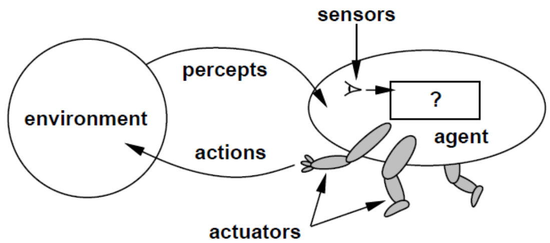
Chapter 1

Rationality

Rational: maximally achieve goals and maximize your expected utility

Rationality only concerns what decision are made not the thought behind them (we are concerning only with acting rationally not thinking rationally)

Chapter 2



Agent = sensors + actuators

Perceive through sensors. Act through actuators.

Agent function vs program

Agent function: from percept histories to actions

Agent program: the implementation of the function

Rationality

Depends on: (Will take the vacuum cleaner as an example)

- Performance measure: one point for each clean square
- Agent's (prior) knowledge: the geography of the environment
- Agent's percepts to date: perceives its location and whether it contains dirt or not
- Available actions: Left, Right, Suck

Definition of rational agent: for each possible percept sequence, agent should select action expected to maximize performance measure, given the percept sequence and the agent's knowledge

We talk about the expected utility not actual utility due to uncertainty in environments. Which mean rationality doesn't mean perfection (maximize actual performance) and doesn't equal omniscient (an agent that know the actual outcomes)

Learning is important in partial or unknown environments (i.e., vacuum cleaner forecasts where and when dirt will appear)

If an agent relies on the prior knowledge of its designer rather than its own percepts, it lacks autonomy

Specifying Task environments

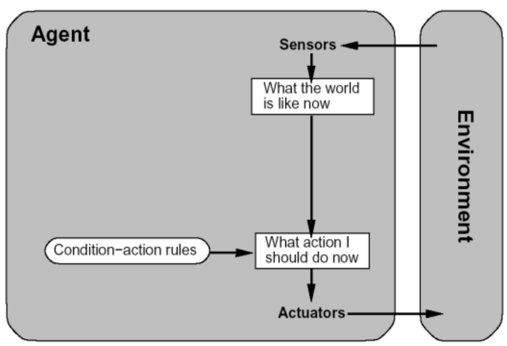
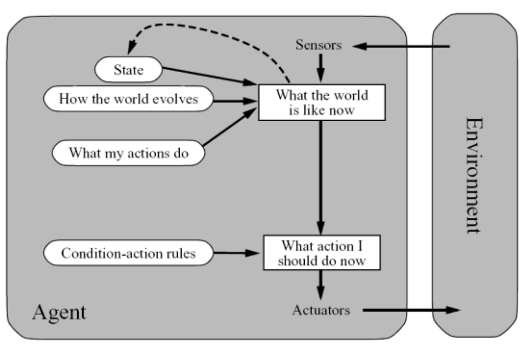
Task environment: a problem spec for which the agent is a solution

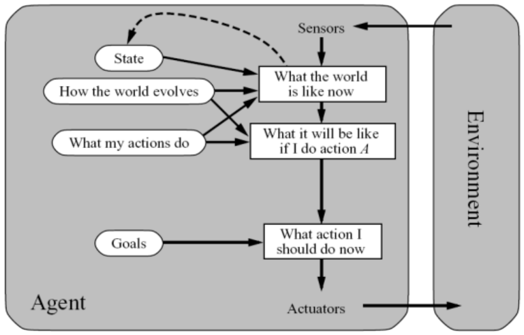
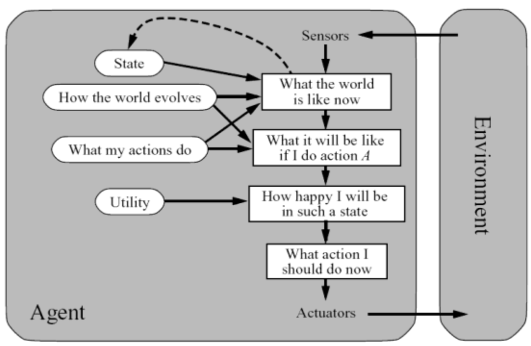
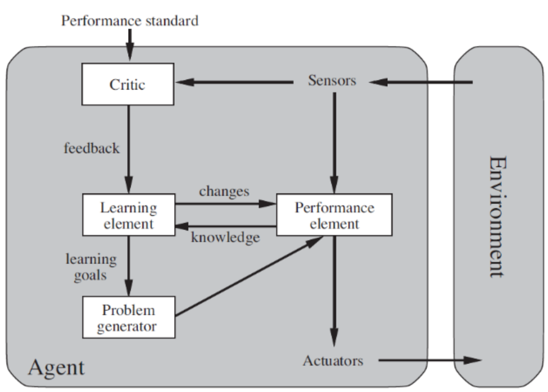
PEAS: Performance measure – Environment – Actuators – Sensors

