CS486Artifitial Intelligence

Introduction

Goal: understand and build intelligent entities.

Turing's Test

- Natural Language Processing
- Knowledge Representation
- Automated Reasoning
- Machine learning

Total Turing's Test

- Computer Vision
- Robotic

"Law of thought" approach

- A precise notation for statements about all kinds of objects and the relations among them
- Programs to solve any solvable problem described in logical notation

Rational Agent Approach

- **Agent**: something that acts
- **Rational Agent**: acts to achieve the best outcome, or (if uncertainty exists) the best expected outcome
- Advantages:
    - More general
    - More amenable

Dscision Theory: combines probability theory with utility theory, provides a formal and complete framework for decisions made under uncertainty

History of Artificial Intelligence

- **The Gestation of AI (1953-1955)**
    - MeCulloch & Pitts (1943) proposed a model of artificial neurons (boolean circuit of the brain)
    - First neural network computer in 1950
    - Alan Turing's lecture "Computer Machinery and Intelligence" and introduced Turing Test

- **Birth of AI (1956)**
  - Dartmouth workshop (1956): to invent computre program capable of thinking non-numerically
- Early Enthusiasm, Great Expectations (1952–1969)
  - program designed from the start to imitate human problem-solving protocols
  - fomulate **physical symbol system** hypothesis
  - Some of the first AI programs:
    - Geometry Theorem Prover
    - Checkers AI player
    - In 1958, MeCarthy defined the high-level language **Lisp**
  - **Perceptron Convergence Theorem**
- **A dose of reality (1966-1973)**
  - difficulties in AI:
    - most early programs knew nothing of their subject matter, they succeeded by means of simple syntactic manipulations
    - intractability of many of the problems that AI was attempting to solve
    - some fundamental limitations on the basic structures being used to generate intelligent behavior
- **Knowledge-based systems (1969-1979)**
  - **Weak Methods**, general but do not scale up to large or difficult problem instances => Use powerful, domain-specific knowledge that allows larger reasoning steps
  - **Expert systems**
- **AI as an industry (1980-present)**
  - First commercial expert systems, R1, helped configure orders for new computer systems (1982)
  - AI industry boomed from a few million dollars in 1980 to billions of dollars in 1988
  - Soon after commes a period called "**AI Winter**"
- **The return of neural networks (1986-present)**
  - **back-propagation** learning algorithm reinvented in mid-1980. The algorithm was applied to many learning problems in computer science and psychology.
  - **Connectionist** models of intelligent systems
- **AI adopts the scientific methods (1987-present)**
  - **hidden Markov models**
  - **data mining** technology
  - **Bayesian network**
- **The emergence of intelligent agents (1995-present)**
  - AI systems becoe so common in Web-based applications

    - search engines, recommender systems and website aggregators
- **The availability of very large data set (2001-present)**
    - previously the emphasis has been on the algorithm, but recent work in IA suggests that data is more important
    - problem of how to express the knowledge may be solved by learning methods rather than hand-coded knowledge engineering

Intelligent Agents

Agent: an **agent** is anything can be viewd as perceiving its **environment** through **sensors** and acting upon that envioronment through **actuators**.

- Human agent: has eyes, ears ... as sensors and hands, legs, vocal tract ... for actuators
- Robotic agents: may have cameras and range finder for sensors and motors for actuators
- Software agent receives keystrokes, file contents and network packets as sensory inputs and acts on environment by displaying on the screen, writing files and send network packets

percept: agent's perceptual inputs at any given instant

percept sequence: the complete history of everything the agent has ever perceived

agent function: maps any given percept sequence to an action

agent program: implement the agent function for an artificial agent

rational agent: agent that does the right thing (every mapping in agent function is filled correctly)

performance measure: evaluates any given sequence of enviroment states

Definition of a Rational Agent: For each possible percept sequence, a rational agent should select an action that is ex- pected to maximize its performance measure, given the evidence provided by the percept sequence and whatever built-in knowledge the agent has.

Task environment

- **PEAS**: Performance, Envoronment, Actuators, Sensors
- **Properties of task environment**
    - **Fully observalbe** VS **partially observable**
        - **Fully observalbe**: agent’s sensors give it access to the complete state of the environment at each point in time
        - **Partially observable**: An environment might be partially observable because of noisy and inaccurate sensors or part of states are missing
    - **Single agent** VS **multiagent**
        - **multiagent** may used in **competitive** multiagent environment (eg. chess) or **cooperative** multiagent environment

        - communication emerges as a rational behavior in multiagent environments
    - **Deterministic** VS **stochastic**
        - **deterministic**: next state of the environment is completely determined by the current state and the action executed by the agent
        - **stochastic**: uncertainty exists and is quantified in terms of probabilities
    - **Episodic** VS **sequential**
        - **episodic**: agent's experience is divided into atomic episodes (one action a time, next action dose not depend on the action taken in previous episodes)
        - **sequential**: the current decision could affect all future decisions
    - **Static** VS **dynamic**
        - **dynamic**: environment can change while an agent is deliberating
    - **Discrete** VS **continuous**
        - the discrete/continuous distinction applies to the state of the environment
        - eg. chess is discrete and auto-driving is continuous
    - **Known** VS **unknown**
        - refers to the agent's state of knowledge about the "laws of physics" of the environment
        - the environment is unknown, the agent will have to learn how it works in order to make good decisions

Searching

uninformed search algorithms: algorithms that are given no information about the problem other than its definition.

Intelligent agents adopt a **goal** and aim at satisfying it.

Goal formulation: organize behavior by limiting the objectives and actions it needs to consider. Consider a goal to be a set of world states (exactly those states in which goal is satisfied)

Problem formulation: the process of deciding what actions and states to consider given a goal.

An agent with several immediate options of unkown value can decide what to do by first examining future actions that eventually lead to states of known value

Search: the process of looking for a sequence of actions that reaches the goal.

formulate, search, execute

- A **problem** can be defined as:
    - **initial state**: the state the agent starts in
    - **possible actions**: given a particular state and returns a set of actions that is **applicable** in s

    - **transition model**: given a state and an action, returns **successor** states results from the action
    - **goal test**: determines whether a given state is a goal state
    - **path cost**: a function returns **step cost** of taking action in a particular state
- a **solution** to a problem is an action sequence that leads from the initial state to a goal state. An **optimal solution** has the lowest path cost amoong all solutions.

Search tree: initial state at the root and the branches are actions and the nodes correspond to states in the state space of the problem

> **Generic Search Algorithm**
>
> - Repeat
>   1. if no candidate nodes can be expanded, **return failure**
>   2. Choose leaf node for expansion, according to **search strategy**
>   3. Test whether this is a goal state, return solution if it is goal state
>   4. **Expanding** the current state: applying each legal action to the current state to generate a new set of states. Add these new child nodes to the parent node

search strategy: decide which state to expand next

frontier: leaf nodes available for expansion (aka. **open list**)

repeated state in the search tree, (usually by a loopy path), can cause the search tree to be **infinite** big.

