Kevin Chen

3/3/19

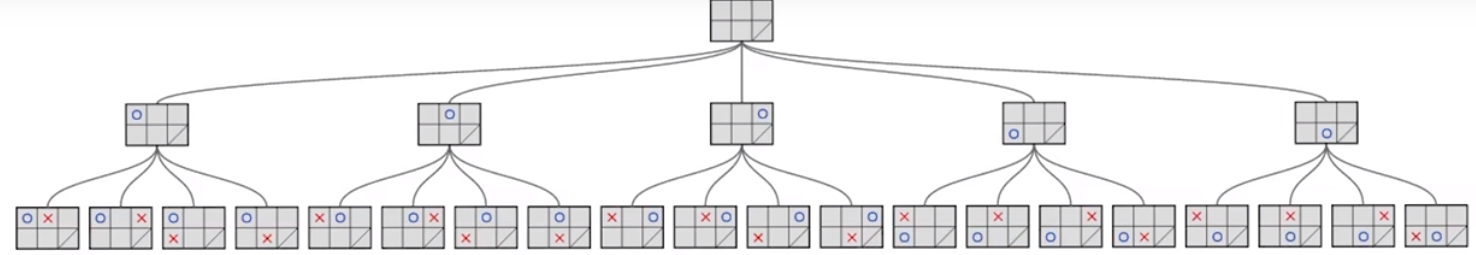
Artificial Intelligence Notes

**Game Playing**

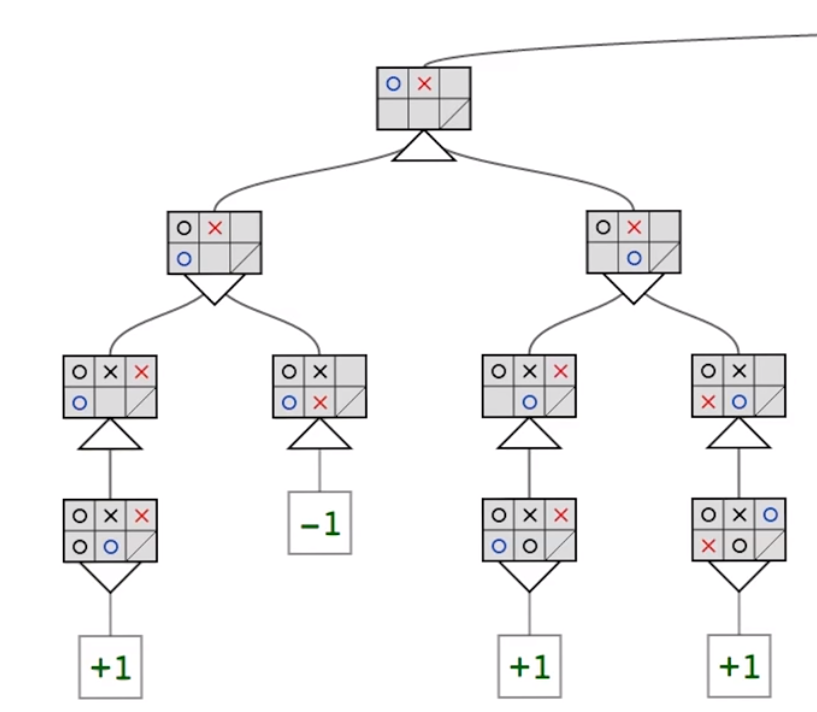
* Isolation game rules:
  + 2 player game taking turns.
  + Objective: to be the last person to move (i.e. not get “isolated”)
  + Played on a 5 x 5 grid.
  + Each player has a single game piece with which they occupy one of the squares in the 5 x 5 grid. First move is to place it in a random square (2nd player cannot put it in the same square as the first person does).
  + Each time, you can move your piece like a chess queen (horizontally, vertically, diagonally), but you cannot move it to a place currently or previously occupied by either of the pieces, and you cannot move it over a square currently or previously occupied.
  + First person to get isolated (i.e. cannot move) loses.
* The mini-max algorithm can be used determine what the best move is.

Mini-max Algorithm Introduction

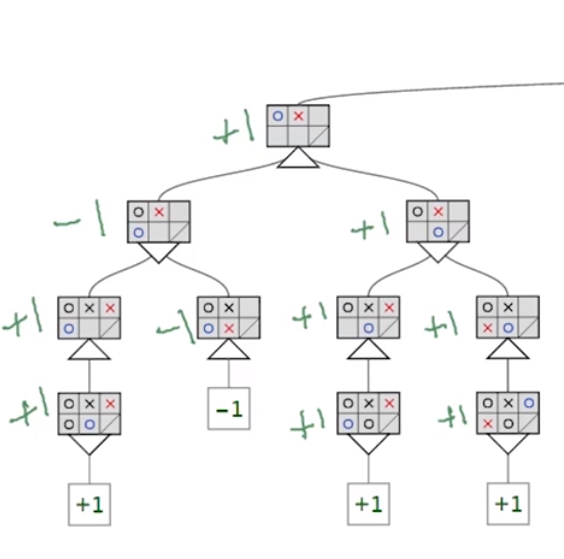
* Can build a search tree, where each node is a state, and each edge represents a possible next state of the game given a particular move. (Assume in this game of isolation, we are playing on a 2 x 3 board, and the bottom right node is blocked.)



* + Use squares to represent terminal nodes, and -1 if the person playing “O” loses and +1 if the person playing “O” wins. In mini-max we assume both players play optimally, so we use an up-triangle on a node’s edges if “O” is trying to win (i.e. on nodes where “X” just moved), and a down-triangle if “X” is trying to win.



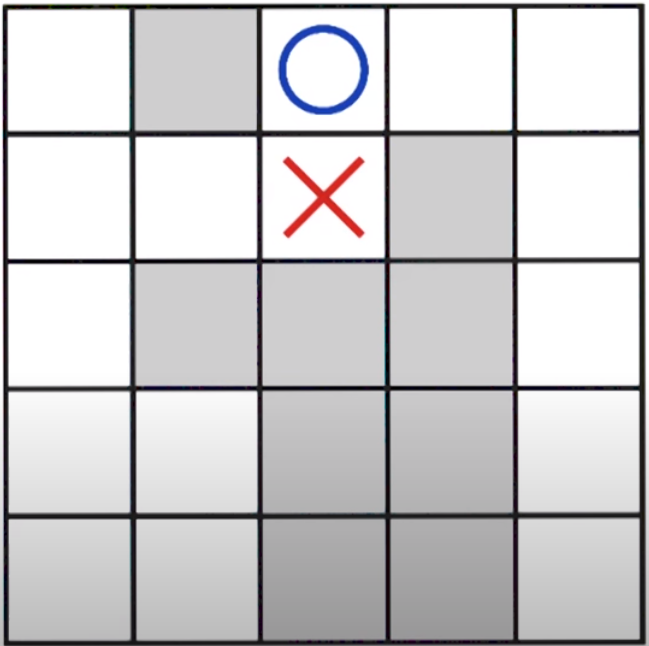
* Propagate values up the tree:
  + Start at the bottom of the tree.
  + Copy the value of the node if its parent only has one child.
  + If a parent has multiple children and it has an up arrow, then “O” will choose the branch that has +1 (i.e. guarantees a victory). If none of the children has +1, then it propagates -1.
  + If a parent has multiple children and it has a down arrow, propagate -1 if any of the children has a value -1. Otherwise, propagate +1.



* The branching factor is the maximum number of moves that can be made in a given turn, and can be used to provide an upper bound estimate of the tree search size: , where b is the branching factor and is the max depth (i.e. number of moves before game is over).
  + You can consider moves separately if you want a tighter bound. For example, the 3rd move may have a lower branching factor than the 1st move.
  + You can use average branching factor for a possibly more realistic estimate (as opposed to an upper bound).

