<u>**Chapter 6 - Constraint Satisfaction**</u> **Constraint Satisfaction Problems (CSP)** Definition: solutions with caveats Example: find a way to take classes such that I graduate in four years. Constraints are pre reqs, course availability, and funding Previously, states were **Atomic** - don't care about internal rep. except w/ respect to goal/heuristic' Mutated by actions to produce **new atomic state;;** Now - **Factored representations** States have **internal structure** Structure can be **manipulated Constraints** related different parts of the structure to one another and provide legal/illegal configurations **CSP Definition Problem = {X, D, C} X - set of variables** X = {X1, X2, … Xn} **D - set of domains** D = {D1, D2, … Dn} such that Xi = xi where xi in Di **C - set of constraints** C = {C1, C2, … Cn} such that Ci = < (Ca, Cb), relationship(Ca, Cb)> **CSP Example: Map Coloring** Color territories on a map using 3 colors such that no two colors are adjacent; Can be represented as a **graph** with adjacent area connected by an edge; All variables have the same domain (the three colors); Constraint set consists of areas adjacent not being equal **Constraint Types Domain values** - time at which task begins {0, 1, 2 …} **Precedence constraints** - a + b <= c **Disjunctive constraints** - e.g. can only do one thing at a time => a + 10 <= b or b + 10 <= a **Unary** - single variable (z <= 10) **Binary** - between two variables z^2 > y **Global** - constraints with 3+ variables can be **reduced to multiple binary/unary constraints Binarization of constraints** Convert n-ary constraints into **unary/binary** ones Any arbitrary n-ary constraint can be converted to equivalent unary constraint <u>Example 1: Individual variables and their domains</u> **Variables: X = {1,2} Y = {3,4} Z = {5,6} Encapsulated U = {(1,3,5), (1,3,6), (1,4,5), (1,4,6), (2,3,5), (2,3,6), (2,4,5), (2,4,6)}** Introduce new encapsulated variable that is a Cartesian product of the domains of the individual variables! Cartesian product U = X x Y x Z This new encapsulated variable U contains all the unique combinations (8) <u>Example 2: Original constraint and variables</u> **Constraint: X + Y = Z Variables: X = {1,2} Y = {3,4} Z = {5,6} Encapsulated U = {(1,3,5), (1,3,6), (1,4,5), (1,4,6), (2,3,5), (2,3,6), (2,4,5), (2,4,6)}; Encapsulated U (reduced)= {(1,4,5), (2,3,5), (2,4,6)};** Create the encapsulated variable like before but perform a reduction based on the constraint! It will reduce the domain of U **CSP Example: House Puzzle** Row of houses each one has: color, Person with nationality, favorite candy, Favorite drink, Pet, All attributes distinct, Associate variables with a location, e.g. milk - 3 for house #3, cat - 4 for house 4, **Implementing a CSP problem: Representation Variables** - simple list **Values** - mapping from variables to value lists e.g. python dictionary **Neighbors** - mapping from variables to list of other variables that participate in constraints **Binary constraints** - explicit value pairs, functions that return a boolean value **General Strategies for Solving CSP Local consistency:** reduce set of possible values through constraint enforcement and propagation; Perform search on **remaining possible states Node Consistency** A variable is **node-consistent** if all values satisfy all unary constraints Other unary conditions could further restrict the domain **Arc Consistency Arc-consistent Variable** - all binary constraints are satisfied for the variable **Network** - if all variables in CSP are arc-consistent Arc consistency only helps when some combinations of values preclude others **3 Outcomes** (when all arcs arc consistent) One domain is empty => **no solution** Each domain has a single value => **unique solution** Some domains have more than one value => **may or may not be a solution** In this case, arc consistency isn't enough to solve **Global Constraints** Consider an "all different" constraint; Each variable has to have a distinct value; Assuming M variables and N distinct values; What happens when M > N? Impossible to solve, have to repeat a value; Extending this idea: Find variables constrained to single value Remove these variables and their values from all variables Repeat until no variable is constrained to a single value Constraints cannot be satisfied if: Variable remains with empty domain, There are more variables than remaining values **Resource contains ("at most")** Consistency checks - minimum values of domains satisfy constraint? Domain restriction - are the largest values consistent with the minimum ones? **Range Bounds** Impractical to store large integer sets Ranges can be used [min, max] instead **Bounds propagation** can be used to restrict domains according to constraints; **Once all constraints propagated, search for solution Naive Search** Action picks a variable and a value.N variables, domain size D -> N x D possible search nodes; Search on next variables; **Backtrack** when search