inhibitory

The vehicle will sense the light with its right-hand sensor first, causing its right wheel to deccelerate. When the vehicle has turned so that the light source is directly ahead of it, the left and right sensors will be receiving the same input and the vehicle will travel in a straight line towards the light source.

The vehicle will slow until it is directly in front of the light source, when it stops.

Fear: Straight and excitatory

The vehicle will sense the light with its right-hand sensor first, causing its right wheel to accelerate. As the vehicle gets closer to the light source it will increase its speed, and will continue turning away from the light source and accelerating until it no longer senses the light.

Curiosity: Crossed and inhibitory

The vehicle will sense the light with its right-hand sensor first, causing its left wheel to deccelerate. The vehicle will approach the light source and slow until a certain point, when the vehicle will turn away from the light, picking up speed in the process.

5.1.2 The Swiss Robots

The Swiss Robots were given a simple set of rule that are used to clean up objects in an area. The rules are as follows:

- Normally, move forward
- If you detect an obstacle to the left or right, turn away from it
- If you detect an obstacle directly in front of you, move forward.

TODO THIS NEEDS COMMENTARY. HALP.

5.2 Behaviour-Based Robotics

Introduced by Rodney Brooks in the late 1980s, behaviour-based robotics was meant as a challenge to GOFAI (Good Old Fashioned AI). It had a few key points:

- robots should be based on insects rather than humans
- tasks like walking around and avoiding obstacles rather than playing Chess
- abandon traditional horizontal decomposition (Sense $\rightarrow$ Plan $\rightarrow$ Act)
- replace vertical decomposition (each layer can connect sensing to an action)

TODO INSERT IMAGES PLZ

5.2.1 Modern Perspective

Each layer in vertical composition is a behaviour. Low-level behaviours such as "avoid hitting things" are reactive, connecting sensors to actuators. Mid-level behaviours like "build maps" make use of a world model. High-level behaviours make use of a world map and planning.

Importantly, higher-level behaviour may take control from lower-level behaviour. e.g. if low-level behaviour has gotten stuck. Lower-level behaviour can also take control from higher-level behaviour, usually in an emergency (i.e. to

avoid getting burned, or falling down a staircase). This is similar to the way that humans' nervous systems have immediate responses to triggers such as heat and pain.

Part IV

Week 4

Chapter 6

Games

6.1 Why Games?

Games have "unpredictable" opponents ⇒ solution is a strategy.

Time limits ⇒ must rely on approximation (tradeoff between speed and accuracy).

Games have been a key driver of new techniques in CompSci and AI.

6.2 Types of Games

6.2.1 Discrete Games

Fully observable, deterministic (chess, checkers, go, othello)

Fully observable, stochastic (backgammon, monopoly)

Partially observable (bridge, poker, scrabble)

6.2.2 Continuous, embodied games

Robocup soccer, pool

6.3 Key Ideas

- Computer considers possible lines of play
- Algorithm for perfect play
- Finite horizon, approximate evaluation
- Machine learning to improve evaluation accuracy
- Pruning to allow deeper search

6.3.1 Minimax

Minimax provides perfect play for deterministic, perfect-information games. The idea is to choose the position with the highest minimax value = best achievable payoff against best play.

Minimax Algorithm

```
function minimax(node, depth)
   if node is a terminal node or depth = 0
      return heuristic value of node
   if we are to play at node
      let α = −∞
      foreach child of node
         let α = max(α, minimax(child, depth − 1))
      return α
   else // opponent is to play at node
      let β = ∞
      foreach child of node
         let β = min(β, minimax(child, depth − 1))
      return β
```

This assumes that all nodes are evaluated with respect to a fixed player (e.g. White in Chess).

Negamax formulation of Minimax Minimax can be simplified to an algorithm called Negamax if we assume that each node is evaluated with respect to the player whose turn it is to move.

```
function negamax(node, depth)
   if node is a terminal node or depth = 0
      return heuristic value of node from perspective of player whose
turn it is
   let α = −∞
   foreach child of node
      let α = max(α, −negamax(child, depth − 1))
   return α
```

Properties of Minimax Complete?
Optimal?
Time complexity?
Space complexity?

6.3.2 Reducing the Search Effort

For chess, $b \approx 35, m \approx 100$ for "reasonable" games $\Rightarrow$ exact solution is completely infeasible.

There are two ways to make the search feasible:

1. don't search to final position; use heuristic evaluation at the leaves
2. $\alpha - \beta$ pruning

Heuristic Evaluation for Chess

Material: Queen = 9, Rook = 5, Knight = Bishop = 3, Pawn=1

Position: some (fractional) score for a particular piece on a particular square

Interaction: some (fractional) score for one piece attacking another piece, etc.

The value of individual features can be determined by reinforcement learning.

Motivation for Pruning

Once we have seen one reply scary enough to convince us the move is really bad, we can abandon this move and search elsewhere.

$\alpha - \beta$ search algorithm

```
function alphabeta(node, depth, α, β, ourTurn)
   if node is a terminal node or depth = 0
      return heuristic value of node
   if ourTurn // we are to play at node
      foreach child of node
         let α = max(α, alphabeta(child, depth − 1, α, β, FALSE))
         if α ≥ β
            return α // β is pruned off
      return α
   else // opponent is to play at node
      foreach child of node
         let β = min(β, alphabeta(child, depth − 1, α, β, TRUE))
         if β ≥ α // α is pruned off
            return β
      return β
alphabeta(origin, depth, −∞, ∞, TRUE)
```

Negamax formulation of $\alpha - \beta$ search

```
function minimax(node, depth)
   return alphabeta(node, depth, −∞, ∞)

function alphabeta(node, depth, α, β)
   if node is terminal or depth = 0
      return heuristic value of node
      // from perspective of player whose turn it is to move
   foreach child of node
      let α = max(α, −alphabeta(child, depth − 1, −β, −α))
      if α ≥ β
         return α
   return α
```

Why is it called $\alpha - \beta$? α is the best value for us found so far, off the current path

β is the best value for opponent found so far, off the current path

If we find a move whose value exceeds α, pass this new value up the tree.

If the current node value exceeds β, it is "too good to be true", so we "prune off" the remaining children.

TODO CAN SOMEONE EXPLAIN THIS? how is it "too good to be true"? what does that mean? lecture slides 6 Games, page 28

Properties of $\alpha - \beta$ $\alpha - \beta$ pruning is guaranteed to give the same result as minimax, but speeds up the computation substantially.

Good move ordering improves effectiveness of pruning.

With perfect ordering, time complexity = $O(b^{m/2})$. To prove that a bad move is bad, we only need to consider one good reply, but to prove that a good move is good, we need to consider all replies.

This means that $\alpha - \beta$ can search twice as deep as plain minimax. An increase in search depth from 6 to 12 could change a very week player into a quite strong one.

6.4 Chess

Deep Blue defeated world champion Gary Kasparov in a six-game match in 1997.

Traditionally, computers played well in the opening (using a database) and in the endgame (by deep search DEFINITION FOR DEEP SEARCH PLZ) but humans could beat them in the middle game by "opening up" the board to increase the branching factor. Kasparov tried this, but Deep Blue was so fast that it was still able to in.

Some experts believe that Kasparov should have been able to defeat Deep Blue, but today chess programs stronger than Deep Blue are running on standard PCs (these programs rely on quiescent search, transposition tables, and pruning heuristics) and could definitely beat the strongest humans.

6.5 Checkers

Chinook failed to defeat human world champion Marion Tinsely prior to his death in 1994, but has beaten all subsequent human champions. Chinook used an endgame database defining perfect play for ass positions involving 8 or fewer pieces on the board - a total of 443,748,401,247 positions. This database has since been expanded to include all positions with 10 or fewer pieces (38 trillion positions).

In 2007, Jonathan Shaeffer released a new version of Chinook and published a roff that it will never lose, His proof method fills out the game tree incrementally, ignoring branches which are likely to be pruned. After many moths of computation, it eventually converges to a skeleton of the real (pruned) tree which is comprehensive enough to complete the proof.

6.6 Go

The branching factor for Go is greater than 300, and static board evaluation is difficult.

Traditional Go programs broke the board into regions and used pattern knowledge to explore each region.

Since 2006, new "Monte Carlo" players have developed using UCB search (TODO WHAT EVEN IS THIS?). A tree is built up stochastically. After a small number of moves, the rest of the game is played out randomly, using fast pattern matching to give prefence to "urgent" moves. Results of random playouts are used to update statistics on early positions.

Computers are now competitive with humans on 9x9 boards, but humans still have the advantage on 19x19 boards.

6.7 Stochastic Games

In stochastic games, chance is introduced by dice, card-shuffling, etc.

Expectimax is an adaptation of Minimax which also handles chance nodes:

if *node* is a chance node

    return average of values of successor nodes

Adaptations of $\alpha-\beta$ pruning are possible, provided the evaluation is bounded.

6.8 Partially Observable Games

Card games are partially observable because some of the opponent's cards are unknown.

This makes the problem very difficult, because some information is known to one player but not the other.

Typically, we can calculate a probability for each possible deal.

The idea is therefore to compute the minimax value of each action in each deal, then choose the action with the highest expected value over all deals.

GIB, the current best bridge program, approximates this idea by

1. generating 100 deals consistent with bidding informatio
2. picking the action that wins most tricks on average

Chapter 7

Motion Planning

7.1 Motion Planning Approaches

7.1.1 Unknown Environments (on-board sensors only)

Occupancy Grid

Divide environment into a Cartesian grid. For each square in the grid, maintain an estimate of the probability of an obstacle in that square.

Potential Field

Treat robot's configuration as a point in a potential field that combines attraction to the goal, and repulsion from objects. Very rapid computation, but can get stuck in local optima, thus failing to find a path.

Vector Field Histogram

Uses a continuously updated Cartesian histogram grid and a Polar histogram based on the current position/orientation of the robot. Candidate valleys are generated (contiguous sectors with low obstacle density). Candidate valleys are then seleced based on proximity to target direction.

7.1.2 Known Environments (overhead cameras)

Delaunay Triangulation

Applicable in situations where the environment is well mapped by overhead cameras (museums, shopping centres, robocup soccer field)

Add line segments between the closest points of obstacles, and sort them according to length in ascending order.

Do not add a segment that crosses an existing segment.

Prune arcs that are too small for the robot to traverse

A* Search can then be applied on the resulting graph.

Parameterized Cubic Splines

Assume that each path segment is of the form
$P(t) = \binom{P_x(t)}{P_y(t)} = \binom{a_x}{a_y}t^3 + \binom{b_x}{b_y}t^2 + \binom{c_x}{c_y}t + \binom{d_x}{d_y}$

Solve these equations for a specified position and velocity at the beginning ($t = 0$) and the end ($t = s$) of the segment. Traditional method was to set $s = 1$. We instead try to minimise s (total time for segment) while satisfying kinematic contraints:

$$|\frac{P\prime\prime(t)}{A} + \frac{P\prime(t)}{V}|^2 \leq 1, \text{ for } 0 \leq t \leq s,$$

where A and V are the maximal acceleration and velocity of the robot.

7.1.3 Minimizing Time instead of Distance

For the problem of a soccer robot getting to the ball, or a wheeled robot navigating a maze, the path with the shortest distance might not be the path that is quickest to traverse. By speeding up and slowing down, the robot could traverse a path with long straight stretches faster than a shorter path with lots of twists and turns.

