CS 188 Introduction to Artificial Intelligence
Fall 2018 Note 1

These lecture notes are heavily based on notes originally written by Nikhil Sharma.

Agents

In artificial intelligence, the central problem at hand is that of the creation of a rational **agent**, an entity that has goals or preferences and tries to perform a series of **actions** that yield the best/optimal expected outcome given these goals. Rational agents exist in an **environment**, which is specific to the given instantiation of the agent. As a very simple example, the environment for a checkers agent is the virtual checkers board on which it plays against opponents, where piece moves are actions. Together, an environment and the agents that reside within it create a **world**.

A **reflex agent** is one that doesn't think about the consequences of its actions, but rather selects an action based solely on the current state of the world. These agents are typically outperformed by **planning agents**, which maintain a model of the world and use this model to simulate performing various actions. Then, the agent can determine hypothesized consequences of the actions and can select the best one. This is simulated "intelligence" in the sense that it's exactly what humans do when trying to determine the best possible move in any situation - thinking ahead.

State Spaces and Search Problems

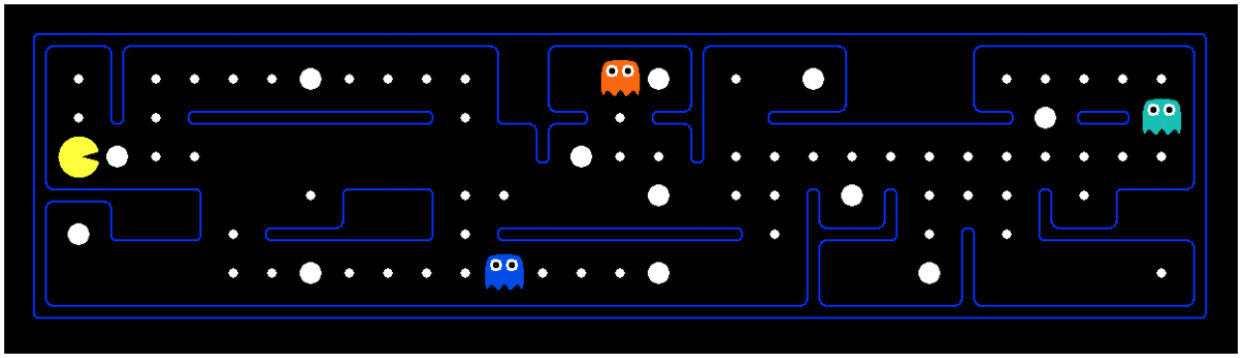
In order to create a rational planning agent, we need a way to mathematically express the given environment in which the agent will exist. To do this, we must formally express a **search problem** - given our agent's current **state** (its configuration within its environment), how can we arrive at a new state that satisfies its goals in the best possible way? Formulating such a problem requires four things:

- A **state space** - The set of all possible states that are possible in your given world
- A **successor function** - A function that takes in a state and an action and computes the cost of performing that action as well as the **successor state**, the state the world would be in if the given agent performed that action
- A **start state** - The state in which an agent exists initially
- A **goal test** - A function that takes a state as input, and determines whether it is a goal state

Fundamentally, a search problem is solved by first considering the start state, then exploring the state space using the successor function, iteratively computing successors of various states until we arrive at a goal state, at which point we will have determined a path from the start state to the goal state (typically called a **plan**). The order in which states are considered is determined using a predetermined **strategy**. We'll cover types of strategies and their usefulness shortly.

Before we continue with how to solve search problems, it's important to note the difference between a **world state**, and a **search state**. A world state contains all information about a given state, whereas a search state

contains only the information about the world that's necessary for planning (primarily for space effiency reasons). To illustrate these concepts, we'll introduce the hallmark motivating example of this course - Pacman. The game of Pacman is simple: Pacman must navigate a maze and eat all the (small) food pellets in the maze without being eaten by the malicious patrolling ghosts. If Pacman eats one of the (large) power pellets, he becomes ghost-immune for a set period of time and gains the ability to eat ghosts for points.



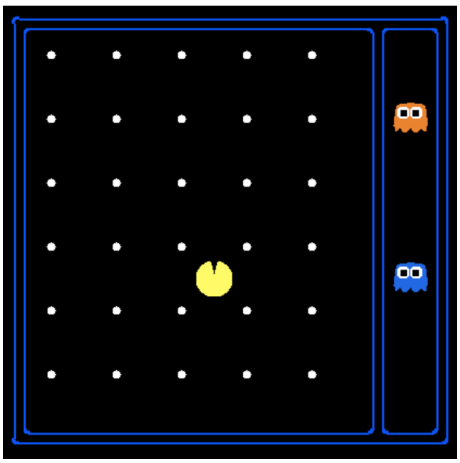
Let's consider a variation of the game in which the maze contains only Pacman and food pellets. We can pose two distinct search problems in this scenario: pathing and eat-all-dots. Pathing attempts to solve the problem of getting from position (x_1, y_1) to position (x_2, y_2) in the maze optimally, while eat all dots attempts to solve the problem of consuming all food pellets in the maze in the shortest time possible. Below, the states, actions, successor function, and goal test for both problems are listed:

- **Pathing**
  - States: (x,y) locations
  - Actions: North, South, East, West
  - Successor: Update location only
  - Goal test: Is (x,y)=END?
- **Eat-all-dots**
  - States: (x,y) location, dot booleans
  - Actions: North, South, East, West
  - Successor: Update location and booleans
  - Goal test: Are all dot booleans false?

Note that for pathing, states contain less information than states for eat-all-dots, because for eat-all-dots we must maintain an array of booleans corresponding to each food pellet and whether or not it's been eaten in the given state. A world state may contain more information still, potentially encoding information about things like total distance traveled by Pacman or all positions visited by Pacman on top of its current (x,y) location and dot booleans.

State Space Size

An important question that often comes up while estimating the computational runtime of solving a search problem is the size of the state space. This is done almost exclusively with the **fundamental counting principle**, which states that if there are n variable objects in a given world which can take on x_1, x_2, ..., x_n different values respectively, then the total number of states is $x_1 \cdot x_2 \cdot \ldots \cdot x_n$. Let's use Pacman to show this concept by example:



Let's say that the variable objects and their corresponding number of possiblilites are as follows:

- *Pacman positions* - Pacman can be in 120 distinct (x, y) positions, and there is only one Pacman
- *Pacman Direction* - this can be North, South, East, or West, for a total of 4 possibilities
- *Ghost positions* - There are two ghosts, each of which can be in 12 distinct (x, y) positions
- *Food pellet configurations* - There are 30 food pellets, each of which can be eaten or not eaten

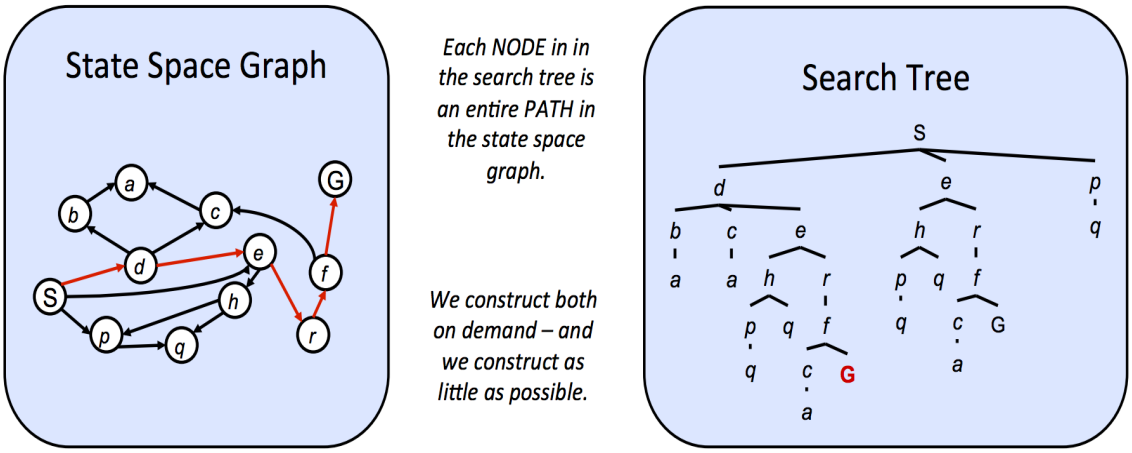
Using the fundamental counting principle, we have 120 positions for Pacman, 4 directions Pacman can be facing, $12 \cdot 12$ ghost configurations (12 for each ghost), and $2 \cdot 2 \cdot \ldots \cdot 2 = 2^{30}$ food pellet configurations (each of 30 food pellets has two possible values - eaten or not eaten). This gives us a total state space size of $\boxed{120 \cdot 4 \cdot 12^2 \cdot 2^{30}}$.

State Space Graphs and Search Trees

Now that we've established the idea of a state space and the four components necessary to completely define one, we're almost ready to begin solving search problems. The final piece of the puzzle is that of state space graphs and search trees.

Recall that a graph is defined by a set of nodes and a set of edges connecting various pairs of nodes. These edges may also have weights associated with them. A **state space graph** is constructed with states representing nodes, with directed edges existing from a state to its successors. These edges represent actions, and any associated weights represent the cost of performing the corresponding action. Typically, state space graphs are much too large to store in memory (even our simple Pacman example from above has $\approx 10^{13}$ possible states, yikes!), but they're good to keep in mind conceptually while solving problems. It's also important to note that in a state space graph, each state is represented exactly once - there's simply no need to represent a state multiple times, and knowing this helps quite a bit when trying to reason about search problems.

Unlike state space graphs, our next structure of interest, **search trees**, have no such restriction on the number of times a state can appear. This is because though search trees are also a class of graph with states as nodes and actions as edges between states, each state/node encodes not just the state itself, but the entire path (or **plan**) from the start state to the given state in the state space graph. Observe the state space graph and corresponding search tree below:



The highlighted path (S $\rightarrow$ d $\rightarrow$ e $\rightarrow$ r $\rightarrow$ f $\rightarrow$ G) in the given state space graph is represented in the corresponding search tree by following the path in the tree from the start state S to the highlighted goal state G. Similarly, each and every path from the start node to any other node is represented in the search tree by a path from the root S to some descendant of the root corresponding to the other node. Since there often exist multiple ways to get from one state to another, states tend to show up multiple times in search trees. As a result, search trees are greater than or equal to their corresponding state space graph in size.

We've already determined that state space graphs themselves can be enormous in size even for simple problems, and so the question arises - how can we perform useful computation on these structures if they're too big to represent in memory? The answer lies in successor functions - we only store states we're immediately working with, and compute new ones on-demand using the corresponding successor function. Typically, search problems are solved using search trees, where we very carefully store a select few nodes to observe at a time, iteratively replacing nodes with their successors until we arrive at a goal state. There exist various methods by which to decide the order in which to conduct this iterative replacement of search tree nodes, and we'll present these methods now.

Uninformed Search

The standard protocol for finding a plan to get from the start state to a goal state is to maintain an outer **fringe** of partial plans derived from the search tree. We continually **expand** our fringe by removing a node (which is selected using our given **strategy**) corresponding to a partial plan from the fringe, and replacing it on the fringe with all its children. Removing and replacing an element on the fringe with its children corresponds to discarding a single length n plan and bringing all length $(n+1)$ plans that stem from it into consideration. We continue this until eventually removing a goal state off the fringe, at which point we conclude the partial plan corresponding to the removed goal state is in fact a path to get from the start state to the goal state. Practically, most implementations of such algorithms will encode information about the parent node, distance to node, and the state inside the node object. This procedure we have just outlined is known as **tree search**, and the pseudocode for it is presented below:

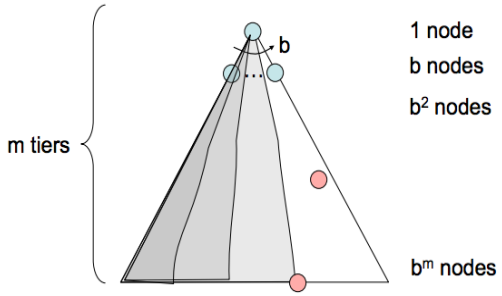
```
function TREE-SEARCH(problem, fringe) return a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE[node]) then return node
        for child-node in EXPAND(STATE[node], problem) do
            fringe ← INSERT(child-node, fringe)
        end
    end
```

When we have no knowledge of the location of goal states in our search tree, we are forced to select our strategy for tree search from one of the techniques that falls under the umbrella of **uninformed search**. We'll now cover three such strategies in succession: **depth-first search**, **breadth-first search**, and **uniform cost search**. Along with each strategy, some rudimentary properties of the strategy are presented as well, in terms of the following:

- The **completeness** of each search strategy - if there exists a solution to the search problem, is the strategy guaranteed to find it given infinite computational resources?
- The **optimality** of each search strategy - is the strategy guaranteed to find the lowest cost path to a goal state?
- The **branching factor** b - The increase in the number of nodes on the fringe each time a fringe node is dequeued and replaced with its children is $O(b)$. At depth k in the search tree, there exists $O(b^k)$ nodes.
- The maximum depth m.
- The depth of the shallowest solution s.

Depth-First Search

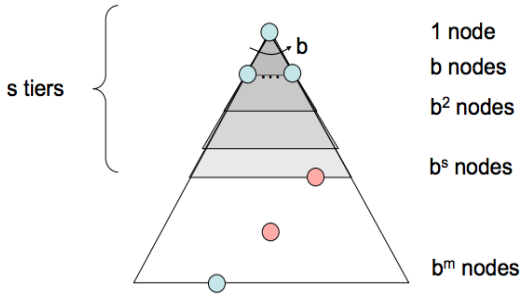
- *Description* - Depth-first search (DFS) is a strategy for exploration that always selects the *deepest* fringe node from the start node for expansion.
- *Fringe representation* - Removing the deepest node and replacing it on the fringe with its children necessarily means the children are now the new deepest nodes - their depth is one greater than the depth of the previous deepest node. This implies that to implement DFS, we require a structure that always gives the most recently added objects highest priority. A last-in, first-out (LIFO) stack does exactly this, and is what is traditionally used to represent the fringe when implementing DFS.



- *Completeness* - Depth-first search is not complete. If there exist cycles in the state space graph, this inevitably means that the corresponding search tree will be infinite in depth. Hence, there exists the possibility that DFS will faithfully yet tragically get "stuck" searching for the deepest node in an infinite-sized search tree, doomed to never find a solution.
- *Optimality* - Depth-first search simply finds the "leftmost" solution in the search tree without regard for path costs, and so is not optimal.
- *Time Complexity* - In the worst case, depth first search may end up exploring the entire search tree. Hence, given a tree with maximum depth m, the runtime of DFS is $O(b^m)$.
- *Space Complexity* - In the worst case, DFS maintains b nodes at each of m depth levels on the fringe. This is a simple consequence of the fact that once b children of some parent are enqueued, the nature of DFS allows only one of the subtrees of any of these children to be explored at any given point in time. Hence, the space complexity of BFS is $O(bm)$.

Breadth-First Search

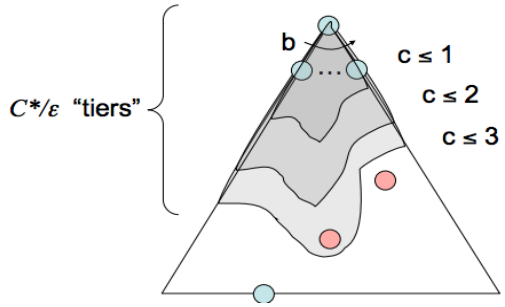
- *Description* - Breadth-first search is a strategy for exploration that always selects the *shallowest* fringe node from the start node for expansion.
- *Fringe representation* - If we want to visit shallower nodes before deeper nodes, we must visit nodes in their order of insertion. Hence, we desire a structure that outputs the oldest enqueued object to represent our fringe. For this, BFS uses a first-in, first-out (FIFO) queue, which does exactly this.



- *Completeness* - If a solution exists, then the depth of the shallowest node s must be finite, so BFS must eventually search this depth. Hence, it's complete.
- *Optimality* - BFS is generally not optimal because it simply does not take costs into consideration when determining which node to replace on the fringe. The special case where BFS is guaranteed to be optimal is if all edge costs are equivalent, because this reduces BFS to a special case of uniform cost search, which is discussed below.
- *Time Complexity* - We must search $1 + b + b^2 + ... + b^s$ nodes in the worst case, since we go through all nodes at every depth from 1 to s. Hence, the time complexity is $O(b^s)$.
- *Space Complexity* - The fringe, in the worst case, contains all the nodes in the level corresponding to the shallowest solution. Since the shallowest solution is located at depth s, there are $O(b^s)$ nodes at this depth.

Uniform Cost Search

- *Description* - Uniform cost search (UCS), our last strategy, is a strategy for exploration that always selects the *lowest cost* fringe node from the start node for expansion.
- *Fringe representation* - To represent the fringe for UCS, the choice is usually a heap-based priority queue, where the weight for a given enqueued node v is the path cost from the start node to v, or the *backward cost* of v. Intuitively, a priority queue constructed in this manner simply reshuffles itself to maintain the desired ordering by path cost as we remove the current minimum cost path and replace it with its children.



- *Completeness* - Uniform cost search is complete. If a goal state exists, it must have some finite length shortest path; hence, UCS must eventually find this shortest length path.
- *Optimality* - UCS is also optimal if we assume all edge costs are nonnegative. By construction, since we explore nodes in order of increasing path cost, we're guaranteed to find the lowest-cost path to a goal state. The strategy employed in Uniform Cost Search is identical to that of Dijkstra's algorithm, and the chief difference is that UCS terminates upon finding a solution state instead of finding the shortest path to all states. Note that having negative edge costs in our graph can make nodes on a path have decreasing length, ruining our guarantee of optimality. (See Bellman-Ford algorithm for a slower algorithm that handles this possibility)
- *Time Complexity* - Let us define the optimal path cost as C^* and the minimal cost between two nodes in the state space graph as ε. Then, we must roughly explore all nodes at depths ranging from 1 to C^*/ε, leading to an runtime of $O(b^{C^*/\varepsilon})$.
- *Space Complexity* - Roughly, the fringe will contain all nodes at the level of the cheapest solution, so the space complexity of UCS is estimated as $O(b^{C^*/\varepsilon})$.

As a parting note about uninformed search, it's critical to note that the three strategies outlined above are fundamentally the same - differing only in expansion strategy, with their similarities being captured by the tree search pseudocode presented above.

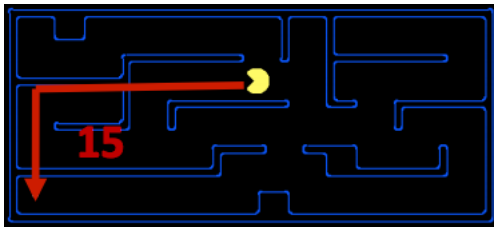
Informed Search

Uniform cost search is good because it's both complete and optimal, but it can be fairly slow because it expands in every direction from the start state while searching for a goal. If we have some notion of the direction in which we should focus our search, we can significantly improve performance and "hone in" on a goal much more quickly. This is exactly the focus of **informed search**.

Heuristics

Heuristics are the driving force that allow estimation of distance to goal states - they're functions that take in a state as input and output a corresponding estimate. The computation performed by such a function is specific to the search problem being solved. For reasons that we'll see in A* search, below, we usually want heuristic functions to be a lower bound on this remaining distance to the goal, and so heuristics are typically solutions to **relaxed problems** (where some of the constraints of the original problem have been removed). Turning to our Pacman example, let's consider the pathing problem described earlier. A common heuristic that's used to solve this problem is the **Manhattan distance**, which for two points (x_1, y_1) and (x_2, y_2) is defined as follows:

$$Manhattan(x_1, y_1, x_2, y_2) = |x_1 - x_2| + |y_1 - y_2|$$



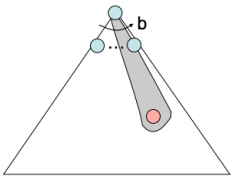
The above visualization shows the relaxed problem that the Manhattan distance helps solve - assuming Pacman desires to get to the bottom left corner of the maze, it computes the distance from Pacman's current location to Pacman's desired location *assuming a lack of walls in the maze*. This distance is the *exact* goal distance in the relaxed search problem, and correspondingly is the *estimated* goal distance in the actual search problem. With heuristics, it becomes very easy to implement logic in our agent that enables them to "prefer" expanding states that are estimated to be closer to goal states when deciding which action to perform. This concept of preference is very powerful, and is utilized by the following two search algorithms that implement heuristic functions: greedy search and A*.

Greedy Search

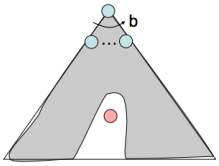
- *Description* - Greedy search is a strategy for exploration that always selects the fringe node with the *lowest heuristic value* for expansion, which corresponds to the state it believes is nearest to a goal.
- *Fringe representation* - Greedy search operates identically to UCS, with a priority queue fringe representation. The difference is that instead of using *computed backward cost* (the sum of edge weights in the path to the state) to assign priority, greedy search uses *estimated forward cost* in the form of heuristic values.
- *Completeness and Optimality* - Greedy search is not guaranteed to find a goal state if one exists, nor is it optimal, particularly in cases where a very bad heuristic function is selected. It generally acts fairly unpredictably from scenario to scenario, and can range from going straight to a goal state to acting like a badly-guided DFS and exploring all the wrong areas.

A* Search

- *Description* - A* search is a strategy for exploration that always selects the fringe node with the *lowest estimated total cost* for expansion, where total cost is the entire cost from the start node to the goal node.



(a) Greedy search on a good day :)



(b) Greedy search on a bad day :(

- *Fringe representation* - Just like greedy search and UCS, A* search also uses a priority queue to represent its fringe. Again, the only difference is the method of priority selection. A* combines the total backward cost (sum of edge weights in the path to the state) used by UCS with the estimated forward cost (heuristic value) used by greedy search by adding these two values, effectively yielding an *estimated total cost* from start to goal. Given that we want to minimize the total cost from start to goal, this is an excellent choice.
- *Completeness and Optimality* - A* search is both complete and optimal, given an appropriate heuristic (which we'll cover in a minute). It's a combination of the good from all the other search strategies we've covered so far, incorporating the generally high speed of greedy search with the optimality and completeness of UCS!

Admissibility and Consistency

Now that we've discussed heuristics and how they are applied in both greedy and A* search, let's spend some time discussing what constitutes a good heuristic. To do so, let's first reformulate the methods used for determining priority queue ordering in UCS, greedy search, and A* with the following definitions:

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

Before attacking the question of what constitutes a "good" heuristic, we must first answer the question of whether A* maintains its properties of completeness and optimality regardless of the heuristic function we use. Indeed, it's very easy to find heuristics that break these two coveted properties. As an example, consider the heuristic function $h(n) = 1 - g(n)$. Regardless of the search problem, using this heuristic yields

$$\begin{aligned} f(n) &= g(n) + h(n) \\ &= g(n) + (1 - g(n)) \\ &= 1 \end{aligned}$$

Hence, such a heuristic reduces A* search to BFS, where all edge costs are equivalent. As we've already shown, BFS is not guaranteed to be optimal in the general case where edge weights are not constant.

The condition required for optimality when using A* tree search is known as **admissibility**. The admissibility constraint states that the value estimated by an admissible heuristic is neither negative nor an overestimate. Defining $h^*(n)$ as the true optimal forward cost to reach a goal state from a given node n, we can formulate the admissibility constraint mathematically as follows:

$$\forall n,\ 0 \leq h(n) \leq h^*(n)$$

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.

Proof. Assume two reachable goal states are located in the search tree for a given search problem, an optimal goal A and a suboptimal goal B. Some ancestor n of A (including perhaps A itself) must currently be on the fringe, since A is reachable from the start state. We claim n will be selected for expansion before B, using the following three statements:

1. $g(A) < g(B)$. Because A is given to be optimal and B is given to be suboptimal, we can conclude that A has a lower backwards cost to the start state than B.
2. $h(A) = h(B) = 0$, because we are given that our heuristic satisfies the admissibility constraint. Since both A and B are both goal states, the true optimal cost to a goal state from A or B is simply $h^*(n) = 0$; hence $0 \leq h(n) \leq 0$.
3. $f(n) \leq f(A)$, because, through admissibility of h, $f(n) = g(n) + h(n) \leq g(n) + h^*(n) = g(A) = f(A)$. The total cost through node n is at most the true backward cost of A, which is also the total cost of A.

We can combine statements 1. and 2. to conclude that $f(A) < f(B)$ as follows:

$$f(A) = g(A) + h(A) = g(A) < g(B) = g(B) + h(B) = f(B)$$

A simple consequence of combining the above derived inequality with statement 3. is the following:

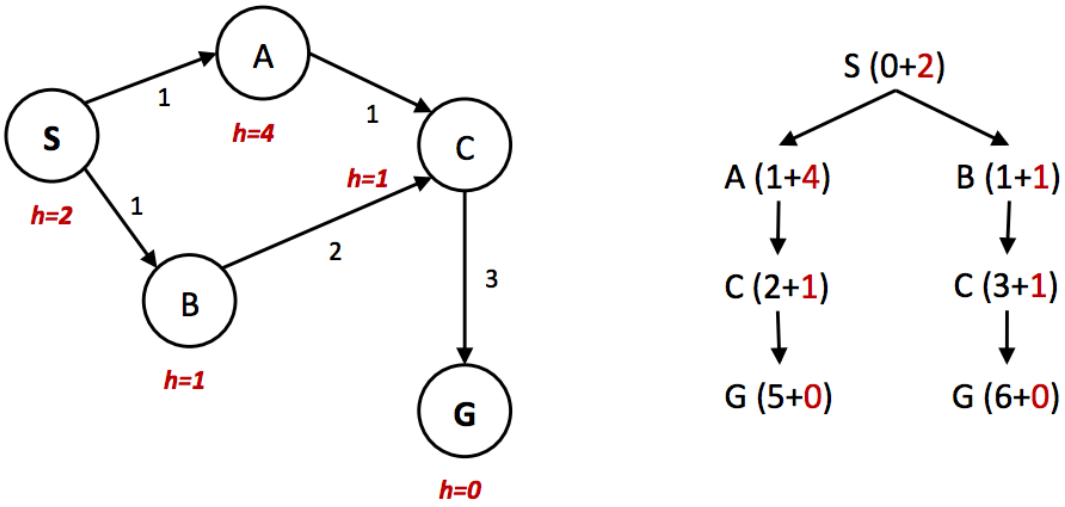
$$f(n) \leq f(A) \wedge f(A) < f(B) \Longrightarrow f(n) < f(B)$$

Hence, we can conclude that n is expanded before B. Because we have proven this for arbitrary n, we can conclude that *all* ancestors of A (including A itself) expand before B. □

One problem we found above with tree search was that in some cases it could fail to ever find a solution, getting stuck searching the same cycle in the state space graph infinitely. Even in situations where our search technique doesn't involve such an infinite loop, it's often the case that we revisit the same node multiple times because there's multiple ways to get to that same node. This leads to exponentially more work, and the natural solution is to simply keep track of which states you've already expanded, and never expand them again. More explicitly, maintain a "closed" set of expanded nodes while utilizing your search method of choice. Then, ensure that each node isn't already in the set before expansion and add it to the set after expansion if it's not. Tree search with this added optimization is known as **graph search**, and the pseudocode for it is presented below:

```
function GRAPH-SEARCH(problem, fringe) return a solution, or failure
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE[node]) then return node
        if STATE[node] is not in closed then
            add STATE[node] to closed
            for child-node in EXPAND(STATE[node], problem) do
                fringe ← INSERT(child-node, fringe)
            end
    end
```

Note that in implementation, it's critically important to store the closed set as a disjoint set and not a list. Storing it as a list requires costs $O(n)$ operations to check for membership, which eliminates the performance improvement graph search is intended to provide. An additional caveat of graph search is that it tends to ruin the optimality of A*, even under admissible heuristics. Consider the following simple state space graph and corresponding search tree, annotated with weights and heuristic values:



In the above example, it's clear that the optimal route is to follow $S \to A \to C \to G$, yielding a total path cost of $1+1+3=5$. The only other path to the goal, $S \to B \to C \to G$ has a path cost of $1+2+3=6$. However, because the heuristic value of node A is so much larger than the heuristic value of node B, node C is first expanded along the second, suboptimal path as a child of node B. It's then placed into the "closed" set, and so A* graph search fails to reexpand it when it visits it as a child of A, so it never finds the optimal solution. Hence, to maintain completeness and optimality under A* graph search, we need an even stronger property than admissibility, **consistency**. The central idea of consistency is that we enforce not only that a heuristic underestimates the *total* distance to a goal from any given node, but also the cost/weight of each edge in the graph. The cost of an edge as measured by the heuristic function is simply the difference in heuristic values for two connected nodes. Mathematically, the consistency constraint can be expressed as follows:

$$\forall A, C \quad h(A) - h(C) \leq cost(A, C)$$

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.

Proof. In order to prove the above theorem, we first prove that when running A* graph search with a consistent heuristic, whenever we remove a node for expansion, we've found the optimal path to that node.

Using the consistency constraint, we can show that the values of $f(n)$ for nodes along any plan are nondecreasing. Define two nodes, n and n', where n' is a successor of n. Then:

$$\begin{aligned} f(n') &= g(n') + h(n') \\ &= g(n) + cost(n, n') + h(n') \\ &\geq g(n) + h(n) \\ &= f(n) \end{aligned}$$

If for every parent-child pair (n, n') along a path, $f(n') \geq f(n)$, then it must be the case that the values of $f(n)$ are nondecreasing along that path. We can check that the above graph violates this rule between $f(A)$

and $f(C)$. With this information, we can now show that whenever a node n is removed for expansion, its optimal path has been found. Assume towards a contradiction that this is false - that when n is removed from the fringe, the path found to n is suboptimal. This means that there must be some ancestor of n, n'', on the fringe that was never expanded but is on the optimal path to n. Contradiction! We've already shown that values of f along a path are nondecreasing, and so n'' would have been removed for expansion before n.

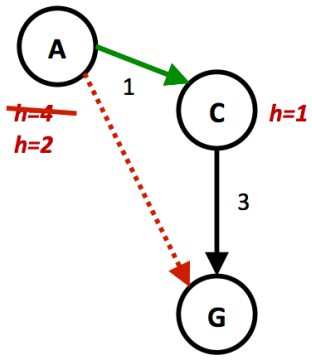
All we have left to show to complete our proof is that an optimal goal A will always be removed for expansion and returned before any suboptimal goal B. This is trivial, since $h(A) = h(B) = 0$, so

$$f(A) = g(A) < g(B) = f(B)$$

just as in our proof of optimality of A* tree search under the admissibility constraint. Hence, we can conclude that A* graph search is optimal under a consistent heuristic. □

A couple of important highlights from the discussion above before we proceed: for heuristics that are either admissible/consistent to be valid, it must by definition be the case that $h(G) = 0$ for any goal state G. Additionally, consistency is not just a stronger constraint than admissibility, consistency *implies* admissibility. This stems simply from the fact that if no edge costs are overestimates (as guaranteed by consistency), the total estimated cost from any node to a goal will also fail to be an overestimate.

Consider the following three-node network for an example of an admissible but inconsistent heuristic:



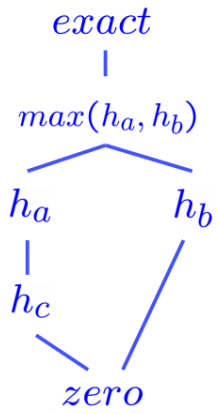
The red dotted line corresponds to the total estimated goal distance. If $h(A) = 4$, then the heuristic is admissible, as the distance from A to the goal is $4 \geq h(A)$, and same for $h(C) = 1 \leq 3$. However, the heuristic cost from A to C is $h(A) - h(C) = 4 - 1 = 3$. Our heuristic estimates the cost of the edge between A and C to be 3 while the true value is $cost(A,C) = 1$, a smaller value. Since $h(A) - h(C) \not\leq cost(A,C)$, this heuristic is not consistent. Running the same computation for $h(A) = 2$, however, yields $h(A) - h(C) = 2 - 1 = 1 \leq cost(A,C)$. Thus, using $h(A) = 2$ makes our heuristic consistent.

Dominance

Now that we've established the properties of admissibility and consistency and their roles in maintaining the optimality of A* search, we can return to our original problem of creating "good" heuristics, and how to tell if one heuristic is better than another. The standard metric for this is that of **dominance**. If heuristic a is dominant over heuristic b, then the estimated goal distance for a is greater than the estimated goal distance for b for every node in the state space graph. Mathematically,

$$\forall n : h_a(n) \geq h_b(n)$$

Dominance very intuitively captures the idea of one heuristic being better than another - if one admissible/consistent heuristic is dominant over another, it must be better because it will always more closely estimate the distance to a goal from any given state. Additionally, the **trivial heuristic** is defined as $h(n) = 0$, and using it reduces A* search to UCS. All admissible heuristics dominate the trivial heuristic. The trivial heuristic is often incorporated at the base of a **semi-lattice** for a search problem, a dominance hierarchy of which it is located at the bottom. Below is an example of a semi-lattice that incorporates various heuristics h_a, h_b, and h_c ranging from the trivial heuristic at the bottom to the exact goal distance at the top:



As a general rule, the max function applied to multiple admissible heuristics will also always be admissible. This is simply a consequence of all values output by the heuristics for any given state being constrained by the admissibility condition, $0 \leq h(n) \leq h^*(n)$. The maximum of numbers in this range must also fall in the same range. The same can be shown easily for multiple consistent heuristics as well. It's common practice to generate multiple admissible/consistent heuristics for any given search problem and compute the max over the values output by them to generate a heuristic that dominates (and hence is better than) all of them individually.

Summary

In this note, we discussed *search problems*, which we characterize formally with four components: a *state space*, a *successor function*, a *start state*, and a *goal state*. Search problems can be solved using a variety of search techniques, including but not limited to the five we study in CS 188:

- *Breadth-first Search*
- *Depth-first Search*
- *Uniform Cost Search*
- *Greedy Search*
- *A* Search*

The first three search techniques listed above are examples of *uninformed search*, while the latter two are examples of *informed search* which use *heuristics* to estimate goal distance and optimize performance.

CS 188 Introduction to Artificial Intelligence
Fall 2018 Note 2

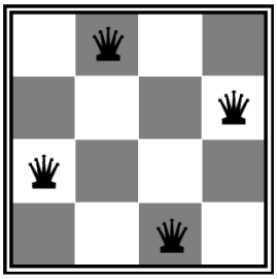
These lecture notes are heavily based on notes originally written by Nikhil Sharma.

Constraint Satisfaction Problems

In the previous note, we learned how to find optimal solutions to search problems, a type of **planning problem**. Now, we'll learn about solving a related class of problems, **constraint satisfaction problems** (CSPs). Unlike search problems, CSPs are a type of **identification problem**, problems in which we must simply identify whether a state is a goal state or not, with no regard to how we arrive at that goal. CSPs are defined by three factors:

1. *Variables* - CSPs possess a set of N variables $X_1, ..., X_N$ that can each take on a single value from some defined set of values.
2. *Domain* - A set $\{x_1, ..., x_d\}$ representing all possible values that a CSP variable can take on.
3. *Constraints* - Constraints define restrictions on the values of variables, potentially with regard to other variables.

Consider the N-queens identification problem: given an $N \times N$ chessboard, can we find a configuration in which to place N queens on the board such that no two queens attack each another?



We can formulate this problem as a CSP as follows:

1. *Variables* - X_{ij}, with $0 \leq i, j < N$. Each X_{ij} represents a grid position on our $N \times N$ chessboard, with i and j specifying the row and column number respectively.
2. *Domain* - $\{0, 1\}$. Each X_{ij} can take on either the value 0 or 1, a boolean value representing the existence of a queen at position (i, j) on the board.
3. *Constraints* -
    - $\forall i, j, k \; (X_{ij}, X_{ik}) \in \{(0,0), (0,1), (1,0)\}$. This constraint states that if two variables have the same value for i, only one of them can take on a value of 1. This effectively encapsulates the condition that no two queens can be in the same row.

