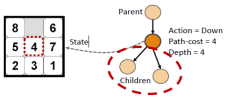
Standard Search problem

State space: Number tiles in each cell position
Initial state: [8,-,6,5,4,7,2,3,1]
Actions: Move tile {Left, Right, Up, Down}
Transition Model: Update tiles in current & target cell pos
Path cost: Number of moves
Goal test: Compare to positions in goal state

State: rep. of a **physical configuration**
Node: data struct, **part of a search tree**
- Comprising state, parent-node, child-node(s), action, path-cost, depth unlike a state.

Solution: Seq of act from init to goal state



Expand function **creates new nodes** (& their various fields) using the Act & Transition Model to create the corresponding states

Search strategy: picking the **order of node expansion**

b: Max branching factor of the search tree
d: depth of the least-cost solution
m: max depth of the state-space (inf possible)

Tree Search:
- Offline, simulated exploration of state-space by expanding states via generation of successors of already-explored states
- However repeated states results in redundant paths that can cause a tractable problem to become intractable (inf loop, m = inf)

Graph Search:
- Modified tree search to keep track of previously visited states (to not revisit) but a potentially large number of states to track)

Types of Search:

a) Uninformed Search
No additional information about states beyond that in the problem definition (blind search)

b) Informed Search
Uses problem-specific knowledge beyond the definition of the problem itself

c) Adversarial Search
Used in a multi-agent environment where the agent needs to consider the actions of other agents and how they affect its own performance.

BFS (FIFO, Graph): Init: A
Expand & pop A: AB, AC
Expand & pop B: AC, ABD, ABE
Expand & pop C: ABD, ABE, ACF
Expand & pop D: ABE, ACF, ABDG, E visited
Expand & pop E: ACF, ABDG, ABEH, F visited
Expand & pop F: ABDG, ABEH, H visited
Expand & pop G: ABEH, ABDGX, H visited
Expand & pop G: ABDGX, H visited
Expand & pop X: X is goal state, return ABDGX

Constraint Satisfaction Problem:
State: Variables, X_i with values from doman, D_i
Goal Test: Set of constraint, C_i specifying allowable combinations of values for subsets of var

(Nodes) Finite set of variables, $X = \{X_1, ..., X_n\}$
Non-empty domain D of k possible values for each variable, $D_i = \{v_1, ..., v_k\}$
(Edges) Finite set of constraints, $C = \{C_1, ..., C_m\}$
where they limit the values variables can take

Complete assignment: every var assigned a val
Consistent assignment meets all constraint
CSP solution = complete & consistent for all var

Formal representation language that can be used to formalize many problems types

Able to use ***general-purpose solver***, which are generally more efficient than standard search.
- Constraints allow us to focus the search to valid branches, invalid branches removed
- Non-trivial to do this for standard search (need manual selection of actions)

Varieties of CSP:
o Discrete variables:
- **Finite domains**: $O(d^n)$ complete assignments for n variables, domain size d (e.g. map-colouring, scheduling with time limits)
- **Infinite domains**: Int, str, etc. (e.g. job scheduling (e.g. StartJob1 + 5 < StartJob3)

o Conti. variables: (e.g. start/end times of an obs)

o Unary/Binary/Higher-order Constraints:
# of variables involved in constraint

o Preference (soft) Constraints:
(e.g. red is better than green represented by some cost of each var, aka Constr. Opti Problem)

Uninformed Searches

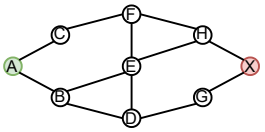
Breadth First Search:
Expand shallowest unexpanded node, implement using (FIFO) Queue
- **Completeness:** Yes (if b is finite)
- **Optimality:** Yes (Not optimal in general)
- **Time complexity:** $O\left(b^d\right)$
- **Space:** Same (keeps every node in mem)

Recall BFS checks level by level, if the step cost != 1/ not uniform, then we might miss out certain nodes with a cheaper step cost

Uniform Cost Search: BFS w g(n)=1
Expand unexpanded node with lowest path cost g(n) implement using Queue ordered by g(n). If visited node, only replace if cost is lower
- **Completeness:** Yes (if step cost some positive constant else lead to loop pattern)
- **Optimality:** Yes
- **Time/Space complexity:** $O\left(b^{\frac{C^\star}{\varepsilon}}\right)$

$C^\star$ is the cost of opt solution
Goal test after popping only, since there might be some shorter path to that node. Only do goal test when popping that node.

Depth First Search:
Expand deepest unexpanded node, implement using (LIFO) Queue
- **Completeness:** No (if m inf, keep going down inf path)
- **Optimality:** No (might be searching down non-optimal path)
- **Time complexity:** $O(b^m)$ terrible if m>>d, as need to back track all the way back up
- **Space:** `O(bm)` linear , store m depth, and at each level, keep track of b child nodes



DFS (LIFO, Graph, insert lowest alphabetical order 1st): Init: A
Expand & pop A: AB, AC
Expand & pop C: AB, ACF
Expand & pop F: AB, ACFE, ACFH
Expand & pop H: AB, ACFE, ACFHG, ACFHX
Expand & pop X: X is goal state, ret ACFHX

DLS (with l = 2): Init: A
Expand & pop A: AB, AC
Expand & pop C: AB, ACF
Expand & pop F: AB, depth limit = 2
Expand & pop B: ABD, ABE
Expand & pop E: ABD, depth limit = 2
Expand & pop D: Empty, depth limit = 2
Frontier empty, ret no solution

Iterative-Deepening Search:
Use increasing DLS to find best depth limit
- **Completeness:** Yes
- **Optimality:** Yes
- **Time complexity:** $O\left(b^d\right)$
- **Space:** $O(bd)$

Best of BFS (optimality) + DFS (space complex). Not wasteful to keep generating states with each increasing DLS since most of the work (node expansion) happens at lower depths)

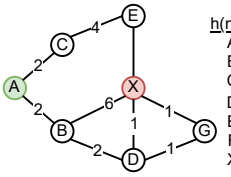
Depth-Limited Search:
DFS with predetermined depth limit l
- **Completeness:** No (l<d)
- **Optimality:** Chance of No (if l>d), might find a solution but is not the least-cost
- **Time complexity:** $O\left(b^l\right)$
- **Space:** `O(bl)`

Solves the infinite problem of DFS

Greedy Best-First Search:
Expand node with lowest h(n), implement using Queue ordered by h(n)
- **Completeness:** No (can get stuck in loop unless we keep track of visited states)
- **Optimality:** No
- **Time complexity:** $O(b^m)$
- **Space:** Same (keeps every node in memory)

UCS complete & optimal but may "waste searches" if search in the wrong direction. G-BFS searches generally in the "right" direction but not complete & optimal since a shorter path from the initial state does not imply we are nearer to the goal state.

