Text Books:

- AI a Modern Approach (Stuart Russell, Peter Norvig)

AI → Thinking/Acting ~~Humanly~~ Humanly
→ Thinking/Acting Rationally

Most Accepted definition

⇒ AI → Act Rationally → Agents act to achieve best possible outcome or the expected outcome.

↓
Eliminate emotions
(Maximize or minimize a metric)

---

1) Thinking Humanly: → Cognitive modelling approach

* ⇒ Cognition → Set of mental abilities and processes related to knowledge. (Memory, reasoning etc)

---

2) Thinking Rationally:

- Laws of thought encoded using logic.

---

3) Acting Humanely:

- Can a computer pass itself as a human (Turing Test)

↓ Robotics
↓ Computer Vision
↓ Machine Learning
↓ Natural Language Processing
↓ Knowledge representation
↓ Auto mat reasoni

Note: GPS (General Problem Solver) → Thinking Humanely approach.

* 4) Acting Rationally :— MOST RELAVENT

- Acts to achieve best / expected outcome under uncertainity.
- Easy to encode compared to humane emotions etc.

→ Perfect Rationality → Impossible → So we use limited rationality

— x —

Note: AI → Probability based (Bayes Thrm)

— x —

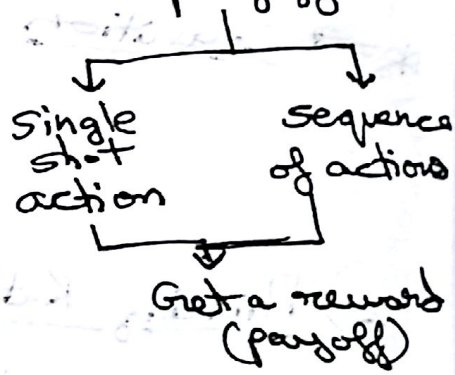
* Utility Values for Actions :

- Any agent can choose an action depending on the actions utility value.

— x —

*Note: AI agents try to optimize on payoff

- AI tries to look into long term payoffs as well (multishot game)



— x —

Singularity :

- When artificial superintelligence overtakes us humans, we expect this to happen in around 20-30yrs

→ Replicability will be achieved by then.

↓

The AI agent can adapt and learn on its own, when this replicability is achieved, we can say the AI overtook us.

*Symbolic Languages: (Lisp or Prolog)
(Programming)

- The code can change itself as # if it were just plain data.
- Normal languages cannot do this.

—— x ——

Note: Neural Networks:

- Tries to compute a weight for all artificial neurons, which are the given inputs, and to match these to outputs, it tries to assign weights. (Connection strengths b/w artificial neurons)

(Hebian learning)

—— x ——

*Developments of AI:

1) Newell & Simon GPS: (General Problem Solver)

- Order of considering subgoals must be similar to humans.
- GPS lead to physical symbol system hypothesis

All data etc can be represented in a physical symbol system.

2) Geometry theorem prover;

3) Solve checkers:

- Chinook system used global optimizations, (try out all possibilities) (took 18 years of computation) ~~come up saying, draw~~

4) Advice Taker: (Hypothetical)

—— x ——

Note: A* search algorithm, Hough transform, visibility graph method

—x—

Note: Shakey robotics project — Stanford (For exam)

—x—

5) Dendral & Mycin:

- two expert systems.
- Dendral — not probablistic, unlike mycin.
  - ↓ Molecular structure analysis.
  - ↓
    - Identify bacteria causing problems
    - Blood infections.

—x—

6) Hidden Markov Models: (Scientific method proof using rigorous theorems)

- Speech recognition.
- Text/Writing recognition etc.

—x—

7) SOAR Architecture: (Latest developments)

- Highly focussed on the "whole agent" problem.
- Considering large datasets, emphasis on data rather than algorithms
- Cognitive architecture.

—x—

Note: Since the amount of data available to us these days is huge, ML etc are thriving. (Machine Learning)

—x—

- SOAR → Set of Rules → called productions (technical term)
  → Inference engine
  - Given input, if rule doesn't exist, inferencing is done & if output is desired, its added to the rules.

———x———

*Rational Agent: → A hardware/software/other entity.

- part of environment
- has a goal (rational)
- senses environment
- acts upon environment (rational) to achieve goal.

⇒ 1) Percept: Input received at any instant by the agent
2) Percept Sequence: History of percepts
3) Agent Function: Maps percept sequence to action.
   → Implemented by Agentprogram.

———x———

Note: In reality there are ∞ percept sequences, so its hard to map all percept sequences to actions. → So we try create agents which mimic the table without having ∞ percept sequences

———x———

*Note: Agent function builds table in planning phase, and executes the actions as required in execution phase

———x———

*Note: Desirability of a percept sequence can also be evaluated by a performance function/measure/metric

———x———

* Exploration vs Exploitation: (Tradeoff)

- Intelligent agents need to decide whether to explore new tasks/actions in search for greater rewards, or to just exploit their known actions to get maximum reward.

——x——

* Specifying Task Environment:

↳ PEAS
- → Performance measure
- → Environment → Fully / Partially observable; Single / Multi agent environment (we work with other agents or all by ourselves)
- → Actuators
- → Sensors

——x——

Note: Cooperative + competitive → co-opetitive
(Agents can be coopetitive)

——x——

* Properties of Task Environment:

) Deterministic or Stochastic

Deterministic:
- Next state from a given state is well defined (Certain environment)

Stochastic:
- Probabilities are given to next states possible (Uncertain environment)

) Episodic (ex: Spotting defective parts) or Sequential → (ex: Chess)

Episodic:
- Multiple but independent episodes (sequence of actions taken by the agent)

Sequential:
- Current decisions affect future decisions.

——x——

3) Static or Dynamic

- Environment does not change while the agent is deciding on an action.

- Environment may change when decision is being made, so the agent needs to look at the world while performing an action that it has decided.

4) Discrete or Continuous

- When you can define timesteps with certain gaps. (ex: Turns in a chess game), this can be the timesteps we consider.

- When time behaves as a continuous variable. we can't divide it into finite states.

⇒ We can model continuous worlds as discrete worlds as well.

5) Known or Unknown

- Outcomes for all actions are specified.

⇒ Can be partially observable where we know the rules of the game but don't know all the info ex: Card Game.

- Agent has to learn outcomes.

⇒ Can be fully observable but we don't know the rules/ outcomes ex: Puzzle game.

Types of Agents (slides for images of each agent)

1) Simple reflex based agent:

- Given a percept, they take an action.
- Condition → Action rule.
- Mainly used in completely observable environment.

2) Model based reflex agent:

- The agent maintains a model of the world.
- Useful in partially observable environment.
- Depends on percept history to take an action.

