Introduction

October 13, 2017 3:31 PM

- Human behaviour vs rationality
- AI should do what human brain does(Turning test) or do what human brain should(rationality)
- major components of AI: - Knowledge, Reasoning, Language Understanding, Learning
- Rational Agents: An entity that perceives and acts
  - Function from percept to actions f:P→A
  - Performance measures - Goal achievement, resource consumption,...
  - Caveat: Computational limitations and environmental constraints mean we do not have perfect rationality
- Task environment: specified
  - Percepts: [Location, Dirty or Clean]
  - Actions: Right, Left, Vacuum, NoOp, Dump
  - Function: ([A,Clean],Right), ([A, Dirty], Vacuum), ([B, Dirty], Vacuum), ([B, Clean], Left)..
  - Properties:
    - Fully observable(chess) vs partially observable
    - Deterministic(guaranteed outcome，next state of the environment is completely deterSTOCHASTIC mined by the current state and the action executed by the agent) vs Stochastic
    - Episodic(treat everything as a unit, examine parts on assembly line) vs dynamic(past experience has consequences， self driving car)
    - Discrete(all board-games have finite amount of states) vs continuous(moving the arms of a robot,Taxi driving)
    - Single agent(only cares about itself) vs multi agent(self-driving car)

Search

September 13, 2017 4:07 PM

Search Problem: it is atomic

- A **state** space
- A **successor function**(actions, cost): (North, 1.0) move 1 unit north
  - RESULT(In(Arad),Go(Zerind)) = In(Zerind)
- A **start space**
- **Goal test**: how do we know we are done. The goal space can have a set of actual nodes or an abstract concept("checkmate")
- **Cost**: assume that the cost of a path can be described as the sum of the costs of the individual actions along the path. The step cost of taking action a in state s to reach state s is denoted by c(s, a, s).
- **Solution**: an optimal path with: minimum cost, or a sequence of states with successor function
- Examples

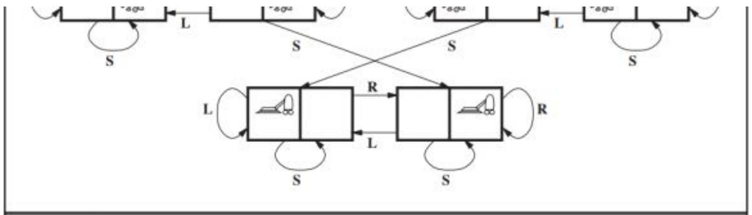


Figure 3.3 The state space for the vacuum world. Links denote actions: L = *Left*, R = *Right*, S = *Suck*.

3.2.1 Toy problems

The first example we examine is the **vacuum world** first introduced in Chapter 2. (See Figure 2.2.) This can be formulated as a problem as follows:

- **States**: The state is determined by both the agent location and the dirt locations. The agent is in one of two locations, each of which might or might not contain dirt. Thus, there are $2 \times 2^2 = 8$ possible world states. A larger environment with n locations has $n \cdot 2^n$ states.
- **Initial state**: Any state can be designated as the initial state.
- **Actions**: In this simple environment, each state has just three actions: *Left*, *Right*, and *Suck*. Larger environments might also include *Up* and *Down*.
- **Transition model**: The actions have their expected effects, except that moving *Left* in the leftmost square, moving *Right* in the rightmost square, and *Suck*ing in a clean square have no effect. The complete state space is shown in Figure 3.3.
- **Goal test**: This checks whether all the squares are clean.
- **Path cost**: Each step costs 1, so the path cost is the number of steps in the path.

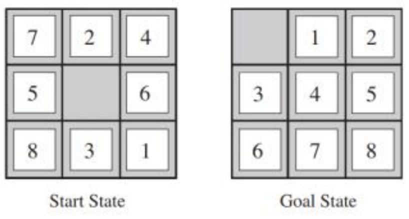
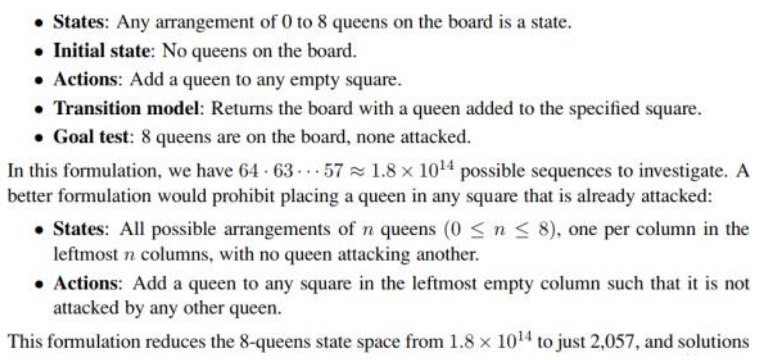


Figure 3.4 A typical instance of the 8-puzzle.

- **States**: A state description specifies the location of each of the eight tiles and the blank in one of the nine squares.
- **Initial state**: Any state can be designated as the initial state. Note that any given goal can be reached from exactly half of the possible initial states (Exercise 3.4).
- **Actions**: The simplest formulation defines the actions as movements of the blank space *Left*, *Right*, *Up*, or *Down*. Different subsets of these are possible depending on where the blank is.
- **Transition model**: Given a state and action, this returns the resulting state; for example, if we apply *Left* to the start state in Figure 3.4, the resulting state has the 5 and the blank switched.
- **Goal test**: This checks whether the state matches the goal configuration shown in Figure 3.4. (Other goal configurations are possible.)
- **Path cost**: Each step costs 1, so the path cost is the number of steps in the path.

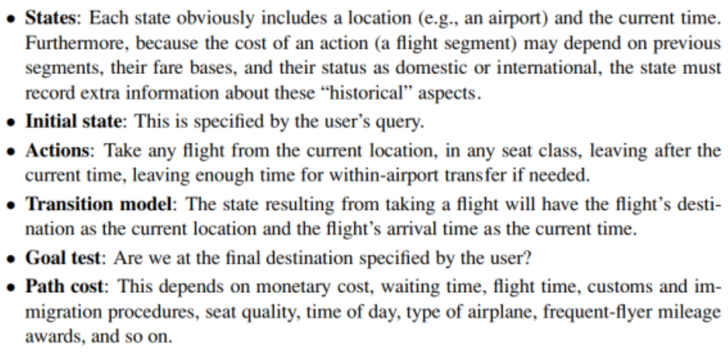


- **States**: Any arrangement of 0 to 8 queens on the board is a state.
- **Initial state**: No queens on the board.
- **Actions**: Add a queen to any empty square.
- **Transition model**: Returns the board with a queen added to the specified square.
- **Goal test**: 8 queens are on the board, none attacked.

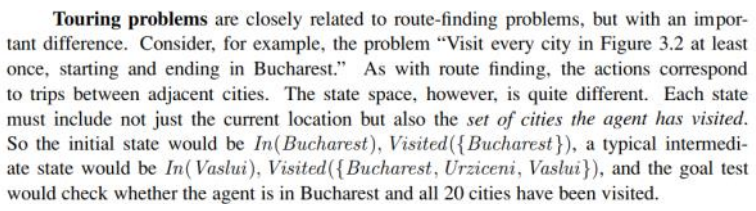
In this formulation, we have $64 \cdot 63 \cdots 57 \approx 1.8 \times 10^{14}$ possible sequences to investigate. A better formulation would prohibit placing a queen in any square that is already attacked:

- **States**: All possible arrangements of n queens ($0 \leq n \leq 8$), one per column in the leftmost n columns, with no queen attacking another.
- **Actions**: Add a queen to any square in the leftmost empty column such that it is not attacked by any other queen.

This formulation reduces the 8-queens state space from 1.8×10^{14} to just 2,057, and solutions



- **States**: Each state obviously includes a location (e.g., an airport) and the current time. Furthermore, because the cost of an action (a flight segment) may depend on previous segments, their fare bases, and their status as domestic or international, the state must record extra information about these "historical" aspects.
- **Initial state**: This is specified by the user's query.
- **Actions**: Take any flight from the current location, in any seat class, leaving after the current time, leaving enough time for within-airport transfer if needed.
- **Transition model**: The state resulting from taking a flight will have the flight's destination as the current location and the flight's arrival time as the current time.
- **Goal test**: Are we at the final destination specified by the user?
- **Path cost**: This depends on monetary cost, waiting time, flight time, customs and immigration procedures, seat quality, time of day, type of airplane, frequent-flyer mileage awards, and so on.



Touring problems are closely related to route-finding problems, but with an important difference. Consider, for example, the problem "Visit every city in Figure 3.2 at least once, starting and ending in Bucharest." As with route finding, the actions correspond to trips between adjacent cities. The state space, however, is quite different. Each state must include not just the current location but also the *set of cities the agent has visited*. So the initial state would be $In(Bucharest)$, $Visited(\{Bucharest\})$, a typical intermediate state would be $In(Vaslui)$, $Visited(\{Bucharest, Urziceni, Vaslui\})$, and the goal test would check whether the agent is in Bucharest and all 20 cities have been visited.

Screen clipping taken: 2017-10-14 12:16 AM

- **Search tree: start state is the root, children are successors, a solution is a path from root to a goal node**
  - When the states create a loop it could be infinite
  - Repeat, if no candidate nodes return failure
  - Keep track of nodes to be expanded(a queue because we need to know the next one fast), adding child nodes to the queue for each iteration
    - Initialize, add starting node to queue
    - If queue is empty return failure
    - Dequeue, if this node is goal node return success
  - **DFS** properties: A LIFO queue/stack means that the most recently generated node is chosen for expansion.
    - **Not optimal: may have loop and return suboptimal branch**
    - **Time complexity: O(b^m)**
    - **space complexity: if there are m levels and up to b nodes each level, frontiers in memory is O(bm), keeping track of the path is O(m). only one successor is generated at a time rather than all successors**
  - **BFS**:g a FIFO queue for the frontier. Thus, new nodes (which are always deeper than their parents) go to the back of the queue, and old nodes, which are shallower than the new nodes, get expanded first
    - if the shallowest goal node is at some finite depth d, breadth-first search will eventually find it after generating all shallower nodes
    - **is optimal (guaranteed to give solution)** if actions have the same cost.
    - time complexity: O(b^d) with d as the level it stopped
    - space complexity: worse than DFS because **it needs to remember all the backtrack nodes for all the nodes on that level. every node generated remains in memory. There will be O(bd—1) nodes in the explored set and O(bd) nodes in the frontier**
    - the memory requirements are a bigger problem for breadth-first search than is the execution time
  - **Iterative Deepening Search:** complete, optimal
    - **DFS with gradually increasing the limit**—first 0, then 1, then 2, and so on—until a goal is found
    - **Like depth-first search, its memory requirements are modest: O(bd) to be precise. Like breadth-first search, it is complete when the branching factor is finite and optimal when the path cost is a nondecreasing function of the depth of the node.**