A-Star Search:
Expand node that has incurred least cost & is nearest to goal state, implement using Queue ordered by eval. fn, $f(n) = g(n) + h(n)$
- **Completeness:** Yes (no need to keep track of explored steps due to path cost g(n))
- **Optimality:** Yes (if adm/consistent h(n))
- **Time/Space complexity:** UCS



h(n) to X
A: 7
B: 4
C: 5
D: 3
E: 4
F: 2
X: 0

A* becomes UCS if heuristic cost of every node is 0. So, UCS is a special case of A* algorithm

---

Informed Searches

$g(n)$ ignores direction of the search, instead we use $h(n)$ heuristic fn: est dist from node n to goal state (how close a state n is to the goal state)

$h(n)$ **admissible** if $h(n) \leq h^\star(n)$
$h^\star(n)$ = true cost from node n to goal state (e.g. straight line distance $\leq$ the actual road distance)

$h(n)$ **consistent** if $h(n) \leq h(n\prime) + c(n, a, n\prime)$
$c(n, a, n\prime)$ = cost of getting from node n to n'

$h_1(n)$ **dominates** $h_2(n)$ if $h_1(n) \geq h_2(n)$

Provided that both $h(n)$ are admissible More dominant $h(n)$ potentially explores lesser branches, hence better

Admissible heuristics can be derived from the exact solution cost of a relaxed
version of the problem

e.g. Relaxed: Tiles can be directly moved to any square. $h_1(n)$: # of misplaced tiles **OR** Relaxed: Tiles can be moved to any adjacent square. $h_2(n)$: total Manhattan distance

Admissible heuristics can be derived from the solution cost of a sub-problem
of the problem (e.g. Solve 4 tiles first)

Greedy-BFS (Graph): Init: A(7)
Expand & pop A: AB(4), AC(5)
Expand & pop B: ABX(0), ABD(3), AC(5)
Expand & pop X: X is goal state, ret ABX

A⋆: Init: A(0+7=7)
Expand & pop A: AB(2+4=6), AC(2+5=7)
Expand & pop B: ABD(2+2+3=7), AC(7), ABX(2+6+0=8)
Expand & pop D: ABDX(2+2+1+0=5), ABDF(2+2+1+2=7), AC(7), ABX(8)
Expand & pop X: X is goal state, ret ABDX

Early Detection:
1) Fwd Checking Algo
- Keep track of remaining lv for unassigned var, terminate search when any var has no lv
- Propagates info from assigned to unassign var only, only provide early detection for some failures. Need to repeatedly enforce constraints locally using constraint propagation.

2) AC-3 Algo $O\left(n^2d^3\right)$ order arcs no matter
- Start with queue of all directed arcs
- **RemoveInconsistent(X,Y) :**
Check for consistency (X-> Y is consistent iff for every val of X there is some allowed y)
- If removed any, add the neighbours (into X) into queue
- Continue till queue is empty

$O\left(n^2\right)$: Need to check all (potentially) n^2 edges (directional)
$O\left(d^2\right)$: For every edge, compare both domains
$O(d)$: Each var change, repropagate to its neighbour at most max d times

AC3: V1 -> V2: D1 = RB, D2 = RG
V2 -> V1: D2 = RG, D1 = RB
V2 -> V3: **D2** = R~~G~~, D3 = G
V3 -> V2: D3 = G, D2 = R
V1 -> V3: D1 = RB, D3 = G
V3 -> V1: D3 = G, D1 = RB
(+) **V1** -> **V2**: **D1** = ~~R~~B, D2 = R
(+) V3 -> **V1**: D3 = G, D1 = B
(+) V2 -> **V1**: D2 = R, D1 = B
Queue empty, all arc consistent, ret ...

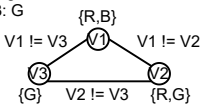
Pure Backtracking:
V1: R
V2: R (Inconsist. assign, bt)
V2: G
V3: G (Inconsist. assign., backtrack)
V1: B (V2 lv none left)
V2: R
V3: G, all var assigned, ret
V1: B, V2: R, V3: G

BT with FC: V1: R (FC, remove any inconsist in remaining var of neighbours)
V2: G (FC)
V3: Empty, terminate
V1: B
V2: R
V3: G all var assigned, ret
V1: B, V2: R, V3: G

Backtracking Search:
(DFS with single-var assignment)
CSP var assignments are **commutative** (order not impt)

Hence only consider assignment to 1 var at each level/depth, branch factor at all levels d, d^n leaves

CSP: $O(d^n)$ If subproblem have c var out of n total, worst case is $O\left(\frac{n}{c} \cdot d^c\right)$



Variable Assignment Order:
1) Minimum remaining values
- Choose var with fewest legal values left (most constraint var, most likely to fail)

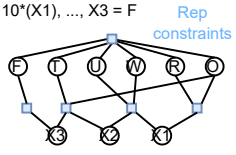
If same MRV, **Degree Heuristic**:
- Choose var with the most constraint on remaining var

2) Least constraining value:
- Choose one that rules out the fewest values in the remaining var

---

E.g. of CSP: Variables: F, T, U, ..., X1, X2, X3
Domains: {0, 1, 2, ..., 9}
Constraints: all_diff(F, T, ..., O),
O + O = R + 10*(X1), ..., X3 = F

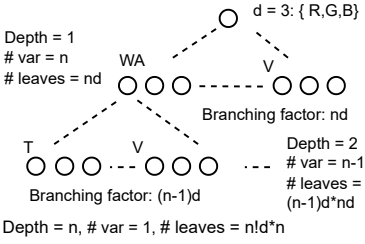
X3 X2 X1
T W O
+ T W O
F O U R



Variables: WA, NT, Q, NSW, V, SA, T
Domains: $D_i = \{R, G, B\}$
Constraints: Adj regions diff colours, ie. WA!=NT
CSP Solution: {WA=R, NT =G, ..., T=G}

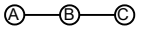
CSP as Std Search:
Initial State: {}
Successor fn: Assign val to unassign var
Path cost: Constant cost for every step
Goal test: Current assign complete & consist.

Seq of actions/path not impt, only final goal state. Also solution will appear at depth n for n var, potentially $n!d^n$ leaves in the search tree. Can reduce using backtracking search

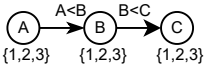


Tree Structured CSP:
If constraint graph has no loop: $O\left(nd^2\right)$ time since tree with n nodes has at most n edges, compare 2 nodes which each side at most d values

Domain = {1,2,3}
Constraints = A<B, B<C



Order var from root (choose one) to leaves st every node's parent precedes it in the ordering



From j = n to 2, apply $RemoveInconsistent\left(parent\left(X_j, X_j\right)\right)$

RI(B,**C**): B{1,2,~~3~~}, C{1,2,3}
RI(A,**B**): A{1,~~2,3~~}, B{1,2}

From j = 1 ton, assign X_j consistently with $Parent\left(X_j\right)$

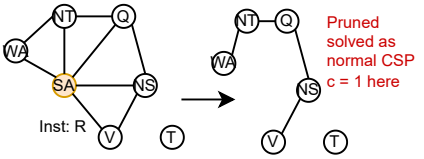
A{**1**}
B{1,**2**}, only 2 is consistent
C{1,2,**3**}, only 3 is consistent

Nearly Tree Structured CSP:
Conditioning: Instantiate a var, prune its neighbours' domain

Cutset conditioning: instantiate (in all ways) a set of variables st the remaining const. graph is a tree

