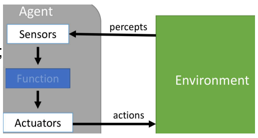
PEAS: **Agent** (AI chatbot perceives **ENVIRONMENT** (percept; chat interface, third-party plugins, user) through **SENSORS** (text input, chat history, context) → agent function (maps from percept histories to actions) → acts through **ACTUATORS** (text output, image output, API).



Based on available information (not necessarily complete), **Rational agent** choose actions that maximise **PERFORMANCE MEASURE** (safety, speed, correctness, legal): Best for whom, Maximise what, Available info, Unintended effects, Costs.

Properties of task environment

- **Fully (Partially) observable:** Sensors give agent access to complete state of environment relevant to choice of action.
- **Deterministic:** Next state of environment completely determined by current state and action executed by agent.
  - (vs **Stochastic: some randomness involved** eg players choose their move randomly in rock paper scissors)
  - **Strategic**: If environment also dependent on the actions of **other agents**, then it is **also** strategic, unless other actions are predictable (e.g dumb)
- **Episodic:** Agent's experience divided into atomic episodes (perceive then act), choice of action depends on current episode itself (**Sequential**: Current decision can affect all future decisions, affected by previous sequence of actions).
- **Static (Dynamic):** Environment unchanged when agent is deliberating. (**Semi-dynamic**: Environment does not change with time, but agent's performance score does eg timer) .
- **Discrete (Continuous):** Limited number of distinct, clearly defined percepts and actions.
- **Single (Multi) agent:** Any object(s) that maximise(s) performance measure (value depends on agent's behavior)

Agent is completely specified by **agent function** (environment → sensors → **agent function** → actuators → environment)

- **Simple reflex agent:** Acts based on current percept, and ignores percept history. Uses condition-action (if-then) rules.
- **Model-based:** Tracks how the world changes (transition model) and information from percept; Simulates environment. Uses condition-action (if-then) rules.
- **Goal-based:** Tracks state of the world and what it will be like if it does action A. Picks action that brings agent closer to goal.
- **Utility-based:** Also tracks state of world and what it will be like if it does action A. **Utility function** to assign score to any percept sequence (how happy will it be in such a state); internalise performance measure,check alignment.
- **Learning:** **Performance element** to select external actions; Critic to give feedback on performance; Learning element makes improvement; Problem generator suggests actions

- **Exploration:** Learn more about the world
- **Exploitation:** Maximise gain based on current knowledge

— — **DEFINING *SEARCH PROBLEM (predictable)*** — —

- **States:** Possible states the environment can be in
- **Initial State:** Where agent starts
- **Goal State(s) / Test:** Can be more than one
- **Actions:** Given a state *s*, ACTION(*s*) returns a finite set of actions that can be executed in *s*
- **Transition Model:** Describes what each action does. RESULT(*s, a*) returns the state that results from doing action *a* in state *s*
- **Action cost function:** ACTION-COST(*s, a, s'*) or c(*s, a ,s'*) gives a numeric cost of applying action a in state *s* to reach *s'*

— — — — — **SEARCH ALGORITHMS** — — — — — —

Goal-based, fully observable, deterministic, static, discrete.

⇒ Solution: A fixed sequence of actions → form a path to a goal

Order of node expansion - which states are prioritised

Tree search: Frontier (data structure) defines the search; where it is rare for two paths to reach the same state

Graph search: Remember all visited nodes in a hashmap

```
create frontier
create visited          // variation that expands less states
insert initial state to frontier and visited
while frontier is not empty:
    state = frontier.pop()              // may skip states
    for action in actions(state):       with less cost
        next state = transition(state, action)
        if next state in visited: continue
        if next state is goal: return solution
        frontier.add(next state)
        visited.add(state)
return failure
```

***UNINFORMED* SEARCH (BFS,uniform-cost,DFS,DLS,IDS)**

Blind search, no clue how close a state is to the goal(s).

