A close up of a mans face

Description automatically generatedAI is anything computers couldn’t do 30 years ago => 1. Systems think like humans 2. Systems think rationally **3. Systems act like humans** 4. Systems act rationally

Agents: Something that does stuff. Includes humans, robots, thermostats…

PEAS model: Performance measure, Environment, Actuators, Sensors

Eg: PEAS for self-driving car 🡺 **Environment**: street, traffic, pedestrians, weather,.. **Sensors**: video, accelerometers, LIDAR, engine sensors,… **Actuators**: steering, brake, horn, display,… **Performance**: safety, legality, speed, comfort,…

**Agent function** is mathematical object and **agent program** is an implementation. AI is about creating agent function.

Rational agent chooses whatever action maximizes the performance 🡺 aiming to produce rational agent or as close as possible.

Environment types: 1. **Observable (vs partially-observable)**: we can see everything we need to know about the world.

2. **Deterministic (vs stochastic)**: actions always have the same result. 3. **Single-agent (vs multi-agent)**: how many actuators. 4. **Discrete (vs continuous)**: the env is distinct states or smoothly varying states. 5. **Episodic (vs sequential**): the agent’s experience is divided independent episodes.

**Chap3**: **state** = unique arrangement of the world. A problem is defined by 4 things: **Initial state, successor function** (define how action influence state), **goal test** (define when we are done), **action cost** (if we care about total num of actions: cost = 1, if don’t care and want any solution reaches the goal: cost = 0.

A screenshot of a cell phone

Description automatically generated🡺 step cost has to be greater than epsilon and not equal to 0. (UCS)

**Complete:** finds solution if exists

**Optimal:** finds the lowest-cost solution

C\*: cost of goal. Epsilon: least-cost action.

BFS: fringe uses queue DFS: fringe uses stack

IDS: Time O(bd+1) 🡺 (d+1)b0 + db1 + (d-2)b2 +…+ bd+1 = O(bd+1)

UCS: fringe uses priority queue

**Checking for repeated states:** applies for linear problem with multiple paths 🡺 no loop 🡺 2n/2 paths 🡺 check for repeated states 2n

Mem using: O(bd) 🡺 may use for BFS or UCS

A screenshot of a cell phone screen with text

Description automatically generatedTime of Greedy is faster than BFS and DFS.

**Uniform search** uses heuristic for faster search 🡺 h(n) estimate cost to goal from n. depends on how good estimate of h(n).

**Greedy search:** fringe uses pqueue ordered by h(n) 🡺 pull out lowest heuristic value and expand it. (**NOT OPTIMAL**)

**A\*:** fringe pqueue ordered by f(n) = h(n) + g(n) estimate total cost thru n

g(n) = cost of path so far… h\*(n) = time optimal cost from n states

h(n) is admissible iff h(n) <= h\*(n) also h(n) >= 0.

ha(n) dominates h**b**(n) iff ha(n) >= hb(n) 🡺 ha(n) is better than hb(n) where hb(n) is lower bound and true value h\*(n) is upper bound.

**Q4:** given f(n) = (2-w)g(n)+wh(n) where w = 0 gives UCS, w=1 gives A\*, w=2 gives Greedy. Complete when 0<=w<2 and optimal when w<=1.

**Chap5**: Utility func is only defined at terminal state.

**Minimax alg**: Idea: assume our opponent picks the best move they can (worst for us). We can determine how good intermediate state is based on its children. Properties of minimax: 1. **Complete**: yes if game is finite 2. **Optimal**: yes if against perfect opponent 3. **Time**: O(bm) (same as DFS) but impractical for most game 4. **Space**: b\*m where **b = branching factor (# of moves), m = maximum # moves in game**.

**Alpha-beta pruning:** calculate strategies for the game 🡺 sequence of actions. Properties: Pruning does not affect result => still optimal. Effectiveness depends on move ordering. With perfect ordering, time complexity O(bm/2). Usually get good ordering.

**Heuristic**: h(state) = estimate utility for A / probability that A wins. Better evaluation function -> less cost from cutting off search. Alternatives to fixed depth: 1. Quiescence: search deeper in volatile board states 2. Search only “good” moves past a certain depth.

