Fa22 CS188 Midterm Cheat Sheet
by Will Tholke (SID: 3036863763)

Alpha-Beta Pruning

Time Complexity: $O(b^{m/2})$
When determining the value of a node n by looking at its children, *stop looking* as soon as you know that n's value can **at best** equal the optimal value of n's parent

Initial: $\alpha = -\infty$, $\beta = +\infty$

α: MAX's best option on path to root
β: MIN's best option on path to root

```
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        if v ≥ β return v
        α = max(α, v)
    return v
```

```
def min-value(state , α, β):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor, α, β))
        if v ≤ α return v
        β = min(β, v)
    return v
```

Properties: Alpha-Beta Pruning
Upper bound (best-case): max # of branches pruned
Lower bound (worst-case): min # of branches pruned, ≥ 0

- The first group of terminal nodes cannot be pruned
- Maximizers are always **suboptimal** when we prune on a lower bound, but are **optimal** when we prune on an upper bound
- **Minimum info needed to prune**: are *all* leaf node values are bounded by some upper bound? Ex: they are all negative (bounded by 0)

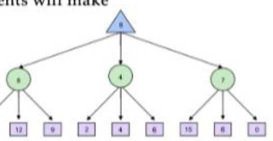
Evaluation Functions
Take in a state and output an estimate of the **true minimax value** of that node; oft used in **depth-limited minimax**, where non-terminal nodes at maximum solvable depth are treated as terminal nodes

Commonly represented as a linear combination of **features**, elements of the game that can be assigned a numerical value:
$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s)$

- Feature $f_i(s)$, extracted from state s
- Feature weight w_i

Note: when designing, make sure the function yields higher scores for better positions as often as possible

Expectimax
Used when facing suboptimal opponents; expected values of states computed using an average over the moves we believe opponents will make



Ex: first subtree $= \frac{1}{3}\times 3 + \frac{1}{3}\times 12 + \frac{1}{3}\times 9 = 8$
$\forall$ chance states, $V(s) = \sum p(s'|s) \times V(s')$
$p(s'|s)$: prob that action results in moving from state s to s'

Mixed Layer Types
In a game of Pacman with 4 ghosts, there is a Pacman (maximizer) layer followed by 4 consecutive ghost (minimizer) layers. If a ghost acts suboptimally, it will be represented by chance node

Monte Carlo Tree Search (MCTS)
Based on:

- **Evaluation by rollouts**: play from state s many times using a policy and count wins/losses
- **Selective search**: explore parts of the tree, without constraints, that will improve decision at the root

UCB Algorithm
Captures trade-off between promising and uncertain actions, using the criteria at each node n:

$$UCB1(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{\log(PARENT(n))}{N(n)}}$$

- $N(n)$: total number of rollouts from n (how **promising** the n is)
- $U(n)$: total number of wins for $Player(PARENT(n))$ (**uncertainty** of utility)
- C: balances the weight we put in the two terms (exploration and exploitation)

MCTS UCT Algorithm
Use UCB criteria in tree search problems to choose the action that leads to the child with highest N. As $N \to \infty$, UCT approaches the behavior of a minimax agent

Repeats these steps multiple times:

- Use criteria to **move down layers** of a tree from root until unexpanded leaf node reached
- **Add new child** to that leaf; **run a rollout** from that child to determine wins
- **Update** wins from the child back up to the root

11 Markov Decision Processes (MDPs)

Properties

- S: set of states
- A: set of actions
- γ: discount factor (implicitly 1)
- $T(s,a,s')$: transition function
- $R(s,a,s')$: reward function
- **Start state**
- **Terminal state(s)**
- **Q-states**, identical to expectimax chance nodes

Additive Utility
$U([s_0, a_0, s_1, a_1, \ldots]) = R(s_0, a_0, s_1) + R(s_1, a_1, s_2) + \ldots$

Discounted Utility
$U([s_0, a_0, s_1, a_1, \ldots]) = R(s_0, a_0, s_1) + \gamma R(s_1, a_1, s_2) + \gamma^2 R(s_1, a_2, s_3) + \ldots$

Markov Property
States that the future and past are conditionally independent given the present; knowing the past has no impact if we know the present: $T(s,a,s') = P(s'|s,a)$

Comparison: $\max$ **vs.** argmax

- $\max$ picks the max $f(a)$ over all $a \in A$
- argmax picks the $a \in A$ that maxes f

The Bellman Equation
An optimal policy yields the max expected total reward or utility for an agent that follows it

- **Policy** $\pi^* : S \to A$ maps each $s \in S$ to an $a \in A$
- $U^*(s)$: The Bellman Equation; optimal value of a state s
$$\max_a \sum_{s'} T(s,a,s')\left[R(s,a,s') + \gamma U^*(s')\right]$$
- $Q^*(s,a)$: optimal value of a Q-state (s,a)
$$\sum_{s'} T(s,a,s')\left[R(s,a,s') + \gamma U^*(s')\right]$$

Simplified Bellman : $U^*(s) = \max_a [Q^*(s,a)]$

Value Iteration
Compute optimal values of states by iterative updates until convergence, which is defined as $\forall s, U_{k+1}(s) = U_k(s)$:
1) $\forall s \in S$, initialize $U_0(s) = 0$
2) $\forall s \in S$, til convergence: $U_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s')\left[R(s,a,s') + \gamma U_k^*(s')\right]$

Policy Extraction
Determine a policy given some state value function
$$\forall s \in S, \pi^*(s) = \mathrm{argmax}_a [Q^*(s,a)] = \mathrm{argmax}_a \sum_{s'} T(s,a,s')\left[R(s,a,s') + \gamma U^*(s')\right]$$

Q-value Iteration
Compute time-limited Q-values; differs from value iteration because the position of the max operator selects an action before transitioning when we're in state, but we transition before selecting a new action when we're in a Q-state
$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s')\left[R(s,a,s') + \gamma \max_{a'}\left[Q_k(s',a')\right]\right]$$

Policy Iteration
Use policy evaluation and policy extraction to iteratively converge to an optimal policy; outperforms value iteration because policies usually converge faster than the values of states
1) Define an **initial policy**: can be arbitrary, but convergence is faster if its closer to the eventual optimal policy
2) Until convergence:
2a) Evaluate current π with **policy evaluation**:
$$U^{\pi}(s) = \sum_{s'} T(s,\pi(s),s')\left[R(s,\pi(s),s') + \gamma U^{\pi}(s')\right]$$
2b) Generate a better policy with **policy improvement**:
$$\pi_{i+1}(s) = \mathrm{argmax}_a \sum_{s'} T(s,a,s')\left[R(s,a,s') + \gamma U^{\pi_i}(s')\right]$$
If $\pi_{i+1} = \pi_i$, the algorithm has converged, meaning $\pi_{i+1} = \pi_i = \pi^*$

Key Takeaways

- A node n cannot find its optimal value if it travels on an infinite path, and it will converge to ∞
- A node with an out-degree of 0 converges to 0

12 Reinforcement Learning (RL)

Definitions
Solving an MDP is an example of **offline planning**, where agents know both the T and R functions, but in **online planning**, an agent has no prior knowledge of those functions.