This requires more work than just path-finding:

1. Delaunay Triangulation
2. A* Search, using paths composed of Parameteric Cubic Splines
3. Smooth entire curve with Waypoint Tuning by Gradient Descent

Part V

Week 5

Chapter 8

Constraint Satisfaction Problems

8.1 Constraint Satisfaction Problems (CSPs)

Constraint Satisfaction Problems are defined by a set of variables X_i, each with a domain D_i of possible values, and a set of contraints C.

The aim is to find an assignment of the variables X_i from the domains D_i in such a way that none of the constraints C are violated.

Path Search Problems and CSPs are significantly different. It is difficult to know the final state for a CSP, but how to get there is easy. Knowing the final state for a Path Search is easy, but it's difficult to get there.

8.1.1 Example: Map Colouring

Variables: WA, NT, Q, NSW, V, SA, T
Domains: $D_i = red, green, blue$
Constraints: adjacent regions must have different colours

The solution is an assignment that satisfies all of the contraints, e.g. WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = green
TODO add images

8.1.2 Example: n-Queens Puzzle

Put n queens on an n-by-n chess board so that no two queens are attacking each other.

Describing the puzzle as a CSP:
First, simplify the problem: 4-Queens puzzle.
Assume one queen in each column. Which row does each one go in?
Variables: Q_1, Q_2, Q_3, Q_4
Domains: $D_i = 1, 2, 3, 4$
Constraints:
$Q_i \neq Q_j$ (cannot be in the same row) $|Q_i - Q_j| \neq |i - j|$ (or same diagonal)

8.1.3 Example: Cryptarithmetic

	S	E	N	D
+	M	O	R	E
M	O	N	E	Y

Variables: D E M N O R S Y
Domains: $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$
Constraints: $M \neq 0, S \neq 0$ (unary contraints)
$Y = D + E$ or $Y = D + E - 10$, etc.
$D \neq E, D \neq M, D \neq N$, etc.

Cryptarithmetic with Hidden Variables

We can add hidden variables to simplify the constraints.

	T	W	O
+	T	W	O
F	O	U	R

Variables: F T U W R O X_1 X_2 X_3
Domains: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Constraints: F, T, U, W, R, O are all different
O + O = R + $10X_i$, etc.

8.1.4 Example: Sudoku

Come on. You know Sudoku.

8.1.5 Real-World CSPs

- Assignment problems (e.g. who teaches which class?)
- Timetabling problems (e.g. which class is offered when and where?)
- Hardware configuration
- Transport scheduling
- Factory scheduling

8.1.6 Types of Constraints for CSPs

Unary constraints involve a single variable, e.g. $M \neq 0$

Binary constraints involve pairs of variables, e.g. $SA \neq WA$

Higher-order constraints involve 3 or more variables, e.g. $Y = D + E$ or $Y = D + E - 10$

Inequality constraints on continuous variables, e.g. $EndJob_1 + 5 \leq StartJob_3$

Soft constraints (preferences) e.g. 11 am lecture better than 8am lecture.

8.1.7 Standard search formulation

Let's start with a simple but slow approach and then see how to improve it. States are defined by the values assigned so far.

Initial state the empty assignment

Successor function assign a value to an unassigned variable that does not conflict with previously assigned variables ⇒ fails if no legal assignments (not fixable)

Goal test the current assignment is complete

This is the same for all CSPs. Every solution appears at depth n with n variables ⇒ use DFS.

8.1.8 Backtracking search

Variable assignments are commutative; [WA = red then NT = green] ≡ [NT = green then WA = red]. Thus, we only need to consider assignments to a single variable at each node. DFS for CSPs with single-variable assignments is called Backtracking search, and is the basic algorithm for all CSPs. It can solve n-Queens for $n \approx 25$.

There are a number of improvements to backtracking search.

General-purpose heuristics can give huge gains in speed:

1. which variable should be assigned next?
2. in what order should its valuen be tried?
3. can we detect inevitable failure early?

Minimum Remaining Values (MRV)

Choose the variable with the fewest legal values.

Degree Heuristic

The degree heuristic acts as a tie-breaker for MRV variables.

The idea is to choose the variable with the most constraints of remaining values.

Least Constraining Value

Given a variable, choose the least constraining value - the one that rules out the fewest values in the remaining variables.

Combining MRV, Degree Heuristic and Least Constraining Value makes 1000-Queens possible.

Forward Checking

The idea is to keep track of remaining legal values for unassigned variables and terminate search when any variable has no legal values.

Constraint propagation Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures. Constraint propagation repeatedly enforces constraints locally. Thus, while Forward Checking will terminate when a variable has no legal values, constraint propagation will terminate one step earlier, when (for example) two variables will be assigned the same value.

Arc Consistency

Simplest form of constraint propagation makes each arc consistent. $X \to Y$ is consistent if for every value x of X there is some allowed value y. If X loses a value, neighbours of X need to be rechecked. For some problems, it can speed up the search by a huge factor. For others, it may slow the search due to computational overheads.

8.1.9 Local Search

Local Search, or Iterative Improvement, is another class of algorithms for solving CSPs.

These algorithms assign all variables randomly in the beginning (thus violating several constraints) and then change one variable at a time, attempting to reduce the number of violations at each step.

Hill-climbing by min-conflicts

Randomly select any variable and select values by the min-conflicts heuristic: choose the value that violates the fewest contraints.

Phase transition in CSPs Given a random initial state, hill climbing by min-conflicts can solve n-Queens in almost constant time for arbitrary n (e.g. $n = 10,000,000$) with high probability.

In general, randomly-generated CSPs tend to be easy if there are very few or very many constraints. They become extra hard in a narrow range of the ratio

$$\mathrm{R} = \frac{numberofconstraints}{numberofvariables}$$

TODO insert image plz.

Flat regions and local optima

TODO insert image plz

Sometimes we have to go sideways or backwards in order to make progress towards the actual solution.

8.1.10 Simulated Annealing

Simulated annealing is stochastic hill-climbing based on the difference between evaluation of previous state h_0 and new state h_1. If $h_1 < h_0$ then definitely make the change, otherwise make the change with probability $e^{-(h_1-h_0)/T}$ where T is a "temperature" parameter.

Simulated annealing reduces to ordinary hill climbing when $T = 0$, and becomes random when $T \to \infty$. Sometimes, we gradually decrease the temperature during the search (TODO WHY?).

Chapter 9

Evolutionary Computation

We use principles of natural selection to evolve a computational mechanism which performs well at a specified task. Start with a randomly initialised population, and then repeat a cycle of

1. evaluation
2. selection
3. reproduction + mutation

We can use any computational paradigm, with appropriately defined reproduction and mutation operators.

9.1 Evolutionary Computation Paradigms

TODO this section is very empty - anything cool to add?

9.1.1 Bit String Operators (Genetic Algorithm)

We can use bitmask/bitstrings to represent gene values of an organism. The correlation goes something like this (source: http://www.doc.ic.ac.uk/~sgc/teaching/pre2012/v231/lecture16.html)

Genetics Terminology	Genetic Algorithm Terminology	Species Terminology
	Population	Population
Chromosome	Bit string	Organism
Gene	Bit (0 or 1)	
	Selection	Survival of the fittest
Crossover	Recombination	Inheritance
Mutation	Mutation	

Crossovers

A crossover, representing inheritance of genes, can be done in different ways.

You can use a one-point crossover, where additions to the population are modified by taking parts of two "parents". We choose some arbitrary point in the string and the "children" become products of the parent strings, like so:

11101001000 00001010101 become 11101010101 and 00001001000
You can also use a two-point crossover, like so:
1101001000 0000101010101 become 11001011000 and 00101000101

Point Mutation

Point mutation represents the mutation of a gene.
11101001000 becomes 11101011000

9.1.2 S-expression trees (Genetic Programming)

S-expression trees can be used similarly. Recombination becomes swapping subtrees of the S-expressions, to give different results upon evaluation.

9.1.3 Continuous Parameters (Evolutionary Strategy)

For Continuous Parameters, reproduction is just copying, and mutation is adding random noise to weights (or parameters) from a Gaussian distribution with a specified standard deviation. Sometimes, the standard deviation can evolve as well.

9.1.4 Lindenmayer System

TODO what is this? :P

9.2 Now and the Future

9.2.1 Grammatical Evolution

Evolution of programs in C, Lisp, Verilog, Assembler

9.2.2 Hierarchical Evolutionary Re-Combination (HERCL)

Scaling up to larger problems
Modularity and evolution
Credit-assignment problem
Transfer from task to task

Part VI

Week 6

Chapter 10

Logic

10.1 Logical Agents

Humans know things, and what we know helps us do things.

We have knowledge-based agents, which use internal representations of knowledge and reason about this knowledge to infer what to do.

We also have logical agents, which use logical sentences to represent knowledge, and reason by inferring new sentences.

10.1.1 Example: General Game-Playing Agents

General Game Players are systems that are able to understand formal descriptions of arbitrary games, and are able to learn to play these games effectively. They don't know the rules of a game until they start playing.

Unlike specialised game players (like Deep Blue (chess) or Chinook (checkers)) they do not use algorithms in advance for specific games.

How to represent game rules

It is possible in principle to communicate game information in the form of tables (for legal moves, state update)

Problem the size of the description - even if state is finite, the necessary tables can be large (for example, there are 10^{44} states in Chess)

Solution in many cases, worlds are best thought of in terms of atomic features tha may change ("position of white queen", "black can castle", etc). We can represent features directly and use logic to describe how actions change individual features rather than entire states.

Game Description Language (GDL): Facts and Rules

In the following, bold/blue symbols are pre-defined keywords in the logic-based GDL.

Facts

```
role(xplayer)
role(oplayer)

init(cell(1,1,b))
init(cell(1,2,b))
...
init(cell(3,3,b))

init(control(xplayer))
```

Rules

```
legal(W,mark(M,N)) <=
    true(cell(M,N,b)) ∧ true(control(W))

next(cell(M,N,x)) <=
    does(xplayer,mark(M,N))

next(cell(M,N,o)) <=
    does(oplayer,mark(M,N))
```

10.1.2 Example: Wumpus World

Agent's actuators

- turn 90° left or right
- move forward one square
- grab the gold
- shoot an arrow in the current direction
- climb out of the cave from [1,1]

Agent's percepts

- a stench when adjacent to the wumpus
- a breeze when adjacent to a pit
- a glitter when in the square with gold
- a bump when walking into a wall
- a scream when the wumpus is killed

Agents in the Wumpus World

The main challenge is the initial ignorance of the configuration of the environment. The solution to this problem is to draw the right inferences from the available information.

10.1.3 Propositional Logic

Vocabulary

Proposition symbols:	p, q, r	
Logical connectives:	$\neg p$	(negation, "not p")
	$p \wedge q$	(conjunction, "p and q")
	$p \vee q$	(disjunction, "p or q")
	$p \Rightarrow q$	(implication, "p implies q")
	$p \Leftrightarrow q$	(equivalence, "p if and only if q")
Sentences:	built from proposition symbols and connectives, e.g. $ready \Leftrightarrow (a4paper \vee a3paper) \wedge \neg jam$	
Order precedence:	$\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$	

Example: Knights and Knaves

Assumptions knights always tell the truth, knaves always lie

Example Two people, A and B. A says "One of us is a knave!"