Environment Type	Description	vs	Notes
Fully observable	Sensors give complete state of environment	Partially observable and unobservable	Unobservable: i.e., agent has no sensors
Deterministic	Next state depends on current state, action	Stochastic	Fully observable + deterministic = no uncertainty

Episodic	Subsequent episodes don't depend on actions in previous episodes	Sequential	Episodic ex: spotting defects on current part doesn't affect next parts Sequential ex: Chess, taxi Both, actions can have long term consequences
Static	Environment doesn't change while thinking	Dynamic and semi-dynamic	Semi-dynamic: clocked chess
Discrete	Limited distinct percepts and actions	Continuous	Chess: discrete
Single Agent	Like solving puzzles	Multi-agent	Multi-agents can cooperate, compete, communicate or randomize its behaviors

Agent = Architecture + Program, Program implements the agent function that maps percepts to actions

Agent	Schematic	Problems
Simple Reflex		• Ignores percept history • Doesn't handle partial observable environments
Reflex Agents with State (Model)		• State is not always enough, how to choose among alternatives. It needs a goal.

Goal-based Agents		• Although it's flexible (can be easily modified) and adaptive, it's less efficient due to the planning and searching for the goal • Goals are not enough to maximize the performance measure (can't decide which path is better)
Utility-Based Agents		• No learning, like previous models
Learning Agents		• **Learning element:** improves performance • **Performance element**: it takes in percepts and decides on actions. (Agent itself in the previous structures or agent function) • **Critic:** gives feedback to the learning element by measuring the performance • **Problem generator:** suggests actions that will lead to new and informative experiences. (exploration)

Chapter 3

Problem Solving Agents

Goal, problem formation – solution is a sequence of actions in observable, discrete and known environments – also called 'Formulate, Search, Execute" agent

Solution is executed eye closed; When a solution is found after searching it executes it one by one without looking into percepts and once the solution sequence has been executed a new goal will be formulated. Also called open loop system (ignoring percepts breaking loop between agent and environment)

Problem Formulation

1. **Initial state**: ex In(Arad)
2. **Actions**: possible actions at state S. ex: Go(Sibiu)
3. **Transition Model**: description of what each action does. Results(S, A) returns state resulting from applying action A at state S. ex: Result(In(Arad), Go(Zerind)) = In(Zerind)
4. **Goal Test**: determines if a certain state is a goal ex In(Bucharest)
5. **Path cost function**: Assign cost to a path. C(S, A, S') step cost of taking action A in state S to reach S'

Solution

Action sequence from initial state too goal. Solution Quality is measured by path cost function. Optimal Solution = lowest cost among solutions

Abstraction

Remove details from real world and actions (no irrelevant actions and no specific actions)

This leads to shrinking the count of possible world states.

Searching

State Space Graph	Search Tree
Nodes => world configuration (state) Arcs => successors – action results Rarely build in memory as it's big for memory	Nodes => states but correspond to plans and paths to achieve these states Each node is entire path in state space graph For most problems, we can never build the whole tree

Graph Search	Tree Search
Each state occurs only once. This is achieved by adding explored set: new node, that is already in explored or the frontier, can be discarded	The same node/state can occur multiple times Search: Expand as few nodes from frontier