- Each (s,a,s',r) tuple is a **sample**, and an **episode** is a collection of samples

Model-Based Learning
$\hat{T}$ is the approximate transition function generated by **normalizing** each of the counts it has collected, dividing the count for each observed tuple (s,a,s') by the sum over the counts when the agent was in $Q(s,a)$

- $\hat{T}(s,a,s') = \frac{\#(s,exit,s')}{\#s(exit)}$

Model-Free Learning

- **Passive RL**: agent follows a policy and learns the values of states as it experiences episodes
- **Active RL**: agent follows a policy and uses feedback to iteratively update its policy

Direct Evaluation (Passive)
Fix some policy π that an agent will follow and experience episodes

s	Total Reward	Times Visited	$V^{\pi}(s)$
A	-10	1	-10
B	16	2	8
C	16	4	4
D	30	3	10
E	-4	2	-2

Temporal Difference Learning (Passive)
Learn at every timestep; give exponentially less weight to older, less accurate samples with an **exponential moving average**; faster than direct evaluation

The **learning rate** $0 \leq \alpha \leq 1$ typically starts out at 1 and shrinks towards 0, at which point all samples are zeroed out and stop affecting our model of $V^{\pi}(s)$
$$\text{sample} = R(s,\pi(s),s') + \gamma V^{\pi}(s')$$
$$V^{\pi}(s) \leftarrow (1-\alpha)\cdot V^{\pi}(s) + \alpha \cdot \text{sample}$$

Q-Learning (Active)
Model free because it learns the values of Q-states directly; can learn the optimal policy directly even by taking suboptimal/random actions (called **off-policy learning**)
$$\text{sample} = R(s,a,s') + \gamma \max_{a'}[Q(s',a')]$$
$Q(s,a) \leftarrow (1-\alpha)\cdot Q(s,a) + \alpha \cdot \text{sample}$ **or**
$Q(s,a) \leftarrow Q(s,a) + \alpha \cdot \text{difference}$

Approximate Q-Learning (Active)
Generalizes learning experiences with **feature-based representation** of states in a **feature vector**. A feature vector for Pacman may encode: **distance** to the closest ghost, **distance** to the closest food pellet, **number** of ghosts, is Pacman trapped (0, 1)

Linear value functions:
$V(s) = w_1 \cdot f_1(s) + w_2 \cdot f_2(s) + \cdots + w_n \cdot f_n(s) = \vec{w} \cdot \vec{f}(s)$ **and** $Q(s,a) = w_1 \cdot f_1(s,a) + w_2 \cdot f_2(s,a) + \cdots + w_n \cdot f_n(s,a) = \vec{w} \cdot \vec{f}(s,a)$
$\text{difference} = [R(s,a,s') + \gamma \max_{a'}[Q(s',a')] - Q(s,a)$
$w_i \leftarrow w_i + \alpha \cdot \text{difference} \cdot f_i(s,a)$

ϵ Greedy Policies
Define a probability to $0 \leq \epsilon \leq 1$ to act randomly and **explore** with; **exploit** with $(1-\epsilon)$

Exploration Functions
Avoid manually tuning ϵ, with exploration function f:
$$Q(s,a) \leftarrow (1-\alpha)\cdot Q(s,a) + \alpha \cdot [R(s,a,s') + \gamma \max_{a'}[f(s',a')]$$

A common choice of f, with k as a predetermined value and $N(s,a)$: being the number of times $Q(s,a)$ has been visited
$$f(s,a) = Q(s,a) + \frac{k}{N(s,a)}$$

Fa22 CS188 Midterm Cheat Sheet
by Will Tholke (SID: 3036863763)

1 State Spaces

Graphs vs. Trees
State space graphs are too large to store in memory; **search trees** are better, and each node encodes the state and the path from the root to that node

Representation
World state: contains all information about a given state
Search state: contains only information necessary for planning
The **minimal state space representation** for a game includes non-static variables and essential information: dot booleans, (x,y) positions of agents that move, etc.

State Space Size
The **size** of a state space is determined by the minimum amount of information needed to know whether the game is complete
The **fundamental counting principle** states that if there are n variable objects in a given world which can take on $x_1, x_2, \ldots, x_n$ different values, then the total # of states is $x_1 \times x_2 \times \cdots \times x_n$

Example: Pacman
Variables: 1 Pacman with 120 unique (x,y) positions and 4 directions to face (NESW), 2 ghosts with 12 unique (x,y) positions, 30 food pellets that can be eaten/not eaten (base 2 since binary)
Size: $120 \times 4 \times 12^2 \times 2^{30}$
Branching Factor: 4, since Pacman can take 4 actions

2 Search Strategies, Overview

Properties

- **Completeness**: is it guaranteed to find the solution given infinite computational resources?
- **Optimality**: is it guaranteed to find the lowest cost path to a goal state?
- **Branching factor** b: the increase in the number of nodes on the frontier each time a frontier node is dequeued and replaced with its children $O(b)$: at depth k, there exists $O(b^k)$ nodes
- **Depth of shallowest solution** s
- **Max depth** m

3 Heuristics

Heuristics allow estimation of distance to goals, and are typically solutions to **relaxed problems**, which are when some of the original problem constraints have been removed

Manhattan distance
Commonly used to solve Pacman games
$M(x_1, y_1, x_2, y_2) = |x_1 - x_2| + |y_1 - y_2|$

Admissibility
A heuristic h is admissible if $\forall n, 0 \leq h(n) \leq h^*(n)$, where $h^*(n)$ is the true cost to a nearest goal. h is admissible $\leftrightarrow$ estimate $\leq$ cost.