UCS: g
Best first search: h
A* : g+h

    - Iterative deepening search may seem wasteful because states are generated multiple times. It turns out this is not too costly. The reason is that in a search tree with the same (or nearly the same) branching factor at each level, most of the nodes are in the bottom level, so it does not matter much that the upper levels are generated multiple times. In an iterative deepening search, the nodes on the bottom level (depth d) are generated once, those on thenext-to-bottom level are generated twice, and so on, up to the children of the root, which are generated d times. So the total number of nodes generated in the worst case is N(IDS)=(d)b + (d − 1)b2 + ··· + (1)bd , which gives a time complexity of O(bd)—asymptotically the same as breadth-first search.
    - **Iterative deepening search is analogous to breadth-first search in that it explores a complete layer of new nodes at each iteration before going on to the next layer.** It would seem worthwhile to develop an iterative analog to uniform-cost search, inheriting the latter algorithm’s optimality guarantees while avoiding its memory requirements. The idea is to use increasing path-cost limits instead of increasing depth limits.
  - **Cost-Sensitive Search(uniformed cost search)**
    - When **all step costs are equa**l, breadth-first search is optimal because it always expands the shallowest unexpanded node. Instead of expanding the shallowest node, **UNIFORM-COST SEARCH expands the node n with the lowest path cost g(n)**. This is done by storing the frontier as a priority queue ordered by g
    - **goal test is applied to a node when it is selected for expansion rather than when it is first generated**
    - Now the algorithm checks to see if this new path is better than the old one; it is, so the old one is discarded.
    - **uniform-cost search is optimal in general.** First, we observe that **whenever uniform-cost search selects a node n for expansion, the optimal path to that node has been found.** (Were this not the case, there would have to be another frontier node n on the optimal path from the start node to n, by the graph separation property of Figure 3.9; by definition, n would have lower g-cost than n and would have been selected first.) Then, **because step costs are nonnegative, paths never get shorter as nodes are added. These two facts together imply that uniform-cost search expands nodes in order of their optimal path cost.** Hence, the first goal node selected for expansion must be the optimal solution.

Bidirectional search is implemented by replacing the goal test with a check to see whether the frontiers of the two searches intersect; if they do, a solution has been found.

Informed Searching

September 18, 2017 4:03 PM

Ch3.5,3.6

- Additional **domain specific knowledge** about the problem, usually the merit of a node
- **Heuristics: a function that estimates the cost of reaching a goal from a given state**
  - Admissible: **h(n) must be less or equal to the (true) shortest path from n to any goal state**
    - Because g(n) is the actual cost to reach n along the current path, and f(n) = g(n) + h(n), we have as an immediate consequence that f(n) never overestimates the true cost of a solution along the current path through n
    - For example, tile placing problem h(n)=number of misplaced tiles->since all misplaced tiles need to move at least once
    - Effective branching factor: if h2 dominates h1, : A∗ using h2 will never expand more nodes than A∗ using h1. every node that is surely expanded by A∗ search with h2 will also surely be expanded with h1, and h1 might cause other nodes to be expanded as well
    - The state-space graph of the relaxed problem is a supergraph of the original state space because the removal of restrictions creates added edges in the graph
  - **Consistency : h(n) ≤ cost(n going to n') + h(n') .** This is a form of the general triangle inequality
    - the triangle is formed by n, n' , and the goal Gn closest to n.
    - if there were a route from n to Gn via n' that was cheaper than h(n), that would violate the property that h(n) is a lower bound on the cost to reach Gn.
    - **every consistent heuristic is also admissible.** Consistency is therefore a stricter requirement than admissibility, but one has to work quite hard to concoct heuristics that are admissible but not consistent.
    - if h(n) is consistent, then the values of f(n) along any path are nondecreasing
- **Best first searc**h: expand the most promising node(result may not be most optimal, not complete)
  - **it evaluates nodes by using just the heuristic function; that is, f(n) = h(n)**.
  - For example, for path finding problem it will expand the node closest to the goal state, as h(n) is the straight-line distance from n to destination. Straight-line distance is admissible because the shortest path between any two points is a straight line
  - Not optimal and incomplete(infinite loop where a dead-end keep being expanded and added to the queue)
- **A* search**
  - f(n)= g(n) + h(n) //**combine the cost of going to that node** and **how promising the node is**
    - estimated cost of the cheapest solution through n
  - Optimal: proof by contradiction. if A* is not optimal let G1 be optimal goal node and G2 be goal node that A* returns. Let n be some node along the path to G1. At some point, let n be a node that is in the queue at the same time as G2, and A* removes G2 instead.(correct behaviour is to remove n). Thus f(G2)<=f(n).
    - f(G2)=g(G2)+h(G2) =g(G2) > g(G1)=cost(start,n)+cost(n,G1)>=g(n)+h(n)=f(n)
  - Time complexity: all tree, O(b^m)
- **Memory-bounded heuristic search:**
  - **Iterative Deepening A* (IDA*)**
    - DFS, using f-value to choose what node to expand
  - **Simplified: drops the worst leaf node when it runs out of memory**

Constraint satisfaction(CSP)

September 20, 2017 4:00 PM

Page 208-222

- Use information inside the states
- Use the state structure to avoid taking bad branches
-

Standard search:	state is a black box, arbitrary data structure. Goal test is any function over states
CSP	**State is defined by variables with values from domains. Goal test is a set of constraints specifying allowable combinations of values for subsets of variables**

- Map colouring:
    - variables v={provinces in Australia }
    - domains D={red, blue, green}
    - Constraints: adjacent regions must have different colour
    - Solution: an assignment satisfying all constraints
    - State: partial assignment of colours
    - Initial: uncoloured graph map
    - Successor: colour one more province
    - GOAL: All regions coloured, no adjacent regions same colour
- N queens problem: place queen and remove
    - Variable: Xij
    - Domains: {0, 1} have or have no queen
    - Constraints:
    - Variables: Qi
    - Domains:{1,2,…N}
    - Constraints: implicit: for all I j (Qi and Qj) are non-threatening
- 3 sat problem:
- **Discrete variables with finite domain, if domain have size d, then there are O(d^n) complete assignments**
- **Unary constraint:** against a single variable
- **Binary constraint**: against the relationship between two variables
- **General CSP: variable, domain,constraints**
    - **State: partial assignment of variables**
    - **Initial: {} empty assignment**
    - **Successor: assign value to one variable**
    - **GOAL: All variable assigned, all constraints satisfied**
- CSP is Commutativity: does not matter which variable assigned first, can be in any order
- Backtracking search is the basic algorithm for csp
    - Select unassigned variable X, for each value {X1,…Xn} in domain of X, assign X= Xi if constraint satisfied
    - Move on to next variable
    - If none of the values X1…Xn work, backtrack
- **Most Constrained Variable(MRV)**
    - Choose the variable which has the fewest legal moves
    - **Most constraining variable**:
        - Choose variable with most constraints on remaining variables(ie many unassigned neighbours)
        - reduce the branching factor on future choices by selecting the variable that is involved in the largest number of constraints on other unassigned variables.
    - Give the **least constraining value**
        - The value that rules out the fewest values in the remaining variables
        - leave the maximum flexibility for subsequent variable assignments
- **Forward checking**
    - When you assign the value to variable, go the ones neighbouring to it and cross off the ones doesn't work from the domain
- **Arc consistency**: given domain D1 and D2, arc is consistent if for all x in D1 there is a y in D2 such that x, y are consistent .
- Structure
    - Break graph into connected components, subproblems
    - Assume n variables with domain size d: O(d^n)
    - Assume each component involves c variables (n/c components) for some constant c: O(d^c *n/c)
    - **Tree decompositions**
        - Each variable must appear in at least 1 subproblem
        - If two nodes are connected by constraint, they must appear together
        - If a variable appears in two subproblem in the tree, it must appear in every subproblem
        - Solve each subproblem independently
        - Tree width w= size of largest subproblem-1, the smaller w is, more tree-like it is
        - O(nd^(w+1))
    - **CSP can be solved in O(nd^2) if it is a tree, d is size of the domain, n is size of the nodes**
        - Topological sort
        - Starting from the end, make Xi arc consistent with parent of it in preprocessing.
          removing conflicting values from its domain
        - Front to back, assign value to Xi consistent with parent
        - Since each link from a parent to its child is arc consistent, we know that for any value we choose for the parent, there will be a valid value left to choose for the child. Avoids backtracking
    - Nontrees: The minimal cycle cut-set problem
        - Choose a subset S of variables such that the constraint graph becomes a tree when S is removed
        - Some node S that when is removed, the rest becomes a tree

local search

September 25, 2017 4:08 PM

- **When path to the goal is unimportant,** ie scheduling
- N queen problem:
  - The global minimum of this function is zero, which occurs only at perfect solutions.
  - start with queens on the board, choose variable at random and reassign value using **min-conflict heuristic**
    - H=#of pairs attacking
  - R = #constraints/#of variables. When the ratio is very big or very small, the machine can detect the solution(or the lack of) very quickly
- **Iterative improvement method**
  - Start WITH A SOLUTION then improve it
- Hill climbing(gradient descent): a loop that continually moves in the direction of increasing value. It terminates when it reaches a "peak" where no neighbor has a higher value. The algorithm does not maintain a search tree, so the data structure for the current node need only **record the state and the value of the objective function**. Hill climbing does not look ahead beyond the immediate neighbors of the current state
  - Evaluate each node that it can move to
  - **Greedy search**, pick the highest
  - **Incomplete. It gets stuck when 1) local maxima 2) plateaux**
    - If we always allow sideways moves when there are no uphill moves, an infinite loop will occur whenever the algorithm reaches a flat local maximum that is not a shoulder. One common solution is to put a limit on the number of consecutive sideways moves allowed
  - Variants:
    - **Stochastic hill climbing** chooses at random from among the uphill moves; the probability of selection can vary with the steepness of the uphill move.
    - **First-choice hill climbing** implements stochastic hill climbing by **generating successors randomly until one is generated that is better than the current state.** This is a good strategy when a state has many (e.g., thousands) of successors.
    - **Random-restart hill climbing** adopts the well-known adage, "If at first you don't succeed, try, try again." It conducts a **series of hill-climbing searches from randomly generated initial states, until a goal is found**. if there are few local maxima and plateaux, random-restart hill climbing will find a good solution very quickly
- **A hill-climbing algorithm that never makes "downhill" moves toward states with lower value (or higher cost) is guaranteed to be incomplete,** because it can get stuck on a local maximum. In contrast, a purely random walk—that is, **moving to a successor chosen uniformly at random from the set of successors—is complete but extremely inefficient**
- **Simulated Annealing**
  - The innermost loop of the simulated-annealing algorithm is quite similar to hill climbing. Instead of picking the best move, however, it picks a random move. If the move improves the situation, it is always accepted. Otherwise, the algorithm accepts the move with some probability less than 1. The probability decreases exponentially with the "badness" of the move. The probability also decreases as the "temperature" T goes down. If the schedule lowers T slowly enough, the

algorithm will find a global optimum with probability approaching 1

- Boltzmann Distribution:
  If T is large, then delta_v/T -> 0, and e^delta_v/T -> 1
  $$e^{\frac{\Delta V}{T}}$$