Truth table:

p	q	$\neg p$	$p \wedge q$	$p \vee q$	$p \Rightarrow q$	$p \Leftrightarrow q$
false	false	true	false	false	true	true
false	true	true	false	truee	true	false
true	false	false	false	true	false	false
true	true	false	true	true	true	true

Example:

A	B	$\neg A$	$\neg B$	$\neg A \vee \neg B$	$A \Leftrightarrow \neg A \vee \neg B$
false	false	true	true	true	false
false	true	true	false	true	false
true	false	false	true	true	true
true	true	false	false	false	false

Models

A **model** $\mathcal{M}$ fixes the truth value (true of false) for every proposition symbol. For example, $\mathcal{M} = \{A = true, B = false\}$.

$\mathcal{M}$ satisfies a proposition p iff $p = \text{true} \in \mathcal{M}$
$\mathcal{M}$ satisfies a sentence $\neg\alpha$ iff $\mathcal{M}$ does not satisfy a.
$\mathcal{M}$ satisfies a sentence $\alpha \wedge \beta$ iff $\mathcal{M}$ satisfies α and $\mathcal{M}$ satisfies β
$\mathcal{M}$ satisfies a sentence $\alpha \vee \beta$ iff $\mathcal{M}$ satisfies α or $\mathcal{M}$ satisfies β or both.
$\mathcal{M}$ satisfies a sentence $\alpha \Rightarrow \beta$ iff $\mathcal{M}$ satisfies β whenever $\mathcal{M}$ satisfies α.
$\mathcal{M}$ satisfies a sentence $\alpha \Leftrightarrow \beta$ iff $\mathcal{M}$ satisfies α if and only if $\mathcal{M}$ satisfies β

Example: $\mathcal{M} = \{A = \text{true}, B = \text{false}\}$ satisfies $A \Leftrightarrow \neg A \vee \neg B$

Logical Reasoning

β **follows logically** from α, written $\alpha \models \beta$ iff every model that satisfies α also satisfies β. $A \wedge \neg B$ follows logically from $A \Leftrightarrow \neg A \vee \neg B$.

Example 1 We have one person, "A". A says "I am a knight".

How many models satisfy this sentence? 2: A = true and A = false

What follows logically from this sentence? $A \vee \neg A$

Example 2 We have one person, "B". B says "I am a knave."

How many models satisfy this sentence? There are no models as the sentence is contradictory. TODO is this right?

Logical Reasoning in the Wumpus World

$P_{x,y}$	there is a pit in [x,y]
$W_{x,y}$	there is a wumpus in [x,y]
$B_{x,y}$	the agent perceives a breeze in [x,y]
$S_{x,y}$	the agent perceives a stench in [x,y]

TODO add image from lecture notes Logic slide 20

- There is no pit in [1,1]:
  $\neg P_{1,1}$
- A breeze in [1,1] means there is a pit in [1,2] or [2,1]:
  $B_{1,1} \Leftrightarrow P_{1,2} \vee P_{2,1}$
- A breeze in [2,1] means there is a pit in [1,1], [2,2] or [3,1]:
  $B_{2,1} \Leftrightarrow P_{1,1} \vee P_{2,2} \vee P_{3,1}$
- The breeze percepts for the first two squares visited are:
  $\neg B_{1,1}$ and $B_{2,1}$

The conjunction KB of the above 4 sentences is

$$
\begin{aligned}
KB : & (\neg P_{1,1}) \wedge (B_{1,1} \Leftrightarrow P_{1,2} \vee P_{2,1}) \wedge \\
& (B_{2,1} \Leftrightarrow P_{1,1} \vee P_{2,2} \vee P_{3,1}) \wedge (\neg B_{1,1} \text{ and } B_{2,1}) \\
& : (\neg P_{1,1}) \wedge (\neg (P_{1,2} \vee P_{2,1})) \wedge (P_{1,1} \vee P_{2,2} \vee P_{3,1}) \qquad \text{because } \neg B_{1,1} \text{ and } B_{2,1} \\
& : (\neg P_{1,2} \wedge \neg P_{2,1}) \wedge (\neg P_{1,1} \wedge P_{1,1} \vee P_{2,2} \vee P_{3,1}) \qquad \text{by de Morgan's} \\
& : (\neg P_{1,2} \wedge \neg P_{2,1}) \wedge (P_{2,2} \vee P_{3,1})
\end{aligned}
$$

Thus,

$$
\begin{aligned}
KB \models & (\neg P_{1,2} \wedge \neg P_{2,1}) \wedge (P_{2,2} \vee P_{3,1}) \\
& \text{or} \\
KB \models & \neg P_{2,1} \wedge (P_{2,2} \vee P_{3,1})
\end{aligned}
$$

10.1.4 First-Order Logic

Vocabulary

Quantifiers	$\forall, \exists$
Constants	e.g. `richard, john`
Variables	e.g. `X, Y, Z`
Predicates	e.g. `king, married, loves`
Functions	e.g. `father, mother`
Logical connectives	$\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$

The **arity** of a function or predicate is the number of arguments that can be supplied.

A **term** can be a variable (X), a constant (richard), or a functional term (mother(father(john)).

A **sentence** can be an atomic structure, e.g. `married(father(richard), mother(john))`, or a complex sentence, e.g. `∀X ∃ Y loves(X,Y) ⇒ ∃Y ∀X loves(X,Y)`

Other examples of First-Order Logic Sentences

TODO insert image from Logic slide 23

Semantics

The **Herbrand base** is the set of all atomic sentences without variables, e.g. `{king(richard), king(john), married(richard,mother(john)), ...}`.

In first-order logic, a **model** $\mathcal{M}$ fixes the truth value (true or false) for every element in the Herbrand base. From the above example, the model would be `{king(richard)=true, king(john)=false, ...}`.

Logic Entailment

$\mathcal{M}$ satisfies a variable-free atomic sentence p iff $p = \text{true} \in \mathcal{M}$
$\mathcal{M}$ satisfies a sentence $\neg\alpha$ iff $\mathcal{M}$ does not satisfy a.
$\mathcal{M}$ satisfies a sentence $\alpha \wedge \beta$ iff $\mathcal{M}$ satisfies α and $\mathcal{M}$ satisfies β
$\mathcal{M}$ satisfies a sentence $\alpha \vee \beta$ iff $\mathcal{M}$ satisfies α or $\mathcal{M}$ satisfies β or both.
$\mathcal{M}$ satisfies a sentence $\alpha \Rightarrow \beta$ iff $\mathcal{M}$ satisfies β whenever $\mathcal{M}$ satisfies α.
$\mathcal{M}$ satisfies a sentence $\alpha \Leftrightarrow \beta$ iff $\mathcal{M}$ satisfies α if and only if $\mathcal{M}$ satisfies β

$\mathcal{M}$ satisfies a sentence $\forall$X α iff $\mathcal{M}$ satisfies $\alpha\{\text{X/t}\}$[1] for all variable-free terms t
$\mathcal{M}$ satisfies a sentence $\exists$X α iff $\mathcal{M}$ satisfies $\alpha\{\text{X/t}\}$[1] for some variable-free term t

We say that β **follows logically** from α ($\alpha \models \beta$) iff every model that satisfies α also satisfies β.

Example A Let α be the sentence `∀X human(X) ⇒ mortal(X)` (all humans are mortal)
Let β by the sentence `human(socrates)` (Socrates is human)

[1] $\alpha\{\text{X/t}\}$ means replace each occurence of X by t in α

$\mathcal{M}_1$ = {`human(socrates)=false, mortal(socrates)=false`} satisfies α because Socrates is not human, and thus does not have to be mortal, but does not satisfy β.

$\mathcal{M}_2$ = {`human(socrates)=true, mortal(socrates)=true`} satisfies both α and β.

$\mathcal{M}_2$ is the only possible model that satisfies α and β, i.e. $\alpha \wedge \beta \models mortal(socrates)$.

Example B Let α be the sentence $\exists$`X card(X)` $\wedge$ `red(X)` (some cards are red)
Let β be the sentence `card(`♡7`)` $\wedge$ `card(`♣8`)` (♡7 and ♣8 are cards).

{`card(`♡7`)=true, card(`♣8`)=true, red(`♡7`)=true, red(`♣8`)=false`} satisfies $\alpha \wedge \beta$.

However, {`card(`♡7`)=true, card(`♣8`)=true, red(`♡7`)=false, red(`♣8`)=true`} also satisfies $\alpha \wedge \beta$.

Hence, $\alpha \wedge \beta \not\models$ `red(`♡7`)`

10.2 Applied Logic: Describing Games with GDL

The GDL uses the following keywords.

role(r)	r is a role (i.e. a player) in the game
init(f)	f is true in the initial position or state
true(f)	f is true in the current state
does(r,a)	role r does action a in the current state
next(f)	f is true in the next state
legal(r,a)	is is legal for r to play a in the current state
goal(r,v)	r gets goal value v in the current state
terminal	the current state is a terminal state
distinct(s,t)	terms s and t are syntactically different

Like Prolog, all variables are implicitly universally quantified.

```
next(cell(M,N,x)) ⇐ does(xplayer,mark(M,N))

means ∀M,N next(cell(M,N,x)) ⇐ does(xplayer,mark(M,N))
```

10.2.1 Example: Noughts and Crosses

Constants

xplayer, oplayer	Players
x, o, b	Marks
noop	Move
0, 50, 100	Scores

Functions

Name	Arguments	
cell	(number,number,mark)	Feature
control	(player)	Feature
mark	(number,number)	Move

Predicates

row	(number,mark)
column	(number,mark)
diagonal	(mark)
line	(mark)
open	

Players and initial state

```
role(xplayer)
role(oplayer)

init(cell(1,1,b))
init(cell(1,2,b))
init(cell(1,3,b))
init(cell(2,1,b))
init(cell(2,2,b))
init(cell(2,3,b))
init(cell(3,1,b))
init(cell(3,2,b))
init(cell(3,3,b))

init(control(xplayer))
```

Move Generator

Define the following sequence of facts as α

```
true(cell(1,1,x))
true(cell(1,2,b))
true(cell(1,3,b))
true(cell(2,1,b))
true(cell(2,2,o))
true(cell(2,3,b))
true(cell(3,1,b))
true(cell(3,2,b))
true(cell(3,3,x))
true(control(oplayer))
```

Define the following rules as β

```
legal(W,mark(M,N)) ⇐
    true(cell(M,N,b)) ∧
    true(control(W))

legal(xplayer,noop) ⇐
    true(control(oplayer))

legal(oplayer,noop) ⇐
    true(control(xplayer))
```

We then have the following logical consequences:

$$\alpha \wedge \beta \models legal(xplayer, noop)$$
$$\alpha \wedge \beta \models legal(oplayer, mark(1,2))$$
$$\ldots$$
$$\alpha \wedge \beta \models legal(oplayer, mark(3,2))$$

Game Physics

At a high level, we can define states in Noughts and Crosses as follows:

If a player marks a cell, that cell belongs to them in the next state (R_1)

```
next(cell(M,N,x)) ⇐ does(xplayer,mark(M,N))
next(cell(M,N,o)) ⇐ does(oplayer,mark(M,N))
```

As long as a player doesn't make an illegal move (moving in an occupied cell), the cells that currently belong to them will still belong to them on the next state. (R_2)

```
next(cell(M,N,W)) ⇔ does(W,mark(J,K)) ∧
                    true(cell(M,N,W)) ∧
                    (distinct(M,J) ∨ distinct(N,K))
```

On the next state, it will be the other player's turn (R_3)

```
next(control(xplayer)) ⇐ true(control(oplayer))
next(control(oplayer)) ⇐ true(control(xplayer))
```

In combination, the above rules have the following logical consequence(s): $R_1 \wedge R_2 \wedge R_3 \wedge \alpha \wedge does(xplayer, noop) \wedge does(oplayer, mark(1,3)) \models next(cell(1,3,o)) \wedge next(control(xplayer)) \wedge \ldots$

Termination and Goal

```
terminal ⇐ line(x)
terminal ⇐ line(o)
terminal ⇐ ¬open

line(X) ⇐ row(M,X)
line(X) ⇐ column(M,X)
line(X) ⇐ diagonal(X)

open ⇐ true(cell(M,N,b))

goal(xplayer,100) ⇐ line(x)
goal(xplayer,50) ⇐ ¬line(x) ∧ ¬line(o) ∧ ¬open
goal(xplayer,0) ⇐ line(o)

goal(oplayer,100) ⇐ line(o)
goal(oplayer,50) ⇐ ¬line(x) ∧ ¬line(o) ∧ ¬open
goal(oplayer,0) ⇐ line(x)
```

where

```
row(M,X) ⇐ true(cell(M,1,X)) ∧ true(cell(M,2,X)) ∧ true(cell(M,3,X))
column(N,X) ⇐ true(cell(1,N,X)) ∧ true(cell(2,N,X)) ∧ true(cell(3,N,X))
diagonal(X) ⇐ true(cell(1,1,X)) ∧ true(cell(2,2,X)) ∧ true(cell(3,3,X))
diagonal(X) ⇐ true(cell(1,3,X)) ∧ true(cell(2,2,X)) ∧ true(cell(3,1,X))
```

10.3 Knowledge Interchange Format

Knowledge Interchange Format (KIF) is a standard for programmatic exchange of knowledge represented in relational logic.