Cutset size c implies $O\left(d^c \cdot (n-c)d^2\right)$, very fast for small c

Rest all pruned to G,B, solve as tree-CSP



Game Types	Deterministic	Chance
Perfect Info (Fully Obs)	Chess, Checker, Go	Backgammon
Imperfect Info (Partially Obs)	Battleship	Bridge, Poker, Scrabble

Supervised (Inductive) Learning: Train + labels (desired output) e.g. classification, regression, obj detection, semantic segmentation
Unsupervised: Train (no desired output) e.g. clust, dim reduction
Semi-supervised: Train +few desired output
Self-supervised: Pretext task (e.g. predict if rotated image), learn the latent features (output labels) intrinsically from data
RL: Rewards from seq actions

Evaluation: Acc (must be class specific in case of imbalanced data)
E.g. If only 30% 1 & predict all 0, we have 70% acc! totally biased

Actual \ Predicted	+ve	-ve
+ve	TP	FN (Type II Error)
-ve	FP (Type I Error)	TN

Recall/ Sensitivity: TP/(TP+FN)
Specificity: TN/(TN+FP)
Precision: TP/(TP+FP)
-ve Pred Val: TN/(TN+FN)
Acc: (TP+TN)/(All)
F1: (Precision * Recall)/(Precision + Recall)

MSE (L2 Loss) for regression:
$$MSE = \sum_{i=1}^{n} (Y_i - \hat{Y}_i)^2$$

Receiver Operating Curve (ROC):
TPR vs FPR (plot against all threshold, AUC = 1 best)

Information: (Generally) Measured in bits ($\log_2$): Amt of "surprise" arising from a given (stochastic) event E: $I(E) = -\log(P(E))$

E.g. Three boys: Tom (20%), Bob (1%), Sam (79%)
Bob = $-\log_2(0.01) = 6.6$, high surprise vs
Sam = $-\log_2(0.79) = 0.34$, low surprise

Entropy: Avg lvl of information inherent in possible outcomes:
Higher Entropy signifies more uncertainty in what action an agent would choose: $H(X) = -\sum_i P_X(x_i) \log_b P_X(x_i)$

$$Entropy = -(0.20\log_2(0.20) + ... + 0.79\log_2(0.79)) = 0.26$$

KL-Divergence (relative entropy):
Expression of Surprise, diff btw 2 prob distribution P & Q:
Under the assumption P & Q are not close, KL divergence high (surprise).
If close then KL divergence low

Likelihood ratio (n indep samples): $LR = \prod_{i=0}^{n} \frac{p(x)}{q(x)}$
(how much more likely samples came from P dist than Q dist)

Log likelihood ratio: $LLR = \sum_{i=0}^{n} \log\left(\frac{p(x)}{q(x)}\right)$

Expected LLR:
(Avg, how much each sample gives evidence of P(x) over Q(x))
$$D_{KL}(P \parallel Q) = \sum_{i=0}^{n} p(x_i) \log\left(\frac{p(x_i)}{q(x_i)}\right)$$
$$D_{KL}(P \parallel Q) = \sum_{i=0}^{n} p(x_i) \log(p(x_i)) - \sum_{i=0}^{n} p(x_i) \log(q(x_i))$$

Entropy (Info content of P) - IC of Q weighted by P
If P is the "true" distribution, then the KL divergence is the amount of information "lost" when expressing it via Q.

Cross-Entropy (min unlike max likelihood):
Cross entropy is, at its core, a way of measuring the "distance" between two probability distributions P and Q. (Entropy on its own is just a measure of a single probability distribution)

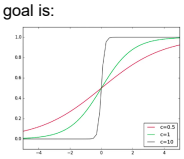
CE = Combination of the entropy of the "true" distribution P and the KL divergence between P and Q:
$$H(p,q) = H(p) + D_{KL}(p \parallel q) = -\sum_{i=0}^{n} p(x_i) \log(q(x_i))$$

E.g. Predict one-hot vector: True {0, 0, 1, 0, 0}
Output (of softmax layer): Predicted {0.01, 0.02, 0.75, 0.05}
Predicted class (argmax): 2 (zero index)

$$-(0 \cdot \log(0.01) + 0 \cdot \log(0.02) + 1 \cdot \log(0.75) + 0 \cdot \log(0.05))$$

Cross-entropy:	**KL-Divergence:**
Avg # of total bits needed to encode data from a source with dist p when we use model q	Avg # of extra bits needed to encode the data, when using dist q instead of the true dist p.

Logistic Regression:
Given an input space X, the goal is to predict for every sample x ∈ X to which class it belongs. For 2 class classification, the goal is: predict output space $Y = \{0, 1\}$ or $\{-1, +1\}$



Linear: $f(x) = w.x + b$, unbounded space
Need to map $(-\infty, \infty)$ to $(0, 1)$ st 0 map to 0.5

Use logistic sigmoid fn $s(a)$:

$$s(a) = \frac{\exp(a)}{1+\exp(a)} = \frac{1}{\exp(-a)+1}\frac{\exp(a)}{\exp(a)} = \frac{1}{\exp(-a)+1}$$

Note if multi class (k) class, use Softmax:
$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad \text{for} \quad i = 1, ..., K \quad \text{and} \quad \mathbf{z} = (z_1, ..., z_K) \in \mathbb{R}^K$$

$h(x) = s(f_{w,b}(x)) \in (0, 1)$ aka prob(sample x class label $y = +1$)

Non-deterministic Game:
Adversarial search where the actions/transitions are non-deterministic
ExpectiMiniMax, similar to **MiniMax** but accounts for chance nodes

Cross-Entropy loss (bin class):

CE Loss: (1 sample: (x_i, y_i))
$-\overline{y}_i \log(p_1(x_i)) - (1-\overline{y}_i)\log(1-p_1(x_i))$ where $p_1(x_i) = 1 - p_2(x_i)$

Avg CE Loss: (dataset: n indep samples)
$$\frac{1}{n}\sum_i -\overline{y}_i \log(p_1(x_i)) - (1-\overline{y}_i)\log(1-p_1(x_i))$$

Opti problem (convex) to be solved is (finding best parameter w min CE loss):
$(w^\star, b^\star) = \text{argmin}_{w,b}$ (avg CE Loss)
where $p_1(x_i) = h(x_i) = \frac{1}{\exp(-w \cdot x - b + 1)} = s(w \cdot x_i)$

$$\nabla_w L = \nabla_w \left((-1) \cdot \sum_{i=1}^{n} y_i \log(s(w \cdot x_i)) + (1-y_i)\log(1 - s(w \cdot x_i)) \right) = \sum_{i=1}^{n} x_i (s(w \cdot x_i) - y_i) = \sum_{i=1}^{n} x_i (h(x_i) - y_i)$$

Optimizing Param:
1. Init weights randomly $\sim N(0, \sigma^2)$
2. Loop until convergence:
   a. Compute gradient: $\frac{\partial J(W_n)}{\partial W_n}$
   b. Update coeff: $W_{n+1} = W_n - \alpha \cdot \frac{\partial J(W_n)}{\partial W_n}$
3. Return weights, grad to 0

Hence gradient wrt w: (sum of points x_i weighted by the diff btw the predicted value $h(x_i) = s(w \cdot x_i)$ and true val y_i

Data Set: (No best split) **Training:** run your learning algo
Development: tune parameters **Test:** evaluate perf

Common Error:
1) Data mismatch (e.g. diff. Resolution img)
2) Bias and variance
   Bias: error rate on the training set
   Variance: how much worse test set error than train (e.g. 15% error (85% acc) on train, bias = 15%. But 16% error on test, variance = 1%)

Set Distribution:
If all same dist, then optimising for web imgs mostly here, which is diff from target dist.

