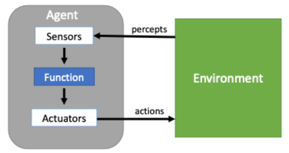
<u>Intelligent Agents</u>

- Feedback loops between AI and the world
- Maps from percept histories to actions

$$[f: P^* \to A]$$

- Runs on physical architecture to produce function f



Performance measures:

- Best for whom?
- What are we optimizing?
- What information is available?
- Any unintended effects?
- What are the costs?

Rational agent: does not require all the information about the environment; chooses actions that maximize performance measure
PEAS: Performance Measure, Environment, Actuators, Sensors
Sensors: camera, LIDAR, speedometer, i.e. hardware/input to detect information about the environment
Actuators: steering wheel, accelerator, brake, i.e. hardware/output to interact with the environment
Performance measure: defines the goodness of a solution; safety, speed, legal, comfort
Environment: defines what an agent can and cannot do; roads, weather, vehicles, obstacles
Properties of task environment:

1. Full/partial observability: agent's sensors give it access to the complete state of the environment at each point in time
2. Deterministic/stochastic: next state of the environment is completely determined by the current state and the action executed by the agent
   - Strategic environment: deterministic except for actions of other agents
3. Episodic/sequential: agent's experience is divided into atomic "episodes" where each episode consists of the agent perceiving and then performing an action, and the choice of action in each episode depends on the episode itself
4. Static/dynamic: environment is unchanged while an agent is deliberating (semi-dynamic if environment does not change but performance score does)
5. Discrete/continuous: limited number of distinct, clearly defined percepts and actions
6. Single/multi agent: an agent operating by itself in an environment

Exploitation vs Exploration: agent must choose between maximizing expected utility according to its current knowledge about the world (exploitation) and trying to learn more about the world (exploration)
<u>Structure of agents</u>

- Completely specified by the agent function mapping percept sequences to actions

Simple reflex agent: moving based on conditions (fixed)
Model-based agent: try all possible moves and pick the one that does not result in immediate termination
Goal-based agent: exploring the options to see which ones reach the end goal
Utility-based agent: like goal-based agent but each action and outcome is weighted so an "ideal" path is taken instead (e.g. dying is negatively weighted and reaching the goal is positively weighted)
Learning agent: mix of different agents like reflex, model-based, goal-based, and utility-based
Problem-solving agent: consider a sequence of actions that form a path to a goal state (via a search)
<u>Problem-solving agent</u>

1. Goal formulation
2. Problem formulation: create an abstract model of the relevant parts of the world
3. Search: simulates sequence of actions using the model, search until goal is reached
4. Execution: execute actions in the solution, one at a time

Assumptions: task environment is fully observable and deterministic

- Solution is a fixed sequence of actions

Problem formulation:

1. States (state space): viewing, at X, positions, locations, etc.
   - **Don't store the "moves" made**
2. Initial state: initial state of the agent
3. Goal state(s)/test: target outcome **(okay to say this is unknown)**
4. Actions: things that the agent can do
5. Transition model: what each action does
6. Action cost function: the cost of performing an action

Search algorithms: takes in a search problem as input and returns a solution/failure and it is defined by the order of node expansion

- **Evaluation criteria:**
  - Time complexity: number of nodes generated (when adding to queue) or expanded (when visiting)
  - Space complexity: maximum number of nodes in memory
  - **Completeness: does it return a solution if it exists?**
  - Optimality: does it always find the least-cost solution?
- **Measure:** branching factor (b), depth (d), and maximum depth (m)

Tree search: may revisit previous states, causing search tree to be infinite

- **Requires terminating condition otherwise may run forever (all searches may not work)**

Graph search: only useful if we have repeated states

- Consider if additional actions have to be made to optimize graph search efficiency which may increase time complexity

Graph search with visited checking while pushing new states: expand less states but may less cost skip states
Graph search with visited checking before pushing new states: expand more states but never skip less cost states
Representation invariant: conditions that ensure that the abstract states have corresponding concrete states (i.e. boundary conditions)
<u>Uninformed search algorithms</u>

- Given no clue about how close a state is to the goal
- E.g. BFS (does it level-wise), uniform-cost search (using a priority queue instead), DFS

Improving performance: pruning by prioritizing more likely candidates to the left or making invariant stricter to terminate branch earlier
Search strategy:

1. Number of goal states
2. Distribution of goal states in search tree
3. Finite/infinite branching factor/depth (number of actions^k)
4. Repeated states
5. Need for optimality?
6. Need to know if there's no solution?