- $\forall i, j, k\ (X_{ij}, X_{kj}) \in \{(0,0),(0,1),(1,0)\}$. Almost identically to the previous constraint, this constraint states that if two variables have the same value for j, only one of them can take on a value of 1, encapsulating the condition that no two queens can be in the same column.
- $\forall i, j, k\ (X_{ij}, X_{i+k,j+k}) \in \{(0,0),(0,1),(1,0)\}$. With similar reasoning as above, we can see that this constraint and the next represent the conditions that no two queens can be in the same major or minor diagonals, respectively.
- $\forall i, j, k\ (X_{ij}, X_{i+k,j-k}) \in \{(0,0),(0,1),(1,0)\}$.
- $\sum_{i,j} X_{ij} = N$. This constraint states that we must have exactly N grid positions marked with a 1, and all others marked with a 0, capturing the requirement that there are exactly N queens on the board.

Constraint satisfaction problems are **NP-hard**, which loosely means that there exists no known algorithm for finding solutions to them in polynomial time. Given a problem with N variables with domain of size $O(d)$ for each variable, there are $O(d^N)$ possible assignments, exponential in the number of variables. We can often get around this caveat by formulating CSPs as search problems, defining states as **partial assignments** (variable assignments to CSPs where some variables have been assigned values while others have not). Correspondingly, the successor function for a CSP state outputs all states with one new variable assigned, and the goal test verifies all variables are assigned and all constraints are satisfied in the state it's testing. Constraint satisfaction problems tend to have significantly more structure than traditional search problems, and we can exploit this structure by combining the above formulation with appropriate heuristics to hone in on solutions in a feasible amount of time.

Constraint Graphs

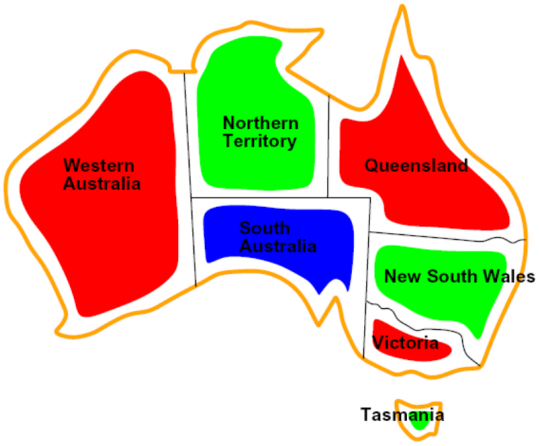
Let's introduce a second CSP example: map coloring. Map coloring solves the problem where we're given a set of colors and must color a map such that no two adjacent states or regions have the same color.

Constraint satisfaction problems are often represented as constraint graphs, where nodes represent variables and edges represent constraints between them. There are many different types of constraints, and each is handled slightly differently:

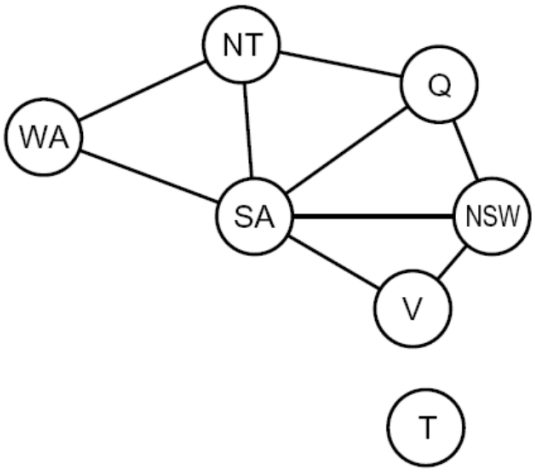
- *Unary Constraints* - Unary constraints involve a single variable in the CSP. They are not represented in constraint graphs, instead simply being used to prune the domain of the variable they constrain when necessary.

- *Binary Constraints* - Binary constraints involve two variables. They're represented in constraint graphs as traditional graph edges.
- *Higher-order Constraints* - Constraints involving three or more variables can also be represented with edges in a CSP graph, they just look slightly unconventional.

Consider map coloring the map of Australia:



The constraints in this problem are simply that no two adjacent states can be the same color. As a result, by drawing an edge between every pair of states that are adjacent to one another, we can generate the constraint graph for the map coloring of Australia as follows:



The value of constraint graphs is that we can use them to extract valuable information about the structure of the CSPs we are solving. By analyzing the graph of a CSP, we can determine things about it like whether it's sparsely or densely connected/constrained and whether or not it's tree-structured. We'll cover this more in depth as we discuss solving constraint satisfaction problems in more detail.

Solving Constraint Satisfaction Problems

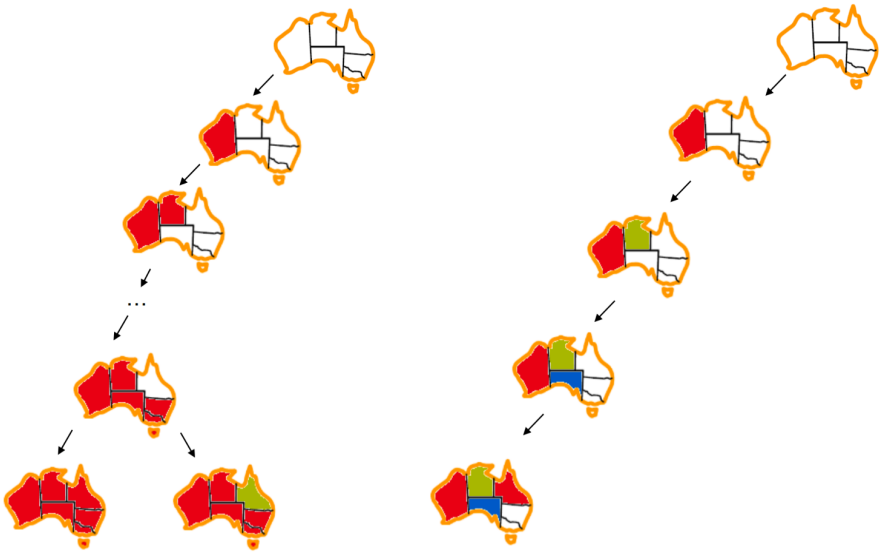
Constraint satisfaction problems are traditionally solved using a search algorithm known as **backtracking search**. Backtracking search is an optimization on depth first search used specifically for the problem of constraint satisfaction, with improvements coming from two main principles:

1. Fix an ordering for variables, and select values for variables in this order. Because assignments are commutative (e.g. assigning $WA = Red$, $NT = Green$ is identical to $NT = Green$, $WA = Red$), this is valid.
2. When selecting values for a variable, only select values that don't conflict with any previously assigned values. If no such values exist, backtrack and return to the previous variable, changing its value.

The pseudocode for how recursive backtracking works is presented below:

```
function BACKTRACKING-SEARCH(csp) returns solution/failure
    return RECURSIVE-BACKTRACKING({ }, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/failure
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
    for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
        if value is consistent with assignment given CONSTRAINTS[csp] then
            add {var = value} to assignment
            result ← RECURSIVE-BACKTRACKING(assignment, csp)
            if result ≠ failure then return result
            remove {var = value} from assignment
    return failure
```

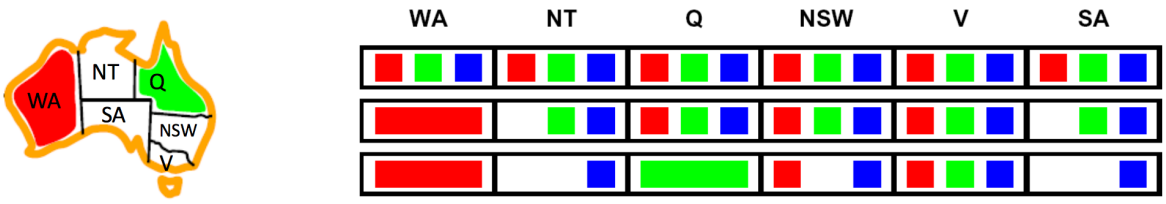
For a visualization of how this process works, consider the partial search trees for both depth first search and backtracking search in map coloring:



Note how DFS regretfully colors everything red before ever realizing the need for change, and even then doesn't move too far in the right direction towards a solution. On the other hand, backtracking search only assigns a value to a variable if that value violates no constraints, leading to a significantly less backtracking. Though backtracking search is a vast improvement over the brute-forcing of depth first search, we can get more gains in speed still with further improvements through filtering, variable/value ordering, and structural explotation.

Filtering

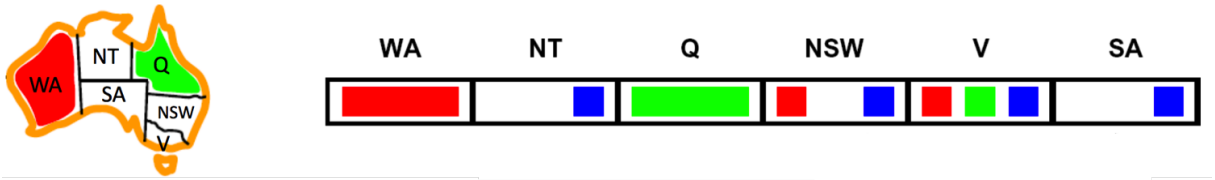
The first improvement to CSP performance we'll consider is **filtering**, which checks if we can prune the domains of unassigned variables ahead of time by removing values we know will result in backtracking. A naïve method for filtering is **forward checking**, which whenever a value is assigned to a variable X_i, prunes the domains of unassigned variables that share a constraint with X_i that would violate the constraint if assigned. Whenever a new variable is assigned, we can run forward checking and prune the domains of unassigned variables adjacent to the newly assigned variable in the constraint graph. Consider our map coloring example, with unassigned variables and their potential values:



Note how as we assign $WA = red$ and then $Q = green$, the size of the domains for NT, NSW, and SA (states adjacent to WA, Q, or both) decrease in size as values are eliminated. The idea of forward checking can be generalized into the principle of **arc consistency**. For arc consistency, we interpret each undirected edge of the constraint graph for a CSP as two directed edges pointing in opposite directions. Each of these directed edges is called an **arc**. The arc consistency algorithm works as follows:

- Begin by storing all arcs in the constraint graph for the CSP in a queue Q.
- Iteratively remove arcs from Q and enforce the condition that in each removed arc $X_i \longrightarrow X_j$, 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 $X_i = v, X_j = w$ does not violate any constraints. If some value v for X_i would not work with any of the remaining values for X_j, we remove v from the set of possible values for X_i.
- If at least one value is removed for X_i when enforcing arc consistency for an arc $X_i \longrightarrow X_j$, add arcs of the form $X_k \longrightarrow X_i$ to Q, for all unassigned variables X_k. If an arc $X_k \longrightarrow X_i$ is already in Q during this step, it doesn't need to be added again.
- Continue until Q is empty, or the domain of some variable is empty and triggers a backtrack.

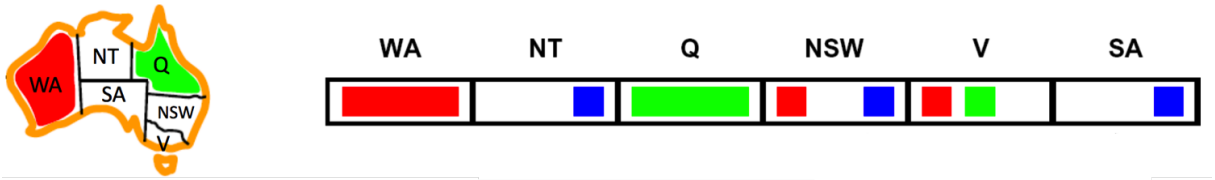
The arc consistency algorithm is typically not the most intuitive, so let's walk through a quick example with map coloring:



We begin by adding all arcs between unassigned variables sharing a constraint to a queue Q, which gives us

$$Q = [SA \to V, V \to SA, SA \to NSW, NSW \to SA, SA \to NT, NT \to SA, V \to NSW, NSW \to V]$$

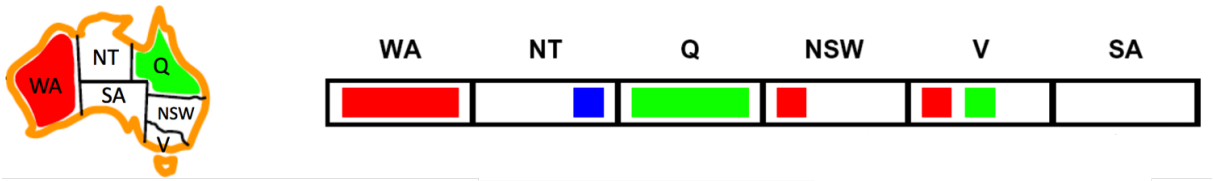
For our first arc, $SA \to V$, we see that for every value in the domain of SA, $\{blue\}$, there is *at least* one value in the domain of V, $\{red, green, blue\}$, that violates no constraints, and so no values need to be pruned from SA's domain. However, for our next arc $V \to SA$, if we set $V = blue$ we see that SA will have no remaining values that violate no constraints, and so we prune $blue$ from V's domain.



Because we pruned a value from the domain of V, we need to enqueue all arcs with V at the head - $SA \to V$, $NSW \to V$. Since $NSW \to V$ is already in Q, we only need to add $SA \to V$, leaving us with our updated queue

$$Q = [SA \to NSW, NSW \to SA, SA \to NT, NT \to SA, V \to NSW, NSW \to V, SA \to V]$$

We can continue this process until we eventually remove the arc $SA \to NT$ from Q. Enforcing arc consistency on this arc removes $blue$ from SA's domain, leaving it empty and triggering a backtrack. Note that the arc $NSW \to SA$ appears before $SA \to NT$ in Q and that enforcing consistency on this arc removes $blue$ from the domain of NSW.



Arc consistency is typically implemented with the AC-3 algorithm (Arc Consistency Algorithm #3), for which the pseudocode is as follows:

```
function AC-3( csp) returns the CSP, possibly with reduced domains
  inputs: csp, a binary CSP with variables {X_1, X_2, ..., X_n}
  local variables: queue, a queue of arcs, initially all the arcs in csp

  while queue is not empty do                          tail   head
    (X_i, X_j) ← REMOVE-FIRST(queue)
    if REMOVE-INCONSISTENT-VALUES(X_i, X_j) then
      for each X_k in NEIGHBORS[X_i] do
        add (X_k, X_i) to queue

---------------------------------------------------------------------

function REMOVE-INCONSISTENT-VALUES( X_i, X_j) returns true iff succeeds
  removed ← false
  for each x in DOMAIN[X_i] do
    if no value y in DOMAIN[X_j] allows (x,y) to satisfy the constraint X_i ↔ X_j
      then delete x from DOMAIN[X_i]; removed ← true
  return removed
```

The AC-3 algorithm has a worst case time complexity of $O(ed^3)$, where e is the number of arcs (directed edges) and d is the size of the largest domain. Overall, arc consistency is more holistic of a domain pruning technique than forward checking and leads to fewer backtracks, but requires running significantly more computation in order to enforce. Accordingly, it's important to take into account this tradeoff when deciding which filtering technique to implement for the CSP you're attempting to solve.

As an interesting parting note about consistency, arc consistency is a subset of a more generalized notion of consistency known as **k-consistency**, which 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. This idea can be further extended through the idea of **strong k-consistency**. A graph that is strong k-consistent possesses the property that any subset of k nodes is not only k-consistent but also $k-1, k-2, \ldots, 1$ consistent as well. Not surprisingly, imposing a higher degree of consistency on a CSP is more expensive to compute. Under this generalized definition for consistency, we can see that arc consistency is equivalent to 2-consistency.

Ordering

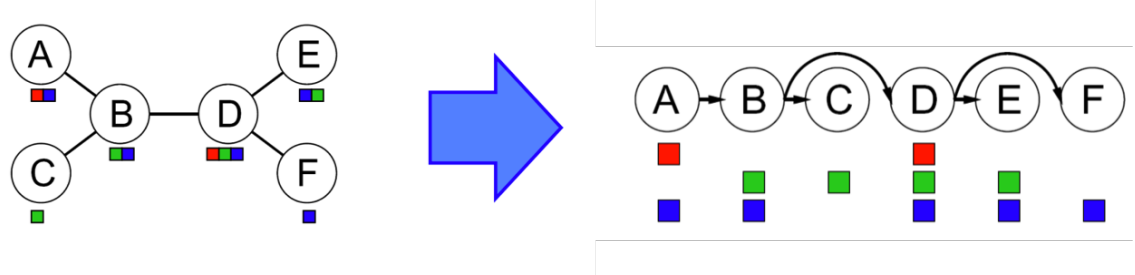
We've delineated that when solving a CSP, we fix some ordering for both the variables and values involved. In practice, it's often much more effective to compute the next variable and corresponding value "on the fly" with two broad principles, **minimum remaining values** and **least constraining value**:

- *Minimum Remaining Values (MRV)* - When selecting which variable to assign next, using an MRV policy chooses whichever unassigned variable has the fewest valid remaining values (the *most constrained variable*). This is intuitive in the sense that the most constrained variable is most likely to run out of possible values and result in backtracking if left unassigned, and so it's best to assign a value to it sooner than later.
- *Least Constraining Value (LCV)* - Similarly, when selecting which value to assign next, a good policy to implement is to select the value that prunes the fewest values from the domains of the remaining unassigned values. Notably, this requires additional computation (e.g. rerunning arc consistency/forward checking or other filtering methods for each value to find the LCV), but can still yield speed gains depending on usage.

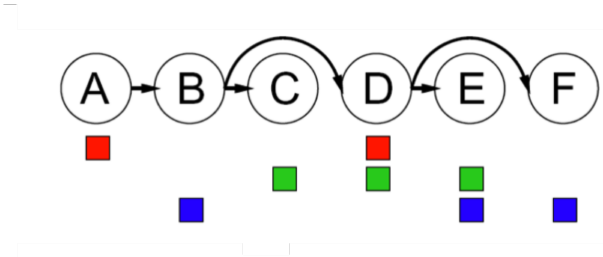
Structure

A final class of improvements to solving constraint satisfaction problems are those that exploit their structure. In particular, if we're trying to solve a **tree-structured CSP** (one that has no loops in its constraint graph), we can reduce the runtime for finding a solution from $O(d^N)$ all the way to $O(nd^2)$, linear in the number of variables. This can be done with the tree-structured CSP algorithm, outlined below:

- First, pick an arbitrary node in the constraint graph for the CSP to serve as the root of the tree (it doesn't matter which one because basic graph theory tells us any node of a tree can serve as a root).
- Convert all undirected edges in the tree to directed edges that point *away* from the root. Then **linearize** (or **topologically sort**) the resulting directed acyclic graph. In simple terms, this just means order the nodes of the graph such that all edges point rightwards. Noting that we select node A to be our root and direct all edges to point away from A, this process results in the following conversion for the CSP presented below:

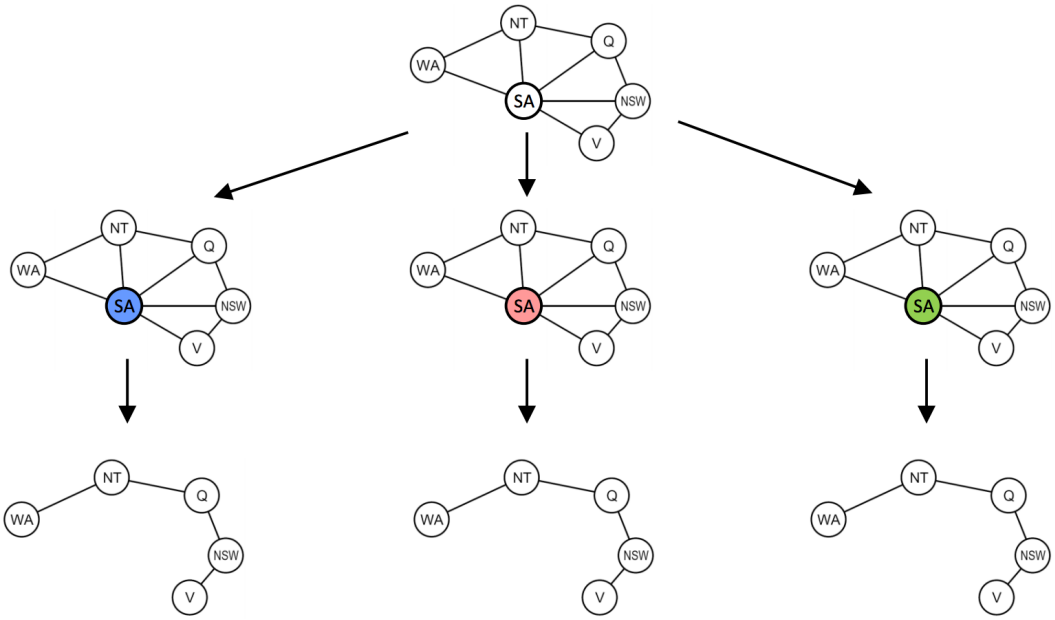


- Perform a **backwards pass** of arc consistency. Iterating from $i = n$ down to $i = 2$, enforce arc consistency for all arcs $Parent(X_i) \longrightarrow X_i$. For the linearized CSP from above, this domain pruning will eliminate a few values, leaving us with the following:



- Finally, perform a **forward assignment**. Starting from X_1 and going to X_n, assign each X_i a value consistent with that of its parent. Because we've enforced arc consistency on all of these arcs, no matter what value we select for any node, we know that its children will each all have at least one consistent value. Hence, this iterative assignment guarantees a correct solution, a fact which can be proven inductively without difficulty.

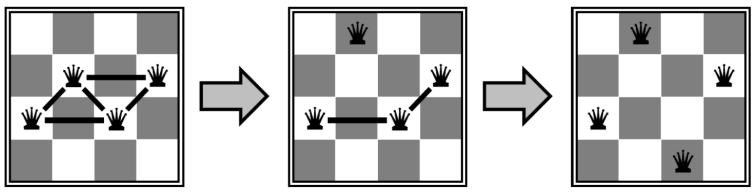
The tree structured algorithm can be extended to CSPs that are reasonably close to being tree-structured with **cutset conditioning**. Cutset conditioning involves first finding the smallest subset of variables in a constraint graph such that their removal results in a tree (such a subset is known as a **cutset** for the graph). For example, in our map coloring example, South Australia (SA) is the smallest possible cutset:



Once the smallest cutset is found, we assign all variables in it and prune the domains of all neighboring nodes. What's left is a tree-structured CSP, upon which we can solve with the tree-structured CSP algorithm from above! The initial assignment to a cutset of size c may leave the resulting tree-structured CSP(s) with no valid solution after pruning, so we may still need to backrack up to d^c times. Since removal of the cutset leaves us with a tree-structured CSP with $(n-c)$ variables, we know this can be solved (or determined that no solution exists) in $O((n-c)d^2)$. Hence, the runtime of cutset conditioning on a general CSP is $O(d^c(n-c)d^2)$, very good for small c.

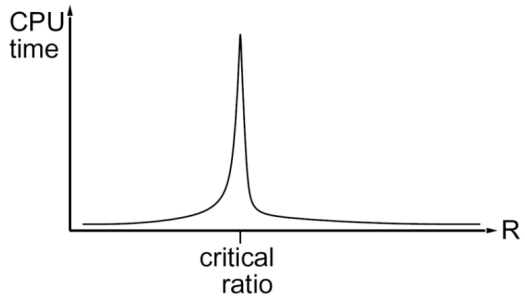
Local Search

As a final topic of interest, backtracking search is not the only algorithm that exists for solving constraint satisfaction problems. Another widely used algorithm is **local search**, for which the idea is childishly simple but remarkably useful. Local search works by iterative improvement - starting with some random assignment to values then repeatedly selecting the variable that violates the most constraints and resetting it to the value that violates the fewest constraints (a policy known as the **min-conflicts heuristic**). Under such a policy, constraint satisfaction problems like N-queens becomes both very time efficient and space efficient to solve. For example, in following example with 4 queens, we arrive at a solution after only 2 iterations:



In fact, local search appears to run in almost constant time and have a high probability of success not only for N-queens with arbitrarily large N, but also for any randomly generated CSP! However, despite these advantages, local search is both incomplete and suboptimal and so won't necessarily converge to an optimal solution. Additionally, there is a critical ratio around which using local search becomes extremely expensive:

$$R = \frac{\text{number of constraints}}{\text{number of variables}}$$



Summary

It's important to remember that constraint satisfaction problems in general do not have an efficient algorithm which solves them in polynomial time with respect to the number of variables involved. However, by using various heuristics, we can often find solutions in an acceptable amount of time:

- *Filtering* - Filtering handles pruning the domains of unassigned variables ahead of time to prevent unnecessary backtracking. The two important filtering techniques we've covered are *forward checking* and *arc consistency*.
- *Ordering* - Ordering handles selection of which variable or value to assign next to make backtracking as unlikely as possible. For variable selection, we learned about a *MRV policy* and for value selection we learned about a *LCV policy*.
- *Structure* - If a CSP is tree-structured or close to tree-structured, we can run the tree-structured CSP algorithm on it to derive a solution in linear time. Similarly, if a CSP is close to tree structured, we can use *cutset conditioning* to transform the CSP into one or more independent tree-structured CSPs and solve each of these separately.

CS 188 Introduction to Artificial Intelligence
Fall 2018 Note 3

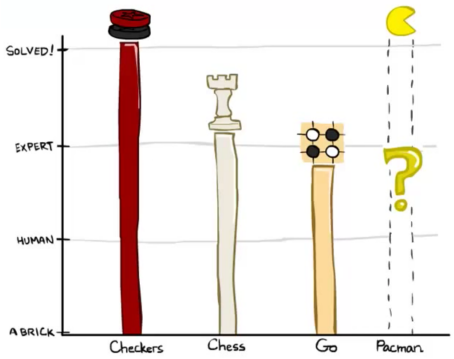
These lecture notes are heavily based on notes originally written by Nikhil Sharma.

Games

In the first note, we talked about search problems and how to solve them efficiently and optimally - using powerful generalized search algorithms, our agents could determine the best possible plan and then simply execute it to arrive at a goal. Now, let's shift gears and consider scenarios where our agents have one or more **adversaries** who attempt to keep them from reaching their goal(s). Our agents can no longer run the search algorithms we've already learned to formulate a plan as we typically don't deterministically know how our adversaries will plan against us and respond to our actions. Instead, we'll need to run a new class of algorithms that yield solutions to **adversarial search problems**, more commonly known as **games**.

There are many different types of games. Games can have actions with either deterministic or **stochastic** (probabilistic) outcomes, can have any variable number of players, and may or may not be **zero-sum**. The first class of games we'll cover are **deterministic zero-sum games**, games where actions are deterministic and our gain is directly equivalent to our opponent's loss and vice versa. The easiest way to think about such games is as being defined by a single variable value, which one team or agent tries to maximize and the opposing team or agent tries to minimize, effectively putting them in direct competition. In Pacman, this variable is your score, which you try to maximize by eating pellets quickly and efficiently while ghosts try to minimize by eating you first. Many common household games also fall under this class of games:

- *Checkers* - The first checkers computer player was created in 1950. Since then, checkers has become a **solved game**, which means that any position can be evaluated as a win, loss, or draw deterministically for either side given both players act optimally.
- *Chess* - In 1997, Deep Blue became the first computer agent to defeat human chess champion Gary Kasparov in a six-game match. Deep Blue was constructed to use extremely sophisticated methods to evaluate over 200 million positions per second. Current programs are even better, though less historic.
- *Go* - The search space for Go is much larger than for chess, and so most didn't believe Go computer agents would ever defeat human world champions for several years to come. However, AlphaGo, developed by Google, historically defeated Go champion Lee Sodol 4 games to 1 in March 2016.

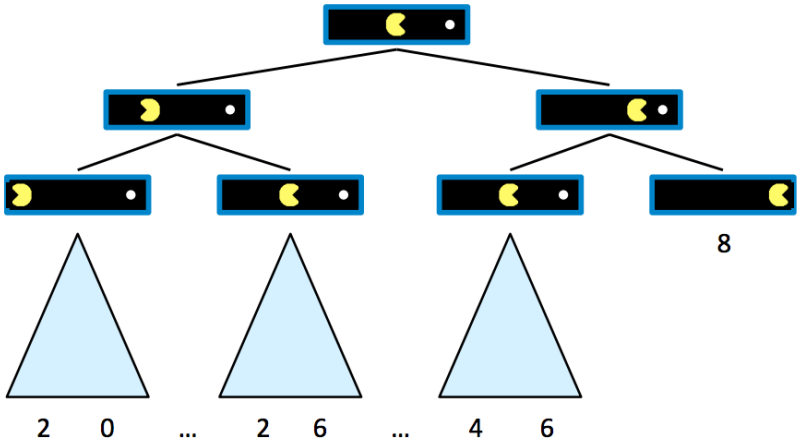


All of the world champion agents above use, at least to some degree, the adversarial search techniques that we're about to cover. As opposed to normal search, which returned a comprehensive plan, adversarial search returns a **strategy**, or **policy**, which simply recommends the best possible move given some configuration of our agent(s) and their adversaries. We'll soon see that such algorithms have the beautiful property of giving rise to behavior through computation - the computation we run is relatively simple in concept and widely generalizable, yet innately generates cooperation between agents on the same team as well as "outthinking" of adversarial agents.

Minimax

The first zero-sum-game algorithm we will consider is **minimax**, which runs under the motivating assumption that the opponent we face behaves optimally, and will always perform the move that is worst for us. To introduce this algorithm, we must first formalize the notion of **terminal utilities** and **state value**. The value of a state is the optimal score attainable by the agent which controls that state. In order to get a sense of what this means, observe the following trivially simple Pacman game board:

Assume that Pacman starts with 10 points and loses 1 point per move until he eats the pellet, at which point the game arrives at a **terminal state** and ends. We can start building a **game tree** for this board as follows, where children of a state are successor states just as in search trees for normal search problems:



It's evident from this tree that if Pacman goes straight to the pellet, he ends the game with a score of 8 points, whereas if he backtracks at any point, he ends up with some lower valued score. Now that we've generated a game tree with several terminal and intermediary states, we're ready to formalize the meaning of the value of any of these states.

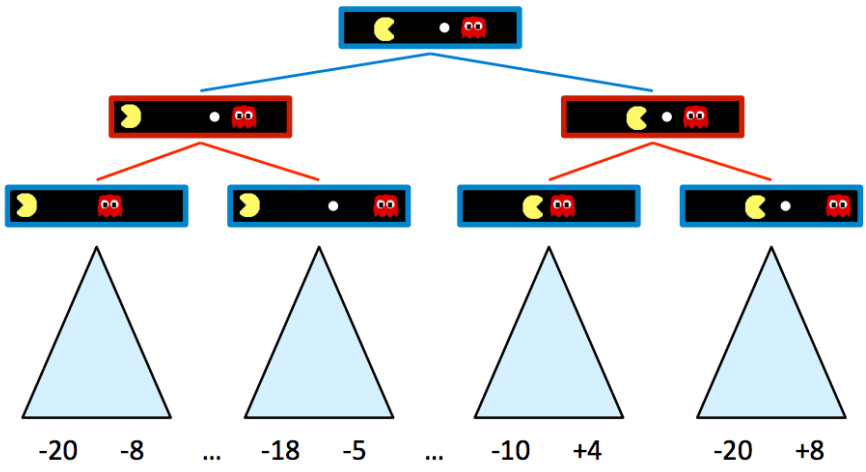
A state's value is defined as the best possible outcome (**utility**) an agent can achieve from that state. We'll formalize the concept of utility more concretely later, but for now it's enough to simply think of an agent's utility as its score or number of points it attains. The value of a terminal state, called a **terminal utility**, is always some deterministic known value and an inherent game property. In our Pacman example, the value of the rightmost terminal state is simply 8, the score Pacman gets by going straight to the pellet. Also, in this example, the value of a non-terminal state is defined as the maximum of the values of its children. Defining $V(s)$ as the function defining the value of a state s, we can summarize the above discussion:

$$\begin{aligned} \forall \text{non-terminal states}, \quad V(s) &= \max_{s' \in successors(s)} V(s') \\ \forall \text{terminal states}, \quad V(s) &= \text{known} \end{aligned}$$

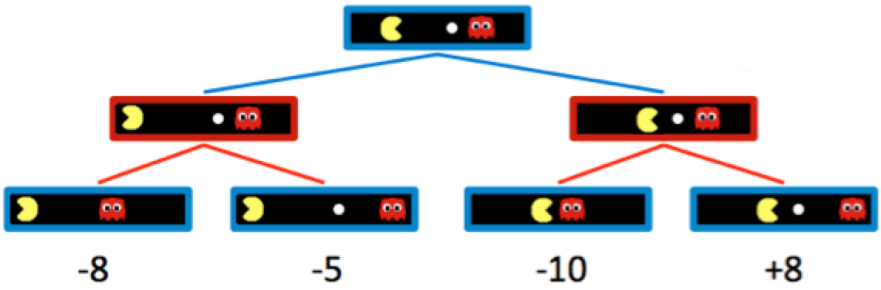
This sets up a very simple recursive rule, from which it should make sense that the value of the root node's direct right child will be 8, and the root node's direct left child will be 6, since these are the maximum possible scores the agent can obtain if it moves right or left, respectively, from the start state. It follows that by running such computation, an agent can determine that it's optimal to move right, since the right child has a greater value than the left child of the start state.

Let's now introduce a new game board with an adversarial ghost that wants to keep Pacman from eating the pellet.

The rules of the game dictate that the two agents take turns making moves, leading to a game tree where the two agents switch off on layers of the tree that they "control". An agent having control over a node simply means that node corresponds to a state where it is that agent's turn, and so it's their opportunity to decide upon an action and change the game state accordingly. Here's the game tree that arises from the new two-agent game board above:



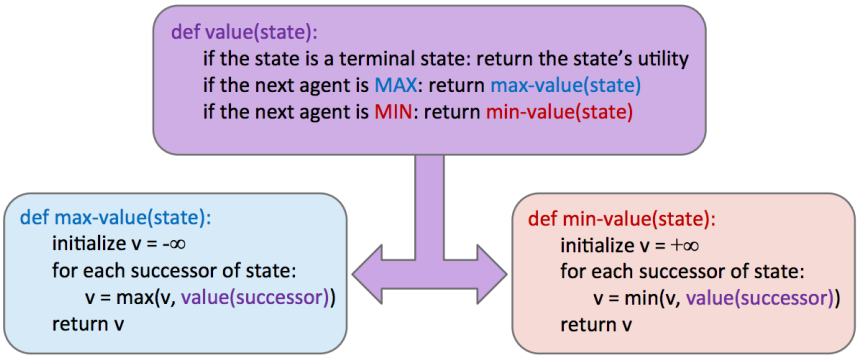
Blue nodes correspond to nodes that Pacman controls and can decide what action to take, while red nodes correspond to ghost-controlled nodes. Note that all children of ghost-controlled nodes are nodes where the ghost has moved either left or right from its state in the parent, and vice versa for Pacman-controlled nodes. For simplicity purposes, let's truncate this game tree to a depth-2 tree, and assign spoofed values to terminal states as follows:



Naturally, adding ghost-controlled nodes changes the move Pacman believes to be optimal, and the new optimal move is determined with the minimax algorithm. Instead of maximizing the utility over children at every level of the tree, the minimax algorithm only maximizes over the children of nodes controlled by Pacman, while minimizing over the children of nodes controlled by ghosts. Hence, the two ghost nodes above have values of $\min(-8,-5) = -8$ and $\min(-10,+8) = -10$ respectively. Correspondingly, the root node controlled by Pacman has a value of $\max(-8,-10) = -8$. Since Pacman wants to maximize his score, he'll go left and take the score of -8 rather than trying to go for the pellet and scoring -10. This is a prime example of the rise of behavior through computation - though Pacman wants the score of $+8$ he can get if he ends up in the rightmost child state, through minimax he "knows" that an optimally-performing ghost will not allow him to have it. In order to act optimally, Pacman is forced to hedge his bets and counterintuitively move away from the pellet to minimize the magnitude of his defeat. We can summarize the way minimax assigns values to states as follows:

$$\begin{aligned} \forall \text{ agent-controlled states}, \quad V(s) &= \max_{s' \in successors(s)} V(s') \\ \forall \text{ opponent-controlled states}, \quad V(s) &= \min_{s' \in successors(s)} V(s') \\ \forall \text{ terminal states}, \quad V(s) &= \text{known} \end{aligned}$$

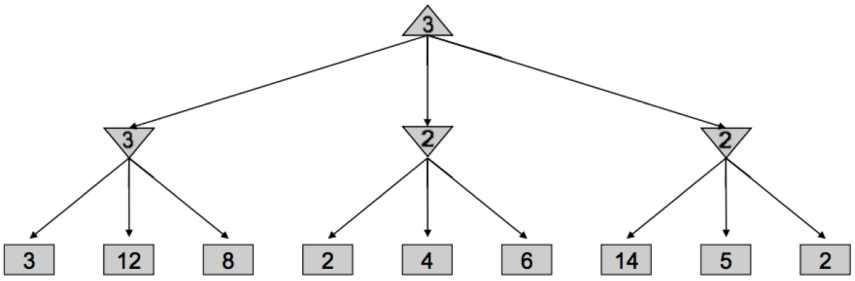
In implementation, minimax behaves similarly to depth-first search, computing values of nodes in the same order as DFS would, starting with the the leftmost terminal node and iteratively working its way rightwards. More precisely, it performs a **postorder traversal** of the game tree. The resulting pseudocode for minimax is both elegant and intuitively simple, and is presented below:



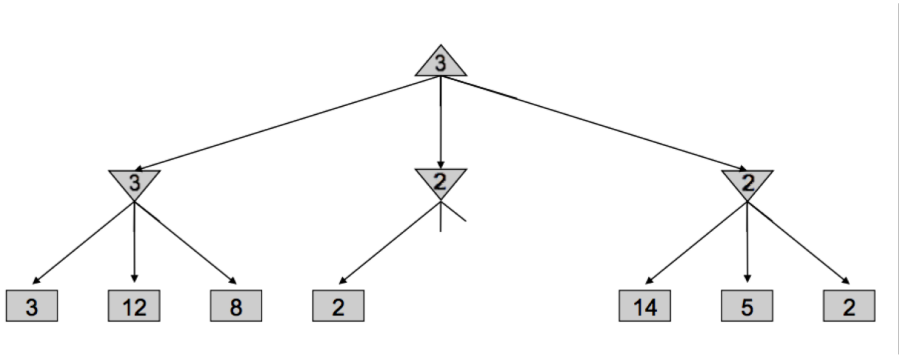
Alpha-Beta Pruning

Minimax seems just about perfect - it's simple, it's optimal, and it's intuitive. Yet, its execution is very similar to depth-first search and it's time complexity is identical, a dismal $O(b^m)$. Recalling that b is the branching factor and m is the approximate tree depth at which terminal nodes can be found, this yields far too great a runtime for many games. For example, chess has a branching factor $b \approx 35$ and tree depth $m \approx 100$. To help mitigate this issue, minimax has an optimization - **alpha-beta pruning**.

