Fully Observable assumption (closed world assumption): All necessary information about a problem domain is accessible so that each state is a complete description of the world; there is no missing information at any point in time**. State Space**: all possible valid configurations of the environment; directed graph (V, E). Goal Test: defines what it means to have achieved/satisfied the goal; is applied to a node's state to determine if it is a good goal node. **Expanding a node**: - generate all of the successor nodes. – add them and their associated arcs to the state-space search tree. A **solution path** is a sequence of actions associated with a path in the state space from a start to a goal node. The generated, but not yet expanded, states define the **Frontier set**. Each node implicitly represents a partial solution path from the start node to the given node; cost of the partial solution path. Changing the Frontier ordering leads to different search strategies. BFS: Queue (FIFO) remove from front, add to back. DFS: Stack (LIFO) remove from front, add to front; performs "chronological backtracking": - when search hits a dead end, backs up one level at a time. – problematic if the mistake occurs because of a bad action choice near the top of search tree. When backtracking the solution path, find the parent of the current node. UCS: Priority Queue to order nodes on the frontier list, sorted by path cost. **IDS**: -do DFS to depth 1 and treat all children of the start node as leaves; – if no solution found, do DFS to depth 2; – start node is at depth 0; "Anytime" algorithm can return a valid solution to a problem even if it's interrupted at any time before it ends. **Bidirectional Search**: BFS from both start and goal; stop when Frontiers meet; **Heuristic function h(n)**: - uses domain-specific information in some way; - is computable form current state description; - it estimates the goodness of node n; h(n) close to 0 means n is close to goal node; the minimal cost path from n to a goal state. **Best-First Search**: sort nodes in the frontier list by increasing values of an evaluation function f(n), that incorporates domain-specific information. **Greedy Best-First Search**: use as an evaluation function f(n) = h(n), sorting nodes in the frontier by increasing values of f. **Beam Search**: use an evaluation function f(n) = h(n) as in Greedy Best-First search, and restrict the maximum size of the Frontier to a constant, k. **A Search**: f(n) = g(n) + h(n), where g(n) is minimum cost path from start to current node; h(n) is the estimated cost from node n to goal node. **A* Search**: same as A, except add constraint that for all nodes n in the search space, h(n) <= h*(n), where h*(n) is the actual cost of the minimum-cost path from n to goal; when h(n) <=h*(n) holds true for all n, h is called **an admissible heuristic function** which guarantees that a node on the optimal path cannot look so bad so that it is never considered. A heuristic, h, is called **consistent** if for every node n and every successor n' of n, the estimated cost of reaching the goal from n is no greater than the step cost of getting to n' + estimated cost of reaching the goal from n': h(n) <= c(n, n') + h(n'). **Devising Heuristic**: the closer h is to h*, the fewer extra nodes that will be expanded. If h1(n) <= h2(n) <=h*(n), then A2* is better informed than A1*. An **operator** is needed to transform one solution to another. **TSP**: 2-swap operator; 2-interchange operator ABCDE with interchange(A,D)=>DCBAE. Those solutions that can be reached with one application of an operator are in the current solution's **neighborhood (move set).** An evaluation function, f, is used to map each solution/state to a number corresponding to the quality/cost of that solution. Maximize f: hill-climbing (does not allow backtracking since no frontier list). **Hill-climbing with random restarts**: 1. When stuck, pick a random new starting state and re-run hill-climbing from there. 2. Repeat this k times. 3. Return the best of the k local optima. Escaping local optima: pick a bad move which accepted with a probability that decreases as the search proceeds. Genetic Algorithm {Crossover(help get out of a local maximum): requires genetic diversity; offspring that have some traits of each parent; for vector representation – pick pairs of individuals as parents and randomly swap their segments; parameter(# of crossover points, positions of crossover points). 1-point crossover: pick a dividing point in the parents' vectors and swap their segments. N-point, Uniform: the values of each element of the vector is randomly chosen from the values in the corresponding elements of the two parents. Mutation(gives worse fitness): randomly change an individual; parameter(mutation rate, size of mutation) Natural selection: better organisms survive in a competitive environment. Evaluation ranks the individuals using some fitness measure that corresponds with the quality of the individual solutions. **Deterministic Selection, Proportional Fitness Selection**: 1. Rank selection: individual selected with a probability proportional to its rank in population sorted by fitness. 