⇒ condition – action rules are used here as well, but depend on the model that the agent has.

3) Goal based agent: (Flexible / Richer type of agent)

- Considers future goals while performing actions as well.
- Has a set of goals instead of condition-action rules.

→ Takes risks when needed, unlike the above types of agents.

4) Utility based agent:

- Each goal is also assigned an importance.
- Utility is a measure of importance &

probability of success, together.

- Actions are based on utility instead of just goals.

*5) Learning agents: (See architecture → slides)

* • Useful in initially unknown environments.

→ Learning element (Formulate a new utility function for an action)
→ Performance element (Knowledge/procedure to do actions)
→ Critic (Observes world and informs learning element)
→ Problem generator (suggests actions that will make the agent learn)

——— x ———

* Components of Agents: (Representation)

1) Atomic: Each state is a single element.

2) Factored: Each state has variables / attributes.

3) Structured: Each state is an object. (Like class etc) (Hierarchy can be present)

——— x ———

Search

— x —

Note: Problem solving agents are kind of goal based with atomic representations.

Note: Problem formulation is deciding what actions & states to consider given a goal.

— x —

* **Note:** State space of a problem ⇒ States + Actions + Transition.
Transition model → the state returned after an action.

— x —

Note: Goal ~~State~~ Test ⇒ Check if current state is a goal state.
Path cost = Sum of step costs → action

$c(s, a, s')$
↓ start state, ↓ next state

— x —

** **Note:** Search Tree: ~~Similar to DFS trees~~

- When applying actions on a state, new states are generated, these are called the frontiers.
- Root of the search tree, is the source, and children of this are all the states that result from actions on the source state.

— x —

* **Note:**
- Graph search keeps visited states. (Explored set) → More memory
- Tree search does not, so we can get redundant paths if we are not careful (loop paths) → less memory

*1) Uninformed Search Strategies

a) **BFS** → Explore shallowest unexplored node → FIFO

↳ Complete, Optimal algorithm → we can prove this

↓ Since all nodes are searched if needed.

↳ Path cost is a non decreasing * function of depth of node, so visiting shallowest first is optimal.

- Goal test when node added to frontier.

b) **Uniform cost search**: (Dijkstra is similar)

↳ Node to node (~~path~~ step cost) can vary.

- Expand node with lowest path cost first. → so its optimal

* - Goal test applied on node, when its selected * for expansion (not when node is added to frontier, unlike BFS).

↳ Optimal

↳ If a node is selected for expansion, optimal path has been found.

** ↳ Time complexity,

$$O\left(b^{\left(1+\lfloor C^*/\varepsilon \rfloor\right)}\right)$$

- C^* → Cost of optimal solution
- $\lfloor \cdot \rfloor$ → lowerbound
- ε → Min step cost
- b → max no of branches per node
- ignoring time costs of heap, etc

c) **DFS** → Non optimal } Returns first occurrence

- Expand deepest node in the frontier
- Space complexity ⇒ $O(bm)$ → Max size of stack.
  - b → branches per node
  - m = depth max
- Time complexity → $O(b^m)$

⇒ tree search can run in loops → since visited state is not maintained

*d) **Depth Limited Search**: → even tree search won't loop infinitely

- DFS for a limited depth l.
- If $l < d$ } Incomplete, $l > d$ } non-optimal

(d ↑ depth of goal node.)

** e) Iterative deepening search: (Depth first)

b^2x(2-1)
(d=2)

- Similar to depth limited search where the depth is increased in each iteration. (Perform search from root every time)
- Complete & Optimal like BFS.

Memory: O(bd)

Time: $d \times b + (d-1) \times b^2 + (d-2) \times b^3 + \dots + 1 \times (b^d) = O(b^d)$

* f) Bidirectional Search:

- BFS from source to path & from path to source as well.

⇒ When they meet we have found a path.

$b^{(d/2)} + b^{(d/2)} = O(b^{(d/2)})$

- If there are multiple goal nodes, make a dummy parent for all of them and backward search from the dummy parent.

— x —

2) Informed Search Strategies: (f(n) & h(n))

→ f(n) should always depend on h(n)

* a) Best First Search: (Greedy)

- Expand node closest to goal.
- $f(n) = h(n)$, The node with smallest f(n) is expanded first.
- h(n) is a heuristic of closeness to the goal. (need not always be perfect.)
- Incomplete on tree search. → Can loop ∞'ly

ex: $h(n)$ = Straight line distance to goal node, where we consider cities connected by nodes.

b) A* Search:

- $\boxed{f(n) = g(n) + h(n)}$

$g(n)$: Cost to come to this node from source.

$h(n)$: Heuristic of closeness to goal

⇒ Here $f(n)$ = Cost of estimated path from source to goal through node n.

- Similar to uniform cost search but $f(n)$ is different here. → Stop search when you expand goal node.

→ We also don't visit already visited nodes again.

* Conditions for optimality:

Tree search optimal ← i) Admissibility: $h(n)$ never overestimates cost to reach the goal

- Optimistic: $h(n)$ thinks cost of solving is lesser than actual.

Graph search optimal. ← ii) Consistency: (monotonicity)

$$h(n) \leq c(n, a, n') + h(n')$$

→ $h(n)$ is the shortest possible to goal.

If we consider distances for $h(n)$ etc ← → We can prove for this using triangle properties

- Every consistent heuristic is admissible. (not necessarily vice versa)

* Proof for optimality

→ If $h(n)$ is consistent, then ① $f(n)$ along any path are non decreasing

* Note: Minimax, Alpha-Beta Pruning, ML, (Monte Carlo Trees)
↑
Adversarial Search

$g(n') = g(n) + c(n, a, n')$

$f(n') = g(n') + h(n') = g(n) + c(n, a, n') + h(n')$

$\geq g(n) + h(n) = f(n)$

$\therefore f(n') \geq f(n)$

2. (2) When we select a node n for expansion, then the optimal path to that node has been found. (Prove by contradiction)

- If we have expanded a node, then any other node reached from an expanded node will have a greater value of f.
- So expanded node will have the most optimal path.

⟹ Properties of A*

- See slides.
- Optimally efficient → It expands out least number of nodes to find the optimal path.

* ∴ A* is optimal, complete & optimally efficient.

Note: Time complexity of A* (Tree search) $= O((b^{\varepsilon})^d)$, $\varepsilon = (h^* - h)/h^*$,

$\Rightarrow$ Tree with b branches per node

h is our heuristic
h^* is actual

—— x ——

Note: The only problem with A* is the space required.

—— x ——

Note: IDA* } Iterative deepening A*

- Similar to iterative deepening search.
- At each iteration, cutoff value is smallest f-cost that exceeded cutoff in previous iteration.

