Artificial Intelligence Revision

Logic Programming *week 2*

Facts and rules infer new facts by asking questions. The system searches the fact database to determine a question's answer by logical deduction, e.g. from "x > y" and "y > z", we can infer "x > z". Inference procedures tell which statements are valid, inferred from others.

Syntax is how something is written, while semantics is what something means.

The syntax of propositional logic consists of propositional symbols (e.g. A, B, C, D), and connectives (∧ (and), ∨ (or), → (then), ¬ (not), ≡ (is the same as)). Its semantics is the assignment of a truth value (true or false) to each statement. For example:

A = "it is night", B = "it is dark", C = "the light is on",

(A ∧ B) → C = "if it is night and it is dark, then the light is on",

B → A = "if it is dark, then it is night",

A proof is a sequence of statements (each being either a premise or derived from an earlier statement by on of the inference rules). A goal is the statement we are trying to prove. For example:

Premises*: if it's night, the room is dark. If the room is dark, the light turns on. The light is not on.*

Goal*: is it night?*

A = it is night, B = the room is dark, C = the light is on.

Given: A→B, B→C, ¬C,

The following truth table shows that A = false, therefore it is not night.

Variables			Given			Trial conclusions	
A	**B**	**C**	**A→B**	**B→C**	**¬C**	**A**	**¬A**
T	T	T	T	T	F	T	F
T	T	F	T	F	T	T	F
T	F	T	F	T	F	T	F
T	F	F	F	T	T	T	F
F	T	T	T	T	F	F	T
F	T	F	T	F	T	F	T
F	F	T	T	T	F	F	T
F	**F**	**F**	**T**	**T**	**T**	**F**	**T**

The problem with truth tables is that the number of rows grows exponentially as the number of propositional values increases (2^n).

A proof procedure is a method of proving statements using inference rules:

1. modus ponens: if A→B and A are true, infer B is true,

2. modus tollens: if A→B and ¬A are true, infer ¬B is true,

3. elimination: if both (A ∧ B) is true, then infer both A and B are true,

4. introduction: if both A and B are true, then infer (A ∧ B),

5. disjunctive syllogism: if (A ∨ B) and ¬A are true, then infer B is true,

For example:

Premises*: if it's night, the room is dark. If the room is dark, the light turns on. the light is not on.* Goal*: is it night?*

A = it is night, B = the room is dark, C = the light is on,

given: A→B, B→C, ¬C,

B→C *and* ¬C, *therefore* ¬B (*modus tollens*)

now, A→B and ¬B, *therefore* ¬A (*modus tollens*)

A is false, it is not night.

The limitation of propositional logic is that it cannot represent, e.g. "x > y" and "y > z" therefore "x > z". To resolve this, extend representation (add predicates), extend operators, and add unification.

In predicate calculus, symbols can consist of any letter, any digit, and underscores, representing constants, functions, predicates (all begin with lowercase), or variables (begin with uppercase). Constants, functions, and variables are known as terms.

Constants name specific objects or properties, variables assign general classes of objects or properties, and functions combine variables and classes, e.g. man(bob), woman(alice). Replacing a function with its value is called evaluation, e.g. plus(2,3) whose value is 5.

Predicates name relationships between objects, e.g. likes(bob, alice). They are special functions with true/false values. The same predicate name with different values is considered distinct.

e.g. ***constants***, ***variables***, ***functions***, ***predicates***
***likes*(*bob*, *alice*), *likes*(*X*, *Y*), *likes*(*father_of*(*bob*))**

Terms are mapped to objects in their domain. Predicate calculus' semantics determine an expression's truth value. The inference system must be able to determine when two expressions match. Two expressions match if they are syntactically identical.

Unification determines the substitution list to make two predicate expressions match. If p and q are logical expressions, unify(p, q) gives a substitution list that either makes p and q identical or fails. The notation X/Y indicates that X can be substituted for Y in the original expression. A substitution is assigned to values and variables. Two terms unify if there is a substitution that makes the two terms identical, e.g. unifying f(X, 2) and f(3, Y) can be written as 3/X, 2/Y.