Conceptually, alpha-beta pruning is this: if you're trying to determine the value of a node n by looking at its successors, stop looking as soon as you know that n's value can at best equal the optimal value of n's parent. Let's unravel what this tricky statement means with an example. Consider the following game tree, with square nodes corresponding to terminal states, downward-pointing triangles corresponding to minimizing nodes, and upward-pointing triangles corresponding to maximizer nodes:



Let's walk through how minimax derived this tree - it began by iterating through the nodes with values 3, 12, and 8, and assigning the value $\min(3,12,8) = 3$ to the leftmost minimizer. Then, it assigned $\min(2,4,6) = 2$ to the middle minimizer, and $\min(14,5,2) = 2$ to the rightmost minimizer, before finally assigning $\max(3,2,2) = 3$ to the maximizer at the root. However, if we think about this situation, we can come to the realization that as soon as we visit the child of the middle minimizer with value 2, we no longer need to look at the middle minimizer's other children. Why? Since we've seen a child of the middle minimizer with value 2, we know that no matter what values the other children hold, the value of the middle minimizer can be at most 2. Now that this has been established, let's think one step further still - the maximizer at the root is deciding between the value of 3 of the left minimizer, and the value that's ≤ 2, it's guaranteed to prefer the 3 returned by the left minimizer over the value returned by the middle minimizer, regardless of the values of its remaining children. This is precisely why we can **prune** the search tree, never looking at the remaining children of the middle minimizer:



Implementing such pruning can reduce our runtime to as good as $O(b^{m/2})$, effectively doubling our "solvable" depth. In practice, it's often a lot less, but generally can make it feasible to search down to at least one or two more levels. This is still quite significant, as the player who thinks 3 moves ahead is favored to win over the player who thinks 2 moves ahead. This pruning is exactly what the minimax algorithm with alpha-beta pruning does, and is implemented as follows:

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
```

Take some time to compare this with the pseudocode for vanilla minimax, and note that we can now return early without searching through every successor.

Evaluation Functions

Though alpha-beta pruning can help increase the depth for which we can feasibly run minimax, this still usually isn't even close to good enough to get to the bottom of search trees for a large majority of games. As a result, we turn to **evaluation functions**, functions that take in a state and output an estimate of the true minimax value of that node. Typically, this is plainly interpreted as "better" states being assigned higher values by a good evaluation function than "worse" states. Evaluation functions are widely employed in **depth-limited minimax**, where we treat non-terminal nodes located at our maximum solvable depth as terminal nodes, giving them mock terminal utilities as determined by a carefully selected evaluation function. Because evaluation functions can only yield estimates of the values of non-terminal utilities, this removes the guarantee of optimal play when running minimax.

A lot of thought and experimentation is typically put into the selection of an evaluation function when designing an agent that runs minimax, and the better the evaluation function is, the closer the agent will come to behaving optimally. Additionally, going deeper into the tree before using an evaluation function also tends to give us better results - burying their computation deeper in the game tree mitigates the compromising of optimality. These functions serve a very similar purpose in games as heuristics do in standard search problems.

The most common design for an evaluation function is a linear combination of **features**.

$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + ... + w_n f_n(s)$$

Each $f_i(s)$ corresponds to a feature extracted from the input state s, and each feature is assigned a corresponding **weight** w_i. Features are simply some element of a game state that we can extract and assign a numerical value. For example, in a game of checkers we might construct an evaluation function with 4 features: number of agent pawns, number of agent kings, number of opponent pawns, and number of opponent kings. We'd then select appropriate weights based loosely on their importance. In our checkers example, it makes most sense to select positive weights for our agent's pawns/kings and negative weights for our opponents pawns/kings. Furthermore, we might decide that since kings are more valuable pieces in checkers than pawns, the features corresponding to our agent's/opponent's kings deserve weights with greater magnitude than the features concerning pawns. Below is a possible evaluation function that conforms to the features and weights we've just brainstormed:

$$Eval(s) = 2 \cdot agent\_kings(s) + agent\_pawns(s) - 2 \cdot opponent\_kings(s) - opponent\_pawns(s)$$

As you can tell, evaluation function design can be quite free-form, and don't necessarily have to be linear functions either. The most important thing to keep in mind is that the evaluation function yields higher scores for better positions as frequently as possible. This may require a lot of fine-tuning and experimenting on the performance of agents using evaluation functions with a multitude of different features and weights.

Expectimax

We've now seen how minimax works and how running full minimax allows us to respond optimally against an optimal opponent. However, minimax has some natural constraints on the situations to which it can respond. Because minimax believes it is responding to an optimal opponent, it's often overly pessimistic in situations where optimal responses to an agent's actions are not guaranteed. Such situations include scenarios with inherent randomness such as card or dice games or unpredictable opponents that move randomly or suboptimally. We'll talk about scenarios with inherent randomness much more in detail when we discuss **Markov decision processes** in the next note.

This randomness can be represented through a generalization of minimax known as **expectimax**. Expectimax introduces *chance nodes* into the game tree, which instead of considering the worst case scenario as minimizer nodes do, considers the *average case*. More specifically, while minimizers simply compute the minimum utility over their children, chance nodes compute the **expected utility** or expected value. Our rule for determining values of nodes with expectimax is as follows:

$$\begin{aligned}
\forall \text{ agent-controlled states}, \quad V(s) &= \max_{s' \in successors(s)} V(s') \\
\forall \text{ chance states}, \quad V(s) &= \sum_{s' \in successors(s)} p(s'|s) V(s') \\
\forall \text{ terminal states}, \quad V(s) &= \text{known}
\end{aligned}$$

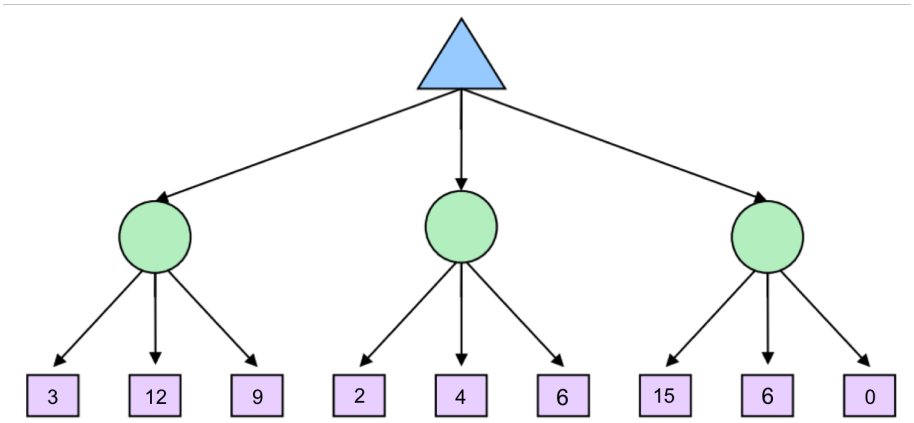
In the above formulation, $p(s'|s)$ refers to either the probability that a given nondeterministic action results in moving from state s to s', or the probability that an opponent chooses an action that results in moving from state s to s', depending on the specifics of the game and the game tree under consideration. From this definition, we can see that minimax is simply a special case of expectimax. Minimizer nodes are simply chance nodes that assign a probability of 1 to their lowest-value child and probability 0 to all other children.

In general, probabilities are selected to properly reflect the game state we're trying to model, but we'll cover how this process works in more detail in future notes. For now, it's fair to assume that these probabilities are simply inherent game properties.

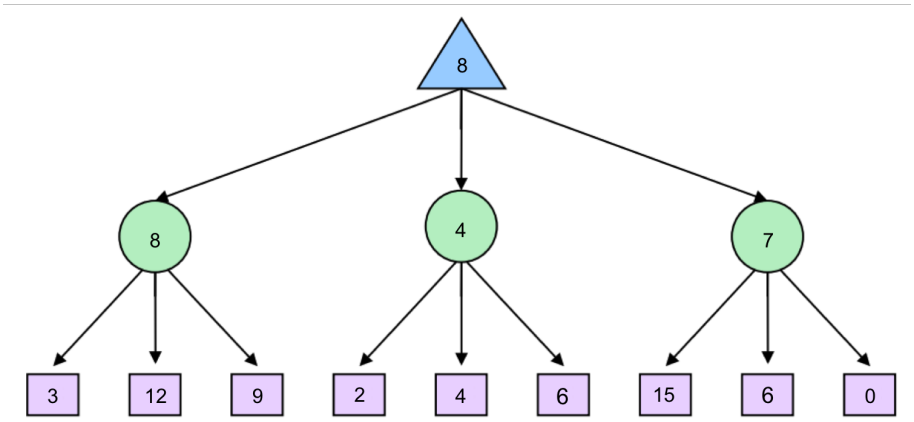
The pseudocode for expectimax is quite similar to minimax, with only a few small tweaks to account for expected utility instead of minimum utility, since we're replacing minimizing nodes with chance nodes:

```
def value(state):
    if the state is a terminal state: return the state's utility
    if the next agent is MAX: return max-value(state)
    if the next agent is EXP: return exp-value(state)

def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor))
    return v

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

Before we continue, let's quickly step through a simple example. Consider the following expectimax tree, where chance nodes are represented by circular nodes instead of the upward/downward facing triangles for maximizers/minimizers.



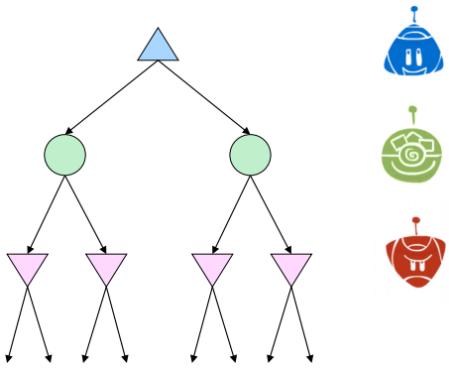
Assume for simplicity that all children of each chance node have a probability of occurrence of $\frac{1}{3}$. Hence, from our expectimax rule for value determination, we see that from left to right the 3 chance nodes take on values of $\frac{1}{3}\cdot 3+\frac{1}{3}\cdot 12+\frac{1}{3}\cdot 9=\boxed{8}$, $\frac{1}{3}\cdot 2+\frac{1}{3}\cdot 4+\frac{1}{3}\cdot 6=\boxed{4}$, and $\frac{1}{3}\cdot 15+\frac{1}{3}\cdot 6+\frac{1}{3}\cdot 0=\boxed{7}$. The maximizer selects the maximimum of these three values, $\boxed{8}$, yielding a filled-out game tree as follows:



As a final note on expectimax, it's important to realize that, in general, it's necessary to look at all the children of chance nodes – we can't prune in the same way that we could for minimax. Unlike when computing minimums or maximums in minimax, a single value can skew the expected value computed by expectimax arbitrarily high or low. However, pruning can be possible when we have known, finite bounds on possible node values.

Mixed Layer Types

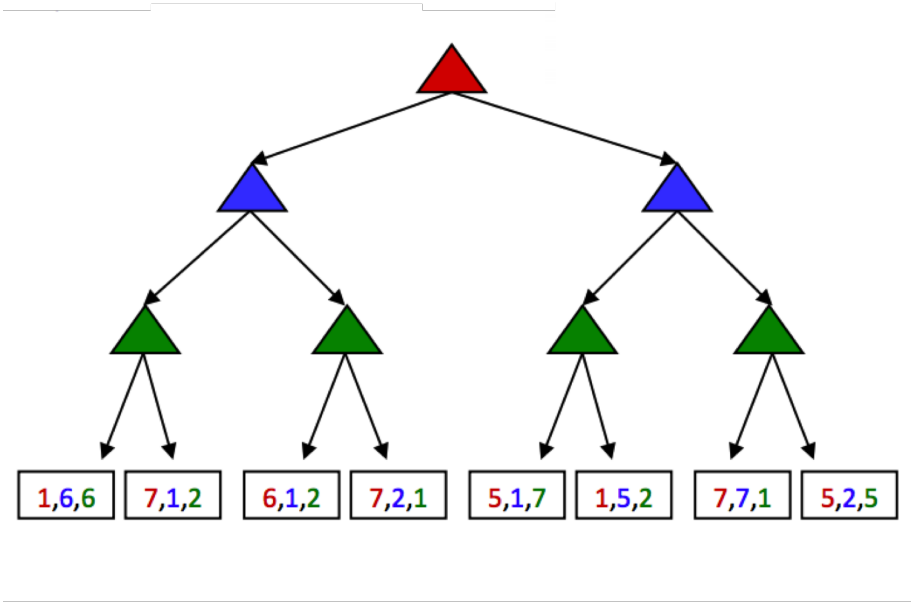
Though minimax and expectimax call for alternating maximizer/minimizer nodes and maximizer/chance nodes respectively, many games still don't follow the exact pattern of alternation that these two algorithms mandate. Even in Pacman, after Pacman moves, there are usually multiple ghosts that take turns making moves, not a single ghost. We can account for this by very fluidly adding layers into our game trees as necessary. In the Pacman example for a game with four ghosts, this can be done by having a maximizer layer followed by 4 consecutive ghost/minimizer layers before the second Pacman/maximizer layer. In fact, doing so inherently gives rise to cooperation across all minimizers, as they alternatively take turns further minimizing the utility attainable by the maximizer(s). It's even possible to combine chance node layers with both minimizers and maximizers. If we have a game of Pacman with two ghosts, where one ghost behaves randomly and the other behaves optimally, we could simulate this with alternating groups of maximizer-chance-minimizer nodes.



As is evident, there's quite a bit of room for robust variation in node layering, allowing development of game trees and adversarial search algorithms that are modified expectimax/minimax hybrids for any zero-sum game.

General Games

Not all games are zero-sum. Indeed, different agents may have have distinct tasks in a game that don't directly involve strictly competing with one another. Such games can be set up with trees characterized by **multi-agent utilities**. Such utilities, rather than being a single value that alternating agents try to minimize or maximize, are represented as tuples with different values within the tuple corresponding to unique utilities for different agents. Each agent then attempts to maximize their own utility at each node they control, ignoring the utilities of other agents. Consider the following tree:



The red, green, and blue nodes correspond to three separate agents, who maximize the red, green, and blue utilities respectively out of the possible options in their respective layers. Working through this example ultimately yields the utility tuple $(5, 2, 5)$ at the top of the tree. General games with multi-agent utilities are a prime example of the rise of behavior through computation, as such setups invoke cooperation since the utility selected at the root of the tree tends to yield a reasonable utility for all participating agents.

Utilities

Thoughout our discussion of games, the concept of utility has come up repeatedly. Utility values are generally hard-wired into games, and agents run some variation of the algorithms discussed in this note to select an action. We'll now discuss what's necessary in order to generate a viable utility function.

Rational agents must follow the **principle of maximum utility** - they must always select the action that maximizes their expected utility. However, obeying this principle only benefits agents that have **rational preferences**. To construct an example of irrational preferences, say there exist 3 objects, A, B, and C, and our agent is currently in possession of A. Say our agent has the following set of irrational preferences:

- Our agent prefers B to A plus \$1
- Our agent prefers C to B plus \$1
- Our agent prefers A to C plus \$1

A malicious agent in possession of B and C can trade our agent B for A plus a dollar, then C for B plus a dollar, then A again for C plus a dollar. Our agent has just lost \$3 for nothing! In this way, our agent can be forced to give up all of its money in an endless and nightmarish cycle.

Let's now properly define the mathematical language of preferences:

- If an agent prefers receiving a prize A to receiving a prize B, this is written $A \succ B$
- If an agent is indifferent between receiving A or B, this is written as $A \sim B$
- A **lottery** is a situation with different prizes resulting with different probabilities. To denote lottery where A is received with probability p and B is received with probability $(1-p)$, we write

$$L = [p, A;\ (1-p), B]$$

In order for a set of preferences to be rational, they must follow the five **Axioms of Rationality**:

- *Orderability*: $(A \succ B) \vee (B \succ A) \vee (A \sim B)$
  A rational agent must either prefer one of A or B, or be indifferent between the two.
- *Transitivity*: $(A \succ B) \wedge (B \succ C) \Rightarrow (A \succ C)$
  If a rational agent prefers A to B and B to C, then it prefers A to C.
- *Continuity*: $A \succ B \succ C \Rightarrow \exists p\, [p, A;\ (1-p), C] \sim B$
  If a rational agent prefers A to B but B to C, then it's possible to construct a lottery L between A and C such that the agent is indifferent between L and B with appropriate selection of p.
- *Substitutability*: $A \sim B \Rightarrow [p, A;\ (1-p), C] \sim [p, B;\ (1-p), C]$
  A rational agent indifferent between two prizes A and B is also indifferent between any two lotteries which only differ in substitutions of A for B or B for A.
- *Monotonicity*: $A \succ B \Rightarrow (p \geq q \Leftrightarrow [p, A;\ (1-p), B] \succeq [q, A;\ (1-q), B]$
  If a rational agent prefers A over B, then given a choice between lotteries involving only A and B, the agent prefers the lottery assigning the highest probability to A.

If all five axioms are satisfied by an agent, then it's guaranteed that the agent's behavior is describable as a maximization of expected utility. More specifically, this implies that there exists a real-valued **utility function** U that when implemented will assign greater utilities to preferred prizes, and also that the utility of a lottery is the expected value of the utility of the prize resulting from the lottery. These two statements can be summarized in two concise mathematical equivalences:

$$U(A) \geq U(B) \quad \Leftrightarrow \quad A \succeq B \tag{1}$$

$$U([p_1, S_1;\ \ldots\ ; p_n, S_n]) \quad = \quad \sum_i p_i U(S_i) \tag{2}$$

If these constraints are met and an appropriate choice of algorithm is made, the agent implementing such a utility function is guaranteed to behave optimally. Let's discuss utility functions in greater detail with a concrete example. Consider the following lottery:

$$L = [0.5, \$0;\ 0.5, \$1000]$$

This represents a lottery where you receive \$1000 with probability 0.5 and \$0 with probability 0.5. Now consider three agents A_1, A_2, and A_3 which have utility functions $U_1(\$x) = x$, $U_2(\$x) = \sqrt{x}$, and $U_3(\$x) = x^2$ respectively. If each of the three agents were faced with a choice between participting in the lottery and receiving a flat payment of \$500, which would they choose? The respective utilities for each agent of participating in the lottery and accepting the flat payment are listed in the following table:

Agent	Lottery	Flat Payment
1	500	500
2	15.81	22.36
3	500000	250000

These utility values for the lotteries were calculated as follows, making use of equation (2) above:

$$\begin{aligned}
U_1(L) &= U_1([0.5, \$0;\ 0.5, \$1000]) = 0.5 \cdot U_1(\$1000) + 0.5 \cdot U_1(\$0) = 0.5 \cdot 1000 + 0.5 \cdot 0 = \boxed{500} \\
U_2(L) &= U_2([0.5, \$0;\ 0.5, \$1000]) = 0.5 \cdot U_2(\$1000) + 0.5 \cdot U_2(\$0) = 0.5 \cdot \sqrt{1000} + 0.5 \cdot \sqrt{0} = \boxed{15.81} \\
U_3(L) &= U_1([0.5, \$0;\ 0.5, \$1000]) = 0.5 \cdot U_3(\$1000) + 0.5 \cdot U_3(\$0) = 0.5 \cdot 1000^2 + 0.5 \cdot 0^2 = \boxed{500000}
\end{aligned}$$

With these results, we can see that agent A_1 is indifferent between participating in the lottery and receiving the flat payment (the utilities for both cases are identical). Such an agent is known as **risk-neutral**. Similarly, agent A_2 prefers the flat payment to the lottery and is known as **risk-averse** and agent A_3 prefers the lottery to the flat payment and is known as **risk-seeking**.

Summary

In this note, we shifted gears from considering standard search problems where we simply attempt to find a path from our starting point to some goal, to considering adversarial search problems where we may have opponents that attempt to hinder us from reaching our goal. Two primary algorithms were considered:

- **Minimax** - Used when our opponent(s) behaves optimally, and can be optimized using α-β pruning. Minimax provides more conservative actions than expectimax, and so tends to yield favorable results when the opponent is unknown as well.
- **Expectimax** - Used when we facing a suboptimal opponent(s), using a probability distribution over the moves we believe they will make to compute the expectated value of states.

In most cases, it's too computationally expensive to run the above algorithms all the way to the level of terminal nodes in the game tree under consideration, and so we introduced the notion of evaluation functions for early termination. Finally, we considered the problem of defining utility functions for agents such that they make rational decisions. With appropriate function selection, we can additionally make our agents risk-seeking, risk-averse, or risk-neutral.

CS 188 Introduction to Artificial Intelligence
Fall 2018 Note 4

These lecture notes are heavily based on notes originally written by Nikhil Sharma.

Non-Deterministic Search

Picture a runner, coming to the end of his first ever marathon. Though it seems likely he will complete the race and claim the accompanying everlasting glory, it's by no means guaranteed. He may pass out from exhaustion or misstep and slip and fall, tragically breaking both of his legs. Even more unlikely, a literally earth-shattering earthquake may spontaneously occur, swallowing up the runner mere inches before he crosses the finish line. Such possibilities add a degree of *uncertainty* to the runner's actions, and it's this uncertainty that will be the subject of the following discussion. In the first note, we talked about traditional search problems and how to solve them; then, in the third note, we changed our model to account for adversaries and other agents in the world that influenced our path to goal states. Now, we'll change our model again to account for another influencing factor – the dynamics of world itself. The environment in which an agent is placed may subject the agent's actions to being **nondeterministic**, which means that there are multiple possible successor states that can result from an action taken in some state. This is, in fact, the case in many card games such as poker or blackjack, where there exists an inherent uncertainty from the randomness of card dealing. Such problems where the world poses a degree of uncertainty are known as **nondeterministic search problems**, and can be solved with models known as **Markov decision processes**, or MDPs.

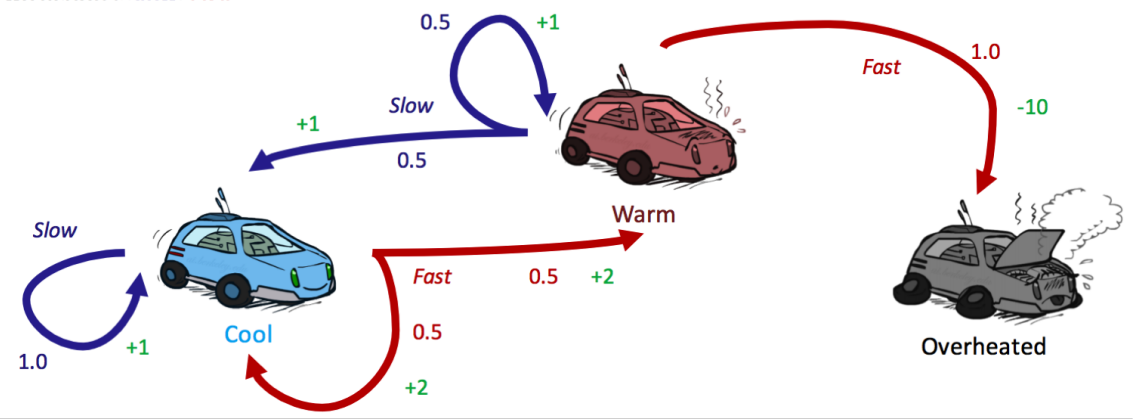
Markov Decision Processes

A Markov Decision Process is defined by several properties:

- A set of states S. States in MDPs are represented in the same way as states in traditional search problems.
- A set of actions A. Actions in MDPs are also represented in the same way as in traditional search problems.
- A start state.
- Possibly one or more terminal states.
- Possibly a **discount factor** γ. We'll cover discount factors shortly.
- A **transition function** $T(s,a,s')$. Since we have introduced the possibility of nondeterministic actions, we need a way to delineate the likelihood of the possible outcomes after taking any given action from any given state. The transition function for a MDP does exactly this - it's a probability function which represents the probability that an agent taking an action $a \in A$ from a state $s \in S$ ends up in a state $s' \in S$.

- A **reward function** $R(s,a,s')$. Typically, MDPs are modeled with small "living" rewards at each step to reward an agent's survival, along with large rewards for arriving at a terminal state. Rewards may be positive or negative depending on whether or not they benefit the agent in question, and the agent's objective is naturally to acquire the maximum reward possible before arriving at some terminal state.

Constructing a MDP for a situation is quite similar to constructing a state-space graph for a search problem, with a couple additional caveats. Consider the motivating example of a racecar:



There are three possible states, $S = \{cool, warm, overheated\}$, and two possible actions $A = \{slow, fast\}$. Just like in a state-space graph, each of the three states is represented by a node, with edges representing actions. *Overheated* is a terminal state, since once a racecar agent arrives at this state, it can no longer perform any actions for further rewards (it's a *sink state* in the MDP and has no outgoing edges). Notably, for nondeterministic actions, there are multiple edges representing the same action from the same state with differing successor states. Each edge is annotated not only with the action it represents, but also a transition probability and corresponding reward. These are summarized below:

- **Transition Function**: $T(s,a,s')$
  - $T(cool, slow, cool) = 1$
  - $T(warm, slow, cool) = 0.5$
  - $T(warm, slow, warm) = 0.5$
  - $T(cool, fast, cool) = 0.5$
  - $T(cool, fast, warm) = 0.5$
  - $T(warm, fast, overheated) = 1$
- **Reward Function**: $R(s,a,s')$
  - $R(cool, slow, cool) = 1$
  - $R(warm, slow, cool) = 1$
  - $R(warm, slow, warm) = 1$
  - $R(cool, fast, cool) = 2$
  - $R(cool, fast, warm) = 2$
  - $R(warm, fast, overheated) = -10$

We represent the movement of an agent through different MDP states over time with discrete **timesteps**, defining $s_t \in S$ and $a_t \in A$ as the state in which an agent exists and the action which an agent takes at timestep t respectively. An agent starts in state s_0 at timestep 0, and takes an action at every timestep. The movement of an agent through a MDP can thus be modeled as follows:

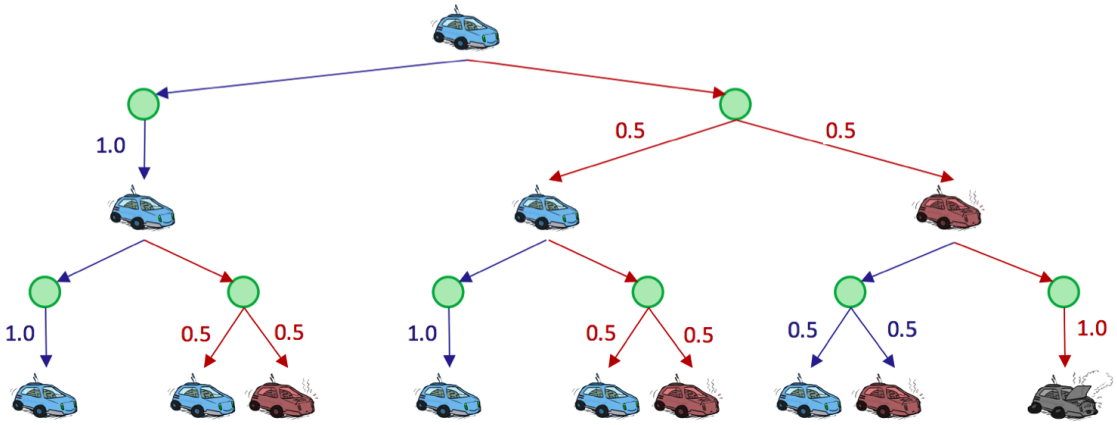
$$s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} s_3 \xrightarrow{a_3} \ldots$$

Additionally, knowing that an agent's goal is to maximize it's reward across all timesteps, we can correspondingly express this mathematically as a maximization of the following utility function:

$$U([s_0, a_0, s_1, a_1, s_2, \ldots]) = R(s_0, a_0, s_1) + R(s_1, a_1, s_2) + R(s_2, a_2, s_3) + \ldots$$

Markov decision processes, like state-space graphs, can be unraveled into search trees. Uncertainty is modeled in these search trees with **q-states**, also known as **action states**, essentially identical to expectimax chance nodes. This is a fitting choice, as q-states use probabilities to model the uncertainty that the environment will land an agent in a given state just as expectimax chance nodes use probabilities to model the uncertainty that adversarial agents will land our agent in a given state through the move these agents select. The q-state represented by having taken action a from state s is notated as the tuple (s,a).

Observe the unraveled search tree for our racecar, truncated to depth-2:



The green nodes represent q-states, where an action has been taken from a state but has yet to be resolved into a successor state. It's important to understand that agents spend zero timesteps in q-states, and that they are simply a construct created for ease of representation and development of MDP algorithms.

Finite Horizons and Discounting

There is an inherent problem with our racecar MDP - we haven't placed any time constraints on the number of timesteps for which a racecar can take actions and collect rewards. With our current formulation, it could routinely choose $a = slow$ at every timestep forever, safely and effectively obtaining infinite reward without any risk of overheating. This is prevented by the introduction of **finite horizons** and/or **discount factors**. An MDP enforcing a finite horizon is simple - it essentially defines a "lifetime" for agents, which gives them some set number of timesteps n to accrue as much reward as they can before being automatically terminated. We'll return to this concept shortly.

Discount factors are slightly more complicated, and are introduced to model an exponential decay in the value of rewards over time. Concretely, with a discount factor of γ, taking action a_t from state s_t at timestep t and ending up in state s_{t+1} results in a reward of $\gamma^t R(s_t, a_t, s_{t+1})$ instead of just $R(s_t, a_t, s_{t+1})$. Now, instead of maximizing the **additive utility**

$$U([s_0, a_0, s_1, a_1, s_2, ...]) = R(s_0, a_0, s_1) + R(s_1, a_1, s_2) + R(s_2, a_2, s_3) + ...$$

we attempt to maximize **discounted utility**

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

Noting that the above definition of a discounted utility function looks dangerously close to a **geometric series** with ratio γ, we can prove that it's guaranteed to be finite-valued as long as the constraint $|\gamma| < 1$

(where $|n|$ denotes the absolute value operator) is met through the following logic:

$$\begin{aligned} U([s_0, s_1, s_2, ...]) &= R(s_0, a_0, s_1) + \gamma R(s_1, a_1, s_2) + \gamma^2 R(s_2, a_2, s_3) + ... \\ &= \sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_{t+1}) \leq \sum_{t=0}^{\infty} \gamma^t R_{max} = \boxed{\frac{R_{max}}{1-\gamma}} \end{aligned}$$

where R_{max} is the maximum possible reward attainable at any given timestep in the MDP. Typically, γ is selected strictly from the range $0 < \gamma < 1$ since values values in the range $-1 < \gamma \leq 0$ are simply not meaningful in most real-world situations - a negative value for γ means the reward for a state s would flip-flop between positive and negative values at alternating timesteps.

Markovianess

Markov decision processes are "markovian" in the sense that they satisfy the **Markov property**, or **memoryless property**, which states that the future and the past are conditionally independent, given the present. Intuitively, this means that, if we know the present state, knowing the past doesn't give us any more information about the future. To express this mathematically, consider an agent that has visited states $s_0, s_1, ..., s_t$ after taking actions $a_0, a_1, ..., a_{t-1}$ in some MDP, and has just taken action a_t. The probability that this agent then arrives at state s_{t+1} given their history of previous states visited and actions taken can be written as follows:

$$P(S_{t+1} = s_{t+1} | S_t = s_t, A_t = a_t, S_{t-1} = s_{t-1}, A_{t-1} = a_{t-1}, ..., S_0 = s_0)$$

where each S_t denotes the random variable representing our agent's state and A_t denotes the random variable representing the action our agent takes at time t. The Markov property states that the above probability can be simplified as follows:

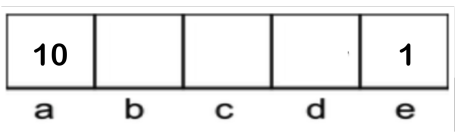
$$P(S_{t+1} = s_{t+1} | S_t = s_t, A_t = a_t, S_{t-1} = s_{t-1}, A_{t-1} = a_{t-1}, ..., S_0 = s_0) = P(S_{t+1} = s_{t+1} | S_t = s_t, A_t = a_t)$$

which is "memoryless" in the sense that the probability of arriving in a state s' at time $t+1$ depends only on the state s and action a taken at time t, not on any earlier states or actions. In fact, it is these memoryless probabilities which are encoded by the transition function: $\boxed{T(s, a, s') = P(s'|s, a)}$.

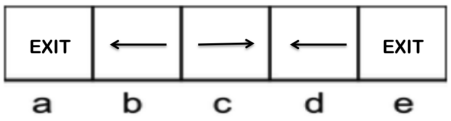
Solving Markov Decision Processes

Recall that in deterministic, non-adversarial search, solving a search problem means finding an optimal plan to arrive at a goal state. Solving a Markov decision process, on the other hand, means finding an optimal **policy** $\pi^* : S \to A$, a function mapping each state $s \in S$ to an action $a \in A$. An explicit policy π defines a reflex agent - given a state s, an agent at s implementing π will select $a = \pi(s)$ as the appropriate action to make without considering future consequences of its actions. An optimal policy is one that if followed by the implementing agent, will yield the maximum expected total reward or utility.

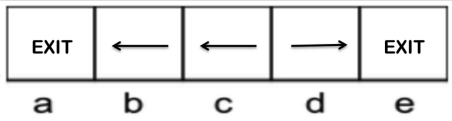
Consider the following MDP with $S = \{a, b, c, d, e\}$, $A = \{East, West, Exit\}$ (with *Exit* being a valid action only in states a and e and yielding rewards of 10 and 1 respectively), a discount factor $\gamma = 0.1$, and deterministic transitions:



Two potential policies for this MDP are as follows:



(a) Policy 1



(b) Policy 2

With some investigation, it's not hard to determine that Policy 2 is optimal. Following the policy until making action $a = Exit$ yields the following rewards for each start state:

Start State	Reward
a	10
b	1
c	0.1
d	0.1
e	1

We'll now learn how to solve such MDPs (and much more complex ones!) algorithmically using the **Bellman equation** for Markov decision processes.