Limiting Search and Evaluation Function

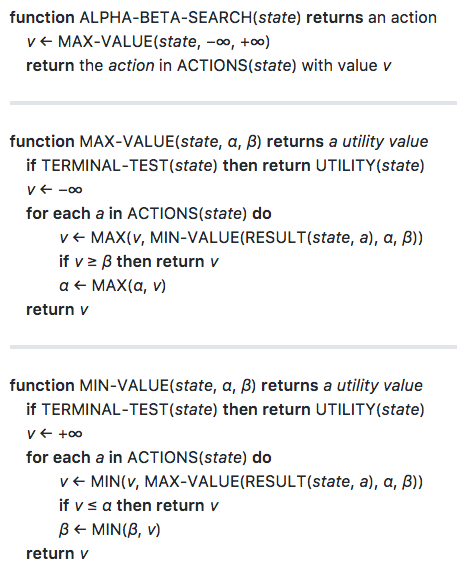
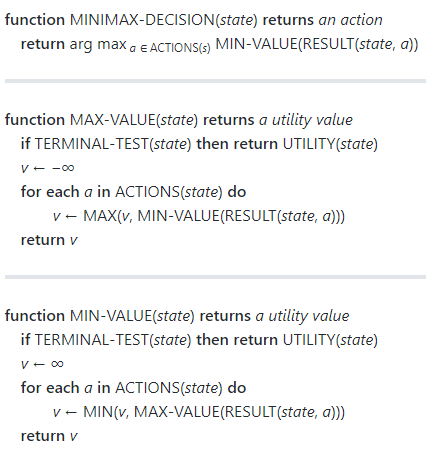
* Depth limited search: limiting the depth of the search algorithm based on time deadline (i.e. you want a move within seconds).
  + Assume that you can search nodes in 2 seconds. If the branching factor is 8, for example, then means that we should limit our search to depth 10 (or 9 to be on the safe side).
  + After searching to depth , we need an evaluation function that tells us the score at that point.
* An evaluation function is a heuristic function that returns an estimate of the score given a game state as input: the higher, the better for the computer player.
  + In the isolation game, perhaps a good heuristic is the number of moves available. We’ll refer to this evaluation function as .
  + All nodes on depth are leaf nodes that should be evaluated using the evaluation function. Then, use the “upward propagation” algorithm described above to propagate the heuristic values up, choosing either the maximum or the minimum heuristic value depending on whose turn it is.
* If possible, try to test the evaluation function using the propagate upwards algorithm by observing whether it suggests the best move to make (i.e. the branch with the highest value represents the best move for the computer on the computer’s turn and the branch with the lowest value represents the best move for the opponent on the opponent’s turn).
  + If the propagated values don’t align well with the best move for the current state, then either the evaluation function is not good OR you are not exploring deep enough to get good answers.
  + One way to check if you’re not exploring enough is to try increasing/decreasing the levels 1 by 1 and observing the propagated values on each branch of the current state. If the values change dramatically, then an important decision is probably being made, so you should explore more. Otherwise, you have achieved a state of quiescence, so you are exploring deep enough.
  + The above process of iteratively increasing the depth of search until branch values don’t change much is known as quiescent search. This may be computationally expensive to do, so you may use this sparingly (e.g. for evaluating your evaluation function or using it at the beginning of the game).
* Iterative deepening is an alternative to quiescent search: try searching 1 level deep. If you have more time, try searching 2 levels deep, then 3 levels, etc. until you run out of time.
  + While it may appear inefficient, iterative deepening doesn’t waste that much time in practice due to the exponential nature of the problem: most of the time is dominated by the last level searched.
  + Assuming a branching factor of , BFS/DFS would use expand nodes while iterative deepening would expand at most nodes, which is not that much worse (i.e. same Big-O).
  + Iterative deepening explores less if the branching factor is high (e.g. in the beginning game) and more if the branching factor is low (e.g. in the endgame).
* Horizon effect: when it is obvious to the human player (e.g. with some other heuristic/rule that can reduce the problem to something easy) what the result of the game (and the best sequence of moves for both sides) is, but is computationally difficult to solve with the general mini-max iterative deepening search.
  + Example, in the following isolation board (circle’s turn), the circle should move to the left of the “x”, thus partitioning the board. Since the left side is a little larger in this partition, the circle player should win. But this defies the heuristic since only 3 moves are available from that position.



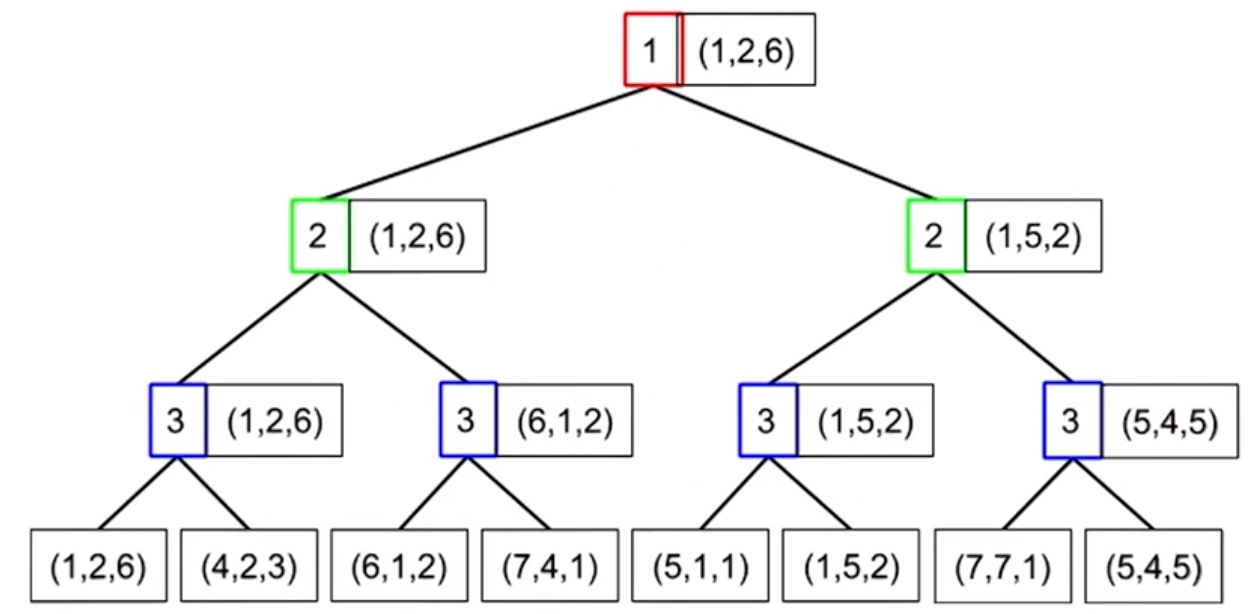
* + Checking for specific rules/heuristics (e.g. partitions in the isolation game) for the evaluation function can be computationally expensive. This creates a tradeoff between having a better evaluation function and reducing the depth of the search.
* One perhaps better evaluation function for isolation is solves the problem (for constant where is # of available moves the circle player can go and is the # of available moves the opponent or X can go).

Alpha-Beta Pruning

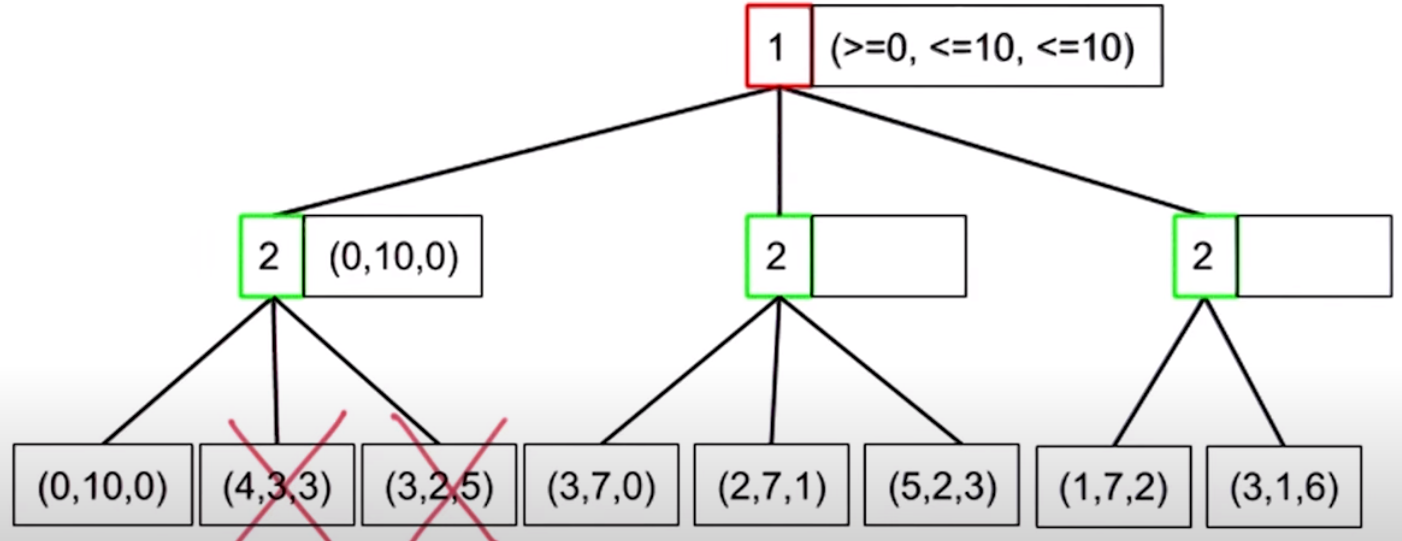
* This technique allows us to ignore whole sections of the game tree (and thus more efficient), but still get the same answer as vanilla mini-max.
* Mini-max algorithm pseudocode (left) vs. Alpha-beta pruning pseudocode (right):