100k imgs (from Web)	8k imgs (from Target dist)

Train (104k imgs)	Dev (2k imgs)	Test (2k imgs)

Instead make test & dev from the same dist such that optimizing to perform well on the target dist.

100k imgs (from Web)	4k imgs	4k imgs

Train (104k imgs)	Dev (2k imgs)	Test (2k imgs)

However, the training dist is now diff from the dev/test distribution: it will take longer & more effort to optimize the model. More importantly, cannot easily tell if the classifier error on the dev set relative to the error on the train set is a variance error, a data mismatch error, or a combination of both.

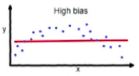
You can use a **bridge set (training data not used for training)** to trouble shoot where the error comes from.

100k imgs (from Web)	4k imgs	4k imgs

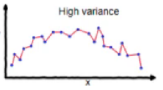
Train (102k)	Bridge (2k)	Dev (2k)	Test (2k)
Error 2%	3%	10%	

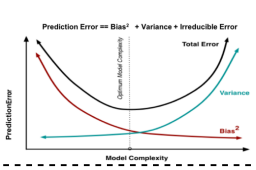
Error 8% (btw dev & train): 1% var + 7% data mismatch
Hence our classifier performed well on data from the same distribution but poorly when from a diff distribution, (data mismatch problem). Else if higher var & lower data mismatch error, implies overfitted to train set.

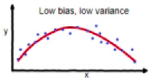
Rules of thumb:
High avoidable bias: (underfitting)
- Increase the size of your model
- Modify the input features based on error analysis
- Reduce or eliminate regularisation (reduces bias but increases variance)
- Modify model architecture



High variance: (overfitting)
- Add data to your training set (e.g. by data augmentation)
- Add regularisation, dropout
- Add early stopping
- Feature selection to decrease # of features (may increase bias)
- Decrease the model size (May increase bias)
- Modify input features based on error analysis
- Modify model architecture







Batch-norm layers:
- Faster training (allow higher lr), improves reg. & acc, reduces dependencies on init
- Ensures received input batch has mean 0 & std dev 1

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$; Parameters to be learned: γ, β
Output: $\{y_i = \text{BN}_{\gamma,\beta}(x_i)\}$

$\mu_\mathcal{B} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$ // mini-batch mean
$\sigma^2_\mathcal{B} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_\mathcal{B})^2$ // mini-batch variance
$\widehat{x}_i \leftarrow \frac{x_i - \mu_\mathcal{B}}{\sqrt{\sigma^2_\mathcal{B} + \epsilon}}$ // normalize
$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(x_i)$ // scale and shift

Minimax-Algo: (Deterministic + Fully obs game), Idea: choose move to position with highest minimax val
Completeness: Yes if tree is finite (exist terminal states)
Optimality: Yes against a rational (opti) opponent
Time Complexity: $O(b^m)$
Space Complexity: $O(bm)$ similar to DFS

Yet, for chess e.g. branching factor (possible moves), $b \approx 35$ & depth (number of plys) $m \approx 100$, exact solution is infeasible

MSE (of unseen test pt): bias & var tradeoff + inherent noise, ε:
$y = f(x) + \varepsilon$ where $\varepsilon \sim rv\left(0, \sigma_\varepsilon^2\right)$
$$\text{Total Error} = E_x\left[\text{bias}\left[\widehat{f}(x)\right]^2\right] + E_x\left[\text{var}\left(\widehat{f}(x)\right)\right] + \sigma_\varepsilon^2$$

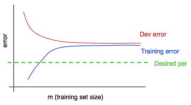
MSE: Avg squared diff of a pred $\widehat{f}(x)$ from its true val y:
$$MSE = E\left[\left(y - \widehat{f}(x)\right)^2\right]$$

Bias: Diff of avg val of pred (over diff realization of train data) to the true underlying function $f(x)$ for a given unseen test point x:
$$bias\left[\left(\widehat{f}(x)\right)\right] = E\left[\left(\widehat{f}(x)\right)\right] - f(x)$$

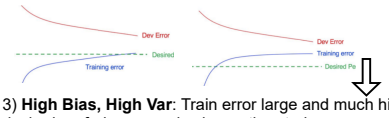
Note realization implies access to only a portion of the underlying data as training data (unrepresentative of the underlying population)

Variance: mean squared deviation of $\widehat{f}(x)$ from its expected value $E\left[\widehat{f}(x)\right]$ over diff realizations of training data:
$$var\left[\left(\widehat{f}(x)\right)\right] = E\left[\left(\widehat{f}(x) - E\left[\widehat{f}(x)\right]\right)^2\right]$$

Learning Curves:
1) **High Bias**: Error plateau, dev error higher than train error, very high train error



2) **High Var, Low Bias**: Train error relatively low, dev error much higher than train error, bias is small but variance is large. Perhaps add more training data will close the gap



3) **High Bias, High Var**: Train error large and much higher than desired perf, dev error also larger than train.

Regularisation:
Used to discourage the complexity of the model by penalizing the loss function (help solve overfitting)

1) L1 (Lasso/L1 Norm)
Feature selection by assigning insig input features with 0 weight & useful features with a non-0 weight

λ is the penalty (reg) param: If 0, back to original loss function no penalty, if too large, weights become close to 0, hence high bias (underfitting)

2) L2 (Ridge)
Forces the weights to be small but $\neq 0$ (non-sparse)
Not robust to outliers as square terms blows up the error differences of the outliers & λ tries to fix it by penalizing the weights.

Performs better when all the input features influence the output and all with
weights are of roughly equal size

$$L(x, y) \equiv \sum_{i=1}^{n} (y_i - h_\theta(x_i))^2 + \lambda \sum_{i=1}^{p} |\theta_i| \text{vs} \sum_{i=1}^{p} \theta_i^2$$

θ_i is the coeff of the i-th term

Put the L1/L2 into the optimizer or use torch.norm(), total_loss = loss + L1/L2, then backward prop

Resource Limitations: Standard Approaches:
a) **Cutoff test**: On the depth limit
b) **Evaluation fn**: Heuristic scoring on the est. desirability of current state

Note for the eval fn values, behaviour is preserved under any monotonic transformation of itself, only the order matters

$\alpha - \beta$ pruning
α: best value (to MAX) found so far off the current path
β: best value (to MIN) found so far off the current path
Prune if $\alpha \geq \beta$

- Pruning only affect the size of the search tree, not result
- Good move ordering improves effectiveness of pruning
- 'Perfect ordering': $O\left(b^{\frac{m}{2}}\right)$, hence we can double the depth of search

$\hat{y} = g(w_0 + X^T w)$ linear decis^n bc matter network size
Nonlinear $g(a) \Rightarrow$ approx. of arbit. complex f^n

i) Threshold : $[a > 0]$
ii) Sigmoid σ : $\frac{1}{1+e^{-a}}$
iii) ReLu : $\max(0, a)$
-non 0 centered, nondiff @ 0, dying problem

XOR : simplest linearly inseperable problem

a) Reg. : Linear $(-\infty, +\infty)$/ Relu$(0, +\infty)$ & MSE loss
b) Bin class. : $\sigma(0,1)$ & BCE loss : $\sigma + CE$ loss = σ CE loss
c) Categorical : Softmax$(0,1)$ $\forall$ class, Σ to 1 & CE loss : Softmax g+ CE loss = Softmax loss
d) Multiclass cat : —|— & BCE loss $\forall$ class

Empirical loss: Total loss over entire data

$J(w) = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}(f(x^{(i)}, w), y^{(i)})$ pred. act.