The process is complicated by the existence of variables, which can be replaced by terms (other variables, constants, or function expressions), e.g.

Unify	Substitutions	Outcome
p(a, X) and p(a, b)	b/X	p(a, b)
p(a, X) and p(Y, b)	a/Y, b/X	p(a, b)
p(a, X) and p(Y, f(Y))	a/Y, f(a)/X	p(a, f(a))
p(a, X) and p(X, b)	a/X, b/X *(X can't be both)*	Fail
p(a, b) and p(X, X)	a/X, b/X	Fail

Logic Programming *week 3*

Prolog itself won't be in the exam, but the logic behind it (see above) will be.

Prolog proves goals by matching them with rules/facts. It tries to find variable bindings making expressions identical.

Backtracking is the process of tracing steps backwards to previous goals and re-satisfying them when a goal fails. Prolog goes through facts/rules top to bottom to try to find matches, keeping track of where it has got to.

Artificial Neural Networks *week 4* & *week 5*

Biological neurons have cell bodies, dendrites (input structures), and axons (output structures). Axons connect to dendrites via synapses. Electro-chemical signals travel from the dendrite, through the cell body, to the axon, then onto other neurons. Neurons only fire if their input signals exceed a threshold in a short period of time. Synapses vary in strength: strong connections allow large signals while weak connections only allow small signals.

Artificial neural networks emulate a biological neural system, consisting of nodes ('neurons') and weights ('neuronal connections'). Each node has weighted connections to several other nodes in adjacent layers and takes input from connected nodes, using the weighted inputs combined and a simple function to compute its output values. Knowledge is stored in the connections between neurons.

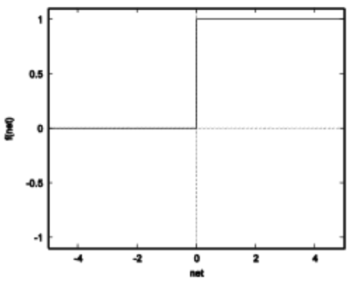
To train a neural network: introduce data; the network computes an output; the output is compared to the desired output; the network's weights are modified based on the difference between the two to reduce error. To use a neural network: introduce new data to the network; the network computes an output based on its training.

In a single layer network, an adder sums up all the inputs, modified by their respective weights (linear combination). An activation function controls the amplitude of the output of the neuron. An acceptable output range is usually between -1 or 0 and 1.

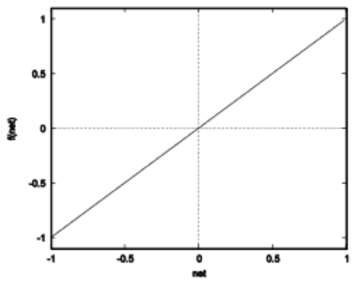
Mathematical model:
$I_1 \ldots I_n$ = inputs,
$w_1 \ldots w_n$ = weights,
net: summation,
f = activation function,
θ = bias/threshold,
O = output,

$$\bar{\mathbf{I}} = \begin{bmatrix} I_1 \\ I_2 \\ \vdots \\ I_n \end{bmatrix} \qquad \bar{\mathbf{w}} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix} \qquad O = f\left(\sum_{j=1}^{n} w_j I_j - \theta\right) = f\left(\bar{\mathbf{w}}^T \bar{\mathbf{I}} - \theta\right) = f(net)$$

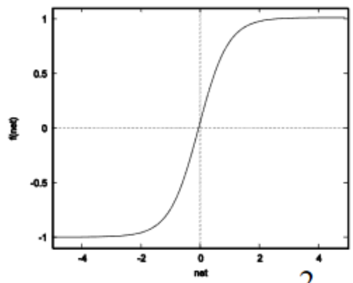
Activation functions:



Step: $f(net) = \begin{cases} 1, & net \geq 0 \\ 0, & net < 0 \end{cases}$



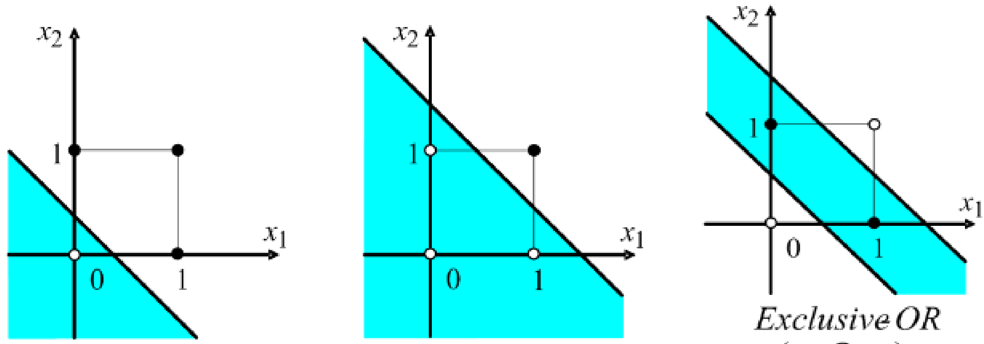
Linear: $f(net) = net$



Sigmoid: $f(net) = \dfrac{2}{1 + e^{-\lambda net}} - 1$

Bias can be implemented as an extra input. Connection strengths are modelled by a set of weights. The training process involves changing the weights and bias values.

Single layer networks are simple and easy to implement and learn quickly: several examples are usually enough. However, they can only learn linearly separable functions. The graphs below show how three Boolean problems should be split, however a single layer network could only perform the left-most two as the third requires two lines.



$OR\ (x_1 \cup x_2)$ $AND\ (x_1 \cap x_2)$ *Exclusive OR* $(x_1 \oplus x_2)$

Multi-layer neural networks are feedforward networks. They consist of an input layer of source neurons, at least one middle or hidden layers of computational neurons, and an output layer of computational neurons. Input signals are propagated in a forward direction on a layer-by-layer basis.

These networks can learn non-linear relationships, can solve 'difficult' problems (e.g. classification, stock prediction), and can usually run well with just one hidden layer. They can, however, be very slow as they have more complex learning methods and require lots of training data.

Artificial neural networks should be used when you are working with lots of data, non-linear and multidimensional input/output mapping, and lots of time. Learning rates should be very small, around 0.1. Artificial neural networks have many uses, for example: pattern recognition, clustering, optimisation, control, and medical or business applications.

Their processing is massively parallel, meaning they are good for real-time applications. Artificial neural networks are self-trainable (they learn by themselves), only needing to increase or decrease connection strengths. They are excellent for generalisations and uncertain information, working well with noisy data. Once they have learned from data, they don't need to be reprogrammed.

However, artificial neural networks have high processing times if they are large, times which can rise quickly as the problem size grows. They don't explain their results, may not be guaranteed to converge to an optimal solution, and can possibly be over-trained.

Here's a link to week 5's step-by step run through on one page.

Games as a Context for AI *week 6*

Games AI should be quick (for real-time use), use predictable resources, and have understandable behaviour (giving the ability to design game play). They should be believable – not too stupid but not too clever. The player's experience is built on 'suspension of disbelief'. This, however, is not always the case with non-games AI.

Pathfinding or 'search' is a generic problem, not only applicable to games, e.g. robots, routing network packets, travel planning. It usually works on a node graph.

Breadth first is visiting each level, one by one. Depth first is visiting each child node to its maximum depth. The implementation is quite similar: depth first uses a stack to expand child nodes, while breadth first uses a queue. Both have the same big O complexity.

In games, breadth first is usually most suited as we often want to look a fixed number of moves ahead, however depth first uses less memory and is slightly easier to implement.

