**CIS-479 Artificial Intelligence**

**With Professor Shengquan Wang**

**Personal Notes – Meech**

**Summer II 2022**

# Live Lecture 1: Introduction & Uniformed/Informed Search (ch1, ch2, & ch3) – 6/29/2022

* Professor has a 3rd project/program that we can ask him for (but it won’t be for a grade).
* For Artificial Intelligence, we need data and probability and deterministic models. Data and probability is the pretense for intelligence – in order to make predictions.
* Voice recognition is AI (i.e., Alexa).
* When your phone guesses the rest of a sentence you are typing = AI.
* Challenge for AI: How to deal with all of the special cases (dynamic environments, like driving) – humans can make good analysis, but AI cannot.
* In well-defined (limited) environments, AI can easily outperform humans (i.e., chess, video games, etc.).
* Goal for AI vehicles: extremely low error rate (probably even better than human error rate).
  + And to somehow adapt and learn to handle special cases.
* Humans are like the parents of AI: we help it to learn and grow.
* Model-based reflex agents: successfully interact with an environment.
* Goal-based agents: successfully interact with an environment and know the state (binary) if some goal is achieved or not.
* Utility-based agents: use model-based agent +goal based + **optimization** within the environment to successfully interact with the environment and achieve some (final) goal.
* Sensor = input hardware, Actuators = output hardware.
* Deterministic: knowing (with usually high or even exact precision) all possible outcomes (and frequency) of an action.
* Stochastic: random; cannot be easily predicted.
  + Definition: randomly determined; having a random probability distribution or pattern that may be analyzed statistically but may not be predicted precisely.
* Implement the environment for the 8-Puzzle environment – will be our first program – will talk about it Wednesday.
* Time complexity and space (memory) are the limitations of search efficient/optimal algorithms.
* A lot of CIS-275 and CIS-276 (Discrete structures 1 and 2) principles (graph and state theory).
* For explore set, we can use a hash table.
* Frontier set can have redundancy that causes a loop.
* It is okay to add a note to the frontier multiple times because there may be a shorter path available on one of the redundantly occurring nodes.
* Windy Maze will be in HW1
  + No diagonal movement allowed in windy maze.
* N-puzzle problem will be helpful for our program 1.
* Breadth-first search (BFS) is really bruit force – it takes a lot of memory since it expands so many levels.
* BFS tries to expand and map the environment.
* DFS (depth-first) is brute force as well, but it risks testing a full path.
* DFS is linear for space (but exponential just like BFS with explore set since you have to maintain which nodes already explored – which requires same amount of memory as BFS).
* Example of a Model-based reflex agent: computer circuit board.
* One of the primary ways we find solutions is by using probability: find/expand all possible combinations in the set (get all possible states). We can do this with computers because they can compute these sets so quickly, so it is a (less efficient/nonoptimal) very reliable way for finding solutions to problems.

## Online Lecture 1 – (Review of live lecture)

* Learning comes through many percept-action cycles.
* Easier to make AI models for a deterministic problem (like with games – i.e. chess, video games, etc.).
  + With cars for example, you cannot describe every possible situation – it is stochastic (needs probability for an *approximation* of all possible scenarios).
* Actuators: interact with the environment.
* General AI Agent
  + Diagram

    Description automatically generated
* Agent Types
  + Graphical user interface, text, application

    Description automatically generated
* Need to understand PEAS (performance, environment, actuators, sensors) when developing an AI agent for some system.
* Deterministic environments are usually fully observable.
* For partially observable environments, we have to use prediction (probability).
* Episodic: not dependent on previous action; sequential does depend on previous action.
* We digitize the state space even for continuous state space problems because computers are digital – they are inherently discrete (unlock analog, which is continuous (infinitely many states and “in-between” states) and thus can have a continuous state).
* Deterministic: we know the outcome of the action. For stochastic, we make a prediction of the outcome of an action (or input from environment).
* Chart, radar chart

  Description automatically generated
  + Above, notice the map above can be converted into a tree.
* Frontier: Set of **all** leaf nodes available for expansion at any given point (in time / during traversal).
  + It is essentially all leaf nodes.
  + For data structure of a frontier: select leaf node, remove it from frontier, add the resulting new nodes to the frontier – so you see it is a add-and-remove process.
  + Explore set data structure is just like frontier but removes redundancies: but we do not ever re-add any nodes that has previously been added to frontier (or that currently exists in the frontier).
    - Use an array or hash table to keep track of nodes already added to frontier, so that before a node is added, we can do a table lookup to make sure the node has not already been added (whether in the past, or currently in the frontier) – it is a membership query.
* Implementation (Data structures) of Frontier and Explored Sets
  + Graphical user interface, text, application

    Description automatically generated
* For frontier set, you still add nodes even that have already been expanded, if they show up again when expanding another node. This is because there could be a path that involves that node that was already explored that could be a part of a shorter path.
* For BFS and DFS, they are brute force uninformed searches; they can find the optimal path, but you have to wait for all possible solutions (path) to be mapped out so that you can compare them all and then finally determine which is the shortest path. So, usually use brute force approaches are used only to find a solution, not necessarily the optimal one.
* Diagram

  Description automatically generated with low confidence
  + Above, b refers to max children for each level (you can get an average number of children per level, which is largely determined by bottom-most levels). As b grows to infinity then, you the only b that really matters is the b (the level) with the most children that will most affect the time and/or space complexity of a search. The same exact logic goes for m (depth), where for very large m, it will greatly determine the complexity of the algorithm.
* UCS is also an uniformed search algorithm but can find optimal path with much better complexity than BFS or DFS.
* UCS and BFS can both find optimal solution if cost = 1. DFS can too, but not automatically – its first solution it finds could be a non-optimal even if cost is 1 everywhere.
  + UCS can find the shortest path from some origin to each node in the tree in one sweep since each explored parent keeps track and adds to its cost for the path cost.
* Application