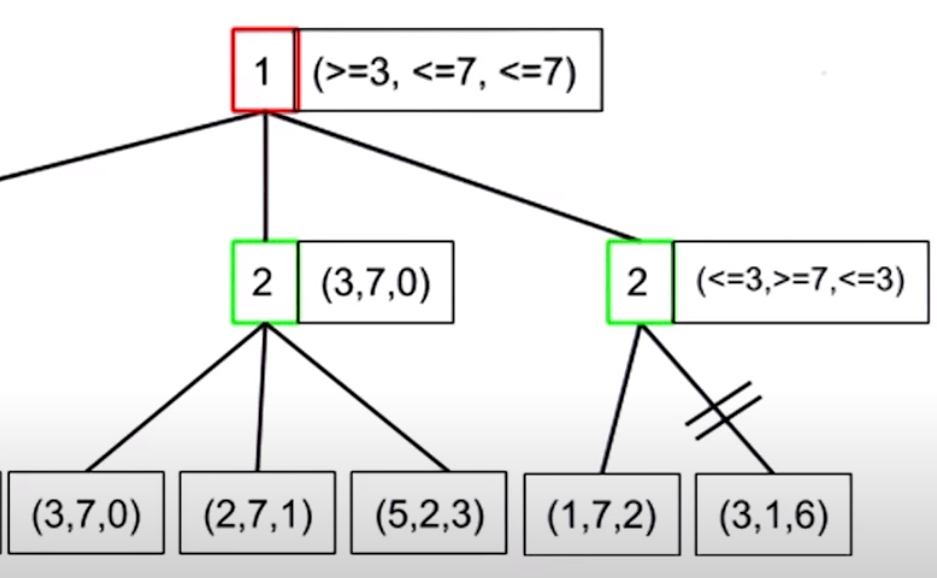
* + is the maximum lower bound of the minimax value and is the minimum upper bound of the minimax value. In other words, at every state in the game tree represents the guaranteed worst-case (i.e. lowest) score the MAX player could achieve, and represents the guaranteed worst-case (i.e. highest) score the MIN player could achieve.
  + If in any state, then the search can be cut off because this state will never be reached in the actual play. This is because your agent could do better by making a different move earlier in the game tree to avoid the pruned state.
  + Recall minimax runs in time. Alpha-beta pruning can run in best time with optimal node ordering, or on average with random node ordering.
* Artificial intelligence problems:
  + Often exponentially complex in time and/or space.
  + AI is joked to be the study of finding clever solutions/hacks to exponential problems. Also is joked to consist of all the NP-hard problems not yet solved.
  + Once a hack has been found (or computers can compute it fast), then we don’t consider it to be AI anymore.
* Strategies for solving isolation on a 5 x 5 board:
  + Use alpha-beta pruning
  + Group positions that are equivalent by rotational/reflectional symmetry.
  + Using heuristics to quickly evaluate a game once a partition formed avoids the need to search to the very end of the game.
* Isolation strategies on a 5 x 5 board:
  + Player 2 will win with best play.
  + Player 1 should take the center square on the first turn and reflect player 2’s moves if possible – if so, player 1 will win. There are 8 squares that player 2 can make that prevents reflection.
  + Create a good book of opening moves.
* 3-player games: consider 3-player isolation, where there is only 1 winner, who is the last to move.
  + We now keep track 3 separate scores, one for each player (i.e. the evaluation function now takes in the game state and the player we’re evaluating).
  + Propagate the appropriate one of the children’s score for each node depending on whose turn it is on that level. For example, if it’s player 3’s turn, select the child that has the highest score for player 3. Example:



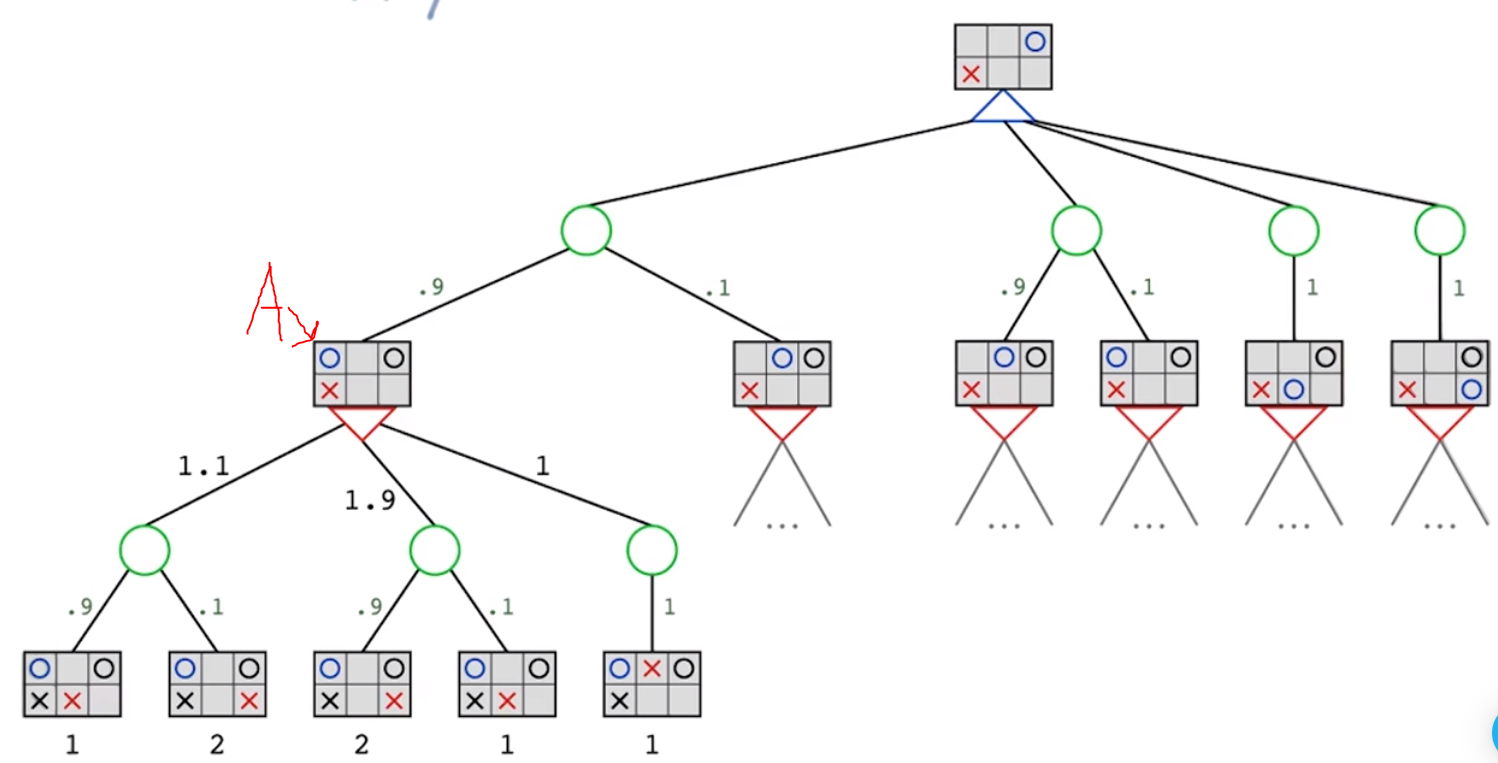
* + The above approach can be generalized for 3+ players.
* Alpha-beta pruning can work with 3+ player games as long as the sum of the values of the evaluation function for all 3+ players has an upper bound and each player’s value has a lower bound.
  + Immediate and shallow pruning is possible, but not deep pruning like before in 2-player games.
  + Immediate pruning example: if the upper bound of the sum of scores is 10, and player 2 can achieve a score of 10 with one if its move choices, then it’s unnecessary to evaluate the rest of player 2’s branches.