Loopy paths are a special case of the more general concept of **redundant paths**

The way to avoid exploring redundant paths is to remember where one has been. Use **explored set** (aka. **closed list**) to remember every expanded node

- **Measuring problem-solving performance**
    - **Completeness**: is this algorithm guaranteed to find a solution?
    - **Optimality**: Does the strategy find the optimal solution?
    - **Time complexity**
    - **Space complexity**
- data structures:
    - V: the set of vertices(nodes)
    - E: set of edges(links)
    - b: **branching factor**, maximum number of successors of any node
    - d: **depth** of the shallowest goal node
    - m: maximum length of any path in the state space

Uninformed search

Uninformed search (aka. **blind search**): the strategies have no additional information about states beyond that provided in the problem definition

Breadth-first search (BFS)

Breadth-first search: root node is expanded first, then all the successors of the root node are expanded next, then their successors and so on. **Shallowest** unexpanded node is chosen for expansion

Completeness: Yes. If the shallowest goal node is at some finite depth, BFS will eventually fin it after generating all shallower nodes. It is the shallowest goal node since all shallower nodes must have been generated already and failed the goal test.

Optimality: No. Shallowest node is not necessarily the optimal one unless the path cost is a nondecreasing function of the depth of the node.

Time complexity: b + b^2 + b^3 + ... + b^d = O(b^d)

Space complexity: need to store every expanded node in the explored set. There will be O(b^d−1) nodes in the explored set and O(b^d) nodes in the frontier. So the space complexity is **O(b^d)**

the memory requirements are a bigger problem for breadth-first search than is the execution time

exponential-complexity search problems cannot be solved by uninformed methods for any but the smallest instances

Depth-first search (DFS)

Depth-first search always expands the **deepest** node in the current frontier of the search tree

Completeness: Yes / No. If graph-search version is used, will avoids repeated states and redundant paths, it is complete. Otherwise, if tree-search version is used, it is not complete.

Optimality: No. Deepest explored node may not be optimal.

Time complexity: bounded by the size of the state space. Worst case generate **O(b^m)** nodes in search tree, where m is maximum depth of any node. (m can be >> d, may be infinite if the tree is unbounded)

Space complexity: needs to store only a single path from the root to a leaf node, along with the remainning unexpanded sibling nodes for each node on the path. It requires storage of only **O(bm)** nodes.

A variant of depth-first search called **backtracking search** uses still less memory.

Iterative deepening depth-first search (IDS)

IDS combines the benefits of DFS and BFS

Finds the best depth limit and gradually increasing the limit until a goal is found.

Completeness: Yes when the branching factor is finite, otherwise No

Optimal: Yes when the path cost is a nondecreasing function of the depth of the node, otherwise No

Space complexity: O(bd)

Time complexity: O(b^d), same as BFS

In general, iterative deepening is the preferred uninformed search method when the search space is large and the depth of the solution is not known.

Uniform-cost search (UCS)

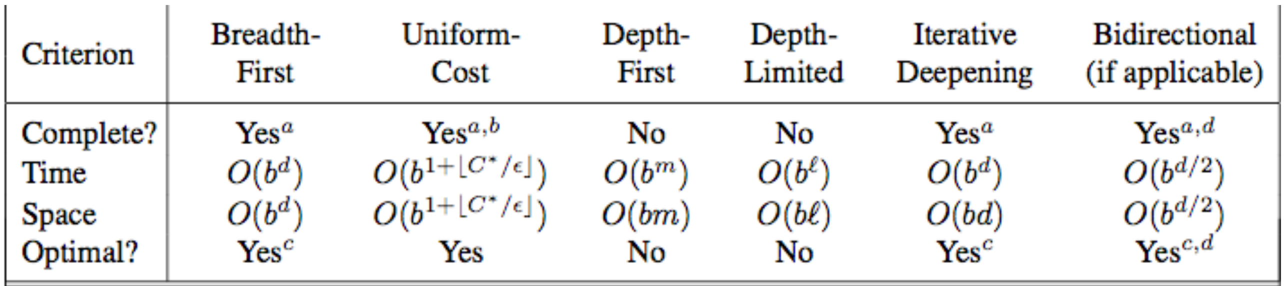
uniform-cost search expands the node n with the lowest path cost g(n), done by storing the frontier as a priority queue ordered by g

Goal test is applied to a node when it is **selected for expansion**, rather than when it is first generated.

Completeness: No, it may get stuck in an infinite loop if there is a path with an infinite sequence of zero-cost actions. Completeness is guaranteed provided the cost of every step exceeds some small positive constant ε

Optimal: Yes, uniform-cost search expands nodes in order of their optimal path cost

Time and Space complexity: O(b^(1 + C*/ε)), where C* is the cost of optiam solution, it can be much greater than b^d



Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes^{a}	Yes^{a,b}	No	No	Yes^{a}	Yes^{a,d}
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes^{c}	Yes	No	No	Yes^{c}	Yes^{c,d}

Figure 3.21 Evaluation of tree-search strategies. b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; l is the depth limit. Superscript caveats are as follows: ^{a} complete if b is finite; ^{b} complete if step costs $\geq \epsilon$ for positive ϵ; ^{c} optimal if step costs are all identical; ^{d} if both directions use breadth-first search.

Informed Search

informed search: uses problem-specific knowledge beyond the definition of the problem itself, it can find solutions more efficiently than uninformed strategy

evaluation function: f(n), is constructed as a cost estimate.

heuristic function: h(0), estimated cost of the cheapest path from the state at node n to a goal state

Best First Search

Best First Search: expand the most promising node according to the heuristic, using f(n) = h(n)

at each step it tries to get as close to the goal as it can

Completeness: No even in a finite state space. consider the neibough sate has a lowest estimated cost but is a dead end.

Optimal: No

Time and Space complexity: Worst case O(b^m), but with good heuristic function, the complexity can be reduced substantially

A* Search

A* search evaluates nodes by combining the cost to reach the node (g(n)) and the estimated cost to get from the node to the goal (h(n))

f(n) = g(n) + h(n) --- f(n) is estimated cost of the cheapest solution through n

Conditions for optimality: require that h(n) to be an **admissible heuristic**

Admissible heurisitc: never **overestimates** the cost to reach the gaol

Admissible heuristics are by nature optimistic because they think the cost of solving the problem is less than it actually is

If the heuristic is admissible then A* with tree-search is optimal

consistency: for every node n and every successor n' of n generated by any action a, the estimated cost of from n is no greater than the cost of getting to n' plus the estimated cost from n, **h(n) <= cost(n, a, n') + h(n')**

A* search on graphs with a consistent heuristic is optimal

Whenever A*selects a node n for expansion, the optimal path to that node has been found. (prove by contradiction)

Completeness: requires that there be only finitely many nodes with cost less than or equal to C*

Optimal: If the heuristic is **admissible** then A* with **tree-search** is optimal or A* search on graphs with a consistent heuristic is optimal

Time complexity: worst case entire tree but the use of a good heuristic still provides enormous savings

Space complexity: need to keep all paths and partial path in memory. A* is not practical for many large-scale problems.