  Description automatically generated with low confidence
  + C\*/E = think of this as depth (cost).
  + UCS == Dijkstra’s algorithm; it works much better when we have some heuristics (informed search information).
* Tree search v graph search: tree search uses frontier set, graph search uses explore set.
* Text

  Description automatically generated with medium confidence
  + Above, the reason why DFS is exponential for graph search is because graph search uses the explore set, which then you have to maintain that table in memory to remember what you already explored, thus each time you search a node you have to check (the hash table or some data structure holding the data for remembering what is in the explore set) to check for each expansion before adding nodes to the set – so that is additional space in memory that must be maintained during the entire search. So, it is better to just be redundant and do tree search (uses frontier set – which allows for some redundancy) with DFS in order to greatly save space: going from exponential with graph search to only linear space with tree search – it is just that the redundancy will obviously exchange the space cost for more time cost.
    - Think about it: if I search a depth of m, then redundantly search the depth of m 5 more times, then that is only 5m. Which for very large m (as m approaches infinity), that factor becomes irrelevant 🡪 reduces to just m = linear complexity.
  + Above, dense refers there being many solutions along many branches (probability of a branch containing a solution is higher) – then it is worth searching depth-wise even for very large depth solutions (very large d); otherwise, for very large d, when solutions are not dense, then time complexity for DFS will be terrible.
  + So DFS is especially useful because it is durable in terms of space – it just costs a lot of time. But if you have time to spare, then DFS is a very reliable way to eventually find a solution.
* Depth limited search (iterative deepening search): helps overcome large depth searches and redundancy issues with DFS.
  + Works especially if solutions do not have very large depths; previous iterations are insignificant in terms of waste if depth is not that large – it will end up saving time especially if solutions is not that deep, but with the state space (m) is very large.
  + It is a combination of DFS and BFS.

## Short Online Lecture 1 – (shortened, past version)

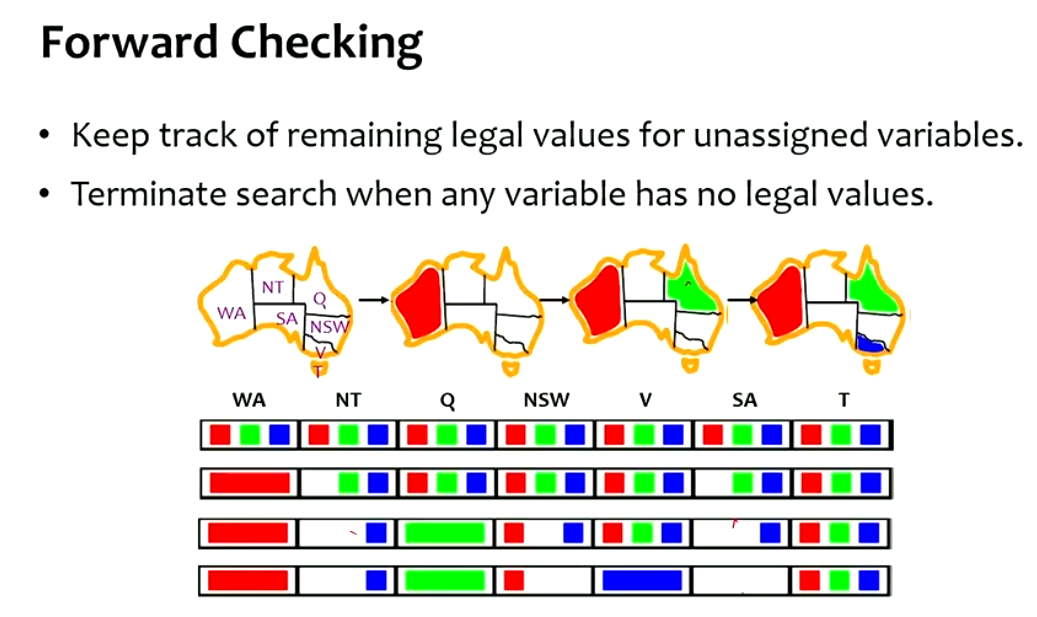
# Live Lecture 2: Uniformed/Informed Search and CSPs (ch3 & ch6) – 7/06/2022

* GBFS – Greedy Best-First Search
  + Uses some heuristic (future estimation) measurement (i.e. Euclidean distance, Manhattan distance = x component + y component to get to a destination, etc. as some lower-bound heuristic estimation parameter) (some standard data – for example, using geometry to get straight lines for best-case (minimum distance)) to determine the next best node to explore.
* A\* Search (finds optimal solution, but takes a long time to search)
  + Combine UCS and greedy search (combine the UCS value (historical = distance from root as we have calculated it from exploring the node) + Heuristic value; whichever node has lowest sum, select that node as the next node to explore).
  + For the Program 1 hash table, you can use a string for the table since there are only single digits for the 2x3 table.
  + Grey dashes (--) in the N-puzzle website that demonstrates the various types of searches means it is in the explore set (node has already been explored).
  + For windy maze example, the heuristic value is based on Manhattan (cartesian) distance (east and west = cost of 2, north = 1, and south = 3).
* If you want to find shortest path to all nodes, then just use UCS to save the extra cost of making the calculation using the heuristic value with A\*.
* A\* applications
  + Require atomic operations and values.
* Search in Constraint Satisfaction Problems (CSPs)
  + Use pruning (cut off branches where constraints are not met, to save time/not waste searching energy).
  + Think about Microsoft project – this is a CSP problem whenever you start a new project – the computer automatically finds optimal solution (or some solution) based on number of tasks, constraints, etc.
  + Think about class scheduling for the university (time constraints, number of students, number of classrooms, type of rooms, number of available professor, type of professor, etc.).
  + Continuous is more complicated (relatively speaking versus discrete).
  + We use inference to solve CSPs
  + Propagation constraints: 1 can only be in a one location; “if 1 is here, it cannot be there also” – thus that reasoning can propagate in helping prune a search tree.
    - i.e. Sudoku game, constraints will propagate to help you to reduce.
  + So we use propagated constraints via process of elimination (inference via deductive reasoning).
  + Many CSPs cannot be solved by inference alone; you need to design a search.
  + For Sudoku, you can use a DFS backtracking search.
  + We can improve backtracking efficiency by using search order and inferences and from constraints.
  + The only goal for using inference based on the constraints is to speed up CSP searching.
  + Can speed up search using decomposition.
  + Decomposition also allows for solving a problem in parallel, which can EXPONENTIALLY reduce search time.
    - Another form of decomposition: Can use tree structures (and cut set to form a set of tree structures) to greatly reduce search as well.
* Big goal for search: prune the tree to reduce search time (and reduce time and space complexity).