- $g(n)$: total backwards cost computed by UCS
- $h(n)$: heuristic value function (estimated forward cost), used by greedy search
- $f(n) = g(n) + h(n)$ - total cost, used by A^* search

Theorem. For a given search problem, if the admissibility constraint is satisfied by a heuristic function h, using A^* *tree* search with h on that search problem will yield an optimal solution

Consistency
A **consistent** heuristic underestimates the *total* distance to a goal from any given node *and* the cost/weight of each edge in the graph
Theorem. For a given search problem, if the consistency constraint is satisfied by a heuristic function h, using A^* *graph* search with h on that search problem will yield an optimal solution
$\forall A, C \quad h(A) - h(C) \leq cost(A,C)$

Dominance
The metric of **dominance** is used to tell if one heuristic is better than another. If a heuristic a is dominant over b, then the estimated goal distance for a is greater than that of b for every node in the state space graph:
$\forall n : h_a(n) \geq h_b(n)$
The trivial heuristic is $h(n) = 0$, and using it reduces A^* to UCS

4 Uninformed Search

Depth-first Search
Select the *deepest* frontier node from the start node for expansion
Complete? No. Vulnerable to cycles, infinite-sized trees
Optimal? No. Finds leftmost solution without regard for path costs
Time Complexity: $O(b^m)$ for depth m. Explores entire tree in worst case
Space Complexity: $O(bm)$. Maintains b nodes at each of m depth levels on the frontier

Breadth-first Search
Explore the *shallowest* frontier node s from the start node for expansion
Complete? Yes. If a solution exists, the depth of the shallowest node must be finite, so BFS must find it
Optimal? No, but it *can* be optimal when all edge costs are equivalent (turns into uniform cost search). Searches without regard for path costs, like DFS
Time Complexity: (b^s) for the shallowest frontier node s. Searches $1 + b + b^2 + \cdots + b^s$ nodes in the worst case, traversing all nodes at every depth from 1 to s
Space Complexity: $O(b^s)$. Frontier contains all nodes in the level corresponding to the shallowest solution in the worst case, at depth s

Uniform Cost Search
Select the *lowest cost* frontier node from the start node for expansion
Complete? Yes. If a solution exists, it must have some finite-length shortest path, which UCS finds
Optimal? Yes, if we assume all edge costs are non-negative
Time Complexity: $O(b^{C^*/\epsilon})$ for the optimal path cost C^* and minimal cost between two nodes as ϵ; we must explore all nodes at depths from 1 to C^*/ϵ
Space Complexity: $O(b^{C^*/\epsilon})$. Frontier contains all nodes in the level corresponding to the cheapest solution

5 Informed Search

Greedy Search
Select the frontier node with the *lowest heuristic value* for expansion, which corresponds to the state it believes is nearest to a goal. Effectively a high-speed greedy search with the optimality and completeness of UCS
Complete? Optimal? No, neither, especially with bad heuristic function. Somewhat unpredictable

A^* Search
Select the frontier node with the *lowest estimated total cost* for expansion
Complete? Optimal? Yes, both complete and optimal, given an appropriate heuristic

6 Constraint Satisfaction Problems

CSPs are used to simply identify whether a state is a goal, regardless of how we arrived there

Attributes

- A set of N **variables** $X_1, \ldots, X_N$ that take on a single value
- A **domain** $\{x_1, \ldots, x_d\}$ of all possible values of a variable
- **Constraints** on the values of variables

Types of Constraints

- **Unary**: involves 1 variable, not represented in constraint graphs
- **Binary**: involves 2 variables, represented as graph edges
- **Higher-order**: involves > 3 variables, may be represented as graph edges

7 Solving CSPs

Backtracking Search
1) Fix an ordering for variables
2) Select unique values for variables that do not conflict with previously assigned values; if no such values exist, backtrack and return to the previous variable, changing its value

Filtering
Filtering: checks if we can prune the domains of unassigned variables ahead of time by removing values that will result in backtracking

Forward checking: when a value is assigned to a variable X_i, we prune the domains of unassigned variables that share a constraint with X_i that would violate the constraint if assigned

Arc Consistency Algorithm
Repeat until Q is empty or the domain of some variable is empty and triggers a backtrack:
1) Store all arcs in the constraint graph in a queue Q
2) Iteratively remove arcs from Q and enforce the condition that, in each removed arc $X_i \to X_j$, for every remaining value v for the tail variable X_i, there is at least one remaining value w for the head variable X_j such that $X_i = v$ and $X_j = w$ do not violate constraints. If some value v for X_i would not work with any remaining values for X_j, remove v from the set of possible values for X_i
3) If at least one value is removed for X_i from step 2, add arcs of the form $X_k \to X_i$ to Q for *all* unassigned variables X_k. If an arc $X_k \to X_i$ is already in Q during this step, it does not need to be added again

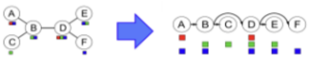
K-consistency
When enforced, guarantees that, for any set of k nodes in the CSP, a consistent assignment to any subset of $k-1$ nodes guarantees that the k^{th} node will have at least one consistent value

Strong K-consistency
Occurs when any subset of k nodes is not only k-consistent but also $k-1, k-2, \ldots, 1$ consistent as well

Ordering
Minimum Remaining Values (MRV) selects the unassigned variable with the smallest domain (most constrained variable), breaking ties alphabetically
Least Constraining Values (LCV) selects the value that prunes the fewest values from the domains of remaining unassigned values; slower than MRV

8 Solving Tree-Structured CSPs

Tree-Structured CSP Algorithm
1) **Pick** an arbitrary node in the CSP constraint graph to be the root
2) **Convert** all undirected edges to directed edges pointing *away* from the root
3) **Linearize** (order the nodes such that all edges point rightwards)



4) **Enforce** arc consistency for all arcs $Parent(X_i) \to X_i$
5) **Assign** each X_i a value consistent with that of its parent