- **Genetic algorithm**: successor states are generated by combining two parent states rather than by modifying a single state
  - **Initialization**: begin with a set of k randomly generated states, called the **POPULATION** . Each encoded candidate solution, or **individual**, is represented as a string over a finite alphabet—a string of 0s and 1s. For example, an 8-queens state must specify the positions of 8 queens, each in a column of 8 squares, and so requires 8 × log2 8 = 24 bits. Alternatively, the state could be represented as 8 digits, each in the range from 1 to 8
  - genetic algorithms combine an **uphill tendency with random exploration and exchange of information among parallel search threads.**
  - **Evaluate**: For each x in P, compute fitness(x). Fitness function: fitness function should return higher values for better states
  - Selection
    - **Fitness proportionate selection**
      - **p(i) = i.fitness/sum(pop.fitness)**
      - **May lead to overcrowding**
    - **Tournament: select I, j with uniform random activity**
    - **Rank: sort by fitness, select based on a probability proportional to rank**
      - **Proportional to rank(small difference) avoids overcrowding in proportional to fitness(huge difference)**
    - **Boltzmann selection**
  - **Crossove**r
    - For each pair to be mated, a crossover point is chosen randomly from the positions in the string
    - The primary advantage of genetic algorithms comes from the crossover operation. Yet it can be shown mathematically that, if the positions of the genetic code are permuted initially in a random order, crossover conveys no advantage. Intuitively, **the advantage comes from the ability of crossover to combine large blocks of letters that have evolved independently to perform useful functions, thus raising the level of granularity at which the search operates**
  - **Mutate**
    - Mutation generates new features that are not present in original population
    - Typically means flipping a bit in the string
    - each location is subject to random mutation with a small independent probability

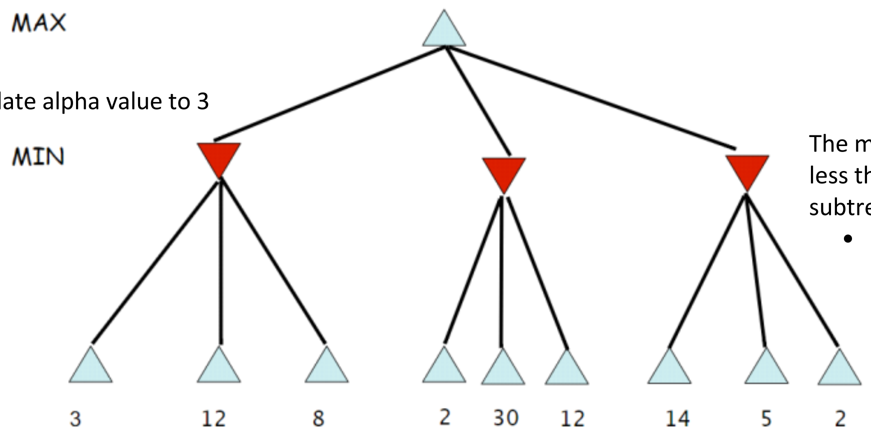
Adversarial search

September 27, 2017 4:03 PM

- Playing against another
- Types of game
  - **Perfect information: entire state in the game**
  - **Imperfect information:**
  - **stochastic(random)**
  - **deterministic(change in state is fully controlled by player**)
- Games as search problem
  - 2 player perfect information
    - State: board config
    - Player(s): whose turn
    - Result(s,a)
    - Action(s)
    - Successor: based on legal moves
    - Terminal states: game ends: win/lose
    - Utility function: +1/-1/0 for win/lose/tie
    - Solution: a strategy tells you what move to make at every stage of the game
- **Optimal strategy**: leads to outcome as good as possible, given that another player is playing optimally
- **Zero sum**: the total payoff to all players is the same for every instance of the game. every game has payoff of either 0+1, 1+0 or 1 2 + 1 2 .
- **game tree** for the game—a tree where the **nodes are game states and the edges are moves**. The **top node is the initial state**
- Minimax value: start from the bottom up

$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if TERMINAL-TEST}(s) \\ \max_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MIN} \end{cases}$$

  - It is complete
  - Similar to DFS in time and space complexity
    - Which is? Time = b^m, space=m
  - Given the opponent is playing optimally, this gives the best result
    - Multiple player, weaker ones make alliance
    - If not zero sum, (terminal state with utilityA=100, utilityB=100, cooperate)
- Alpha pruning: t prunes away branches that cannot possibly influence the final decision



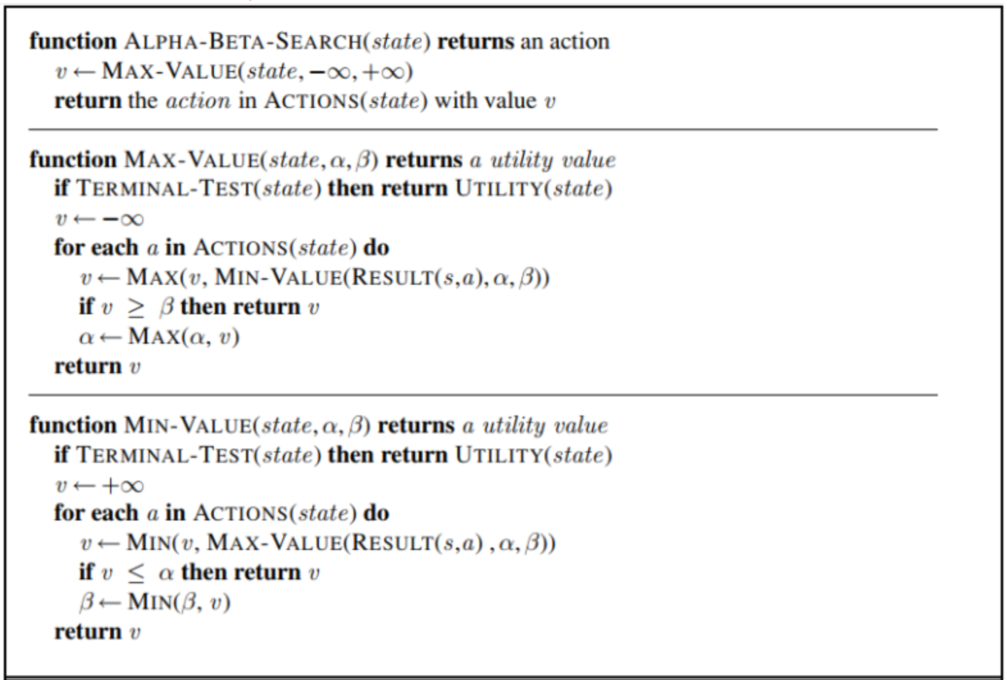
Update alpha value to 3

The moment when encounter 2, which is less than the alpha value, gives up on this subtree

- If 2 on the second subtree does not exists, the alpha will be updated to be 12

  - Assume node n, player has a choice of moving to that node. If Player has a better choice m either at the parent node of n or at any choice point further up, then n will never be reached in actual play. So once we have found out enough about n (by examining some of its descendants) to reach this conclusion, we can prune it.
  - Proof: let x,y>2.
    - MINIMAX(root) = max(min(3, 12, 8), min(2, x, y), min(14, 5, 2)) = max(3, min(2, x, y), 2) = max(3, z, 2)

        - Independent of values of x, y
    - Oder: we could not prune any successors if the worst were generated first. examine first the successors that are likely to be best.



```
function ALPHA-BETA-SEARCH(state) returns an action
  $v \leftarrow$ MAX-VALUE(state, $-\infty$, $+\infty$)
  return the action in ACTIONS(state) with value $v$

function MAX-VALUE(state, $\alpha$, $\beta$) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  $v \leftarrow -\infty$
  for each $a$ in ACTIONS(state) do
    $v \leftarrow$ MAX($v$, MIN-VALUE(RESULT(s,a), $\alpha$, $\beta$))
    if $v \geq \beta$ then return $v$
    $\alpha \leftarrow$ MAX($\alpha$, $v$)
  return $v$

function MIN-VALUE(state, $\alpha$, $\beta$) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  $v \leftarrow +\infty$
  for each $a$ in ACTIONS(state) do
    $v \leftarrow$ MIN($v$, MAX-VALUE(RESULT(s,a) , $\alpha$, $\beta$))
    if $v \leq \alpha$ then return $v$
    $\beta \leftarrow$ MIN($\beta$, $v$)
  return $v$
```

Figure 5.7 The alpha–beta search algorithm. Notice that these routines are the same as the MINIMAX functions in Figure 5.3, except for the two lines in each of MIN-VALUE and MAX-VALUE that maintain α and β (and the bookkeeping to pass these parameters along).

    - **Heuristic evaluation function and cut-off tests:** programs should cut off the search earlier and apply a heuristic evaluation function to states in the search, effectively turning nonterminal nodes into terminal leaves. (assignment2)
        - Alter minimax or alpha–beta in two ways:
            - **replace the utility function by a heuristic evaluation function EVAL,** which estimates the position's utility
                - If terminal state, function returns actual utility - If non-terminal, function returns estimate of the expected utility
            - **CUTOFF TEST replace the terminal test, decides when to apply EVAL.**
                - **The evaluation function should be applied only to positions that are Quiescent(not interesting)**
                - Singular extensions(especially promising states):strategy to mitigate the horizon effect
                    - Once discovered anywhere in the tree in the course of a search, this **singular move is remembered.** When the search **reaches the normal depth limit, the algorithm checks to see if the singular extension is a legal move; if it is, the algorithm allows the move to be considered.**
- **Stochastic**
    - Expectiminimax: minimax but at chance nodes compute the expected value
- Monte Carlo Trees for Go
    - An expanding tree
    - Selection: **pick a promising path(estimate, a path that has been visited)** through the record tree, if terminal node reach, end. Otherwise
    - Expansion: try out many steps in imagination. expand the record tree by picking a **random move**
    - **Simulate: taking bunch of random move, building an evaluation**
    - Back propagation: **update record tree along the path. Now it feels confident, update the policy**
    - Upper confidence bonds: **when it runs out of time , pick the best node**

$$v_i + C\sqrt{\frac{\ln N}{n_i}}$$

# of tries at parent
# of tries at current node
Vi is win ratio
C is exploration parameter

Start with an alpha–beta (or other) search algorithm. From a start position, have the algorithm play thousands of games against itself, using random dice rolls. In the case of backgammon, the resulting win percentage has been shown to be a good approximation of the value of the position, even if the algorithm has an imperfect heuristic and is searching only a few plies ROLLOUT. For games with dice, this type of simulation is called a rollout.

