Agents Any device that can sense and act upon its environment
Rational: for any percept sequence acts so as to max. expected performance
Autonomous: behaviour depends on own percept sequence
Reflex: act on current percept

Search Start state, actions (with known results), goal test function
Test the root to see if it is a goal, if not then expand, move to one of the resulting states: states not yet expanded called *fringe*

Breadth-First Complete, optimal if path cost non-decreasing function of depth, memory and time complexity $O(x^d)$ for shortest depth d

Uniform Cost Always expand node with lowest path cost first. Optimal if path cost of node is at least that of its parent node

Depth-First Not complete or optimal, memory O(xd) and time $O(x^d)$ for depth d
Good if there are many solutions

Back-tracking If each node knows how to generate the next possibility, memory is O(d). Can optimise by having undoable actions which modify a shared state

Depth-Limited As depth-first but complete if a solution is within the depth

Iterative Deep. Complete and optimal, memory O(xd), time still exponential but less than naïve breadth-first since most nodes are in bottom layer

Bi-direction If methods are $O(x^d)$ then we can convert this to $O(2x^{d/2})$, but might not be possible esp. if > 1 goal: also need to store all nodes of one of the searches to test meet

Graph Have a closed list of expanded nodes, never add a node to it twice. Time and space complexity proportional to state space size (esp. problem with depth-first), may discard new state that is better than an older one

Best-First Expand nodes using ordering of evaluation function. Can use heuristic (w/ h(goal) = 0) for greedy search, but time and space complexity $O(x^d)$ and not optimal or complete!

A* f(n) = p(n) + h(n) with known path cost, p and estimated cost of path to goal state, h
Admissible: if h(n) never overestimates the cost of best path from n to a goal
Optimal if h is admissible (show this by contradiction)

A* Graph Monotonic: if f(n′) ≥ f(n) for n′ following n, can use pathmax f(n′)=max{f(n), p(n′) + h(n′)}. Same as triangle inequality, implies admissibility
Optimal if h is monotonic since everything with f(n″) < f_{opt} gets explored first, then one or more such things get found
Complete provided branching factor finite and there is a c st. each operator has a cost at least c
No other optimal algorithm that constructs paths from the root can guarantee to examine fewer nodes, but still has exponential complexity unless |h(n)-h′(n)| ≤ O(log[h′(n)])

IDA* Iterative deepening with A*
Does not require queue of nodes: space O(p) where p is the longest path length, time taken depends on the # of values h can take
Heart is a `contour` function which returns node found (if any) and minimum f limit required to progress at least 1 node

RBFS Remember the f(n) for the best alternative node we've seen on the way to current node n′. If f(n′) > f(n) then go back and explore the best alternative, replacing f cost of every node in path with f(n′) to remember path goodness (note: must ensure f remains monotonic under replacement)
Optimal if h admisable, memory = O(xd), time can be exponential

Games Can be modelled with search

MiniMax If A is rational he plays to reach a position with maximum utility, if B is rational she plays to minimise the utility available to A
Generate complete tree and work

from leaves upwards computing utility: has time $O(x^p)$ for p-ply
Introduce cutoff test, evaluation function to limit tree generated
Evaluation f typ. weighted linear

Alpha-Beta Pruning Alpha: highest utility seen so far for Max, Beta: lowest utility seen so far for Min. If α ≥ β at any point we can stop the search

```
Max(alpha, beta, node) {
 For (successor s) { Alpha =
max_of(alpha, min(alpha, beta, s))
  Return beta if alpha >= beta }
 Return alpha
```

Try good moves first: if ordering perfect then $O(x^{p/2})$, $O(x^{3p/4})$ for realistic x and random order

Constraint Satisfaction Set of variables $V_1, V_2..V_n$
Domain D_i for each V_i
Constraints $C_1, C_2,..C_m$
State: assignment of values to variables, consistent if violates no constraints, complete if assigns a value to all variables
Binary constraints: with finite domains this is sufficient even for higher order constraints

Backtrack. Search Depth first, single variable at a time, backtrack when no assignment is possible
Minimum remaining values: assign such variables first
Degree heuristic: choose the variable involved in the most constraints on yet unassigned variables (good tie breaker)
Least constraining value: choose variable value that gives max neighbour freedom

Forward Checking When we assign a value to a variable, delete the value from the current domains of its neighbours (good with MRV)

Constraint Propagation Arc consistency: $i \Rightarrow j$ is consistent if for all assignments to i, can assign something to j
Enforce this each time a variable is assigned (may need cascade): called AC-3, $O(n^2d^3)$
K-consistency: if we have k-1 variables w/ a consistent assignment then we can find a consistent assignment to any kth variable
Strong k-consistency: if k consistent and strong k-1 consistency. Find assignment in O(nd), n = var. count

Back-jumping Backtracks to the conflict set: set of assigned variables connected to x
Accumulate conflicts as we make assignments: when we cause the trimming of another vars domain we join their set, if remove last then their conflict set joins ours
Forward checking makes this redundant, but can redefine conflict set to be collection of preceding variables causing x not to have a valid assignment set: when backtracking to x from x' union x set with conflict set of x' (without x itself)

Planning Planners can add actions anywhere, their state descriptions are not complete, assume element independence

STRIPS States: conjunctions of ground literals with no functions
Goals: conjunctions of literals with existentially quantified free vars.
Operators: tuples of description (name), precondition (conjunction of positive literals), effect (conjunction of literals)
Plans: tuples of set of steps (operator instantiations), ordering constraints between steps, variable bindings (to variables or constants), causal links (which preconditions of steps our steps achieve)
Initial plan: just has Start (effects: start state) and Finish (preconditions: goal state) steps and Start < Finish
Complete: every precondition is achieved by a causal link with associated order, unless some step exists that cancels it
Consistent: variable binding is a function and ordering consistent

Threats A step that might invalidate a precondition: can try and

eliminate with ordering constraint
Order before causal link is demotion, else promotion
In general, may have to introduce other steps: implies backtracking

Objects Frames or semantic networks: objects with relationships
Subclass, instance relationships control inference paths
Frames have slots with slot values, starred slots are defaults for instances / subclasses
MI, frames as slot values etc..