Memory-bounded heuristic search

Iterative Deepening A* (IDA*)

DFS with f-value to decide which order to consider nodes

Use f-limit instead of depth limit

IDA* has same properties as A* but with less memory used

Simplified Memory-Bounded A* (SMA*)

Uses all available memory

Proceeds like A* but when it runs out of memory it **drops the worst leaf node**. If all leaf nodes have same f-value, **drop oldest and expand newest**

Optimal and complete if depth of shallowest goal node is less than memory size

Heurisitcs functions

Generating admissible heuristics from relaxed problems

relaxed problem: a problem with fewer restrictions on the actions

Generating admissible heuristics from subproblems

Precompute solution costs of **subproblems** and storing them in a **pattern database**

derive heuristic function from the solution cost of a **subproblem** of a given problem

pattern databases: to store exact solution costs for every possible subproblem instance

Learning heuristics from experience

Constraint Satisfaction Problems (CSP)

- **constraint satisfaction problem**
    - a set of variables with values from domains
    - problem is solved when each variable has a value taht satisfies all the constraints on the variable
- Defining CSP:
    - X is a set of variables $\{X_1, X_2, \dots, X_n\}$

    - D is a set of domains $\{D_1, D_2, \ldots, D_n\}$
    - C is a set of constraints that specify allowable combinations of values
    - each **state** in a CSP is defined by an **assignment** of avalues to some or all of the varibales
    - An **assignment** is **consistent** if it does not violate any constraints
    - A **complete assignment** is which every variable is assigned
    - A **partial assignment** is one that assigns values to only some of the variables
    - A **solution** is a **consistent**, **complete** assignment
- Types of CSPs:
    - **Finite domains** VS **Infinite domains**
        - Finite domains: if domain has size d, there are O(d^n) complete assignments (eg. boolean CSPs)
        - Infinite domains: eg. set of intergers or strings.
            - a **constraint language** must be used that understands constraints such as
            - **linear constraints** are solvable but **non-linear** are undecidable
    - **Continuous domains** VS **Discrete domains**
        - best-known category of **continuous-domain CSPs** is **linear programming problems**, where constraints must be linear equalities or inequalities. Linear programming problems can be solved in polynomial time
    - **Varieties of Constraints**
        - **unary constraint**: restricts the value of a single variable
        - **binary constraint**: constraints relate two variables
        - **higher-order constraints**: involve more than two variables
        - **global constraint**: constraints involving an arbitrary number of variables
    - **Preference of Constraints (soft constraints)**
        - indicating which solutions are preferred
        - such problem is **constraint optimization problem**

node-consistent: A single variable is **node-consistent** if all the values in the variable's domain satisfy the variable's unary constraints

arc-consistent: A variable in a CSP is **arc-consistent** if every value in its domain satisfies the variable's binary constraints

A network is **arc-consistent** if every variable is arc consistent with every other variable.

Backtracking Search for CSPs

CSPs are commutative

commutativity: Order of actions taken does not effect outcome, can assign variables in any order

- **Backtracking Search** is used for a depth-first search that chooses values for one variable at a time and backtracks when a variable has no legal values left to assign
- repeatedly chooses an unassigned variable, tries all values in the domain of that variable in turn, trying to find a solution
- If an **inconsistency** is detected, then **BACKTRACK** returns failure, causing the previous call to try another value

> 1. Select unassigned variable X
> 2. For each value {x1, ..., xn} in domain of X
>   - if value satisfies constraints, assign X=xi and exit loop
> 3. If an assignment is found, move to next variable
> 4. If no assignment found, **back up** to preceding variable and try a different assignment for it

-

Variable and value ordering

Most Constrained Variable (aka. minimum-remaining-values): choose the variable with **fewest legal moves**.

Most Constraining Variable: Choose variable with **most constraints** on remaining variables. **degree heuristic** attempts to reduce the branching factor on future choices by selecting the variable that is involved in the largest number of constraints

Least Constraining Value: Given a variable, choose the least constraining value. It prefers the value that rules out the fewest choices for the neighboring variables in the constraint graph

Filtering

Forward Checking

Keep track of remaining legal values for unassigned variables

Terminate search when any variable has no legal values

Arc Consistency

Given domains D1 and D2, an arc is consistent if for all x in D1 there is a y in D2 such that x and y are consistent

Structure

Independent Subproblems

Break down the graph into its connected components. Solve each component separately.

Significant potential savings: Assume each component involves c variables (n/c components) for some constant c: O(d^c * n/c)

Tree Structures

CSPs can be solved in O(nd^22) if there are no loops in the constraint graph

By **topological sort**

> 1. For i=n to 1, make Xi arc-consistent with parent(Xi) (Pre-processing)
> 2. For i=1 to n, assign value to Xi consistent with parent(Xi) (No backtracking!)

Cutsets

- Choose a subset S of variables such that the constraint graph **becomes a tree** when S is removed (S is the cycle cutset)
- For each possible valid assignment to the variables in S
    1. **Remove from the domains** of remaining variables, all values that are inconsistent with S
    2. If the remaining CSP has a solution, return it
- Running time: O(d^c * (n-c) * d^2) where c is the size of the cutset

Tree Decomposition

- tree decomposition is possible if:
    1. Each variable appears in at least one subproblem
    2. If two variables are connected by a constraint, then they (and the constraint) must appear together in at least one subproblem
    3. If a variable appears in two subproblems in the tree, it must appear in every subproblem along the path connecting those subproblems
- Algorithm:
    1. solve each **subproblem independently**
    2. Solve constraints connecting the subproblems using tree-based algorithm

-

Iterative Improvement and Local Search

Local search: purely performed in the state space, evaluating and modifying one or more current states rather than exploring

These algorithms are suitable for problems in which **all that matters is the solution state**, not the path cost to reach it

- **Local search** algorithms operate using a single current node and move only to neighbors
- Advantages:
    - use very little memory (usually constant)
    - often find reasonable solution in a large or infinite (continuous) state spaces
- useful for solving pure **optimization problems** (find the best state according to an **objective function**
- **state-space landscape**: consist "location" (defined by state) and "elevation" (defined by the value of objective function)
    - Local search algorithms explore this landscape
    - A **complete** local search algorithm always finds a goal if one exists
    - an **optimal** algorithm always finds a global minimum/maximum

Iterative Improvement Methods

Goal: find the highest (or lowest) point

> 1. start at some random point
> 2. Generate all possible points to move
> 3. If the set is not empty, choose a point and move to it
> 4. If stuck (set is empty), restart

Hill Climbing Search

Idea: Always move in the direction of increasing objective value, it terminates when reaches a "peak" (no neighbor has a higher value).

Do not maintain a search tree, need only record the **state** and the value of the **object function**

Local maxima: a peak that is higher than each of its neighbour state but lower than the global maximum

Ridges: results in a sequence of local maxima that is very difficult for a greedy algorithms to navigate

Plateaux: a flat area of the state-space landscape. A hill-climbing search might get lost on the plateau

> 1. Start with some initial configuration S, with value V(S)
> 2. Generate Moveset(S) = {S1, S2, ..., Sn}
> 3. Smax = argmax_Si V(Si)
> 4. If V(Smax) < V(S): return S   (local optimium)
> 5. Let S = Smax Go to 2

- Good:
    - Easy to program

    - Requires no memory of the path or tree
- Bad:
    - Not necessarily complete
    - Not optimal
    - It can get stuck in local optima/plateaus
- Improvement:
    - **sideways move**: allow a sideways move in plateaus, but be carefull for an infinite loop when there are no uphill moves
    - **random restarts**: "If at first you don’t succeed, try, try again"
    - **Stochastic hill climbing**: chooses at random from among the uphill moves; the probability of selection can vary with the steepness of the uphill move
    - **Simulated annealing**

Simulated annealing

A hill-climbing algorithm that never makes "downhill" moves toward states with lower value (or higher cost) is guaranteed to be incomplete. (since it will stuck at local maximum

Simulated annealing: combine hill climbing with a random walk in some way that yields both efficiency and completeness

annealing: the process used to temper or harden metals and glass by heating them to a high temperature and then gradually cooling them