fails; **Problems** N variables with domain of size D; NxD choices for first, (N-1)D for second -> N!d^n; Only d^n possible assignments **Modified Search** CSPs are **commutative** => order of variable selection does not affect correctness; Each level of search handles specific variable; Levels have d choices, leaving us with d^n leaves **Backtracking Search Select-Unassigned-Variable** Fail-first strategies: **Minimum remaining value heuristic:** Select the most **constrained** value; the one with the smallest domain Rationale: probably most likely variable to **fail Degree heuristic:** Use the variable with the highest number of constraints on other unassigned variables **Order-Domain-Values** Order of values within a domain **may or may not make a difference** If goal is to produce all solutions or if there are no solution => order has no consequence In other cases, we use a fail-last strategy and pick the value that reduces neighbors' domains as little as possible **Why fail-first for variable selection and fail-last for value selection?** For variables, order does not matter. So we want to eliminate as many possible. For values, order can matter so if we eliminate a value, we might hurt ourselves later when we go to check it out. **Inference in Search => Forward Checking** Check arc consistency with neighboring variables Not needed if arc-consistency was performed prior to search **Maintaining Arc Consistency (MAC)** Algorithm that propagates constraints beyond the node; AC3 algorithm with modified initial queue **Typical AC3** - all constraints **MAC** - constraints between selected variable and its neighbors **Back-Jumping** Maintain a **conflict set** for each variable X: <u>A set of assignments that restricted values in X's domain</u> When conflict occurs, backtrack to last conflict added **Back-Jumping Implementation:** On forward checks of X assigned to x, When X deletes a value form Y's domain, add X=x to Y's conflict set If Y is emptied, add Y's conflict set to X's and backjump; Easy to implement, build conflict set during forward check; However, what we prune is redundant to what we'd prune from forward checking or MAC searches; **Constraint-Learning and No-Goods;** Minimal set of assignments that caused problem - **no-goods** Avoid running into problem by adding new constraint (or checking no-good cache) **Local Search CSPs** Alternative to what we have seen so far; Assign **everything** at once; Search changes **one variable** at a time **Min-Conflicts Local Search;** Pretty effective for many problems, e.g. million queens problem can be solved in about 50 steps; Essentially **greedy search**, consequently: **Local extrema; Can plateau;** Techniques discussed for **hill climbing** can be applied (e.g. simulated annealing, plateau search) **Structure of CSP Problems;** Improve search by **exploiting structure;** Independent **subproblem**s - solve separately; <u>**Ch 18 - Learning Agents**</u> can learn to improve **Inference from percepts** Information about world evolution as the result of **changing world** or **action** Utility estimators; Action choices either update condition-action maps or involve goal modification to maximize utility; What we want to learn **Mapping function** Inputs are factored representation e.g. vector of values **Outputs are** Discrete (e.g. categorical), Continuous **Type of Learning** Inductive - learn map between input / output pairs Deductive - creating rules that are logically entailed Learners vary based on **feedback Unsupervised learning** No explicit feedback; Goal is to cluster similar things; **Reinforcement learning** Learner given rewards/punishments for actions Ex: animal training w rewards **Hybrids** are possible such as semi-supervised learning where a small set of labeled data accompanies large set of unlabeled data Caveat about labeled data sets Labels referred to as "ground truth" **Why be cautious with "ground truth"?** Error can be systematic / important? **How to choose amongst functions?** Hypothesis spaces **Decision Tree Learner** Answers a series of **questions** to arrive at a solution For now, restrict discussion to Questions that have **categorical** (discrete answers) **Binary** classification decisions **Decision trees** -> good for justifying decision because you can see the decisions (white box) Constructing a tree from examples **Quantity of information** Amount of **surprise** that one sees when observing an event If an event is **rare,** we can derive a large quantity of **information** (measured in **bits**) from it **Entropy -** defined as expected amount of information (average amount of surprise) and is usually denoted by the symbol H **Tree Questions** Eq's have binary response Suppose goal: separate mammals from birds Question: does it fly? **Tree Questions Entropy** Entropy is E[I(P(X))] = E[-log2P(X)] For binary categories, define short hand: Q = p / (p + n) => the positive rate - 1 - Q = Q / (p + n) => the negative rate - B(q) = E[IP(X))] = -qlog2q - (1-q)log2(1-q) Non-binary entropy calc: $-\sum_{i \in num\_classes} P(v_i)\log_2(v_i) == \sum_{i \in num\_classes} P(v_i)\log_2(1/P(v_i))$ **Information Gain** Goal: **reduce** the amount of information needed to represent the problem We can represent the remaining entropy after dividing **data into d groups** with question **A** as follows: **Remainder(A)** = some questions from slide Expected value of entropy for all the categories And information gain as: **Gain(A) = B(p/(p+n)) - Remainder(A) B = Binary Entropy Features and Overlearning** Useless features are not good for prediction, but a learner may pick up on random patterns in the training data and incorporate into rules **Generalization and overfitting** Learning random patterns that don't affect function is called **overfitting** For each leaf node, ask ourselves if **good information gain** If node informative, keep it, otherwise discard How do we know if our decisions were any good? Goal was to **separate into positive and negative classes as well as possible** Can we devise a statistic that lets us know if our observed split is statistically significant from the expected ratio? => **Chi squared Chi Squared Test** Supposed decision tree splits a node into V categories If node does not add any new info, we would exotic # of class examples in each split roughly according to same proportion of examples Let's restrict an example to We can look at how much our categories **differ** from what would be expected if the proportion of categories did not change When value is **small**, we are close to the original distribution Test stat has distribution related to (# of categories - 1) => **degrees of freedom** Cumulative density function (CDF) -> integrate P(x dof) up to delta The chi-squared test is used to determine **whether there is a significant difference** between the **expected frequencies** and the **observed frequencies** in one or more categories. **Decision Trees w/ Continuous/integer-valued attributes** Don't create **infinite branches** Select split point Sort values Keep running total of number of +/- examples for each point in sorted list and pick separating point that give best separation

Decision Tree Summary Relatively **straightforward** learners Recursively portion the feature space into **hyperplanes** Sensitive to **overtraining**, have methods to **prune** Easy for **humans** to understand **Do I have a good hypothesis function?** We cross-validate the learner on a separate validation set Problem: don't exploit all data **K-fold cross validation** Extreme case - leave-one-out cross validation (aka jackknife) **Model selection** More complex models (e.g. more nodes in a decision tree) learn the training data better, but are they really better? Look at validation error **Loss** Loss functions are form o**f utility function that provide cost for misclassification** Could be good to find whale and noted to misclassify nonwhale Some learners attempt to minimize loss **Common loss functions Generalization Loss** What is our loss when we use a novel data set e? The expected loss requires the distribution (X,Y) which we probably do not have: But we can estimate it empirically on a finite set of examples E of N samples: **Unreliability:** f may not be in H Variance: learners return different f's for different training sets **Noise:** F may be noisy (e.g. stochastic component - different y's for the same x) The training samples may have mis-measured attributes or incorrect labels Might not have measured important attributes **Complexity:** learner may not achieve a global minimum **Regularization of Regression Linear Classification** Regression lines can be used to classify examples We look for a **linear separating line** (hyperplane when data is in R3 or higher) Suppose we pick a **vector w** perpendicular to the decision boundary Consider the **dot product** between w and an arbitrary point Once we have w, we can **classify** by taking the dot product and looking at the sign **How do we choose w?** Variant of the **gradient descent** rule used for regression is applicable Recall **regression rule Perceptron Learning Rule Iterate** through data (1 iteration = 1 epoch) and repeat until no errors Convergence may be slow, but **guaranteed** for linearly separable data Most datasets **not** linearly separable When non linearly separable examples presented in random order, we will converge to a stable classifier if alpha decays **linearly** with epoch number **Learning with Logistic Regression** Remember update rule was: wi = wi - a(t/tw)loss(w) (t = theta) When the hard threshold is replaced with logistic regression: Easier if we use labels [0,1] instead of [-1, 1] as this range covered by function Derivative of loss needs to be recomputed; wi = wi - a(y - hw(x))hw(x)(1-hw(x))xi This results in smoother + more predictable learning curve **Connectionist Networks (Artificial Neural Networks)** Activation functions for perceptrons are **nonlinear:** Hard threshold