2. Proportional selection: individual selected with a probability: Fitness(individual)/∑ Fitness for all individuals. Tournament: $(2s-2r+1)/s^2$ (s is the size of the population, r [start from 0] is the rank of the "winning" individuals). Crowding: occurs when the individuals that are most-fit quickly reproduce so that a large percentage of the entire population looks very similar; reduces diversity in the population; may hinder the long-run progress of the algorithm. Zero-sum: one player's gain is the other player's loss. Does not mean fair. Perfect information: each player can see the complete game state. No simultaneous decisions. Deterministic: no coin flips, no chance. Minimax principle: max-choose the max from its children; min-choose the min from its children. Time Complexity: $O(b^d)$ $[branching\ factor, depth]$. $Static\ Evaluation\ function\ use\ heuristics\ to\ estimate\ the\ value\ of\ non-terminal\ states$. More effective (more cutoffs). Worse Case: ordered so that no pruning takes place. Best Case: each player's best move is visited first. **Dealing with Limited time**: [have a depth limit] use IDS (run alpha-beta search with DFS and an increasing depth limit). **The Horizon Effect**: quiescence search (when SBE values is frequently changing, look deeper than the depth limit; look for point when game quiets down); Secondary Search (find best move looking to depth d; look k steps beyond to verify that it still looks good; if it does not, go back to last step). Non-Deterministic Games: computer moves; chance nodes; opponent moves. **Chance nodes**: representing random events. Expected value for move: instead of using max or min, compute the average, weighted by the probabilities of each child. Minimax: determine the "optimal" moves by assuming that both players always chooses their best move. Alpha-beta: can avoid large parts of the each tree, thus enabling the search to go deeper. Inductive learning: generalize from a given set of examples so that accurate predictions can be made about future examples. An example or instance, x represents a specific object. Each dimension is called a feature or attribute. X is a point in the D-dimensional feature space. A training set is a collection of examples, which is the input to the learning process. Hierarchical Agglomerative Clustering: build a binary tree (dendrogram [the tree can be cut at any level to produce different numbers of cluster]) over the dataset by repeatedly merging clusters. **Single-linkage**: the shortest distance from any member of one cluster to any member of other cluster. **Complete-linkage**: the largest distance from any member of one cluster to any member of the other cluster. **Average-linkage**: the average distance between all pairs of members, one from each cluster. **K-Means Clustering**: specify the desired number of clusters and use an iterative algorithm to find them. Each iteration will reduce the distortion of the clustering. **Supervised learning**: a functino h: x -> y in H, such thatn h(x) predicts the true label y on the future data, x. classification: y is discrete; regression: y is continuous. A label y is the desired prediction for an instance x. **Discrete label**: classes. Determine if a given eample is or is not an instance of the concept/class: if it is, positive example; otherwise, negative. Given a training set of positive and negative examples of a concept. Inductive inference is "falsity preserving". Learning can be viewed as searching the Hypothesis Space H of possible h fuctions**. Inductive bias**: is used when one h is chosen over another; is needed to generalize beyond the specific training examples. Preference Bias: define a metric for comparing h's so as to determine whether one is better than another. A decision tree is a tree: each non-leaf node has associated with it an attribute; each leaf node had associated with it a classification; each arc has associated with it one of the possible values of the attribute of its parent node. **Ockham's Razor**: the simplest hypothesis that is consistent with all observations is most likely; the smallest decision tree that correctly classifies all of the training examples is best. Max-Gain: the attribute that has the largest expected information gain. Information gain: symmetric; mutual information (I(A; B) = I(B; A)); select attribute that will result in the smallest expected tree size. ***I(Y ; X) = H(Y) - H(Y|X)***. Information Theory: $(p/(p+n)\log_2 p + (n/(p+n)log\log_2 n))$ [p = set P, n = set N]. Entropy: H(Y)>=0, where 0 is no information, and 1 is maximum information; small entropy predicts a small tree size; large entropy predicts a large tree size. The best attribute is maximum information gain or minimum conditional gain. Use Test set to evaluate accuracy = #correct / #total. Overfitting: -meaningless regularity if found in the data. – irrelevant attributes confound the true, important, distinguishing features. – fixed by pruning some nodes in the decision tree; As d increase, Mean Square Error on training set improves, but prediction on test data worsens. Build (decision tree) classifier using the Training set. Estimate future performance using the Test set. If state space is not a tree, problem will be repeat states we have to remember Explored set: already-expanded states. **Random Forest**: collection of independently-trained binary decision trees; 1.choose a training set by choosing n times with replacement from all N available training examples 2. At each node of decision tree during construction, choose a random subset of m attributes from the total #, M, of possible attributes. 3. Select best attribute at node using Max-Gain. **Bagging:** create classifiers using different training sets. Expectimax: the nodes after chance node add up * chance node. Numerical attributes can repeat used, categorical attribute cannot repeat used.