* + Shallow pruning example: assume again that the sum of evaluation scores can be at most 10. Player 1 is guaranteed at least a score of 3 by taking the middle branch (left of the two shown). However, by taking the right branch, player 2 can earn at least 7 points, and thereby limit player 1’s score to 3 by taking the right branch. Therefore, there is no need to evaluate further, since player 1 can earn at least the same score by taking the middle branch.



* Probabilistic games: games that depend on probability (e.g. games with dice). You can still use a game tree, and then solve for the best move via the expectimax problem. We’ll demonstrate this with “sloppy isolation” game.
* Sloppy isolation game rules: same rules as isolation, except there is an 80% chance of the player moving to desired square, 10% chance of undershooting by 1 square, and 10% chance of overshooting by 1 square.
  + If undershooting is an illegal move (i.e. lands on a square already occupied), then you have 90% chance of moving to desired square and 10% chance of overshooting by 1 square.
  + If overshooting is an illegal move (i.e. lands on square already occupied or goes outside game board), then you have 90% chance of moving to desired square and 10% chance of undershooting by 1 square.
  + If both undershooting and overshooting are illegal moves, then 100% chance of getting to desired square.
* Expectimax algorithm demonstrated using sloppy isolation:



* + Key: X is opponent and O is our agent. Red X and Blue O are current players’ positions. Black X/O are previously occupied positions. Numbers at very bottom are the output of the evaluation function on the position.
  + Green circle nodes to represent probability nodes. Edges coming out of probability nodes have probabilities.
  + For each edge coming out of a game state, calculate the expected value. For example, for the node marked A, it is X’s (opponent’s) turn. Expected value of taking left branch is 0.9 \* 1 + 0.1 \* 2 = 1.1. Expected value of middle branch is 0.9 \* 2 + 0.1 \* 1 = 1.9. Expected value of right branch is 1 \* 1 = 1. Since this is opponent’s turn, we’ll propagate the values with min value, which is 1.
* Alpha-beta pruning in expectimax:
  + Assume you have some bounds on the evaluation function.
  + You can ignore the remaining branches of a probability node if even under the best case scenario they can’t offer a good enough of a score compared to some other option you have. Here’s an example (assume all evaluation scores are between -10 and 10 inclusive):



Blue gets an average value of 6.5 by taking the left branch (of course assuming opponent picks choice with min evaluation score). Hence, we can prune remaining choices if we know they will give us less than 6.5. After calculating the left middle branch will give us 0, we realize the middle branch is upper bounded by 5 (since there is 0.5 probability left and the remaining branches can each have a value of at most10, which means the remaining branches can at most give us 0.5 \* 10 = 5). This is inferior than 6.5 from left branch, so we can prune the right middle branch.

* + As a general heuristic, evaluating edges with higher probability first (i.e. putting them to the left of each probability node) often can result in the most pruning. This is because they can often restrict the upper bound of the remaining branches most.

**Search** (Skipped – see CS 3600 notes)

**Simulated Annealing**

Iterative Improvement Problems

* Iterative improvement problems are a class of problems where just adding a little bit of intelligence and iteratively improving the solution gets you very close to an optimal solution
  + Ex: traveling salesman problem. Draw a random path across all cities. Anytime the paths cross, revise the path to uncross the situation. For large problems, you can get a solution that is within 1% of the optimal solution.
* Ex: N-queens on a n x n chess board so that they don’t attack each other is another iterative improvement problem.
  + One solution: first randomly place the queens such that at most one queen is in one column.
  + Iteratively try moving the queens (keeping them in the same column as they were originally in) by trying to minimize a heuristic cost, such as the number of attacks between two queens:

While number of conflicts > 0:

Min\_conflicts = infinity

Best\_action = null

For each action a (where an action is moving a queen up/down):

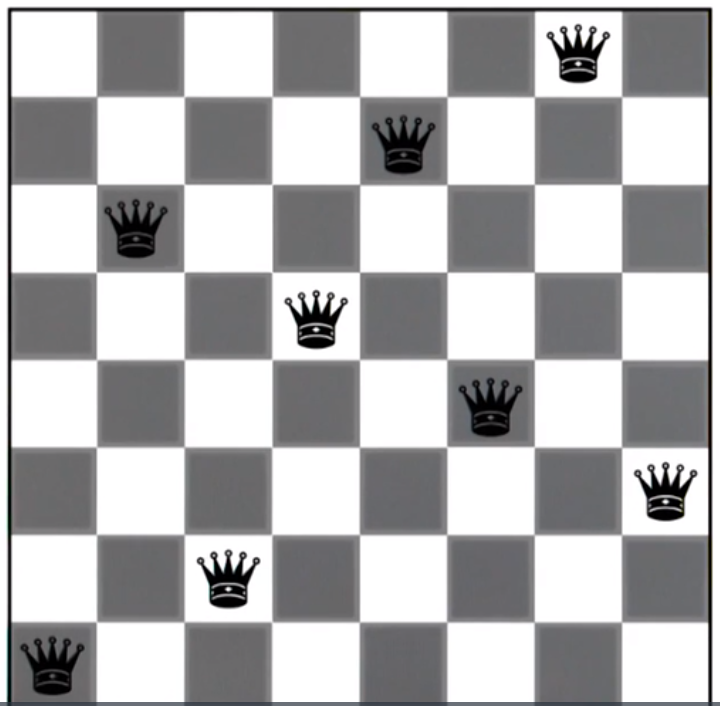
If result(a) has fewer conflicts than min\_conflicts:

Min\_conflicts = result(a)’s number of conflicts

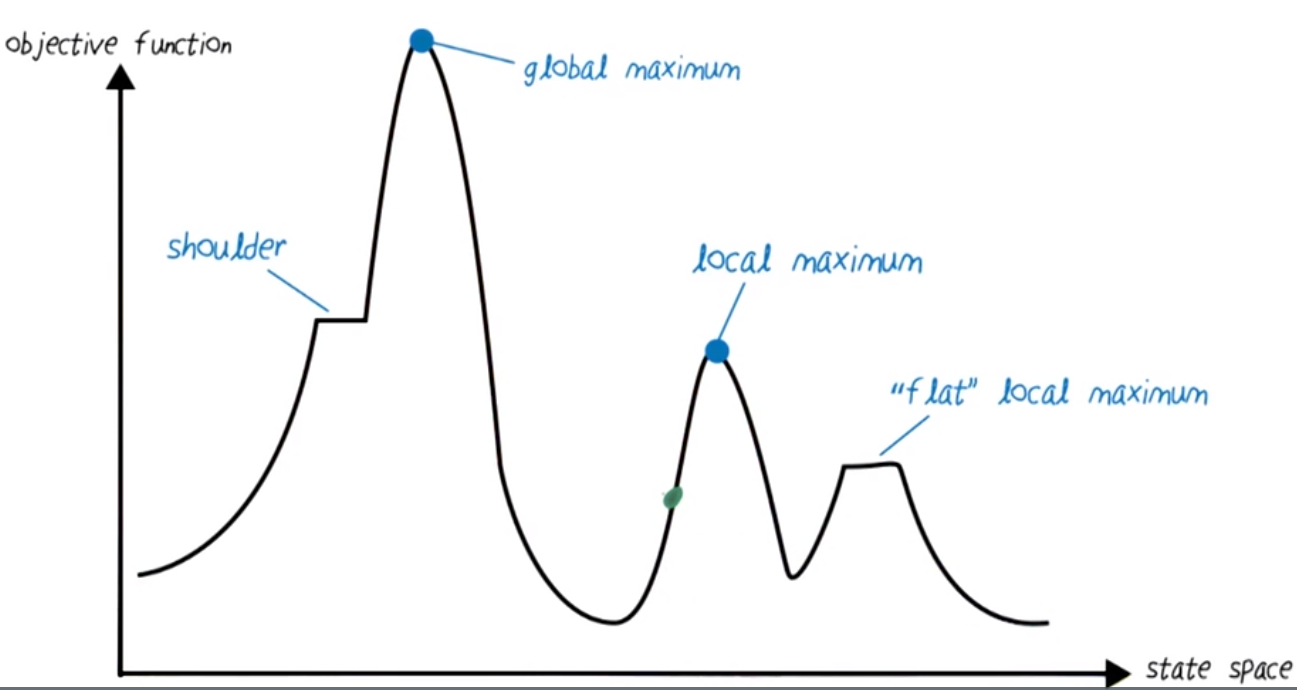
Best\_action = a

Perform a

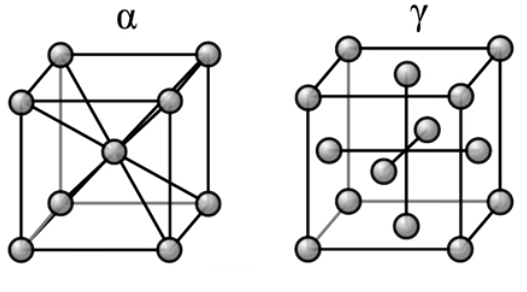
* + You may get into a situation where no move decrease the number of conflicts, which is a local optimum. Example:



* Hill climbing: given an objective function, goal of trying to find the global maximum.



* + If you start at the green dot, you might try going right, but you’ll get stuck at a local maximum.
  + Performing more iterations of hill climbing by using random restarts may help you get out of local maximums. Take the maximum of all the iterations. As number of samples approaches infinity, chance of hitting global maximum approaches infinity.
  + Taboo search: break out of the current iteration if you visited a place you already visited before.
* Step size
  + Having too small of a step size would increase the number of iterations required to get to the maximum. It also makes it longer to traverse shoulders if our implementation continues stepping in the same direction as the previous gradient for shoulders.
  + Having too large of a step size may skip entire hills (and miss certain maximums). The algorithm may also get into an infinite loop (i.e. diverge) and never terminate.
  + We can fix both problems by starting with a large step size and decreasing it gradually. Simulated annealing also accomplishes this.
* Simulated annealing comes from the physics concept of annealing.
  + Principle of energy minimization: as the mobility of molecules reduces from cooling of the material, the molecules arrange themselves in the lowest energy configuration. Ex: mud-cracks (left), Iron/Carbon lattice structures (right) which is useful for Iron sword making.

* + Annealing involves the heating and cooling to achieve certain properties of the material.
* Simulated annealing is useful for getting out of local minima.
  + We use the idea of heating/cooling to get us out of local minima. High temperature means more randomness and gradual cooling will decrease the randomness till we converge on a solution.
* The algorithm with fitness function

Pick a random *x* within the domain. Let

For each iteration :

If :

Return *current*

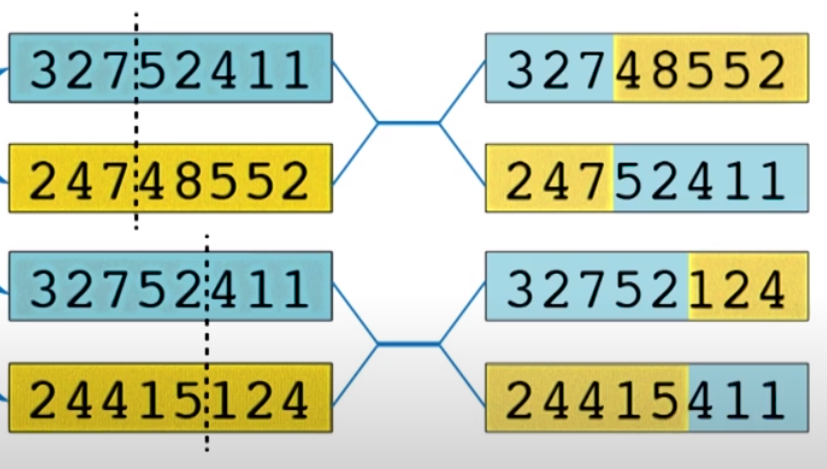
a randomly selected successor of

If :

Else:

with probability

* + Where is very large when is small and becomes 0 as approaches infinity.
  + When the temperature is large, we have a high probability of choosing to explore points with low fitness . As becomes small (e.g. 0.01), then there is very small probability that we choose to explore points with low fitness.
  + For plateaus (i.e. ), we always choose to go to and , which results in a random walk.
* Local beam search
  + Instead of keeping track of one point (i.e. particle) as we did in simulated annealing, we keep track of particles.
  + At each timeframe look at the randomly generated neighbors of the particles, and keep the best particles among the neighbors and original particles. Terminate once we reach some goal.
  + This is different than performing random restart since we compare the neighbors between original particles, but each restart in random research is independent.
  + Stochastic beam search: same as local beam such, except there is a chance we pick worse particles (initially more stochasticity, less later on) to avoid local maxima, similar to simulated annealing.
* Genetic algorithms is another approach for finding global maximum
  + Requires a string representation of the game state. For n-Queens, assuming one queen per column, concatenate the rows that each queen in on starting from the leftmost column to the rightmost column. Ex: “86427531”
  + Define a fitness function: the higher the function value, the better the state. For n-Queens, the score is a reasonable choice, as a solution with no attacking queens has the best score of 28 and a solution with all queens attacking each other has score of 0.
  + Start with random samples, where each sample is a string representing a state.
  + Evaluate the fitness of each sample, and normalize the fitness across all samples to find the chance that they can breed. (Ex if fitness of particle 1 is 10, fitness of particle 2 is 20, and fitness of particle 3 is 20, then particle 1 has 20% chance of breeding, particle 2 has 40%, and particle has 40%.)
  + Using the above chances, select parents.
  + For each pair of parents (i.e. parents 1-2, parents 3-4, etc), create two children using crossover: pick a random point between both parents. Create the first child by taking the first part of the first parent and the second part of second parent. Create second child by taking second part of first parent and first part of second parent. Example:



* + With crossovers, one child may get all the good aspects of the parents (while the other gets all the bad aspects).
  + One issue: a critical part of the solution might not exist in any part of the parents. To fix this, we add mutations in genetic algorithms: for each symbol of the string representation of each child, have a small chance that the symbol will change to a random symbol.
  + Select the best particles among parents and children, and repeat until desired goal is reached.
* Similarities between optimizers
  + In the original form, genetic algorithms don’t reduce randomness over time, but there are alternatives of the algorithm that change the number of parents/children, chance of mutation, etc.
  + Genetic algorithms is a “fancy version” of beam search.
* Recap of methods to get unstuck from local minima:
  + Random restarts, genetic algorithms, simulated annealing

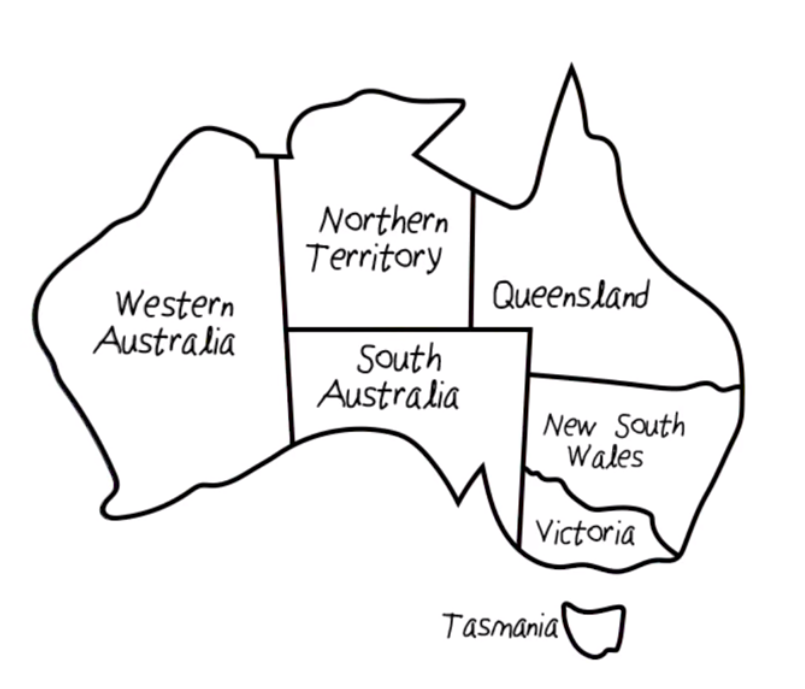
**Search**

This section is highly abbreviated. See CS 1332/3600 notes before these notes.