## Online Lecture 2 – (Review of live lecture)

* BFS, UCS, and DFS are blind (uninformed searches), and only use historical information (path-cost so far); does not use any heuristics (future/prediction value for solution or some future state that works toward the goal).
* Remember the four characteristics to consider for any search (or any) algorithm: completeness, time complexity, space complexity, optimality.
* Informed search is faster than uniformed because it can narrow down its search using information, which can save memory and time.
* GBFS (greedy best first search) uses strictly heuristics – does not consider historical data; it only cares to choose to expand the next node with the current lowest heuristic cost, regardless of historical data (cost) for any node in the current frontier.
* Heuristic is always **minimum (lower) bound** value (best-case scenario) to the goal; actual path cost will be >= heuristic value – thus we can safely say h(n) (heuristic function) is a good evaluation function.
* g(n) = path cost so far (historical data); h(n) = heuristic future prediction, minimum bound value.
* for GBFS, can get stuck in loops for tree search; but for finite space in graph search it can always find a solution (since graph-search takes out redundancies – using explore set – unlike tree search which uses frontier set and does not get rid of redundancies).
* A\* search simply combines UCS (which only uses historical information) and GBFS (which only uses heuristic information); f(n) = g(n) + h(n).
  + Diagram

    Description automatically generated
* **Important**: For A\* search, even if goal is found, still need to expand any nodes with a lower f(n) value in case they can also reach the goal with a final lower f(n) value than a previous path found for the goal.
* Worst case: A\* performs same as UCS; but on average case it is faster than UCS.
* So far, A\*, BFS, GBFS, DFS, and UCS are all use atomic states in the search algorithm (no internal structure for variables); there is only cost along path (whether historical or heuristic).
* CSPs account for variations inside of variables **and** along edges.
  + Text

    Description automatically generated
* For A\*, GBFS, and other narrowing searches like CSP searching, they are all just a form of **pruning** (trying to reduce the state space so as to more quickly and effectively search).
* Example of unary constraint (involves single variable): map color – color must be from the set of allowed colors (red, green, or blue) (from map example).
  + Node consistency
* Example of binary constraint (involves pair of variables): no two **adjacent** states on the map can have same color.
  + Arc consistency
* Example of High-order constraints (involves 3 or more variables): Sudoku game; row, column, and sub grid constraint. (note: Sudoku can be solved (for some games) purely by inference; no search necessary).
  + Path consistency
* Remember, we can use inference through constraints (constraint propagation and elimination) to reduce (prune = reduce search space) the search for a solution.
* Fail-first is for pruning (for variable selection), while fail-last (for value selection for a variable) is for **avoiding** situations that are more likely to cause a failure (since we are only trying to find one solution, we want to go as deep in the tree as possible closer toward the goal; no need to waste time on choosing values that are more constraining to other variables later in the tree, thus limiting m(more likely) how far down (and closer to the goal) you will go).
* We can also use inference in CSPs for pruning. So, there are some methods for pruning: prune by checking for legality directly, and prune by keep track of how other variables are affected when a node is chosen so as to also check for legality by tracked inferences, and finally, you can prune by avoiding repeating a failure by maintaining/tracking conflict sets so that you can back jump (back track to where the conflict set is to avoid unnecessary backtracking where the issue is essentially repeated and thus is a waste of search time/space). Also, you can use constraint graphs via decompositions and cut sets.
  + 
  + Diagram

    Description automatically generated
* Note, n = total variables, c is a set of n, and d= domain = the set of values you can select for any of the n variables. The goal of decomposition is to make a tree structure, which can search for solutions in CSPs significantly faster.
  + Graphical user interface, text, application, email

    Description automatically generated
  + Diagram

    Description automatically generated
* For cutset: you instantiate (initialize/fix some set of variables (nodes) in the graph so that the remaining graph to solve is just a tree – thus we can solve much faster when we have a tree structure). Below, notice cut set runtime is very fast for small c, and total run time compared to a non-cut tree would just be the sum of the runtime for all cut sets; thus the cut set with the largest c will determine the overall runtime for the search using cut sets. For example, if you have 2 cut sets, c1 = 5, and c2 = 10; then:
  + O[(n-5)d^(5+2) + (n-10)d^(10+2)],
  + O[(n-5)d^(5+2)] < O[(n-10)d^(10+2)],
  + thus worst-case run time is determined by O[(n-10)d^(10+2)].
  + Diagram, schematic

    Description automatically generated
  + Diagram

    Description automatically generated with low confidence
* Goal is simple tree structure (with no cycles = graphs), so that we can reduce complexity for DFS search with CSPs.