— — — — —

Breadth-First Search (queue (layer by layer), **FIFO)**

- Special case of uniform-cost search (equal step costs).
- f(n) = depth of node.
- Save space for BFS while preserving its completeness: Apply goal-test when PUSHING a successor state to the frontier; *check if next state is goal BEFORE pushing to frontier queue*.

```
create frontier : queue
create visited
insert initial state to frontier
while frontier is not empty:
  state = frontier.pop()
  // does the goal test on the node popped from the queue
  if state is goal: return solution
  if state in visited: continue
  visited.add(state)
  for action in actions(state):
      next state = transition(state, action)
      // if next state is goal: return solution
      frontier.add(next state)
return failure
```

Uniform-Cost Search (PRIORITY queue (path cost))

- **A*** with h(n)=0. Expand the least cost unexpanded node.
- **Best-First Search** with f(n) = path cost (**FROM INITIAL state)**

Depth-First Search (stack, LIFO)

- Can be optimised with backtracking search, where only one successor generated at a time., i,e, O(m) space.

Depth-Limited Search (Dls) DFS with depth limit ℓ.

Iterative Deepening Search (IDS) DLS from 0 … ∞

- Search with depth limit = 0, 1, … ∞. Return solution if found.
- IDS is NOT always faster than DFS for time complexity. If each state has only a single successor and the goal node is at depth n, then IDS will run in $O(n^2)$ while DFS will run in O(n).

***INFORMED* SEARCH (Best-first Search)** domain info to *guide*

Best-First Search (priority queue f(n))

- Uses evaluation function **f(n)** for each node, estimates how good a state is. Expand the most desirable unexpanded node.
- **GREEDY Best-First Search** (**A*** with g(n) = 0)
  - **f(n) = h(n)** Heuristic: estimated cost from n to goal
  - Expand node with lowest h(n), appears closest to goal.
- **A* Search**
  - **f(n) = g(n) + h(n)**
  - g(n) = cost so far to reach n
  - h(n) = estimated cost from n to goal
  - f(n) = estimated total cost of path, through n, to goal

Admissible Heuristics

- h(n) never overestimates cost to reach a goal; conservative.
- For each node n, **h(n) ≤ h*(n)**. h*(n) = true cost to reach goal.
- **If h(n) is admissible ⇒ A* using TREE search is optimal.**

Dominance If $h_2(n) \geq h_1(n)$ **for all n**, then h_2 dominates h_1 and is closer to the true cost h*(n), incurs less search cost (on average). h_2 is better for search if admissible.

Inventing admissible heuristics

- Relaxed problem: Fewer restrictions on actions.
- Cost of optimal solution to a relaxed problem is an admissible heuristic for the original problem.

Consistent Heuristics (is admissible)(satisfies △ inequality)

- 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**.
- ⇒ f(n) is non-decreasing along any path
- **If h(n) is consistent ⇒ A* using graph search is optimal.**
- **Consistent heuristic is admissible, but converse not true.**

** No notion of admissibility and consistency in local and adversarial search. Only informed search heuristics.*

Search	Informed,Uninformed	Local
Agent	Goal-based	Goal &/or Utility-
State space	Low to moderate	Very large. Games.
Time constraint	A solution / No solution	Good enough solutn
Solution	Search path (usually)	State

LOCAL SEARCH FORMULATION

- States (state space), Initial state, Goal test (optional)
- **Successor function:** Possible states from a state
- Objective functions: Evaluate value / goodness of a state

State space landscape: Local max, Global max, Shoulder (same values, but progress possible)

Can still be used to solve shortest path problems (state = random path; successor = add / remove subpath(s))

Hill-Climbing / Steepest Ascent / Greedy Local Search

***Pick the best** among neighbors, repeat.*

```
current = initial state
while True:
   neighbor = highest valued successor of current
   if value(neighbor) <= value(current):
      return current
   current = neighbor
```

Find local maxima by traveling to neighbouring states with the highest value. If node has higher value than successors, then return it. It is a local maximum, but may not be a global max.

Simulated Annealing (*hill climbing, but allow bad moves*)

```
current = initial state
T = large positive value
while T > 0:
   next = randomly selected successor of current
   if value(next) > value(current): current = next
   else with prob(current,next,T): current = next
   decrease T
return current
```

Randomly pick successor/action. If the successor has higher value, recurse. Else recurse with probability (=curr, next, T) of choosing a bad move that exponentially decreases. *Escapes local maxima by allowing bad moves occasionally*. If T decrease slowly enough, then finds a global optimum with high probability.