Logistic regression (frequently called sigmoid function) **Linking perceptrons together** provides complex function modeling capability **An Intuitive View of Neural Nets** Suppose we combine two perceptrons whose output functions are reversed This could be used to model a ridge in output space **Learning in a Neural Network** Similar to the regression problem, for output a and desired output y, we can find the loss gradient for each output node Use perceptron learning rule for the sum of the gradients at the output layer **Back-Propagation** What should the targets be for the previous input layer? **Back-propagating error (overview)** Error of the kth output: Errk = yk - ak We can compute gradient for any input node and apply **regression rule** This gives new set of **weights** for output node After applying update to the output layer, there still exists **loss** We assign a portion of the loss to each of the input nodes based on their **weight** This contribution is computed for each node of current layer Look at sum of losses attributable to each node in previous layer -> sum of these provides us with loss to minimize **Neural Net Summary Supervised Learner** Training labels either High value for class (n classes -> n output nodes) Encoding of class information Iterative training typically using a gradient descent algorithm (e.g. back propagation **Classification** Present features to input nodes Interpret output nodes for category **Disadvantages** Frequently hard to interpret Many parameters require large data sets Bad w imbalanced examples Slow to train Overfits easily regularization important **Advantages** Flexible, nonlinear learner Deep architectures very powerful **Non-Parametric Models** Neural nets and decision trees have models with parameters Decision node parameters: **attribute and cut-point/categories for sub-trees** Neural nets: **weights and connections Non-Parametric models:** Cannot be characterized by **bounded set** of parameters Simplest case: look at every example and use it to classify novel examples Called **instance** or **memory-based learning Nearest Neighbors Models** Use a distance metric to find the k closest neighbors, e.g. for continuous attributes Use the plurality of labels that are the k closest **Good** Simplicity Effective technique for low-dimensional data **Bad** - searching is expensive with large training sets, but we can mitigate for this: Trees - similar to decision tree (split on value, may at times need to search both sides) **Locally sensitive hash tables** Hash functions Set of projections on to lines Line projections discretized into buckets Can be much more effective than tree approach **Ugly** N points uniformly distribution in R^D unit hypercube To capture r = 0.01, what edge length would we need in a random sample? Samples are randomly distributed and total volume is 1, so we need a volume of r (0.01) **Support Vector Machines** A **margin** is the distance to the closest examples on either side of a hyperplane SVM approaches attempt to **maximize the margin** Can only separate linear problems, but **kernel function** can project the data into a higher dimensional space where perhaps the data can be better separated Maximal margins computed as functions of training examples Summary **SVM - non parametric technique** In practice, **only small subset of training examples, the support vectors are required Error in learning comes from two sources: bias and variance Bias** - large when learners make **consistently** incorrect predictions **Variance** - large when **different training sets** result in different predictions **Ensemble Learning Ensemble learners** are collections of weak learners that are combined to form robust classifier **Weak learner** - simple learning algorithm that is likely to have high bias (e.g. **single node/stump of decision tree** Ensemble learners typically use collections of weak classifiers to **reduce both bias and variance** <u>Chapter 7 - Logical Agents</u> **Logical Agents** Two key components: **Representation** of the world **Background information Percepts; Ability to reason** : derive new information based on inference **Knowledge Base (KB) Sentence** - statement about the agent's world Based on **sensors** Based on **background knowledge** (or possible learning) **Satisfaction and Entailment** Suppose a is true in model M, then we state: M satisfies a or equivalent Or: M is a model of a M(a) means all models that satisfy a Reason - entailment **Examples of Entailment** House is cornflower blue entails house is a shade of blue X = 0 entails xy = 0 Second set must be bigger and contain it (basically subset) **Inference Algorithms** Are sound if inference only devices entailed sentences If our algorithm entailed KB |= a2, we might fall into a pit! Are complete if they can service any sentence that is entailed. Becomes an issue when the left-hand side is infinite If KB is well grounded, our entailments should follow in the real world **Sentence