## Short Online Lecture 2 – (shortened, past version)

# Live Lecture 3: Chapter 5 (search in games) and Review and Hints for P1 – 7/11/2022

* What should we use for our hash function?
* g function for the next expanded node = parent cost + cost of new action.
* For h function, you can use the value from parent as well, to calculate the new heuristic value for a new node that resulted from expanding a parent.
* Think about zero-sum games.
* Note: you can reduce the search space by using symmetry when possible!
* Optimal strategy: always assume worst-case scenario == opponent always chooses best move.
  + Minimax strategy: minimize the max gain for all moves of your opponent.
* To calculate gain (utility) we need to first go to terminal node and trace back up the tree (via recursion) to calculate.
  + Data structure is DFS for minimax algorithm.
  + Use LIFO (stack)
  + Use DFS so we can return gain values earlier (since you need terminal nodes to evaluate gain value).
  + Maintaining game trees is very expensive; so to evaluate gain value for a node, you can use DFS to find terminal, use recursion to get value, then take that out of memory, then evaluate next path to a terminal node, calculate value, etc…add up all paths from a parent node to get the gain value for that parent while not needing to keep each path from the parent to a terminal node in memory.
* For minimax (esp. for games with very large (even infinite) state space, even if discrete), we use frontier set, since explore set would need to store so many nodes (exponential memory (space) usage).
* We can improve minimax by: **pruning** (depth-wise or branch-wise).
* For depth-wise pruning, use Eval = evaluation (approximation gain value)
  + Risky because if you were to go all the way to the terminal node, the actual gain value could be much different.
  + Think about chess, how we give certain pieces a given value (i.e., queen = +9).
  + But not all depths are equal; some depths you want to study further, while other depths you can sometimes more confidently prune.
* For Lookup: store a tables in memory for common moves (usually from start or end of game when there are less available moves or most common moves), so you can evaluate without calculation the next best move; then when you are in rarer or uncommon positions, start to do calculations for the search since it will not be in the table lookup in memory.
* Monte Carlo Tree Search: uses sampling (statistical approach) from more and more games played to evaluate the value of a state/position.
  + We need to do this for deterministic games when the state space is just too large.
* Game Tree with Chance Node
  + Circles are chance nodes
  + For stochastic, there is often some deterministic features involved. So we can use a mixture of search types: use expected value + minimax.

# Live Lecture 4: Chapter 5 (search in games) and Review and Hints for P1 – 7/11/2022

* Need inference engine for knowledge-base agent in order to not have to store everything in memory. For example, you do not need to repeat certain parts of a sentence structure, just change the verb or noun, for example.
* Logical inference = deduction.
* We can build complex sentences with just a few basic atomic sentences from propositional logic elements (and, or, not, if then, etc.).
* We have to define semantics for connectives in propositional logic.
* For P 🡪 Q, it means “we only consider Q’s value if P is true”, then if P is true, the output then depends on value of Q; if P is false, then we say statement is always true no matter the value of Q since then it does not depend on Q 🡪 so we assume statement to be true by default.
  + Implication and biconditional are sentences derived from their logical equivalences; this explains why when P is false for P 🡪 Q, then we assume statement as true (and not dependent on Q’s value).
* For P🡨 🡪 Q, it means both P and Q must have the same value to get a T output of that sentence.
* We can use logical equivalence to help with proofing in order to not have to build a truth table (in memory) to determine if some sentence will always have some value, or for some values for variables it can more quickly determine all possible outputs (basically using inference).
* Contraposition = proof by contradiction.
* Most exhaustive way to do a proof is through proof table 🡪 but takes a lot of memory depending on complexity of the proof (for example, many variables involved).
* For Wumpus World example, the Wump does not react to anything (even if it dies, only the world/squares become non smelly), so it is not considered as another agent; so scenario is a single-agent scenario (the player).
* KB = knowledge base
* Given a KB, how can we prove inferences are true?
  + Use propositional logic.
* For the example R\_# refers to all the sentences/rules in the current knowledge base.
* O(2^n) for n symbols; this is why we cannot always rely on proof/inference through enumeration (truth table); we need to use a better inference method.
  + apply pruning through a search.
  + But we have to consider interdependencies.
* CNF = Conjunctive normal form
* For solution inference rule: it makes sense that you can cancel out complementary liters, because if some variable, let’s say P, is in two separate or statements, we know that if it is true, then all other disjunctions that it is in will also be true.
* Converting to CNF
  + Implication elimination: convert implication into disjunction (use the proper logical equivalence).
  + Bidirectional elimination, convert bidirectional implication into conjunction(use the proper logical equivalence).
* Resolution rule: combine P and negation of P from two separate disjunct statements, to cancel out P and be left with one combined disjunct statement of the other variables.
* The Knowledge Base (KB) is a set of conjunctive statements.
  + We can use resolution algorithm to solve by contradiction to tell us whether the KB is true or not.
* Resolution algorithm can solve can attempt to find an empty clause in polynomial time.
* Use Horn clause to speed up resolution algorithm
* Modus Ponens is a format for representing implication statements (the / symbol is not a division symbol like in traditional mathematics).
* Forward chaining and backward chaining are horn clause (from resolution algorithm) inference methods we can use to reveal the value of unknown variables.
  + Can be represented as a graph. And also, we can try to avoid discovering unknown variables that we do not need in order to find some particular unknown (i.e., prune).
* FOL = first order logic
* Proposition logic is deterministic; but eventually we will use probabilistic logic (Bayesian Networks).