Instead of picking the best move, however, it picks a **random** move

> 1. Start with some initial configuration S, with value V(S)
> 2. Generate Moveset(S) = {S1, S2, ..., Sn}
> 3. Randomly choose Si from Moveset(S)
> 4. Define ΔV = V(Si) - V(S)
>   - If ΔV > 0: then S = Si
>   - Else: S = Si with probability S(S, Si)
> 5. goto 2

- If new value Vi is **better** than old value V then definitely move to new solution,
- If new value Vi is **worse** than old value V then move to new solution with probability e^(ΔV/T)
- **“temperature” T**: T a positive parameter called temperature. It starts high and decreases over time towards 0.
    - when T is high **(Exploratory phase)**: bad move have a chance of being picked (random walk)
    - when T is low **(Exploitation phase)**: bad move is not likely being selected

- If T is ecreased slowly enough then simulated annealing is (theoretically) guaranteed to reach optimal solution

Genetic algorithms

genetic algorithm (GA): successor states are generated by combining two parent states rather than by modifying a single state

population: begin with a set of k randomly generated states

individual: each candidate solution is a individual, is represented as a string over a finite alphabet

fitness function: the value rated by the objective function for each state

Populations change over **generations** by applying operators to them **(selection, mutation, crossover)**

> Initialize: Population P <- N random individuals
>
> Evaluate: For each x in P, compute fitness(x)
>
> Loop:
>
> ```
> - For i = 1 to N
>     1. **select** 2 parents each with probability proportional to fitness scores
>     2. **crossover** the 2 parents to produce a new bitstring (child)
>     3. With some small probability **mutate** child
>     4. Add child to population
> ```
>
> - Until some child is fit enough or you get bored
>
> Return best child in the population according to fitness function

- Selection:
  - Fitness proportionate selection
    - $P(i) + \frac{fitness(u)}{\sum_j fitness(j)}$
    - can lead to overcrowding
  - Tournament selection
    - Pick two random individual with uniform probability
    - With probability p select filter one
  - Rank selection
    - sort all by fitness
    - Probability of selection is proportional to rank
  - Softmax (Boltzmann) selection

        - $P(i) = \frac{e^{fitness(i)/T}}{\sum_j e^{fitness(j)/T}}$

- Crossover
    - Combine parts of individual to create new ones
    - a crossover point is chosen randomly from the positions in the string
- Mutation:
    - generates new features that are not present in original population
    - typically one digit was mutated (fliped)
    - can allow mutation in all individuals or just new offspring

In practice, genetic algorithms have had a widespread impact on optimization problems

second-best way to solve a problem

Adversarial Search

In **multiagent environments**, each agent needs to consider the actions of other agents and how they affect its own welfare

competitive environments: agents' goals are in conflict

game theory: views any multiagent environment as a game, provided that the impact of each agent on the others is "significant," regardless of whether the agents are cooperative or competitive

- **Perfect vs Imperfect Information**
    - **Perfect information**: agen can see the entire state of the game
    - **Imperfect information**: only have partial information
- **Deterministic vs Stochastic**
    - **Deterministic**: change in state is fully controlled by the players
    - **Stochastic**: change in state is partially determined by chance

hard to solve: make **some decision** even when calculating the optimal decision is infeasible

- S_0: The **initial state**, specifies how the game is set up at the start
- PLAYER(s): Defines which player has the move in a state
- ACTIONS(s): Returns the set of legal moves in a state
- RESULT(s, a): The **transition model**, which defines the result of a move
- TERMINAL_TEST(s): A **terminal test**, returns true when game is over. States where the game is ended is called **terminal states**
- UTILITY(s, p): A **utility function** (aka. objective or payoff function) defines the final numeric value for a game ends in terminal state s for player p.

The initial state, ACTIONS function, and RESULT function define the **game tree** for the game

- Notation in a 2 player competitive game:

- **MAX** player wants to maximize its utility
- **MIN** player wants to minimize its utility

Optimal Strategies

ply: tree is one move deep, consisting of two half-moves, each of which is called ply

- MINIMAX(s) =
  - if TERMINAL_TEST(s): UTILITY(s)
  - if PLAYER(s) = MAX: max_a∊Actions(s) {MINIMAX(RESULT(s,a))}
  - if PLAYER(s) = MIN: min_a∊Actions(s) {MINIMAX(RESULT(s, a))}

Player MAX assumes that MIN plays optimally, it maximizes the worst-case outcome for MAX.

The minimax algorithm

The **minimax algorithm** computes the minimax decision from the current state.

Completeness: No (eg. infinite tree)

Optimality: Yes, if the opponent is also optimal

Time complexity: $O(b^m)$, The minimax algorithm performs a complete depth-first exploration of the game tree.

Space complexity: $O(bm)$ if generates all actions at once, or $O(m)$ if generates actions one at a time

for multiplayer games

Replace the single value for each node with a vector of values. For terminal states, this vector gives the utility of the state from each player's viewpoint. Each player choose the action that maximize its utility value

Alpha–Beta Pruning

- Goal: **pruning** to eliminate large parts of the tree
- Idea: If Player has a better choice m either at the parent node of n, then n will never be reached in actual play. So we can prune it.
- α = the value of the best choice we have found so far at any choice point along the path for MAX.
- β = the value of the best choice we have found so far at any choice point along the path for MIN.
- Update alpha and beta as search continues
- Prune as soon as value of current node is known to be worse than current alpha or beta values for MAX or MIN
- This algorithm returns the same move as minimax would

Evaluation Functions

Apply an evaluation function to state. If not terminal state, returns estimate of the expected utility.

- Requirements:
  - wins must evaluate better than draws, which in turn must be better than losses
  - computation must not take too long evaluation function should be strongly correlated with the actual chances of winning
  - evaluation function should be strongly correlated with the actual chances of winning
- How to find
  - expert knowledge / learn from experience
  - Look for features:
    - expected value
    - material value
    - weighted linear function

evaluation function effectively turning nonterminal nodes into terminal leaves

Cutting Off Search

- When?
  - Arbitrarily (deeper is better)
  - **Quiescent** states (unlikely to exhibit wild swings in value in the near future (stable))
  - Singular extensions
    - Searching deeper when you have a move that is "clearly better" than all other moves in a given position
    - Can be used to avoid **horizon effect** (program is facing an opponent's move that causes serious damage and is ultimately unavoidable)

Stochastic Games

stochastic games: games has unpredictable external events by including a random element

A game tree in stochastic games must include **chance nodes**

Chance node: a branches leading from each chance node denote the possible outcome with its probability

idea: generalize the **minimax value** for deterministic games to an **expecti-minimax value** for games with chance nodes

- EXPECTIMINIMAX(s) =
  - if TERMINAL_TEST(s): UTILITY(s)

    - if PLAYER(s) = MAX: Max_a EXPECTIMINIMAX(RESULT(s, a))
    - if PLAYER(s) = MIN: Min_a EXPECTIMINIMAX(RESULT(s, a))
    - if PLAYER(s) = CHANCE: Sum_r(P(r) * EXPECTIMINIMAX(RESULT(s,r)))

Introduction to Decision Making

Decision theory: choosing among actions based on the desirability of their immediate outcomes

utility function: assigns a single number to express the desirability of a state

expected utility: average utility value of the outcomes, weighted by the probability that the outcome occurs

maximum expected utility (MEU): a rational agent should choose the action that maximizes the agent's expected utility