- We expand the smallest ** node till its f value is smaller than the smallest cousin node of the node we are expanding.
- Then we update the nodes f-value

Recursive Best first Search (RBFS) → save space

- Uses f-limit variables to keep track of f-values.
- This is optimal if h(n) is admissible
- Similar to A*, but we try reduce space for increased time.

running(F) with the f value of this child and again repeat the process

—— x ——

C) Simplified Memory Bounded A* (SMA*)

(Uses all the given memory) (Space adaptive)

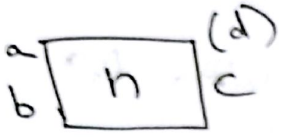
- Expand best leaf node till memory is full.
- If memory is full, we drop the worst leaf. We store the f-value of this node in its parent, so we can use it if needed later. (f-limit variables)

The parent stores the best f-value of all its dropped children.

- We will use the f-limit values and re-expand a dropped node only if the f-limit is the least of all current f-values of leaf nodes.

———x———

* ex:



$a = g(n)$
$b = h(n)$
$c = f(n)$
d = f-limit

- Use this notation! to represent nodes

———x———

*Note: See slides for an example.

———x———

* backing up.
- When we encounter the goal state through a certain path, we backup its value to its parent as f-limit ~~till root node~~.
- We also change the f value of this node to the f value of ~~the~~ over goal since we know the cost to G through this node.
- Also if we visit all the children of a node, we update its f-value to min of children

? ⇒ Now on encountering a goal, if the f-value is the least among all leaf nodes f-values & least among stored f-limit for the current node. Then we can finish ~~over~~ search.

- If current f value is lesser than f-limit, we discard the limit. (After visiting all children)

———x———

Note: ⇒ Expands newest best leaf & drops oldest worst leaf in case of multiple best/worst nodes

———x———

- When we come to a node but can't expand it due to space limit, we set its $h(n)$ as ∞, since we cannot expand it anymore due to memory constraints.

* ⇒ SMA* is complete if the space provided is greater or equal to length of path from source to goal.

———x———

3) Multiagent & Game Theory:

* a) Minimax: (Slides for example)

- Maximizer – Player
  Minimizer – Opponent
- Ply is one move deep.
  (ex: A 3 ply game tree has a depth of 3)
- $O(bm)$ space & $O(b^m)$ time complexity.

* b) Alpha-Beta Pruning:

* *

- Skip some nodes, if the utility you get in one of their children is not what you want.

ex: $\max(\min(3,12,8), \min(2,x,y), \min(14,5,2))$

$\Rightarrow \max(3, \leq 2, 2) = 3$

- So since our min value 2 itself is lesser than our current best which is 3, we can stop expanding that subtree.

- For a node n, if there is a better choice m at the parent node, then we can skip n.

⇒ α = Best choice value along the path, for MAX.

β = Best choice value along the path, for MIN.

⇒ For leaf $\alpha = \beta$ = Utility.

- At Max node, α = Largest child utility so far, else use parent's α.

  β = β of parent.

- At Min node, α = α of parent.

  β = Smallest child utility so far.

* ∴ For any node,

$$\alpha \leq \text{Utility} \leq \beta$$

- So at a certain node if $\alpha > \beta$, then
* stop search at that branch.

ex:- This means at the parent of this min node, the MAX parent has another path which we will choose for sure. So we can ignore this subtree containing this MIN node.

vice versa current max node

Note:- ⇒ We can't do this for our own node unless we store α', β' which are the opposites of α, β for each node.

ie. At max node, α' = Smallest child utility so far etc.

Note: Use iterative deepening search with
* minimax assuming a good heuristic along with α-β pruning to make a good bot. (Timer to try the deepest search possible)

---

* Note: Dynamic Move Ordering:
  - When re-traversing a same node again in iterative deepening search etc. Traverse the nodes which you found to be best before, first.

---

* Transposition Table: (DP kind of)
  - Store a hash table of visited states utility values since if we use a different move sequence to end up in the same state (transposition), we don't need to calculate the utility of the same state again.

Note: In ~~Hee~~ Heuristic Minimax since we can't calculate the utility of leaf nodes without traversing a large tree, we set a limit on the depth & use a heuristic → cut off test. (Not a good test)

---

* → This isn't very effective, so we use linear weighted functions of features

Note: Features of a game can determine equivalence classes, which all get the same value from our heuristic but they may lead to different game states.

Heuristic returns ex: count of unit ignoring position

---

* Note: Quiescence search → Do more search at peak periods of a game. (Do not cut off search at this point)
(Prune less at peak periods)

Singular extensions → Remember a move which is clearly better than all other at a given state.

Forward pruning → Some moves are directly pruned

↳ Beam search: consider top n moves given m possible moves at a given state

—x—

!?

* Probcut (Probabilistic cut) (Variant of α-β)

- Uses statistics from prior experience and performing a shallow search.

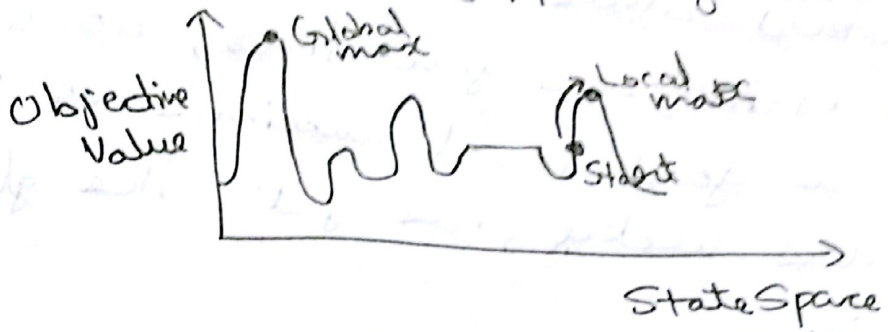
⇒ Then checks for likelihood of our value we get from the search, that it would stay in the row of (α, β).

—x—

4) Local Search:

a) Hill Climb Search:

- To get maximum, we move in the increasing direction till the peak is reached. (Opposite for minimizing cost)



- Memory requirements are low since we don't need to store path to get to local maxima/minima. (No tree nothing, just the current state)

---

* exc: 8 Queens problem, we don't use a tree, but consider the 8 × 7 possibilities from current state.
  - 8 → The no. of the queen
  - 7 → The column possibilities (We ignore the column where we currently are)

  ⇒ At each point, expand on the neighbour with lowest heuristic which is also lower than us (or equal)

  (Greedy local search)

---

* Note: If we have all adjacent states with equal reward as current state, we set a limit on no. of moves, and move to the adjacent states which are equal till our limit exists.

  ↙ Hill Climb Search

---

b) Stochastic Hill Climbing:

- Use probabilities, to choose best adjacent state.