Dijkstra's Algorithm *week 7*

Key ideas with Dijkstra's: planning (it works out the complete route in advance), divide and conquer (decomposes the planning into an iterative process). It eliminates suboptimal search directions early in the search process. Dijkstra's explores all possible paths, including dead ends and works on general graphs (not just grids).

To perform Dijkstra's algorithm, we need a set of nodes (the map) and pair costs to traverse from one node to its neighbours. We obviously need to know the start and end nodes. For each node, we keep the total cost from the start node: the current shortest total distance from the start and a link to the neighbour that got us to that node with the lowest cost. The link could just be the neighbour's coordinates. We also need two lists of nodes, those that are open (not yet found their shortest route) and those that are closed (have found the shortest route).

Before starting the algorithm, each node's total costs should be set to either infinity or a very large number. The total cost of the start node should be zero. Links don't matter, but all nodes should be in the open list.

To find the path, do the following, iteratively:

1. Find the node, N, in the open list with the current lowest total cost.
2. Move that node to the closed list (take its current estimate as final).
3. Re-estimate the cost of each open neighbour.
4. If the new estimate is lower than the current total, update it and link to N.

When the end node is closed, we are finished. Follow the links from the end node back to the start node to get the shortest path.

The limitations of Dijkstra's are that it needs to work out the entire solution in advance, it doesn't adapt to changes in the environment, it doesn't adapt to changes in the target position, and it doesn't take anything else moving in the environment into account.

A* modifies Dijkstra's to search more likely paths first in a fairly simple way: adding a heuristic function (a guess as to how far each node is from the target) to the open node total costs when choosing the next node to close.

The new cost function is the cost from the start for a node (determined as in Dijkstra's) plus the heuristic function for that node (an estimate of the minimum cost to get to the target). The heuristic function can be defined in many ways.

A* and Further Optimisations *week 8*

The heuristic function should be admissible – it should not over-estimate the actual shortest distance to the target from the node, otherwise it isn't guaranteed to get the actual shortest path.

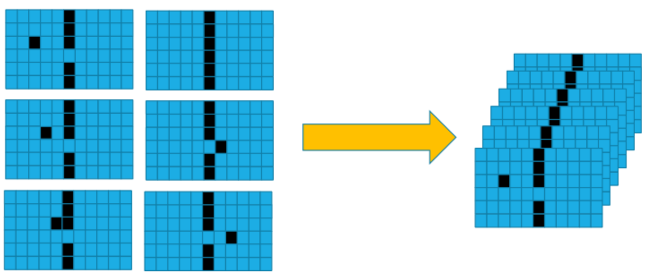
Manhattan distance: difference in x + difference in y.

Diagonal distance: (dx + dy) – 0.6 * smallest of (dx, dy), for a square grid

Euclidian distance: $\text{sqrt}((dx)^2 + (dy)^2)$

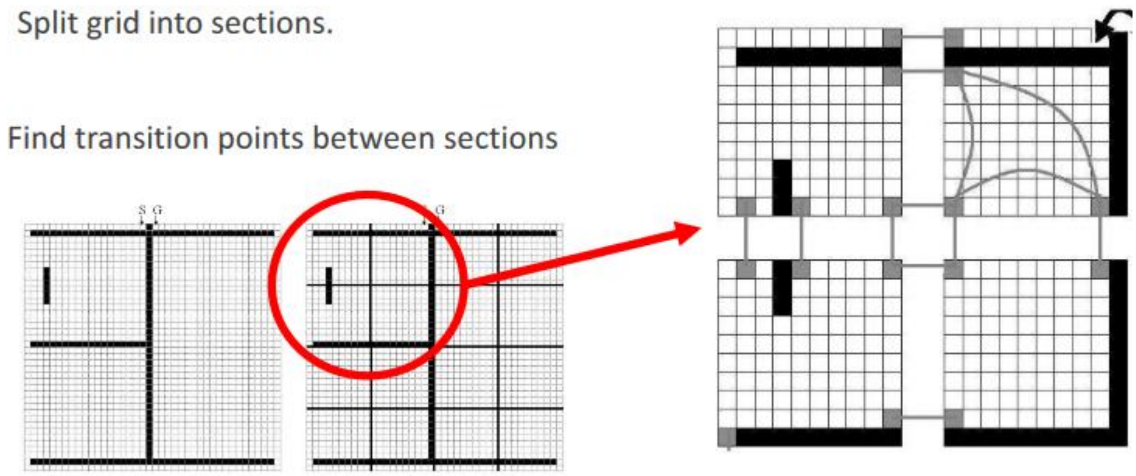
As A* is an optimisation of Dijkstra, it has the same limitations. If the environment changes, we may have to re-run the algorithm, but that could mean lots of recalculations. We can path patch, only rerunning the algorithm from where the environment change has happened. This can save some computational overhead, but it doesn't look as believable.