Syntax is prefix version of standard syntax, with some operators renamed (not, and, or). KIF is case-insensitive. Variables are prefixed with ?.

```
GDL: r(X,Y) ⇐ p(X,Y) ∧ ¬q(Y)
KIF: (⇐ (r ?x ?y) (and (p ?x ?y) (not (q ?y))))
 or: (⇐ (r ?x ?y) (p ?x ?y) (not (q ?y)))
```

10.3.1 Noughts And Crosses in KIF

```
(role xplayer)
(role oplayer)
(init (cell 1 1 b))
(init (cell 1 2 b))
(init (cell 1 3 b))
(init (cell 2 1 b))
(init (cell 2 2 b))
(init (cell 2 3 b))
(init (cell 3 1 b))
(init (cell 3 2 b))
(init (cell 3 3 b))
(init (control xplayer))

(<= (next (cell ?m ?n x))
    (does xplayer (mark ?m ?n))
(<= (next (cell ?m ?n o))
    (does oplayer (mark ?m ?n))
(<= (next (cell ?m ?n b))
    (does ?w (mark ?j ?k))
    (true (cell ?m ?n b))
    (or (distinct ?m ?j)
        (distinct ?n ?k)))
(<= (next (control xplayer))
    (true (control oplayer)))
(<= (next (control oplayer))
    (true (control xplayer)))
```

```
(<= (legal ?w (mark ?m ?n))
    (true (cell ?m ?n b))
    (true (control ?w)))
(<= (legal xplayer noop)
    (true (control oplayer)))
(<= (legal oplayer noop)
    (true (control xplayer)))

(<= (row ?m ?x)
    (true (cell ?m 1 ?x))
    (true (cell ?m 2 ?x))
    (true (cell ?m 3 ?x)))
(<= (column ?n ?w)
    (true (cell 1 ?n ?x))
    (true (cell 2 ?n ?x))
    (true (cell 3 ?n ?x)))
(<= (diagonal ?x)
    (true (cell 1 1 ?x))
    (true (cell 2 2 ?x))
    (true (cell 3 3 ?x)))
(<= (diagonal ?x)
    (true (cell 1 3 ?x))
    (true (cell 2 2 ?x))
    (true (cell 3 1 ?x)))

(<= (line ?x) (row ?m ?x))
(<= (line ?x) (column ?n ?x))
```

```
(<= (line ?x) (diagonal ?x))

<= open
    (true (cell ?m ?n b)))

(<= terminal (line x))
(<= terminal (line o))
(<= terminal (not open))

(<= (goal xplayer 100)
    (line x))
(<= (goal xplayer 50)
    (not (line x))
    (not (line o))
    (not open))
(<= (goal xplayer 0)
    (line o))
(<= (goal oplayer 100)
    (line o))
(<= (goal oplayer 50)
    (not (line x))
    (not (line o))
    (not open))
(<= (goal oplayer 0)
    (line x))
```

10.4 Inference

We will cover inference algorithms for propositional logic, and inference algorithms for first-order logic.

10.4.1 Propositional Inference: Example

Assumptions: knights always tell the truth, knaves always lie.

We have two people, A and B. A says "one of us is a knave".

It follows that A is a knight and B is a knave.

$A \leftrightarrow$ A is a knight $B \leftrightarrow$ B is a knight

$A \Leftrightarrow \neg A \vee \neg B \models A \wedge \neg B$

10.4.2 Converting a sentence to CNF

A **literal** is an atomic sentence or its negation.
A **clause** is a disjunction of literals.
A sentence is **conjunctive normal form** (CNF) is a conjunction of clauses.
To convert a propositional sentence into CNF:

- eliminate $\Leftrightarrow$ by replacing $\alpha \Leftrightarrow \beta$ with $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$
- eliminate $\Rightarrow$ by replacing $\alpha \Rightarrow \beta$ vith $\neg\alpha \vee \beta$
- move $\neg$ inwards by repeated application of
  - $\neg\neg\alpha \equiv \alpha$ (double negation elimination)
  - $\neg(\alpha \wedge \beta) \equiv (\neg\alpha \vee \neg\beta)$ (de Morgan rule)
  - $\neg(\alpha \vee \beta) \equiv (\neg\alpha \wedge \neg\beta)$ (de Morgan rule)
- Apply distributivity laws whenever possible
  - $\alpha \wedge (\beta \vee \gamma) \equiv ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma))$
  - $\alpha \vee (\beta \wedge \gamma) \equiv ((\alpha \vee \beta) \wedge (\alpha \vee \gamma))$

10.4.3 A Possible Resolution Algorithm

Resolution uses the principle of proof by contradiction: $\alpha \models \beta$ iff $\alpha \wedge \neg\beta$ is unsatisfiable.

1. Convert $\alpha \wedge \neg\beta$ to CNF
2. Repeatedly apply the resolution rule (http://en.wikipedia.org/wiki/Resolution_(logic)#Resolution_in_propositional_logic that takes two clauses as input and produces a new clause
3. If the resolution rule produces the empty clause we have no conflict, and so β follows from α.
4. Otherwise, if there are no more new clauses, then β does not follow from α.

Example

A says "one of us in a knave" Does it follow that A is a knight and B is a knave?
Prove that A $\Leftrightarrow \neg$A $\vee \neg$B $\models$ A $\wedge \neg$B
Prove that (A $\Leftrightarrow \neg$A $\vee \neg$B) $\wedge \neg$(A $\wedge \neg$B) is unsatisfiable.
Convert to CNF:

```
(A ⇔ ¬A ∨ ¬B) ∧ ¬(A ∧ ¬B)
((A ⇒ ¬A ∨ ¬B) ∧ (¬A ∨ ¬B ⇒ A)) ∧ ¬(A ∧ ¬B)
((¬A ∨ (¬A ∨ ¬B)) ∧ (¬(¬A ∨ ¬B) ∨ A)) ∧ ¬(A ∧ ¬B)
(¬A ∨ ¬B) ∧ A ∧ (¬A ∨ B)
¬B ∧ A ∧ B
B ∧ ¬B ∧ A
```

which is a contradiction.
Thus we have shown that (A $\Leftrightarrow \neg$A $\vee \neg$B) $\wedge \neg$(A $\wedge \neg$B) is unsatisfiable.
And so A $\Leftrightarrow \neg$A $\vee \neg$B $\models$ A $\wedge \neg$B.

10.4.4 First-Order Resolution for Logic Programs/GDL

Given

```
true(cell(1,1,b))
...
true(cell(3,3,b))
true(control(xplayer))

legal(P,mark(M,N)) <= true(cell(M,N,b))     true(control(P))
legal(xplayer,noop) <= true(control(oplayer))
legal(oplayer,noop) <= true(control(xplayer))
```

answer the query `?- legal(xplayer,M)`

```
(true(control(oplayer)) ∧ M = noop) ∨
    (true(control(xplayer)) ∧ M = mark(M,N) ∧ cell(M,N,b))
    ⇒ legal(xplayer,M)

(control(xplayer) ∧ mark(M,N) ∧ cell(M,N,b))
    ⇒ legal(xplayer,M)

control(xplayer) ∧ (M = mark(M,N) ∧ cell(M,N,b)
                    ⇒ legal(xplayer,M)
```

And, since `control(xplayer) = true` we have

```
M = mark(1,1)
M = mark(1,2)
M = mark(1,3)
M = mark(2,1)
..
..
M = mark(3,2)
M = mark(3,3)
```

Substitutions

To compute logical consequences you need

1. unification: using substitutions in order to match two expressions
2. resolution steps (a single derivation step)
3. derivations (computing a result for a query)

A **substitution** is a finite set of replacements of variables by terms, e.g. {X/a, Y/f(b), V/W}

The result of applying a substitution Θ to an expression p is the expression SUBST(Θ, p) obtained from p by replacing every occurrence of every variable in the substitution by its replacement.

`SUBST({X/a, Y/f(b), V/W}, p(X,X,Y,Z)) = p(a,a,f(b),Z)`

Unification

A substitution Θ is a **unifier** for an expression p and an expression q iff SUBST(Θ, p) = SUBST(Θ, q).

```
SUBST(\{M/1,N/3,X/3\},mark(M,N)) = mark(1,3)
SUBST(\{M/1,N/3,X/3\},mark(1,X)) = mark(1,3)
```

If two expressions have a unifier, we say they are **unifiable**. mark(X,X) and mark(1,3) are not unifiable.

Most General Unifiers

A substitution Θ is **more general** than a substitution θ iff Θ places fewer restrictions than θ on the values of variables.

A substitution is a **most general unifier** (mgu) of two expressions iff it is more general than any other unifier.

Theorem If two expressions are unifiable, then they have an mgu that is unique up to variable permutation.

```
SUBST({M/1,N/X},mark(M,N)) = mark(1,X)
SUBST({M/1,N/X},mark(1,X)) = mark(1,X)

SUBST({M/1,X/N},mark(M,N)) = mark(1,N)
SUBST({M/1,X/N),mark(1,X)) = mark(1,N)
```

Resolution Step for Queries + Logic Program Clauses

Given: Query $L_1 \wedge L_2 \wedge ... \wedge L_m$ (without negation)
clauses (without negation)

Let: $A \Leftarrow B_1 \wedge ... \wedge B_n$ "fresh" variant of clause
Θ mgu of L_1 and A

Then: $L_1 \wedge L_2 \wedge ... \wedge L_m \rightarrow$ SUBST(Θ,$B_1 \wedge ... \wedge B_n \wedge L_2 \wedge ... \wedge L_m$) is a **resolution step**

```
legal(P5,mark(M5,N5)) ⇐ true(cell(M5,N5,b)) ∧ true(control(P5))

legal(xplayer,M) → with Θ = {P5/xplayer,M/mark(M5,N5)}

true(cell(M5,N5,b)) ∧ true(control(xplayer))
```

Derivation 1: Query Answering

A sequence of resolution steps is called a **derivation**. A **successful** derivation ends with the empty query ($\square$).