# Live Lecture 5: Chapters 12 and 13: Bayesian Networks

* Complementary probability = total probability adds up to 1.
* When variables are **independent**, we can do probx \* proby for joint probability of two (or more) events occurring.
* P(X|Y) means probability of X given Y
* P(X|Y,Z) means probability of X given Y and Z
* Bayes Rule
  + Bayes rule can give the posterior probability.
  + Text

    Description automatically generated with low confidence
    - The hosts decisions says a lot, it removes some uncertainty since he will not choose the door with the car behind it. Thus, as Bayes Rule shows, you have better chances of getting the car by switching cars.
  + Bayes rule uses evidence to remove uncertainty and give the true whole joint probability distribution (of a given situation).
  + 50/50 is worst probability (50%). The chaos (entropy) is highest when the probability is uniform (all options have = probability of occurrence).
  + Judea Pearl extended Bayes Rule into a network so that we can represent Bayes rule in a more practical way especially when there are many variables involved; he made Bayes Rule tractable.
  + Intractability 🡪 refers to time or space complexity being exponential
* You can do **sampling** to help give a more real-time/recent probability of some event.
  + You can still use the historical probability or theoretical probability values.

# Live Lecture 6: Chapters 14: Bayesian Networks over Time

* HMM’s: Hidden Markovian Chains
* Robot localization: move around and use Bayes theory to predict previous and current locations with more and more certainty as the robot moves around and obtains evidences.
* Remember: worst case for predictions is uniform probability.
* For door example: S\_1 == position 1. So P(S\_1 | Z\_1) means the probability that the robot is at location 1 given the probability that the sensor accurately sensed a door.
* Error can accumulate; so in robots, we may need to do adjustments after a certain amount of time (calibration). Or, depending on how accurate of an application you need, you may especially need probability matrix using Bayes Theory to calculate posterior probabilities.
* Posterior probability becomes the prior for the next step (when using HMM’s).
  + Each step will help you refine your predictions/probabilities, as evidences accumulate.
  + Notice in the robot localization example, each step/move increases your certainty for the next step, and for all previous steps because now you can recalculate even for previous probabilities (steps) calculated; it is referred to as “smoothing” (improving previous predictions/probabilities).
* Filtering and prediction == Forward Algorithm.
* Particle filters == sampling approach.
* Gaussian distribution = bell curve.
* HMM and Particle Filters == used for localization 🡪 sampling/getting evidence
  + Track something relative to you moving 🡪 include in your approximation based on sampling in your approximation of a continuous state space.
* Kalman is for tracking something relative to an origin 🡪 track moving objects relative to yourself as you are stationary(assuming you know your own location == the origin).
  + Kalman == Gaussian model 🡪does not use sampling, but still makes an approximation of a continuous state space.
* The posterior probability is a type of conditional probability that results from updating the prior probability with information summarized by the likelihood, through an application of Bayes' theorem

# Live Lecture 7: Chapters 4 and 17: Planning and Decision-making processes

* Review of Program 2:
  + Professor Recommends we don’t use a matrix (large memory, especially for calculations); recommends that we just calculate probabilities that would be in the matrix at runtime (just like calculating heuristic at run time – and not storing the x and y of each state; but rather searching for the value and then calculating h value based on goal – it is all done at runtime: trade space cost for (run) time cost).
  + Goal is to find initial location.
    - Using filtering/bayes theorem
  + Compare our output to the output he has on the requirements document in order to tune our program and make sure it is running correctly: it has the example for how our output should be for each move. The probability at each location refers to probability that the location is the robots initial location. After each move, we become more and more certain as to where the robot is (generally speaking).
* For state space of vacuum cleaner example: we have 3 binary variables: vacuum movement, left or right square location, and clean or not clean: 2^3 = 8 possible combinations.
* Markov Decision Process (MDP): remember, it shows relationship between dependent events, but with a short term memory (predicts future only depend on current state).
  + Also, it shows that dependent events converge on some deterministic ratio, just like independent probabilistic events.
  + No real “path” or sequence to get to a goal: simply predicts best action to take based on current state in order to take the best local actions based on each state reached during the process. So, we need to know transitional probability between each state.
  + Not about path; the plan is not a path, but rather the plan is a “policy”.
* Need a utility function in order to give each choice/action a measure of value.
  + Discounted reward for the utility function: avoid infinite loops, and essentially causes max reward to converge on a value (namely, 0) so that the longer it takes to get the reward, the smaller the reward becomes.
  + For stochastic sequence, we have an Expected Utility function.
* Methodology: Utility Theory + Probability Theory = Decision theory.
* Think about game theory with stochastic search tree games. It is very similar (think about mini max theory with probability nodes, except there is no opponent; you are just always choosing the max value (max expected value of utility function). So it is like the MAX theory.
  + But the tree is even larger for decision theory: so we need to do sampling in order to find an approximate expected value.
  + There are a lot of loops.
  + The tree is actually infinite.
    - There is a lot of redundancy that can and will happen.
    - Utility function with discounted utility can help parts of tree to be pruned as they converge to 0.
* Approach to use for decision theory (especially with stochastic situations) is called Value Iteration Approach: applies the Fixed-Point Theory.
* Maximum Expected Utility (Bellman Equation)
* Max expected utility: sum of max utility of each action from each state – including factoring in the expected value utility of states that come before it.
* Value Iteration
  + ArgMax operator makes calculation non-linear: need to use any random value and then do many iterations until we converge on a value. No matter what value you choose, you will always converge on a set of expected values (see slides) (use Fixed-Point theory).
* Policy Iteration
* The point of Value Iteration and Policy iteration: MDP (Markov decision Process: find best action at each state no matter the past; no matter what sequence of states you might have been in, it gives the current best next decision): find the best policy for each state, regardless of past.
  + See slides (which show policy for each location given utility function).
* Partially Observable Markov Decision Process (POMDP)
  + HMM (Hidden Markov Model) and MDP together.
  + Very complex, but very useful.
  + Very advance study for AI applications.
* So we model many situations that are stochastic with MDP/MODP models.
  + Planning problems.
* Adding uncertainty complicates things.
* If you don’t know the probabilities: take samples; machine learning: run an update of all decision values every time you learn (collect data) using some sampling model.
* Value v Policy iteration:
  + Both will find the best policy pi\*, and the calculations are very similar and related.
  + It can be difficult to spot the difference, However:
    - Value iteration **sets/fixes/selects the utility** (==given s, a) at each location and finds utility of all actions and iterates until convergence.