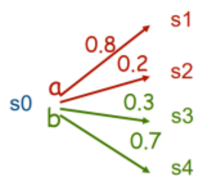
*partially observable state
Optimal play in games of imperfect information, such as Kriegspiel and bridge, requires reasoning about the current and future belief states of each player. **A simple approximation can be obtained by averaging the value of an action over each possible configuration of missing information.**For Kriegspiel, **a winning strategy, or guaranteed checkmate, is one that, for each possible percept sequence, leads to an actual checkmate for every possible board state in the current belief state, regardless of how the opponent moves.** With this definition, the opponent's belief state is irrelevant—the strategy has to work even if the opponent can see all the pieces. T

Decision making

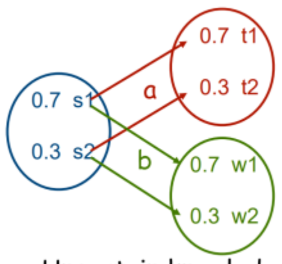
October 2, 2017 4:03 PM

- **Preference ordering is a ranking over all possible states of the world**
  - **ORDERABILITY: must be able to compare and rank**
  - **TRANSIVITIVEA>B, B>C, A>C**
  - **CONTINUE: A>B>C, getting b for sure or getting a or c**
  - **SUBSTITUTABLE: A~B-> getting A or C~ getting B or C**
  - **MONOTONIC:**If an agent prefers A to B, then the agent must prefer the lottery that has a higher probability for A
  - **DECOMPOSABLE:**: Compound lotteries can be reduced to simpler ones
- Decision problem under certainty
  - D is a set of decisions, S is a set of outcomes, preference ordering over S
  - A solution is any d in D such that **f(d)** is greater than other
- Suppose actions do not have deterministic outcome
- Utility function: We need to quantify our degree of preference
  - **Expected utility = sum of probability* preference**

    - Uncertainty in action outcomes
    - Uncertainty in state of knowledge
    - Any combination of the two



Stochastic actions



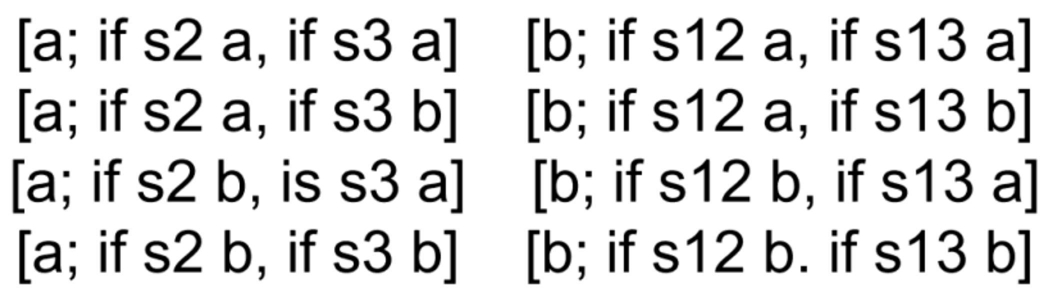
Uncertain knowledge

What caused what, even the state is not certain

- Decision problem under uncertainty
  - D is a set of decisions, S is a set of outcomes, **outcome function P that maps d to a set of distribution of s,** utility function
  - A solution is any d in D such that **EU(d)** is greater than other
  - Difficulties?
    - Outcome space can be huge(many possible outcomes)
    - Decision space is large(decisions can be sequential)
  - Sequence: states and probability.
    - We associate utilities with the final outcome
    - But it is hard to add more detail/constraints like: do a so we can reach s1 or s2. if arrive at state s1 then….
  - Plans/sequences

    [a;a], [a;b], [b;a], [b;b]

    - K = length. If A is the action set, # = A^k
  - Policies



[a; if s2 a, if s3 a] [b; if s12 a, if s13 a]
[a; if s2 a, if s3 b] [b; if s12 a, if s13 b]
[a; if s2 b, is s3 a] [b; if s12 b, if s13 a]
[a; if s2 b, if s3 b] [b; if s12 b. if s13 b]

        - # = (Outcome*Action)^k
        - Use dynamic programming. - Suppose EU(a)>EU(b) at s, only look at decision a. Back values up the tree
- Decision tree: used to evaluate policy
    - Square->choice/decision, circle->chance
    - A policy assigns a decision to each choice node in the tree
    - U(n)=average if n is a chance node
    - U(n)=max if n is a choice node
    - Two policies are **implementational indistinguishable** if they disagree only on unreachable nodes
    - Total computation for explicitly evaluating each policy would be: number of policy((OA)^k)* time to evaluate each policy(O^k)
    - we evaluate the policy as we build the tree, which is why the cost of evaluation is the same as the number of nodes in the tree: O((nm)^d. Total computational cost is thus O((OA)^k)
    - **Tree size** grows exponentially with depth, maybe use lookahead , heuristic
    - **Key Assumption: Full observability**: we must know the initial state and outcome of each action: to implement a policy we must be able to solve the uncertainty of any chance node followed by a decision node
        - Push chance node to the end of tree
        - Read more
    - Specification
        - Bayes net representation
        - Decision networks

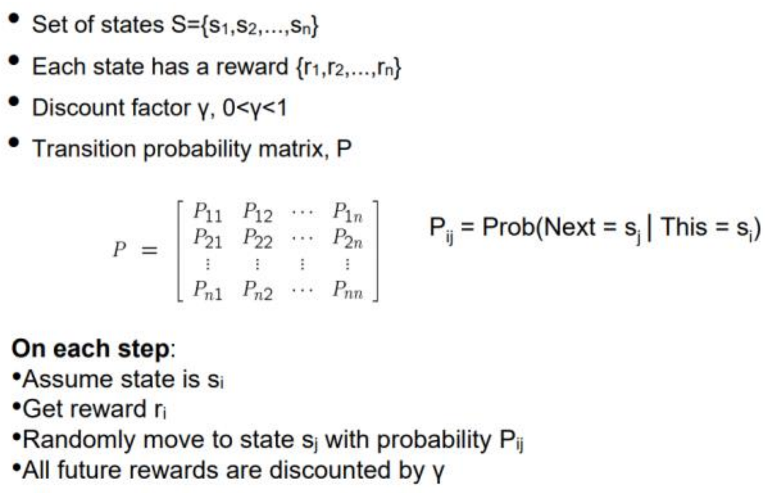
Markov Decision Processes

October 4, 2017 4:30 PM

- The probability of the next state st+1 does not depend on how the agent got to the current state st (Markov Property)
- Game is completely described by the probability distribution of the next state given the current state
  P_{ij}=Prob(Next=s_j| This=s_i)
  - If there is 11 states , the transition matrix will be a 11x11 matrix
- Discounted Rewards
  - A reward in the future is not worth quite as much as a reward now, due to inflation or obliteration
  - i.e. N years into the future is worthy (0.9)^n of payment now
  - U(A): expected discounted sum of future rewards for starting in A
  - **Value iteration: dynamic programming**
    - Only look at expected discounted sum of rewards over next k time steps
    - Initialized k=1(just the award of that node)
    - While the maximum change is still above err

$$U^k = R + \gamma P U^{k-1}$$

- Formal definition

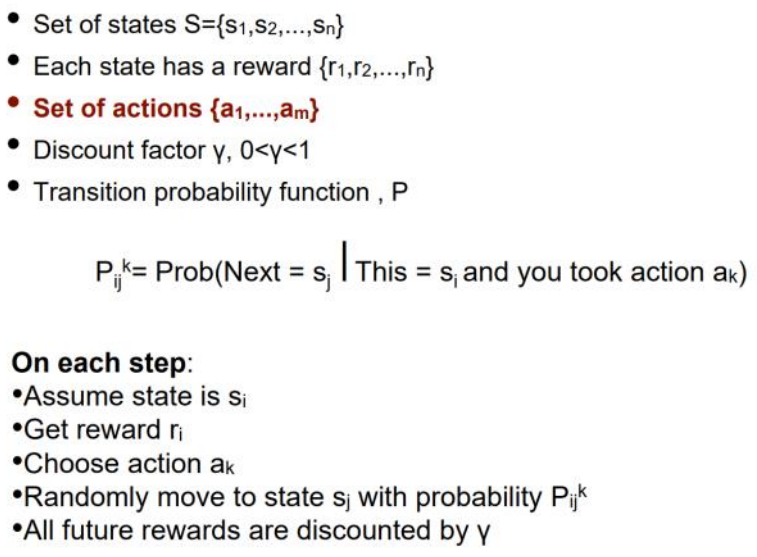


- Set of states S={s_1,s_2,…,s_n}
- Each state has a reward {r_1,r_2,…,r_n}
- Discount factor γ, 0<γ<1
- Transition probability matrix, P

$$P = \begin{bmatrix} P_{11} & P_{12} & \cdots & P_{1n} \\ P_{21} & P_{22} & \cdots & P_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ P_{n1} & P_{n2} & \cdots & P_{nn} \end{bmatrix} \qquad P_{ij} = \text{Prob}(\text{Next} = s_j \mid \text{This} = s_i)$$

On each step:
•Assume state is s_i
•Get reward r_i
•Randomly move to state s_j with probability P_{ij}
•All future rewards are discounted by γ

-



- Set of states S={s_1,s_2,…,s_n}
- Each state has a reward {r_1,r_2,…,r_n}
- **Set of actions {a_1,…,a_m}**
- Discount factor γ, 0<γ<1
- Transition probability function , P