Time Complexity: $O(nd^2)$, reduced from $O(d^N)$

Cutset Conditioning
Find the smallest subset, a **cutset**, of variables such that their removal results in a tree
Time Complexity: $O(d^c(n-c)d^2)$, good for small c; backtrack up to d^c times, tree-structured CSP with $(n-c)$ variables can be solved in $O((n-c)d^2)$

Local Search
Implements the **min-conflicts heuristic**, which iteratively selects a random violated variable and reassigns its value such that its new value violates the fewest constraints; continues until no violations

9 Optimizing an Objective Function

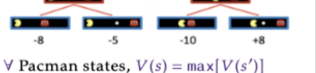
Hill-Climbing Search
Moves from the current state towards a neighboring state that increases the objective value
Complete? No. Only **restart hill-climbing** is trivially complete as at some point the randomly chosen initial state will coincide with the global maximum. **Stochastic hill-climbing** converges to a higher maxima, at the cost of more iterations
Vulnerabilities: Greedy, so can be trapped in flat areas and local maxima (points register as global maxima, but are not)

Simulated Annealing Search
Randomly moves to nearby states and uses hill-climbing to obtain a **complete** and **efficient** search algorithm. Allowed to move to states that decrease the objective, but always moves to higher objective when given the option

10 Game Trees

In **zero-sum** games, our gain is directly equivalent to out opponent's loss and vice versa (direct competition)

Minimax
Time Complexity: $O(b^m)$
Used when opponents behave optimally; optimized with $\alpha - \beta$ pruning; behaves similarly to DFS, performing a **postorder traversal** by processing all subtrees *before* the root



$\forall$ Pacman states, $V(s) = \max[V(s')]$
$\forall$ Ghost states, $V(s) = \min[V(s')]$

Week 1:

What is AI? (According to Ling)

It is the science of making machines that...

- Think like people (think rationally)
- Act like people (act rationally)

Turing Test:

- Interrogator speaks with a machine for 5 minutes through a typewriter
- Then has to guess if they are speaking with a person or machine
- If the program can successfully fool them 30% of the time, it is considered "intelligent"

Flaws with the Turing Test:

- Answers can vary depending on individual interpretation, a person could guess instead of accurately answering
- Imitating humans may not be an ideal goal for AI
- The 30% benchmark has already been surpassed, yet it is debatable if current AI is really "intelligent" as humans are

Rational: To maximally achieve predetermined goals, regardless of the thought process behind those goals

- So even if an agent is not **optimal** (giving us the least cost solution), it is still considered rational if it gives us the correct answer

Complete: If the search algorithm is guaranteed to find a solution if one exists

We can judge search algorithms based on if...

- They are *complete*
- They are *optimal*
- They are *rational*
- Time and Space complexity

Week 2: Uninformed Search

Reflex Agent: One that cannot plan ahead, it simply acts based on its current perception/memory

- They do not consider the consequences of their actions
- Often have a model of the world's current state
- Ex: A robot traversing a maze by making random turns until it reaches the end

Planning Agent: Instead asks "what if?", it makes decisions based on hypothesized consequences

- Must have a complete model of how the world responds to action
- Must formulate a goal and perform tests to see if the goal has been met
- Often cannot account for everything, so may require re-planning

Search problems generally consist of...

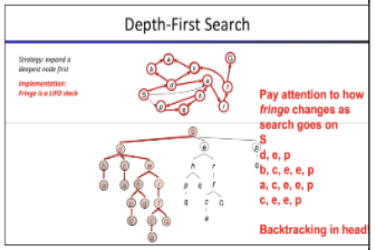
- A **State Space** is a discrete representation of the situation, where each consists of a...
  - **World state** (every detail of the world)
  - **Search state** (only the details needed for planning)
- A **successor function** that translates each state into actions and costs
- A **start state** and a **goal state**

Search Algorithm Properties:

- "**b**" is the branching factor (the number of nodes per row)
- "**m**" is the maximum depth (how deep the tree goes)
- There are b^m nodes in the entire tree

Depth-first Search: (has better space complexity)

- Always finds the leftmost solution, regardless of depth/cost
- Is not optimal
- Has a space complexity of O(b*m)
- It is complete but only if we prevent cycles (since otherwise m would be infinite)



Depth-First Search

Strategy: expand a deepest node first

Implementation: Fringe is a LIFO stack

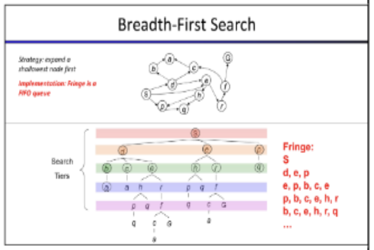
Pay attention to how *fringe* changes as search goes on

S
d, e, p
b, c, e, e, p
a, c, e, e, p
c, e, e, p

Backtracking in head!

Breadth-first Search: (has better time complexity)

- Uses level-order traversal, will search across each level until it finds the target node (so it traverses all nodes above the shallowest solution)
- "**s**" is the depth of the shallowest solution, and it only visits b^s nodes
- Space complexity is O(b^s)
- It is complete, as "s" must be finite if a solution exists
- It is optimal as long as all costs are the same (ie - we do not have a weighted graph)



Breadth-First Search

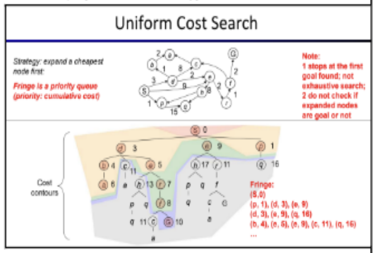
Strategy: expand a shallowest node first

Implementation: Fringe is a FIFO queue

Fringe:
S
d, e, p
e, p, b, c, e
p, b, c, e, h, r
b, c, e, h, r, q
...

Uniform Cost Search: (used to get the shortest path on weighted graphs)

- It visits all nodes with a cost <= the cheapest solution
- If the solution costs C and the actions along the way cost at least ε, then the ***effective depth*** is C/ε
- Space Complexity: O(b^(C/ε))
- Is both complete and optimal
- Downside of exploring in each direction, not affected by goal location



Uniform Cost Search

Strategy: expand a cheapest node first:

Fringe is a priority queue (priority: cumulative cost)

Note:
1 stops at the first goal found; not exhaustive search;
2 do not check if expanded nodes are goal or not

Fringe:
(S,0)
(p, 1), (d, 3), (e, 9)
(d, 3), (e, 9), (q, 16)
(b, 4), (e, 5), (e, 9), (c, 11), (q, 16)
...