The **answer substitution** (computed by a successful derivation) is obtained by composing the mgu's $\Theta_1 \circ ... \circ \Theta_n$ of each step (and restricting the result to the variables in the original query).

A **failed** derivation ends with a query to which no clause applies.

```
true(cell(1,1,b))
...
true(cell(3,3,b))
true(control(xplayer)

legal(P,mark(M,N)) ⇐ true(cell(M,N,b)) ∧ true(control(P))
legal(xplayer,noop) ⇐ true(control(oplayer))
legal(oplayer,noop) ⇐ true(control(xplayer))

legal(xplayer,M)
    → with Θ1 = {P5/xplayer,M/mark(M5,N5)}
true(cell(M5,N5,b)) ∧ true(control(xplayer))
    → with Θ2 = {M5/1,N5/1}
true(control(xplayer))
    → with Θ3 = {}
□, with answer substitution {M/mark(1,1)}
```

The query `?- legal(xplayer,M)` has 9 successful derivations, with the following answers:

```
{M/mark(1,1)}
...
{M/mark(3,3)}
```

The query `?- legal(oplayer,M)` has 1 successful derivation, with the answer `{M/noop}`.

Derivation 2: Query Answering with Negation

Given a query $L_1 \wedge L_2 \wedge ... \wedge L_m$ and clauses:

- If L_1 is an atom, proceed as before
- If L_1 is of the form $\neg$A
  - if all derivations for A fail then
    $\neg A \wedge L_2 \wedge ... \wedge L_m \rightarrow L_2 \wedge ... \wedge L_m$
  - if there is a successful derivation for A then
    $\neg A \wedge L_2 \wedge ... \wedge L_m \rightarrow$ fail

```
role(red)
role(blue)
role(green)
true(freecell(blue))
trapped(P) ⇐ role(P) ∧ ¬true(freecell(P))
goal(P,100) ⇐ role(P) ∧ ¬trapped(P)

goal(P,100) → role(P) ∧ ¬trapped(P) → ¬trapped(blue)
  and
    trapped(blue) → role(blue) ∧ ¬true(freecell(blue))
  but we have true(freecell(blue)) so trapped(blue) fails

as such, we have goal(P,100) → role(P) ∧ ¬[fails]
```

and so the answer substitution is `{P/blue}`, which is the only answer to this query.

Derivation 3: Query Answering with Disjunction

A GDL/Prolog rule with a disjunction
A $\Leftarrow B \wedge (C_1 \vee C_2) \wedge D$
is logically equivalent to the conjunction of the clauses
A $\Leftarrow B \wedge C_1 \wedge D$
A $\Leftarrow B \wedge C_2 \wedge D$

As such,
line(W) $\Leftarrow$ row(M,W) $\vee$ column(N,W) $\vee$ diagonal(W)
is logically equivalent to

```
line(W) ⇐ row(M,W)
line(W) ⇐ column(N,W)
line(W) ⇐ diagonal(W)
```

10.5 Logic in Action: Inference Tasks for General Game-Playing Agents

Let KB be the logical description of a game.

1. To infer the initial position, compute all answers to KB $\models$ init(F)
2. To infer the legal moves of player p in a given state $\{f_1, ..., f_n\}$ compute all answers to KB $\wedge true(f_1) \wedge ... \wedge true(f_n) \models legal(p, M)$
3. To infer the next state from a given state and moves, compute al answers to KB $\wedge true(f_1) \wedge ... \wedge true(f_n) \wedge does(p_1, f_1) \wedge ... \wedge does(p_n, f_n) \models$ next(F)
4. To check if a state is terminal: KB $\wedge true(f_1) \wedge ... \wedge true(f_n) \models$ terminal
5. To compute player p's goal value: KB $\wedge true(f_1) \wedge ... \wedge true(f_n) \models$ goal(p,N)

Part VII

Week 7

Chapter 11

Planning

11.1 Planning Agents

A critical part of Artifical Intelligence is devising a plan of action to achieve one's goal.

11.1.1 Example: Planning as Single-Player Game in GDL

We have an initial state, say

```
init(have(cake))
```

and a goal, say

```
goal(agent,100) ⇐ true(have(cake)) ∧ true(eaten(cake))
```

Actions can be defined as follows:

```
legal(agent,eat(cake)) ⇐ true(have(cake))
legal(agent,bake(cake)) ⇐ ¬true(have(cake))
```

And effects can be similarly defined:

```
next(eaten(cake)) ⇐ does(agent,eat(cake))
next(eaten(cake)) ⇐ true(eaten(cake))
next(have(cake)) ⇐ does(agent,bake(cake))
next(have(cake)) ⇐ true(have(cake)) ∧ ¬does(agent,eat(cake))
```

A Special-Purpose Planning Domain Definition Language (PDDL)

```
Init(conjunction of atomic sentences without variables)
Goal(conjunction of literals)
Action(Name(parameters),
    PRECOND: conjunction of literals,
    EFFECT:  conjunction of literals)
```

For example,

```
Init(have(cake))
Goal(have(cake)∧eaten(cake))
Action(eat(C),
    PRECOND: have(C),
    EFFECT:  ¬have(C) ∧ eaten(C))
Action(bake(C),
    PRECOND: ¬have(C),
    EFFECT:  have(C))
```

11.1.2 Example: Blocks World Planning

TODO insert image here. Slide 6, Planning.

A robot arm can pick up a block and hmove it to another position. The arm can only pick up one block at a time.

Describing in PDDL

```
Init(on(a,table) ∧ on(b,table) ∧ on(c,a) ∧ clear(b) ∧ clear(c))

Goal(on(a,b) ∧ on(b,c))

Action(move(B,X,Y),
    PRECOND: on(B,X) ∧ clear(B) ∧ clear(Y) ∧ X ≠ Y,
    EFFECT:  on(B,Y) ∧ clear(X) ∧ ¬on(B,X) ∧ ¬clear(Y))

Action(moveToTable(B,X),
    PRECOND: on(B,X) ∧ clear(B),
    EFFECT:  on(B,table) ∧ clear(X) ∧ ¬on(B,X))

\section{Planning as State-Based Search}
A state space is some collection of states with actions that lead to other
states. The state space includes an Inital State and states that satisfy th
goal.

TODO insert image, slide 9, Planning.

\subsection{Example: State Space for Have-The-Cake-And-Eat-It-Too}
TODO insert image, slide 10, Planning.

\subsection{Example: State Space for Blocksworld}
\begin{lstlisting}[morekeywords={Action}]
Action(move(B,X,Y),
    PRECOND: on(B,X) ∧ clear(B) ∧ clear(Y) ∧ B≠Y ∧ X≠Y,
    EFFECT:  on(B,Y) ∧ clear(Y) ∧ ¬on(B,X) ∧ ¬clear(Y))
Action(moveToTable(B,X),
    PRECOND: on(B,X) ∧ clear(B),
    EFFECT:  on(B,table) ∧ clear(X) ∧ ¬on(B,X))
```

TODO what is going on with Forward/Backward/Bidirectional Search on

slides 12,13,14?

11.2 Partial-Order Planning

Partial-Order Plans (POPs) give us a general idea of the order that steps must be performed in, but not a strict order. Let us consider the following:

TODO insert image from slide 16, Planning.

To reach our goal state, we must:

1. Move peg from 4,1 to 4,3, taking the peg in 4,2
2. Move peg from 4,3 to 4,5, taking the peg in 4,4

1. Move peg from 9,5 to 7,5, taking the peg in 8,5
2. Move peg from 7,5 to 5,5, taking the peg in 6,5

And finally, move peg from 4,5 to 6,5, taking the peg in 5,5.

This partial-order plan represents 6 sequential plans, as the only condition is that the '2' moves must be performed after the '1' moves, and the final move must be performed after all other moves.

```
Right(41), Right(43), Up(95), Up(75), Down(45)
Right(41), Up(95), Right(43), Up(75), Down(45)
Right(41), Up(95), Up(75), Right(43), Down(45)
Up(95), Right(41), Right(43), Up(75), Down(45)
Up(95), Right(41), Up(75), Right(43), Down(45)
Up(95), Up(75), Right(41), Right(43), Down(45)
```

11.2.1 Plan-Based Search

In the below image, each node is a plan.

TODO insert image from slide 18, Planning.

11.2.2 Partial-Order Planning as a Search Problem

Search nodes are (mostly unfinished) partial-order plans. The inital plan contains only the Start and Finish actions. We define the EFFECT of Start as the inital state of the planning problem, and the PRECOND of Finish as the goal of the planning problem.

Plans have 4 components:

1. A set of actions (steps of the plan)
2. A set of ordering constraints $A < B$ (A before B)
3. A set of **causal links** A $\xrightarrow{p}$ B ("action A achieves p for action B")
4. A set of open preconditions

An action C **conflicts** with a causal link A $\xrightarrow{p}$ B if C has the effect $\neg$p and C could come after A and before B.

A plan with no conflicts and no open preconditions is a **solution**.

Example: Partial-Order Planning for Have-The-Cake-And-Eat-It-Too

Our inital plan has just "Start" and "Finish". We add a causal link, to obtain: Start $\xrightarrow{have(cake)}$ Finish

We then add our action, eat(cake). Now we have eat(cake) causing a conflict for the causal link from Start to Finish: as eat(cake) $\xrightarrow{eaten(cake)}$ Finish, eat(cake) has the effect ¬have(cake).

In other words:

```
Init(have(cake))
Goal(have(cake) ∧ eaten(cake))
Action(eat(C),
    PRECOND: have(C),
    EFFECT:  ¬have(C) ∧ eaten(C))
```

Solution: Plan Without Conflict or Open Precondition TODO insert image from slide 21, Planning.

```
Init(have(cake))
Goal(have(cake) ∧ eaten(cake))
Action(eat(C),
    PRECOND: have(C),
    EFFECT:  ¬have(C) ∧ eaten(C))
Action(bake(C),
    PRECOND: ¬have(C),
    EFFECT:  have(C))
```

11.2.3 Algorithm for Solving POPs

The initial plan contains *Start* and *Finish*, the ordering constraint $Start < Finish$, and no causal links. All preconditions of *Finish* are open.

We repeat the following:

1. Pick an open precondition p (of an action B in the plan)
2. Pick an action A with effect p
3. Add the causal link A $\xrightarrow{p}$ B and the ordering constraint A¡B (if A is new to the plan and $Start < A$ and $A < Finish$)
4. If a conflict arises between the causal link A $\xrightarrow{p}$ B and an action C: add either $B < C$ or $C < A$ if consistent with the existing ordering.

Retry (with different choices) if conflict cannot be resolved.
Stop if a solution is found.

11.2.4 Example: Spare Tire Problem

```
Init(at(flat,axle) ∧ at(spare,trunk))
Goal(at(spare,axle))
Action(remove(Obj,Loc),
    PRECOND: at(Obj,Loc),
    EFFECT:  ¬at(Obj,Loc) ∧ at(Obj,ground))
Action(putOn(T,axle),
    PRECOND: at(T,ground) ∧ ¬at(flat,axle),
    EFFECT:  ¬at(T,ground) ∧ at(T,axle))
```

Solving the Spare Tire Problem with POP Planning

TODO insert images from slides 24, 25 Planning.