Repeated states: keep a "visited" set to avoid revisiting nodes

- Useful for graph searches (not really needed for trees)
- If adding to visited while visiting neighbors, can reduce space but skip over states with less cost
- If adding to visited after visited, will use more space but never skip states with less cost

	Time Complexity	Space Complexity	Complete?	Optimal?
BFS	O(b^d)	O(b^d)	Yes, if B is finite	Yes, if step cost is consistent
	Always finds shortest path, uses more space than DFS if finite search; consider pre-processing (like greedy)			
Uniform	O(b^(C*/e))	O(b^(C*/e))	Yes, if e > 0 and C* is finite	Yes, if e > 0
	Useful if decisions (i.e. edges) can be mapped as different weights			
DFS	O(b^m)	O(bm)	No, when depth is infinite or can go back and forth (loops)	No
	No guarantee to find shortest path, uses less space if finite (compared to BFS); may not find solution; complete if tree is finite			
DLS	O(b^l)	O(bl)	No, see DFS	No, see DFS
	May not be useful if cannot find solution when depth limited or <u>depth to limit is unknown</u>; does not guarantee minimal solution			
IDS	O(b^d)	O(bd)	Yes	Yes, if step cost is consistent
	Paying extra for no good reason			

Uniform-cost search: BFS but continue adding the edge weight while visiting so that the cost increases and the min heap weights it later

- e = 0 can cause zero cost cycles
- C*/e refers to the number of tiers that run where C* is the cost of the optimal solution and e is the minimum edge cost
- Each cost refers to the minimum path cost from root to the node
- Edge weight of 0 may result in zero-cost cycles that cause infinite loops (if using tree search)
- Similar to Dijkstra but only cares about S -> G

Depth-limited search (DLS): limit the search depth and backtrack once depth limit is reached (for DFS)

- Not the same as depth-limited BFS

Iterative deepening search (IDS): depth-limited search with max depth 0..N and return solution if found

- Increase the depth otherwise
- Repeats the computation too many times (introducing overhead)

Backward search: search from the goal and reach root
Bidirectional search: start from root and find goal and start from goal and find root

- Stop when two searches meet
- Intuitively: 2 x O(b^(d/2)) < O(b^d)
- Issues: operators need to be reversible, many goal states, how to efficiently check if a node appears in another search tree?

<u>Informed search algorithms</u>

- Use domain information to guide the search
- **Don't use if no solution at all (so just find configuration with minimum violation); also don't use if search space is very huge since informed search can take very long**

	Time Complexity	Space Complexity	Complete?	Optimal?
Greedy Best-fit Search	O(b^m), good heuristics gives improvement	O(b^m), keep all nodes in memory	No	No, does not account for cost so far
A*	O(b^m)	O(b^m)	Yes	Yes, if heuristic is admissible (tree) or consistent (graph)

Best-first search: using an evaluation function + priority queue to decide the "goodness" of a state

- Special cases: greedy best-fit search and A* search

Greedy best-fit search: use a heuristic to estimate the cost from n to goal instead of actual evaluation function (f(n) = h(n))
A* search: f(n) = g(n) + h(n) where g(n) is the cost so far to reach n

- Works with negative weights but not negative weight cycles
- Variants
  - Iterative deepening A* (IDA*): use iterative deepening search with cutoff using f-cost [f(n) = g(n) + h(n)] instead of depth
  - Simplified memory-bounded A* (SMA*): drop the nodes with worst f-cost if memory is full

Implementation: like BFS but use priority queue instead with f(n) depending on implementation
<u>Heuristics</u>
Admissible: h(n) is admissible if for every node n, h(n) <= h*(n) where h*(n) is the true cost to reach the goal state from n

- Never over-estimates the cost to reach the goal (i.e. conservative estimate)
- **If h(g) > 0, then not admissible**
- Theorem: if h(n) is admissible, A* using tree search is optimal

Consistent heuristics: h(n) is consistent if for every node n, every successor n' of n generated by any action a, h(n) <= c(n, a, n') + h(n') and h(G) = 0 where c(n, a, n') is the cost to reach n' from n given action a

- Theorem: if h(n) is consistent, A* using graph search is optimal (tree search is also optimal)
- A* expands nodes in order of increase f-cost
  - Gradually adds "f-contours" of nodes where contour I has all nodes with f = f^I where f^I < f^(i+1)
- **Heuristic decreases no more than actual cost with each time step (increasing is fine), otherwise it is inconsistent (overestimates true step cost)**
- Consistency => admissibility, ~admissible => ~consistent
- **Check for overlapping goal states causing heuristic to decrease more than step cost once one goal is achieved**