**Chap6**: **Constraint sat problem (CSP)** 3 parts: Variables, Domains, Constraints.

Using **search** to solve **CSP** 🡺 **State**: partial asg to vars (what the inputs are). **Action**: assign val does not violate constraints. **Goal**: full asg (desired result). 🡪 may use **heuristic for improvement.**

All solns at depth n 🡪 using DFS. Time complexity: n = # of vars, d = size of domains 🡺 # of actions: n\*d. depth: m=n 🡺 **Time**: O(bm) = O((dn)n) = O(dnnn) = O(n!dn). **Space**: O(bm) = O(dn2).

**Backtracking search** properties: Complete (yes), Optimal (N/A) all slns are similar, Time O(dn), Space O(dn) (the parameter assignment grows at rate of n)🡺 n = # of vars, d = size of domains(# of vals) 🡺 use heuristic to run faster. Improvements:

**select\_variable()**: 1. Minimum remaining values 🡺 choose variable with the fewest legal values 2. Maximum degree 🡺 tiebreaker among min remaining value vars. Choose var with the most constraint on remaining vars. **Order\_values():** Least constraining value 🡺 given a variable, choose the value that rules out the fewest values in remaining variables.

**Is\_consistent():** Forward checking 🡺 keep track of remaining legal values for unassigned variables and terminate search when any variable has no legal value. Arc consistency 🡺 X->Y is consistent iff for every val x of X there is some allowed y in Y. <= Expensive but helpful bc reduce search depth of backtracking O(n2d2)

**Local search:** starts will full solution but violate constraints and slowly improve until satisfactory. This alg is complete but takes long searching time.

Framework : 1. Initialize randomly to vars 2. Iterate: choose var and change it. 3. Stop when solution is satisfactory.

**Hill climbing search:** choose vars that violate some constraints, choose pos that generate least conflicts 🡺 constraint factor completes without running out of max\_steps.

Downside: local optima. We can get stuck where where allvariables cannot be changed without increasing number of violated constraints.

**Independent subproblems:** we can improve our algs if our problems have structure.

**Tree-structured CSPs:** trackable to solve. Can solve in polunomial time 🡪 No loops: O(n\*d2).

**Good heuristic** chooses the most constrained variable in order to cause a failure and it is more efficiemt to fail as early as possible. It chooses the least constraining value bc it allows the most chances for future assignments to avoid conflict.

**A close up of text on a white background

Description automatically generatedChap7**: **Logic agents:** knowledge base (KB): set of known sentences.

**Entailment** is one way: A |= B iff B logically follows from A. M(A) M(B) where M(B) is set of models.

KB |= a iff (KB => a) is valid. KB |= a iff (KB ^ not a) is unsat.

Satisfiability is a CSP:

Variables: propositional symbols

Domains: T/F

Constraints: Propositional sentences.

🡸 **USE THIS TABLE TO CONVERT TO CNF (ONLY INCLUDE OR, AND )**

**DPLL alg:** look for **unit clauses** and get rid of them. Look for **pure literals** and get rid of them.

Otherwise run func recursively on remaining vars until there is no more clause then it’s satisfied or there is empty clause which is unsat.

**WalkSAT alg:** hill climbing search alg. Randomly assign symbols and pick probability p where higher value of P may or may not get more accurate.

**Problem:** local optimum where it flips back andn forth 🡺 infinite loop.

**Logical inference:** 1. Proof by contradiction using satisfiability (if KB and not alpha is unsat then KN |= alpha) 2. Apply inference rules (generate new sentence until we get alpha)

**Note:** 1. **WalkSAT** does not tell us whether clause is sat. 🡺 cannot use **WalkSAT** to check KB |= S where S is sentence.

2**. Instruction for inference rules**: take 2 CNF clauses and output a new clause that is entailed by the first two. 🡺 get rid of complementary literals and concatenate everything else.

**Chap13**: Random variable (**Capital letters)** is a mapping from atomic events to values.

Domain of random variable: Set of possible values it can take. Using lowercase letters for values: true, false, heads, …



A screenshot of a cell phone

Description automatically generated**Conditional prob.** P(a|b) = P(a,b)/P(b) 🡺 P(a,b) = P(a|b)P(b) = P(b|a)P(a).

Each random variable is in one of 3 sets: **Query Y, Evidence E, Hidden H**.

🡺 denominator = 1/alpha where alpha = sum of all prob

A and B are **independent** iff: P(A,B) = P(A)P(B), P(A|B) = P(A), P(B|A) = P(B).

