Introduction to AI
An intelligent **agent** will precept the **environment** using its **sensors**, then using a **function**, it will turn information into **actions** using **actuators** that will affect the **environment** again.
Agent: anything that can be viewed as **perceiving environment through sensors** and **acting upon that environment through actuators**
A agent function maps from percept histories to actions.
Performance measure: measure of goodness, to know the agent is doing the right thing.
The performance measure takes into account:
- Best for who?
- What is being optimized
- Unintended Effects
- Costs
- Performance vs cost

Types of Agents:
Table-up agent: Have a lookup table that maps percepts to actions, and then just look up action for each percent.
Drawbacks: storage, cannot react to changes, need time to access table
Vacuum Cleaner agent: Use if/else statements to capture state+action
Models for Agent Organization
1. Simple Reflex Agent
2. Model-based Reflex Agent
3. Goal-based Agent
4. Utility-based Agent
5. Learning Agent

An agent operating in the real world must often choose between maximized expected utility currently (**Exploitation**) or learning more about the world to Improve future gains (**Exploration**)

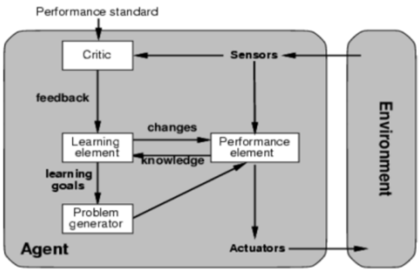
Problem Formulation
Define initial state
Define actions/successor function
Goal test: reach goal or not
Path cost: e.g. 1 per move
State space must be abstracted for problem solving
An **abstraction function** maps the representation to a real world state. There needs to be a Representation Invariant I, such that I(c) = true for all valid representations of the state.
Search Strategies:
Search Strategies are evaluated using 4 dimensions:
5. Completeness: if solution exists, does it find
6. Time Complexity: no. of nodes generated
7. Space Complexity: max no. of nodes stored
8. Optimality: Does it always find a least cost solution

Bi-directional Search:
Search from both back and front, stop when the searches meet. Better because $2 \times O(b^{d/2}) < O(b^d)$
Issues:
Successor function must be reversible
There may be many possible goal states
How to check if node appears in the other search tree, if the 2 searches is done simultaneously.

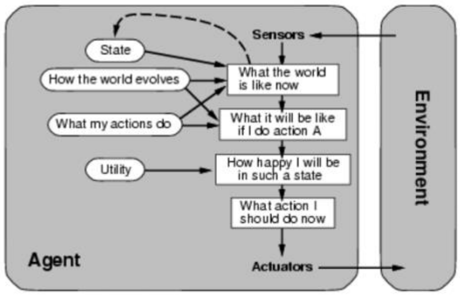
Rational Agent:
Rational agent chooses an action that **maximizes its performance measure,** using the **percept sequence and built-in knowledge**
PEAS Framework:
1. Performance Measure
2. Environment
3. Actuators
4. Sensors

Use the PEAS Framework to frame AI problems
For example, in Autonomous Driving,
P -> Safety, speed, legality, comfort
E-> Roads, visibility, pedestrains etc
A -> Steering Wheel, Accelerator, Brake etc (things that allow input)
S -> Camera, Speedometer, GPS etc

Learning Agent



Utility-based Agent



Memoization (Graph Search))

create queue ← create hashtable
insert initial state

while queue not empty:
    current ← queue.pop()
    is_goal(current)? → return ans

    if current not in hashtable:
        hashtable←current
        queue ← expand(current)

In Python, ke needs to be hashable

return not found

Environment
Next, we characterize the environment:

Fully Observable: Agent's sensors give complete state.	Partially Observable: Agent's sensors cannot perceive complete state
Deterministic, can be strategic: Next state completely determined by current state and action taken by agent. IF deterministic except for actions by other agents then **strategic**	Stochastic: Next state may be random/ not completely defined by current state and action
Episodic: Previous states do not matter/memoryless	Sequential: Choice of action may differ depending on previous states
Static: Environment does not change with time. If environment does not change but agent's performance score does, then semi-dynamic	Dynamic: Environment changes with time
Discrete: Limited number of distinctly defined percepts and actions	Continuous: No distinctly defined percepts and actions
Single Agent	Multi-Agent