Dominance: if h2(n) >= h1(n) for all n, then h2 dominates h1 and h2 is better for search

- Between 2 admissible heuristics, always pick the one that dominates (since this means we're always approximating closer to the actual cost)
- Without admissibility/consistency, dominance relationship still possible
- When building dominant heuristic, be careful not to add too much to the original (use min{}/max{} to control)
- **Not dominant if hA >= hB sometimes and hA < hB other times**
- **Choose the heuristic that's admissible if possible**

Relaxed problem: problem with fewer restrictions on the actions

- Cost of optimal solution (of relaxed problem) is admissible heuristic for the original problem **(can also be consistent if exact cost is heuristic)**
- **Admissible heuristic of relaxed problem is admissible for constrained problem**
- Think of ways to "restrict" or "expand" the relaxed heuristic to work with the new problem's constraints

<u>Local search</u>

- Path to goal is not important, the state could be the solution
- No specific goal state, goal found when condition is met

Solving: start with a state and improve it till the solution state is found
Trivial algorithms:

1. Random sampling: generate a state randomly until a solution is found
2. Random walk: randomly pick a neighbor from the current state

Formulation:

1. States (state space)/state representation
2. Initial state **(be as clear with how to generate this)**
3. Goal test **(try maintaining as many constraints in state)**
4. Successor function **(discuss how the swapping works, usually pick 2 sets and swap and element)**

Evaluation function: output the "goodness" of a state

- May be heuristic function or objective function (loss function)
- Measure some invariant so goal is just h(n) = 0/n

Adding constraints to measure: introduce variables for each constraint and create linear combination of constraints (ax1 + bx2)

- a and b trade offs relative importance of constraints
- Alter initial state and transition function works too

Uninformed vs local search: if n is large enough and sufficiently large constraints for pruning, uninformed search may be better; if n is small and many possible solutions, local search to be better
Hill climbing: finding local maximum by moving in direction of successor with highest value; if highest successor still less than current evaluation, return current as local max found

Simulated annealing: accepting a sub-optimal solution every once in a while as it may lead to a new solution later on

```
Current = initial state
For t = 1..infty:
  T = schedule(t)
  If T = 0:
    Return current
  Next = random(successor(current))
  If value(next) > value(current) or prob(next, current, T):
    Current = next
```

Prob(next, current, T) = $e^{\frac{value(next)-value(current)}{T}}$

- Prob is the threshold to accept bad moves
- **Theorem:** if T decreases slowly enough, simulated annealing will find a global optimum with high probability

Beam search: performing k hill climbing in parallel; pick k successors and hill climb and repeat

- Variants: local (deterministic k successors) or stochastic (probabilistic k successors)

Genetic algorithm: marry the best, mutate, repeat

- Initial population (states) -> fitness function -> selection -> cross over (combining states) -> mutation (new state)

Adversarial search

- Against an opponent that may react unpredictably (i.e. strategy environment)
  - Search problems are more predictable
- Usually involving time limits

Designing heuristics: understanding the problem domain and knowing what a good estimate is

- For chess: linear weighted sum of features such as number of white queens – number of black queens, number of player's pieces – number of opponents pieces, etc

Optimizing search: transposition table (cache of previously seen moves and evaluations) and precomputation of best moves in the opening and closing games (pre-recommended next set of actions)

Minimax

- Assuming both players play optimally where the main player tries to maximize their chances to win while the opponent tries to minimize the main player's chances to win
- Reaches terminal state before backtracking
  In max_value and min_value, look at all possible actions and resulting states by considering that the other player plays optimally
- **Always assumes terminal states are computed with MAX player perspective (i.e. score of first player at this state)**
- **If first player root score is positive, then he always wins, else second player always wins**
- **Default to compacting states across same level**
  - **Cannot compact state across levels since state of game includes turn**

Time complexity: O(b^m)
Space complexity: O(bm) with depth first exploration
Complete: if tree is finite
Optimal: against optimal opponent**
Handling large trees: alpha-beta pruning or cutoffs