Construction: Propositional Logic** Propositional logic is for the most part the logic you learned to program with Sentence -> AtomicSentence | Complex Sentence; AtomicSentence -> true | false | literal; ComplexSentence -> (Sentence) | [Sentence] | ~ Sentence => (negation) | Sentence ^ Sentence (conjunction) | Sentence v Sentence (disjunction) | Sentence => Sentence (implication) | Sentence <=> Sentence (biconditional) **Operator priority is shown by order Sentence Semantics** Sentences reduced to true or false with respect to specific model In the Wumpus cave we might denote presence or absence of pit by literals indexed by location: Px,y Example: P1,2, P2,2, and P3,1 that have true/false values in any given model Models must specify values for each proposition To resolve sentence: apply logical connectives to truth values **Knowledge Base Rules** In the Wumpus save, Denote pit & rumpus presence/absence y $P_{i,j}$ and $W_{i,j}$ There is a breeze if and only if there is a neighboring pit: $B_{2,2}$ <=> $P_{2,1}$ v $P_{2,3}$ V $P_{1,2}$ v $P_{3,2}$ $B_{1,1}$ <=> $P_{1,2}$ v $P_{2,1}$ There is stench if the rumpus is in the next cavern $S_{2,2}$ <=> $W_{2,1}$ v $W_{2,3}$ V $W_{1,2}$ v $W_{3,2}$ $S_{1,1}$ <=> $W_{1,2}$ v $W_{2,1}$ **Percepts** There is no pit at 1,1 ~$P_{1,1}$ There is no breeze at 1,1 ~$B_{1,1}$ There is a breeze at 2,1 $B_{2,1}$ **Simple Entailment Algorithm** `def TruthTable-Entails(KB, a): symbols = proposition-symbols in KB and a return TT-Check-All(KB, a, symbols, {}) def TT-Check-All(KB, a, symbols, model): if empty(symbols): if pl-true(kb, model): # Does the KB entail the model? return pl-true(a, model) # Is entail the model? else return True # return true when KB does not hold else: # recursively enumerate the models (s, others) = (first(symbols), rest(symbols)) return TT-Check-ALL(KB, a, rest, model U {s=True}) and TT-Check-ALL(KB, a, rest, model U {s=False})` **Summary of Model Checking** Recursively generates all possible combinations of truth values for every symbol. Checks if the knowledge base is a subset of a specific symbol value assignment If so, returns if sentence a hold as well Otherwise returns true as we don't care about whether as holds outside the KB (implies) Can we prove things without enumerating everything? **Concepts Logical equivalence** a = B if they are true in the same set of models Alternative definition: Equivalent if they entail one another a =| B <=> b |= a **Validity** Sentences are **valid** (called **tautologies**) if they are true in all models e.g. (a^b)v~aV~a **Deduction theorem** For arbitrary sentences a and B, a |= B iff (a => B) is valid So if we can show (a <=> B) is a tautology, we know a |= B **Satisfiability** There exists **some model** such that sentence is true Sometimes easier to show something is valid by **showing its contradiction is not satisfiable**: a is valid iff ~a is not satisfiable if no model satisfies ~a, then a must be true which leads to: a |= B iff (a ^ ~B) is not satisfiable Remember a |= B iff a => === ~a v B.

$$
\begin{aligned}
(\alpha \wedge \beta) &\equiv (\beta \wedge \alpha) && \text{commutativity of } \wedge \\
(\alpha \vee \beta) &\equiv (\beta \vee \alpha) && \text{commutativity of } \vee \\
((\alpha \wedge \beta) \wedge \gamma) &\equiv (\alpha \wedge (\beta \wedge \gamma)) && \text{associativity of } \wedge \\
((\alpha \vee \beta) \vee \gamma) &\equiv (\alpha \vee (\beta \vee \gamma)) && \text{associativity of } \vee \\
\neg(\neg\alpha) &\equiv \alpha && \text{double-negation elimination} \\
(\alpha \Rightarrow \beta) &\equiv (\neg\beta \Rightarrow \neg\alpha) && \text{contraposition} \\
(\alpha \Rightarrow \beta) &\equiv (\neg\alpha \vee \beta) && \text{implication elimination} \\
(\alpha \Leftrightarrow \beta) &\equiv ((\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)) && \text{biconditional elimination} \\
\neg(\alpha \wedge \beta) &\equiv (\neg\alpha \vee \neg\beta) && \text{De Morgan} \\
\neg(\alpha \vee \beta) &\equiv (\neg\alpha \wedge \neg\beta) && \text{De Morgan} \\
(\alpha \wedge (\beta \vee \gamma)) &\equiv ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma)) && \text{distributivity of } \wedge \text{ over } \vee \\
(\alpha \vee (\beta \wedge \gamma)) &\equiv ((\alpha \vee \beta) \wedge (\alpha \vee \gamma)) && \text{distributivity of } \vee \text{ over } \wedge
\end{aligned}
$$

(p⇒q) read "if p is true then I am claiming that q is true, otherwise I'm not making a claim") so if not p then q is always

(p<=>q) read "is true iff p and q are both true or both false in Model of truth tables(what model is set as)" or when both p⇒q & q⇒p r true

conjunctive normal form: combine all sentences through ands but the each complex sentence can contain only ors, (aV~bVc) ^ (~dVe); (a V b) ^ c; a v b; a ; Resolution rule: ((sentence1, sentence2)/(inference (claim from these 2 sentences))

Overall goal: if trying to prove kb |= w22 prove that all boxes around w22 that there is a wumpus there, crossing off possibilities as we go