If there is 1 variable not depends from other 3 vars (P(C, T, C)P(W))🡺 8 vars needed (8-1 + 2-1) otherwise 16-1 = 15 vars needed.

For n coins, 2n-1 if coins are independent and n if coins are identical.

A is **conditionally independent** of B given C iff: P(A,B|C) = P(A|C)P(B|C) or (A ind B)|C

A screenshot of a cell phone

Description automatically generated**Bayes’ rule**: P(a|b) = P(b|a)P(a)/P(b)

🡸 alpha-beta pruning

A screenshot of a cell phone

Description automatically generated🡸 minimax

**Chap14:** **A Bayesian network** consists of: 🡪 a set of random vars (nodes). 🡪 a directed, acyclic graph over vars. 🡪 a conditional distribution for each node given its parents P(Xi | Parents(Xi)

Bayes nets are the way to represent conditional dependence. **Ful joint probability tables** are not useful in practice bc (1) too big. (2) not intuitive.

Bayes network is a way to represent a probability distribution using just **conditional probability tables**.

**The probabilities** do not add up to 1 bc they do not show probability of (what we want on the left) to be false

Eg: P(j,m,a,not b, not e) = P(j | a) P(m | a) P(a | not b ^ not e) P(not b) P(not e) where J and B are parents of A and A is parent of J and M.

**A bayesian network defines a full joint distribution:** A picture containing furniture, table

Description automatically generated

n Boolean variables 🡺 joint distribution parameters = 2n -1. Max k parents in Bayes net 🡺 O(2k) parameters per variable. 🡺 O(n2k)

**Conditional Independence:** A close up of a logo

Description automatically generatedfirst rule: P(A) -> P(B | A) -> P(C | B). Third rule: A,C are independent. A close up of a logo

Description automatically generatedfirst rule: Given B that A,C are independent.

**Continuous random variables:** we get a probability density function f(x) rather than a probability mass function P(x). 🡪 we need to reason the ranges of values.

**Inference in Bayes networks:** Query: A, B. Evidence: C. Hidden: D,E ?? **Methods**: - Construct joint dist. - Enumeration. - Variable elimination. -Sampling.

Simple query on the burglary network: P(B | j,m) = P(B,j,m) / P(j,m) = alpha\*P(B,j,m) = alpha\*

**Runtime of enumeration alg:** O(DH) where H: # hidden and D: domain size

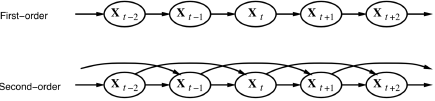
**Smoothing:** O(DH)

**Factor:** generalized probability table, without the sum-to-onr requirement. A fater *f* defined over random variables A,B and C defines a valie for each assignment f(A=a, B =b, C= c) = <value>

Operation on factors: Fix: restrict the domain of a factor (used to incorporate evidence), Marginalize: sum out a hideen variable, Product: input factors f1 and f2, output a factor with the union of the two domains.

**Chap15: Temporal Bayesian networks:** copy state and evidence variables for each time steps: Xt = set of unobservable state variables at time t and Et = set of observable evidence variables at time t.

**Markov process** First order: P(Xt | X0:t-1) = P(Xt | Xt-1). Second order: P(Xt | X0:t-1) = P(Xt | Xt-2, Xt-1)

A screenshot of text

Description automatically generated**Inference tasks:** Filtering P(Xt | e1:t) Smoothing: P(Xt+k | e1:t) 1 k < t Most likely sequence:

**Chap18:**

A screenshot of a cell phone

Description automatically generatedPerformance element: The actual program that chooses actions

Critic + learning: a program that changes the performance element based on feedback.

Feedback: how do we get feedback on whether choices were good or not? 🡪 performance.

**Inductive learning:** we want h(x) f(x). we have traning set:

Trade-off: consistency with training data vs generalizability.

Linear classifier: where w is the weight, x is value of feature and g is threshold function.

**Stochastic gradient descent:** a local search algorithm.

**Perceptron learning:** learn by adjusting weights to reduce error on training set. 🡪 using squared error to “smooth” error.

🡪 It is impossible to implement XOR using perceptron.

🡪 Perceptron learning rule converges to a consistent function for any linearly seperatable data set.