Uninformed Search Algorithms Evaluation

	BFS	Uniform-Cost Search	DFS	Depth-Limited Search	Iterative Deepening	Bi-directional Search
Completeness: (Guaranteed to find solution if any)	Yes	Yes, if best solution has finite cost and ε is positive, step costs >= ε	Graph search: Yes, in finite spaces Tree Search: No	Yes if (depth limit) l >= d	Yes	Yes, if both directions are BFS
Time Complexity: (How long to find solution in terms of number of nodes generated)	$O(b^d)$	$O(b^{C^*/\varepsilon})$ If that solution costs C* and step costs at least ε	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$
Space Complexity: (How much memory in terms of maximum number of nodes stored)	$O(b^d)$	$O(b^{C^*/\varepsilon})$	$O(bm)$	$O(bl)$	$O(bd)$	$O(b^{d/2})$
Optimality: (Lowest path cost solution?)	Yes, if all actions have same cost	Yes	No	No	Yes, if all actions have same cost	Yes, if both directions use BFS and all actions have same cost
Frontier Data structure	Queue (FIFO)	Priority Queue	Stack LIFO	Stack LIFO	Stack LIFO	Can use FIFO or LIFO frontier
Disadvantages	Memory and time requirements in big depths	Explores in every direction. No info about goal location	Stuck down in the wrong path. Many problems have deep/ infinite search trees so DFS should be avoided	If depth is too small it's not even complete	Redundancy from repeating all work of the previous phase	Can't search backward in all problems: not reversible – multiple goals – can be abstract like n-queens

Complexities are estimated in terms of (b: branching factor, d: depth, m: maximum length of any path, l: depth limit, C^* total cost of the solution of uniform-cost search, ε is minimum step cost)

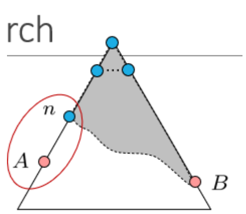
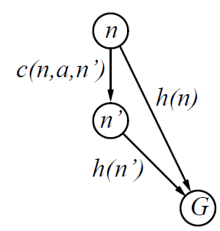
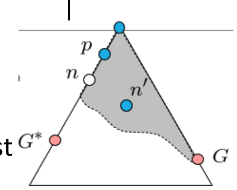
Informed Search

Heuristic function h(n):

- Estimates how close state to goal

	Greedy best-first (Pick the smallest h(n))	A* Search (Pick the smallest g(n) + h(n))	Iterative-deepening A* (IDA*)
Completeness	Tree: No Graph: Yes, in finite spaces	Yes	Adapt the iterative deepening strategy to reduce memory requirements. Use the f-cost rather than the depth At each iteration, cutoff value is the smallest f-cost that exceeded the threshold on previous iteration. Ex: if threshold = 5 in 1st iteration, and a node with cost 6.5 was discovered. It exits the 1st iteration and starts 2nd iteration with threshold = 6.5
Time Complexity	$O(b^m)$	$O(b^{\varepsilon d})$	
Space Complexity	$O(b^m)$	$O(b^d)$	
Optimality	No	Tree: Yes, if h(n) is admissible (h(n) <= actual cost to goal) Graph: Yes, if h(n) is consistent (heuristic arc/edge cost <= actual cost of the edge)	
Disadvantages	Time and space requirements. Depends on the heuristic & problem. In worst-case behaves like badly-guided DFS.	exponential time and space complexity.	

A* Star Optimality Proof

Tree Search: Admissible	Graph Search: consistent
Assume: A is an optimal goal node B is suboptimal goal node h is admissible Claim: A will exit the frontier before B i.e. f(A) <= f(B) assume a node n that is f(n) <= f(B) will be expanded before B since h is admissible f(n) <= g(A) since f(A) is goal -> f(A) = g(A) f(n) <= f(A)  since B is subgoal -> f(A) < f(B) therefore f(n) <= f(A) < f(B) therefore, A exits before B, A* is optimal	Since h(n) is consistent -> h(n) – h(n') <= c(n, a, n') f(n') = g(n') + h(n') = g(n) + c(n; a; n') + h(n') >= g(n) + h(n) = f(n) Then f(n') >= f(n) So, Values of f(n) along any path are nondecreasing 
	Proof: for state s, any node that reach it optimally will be expanded before nodes that reaches it sub optimally. Assume: some n on path to G* isn't in queue when we need it, because worse n' on path to G* expanded first  Let p be the ancestor of n that was on the queue when n was popped Since h is consistent -> f(p) < f(n) Since n' is suboptimal -> f(n) < f(n') Contradiction from this and previous fact A* is optimal