—x—

*c) Random Restart Hill Climbing:

- The start state is reset & algo is run again for a few times, so we get a good local maxima or if we are lucky, we get the global maxima.
- No of restarts $= \frac{1}{p}$ (p = Success Probability to get global maxima/minima)

(Last one is success, all others failure)

* $\Rightarrow$ Total No of Steps = No of steps for success $+ \left(\frac{1}{p} - 1\right)$ (No of steps for failure)

(A very effective strategy)

—x—

*d) ~~Simulated Annealing~~ Random Walk:

- A very inefficient but complete algorithm (Since we select all nodes at ~~random~~ random)
- From every step we just keep randomly selecting a state, which is allowed.

—x—

*e) Simulated Annealing:

- Its a combination of hill climbing search & random walk.
- We pick a random state, if it improves situation, we accept it. If it does not improve situation, we accept it at a lower probability, and reset probability to high again.

⇒ Initially temp = T, at each iteration of random restart, drop temp by some value x & this would allow us to pick even bad moves if prob is high, & then we reset it. and choose random start point

If the new state is worse than previous state or vice versa a

⇒ $p' = e^{(\Delta E/T)}$ (T: some constant)

p' ↑ Prob to accept bad moves

ΔE = Next Value − Current Value.

−ve if we choose a bad move.

- This T is the temperature, which is dropped on each reset of the simulated annealing search.

*f) Local Beam Search: (Memory stores k nodes)

- Keep k states in memory,

⇒ generate all the best successors.

⇒ Go to the successor if state is improving (similar to Hill Climbing search)

- The k states can communicate talking about how close/far the goal is.

g) Stochastic Beam Search:

- Similar to normal beam search, but uses probabilities to pick next states.

*h) Genetic Algorithm:

- Start with k random states (initial population)
- Each state represented by a string of characters.

⇒ A fitness function which returns higher value for a better solution.

1) • Select a few parents based on fitness score
2) ⇒ When we combine 2 states (parents) to generate new states, it is called a crossover.

3) ⇒ Also when generating new states, we can randomly mutate them, (Change 1 or 2 chars of the string).

- In step 1, we pick k parents based on their probabilities given by the fitness score. (Similar to stochastic beam search)

* • The crossover can be done by selecting a point & splitting the parents,

child 1 = $P1_a + P2_b$ | $P1_a$ $P1_b$

child 2 = $P2_a + P1_b$ | $P2_a$ $P2_b$

↑ Cut here

This is dependent on the application (where the cut would be the best)

- In mutations, randomly change one character of so of each child.

———×———

Note: The crossover tries to ensure children get good characteristics from parents & leave the bad ones.

———×———

* Note: Swarm Intelligence:
  - Similar to how ants etc behave.
  - Use localized rules to generate global best solutions.

———×———

Note: A good heuristic will ensure that the branching factor is close to 1 (Many children should not have same heuristic values).

———×———

*Proving Heuristics' domination:

⇒ If C^* is the optimal solution, $f(n) < C^*$, any such n will be expanded.

⇒ $h(n) < C^* - g(n)$ will be expanded.

- Suppose we have h_1 & h_2 which are admissible.

⇒ Suppose $h_2 > h_1$, then h_1 expands more nodes than h_2 & it expands all nodes of h_2.

∴ h_2 dominates h_1. (∵ h_2 expands least no of nodes)

———×———

Note: Our heuristics we develop are the best possible solutions for altered versions of the game. → relaxed game

↳ This is how good admissible heuristics are

- A solution to the original game must be a solution to the relaxed game.

ex: In the 8 puzzle game, we can relax it by, allowing tile to move to adjacent square even if there is another square adjacent to the tile (and its not an empty sq.

———x———

* Merging Heuristics:

- Suppose we have multiple heuristics, then $h[n] = max\{h_1[n], h_2[n], \ldots, h_m[n]\}$
- This $h[n]$ is admissible, consistent & dominates all the other heuristics $h_1, \ldots, h_m$. (Considering the component heuristics are all admissible).

———x———

* Pattern Databases:

- If a pattern (subproblem) of the given input is present in the database, then we use its cost from the database (Admissible Heuristic)

———x———

Note: Disjoint Pattern Database:

- Cost of a problem = sum of costs of all the subproblems.

⇒ But the cost of the subproblem is solely the cost of blocks/elements which are a part of the subproblem but not any other subproblems.

- This will also be admissible.

———x———

* Inductive Learning:

- Give some features & a training dataset to the agent and it will assign weights to features.
- It will then use these features to compute the heuristic

———x———

Decision & Probability Theory

Decision Theory → Prob. Theory + Utility Theory

* ↳ MEU (Maximum Expected Utility)

*Conditional Probability:

- $P(A \mid B \wedge C)$ ⟹ A occurs given both B & C.

- $P(A \mid B) = \frac{P(A \cap B)}{P(B)}$

—— x ——

Note: If we use 2 variables then its a joint table distribution is given. → probabilities of each combination

—— x ——

*Marginalization (Sum)

$$P(Y) = \sum_{z \in Z} P(Y, z) = \sum_{z} P(Y \mid z) \cdot P(z)$$

- This is conditioning on z.

—— x ——

* Independence: (ex: Slides)

- Reduce no. of entries in distribution table, by splitting it up.

$P(A,B,C)$ (2, 2, 2 → 3 No. of values) $= P(A \mid B \wedge C) \times P(B \wedge C)$ ← $(P(A,B,C) = P(A \wedge B \wedge C))$

$= P(A) \times P(B \wedge C)$ ⟹ Considering A is independent on B & C.

∴ We only need 2×2 & 2 values = 6 instead of 8

Note: P(effect|cause) → Doesn't change much = Causal Knowledge → use Bayes rule to calculate

P(cause|effect) = Diagnostic Knowledge

---

* Relative Likelihood:

$\Rightarrow P(M|S) = P(S|M) \times P(M) / P(S)$

$\Rightarrow P(W|S) = P(S|W) \times P(W) / P(S)$

- Suppose S = Effect, M, W = Causes, P(S) can be hard to obtain at times.

$\therefore P(M|S)/P(W|S) = (P(S|M) \times P(M))/(P(S|W) \times P(W))$

So we relatively compare diagnostic knowledge.

---

* Computing effect:

- $P(M|S) = P(S|M) \times P(M)/P(S)$ (causal)
- $P(\sim M|S) = P(S|\sim M) \times P(\sim M)/P(S)$ (causal)

(Assuming we can find $P(S|\sim M)$ easily)

$\Rightarrow P(M|S) + P(\sim M|S) = 1$

$\Rightarrow P(S) = P(S|M) \times P(M) + P(S|\sim M) \times P(\sim M)$

---

* Conditional Independence: (From Bayes rule)

- $P(A \wedge B|C) = P(A|B \wedge C) \times P(B|C)$

Now, $P(A|B \wedge C) = P(A|B)$, if A & C are conditionally independent given B.