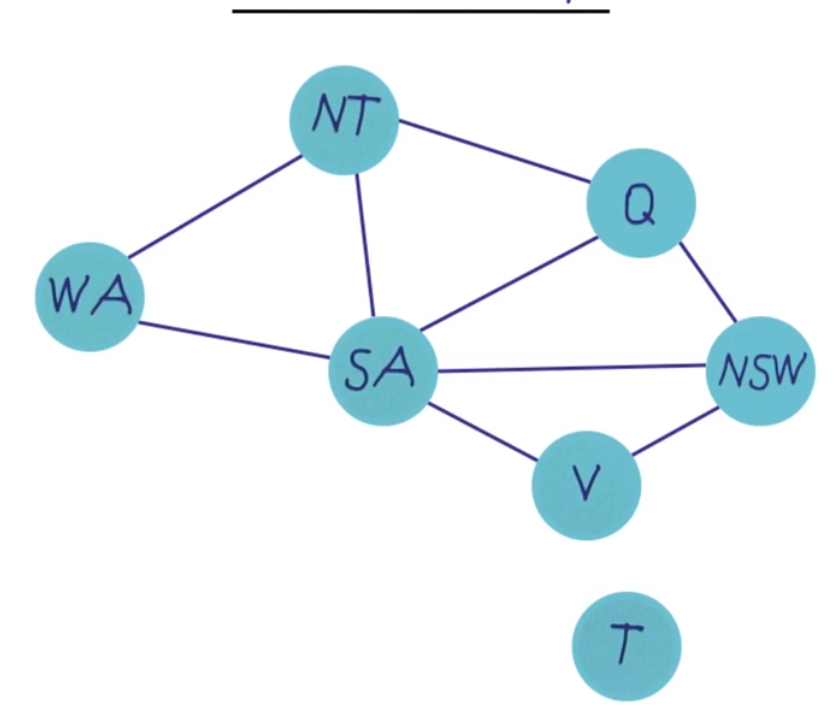
**Constraint Satisfaction**

See CS 4649 notes for more details

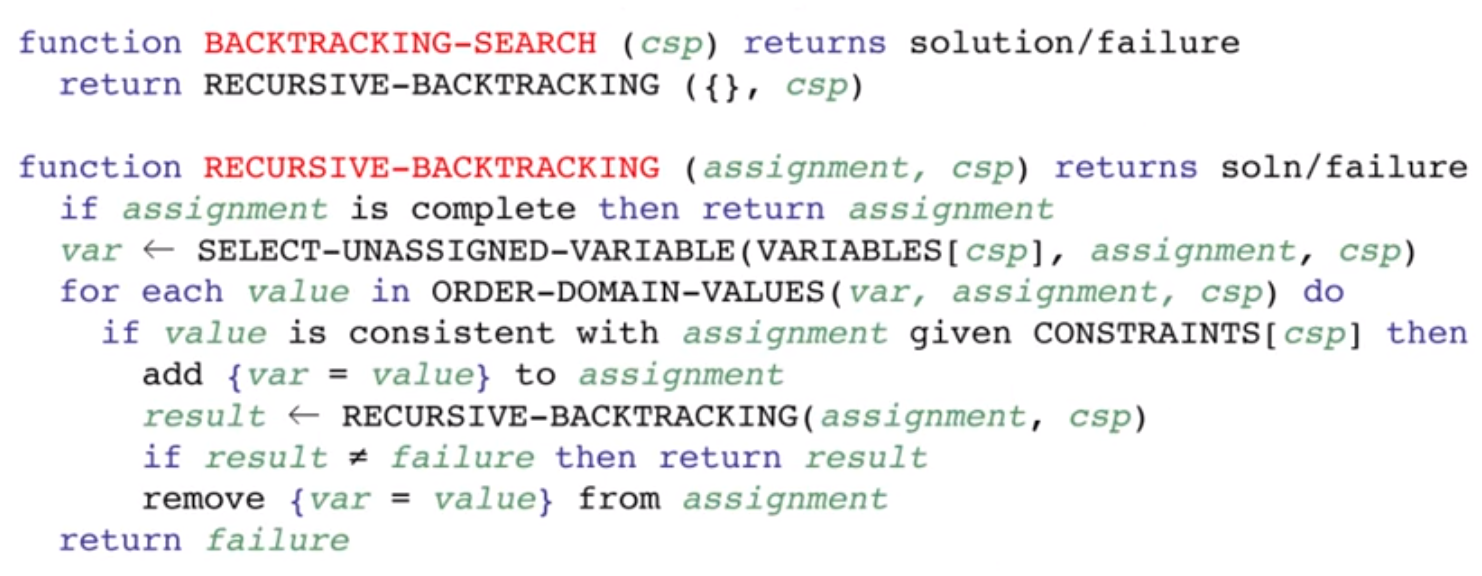
* Constraint satisfaction problems are a class of problems where, for each variable, you must assign it a value from a domain set such that they do not violate any given constraints.
* Ex problem: Map coloring – color territories on a map such that no two adjacent territories have the same color.



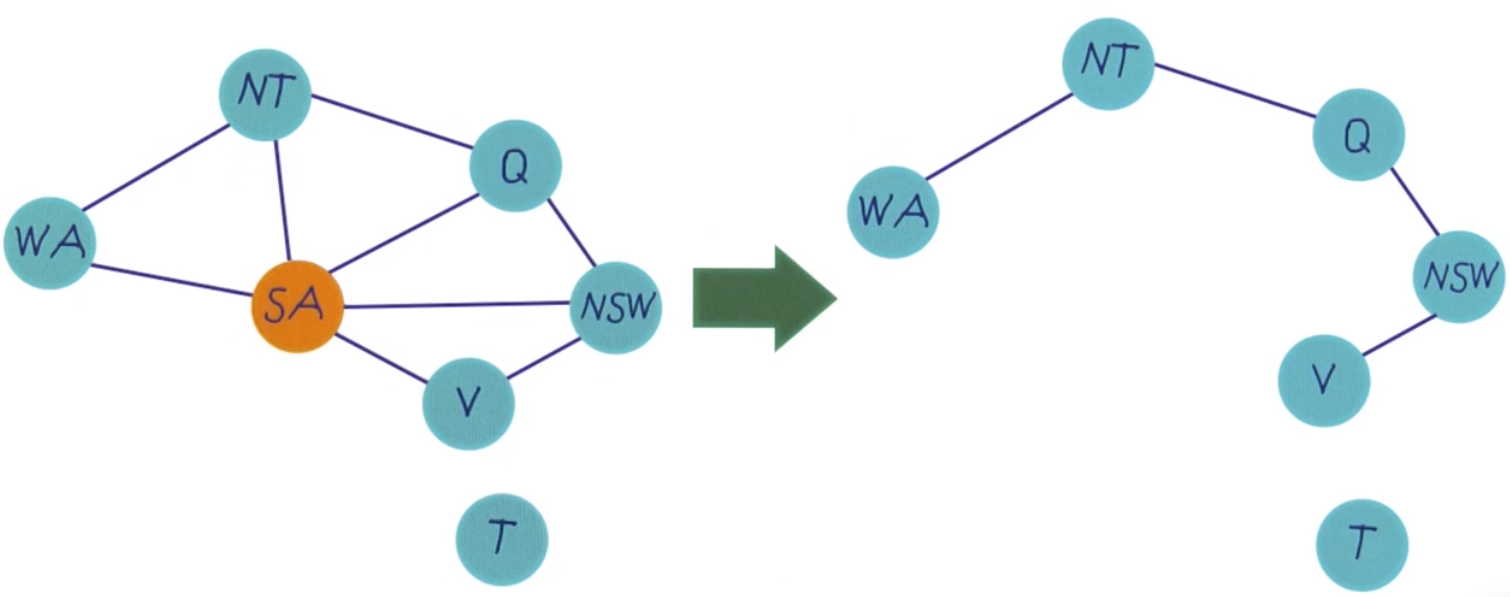
* + Variables: WA, NT, Q, NSW, V, SA, T
  + Domains:
  + Constraints: adjacent regions must have different colors (e.g. )
* Unary constraints restrict a single variable (e.g. Tasmania cannot be blue). Binary constraints can relate at most two variables, as does the map coloring problem. We can create a constraint graph:



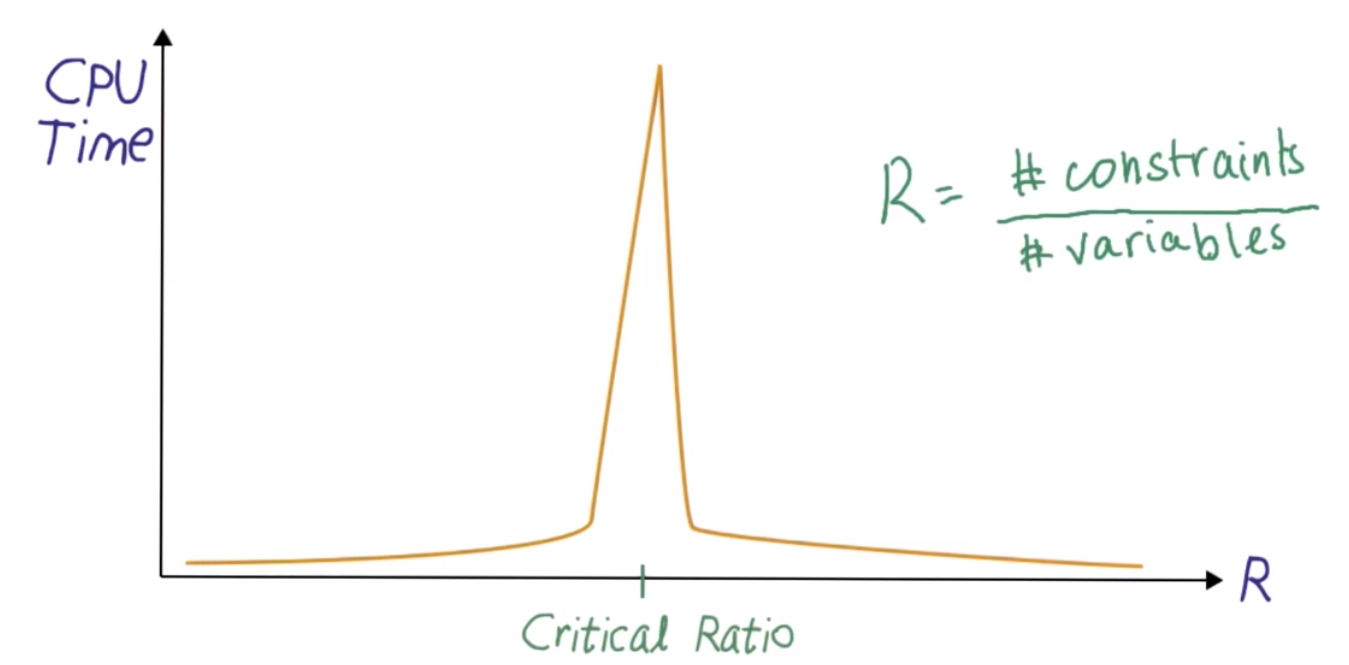
* Problems with preference constraints are called constrained optimization problems and are solved with linear programming.
* Backtracking search is one way to simplest ways to solve constraint satisfaction problems



* Heuristics to improve efficiency of backtracking search:
  + Least constraining value: choose the variable that rules out the fewest values in the remaining variables (line 3 of the algorithm above).
  + Minimum remaining values (MRV): choose the variable with the fewest legal values (line 2 of the algorithm above).
  + Degree heuristics: if there’s a tie for MRV (2+ variables have the fewest legal values), then choose the variable with the most constraints on remaining variables. This allows us to run into problems sooner rather than later if they exist.
* Backtracking with forward checking:
  + While trying a partial assignment of variables to domain values, we check if all the remaining variables have at least one possible domain value that doesn’t violate a constraint with the partial assignment of variables.
  + In the Australia map coloring example above, if we assign orange to WA, then Green to Q, then blue to V, we see that there are no possible values remaining for SA, so we backtrack now instead of exploring deeper.
* Constraint propagation is more powerful than forward checking. A simple version of constraint propagation is arc consistency.
  + A variable is arc consistent to another variable if there’s still a value available to the second variable after we assign a value to the first variable. If all variables in the graph satisfy this condition, then the network is arc consistent.
  + Ex: color WA orange and Q green. As SA only has blue left, we check its neighbors (in the order of NSW then NT). We remove blue from NSW, leaving it with just orange, and thus remove orange from NSW’s neighbor of V, leaving V with green and blue. Now that we explored SA’s neighbor of NSW, we explore SA’s neighbor of NT. We remove blue from NT, leaving NT without any colors and thus detect a conflict even earlier than Backtracking with Forward Checking.
* Structured CSPs: sometimes we can decompose a CSP into several independent problems
  + In the Australia map color example, we can solve Tasmania independently of the other regions.
  + If our constraint graph is actually a tree, we can solve it via structured CSPs in time instead of time. Choose a variable as root, and order variables from root to leaves such that every node’s parent precedes it in the ordering. Then start at the end of the ordering, making the parent arc consistent, until we finish going up the tree till we get to the beginning of the ordering. If we fail, then return that the CSP is unsolvable.
  + Sometimes we can condition a problem by assigning one or few variables to a fixed value and result in a tree. In the Australia map coloring example, we can set SA to a color, remove that color from neighbors, and get that to be a tree.



* + If conditioning on several variables, you’ll need to iterate across conditioned values, which may defeat the purpose of this optimization if this need to be done for too many variables.
* Some problems can be solved both iteratively (e.g. via optimization) and CSPs, such as n-queens problem.
  + If the number of solutions is very few or very high, then iterative algorithms are faster. There is a critical ratio of for which the problem is hard to solve.



**Probability**

* Probability gives us a method to representing uncertainty
* Bayes Network is a directed acyclic graph consisting of nodes representing random variables and directed edges representing dependencies: an edge from node A to node B means that random variable B depends on random variable A.
  + Assume binary events.
* Complementary probability:
* Independence: If (X is independent of Y): .
* Total probability:
* Bayes Rule:
  + is the likelihood.
  + is the prior
  + is the marginalized likelihood
  + is the posterior
* Bayes Rules swaps what we care about , which represents the cause A given evidence B, to , which represents the evidence B given the cause A. We have to correct for this conversion by incorporating and

**Bayes Net**

Typical Use of Bayes Network

* We have a not observable variable A that can be tested by variable B
  + Aka variable B depends on A.
  + We know .
  + We know probabilities associated with causal reasoning: and .
  + We care about the diagnostic reasoning: and .

Bayes Rule

* We know based on Bayes Rule that:

and

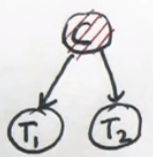
* We also know that .
  + Define and as follows:
  + We realize that:

Where

* So we can compute by calculating the values of and , and then normalize those two values so that they add up to 1.)

Independence and Conditional Independence

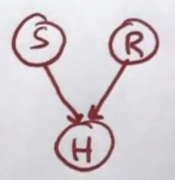
* If A and B are independent, then . This is written as
* A is conditionally independent of B given C means that . This is written as
  + If the value of C in the figure below is given, then and get “cut off”, so and become independent. Thus, and are conditionally independent given . This is written as .
  + Note that does not imply .



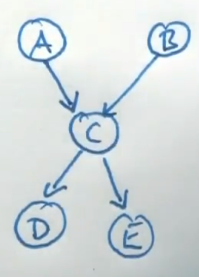
* Absolute independence does not imply conditional independence, and conditional independence does not imply absolute independence:
  + does not imply , and does not imply .