MSE: $\frac{1}{n}\sum_{i=1}^{n} (y^{(i)} - f(x^{(i)}, w))^2$

BCE: $\frac{1}{n}\sum_{i=1}^{n} [y^{(i)} \log(f(x^{(i)}, w)) + (1 - y^{(i)}) \log(1 - f(x^{(i)}, w))]$

Training : Find $w^* = \arg\min_w J(w)$, lowest loss
i) Init weights rand $\sim N(0, \sigma^2)$
ii) Loop until converg. $\nabla w = \frac{\partial J(w)}{\partial w}$, $w \leftarrow w - \eta \nabla w$
iii) Return w
- Small η : converge quickly (local optima) } Use adap!
- large η : unstable & diverges
- Easy compute, implement & understand
- May trap at local min (need convex f^n too)
- Update weights only after calculating ∇ on entire dataset (too large $\Rightarrow$ years to compute + large mem)

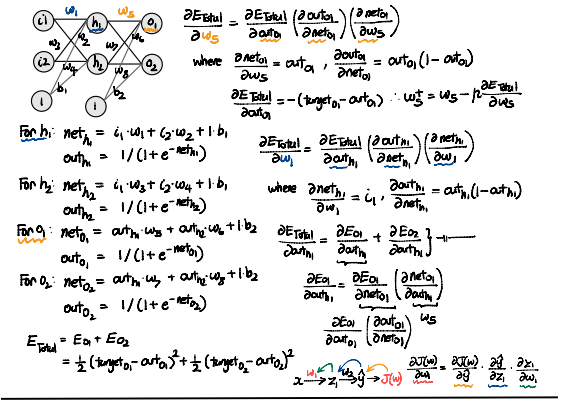
SGD: Update model param more freq.
- Converges in less time & may get new minima's
- High var in model param (may shoot ↑ even after global min)
- To get some convergence as GD, need slowly ↓ η

Pick Batch B of data } Fast compute & allow // compute
$\nabla w = \frac{1}{B}\sum_{k=1}^{B} \frac{\partial J_w(w)}{\partial w}$ } better est. of true ∇ than 1 datapoint, smoother converge, allow larger η

Adagrad : Changes η $\forall$ param & t (need 2nd order der.)
- No need tune η manually & trainable on sparse data
- η always ↓ → slow training - Comp. exp

Adadelta RMS Prob

ADAM : momentum of 1st+2nd order - Comp. exp.
- Very fast converge + rectifies decaying ∇ & high var



Generative Modeling: Take input train samples from some dist and learn a model that represents that dist (Debiasing)
- Capable of uncovering underlying features in a dataset
- Detect outliers in the dist for training

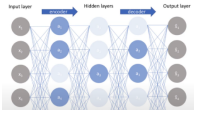
AE: Encoder: Unsupervised learning of a lower dim feature space (under complete), z (salient features + easier to work with) representation from unlabelled train data (Note no labels used.)
Decoder: Use z to reconstruct the obs $\hat{x}$
$L(x, \hat{x}) = ||x - \hat{x}||^2$

- Form of compression, smaller z -> larger training bottleneck
- Bottleneck hidden layers forces network to learn compressed latent rep. while reconstruct loss forces z to capture as much information of the data

Regularizer AE: Overcomplete case with a loss function that encourages the model to have other properties beside copy-paste

- sparsity rep, smallness of rep derivatives (similar img, sim lv), robust to noise/missing inputs

Sparse AE: $L(x, g(f(x)) + \Omega(z)$ ⇒
decoder, encoder output + sparse penalty
Denoising AE: Add noise to input image, AE learns to remove noise



Bagword (BoW): Columns: Sentences, Rows: all words
Uses Binary one-hot encoding (very sparse)

Term-Freq-Inv doc. freq (TF-IDF): Rare words carry more meaning (also very sparse)

TF = # appearances for a term / Total # words in document
IDF = log(# docs / # docs term appear in)
TF-IDF = TF × IDF

Less-sparse Rep: Co-occurence models (Singular-value-decomp)
Vector embedding models (word2Vec)

Word Embeddings: Each word = a point in a n dimen space
Represented by vector of len n (~100-300), 1 layer

i) Cont.-BoW: Pred center word from surrounding context
ii) Skip-grams: Pred surr context words from given center word

- Maximum likelihood of context, given focus word
e.g. 1 love *pizza*, P(1|*pizza*) & P(love|*pizza*)

Train word2vec: (weights of layer) but no preserve order!
- Decide window size & embedding size & vocab size
- Training Obj: Negative sampling (binary LR class)
- Fast accurate model that captures semantic content
"Gensim" library, pre-trained word embedding
- Interested in the 1 hidden (latent rep) layer, not the output

CNN: Preprocessing cals e.g. Normalize to increase perf
- Decrease # params to learn, preserves locality (no need flatten image)
- For all conv. layer, # activation map = # fliters/feature maps/kernels
- Depth of input (channels) = Depth of filters too (Note depth of activation map output is 1 nevertheless)
- For 2D image H with 2D kernel F, Conv operator:

$$G = H \star F,\ G[i,j] = \sum_{u=-k}^{k}\sum_{v=-k}^{k} H[u,v]F[i-u, j-v]$$

However in CNN, do not flip kernel; instead of more complex above equation, use:

Cross-correlation Operator: [+] symmetric, kernels transposed

$$G[i,j] = \sum_{u=-k}^{k}\sum_{v=-k}^{k} H[u,v]F[i+u, j+v]$$

pos in filter (given dim 2k+1) — pos in resulting act map, centred at 0

- Still no difference in results, same output, just convention
- Stride: Output size = (N, input size - F, feature size / Stride) +1
- Padding: Add padding to output, decrease problem size

To preserve spatial size: Stride = 1, F by F filter size, with 0-padding of (F-1)/2 size

- Pooling: Non-linear down sampling (common max/ avg), no learning here, just the stride size and pooling window size, non overlapping best, if too large lose information

L to R, Top -down
Input -> Conv+Relu, Pooling -> ..... -> **Flatten**+FC+Softmax
Feature learn | Classification

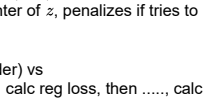
- Can be used for any vector/matrix representation
e.g. Audio: Spectrogram, Word: Embbedings

- Waveforms: Bunch of numbers over time
- More informative are spectrograms: a visual representation of the spectrum of frequencies of sound or another signal as they vary with time:
- Typical spectrogram of a few spoken words.
Y-axis: frequencies, X-axis: time, lower frequencies can be seen as denser (brighter), like a male voice
- The power spectrum of a time signal describes the power distribution into freq components composing that signal. Fourier analysis can decompose a signal into discrete freq (a spectrum of freq) over a continuous range.