The Bellman Equation

In order to talk about the Bellman equation for MDPs, we must first introduce two new mathematical quantities:

- The optimal value of a state s, $V^*(s)$ – the optimal value of s is the expected value of the utility an optimally-behaving agent that starts in s will receive, over the rest of the agent's lifetime.
- The optimal value of a q-state (s,a), $Q^*(s,a)$ - the optimal value of (s,a) is the expected value of the utility an agent receives after starting in s, taking a, and acting optimally henceforth.

Using these two new quantities and the other MDP quantities discussed earlier, the Bellman equation is defined as follows:

$$V^*(s) = \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$$

Before we begin interpreting what this means, let's also define the equation for the optimal value of a q-state (more commonly known as an optimal **q-value**):

$$Q^*(s,a) = \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$$

Note that this second definition allows us to reexpress the Bellman equation as

$$V^*(s) = \max_a Q^*(s,a)$$

which is a dramatically simpler quantity. The Bellman equation is an example of a *dynamic programming equation*, an equation that decomposes a problem into smaller subproblems via an inherent recursive structure. We can see this inherent recursion in the equation for the q-value of a state, in the term

$[R(s,a,s') + \gamma V^*(s')]$. This term represents the total utility an agent receives by first taking a from s and arriving at s' and then acting optimally henceforth. The immediate reward from the action a taken, $R(s,a,s')$, is added to the optimal reward attainable from s', $V^*(s')$, which is discounted by γ to account for the passage of the timestep in taking a. Though in most cases there exists a vast number of possible sequences of states and actions from s' to some terminal state, all this detail is abstracted away and encapsulated in a single recursive value, $V^*(s')$.

We can now take another step outwards and consider the full equation for q-value. Knowing $[R(s,a,s') + \gamma V^*(s')]$ represents the utility attained by acting optimally after arriving in state s' from q-state (s,a), it becomes evident that the quantity

$$\sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$$

is simply a weighted sum of utilities, with each utility weighted by its probability of occurrence. This is definitionally the *expected utility* of acting optimally from q-state (s,a) onwards! This completes our analysis and gives us enough insight to interpret the full Bellman equation - the optimal value of a state, $V^*(s)$, is simply the *maximum expected utility* over all possible actions from s. Computing maximum expected utility for a state s is essentially the same as running expectimax - we first compute the expected utility from each q-state (s,a) (equivalent to computing the value of chance nodes), then compute the maximum over these nodes to compute the maximum expected utility (equivalent to computing the value of a maximizer node).

One final note on the Bellman equation – its usage is as a *condition* for optimality. In other words, if we can somehow determine a value $V(s)$ for every state $s \in S$ such that the Bellman equation holds true for each of these states, we can conclude that these values are the optimal values for their respective states. Indeed, satisfying this condition implies $\forall s \in S,\ V(s) = V^*(s)$.

Value Iteration

Now that we have a framework to test for optimality of the values of states in a MDP, the natural follow-up question to ask is how to actually compute these optimal values. To answer this question, we need **time-limited values** (the natural result of enforcing finite horizons). The time-limited value for a state s with a time-limit of k timesteps is denoted $V_k(s)$, and represents the maximum expected utility attainable from s given that the Markov decision process under consideration terminates in k timesteps. Equivalently, this is what a depth-k expectimax run on the search tree for a MDP returns.

Value iteration is a **dynamic programming algorithm** that uses an iteratively longer time limit to compute time-limited values until convergence (that is, until the V values are the same for each state as they were in the past iteration: $\forall s, V_{k+1}(s) = V_k(s)$). It operates as follows:

1. $\forall s \in S$, initialize $V_0(s) = 0$. This should be intuitive, since setting a time limit of 0 timesteps means no actions can be taken before termination, and so no rewards can be acquired.

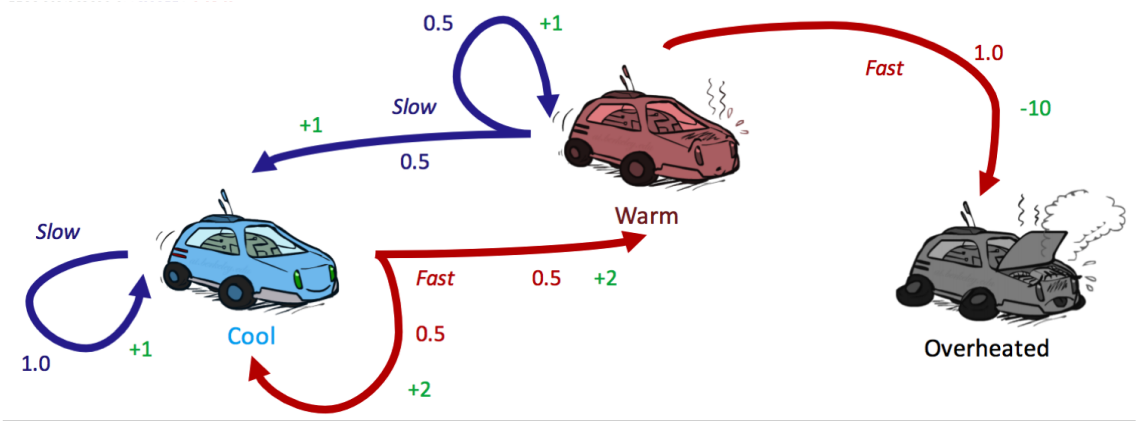
2. Repeat the following update rule until convergence:

$$\forall s \in S,\ V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V_k(s')]$$

   At iteration k of value iteration, we use the time-limited values for with limit k for each state to generate the time-limited values with limit $(k+1)$. In essence, we use computed solutions to subproblems (all the $V_k(s)$) to iteratively build up solutions to larger subproblems (all the $V_{k+1}(s)$); this is what makes value iteration a dynamic programming algorithm.

Note that though the Bellman equation looks essentially identical in construction to the update rule above, they are not the same. The Bellman equation gives a condition for optimality, while the update rule gives a method to iteratively update values until convergence. When convergence is reached, the Bellman equation will hold for every state: $\forall s \in S,\ V_k(s) = V_{k+1}(s) = V^*(s)$.

Let's see a few updates of value iteration in practice by revisiting our racecar MDP from earlier, introducing a discount factor of $\gamma = 0.5$:



We begin value iteration by initialization of all $V_0(s) = 0$:

	cool	**warm**	**overheated**
V_0	0	0	0

In our first round of updates, we can compute $\forall s \in S,\ V_1(s)$ as follows:

$$\begin{aligned}
V_1(cool) &= \max\{1\cdot[1+0.5\cdot 0],\ 0.5\cdot[2+0.5\cdot 0]+0.5\cdot[2+0.5\cdot 0]\} \\
&= \max\{1,2\} \\
&= \boxed{2} \\
V_1(warm) &= \max\{0.5\cdot[1+0.5\cdot 0]+0.5\cdot[1+0.5\cdot 0],\ 1\cdot[-10+0.5\cdot 0]\} \\
&= \max\{1,-10\} \\
&= \boxed{1} \\
V_1(overheated) &= \max\{\} \\
&= \boxed{0}
\end{aligned}$$

	cool	**warm**	**overheated**
V_0	0	0	0
V_1	2	1	0

Similarly, we can repeat the procedure to compute a second round of updates with our newfound values for

$V_1(s)$ to compute $V_2(s)$.

$$\begin{aligned}
V_2(cool) &= \max\{1\cdot[1+0.5\cdot 2],\ 0.5\cdot[2+0.5\cdot 2]+0.5\cdot[2+0.5\cdot 1]\} \\
&= \max\{2, 2.75\} \\
&= \boxed{2.75} \\
V_2(warm) &= \max\{0.5\cdot[1+0.5\cdot 2]+0.5\cdot[1+0.5\cdot 1],\ 1\cdot[-10+0.5\cdot 0]\} \\
&= \max\{1.75, -10\} \\
&= \boxed{1.75} \\
V_2(overheated) &= \max\{\} \\
&= \boxed{0}
\end{aligned}$$

	cool	**warm**	**overheated**
V_0	0	0	0
V_1	2	1	0
V_2	2.75	1.75	0

It's worthwhile to observe that $V^*(s)$ for any terminal state must be 0, since no actions can ever be taken from any terminal state to reap any rewards.

Policy Extraction

Recall that our ultimate goal in solving a MDP is to determine an optimal policy. This can be done once all optimal values for states are determined using a method called **policy extraction**. The intuition behind policy extraction is very simple: if you're in a state s, you should take the action a which yields the maximum expected utility. Not surprisingly, a is the action which takes us to the q-state with maximum q-value, allowing for a formal definition of the optimal policy:

$$\forall s \in S,\ \pi^*(s) = \operatorname*{argmax}_a Q^*(s,a) = \operatorname*{argmax}_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$$

It's useful to keep in mind for performance reasons that it's better for policy extraction to have the optimal q-values of states, in which case a single argmax operation is all that is required to determine the optimal action from a state. Storing only each $V^*(s)$ means that we must recompute all necessary q-values with the Bellman equation before applying argmax, equivalent to performing a depth-1 expectimax.

Policy Iteration

Value iteration can be quite slow. At each iteration, we must update the values of all $|S|$ states (where $|n|$ refers to the cardinality operator), each of which requires iteration over all $|A|$ actions as we compute the q-value for each action. The computation of each of these q-values, in turn, requires iteration over each of the $|S|$ states again, leading to a poor runtime of $O(|S|^2|A|)$. Additionally, when all we want to determine is the optimal policy for the MDP, value iteration tends to do a lot of overcomputation since the policy as computed by policy extraction generally converges significantly faster than the values themselves. The fix for these flaws is to use **policy iteration** as an alternative, an algorithm that maintains the optimality of value iteration while providing significant performance gains. Policy iteration operates as follows:

1. Define an *initial policy*. This can be arbitrary, but policy iteration will converge faster the closer the initial policy is to the eventual optimal policy.

2. Repeat the following until convergence:

   - Evaluate the current policy with **policy evaluation**. For a policy π, policy evaluation means computing $V^{\pi}(s)$ for all states s, where $V^{\pi}(s)$ is expected utility of starting in state s when following π:

     $$V^{\pi}(s) = \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V^{\pi}(s')]$$

     Define the policy at iteration i of policy iteration as π_i. Since we are fixing a single action for each state, we no longer need the max operator which effectively leaves us with a system of $|S|$ equations generated by the above rule. Each $V^{\pi_i}(s)$ can then be computed by simply solving this system. Alternatively, we can also compute $V^{\pi_i}(s)$ by using the following update rule until convergence, just like in value iteration:

     $$V^{\pi_i}_{k+1}(s) \leftarrow \sum_{s'} T(s, \pi_i(s), s')[R(s, \pi_i(s), s') + \gamma V^{\pi_i}_{k}(s')]$$

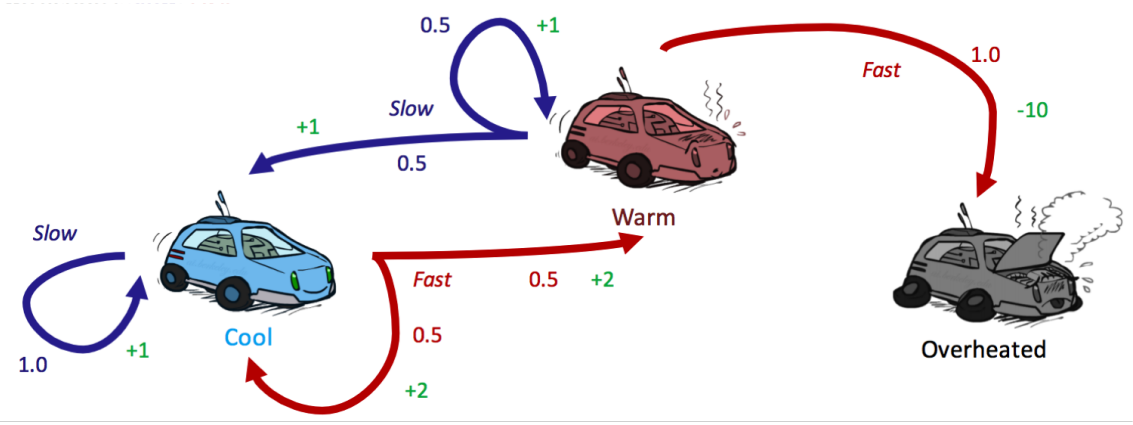
     However, this second method is typically slower in practice.

   - Once we've evaluated the current policy, use **policy improvement** to generate a better policy. Policy improvement uses policy extraction on the values of states generated by policy evaluation to generate this new and improved policy:

     $$\pi_{i+1}(s) = \operatorname*{argmax}_{a} \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V^{\pi_i}(s')]$$

     If $\pi_{i+1} = \pi_i$, the algorithm has converged, and we can conclude that $\pi_{i+1} = \pi_i = \pi^*$.

Let's run through our racecar example one last time (getting tired of it yet?) to see if we get the same policy using policy iteration as we did with value iteration. Recall that we were using a discount factor of $\gamma = 0.5$.



We start with an initial policy of *Always go slow*:

	cool	**warm**	**overheated**
π_0	*slow*	*slow*	–

Because terminal states have no outgoing actions, no policy can assign a value to one. Hence, it's reasonable to disregard the state *overheated* from consideration as we have done, and simply assign $\forall i,\ V^{\pi_i}(s) = 0$ for

any terminal state s. The next step is to run a round of policy evaluation on π_0:

$$\begin{aligned} V^{\pi_0}(cool) &= 1 \cdot [1 + 0.5 \cdot V^{\pi_0}(cool)] \\ V^{\pi_0}(warm) &= 0.5 \cdot [1 + 0.5 \cdot V^{\pi_0}(cool)] + 0.5 \cdot [1 + 0.5 \cdot V^{\pi_0}(warm)] \end{aligned}$$

Solving this system of equations for $V^{\pi_0}(cool)$ and $V^{\pi_0}(warm)$ yields:

	cool	**warm**	**overheated**
V^{π_0}	2	2	0

We can now run policy extraction with these values:

$$\begin{aligned} \pi_1(cool) &= \text{argmax}\{slow : 1 \cdot [1 + 0.5 \cdot 2],\ fast : 0.5 \cdot [2 + 0.5 \cdot 2] + 0.5 \cdot [2 + 0.5 \cdot 2]\} \\ &= \text{argmax}\{slow : 2,\ fast : 3\} \\ &= \boxed{fast} \\ \pi_1(warm) &= \text{argmax}\{slow : 0.5 \cdot [1 + 0.5 \cdot 2] + 0.5 \cdot [1 + 0.5 \cdot 2],\ fast : 1 \cdot [-10 + 0.5 \cdot 0]\} \\ &= \text{argmax}\{slow : 3,\ fast : -10\} \\ &= \boxed{slow} \end{aligned}$$

Running policy iteration for a second round yields $\pi_2(cool) = fast$ and $\pi_2(warm) = slow$. Since this is the same policy as π_1, we can conclude that $\pi_1 = \pi_2 = \pi^*$. Verify this for practice!

	cool	**warm**
π_0	*slow*	*slow*
π_1	*fast*	*slow*
π_2	*fast*	*slow*

This example shows the true power of policy iteration: with only two iterations, we've already arrived at the optimal policy for our racecar MDP! This is more than we can say for when we ran value iteration on the same MDP, which was still several iterations from convergence after the two updates we performed.

Summary

The material presented above has much opportunity for confusion. We covered value iteration, policy iteration, policy extraction, and policy evaluation, all of which look similar, using the Bellman equation with subtle variation. Below is a summary of when to use each algorithm:

- *Value iteration*: Used for computing the optimal values of states, by iterative updates until convergence.
- *Policy evaluation*: Used for computing the values of states under a specific policy.
- *Policy extraction*: Used for determining a policy given some state value function. If the state values are optimal, this policy will be optimal. This method is used after running value iteration, to compute an optimal policy from the optimal state values; or as a subroutine in policy iteration, to compute the best policy for the currently estimated state values.

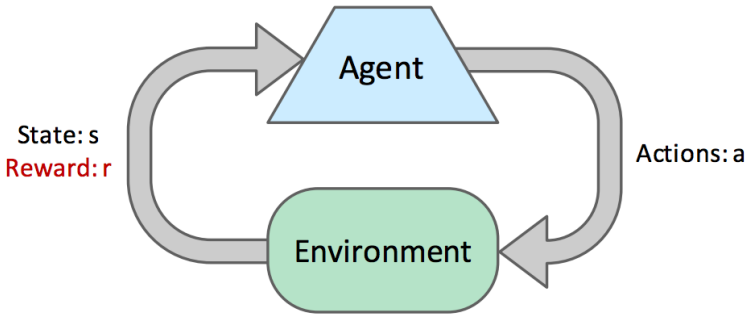
- *Policy iteration*: A technique that encapsulates both policy evaluation and policy extraction and is used for iterative convergence to an optimal policy. It tends to outperform value iteration, by virtue of the fact that policies usually converge much faster than the values of states.

CS 188 Introduction to Artificial Intelligence
Fall 2018 Note 5

These lecture notes are heavily based on notes originally written by Nikhil Sharma.

Reinforcement Learning

In the previous note, we discussed Markov decision processes, which we solved using techniques such as value iteration and policy iteration to compute the optimal values of states and extract optimal policies. Solving Markov decision processes is an example of **offline planning**, where agents have full knowledge of both the transition function and the reward function, all the information they need to precompute optimal actions in the world encoded by the MDP without ever actually taking any actions. In this note, we'll discuss **online planning**, during which an agent has no prior knowledge of rewards or transitions in the world (still represented as a MDP). In online planning, an agent must try **exploration**, during which it performs actions and receives **feedback** in the form of the successor states it arrives in and the corresponding rewards it reaps. The agent uses this feedback to estimate an optimal policy through a process known as **reinforcement learning** before using this estimated policy for **exploitation**, or reward maximization.



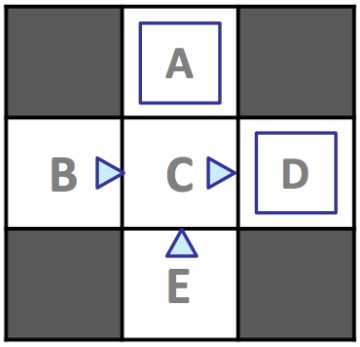
Let's start with some basic terminology. At each timestep during online planning, an agent starts in a state s, then takes an action a and ends up in a successor state s', attaining some reward r. Each (s, a, s', r) tuple is known as a **sample**. Often, an agent continues to take actions and collect samples in succession until arriving at a terminal state. Such a collection of samples is known as an **episode**. Agents typically go through many episodes during exploration in order to collect sufficient data needed for learning.

There are two types of reinforcement learning, **model-based learning** and **model-free learning**. Model-based learning attempts to estimate the transition and reward functions with the samples attained during exploration before using these estimates to solve the MDP normally with value or policy iteration. Model-free learning, on the other hand, attempts to estimate the values or q-values of states directly, without ever using any memory to construct a model of the rewards and transitions in the MDP.

Model-Based Learning

In model-based learning an agent generates an approximation of the transition function, $\hat{T}(s, a, s')$, by keeping counts of the number of times it arrives in each state s' after entering each q-state (s, a). The agent can

then generate the the approximate transition function $\hat{T}$ upon request by **normalizing** the counts it has collected - dividing the count for each observed tuple (s,a,s') by the sum over the counts for all instances where the agent was in q-state (s,a). Normalization of counts scales them such that they sum to one, allowing them to be interpreted as probabilities. Consider the following example MDP with states $S = \{A,B,C,D,E,x\}$, with x representing the terminal state, and discount factor $\gamma = 1$:



Assume we allow our agent to explore the MDP for four episodes under the policy $\pi_{explore}$ delineated above (a directional triangle indicates motion in the direction the triangle points, and a blue squares represents taking *exit* as the action of choice), and yield the following results:

Episode 1

B, east, C, -1
C, east, D, -1
D, exit, x, +10

Episode 2

B, east, C, -1
C, east, D, -1
D, exit, x, +10

Episode 3

E, north, C, -1
C, east, D, -1
D, exit, x, +10

Episode 4

E, north, C, -1
C, east, A, -1
A, exit, x, -10

We now have a collective 12 samples, 3 from each episode with counts as follows:

s	a	s′	count
A	*exit*	x	1
B	*east*	C	2
C	*east*	A	1
C	*east*	D	3
D	*exit*	x	3
E	*north*	C	2

Recalling that $T(s,a,s') = P(s'|a,s)$, we can estimate the transition function with these counts by dividing the counts for each tuple (s,a,s') by the total number of times we were in q-state (s,a) and the reward function directly from the rewards we reaped during exploration:

- **Transition Function**: $\hat{T}(s,a,s')$
  - $\hat{T}(A,exit,x) = \frac{\#(A,exit,x)}{\#(A,exit)} = \frac{1}{1} = 1$
  - $\hat{T}(B,east,C) = \frac{\#(B,east,C)}{\#(B,east)} = \frac{2}{2} = 1$
  - $\hat{T}(C,east,A) = \frac{\#(C,east,A)}{\#(C,east)} = \frac{1}{4} = 0.25$
  - $\hat{T}(C,east,D) = \frac{\#(C,east,D)}{\#(C,east)} = \frac{3}{4} = 0.75$
  - $\hat{T}(D,exit,x) = \frac{\#(D,exit,x)}{\#(D,exit)} = \frac{3}{3} = 1$
  - $\hat{T}(E,north,C) = \frac{\#(E,north,C)}{\#(E,north)} = \frac{2}{2} = 1$
- **Reward Function**: $\hat{R}(s,a,s')$
  - $\hat{R}(A,exit,x) = -10$
  - $\hat{R}(B,east,C) = -1$
  - $\hat{R}(C,east,A) = -1$
  - $\hat{R}(C,east,D) = -1$
  - $\hat{R}(D,exit,x) = +10$
  - $\hat{R}(E,north,C) = -1$

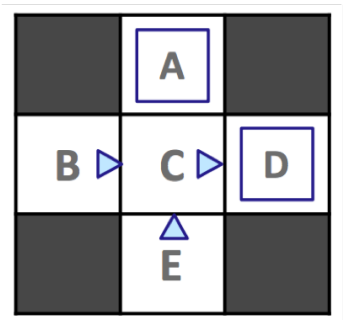
By the **law of large numbers**, as we collect more and more samples by having our agent experience more episodes, our models of $\hat{T}$ and $\hat{R}$ will improve, with $\hat{T}$ converging towards T and $\hat{R}$ acquiring knowledge of previously undiscovered rewards as we discover new (s,a,s') tuples. Whenever we see fit, we can end our agent's training to generate a policy $\pi_{exploit}$ by running value or policy iteration with our current models for $\hat{T}$ and $\hat{R}$ and use $\pi_{exploit}$ for exploitation, having our agent traverse the MDP taking actions seeking reward maximization rather than seeking learning. We'll soon discuss methods for how to allocate time between exploration and explotation effectively. Model-based learning is very simple and intuitive yet remarkably effective, generating $\hat{T}$ and $\hat{R}$ with nothing more than counting and normalization. However, it can be expensive to maintain counts for every (s,a,s') tuple seen, and so in the next section on model-free learning we'll develop methods to bypass maintaining counts altogether and avoid the memory overhead required by model-based learning.

Model-Free Learning

Onward to model-free learning! There are several model-free learning algorithms, and we'll cover three of them: direct evaluation, temporal difference learning, and Q-learning. Direct evaluation and temporal difference learning fall under a class of algorithms known as **passive reinforcement learning**. In passive reinforcement learning, an agent is given a policy to follow and learns the value of states under that policy as it experiences episodes, which is exactly what is done by policy evaluation for MDPs when T and R are known. Q-learning falls under a second class of model-free learning algorithms known as **active reinforcement learning**, during which the learning agent can use the feedback it receives to iteratively update its policy while learning until eventually determining the optimal policy after sufficient exploration.

Direct Evaluation

The first passive reinforcement learning technique we'll cover is known as **direct evaluation**, a method that's as boring and simple as the name makes it sound. All direct evaluation does is fix some policy π and have the agent that's learning experience several episodes while following π. As the agent collects samples through these episodes it maintains counts of the total utility obtained from each state and the number of times it visited each state. At any point, we can compute the estimated value of any state s by dividing the total utility obtained from s by the number of times s was visited. Let's run direct evaluation on our example from earlier, recalling that $\gamma = 1$.



Episode 1
B, east, C, -1
C, east, D, -1
D, exit, x, +10

Episode 2
B, east, C, -1
C, east, D, -1
D, exit, x, +10

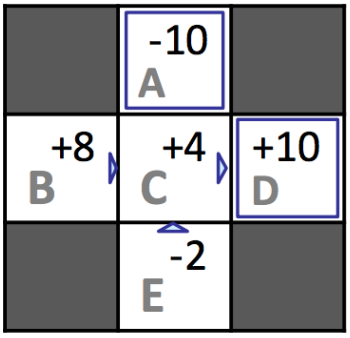
Episode 3
E, north, C, -1
C, east, D, -1
D, exit, x, +10

Episode 4
E, north, C, -1
C, east, A, -1
A, exit, x, -10

Walking through the first episode, we can see that from state D to termination we acquired a total reward of 10, from state C we acquired a total reward of $(-1)+10=9$, and from state B we acquired a total reward of $(-1)+(-1)+10=8$. Completing this process yields the total reward across episodes for each state and the resulting estimated values as follows:

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

Though direct evaluation eventually learns state values for each state, it's often unnecessarily slow to converge because it wastes information about transitions between states.



In our example, we computed $V^{\pi}(E)=-2$ and $V^{\pi}(B)=8$, though based on the feedback we received both states only have C as a successor state and incur the same reward of -1 when transitioning to C. According to the Bellman equation, this means that both B and E should have the same value under π. However, of the 4 times our agent was in state C, it transitioned to D and reaped a reward of 10 three times and transitioned to A and reaped a reward of -10 once. It was purely by chance that the single time it received the -10 reward it started in state E rather than B, but this severely skewed the estimated value for E. With enough episodes, the values for B and E will converge to their true values, but cases like this cause the process to take longer than we'd like. This issue can be mitigated by choosing to use our second passive reinforcement learning algorithm, temporal difference learning.

Temporal Difference Learning

Temporal difference learning (TD learning) uses the idea of *learning from every experience*, rather than simply keeping track of total rewards and number of times states are visited and learning at the end as direct evaluation does. In policy evaluation, we used the system of equations generated by our fixed policy and the Bellman equation to determine the values of states under that policy (or used iterative updates like with value iteration).

$$V^{\pi}(s) = \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V^{\pi}(s')]$$

Each of these equations equates the value of one state to the weighted average over the discounted values of that state's successors plus the rewards reaped in transitioning to them. TD learning tries to answer the question of how to compute this weighted average without the weights, cleverly doing so with an **exponential moving average**. We begin by initializing $\forall s,\ V^{\pi}(s) = 0$. At each timestep, an agent takes an action $\pi(s)$ from a state s, transitions to a state s', and receives a reward $R(s, \pi(s), s')$. We can obtain a **sample value** by summing the received reward with the discounted current value of s' under π:

$$sample = R(s, \pi(s), s') + \gamma V^{\pi}(s')$$

This sample is a new estimate for $V^{\pi}(s)$. The next step is to incorporate this sampled estimate into our existing model for $V^{\pi}(s)$ with the exponential moving average, which adheres to the following update rule:

$$V^{\pi}(s) \leftarrow (1-\alpha)V^{\pi}(s) + \alpha \cdot sample$$

Above, α is a parameter constrained by $0 \leq \alpha \leq 1$ known as the **learning rate** that specifies the weight we want to assign our existing model for $V^{\pi}(s)$, $1-\alpha$, and the weight we want to assign our new sampled estimate, α. It's typical to start out with learning rate of $\alpha = 1$, accordingly assigning $V^{\pi}(s)$ to whatever the first *sample* happens to be, and slowly shrinking it towards 0, at which point all subsequent samples will be zeroed out and stop affecting our model of $V^{\pi}(s)$.

Let's stop and analyze the update rule for a minute. Annotating the state of our model at different points in time by defining $V_k^{\pi}(s)$ and $sample_k$ as the estimated value of state s after the k^{th} update and the k^{th} sample respectively, we can reexpress our update rule:

$$V_k^{\pi}(s) \leftarrow (1-\alpha)V_{k-1}^{\pi}(s) + \alpha \cdot sample_k$$

This recursive definition for $V_k^{\pi}(s)$ happens to be very interesting to expand:

$$\begin{aligned}
V_k^{\pi}(s) &\leftarrow (1-\alpha)V_{k-1}^{\pi}(s) + \alpha \cdot sample_k \\
V_k^{\pi}(s) &\leftarrow (1-\alpha)[(1-\alpha)V_{k-2}^{\pi}(s) + \alpha \cdot sample_{k-1}] + \alpha \cdot sample_k \\
V_k^{\pi}(s) &\leftarrow (1-\alpha)^2 V_{k-2}^{\pi}(s) + (1-\alpha) \cdot \alpha \cdot sample_{k-1} + \alpha \cdot sample_k \\
&\vdots \\
V_k^{\pi}(s) &\leftarrow (1-\alpha)^k V_0^{\pi}(s) + \alpha \cdot [(1-\alpha)^{k-1} \cdot sample_1 + \ldots + (1-\alpha) \cdot sample_{k-1} + sample_k] \\
V_k^{\pi}(s) &\leftarrow \alpha \cdot [(1-\alpha)^{k-1} \cdot sample_1 + \ldots + (1-\alpha) \cdot sample_{k-1} + sample_k]
\end{aligned}$$

Because $0 \leq (1-\alpha) \leq 1$, as we raise the quantity $(1-\alpha)$ to increasingly larger powers, it grows closer and closer to 0. By the update rule expansion we derived, this means that older samples are given exponentially less weight, exactly what we want since these older samples are computed using older (and hence worse) versions of our model for $V^{\pi}(s)$! This is the beauty of temporal difference learning - with a single straightfoward update rule, we are able to:

- learn at every timestep, hence using information about state transitions as we get them since we're using iteratively updating versions of $V^{\pi}(s')$ in our samples rather than waiting until the end to perform any computation.
- give exponentially less weight to older, potentially less accurate samples.
- converge to learning true state values much faster with fewer episodes than direct evaluation.

Q-Learning

Both direct evaluation and TD learning will eventually learn the true value of all states under the policy they follow. However, they both have a major inherent issue - we want to find an optimal *policy* for our agent, which requires knowledge of the q-values of states. To compute q-values from the values we have, we require a transition function and reward function as dictated by the Bellman equation.

$$Q^*(s,a) = \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$$

Resultingly, TD learning or direct evaluation are typically used in tandem with some model-based learning to acquire estimates of T and R in order to effectively update the policy followed by the learning agent. This became avoidable by a revolutionary new idea known as **Q-learning**, which proposed learning the q-values of states directly, bypassing the need to ever know any values, transition functions, or reward functions. As a result, Q-learning is entirely model-free. Q-learning uses the following update rule to perform what's known as **q-value iteration**:

$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma \max_{a'} Q_k(s',a')]$$

Note that this update is only a slight modification over the update rule for value iteration. Indeed, the only real difference is that the position of the max operator over actions has been changed since we select an action before transitioning when we're in a state, but we transition before selecting a new action when we're in a q-state.

With this new update rule under our belt, Q-learning is derived essentially the same way as TD learning, by acquiring **q-value samples**:

$$sample = R(s,a,s') + \gamma \max_{a'} Q(s',a')$$

and incoporating them into an exponential moving average.

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha \cdot sample$$

As long as we spend enough time in exploration and decrease the learning rate α at an appropriate pace, Q-learning learns the optimal q-values for every q-state. This is what makes Q-learning so revolutionary - while TD learning and direct evaluation learn the values of states under a policy by following the policy before determining policy optimality via other techniques, Q-learning can learn the optimal policy directly even by taking suboptimal or random actions. This is called **off-policy learning** (contrary to direct evaluation and TD learning, which are examples of **on-policy learning**).

Approximate Q-Learning

Q-learning is an incredible learning technique that continues to sit at the center of developments in the field of reinforcement learning. Yet, it still has some room for improvement. As it stands, Q-learning just stores

all q-values for states in tabular form, which is not particularly efficient given that most applications of reinforcement learning have several thousands or even millions of states. This means we can't visit all states during training and can't store all q-values even if we could for lack of memory.

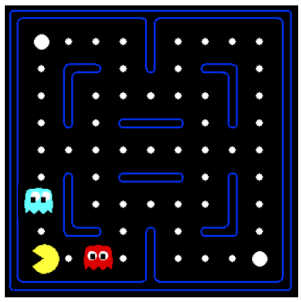


Figure 1

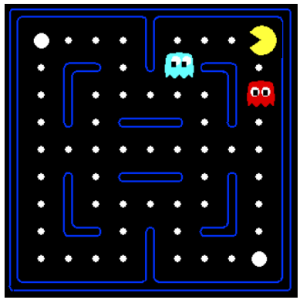


Figure 2

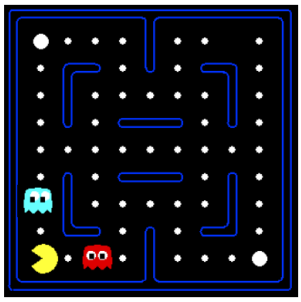


Figure 3

Above, if Pacman learned that Figure 1 is unfavorable after running vanilla Q-learning, it would still have no idea that Figure 2 or even Figure 3 are unfavorable as well. **Approximate Q-learning** tries to account for this by learning about a few general situations and extrapolating to many similar situations. The key to generalizing learning experiences is the **feature-based representation** of states, which represents each state as a vector known as a **feature vector**. For example, a feature vector for Pacman may encode

- the distance to the closest ghost.
- the distance to the closest food pellet.
- the number of ghosts.
- is Pacman trapped? 0 or 1

With feature vectors, we can treat values of states and q-states as **linear value functions**:

$$\begin{aligned} V(s) &= w_1 \cdot f_1(s) + w_2 \cdot f_2(s) + \ldots + w_n \cdot f_n(s) = \vec{w} \cdot \vec{f}(s) \\ Q(s,a) &= w_1 \cdot f_1(s,a) + w_2 \cdot f_2(s,a) + \ldots + w_n \cdot f_n(s,a) = \vec{w} \cdot \vec{f}(s,a) \end{aligned}$$

where $\vec{f}(s) = \begin{bmatrix} f_1(s) & f_2(s) & \ldots & f_n(s) \end{bmatrix}^T$ and $\vec{f}(s,a) = \begin{bmatrix} f_1(s,a) & f_2(s,a) & \ldots & f_n(s,a) \end{bmatrix}^T$ represent the feature vectors for state s and q-state (s,a) respectively and $\vec{w} = \begin{bmatrix} w_1 & w_2 & \ldots & w_n \end{bmatrix}$ represents a weight vector. Defining $difference$ as

$$difference = [R(s,a,s') + \gamma \max_{a'} Q(s',a')] - Q(s,a)$$

approximate Q-learning works almost identically to Q-learning, using the following update rule:

$$w_i \leftarrow w_i + \alpha \cdot difference \cdot f_i(s,a)$$

Rather than storing Q-values for each and every state, with approximate Q-learning we only need to store a single weight vector and can compute Q-values on-demand as needed. As a result, this gives us not only a more generalized version of Q-learning, but a significantly more memory-efficient one as well.

As a final note on Q-learning, we can reexpress the update rule for exact Q-learning using $difference$ as follows:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \cdot difference$$

This second notation gives us a slightly different but equally valuable interpration of the update: it's computing the difference between the sampled estimated and the current model of $Q(s,a)$, and shifting the model in the direction of the estimate with the magnitude of the shift being proportional to the magnitude of the difference.

Exploration and Exploitation

We've now covered several different methods for an agent to learn an optimal policy, and harped on the fact that "sufficient exploration" is necessary for this without really elaborating on what's really meant by "sufficient". In the upcoming two sections, we'll discuss two methods for distributing time between exploration and exploitation: ε-greedy policies and exploration functions.

ε-Greedy Policies

Agents following an **ε-greedy policy** define some probability $0 \leq \varepsilon \leq 1$, and act randomly and explore with probability ε. Accordingly, they follow their current established policy and exploit with probability $(1-\varepsilon)$. This is a very simple policy to implement, yet can still be quite difficult to handle. If a large value for ε is selected, then even after learning the optimal policy, the agent will still behave mostly randomly. Similarly, selecting a small value for ε means the agent will explore infrequently, leading Q-learning (or any other selected learning algorithm) to learn the optimal policy very slowly. To get around this, ε must be manually tuned and lowered over time to see results.