Formalizing Search in a State Space

State-space search is the process of searching through a state space for a solution by **making *explicit* a sufficient portion of an implicit *state-space graph*, in the form of a *search tree*, to include a goal node:**

***TREE SEARCH Algorithm*:**

Frontier = {*S*}, where *S* is the start node
Loop do
  if *Frontier* is empty **then return** failure
  pick a node, *n*, from *Frontier*
  if *n* is a goal node **then return** solution
  Generate all *n*'s successor nodes and add them all to *Frontier*
  Remove *n* from *Frontier*

Performance of Search Algorithms on Trees

b: branching factor (assume finite) *d*: goal depth *m*: graph depth

	Complete	optimal	time	space
Breadth-first search	Y	Y, if [1]	$O(b^d)$	$O(b^d)$
Uniform-cost search[2]	Y	Y	$O(b^{C^*/\varepsilon})$	$O(b^{C^*/\varepsilon})$
Depth-first search	N	N	$O(b^m)$	O(bm)
Iterative deepening	Y	Y, if [1]	$O(b^d)$	O(bd)
Bidirectional search[3]	Y	Y, if [1]	$O(b^{d/2})$	$O(b^{d/2})$

1. edge cost constant, or positive non-decreasing in depth
2. edge costs ≥ ε > 0. C^* is the best goal path cost
3. both directions BFS; not always feasible

A and A* Algorithm for General State-Space **Graphs**

Frontier = {*S*} where *S* is the start node
Explored ={}
Loop do
  if *Frontier* is empty **then return** failure
  pick node, *n*, with min *f* value from *Frontier*
  if *n* is a goal node **then return** solution
  foreach each child, *n'*, of *n* **do**
    if *n'* is not in *Explored* or *Frontier*
      then add *n'* to *Frontier*
    else if *g*(*n'*) ≥ *g*(*m*) **then** throw *n'* away
      else add *n'* to *Frontier* and remove *m*
  Remove *n* from *Frontier* and add *n* to *Explored*

Note: *m* is the node in *Frontier* or *Explored* that is the same state as *n'*

Simulated Annealing (Stochastic Hill-Climbing)

1. Pick initial state, *s*
2. Randomly pick state *t* from neighbors of *s*
3. **if** *f*(*t*) better than *f*(*s*)
   then *s* = *t*
   else with small probability *s* = *t*
4. Goto Step 2 until some stopping criterion is met

Pick initial state, *s*
k = 0
while *k* < *kmax* {
  T = temperature(*k*)
  Randomly pick state *t* from neighbors of *s*
  if *f*(*t*) > *f*(*s*) **then** *s* = *t*
  else if ($e^{(f(t)-f(s))/T}$) > random() **then** *s* = *t*
  k = *k* +1
  }
return *s*

Escaping Local Maxima

Let $\Delta E = f(newNode) - f(currentNode) < 0$
$p = e^{\Delta E / T}$ (Boltzman's equation*) **Idea: *p* decreases as neighbor gets worse**

- $\Delta E \to -\infty$, $p \to 0$
  as move gets worse,
  probability of taking it ***decreases*** exponentially
- $T \to 0$, $p \to 0$
  as temperature ***decreases***
  probability of taking bad move ***decreases***

Escaping Local Maxima

Let $\Delta E = f(newNode) - f(currentNode)$
$p = e^{\Delta E / T}$

- $\Delta E << T$
  if badness of move is small compared to *T*, move is ***likely*** to be accepted
- $\Delta E >> T$
  if badness of move is large compared to *T*, move is ***unlikely*** to be accepted

Control of Annealing Process

Cooling Schedule:

+ *T*, the *annealing temperature*, is the parameter that control the frequency of acceptance of bad steps
+ We gradually reduce temperature *T*(*k*)
+ At each temperature, the search is allowed to proceed for a certain number of steps, *L*(*k*)
+ The choice of parameters $\{T(k), L(k)\}$ is called the ***cooling schedule***

Decision-Tree-Learning Algorithm

buildtree(*examples*, *attributes*, *default-label*)
if empty(*examples*) then return *default-label*
if (*examples* all have same label *y*) then return *y*
if empty(*attributes*) then return majority-class of *examples*
q = best_attribute(*examples*, *attributes*)
tree = create-node with attribute *q*
foreach value *v* of attribute *q* do
  v-ex = subset of examples with *q* == *v*
  subtree = **buildtree**(*v-ex*, *attributes* - {*q*}, *majority-class*(*examples*))
  add arc from *tree* to *subtree*
return *tree*