11.3 Planning with Propositional Logic

11.3.1 Encoding Planning Problems in Propositional Logic

Planning can be done by testing the **satisfiability** of a logical sentence

$\texttt{initial-state}^0 \wedge \texttt{transition}^0 \wedge ... \wedge \texttt{transition}^{max-1} \wedge \texttt{goal}^{max}$

A sentence transition^t contains propositions A^t for every potential occurrence of an action. A model will assign *true* to A^t iff doing action A at time $t \in \{0, ..., max-1\}$ is part of a correct plan. For example, movetoTable(c,a)^0 = *true*, move(b,table,c)^1 = *true*, ...

Planners based on satisfiability can handle large planning problems.

Example: Propositional Logic for Have-The-Cake-And-Eat-It-Too

$$\texttt{Initial and goal state:}$$
$$\texttt{have-cake}^0 \wedge$$
$$\texttt{have-cake}^2 \wedge \texttt{eaten-cake}^2 \wedge$$

$$\texttt{Action preconditions:}$$
$$(\texttt{eat-cake}^0 \Rightarrow \texttt{have-cake}^0) \wedge$$
$$(\texttt{eat-cake}^1 \Rightarrow \texttt{have-cake}^1) \wedge$$
$$(\texttt{bake-cake}^0 \Rightarrow \neg\texttt{have-cake}^0) \wedge$$
$$(\texttt{bake-cake}^1 \Rightarrow \neg\texttt{have-cake}^1) \wedge$$

$$\texttt{Transitions } (\text{F}^{t+1} \Leftrightarrow \texttt{ActionCausesF}^t \vee (\text{F}^t \wedge \neg\texttt{ActionCausesNotF}^t)$$
$$(\texttt{have-cake}^1 \Leftrightarrow \texttt{bake-cake}^0 \vee (\texttt{have-cake}^0 \wedge \neg\texttt{eat-cake}^0)) \wedge$$
$$(\texttt{have-cake}^2 \Leftrightarrow \texttt{bake-cake}^1 \vee (\texttt{have-cake}^1 \wedge \neg\texttt{eat-cake}^1)) \wedge$$
$$(\texttt{eaten-cake}^1 \Leftrightarrow \texttt{eat-cake}^0 \vee \texttt{eaten-cake}^0) \wedge$$
$$(\texttt{eaten-cake}^2 \Leftrightarrow \texttt{eat-cake}^1 \vee \texttt{eaten-cake}^1)$$

$$\texttt{have-cake}^0 = \textit{true},$$
$$\texttt{eaten-cake}^0 = \textit{false},$$
$$\texttt{eat-cake}^0 = \textit{true},$$
$$\texttt{bake-cake}^0 = \textit{false},$$

$$\texttt{have-cake}^1 = \textit{false},$$

$\texttt{eaten-cake}^1 = true$,
$\texttt{eat-cake}^1 = false$,
$\texttt{bake-cake}^1 = true$,

$\texttt{have-cake}^2 = true$,
$\texttt{eaten-cake}^2 = true$

Example: Blocks World Planning as Satisfiability

The initial state is represented as such:

$\texttt{on(a,table)}^0 \wedge\ \texttt{on(b,table)}^0 \wedge\ \texttt{on(c,a)}^0 \wedge\ \texttt{clear(b)}^0 \wedge\ \texttt{clear(c)}^0$

And the goal is encoded as follows:

$\texttt{on(a,b)}^{max} \wedge\ \texttt{on(b,c)}^{max}$

The action preconditions:

$\texttt{move(B,X,Y)}^t \Rightarrow\ \texttt{on(B,X)}^t \wedge\ \texttt{clear(B)}^t \wedge\ \texttt{clear(Y)}^t$
$\texttt{moveToTable(B,X)}^t \Rightarrow\ \texttt{on(B,X)}^t \wedge\ \texttt{clear(B)}^t$
$\quad (\forall\ \texttt{B,X,Y} \in \{\texttt{a,b,c,table}\},\ \texttt{t} \in \{\texttt{0,1,2,...,}max\texttt{-1}\},\ \texttt{B} \neq \texttt{Y},\ \texttt{X} \neq \texttt{Y})$

Constraints: actions can't be executed in parallel

$\neg(\texttt{move(B,X,Y)}^t \wedge\ \texttt{moveToTable(C,Z)}^t)$
$\neg(\texttt{moveToTable(B,X)}^t \wedge\ \texttt{moveToTable(B}'\texttt{,X}')^t)$
$\neg(\texttt{move(B,X,Y)}^t \wedge\ \texttt{move(B}'\texttt{,X}'\texttt{,Y}')^t)$
$\quad (\forall\ \texttt{B,B}'\texttt{,C,X,X}'\texttt{,Y,Y}'\texttt{,Z} \in \{\texttt{a,b,c,table}\},$
$\quad\ \texttt{t} \in \{\texttt{0,1,2,...,}max\texttt{-1}\},\ \texttt{B} \neq \texttt{B}',\ \texttt{...})$

Encoding transitions:

$\texttt{on(B,X)}^{t+1} \Leftrightarrow\ \texttt{move(B,Y,X)}^t \vee$
$\qquad\qquad (\texttt{moveToTable(B,Y)}^t \wedge\ \texttt{X=table}) \vee$
$\qquad\qquad (\texttt{on(B,X)}^t \wedge \neg\texttt{move(B,X,Y)}^t \wedge \neg\texttt{moveToTable(B,X)}^t)$

$\texttt{clear(B)}^{t+1} \Leftrightarrow\ \texttt{move(X,B,Y)}^t \vee$
$\qquad\qquad\ \texttt{moveToTable(X,B)}^t \vee$
$\qquad\qquad\ (\texttt{clear(B)}^t \wedge \neg\texttt{move(X,Y,B)}^t)$

$\quad (\forall\ \texttt{B,X,Y} \in \{\texttt{a,b,c,table}\},\ \texttt{t} \in \{\texttt{0,1,2,...,}max\texttt{-1}\},$
$\quad\ \texttt{B} \neq \texttt{X},\ \texttt{B} \neq \texttt{Y},\ \texttt{X} \neq \texttt{Y})$

For $max = 3$ there is 1 model that satisfies all sentences. The propositions that are true in this model include:

$\texttt{moveToTable(c,a)}^0,\ \texttt{move(b,table,c)}^1,\ \texttt{move(a,table,b)}^2$

Chapter 12

General Game Playing

12.1 Blind General Game Playing

12.1.1 Monte-Carlo Tree Search: The Idea

Monte-Carlo Tree Search (MCTS) tries to analyse the most promising moves based on the expansion of the game tree on random sampling of the search space. MCTS is based on playouts: games are played out to the very end, selecting moves at random. The final result is used to weight the nodes in the game tree so that better nodes are more likely to be chosen in future playouts.

Doing this should allow expanding nodes in the game tree so that fewer nodes are examined, but game play is not significantly negatively effected - all while reducing the amount of space expanded.

TODO insert image from slide 4, General Game Playing.

12.1.2 MCTS: How it Works

The value of a move = average score returned by simulation

TODO insert image from slide 5, General Game Playing

12.1.3 Improvement: Confidence Bounds

Play one random game for each move

For the next simulation choose move i with

$argmax_i(v_i + C \times \sqrt{\frac{\log n}{n_i}})$, where $\sqrt{\frac{\log n}{n_i}}$ is the confidence bound. Here,

i move

v_i average result for i

n_i number of sample runs for i

n number of sample runs

C a constant

12.1.4 Assessment

Upper Confidence Bound Monte-Carlo Tree Search (UCT) works well for games that

- **converge** to the goal (e.g. Othello)
- have a large branching factor (e.g. Amazons)
- do not have good/perfect heuristics (e.g. Ultimate Tic Tac Toe)

12.2 Informed General Game Playing

12.2.1 Recognising structures

Informed Search: Exploiting Symmetries

Symmetries can be logically derived from the rules of a game.

A **symmetry relation** over the elements of a domain is an equivalence relation such that

- two symmetric states are either both terminal or non-terminal
- if they are terminal, they have the same goal value
- if they are non-terminal, the legal moves in each of them are symmetric and yield symmetric states.

You can have reflectional symmetry (for example, in Connect 3)
TODO insert image from slide 12, GGP
You can have rotational symmetry (for example, in Capture Go)
TODO insert image from slide 13, GGP

Informed Search: Factoring

Take the game "hodepodge", a combination of chess and othello. If the branching factor of chess is a and the branching factor of Othello is b, the branching factor as given to players is ab. The fringe of the tree at depth n is given by $(ab)^n$ or, factored, $a^n + b^n$.

TODO what the hell is happening on slide 15 of GGP?

Game Factoring and its Use

1. Compute factors
    - Behavioural factoring
    - Goal factoring
2. Play factors
3. Reassemble solution
    - Append plans
    - Interleave plans
    - Parallelise plans with simultaneous actions

12.2.2 Discovering heuristics

Blind search: only assign scores to nodes based on the evalutanio of the complete subtrees at those nodes

Problem: can relatively rarely see all the way to the bottom of a tree for a single node, even less so for every successor node

Solution: improve efficiency of inference

Solution: assign intermediate scores to nodes based on an **evaluation function**.

Evaluation Functions

Typically designed by programmers and humans, a great deal of thought and empirical testing goes into choosing good functions. (for example, piece count and piece values in Chess, or holding corners in Othello).

This requires knowledge of the game's structure, semantics, play order, etc.

In the general case, we have no knowledge of features, no insight into the game structure, and no intuition about what is a good feature for a game. Some general ideas work in many cases, but sometimes they don't.

Example: the "mobility" heuristics

Mobility "More moves means better state". Optionally, you can include "limiting opponent moves is preferable".

This is good because in many games, being cornered and forced into making a move is bad (Chess, Othello).

This can be bad when mobility is counterproductive, e.g. in Checkers.

Inverse Mobility Sometimes, having fewer things to do is preferable (or, optionally, giving the opponent things to do is better).

This works in games like nothello (Suicide Othello), where you want to lose pieces.

We still have the problem of how to decide between mobility and inverse mobility.

Generating Evaluation Functions: Goal Distance

The better an intermediate state satisfies the goal specification, the better it is.

We use **Fuzzy Logic** to evaluate the "degree of truth" of a goal formula.

True literals take the value $0.5 < p < 1.0$, and false literals take $1 - p$.

For example: take $p = 0.9$.

```
A = true    B = true    C = false
⇒ fuzzy_eval(A ∧¬B ∧ C) = (0.9)(1-0.9)(0.1) = 0.009

A = false   B = false   C = true
⇒ fuzzy_eval(A ∧¬B ∧ C) = (0.1)(1-0.1)(0.9) = 0.081
⇒ fuzzy_eval(A ∨¬B ∨ C) = 1 - fuzzy_eval(¬A ∧ B ∧¬C) = 0.991
```

Example: Noughts and Crosses

```
goal(xplayer, 100) ⇐ true(cell(M,1,x)) ∧
                     true(cell(M,2,x)) ∧
                     true(cell(M,3,x))
                     ∨
                     true(cell(1,N,X)) ∧
                     true(cell(2,N,X)) ∧
                     true(cell(3,N,X))
                     ∨
                     true(cell(1,1,x)) ∧
                     true(cell(2,2,x)) ∧
                     true(cell(3,3,x))
                     ∨
                     true(cell(1,3,x)) ∧
                     true(cell(2,2,x)) ∧
                     true(cell(3,1,x))
```

Evaluation of Intermediate States TODO insert image from slide 26, GGP

```
fuzzy_eval(goal(xplayer,100)) after does(xplayer,mark(2,2))
> fuzzy_eval(goal(xplayer,100)) after does(xplayer,mark(1,1))
> fuzzy_eval(goal(xplayer,100)) after does(xplayer,mark(1,2))
```

Part VIII

Week 8

Chapter 13

Learning and Decision Trees

13.1 Learning Agents

This is a callback to early in the course:
TODO insert image from slide 1, Learning

13.2 Types of Learning

13.2.1 Supervised Learning

The agent is presented with examples of inputs and their target outputs

We have a training set and a test set, each consisting of a set of items; for each item, a number of input attributes and a target value are specified.