Alpha-beta pruning

- Dealing with minimax trees that are large
- Visiting some nodes are unnecessary
- Good move ordering improves the effectiveness of pruning
  - Perfect ordering: O(b^(m/2)) in theory
- Core idea is keeping tracking of the parent node's smallest and largest values seen so far
  - Depending on whether parent is min or max, they may or may not pick children nodes once a certain value is found

```
Def minimax(state):
  V = max_value(state, -infty, infty)
  Return action in successors(state) with value v
Def max_value(state, a, b):
  If is_terminal(state): return utility(state)
  V = -infty
  For action, next_state in successors(state):
    V = max(v, min_value(next_state))
    If v >= b: return v // we know that the min above won't use it
    A = max(a, v) // MAX increases MIN's lower threshold
  Return V
Def min_value(state, a, b):
  If is_terminal(state): return utility(state)
  V = infty
  For action, next_state in successors(state):
    V = min(v, max_value(next_state))
    If v <= a: return v // we know the max above won't use it
    B = min(b, v) // MIN decreases MAX's upper threshold
  Return V
```

Minimax with cutoff

- Cutoff can be determined by time limit or depth limit

```
Def minimax(state):
  V = max_value(state)
  Return action in successors(state) with value v
Def max_value(state):
  If is_cutoff(state): return eval(state)
  V = -infty
  For action, next_state in successors(state):
    V = max(v, min_value(next_state))
  Return V
Def min_value(state):
  If is_cutoff(state): return eval(state)
  V = infty
  For action, next_state in successors(state):
    V = min(v, max_value(next_state))
  Return V
```

Is_cutoff: returns true if state is terminal or cutoff is reached
Eval: return utility of terminal states or heuristic for non-terminal states

Types of ML feedback

Supervised: feedback of actual vs prediction
Unsupervised: no feedback, let the model figure out the patterns itself
Reinforcement: trial and error, model does not know the full rules or game

Supervised learning

Types: regression (continuous output) and classification (discrete)
True mapping function: f : x -> y (actual labels)
Hypothesis: h : x -> y' (from the hypothesis class H) such that h ~= f given a training set (i.e. ML model)
Algorithm order: hypothesis class -> type -> algorithm

Performance measure

- Testing if a hypothesis is good

1. Use theorems from statistical learning theory
2. Try the hypothesis on a new set of examples (test set)

Regression:

- Absolute error: |y' – y|
- Squared error: (y' - y)^2
- Mean squared error: $\frac{1}{N}\sum_{i=1}^{N}(y_i' - y_i)^2$
- Mean absolute error: $\frac{1}{N}\sum_{i=1}^{N}|y_i' - y_i|$

Classification:

- Average correctness: $\frac{1}{N}\sum_{i=1}^{N}\mathbb{1}_{y_i'=y_i}$ where 1 if y' = y
- Confusion matrix

		Actual Value	
		Positive	Negative
Predicted Value	Positive	TP	FP
	Negative	FN	TN

- FP: type 1 error, FN: type 2 error
- Accuracy: (TP+TN)/(TP+FN+FP+TN)
- Precision: TP/(TP+FP)
  - How many selected items are relevant
  - How precise were positive predicted instances (i.e. hypothesis)
  - Maximize this if false positives are costly
  - Email spam, satellite launch date prediction
- Recall: TP/(TP+FN)
  - How many relevant items are selected
  - How many positive instances can be recalled (predicted)
  - Maximize this if false negative is dangerous
  - Cancer prediction but not music recommendation
- F1 score: 2/((1/P) + (1/R))
  - Geometric average

Decision tree

- Flow diagram tracing each factor of a dataset to reach a conclusion

Expressiveness: decision trees can express any function of the input attributes

- Consistent decision tree for any training set but won't generalize to new examples
- Prefer a more compact tree

Size of the decision tree hypothesis class: number of distinct decision trees with n attributes: 2^{2^n}
Decision tree learning: greedy, top-down, recursive algorithm

- Select best attribute and repeat
- **If decision is partial, check if sufficient information is given to decide from decision tree, else conclude it's insufficient**
- **One level decision tree uses majority for other branch**

```
Def DTL(examples, attributes, default):
  If examples is empty: return default
  If examples have same classification: return classification
  If attributes is empty: return mode(examples)

  Best = choose_attribute(attributes, examples)
  Tree = new decision tree with root best
  For each value v_i of best:
    Examples_i = {rows in examples with best = v_i}
    Subtree = DTL(examples, attributes – best, mode(examples))
    Add branch to tree with label v_i and subtree subtree
```