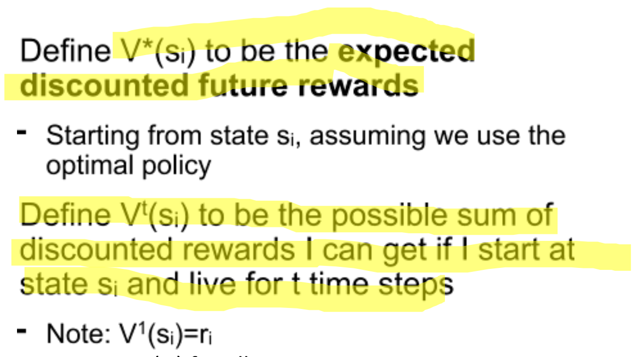
$$P_{ij}^{k} = \text{Prob}(\text{Next} = s_j \mid \text{This} = s_i \text{ and you took action } a_k)$$

On each step:
•Assume state is s_i
•Get reward r_i
•Choose action a_k
•Randomly move to state s_j with probability P_{ij}^{k}
•All future rewards are discounted by γ

- Goal: maximize expected utilities long term
- **A policy is a mapping from states to actions**
- For every MDP there exists an optimal policy
- For each possible start state, there is no better option than to follow that policy
- **Optimal value function**

[illegible]

- For each possible start state, there is no better option than to follow that policy
- **Optimal value function**



Define $V^*(s_i)$ to be the **expected discounted future rewards**

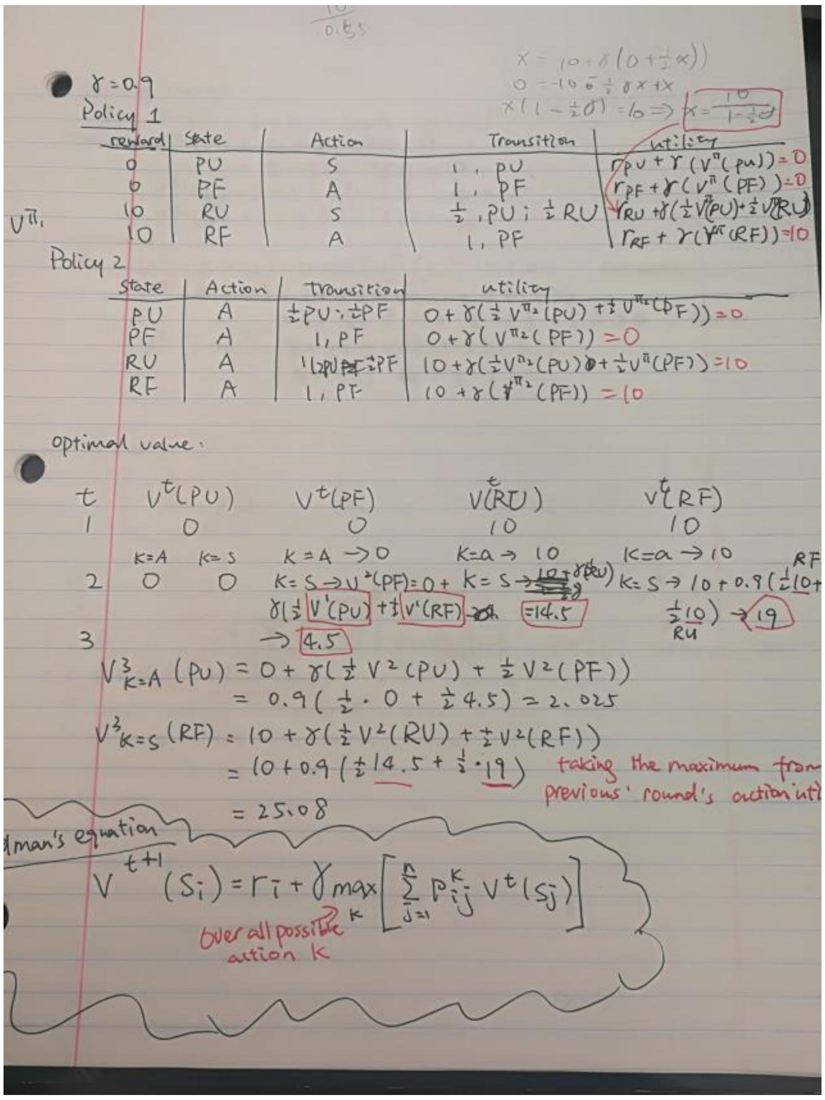
- Starting from state s_i, assuming we use the optimal policy

Define $V^t(s_i)$ to be the possible sum of discounted rewards I can get if I start at state s_i and live for t time steps

- Note: $V^1(s_i)=r_i$

  - Compute V1 (si) for all i
  - Compute V2 (si) for all i
  - Pick a decision that maximize the utility for that turn
  - Wait for convergence
  - Bellman equation
  - Policy iteration

Agent maintains a belief state, b - Probability distribution over all possible states - b(s) is the probability assigned to state s
Policy is a mapping from belief states to actions



$\frac{10}{0.55}$

$x = 10 + \gamma(0 + \frac{1}{2}x))$

$0 = -10 - \frac{1}{2}\gamma x + x$

$x(1 - \frac{1}{2}\gamma) = 10 \Rightarrow \boxed{x = \frac{10}{1-\frac{1}{2}\gamma}}$

$\gamma = 0.9$

Policy 1

reward	State	Action	Transition	utility
0	PU	S	1, PU	$r_{PU} + \gamma(V^{\pi}(PU)) = 0$
0	PF	A	1, PF	$r_{PF} + \gamma(V^{\pi}(PF)) = 0$
10	RU	S	$\frac{1}{2}$, PU ; $\frac{1}{2}$ RU	$r_{RU} + \gamma(\frac{1}{2}V^{\pi}(PU) + \frac{1}{2}V^{\pi}(RU))$
10	RF	A	1, PF	$r_{RF} + \gamma(V^{\pi}(RF)) = 10$

V^{π_1}

Policy 2

State	Action	transition	utility
PU	A	$\frac{1}{2}$PU ; $\frac{1}{2}$PF	$0 + \gamma(\frac{1}{2}V^{\pi_2}(PU) + \frac{1}{2}V^{\pi_2}(PF)) = 0$
PF	A	1, PF	$0 + \gamma(V^{\pi_2}(PF)) = 0$
RU	A	1/2PU ~~PF~~ $\frac{1}{2}$PF	$10 + \gamma(\frac{1}{2}V^{\pi_2}(PU) + \frac{1}{2}V^{\pi}(PF)) = 10$
RF	A	1, PF	$10 + \gamma(V^{\pi_2}(PF)) = 10$

optimal value:

t	$V^t(PU)$		$V^t(PF)$	$V^t(RU)$	$V^t(RF)$
1	0		0	10	10
	K=A	K=S	K=A → 0	K=a → 10	K=a → 10
2	0	0	K=S → $V^2(PF) = 0 + \gamma(\frac{1}{2}\boxed{V^1(PU)} + \frac{1}{2}\boxed{V^1(RF)}$ → $\boxed{4.5}$	K=S → ~~10 + γ(RU)~~ $\boxed{=14.5}$	K=S → $10 + 0.9(\frac{1}{2}\underline{10} + \frac{1}{2}\underline{10}) \to 19$ (RF, RU)
3					

$V^3_{K=A}(PU) = 0 + \gamma(\frac{1}{2}V^2(PU) + \frac{1}{2}V^2(PF))$
$= 0.9(\frac{1}{2}\cdot 0 + \frac{1}{2}4.5) = 2.025$

$V^3_{K=S}(RF) = 10 + \gamma(\frac{1}{2}V^2(RU) + \frac{1}{2}V^2(RF))$
$= 10 + 0.9(\frac{1}{2}\underline{14.5} + \frac{1}{2}\cdot\underline{19})$ taking the maximum from previous' round's action uti
$= 25.08$

lman's equation

$$V^{t+1}(S_i) = r_i + \gamma \max_{K} \left[\sum_{j=1}^{n} P^{K}_{ij} V^t(S_j)\right]$$

over all possible action K

Reinforcement learning

October 16, 2017 4:03 PM

- Learner is not told what actions to take(unknown rule)
- **Learner discovers value of action by trying actions out and seeing what the reward is**
- Learn to choose actions that maximize $r_0+\gamma r_1+\gamma^2 r_2+\ldots$, where 0·<1
- Example: slot machine
    - State: configuration of slots
    - Action choosing machine; stopping time
    - Reward: money
    - Problem: find a policy S->A that maximizes the reward
- Example: inverted pendulum
    - State: the position of the car and the angle between the pendulum and the car
    - Action: driving forward or backward
    - Reward: balanced pendulum
    - Problem find S->A that maximizes the reward
- Example: mobile robot
    - State: location of robot, people around the robot
    - Action: motion
    - Reward: detect the number of happy faces in the crowd
    - problem: find a policy S->A that maximizes the reward
- **Reinforcement learning characteristics**
    - **Delayed reward - Credit assignment problem**
    - **Exploration and exploitation: gaining useful information from the world, use this information or go explore more?**
        - Exploiting : taking greedy actions
        - Exploring: randomly choosing actions

        Boltzmann exploration

        $$P(a) = \frac{e^{Q(s,a)/T}}{\sum_a e^{Q(s,a)/T}}$$
    - Possibility that a state is only partially observable
    - Life-long learning
- **Model:**
    - Set of states S
    - Set of actions A(but we don't know what the consequences will be)
    - Set of reinforcement signals(rewards), may be delayed
    - Similar to MDP, our goal is to f**ind the optimal policy but we start without knowing the model** - Not given rewards and transition probabilities
- Types of Agent
    - **A utility-based agent** learns **a utility function on states** and uses it to select actions that maximize the expected outcome utility.
        - must also have a model of the environment in order to make decisions, because it must know the states to which its actions will lead
    - **A Q-learning agent l**earns an **action-utility function, or Q-function,** giving the expected utility of taking a given action in a given state.
        - can compare the expected utilities for its available choices without needing to know their outcomes, so it does not need a model of the environment
        - cannot look ahead
    - A **reflex agent l**earns **a policy that maps directly from states to actions.**

- Types of RL:
    - **Model-based vs Model-free**

        - Model-based: Learn the model of the environment -
        - Model-free: Never explicitly learn transition model state->action->newstate
    - **Passive vs Active**
        - Passive: Given a **fixed policy**, evaluate it
            - in state s, it always executes the action π(s). Its goal is simply to learn how good the policy is
            - Similar to policy evaluation, except passive learning agent does not know the transition model nor does it know the reward function R(s),
        - Active: Agent must learn what to do
- Passive:
    - **Direct utility estimation (sampling)**
        - The average of rewards for all the trials that start with that state
        - the sample average will converge to the true expectation
        - But utility of each state is not independent, so it misses the link between states(an average state may lead to high reward state thus have high utility)
    - **Adaptive Dynamic Programming**
        - Estimate probability transition matrix from the data(every time I arrive at A I move to B 2/3 of the time and C 1/3 of the time)
        - Use Bellman equation
    - **Temporal Difference learning**
        - t TD does not need a transition model to perform its updates. The environment supplies the connection between neighboring states in the form of observed transitions

Key Idea: Use observed transitions to adjust values of observed states so that they satisfy Bellman equations

$$V^{\pi}(s) = V^{\pi}(s) + \underbrace{\alpha}_{\text{Learning rate}}(\underbrace{r(s) + \gamma V^{\pi}(s') - V^{\pi}(s)}_{\text{Temporal difference}})$$

Theorem: If α is appropriately decreased with the number of times a state is visited, then $V^{\pi}(s)$ converges to the correct value.