The Basis of Utility Theory

- Notations:
    - $s \succ t$: state s is strictly preferred to state
    - $s \succsim t$: state s is at least as good as state t
    - $s \sim t$: agent is ambivalent between states s and t

If an agent's actions are not deterministic, we represent the set of outcomes for each action as a **lottery**. $L = [p_1, s_1; p_2, s_2; \ldots; p_n, s_n]$, where p_i will occurs with probability p_i

- Rules:
    - **Orderability**: Exactly one of$(A \succ B)$,$(B \succ A)$ or$(A \sim B)$ holds
    - **Transitivity**: $(A \succ B) \wedge (B \succ C) \Rightarrow (A \succ C)$
    - **Continuity**: $A \succ B \succ C \Rightarrow \exists p \text{ s.t.} [p, A; 1-p, C] \sim B$
    - **Substitutability**: $A \sim B \Rightarrow [p, A; 1-p, C] \sim [p, B; 1-p, C]$
    - **Monotonicity**: $A \succ B \Rightarrow (p > q \Leftrightarrow [p, A; 1-p, B] \succ [q, A; 1-q, B])$
    - **Decomposability**: $[p, A; 1-p, [q, B, 1-q, C]] \sim [p. A; (1-p)q, B; (1-p)(1-q), C]$

- A decision problem uder uncertainty is $< D, S, P, U >$
    - Set of decisions D
    - Set of outcomes S
    - Outcome function $P : D \rightarrow \Delta(S)$
        - $\Delta(S)$ is the set of **distributions** over S
    - Utility function $U : S \rightarrow R$
- A **solution** to a decision problem is any d^* in D such that $EU(d^*) \succsim EU(d)$ for all d in D

Goal: Pick action leads to the maximal expected utility.

- Problems:
  - Outcome space can be large
    - solution: use Bayes Nets
  - Decision space is large
    - solution: use dynamic programming to construct optimal plans

Policies

- **policy**: the form $[d_1, if\ s_i\ d_i, if\ s_j\ d_k, \dots]$
- **plan**: a sequence of decisions, $[d_i, d_k, d_j, \dots]$

Evaluating Policies

- Number of plans(sequences) of length k
  - **Exponential** in k: $|A|^k$, where A is the action set
- Number of policies is much larger, $(|A||O|)^k$, where A is action set, O is outcome set
- Use **dynamic programming**:
  - If $EU(a) > EU(b)$ at Si,
  - Never consider a policy that does anything else at S_i

Decision Trees

- Choce nodes (decision nodes)
- chance nodes, uncertainty regarding action effects
- Terminal nodes labelled with **utilities**

Evaluating a Decision Tree

- U(s) =
  - if TERMINAL_TEST(s): UTILITY(s)
  - if PLAYER(s) = CHANCE: avg{U(C): c is a child of s}
  - if PLAYER = CHOICE: max{U(c: c is child of s}}

Decision Tree Policies

A **policy** assigns a decision to each **choice node** in the tree

Two policies are **implementationally indistinguishable** if they disagree only on **unreachable nodes**

Savings compared to **explicit** policy evaluation is **substantial**

- Let $n = |A|$, $m = |O|$
  - need to evaluate only $O((nm)^d)$ nodes in tree of depth d
  - Evaluating a single policy requires O(m^d), O(n^d * m^2d) in total

- Issues:
    - Tree size: Grows **exponentially**
        - possible solutions: bounded lookahead, heuristic search procedures
    - Full Observability
        - possible solutions: Handcrafted decision trees, more general policies based on observations
    - Specification: Suppose each state is an assignment of values to variables
        - possible solutions: Bayes Net representations, decision network

Markov Decision Processes

SEQUENTIAL DECISION PROBLEMS

sequential decision problems: agent’s utility depends on a sequence of decisions. (Sequential decision problems incorporate utilities, uncertainty, and sensing, and include search and planning problems as special cases)

Transition model: $P(s'|s, a)$ to denote the probability of reaching state s' if action a is done in state s

Markov Property: the probability of reaching s from s depends only on s and not on the history of earlier states

utility function will depend on a sequence of states (an **environment history**)

utility of a given state sequence is the **sum of discounted rewards** obtained during the sequence

compare **policies** by comparing the **expected utilities** obtained when executing them

expectation is with respect to the probability distribution over state sequences de- termined by s and π.

- **Markov decision process (MDP)**: a sequential decision problem for a fully observable, stochastic environment with a Markovian transition model and additive rewards
- consists of:
    - a set of states with initial state
    - a set of ACTIONS(s) for actions in each state
    - a transition model P(s'|s, a)
    - a reward function R(s)
- a "solution" must specify what the agent should do for any state that the agent might reach, called **policy**. Denote a policy by π

executed starting from the initial state, the stochastic nature of the environment may lead to a different environment history. The quality of a policy is measured by the expected utility of the possible environment histories generated by that policy

An **optimal policy** is a policy that yields the highest expected utility. We use π* to denote an optimal policy. Given π*

Markov Chain

- Discrete **clock** pacing interaction of agent with environment, t = 0, 1, 2, ...
- Agent can be in one of a **set of states** S={...}
- **Initial state**
- **Markov Property**: The probability of the next state st+1 does not depend on how the agent got to the current state st
- Game is completely described by the **probability distribution of the next state given the current state**

Discounted Rewards

A reward in the future is **not worth as much as** a reward now (inflation or obliteration, ...)

Discount Factors: Used in economics and probabilistic decision-making

Discounted sum of future awards: Reward now + γ(reward in 1 time step) + γ2(reward in 2 time steps) + γ3(reward in 3 time steps) + ...

Markov System

- Set of **states** $S = \{s_1, s_2, \ldots, s_n\}$
- Each state has a reward
- **Discount factor** γ, $0 < \gamma < 1$
- Transition probability matrix, P
  - P is n*n matrix
  - $P_{ij} = Pr(Next = s_j | Current = s_i)$
- On each step:
  - get reward ri for state current state si
  - randomly move to state sj with probability Pij
  - All future rewards are discounted by γ
- **Solving a markov process**
  - U*(si) = expected discounted sum of future rewards starting at state s_i
    - $U_*(s_i) = r_i + \gamma(P_{i1}U_*(s_1) + P_{i2}U_*(s_2) + \ldots + P_{in}U_*(s_n))$
    - **Closed form**: $U = (I - \gamma P)^{-1} * R$

  - Advantage:
    - get an exact number
  - Disadvantage:
    - need to solve an n by n systems with n states
- **Value Iteration**:
  - $U^i(S_j)$ = Expected discounted sum of rewards over next i time step
  - $U^1(S_1) = r_i$
  - $U^2(S_i) = r_i + \gamma \sum_{j=1}^{n} p_{ij} U^1(s_j)$
  - $U^{k+1}(S_i) = r_i + \gamma \sum_{j=1}^{n} p_{ij} U^k(s_j)$
  - Note: As $k \to \infty$, $U^k(s_i) \to U^*(s_i)$
  - This is often **faster** than matrix inversion

Markov Decision Process

- Set of states S
- Each state has a reward
- Set of actions {a1,...}
- Discount facotr, 0 < γ < 1
- Transition probability function P
  - $P_{ij}^k = Prob(Next = s_j \mid This = s_i \wedge action = a_k)$
- On each step:
  - Get reward ri for current state s_i
  - **Choose action** a_k
  - Randomly move to state s_j with probability P_{ij}^k
  - All future rewards are discounted by γ