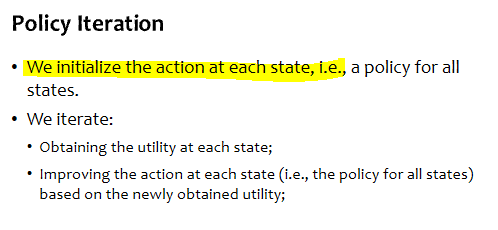
Text

Description automatically generated

Graphical user interface, text, application

Description automatically generated

* + - Policy iteration **fixes/sets/selects the action** (== given s, pi(s)) at each location and finds utility of all other actions and iterates until convergence.



Graphical user interface, text, application

Description automatically generated

# Live Lecture 8: Chapter 22: Reinforcement Learning

* Use incremental mean to save memory.
* Think about it: if you have a sample size n=3, say 1, 1 and 1, then average is 1+1+1/3 = 1. But if you said 1+1 = 2 / 2 = 1, and then added another sample (2) and did current average 2 + 1 /2 = 1.5; but if you had it as a whole sample: 1 + 1 + 2 = 4 / 3 = 1.333, which is different than 1.5; so you see you need the special equation to treat any added samples as one whole data set that you find the average for.
* Monte Carlo (MC) stores entire trajectory for estimating the utility function (memory intensive).
* Alternative to MC approach: TD (Temporal Difference) Method: Trades accuracy for saving memory.
  + Method uses bootstrapping (just like the value iteration approach for finding optimal policy).
  + Seems like it would take more time (more iterations to converge and uses more time because of all of the calculations needed to throw away old samples and save memory), but even still the overall time (according to professor) would be faster. But it seems like it would be a trade off of space for time.
  + But, the reason why it may be faster still is because it requires the transitional model: which is something we want to avoid because it is hard to use it.
  + But there is a remedy for it: another utility model: define utility for states and actions.
    - Called Q value: state-action pair.
* Q Value
  + The same as the other utility function: but instead of throwing away non-max values, we keep track of all of the other values and associate the values with the action, and then select the max value AND the action associated with that max value; utility function only cares about the max utility value and does not care about which action at a given state is associated with the max action.
  + Learn Q value using TD and MC methods – which require the transition model – but now we can use the Q value model instead.
  + Remember, for utility function and Q function: U(s) or U(s,a) means utility from a given state sequence all the way to the observing state.
    - That is why, as you get closer to positive observing state (as in grid world example and the program I wrote for it), the U(s) or U(s,a) gets more positive (or negative, say if observing state was negative).
  + Need to keep track of sample size so that you can do incremental mean: N(s,a).
* SARSA Learning Updating access frequency using Q values (iteratively) is very similar to Bellman Updates (Value Iteration)
  + Except: for Value Iteration, we KNOW the Transition Probability model and the reward. For SARSA Learning, we do not have the given Transitional Probability or reward– We only have the environment: so we must **sample** in order to approximate the transitional probability model (embedded in another model that uses sampling to approximate reward and transition probability).
  + But both can help give you the optimal policy eventually – but with some accuracy differences.
* Remember: the value of the observing states (goal/end states) will propagate to other states.
* For updating access frequency and Q values using Q Learning Illustration, the next state you still use E-greedy value for taking next action; but you always take optimal action at next state, but you update the previous state based on the actual action generated (even if that action generated was not optimal) – it essentially more quickly skews more quickly to what may be the best trajectory.
* side note: Simulated environment == transition model; and we can create an agent that implements sampling and Q-values (for example) to approximate the transition model and make the best decisions.
* So using Q values is sampling for MDP approach (Markov Decision Approach).
* One possible strategy for picking an initial state to start running iterations to find the best action at each state: start at state closest to observing state so that values propagate faster.
* Reinforcement aspect of Q-Learning: when you pick the best action: you will eventually continue to keep picking that best action - which will continue to increase and converge at some value for that best action.
* MDP (model): We have transition model.
* Q-Learning (model-free learning): we sample to approximate transition model.
  + Think about autonomous driving.
* AI has a hard time generalizing the environment with a few samples, unlike humans.
  + Perhaps because humans have so much data to work with actually: we can take data that may seem unrelated or unavailable to a computer and use it to make generalizations.
* HMM (hidden Markov Models is about localization).