Exploration Functions

This issue of manually tuning ε is avoided by **exploration functions**, which use a modified q-value iteration update to give some preference to visiting less-visited states. The modified update is as follows:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha \cdot [R(s,a,s') + \gamma \max_{a'} f(s',a')]$$

where f denotes an exploration function. There exists some degree of flexibility in designing an exploration function, but a common choice is to use

$$f(s,a) = Q(s,a) + \frac{k}{N(s,a)}$$

with k being some predetermined value, and $N(s,a)$ denoting the number of times q-state (s,a) has been visited. Agents in a state s always select the action that has the highest $f(s,a)$ from each state, and hence never have to make a probabilistic decision between exploration and exploitation. Instead, exploration is automatically encoded by the exploration function, since the term $\frac{k}{N(s,a)}$ can give enough of a "bonus" to some infrequently-taken action such that it is selected over actions with higher q-values. As time goes on and states are visited more frequently, this bonus decreases towards 0 for each state and $f(s,a)$ regresses towards $Q(s,a)$, making exploitation more and more exclusive.

Summary

It's very important to remember that reinforcement learning has an underlying MDP, and the goal of reinforcement learning is to solve this MDP by deriving an optimal policy. The difference between using reinforcement learning and using methods like value iteration and policy iteration is the lack of knowledge of the transition function T and the reward function R for the underlying MDP. As a result, agents must *learn* the optimal policy through online trial-by-error rather than pure offline computation. There are many ways to do this:

- Model-based learning - Runs computation to estimate the values of the transition function T and the reward function R and uses MDP-solving methods like value or policy iteration with these estimates.
- Model-free learning - Avoids estimation of T and R, instead using other methods to directly estimate the values or q-values of states.
  - Direct evaluation - follows a policy π and simply counts total rewards reaped from each state and the total number of times each state is visited. If enough samples are taken, this converges to the true values of states under π, albeit being slow and wasting information about the transitions between states.
  - Temporal difference learning - follows a policy π and uses an exponential moving average with sampled values until convergence to the true values of states under π. TD learning and direct evaluation are examples of on-policy learning, which learn the values for a specific policy before deciding whether that policy is suboptimal and needs to be updated.
  - Q-Learning - learns the optimal policy directly through trial and error with q-value iteration updates. This an example of off-policy learning, which learns an optimal policy even when taking suboptimal actions.
  - Approximate Q-Learning - does the same thing as Q-learning but uses a feature-based representation for states to generalize learning.

CS 188 Introduction to Artificial Intelligence
Fall 2018 Note 6

These lecture notes are heavily based on notes originally written by Josh Hug and Jacky Liang.

Probabilistic Inference

In artificial intelligence, we often want to model the relationships between various nondeterministic events. If the weather predicts a 40% chance of rain, should I carry my umbrella? How many scoops of ice cream should I get if the more scoops I get, the more likely I am to drop it all? If there was an accident 15 minutes ago on the freeway on my route to Oracle Arena to watch the Warriors' game, should I leave now or in 30 minutes? All of these questions (and innumerable more) can be answered with **probabilistic inference**.

We're assuming that you've learned the foundations of probability in CS70, so these notes will not review basic concepts of probability like PDFs, conditional probabilities, independence, and conditional independence.

In previous sections of this class, we modeled the world as existing in a specific state that is always known. For the next several weeks, we will instead use a new model where each possible state for the world has its own probability. For example, we might build a weather model, where the state consists of the season, temperature and weather. Our model might say that $P(winter, 35^\circ, cloudy) = 0.023$. This number represents the probability of the specific outcome that it is winter, 35°, and cloudy.

More precisely, our model is a **joint distribution**, i.e. a table of probabilities which captures the likelihood of each possible **outcome**, also known as an **assignment**. As an example, consider the table below:

Season	Temperature	Weather	Probability
summer	hot	sun	0.30
summer	hot	rain	0.05
summer	cold	sun	0.10
summer	cold	rain	0.05
winter	hot	sun	0.10
winter	hot	rain	0.05
winter	cold	sun	0.15
winter	cold	rain	0.20

This model allows us to answer questions that might be of interest to us, for example:

- What is the probability that it is sunny? $P(W = sun)$
- What is the probability distribution for the weather, given that we know it is winter? $P(W \mid S = winter)$
- What is the probability that it is winter, given that we know it is rainy and cold? $P(S = winter \mid T = cold, W = rain)$
- What is the probability distribution for the weather and season give that we know that it is cold? $P(S, W \mid T = cold)$

Given a joint PDF, we can trivially perform compute any desired probablity distribution $P(Q_1 \dots Q_k \mid e_1 \dots e_k)$ using a simple and intuitive procedure known as **inference by enumeration**, for which we define three types of variables we will be dealing with:

1. **Query variables** Q_i, which are unknown and appear on the left side of the probability distribution we are trying to compute.
2. **Evidence variables** e_i, which are observed variables whose values are known and appear on the right side of the probability distribution we are trying to compute.
3. **Hidden variables**, which are values present in the overall joint distribution but not in the distribution we are currently trying to compute.

In this procedure, we collect all the rows consistent with the observed evidence variables, sum out all the hidden variables, and finally normalize the table so that it is a probability distribution (i.e. values sum to 1).

For example, if we wanted to compute $P(W \mid S = winter)$, we'd select the four rows where S is winter, then sum out over T and normalize. This yields the following probability table:

W	S	Unnormalized Sum	Probability
sun	winter	$0.10+0.15=0.25$	$0.25/(0.25+0.25)=0.5$
rain	winter	$0.05+0.20=0.25$	$0.25/(0.25+0.25)=0.5$

Hence $P(W = sun \mid S = winter) = 0.5$ and $P(W = rain \mid S = winter) = 0.5$, and we learn that in winter there's a 50% chance of sun and a 50% chance of rain (classic California weather).

So long as we have the joint PDF table, inference by enumeration (IBE) can be used to compute any desired probablity distribution, even for multiple query variables $Q_1 ... Q_k$.

Bayes Nets (Representation)

While inference by enumeration can compute probabilities for any query we might desire, representing an entire joint distribution in the memory of a computer is impractical for real problems - if each of n variables we wish to represent can take on d possible values (it has a **domain** of size d), then our joint distribution table will have d^n entries, exponential in the number of variables and quite impractical to store!

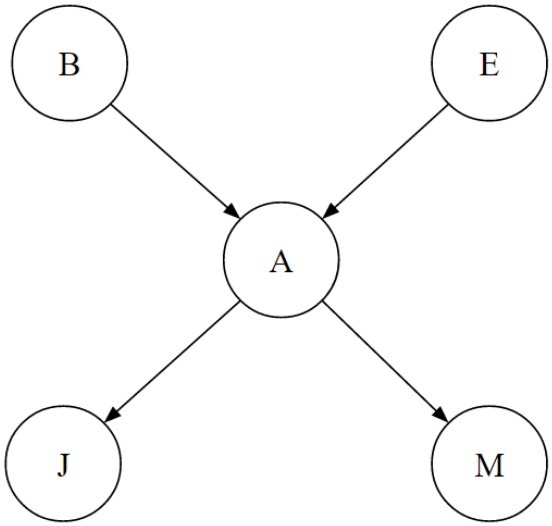
Bayes nets avoid this issue by taking advantage of the idea of conditional probability. Rather than storing information in a giant table, probabilities are instead distributed across a large number of smaller local probability tables along with a **directed acyclic graph** (DAG) which captures the relationships between variables. The local probability tables and the DAG together encode enough information to compute any probability distribution that we could have otherwise computed given the entire joint distribution.

Specifically, each node in the graph represents a single random variable and each directed edge represents one of the conditional probability distributions we choose to store (i.e. an edge from node A to node B indicates that we store the probability table for $P(B|A)$). **Each node is conditionally independent of all its ancestor nodes in the graph, given all of its parents**. Thus, if we have a node representing variable X, we store $P(X|A_1, A_2, ..., A_N)$, where $A_1, ..., A_N$ are the parents of X.

As an example of a Bayes Net, consider a model where we have five binary random variables described below:

- B: Burglary occurs.
- A: Alarm goes off.
- E: Earthquake occurs.
- J: John calls.
- M: Mary calls.

Assume the alarm can go off if either a burglary or an earthquake occurs, and that Mary and John will call if they hear the alarm. We can represent these dependencies with the graph shown below.



As a reality check, it's important to internalize that Bayes Nets are only a type of model. Models attempt to capture the way the world works, but because they are always a simplification they are always wrong. However, with good modeling choices they can still be good enough approximations that they are useful for solving real problems in the real world. In general, they will not account for every variable or even every interaction between variables.

Returning to our discussion, we formally define a Bayes Net as consisting of:

- A directed acyclic graph of nodes, one per variable X.
- A conditional distribution for each node $P(X|A_1 \dots A_n)$, where A_i is the i^{th} parent of X, stored as a **conditional probability table** or CPT. Each CPT has $n+2$ columns: one for the values of each of the n parent variables $A_1 \dots A_n$, one for the values of X, and one for the conditional probability of X.

In the alarm model above, we would store probability tables $P(B), P(E), P(A \mid B,E), P(J \mid A)$ and $P(M \mid A)$.

Given all of the CPTs for a graph, we can calculate the probability of a given assignment using the chain rule: $P(X_1, X_2, \dots, X_n) = \prod_{i=1}^{n} P(X_i | parents(X_i))$.

For the alarm model above, we might calculate the probability of one event as follows: $P(-b,-e,+a,+j,-m) = P(-b) \cdot P(-e) \cdot P(+a|-b,-e) \cdot P(+j|+a) \cdot P(-m|+a)$.

This works because of the conditional independence relationships given by the graph. Specifically, we rely on the fact that $P(x_i|x_1,\ldots,x_{i-1}) = P(x_i|parents(X_i))$. Or in other words, that the probability of a specific value of X_i depends only on the values assigned to X_i's parents.

Bayes Nets (Inference)

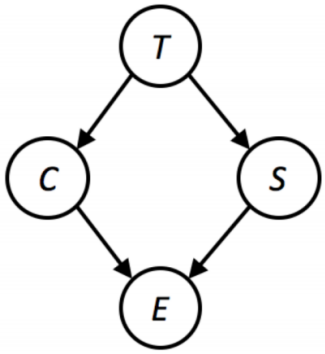
Inference is the process of calculating the joint PDF for some set of query variables based on some set of observed variables. We can solve this problem naively by forming the joint PDF and using inference by enumeration as described above. This requires the creation of and iteration over an exponentially large table.

An alternate approach is to **eliminate** variables one by one. To eliminate a variable X, we:

1. Join (multiply together) all factors involving X.
2. Sum out X.

A **factor** is defined simply as an *unnormalized probability*. At all points during variable elimination, each factor will be proportional to the probability it corresponds to but the underlying distribution for each factor won't necessarily sum to 1 as a probability distribution should.

Let's make these ideas more concrete with an example. Suppose we have a model as shown below, where T, C, S, and E can take on binary values, as shown below. Here, T represents the chance that an adventurer takes a treasure, C represents the chance that a cage falls on the adventurer given that he takes the treasure, S represents the chance that snakes are released if an adventurer takes the treasure, and E represents the chance that the adventurer escapes given information about the status of the cage and snakes.



In this case, we have the factors $P(T)$, $P(C|T)$, $P(S|T)$, and $P(E|C,S)$. Suppose we want to calculate $P(T|+e)$. The inference by enumeration approach would be to form the 16 row joint PDF $P(T,C,S,E)$, select only the rows corresponding to +e, then summing out C and S and finally normalizing.

The alternate approach is to eliminate C, then S, one variable at a time. We'd proceed as follows:

- Join (multiply) all the factors involving C, forming $P(C,+e|T,S) = P(C|T) \cdot P(+e|C,S)$.
- Sum out C from this new factor, leaving us with a new factor $P(+e|T,S)$.
- Join all factors involving S, forming $P(+e,S|T) = P(S|T) \cdot P(+e|T,S)$.

- Sum out S, yielding $P(+e|T)$.

Once we have $P(+e|T)$, we can easily compute $P(T|+e)$.

While this process is more involved from a conceptual point of view, the maximum size of any factor generated is only 8 rows instead of 16 as it would be if we formed the entire joint PDF.

An alternate way of looking at the problem is to observe that the calculation of $P(+e,T)$ can either be done, as it is in inference by enumeration, as follows:

$$\sum_s \sum_c P(T)P(s|T)P(c|T)P(+e|c,s)$$

Variable elimination is equivalent to calculating $P(+e,T)$ as follows:

$$P(T)\sum_s P(s|T)\sum_c P(c|T)P(+e|c,s)$$

Bayes Nets (Sampling)

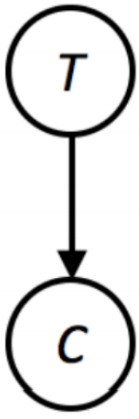
An alternate approach for probabilistic reasoning is to implicitly calculate the probabilities for our query by simply counting samples.

For example, suppose we wanted to calculate $P(T|+e)$. If we had a magic machine that could generate samples from our distribution, we could collect all samples for which the adventurer escapes the maze, and then compute the fraction of those escapes for which the adventurer also took the treasure. Put differently, if we could run simulations of say, a few million adventurers, we'd easily be able to compute any inference we'd want just by looking at the samples.

Given a Bayes Net model, we can easily write a simulator. For example, consider the CPTs given below for the simplified model with only two variables T and C.

T	P(T)
+t	0.99
-t	0.01

T	C	P(C|T)
+t	+c	0.95
+t	-c	0.05
-t	+c	0.0
-t	-c	1.0



A simple simulator in Python would be as follows:

```
import random

def get_t():
    if random.random() < 0.99:
        return True
    return False

def get_c(t):
    if t and random.random() < 0.95:
        return True
    return False

def get_sample():
    t = get_t()
    c = get_c(t)
    return [t, c]
```

We call this simple approach **prior sampling**. The downside of this approach is that it may require the generation of a very large number of samples in order to perform analysis of unlikely scenarios. If we wanted to compute $P(C|-t)$, we'd have to throw away 99% of our samples.

One way to mitigate this this problem, we can modify our procedure to early reject any sample inconsistent with our evidence. For example, for the query $P(C|-t)$, we'd avoid generating a value for C unless t is true. This still means we have to throw away most of our samples, but at least the bad samples we generate take less time to create. We call this approach **rejection sampling**.

These two approaches work for the same reason, which is that any valid sample occurs with the same probability as specified in the joint PDF. In other words, the probability of every sample is based on the product of every CPT, or as I personally call it, the "every CPT participates principle".

A more exotic approach is **likelihood weighting**, which ensures that we never generate a bad sample. In this approach, we manually set all variables equal to the evidence in our query. For example, if we wanted to compute $P(C|-t)$, we'd simply declare that t is false. The problem here is that this may yield samples that are inconsistent with the correct distribution. As an example, consider the more complex four variable model for T, C, S, and E given earlier in these notes. If we wanted to compute $P(T,S,+c,+e)$, and simply picked values for T and S without taking into account the fact that c = false, and e = true, then there's no guarantee that our samples actually obey the joint PDF given by the Bayes Net. For example, if the cage only ever falls if the treasure is taken, then we'd want to ensure that T is always true instead of using the $P(T)$ distribution given in the Bayes Net.

Put differently, if we simply force some variables equal to the evidence, then our samples occur with probability given only equal to the products of the CPTs of the non-evidence variables. This means the joint PDF has no guarantee of being correct (though may be for some cases like our two variable Bayes Net). Instead, if we have sampled variables Z_1 through Z_p and fixed evidence variables E_1 through E_m a sample is given by the probability $P(Z_1...Z_p,E_1...E_m) = \prod_i^p P(Z_i)|Parents(Z_i)$. What is missing is that the probability of a sample does not include all the probabilities of $P(E_i|Parents(E_i))$, i.e. not every CPT participates.

Likelihood weighting solves this issue by using a weight for each sample, which is the probability of the evidence variables given the sampled variables. That is, instead of counting all samples equally, we can

define a weight w_j for sample j that reflects how likely the observed values for the evidence variables are, given the sampled values. In this way, we ensure that every CPT participates. To do this, we iterate through each variable in the Bayes net, as we do for normal sampling), sampling a value if the variable is not an evidence variable, or changing the weight for the sample if the variable is evidence.

For example, suppose we want to calculate $P(T|+c,+e)$. For the jth sample, we'd perform the following algorithm:

- Set w_j to 1.0, and c = true and e = true.
- For T: This is not an evidence variable, so we sample t_j from $P(T)$.
- For C: This is an evidence variable, so we multiply the weight of the sample by $P(+c|t_j)$, i.e. $w_j = w_j \cdot P(+c|t_j)$.
- For S: sample s_j from $P(S \mid t_j)$.
- For E: multiply the weight of the sample by $P(+e|+c,s_j)$, i.e. $w_j = w_j \cdot P(+e|+c,s_j)$.

Then when we perform the usual counting process, we weight sample j by w_j instead of 1, where $0 <= w_j <= 1$. This approach works because in the final calculations for the probabilities, the weights effectively serve to replace the missing CPTs. In effect, we ensure that the weighted probability of each sample is given by $P(z_1 ... z_p, e_1 ... e_m) = [\prod_i^p P(z_i \mid Parents(z_i))] \cdot [\prod_i^m P(e_i) \mid Parents(e_i))]$.

For all three of our sampling methods (prior sampling, rejection sampling, and likelihod weighting), we can get increasing amounts of accuracy by generating additional samples. However, of the three, likelihood weighting is the most computationally efficient, for reasons beyond the scope of this course.

Gibbs Sampling is a fourth approach for sampling. In this approach, we first set all variables to some totally random value (not taking into account any CPTs). We then repeatedly pick one variable at a time, clear its value, and resample it given the values currently assigned to all other variables.

For the T,C,S,E example above, we might assign t = true, c = true, s = false, and e = true. We then pick one of our four variables to resample, say S, and clear it. We then pick a new variable from the distribution $P(S|+t,+c,+e)$. This requires us knowing this conditional distribution. It turns out that we can easily compute the distribution of any single variable given all other variables. More specifically, $P(S|T,C,E)$ can be calculated only using the CPTs that connect S with its neighbors. Thus, in a typical Bayes Net, where most variables have only a small number of neighbors, we can precompute the conditional distributions for each variable given all of its neighbors in linear time.

We will not prove this, but if we repeat this process enough times, our later samples will eventually converge to the correct distribution even though we may start from a low-probability assignment of values. If you're curious, there are some caveats beyond the scope of the course that you can read about under the Failure Modes section of the Wikipedia article for Gibbs Sampling.

Bayes Nets (D-Separation)

One useful question to ask about a set of random variables is whether or not one variable is independent from another, or if one random variable is conditionally independent of another given a third random variable. Bayes' Nets representation of joint probability distributions gives us a way to quickly answer such questions by inspecting the topological structure of the graph.

We already mentioned that **a node is conditionally independent of all its ancestor nodes in the graph given all of its parents.**

We will present all three canonical cases of connected three-node two-edge Bayes' Nets, or triples, and the conditional independence relationships they express.

Causal Chains

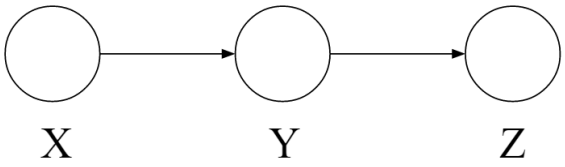


Figure 1: Causal Chain with no observations.

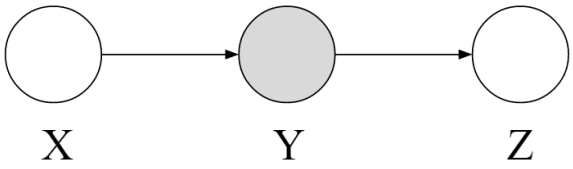


Figure 2: Causal Chain with Y observed.

Figure 1 is a configuration of three nodes known as a **causal chain**. It expresses the following representation of the joint distribution over X, Y, and Z:

$$P(x,y,z) = P(z|y)P(y|x)P(x)$$

It's important to note that X and Z are not guaranteed to be independent, as shown by the following counterexample:

$$P(y|x) = \begin{cases} 1 & \text{if } x = y \\ 0 & \text{else} \end{cases} \qquad P(z|y) = \begin{cases} 1 & \text{if } z = y \\ 0 & \text{else} \end{cases}$$

In this case, $P(z|x) = 1$ if $x = z$ and 0 otherwise, so X and Z are not independent.

However, we can make the statement that $X \perp\!\!\!\perp Z \mid Y$, as in Figure 2. Recall that this conditional indepdence means:

$$P(X|Z,Y) = P(X|Y)$$

We can prove this statement as follows:

$$\begin{aligned} P(X|Z,y) &= \frac{P(X,Z,y)}{P(Z,y)} = \frac{P(Z|y)P(y|X)P(X)}{\sum_x P(X,y,Z)} = \frac{P(Z|y)P(y|X)P(X)}{P(Z|y)\sum_x P(y|x)P(x)} \\ &= \frac{P(y|X)P(X)}{\sum_x P(y|x)P(x)} = \frac{P(y|X)P(X)}{P(y)} = P(X|y) \end{aligned}$$

An analogous proof can be used to show the same thing for the case where X has multiple parents. To summarize, in the causal chain chain configuration, $X \perp\!\!\!\perp Z \mid Y$.

Common Cause

Another possible configuration for a triple is the **common cause**. It expresses the following representation:

$$P(x,y,z) = P(x|y)P(z|y)P(y)$$

Just like with causal chain, we can show that X is not guaranteed to be independent of Z with the following counterexample distribution:

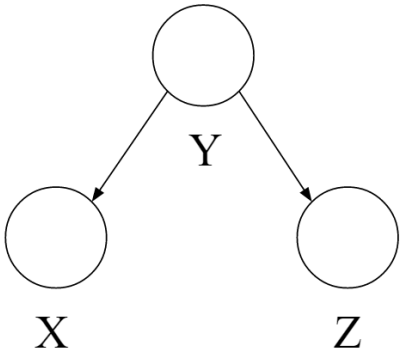


Figure 3: Common Cause with no observations.

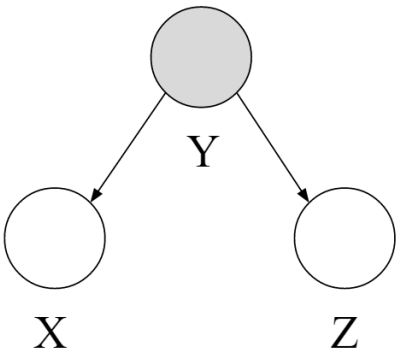


Figure 4: Common Cause with Y observed.

$$P(x|y) = \begin{cases} 1 & \text{if } x = y \\ 0 & \text{else} \end{cases}$$

$$P(z|y) = \begin{cases} 1 & \text{if } z = y \\ 0 & \text{else} \end{cases}$$

Then $P(x|z) = 1$ if $x = z$ and 0 otherwise, so X and Z are not independent.

But it is true that $X \perp\!\!\!\perp Z \mid Y$. That is, X and Z are independent if Y is observed as in Figure 4. We can show this as follows:

$$P(X|Z,y) = \frac{P(X,Z,y)}{P(Z,y)} = \frac{P(X|y)P(Z|y)P(y)}{P(Z|y)P(y)} = P(X|y)$$

Common Effect

The final possible configuration for a triple is the **common effect**, as shown in the figures below.

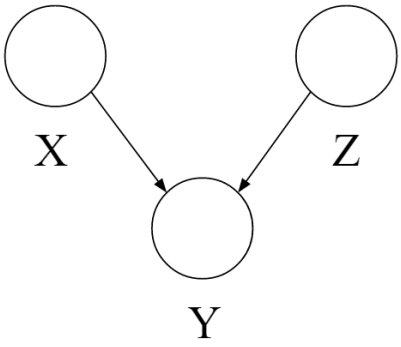


Figure 5: Common Effect with no observations.

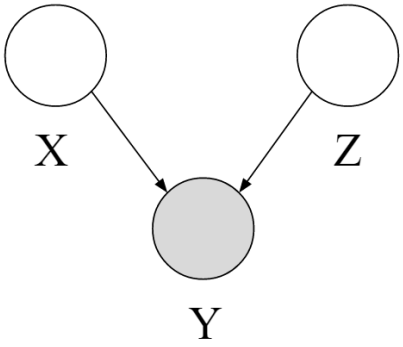


Figure 6: Common Effect with Y observed.

It expresses the representation:

$$P(x,y,z) = P(y|x,z)P(x)P(z)$$

In the configuration shown in Figure 5, X and Z are independent: $X \perp\!\!\!\perp Z$. However, they are not necessarily independent when conditioned on Y (Figure 6). As an example, suppose all three are binary variables. X

and Z are true and false with equal probability:

$$\begin{aligned} P(X = true) &= P(X = false) = 0.5 \\ P(Z = true) &= P(Z = false) = 0.5 \end{aligned}$$

and Y is determined by whether X and Z have the same value:

$$P(Y|X,Z) = \begin{cases} 1 & \text{if } X = Z \text{ and } Y = true \\ 1 & \text{if } X \neq Z \text{ and } Y = false \\ 0 & \text{else} \end{cases}$$

Then X and Z are independent if Y is unobserved. But if Y is observed, then knowing X will tell us the value of Z, and vice-versa. So X and Z are *not* conditionally independent given Y.

Common Effect can be viewed as "opposite" to Causal Chains and Common Cause – X and Z are guaranteed to be independent if Y is not conditioned on. But when conditioned on Y, X and Z may be dependent depending on the specific probability values for $P(Y \mid X,Z)$).

This same logic applies when conditioning on descendents of Y in the graph. If one of Y's descendent nodes is observed, as in Figure 7, X and Z are not guaranteed to be independent.

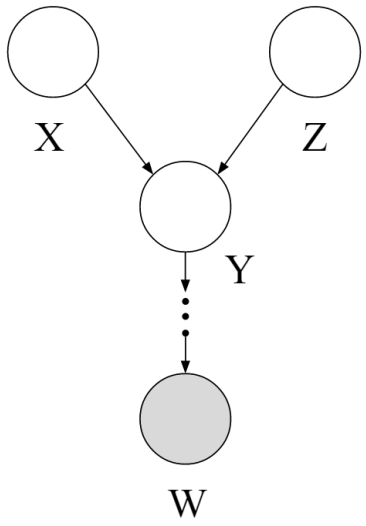


Figure 7: Common Effect with child observations.

General Case, and D-separation

We can use the previous three cases as building blocks to help us answer conditional independence questions on an arbitrary Bayes' Net with more than three nodes and two edges. We formulate the problem as follows:

Given a Bayes Net G, two nodes X and Y, and a (possibly empty) set of nodes $\{Z_1, \ldots Z_k\}$ that represent observed variables, must the following statement be true: $X \perp\!\!\!\perp Y | \{Z_1, \ldots Z_k\}$?

D-separation (directed separation) is a property of the structure of the Bayes Net graph that implies this conditional independence relationship, and generalizes the cases we've seen above. If a set of variables $Z_1, \cdots Z_k$ d-separates X and Y, then $X \perp\!\!\!\perp Y \mid \{Z_1, \cdots Z_k\}$ in all possible distributions that can be encoded by the Bayes net.

We start with an algorithm that is based on a notion of reachability from node X to node Y. (**Note: this algorithm is not quite correct! We'll see how to fix it in a moment.**)

1. Shade all observed nodes $\{Z_1, \ldots Z_k\}$ in the graph.
2. If there exists an undirected path from X and Y that is not blocked by a shaded node, X and Y are "connected".
3. If X and Y are connected, they're not conditionally independent given $\{Z_1, \ldots Z_k\}$. Otherwise, they are.

However, this algorithm only works if the Bayes' Net has no Common Effect structure within the graph, because if it exists, then two nodes are "reachable" when the Y node in Common Effect is activated (observed). To adjust for this, we arrive at the following **d-separation algorithm**:

1. Shade all observed nodes $\{Z_1, \ldots, Z_k\}$ in the graph.
2. Enumerate all undirected paths from X to Y.
3. For each path:
   (a) Decompose the path into triples (segments of 3 nodes).
   (b) If all triples are active, this path is active and *d-connects* X to Y.
4. If no path d-connects X and Y, then X and Y are d-separated, so they are conditionally independent given $\{Z_1, \ldots, Z_k\}$

Any path in a graph from X to Y can be decomposed into a set of 3 consecutive nodes and 2 edges - each of which is called a triple. A triple is active or inactive depending on whether or not the middle node is observed. If all triples in a path are active, then the path is active and *d-connects* X to Y, meaning X is not guaranteed to be conditionally independent of Y given the observed nodes. If all paths from X to Y are inactive, then X and Y are conditionally independent given the observed nodes.

Active triples: We can enumerate all possibilities of active and inactive triples using the three canonical graphs we presented above in Figure 8 and 9.

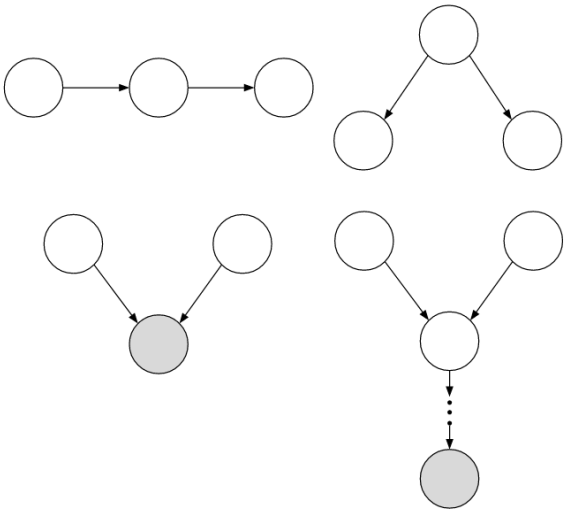


Figure 8: Active triples

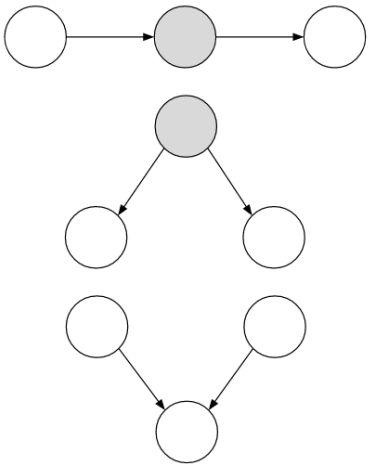
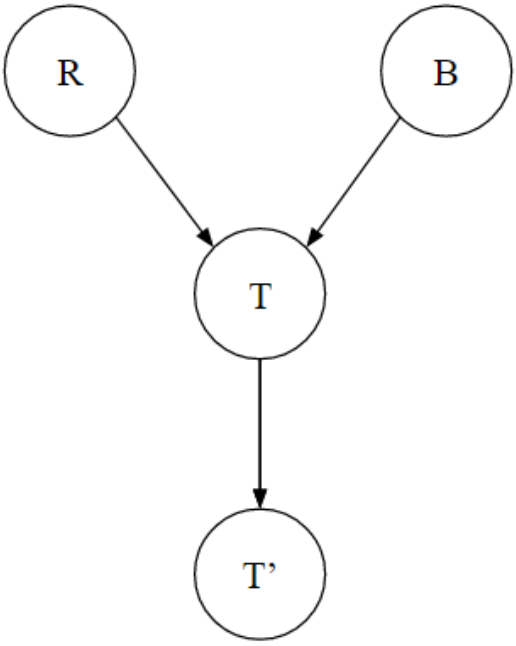


Figure 9: Inactive triples

Examples

Here are some examples of applying the *d*-separation algorithm:



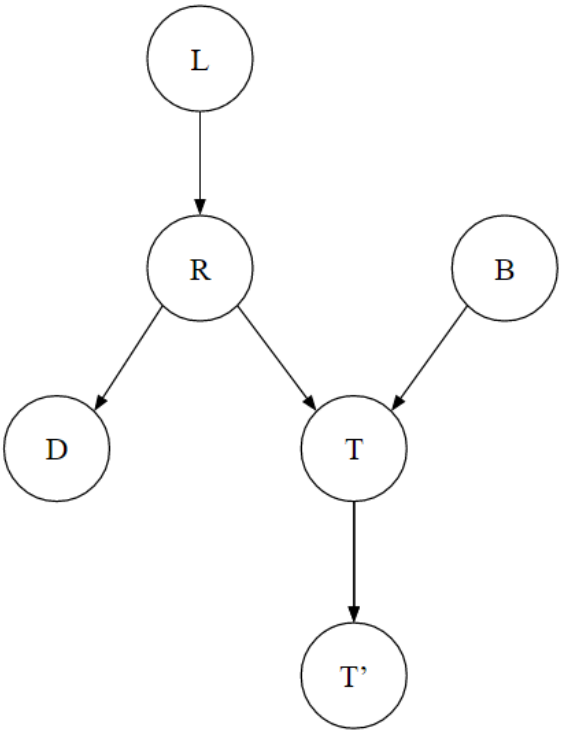
This graph contains the common effect and causual chain canonical graphs.

a) $R \perp\!\!\!\perp B$ – Guaranteed

b) $R \perp\!\!\!\perp B \mid T$ – Not guaranteed

c) $R \perp\!\!\!\perp B \mid T'$ – Not guaranteed

d) $R \perp\!\!\!\perp T' \mid T$ – Guaranteed



This graph contains combinations of all three canonical graphs (can you list them all?).

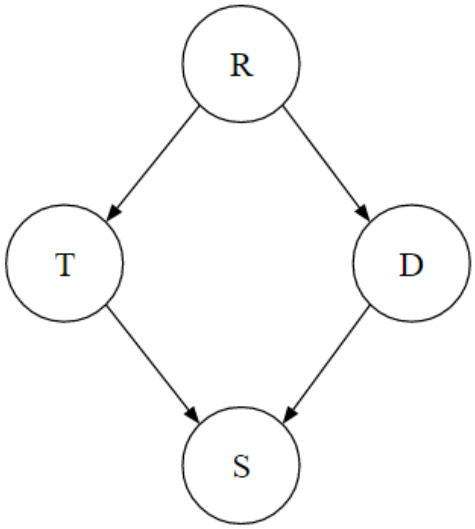
a) $L \perp\!\!\!\perp T' \mid T$ – Guaranteed

b) $L \perp\!\!\!\perp B$ – Guaranteed

c) $L \perp\!\!\!\perp B \mid T$ – Not guaranteed

d) $L \perp\!\!\!\perp B \mid T'$ – Not guaranteed

e) $L \perp\!\!\!\perp B \mid T, R$ – Guaranteed



This graph contains combinations of all three canonical graphs.

a) $T \perp\!\!\!\perp D$ – Not guaranteed

b) $T \perp\!\!\!\perp D \mid R$ – Guaranteed

c) $T \perp\!\!\!\perp D \mid R, S$ – Not guaranteed

Conclusion

To summarize, Bayes' Nets is a powerful representation of joint probability distributions. Its topological structure encodes independence and conditional independence relationships, and we can use it to model arbitrary distributions to perform inference and sampling.

CS 188 Introduction to Artificial Intelligence
Fall 2018 Note 7

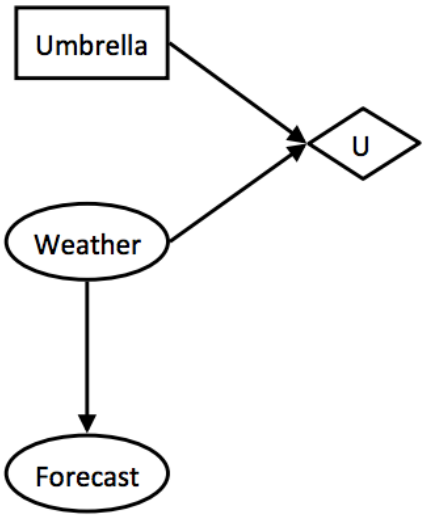
These lecture notes are heavily based on notes originally written by Nikhil Sharma.

Decision Networks

In the third note, we learned about game trees and algorithms such as minimax and expectimax which we used to determine optimal actions that maximized our expected utility. Then in the sixth note, we discussed Bayes' nets and how we can use evidence we know to run probabilistic inference to make predictions. Now we'll discuss a combination of both Bayes' nets and expectimax known as a **decision network** that we can use to model the effect of various actions on utilities based on an overarching graphical probabilistic model. Let's dive right in with the anatomy of a decision network:

- **Chance nodes** - Chance nodes in a decision network behave identically to Bayes' nets. Each outcome in a chance node has an associated probability, which can be determined by running inference on the underlying Bayes' net it belongs to. We'll represent these with ovals.
- **Action nodes** - Action nodes are nodes that we have complete control over; they're nodes representing a choice between any of a number of actions which we have the power to choose from. We'll represent action nodes with rectangles.
- **Utility nodes** - Utility nodes are children of some combination of action and chance nodes. They output a utility based on the values taken on by their parents, and are represented as diamonds in our decision networks.