The goal of an agent in an MDP is **to be rational**

policy: a mapping from **states** to **actions**

For every MDP there exists an optimal policy

- $V^*(s_i)$: **expected discounted future rewards**
- $V^t(s_i)$: possible sum of discounted rewards I can get if I start at state s_i and live for t time steps
  - $V^1(s_i) = r_i$
- **Bellman's Equation**: $V^{t+1}(s_i) = max_k[r_i + \gamma \sum_{j=1}^{n}(P_{ij}^k V^t(s_j))]$
- Then do Value Iteration (aka. **Dynamic Programming**)
- Finding the Optimal Policy:

- Value Iteration
    - Compute $V^*(s_i)$ for all i using value iteration
    - Define the best action in state si as: $argmax_k[r_i + \gamma \sum_j P_{ij}^k V^*(s_j)]$
- Policy Iteration
    1. Start with random policy π
    2. Repeat
        1. Compute long term reward for each s_i using π
        2. For each state s_i
            1. If $max_k[r_i + \gamma \sum_j P_{i,j}^k V^*(s_j)] > r_i + \gamma \sum P_{i,j}^{\pi(s_i) V^*(s_j)}$ then $\pi(s_i) = argmax_k[r_i + \gamma \sum_j P_{i,j}^k V^*(s_j)]$

Bellman's Equation for utilities

the utility of a state is the immediate reward for that state plus the expected discounted utility of the next state, assuming that the agent chooses the optimal action.

utility of a state: U(s) = R(s) + γ∑_s' P(s'|s, a) U(s')

Called **Bellman equation**, they are the **unique** solutions

The value iteration algorithm

Bellman equation is the basis of the value iteration algorithm for solving MDPs. n Bellman equations if there are n states. It is a **nonlinear** equations, which is problematic to solve.

> ### value Iteration approach
>
> - loop
>   1. start with arbitrary initial values for the utilities, calculate the right-hand side of the equation
>   2. updating the utility of each state from the utilities of its neighbors
> - until reach equilibrium

$U^{i+1}(s) = R(s) + \gamma \max_a \sum_{s'} P(s'|s, a) U_i(s)$

If we apply the Bellman update infinitely often, we are guaranteed to reach an equilibrium. The final utility values must be unique solutions and are solutions to the Bellman equations.

the Bellman update is a contraction by a factor of γ on the space of utility vectors

Policy Iteration

Policy evaluation: given a policy πi , calculate $U_i = U\pi_i$, the utility of each state if π_i were to be executed

Policy improvement: Calculate a new MEU policy $\pi(i+1)$, using one-step look-ahead based on U_i

The algorithm terminates when the policy improvement step yields no change in the utilities

$U_i(s) = R(s) + \gamma \sum P(s'|s, \pi_i(s))U_i(s')$

these equations are linear, can be solved exactly in time O(n3) by standard linear algebra methods.

Partially Observable MDPS (POMDP)

The agent does **not** necessarily know which state it is in, so it cannot execute the action π(s) recommended for that state

utility of a state s and the optimal action in s depend not just on s, but also on **how much the agent knows** when it is in s.

- Definition:
    - transition model P (s' | s, a)
    - actions A(s)
    - reward function R(s)
    - Set of **observations** O={o1,...,ok}
    - Observation model P(ot | st)
    - **belief state**: the set of actual states the agent might be in
        - Probability **distribution** over all possible states
        - optimal action depends only on agent’s **current belief state**
        - write b(s) for the probability assigned to the actual state s by belief state b
- **Decision cycle**:
    - Given current **belief**, execute **action** a=π*(b)
    - Receive observation o
    - **Update** current belief state: b'(s') = aO(o|s')∑_s (P(s'|a, s)b(s)
        - a is a normalizing constant that makes the belief state sum to

the optimal policy can be described by a mapping π*(b) from belief states to action

POMDP belief-state space is **continuous**, because a POMDP belief state is a probability distribution. Finding optimal policies is difficult.

Reinforcement Learning

- **Goal**: to use observation to learn an optimal policy for the enviroment (without prior knowledge)
- Learner is not told what actions to take. Instead, discover value of actionsby trying and receiving feedback
- **Feedback**: reward and reinforcement
- **Utility-based agent**: learns a utility function on states and uses it to select actions that maximize the expected outcome utility
- **Q-learning agent**: learns an **action-utility** function or **Q-function** given the expected utility of taking a given action in a given state
- **Reflex agent**: learns a policy
- **Reinforcement Learning Model**:
  - Set of states S
  - Set of actions A
  - Set of **reinforcement signals** (rewards). Rewards may be delayed
- MDPs VS RL:
  - Goal of MDP is to find the optimal policy **given the model** (with rewards and transition probabilities)
  - RL is to find the optimal policy **without knowing the model** (No known rewards and transition probabilities)
- **Learning Task**:
  - Execute actions in the environment
  - Observe the result
  - **Learn policy** $\pi : S \rightarrow A$ that maximizes $E[r_t + \gamma r_{t+1} + \gamma^2 rt + 2 + \ldots]$ from any starting state in S **(Optimal Policy)**
- Types of RL
  - **Model-based** VS **Model-free**
    - Model-based: Learn the model of the environment
    - Model-free: Never explicitly learn $P(s'|s,a)$
  - **Passive** VS **Active**
    - Passive: Given fixed policy, evaluate it
    - Active: Agent must learn what to do
- **Passive learning**: agent's policy is fixed and the task is to learn the utilities of states
  - Agent's policy π is fixed (similar to policy evaluation)
  - Passive learning agent does not know the **transition model** $P(s'|s,a)$ and reward function $R(s)$

- Agents executes a set of trails in the environment using policy π
- Direct utility estimination: $V^*(S) = E[\sum_{t=0}^{\infty} \gamma^t R(S_t)]$.
- Utility of a state is the expected total reward from that state onward. At end of each sequence, the algorithm calculates the observed reward-to-go for each state and updates the estimated utility for the state accordingly.

- **Adaptive dynamic programming (ADP)**
  - Learn the transition model that connects the states and solving the corresponding MDP using a dynamic programming method.
  - **Modified policy iteration**: use a simplified value iteration process to update the utility estimates after each change to the learned model.
  - **Temporal DIfference**: Use observed transitions to adjust values of observed states. $V^\pi(s) = V^\pi(s) + \alpha(r(s) + \gamma V^\pi(s') - V^\pi(s))$ where α is the **learning rate**
    - **No explicit model** of P or R
    - Estimate V and expectation through samples
- **Active learning**: an agent must experience as much as possible its environment in order to learn how to behave in it.
  - **Exploration**: **randomly choose actions** to maximize its long-term well-being
  - **Exploitation**: **taking greedy actions** to maximize its reward
  - It needs to learn a utility function and the model in order to be able to choose via one-step look-ahead
- **Q-Learnning**: Learns an action-utility representation $Q : S, A \rightarrow R$ of the enviroment.
  - Optimal Policy: $\pi^* = argmax_a Q(s, a)$
  - $V^*(s) = max_a Q(s, a)$
  - model-free method
  - **Q-value**: $Q(s, a) = R(s) + \gamma \sum_{s'} P(s'|s, a) \max_a Q(s', a')$
- On-Policy / Off-Policy:
  - On-Policy: Behaviour = Learning
  - Off-Policy: Behaviour ≠ Learning
- **SARSA** (State-Action-Reward-State-Action)
  - On Policy
  - Agent learns policy being used, including exploration actions
  - $Q(s, a) = Q(s, a) + \alpha(R(s) + \gamma Q(s', a') - Q(s, a))$
- **Q-Learning** (Off-Policy)
  - Learned regardless of action chosen from exploring (agent follows a policy but learns the value of a different policy)
  - $Q(s, a) = Q(s, a) + \alpha(r + \gamma \max_{a'} Q(s', a') - Q(s, a))$
  - Q-Learning converges to the **optimal** Q-values if
    - Every state is visited **infinitely often**