- α must satisfy $\Sigma_n \alpha(n) \to \infty$ and $\Sigma_n \alpha^2(n) < 1$

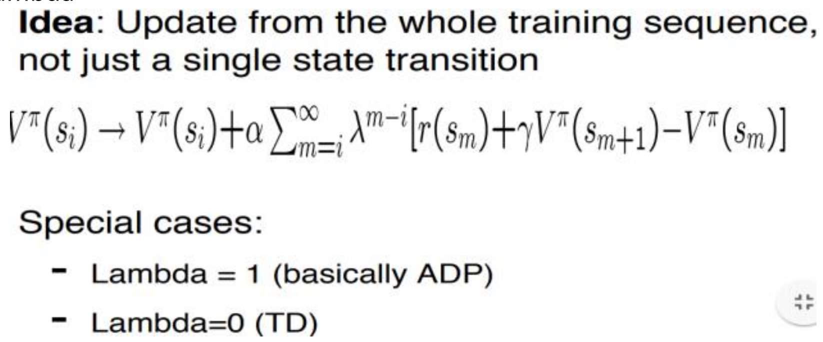
Sample of $V^{\pi}(s)$: sample=$R(s)+\gamma V^{\pi}(s')$

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

        - No explicit model of T or R
        - Estimate V and expectation through samples
        - Update from each experience: update V(s) after each state transition
            - If the new difference is huge, how much the updated value should be affected by the new difference depends on the learning rate
        - Both try to make local adjustments to the utility estimates in order to make each state "agree" with its successors. One difference is that TD adjusts a state to agree with its observed successor (Equation (21.3)), whereas ADP adjusts the state to agree with all of the successors that might occur, weighted by their probabilities (Equation (21.2)). This difference disappears when the effects of TD adjustments are averaged over a large number of transitions, because the frequency of each successor in the set of transitions is approximately proportional to its probability. A more important difference is that whereas TD makes a single adjustment per observed transition, ADP makes as many as it needs to restore consistency between the utility estimates U and the environment model P. Although the observed transition makes only a local change in P, its effects might need to be propagated throughout U . Thus, TD can be viewed as a crude but efficient first

approximation to ADP.

    - TD-Lambda



Idea: Update from the whole training sequence, not just a single state transition

$$V^{\pi}(s_i) \rightarrow V^{\pi}(s_i)+\alpha \sum_{m=i}^{\infty} \lambda^{m-i}[r(s_m)+\gamma V^{\pi}(s_{m+1})-V^{\pi}(s_m)]$$

Special cases:

- Lambda = 1 (basically ADP)
- Lambda=0 (TD)

- Active Learning
- **Behaviour policy: generate action and data, learning: target policy to learn**
    - **On-Policy learning: behaviour== learning**
    - **Off-Policy learning: behaviour !=learning**
    - **Q-learning**: learns an action-utility representation instead of learning utilities, Q: SxA->R
        - a TD agent that learns a Q-function does not need a model, either for learning or for action selection.
        - Optimal policy: the Q(s,a) that returns biggest R
        - Bellman's equation: Q(s,a)=r(s)+Σs'P(s'|s,a)maxa'Q(s',a')
            - The reward of this state+ probability of new state*bestR of new state
        - For each (s,a) initialize Q(s,a)
        - Observe current state
            - Select action a and execute it
            - Observe r and s'
            - Update Q(s,a): **Q(s,a)=Q(s,a)+α(r+γ maxa'Q(s',a')-Q(s,a))**
            - s=s
    - **SARSA**:
        - For each (s,a) initialize Q(s,a)
        - Observe current state S
        - Choose action a from s using policy
            - Take action a, observe r and s'
            - Choose a' from s' according to policy
            - Update Q(s,a): **Q(s,a)=Q(s,a)+α(r+γ Q(s',a')-Q(s,a))**
            - s=s', a=a
        - where a is the action actually taken in state s
        - whereas Q-learning backs up the best Q-value from the state reached in the observed transition, SARSA waits until an action is actually taken and backs up the Q-value for that action. Now, for a greedy agent that always takes the action with best Q-value, the two algorithms are identical. When exploration is happening, however, they differ significantly. Because Q-learning uses the best Q-value, it pays no attention to the actual policy being followed—it is an off-policy learning algorithm, whereas SARSA is an on-policy algorithma Q-learning agent can learn how to behave well even when guided by a random or adversarial exploration policy. On the other hand, SARSA is more realistic: for example, if the overall policy is even partly controlled by other agents, it is better to learn a Q-function for what will actually happen rather than what the agent would like to happen.
            - Q-learning:

Probability

- Axioms
  - $0 \leq P(A) \leq 1$
  - $P(True)=1$
  - $P(False)=0$
  - $P(A \lor B)=P(A)+P(B)-P(A \land B)$
- Marginalization
  - sum of values of the **joint** probability distribution

sunny

	cold	~cold
headache	0.108	0.012
~headache	0.016	0.064

~sunny

	cold	~cold
headache	0.072	0.008
~headache	0.144	0.576

P(headache^sunny^cold)=0.108 P(~headache^sunny^~cold)=0.064

P(headache)+P(sunny)-P(headache and sunny)
P(headache V sunny) = 0.108 + 0.012 + 0.072 + 0.008 + 0.016 + 0.064 = 0.28

P(headache)=0.108+0.012+0.072+0.008=0.2

- **Conditional probability**

  $P(A|B)=P(A \land B)/P(B)$

  Chain Rule:

  - $P(A \land B)=P(A|B)P(B)$
- **Bay's Rule**
  - $P(A \mid B \land C) = P(B \land C \mid A) * P(A)/P(B \land C)$
    $= \alpha P(B \land C \mid A) P(A)$
  - General form of bays rule

Prior probability

Likelihood

$$P(H|e) = \frac{P(e|H)P(H)}{P(e)}$$

Posterior probability

Normalizing constant

- **Independence**
  Two variables A and B are **independent** if knowledge of A does not change uncertainty of B (and vice versa)
  - $P(A|B)=P(A)$
  - $P(B|A)=P(B)$
  - $P(A \land B)=P(A)P(B)$
  - In general: $P(X_1,X_2,\ldots,X_n)=\prod_i P(X_i)$
- Conditional independence: tooth ache and catch are conditionally independent given cavity

- Full independence is often too strong a requirement
- Two variables A and B are **conditionally independent** given C if
  - P(a|b,c)=P(a|c) for all a,b,c
  - i.e. knowing the value of B does not change the prediction of A ***if the value of C is known***

P(a,b|c) = P(a|c)P(b|c)

- Full distribution:

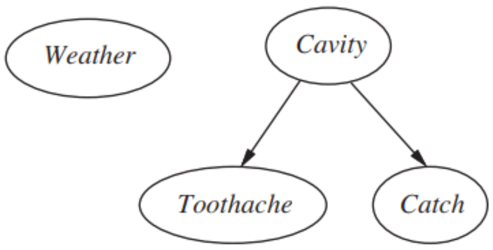
$$\mathbf{P}(Cause, Effect_1, \ldots, Effect_n) = \mathbf{P}(Cause) \prod \mathbf{P}(Effect_i \mid Cause)$$

$$P(x_1, x_2, x_3) = P(x_1)P(x_2|x_1)P(x_3|x_1, x_2)$$

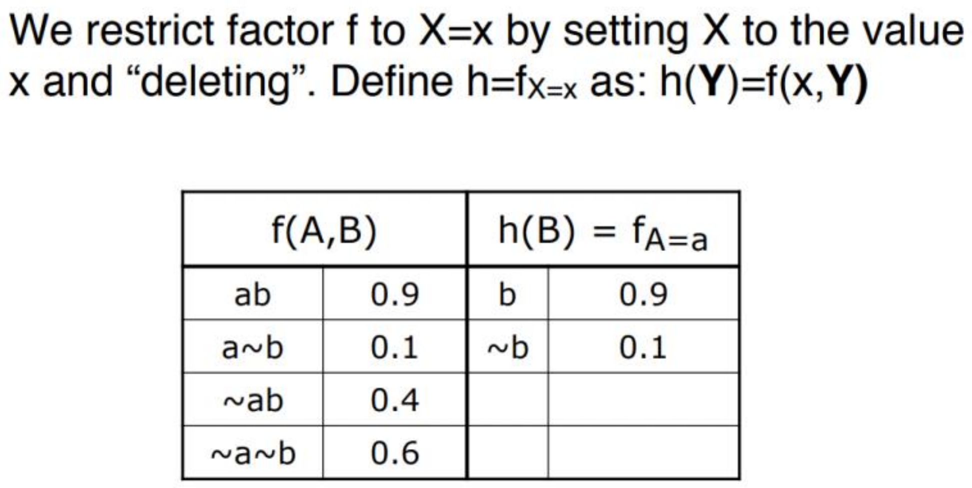
Bayes net

October 30, 2017 11:45 AM

- P(x1,...,xn) = ∏ P(xi | parents(Xi))
- P(Xi | Xi—1,...,X1) = P(Xi | Parents(Xi)) ,
- Probabilities in bayes net :
    - Joint distribution
    - Assume conditional independence
    - **P(x1,x2,...,xn)= P(x1|parent(x1))*p(x2|parent(x2))*....**
- **Directly acylic graph with every node representing random variable**