Iterative Deepening: An algorithm that tries to get the advantages of both DFS and BFS

- Run a DFS with a depth-limit of 1
- If you did not find it, run a DFS with a depth-limit of 2
- If you did not find it, run a DFS with a depth-limit of 3 ...

Week 3: Informed Search + Heuristics

Search heuristic: A function that estimates how close a state is to a goal

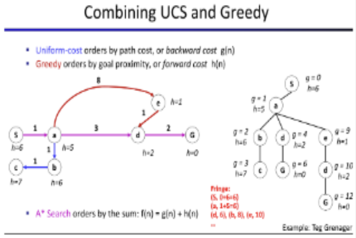
- So it is a mathematical function that takes a state as its parameter, and outputs a number (ex - f(s) = 8)
- It is designed for a particular search problem
- Ex: Manhattan distance, Euclidean distance for pathing

Greedy Search: (uses "forward cost")

- Expand the node that seems closest
  - **Based on straight-line distance**, so how far the node seems to be from the current one, regardless of how long the path actually is
- Not optimal, because it does not account for cost
  - In its worst case, greedy search is basically like DFS (going through the whole tree)
- Problem is that even if two nodes are close, a path may not exist between them

A* Search: (Sums UCS and Greedy Search)

- An algorithm that combines UCS and greedy search (greedy search is like a hare, UCS is like a tortoise)
  - UCS uses backwards cost (or "path cost") **g(n)**
  - Greedy search uses forward cost (or "node proximity") **h(n)**
- Uses a priority queue ordered by **f(n) = g(n) + h(n)**



Combining UCS and Greedy

- Uniform-cost orders by path cost, or *backward cost* g(n)
- Greedy orders by goal proximity, or *forward cost* h(n)
- A* Search orders by the sum: f(n) = g(n) + h(n)

Fringe:
(S, 0+6=6)
(a, 1+5=6)
(d, 6), (b, 8), (e, 10)
...

Example: Teg Grenager

Admissible Heuristics:

- A heuristic *h* is admissible (or "optimistic") if...
  0 <= h(n) <= h(n)*
- Where h*(n) is the **true cost** to the nearest goal
- Note that it just has to be <= the true cost, not equal, so even if h=0 it would still be considered admissible

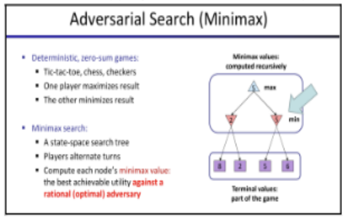
(A* is optimal only if it is admissible)

Week 4 and 5: Adversarial Search

- A **deterministic** game is one that lacks randomness
  - Specifically, it is where the next state of the game is completely determined by the action of the current state, there is no randomness involved
  - Ex: Pacman, Chess, Go, etc
- A **stochastic** game is one that involves randomness (ex - poker)
- Any deterministic game consists of...
  - States represented by "S" (ex - start state of S0)
  - Players
  - Actions (represented by "A"), may depend on player and state
  - Transition Function (maps an S and A to another S)
  - Terminal Test (maps S to a test "T" and a final goal "F")
  - Terminal Utilities (whether or not the action has any positive utility towards the goal)
- "**General games**" have agents with independent utility, so cooperation, indifference, and competition are all possible
- A **zero-sum game** is one where agents have opposite utilities, when one side gains the other loses

Adversarial search is any where you have to anticipate the moves of the opponent (making them ideal for zero-sum games)

- **Minimax** is one form of adversarial search, where each node will choose the child node that has the minimum or maximum value (depending on if the current one is a *min node* or a *max node*)



Adversarial Search (Minimax)

- Deterministic, zero-sum games:
  - Tic-tac-toe, chess, checkers
  - One player maximizes result
  - The other minimizes result
- Minimax search:
  - A state-space search tree
  - Players alternate turns
  - Compute each node's minimax value: the best achievable utility against a rational (optimal) adversary

- Minimax efficiency:
  - Just like an (exhaustive) DFS, it needs to traverse through the entire tree...
  - Time complexity: O(b^m)
  - Space complexity: O(b^M)
  - An exact solution is typically not feasible

How to deal with resource limits in minimax?

- Perform a ***depth-limited search***, where we only search until a certain depth
- Ex: Search 8 steps then estimate how far we are from the goal (using an evaluation function)
- Issue with this is that the **guarantee of optimal play is gone** (as the estimate may give errors)

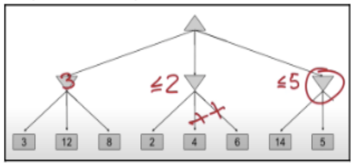
Evaluation Functions: Used to estimate cost for depth-limited search, and it is typically a weighted linear sum of features

- Ex: In Chess we can use "queen advantage", that checks if one side has more queens than the other side, summed together with knight advantage, rook advantage, etc

Dangerous optimism: Assuming chance when the world is adversarial
Dangerous pessimism: Assuming the worst case when it has unlikely chance

Alpha-Beta Pruning: Used in minimax to make traversing easier, as we know to ignore certain branches

- For example below, because it is a *min* node, we know from seeing the 2 that the value of the middle path must be <=2, and since the *max* node (the root) already saw a 3, it can exclude that middle path, though it will still need to check the rightmost one



Properties of Alpha-Beta pruning:

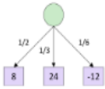
- Has no effect on the minimax value computed for the root
- Good child node ordering improves effectiveness of the pruning
- With perfect ordering...
  - Time complexity drops to O(b^(m/2))
  - Doubles solvable depth
  - Full searches (such as for chess) is still hopeless

(Continued in the right box)

Expectimax Search: Similar to minimax, except that it is used for stochastic games (where there is an element of randomness)

- It does this by assigning a probability to each branch
- Values reflect the ***average case*** (the expected probability), rather than the **worst case** (the minimum)
- When taking a decision, you take the **weighted sum** of probabilities

```
def exp-value(state):
    initialize v = 0
    for each successor of state:
        p = probability(successor)
        v += p * value(successor)
    return v
```



v = (1/2) (8) + (1/3) (24) + (1/6) (-12) = 10

(No pruning for expectimax since you don't know the probabilities of the remaining nodes)

Week 6: Probability

Product Rule: **P(y) * P(x|y) = P(x, y)**

- Rearranged gives us: **P(x|y) = P(x, y) / P(y)**

Bayes Rule: **P(x|y) = [P(y|x) / P(y)] * P(x)**

Why is Bayes Rule helpful?