ADVERSARIAL SEARCH AND *GAMES (unpredictable)*

Fully observable, deterministic, discrete, terminal states exist (~~infinite run~~), two-player zero-sum (~~win-win~~), turn-taking.

- **States, Initial state, Terminal states** (where game ends)
- **Actions:** *Action(s)* gives a set of legal moves in *s*.
- **Transition**, **Utility function**(output the value of a state)

Minimax (recursive like DFS; choose highest **minimax** value)

- *Assume the MIN player plays optimally, trying to minimise player's value. Not optimal against a non-optimal MIN player.*
- Pick a move that **maximises** player's **utility**
- MAX player will choose MAX, MIN player will choose MIN.

Alpha-beta Pruning (prune useless branches)

- Will not change the decision

```
def alpha-beta(state):        [minimax: not highlighted]
  v = max_value(state, -∞, ∞)
  return action in successors(state) with value v

def max_value(state, α , β):
  if is_cutoff(state): return eval(state)
  if is_terminal(state): return utility(state)
  v = -∞
  for action, next_state in successors(state):
    v = max(v, min_value(next_state))
    α = max(α, v)             // α = highest value for MAX
    if v >= β : return v      // stop if v >= β
  return v
                              β is the best choice for MIN
def min_value(state, α , β):
  if is_cutoff(state): return eval(state)
  if is_terminal(state): return utility(state)
  v = ∞
  for action, next_state in successors(state):
    v = min(v, max_value(next_state))
    β = min(β, v)             // β = lowest value for MAX
    if v <= α : return v      // stop if v <= α
  return v
```

Minimax with cutoff at a certain depth (refer: ~~strikethrough~~)

- For large/infinite game trees: Cutoff, Use evaluation function
- Replace is_terminal / utility with is_cutoff / eval
- is_cutoff return true if state is terminal, exceed time/depth limit
- eval return utility for terminal, heuristic value for non-terminal
- Evaluation (heuristic) function: Estimate how good a state is

b: branching factor | d: depth | m: maximum depth | C: path cost of optimal solution | ∈: minimum edge cost | G: goal state*

SEARCH	**TIME**: #nodes generated or expanded	**SPACE**: maximum #nodes in memory	**COMPLETE:** Solution / failure if there is	**OPTIMAL:** Always find least cost
BFS	$1 + b + b^2 + \ldots + b^d$ = **$O(b^d)$**.	**$O(b^d)$**. Worst case: expand last child in a branch.	Complete if b is finite.	Yes if same step cost everywhere.
Uniform-cost	**$O(b^{C^*/\in})$**. Estimated depth = C*/∈	**$O(b^{C^*/\in})$**.	Yes if ∈ > 0 and finite C*.	Yes if ∈>0. ∈=0 may cause zero cost cycle
DFS	**$O(b^m)$**.	**O(bm)**.	No, when infinite depth / back&forth loop.	No.
DLS	$1 + b + b^2 \ldots b^\ell$ = **$O(b^\ell)$**. ℓ = depth limit.	**O(bℓ)** if used with DFS.	No if ℓ is not the depth of optimal solution.	No if used with DFS.
IDS overhead $= \frac{nIDS - nDLS}{nDLS}$	$b^0 + (b^0 + b^1) \ldots + \ldots (b^0 \ldots + b^d) = (d+1)b^0 + (d)b^1 + \ldots (1)b^d$ = **$O(b^d)$**.	**O(bd)** if used with DFS.	Yes.	Yes, if same step cost everywhere.Is
Greedy Best-First	**$O(b^m)$**. Good heuristic h(n) gives improve.	**$O(b^m)$**, keep all nodes in memory.	No, can go into loops.	No, heuristic function can be wrong.
A*	**$O(b^m)$**. Good heuristic h(n) gives improve.	**$O(b^m)$**, keep all nodes in memory.	Yes, tracks path so no back and forth. Unless infinite nodes with f ≤ f(G).	Depends on heuristics.
Minimax	**$O(b^m)$**.	**O(bm)**. **Depth first** exploration (then backtrack)	Yes, if the tree is finite.	Yes, against optimal opponent.
α-β Pruning	**$O(b^{m/2})$** with perfect ordering.			

M A C H I N E   L E A R N I N G

A computer program learns from experience E w.r.t some class of tasks T and performance measure P, if its performance at tasks in T, measured by P, improves with E.