🡸 **McCulloch-Pitts unit (neural network unit).**

Multi-layer perceptron (MLP) 🡪 # of hidden layers is chosen by hand. (can have more than 1 hidden layer).

A picture containing object

Description automatically generated

A screenshot of a cell phone

Description automatically generated

**Expressiveness:** Decision trees can represent any function of the input. 2N possible assignments of inputs 🡪 possible functions.

Complexity vs generalizability: We want a compact decision tree – i.e. maximum depth.

**Choosing an attribute:** a good attribute splits the examples into subsets that are ideally “all positive” (H = 0) or “all negative” (H= 0) or mixed (0 < H < 1) 🡺 Choose attribute that has the lowest entropy.**A close up of a logo

Description automatically generated**

**Entropy:** how much uncertainty is in a sample. 🡺 H = 0 when p = 0 or 1. H = 1 when p = 0.5

A picture containing object, watch

Description automatically generatedwhere N is # of examples. P: probability of T/F. T = # of True samples at leaf i. F: # of False samples at leaf i.

**Performance measurement**: H f ? 🡺 using traint/test split. Bigger train set 🡪 better model. Larger test set 🡪 more accurate model.

Can use multiple splits: Train | Validation | test

Validation is used to choose model or parameters ( such as numbers in neural network)

Underfit when the model is too simple. Solving by increase the depth of the decision tree. A close up of a logo

Description automatically generated

Overfit when the model is too complex. Solving by decrease the depth of the decision tree.

At infinite examples, training set = test set.

Accuracy decreases when more examples are added.

**Performance measurement in practice:**

* Make sure evaluation set is a good model of real data.
* What performance measurement is appropriate for your setting.

(a) Neural network questionA screenshot of a cell phone

Description automatically generated: Input x1 = 0 0 1 1 x2 = 0 1 0 1 Output y = 1 1 0 1 Given bias input = -1.

A close up of text on a white background

Description automatically generated w0 is bias weight.

(b) all non-bias weights (w1,2, w2,3,…) = 1 and all the bias weights (w0,1, w0,2) = 0.5. giving an input x = 0.5 causes all outputs to become .

Suppose given x = 0.5 and y = 1 and use backpropagation to make one stochastic gradient descent update. 🡺 the update increases the input (non-bias) weights.

After the update, as non bias weight increase, w1,2 < w2,3 < w3,4 < w4,5

in = 0.5 – 0.5 = 0 Therefore, 0.5 🡪 a2 = g(0) = 0.5 🡪 a3 = 0.5 🡪 a4 = 0.5 🡪 a5 = 0.5 🡪 1

Backprop: Δ2 = 2^-8 🡪 Δ3 = 2^-6 Δ4 = g’(in4)\* w4,5\*Δ5 = ¼ \* 1 \* ¼ = 2^-4 Δ5 = err \* in5 = (y – a5) \* 0.5 = 0.5 \* 0.5 = ¼

w4,5 🡨 w4,5 + alpha \*a4 \* Δ5 = 1 + alpha 0.5\*1/4 = 1 + alpha\*1/8 w1,2 🡨 w1,2 + alpha\*0.5\*2^-8 = 1 + alpha\*2^-9

alpha = learning rate, we get to pick 🡺 too high causes divergent , too small takes too long.

**Guest:** Reinforcement learning framework: At each step t the **agent**: Executex action At, receives pbservation Ot, receives scalar reward Rt.

The **environment** receives action At, emits observation Ot+1, emits scalar reward Rt+1.

- Markov process + Reward Rs associates with state.

- Markov decision process = Markov reward process + actions 🡪 transition probs depend on actions. RL use factored representation.

**RL framwork**: Data (learning)🡪 trans probs<-full defined MDP (dynamic prog.) 🡪value func. Trajectory is a (possibly infinite) sequence. The return is the total sum of rewards. Given a policy and a MDP, we have the expected return over all infinite trajectories, from using a policy at a state.

A policy π\* is optimal if for any other policy and for all states s: Vπ\*(s) ≥ Vπ(s).

Bellmann equation for computing the value function with deterministic transitions:

**Summary:** Reinforcement Learning: learning to act. Adds *actions* and *rewards* to a temporal Markov model. Learning problems: -**Value function**: Compute the expected cummulative reward given a state for a given policy/an optimal lolicy. -**Policy optimization**: find an optimal policy. Value functions and policies can be represented in a neural net 🡺 Deep RL.