With two A* agents at the same time, we can convert a 2D map to a 3D map. Each layer of the 3D map includes the standard environment, with the first agent's position on a different layer per time-step. This then allows us to shift the second agent to a different layer after each step, meaning it sees the first agent's position as an obstacle and avoids it based on its position at that specific time. This works if the first agent runs its algorithm first, then a 3D map is created, then the second agent runs its algorithm.

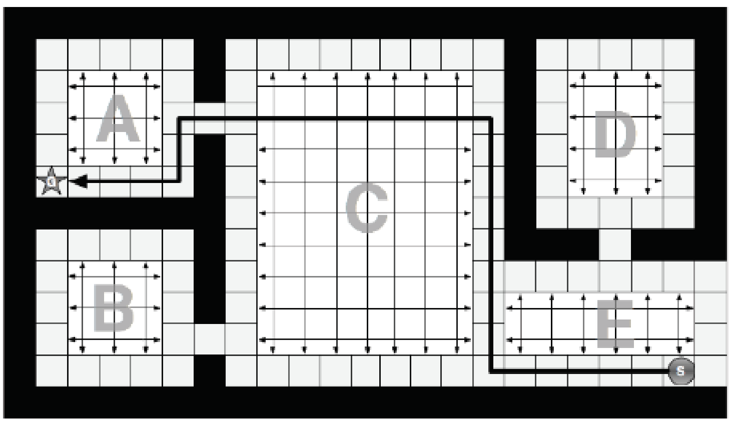


The problems with a 3D approach are that optimal paths are not guaranteed, but this makes the agents somewhat believable. If the agents move at different speeds, this could be an issue. Paths generally get longer and can't be easily patched.

A* is still not very fast. We can pre-calculate routes on the map (but this doesn't respond to any environmental changes) or we can break the problem down hierarchically. This splits maps into sections, each with set inter-sectional routes. These routes can be pre-calculated, with only the final section needing a real-time calculation, as the target is within that section, rather than in a neighbour. As this is pre-calculated, it is quicker and can be patched more easily. It is also faster (10x in some cases) and nearly as optimal (99%).



Another way of improving the search's speed is symmetry. Rectangular Symmetry Reduction is a simple algorithm to help optimise searches. It identifies rectangular blocks of clear nodes, leaving edge nodes in place. RSR then creates direct connections (bridges) between nodes on opposite sides of the clear rectangles and searches using the edges and bridges only.



AI Planning *week 9*

Requirements for a planning problem: initial state and goal state; actions, applied to change from one state to another; preconditions, the previous state before an action is applied; effect, the new state after an action is applied.

In a planning problem, you are given the initial and goal state and are asked to find a sequence of actions leading from the initial state to the goal state. You can create a graph of all possible sequences which can be searched breadth first, depth first, or with a heuristic search.

Sometimes, actions are also associated with parameters (objects that actions are performed on).

AI Planning *week 10*

Heuristic functions evaluate the distance to the goal state and are used to search the shortest path to a goal.

Bayesian Reasoning *week 14*

-