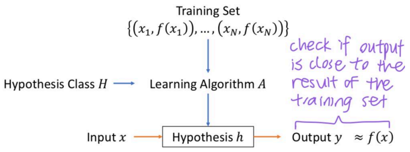
— — **Types of Feedback** — —

SUPERVISED LEARNING (with teacher)

- Agent observes input-output pairs and learns a function that maps from input to output. **Learn from being given the right answers X → Y**. Each example has correct answer.

- **Regression:** Predict **continuous** output (eg number)
- **Classification:** Predict **discrete** output (finite set of values)

- Assume y is generated by a true mapping function *f: x → y* . Use a learning algorithm to find a hypothesis *h: x → ŷ* (from Hypothesis class H) s.t. $h \approx f$, given a training set (set may contain errors) *{ (x_1,f(x1) … (x_N, f(x_N)) }*.



Performance: True measure of hypothesis h (Good if h ≈ f) is how well it handles inputs it has not seen yet, ie the **test set**.

WEAKLY / SEMI-SUPERVISED LEARNING

- Correct answer given, but not precise (eg image contains face, but exact location not specified)

UNSUPERVISED LEARNING

- Agent learns patterns in the pattern without explicitly feedback. No answers given. Most common task is clustering.

REINFORCEMENT LEARNING

- Agent learns from reinforcements (rewards, punishments), eg game, navigating a maze

— Performance measure of output of hypothesis —

REGRESSION (output is continuous value) → Measure **error**:

- Squared error = $(\hat{y} - y)^2$ Absolute error = $|\hat{y} - y|$
  where predicted value = ŷ = h(x); true value = y
- **(Mean) Squared Error** **(Mean) Absolute Error**

MSE = $\frac{1}{N}\sum_{i=1}^{N}(\hat{y} - y)^2$ MAE = $\frac{1}{N}\sum_{i=1}^{N}|\hat{y} - y|^2$

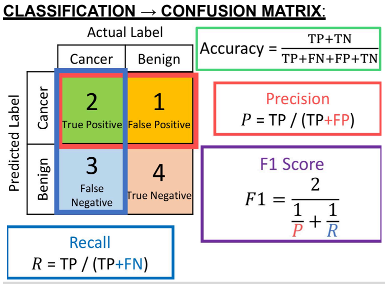
where $\hat{y}_i = h(x_i)$; $y_i = f(x_i)$ for N samples *{(x_1,f(x1)…(x_N, f(x_N))}*

CLASSIFICATION → Measure **Correctness** and **Accuracy**:

Accuracy = $\frac{1}{N}\sum_{i=1}^{N} 1_{\hat{y}_i = y_i}$ where $\hat{y}_i = h(x_i)$; $y_i = f(x_i)$

Range is 0-1. 1 if **prediction** $\hat{y}_i = y_i$ **(true label)**. Take mean.

CLASSIFICATION → CONFUSION MATRIX:



Recall TP / (TP+FN)	**Precision** TP / (TP+FP)
How many **relevant** items are **selected**? How many positive instances can be **recalled** (predicted)?	How many **selected** items are **relevant**? How **precise** were the positive predicted instances?
Maximise if False Negative (FN, type II) is very dangerous. Eg Cancer prediction, but not music prediction.	Maximise this if False Positive (FP, type I error) is very costly. Eg Email spam, Satellite launch date prediction.

D E C I S I O N   T R E E

FALSE that we can always find at least one decision tree that can perfectly label every example in the training set, no matter the amount of data and attributes we have. - Data may not be consistent. Suppose we train a decision tree that determines whether a student passes or fails. No way to cleanly split using the training set gender as only attribute.

Attributes Binary valued, Discrete valued, Binned / Discretised continuous valued (<10, 10-20, 20-40, >40)

Expressiveness

- Decision tree can express any function of the input attributes.
- Function maps a vector of attribute values to a single output value (a decision eg T/F).
- Internal nodes are tests, leaf specifies value to be returned.
- *Trivially*, there is a consistent decision tree for ANY TRAINING SET, but probably **will not *generalise (prefer a more compact tree)*** **to new examples**.
- Size of Hypothesis class: For Bool functions with *n* Bool attributes, #distinct decision trees = #Bool functions = #distinct truth tables with 2^n rows (each row outputs T/F) ⇒ 2^{2^n}