Consider a situation when you're deciding whether or not to take an umbrella when you're leaving for class in the morning, and you know there's a forecasted 30% chance of rain. Should you take the umbrella? If there was a 80% chance of rain, would your answer change? This situation is ideal for modeling with a decision network, and we do it as follows:



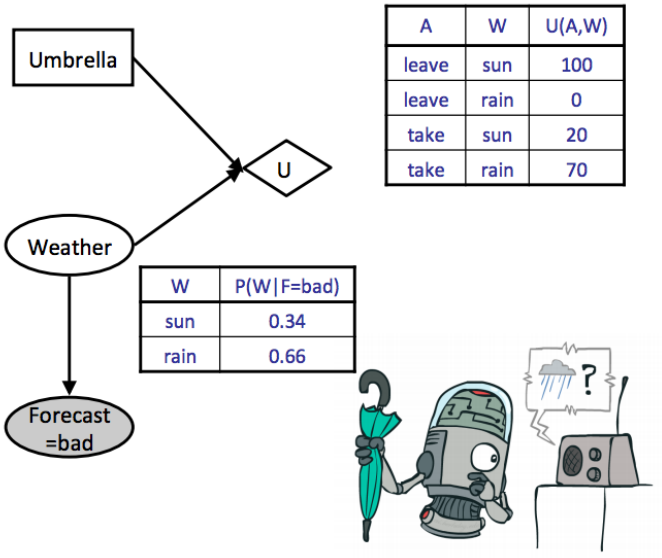
As we've done throughout this course with the various modeling techniques and algorithms we've discussed, our goal with decision networks is again to select the action which yields the **maximum expected utility** (MEU). This can be done with a fairly straightforward and intuitive procedure:

- Start by instantiating all evidence that's known, and run inference to calculate the posterior probabilities of all chance node parents of the utility node into which the action node feeds.
- Go through each possible action and compute the expected utility of taking that action given the posterior probabilities computed in the previous step. The expected utility of taking an action a given evidence e and n chance nodes is computed with the following formula:

$$EU(a|e) = \sum_{x_1,\ldots,x_n} P(x_1,\ldots,x_n|e)U(a,x_1,\ldots,x_n)$$

  where each x_i represents a value that the i^{th} chance node can take on. We simply take a weighted sum over the utilities of each outcome under our given action with weights corresponding to the probabilities of each outcome.
- Finally, select the action which yielded the highest utility to get the MEU.

Let's see how this actually looks by calculating the optimal action (should we *leave* or *take* our umbrella) for our weather example, using both the conditional probability table for weather given a bad weather forecast (forecast is our evidence variable) and the utility table given our action and the weather:



Note that we have omitted the inference computation for the posterior probabilities $P(W|F = \text{bad})$, but we could compute these using any of the inference algorithms we discussed for Bayes Nets. Instead, here we simply assume the above table of posterior probabilities for $P(W|F = \text{bad})$ as given. Going through both our actions and computing expected utilities yields:

$$\begin{aligned} EU(\text{leave}|\text{bad}) &= \sum_w P(w|\text{bad})U(\text{leave},w) \\ &= 0.34 \cdot 100 + 0.66 \cdot 0 = \boxed{34} \end{aligned}$$

$$\begin{aligned} EU(\text{take}|\text{bad}) &= \sum_w P(w|\text{bad})U(\text{take},w) \\ &= 0.34 \cdot 20 + 0.66 \cdot 70 = \boxed{53} \end{aligned}$$

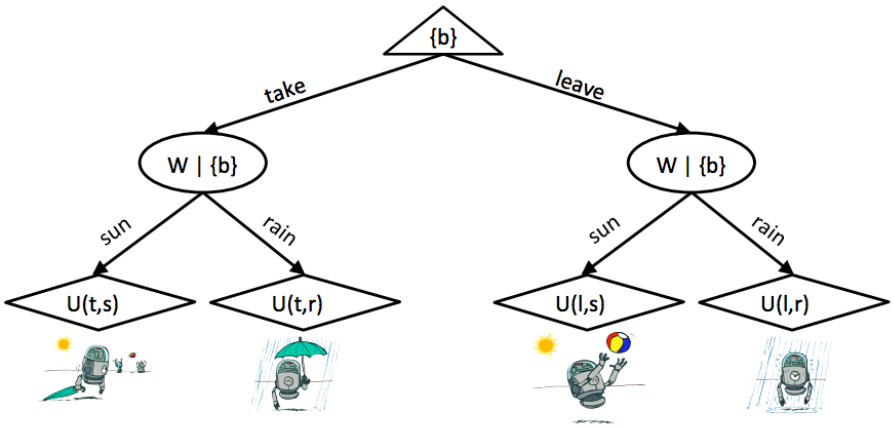
All that's left to do is take the maximum over these computed utilities to determine the MEU:

$$MEU(F = \text{bad}) = \max_a EU(a|\text{bad}) = \boxed{53}$$

The action that yields the maximum expected utility is *take*, and so this is the action recommended to us by the decision network. More formally, the action that yields the MEU can be determined by taking the **argmax** over expected utilities.

Outcome Trees

We mentioned at the start of this note that decision networks involved some expectimax-esque elements, so let's discuss what exactly that means. We can unravel the selection of an action corresponding to the one that maximizes expected utility in a decision network as an **outcome tree**. Our weather forecast example from above unravels into the following outcome tree:



The root node at the top is a maximizer node, just like in expectimax, and is controlled by us. We select an action, which takes us to the next level in the tree, controlled by chance nodes. At this level, chance nodes resolve to different utility nodes at the final level with probabilities corresponding to the posterior probabilities derived from probabilistic inference run on the underlying Bayes' net. What exactly makes this different from vanilla expectimax? The only real difference is that for outcome trees we annotate our nodes with what we know at any given moment (inside the curly braces).

The Value of Perfect Information

In everything we've covered up to this point, we've generally always assumed that our agent has all the information it needs for a particular problem and/or has no way to acquire new information. In practice, this is hardly the case, and one of the most important parts of decision making is knowing whether or not it's worth gathering more evidence to help decide which action to take. Observing new evidence almost always has some cost, whether it be in terms of time, money, or some other medium. In this section, we'll talk about a very important concept - the **value of perfect information** (VPI) - which mathematically quantifies the amount an agent's maximum expected utility is expected to increase if it observes some new evidence. We can compare the VPI of learning some new information with the cost associated with observing that information to make decisions about whether or not it's worthwhile to observe.

General Formula

Rather than simply presenting the formula for computing the value of perfect information for new evidence, let's walk through an intuitive derivation. We know from our above definition that the value of perfect information is the amount our maximum expected utility is expected to increase if we decide to observe new evidence. We know our current maximum utility given our current evidence e:

$$MEU(e) = \max_a \sum_s P(s|e)U(s,a)$$

Additionally, we know that if we observed some new evidence e' before acting, the maximum expected utility of our action at that point would become

$$MEU(e,e') = \max_a \sum_s P(s|e,e')U(s,a)$$

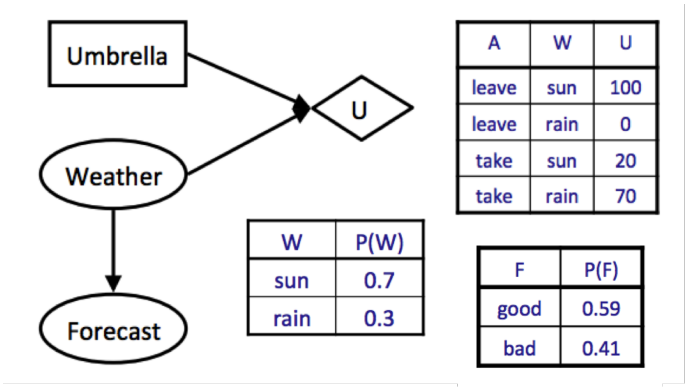
However, note that *we don't know what new evidence we'll get*. For example, if we didn't know the weather forecast beforehand and chose to observe it, the forecast we observe might be either *good* or *bad*. Because we don't know what what new evidence e' we'll get, we must represent it as a random variable E'. How do we represent the new MEU we'll get if we choose to observe a new variable if we don't know what the evidence gained from observation will tell us? The answer is to compute the expected value of the maximum expected utility which, while being a mouthful, is the natural way to go:

$$MEU(e,E') = \sum_{e'} P(e'|e)MEU(e,e')$$

Observing a new evidence variable yields a different MEU with probabilities corresponding to the probabilities of observing each value for the evidence variable, and so by computing $MEU(e,E')$ as above, we compute what we expect our new MEU will be if we choose to observe new evidence. We're just about done now - returning to our definition for VPI, we want to find the amount our MEU is expected to increase if we choose to observe new evidence. We know our current MEU and the expected value of the new MEU if we choose to observe, so the expected MEU increase is simply the difference of these two terms! Indeed,

$$\boxed{VPI(E'|e) = MEU(e,E') - MEU(e)}$$

where we can read $VPI(E'|e)$ as "the value of observing new evidence E' given our current evidence e". Let's work our way through an example by revisiting our weather scenario one last time:



If we don't observe any evidence, then our maximum expected utility can be computed as follows:

$$\begin{aligned} MEU(\varnothing) &= \max_a EU(a) \\ &= \max_a \sum_w P(w)U(a,w) \\ &= \max\{0.7\cdot 100 + 0.3\cdot 0, 0.7\cdot 20 + 0.3\cdot 70\} \\ &= \max\{70, 35\} \\ &= 70 \end{aligned}$$

Note that the convention when we have no evidence is to write $MEU(\varnothing)$, denoting that our evidence is the empty set. Now let's say that we're deciding whether or not to observe the weather forecast. We've already computed that $MEU(F = \text{bad}) = 53$, and let's assume that running an identical computation for $F = \text{good}$

yields $MEU(F = \text{good}) = 95$. We're now ready to compute $MEU(e, E')$:

$$\begin{aligned} MEU(e, E') &= MEU(F) \\ &= \sum_{e'} P(e'|e) MEU(e, e') \\ &= \sum_{f} P(F = f) MEU(F = f) \\ &= P(F = \text{good}) MEU(F = \text{good}) + P(F = \text{bad}) MEU(F = \text{bad}) \\ &= 0.59 \cdot 95 + 0.41 \cdot 53 \\ &= 77.78 \end{aligned}$$

Hence we conclude $VPI(F) = MEU(F) - MEU(\varnothing) = 77.78 - 70 = \boxed{7.78}$.

Properties of VPI

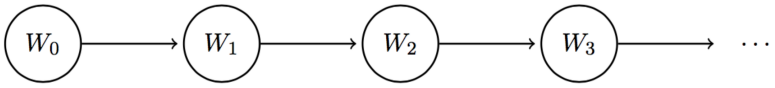
The value of perfect information has several very important properties, namely:

- **Nonnegativity.** $\forall E', e \; VPI(E'|e) \geq 0$
  Observing new information always allows you to make a *more informed* decision, and so your maximum expected utility can only increase (or stay the same if the information is irrelevant for the decision you must make).
- **Nonadditivity.** $VPI(E_j, E_k|e) \neq VPI(E_j|e) + VPI(E_k|e)$ in general.
  This is probably the trickiest of the three properties to understand intuitively. It's true because generally observing some new evidence E_j might change how much we care about E_k; therefore we can't simply add the VPI of observing E_j to the VPI of observing E_k to get the VPI of observing both of them. Rather, the VPI of observing two new evidence variables is equivalent to observing one, incorporating it into our current evidence, then observing the other. This is encapsulated by the order-independence property of VPI, described more below.
- **Order-independence.** $VPI(E_j, E_k|e) = VPI(E_j|e) + VPI(E_k|e, E_j) = VPI(E_k|e) + VPI(E_j|e, E_k)$
  Observing multiple new evidences yields the same gain in maximum expected utility regardless of the order of observation. This should be a fairly straightforward assumption - because we don't actually take any action until after observing any new evidence variables, it doesn't actually matter whether we observe the new evidence variables together or in some arbitrary sequential order.

CS 188 Introduction to Artificial Intelligence
Fall 2018 Note 8

Markov Models

In previous notes, we talked about Bayes' nets and how they are a wonderful structure used for compactly representing relationships between random variables. We'll now cover a very intrinsically related structure called a **Markov model**, which for the purposes of this course can be thought of as analogous to a chain-like, infinite-length Bayes' net. The running example we'll be working with in this section is the day-to-day fluctuations in weather patterns. Our weather model will be **time-dependent** (as are Markov models in general), meaning we'll have a separate random variable for the weather on each day. If we define W_i as the random variable representing the weather on day i, the Markov model for our weather example would look like this:



What information should we store about the random variables involved in our Markov model? To track how our quantity under consideration (in this case, the weather) changes over time, we need to know both it's **initial distribution** at time $t = 0$ and some sort of **transition model** that characterizes the probability of moving from one state to another between timesteps. The initial distribution of a Markov model is enumerated by the probability table given by $Pr(W_0)$ and the transition model of transitioning from state i to $i+1$ is given by $Pr(W_{i+1}|W_i)$. Note that this transition model implies that the value of W_{i+1} is conditionally dependent only on the value of W_i. In other words, the weather at time $t = i+1$ satisfies the **Markov property** or **memoryless property**, and is independent of the weather at all other timesteps besides $t = i$.

Using our Markov model for weather, if we wanted to reconstruct the joint between W_0, W_1, and W_2 using the chain rule, we would want:

$$Pr(W_0, W_1, W_2) = Pr(W_0)Pr(W_1|W_0)Pr(W_2|W_1, W_0)$$

However, with our assumption that the Markov property holds true and $W_0 \perp\!\!\!\perp W_2|W_1$, the joint simplifies to:

$$Pr(W_0, W_1, W_2) = Pr(W_0)Pr(W_1|W_0)Pr(W_2|W_1)$$

And we have everything we need to calculate this from the Markov model. More generally, Markov models make the following independence assumption at each timestep: $W_{i+1} \perp\!\!\!\perp \{W_0, ..., W_{i-1}\}|W_i$. This allows us to reconstruct the joint distribution for the first $n+1$ variables via the chain rule as follows:

$$Pr(W_0, W_1, ..., W_n) = Pr(W_0)Pr(W_1|W_0)Pr(W_2|W_1)...Pr(W_n|W_{n-1}) = Pr(W_0)\prod_{i=0}^{n-1} Pr(W_{i+1}|W_i)$$

A final assumption that's typically made in Markov models is that the transition model is **stationary**. In other words, for all values of i, $Pr(W_{i+1}|W_i)$ is identical. This allows us to represent a Markov model with only two tables: one for $Pr(W_0)$ and one for $Pr(W_{i+1}|W_i)$.

The Mini-Forward Algorithm

We now know how to compute the joint distribution across timesteps of a Markov model. However, this doesn't explicitly help us answer the question of the distribution of the weather on some given day t. Naturally, we can compute the joint then **marginalize** (sum out) over all other variables, but this is typically extremely inefficient, since if we have j variables each of which can take on d values, the size of the joint distribution is $O(d^j)$. Instead, we'll present a more efficient technique called the **mini-forward algorithm**.

Here's how it works. By properties of marginalization, we know that

$$Pr(W_{i+1}) = \sum_{w_i} Pr(w_i, W_{i+1})$$

By the chain rule we can reexpress this as follows:

$$\boxed{Pr(W_{i+1}) = \sum_{w_i} Pr(W_{i+1}|w_i)Pr(w_i)}$$

This equation should make some intuitive sense - to compute the distribution of the weather at timestep $i+1$, we look at the probability distribution at timestep i given by $Pr(W_i)$ and "advance" this model a timestep with our transition model $Pr(W_{i+1}|W_i)$. With this equation, we can iteratively compute the distribution of the weather at any timestep of our choice by starting with our initial distribution $Pr(W_0)$ and using it to compute $Pr(W_1)$, then in turn using $Pr(W_1)$ to compute $Pr(W_2)$, and so on. Let's walk through an example, using the following initial distribution and transition model:

W_0	$Pr(W_0)$
sun	0.8
rain	0.2

W_{i+1}	W_i	$Pr(W_{i+1}\|W_i)$
sun	*sun*	0.6
rain	*sun*	0.4
sun	*rain*	0.1
rain	*rain*	0.9

Using the mini-forward algorithm we can compute $Pr(W_1)$ as follows:

$$\begin{aligned}
Pr(W_1 = sun) &= \sum_{w_0} Pr(W_1 = sun|w_0)Pr(w_0) \\
&= Pr(W_1 = sun|W_0 = sun)Pr(W_0 = sun) + Pr(W_1 = sun|W_0 = rain)Pr(W_0 = rain) \\
&= 0.6 \cdot 0.8 + 0.1 \cdot 0.2 = \boxed{0.5} \\
Pr(W_1 = rain) &= Pr(W_1 = rain|w_0)Pr(w_0) \\
&= Pr(W_1 = rain|W_0 = sun)Pr(W_0 = sun) + Pr(W_1 = rain|W_0 = rain)Pr(W_0 = rain) \\
&= 0.4 \cdot 0.8 + 0.9 \cdot 0.2 = \boxed{0.5}
\end{aligned}$$

Hence our distribution for $Pr(W_1)$ is

W_1	$Pr(W_1)$
sun	0.5
rain	0.5

Notably, the probability that it will be sunny has decreased from 80% at time $t = 0$ to only 50% at time $t = 1$. This is a direct result of our transition model, which favors transitioning to rainy days over sunny days. This gives rise to a natural follow-up question: does the probability of being in a state at a given timestep ever converge? We'll address the answer to this problem in the following section.

Stationary Distribution

To solve the problem stated above, we must compute the **stationary distribution** of the weather. As the name suggests, the stationary distribution is one that remains the same after the passage of time, i.e.

$$Pr(W_{t+1}) = Pr(W_t)$$

We can compute these converged probabilities of being in a given state by combining the above equivalence with the same equation used by the mini-forward algorithm:

$$Pr(W_{t+1}) = Pr(W_t) = \sum_{w_t} Pr(W_{t+1}|w_t)Pr(w_t)$$

For our weather example, this gives us the following two equations:

$$\begin{aligned}
Pr(W_t = sun) &= Pr(W_{t+1} = sun|W_t = sun)Pr(W_t = sun) + Pr(W_{t+1} = sun|W_t = rain)Pr(W_t = rain) \\
&= 0.6 \cdot Pr(W_t = sun) + 0.1 \cdot Pr(W_t = rain) \\
Pr(W_t = rain) &= Pr(W_{t+1} = rain|W_t = sun)Pr(W_t = sun) + Pr(W_{t+1} = rain|W_t = rain)Pr(W_t = rain) \\
&= 0.4 \cdot Pr(W_t = sun) + 0.9 \cdot Pr(W_t = rain)
\end{aligned}$$

We now have exactly what we need to solve for the stationary distribution, a system of 2 equations in 2 unknowns! We can get a third equation by using the fact that $Pr(W_t)$ is a probability distribution and so must sum to 1:

$$\begin{aligned}
Pr(W_t = sun) &= 0.6 \cdot Pr(W_t = sun) + 0.1 \cdot Pr(W_t = rain) \\
Pr(W_t = rain) &= 0.4 \cdot Pr(W_t = sun) + 0.9 \cdot Pr(W_t = rain) \\
1 &= Pr(W_t = sun) + Pr(W_t = rain)
\end{aligned}$$

Solving this system of equations yields $Pr(W_t = sun) = 0.2$ and $Pr(W_t = rain) = 0.8$. Hence the table for our stationary distribution, which we'll henceforth denote as $Pr(W_\infty)$, is the following:

W_∞	$Pr(W_\infty)$
sun	0.2
rain	0.8

To verify this result, let's apply the transition model to the stationary distribution:

$$\begin{aligned}
Pr(W_{\infty+1} = sun) &= Pr(W_{\infty+1} = sun|W_\infty = sun)Pr(W_\infty = sun) + Pr(W_{\infty+1} = sun|W_\infty = rain)Pr(W_\infty = rain) \\
&= 0.6 \cdot 0.2 + 0.1 \cdot 0.8 = \boxed{0.2} \\
Pr(W_{\infty+1} = rain) &= Pr(W_{\infty+1} = rain|W_\infty = sun)Pr(W_\infty = sun) + Pr(W_{\infty+1} = rain|W_\infty = rain)Pr(W_\infty = rain) \\
&= 0.4 \cdot 0.2 + 0.9 \cdot 0.8 = \boxed{0.8}
\end{aligned}$$

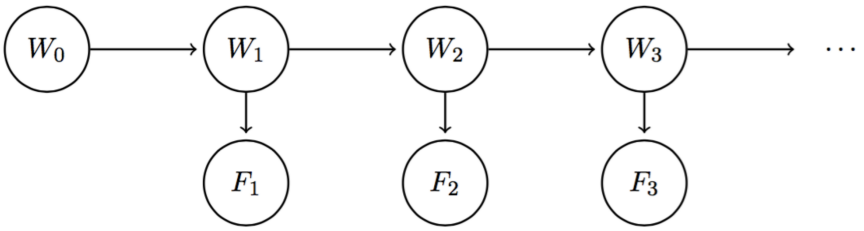
As expected, $Pr(W_{\infty+1}) = Pr(W_\infty)$. In general, if W_t had a domain of size k, the equivalence

$$Pr(W_t) = \sum_{w_t} Pr(W_{t+1}|w_t)Pr(w_t)$$

yields a system of k equations, which we can use to solve for the stationary distribution.

Hidden Markov Models

With Markov models, we saw how we could incorporate change over time through a chain of random variables. For example, if we want to know the weather on day 10 with our standard Markov model from above, we can begin with the initial distribution $Pr(W_0)$ and use the mini-forward algorithm with our transition model to compute $Pr(W_{10})$. However, between time $t = 0$ and time $t = 10$, we may collect new meteorological evidence that might affect our belief of the probability distribution over the weather at any given timestep. In simpler terms, if the weather forecasts an 80% chance of rain on day 10, but there are clear skies on the night of day 9, that 80% probability might drop drastically. This is exactly what the **Hidden Markov Model** helps us with - it allows us to observe some evidence at each timestep, which can potentially affect the belief distribution at each of the states. The Hidden Markov Model for our weather model can be described using a Bayes' net structure that looks like the following:



Unlike vanilla Markov models, we now have two different types of nodes. To make this distinction, we'll call each W_i a **state variable** and each weather forecast F_i an **evidence variable**. Since W_i encodes our belief of the probability distribution for the weather on day i, it should be a natural result that the weather forecast for day i is conditionally dependent upon this belief. The model implies similar conditional indepencence relationships as standard Markov models, with an additional set of relationships for the evidence variables:

$$
\begin{array}{lcll}
F_1 & \perp\!\!\!\perp & W_0|W_1 & \\
\forall i & = & 2,\ldots,n; & W_i \perp\!\!\!\perp \{W_0,\ldots,W_{i-2},F_1,\ldots,F_{i-1}\}|W_{i-1} \\
\forall i & = & 2,\ldots,n; & F_i \perp\!\!\!\perp \{W_0,\ldots,W_{i-1},F_1,\ldots,F_{i-1}\}|W_i
\end{array}
$$

Just like Markov models, Hidden Markov Models make the assumption that the transition model $Pr(W_{i+1}|W_i)$ is stationary. Hidden Markov Models make the additional simplifying assumption that the **sensor model** $Pr(F_i|W_i)$ is stationary as well. Hence any Hidden Markov Model can be represented compactly with just three probability tables: the initial distribution, the transition model, and the sensor model.

As a final point on notation, we'll define the **belief distribution** at time i with all evidence $F_1,\ldots,F_i$ observed up to date:

$$B(W_i) = Pr(W_i|f_1,\ldots,f_i)$$

Similarly, we'll define $B'(W_i)$ as the belief distribution at time i with evidence $f_1,\ldots,f_{i-1}$ observed:

$$B'(W_i) = Pr(W_i|f_1,\ldots,f_{i-1})$$

Defining e_i as evidence observed at timestep i, you might sometimes see the aggregated evidence from timesteps $1 \leq i \leq t$ reexpressed in the following form:

$$e_{1:t} = e_1,\ldots,e_t$$

Under this notation, $Pr(W_i|f_1,\ldots,f_{i-1})$ can be written as $Pr(W_i|f_{1:(i-1)})$. This notation will become relevant in the upcoming sections, where we'll discuss time elapse updates that iteratively incorporate new evidence into our weather model.

The Forward Algorithm

Using the conditional probability assumptions stated above and marginalization properties of conditional probability tables, we can derive a relationship between $B(W_i)$ and $B'(W_{i+1})$ that's of the same form as the update rule for the mini-forward algorithm. We begin by using marginalization:

$$B'(W_{i+1}) = Pr(W_{i+1}|f_1,\ldots,f_i) = \sum_{w_i} Pr(W_{i+1},w_i|f_1,\ldots,f_i)$$

This can be reexpressed then with the chain rule as follows:

$$B'(W_{i+1}) = Pr(W_{i+1}|f_1,\ldots,f_i) = \sum_{w_i} Pr(W_{i+1}|w_i,f_1,\ldots,f_i)Pr(w_i|f_1,\ldots,f_i)$$

Noting that $Pr(w_i|f_1,\ldots,f_i)$ is simply $B(w_i)$ and that $W_{i+1} \perp\!\!\!\perp \{f_1,\ldots f_i\}|W_i$, this simplies to our final relationship between $B(W_i)$ to $B'(W_{i+1})$:

$$\boxed{B'(W_{i+1}) = \sum_{w_i} Pr(W_{i+1}|w_i)B(w_i)}$$

Now let's consider how we can derive a relationship between $B'(W_{i+1})$ and $B(W_{i+1})$. By simple application of Bayes' rule, we can see that

$$B(W_{i+1}) = Pr(W_{i+1}|f_1,\ldots,f_{i+1}) = \frac{Pr(W_{i+1},f_{i+1}|f_1,\ldots,f_i)}{Pr(f_{i+1}|f_1,\ldots,f_i)}$$

When dealing with conditional probabilities a commonly used trick is to delay normalization until we require the normalized probabilities, a trick we'll now employ. More specifically, since the denominator in the above expansion of $B(W_{i+1})$ is common to every term in the probability table represented by $B(W_{i+1})$, we can omit actually dividing by $Pr(f_{i+1}|f_1,\ldots,f_i)$. Instead, we can simply note that $B(W_{i+1})$ is proportional to $Pr(W_{i+1},f_{i+1}|f_1,\ldots,f_i)$:

$$B(W_{i+1}) \propto Pr(W_{i+1},f_{i+1}|f_1,\ldots,f_i)$$

with a constant of proportionality equal to $Pr(f_{i+1}|f_1,\ldots,f_i)$. Whenever we decide we want to recover the belief distribution $B(W_{i+1})$, we can divide each computed value by this constant of proportionality. Now, using the chain rule we can observe the following:

$$B(W_{i+1}) \propto Pr(W_{i+1},f_{i+1}|f_1,\ldots,f_i) = Pr(f_{i+1}|W_{i+1},f_1,\ldots,f_i)Pr(W_{i+1}|f_1,\ldots,f_i)$$

By the conditional independence assumptions associated with Hidden Markov Models stated previously, $Pr(f_{i+1}|W_{i+1},f_1,\ldots,f_i)$ is equivalent to simply $Pr(f_{i+1}|W_{i+1})$ and by definition $Pr(W_{i+1}|f_1,\ldots,f_i) = B'(W_{i+1})$. This allows us to express the relationship between $B'(W_{i+1})$ and $B(W_{i+1})$ in it's final form:

$$\boxed{B(W_{i+1}) \propto Pr(f_{i+1}|W_{i+1})B'(W_{i+1})}$$

Combining the two relationships we've just derived yields an iterative algorithm known as the **forward algorithm**, the Hidden Markov Model analog of the mini-forward algorithm from earlier:

$$\boxed{B(W_{i+1}) \propto Pr(f_{i+1}|W_{i+1})\sum_{w_i} Pr(W_{i+1}|w_i)B(w_i)}$$

The forward algorithm can be thought of as consisting of two distinctive steps: the **time elapse update** which corresponds to determining $B'(W_{i+1})$ from $B(W_i)$ and the **observation update** which corresponds to determining $B(W_{i+1})$ from $B'(W_{i+1})$. Hence, in order to advance our belief distribution by one timestep (i.e. compute $B(W_{i+1})$ from $B(W_i)$), we must first advance our model's state by one timestep with the time elapse update, then incorporate new evidence from that timestep with the observation update. Consider the following initial distribution, transition model, and sensor model:

W_0	$B(W_0)$
sun	0.8
rain	0.2

W_{i+1}	W_i	$Pr(W_{i+1}\|W_i)$
sun	*sun*	0.6
rain	*sun*	0.4
sun	*rain*	0.1
rain	*rain*	0.9

F_i	W_i	$Pr(F_i\|W_i)$
good	*sun*	0.8
bad	*sun*	0.2
good	*rain*	0.3
bad	*rain*	0.7

To compute $B(W_1)$, we begin by performing a time update to get $B'(W_1)$:

$$\begin{aligned}
B'(W_1 = sun) &= \sum_{w_0} Pr(W_1 = sun|w_0)B(w_0) \\
&= Pr(W_1 = sun|W_0 = sun)B(W_0 = sun) + Pr(W_1 = sun|W_0 = rain)B(W_0 = rain) \\
&= 0.6 \cdot 0.8 + 0.1 \cdot 0.2 = \boxed{0.5} \\
B'(W_1 = rain) &= \sum_{w_0} Pr(W_1 = rain|w_0)B(w_0) \\
&= Pr(W_1 = rain|W_0 = sun)B(W_0 = sun) + Pr(W_1 = rain|W_0 = rain)B(W_0 = rain) \\
&= 0.4 \cdot 0.8 + 0.9 \cdot 0.2 = \boxed{0.5}
\end{aligned}$$

Hence:

W_1	$B'(W_1)$
sun	0.5
rain	0.5

Next, we'll assume that the weather forecast for day 1 was good (i.e. $F_1 = good$), and perform an observation update to get $B(W_1)$:

$$\begin{aligned}
B(W_1 = sun) &\propto Pr(F_1 = good|W_1 = sun)B'(W_1 = sun) = 0.8 \cdot 0.5 = \boxed{0.4} \\
B(W_1 = rain) &\propto Pr(F_1 = good|W_1 = rain)B'(W_1 = rain) = 0.3 \cdot 0.5 = \boxed{0.15}
\end{aligned}$$

The last step is to normalize $B(W_1)$, noting that the entries in table for $B(W_1)$ sum to $0.4 + 0.15 = 0.55$:

$$\begin{aligned}
B(W_1 = sun) &= 0.4/0.55 = \frac{8}{11} \\
B(W_1 = rain) &= 0.15/0.55 = \frac{3}{11}
\end{aligned}$$

Our final table for $B(W_1)$ is thus the following:

W_1	$B'(W_1)$
sun	$8/11$
rain	$3/11$

Note the result of observing the weather forecast. Because the weatherman predicted good weather, our belief that it would be sunny increased from $\frac{1}{2}$ after the time update to $\frac{8}{11}$ after the observation update.

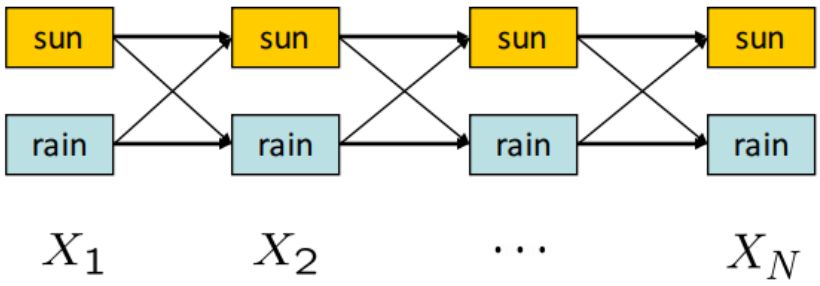
As a parting note, the normalization trick discussed above can actually simplify computation significantly when working with Hidden Markov Models. If we began with some initial distribution and were interested in computing the belief distribution at time t, we could use the forward algorithm to iteratively compute $B(W_1), \ldots, B(W_t)$ and normalize only once at the end by dividing each entry in the table for $B(W_t)$ by the sum of it's entries.

Viterbi Algorithm

In the Forward Algorithm, we used recursion to solve for $P(X_N|e_{1:N})$, the probability distribution over states the system could inhabit given the evidence variables observed so far. Another important question related to Hidden Markov Models is: *What is the most likely sequence of hidden states the system followed given the observed evidence variables so far?* In other words, we would like to solve for $\arg\max_{x_{1:N}} P(x_{1:N}|e_{1:N}) = \arg\max_{x_{1:N}} P(x_{1:N}, e_{1:N})$. This trajectory can also be solved for using dynamic programming with the **Viterbi algorithm**.

The algorithm consists of two passes: the first runs forward in time and computes the probability of the best path to that (state, time) tuple given the evidence observed so far. The second pass runs backwards in time: first it finds the terminal state that lies on the path with the highest probability, and then traverses backward through time along the path that leads into this state (which must be the best path).

To visualize the algorithm, consider the following **state trellis**, a graph of states and transitions over time:



In this HMM with two possible hidden states, sun or rain, we would like to compute the highest probability path from X_1 to X_N. The weights on the edges are equal to $P(X_t|X_{t-1})P(E_t|X_t)$ and the probability of a path is computed by taking the *product* of its edge weights. The first term in the weight estimates how likely a particular transition is and the second term weights how well the observed evidence fits the resulting state.

Recall that:

$$P(X_{1:N}, e_{1:N}) = P(X_1)P(e_1|X_1)\prod_{t=2}^{N} P(X_t|X_{t-1})P(e_t|X_t)$$

The Forward Algorithm computes (up to normalization)

$$P(X_N, e_{1:N}) = \sum_{x_1,..,x_{N-1}} P(X_N, x_{1:N-1}, e_{1:N})$$

In the Viberbi Algorithm, we want to compute

$$\arg\max_{x_1,..,x_N} P(x_{1:N}, e_{1:N})$$

to find the maximum likelihood estimate of the sequence of hidden states. Notice that each term in the product is exactly the expression for the edge weight between layer $t-1$ to layer t. So, the product of weights along the path on the trellis gives us the probability of the path given the evidence.

We could solve for a joint probability table over all of the possible hidden states, but this results in an exponential space cost. Given such a table, we could use dynamic programming to compute the best path in polynomial time. However, because we can use dynamic programming to compute the best path, we don't necessarily need the whole table at any given time.

Define $m_t[x_t] = \max_{x_{1:t-1}} P(x_{1:t}, e_{1:t})$, or the maximum probability of a path starting at any x_0 and the evidence seen so far to a given x_t at time t. This is the same as the highest weight path through the trellis from step 1 to t. Also note that

$$\begin{aligned} m_t[x_t] &= \max_{x_{1:t-1}} P(e_t|x_t)P(x_t|x_{t-1})P(x_{1:t-1}, e_{1:t-1}) && (1)\\ &= P(e_t|x_t)\max_{x_{t-1}} P(x_t|x_{t-1})\max_{x_{1:t-2}} P(x_{1:t-1}, e_{1:t-1}) && (2)\\ &= P(e_t|x_t)\max_{x_{t-1}} P(x_t|x_{t-1})m_{t-1}[x_{t-1}]. && (3) \end{aligned}$$

This suggests that we can compute m_t for all t recursively via dynamic programming. This makes it possible to determine the last state x_N for the most likely path, but we still need a way to backtrack to reconstruct the entire path. Let's define $a_t[x_t] = P(e_t|x_t)\arg\max_{x_{t-1}} P(x_t|x_{t-1})m_{t-1}[x_{t-1}] = \arg\max_{x_{t-1}} P(x_t|x_{t-1})m_{t-1}[x_{t-1}]$ to keep track of the last transition along the best path to x_t. We can now outline the algorithm.