Uninformed Search: only use info available in the problem definition. (no heuristic involved)
6. Uniform-cost search
7. Breadth-first search (BFS)
8. Depth-first search(DFS)
9. Depth-Limited search
10. Iterative Deepening search

b = maximum branching factor of search tree
d = depth of least-cost solution
C* = Cost of optimal solution
m = max depth of state space, and can be ∞

Uninformed Search: only use info available in the problem definition. (no heuristic involved)
1. Uniform-cost search
2. Breadth-first search (BFS)
3. Depth-first search(DFS)
4. Depth-Limited search
5. Iterative Deepening search

b = maximum branching factor of search tree
d = depth of least-cost solution
C* = Cost of optimal solution
m = max depth of state space, and can be ∞

	Implementation Details	Complete	Time	Space	Optimal
Uniform Cost	Expand least cost unexpanded node. Frontier = pq. Apply goal-test when popping from frontier and NOT pushing.	Yes, if b is finite and step cost >= ε	$O(b^{C^*/\varepsilon})$ C* is the cost of optimal solution, so this is the min no. of nodes we need to expand	$O(b^{C^*/\varepsilon})$, approximately O(n)	Yes
BFS	Expand shallowest unexpanded node. Frontier = FIFO queue	Yes, if b is finite	$1+b+b^2+b^3+\ldots+b^d = O(b^d+1)$	$O(b^d)$, Worst case need to keep last layer **Is likely the issue**	Yes
DFS	Expand deepest unexpanded node. Frontier = stack	No, in infinite depth space, there might be loops.	$O(b^m)$, Need to traverse the entire depth no matter what. Terrible if m >> d, but better if there is a high density of solutions	O(bm), for each node in depth m, need to store all b node. Potentially linear space.	No
DLS	DFS but with a limit, nodes at depth l has no successors. Once limit is reached, backtrack	No, if solution at depth > limit	Same as DFS $N_{DLS} = b^0 + b^1 + b^2 + \ldots + b^{d-1} + b^d$	Same as DFS	No
IDS	Start depth at 0. Iteratively increase the depth, then run DLS at the depth until find solution	Yes	$O(b^d)$ $N_{IDS} = b^0 + b^0 + b^1 + b^0 + b^1 + b^2 + \ldots + b^0 + b^1 + b^2 + \ldots + b^{d-1} + b^d$ There is overhead	O(bd) much less space than BFS	Yes, if step cost = 1

Informed Search
Performance of search strategies depend on the order of node expansion. Informed search makes use of information to decide on a better way to perform node expansion.
Types of Informed Search
Best-first Search
Greedy Best-first Search
A* Search
Local Search Algorithms
Adversarial Search Algorithms

Local Search Algorithms
Path to goal is relevant, the goal state itself is the solution.
Local search keeps a single "current state" and improves it.
Have a state space, and find configurations that satisfies the constraints.
Formulate Local Search:
1. Initial State
2. Actions/successor function
3. Good heuristic
4. Goal state/test

Hill Climbing Search:
At each iteration pick the successor with the HIGHEST heuristic value (diff from A*)

```
function HILL-CLIMBING(problem) returns a state that is a local maximum
  inputs: problem, a problem
  local variables: current, a node
                   neighbor, a node
  current ← MAKE-NODE(INITIAL-STATE[problem])
  loop do
     neighbor ← a highest-valued successor of current
     if VALUE[neighbor] ≤ VALUE[current] then return STATE[current]
     current ← neighbor
```

Hill Climbing Search might encounter local maximas when we want to find global maxima.
Solution: Introduce Randomness

Types of feedback
Supervised:
•Correct answer given for each example
•e.g. image of "A" and its unicode 0041
Unsupervised
•No answers given
•e.g. are there patterns in the data?
Weakly supervised
•Correct answer given, but not precise
•e.g. This slide contains a face (but not exact location)
Reinforcement
•Occasional rewards given
•e.g. robot navigating a maze

Precision: Correctly convict a guilty person
Recall: Percentage of correct convictions