GREEDY, TOP-DOWN, RECURSIVE algorithm for DTL

```
def DTL(examples, attributes, default):
  if examples is empty: return default
  if examples have the same classification:
    return classification
  if attributes is empty:
    return mode(examples)
  best = choose_attribute(attributes, examples)
  tree = a 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_i, attributes − best, mode(examples))
    add a branch to tree with label v_i and subtree subtree
```

decision. Eg True/False

class/category with highest number

choose best attribute based on INFORMATION GAIN

Define this

recurse

1. If remaining e.g.s are all True or False, return that value.
2. If a mix, then choose best attribute to split and recurse.
3. If no examples are left, then return the most common value from node's parent's examples.
4. If no attributes are left for splitting, then return the most common value of current examples.

Decision Tree Learning with INFORMATION GAIN

Entropy (randomness): $I(P(v_1)\ldots P(v_n)) = -\sum_{i=1}^{n} P(v_i)log_2P(v_i)$

- Entropy = 1 if 50:50 eg #true = #false
- Bool variables: $B(q) = -(qlog_2q + (1-q)log_2(1-q))$
  q = probability of it being true
- Dataset with p positive and n negative examples:

$I(\frac{p}{p+n}, \frac{n}{p+n}) = -\frac{p}{p+n}log_2\frac{p}{p+n} - \frac{n}{p+n}log_2\frac{n}{p+n}$ $plog\ p\ ,\ nlog\ n$

Entropy of output of entire set is $B(\frac{p}{p+n})$.

A **chosen attribute** A **with** v **distinct values** divides training set E into subsets $E_1 \ldots E_v$ according to their values.

$$remainder(A) = \sum_{k=1}^{v}\frac{p_k+n_k}{p+n} I(\frac{p_k}{p_k+n_k}, \frac{n_k}{p_k+n_k})$$

Each subset E_k has p_k positive examples and n_k negative examples: $Remainder(A) = \sum_{k=1}^{v}\frac{p_k+n_k}{p_n} B(\frac{p_k}{p_k+n_k})$

Information Gain (highest = impt attrib) / Reduction in Entropy:

$IG(A)\ (in\ bits) = I(\frac{p}{p+n}, \frac{n}{p+n}) - remainder(A)$
= Entropy of this node - Entropy of children nodes

Issues

- **Continuous-valued Attributes:** Define a discrete-valued input attribute to partition values into a discrete set of intervals.
- **Missing values:** If node n tests A, then assign most common value of A at node n to the example with missing data for A and is at node n. Or further filter by most common value amongst examples with the same output. Or assign each possible value of A some probability, then split examples with missing data based on this. Or drop the attribute. Or drop the row.
- **Attribute with many values** (eg dates, phone numbers)**:** Selected by IG as it splits the data well. In extreme cases, each branch has one example, all positive or negative. So balance $Information\ Gain$ with the number of branches.
  - Use $Gain\ Ratio = \frac{IG(A)}{SplitInformation(A)}$.

$Split\ Information(A) = -\sum_{i=1}^{d}\frac{|E_i|}{|E|}log_2\frac{|E_i|}{|E|}$, where E = example.

- **Attributes with Differing Costs** (eg biopsy cost): Make decision tree use low-cost attributes, use Cost-Normalised-Gain $\frac{IG^2(A)}{Cost(A)}$ or $\frac{2^{IG(A)}-1}{(Cost(A)+1)^w}$, where $w \in [0, 1]$ determines the relative importance of the cost vs information gain.
- **Continuous-valued output**: Need regression tree.

Overfitting: Decision Tree's performance is perfect on training data, but worse on training data. Decision tree captures data perfectly, **including noise**. *Dealt with using Oscam's Razor:*

- Prefer short / simple hypotheses.
- More susceptible to overfitting if too many attributes used
- In favor: Short/simple (or Long/complex) hypothesis that fits the data is unlikely to (or may) be coincidence.
- Against: Many ways to define small sets of hypotheses. Different hypotheses representations may be used instead.

PRUNING (eg minimum sample leaf pruning, max depth)

- Prevent nodes from being split if fail to cleanly separate e.g.s
- Results in **smaller tree**. DT has **higher accuracy** if tested using sample data as **noise can be ignored** through pruning.