- Locally structured(compactness):
    - A joint distribution over N boolean variables: 2^N
    - In bayes net, N nodes, each of node has K parents: (2^k+1)*N
- How to construct bayes net:
    - Bayesian network is a correct representation of the domain only if each node is conditionally independent of its other predecessors in the node ordering, given its parents
    - Nodes: order by cause then effect
    - Links: for I=1 to n, choose a minimal set of parent for Xi such that (Xi | Xi—1,...,X1) = P(Xi | Parents(Xi)). Each parent has a link to Xi
- **a node is conditionally independent of all other nodes in the network, given its parents, children, and children's parents**
- Common cause: y->x, y->z
- Causal chain: x->y->z
- Common effect: c->y, z->y
    - X and z independent
    - X and z are not independent given y
- Inference: calculate the posterior probability distribution for a set of query variables, given some observed event—that is, some assignment of values to a set of evidence variables
    - Enumeration: need to calculate hidden variable k^2 possibilities
        - p(+a,+b,+c) = p(+a,+b,+c,+d,+e)+p(+a,+b,+c,+d,-e)+
          p(+a,+b,+c,-d,+e) + p(+a,+b,+c,-d,-e)
    - Forward inference
    - Forward inference with upstream evidence
    - Forward inference with multiple parents
    - Forward inference with evidence
    - Backward inference
    - Variable elimination: join, then sum out
        - Restricted factor



We restrict factor f to X=x by setting X to the value x and "deleting". Define h=$f_{X=x}$ as: h(**Y**)=f(x,**Y**)

f(A,B)		$h(B) = f_{A=a}$	
ab	0.9	b	0.9
a~b	0.1	~b	0.1
~ab	0.4		
~a~b	0.6		

        - Product factor

f(A,B)		g(B,C)		h(A,B,C)			
ab	0.9	bc	0.7	abc	0.63	ab~c	0.27
a~b	0.1	b~c	0.3	a~bc	0.08	a~b~c	0.02
~ab	0.4	~bc	0.8	~abc	0.28	~ab~c	0.12
~a~b	0.6	~b~c	0.2	~a~bc	0.48	~a~b~c	0.12

- Result factor
    - Sum out

      We sum out variable X from f to produce $h=\sum_X f$
      where $h(\mathbf{Y})=\sum_{x\in Dom(X)} f(x,\mathbf{Y})$

f(A,B)		h(B)	
ab	0.9	b	1.3
a~b	0.1	~b	0.7
~ab	0.4		
~a~b	0.6		

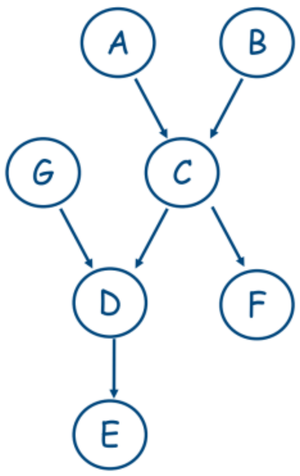
A,B,C

    - Inference

- Elimination ordering
    - eliminate only “singly-connected” nodes first(low degree)
- Relevance
    - Given query Q with evidence E, Q is relevant. If any node Z is relevant, it's parents are relevant. If Evidence is a descendent of a relevant node, E is also relevant

The entire graph is relevant

- P(F)
- P(F|E)
- P(F|E,C)



$$\begin{tikzcd}
 & A \arrow[dr] & & B \arrow[dl] \\
G \arrow[dr] & & C \arrow[dl] \arrow[dr] & \\
 & D \arrow[d] & & F \\
 & E & &
\end{tikzcd}$$

Hidden Markov Models

October 28, 2017 2:28 PM

- Uncertainty over time: set of states, time slices, different possibility distribution over states at different time slices
- Stochastic process: Bayes Net with one random variable per time-slice
    - Problem: Infinitely many variables, infinitely large conditional probability table(n*n)
    - Solution:
        - assume stationary process: dynamics do not change over time
        - Markov assumption: current state depends only on a finite history of past states
            - k-order Markov process: only care about the last k states. Can specify the entire process using finitely many time slices.
            - However, uncertainty increases over time.
- Hidden Markov Model: add sensor data(observations)
    - States are not directly observable
    - Uncertain dynamics increase state uncertainty
    - Emits evidence node to observe when arrive at a state
    - Observation model: If we end up in a state, what is the possibility of making that observation
    - Robot example:
        - S: (x,y) coordinates of the robot on the map
        - O: distances to surrounding obstacles (measured by laser range fingers or sonar)
        - P(st|st-1): movement of the robot with uncertainty
        - P(ot|st): uncertainty in the measurements provided by the sensors
        - Localization: given some sensor data, what is my current state
- Inference:
    - **Monitoring: P(st|ot,...o1) : where am I**
        - The sum of probabilities of the states that lead us here and the observations
        - ∑P(S0,S1,...,St,O1,O2,...,Ot)
          = ∑P(Ot|S0,..,St,O1,…,Ot-1)P(S0,..,St,O1,…,Ot-1)
          = ∑P(Ot|St)P(St|St-1)P(S0,…,St-1,O1,…,Ot-1)
          =∑P(S0)∏P(Si|Si-1)P(Oi|Si)
        - Factor:
            - P(Si|Si-1)P(Oi|Si), 1<=i<=t, P(S0)
            - Restriction O1,…,Ot
            - Eliminate S1,…,St-1
    - **Prediction: P(st+k|ot,...,o1) : where will I be**
        - ∑P(Si|Si-1)P(Si|O1,..,Ot)
    - **Hindsight: P(sk|ot,...,ok,..,o1) : what was my state given a history**
        - P(sk|ot,...,ok,..,o1) = αP(sk|o1,...,ok)<u>P(Ok+1,...,Ot|Sk,O1,...,Ok)</u>
          = α∑P(Ok+1,..,Ot,Sk, Sk+1,...,St)
          = α∑P(Ok+1|Sk+1)P(Sk+1|Sk)<u>P(Ok+2,…,Ot|Sk+1)</u>
        - Factor:
            - P(Si|Si-1)P(Oi|Si), 1<=i<=t, P(S0)
            - Restriction O1,…,Ot
            - Eliminate S1,…,Sk-1, Sk+1,…,St
    - **Most likely explanation: argmaxst,...,s1 P(st,...,s1|ot,...,o1): what is the most likely path**
        - Viterbi algorithm: **autocorrect**
            - Evidence: CAH
            - Most likely sequence leading to this node based on the most likely sequence that lead to this node
            - Transitional score*
            - Caveats: Naïve transition pobability, not guaranteed to produce a real word,
- Dynamic Bayes net

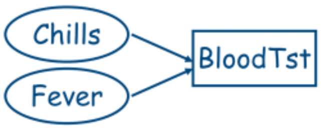
    - Encode states and observations with several random variables
- What if the process is not stationary?
    - Add new state components until dynamics are stationary
    - Example: Robot navigation based on (x,y,θ) is nonstationary when velocity varies. Add velocity to state description (x,y,v,θ)
- What if the process is not Markovian?
    - Solution: Add new state components until the dynamics are Markovian -
    - Example: Robot navigation based on (x,y,θ) is non-Markovian when influenced by battery level. Solution: Add battery level to state description (x,y,θ,b)
- Problem: Adding components to the state description to force a process to be Markovian and stationary may significantly increase computational complexity

Decision Network

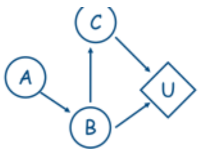
November 1, 2017 4:24 PM

- a representation for sequential decision making
- Decision nodes
    -
- Value nodes: utility of a state
    - Utility depends only on state of parents
- Decision nodes are totally ordered
    - Given decision variables D1,..., Dn, decisions are made in sequence
- No forgetting property
    - Any information available for decision Di remains available for decision Dj where j>i - All parents of Di are also parents for Dj
- A policy δ is a set of mappings δi, one for each decision node Di
    - δi(Di) associates a decision for each parent assignment, δi:Dom(Par(Di))→Dom(Di)
    - EU(δ)=∑xP(x,δ(x))U(x,δ(x))

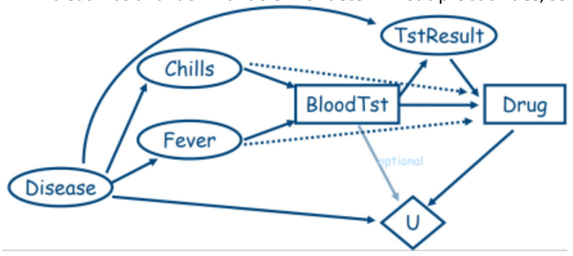
$\delta_{BT}(c,f)=bt$
$\delta_{BT}(c,\sim f)=\sim bt$
$\delta_{BT}(\sim c,f)=bt$
$\delta_{BT}(\sim c,\sim f)=\sim bt$



$$EU = \Sigma_{A,B,C}\ P(A,B,C)\ U(B,C)$$
$$= \Sigma_{A,B,C}\ P(C|B)\ P(B|A)\ P(A)\ U(B,C)$$

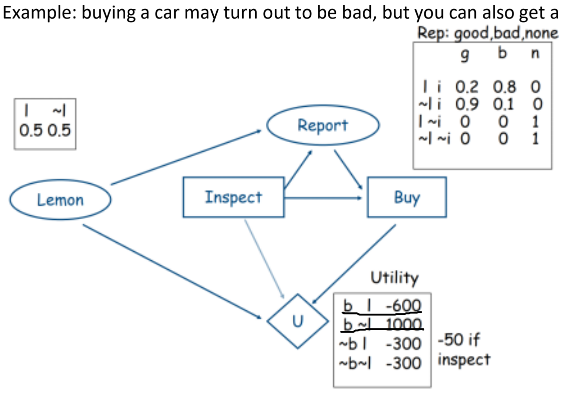


- Optimal policy:
    - Solve backwards.
        - Solve for Drug first
        - treat D as a random variable with deterministic probabilities, solve for BloodTest



- Example: buying a car may turn out to be bad, but you can also get a mechanic for 50. U(good car)>U(no car)>U(bad car)