Measuring Cluster Quality

- **Distortion** = Sum of squared distances of each data point to its cluster center:

$$\sum_{\text{clusters } i} \sum_{\text{points p in cluster } i} \|p - c_i\|^2$$

- The "optimal" clustering is the one that minimizes distortion (over all possible cluster center locations and assignment of points to clusters)

Information Extremes

- 2 classes: + and –
- Perfect Balance (Maximum Inhomogeneity):
  given $p_+ = p_- = ½$
  $H(Y) = H(½, ½) = -½ \log_2 ½ - ½ \log_2 ½$
  $= -½ (\log_2 1 - \log_2 2) - ½ (\log_2 1 - \log_2 2)$
  $= -½ (0 - 1) - ½ (0 - 1)$
  $= ½ + ½$
  $= 1$ (➔ the entropy is large)
  A histogram of the frequency distribution of values of *Y* is nearly flat
- "High Entropy" means *Y* is from a nearly ***uniform*** distribution

- 2 classes: + and –
- Perfect Homogeneity:
  given $p_+ = 1$ and $p_- = 0$
  $H(Y) = H(1, 0) = -1 \log_2 1 - 0 \log_2 0$
  $= -1 (0) - ???$
  $= 0 - 0$
  $= 0$ (➔ no information content)
  A histogram of the frequency distribution of values of *Y* has many low values and a few high values
- "Low Entropy" means *Y* is from a varied (peaks and valleys) distribution

Entropy

When there are *k* classes, entropy is defined as

$$H(Y) = \sum_{i=1}^{k} -p_i \log_2 p_i$$

- p_i is the proportion of *Y* that belong to class *i*
- Log is still base 2 because entropy is a measure of expected encoding length measured in bits
- Maximum value of *H* is $\log_2 k$
  – For example, when *k* = 3, 0 ≤ *H* ≤ 1.58

Conditional Entropy

$$H(Y \mid X = v) = \sum_{i=1}^{k} -\Pr(Y = y_i \mid X = v) \log_2 \Pr(Y = y_i \mid X = v)$$

$$H(Y \mid X) = \sum_{v \text{values of } X} \Pr(X = v) H(Y \mid X = v)$$

called "Specific Conditional Entropy"

- *Y*: a label (or attribute)
- *X*: an attribute (i.e., feature or question)
- *v*: a value of the attribute
- *Pr*(*Y*|*X*=*v*): conditional probability
- Textbook calls *H*(*Y*|*X*) the ***Remainder***(*X*)

Pruning using a Greedy Algorithm

Prune(tree T, TUNE set)
1. Compute T's accuracy on TUNE; call it Acc(T)
2. For every internal node N in T:
   a) New tree T_N = copy of T, but prune (delete) the subtree under N
   b) N becomes a leaf node in T_N. The class is the majority vote of TRAIN examples reaching N
   c) Acc(T_N) = T_N's accuracy on TUNE
3. Let T* be the tree (among the T_N's and T) with the largest Acc()
   Set T = T* /* prune */
4. Repeat from Step 1 until no more improvement
5. Return T

Genetic Algorithm (1 version*)

1. Let $s = \{s_1, \ldots, s_N\}$ be the current population
2. Let $p[i] = f(s_i)/\text{SUM}_j f(s_j)$ be the fitness probabilities
3. for $k = 1$; $k < N$; k += 2
   - Parent1 = randomly pick s_i with prob. $p[i]$
   - Parent2 = randomly pick another s_j with prob. $p[j]$
   - Randomly select 1 crossover point, and swap strings of parents 1 and 2 to generate two children $t[k]$ and $t[k+1]$
4. for $k = 1$; $k \le N$; k++
   - Randomly mutate each position in $t[k]$ with a small probability
5. New generation replaces old generation: $s = t$

Minimax Algorithm

```
function Max-Value(s)
inputs:
    s: current state in game, Max about to play
output: best-score (for Max) available from s
    if ( s is a terminal state or at depth limit )
    then return ( SBE value of s )
    else
        v = −∞          // v is current best minimax value at s
        foreach s' in Successors(s)
            v = max( v , Min-Value(s'))
    return v

function Min-Value(s)
output: best-score (for Min) available from s
    if ( s is a terminal state or at depth limit )
    then return ( SBE value of s)
    else
        v = +∞          // v is current best minimax value at s
        foreach s' in Successors(s)
            v = min( v , Max-Value(s'))
    return v
```

General Minimax Algorithm

```
For each move by the computer:
1. Perform depth-first search, stopping
   at terminal states
2. Evaluate each terminal state
3. Propagate upwards the minimax values
     if opponent's move, propagate up
       minimum value of its children
     if computer's move, propagate up
       maximum value of its children
4. Choose move at root with the maximum
   of the minimax values of its children
```