- Effective: Mel Freq Cepstrum (non-linear) represents human hearing better. TA library nnAudio

Variational AE (VAE): Sample from mean & std dev to compute z (probabilistic, no longer deterministic).

E.g. $z =$ samp 1, samp 2 — smile: ⋮ latent dist.



$P_\phi(z \mid x) q_\phi(x \mid z)$

$L(\Theta, \theta, x) = \text{reconst loss} + \text{reg term}$
$= ||x - \hat{x}||^2 + \sum_j KL(P_j(z \mid x) \,||\, p(z))$ fixed prior ld, learned ld } KL diverg

- Choice of prior on ld: common $p(z) \sim N(0,1)$, encourages encodings to distribute evenly around center of z, penalizes if tries to cluster points tgt (memorise data)

- Generation (only sampling for z + decoder) vs
- Training (Encoder: produces z (μ + σ), calc reg loss, then ....., calc reconstruct loss, minimize both losses tgt

- Cannot backdrop grad through sampling layers, ∴ reparametrize sampling layer (sum fixed μ & σ scaled by random const drawn from prior dist. $z = \mu + \sigma \cdot \varepsilon \sim N(0,1)$

- Goal is to find disentangled uncorrelated meaningful lv
Latent Perturbation: Slowly increase/dec 1 lv, keep rest same for intepretation

Conditional VAE: Feed label Y into model, since VAE no control over the generated data

RNN: Input x_t vector, Output $\hat{y} = W_{hy}^T h_t$
Update hidden state: $h_t = \tanh\left(W_{hh}^T h_{t-1} + W_{xh}^T x_t\right)$
- Many values >1: Exploding gradients, need grad clipping
- Many values <1: Small numbers multiplication, gradient increasingly smaller, bias, param unable to capture short term dependencies
- Weights matrices updated after entire timesteps

a) ReLU, prevent f' from shrinking grad when $x > 0$
b) Weight Init: Weights use Identity matrix, Bias use 0, prevent weights shrinking to 0
c) Using gates to track and control information through time

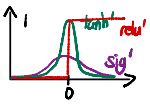
LSTM: - Forget irrelevant info of prev states
$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$

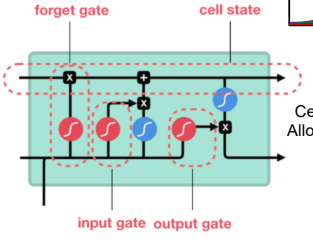
- Store relevant new info into cell state
$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$
$\overline{C}_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_c)$

○: pointwise multiplication, Hadamard mult

- Update cell state selectively
$C_t = f_t \circ C_{t-1} + i_t \circ \overline{C}_t$

- Output gate: Control what to send to next t
$O_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$
$h_t = O_t \circ \tanh(C_t)$

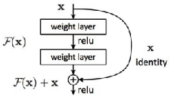




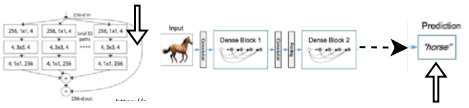
Cellstate highway: Allows uninterrupted grad flow

Transfer Learning (Opti): Higher start, slope & asymptote
- Identify related task with abundant data, reuse pretrained model

E.g. LeNet 5, AlexNet, VGG-16, Inception (GoogleLeNet)
- More layers -ve effect, degradation pro

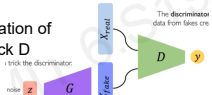


ResNet: residual blocks at intermediate layers (winner)
ResNext: "Split-transform-merge" + residual block connection



DenseNet: Each layer's feature map concatenated to the input of every successive layer within dense block

Generative Adversarial Networks (GAN):
Generator: Noise -> imitation of data to try trick D
Discriminator: Tries identify data from fakes generated by G



2 nn competing

Data: Real -> D use as +ve e.g., Fake -> D use as -ve e.g.
Alternative training:
D Training: D ignores G loss & uses its own loss
- Sample random noise to produce G output
- classify & calculate D loss
- BP. through D & G weight, only update G weights
G Training: Opposite
- G loss penalizes G for failing to fool D

If G perfect, D acc = 50%, D feedback gets less meaningful overtime, may collapse, sin ce GAN not stable, but fleeting convergence

Minimax loss: $\min_{\theta_g}\max_{\theta_d} \left[E_{X \sim P_{data}} \log D_{\theta_d}(c) + E_{Z \sim P(z)} \log\left(1 - D_{\theta_d}\left(G_{\theta_g}(z)\right)\right)\right]$
D output real; G output $|z$; D output fake

(CE loss btw real & fake dist)
- D max probability real & output prob fake low
- G no influence on 1st term, tries min 2nd term (1- ...) -> get high output for fake to fool D
- Can get stuck at early stages of training,
∴ use **Modified minimax loss:** $\max \log D(G(z))$ instead of $\min(1 - D(G(z)))$

Deep Conv GANs: Uses leaky ReLu (-ve slope before x = 0), sin ce ReLU is non-0 centered, non diff at 0 & dying problem

Conditional GANs: same feed label Y **(paired data)**
Used for object transfiguration
a) Pix2pix(cGAN with original img as condition)
- Paired data, D compares $G(x)$ and y given x aka generate img in different style from original img
- To make img less blurry use L1/L2

$G\star = arg \min_G \max_D L_{cGAN}(G, D) + \lambda L_{L1}(G)$

Cycle GAN (Unpaired data): Domain transformation
- Uses adversarial loss + inverse mapping (sin ce mapping highly constrained). Introduces a cycle consistency loss to push $F(G(x)) \approx X$
- Not just img, e.g. waveform, but not spectrogram
- Used to convert accented speech

Planning is the process of computing several steps of a problem-solving procedure before executing any of them (can be solved by search)
- Main difference between search and planning is the representation of states

- **Search:** rep as a single entity (which may be a complex object, but the search algorithm does not use its internal structure algorithm)
- **Planning:** states have structured representations (collections of properties) which are used by the planning algorithm

Types of Planners:
1. Domain-specific: Made or tuned for a specific planning domain, won't work well (if at all) in other planning domains
- Most successful real-world planning systems work this way (e.g. Mars exploration, sheet-metal bending, playing bridge)
- Often use problem-specific techniques that are difficult to generalize to other planning domains

2. Domain-independent: In principle, it works in any planning domain, no domain-specific knowledge except the description of the system
- In practice, need restrictions on what kind of planning domain, not feasible to make domain-independent planners work well in all possible planning domains
- Make simplifying assumptions to restrict the set of domains
  - Classical planning
  - Historical focus of most research on automated planning

A domain-specific planner may be able to go directly toward a solution in situations where a domain-independent planner would explore many alternative paths.