Picking an attribute: using entropy and highest information gain
Entropy: measure of randomness, more random = higher entropy

$$I(P(v_1), \ldots, P(v_n)) = -\sum_{i=1}^{n} P(v_i)\log_2 P(v_i)$$

- Given 2 probabilities
  $I\left(\frac{p}{p+n}, \frac{n}{p+n}\right) = -\left(\frac{p}{p+n}\log_2\frac{p}{p+n} + \frac{n}{p+n}\log_2\frac{n}{p+n}\right)$
- If L = R, I = 1
- If L – R = L or vice versa, I = 0

Information gain: given some attribute a, pick the entropy before splitting against a and after and compare (i.e. reduction in entropy)

$$IG(A) = I\left(\frac{p}{p+n}, \frac{n}{p+n}\right) - \sum_{i=1}^{v}\frac{p_i+n_i}{p+n} I\left(\frac{p_i}{p_i+n_i}, \frac{n_i}{p_i+n_i}\right)$$

Arbitrary decisions: if no examples exist of a given attribute value or no mode exists, then pick an arbitrary decision to be made

- Pick the majority otherwise

Problems:

1. Attributes with many values: cause splitting where each branch has a single example
   - Balance the information gain with number of branches

$$GainRatio(A) = \frac{IG(A)}{SplitInformation(A)}$$

$$SplitInformation(A) = -\sum_{i=1}^{d}\frac{|E_i|}{|E|}\log_2\frac{|E_i|}{|E|}$$

   - Split information (E for examples) increases with more branches
   - Gain ratio decreases with more split information
2. Attributes with differing costs: some attributes have costs (monetary or comfort)
   - Use low-cost attributes where possible (balancing IG with cost)

$$\frac{IG^2(A)}{Cost(A)}$$

$$\frac{2^{IG(A)} - 1}{(Cost(A) + 1)^w}, w \in [0, 1]$$

   - w determines the importance of the cost
3. Continuous-valued attributes: define a discrete-valued input attribute to partition the values into a discrete set of intervals
4. Missing values: several options
   - Assign the most common value
   - Assign the most common value of the attribute with same output
   - Assign probability to each possible value and sample
   - Drop attribute
   - Drop rows

Overfitting: decision trees perfectly fit training data; but may perform worse on test data since they capture the data perfectly including noise

- Solutions: Occam's razor or pruning (preferred)

Occam's razor: prefer short/simple hypotheses

- For Occam's razor:
  - Short/simple hypothesis that fits the data is unlikely to be coincidence
  - Long/complex hypothesis that fits the data may be coincidence
- Against Occam's razor:
  - Many ways to define small sets of hypotheses
  - Different hypotheses representations maybe used instead

Pruning: prevent nodes from splitting if it fails to cleanly separate examples; increases the chances of ignoring noise

- Min sample: each node must have at least n samples involved
  - Otherwise, merge with sibling and use the majority outcome
- Max depth: nodes after a given depth d will be merged with sibling
  - Use majority outcome
- Pruned tree is smaller and likely to have higher accuracy

Ensemble methods

- Combining multiple models together

Bootstrap aggregation (bagging): random forests
Boosting: Adaboost, XGBoost
Learning: removing the use of iteration, applications of calculus and linear algebra, fundamentals of good search algorithms

	1	2	3	4	5	6	7	8	9	10
1	1.00000	0.91830	0.81128	0.72193	0.65002	0.59167	0.54356	0.50326	0.46900	0.43950
2	0.91830	1.00000	0.97095	0.91830	0.86312	0.81128	0.76420	0.72193	0.68404	0.65002
3	0.81128	0.97095	1.00000	0.98523	0.95443	0.91830	0.88129	0.84535	0.81128	0.77935
4	0.72193	0.91830	0.98523	1.00000	0.99108	0.97095	0.94566	0.91830	0.89049	0.86312
5	0.65002	0.86312	0.95443	0.99108	1.00000	0.99403	0.97987	0.96124	0.94029	0.91830
6	0.59167	0.81128	0.91830	0.97095	0.99403	1.00000	0.99573	0.98523	0.97095	0.95443
7	0.54356	0.76420	0.88129	0.94566	0.97987	0.99573	1.00000	0.99679	0.98870	0.97742
8	0.50326	0.72193	0.84535	0.91830	0.96124	0.98523	0.99679	1.00000	0.99750	0.99108
9	0.46900	0.68404	0.81128	0.89049	0.94029	0.97095	0.98870	0.99750	1.00000	0.99800
10	0.43950	0.65002	0.77935	0.86312	0.91830	0.95443	0.97742	0.99108	0.99800	1.00000