# Live Lecture 9 & 10: Chapters 18 & 20: Supervised and Unsupervised Learning

* For types of model from example given in class: for example, if you generate numbers from 1-100, and the first data you get is “16”, then a “multiple pf 3” model is impossible since you cannot get “16” using that model to generate numbers from 1 to 100.
  + Then use Bayes Rule to generate the posterior.
* The prior can be very hard to get: need mathematics or lots of data: but if you have enough data, you can “ignore” the prior.
* MLE: Maximum Likelihood Estimation
  + Once you have enough data, the prior does not matter since all of the data essentially forms the most recent, and true model – the prior becomes negligible.
* A model is associated with a random variable: a random variable is associated with a distribution (distribution model).
* Bernoulli Distribution: binary.
  + It is a class of distributions.
* We aim to learn the parameter – since we don’t have any prior information: but we ignore the prior and just collect data and depend on the likelihood found from the data collected.
* Categorical (generalized Bernoulli) is for any discrete random variables (including the binary case and above: i.e. 3 possible values, 4 possible values, etc.).
  + Using Reinforcement Learning for Markov Decision Process is a form of Categorical Distribution
* Laplace Smoothing helps avoid the P=0 case, which is not likely to be 0; we should have non-0 probability because with small data sets, it does not show strong MLE accurately, and handles even a rare occurrence of something showing up (and thus it should not be P=0).
* Statistical Learning Goal: find distribution model of a random variable
* Supervised Learning: speed up learning process of AI agent; teach it by labeling data so that machine learns faster how to associate data to some characteristic(s) and features.
* For supervised learning: we aim to learn the feature function (a regression problem, because for example the label is a continuous variable (not discrete)).
* Don’t want to overkill: you often want a general model; so you don’t need to make the agent rely 100% on training set; you want to do about 80%, and let the rest of the model be built from testing (20%).
* Example of supervised learning function: spam filters
* How to build decision search tree with many attributes involved in a problem?
  + Identify the attributes that have the greatest affect (weight) and segment categories as best as possible to form the decision tree.
* Remember for uniform distribution: entropy is highest.
  + Deterministic is lowest entropy = 0.
* We use Information Theory: Information Entropy – in order to reduce chaos, the most when choosing an attribute.
* Gain information == remove the chaos == information gain.
* Information Entropy: use the distribution of a random variable to describe the chaos.
* What defines information? A model: for example, a distribution of a random variable.
* One way to prevent overfitting is using the Laplace smoothing factor, k
* You do not want to over train (also considered a form of overfitting)
  + One way to prevent is to train based on multiple decision trees, which is just a sample of the entire population.
  + Single decision tree may not be as accurate even with choosing best information gain, than training with a subset of multiple decision trees.
* Linear Regression
  + Trying to fit data to some line (function) = a model.
  + We need to figure out which model (line/function) is closest (by some standard, let’s say distance each actual data point is from the line/function)
  + Use MSE for the standard: Mean Squared Error
    - We like to do mean squared error to make magnify difference using the ^2 operator.
    - Formula in slides does not use 1/N because goal is to find minimum, and so you don’t have to have 1/N in the formula, as it will end up producing the same optimal answer – although personally I like to have it in there since it is called “Mean Square Error”
  + Spare model is less complex than a dense model – easier to calculate and make estimations – but of course it’s a tradeoff of accepting more accuracy.
  + Lambda in the overfitting is used to control the error since some values that are close to 0 we treat as 0 to make model more sparse – you can train/adjust lambda with validation data.
  + Now, we can use logistic regression for classification.
* Logistic Regression
  + Use Sigmod (S-shaped) function to predict output between 0 and 1 for all real numbers (essentially, can estimate probability).
  + Gradient descent eventually point to lowest value.
    - Step size is same as lambda from MSE equation.
  + Sparse model can prevent overfitting.
    - Overfitting using sampled data can sometimes cause error – you do not want approximation to be too rigid to the data since its just a model anyway.
* KNN model: K Nearest Neighbor
  + You want to usually pick a k that is an odd number so you do not have to deal with ties.
* Unsupervised Learning
  + We can learn faster with supervised learning.
  + Instead of using “labels” that can also be used to teach (supervise learning), we use a “clustering model” – the cluster structures essentially make up the “labels”.
  + Other than using “clusters” instead of “labels”, the two types of learning are largely the same: but now our focus is on how to define clusters since there are no given labels of the data – we have to establish the “cluster labels”.
* K-means: used for unsupervised learning to determine groups (grouping/clusters).
* Depending on where you place centroids can determine if they will all converge – they may overlap and not converge to their own cluster – but for large data sets you don’t really have to worry about this issue – every added centroid will converge to its own group.
* Expectation Maximization (EM) is similar to K-means, but uses a probabilistic model version of the K-means
  + K-means uses hard clusters.
  + EM uses soft clusters – meaning each cluster is partially another cluster.
  + Technically, all data points are a part of all clusters – even if it’s a very small probability
  + We need to learn the Gaussian distribution model curve for each cluster: need to use Maximum Likelihood Estimation (MLE)
  + Need to find the average and standard deviation^2 (variance) in order to find the gaussian distribution.
    - Then you can plug in average and variance into probability density function to get gaussian bell curve 🡪 represented by N(mue\_k, E\_k)(x) == Gaussian Source
  + So with Gaussian Mixture Model: we assume each centroid originates from a different Gaussian Distribution source (average and variance) == center of a cluster 🡪 equivalent of a centroid..
  + E-step: get the contribution by each data point for each cluster 🡪 similar to assigning each point to the nearest centroid.
  + When average and variance are updated in the M-step 🡪 it changes the shape of the curve 🡪 distribution is updated 🡪 it is similar to K-means updating the centroid location and accordingly the deterministic dividing line, which will determine what cluster all points will now belong to.
  + Pi\_k = the accumulated contribution of all points to a Gaussian Source (cluster).
    - Then if we have that, we can accordingly find the impact of each cluster on every point.
* Next topic is deep learning: GPUs are very good with matrix algebra – which is important for neural networks == deep learning, which depends on a lot of connections.