- Lets us build a conditional from its reverse
- Sometimes one conditional is tricky but the other is simple

Other rules:

P(x) = P(x, y) + P(x, ~y)
P(x|y) = P(x, y) / P(y)
P(A,B|C) = P(A|C) x P(B|A,C)

General Probability Rules:

- Probability values must be >= 0
- The sum of probability values = 1

$$\forall x \;\; P(X = x) \geq 0 \;\; \text{and} \;\; \sum_x P(X = x) = 1$$

Conditional vs Joint Distributions:

Joint Distribution (given)

$P(T, W)$

T	W	P
hot	sun	0.4
hot	rain	0.1
cold	sun	0.2
cold	rain	0.3

Conditional Distributions

$P(W|T)$

$P(W|T = hot)$

W	P
sun	0.8
rain	0.2

$P(W|T = cold)$

W	P
sun	0.4
rain	0.6

Week 7: Bayesian Networks

Independence: Say we have a distribution over 2 variables (X and Y), these two are independent if...

P(x,y) = P(x) * P(y)

Another way of saying it is...

P(x|y) = P(x)

You can check independence by doing the following:

$P(T)$

T	P
hot	0.5
cold	0.5

$P_1(T, W)$

T	W	P
hot	sun	0.4
hot	rain	0.1
cold	sun	0.2
cold	rain	0.3

$P_2(T, W)$

T	W	P
hot	sun	0.3
hot	rain	0.2
cold	sun	0.3
cold	rain	0.2

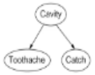
$P(W)$

W	P
sun	0.6
rain	0.4

- So, we split the joint probability table P1 into two tables (one for each variable)
- Then we multiply them together to create a new table, P2
- If P1 = P2, then we know the two variables are independent

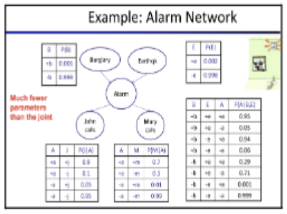
Bayes Nets: A technique for describing complex joint distributions using simple, local distributions (conditional probabilities)

- Also called "graphical models"
- Describes how variables interact locally
- Ex: Below is one representing a relation between Cavity, toothache and catch...



Bayes Nets Semantics:

- Nodes must be connected in a **directed acyclic graph** (so there should be no cycles)
- Ex: The one below lacks a cycle...



Example: Alarm Network

Much fewer parameters than the joint

B	P(B)
+b	0.001
-b	0.999

E	P(E)
+e	0.002
-e	0.998

A	J	P(J|A)
+a	+j	0.9
+a	-j	0.1
-a	+j	0.05
-a	-j	0.95

A	M	P(M|A)
+a	+m	0.7
+a	-m	0.3
-a	+m	0.01
-a	-m	0.99

B	E	A	P(A|B,E)
+b	+e	+a	0.95
+b	+e	-a	0.05
+b	-e	+a	0.94
+b	-e	-a	0.06
-b	+e	+a	0.29
-b	+e	-a	0.71
-b	-e	+a	0.001
-b	-e	-a	0.999

Final Weeks:

Difference between supervised, self-supervised and unsupervised learning...

Supervised learning:

- Uses a labelled (known) dataset, and maps each input to a known output
- Ex: You could have a set of images of animals, with each picture labeled
- Good for regression or classification problems
- **Decision trees**, **K-NN** (K-nearest neighbor) and **Neural Networks** would all be examples of supervised learning

Unsupervised learning:

- Given an unlabelled dataset, it must find patterns or relationships on its own
- Common example would be clustering (includes K-means and agglomerative)

Self-supervised learning:

- A specific type of unsupervised learning, it uses unlabelled data, discovers patterns, and then uses that to interpret the data
  - Another way of thinking about this is that "the learning task is generated from the data itself"
- Ex: Given some video, images, text, etc, it could be used to predict some "missing pieces"
- A modern example would be using neural networks to perform Natural Language Processing (NLP), where it predicts the next word in a sentence

Reinforcement learning: (uses rewards and punishments for behavior)

- Where an agent learns to make decisions by interacting with the environment, rather than being based on some initial dataset (as in supervised and unsupervised learning)
- Similar to unsupervised learning, it does not need any supervision
- It is usually not applicable to the real world, as the model needs to generate its own data

Lazy vs Eager learning approaches: (mainly only applies to supervised models)

Eager Learning algorithms: Also known as "model based learning", since they make a generalized model of the data in the "training phase"

- Examples of this would be ***Decision Trees*** and ***Neural Networks***
- Decision trees and neural networks both create their respective models during the training phase, but afterwards new predictions can be made without going back to the original training data
- So they are **slow at training** (being created), but **fast at testing**

Lazy Learning algorithms: Also known as "memory based learning", they make use of functions rather than doing explicit learning (so the "learning" is just memorizing)

- An example of this would be ***K-NN***
- Delay learning until provided with a specific test example

Immediate Feedback vs Delayed Feedback:

Supervised Learning involves "**immediate feedback**", meaning the model learns immediately if its prediction was correct or not

- For example, with a decision trees, after having the tree, we immediately know how to make decisions
- K-NN, despite being a "lazy learner", still has immediate feedback, as immediately after we perform the comparison function calculations, we know the result for the Kth neighbor

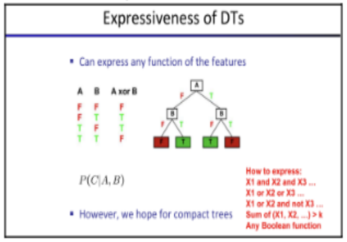
Reinforcement Learning involves "**delayed feedback**", meaning there is a delay between when the model learns and when we see the results

- The agent first needs to take actions in its environment before assessing if they were good or bad

The different algorithms we learned:

Decision Trees:

- An example of **supervised** learning, it is a tree-like structure where each internal node is a "test" on an attribute, so that it can be used to map attributes to conclusions
  - So it is a representation of a decision function, and every tree can be encoded as a truth table
  - Ex: Below is a tree encoding a XOR function...