        - The action selection becomes greedy as time approaches infinity
        - The learning rate is **decreased appropriately**

- Exploration Methods
    - Use an optimistic estimate of utility
    - Choose best action with probability p and a random action otherwise
    - Boltzmann exploration $P(a) = \frac{e^{Q(s,a)/T}}{\sum_{\alpha} e^{Q(s,a)/T}}$

Probability

- **Random variable**: describes an outcome that can not be determined in advance
- **Discrete random variable**: random variable with a countable domain
- **Event**: complete specification of the state of the world
- Events must be mutually **exclusive** and **Exhausitive**
- **P(A)**: **degree of belief** of the statement A is true
- **Probability Distribution**: A specification of a probability for each event in the sample space
- **Joint probability distribution**: Specification of probabilities for **all combination** of events
- **Unconditional** or **Prior Probabilities**: degrees of belief in propositions in the absence of any other information
- **Conditional** or **Posterior Probabilities**: $P(A|B)$ is probability of A given that B is true
- Axioms
    - $0 \leq P(A) \leq 1$
    - $P(True) = 1$
    - $P(False) = 0$
    - $P(A \vee B) = P(A) + P(B) - P(A \wedge B)$
- Useful Facts:
    - $P(A = v_i, A = v_j) = 0$ if $i \neq j$
    - $P(A = v_1 or A = v_2 or \ldots or A = v_k) = 1$ if the domain of A is $\{v_1, v_2, \ldots, v_k\}$
    - $P(A = v_1 or A = v_2 or \ldots or A = v_i) = \sum_{j=1}^{i} P(A = v_j)$
    - $P(B, (A = v_1 \vee A = v_2 \vee \ldots \vee A = V_i)) = \sum_{j=1}^{i} P(B, A = v_j)$
    - **Independence**: If X and Y are independent, $P(Y|X) = P(Y)$ or $P(X|Y) = P(Y)$ or $P(X, Y) = P(X)P(Y)$
        - In general: $P(X_1, X_2, \ldots, X_n) = \prod_{i}^{n}(X_i)$
    - **Marginalization**: $P(Y) = \sum_{z \in Z} P(Y, z)$
    - **Conditioning**: $P(Y) = \sum_{z \in Z} P(Y|Z)P(Z)$
    - **Chain Rule**: $P(A \wedge B) = P(A|B)P(B)$

- **Bayes' Rule**: $P(B|A) = \frac{P(A|B)P(B)}{P(A)}$
- **Naive Bayes**: $P(Cause, Effect_1, \ldots, Effect_n) = P(Cause) \prod_i^n P(Effect_i|Cause)$

- **Conditional independence**
  - **Full independence** is often too strong a requirement
  - Two variables A and B are **conditionally independent** given C and $P(a|b,c) = P(a|c)$ for all a, b, c. **Knowing the value of B does not change the prediction of A if value of C is known**
- **Probabilistic Inference**: the computation of posterior probabilistic for query propositions given observed evidence.

Bayes Networks

- **Bayesian network**: represent the dependencies among variables, can represent essentially any ful joint probability distribution and in many cass can do so cousely
  - Each node corresponds to a random variable
  - Direct links $X \to Y$: X is parent of Y and X has a **direct influence** on Y
  - No directed cycles
  - each X_i has conditional probability distribution $P(X_i|Parent(X_i))$ that quantifies the effect of the parent on the node
  - Constructing a BN
    1. Take any ordering of the variables and go through the following procedures for X_n down to X_1
    2. Let $Par(X_n)$ be any subset of $S \subseteq \{X_1, \ldots, X_{n-1}\}$ such that X_n is independent of $\{X_1, \ldots, X_{n-1}\} - S$ given S, such a subset must exist
    3. Then determine the parents of X_{n-1} in the same way, finding a similar $S \subseteq \{X_1, \ldots, X_{n-2}\}$ and so on. In the end, a DAG is produced and the BN semantics must hold by construction.
- **Conditional Probability Table (CPT)**: each row in a CPT contains the conditional probability of each node value for a conditioniong case.
- A **Bayesian network** is a directed acyclic graph with some numeric parameters attached to each node. $P(x_1, \ldots, x_n) = \prod_{i=1}^n \Theta(x_i|parents(X_i))$. Each entry in the joint distribution is represented by the product of the appropriate elements of the CPTs in the Baysian Network
- **D-separation**: A set of variables E d-separates X and Y if it **blocks** every undirected path between X and Y.
- X and Y are **conditionally independent** given E if E d-separates X and Y.
- Evidence set E **blocks** path P iff there is some node in Z on the path such that
  - one arc on P goes into Z and one goes out of Z and Z in E
  - both arcs on P leave Z and Z in E

- both arcs on P enter Z and neither Z nor its descendents are in E

- Other ways of determning conditional independence:
  - **Non-descendents**: A node is conditionally independent of its non-descendents given its parents.
  - **Markov Blanket**: A node is conditionally independent of all other nodes given its parents, children and children's parents

- Inference in Bayes
  - Goal: compute the posterior probability distribution for a set of **query variables** given some observed **event** (some assignment of **evidence variables**)

- **Variable Elimination**: evaluating expression in right-to-left order. Intermediatee results are stored and summations over each variable are done only for those portions of the expression that dependent on the variables

- A function $f(X_1, \dots, X_k)$ is called a **factor**. Can be viewed as a table of numbers, one for each instantiation of the variables. Exponential in k.

- Each **CPT** in BN is a factor

- **Product** of two factor: $h(X, Y, Z) = f(X, Y) * g(Y, Z)$
- **Summing** a variable out: $h(Y) = \sum_{x \in X} f(x, Y)$
- **Restrict** a factor to X=x by setting X to the value x and deleting. $h(Y) = f(x, Y)$
- Variable Elimination:
  - Write out computation using chain rule and exploiting independence relations in networks
  - Arrange terms in convenient fashion
  - Distribution each sum
  - Apply operations inside out

- Complexity of variable elinimation is **linear** in number of variables and exponential in size of the largest factor

Hidden Markov Models

- **Stochastic Process**:
  - Problems: Infinitely many variables and infinitely large CPTs
  - Solution:
    - **Stationary process**: Dynamics do not change over time
    - **Markove assumption**: Current statedepends only on finite history of past states.

- View the world as a series of snapshots or **time slices** each contains a set of random variables.

- S_t to denote the set of states at time t.