The aim is to predict the target value, based on the input attributes.

The agent is presented with the input and target output for each item in the training set; it must then predict the output for each item in the test set.

Various learning paradigms are available:

- Decision Tree
- Neural Network
- Support Vector Machine

Issues with Supervised Learning

- framework (decision tree, neural network, SVM, etc.)
- representation (of inputs and outputs)
- pre-processing / post-processing
- training method (perceptron learning, back-propagation, etc.)
- generalization (avoid over-fitting)
- evaluation (separate training and testing sets)

Curve Fitting

Which curve gives the "best fit" for data? A straight line? Parabola? 4th order polynomial? something else?

TODO insert image from slide 5-9, Learning.

Ockham's Razor is a principle which states "the most likely hypothesis is the simplest one consistent with observed data."

Since there can be noise in measurements, we need to make a tradeoff between simplicity of the hypothesis and how well it fits the data.

TODO insert image from slide 10, Learning.

TODO what is going on with restaurant stuff on slides 13, 14?

Generalisation

Provided the training data are not inconsistent, we can split the attributes in any order and still produce a tree that correctly classifies all examples in the training set.

However, we really want a tree which is likely to generalize to correctly classify the [unseen] examples in the test set.

In view of Ockham's Razor, we prefer a simpler hypothesis, i.e. a smaller tree.

But how can be choose attributes in order to produce a small tree?

Choosing an Attribute Patrons is a "more informative" attribute than Type, because it splits the examples more nearly into sets that are "all positive" or "all negative".

This notion of "informativeness" can be quantified using the mathematical concept of entropy, A parsimonious tree can be built by minimizing the entropy at each step.

Entropy Entropy is the measure of how much information we gain when the target attribute is revealed to us. In other words, it is not a measure of how much me know, but a measure of how much we don't know.

If the prior probabilities of the n target attribute values are $p_1, ..., p_n$ then the entropy is

$$H(\langle p_1, ..., p_n \rangle) = \sum_{i=1}^{n} -p_i \log_2 p_i$$

Entropy is the number of bits per symbol achieved by a (block) Huffman Coding scheme. For example, $H(\langle 0.5, 0.5 \rangle) = 1$ bit, $H(\langle 0.5, 0.25, 0.25 \rangle) = 1.5$ bits.

Suppose we have p positive and n negative examples at a node. To classify a new example, $H(\langle \frac{p}{p+n}, \frac{n}{p+n} \rangle)$ bits are needed.

An attribute splits the examples E into subsets E_i, each of which needs less information to complete the classification.

Let E_i have p_i positive and n_i negative examples. $H(\langle \frac{p_i}{p_i+n_i}, \frac{n_i}{p_i+n_i} \rangle)$ bits are needed to classify a new example. The expected number of bits per example over all branches is

$$\sum_i \frac{p_i+n_i}{p+n} H(\langle \frac{p_i}{p_i+n_i}, \frac{n_i}{p_i+n_i} \rangle)$$

Laplace Error and Pruning According to Ockham's Razor, we may wish to pune off branches that do not provide much benefit to classifying the items.

When a node becomes a leaf, all items will be assigned to the majority class at that node. We can estimate the error rate on the (unseen) test items using the Laplace error:

$$E = 1 - \frac{n+1}{N+k}$$

N = total number of traning items at the node n = number of training items in the majority class k = number of classes

If the average Laplace error of the children exceeds that of the parent node, we prune off the children.

Minimal Error Pruning TODO what is going on on slide 22, Learning.

Learning Actions

Supervised Learning can be used to learn Actions, if we construct a training set of situation-action pairs (called Behavioural Cloning).

It is not always easy, appropriate, or even possible to provide a "training set". Examples include

- optimal control (mobile robots, pole balancing, flying a helicopter)
- resource allocation (job shop scheduling, mobile phone channel allocation)
- mix of allocation and contol (elevator control, backgammon, ...)

13.2.2 Reinforcement Learning

Agent is not presented with target outputs, but is given a reward signal which it aims to maximize

Framework

An agent interacts with its environment. There is a set S of states and a set A of actions.

At each time step t, the agent is in some state s_t. It must choose an action a_t, whereupon it goes into state $s_{t+1} = \delta(s_t, a_t)$ and receives reward $r(s_t, a_t)$.

In general, $r()$ and $\delta()$ can be multi-valued, with a random element.

The aim is to find an optimal **policy** $\pi : S \to A$ which will maximize the cumulative reward.

Models of optimality

Finite horizon reward	$\sum_{i=0}^{h} r_{t+i}$
Average reward	$\lim_{h\to\infty} \frac{1}{h} \sum_{i=0}^{h-1} r_{t+i}$
Infinite discounted reward	$\sum_{i=0}^{\infty} \gamma^i r_{t+i}$, $0 \leq \gamma < 1$

TODO insert image from slide 10, Reinforcement.

Finite horizon reward is computationally simple.

Infinite discounted reward is easier for proving theorems

Average reward is hard to deal with, because can't sensibly choose between small reward and large reward very far in the future.

Value Function

For each state $s \in S$, let $V^*(s)$ be the maximum discounted reward obtainable from s.

TODO insert image from slide 11, Reinforcement.

Learning the Value Function can help to determine the optimal strategy.

Environment Types

Environments can be passive or stochastic, active and deterministic (e.g. Chess), and active and stochastic (e.g. backgammon).

Active, stochastic environments introduce the K-armed Bandit Problem. Imagine being in a room with several friendly slot machines, for a limited time, and trying to maximize the payout. Each action (slot machine) provides a different average reward.

Most of the time, we should choose the action that we think is best (highest average reward), but in order to ensure convergence to the optimal strategy, we must occasionally choose something different from our preferred action. A simple solution can be choosing a random action 5% of the time. Something more strategic would be using a Bolzmann distribution to choose the next action: $P(a) = \frac{e^{\hat{V}(a)/T}}{\sum_{b \in A} e^{\hat{V}(b)/T}}$

Delayed Learning

TODO what. :P

Temporal Difference Learning

TD(0) (also called AHC or Widrow-Hodd Rule): $\hat{V}(s) \leftarrow \hat{V}(s) + \eta[r(s,a) + \gamma\hat{V}(\delta(s,a)) - \hat{V}(s)]$, where η is the learning rate.

The (discounted) value of the next state, plus the immediate reward, is used as the target value for the current state.

A more sophisticated version, called TD(λ), uses a weighted average of future states.

Q-Learning

For each $s \in S$, let $V^*(s)$ be the maximum discounted reward obtainable from s, and let $Q(s,a)$ be the discounted reward available by first doing action a and then acting optimally.

The optimal policy is $\pi^*(s) = argmax_a Q(s,a)$, where $Q(s,a) = r(s,a) + \gamma V^*(\delta(s,a))$

then we have $V^*(s) = max_a Q(s,a)$, so $Q(s,a) = r(s,a) + \delta max_B Q(\delta(s,a),b)$

which allows us to iteratively approximate Q by $\hat{Q}(s,a) \leftarrow r + \delta max_b \hat{Q}(\delta(s,a),b)$.

Theoretical Results

Theorem a: Q-Learning will eventually converge to the optimal policy, for any deterministic Markov decision process, assuming an appropriately randomized strategy. (Watkins and Dayan 1992)

Theorem b: TD-Learning will also converge, with probability 1. (Sutton 1988, Dayan 1992, Dayan & Sejnowski 1994).

There are limitations on theoretical results, however. We have delayed reinforcement (reward resulting from an action may not be received until several time steps later, which also slows down the learning). The search space must be finite; convergence is slow if the state space is large, and it relies on visiting every state infinitely often.

For "real world" problems, we can't rely on a lookup table. We need to have some kind of generalization.

13.2.3 Unsupervised Learning

Agent is only presented with inputs, and aims to find structure in these inputs.

Part IX

Week 9

Chapter 14

Perceptrons

14.1 Neural Networks

Neural networks in brains are networks made of neurons. Neurons are cells with a cell body (soma), dendrites (inputs), an axon (outputs), and synapses (connections between cells). Synapses can be exitatory or inhibitory, and may change over time.

When inputs reach a certain threshold an action potential (electrical pulse) is sent along the axon to the outputs.

The human brain has 100 billion neurons, with an average of 10,000 synapses each.

Artificial neural networks are made up of nodes. Nodes have input edges (weighted), output edges (also weighted), and an activation level, which is a function of the inputs.

Weights can be positive or negative and may change over time (as a result of learning). The input function is a weighted sum of the activation levels of inputs, and the activation level is a non-linear transfer function g of this input:

$$\text{activation}_i = g(s_i) = g(\textstyle\sum_j w_{ij} x_j)$$

Some nodes are inputs (sensing), some are outputs (action).

14.2 Rosenblatt Perceptron

TODO insert image from slide 10, Perceptrons.

A Rosenblatt Perceptron has inputs (x_1, x_2) and weights (w_1, w_2). A bias weight is also included (w_0), with a corresponding input (1). Finally, we have a threshold, th. We take $w_0 = -th$.

Inputs are summed

$$\begin{aligned} s &= w_1 x_1 + w_2 x_2 + w_0 \\ &= w_1 x_1 + w_2 x_2 - th \end{aligned}$$

and passed to the transfer function g to get some output $g(s)$.

14.2.1 Transfer Function

A discontinuous step function was originally used for the transfer function

$$g(s) = \begin{cases} 1, & \text{if } s \geq 0 \\ 0, & \text{if } s < 0 \end{cases}$$

Later, other transfer functions were introduced, which are continious and smooth.

14.3 Perceptron Learning

Perceptrons can compute linearly separable functions, e.g. AND, OR, NOR.

14.3.1 Perceptron Learning Rule

We can train a perceptron to learn a new function by adjusting weights as each input is presented.

In our previous example, we had $s = w_1 x_1 + w_2 x_2 + w_0$.

If $g(s) = 0$ but should be 1,

$$w_k \leftarrow w_k + \eta x_k$$
$$w_0 \leftarrow w_0 + \eta$$

so we get $s \leftarrow s + \eta(1 + \sum_k x_k^2)$.

If $g(s) = 1$ but should be 0,

$$w_k \leftarrow w_k - \eta x_k$$
$$w_0 \leftarrow w_0 - \eta$$

so we get $s - \eta(1 + \sum_k x_k^2)$.

otherwise, the weights are unchanged. (note, η is the learning rate).

Theorem: this will eventually learn to classify data correctly, as long as they are linearly separable.

There is a large limitation on perceptron learning, which is that many useful functions are not linearly separable (e.g. XOR). One possible solution is to split the required function up into other functions that can be implemented by perceptrons. For example, x_1 XOR x_2 can be written as (x_1 AND x_2) NOR (x_1 NOR x_2).

To implement a function like XOR, we use multi-layer neural networks

TODO insert image from slide 20, Perceptrons.