# Live Lecture 10: Deep Learning (Chapter 18) – Final New Material Learning

* Deep Learning is just another name for Neural Networks (it was just rebranded as Deep Learning due to its great success).
* Deep learning is especially concerned with large data sets – especially those that are unlabeled (focuses on clustering/grouping).
* The more data, the better the performance of a deep learning algorithm (has greater precision).
  + But there is a limitation where more data does not really improve performance due to hardware and algorithmic limitations.
    - Also because of convergence, or limitation to data (and collection method).
  + Neural networks greatly increase capacity of a technique (algorithm) and greatly continue to improve performance with more data – but obviously the more data, the more memory and processing power needed (relies heavily on GPUs, which are very good at matrix algebra).
* Remember, we use MSE (mean square error) to minimize loss of a regression curve that we are trying to fit to some data.
* Neural networks are all about “neurons”🡪 which is just a function that gives an output for the input of another function until the final function (neuron link) points to the predicted solution (final output function).
* The brain uses something called an “activation function”, which is what we dub the term for the nonlinear activation voltage threshold function a neuron uses to determine whether or not to send a weak or strong signal to the next neuron as input.
  + nonlinear function is more powerful (which is what our brain uses).
* The idea is that all neurons between layers are connected to each other, with some function that has some weight where its output is used as input for the next neuron(s).
* In deep learning, we need to learn those weights of each function, which form the model.
  + We do not need to learn the actual function themselves, those are give; we need to learn the weights; the more complex of functions we use for each neuron, however, the more powerful the network will be (and of course more neurons (functions) as well).
* ReLU (Rectified Linear Unit)
  + Activation function perform filtering (because if signal is negative, then do not pass to next neuron (function), if it is positive, then pass it to the next function).
  + We call the z as “z+” because we only do the z function if a(z) is positive.
  + B is the bias 🡪 similar to y = mx +b 🡪 there is always a bias b added to it no matter what (similar idea but nor exactly).
* All neuron functions going to the same neuron can be combined into one average function?
* For cost function, we still use MSE.
* Of course objective is to minimize C
* We still prefer sparse model (you can still over train (over feed) in a neural network.)
* You can prevent over fitting by artificially expanding the training data: use an algorithm (for example) to input rotations or location shifts of the same data and feed that data to itself (the neural network).
  + Notice: overfitting makes the model developed from the data/training data to be too rigid, so that the model might actually become less accurate.
* We can use a sampling approach for finding lowest cost: use the stochastic gradient descent (SGD approach).
  + Of course sample is only an approximation of the lowest cost gradient, but it is faster to calculate.
  + Sampling eventually can get you to the same answer as using all of the data for the lowest gradient calculation (cost); and can overall still save more time (less calculations).
* Use backpropagation to get partial derivatives to get analytical result to obtain gradients.
* We need to use back propagation to figure out the partial derivatives of all weights; which allows you to find the gradient so that we can do a gradient descent.
* SGD can be better than GD because it can avoid from getting stuck in local minimums and maxima (because goal is to get to absolute minimum).
  + Just like with a single dimension (y = x), you want to follow along the line in order to eventually find the minimum, but we can use derivatives for single dimensions; for multiple dimensions, we need to use gradient approach to follow along the partial derivative of each dimension to follow along the path that definitely connects to the absolute minimum.
* Use CNN (Convolutional Neural Network) to help reduce some of the computational power needed in a neural network (especially very large ones, with large depth, which is harder for the network to learn). Can be used for natural language processing and also still for images as well.
  + We can use filters with convolved images.
  + A filter is a smaller matrix/data range of the full data.
* Convolution is just a way to get an output number from a data set that we can provide a label to describe the data with some characteristic (based on the filter applied).
  + We use convolution to create more neuron layers.
* Goal is to detect features.
* Pooling throws away the exact positional information.
* RNN (Recurrent Neural Network) 🡪 designed for sequential data
* For labels, do not use numbers; use the not vector which has only values 0 to 1, so that the label does not have any numerical significance over other features such that it will impact calculations.
* Sparse model is better to generalize in order to learn things.
* Basic structure for neural networks: linear combinations, but non linear activation functions.
* Limitations of neural network: feedback loops (something we humans are able to evade).
* MONDAY IS REVIEW FOR HW 3 AND HW 4 SO WE CAN BE PREPARED FOR THE EXAM. EXAM WILL BE 1 DAY, 3 HRS, ON AUGUST 18.

# Notes on Normalization (Norms)

* Below, is the L2 norm (aka, Euclidean distance) of a vector (a 2d point) from the origin:
  + A picture containing diagram

    Description automatically generated
  + Notice, it is really just using Pythagorean Theorem: A^2 + B^2 = C^2, to solve the straight-line distance of a point to the origin.
  + Also, notice the formula above, it is referring to a vector: for example, a point two-dimensional point (x,y) is represented as [X1,X2] vector; so the formula means take each dimension, square it, and sum them up, then take the square root of it (aka, Pythagorean Theorem, solving for the hypotenuse (C).
* L1 Norm == Manhattan distance == just add the x and y components.
  + Diagram

    Description automatically generated
* Here is the general formula for any level of norm:
  + A picture containing text

    Description automatically generated
* Notice, for MSE (Mean Square Error), the “Error” refers to the (straight-line == EUCLIDEAN == L2) distance from the predicted value (the function) and a measured (sample) value. It is the exact same as the L2 norm function, except instead of distance from origin, it is the perpendicular distance from a line (of a function == predicted value point):
  + A picture containing diagram

    Description automatically generated
  + Text

    Description automatically generated
  + Text

    Description automatically generated
  + So another way to look at it is like this: MSE is really just subtracting one vector from another to get the resulting vector (the difference vector), then finding the L2 NORM (Euclidean Distance) from origin of that resulting vector.