Utility Theory:

⇒ Risk Aversion:

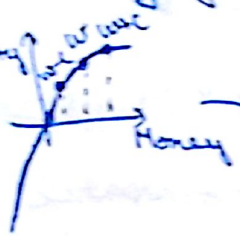
- Slope of our utility function keeps decreasing then we can say that we have risk aversion.

———x———

Note:

⇒ Monetarily fair bet → Utilities do not give us an expectation of 0 → utility of money varies

→ Bets don't prefer this since it gives us a -ve utility.



———x———

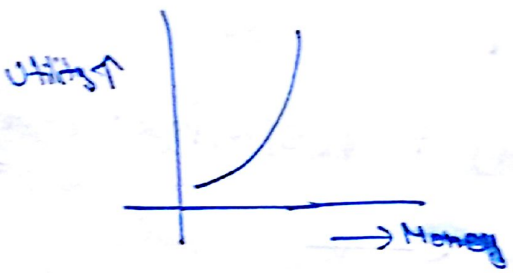
Currently we have w, on loosing → w−c (0.5 prob)
on winning → w+c (0.5 prob)

∴ Our expected utility is lesser than neutral

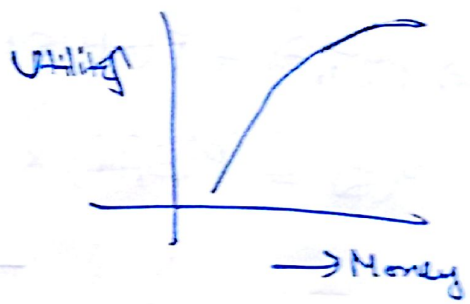
$$\because \frac{U(w-c)+U(w+c)}{2} < U(w)$$

———x———

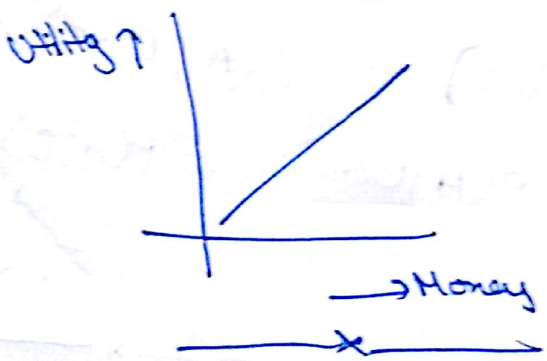
Note: Risk Seeker



Risk Averse



Risk Neutral



———x———

Note: If an agent is maximizing its utility function * then it's always a rational agent.

---

* Markov Decision Process: (MDP)

- This is a stochastic model which mimics reality well.
- MDP Tuple ⇒ $< S, A, P, R >$
  - S → current state
  - A → Action
  - P → Transition function (ex: $P(s' | s, a) = 0.3$)
  - R → reward function
- Policy is a collection of MDP tuples.
- Markov Assumption → Transition probabilities & rewards are only dependent on the current state but not the previous history.

*

⇒ **Decision Epoch:**

- Points at which decisions are made

a) Finite horizon MDP ⇒ No. of decision epochs are finite.
   - the policy (MDP) need not be stationary, ex: Policy = {D1, D2, D3, D4}
   - ex: fixed time blocks for game to end.

b) Infinite horizon MDP ⇒ No. of decision epochs are infinite.
   - Policy is the same at any decision epoch = D
   - ex: Time independent.

⇒ **Absorbing State:**

- Goal states * in our model.

- Reward function ⇒ R(S, A) (If all actions have the same reward then R(S) can be used)
  - S → State, A → Action

* ↳ $R(S,A) = \sum_{J} R(S,A,J) \times P(J|S,A)$

Stationary Policy:

* • At any decision epoch, the policy is always the same = D
  (Like infinite horizon MDP's)

Non stationary Policy:

• The policy can change depending on the decision epoch.

———x———

* Deterministic Policy:

*

• Given a state, we are forced to take a certain action since its probability is 1.

Random Policy:

• Given a state, each action has a probability defined.

———x———

Note: Standard → Infinite horizon (If not specified)

* • Policy at a time horizon → just that time horizon
  • Policy for a time horizon → for all 1 to x time horizons

———x———

Note: Stationary + Deterministic policy ⟹ Pure policies

———x———

Note: In plans → Failure of bot can happen (If we go to an unintended state we are lost)
In policies → No failure ever occurs. (Because we check which state we reach)

———x———

* Computing optimal policies:

I) Value Iteration: (Converging Algorithm) (Some sort of DP)

* • Iterating, for each iteration, for every state I, we update its utility considering older utility of adjacent states "J".

$$\rightarrow U_{t+1}(I) = \max_{A}\left[R(I,A) + \sum_{J} P(J/I,A) * U_t(J)\right]$$

- $U_{t+1}(I)$: * Bellman Update.
- $\max_{A}$: Actions allowed by policy
- $R(I,A)$: Reward of action A on state I.
- $\sum_{J} P(J/I,A) * U_t(J)$: Going to the next state and its reward.

• Algorithm stops if $U_{t+1}(I) \approx U_t(I)$ for all states. (Or within a certain threshold gap.)

$\Rightarrow$ This is when we say the algorithm converges.

* $\Rightarrow$ This can be used for infinite horizon MDP's & also finite horizon MDP's with stationary policies. (We just run this algo till it stops & use the values in any of the above MDP's)

Note: For 2 policies use the value iteration ** algorithm (Note: Each policy only allows one action at a certain state) and the better policy is the one which has a better utility for your start state.

Note: If we need to figure out the best move at a certain state, we run the algo with A = All possible actions, since here we are not constrained by policy.

Note: π → A policy
π^* → Optimal policy

---

* Markov Chain: (Fixed Policy)

- From the MDP if our policy is fixed, we get a markov chain (From each state, we have a determined action we want to take).

⇒ So we can call a fixed policy, a markov chain. $\left(U_{t+1}(I) = R(I,A) + \sum_j P(j|I,A) \times U_t(j)\right)$

Only one action possible

↑ Discounting:

- Rewards in future timestep i is discounted by γ^i. $(\gamma^i \leq 1 \ \& \ \gamma^i \geq 0)$
- If we don't do this, then the agents accumulate ∞ rewards. (We want the rewards quickly :P)

(Typically $\gamma \approx 0.95$). ∴ $U_{t+1}(I) = \max_A \left[R(I,A) + \gamma \times \Sigma \ldots\right]$

?? Note: If we say its a finite horizon MDP with x epochs, then in our algo we can store no of steps to reach & if its $> x$, then utility $= -\infty$.