Alpha-Beta Algorithm

```
function Max-Value (s, α, β)          Starting from the root:
inputs:                               Max-Value(root, -∞, +∞)
  s: current state in game, Max about to play
  α: best score (highest) for Max along path from s to root
  β: best score (lowest) for Min along path from s to root
  if ( s is a terminal state or at depth limit )
  then return ( SBE value of s )
  v = -∞          // v = best minimax value found so far at s
  for each s' in Successors(s)
    v = max( v, Min-Value(s', α, β))
    if (v ≥ β ) then return v   // prune remaining children
    α = max(α, v)
  return v        // return value of best child
```

```
function Min-Value(s, α, β)
  if ( s is a terminal state or at depth limit )
  then return ( SBE value of s)
  v = +∞          // v = best minimax value found so far at s
  for each s' in Successors(s)
    v = min( v, Max-Value(s', α, β))
    if (α ≥ v ) then return v   // prune remaining children
    β = min(β, v)
  return v        // return value of best child
```

K-Means Algorithm

- Input: $\mathbf{x}_1, \ldots, \mathbf{x}_n, k$ where each $\mathbf{x}_i$ is a point in a *d*-dimensional feature space
- **Step 1**: Select *k* cluster centers, $\mathbf{c}_1, \ldots, \mathbf{c}_k$
- **Step 2**: For each point $\mathbf{x}_i$, determine its cluster: Find the closest center (using, say, Euclidean distance)
- **Step 3**: Update all cluster centers as the centroids

$$\mathbf{c}_i = \frac{1}{num\_pts\_in\_cluster\_i} \sum_{\mathbf{x} \in \text{cluster } i} \mathbf{x}$$

- Repeat steps 2 and 3 until cluster centers no longer change

MCTS Algorithm

Recursively build search tree, where each round consists of:

1. Starting at root, successively select best child nodes using scoring method until leaf node *L* reached
2. Create and add best new child node, *C*, of *L*
3. Perform a random playout from *C*
4. Update score at *C* and all of *C*'s ancestors in search tree

Hierarchical Agglomerative Clustering Algorithm

Input: a training sample $\{\mathbf{x}_i\}_{i=1}^n$*; a distance function d*().
1. Initially, place each instance in its own cluster (called a singleton cluster).
2. ***while*** *(number of clusters* > 1*)* ***do:***
3. Find the closest cluster pair A, B, i.e., they minimize d(*A*, *B*).
4. Merge A, B to form a new cluster.
Output: a binary tree showing how clusters are gradually merged from singletons to a root cluster, which contains the whole training sample.

Cross-Validation

Often 3, 5 or

1. Divide all examples into *K* disjoint subsets $E = E_1, E_2, \ldots, E_K$
2. For each *i* = 1, …, *K*
   - let TEST set = E_i and TRAIN set = $E - E_i$
   - build decision tree using TRAIN set
   - determine accuracy PA_i using TEST set
3. Compute ***K*-fold cross-validation** estimate of performance = mean accuracy = $(PA_1 + PA_2 + \ldots + PA_K)/K$

Leave-One-Out Cross Validation

For *i* = 1 to *N* do // *N* = number of examples
1. Let (x_i, y_i) be the i^{th} example
2. Remove (x_i, y_i) from the dataset
3. Train on the remaining *N*-1 examples
4. Compute accuracy on i^{th} example

- Accuracy = mean accuracy on all *N* runs
- Doesn't waste data but is expensive
- Use when you have a small dataset (< ~100)

k-Nearest-Neighbors (k-NN)

Input: Training data $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)$*; distance function d*(); *number of neighbors k; test instance* $\mathbf{x}^*$
1. Find the k training instances $\mathbf{x}_{i_1}, \ldots, \mathbf{x}_{i_k}$ *closest to* $\mathbf{x}^*$ *under distance d*().
2. Output y^* *as the majority class of* $y_{i_1}, \ldots, y_{i_k}$*. Break ties randomly.*

Classify each pixel **x** in image *I* using *all T* decision trees and ***average*** the results at the leaves:

$$P(c|I, \mathbf{x}) = \frac{1}{T} \sum_{t=1}^{T} P_t(c|I, \mathbf{x})$$

$0 \leq H(Y \mid X) \leq 1$

Hill-Climbing Algorithm

1. Pick initial state *s*
2. Pick *t* in neighbors(*s*) with the largest *f*(*t*)
3. **if** *f*(*t*) ≤ *f*(*s*) **then** stop and **return** *s*
4. *s* = *t*. **Goto** Step 2.