Rule Systems If-then rules (implication), with facts which the system "knows"

Forward Chaining Find rules that can fire based on current working memory, choose one to fire, update WM until halt

Backward Chaining Find the goal and find rules that would achieve it, backtrack if you have to make a choice and turn out to be wrong

Conflict Resolution Rule choice affects the outcome, avoid inferring useless info
Prefer specifics, recent facts

Reason Maint. When facts removed from WM need to remove old inferences

Neural Networks Feature vector: list of information, continuous/discrete
Training sequence: list of pairs of features with Ω
Hypothesis: function from feature vectors to Ω
Hypothesis space: hypotheses available to learning algorithm L
Target concept: the perfect hypothesis

Generaliz. Ability of L to pick hypothesis which is close to concept
Give a distribution P to X*Ω
Assume examples are IID
Measure error L(h, (x, y)), then set er(h) to be expected L

Perceptron $h(\vec{x}) = \mathrm{sgn}(\vec{w}^T \vec{x} + w_0)$
$R = \max_i |x_i|$, describes hypersphere which training data lies in

Primal Each round, for each misclassification $\vec{w} = \vec{w} + \eta y_i \vec{x}_i$, $w_0 = w_0 + \eta y_i R^2$ where y_i is sign of the error (-1 = too big)
Novikoff: this will converge in not more than $(\frac{2R}{\gamma})^2$ mistakes if linearly separable and exists a normalized hyperplane for all i with $y_i(\vec{w}^T \vec{x}_i + w_0) \geq \gamma$

Dual Form If η = 1 can characterise final weights as $\sum_{i=1}^{m} \alpha_i y_i \vec{x}_i$, and we can rewrite the hypothesis as
$h(\vec{x}) = \mathrm{sgn}(\sum_{i=1}^{m} \alpha_i y_i (\vec{x}_i^T \vec{x}) + w_0)$
Core training loop is as below:

```
For (example I in s)
{ if (misclassification) {
ai++; w0 += …; } }
```

Can map to a bigger space by setting up a Φ, hypothesis is:
$h(\vec{x}) = \mathrm{sgn}(\sum_{i=1}^{m} \alpha_i y_i \vec{\Phi}(\vec{x}_i)^T \vec{\Phi}(\vec{x}) + w_0)$
The sum is potentially smaller, but the Φ multiplication may be calculated multiple times
A kernel is a function K such that $K(\vec{x}, \vec{y}) = \vec{\Phi}(\vec{x})^T \vec{\Phi}(\vec{y})$: design this to be calculated easily, and so that d (dimension) does not effect kernel calculation: makes h easy

Gradient Descent Extend x, w with 1, w_0 for bias
Define a measure of error E for weights, involving sample space
Take random initial **w** and iterate: $\vec{w}_{i+1} = \vec{w}_i - \eta \frac{\partial E(\vec{w})}{\partial \vec{w}}\Big|_{\vec{w}_i}$: if E has a global minima we fall into it
For each node j: $w_i^{(j)}$ is weight from input i, $a_j = \sum_i w_i^{(j)} z_i$ is activation for j, g is activation function, $z_j = g(a_j)$ is output
$\frac{\partial E(\vec{w})}{\partial \vec{w}} = \sum_{p=1}^{m} \frac{\partial E_p(\vec{w})}{\partial \vec{w}}$ for example p

Forward Propagat. Place p at the inputs and get a_j, z_j for all nodes j

Backward Propagat. $\frac{\partial E_p(\vec{w})}{\partial w_i^{(j)}} = \frac{\partial E_p(\vec{w})}{\partial a_j} \frac{\partial a_j}{\partial w_i^{(j)}}$
$= \delta_j (\frac{\partial}{\partial w_i^{(j)}} (\sum_i w_i^{(j)} z_i)) = \delta_j z_i$

Output Node $\delta_j = \frac{\partial E_p(\vec{w})}{\partial a_j} = \frac{\partial E_p(\vec{w})}{\partial z_j} \frac{\partial z_j}{\partial a_j} = \frac{\partial E_p(\vec{w})}{\partial z_j} g'(a_j)$

Input Node Where k are the nodes which connect back to us, we can say:

$$\delta_j = \frac{\partial E_p(\vec{w})}{\partial a_j} = \sum_k \frac{\partial E_p(\vec{w})}{\partial a_k} \frac{\partial a_k}{\partial a_j} = \sum_k \delta_k \frac{\partial a_k}{\partial a_j}$$
$$= \sum_k \delta_k \frac{\partial}{\partial a_j} \left(\sum_i w_i^{(k)} z_i\right) = \sum_k \delta_k w_j^{(k)} g'(a_j)$$
$$= g'(a_j) \sum_k \delta_k w_j^{(k)}$$

And now we have enough information to perform the algorithm for training outlined above