II) Linear Programming:

- $Z = c_1x_1 + c_2x_2 + \dots + c_nx_n$

(We try to maximize our objective function)

$\Rightarrow x_1, \dots x_n \rightarrow$ Decision Variables, $x_i \geq 0$

Constraints:

$$a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n \leq b_1$$

$$\vdots$$

$$a_{m1}x_1 + a_{m2}x_2 + \dots + a_{mn}x_n \leq b_m$$

$\Rightarrow$ Applying this to MDP's, (This is a primal form, but we prefer the dual form)

$\Rightarrow$ we maximize $\sum V_j$; $j \in S$

Constraint,

$$V_i \leq \left[R(I,A) + \gamma \sum P(J/I,A) \times V_j\right]$$

---

* Note: Simplex Programming → Best way to solve linear programming problems!

$\Rightarrow$ We try to maximize, $\sum_i \sum_a x_{ia} r_{ia}$

$\Rightarrow$ $\sum_a x_{ja} - \sum_i \sum_a x_{ia} p_{ij}^a = \alpha_j$, $x_{ia} \geq 0$

- x_{ia} = Expected no of times action a is taken in state i
  r_{ia} → Reward for taking a from state i.
- α_j = Initial probability of being in state j.

⇒ So the constraints can easily be modelled as (ex: in slides)

$$Ax = \alpha$$

- x: Length = Sum of all possible actions from all possible states
- α: Initial State Probabilities

- Each element of A is.

$a_{j,(i,a)}$ → (i,a) denotes columns, j denotes rows

$$(\delta_{ij} - P_{ij}^{a})$$

- The constraints can then be used to find the values of all x_{ia}.

LP for MDP's:

** Now for each action we get the flows, so in our policy, it can be a non-deterministic policy where each action is assigned a probability, depending on the flow.

Value Iteration for MDP's

** For each state we only store the action which gives us the max utility, so our policy is deterministic. (Single action from each state).

---

*Note: Noop actions can be used on terminal states to direct flow out of the system.

---

*Multiagent MDP: (Teamwork models)

- A, R are team based.

⇒ A = $\langle a_1, a_2, \ldots, a_n \rangle$ } If n agents are present.

→ A = Set of joint actions

- State is fully observable by all agents. (All agents can talk to each other).
- In absence of communication, ~~random~~ policies can cause miscoordination

*Dec-MDP: (Teamwork models)

- Similar to MMDP, but we only have information on which state we have reached & make assumptions on where other agents have reached etc.

⇒ Based on these assumptions, we choose the reward function required.

—x—

—END of Mid 2—

*POMDP's: (Observational Uncertainties)

I) ⇒ Search with Partial Observations: (N. observations?)

- We don't know the start states, so action can lead to several outcomes. (even if the environment is deterministic).
- We search in belief states rather than just states.

a) * • Belief states → 2^N, if there are n states.
   ↳ We believe that we could be present in any of these belief states.

⇒ Initial state = Set of all states (largest belief state)

b) • Action set for a belief state = Union of actions for all states in belief state.

Note: * (For some types of problems, (actions rise to negative effects if its undefined), then we take the intersection of actions of all states in belief state.

---

c) * ⇒ Transition Model:

↳ Deterministic: $b' = Result(b,a)$
$= \{s' : s' = Result(s,a), s \in b\}$
(Each action on a state leads to another single state)

↳ Non Deterministic: $b' = \bigcup_{s \in b} Results(s,a)$
(Each action on a certain state can lead to several states).

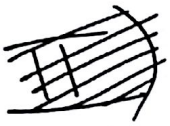
d) ⇒ Goal State: If all physical states ∈ Belief state satisfy the goal test, then the belief states is a goal.

* ⇒ Step Cost: Assuming action cost among all states is the same, the action cost of belief state is the same.

* - Pruning out belief states is essential (remove identical belief state.

* ⇒ If a belief state is a goal state & a subset already exists, then we can discard our current belief state. (Since subset is already a goal state)

——— x ———



Note: Algorithms for belief state: search

** 1) Incremental belief state search:

AND operation (All physical states belonging to belief states)

- Build up the solution, one physical state at a time, so that most states get pruned & memory does not become a constraint.

⇒ exc: Find solution for state 1, (→ set of actions) then check if it works for state 2, and so on.... (If its: invalid etc, backtrack to state 1 & try another action).

(If belief state = {1, 2, ... n})

——— x ———

* II) Search with observations: (Partially observable environments)

- If sensing is non-deterministic,
  PERCEPT(s) = Set of possible percepts } Partially observable
- In fully observable PERCEPT(s) = s, if no sensor PERCEPT(s) = NULL.

* Transition Model: (Example in slides)

* a) • Prediction stage, $\hat{b} = PREDICT(b, a)$
  (Possible set of states from b on action a)
  (belief state)

  b) Observation prediction,
  $POSSIBLE\text{-}PERCEPTS(\hat{b}) = \{o : o = PERCEPT(s), s \in \hat{b}\}$
  - These are all the percepts that are currently available.

* c) Update stage, (Create clusters)
  $b\_o1, b\_o2, \ldots$ = Clusters, where each one is a belief state, that from state b, on action a, from $\hat{b}$ which states satisfy o1, o2 .....

(Vacuum world) exc: o1 = B.Dirty
b_o1 = The states from $\hat{b}$ where agent is in cell B & observes it's dirty

---

Note: Generally agent can only sense observations within the cell they are in.

Note: * Non determinism in partially observable problem comes mainly due to the inability to predict which percept we will get after taking the action.

Note: The results from the transition model gives us a tree on which we can run a search algorithm to get our solutions (AND-OR)

---

realtime search → conditional plan, not a sequence

*Note: In MDP's → Observations are directly observable. They determine the specific state.

In POMDP's → Observations can lead to many possible states.

---

*Bayesian Networks: (Belief Nets)

*(Example in slides)

- Efficiently represent joint probability distributions
- 2^N joint probability values need not be specified. → So domain experts can get values much more easily.

⇒ Bayes Nets → Knowledge based system.

- Nodes are random variables.
- If X influences Y directly X → Y. X is a parent of Y.

⇒ No cycles present, so it's a DAG.

* • We need to give CPT data for each node's CPT (Conditional Prob Table)
  - → Considers all parents of the node.
  - → For nodes with no parents, we just give it a probability of occurrence.

* ⇒ $P(X_1 = x_1 \text{ (True/False etc)} \wedge X_2 = x_2 \wedge \dots -)$

$$= \prod_{i=1}^{N} P(x_i \mid \text{Parents}(x_i))$$

→ Derivation is simple, just use $P(A \wedge B) = P(A|B) \cdot P(B)$ recursively.

* For a node, if its dependent on all of it's parents, then it becomes conditionally independent on all other nodes.
→ This is also used in the proof derivation.