Explaining Away

* Take a look at this Bayes Network, where means it is sunny, means we received a raise, and means we’re happy. Further assume that the chance of being happy is a little higher if it’s sunny and a lot higher if I got a raise.



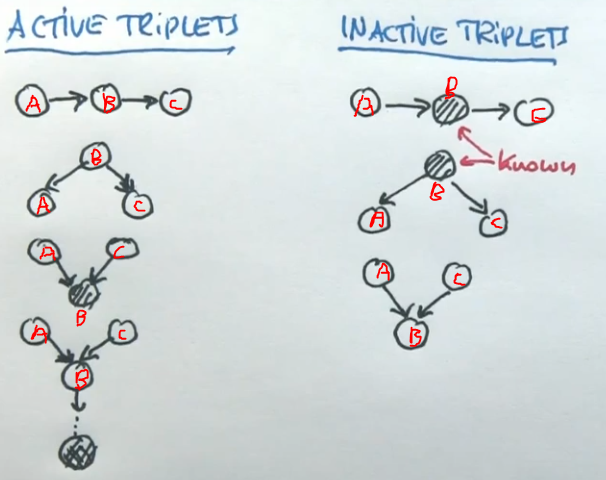
* + Variables and are independent. This is because, without any knowledge of , the value of does not have any effect on .
  + However, and are not conditionally independent given due to “explaining away”. This is because, for example, if I am happy and it is sunny outside, it is not as likely that my cause of happiness is that I got a raise (i.e. . Similarly, if I am happy but it is not sunny outside, it is more likely I got raise to explain why I’m still happy (i.e. ).
  + Concrete example: assume . We can calculate that , , and .
  + This example also serves as an example where independence does not imply conditional independence. We see , but is not conditionally independent of given .
* A Bayes network is fully defined if each random variable has the following probabilities provided:
  + If the RV has no children, then the probability of the RV is provided
  + If the RV has children, then the RV’s conditional probability given all possible combinations of its children is provided.
* For example, the following probabilities need to be defined for the following Bayes network:



* + The joint/conditional probability involving any of the above variables (e.g. can be calculated.
  + The joint distribution over any variables requires probability values, whereas Bayesian networks often require much fewer (e.g. 10 values for the above Bayesian network).

D Separation

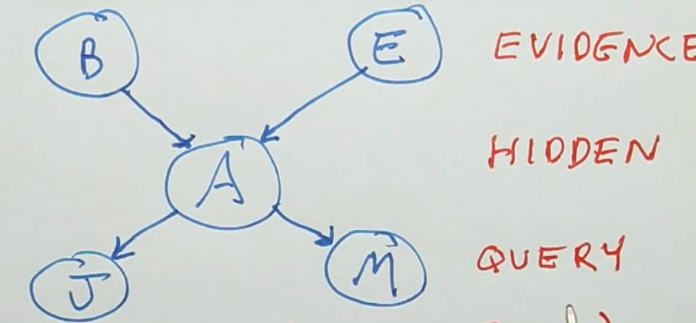
* D separation (aka reachability) refers to the conditional independence of random variables.
  + Active triplets render variables dependent and inactive triplets render variables independent.
  + Here are a list of active and inactive triplets that can be used as rules to figure out if two variables are conditionally dependent for any Bayesian Network:



* + In the active triplets list, random variables A and C are dependent. In the inactive triplets, random variables A and C are independent.

Probabilistic Inference

* In probabilistic inference, evidence random variables (RVs) are variables we know the values of, query RVs are variables we want to find the values of, and hidden RVs are the remaining variables (which we’ll have to compute internally).



* + The output is not a single number for each of the output variables but rather a joint probability distribution over the query variables, known as the posterior distribution (given the evidence).
  + We write the output in the format of: , where are query variables and are evidence variables.
* Another type of question is: what is the most likely outcome?
  + Where represents
  + We want to find the set of values with the maximum likelihood given the value of the evidence variables.
* Any variable in a Bayesian Network can be an evidence, query, or hidden variable.
* Enumeration is a naïve technique for performing inference, or finding the conditional probability of any event in a Bayes network. We go through all the possibilities and add them up.
  + Example: for the below Bayesian network, assuming all RV’s are binary, find :



* + means summing across all values of . Since is binary . So, for example,
* Enumeration is slow because it requires summing across up to terms, where is the number of variables. It gets even worse if the variables are not binary, in which there can be up to terms.
* Bringing terms out of the summation can help reduce some computations in enumeration.
  + Using the same above example:

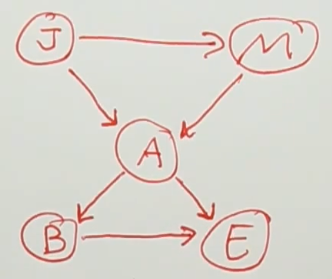


Consider just the numerator of , which is . The and terms can be pulled out as shown because doesn’t depend on or and doesn’t depend on .

* The next step to improve enumeration is to try to maximize the independence of the Bayes Network (i.e. draw the fewest edges possible). Assume you create the Bayesian Network one node at a time: each time you add a node, you add all associated dependency edges. Using this method, the order you create the nodes in the Bayes Network can affect the number of edges you need.
  + The optimal order of creating the nodes in a Bayes Network is to do it in the causal direction (i.e. ordered from causes to effects).
  + Using the same example above:

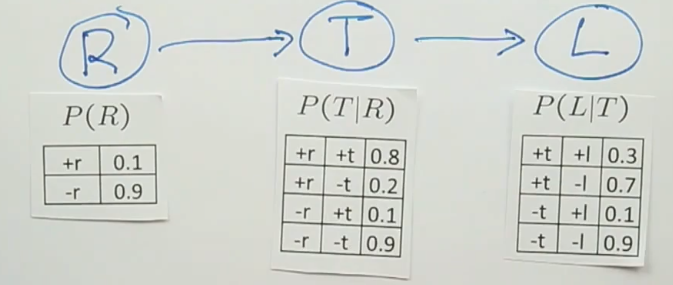


* + If we create nodes in the order of , we get the above Bayesian Network with 4 edges.
  + If we create nodes in the order of , we get the below Bayesian Network with 6 edges:

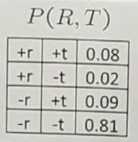


* Though still an NP-Hard computation, variable elimination is a technique for performing inference that is often faster than enumeration.
  + This is done by combining two nodes at a time into a single table, also known as “joining factors”.
  + Example: Given the following Bayesian Network, where each random variable is binary, compute . With enumeration, we would perform .

Below are the steps for variable elimination:



We first combine and , finding for and . We get the following table:



We then see that and . We replace the table containing to be the table with just .

We repeat the above set of steps for combining and .

* Approximate inference can be done via sampling.
  + For each sample, we sample all variables in a topological ordering. Each sample is a joint distribution across all variables, which can be used to approximate any non-conditional probabilities. We use the frequency of samples to infer approximate distributions.
  + Ex: if and and , then we can use sampling to approximate . We use an RNG that returns true with 20% chance to simulate . If the RNG returns true, we use an RNG that returns true with chance to simulate . If A was simulated as false, then we use an RNG that returns true with 40% chance to simulate . We count the number of samples where is true and compare it against the total number of samples: the division of these two numbers gives us an approximation for .
  + The number of occurrences of each event is more accurate as the number of samples increases. We refer to this as the sampling is consistent.
  + Sampling is an easy way to come up with an approximate value for the likelihood of the event as opposed to exact inference, where the computation is more complex.
  + Sampling is useful when not all probabilities are known in a Bayesian network, but when we can still simulate.
* For approximate sampling for a conditional probability, we can use rejection sampling: reject any samples that violate the assumptions in a conditional probability.
  + Ex: assume there are two events and and is dependent on . If we want to find , we can reject (i.e. discard) any sample of where .
  + Rejection sampling is a problem if the condition is rare, in which case we will be rejecting a lot of samples.
* Likelihood weighting is a method that fixes the above problem with rejection sampling, allowing us to keep all samples.
  + We keep the condition we are assuming fixed (e.g. keeping fixed to being true in the above example) and then sample the remaining variables.
  + There is one problem with the above approach that we need to fix: the samples are inconsistent. We fix this by weighing the samples correctly.
  + For non-conditional probabilities, we weigh all samples equally.