Expressiveness of DTs

- Can express any function of the features

A	B	A xor B
F	F	F
F	T	T
T	F	T
T	T	F

$P(C|A, B)$

- However, we hope for compact trees

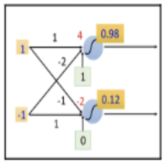
How to express:
X1 and X2 and X3 ...
X1 or X2 or X3 ...
X1 or X2 and not X3 ...
Sum of (X1, X2, ...) > k
Any Boolean function

Neural Networks:

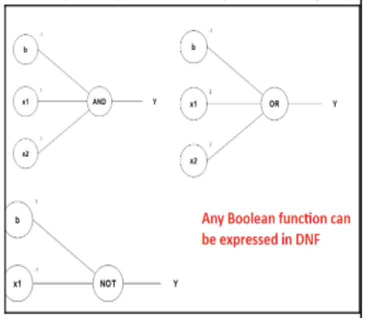
- Can be used for both **supervised** and **self-supervised** learning
- They are capable of forming complex patterns of data, making them useful for a wide range of tasks like image recognition, NLP, etc

Deep Neural Networks:

- In addition to nodes and weighted branches, these include "biases"
- For example, below is one that uses a "sigmoid function", which for any positive input provides a value rounded to 1, and for any negative input a value rounded to 0...



(Biases are the squares with 1 and 0, the "hidden layers" are the values containing 4 and -2)



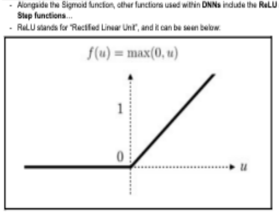
Any Boolean function can be expressed in DNF

Decision Trees vs Neural Networks:

- Decision trees work well on labeled data, whereas neural networks can be used for unstructured data as well
- Decision trees are easier for humans to read through and interpret, whereas neural networks can be more complex
- Neural networks are better for accomplishing multiple tasks
- Decision trees are better for simple classification tasks

(Continuation from left side)

- Alongside the Sigmoid function, other functions used within **DNNs** include the **ReLU** and **Step functions**...
- ReLU stands for "Rectified Linear Unit", and it can be seen below:



(So between (0, u), it returns *u* if it is positive, otherwise it returns 0)

- A "step function" is any function of the following form:

$$f(x) = \{ \begin{matrix} a & \text{if } x < c \\ b & \text{if } x \geq c \end{matrix}$$

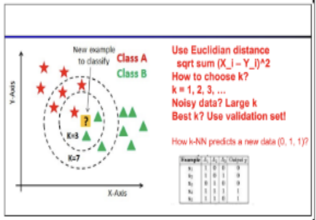
(So it could be used to assign the output to be 0 or 1 exactly)

What are the advantages and disadvantages of DNN (Deep Neural Networks)?

- Advantages include...
  - It can be used to represent any boolean function
  - It can handle large amounts of data
  - They are good at things like image recognition and NLP
- Disadvantages include...
  - They require a large amount of training data
  - They can be hard to design
  - They can be "black boxes", where we have a hard time understanding why they make a certain prediction

K-NN (K-Nearest Neighbour, also known as "case-based learning"):

- An example of **supervised** learning used for classification and regression
- It is "**non-parametric**", meaning it doesn't make any assumptions about the underlying data
- It uses the euclidean distance in order to find the Kth nearest neighbor of a value



Use Euclidian distance
sqrt sum (X_i – Y_i)^2
How to choose k?
k = 1, 2, 3, ...
Noisy data? Large k
Best k? Use validation set!

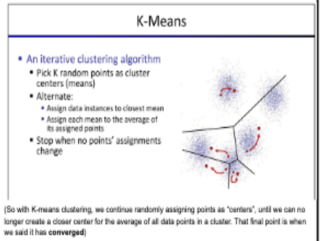
How k-NN predicts a new data (0, 1, 1)?

Clustering:

- An example of **unsupervised** learning, as the aim is to detect patterns in unlabelled data
- Useful when we do not know what we are looking for (reveals general patterns)
- General idea is to group together similar instances
  - One way of doing this could be using small (squared) Euclidean distance:

$$\text{dist}(x, y) = (x - y)^{\mathsf{T}}(x - y) = \sum_i (x_i - y_i)^2$$

Another form of clustering is **K-Means Clustering**...



K-Means

- An iterative clustering algorithm
  - Pick K random points as cluster centers (means)
  - Alternate:
    - Assign data instances to closest mean
    - Assign each mean to the average of its assigned points
  - Stop when no points' assignments change

(So with K-means clustering, we continue randomly assigning points as "centers", until we can no longer create a closer center for the average of all data points in a cluster. That final point is when we said it has **converged**)

- K-means is **non-deterministic**, meaning it involves a degree of randomness, as even with the same set of data points, we could get different clusters depending on the centers chosen

Will K-means converge? Yes

- Will K-means converge to a global optimum? Not necessarily

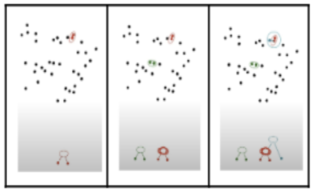
Will K-means always find true patterns in the data? Not necessarily

- Nonetheless, people do use this algorithm to find interesting patterns

The final form of clustering we learn is **Agglomerative Clustering**...

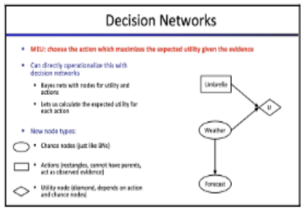
- It's an iterative algorithm where we first merge very similar instances (ex - the two points closest together)
- Then we repeatedly build larger clusters out of smaller clusters (ex - merging a cluster with its closest cluster or datapoint)
- Stops when there is only one cluster left

Agglomerative clustering can be represented as a tree called a **dendrogram**, increasing from the leaf-nodes upwards...