3. Configurable: Domain-independent planning engine
- Input includes info about how to solve problems in some domain
- Generally this means one can write a planning engine with fewer restrictions than domain-independent planners.

Classical planning: requires certain assumptions.
- Offline generation of action sequences for an environment that is fully observable, deterministic, finite, static and discrete
- Given a planning problem $P = (A, s_0, s_g)$
- Find a sequence of actions $(a_1, a_2, \dots a_n)$ that produces a sequence of state transitions $(s_1, s_2, \dots, s_n)$ such that s_n is in s_g
- Just path-searching in a graph (Nodes = states, Edges = actions)

STRIPS: Stanford Research Institute Problem Solver (1971)
- Originally a planner software, today mostly used to name a formal language to describe planning problems.
- (Logic-based) Language expressive enough to describe a wide variety of problems but restrictive enough to allow efficient algorithms to operate over it
- Planning is about finding a sequence of actions to achieve a goal

Propositional variables/facts: T/F variables that model and describe some aspect (state) of the world

STRIPS Instance: 1) An initial state
2) Specification of the goal states – situations planner is trying to reach
3) Set of actions. For each action, the following are included:

- preconditions: T/F facts that must hold st. an action can be performed
- postcondition: facts that change when an action is performed

P - a set of propositional variables
O - a set of operators (actions), 3-tuple:

- pre_a: facts that must be true before the action can be performed
- add_a: facts that will change to T when/after the action can be performed
- del_a: facts that will change to F when/after the action can be performed
- (requirement: $\text{add}_a \bigcap \text{del}_a = \emptyset$)

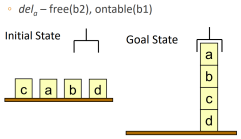
I - the initial state of the world, T/F assignments to variables from P
G - the goal state of the world

State Representation:
World states are rep by sets of facts: conjunction of propositions (conditions) where each proposition states a fact that attributes "values" to features
(e.g. Robot World: State 1 = { inA(robot) ∧ inA(ball) ∧ dooropen(A,B)})

- Goals states are also rep as sets of facts (e.g. state { **inB(ball)** })
- Goal state is any state that includes all the goal facts
(e.g. State 3 = { dooropen(a,b) ∧ inB(robot) ∧ **inB(ball)** })

Closed World Assumption (CWA): Facts not listed in a state are assumed to be F. Agent has full observability and only +ve facts stated

- ***P*** - a set of propositional variables.
  - ontable(block1): block1 is on the table
  - on(block1,block2): block1 is on block2
  - free(block1): top of block1 is free
  - clawempty: the robot claw is not holding any blocks
- ***I*** - the initial state of the world.
  - ontable(c), ontable(a), … , free(c), … , clawempty
- ***G*** - the goal state of the world.
  - on(a,b), on(b,c), on(c,d), ontable(d)
- ***O*** - a set of operators (i.e., actions). E.g., for the grabandstack(block1,block2) operator:
  - pre_o – free(b1), free(b2), clawempty, ontable(b1)
  - add_o – on(b1,b2)
  - del_o – free(b2), ontable(b1)



Planning Domain Definition Language (PDDL): language to describe planning problems that can be used as input to planner software.Std language for defining classical planning problem.

2 files: Problem file for objects, initial state and goal specification
Domain file for predicates (= facts) and actions.

PDDL and STRIPS:
- PDDL includes STRIPS as a special case along with more adv features
- PDDL includes some simple additional features, such as type specification for objects, negated preconditions, conditional add/del effects
- Adv features: e.g. allowing numeric variables and durative actions

Components:
- Objects: Things in the world that interest us.
- Predicates: Properties of objects that we are interested in (can be T/F)
- Initial state: The state of the world that we start in.
- Goal specification: Things that we want to be true.
- Actions/Operators: Ways of changing the state of the world.

Domain Files:
(define (domain <domain name: str that identifies the planning domain, e.g. gripper>)
  <PDDL code for predicates>
  <PDDL code for first actions>
  <...>
  <PDDL code for last actions>)

Problem Files
(define (problem <problem name: str that identifies the planning task, e.g. gripper-four>)
  (: domain <domain name, e.g. gripper>)
  <PDDL code for objs>
  <PDDL code for initial state>
  <PDDL code for goal specification>)

E.g. Gripper task with four balls:
A robot that can move between two rooms and pick up or drop balls with either of his two arms. Initially, all balls and the robot are in the 1st room. We want the balls to be in the 2nd room.
• Objects:
- Rooms: rooma, roomb
- Balls: ball1, ball2, ball3, ball4
- Robot arms: left, right
• Predicates:
- ROOM(x) – true iff x is a room
- BALL(x) – true iff x is a ball
- GRIPPER(x) – true iff x is a gripper (robot arm)
- at-robby(x) – true iff x is a room and the robot is in x
- at-ball(x, y) – true iff x is a ball, y is a room, and x is in y
- free(x) – true iff x is a gripper and x does not hold a ball
- carry(x, y) – true iff x is a gripper, y is a ball, and x holds y
• Initial state:
- ROOM(rooma), ROOM(roomb) are T
- BALL(ball1), ..., BALL(ball4) are T
- GRIPPER(left), GRIPPER(right), free(left), free(right) are T
- at-robby(rooma), at-ball(ball1, rooma), ..., at-ball(ball4, rooma) are T
- Everything else is F
• Goal specification:
- at-ball(ball1, roomb), ..., at-ball(ball4, roomb) must be T.
- Everything else we don't care about.
• Actions/Operators:
- Move Operator:
  Description: Robot can move from x to y
  Precondition: ROOM(x), ROOM(y), at-robby(x) are T
  Effect: at-robby(y) becomes T. at-robby(x) becomes F.
  Everything else doesn't change.
...

In PDDL:
```
(:objects rooma roomb
          ball1 ball2 ball3 ball4
          left right)
(:predicates (ROOM ?x) (BALL ?x) (GRIPPER ?x)
             (at-robby ?x) (at-ball ?x ?y)
             (free ?x) (carry ?x ?y))
(:init (ROOM rooma) (ROOM roomb)
       (BALL ball1) (BALL ball2) (BALL ball3) (BALL ball4)
       (GRIPPER left) (GRIPPER right) (free left) (free right)
       (at-robby rooma)
       (at-ball ball1 rooma) (at-ball ball2 rooma)
       (at-ball ball3 rooma) (at-ball ball4 rooma))
(:goal (and (at-ball ball1 roomb)
            (at-ball ball2 roomb)
            (at-ball ball3 roomb)
            (at-ball ball4 roomb)))
(:action move :parameters (?x ?y)
      :precondition (and (ROOM ?x) (ROOM ?y)
                         (at-robby ?x))
 :effect (and (at-robby ?y)
              (not (at-robby ?x))))
```

Heuristics: $h(n)$ = est cost from n to goal
Admissible: $h(n) \leq h^\star(n)$ where $h^\star(n)$: true cost from n
- Derived from the exact solution cost of a relaxed version of the problem
- Opt. solution cost of a relaxed $\leq$ real