Good, bad or no report. You can always access the inspection report when you buy
Decision node: buy
EU(buy) = sum P(Lemon, Inspect, Report, buy)U(Lemon, Inspect, Buy)
EU(Buy(Inspect, Report)) = sum P(L|I, R, B)***P(I,R,B)***U(L,I,B) //no uncertainty here
= sum P(L|I, R, B)*U(L,I,B)
Solve fir Eu for every information of L, I, B
Case1: I = inspect, R =good
    EU(~buy|inspect, good) = -350
    EU(buy|inspect, good) = sum of L, P(L| inspect, g, buy)U(L, inspect, buy)
    = P(lemon |inspect, good, buy) *U(lemon, inspect, buy)* + P(~lemon |inspect, good, buy) *U(~lemon, inspect, buy)*
    =0.18*-650+0.82*950
Case 2: I= inspect, R= bad
    EU(~buy|inspect,bad) = -350
    EU(buy|inspect, bad) = -474
Case 3: I= no inspect, R= none
    EU(~buy|~inspect,none) = -300
    EU(buy|~inspect, none) = 200



l	~l
0.5	0.5

Rep: good,bad,none

	g	b	n
l i	0.2	0.8	0
~l i	0.9	0.1	0
l ~i	0	0	1
~l ~i	0	0	1

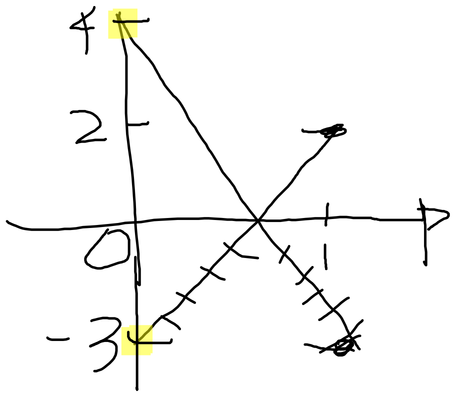
b	l	-600
b	~l	1000
~b	l	-300
~b	~l	-300

    - Decision node: inspect
      EU(I) = sum(L,R) P(L,R,I,B)U(L,I, policy(IR))
- Information has value
    - To the extent it is likely to cause a change of plan
    - To the extent that the new plan will be significantly better than the old plan
    - The value of information is non-negative. This is true for any decision-theoretic agent

Game theory

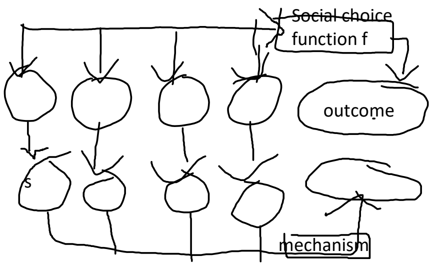
November 6, 2017 4:09 PM

- Analyze interaction among a group of rational agents that behave strategically
    - Action directly influence other
- Set of agents
- Set of actions
- Prisoner's dilemma: logic says confess because it is better no matter what other person does
- Outcome of a game is defined by a profile(tuple)
- Agents have preferences over outcomes
- Two players, a row player and a column player
- Zero sum game: utility add up to zero
- Agents are rational
    - Let Pi be agent I's belief of what its' opponents will do
    - Best response is the most optimal choice given your belief
- Strictly dominates strategy: for every action from the player, it will gives you better utility
- Nash equilibrium:
    - An agents best response depends on the actions of other agents
    - For every single player there is no incentive to change their action
    - Assumed every one has chosen
- Mixed Nash equilibrium
    - A probability distribution over Ai
    - Supposed the row player goes first, plays One with probability P
        - When p=1, row plays One, utility=2 or =-3
        - When p=0, row plays Two, utility=-3 or =4
        - Pick a p value such that col cannot take advantage of your choice
        - If ActionCol =One, Ucol = -2p+3(1-p)
        - If ActionCol =Two, Ucol = 3p-4(1-p), set 1=2, solve for p
- Repeated games: S: H->A
    - Grim strategy: cooperate first, defect forever
    - Tit-for-tat: copy every opponent did
- Extensive form game normal form game
- Subgame perfect equilibrium
    - Every finite extensive form game has an SPE
    - Solve SPE using backward induction
    - Identify sub eq in bottom

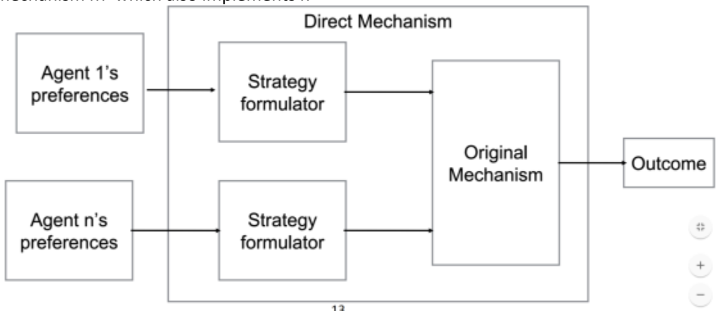


Mechanism design

November 8, 2017 4:11 PM



- What sort of game should we design
- Fundamentals:
  - Set of possible outcomes: O
  - Set of agents N, size of N is n
    - Each agent has a type theta, the type is the agent's preference
  - Utility function u(o, theta), the utility of an outcome given agent's type
  - Social choice function f(theta1 * theta2*,…,*thetan) ->O
    - Voting: choose the candidate from population
  - Agent types are not public, agents are acting in their own self interest
  - Mechanism: M(S1,..,Sn, g())
    - Si is the strategy space of agent i
    - g:S1 × … × Sn → O is the outcome function
- A mechanism M=(S1,…,Sn,g()) implements social choice function f(θ) if there is an equilibrium s* that produce same outcome
  - Direct mechanism has strategy space that is same as the type space
  - A direct mechanism is incentive compatible if it has an equilibrium s* where s*(theta) = theta
    - Being honest
- A direct mechanism is **strategy proof** if the equilibrium above is a dominant strategy equilibrium
- Suppose there exists a mechanism M that implements social choice function f in dominant strategies. Then there is a direct st rategy-proof mechanism M' which also implements f.



- Gibbard-Satterthwaite Theorem: O is finite and |O|>2 - Each o in O can be achieved by SCF f for some θ - Θ includes all possible strict orderings over O Then f is implementable in dominant strategies if and only if f is dictatorial.
- Circumventing Gibbard-Satterthwaite:
  - Single peak: a certain preference at a certain point, and gradually reducing around the point
  - Median voter rule
  - Outcome o=(x,t1,…,tn)
    - x is a "project choice"
    - ti in ℝ are transfers ("money")
  - Utility functions: ui(o,𝜃i)=vi(x,𝜃i)-ti
  - Quasilinear mechanism M=(S1,…,Sn,g()) where - g()=(x(),t1,…,tn)
- Groves algorithm
  - Choice rule

    Sum over the utility of each outcome for everyone, then choose the one that benefit everyone the best "social welfare"

    $$x^*(\theta) = \arg\max_x \sum_i v_i(x, \theta_i)$$

  - Transfer rules

    If t is positive, the demand payment from that person
    h(theta) is determined by everyone else but I
    The summation of how happy everyone else is given the current outcome.

    $$t_i(\theta) = h_i(\theta_{-i}) - \sum_{j \neq i} v_j(x^*(\theta), \theta_j)$$

  - Groves mechanisms are strategy-proof and efficient.
  - Groves mechanisms are unique (up to hi(𝜃-i))
  - VCG mechanism: Efficient and strategy-proof, Agents' equilibrium utility is their marginal contribution to the welfare of the system

    Outcome

    $$x^* = \arg\max_x \sum_i v_i(x, \theta_i)$$

    Transfers

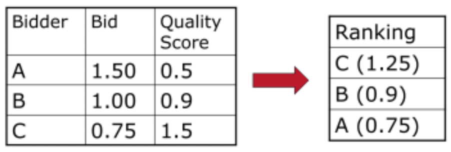
    $$t_i(\theta) = \sum_{j \neq i} v_j(x^{-i}, \theta_j) - \sum_{j \neq i} v_j(x^*(\theta), \theta_j)$$

    Marginal contribution to social welfare:
    What happens if you are not participating - everyone else happiness now that you are

- Example:
  - Social choice function: Maximize social welfare (i.e. give item to agent who values it the most)
  - Utility functions: ui=vi(o)-ti
  - Mechanism (Vickrey Auction)
    - Si: a bid of any non-negative number
    - Outcome function g:
      - Give item to agent who submits highest bid
      - Highest bidder pays amount of second highest bid, all else pay nothing
  - Sponsored search: 1. Advertisers are ranked and assigned slots based on the ranking. 2. If an ad is clicked on, only then does the advertiser pay.
    - Assign slots of order of (quality score)*(bid)
    - The lowest price it could have bid and still been in the same position

ti=sum of everyone else's welfare if you don't participate-current outcome

V1= 6, winner	v(x) = 6	5	0	5
V2 =5	0	6	6	0
V3=2	0	6	6	0



Bidder	Bid	Quality Score
A	1.50	0.5
B	1.00	0.9
C	0.75	1.5

Ranking
C (1.25)
B (0.9)
A (0.75)

C will pay p=0.9/1.5=0.6
B will pay p=0.75/0.9 = 0.83

0
How much will A pay?

Machine Learning

November 13, 2017 4:03 PM

- Autonomous driving
  - Task: driving on a public highway using vision sensor
  - Performance matric: average distance traveled before an error was made
  - E: sequence of image and driving command while observing a human driver
- Types of learning
  - Supervised(inductive) learning
    - Learn a function from inputs and outputs
    - Example: handwriting recognition
    - Techniques: decision trees, neural networks
  - Unsupervised learning
    - Learn patterns in the input when no specific output is given
    - Web log data, access patterns
    - clustering
  - Reinforcement learning
    - Learn from feedback
- Representation: of learned information
  - Linear weighted polynomials
  - Proposition logic
  - First order logic
  - Bayes net
- Hypothesis representation:
  - Conjunction of constraints
  - If some instance x satisfy all constraints of hypothesis h, h(x) =1
  - Hypothesis space:
    - Most general: accept all instances
    - Set f all possible hypothesis that the learner may consider
    - Learning is a search through a hypothesis space
- Inductive learning hypothesis
  - Given enough training data, we can assume it can actually perform well on unobserved data
- Construct h to agree with f on training set. H is consistent if it agrees with f on all examples
- Realizable: hypothesis space contains the true value
  - Large hypothesis space: expressiveness of hypothesis vs complexity of finding it
- Linear threshold classifiers
  - H(x) = 1 or 0 depending on if w*x >0. Find weights w to minimize loss
  - Loss function: evaluate how good a particular classifier is
    - L(y, h(x)) = sum(y-h(x))^2
    - Input: real label and the label assigned by our classifier
    - Square of the difference between ground truth and classification by hypothesis
    - Start with arbitrary line and change the line every time it is wrong
    - Gradient descent to the space of all linear classifier(hypothesis space)
      - Start: initialize w
      - Select: an example (x, y)
      - Update: wi<-wi + learningFactor*(y-h(x))xi
        - If y = h(x), no change in wi
        - If y =1, h(x) =0, if xi>0 wi increases, if xi<0, wi decreases
- Decision tree: classify instances by sorting them down the tree from robot to leaf
  - Fully expressive within the class of propositional languages
  - No representation is efficient enough for all situation
  - Leaf node:
  - Choose a good attribute that split examples into subsets that are all positive and all

    negative

- Information theory
    - Its better asking questions that can get rid of high uncertainty
    - Assign a numeric value to uncertainty
    - Information content(Entropy)
        - Given a random variable X with size n, entropy H(X) = sum of -P(xi)logP(Pxi)