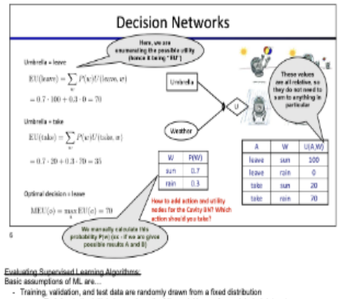
K-NN vs K-Means - Don't Mix these up!

- K-NN is an example of supervised learning, it is ideal for labeled datasets
- K-means is a form of clustering, so it is unsupervised and good for unlabelled data
- Both of these algorithms can use Euclidean distance, but for different reasons:
  - beled dataset, and you want to classify a new datapoint according to it (by finding its nearest neighbor)
  - K-means: When we have unlabelled data and want to organize that data into clusters



Decision Networks

- MEU: choose the action which maximizes the expected utility given the evidence
- Can directly operationalize this with decision networks
  - Bayes nets with nodes for utility and actions
  - Lets us calculate the expected utility for each action
- New node types:
  - Chance nodes (just like BNs)
  - Actions (rectangles, cannot have parents, act as observed evidence)
  - Utility node (diamond, depends on action and chance nodes)



Decision Networks

Umbrella = leave

$$\text{EU}(\text{leave}) = \sum_w P(w)U(\text{leave}, w)$$

$$= 0.7 \cdot 100 + 0.3 \cdot 0 = 70$$

Umbrella = take

$$\text{EU}(\text{take}) = \sum_w P(w)U(\text{take}, w)$$

$$= 0.7 \cdot 20 + 0.3 \cdot 70 = 35$$

Optimal decision = leave

$$\text{MEU}(\phi) = \max_a \text{EU}(a) = 70$$

Here, we are enumerating the possible utility (hence it being "EU")

These values are all relative, so they do not need to sum to anything in particular

We manually calculate this probability P(w) (ex - if we are given possible results A and B)

W	P(W)
sun	0.7
rain	0.3

A	W	U(A,W)
leave	sun	100
leave	rain	0
take	sun	20
take	rain	70

How to add action and utility nodes for the Cavity BN? Which action should you take?

6

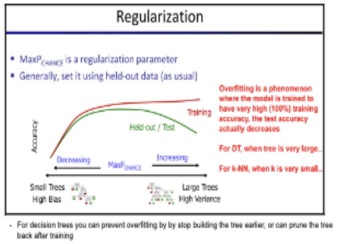
Evaluating Supervised Learning Algorithms:

Basic assumptions of ML are...

- Training, validation, and test data are randomly drawn from a fixed distribution
  - **Training** is the initial phase where the ML-model learns from a labeled dataset
  - **Testing** (or "evaluation") occurs after training, where the model's performance needs to be tested on a separate set of data (which it has not already seen during training)
  - A model that performs well on the test data is assumed to be likely to work well for new real-world data
- Most theories of ML are **PAC**-based (Probably Approximately Correct), meaning when training size is large enough, the test-accuracy will be accurate enough with a high probability
- **Cross Validation:** A key step in finding well-performing models, where you split the whole training set into k-folds randomly, where one fold is used for testing and the rest for training. Repeat this "k" times, and take the average testing performance
- ML algorithms may have "hyper parameters", and cross-validation for this is a bit different: Choose a parameter to evaluate, and from the k-folds, have 1 for validation (of the parameter), another for testing, and the rest for training
  - **Hyper Parameters** are any decisions you make before the training process begins, so they are not learned by the training data but pre-assigned

Overfitting:

- When you stop modeling the patterns in the training data and start modeling noise
- Prevents the model from generalizing well to new data sets
- One common solution is **regularization**, which means constraining the model during training so it has a hard time learning the noise (example below)



Regularization

- $\text{MaxP}_{\text{CHANCE}}$ is a regularization parameter
- Generally, set it using held-out data (as usual)

Overfitting is a phenomenon where the model is trained to have very high (100%) training accuracy, the test accuracy actually decreases

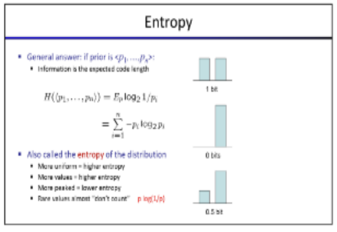
For DT, when tree is very large...

For k-NN, when k is very small...

- For decision trees you can prevent overfitting by by stop building the tree earlier, or can prune the tree back after training

Entropy:

- Measures the level of disorder/uncertainty
- Because it measures uncertainty, a more peaked distribution would have lower entropy because it is certain which value it will be...



Entropy

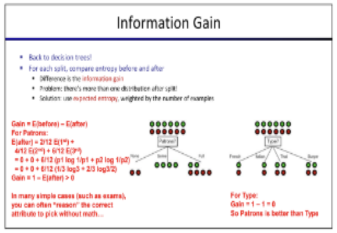
- General answer: if prior is $<p_1, \ldots, p_n>$:
  - Information is the expected code length

$$H(\langle p_1, \ldots, p_n \rangle) = E_p \log_2 1/p_i$$

$$= \sum_{i=1}^{n} -p_i \log_2 p_i$$

- Also called the entropy of the distribution
  - More uniform = higher entropy
  - More values = higher entropy
  - More peaked = lower entropy
  - Rare values almost "don't count" p log(1/p)

- Used to determine the quality of splits for decision trees
  - Ideally, a good attribute would split the tree into subsets that are "all positive" or "all negative", but since this is often not possible we use entropy instead
- This relates to decision trees, as if the children of a node are all homogeneous (belong to the same class), it would mean it has low entropy



Information Gain

- Back to decision trees!
- For each split, compare entropy before and after
  - Difference is the information gain
  - Problem: there's more than one distribution after split!
  - Solution: use expected entropy, weighted by the number of examples

Gain = E(before) – E(after)
For Patrons:
E(after) = 2/12 E(1st) +
4/12 E(2nd) + 6/12 E(3rd)
= 0 + 0 + 6/12 (p1 log 1/p1 + p2 log 1/p2)
= 0 + 0 + 6/12 (1/3 log3 + 2/3 log3/2)
Gain = 1 – E(after) > 0

In many simple cases (such as exams), you can often "reason" the correct attribute to pick without math...

For Type:
Gain = 1 – 1 = 0
So Patrons is better than Type