Chapter 15

Neural Networks

To train a neural network to learn a new function, the key idea is to replace the discontinuous step function with a differentiable function, such as the sigmoid: $g(s) = \frac{1}{1+e^{-s}}$

or hyperbolic tangent $g(s) = \tanh(s) = \frac{e^s - e^{-s}}{e^s + e^{-s}} = 2(\frac{1}{1+e^{-2s}}) - 1$

We use a forward pass as we did with simple perceptrons: weights, biases and values are passed through multiple layers to get a final output value.

TODO insert image from slide 5, NeuralNets

15.1 Gradient Descent

We include an error function E to define an error 'landscape' on the weight space. The aim is to find a set of weights for which E is very low. This is done by moving in the steepest downhill direction; $w \leftarrow w - \eta\frac{\partial E}{\partial w}$, where $E = \frac{1}{2}\sum(z-t)^2$. In words, the error function is half of the sum over all input patterns of the square of the difference between actual output and desired output. As always, we are using η as the learning rate.

This method is called "Gradient Descent".

15.2 Chain Rule

First, recall the chain rule. If we have

$$y = y(u)$$
$$u = u(x)$$

Then

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial u}\frac{\partial u}{\partial x}$$

This principle can be used to compute the partial derivatives in an efficient and localised manner. Note that the transfer function must be differentiable (usually sigmoid or tanh, from before).

Note:

$$\text{if } z(s) = \frac{1}{1+e^{-s}}, \qquad z'(s) = z(1-z)$$
$$\text{if } z(s) = tanh(s), \qquad z'(s) = 1 - z^2$$

15.3 Backpropagation

In conjunction with gradient descent, we use backpropagation (abbreviation for "backward propagation of errors"). We calculate the gradient of a loss function with respect to all the weights in the network.

We have the following partial derivatives:

$$\begin{aligned}\frac{\partial E}{\partial z} &= z - t \\ \frac{dz}{ds} &= g'(s) = z(1-z) \\ \frac{\partial s}{\partial y_1} &= v_1 \\ \frac{dy_1}{du_1} &= y_1(1-y_1)\end{aligned}$$

We introduce the following useful notation:

$$\delta_{out} = \frac{\partial E}{\partial s},\ \delta_1 = \frac{\partial E}{\partial u_1},\ \delta_2 = \frac{\partial E}{\partial u_2}$$

Combining these, we get:

$$\begin{aligned}\delta_{out} &= (z-t)z(1-z) \\ \frac{\partial E}{\partial v_1} &= \delta_{out} y_1 \\ \delta_1 &= \delta_{out} v_1 y_1 (1-y_1) \\ \frac{\partial E}{\partial w_{11}} &= \delta_1 x_1\end{aligned}$$

Partial derivatives can be calculated efficiently by backpropagating deltas through the network.

15.4 Applications of Neural Networks

Applications include

- Autonomous driving (e.g. ALVINN)
- Game playing
- Credit card fraud detection
- Handwriting recognition
- Financial Prediction

15.4.1 ALVINN

ALVINN is the Autonomous Land Vehicle In a Neural Network. It had a 30x32 sensor, which fed into a neural network with 30 output units (ranging from "sharp left" to "straight ahead" to "sharp right"). Later versions included a

sonar range finder, an 8.32 range finder input retina, with 29 hidden input units and 45 output units.

This is an example of supervised learning, from human actions (behavioural cloning), with additional "transformed" training items to cover emergency situations.

With just this, ALVINN was able to drive autonomously from coast to coast.

15.5 Variations on backpropagation

15.5.1 Cross Entropy

Problem: least squares error function is unsuitable for classification where target = 0 or 1

Mathematical theory: maximum likelihood

Solution: replace with cross entropy error function

Cross Entropy

For classification tasks, target t is either 0 or 1, so better to use

$$E = -t\log(z) - (1-t)\log(1-z)$$

This can be justified mathematically, and works well in practice - especially when negative examples vastly outweigh positive ones. It also makes the backpropagation computations simpler: $\frac{\partial E}{\partial z} = \frac{z-t}{z(1-z)}$

if $z = \frac{1}{1+e^{-s}}$, then $\frac{\partial E}{\partial s} = \frac{\partial E}{\partial z}\frac{\partial z}{\partial s} = z - t$.

Maximum Likelihood

H is a class of hypotheses.

$P(D|h)$ = probability of data D being generated under hypothesis $h \in H$.

$\log P(D|h)$ is called the likelihood.

Machine Learning principle: choose $h \in H$ which maximizes the likelihood, i.e. maximizes $P(D|h)$, or maximizes $\log P(D|h)$.

Least Squares Method - Finding a Line of Best Fit

Suppose data generated by a linear function h, plus Gaussian noise with standard deviation σ.

$$\begin{aligned}
P(D|h) &= \prod_{i=1}^{m} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2(d_i - h(x_u))^2}} \\
\log P(D|h) &= \sum_{i=1}^{m} -\frac{1}{2\sigma^2}(d_i - h(x_i))^2 - \log(\sigma) - \frac{1}{2}\log(2\pi) \\
h_{ML} &= \text{argmax}_{h\in H} \log P(D|h) \\
&= \text{argmin}_{h\in H} \sum_{i=1}^{m} (d_i - h(x_i))^2
\end{aligned}$$

(Note: we do not need to know σ)

Derivation of Cross Entropy

For classification tasks, d is either 0 or 1.

Assume D generated by hypothesis h as follows:

$$P(1|h(x_i)) = h(x_i)$$
$$P(0|h(x_i)) = (1 - h(x_i))$$

i.e. $P(d_i|h(x_i)) = h(x_i)^{d_i}(1 - h(x_i))^{1-d_i}$

then,

$$\log P(D|h) = \sum_{i=1}^{m} d_i \log h(x_i) + (1 - d_i) \log(1 - h(x_i))$$

$$h_{ML} = \text{argmax}_{h \in H} \sum_{i=1}^{m} d_i \log h(x_i) + (1 - d_i) \log(1 - h(x_i))$$

This can be generalized to multiple classes.

Bayes' Rule

H is a class of hypotheses.

$P(D|h)$ = probability of data D being generated under hypothesis $h \in H$

$P(h|D)$ = probability that h is correct, given that data D were observed.

Bayes' Theorem:

$$P(h|D)P(D) = P(D|h)P(h)$$
$$P(h|D) = \frac{P(D|h)P(h)}{P(D)}$$

$P(h)$ is called the **prior**.

Example: Medical Diagnosis Suppose we have a 98% accurate test for a type of cancer which occurs in 1% of patients. If a patient tests positive, what is the probability that they have the cancer?

$$P(positive|cancer) = 0.98$$
$$P(cancer) = 0.01$$
$$P(cancer|positive) = \frac{P(positive|cancer)P(cancer)}{P(positive)}$$
$$= \frac{0.98 \times 0.01}{0.98 \times 0.01 + P(\text{false positive}) \times 0.99}$$

15.5.2 Weight Decay

Problem: weights "blow up" and inhibit further learning

Mathematical theory: Bayes' rule

Solution: add weight decay term to error function

Assume that small weights are more likely to occur than large weights, i.e.

$$P(w) = \frac{1}{Z} e^{-\frac{\lambda}{2} \sum_j w_j^2}$$

where Z is a normalizing constant. Then the cost function becomes:

$$E = \frac{1}{2}\sum_i (z_i - t_i)^2 + \frac{\lambda}{2}\sum_j w_j^2$$

This can prevent the weights from "saturating" to very high values.

Problem: need to determine λ from experience, or empirically.

15.5.3 Momentum

Problem: weights oscillate in a "rain gutter"

Solution: weighted average of gradient over time

If landscape is shaped like a "rain gutter", weights will tend to oscillate without much improvement.

Solution: add a momentum factor

$$\delta w \leftarrow \alpha \delta w + (1 - \alpha)\frac{\partial E}{\partial w}$$
$$w \leftarrow w - \eta \delta w$$

Hopefully, this will dampen sideways oscillations but amplify downhill motion by $\frac{1}{1-\alpha}$.

15.6 Conjugate Gradients

Compute matrix of second derivatives $\frac{\partial^2 E}{\partial w_i \partial w_j}$ (called the Hessian).

Approximate the landscape with a quadratic function (parabaloid).

Jump to the minimum of this quadratic function.

15.7 Natural Gradients (Amari, 1995)

Use methods from information geometry to find a "natural" re-scaling of the partial derivatives.

15.8 Training Tips

- Rescale inputs and outputs to be in the range 0 to 1 or -1 to 1
- Initialize weights to very small random values
- On-line or batch learning
- Three different ways to prevent overfitting:
  - limit the number of hidden nodes or connections
  - limit the training time, using a validation set
  - weight decay
- Adjust learning rate and momentum to suit the particular task

Chapter 16

Temporal Difference Learning

Temporal difference learning is a prediction method. Mostly used to solve the reinforcement learning problem, TD learning is a combination of Monte Carlo ideas and dynamic programming ideas.

16.1 Backgammon

Imagine a backgammon neural network; we have 196 inputs, 20 hidden units, 1 output unit in a two-layer neural network. The board encoding is 4 units $\times$ 2 player $\times$ 24 points, 2 units for the bar, and 2 units for off the board.

The input s is the encoded board postition (state), and the output $V(s)$ is the value of this position (the probability of winning).

For backgammon play, we consider the following:

Question: how do we play?

Answer: at each move, roll the dice, find all possible "next board positions", convert them to the appropriate input format, feed them to the network, and choose the one which produces the largest output.

Question: how do we train the network?

Answer: by supervised learning (from expert preferences) or by temporal difference learning (from self-play).

16.2 Backpropagation

$$w \leftarrow w + \eta(T - V)\frac{\partial V}{\partial w}$$

V = actual output
T = target output
w = weight
η = learning rate.

How do we choose T?

- learn move from example games?
- T = final outcome of game?

- Temporal Difference Learning (Sutton)

16.3 Temporal Difference Learning

(current estimate) $V_k \to V_{k+1} \to ... \to V_m \to V_{m+1}$ (final result)

$TD(0)$: use V_{k+1} as the training value for V_k

$TD(\lambda)$: use T_k as the training value for V_k, where

$$T_k = (1-\lambda)\sum_{t=k+1}^{m} \lambda^{t-1-k} V_t + \lambda^{m-k} V_{m+1}$$

T_k is a weighted average of future estimates, λ = discount factor ($0 \leq \lambda < 1$).

TD-learning can be seen as an alternative to Q-learning, but it learns only a value function $V(s)$, rather than a Q-function $Q(s,a)$.

16.4 TD-Gammon

Question:Why is it better to learn from next postition instead of final outcome?

Answer: If we learn from next position, we won't assign credit indiscriminately in the case where we go from a bad move, to a good move, to a win.

Tesauro trained two networks: EP-network was trained on Expert Preferences. TD-network was trained by self play.

TD-network outperformed the EP-network.

With modifications such as 3-step lookahead and additional hand-crafted input features, TD-Gammon became the best Backgammon player in the world (Tesauro, 1995).

16.5 General Ideas

Other games have been trained by TD-learning, but generally against humans rather than self-play.

Evolutionary algorithms can also produce surprisingly strong players, but a gradient-based method such as TD-learning is better able to fine-tune the rarely used weights, and exploit the limited nonlinear capabilities of the neural network.

A more recent algorithm called TreeStrap learns at all branches of the (alpha-beta) game tree. It can learn chesse to master-level play (but relies on knowing the "world model", i.e. the rules of the game).