Best-First search
Use an evaluation function f(n) for each node to decide desirability of unexpanded nodes. Order nodes in **decreasing order of desirability (use pq).**
Greedy Best-First search
Use heuristic, h(n), as evaluation function.
Expands node that appears to be closest to the goal, lowest heuristic value. m =depth
Completeness: No, can be stuck in loops
Time: $O(b^m)$, but good heuristic can improve
Space: $O(b^m)$, need to store all nodes
Optimal: no

Dominance:
If $h_2(n) >= h_1(n)$ for all n, where h_2 and h_1 are both admissible,
Then h_2 dominates h_1.
h_2 is better for search (find solution faster)
To invent heuristics:
Consider relaxed problem
Relaxed problem: problem with fewer restrictions. Solution to relaxed problem = admissible heuristic to original problem
If cannot find admissible heuristic, can use non-admissible heuristics but will lose optimality
To save space, use Memory-bounded heuristic search.

Simulated Annealing
Escape local maxima by allowing some "bad" moves but gradually decrease the frequency -> randomly select successor every once in a while
If the T(frequency of bad moves) decrease slowly enough, simulated annealing will find global optimum with probability -> 1
Beam Search
Perform k hill-climbing searches in parallel
Local beam search: k threads share info
Stochastic beam search: k independent threads
Formulating successors:
Genetic algo- select, crossover, mutate

Information Content (Entropy):

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

- A chosen attribute A divides the training set E into subsets $E_1, \ldots, E_v$ according to their values for A, where A has v distinct values.

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

- Information Gain (IG) or reduction in entropy from the attribute test

$$IG(A) = \underbrace{\boxed{I\left(\frac{p}{p+n}, \frac{n}{p+n}\right)}}_{\text{Entropy of this node}} - \underbrace{\boxed{remainder(A)}}_{\text{Entropy of children nodes}}$$

- Choose the attribute A with the largest IG

Inst.	Predicted $\hat{y}$	Actual y	
1	Alert	Alert	TP
2	Alert	Alert	TP
3	Sleepy	Alert	FN
4	Sleepy	Alert	FN
5	Sleepy	Alert	FN
6	Sleepy	Sleepy	TN
7	Sleepy	Sleepy	TN
8	Sleepy	Sleepy	TN
9	Sleepy	Sleepy	TN
10	Alert	Sleepy	FP

App case: student alertness prediction

Predicted Label \ Actual Label	Alert	Sleepy
Alert	2	1
Sleepy	3	4

Predicted Label \ Actual Label	+ve	-ve
+ve	TP True Positive	FP False Positive
-ve	FN False Negative	TN True Negative

$$\text{Accuracy} = \frac{TP+TN}{TP+FN+FP+TN}$$

Problems with Greedy Best-first search:
We do not consider the cost incurred thus far
A* Search:
Add the cost so far, to avoid expanding paths that are already expensive.
f(n) = g(n) + h(n)
f(n) = estimated total cost of path from start, through n to goal
g(n) = cost from start to n
h(n) = estimated cost from n to goal
Completeness: Yes
Time: $O(b^d)$, d = depth of optimal Solution
Space: $O(b^d)$
Optimal: No

Iterative Deepening A*
Similar to IDS, instead of using depth for cutoff, use f-cost, takes linear space.
To utilize more of the available memory, can use simplified Memory-bounded A*.
Do A*, but when memory is full, drop nodes with worst f-value.
Might be problematic if the optimal solution makes used of dropped nodes.

Adversarial Search:
Used in games with opponent
Key Assumption: Opponent will react optimally
Minimax Algorithm
Idea: player will choose to move to a position with highest minimax value depending on whether they are min or max -> determines the perfect play for **deterministic games.**

```
function MINIMAX-DECISION(state) returns an action
  v ← MAX-VALUE(state)
  return the action in SUCCESSORS(state) with value v

function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for a, s in SUCCESSORS(state) do
     v ← MAX(v, MIN-VALUE(s))
  return v

function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for a, s in SUCCESSORS(state) do
     v ← MIN(v, MAX-VALUE(s))
  return v
```

Minimax:
Complete: Yes if tree is finite
Time: $O(b^m)$
Space: O(bm)
Optimal: Yes, against an optimal opponent
Problem: there are a lot of potential branches to search.
We can ignore paths that will never be chosen using:
Alpha-Beta pruning:
Idea, keep alpha, beta value along the search path. Whenever alpha >= beta, can prune because the next branches will never be selected.