Compactness:-

- If each node has ∃ k parents,
  → $N \cdot 2^k$ entries in CPTs.
  compared to
  2^N entries in JPD tables

Building the Belief Net:

** • Choose a good ordering of nodes.
  • Pick variables one by one, add a node to network & then set parents(X_i) to a minimal set of nodes already in the network which directly influence X_i.
  Now create the CPT for X_i.

(One way links, so we always get a DAG)

——— x ———

Note: Causal models are better, Causes → Symptoms

** P(effect | cause) } Causal prob.
↳ these are easy to calculate.

(So node ordering should follow this kind of causal selection).

• Random orders may increase links etc.

——— x ———

Inference in belief nets:

(ex: in slides)

- Causal
- Diagnostic
- Mixed
- Explaining away

———x———

*

*Note: Path Finder → Diagnostic expert system for lymph node disease.
→ Uses belief nets.

———x———

* Constraint Satisfaction Problems: (CSPs)

(eoc: in slides)

- Assume states have a factored representation
  - → They are richer
  - → Problem solved when each variable satisfies all constraints put on the variable.

⇒ We try eliminate large regions of search space which violate some or the other given conditions.

* • CSP → X set of variables
D set of domains for variables
C → constraints.
→ <scope, rel>
scope → Variables involved
rel → Relation among them.

- A state = An assignment of values to some or all variables $(X_1, X_2, \ldots)$

* ⇒ Consistent or Legal assignment → state where no constraints are violated
  Complete → All variables are assigned values.

- A solution to a CSP is a consistent & complete assignment.

*Constraint Graph:

- Nodes are variables
- Arcs are constraints

⇒ Using this graph, we play around with our state, narrowing down our variables accepted values so that every constraint is satisfied.

⇒ This graph helps us prune out large parts of our search space.

———x———

*Note: Jobshop scheduling → slides

———x———

* Types of CSPs:

1) Discrete finite domains → {0, 1}

2) Discrete infinite domains → Integers etc

3) Continuous variables → Real numbers

———x———

Types of Constraints:

- Unary, Binary, ----
- Global constraints → Constraints involving any number of variables.

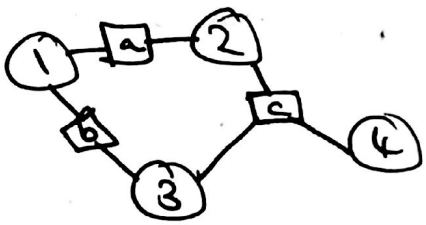
ex: All diff constraint, all variables in given set must be different.

Hypergraph: (n-ary constraints)

- To show relations b/w multiple variables we use circles for nodes & squares (hyperedge nodes) for linking multiple nodes for a certain constraint.

ex:

[x] = n-ary constraint.



Binary CSPs:

- We can convert any CSP to a binary CSP which has a lot of solutions.
- We use dual graph transformation for this conversion.

*Dual Graph Transformation:

- One variable for each constraint
- One binary constraint for each pair of constraints.

———x———

*Note: Preference constraints (Soft constraints) → optimization problem → Flexible,
ex: Prefer tea over coffee

Hard constraint → Hard rule, strict

———x———

*Note: performing Inference (choosing a value of a variable)
* is called constraint propagation → reduce no. of legal values for neighbours

———x———

*Note: Local consitency (Ensuring) → Neighbours don't have conflicting values in domains.
↳ So that there are no inconsistencies globally.

(CSP Inference) ———x———

*Types of Local Consistency: (Inference algorithms → cut down search space.)

1) Node consistency: If all values in the variable's domain satisfy the variable's unary constraints.

(Network is node consistent if every variable is node consistent)

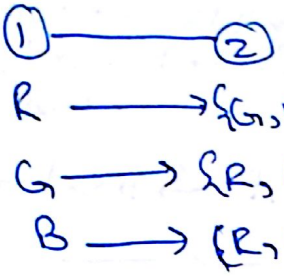
2) Arc consistency: If every value in a variable's domain satisfies the variable's binary constraints with all nodes connected to it.

* (Inference algorithms)
  ↳ Cut down search space

(Network is arc consistent if all pairs of variables (arcs) are arc consistent)

———x———

* Note: Arc consistency does not prune out the domain of any node, in graph coloring (face) since.



→ No pruning can be done using this one arc.

———x———

** Arc consistency Algorithm (Slides) (AC-3)
   O(c·d³)

———x———

* ↳ If X_i's domain changes when trying to make it arc consistent with X_k, then we will be required to check for arc consistency with all other $X_k \rightarrow \{X_i, X_k\}$ recursively. (we use a queue to get this done) → put them in queue again, else discard

———x———

* Note: In the Arc consistency Algo (AC-3), if a CSP has n-variables with a domain of size d each,
  ⟹ On each removal from queue, d → d-1 & reinsertion atleast & then checking consistency for a single arc = O(d^2)
  ∴ Total = O(c·d^3). since each constraint can be inserted d times.

3) Path consistency: (Inference algorithm)

- For a 2 variable (nodes) set $\{X_i, X_j\}$ is path consistent wrt X_m if, $\{X_i, X_j\}$ is arc consistent and
→ $\{X_i, X_m\}$ & $\{X_m, X_j\}$ are arc consistent
⇒ Some assignment for X_m, satisfies this

- PC-2 algorithm used to obtain path consistency.

4) K-Consistency:

- A CSP is k-consistent if for any set of (k-1) variables and for any consistent assignment to them, a consistent value can be assigned to the k^{th} variable.

———x———

*Note: Strongly k-consistent → k consistent, k-1 consistent, k-2 consistent..... so on

———x———

*Note:
⇒ For an n-consistent CSP, we are guaranteed to find a solution in $O(n^2d)$
($O(n \cdot d)$ for n ordered)
↑ For each variable scan its domain to find a solution

———x———

* Global Constraints: (Required if constraints are of higher order)

- ex: All diff constraint:

⇒ Specialized algo.: (Inference algorithm) 1) Remove variables with single values assigning it to the variable & remove it from domain of others.
Or if the variables > domain values 2) If empty domain → reject else repeat

*5) Naked Triples Strategy: (Another inference strategy)

- Used in sudoku.

⇒ If three cells in a row, column of a box (3×3 grid out of 9×9 sudoku), such that their domains are a subset of equal to a domain with just 3 values.

- We can then say that these 3 values will be used in these cells only, so we can remove these 3 values from all other cells.

———— x ————

*CSP search:

*1) Backtracking Search: (Uninformed search)

- 1st layer of the tree → Assigning values from d to n variables we have, $n \times d$ possibilities → Select a node & assign a value
- 2nd layer we have $n \times d \times (n-1) \times d$

Since we will fix a value from d for one of our variables

∴ Direct search → $n! \times d^n$

⇒ So we use inference.