Result: Most likely sequence of hidden states $x^*_{1:N}$
`/* Forward pass */`
for $t = 1$ *to* N **do**
  for $x_t \in \mathscr{X}$ **do**
    if $t = 1$ **then**
      $m_t[x_t] = P(x_t)P(e_0|x_t)$
    else
      $a_t[x_t] = \arg\max_{x_{t-1}} P(x_t|x_{t-1})m_{t-1}[x_{t-1}]$;
      $m_t[x_t] = P(e_t|x_t)P(x_t|a_t[x_t])m_{t-1}[a_t[x_t]]$;
    end
  end
end
`/* Find the most likely path's ending point */`
$x^*_N = \arg\max_{x_N} m_N[x_N]$;
`/* Work backwards through our most likely path and find the hidden states */`
for $t = N$ *to* 2 **do**
  $x^*_{t-1} = a_t[x^*_t]$;
end

Notice that our a arrays define a set of N sequences, each of which is the most likely sequence to a particular end state x_N. Once we finish the forward pass, we look at the likelihood of the N sequences, pick the best

one, and reconstruct it in the backwards pass. We have thus computed the most likely explanation for our evidence in polynomial space and time.

Particle Filtering

Recall that with Bayes' nets, when running exact inference was too computationally expensive, using one of the sampling techniques we discussed was a viable alternative to efficiently approximate the desired probability distribution(s) we wanted. Hidden Markov Models have the same drawback - the time it takes to run exact inference with the forward algorithm scales with the number of values in the domains of the random variables. This was acceptable in our current weather problem formulation where the weather can only take on 2 values, $W_i \in \{sun, rain\}$, but say instead we wanted to run inference to compute the distribution of the actual temperature on a given day to the nearest tenth of a degree. The Hidden Markov Model analog to Bayes' net sampling is called **particle filtering**, and involves simulating the motion of a set of particles through a state graph to approximate the probability (belief) distribution of the random variable in question.

Instead of storing a full probability table mapping each state to its belief probability, we'll instead store a list of n particles, where each particle is in one of the d possible states in the domain of our time-dependent random variable. Typically, n is significantly smaller than d (denoted symbolically as $n << d$) but still large enough to yield meaningful approximations; otherwise the performance advantage of particle filtering becomes negligible. Our belief that a particle is in any given state at any given timestep is dependent entirely on the number of particles in that state at that timestep in our simulation. For example, say we indeed wanted to simulate the belief distribution of the temperature T on some day i and assume for simplicity that this temperature can only take on integer values in the range $[10, 20]$ ($d = 11$ possible states). Assume further that we have $n = 10$ particles, which take on the following values at timestep i of our simulation:

$$[15, 12, 12, 10, 18, 14, 12, 11, 11, 10]$$

By taking counts of each temperature that appears in our particle list and diving by the total number of particles, we can generate our desired empirical distribution for the temperature at time i:

T_i	10	11	12	13	14	15	16	17	18	19	20
$B(T_i)$	0.2	0.2	0.3	0	0.1	0.1	0	0	0.1	0	0

Now that we've seen how to recover a belief distribution from a particle list, all that remains to be discussed is how to generate such a list for a timestep of our choosing.

Particle Filtering Simulation

Particle filtering simulation begins with particle initialization, which can be done quite flexibly - we can sample particles randomly, uniformly, or from some initial distribution. Once we've sampled an initial list of particles, the simulation takes on a similar form to the forward algorithm, with a time elapse update followed by an observation update at each timestep:

- *Time Elapse Update* - Update the value of each particle according to the transition model. For a particle in state t_i, sample the updated value from the probability distribution given by $Pr(T_{i+1}|t_i)$. Note the similarity of the time elapse update to prior sampling with Bayes' nets, since the frequency of particles in any given state reflects the transition probabilities.

- *Observation Update* - During the observation update for particle filtering, we use the sensor model $Pr(F_i|T_i)$ to weight each particle according to the probability dictated by the observed evidence and the particle's state. Specifically, for a particle in state t_i with sensor reading f_i, assign a weight of $Pr(f_i|t_i)$. The algorithm for the observation update is as follows:

  1. Calculate the weights of all particles as described above.
  2. Calculate the total weight for each state.
  3. If the sum of all weights across all states is 0, reinitialize all particles.
  4. Else, normalize the distribution of total weights over states and resample your list of particles from this distribution.

  Note the similarity of the observation update to likelihood weighting, where we again downweight samples based on our evidence.

Let's see if we can understand this process slightly better by example. Define a transition model for our weather scenario using temperature as the time-dependent random variable as follows: for a particular temperature state, you can either stay in the same state or transition to a state one degree away, within the range $[10, 20]$. Out of the possible resultant states, the probability of transitioning to the one closest to 15 is 80% and the remaining resultant states uniformly split the remaining 20% probability amongst themselves.

Our temperature particle list was as follows:

$$[15, 12, 12, 10, 18, 14, 12, 11, 11, 10]$$

To perform a time elapse update for the first particle in this particle list, which is in state $T_i = 15$, we need the corresponding transition model:

T_{i+1}	14	15	16
$Pr(T_{i+1}\|T_i = 15)$	0.1	0.8	0.1

In practice, we allocate a different range of values for each value in the domain of T_{i+1} such that together the ranges entirely span the interval $[0, 1)$ without overlap. For the above transition model, the ranges are as follows:

1. The range for $T_{i+1} = 14$ is $0 \leq r < 0.1$.
2. The range for $T_{i+1} = 15$ is $0.1 \leq r < 0.9$.
3. The range for $T_{i+1} = 16$ is $0.9 \leq r < 1$.

In order to resample our particle in state $T_i = 15$, we simply generate a random number in the range $[0, 1)$ and see which range it falls in. Hence if our random number is $r = 0.467$, then the particle at $T_i = 15$ remains in $T_{i+1} = 15$ since $0.1 \leq r < 0.9$. Now consider the following list of 10 random numbers in the interval $[0, 1)$:

$$[0.467, 0.452, 0.583, 0.604, 0.748, 0.932, 0.609, 0.372, 0.402, 0.026]$$

If we use these 10 values as the random value for resampling our 10 particles, our new particle list after the full time elapse update should look like this:

$$[15, 13, 13, 11, 17, 15, 13, 12, 12, 10]$$

Verify this for yourself! The updated particle list gives rise to the corresponding updated belief distribution $B(T_{i+1})$:

T_i	10	11	12	13	14	15	16	17	18	19	20
$B(T_{i+1})$	0.1	0.1	0.2	0.3	0	0.2	0	0.1	0	0	0

Comparing our updated belief distribution $B(T_{i+1})$ to our initial belief distribution $B(T_i)$, we can see that as a general trend the particles tend to converge towards a temperature of $T = 15$.

Next, let's perform the observation update, assuming that our sensor model $Pr(F_i|T_i)$ states that the probability of a correct forecast $f_i = t_i$ is 80%, with a uniform 2% chance of the forecast predicting any of the other 10 states. Assuming a forecast of $F_{i+1} = 13$, the weights of our 10 particles are as follows:

Particle	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}
State	15	13	13	11	17	15	13	12	12	10
Weight	0.02	0.8	0.8	0.02	0.02	0.02	0.8	0.02	0.02	0.02

Then we aggregate weights by state:

State	10	11	12	13	15	17
Weight	0.02	0.02	0.04	2.4	0.04	0.02

Summing the values of all weights yields a sum of 2.54, and we can normalize our table of weights to generate a probability distribution by dividing each entry by this sum:

State	10	11	12	13	15	17
Weight	0.02	0.02	0.04	2.4	0.04	0.02
Normalized Weight	0.0079	0.0079	0.0157	0.9449	0.0157	0.0079

The final step is to resample from this probability distribution, using the same technique we used to resample during the time elapse update. Let's say we generate 10 random numbers in the range $[0,1)$ with the following values:

$$[0.315, 0.829, 0.304, 0.368, 0.459, 0.891, 0.282, 0.980, 0.898, 0.341]$$

This yields a resampled particle list as follows:

$$[13, 13, 13, 13, 13, 13, 13, 15, 13, 13]$$

With the corresponding final new belief distribution:

T_i	10	11	12	13	14	15	16	17	18	19	20
$B(T_{i+1})$	0	0	0	0.9	0	0.1	0	0	0	0	0

Observe that our sensor model encodes that our weather prediction is very accurate with probability 80%, and that our new particles list is consisistent with this since most particles are resampled to be $T_{i+1} = 13$.

Summary

In this note, we covered two new types of models:

- *Markov models*, which encode time-dependent random variables that possess the Markov property. We can compute a belief distribution at any timestep of our choice for a Markov model using probabilistic inference with the mini-forward algorithm.
- *Hidden Markov Models*, which are Markov models with the additional property that new evidence which can affect our belief distribution can be observed at each timestep. To compute the belief distribution at any given timestep with Hidden Markov Models, we use the forward algorithm.

Sometimes, running exact inference on these models can be too computationally expensive, in which case we can use particle filtering as a method of approximate inference.

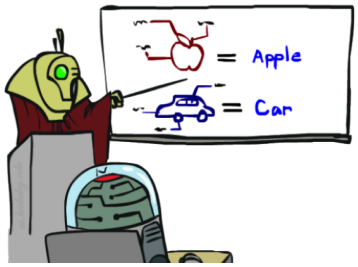
CS 188 Introduction to Artificial Intelligence
Fall 2018 Note 9

These lecture notes are heavily based on notes originally written by Nikhil Sharma.

Machine Learning

In the previous few notes of this course, we've learned about various types of models that help us reason under uncertainty. Until now, we've assumed that the probabilistic models we've worked with can be taken for granted, and the methods by which the underlying probability tables we worked with were generated have been abstracted away. We'll begin to break down this abstraction barrier as we delve into our discussion of **machine learning**, a broad field of computer science that deals with constructing and/or learning the parameters of a specified model given some data.

There are many machine learning algorithms which deal with many different types of problems and different types of data, classified according to the tasks they hope to accomplish and the types of data that they work with. Two primary subgroups of machine learning algorithms are **supervised learning algorithms** and **unsupervised learning algorithms**. Supervised learning algorithms infer a relationship between input data and corresponding output data in order to predict outputs for new, previously unseen input data. Unsupervised learning algorithms, on the other hand, have input data that doesn't have any corresponding output data and so deal with recognizing inherent structure between or within datapoints and grouping and/or processing them accordingly. In this class, the algorithms we'll discuss will be limited to supervised learning tasks.



(a) Training

(b) Validation

(c) Testing

Once you have a dataset that you're ready to learn with, the machine learning process usually involves splitting your dataset into three distinct subsets. The first, **training data**, is used to actually generate a model mapping inputs to outputs. Then, **validation data** (also known as **hold-out** or **development data**) is used to measure your model's performance by making predictions on inputs and generating an accuracy score. If your model doesn't perform as well as you'd like it to, it's always okay to go back and train again, either by adjusting special model-specific values called **hyperparameters** or by using a different learning algorithm altogether until you're satisfied with your results. Finally, use your model to make predictions on the third and final subset of your data, the **test set**. The test set is the portion of your data that's never seen by

your agent until the very end of development, and is the equivalent of a "final exam" to gauge performance on real-world data.

Naive Bayes

We'll further motivate our discussion of machine learning with a concrete example of a machine learning algorithm. Let's consider the common problem of building an email spam filter which sorts messages into spam (unwanted email) or ham (wanted email). Such a problem is called a **classification problem** – given various datapoints (in this case, each email is a datapoint), our goal is to group them into one of two or more **classes**. For classification problems, we're given a training set of datapoints along with their corresponding **labels**, which are typically one of a few discrete values. As we've discussed, our goal will be to use this training data (emails, and a spam/ham label for each one) to learn some sort of relationship that we can use to make predictions on new and previously unseen datapoints. In this section we'll describe how to construct a specific type of model for solving classification problems known as a **Naive Bayes Classifier**.

To train a model to classify emails as spam or ham, we need some training data consisting of preclassified emails that we can learn from. However, emails are simply strings of text, and in order to learn anything useful, we need to extract certain attributes from each of them known as **features**. Features can be anything ranging from specific word counts to text patterns (e.g. whether words are in all caps or not) to pretty much any other attribute of the data that you can imagine. The specific features extracted for training are often dependent on the specific problem you're trying to solve and which features you decide to select can often impact the performance of your model dramatically. Deciding which features to utilize is known as **feature engineering** and is fundamental to machine learning, but for the purposes of this course you can assume you'll always be given the extracted features for any given dataset.

Now let's say you have a dictionary of n words, and from each email you extract a feature vector $F \in \mathbb{R}^n$ where the i^{th} entry in F is a random variable F_i which can take on a value of either a 0 or a 1 depending on whether the i^{th} word in your dictionary appears in the email under consideration. For example, if F_{200} is the feature for the word *free*, we will have $F_{200} = 1$ if *free* appears in the email, and 0 if it does not. With these definitions, we can define more concretely how to predict whether or not an email is spam or ham – if we can generate a joint probability table between each F_i and and the label Y, we can compute the probability any email under consideration is spam or ham given it's feature vector. Specifically, we can compute both

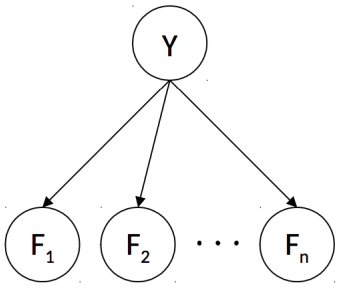
$$P(Y = spam | F_1 = f_1, \ldots, F_n = f_n)$$

and

$$P(Y = ham | F_1 = f_1, \ldots, F_n = f_n)$$

and simply label the email depending on which of the two probabilities is higher. Unfortunately, since we have n features and 1 label, each of which can take on 2 distinct values, the joint probability table

corresponding to this distribution mandates a table size that's exponential in n with 2^{n+1} entries - very impractical! This problem is solved by modeling the joint probability table with a Bayes' net, making the critical simplifying assumption that each feature F_i is independent of all other features given the class label. This is a very strong modeling assumption (and the reason that Naive Bayes is called *naive*), but it simplifies inference and usually works well in practice. It leads to the following Bayes' net to represent our desired joint probability distribution



Note that the rules of d-separation delineated earlier in the course make it immediately clear that in this Bayes' net each F_i is conditionally independent of all the others, given Y. Now we have one table for $P(Y)$ with 2 entries, and n tables for each $P(F_i \mid Y)$ each with $2^2 = 4$ entries for a total of $4n+2$ entries - linear in n! This simplifying assumption highlights the tradeoff that arises from the concept of **statistical efficiency**; we sometimes need to compromise our model's complexity in order to stay within the limits of our computational resources.

Indeed, in cases where the number of features is sufficiently low, it's common to make more assumptions about relationships between features to generate a better model (corresponding to adding edges to your Bayes' net). With this model we've adopted, making predictions for unknown datapoints amounts to running inference on our Bayes' net. We have observed values for $F_1, \cdots F_n$, and want to choose the value of Y that has the highest probability conditioned on these featuers:

$$\begin{aligned} \textit{prediction}(f_1, \cdots f_n) &= \operatorname*{argmax}_y P(Y = y | F_1 = f_1, \ldots F_N = f_n) \\ &= \operatorname*{argmax}_y P(Y = y, F_1 = f_1, \ldots F_N = f_n) \\ &= \operatorname*{argmax}_y P(Y = y) \prod_{i=1}^{n} P(F_i = f_i | Y = y) \end{aligned}$$

where the first step is because the highest probability class will be the same in the normalized or unnormalized distribution, and the second comes directly from the Naive Bayes' independence assumption that features are independent given the class label (as seen in the graphical model structure).

Generalizing away from a spam filter, assume now that there are k class labels (possible values for Y). Additionally, after noting that our desired probabilities - the probability of each label y_i given our features, $P(Y = y_i | F_1 = f_1, \ldots, F_n = f_n)$ - is proportional to the joint $P(Y = y_i, F_1 = f_1, \ldots, F_n = f_n)$, we can compute:

$$P(Y, F_1 = f_1, \ldots, F_n = f_n) = \begin{bmatrix} P(Y = y_1, F_1 = f_1, \ldots, F_n = f_n) \\ P(Y = y_2, F_1 = f_1, \ldots, F_n = f_n) \\ \vdots \\ P(Y = y_k, F_1 = f_1, \ldots, F_n = f_n) \end{bmatrix} = \begin{bmatrix} P(Y = y_1) \prod_i P(F_i = f_i | Y = y_1) \\ P(Y = y_2) \prod_i P(F_i = f_i | Y = y_2) \\ \vdots \\ P(Y = y_k) \prod_i P(F_i = f_i | Y = y_k) \end{bmatrix}$$

Our prediction for class label corresponding to the feature vector F is simply the label corresponding to the maximum value in the above computed vector:

$$prediction(F) = \underset{y_i}{\text{argmax}} \; P(Y = y_i) \prod_j P(F_j = f_j | Y = y_i)$$

We've now learned the basic theory behind the modeling assumptions of the Naive Bayes' classifier and how to make predictions with one, but have yet to touch on how exactly we learn the conditional probability tables used in our Bayes' net from the input data. This will have to wait for our next topic of discussion, **parameter estimation**.

Parameter Estimation

Assume you have a set of N **sample points** or **observations**, $x_1, \ldots, x_N$, and you believe that this data was drawn from a distribution **parametrized** by an unknown value θ. In other words, you believe that the probability $P_\theta(x_i)$ of each of your observations is a function of θ. For example, we could be flipping a coin which has probability θ of coming up heads.

How can you "learn" the most likely value of θ given your sample? For example, if we have 10 coin flips, and saw that 7 of them were heads, what value should we choose for θ? One answer to this question is to infer that θ is equal to the value that maximizes the probability of having selected your sample $x_1, \ldots, x_N$ from your assumed probability distribution. A frequently used and fundamental method in machine learning known as **maximum likelihood estimation** (MLE) does exactly this. Maximum likelihood estimation typically makes the following simplifying assumptions:

- Each sample is drawn from the same distribution. In other words, each x_i is **identically distributed**. In our coin flipping example, each coin flip has the same chance, θ, of coming up heads.
- Each sample x_i is conditionally **independent** of the others, given the parameters for our distribution. This is a strong assumption, but as we'll see greatly helps simplify the problem of maximum likelihood estimation and generally works well in practice. In the coin flipping example, the outcome of one flip doesn't affect any of the others.
- All possible values of θ are equally likely before we've seen any data (this is known as a **uniform prior**).

The first two assumptions above are often referred to as **independent, identically distributed** (i.i.d.).

Let's now define the **likelihood** $\mathscr{L}(\theta)$ of our sample, a function which represents the probability of having drawn our sample from our distribution. For a fixed sample x_1, x_N, the likelihood is just a function of θ:

$$\mathscr{L}(\theta) = P_\theta(x_1, \ldots, x_N)$$

Using our simplifying assumption that the samples x_i are i.i.d., the likelihood function can be reexpressed as follows:

$$\mathscr{L}(\theta) = \prod_{i=1}^{N} P_\theta(x_i)$$

How can we find the value of θ that maximizes this function? This will be the value of θ that best explains the data we saw. Recall from calculus that at points where a function's maxima and minima are realized, its

first derivative with respect to each of its inputs (also known as the function's **gradient**) must be equal to zero. Hence, the maximum likelihood estimate for θ is a value that satisfies the following equation:

$$\frac{\partial}{\partial \theta}\mathscr{L}(\theta) = 0$$

Let's go through an example to make this concept more concrete. Say you have a bag filled with red and blue balls and don't know how many of each there are. You draw samples by taking a ball out of the bag, noting the color, then putting the ball back in (sampling with replacement). Drawing a sample of three balls from this bag yields *red*, *red*, *blue*. This seems to imply that we should infer that $\frac{2}{3}$ of the balls in the bag are red and $\frac{1}{3}$ of the balls are blue. We'll assume that each ball being taken out of the bag will be red with probability θ and blue with probability $1-\theta$, for some value θ that we want to estimate (this is known as a Bernoulli distribution):

$$P_\theta(x_i) = \begin{cases} \theta & x_i = red \\ (1-\theta) & x_i = blue \end{cases}$$

The likelihood of our sample is then:

$$\mathscr{L}(\theta) = \prod_{i=1}^{3} P_\theta(x_i) = P_\theta(x_1 = red)P_\theta(x_2 = red)P_\theta(x_2 = blue) = \theta^2 \cdot (1-\theta)$$

The final step is to set the derivative of the likelihood to 0 and solve for θ:

$$\frac{\partial}{\partial \theta}\mathscr{L}(\theta) = \frac{\partial}{\partial \theta}\theta^2 \cdot (1-\theta) = \theta(2-3\theta) = 0$$

Solving this equation for θ yields $\theta = \frac{2}{3}$, which intuitively makes sense! (There's a second solution, too, $\theta = 0$ – but this corresponds to a minimum of the likelihood function, as $\mathscr{L}(0) = 0 < \mathscr{L}(\frac{2}{3}) = \frac{4}{27}$.)

Maximum Likelihood for Naive Bayes

Let's now return to the problem of inferring conditional probability tables for our spam classifier, beginning with a recap of variables we know:

- n - the number of words in our dictionary.
- N - the number of observations (emails) you have for training. For our upcoming discussion, let's also define N_h as the number of training samples labeled as ham and N_s as the number of training samples labeled as spam. Note $N_h + N_s = N$.
- F_i - a random variable which is 1 if the i^{th} dictionary word is present in an email under consideration, and 0 otherwise.
- Y - a random variable that's either *spam* or *ham* depending on the label of the corresponding email.
- $f_i^{(j)}$ - this references the resolved value of the random variable F_i in the j^{th} item in the training set. In other words, each $f_i^{(j)}$ is a 1 if word i appeared in j^{th} email under consideration and 0 otherwise. This is the first time we're seeing this notation, but it'll come in handy in the upcoming derivation.

Disclaimer: Feel free to skip the following mathematical derivation. For CS 188, you're only required to know the result of the derivation summarized in the paragraph at the end of this section.

Now within each conditional probability table $P(F_i|Y)$, note that we have two distinct Bernoulli distributions: $P(F_i|Y = ham)$ and $P(F_i|Y = spam)$. For simplicity, let's specifically consider $P(F_i|Y = ham)$ and try to find the maximum likelihood estimate for a parameter $\theta = P(F_i = 1|Y = ham)$ i.e. the probability that the i^{th} word in our dictionary appears in a ham email. Since we have N_h ham emails in our training set, we have N_h observations of whether or not word i appeared in a ham email. Because our model assumes a Bernoulli distribution for the appearance of each word given its label, we can formulate our likelihood function as follows:

$$\mathscr{L}(\theta) = \prod_{j=1}^{N_h} P(F_i = f_i^{(j)}|Y = ham) = \prod_{j=1}^{N_h} \theta^{f_i^{(j)}}(1-\theta)^{1-f_i^{(j)}}$$

The second step comes from a small mathematical trick: if $f_i^{(j)} = 1$ then

$$P(F_i = f_i^{(j)}|Y = ham) = \theta^1(1-\theta)^0 = \theta$$

and similarly if $f_i^{(j)} = 0$ then

$$P(F_i = f_i^{(j)}|Y = ham) = \theta^0(1-\theta)^1 = (1-\theta)$$

In order to compute the maximum likelihood estimate for θ, recall that the next step is to compute the derivative of $\mathscr{L}(\theta)$ and set it equal to 0. Attempting this proves quite difficult, as it's no simple task to isolate and solve for θ. Instead, we'll employ a trick that's very common in maximum likelihood derivations, and that's to instead find the value of θ the maximizes the log of the likelihood function. Because $\log(x)$ is a strictly increasing function (sometimes referred to as a **monotonic transformation**), finding a value that maximizes $\log \mathscr{L}(\theta)$ will also maximize $\mathscr{L}(\theta)$. The expansion of $\log \mathscr{L}(\theta)$ is below:

$$\begin{aligned}
\log \mathscr{L}(\theta) &= \log\left(\prod_{j=1}^{N_h} \theta^{f_i^{(j)}}(1-\theta)^{1-f_i^{(j)}}\right) \\
&= \sum_{j=1}^{N_h} \log\left(\theta^{f_i^{(j)}}(1-\theta)^{1-f_i^{(j)}}\right) \\
&= \sum_{j=1}^{N_h} \log\left(\theta^{f_i^{(j)}}\right) + \sum_{j=1}^{N_h} \log\left((1-\theta)^{1-f_i^{(j)}}\right) \\
&= \log(\theta)\sum_{j=1}^{N_h} f_i^{(j)} + \log(1-\theta)\sum_{j=1}^{N_h}(1-f_i^{(j)})
\end{aligned}$$

Note that in the above derivation, we've used the properties of the log function that $\log(a^c) = c \cdot \log(a)$ and $\log(ab) = \log(a) + \log(b)$. Now we set the derivative of the log of the likelihood function to 0 and solve for θ:

$$
\begin{aligned}
\frac{\partial}{\partial\theta}\left(\log(\theta)\sum_{j=1}^{N_h} f_i^{(j)} + \log(1-\theta)\sum_{j=1}^{N_h}(1-f_i^{(j)})\right) &= 0 \\
\frac{1}{\theta}\sum_{j=1}^{N_h} f_i^{(j)} - \frac{1}{(1-\theta)}\sum_{j=1}^{N_h}(1-f_i^{(j)}) &= 0 \\
\frac{1}{\theta}\sum_{j=1}^{N_h} f_i^{(j)} &= \frac{1}{(1-\theta)}\sum_{j=1}^{N_h}(1-f_i^{(j)}) \\
(1-\theta)\sum_{j=1}^{N_h} f_i^{(j)} &= \theta\sum_{j=1}^{N_h}(1-f_i^{(j)}) \\
\sum_{j=1}^{N_h} f_i^{(j)} - \theta\sum_{j=1}^{N_h} f_i^{(j)} &= \theta\sum_{j=1}^{N_h} 1 - \theta\sum_{j=1}^{N_h} f_i^{(j)} \\
\sum_{j=1}^{N_h} f_i^{(j)} &= \theta \cdot N_h \\
\theta &= \frac{1}{N_h}\sum_{j=1}^{N_h} f_i^{(j)}
\end{aligned}
$$

We've arrived at a remarkably simple final result! According to our formula above, the maximum likelihood estimate for θ (which, remember, is the probability that $P(F_i = 1|Y = ham)$) corresponds to counting the number of ham emails in which word i appears and dividing it by the total number of ham emails. You may think this was a lot of work for an intuitive result (and it was), but the derivation and techniques will be useful for more complex distributions than the simple Bernoulli distribution we are using for each feature here. To summarize, in this Naive Bayes model with Bernoulli feature distributions, within any given class the maximum likelihood estimate for the probability of any outcome corresponds to the count for the outcome divided by the total number of samples for the given class. The above derivation can be generalized to cases where we have more than two classes and more than two outcomes for each feature, though this derivation is not provided here.

Smoothing

Though maximum likelihood estimation is a very powerful method for parameter estimation, bad training data can often lead to unfortunate consequences. For example, if every time the word "minute" appears in an email in our training set, that email is classified as spam, our trained model will learn that

$$P(F_{minute} = 1|Y = ham) = 0$$

Hence in an unseen email, if the word *minute* ever shows up, $P(Y = ham)\prod_i P(F_i|Y = ham) = 0$, and so your model will never classify any email containing the word *minute* as ham. This is a classic example of **overfitting**, or building a model that doesn't generalize well to previously unseen data. Just because a specific word didn't appear in an email in your training data, that doesn't mean that it won't appear in an email in your test data or in the real world. Overfitting with Naive Bayes' classifiers can be mitigated by **Laplace smoothing**. Conceptually, Laplace smoothing with strength k assumes having seen k extra of each outcome. Hence if for a given sample your maximum likelihood estimate for an outcome x that can take on

$|X|$ different values from a sample of size N is

$$P_{MLE}(x) = \frac{count(x)}{N}$$

then the Laplace estimate with strength k is

$$P_{LAP,k}(x) = \frac{count(x)+k}{N+k|X|}$$

What does this equation say? We've made the assumption of seeing k additional instances of each outcome, and so act as if we've seen $count(x)+k$ rather than $count(x)$ instances of x. Similarly, if we see k additional instances of each of $|X|$ classes, then we must add $k|X|$ to our original number of samples N. Together, these two statements yield the above formula. A similar result holds for computing Laplace estimates for conditionals (which is useful for computing Laplace estimates for outcomes across different classes):

$$P_{LAP,k}(x|y) = \frac{count(x,y)+k}{count(y)+k|X|}$$

There are two particularly notable cases for Laplace smoothing. The first is when $k = 0$, then $P_{LAP,0}(x) = P_{MLE}(x)$. The second is the case where $k = \infty$. Observing a very large, infinite number of each outcome makes the results of your actual sample inconsequential and so your Laplace estimates imply that each outcome is equally likely. Indeed:

$$P_{LAP,\infty}(x) = \frac{1}{|X|}$$

The specific value of k that's appropriate to use in your model is typically determined by trial-and-error. k is a hyperparameter in your model, which means that you can set it to whatever you want and see which value yields the best prediction accuracy/performance on your validation data.

Perceptron

Linear Classifiers

The core idea behind Naive Bayes is to extract certain attributes of the training data called features and then estimate the probability of a label given the features: $P(y|f_1, f_2, ...f_n)$. Thus, given a new data point, we can then extract the corresponding features, and classify the new data point with the label with the highest probability given the features. This all, however, this requires us to estimate distributions, which we did with MLE. What if instead we decided not to estimate the probability distribution? Lets start by looking at a simple linear classifier, which we can use for **binary classification**, which is when the label has two possibilities, positive or negative.

The basic idea of a **linear classifier** is to do classification using a linear combination of the features– a value which we call the **activation**. Concretely, the activation function takes in a data point, multiplies each feature of our data point, $f_i(x)$, by a corresponding weight, w_i, and outputs the sum of all the resulting values. In vector form, we can also write this as a dot product of our weights as a vector, $\mathbf{w}$, and our featurized data point as a vector $\mathbf{f}(x)$:

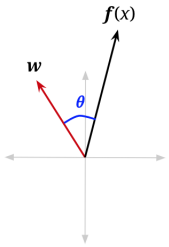
$$\text{activation}_w(x) = \sum_i w_i f_i(x) = \mathbf{w}^T \mathbf{f}(x) = \mathbf{w} \cdot \mathbf{f}(x)$$

How does one do classification using the activation? For binary classification, when the activation of a data point is positive, we classify that data point with the positive label, and if it is negative, we classify with the negative label.

$$\text{classify}(x) = \begin{cases} + & \text{if activation}_w(x) > 0 \\ - & \text{if activation}_w(x) < 0 \end{cases}$$

To understand this geometrically, let us reexamine the vectorized activation function. Using the Law of Cosines, we can rewrite the dot product as follows, where $\|\cdot\|$ is the magnitude operator and θ is the angle between $\mathbf{w}$ and $\mathbf{f}(x)$:

$$\text{activation}_w(x) = \mathbf{w} \cdot \mathbf{f}(x) = \|\mathbf{w}\|\|\mathbf{f}(\mathbf{x})\| \cos(\theta)$$



Since magnitudes are always non-negative, and our classification rule looks at the sign of the activation, the only term that matters for determining the class is $\cos(\theta)$.

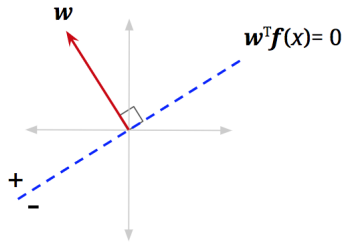
$$\text{classify}(x) = \begin{cases} + & \text{if } cos(\theta) > 0 \\ - & \text{if } cos(\theta) < 0 \end{cases}$$

We, therefore, are interested in when $cos(\theta)$ is negative or postive. It is easily seen that for $\theta < \frac{\pi}{2}$, $cos(\theta)$ will be somewhere in the interval $(0,1]$, which is positive. For $\theta > \frac{\pi}{2}$, $cos(\theta)$ will be somewhere in the interval $[-1,0)$, which is negative. You can confirm this by looking at a unit circle. Essentially, our simple linear classifier is checking to see if the feature vector of a new data point roughly "points" in the same direction as a predefined weight vector and applies a positive label if it does.

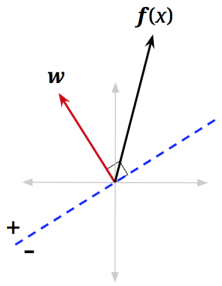
$$\text{classify}(x) = \begin{cases} + & \text{if } \theta < \frac{\pi}{2} \\ - & \text{if } \theta > \frac{\pi}{2} \end{cases} \qquad \begin{array}{l} \text{(i.e. when } \theta \text{ is less than } 90°\text{, or acute)} \\ \text{(i.e. when } \theta \text{ is greater than } 90°\text{, or obtuse)} \end{array}$$

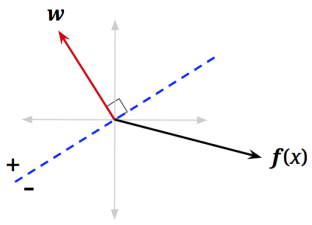
Up to this point, we haven't considered the points where $\text{activation}_w(x) = \mathbf{w}^T\mathbf{f}(x) = 0$. Following all the same logic, we will see that $cos(\theta) = 0$ for those points. Furthermore, $\theta = \frac{\pi}{2}$ (i.e θ is $90°$) for those points. In otherwords, these are the data points with feature vectors that are orthogonal to $\mathbf{w}$. We can add a dotted blue line, orthogonal to $\mathbf{w}$, where any feature vector that lies on this line will have activation equaling 0.

We call this blue line the **decision boundary** because it is the boundary that separates the region where we classify data points as positive from the region of negatives. In higher dimensions, a linear decision boundary is generically called a **hyperplane**. A hyperplane is a linear surface that is one dimesion lower than the latent space, thus dividing the surface in two. For general classifiers (non-linear ones), the decision boundary may not be linear, but is simply defined as surface in the space of feature vectors that separates the classes. To classify points that end up on the decision boundary, we can apply either label since both classes are equally valid (in the algorithms below, we'll classify points on the line as positive).



Decision Boundary





x **classified into positive class** *x* **classified into negative class**

Binary Perceptron

Great, now you know how linear classifiers work, but how do we build a good one? When building a classifier, you start with data, which are labeled with the correct class, we call this the **training set**. You build a classifier by evaluating it on the training data, comparing that to you training labels, and adjusting the parameters of your classifier until you reach your goal.

Let's explore one specific implementation of a simple linear classifier: the binary perceptron. The perceptron is a binary classifier–though it can be extended to work on more than two classes. The goal of the binary perceptron is to find a decision boundary that perfectly separates the training data. In other words, we're seeking the best possible weights– the best **w**– such that any featured training point that is multiplied by the weights, can be perfectly classified.

The Algorithm

The perceptron algorithm works as follows:

1. Initialize all weights to 0: $\mathbf{w} = \mathbf{0}$
2. For each training sample, with features $\mathbf{f}(x)$ and true class label $y^* \in \{-1, +1\}$, do:
   (a) Classify the sample using the current weights, let y be the class predicted by your current $\mathbf{w}$:

$$y = \text{classify}(x) = \begin{cases} +1 & \text{if activation}_w(x) = \mathbf{w}^T \mathbf{f}(x) > 0 \\ -1 & \text{if activation}_w(x) = \mathbf{w}^T \mathbf{f}(x) < 0 \end{cases}$$

   (b) Compare the predicted label y to the true label y^*:
   - If $y = y^*$, do nothing
   - Otherwise, if $y \neq y^*$, then update your weights: $\mathbf{w} \leftarrow \mathbf{w} + y^* \mathbf{f}(x)$

3. If you went through **every** training sample without having to update your weights (all samples predicted correctly), then terminate. Else, repeat step 2

Updating weights

Let's examine and justify the procedure for updating our weights. Recall that in step 2b above, when our classifier is right, nothing changes. But when our classifier is wrong, the weight vector is updated as follows:

$$\mathbf{w} \leftarrow \mathbf{w} + y^* \mathbf{f}(x)$$

where y^* is the true label, which is either 1 or -1, and x is the training sample which we mis-classified. You can interpret this update rule to be:

Case 1 : mis-classified positive as negative	$\mathbf{w} \leftarrow \mathbf{w} + \mathbf{f}(x)$
Case 2 : mis-classified negative as positve	$\mathbf{w} \leftarrow \mathbf{w} - \mathbf{f}(x)$

Why does this work? One way to look at this is to see it as a balancing act. Mis-classification happens either when the activation for a training sample is much smaller than it should be (causes a Case 1 misclassification) or much larger than it should be (causes a Case 2 misclassification).