==Heuristics==
Admissible Heuristic:
Heuristic h(n) is admissible if for **every node n,**
h(n) <= h*(n), where h*(n) is the true cost to reach goal state from n.
Idea: heuristic never over-estimates the cost to reach the goal.
IF heuristic is admissible, A* using tree-search is OPTIMAL
For proving purposes:

Optimality of A* (proof)

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

f(G) = g(G) since h(G) = 0
f(G_2) = g(G_2) since h(G_2) = 0
g(G_2) > g(G) since G_2 is suboptimal
f(G_2) > f(G) since f(n) = g(n)+h(n)
h(n) ≤ h*(n) since h is admissible
g(n) + h(n) ≤ g(n) + h*(n)
f(n) ≤ f(G)

Hence $f(G_2) > f(n)$, and A* will never select G_2 for expansion

Consistent Heuristic:
Heuristic 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')
If h is consistent:
f(n') = g(n') + h(n')
<= g(n) + c(n, a, n') + h(n')
<= g(n) + h(n)
<= f(n), f is non-decreasing along any path.
IF heuristic is consistent, A* using graph-search is OPTIMAL
CONSISTENT => ADMISSIBLE

Summary: α-β algorithm
- Initially, α = -∞, β = +∞
- α is max along search path
- β is min along search path
- If at MIN node, can stop if we find a node that is smaller or equal to α
- If at MAX node, can stop if we find a node that is larger or equal to β

Pruning **does not** affect the final result.
Good move ordering improves effectiveness of pruning.
With "perfect" ordering, time complexity is $O(b^{m/2})$: prune away half the nodes
In the event of resource limits, Can limit depth, memorize, or pre-compute standard opening/closing moves.
In the event of infinite game trees, cannot calculate utility because no end => use cutoff at depth + replace utility with eval function(usually a weighted sum)

```
function DTL(examples, attributes, default) returns a decision tree
  if examples is empty then return default
  else if all examples have the same classification then return the classification
  else if attributes is empty then return MODE(examples)
  else
     best ← CHOOSE-ATTRIBUTE(attributes, examples)
     tree ← a new decision tree with root test best
     for each value v_i of best do
        examples_i ← {elements of 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
  return tree
```

```
function ALPHA-BETA-SEARCH(state) returns an action
  inputs: state, current state in game
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in SUCCESSORS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
  inputs: state, current state in game
          α, the value of the best alternative for MAX along the path to state
          β, the value of the best alternative for MIN along the path to state
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for a, s in SUCCESSORS(state) do
     v ← MAX(v, MIN-VALUE(s, α, β))
     if v ≥ β then return v
     α ← MAX(α, v)
  return v

function MIN-VALUE(state, α, β) returns a utility value
  inputs: state, current state in game
          α, the value of the best alternative for MAX along the path to state
          β, the value of the best alternative for MIN along the path to state
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for a, s in SUCCESSORS(state) do
     v ← MIN(v, MAX-VALUE(s, α, β))
     if v ≤ α then return v
     β ← MIN(β, v)
  return v
```

Inst.	Predicted $\hat{y}$	Actual y
1	Alert	Alert
2	Alert	Alert
3	Sleepy	Alert
4	Sleepy	Alert
5	Sleepy	Alert
6	Sleepy	Sleepy
7	Sleepy	Sleepy
8	Sleepy	Sleepy
9	Sleepy	Sleepy
10	Alert	Sleepy

App case: student alertness prediction

Predicted Label \ Actual Label	Alert	¬Alert	
Alert	2 True Positive	1 False Positive	3 Σ Pred. Pos.
¬Alert	3 False Negative	4 True Negative	7 Σ Pred. Neg.
	5 Σ Actual Pos.	5 Σ Actual Neg.	

Precision
P = TP / (TP+FP)

Recall
R = TP / (TP+FN)

FP: Type I error
FN: Type II error

60