Delete-relaxed Problem: For STRIPS planning problem, remove the negation of facts in all operators (remove del_a)

Every plan (consists of a sequence of actions) that solves the original problem (with deletes), also solves the delete-relaxed problem. ∵ the goal is for certain facts to be set to true, it does not matter if other additional facts are true.

h^+ **heuristic:** Optimal plan (or min number of actions) for a delete-relaxed problem (no deletes)
- Can use the optimal plan for a delete-relaxed problem as a heuristic for a state in the original problem
- Never overestimates the cost in the original problem, $h(n) \leq h^\star(n)$
- Admissible by design for every state of the original problem
- Need to calculate the h^+ at each newly generated state:
- Solving for multiple delete-relaxed problems, each time based on a generated state as a start state (very expensive, hence need approximate h^+)

Faster Planner Search Heuristics ($h_{add}, h_{\max}, h_{ff}$)
- Assumes delete relaxed problem
- F_0 is the initial set of true facts
- Iterate: A_i is the set of all possible actions that can be applied on F_i (no deletes), resulting in $F_{i+1} = F_i + \text{some other facts}$
- Terminate at M^{th} iteration when set F_m of facts contains all the goal facts

Concise representation using F & A allows us to tell the **level/index of a fact**: which tells us the **number of actions (cost) required** to achieve that fact, useful to find $h_{add}, h_{\max}, h_{ff}$

$h_{\max}$**:** cost of the single most costly goal fact (out of all goal facts), the max number of actions needed for achieving one of the goal facts **(admissible)**

- Optimistic heuristic: Implicit assumption is that an action can set multiple facts to T
- Tends to underestimate the true cost (too loose, low)
- Only considers the longest path of actions (assumes that all goals are on that, ignores that there can be partially non-overlapping or independent paths of actions leading to multiple goals.

h_{add}**:** summed cost of all goal facts, adds up the number of actions needed to set a fact to T **(inadmissible)**

- Pessimistic heuristic: Implicit assumption is that an action can only set one fact to T, v.versa
- Tends to overestimate true cost (too large, > h^+ we approx)
- Assumes that there are only independent paths leading to each goal. However, multiple goals or facts can exist on a single path of action

h_{ff}**:** Uses backward pass, unlike prev 2 fwd pass **(inadmissible, practical)**
-In reality, paths of actions can be partially independent and partially overlapping/shared, h_{ff} allows it to find a common set of shared actions, i.e., paths to make all goal facts T

a) Loop through all goal facts, starting from the last one at level M
b) ∀ goal fact, check backwards for action that made it possible
c) Continue all the way until the initial state

- Concise representation using facts (F) and actions (A)
  - F_0 = x_1
  - A_0 = o_1,
  - F_1 = x_1, x_2
  - A_1 = o_2
  - F_2 = x_1, x_2, x_3
  - A_2 = o_3
  - F_3 = x_1, x_2, x_3, x_4
  - A_4 = o_4
  - F_4 = x_1, x_2, x_3, x_4, x_5
  - A_5 = o_5
  - F_5 = $x_1, x_2, x_3, x_4, x_5, x_6$

Variables: $x_1, x_2, x_3, x_4, x_5, x_6$
Initial State: x_1
Goal: x_2, x_5, x_6
Actions:
o_1 : pre: x_1, post: x_3
o_2 : pre: x_3, post: $\neg x_1$, x_3
o_3 : pre: x_3, post: $\neg x_3$, x_1, x_4
o_4 : pre: x_3, x_4, post: x_5
o_5 : pre: x_1, x_5, post: $\neg x_1$, x_6

Task: Calculate the h_{max} and h_{add} heuristics
- h_{max} = max(1, 4, 5) = 5
- h_{add} = 1 + 4 + 5 = 10

Bayes Rule:
$$P(A \mid B) = \frac{P(A \cap B)}{P(B)} = \alpha P(A \cap B) \Rightarrow \frac{P(B \mid A)P(A)}{P(B)}$$
Chain rule: $P(x_1, ..., x_n)$
$= P(x_n \mid x_1, ...x_{n-1})P(x_1, ...x_{n-1})$
$= P(x_n \mid x_1, ...x_{n-1})P(x_{n-1} \mid x_1, ..., x_{n-2})P(x_1, ..., x_{n-2})$
$= \prod_{i=1}^{n} P(x_i \mid x_1, ..., x_{i-1})$

	toothache		¬toothache	
	catch	¬catch	catch	¬catch
cavity	.108	.012	.072	.008
¬cavity	.016	.064	.144	.576

$$P(\text{-cavity} \mid \text{toothache}) = \frac{P(\text{-cavity} \cap \text{toothache})}{P(\text{toothache})}$$
$$= \frac{0.016 + 0.064}{0.108 + 0.012 + 0.016 + 0.064} = 0.4$$
$$= \alpha \cdot P(\text{-cavity} \cap \text{toothache})$$
$$= \alpha \cdot [P(\text{-cavity}, \text{ta}, \text{catch}) + P(\text{-cavity}, \text{ta}, \text{-catch})]$$
$$= \alpha \cdot (0.016 + 0.064)$$

Let Y = query var, E = explanatory var, H = hidden var
$$P(Y \mid E = e) = \alpha P(Y, E = e) = \alpha \sum_h P(Y, E = e, H = h)$$

1) Worst-case time complexity $O(d^m)$, d is the largest arity
2) Space complexity $O(d^m)$ to store the joint distribution

Independence: $A \perp B \Leftrightarrow P(A, B) = P(A)P(B)$
$P(A \mid B) = P(A)$ vice versa

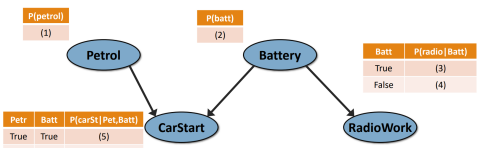
Conditional Indep: A, B independent given C:
$P(A, B \mid C) = P(A \mid C) \cdot P(B \mid C)$
$P(A \mid B, C) = P(A \mid C)$

If all have 2 values each: $P(A, B, C) = 2 \cdot 2 \cdot 2 = 8$ var
$P(A)P(B)P(C) = 2 + 2 + 2 = 6$ var

$$P(A) = P(A \mid B)P(B) + P(A \mid -B)P(-B)$$

Join-table: Way too big to represent explicitly
- Hard to learn (est.) anything empirically about more than a few var at a time

Bayes' Network: (Graphical models): A simple, graphical notation for cond. indep. assertions and hence for compact specification of full joint distributions
- a set of nodes, one per variable
- a set of arcs, representing "directly influences"
- a cond. dist. ∀ node given its parents (rep by cond. prob table)



Global semantics: full joint dist. as the pdt of local cond. dist.

$$P(!p \mid !c, r) = \frac{P(!p, !c, r)}{P(!c, r)}$$
$$= \frac{\sum_b P(!p, !c, b, r)}{\sum_b \sum_p P(p, !c, b, r)}$$
$$= \frac{\sum_b P(!p)P(!c \mid !p, b)P(b)P(r \mid b)}{\sum_b P(b) \sum_p P(!c \mid p, b)P(p)P(r \mid b)}$$