Consider Case 1, where activation is negative when it should be positive. In otherwords, the activation is too small. How we adjust $\mathbf{w}$ should strive to fix that and make the activation larger for that training sample. To convince yourself that our update rule $\mathbf{w} \leftarrow \mathbf{w} + \mathbf{f}(x)$ does that, let us update $\mathbf{w}$ and see how the activation changes.

$$\text{activation}_{w+x}(x) = (\mathbf{w} + \mathbf{f}(x))^T \mathbf{x} = \mathbf{w}^T \mathbf{f}(x) + \mathbf{f}(x)^T \mathbf{f}(x) = \text{activation}_w(x) + \mathbf{f}(x)^T \mathbf{f}(x)$$

Using our update rule, we see that the new activation increases by $\mathbf{f}(x)^T \mathbf{f}(x)$, which is a postive number, therefore showing that our update makes sense. Activation is getter larger– closer to becoming positive. You can repeat the same logic for when the classifier is mis-classifying because the activation is too large (activation is positive when it should be negative). You'll see that the update will cause the new activation to decrease by $\mathbf{f}(x)^T \mathbf{f}(x)$, thus getting smaller and closer to classifying correctly.

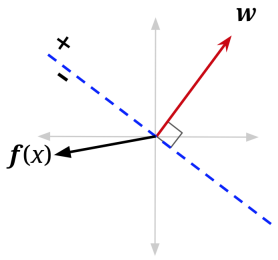
While this makes it clear why we are adding and subtracting *something*, why would we want to add and subtract our sample point's features? A good way to think about it, is that the weights aren't the only thing that determines this score. The score is determined by multiplying the weights by the relevant sample. This means that certain parts of a sample contribute more than others. Consider the following situation where x is a training sample we are given with true label $y^* = -1$:

$$\mathbf{w}^T = \begin{bmatrix} 2 & 2 & 2 \end{bmatrix}, \mathbf{f}(x) = \begin{bmatrix} 4 \\ 0 \\ 1 \end{bmatrix} \qquad \text{activation}_w(x) = (2*4) + (2*0) + (2*1) = 10$$

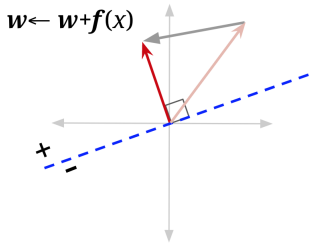
We know that our weights need to be smaller because activation needs to be negative to classify correctly. We don't want to change them all the same amount though. You'll notice that the first element of our sample, the 4, contributed much more to our score of 10 than the third element, and that the second element did not contribute at all. An appropriate weight update, then, would change the first weight alot, the third weight a little, and the second weight should not be changed at all. After all, the second and third weights might not even be that broken, and we don't fix want to fix what isn't broken!

When thinking about a good way to change our weight vector in order to fulfill the above desires, it turns out just using the sample itself does in fact do what we want; it changes the first weight by alot, the third weight by a little, and doesn't change the second weight at all! *Note that this is similar to our updates in approximate Q-learning, which also updated weights by multiples of the feature vectors.*

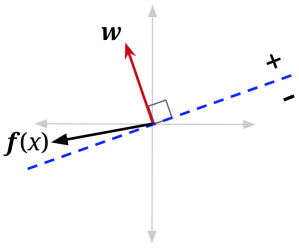
A visualization may also help. In the figure below, $\mathbf{f}(x)$ is the feature vector for a data point with positive class ($y^* = +1$) that is currently misclassified – it lies on the wrong side of the decision boundary defined by "old $\mathbf{w}$". Adding it to the weight vector produces a new weight vector which has a smaller angle to $\mathbf{f}(x)$. It also shifts the decision boundary. In this example, it has shifted the decision boundary enough so that x will now be correctly classified (note that the mistake won't always be fixed – it depends on the size of the weight vector, and how far over the boundary $\mathbf{f}(x)$ currently is).



Mis-classifying x with old w



Updating w



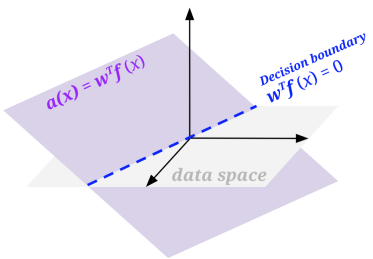
Updated classification of x

Bias

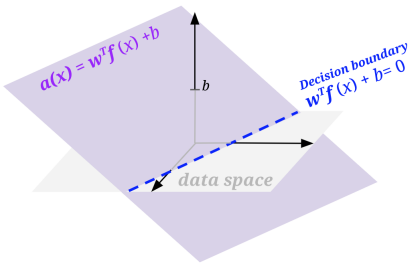
If you tried to implement a perceptron based on what has been mentioned thus far, you will notice one particularly unfriendly quirk. Any decision boundary that you end up drawing will be crossing the origin. Basically, your perceptron can only produce a decision boundary that could be represented by the function $\mathbf{w}^\top \mathbf{f}(x) = 0$, $\mathbf{w}, \mathbf{f}(x) \in \mathbb{R}^n$. The problem is, even among problems where there is a linear decision boundary that separates the positive and negative classes in the data, that boundary may not go through the origin, and we want to be able to draw those lines.

To do so, we will modify our feature and weights to add a bias term: add a feature to your sample feature vectors that is always 1, and add an extra weight for this feature to your weight vector. Doing so essentially allows us to produce a decision boundary representable by $\mathbf{w}^\top \mathbf{f}(x) + b = 0$, where b is the weighted bias term (i.e. 1 * the last weight in the weight vector).

Geometrically, we can visualize this by thinking about what the activation function looks like when it is $\mathbf{w}^\top \mathbf{f}(x)$ and when there is a bias $\mathbf{w}^\top \mathbf{f}(x) + b$. To do so, we need to be one dimension higher than the space of our featurized data (labeled data space in the figures below). In all the above sections, we had only been looking at a flat view of the data space.



Without bias



With bias

Example

Up until now, we have been focused on alot of reading, let's shake things up a little and see an example, and run the perceptron algorithm step by step.

Let's run one pass through the data with the perceptron algorithm, taking each data point in order. We'll start with the weight vector $[w_0, w_1, w_2] = [-1, 0, 0]$ (where w_0 is the weight for our bias feature, which remember is always 1).

Training Set

#	f_1	f_2	y^*
1	1	1	-
2	3	2	+
3	2	4	+
4	3	4	+
5	2	3	-

Single Perceptron Update Pass

step	Weights	Score	Correct?	Update
1	[-1, 0, 0]	$-1 \cdot 1 + 0 \cdot 1 + 0 \cdot 1 = -1$	yes	none
2	[-1, 0, 0]	$-1 \cdot 1 + 0 \cdot 3 + 0 \cdot 2 = -1$	no	+[1, 3, 2]
3	[0, 3, 2]	$0 \cdot 1 + 3 \cdot 2 + 2 \cdot 4 = 14$	yes	none
4	[0, 3, 2]	$0 \cdot 1 + 3 \cdot 3 + 2 \cdot 4 = 17$	yes	none
5	[0, 3, 2]	$0 \cdot 1 + 3 \cdot 2 + 2 \cdot 3 = 12$	no	-[1, 2, 3]
6	[-1, 1, -1]			

We'll stop here, but in actuality this algorithm would run for many more passes through the data before all the data points are classified correctly in a single pass.

Multiclass Perceptron

The perceptron presented above is a binary classifier, but we can extend it to account for multiple classes rather easily. The main difference is in how we set up weights and how we update said weights. For the binary case, we had one weight vector, which had a dimension equal to the number of features (plus the bias feature). For the multi-class case, we will have one weight vector for each class, so in the 3 class case, we have 3 weight vectors. In order to classify a sample, we compute a score for each class by taking the dot product of the feature vector with each of the weight vectors. Whichever class yields the highest score is the one we choose as our prediction.

For example, consider the 3-class case. Let our sample have features $\mathbf{f}(x) = \begin{bmatrix} -2 & 3 & 1 \end{bmatrix}$ and our weights for classes 0, 1, and 2 be:

$$\mathbf{w}_0 = \begin{bmatrix} -2 & 2 & 1 \end{bmatrix}$$
$$\mathbf{w}_1 = \begin{bmatrix} 0 & 3 & 4 \end{bmatrix}$$
$$\mathbf{w}_2 = \begin{bmatrix} 1 & 4 & -2 \end{bmatrix}$$

Taking dot products for each class gives us scores $s_0 = 11, s_1 = 13, s_2 = 8$. We would thus predict that x belongs to class 1.

An important thing to note is that in actual implementation, we do not keep track of the weights as separate structures, we usually stack them on top of each other to create a weight matrix. This way, instead of doing as many dot products as there are classes, we can instead do a single matrix-vector multiplication. This tends to be much more efficient in practice (because matrix-vector multiplication usually has a highly optimized implementation).

In our above case, that would be:

$$W = \begin{bmatrix} -2 & 2 & 1 \\ 0 & 3 & 4 \\ 1 & 4 & -2 \end{bmatrix}, x = \begin{bmatrix} -2 \\ 3 \\ 1 \end{bmatrix}$$

And our label would be:

$$\arg\max(Wx) = \arg\max(\begin{bmatrix} 11 \\ 13 \\ 8 \end{bmatrix}) = 1$$

Along with the structure of our weights, our weight update also changes when we move to a multi-class case. If we correctly classify our data point, then do nothing just like in the binary case. If we chose incorrectly, say we chose class $y \neq y^*$, then we add the feature vector to the weight vector for the true class to y^*, and subtract the feature vector from the weight vector corresponding to the predicted class y. In our above example, let's say that the correct class was class 2, but we predicted class 1. We would now take the weight vector corresponding to class 1 and subtract x from it,

$$w_1 = \begin{bmatrix} 0 & 3 & 4 \end{bmatrix} - \begin{bmatrix} -2 & 3 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 0 & 3 \end{bmatrix}$$

Next we take the weight vector corresponding to the correct class, class 2 in our case, and add x to it:

$$w_2 = \begin{bmatrix} 1 & 4 & -2 \end{bmatrix} + \begin{bmatrix} -2 & 3 & 1 \end{bmatrix} = \begin{bmatrix} -1 & 7 & -1 \end{bmatrix}$$

What this amounts to is 'rewarding' the correct weight vector, 'punishing' the misleading, incorrect weight vector, and leaving alone an other weight vectors. With the difference in the weights and weight updates taken into account, the rest of the algorithm is essentially the same; cycle through every sample point, updating weights when a mistake is make, until you stop making mistakes.

In order to incorporate a bias term, do the same as we did for binary perceptron – add an extra feature of 1 to every feature vector, and an extra weight for this feature to every class's weight vector (this amounts to adding an extra column to the matrix form).

Summary

In this note, we introduced several fundamental principles of machine learning, including:

- Splitting your data into training data, validation data, and test data.
- The difference between supervised learning, which learns from labeled data, and unsupervised learning, which doesn't have labeled data and so attempts to infer inherent structure from it.

We then proceeded to discuss an example of a supervised learning algorithm for classification, Naive Bayes', which uses parameter estimation to construct conditional probability tables within a Bayes' net before running inference to predict the class labels of datapoints. We extended this idea to discuss the problem of overfitting in the context of Naive Bayes' and how this issue can be mitigated with Laplace smoothing. Finally, we talked about perceptrons and decision boundaries - methods for classification that don't explicitly store conditional probability tables.

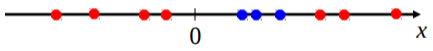
CS 188 Introduction to Artificial Intelligence
Fall 2017 Note 10

Neural Networks: Motivation

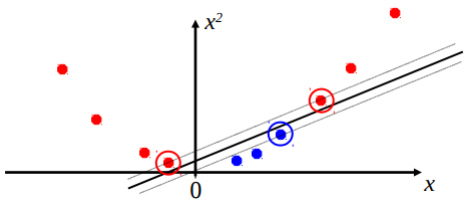
Non-linear Separators

We know how to construct a model that learns a linear boundary for binary classification tasks. This is a powerful technique, and one that works well when the underlying optimal decision boundary is itself linear. However, many practical problems involve the need for decision boundaries that are nonlinear in nature, and our linear perceptron model isn't expressive enough to capture this relationship.

Consider the following set of data:



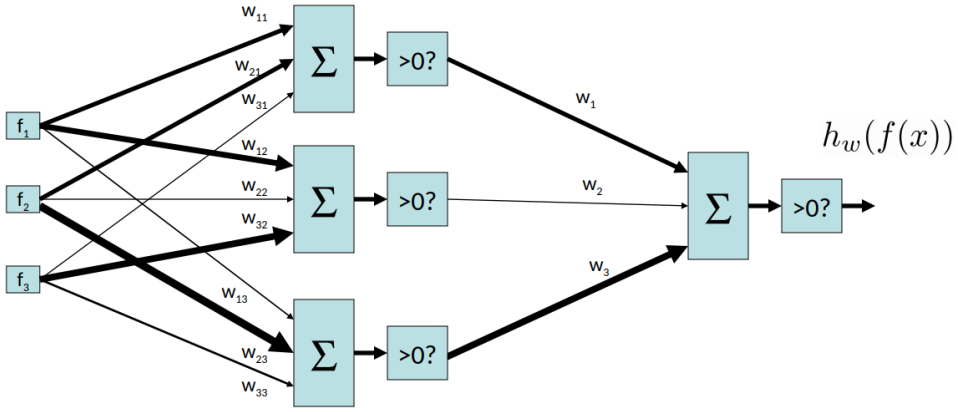
We would like to separate the two colors, and clearly there is no way this can be done in a single dimension (a single dimensional decision boundary would be a point, separating the axis into two regions).
To fix this problem, we can add additional (potentially nonlinear) features to construct a decision boundary from. Consider the same dataset with the addition of x^2 as a feature:



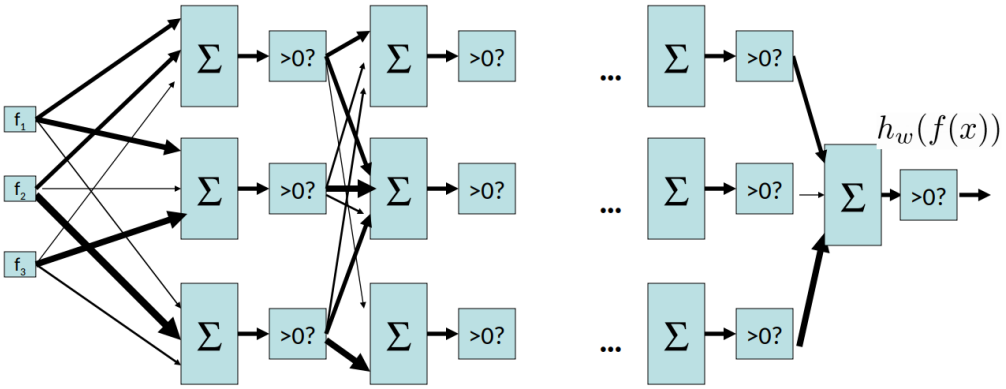
With this additional piece of information, we are now able to construct a linear separator in the two dimensional space containing the points. In this case, we were able to fix the problem by mapping our data to a higher dimensional space by manually adding useful features to datapoints. However, in many high-dimensional problems, such as image classification, manually selecting features that are useful is a tedious problem. This requires domain-specific effort and expertise, and works against the goal of generalization across tasks. A natural desire is to learn these featurization or transformation functions as well, perhaps using a nonlinear function class that is capable of representing a wider variety of functions.

Multi-layer Perceptron

Let's examine how we can derive a more complex function from our original perceptron architecture. Consider the following setup, a two-layer perceptron, which is a perceptron that takes as input the outputs of another perceptron.



In fact, we can generalize this to an N-layer perceptron:



With this additional structure and weights, we can express a much wider set of functions.

By increasing the complexity of our model, we in turn greatly increase its expressive power. Multi-layer perceptrons give us a generic way to represent a much wider set of functions. In fact, a multi-layer perceptron is a **universal function approximator** and can represent *any* real function, leaving us only with the problem of selecting the best set of weights to parameterize our network. This is formally stated below:

Theorem. (Universal Function Approximators) A two-layer neural network with a sufficient number of neurons can approximate any continuous real-valued function to any desired accuracy.

Measuring Accuracy

The accuracy of the binary perceptron after making m predictions can be expressed as:

$$l^{acc}(\boldsymbol{w}) = \frac{1}{m}\sum_{i=1}^{m} \mathbb{1}(\text{sgn}(\boldsymbol{w}\cdot\mathbf{f}(x^{(i)})) == y^{(i)})$$

where $x^{(i)}$ is datapoint i, $\mathbf{w}$ is our weight vector, $\mathbf{f}$ is our function that derives a feature vector from a raw datapoint, and $y^{(i)}$ is the actual class label of $x^{(i)}$. In this context, $\text{sgn}(x)$ represents the **sign function**, which

evaluates to -1 when x is negative, and 1 when x is positive. Similarly, $\mathbb{1}(x)$ is an **indicator function**, which evaluates to 1 if the quantities within are equivalent and 0 otherwise. Taking this notation into account, we can note that our accuracy function above is equivalent to dividing the total number of correct predictions by the raw total number of predictions.

Sometimes, we want an output that is more expressive than a binary label. It then becomes useful to produce a probability for each of the N classes we want to classify into, which reflects our a degree of certainty that the datapoint belongs to each of the possible classes. To do so, we transition from storing a single weight vector to storing a weight vector for *each* class j, and estimate probabilities with the **softmax function** $\sigma(x)$. The softmax function defines the probability of classifying $x^{(i)}$ to class j as:

$$\sigma(x^{(i)})_j = \frac{e^{f(x^{(i)})^T w_j}}{\sum_{k=1}^{N} e^{f(x^{(i)})^T w_k}} = P(y^{(i)} = j | x^{(i)})$$

Given a vector that is output by our function f, softmax performs normalization to output a probability distribution. To come up with a general loss function for our models, we can use this probability distribution to generate an expression for the likelihood of a set of weights:

$$\ell(\boldsymbol{w}) = \prod_{i=1}^{m} P(y^{(i)} | x^{(i)}; \boldsymbol{w})$$

This expression denotes the likelihood of a particular set of weights explaining the observed labels and datapoints. We would like to find the set of weights that maximizes this quantity. This is identical to finding the maximum of the log-likelihood expression (since log is an increasing function, the maximizer of one will be the maximizer of the other):

$$\ell\ell(\boldsymbol{w}) = \log \prod_{i=1}^{m} P(y^{(i)} | x^{(i)}; \boldsymbol{w}) = \sum_{i=1}^{m} \log P(y^{(i)} | x^{(i)}; \boldsymbol{w})$$

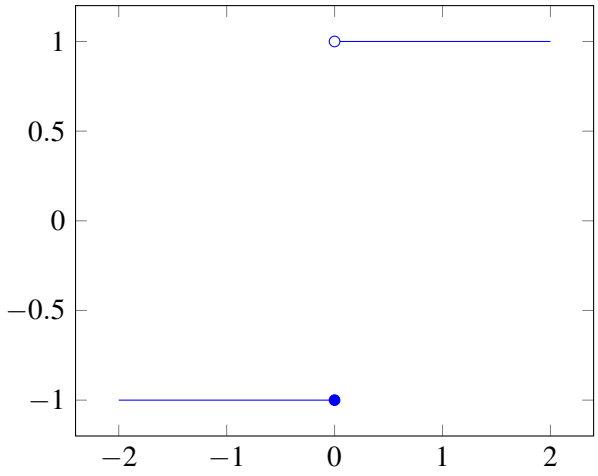
(Depending on the application, the formulation as a sum of log probabilities may be more useful – for example in mini-batched or stochastic gradient descent; see the *Neural Networks: Optimization* section below.) In the case where the log likelihood is differentiable with respect to the weights, we will discuss a simple algorithm to optimize it.

Multi-layer Feedforward Neural Networks

We now introduce the idea of an (artificial) neural network. This is much like the multi-layer perceptron, however, we choose a different non-linearity to apply after the individual perceptron nodes. Note that it is these added non-linearities that makes the network as a whole non-linear and more expressive (without them, a multi-layer perceptron would simply be a composition of linear functions and hence also linear). In the case of a multi-layer perceptron, we chose a step function:

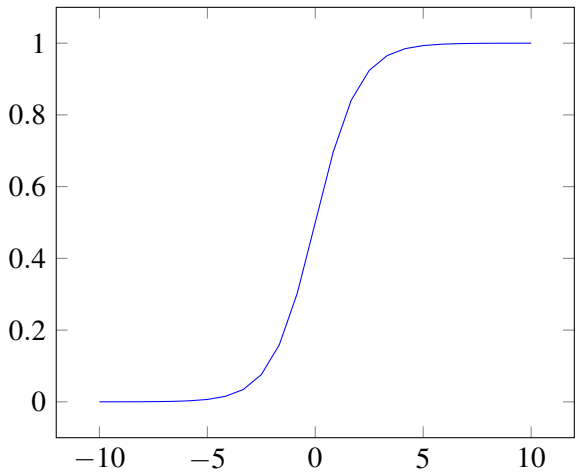
$$f(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

Let's take a look at its graph:

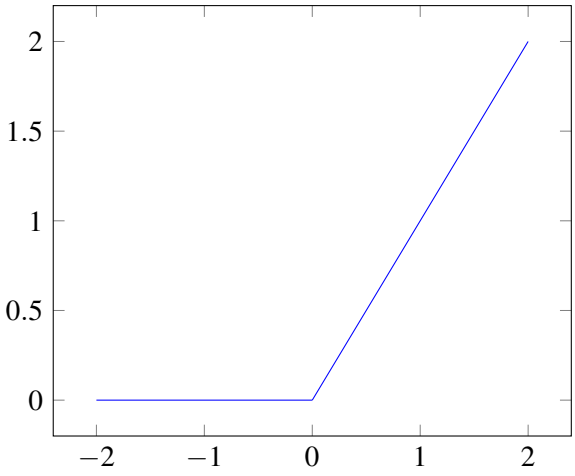


This is difficult to optimize for a number of reasons which will hopefully become clearer when we address gradient descent. Firstly, it is not continuous, and secondly, it has a derivative of zero at all points. Intuitively, this means that we cannot know in which direction to look for a local minima of the function, which makes it difficult to minimize loss in a smooth way.
Instead of using a step function like above, a better solution is to select a continuous function. We have many options for such a function, including the **sigmoid function** (named for the Greek σ or 's' as it looks like an 's') as well as the **rectified linear unit** (ReLU).Let's look at their definitions and graphs below:

Sigmoid: $\sigma(x) = \frac{1}{1+e^{-x}}$



ReLU: $f(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$



Calculating the output of a multi-layer perceptron is done as before, with the difference that at the output of each layer we now apply one of our new non-linearities (chosen as part of the architecture for the neural network) instead of the initial indicator function. In practice, the choice of nonlinearity is a design choice that typically requires some experimentation to select a good one for each individual use case.

Loss Functions and Multivariate Optimization

Now we have a sense of how a feed-forward neural network is constructed and makes its predictions, we would like to develop a way to train it, iteratively improving its accuracy, similarly to how we did in the case of the perceptron. In order to do so, we will need to be able to measure their performance. Returning to our log-likelihood function that we wanted to maximize, we can derive an intuitive algorithm to optimize our weights given that our function is differentiable.

Gradient Ascent / Descent

To maximize our log-likelihood function, we differentiate it to obtain a **gradient vector** consisting of its partial derivatives for each parameter:

$$\nabla_w \ell\ell(\mathbf{w}) = \left[\frac{\partial \ell\ell(\mathbf{w})}{\partial \mathbf{w}_1}, \ldots, \frac{\partial \ell\ell(\mathbf{w})}{\partial \mathbf{w}_n}\right]$$

This gradient vector gives the local direction of steepest ascent (or descent if we reverse the vector). **Gradient ascent** is a greedy algorithm that calculates this gradient for the current values of the weight parameters, then updates the parameters along the direction of the gradient, scaled by a **step size**, α. Specifically the algorithm looks as follows:

Initialize weights $\boldsymbol{w}$
For i = 0, 1, 2, ...
  $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha \nabla_w \ell\ell(\boldsymbol{w})$

If rather than minimizing we instead wanted to minimize a function f, the update should subtract the scaled gradient ($\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla_{\mathbf{w}} f(\mathbf{w})$) – this gives the **gradient descent** algorithm.

Neural Networks: Optimization

Now that we have a method for computing gradients for all parameters of the network, we can use gradient descent methods to optimize the parameters to get high accuracy on our training data. For example, suppose we have designed some classification network to output probabilities of classes y for data points x, and have m different training datapoints (see the *Measuring Accuracy* section for more on this). Let $\mathbf{w}$ be all the parameters of our network. We want to find values for the parameters $\mathbf{w}$ that maximize the likelihood of the true class probabilities for our data, so we have the following function to run gradient ascent on:

$$\ell\ell(\mathbf{w}) = \log \prod_{i=1}^{m} P(y^{(i)} \mid x^{(i)}; \mathbf{w}) = \sum_{i=1}^{m} \log P(y^{(i)} \mid x^{(i)}; \mathbf{w})$$

where $x^{(1)}, \ldots, x^{(m)}$ are the m datapoints in our training set.

One way to try to minimize this function is, at each iteration of gradient descent, to use all the data points $x^{(1)}, \ldots, x^{(m)}$ to compute gradients for the parameters $\mathbf{w}$, update the parameters, and repeat until the parameters converge (at which point we've reached a local minimum of the function).

This technique, known as **batch gradient descent**, is rarely done in practice, since datasets are typically large enough that computing gradients for this full likelihood function will be very slow. Instead, we'll typically use **mini-batching**. Mini-batching rotates through randomly sampled batches of k data points at a time, taking one batch for each step of gradient descent and computing gradients of the loss function using only that batch (so that the sum above is over the k datapoints in the batch, rather than all m datapoints in the training set). This allows us to compute each gradient update much more quickly, and often still makes fast progress toward the minimum of the function. The limit where the batch size $k = 1$ is known as **stochastic gradient descent (SGD)**. In SGD, we randomly sample a single example from the training dataset at each step of gradient descent, compute parameter gradients using the network's loss on that single example, update the parameters, and repeat (sampling another example from the training set).

Neural Networks: Backpropagation (Optional)

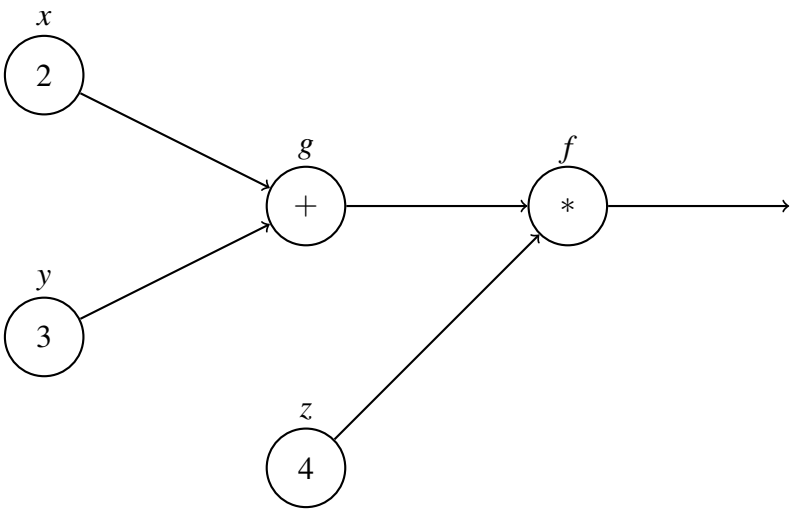
To efficiently calculate the gradients for each parameter in a neural network, we will use an algorithm known as **backpropagation**. Backpropagation represents the neural network as a dependency graph of operators and operands, called a **computational graph**, such as the one shown below:

The graph structure allows us to efficiently compute both the network's error (loss) on input data, as well as the gradients of each parameter with respect to the loss. These gradients can be used in gradient descent to adjust the network's parameters and minimize the loss on the training data.

The Chain Rule

The chain rule is the fundamental rule from calculus which both motivates the usage of computation graphs and allows for a computationally feasible backpropagation algorithm. Mathematically, it states that for a variable z which is a function of n variables $x_1, \ldots, x_n$ and each x_i is a function of m variables $t_1, \ldots, t_m$, then we can compute the derivative of z with respect to any t_i as follows:

$$\frac{\partial f}{\partial t_i} = \frac{\partial f}{x_1} \cdot \frac{\partial x_1}{\partial t_i} + \frac{\partial f}{x_2} \cdot \frac{\partial x_2}{\partial t_i} + \ldots + \frac{\partial f}{x_n} \cdot \frac{\partial x_n}{\partial t_i}$$



In the context of computation graphs, this means that to compute the gradient of a given node t_i with respect to the output z, we take a sum of children(t_i) terms.

The Backpropagation Algorithm

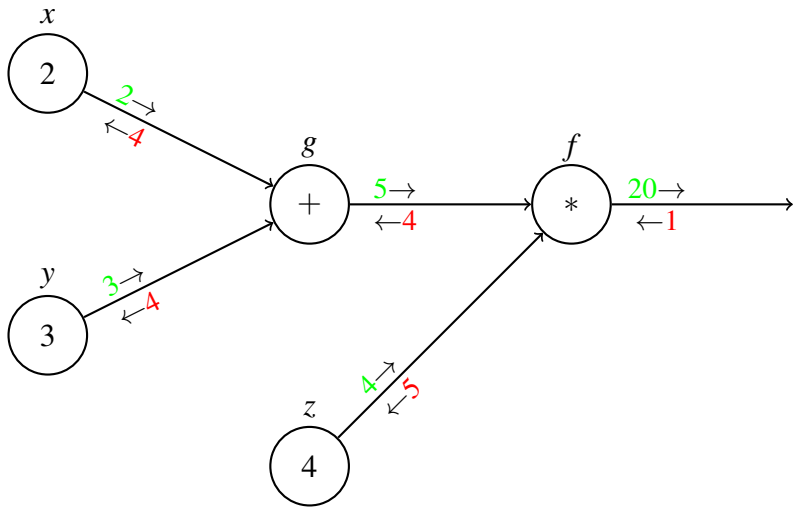


Figure 1: A computation graph for computing $(x+y)*z$ with the values $x=2$, $y=3$, $z=4$.

Figure 1 shows an example computation graph for computing $(x+y)*z$ with the values $x=2$, $y=3$, $z=4$. We will write $g=x+y$ and $f=g*z$. Values in green are the outputs of each node, which we compute in the **forward pass**, where we apply each node's operation to its input values coming from its parent nodes.

Values in red after each node give gradients of the function computed by the graph, which are computed in the **backward pass**: the value after each node is the partial derivative of the last node f value with respect to the variable at that node. For example, the red value 4 after g is $\frac{\partial f}{\partial g}$, and the red value 4 after x is $\frac{\partial f}{\partial x}$. In our simple example, f is just a multiplication node which outputs the product of its two input operands, but in a real neural network the final node will usually compute the loss value that we are trying to minimize.

The backward pass computes gradients by starting at the final node (which has a gradient of 1 since $\frac{\partial f}{\partial f}=1$) and passing and updating gradients backward through the graph. Intuitively, each node's gradient measures

how much a change in that node's value contributes to a change in the final node's value. This will be the product of how much the node contributes to a change in its child node, with how much the child node contributes to a change in the final node. Each node receives and combines gradients from its children, updates this combined gradient based on the node's inputs and the node's operation, and then passes the updated gradient backward to its parents. Computation graphs are a great way to visualize repeated application of the chain rule from calculus, as this process is required for backpropagation in neural networks.

Our goal during backpropagation is to determine the gradient of output with respect to each of the inputs. As you can see in Figure 1, in this case we want to compute the gradients $\frac{\partial f}{\partial x}$, $\frac{\partial f}{\partial y}$, and $\frac{\partial f}{\partial z}$:

1. Since f is our final node, it has gradient $\frac{\partial f}{\partial f} = 1$. Then we compute the gradients for its children, g and z. We have $\frac{\partial f}{\partial g} = \frac{\partial}{\partial g}(g \cdot z) = z = 4$, and $\frac{\partial f}{\partial z} = \frac{\partial}{\partial z}(g \cdot z) = g = 5$.

2. Now we can move on upstream to compute the gradients of x and y. For these, we'll use the chain rule and reuse the gradient we just computed for g, $\frac{\partial f}{\partial g}$.

3. For x, we have $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial x}$ by the chain rule – the product of the gradient coming from g with the partial derivative for x at this node. We have $\frac{\partial g}{\partial x} = \frac{\partial}{\partial x}(x+y) = \frac{\partial}{\partial x}x + \frac{\partial}{\partial x}y = 1+0$, so $\frac{\partial f}{\partial x} = 4 \cdot 1 = 4$. Intuitively, the amount that a change in x contributes to a change in f is the product of the amount that a change in g contributes to a change in f, with the amount that a change in x contributes to one in g.

4. The process for computing the gradient of the output with respect to y is almost identical. For y we have $\frac{\partial f}{\partial y} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial y}$ by the chain rule – the product of the gradient coming from g with the partial derivative for y at this node. We have $\frac{\partial g}{\partial y} = \frac{\partial}{\partial y}(x+y) = \frac{\partial}{\partial y}x + \frac{\partial}{\partial y}y = 0+1$, so $\frac{\partial f}{\partial y} = 4 \cdot 1 = 4$.

Since the backward pass step for a node in general depends on the node's inputs (which are computed in the forward pass), and gradients computed "downstream" of the current node by the node's children (computed earlier in the backward pass), we cache all of these values in the graph for efficiency. Taken together, the forward and backward pass over the graph make up the backpropagation algorithm.

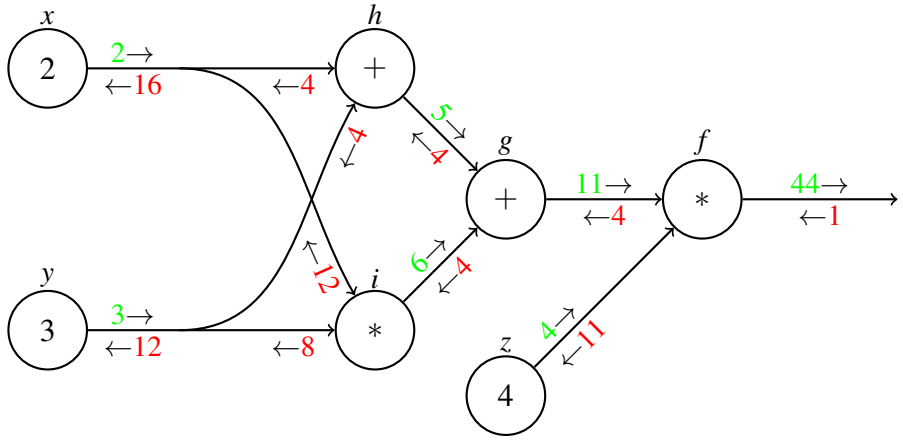


Figure 2: A computation graph for computing $((x+y)+(x \cdot y)) \cdot z$, with $x = 2$, $y = 3$, $z = 4$.

For an example of applying the chain rule for a node with multiple children, consider the graph in Figure 2, representing $((x+y)+(x \cdot y)) \cdot z$, with $x = 2$, $y = 3$, $z = 4$. x and y are each used in 2 operations, and so each has two children. By the chain rule, their gradient values are the sum of the gradients computed for them

by their children (i.e. gradient values add at path junctions). For example, to compute the gradient for x, we have

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial h}\frac{\partial h}{\partial x} + \frac{\partial f}{\partial i}\frac{\partial i}{\partial x} = 4 \cdot 1 + 4 \cdot 3 = 4 + 12 = 16$$

Conclusion

In this note, we generalized the perceptron to neural networks, models which are powerful (and universal!) function approximators but can be difficult to design and train. We talked about how the expressiveness of neural networks comes from the activation functions they employ to introduce nonlinearities, as well as how to optimize their parameters with backpropagation and gradient descent. Currently, a large portion of new research in machine learning focuses on various aspects of neural network design such as:

1. *Network Architectures* - designing a network (choosing activation functions, number of layers, etc.) that's a good fit for a particular problem
2. *Learning Algorithms* - how to find parameters that achieve a low value of the loss function, a difficult problem since gradient descent is a greedy algorithm and neural nets can have many local optima
3. *Generalization and Transfer Learning* - since neural nets have many parameters, it's often easy to overfit training data - how do you guarantee that they also have low loss on testing data you haven't seen before?