- One of the orderings of values given to the nodes is enough, ie: $a=1$ & then $b=2$ or $b=2$ & then $a=1$ are the same

If all variables can take similar values

⇒ So 1st layer → d possibilities
2nd layer → d^2 possibilities

∴ Complexity → d^n
(Similar to considering only combinations instead of permutations.

- This is called a backtracking search
  - ↳ Try find a path from root to leaf which would be the values we assign to our n nodes.
  - ↳ If none found which satisfy the constraints
    - ↳ No solution.

⇒ This is an uninformed search algorithm for CSPs
(example in slides)

———x———

2) Minimum Remaining Values (MRV) heuristic:

*
- In the search (backtracking), select the variable to expand who's current domain is the smallest.

⇒ A huge speedup provided.

———x———

3) Degree heuristic:

*
- We use this heuristic to select the first state of our tree.

⇒ Select the variable with the largest no. of constraints as the start variable.

———

4) Least constraining value heuristic:

* to expand first
* Given a variable, choose a value for it which will be the least constraining for other unassigned variables.

⇒ So we have higher chances of finding a solution.

—x—

* Note: Using these heuristics, variable selection → fail first and value assignment → fail last.

—x—

*Combining Search & Inference:

*1) Forward Checking: (Reduce neighbours domain on value assignment)

- Once we assign a value to a variable, for each unassigned neighbour, reduce the domain according to the constraints.
- Terminate if domain for a variable is empty.

⇒ Forward Checking along with MRV is very effective.

—x—

*2) Maintaining Arc Consistency: (MAC)

- Once a value is assigned to a variable,

⇒ AC-3 is called and domain reduction happens on all unassigned variables.

↳ For the start of the queue start only with arcs (X_j, X_3)

where X_j = Neighbours of X
X = Start node which we choose

⇒ It recursively checks for neighbours as long as needed since it uses AC-3.

———x———

Note: MAC is strictly more powerful compared to Forward Checking.

———x———

* POMDPs (cont)

- Similar to MDPs. but we also have

$\Omega = \{o_1, o_2, \ldots o_n\}$ } Observations

$\Rightarrow O(o_i | s', a)$ = Prob of observation i when action a used to reach s'.

↑ Observation probabilities

- A state estimator uses our current belief states, the observation received & the previous action to determine our current state.

- Once a new belief state is decided, we use our policy to determine an action for our current belief state.

? (Policy table can be updated based on the history of observations)

---

* Tiger Problem: (Slides)

- Two doors, one has a tiger, other has food.

$\Rightarrow$ Two states = S_L (Tiger on left) & S_R (Tiger on right)

$\Rightarrow$ 3 Actions → Left, Right & Listen

- Transition probabilities table $\Rightarrow$ Column is the initial state
Rows are new belief state

- The game is reset when we take Left & Right action with tiger in either of the rooms.

⇒ Observations → TL , TR

TL ↑ Tiger Left TR ↑ Tiger Right

- Rewards of the form $R(s, a)$ → Taking action a from state s.

---

* **Policy Tree:** → (Tiger ex: in slides) ↳ Photos

- No. of child nodes per parent = No. of observations received.

⇒ No. of nodes in a tree,

$$N = \sum_{i=0}^{T-1} |O|^i = (|O|^T - 1)/(|O| - 1)$$

- So the no. of possible trees = $|A|^N$

---

* **Computing Belief States:**

$$b'(s') = Pr(s' | o, a, b) = Pr(s' \wedge o \wedge a \wedge b) / Pr(o \wedge a \wedge b)$$

(Previous belief state)

$$= \frac{Pr(o | s', a, b) \cdot Pr(s' | a, b) \cdot Pr(a \wedge b)}{Pr(o | a, b) \cdot Pr(a \wedge b)}$$

Normalizing factor

$$b'(s') = \frac{Pr(o | s', a) \cdot Pr(s' | a, b)}{Pr(o | a, b)}$$

b'(s') ↑ New belief state probs

$b'(s') = O(s', a, o) \sum_s Pr(s' | a, b, s) \cdot Pr(s | a, b)$

$= O(s', a, o) \cdot \sum_s Pr(s' | a, b, s) \cdot b(s)$

$\because Pr(s | a, b) = Pr(s | b) = b(s)$

* $b'(s') = O(s', a, o) \sum_s T(s, a, s') b(s)$

↑ Observation function

↓ Transition function

(We also need to divide by $Pr(o | a, b)$ for normalization)

⇓ or just divide by sum of probabilities.

example in slides → Photo

Note: Unnormalized prob denote with $\bar{b}'(s)$

Note: In belief MDPs we dont use states
* 's', 's'' anywhere, only belief states are used. → ex: $T(b, a, b')$, $R(b, a) = \sum_s R(s, a) \cdot b(s)$

↳ So we calculate for states & then get $T(b, a, b')$ etc.

Note: * We can use value iteration for belief MDPs, but we have ∞ belief states, a continuous state space → if we don't know the start belief state.

↓ Generally this is the case

---

* CSP: (Cont)

* 3) Intelligent backtracking → Looking back:

- Going back to parent node when we don't find a solution is chronological backtracking

* ⟹ In intelligent backtracking, if I don't have a value for my state, I back jump to the state which does effect my current state.
  - ↳ Skip states which don't effect me.

* • For each node maintain a conflict set, use this for the jumps
  - → Go to most recently used in conflict set.

⟹ Conflict set is easy to create, with forward checking,

* • If X=x deletes value from Y's
* domain, add X=x to Y's conflict set.
- If last value deleted from Y's domain, add conflict set of Y to X.

- This simple back jumping is redundant since in forward checking, we don't visit a state at all if its domain is empty.

4) Conflict Directed Backjumping: (Slides for example)

- We learn from conflict by updating conflict set of the variable we have jumped to. → Discovering knowledge
  ↳ Algo learns these new conflicts

⇒ We would have a better conflict state.

- $Conf(X_i) \leftarrow Conf(X_i) \cup Conf(X_j) - \{X_i\}$
- When from X_j we backjump to X_i.

⇒ We use information of both parents & children of a current node. So we have a richer conflict set.

5) Constraint Learning: (Remember bad assignments sort of thing)

- Find a minimum set of variables from conflict set which cause the problem and add a new constraint to our CSP.

⇒ If we don't want to add directly to the CSP, we maintain another set of "no-goods" constraints where this can be added.

*Local Search:

- Use complete state formulation

⇒ Initially assign values to all the variables

⇒ Change values of variables during the search to reduce conflicts.

* • Min conflicts heuristic ↑ Select value of variable so it results in minimum no. of conflicts with other variables.