- **k-Order Markov Process**
    - **Markov Assumption**: The current state depends on only a **finite fixed number** of previous states
    - **first-order Markov process**: the current state depends only on the previous state and not on any earlier states. $P(s_t|s_{t-1},...,s_0) = P(s_t|s_{t-1})$
    - **k-Order**: $P(s_t|s_{t-1},...,s_0) = P(s_t|s_{t-1},...,s_{t-k})$
    - Advantage: Can specify the entire process using finitely many time slices
    - Problem: uncertainty increases over time
- **First Order Hidden Markov Model (HMM)**:
    - Set of states: S
    - Set of observation: O
    - Transition model: $P(s_t|s_{t-1})$
    - **Observation model**: $P(o_t|s_t)$. The evidence(observation) variable could depend on previous variables as well as the current state variables.
    - Prior: $P(s_0)$
- Common Tasks:
    - **Monitoring**: $P(s_t|p_t,...,o_1)$. interested in distribution over **current states** given observation.
    - **Prediction**: $P(s_{t+k}|o_t,..,o_1)$. Interested in distributions over **future states** given observations
    - **Hindsight**: P(s_k|o_t,..., o_1). Interested in distribution over a **past state** given observation
    - **Most likely explanation**: $argmax_{s_t,...,s_1} P(s_t,...,s_1|o_t,...,o_1)$. Interested in the **most likely sequence** of states given the observations.
- **Dynamic Bayes Nets**
    - Idea: Encode states and observations with **several random variables**
    - Advantage: Exploit **conditional independence** and save time
    - Note: HMMs are just DBNs with one state variable and one observation variable
    - Complexity: Inference tends to be exponential in number of state variables
- Solution for **Non-Stationary Process**: Add new state components until dynamics are stationary
- Solution for **Non-Markovian Processes**: Add new state components until the dynamics are Markovian

Decision Networks （Influence Diagrams)

- Goal: A decision network represents information about the agent's current state, its possible actions, the state that will result from the agent's action, and the utility of that state
- **Random variables** （Circles): Or chance node, Same as in BN. Each node associated with it a conditional distribution that is indexed by the state of the parents nodes.

- **Decision nodes**(rectangles): represent points where the decision maker has a choice actions. Parents reflect **information available** at time of decision.
- **Utility nodes**(diamonds): or value nodes, represent the agent's utility function. The utility node has as parents all variables describing the outcome that directly afffect utility.
- Assumptions:
    - Decision nodes are **totally ordered**. (decisions are made in sequence)
    - **No forgetting property**: Any information available for previous decision maker are available for the later decisions.
- **Policies**:
    - Let $Par(D_i)$ be the parents of decison node D_i. $Dom(Par(D_i))$ is the set of assignments to $Par(D_i)$
    - A **policy** δ is a set of maping δ_i one for each decision node D_i
    - Given assignment x to random variables X, $\delta(x)$ be the assignment to decison variables ditated by δ .$EU(\delta) = \sum_x P(x, \delta(x)U(\delta(x)))$
- Computing the **optimal ploicy**
    - compute from backwards
    - For each assignment to parents, and for each decision value, compute the expected value of choosing that value
    - Set policy choice for each value of parents to be the value of D that has maximal value
    - Treate the later decision variable as random variable (since they are computed and EU is fixed)
- Computing **Expected Utility**
    - Utility nodes are just factors that can be dealt using variable elimination
- If decision node D has no decisions that follow it, we can find its policy by instantiating its parents and computing the expected utility for each decision given parents

Game Theory

- **MAS:** Multiagent Agent System
- **Self-interested MAS**: Agents have their own description of states of the world. Agents take actions based on these descriptions.
- **Game Theory**: formal way to analyze **interactions** among a **group** of **rational** agents that behave **strategically**.
    - **Interaction**: What one agent does directly affects at least one other
    - **Strategic**: Agents take into account that their actions influence the game
    - **Rational**: Agents chose their best actions
- **Strategic Game**:
    - Set of **agents**: $I = \{1, 2, \ldots, N\}$
    - Set of **actions**: $A_i = \{a_i^1, \ldots, a_i^m\}$. A **strategy profile** is an assignment of a strategy to

each p
  - Outcome of a game is defined by a profile $a = (a_1, \ldots, a_n)$ and a game's outcome is a numeric value for each player.
  - Agents have **preferences** over outcomes (Utility functions $u_i : A \rightarrow R$)
- **Best response**: $a_i = argmax \sum_{a-i} u_i(a_i, a_{-1}) p_i(a_{-1})$. Where $a_{-i} = (a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n)$
- We say that a strategy s for player p **Stictly dominates** strategy s' if the outcome for s is better for p than the outcome for s' for every choice of strategies by the other player(s). $u_i(a'_i, a_{-1}) > u_i(a_i, a_{-i}) \forall a_{-i}$
  - A rational agent will never play a dominated strategy
- **Equilibrium**
  - An equilibrium is essentially a **local optimum** in the space of policies.
  - Every game has at least one equilibrium (Proved by John Nash)
  - **Nash equilibrium**: when there is no agent has incentive to change given that others do not change. $\forall i, u_i(a^*_i, a^*_{-i}) \geq u_i(a'_i, a^*_{-i}) \forall a'_i$
- **Extensive Form Games** $G = (I, A, H, Z, \alpha, \rho, \sigma, u)$
  - I: player set
  - A: action space
  - H: non-terminal choice nodes
  - Z: terminal nodes
  - α: action function $\alpha : H \rightarrow 2^A$
  - ρ: player function $\rho : H \rightarrow N$
  - σ: successor function $\sigma : H * A \rightarrow H \cup Z$
  - $u = (u_1, \ldots, u_n)$ where u_i is a utility function $u_i : Z \rightarrow R$
- **Subgame perfect equilibrium**: A strategy profile is a subgame perfect equilibrium if it represents a Nash equilibrium of every subgame of the original game. Computed by using backward induction.

Mechanism Design

- Goal:
  - Understand the ways in which agents interact and behave
    Design systems so that agents behave the way we want them to.
- **Mechanism Design**:
  - Given rational agents, what sort of game should we design
  - Can we guarantee that agents will reach an outcome with properties we want
  - **To design a game whose solutions, consisting of each agent pursuing its own rational strategy, result in the maximization of some global utility function**
- **Fundamentals**

- Set of possible **outcomes**: O
- Set of agents: N, $|N| = n$
  - Each agent has a type θ_i from Θ_i
  - The type captures all private information relevant to the agent's decision making
- Utility functions: $u_i(o, \theta_i)$
- Social choice function: $f : \theta_1, \ldots, \theta_n \to o$

- **Mechanism** $M = (S_1, \ldots, S_n, g(\cdot))$
  - S_i is the **strategy space** of agent i
  - $g : S_1, \ldots, S_n \to O$ is the **outcome function**
  - A mechanism M implements the **social choice function** $f(\Theta)$ if there is an **equilibrium** $s_* = (s_1^*(\theta_1), \ldots, s_n^*(\theta_n))$ such that $g(s_1^*(\theta_1), \ldots, s_n^*(\theta_n)) = f(\theta_1, \ldots, \theta_n)$ for all $(\theta_1, \ldots, \theta_n) \in \Theta_1 * \Theta_n$
  - A **mechanishm** consists of (1) a language for describing the **set of allowable strategies** that agents may adopt (2) a distinguished agent, called **center** that **collects reports of strategy choices** from the agetns (3) an **outcome rule** known to all agents that to uses to determine the payoffs to each agetns given strategy
  - **Direct Mechanisms** if $S_i = \Theta_i, \forall i$ and $g(\theta) = f(\theta), \forall \theta \in \Theta_1 * \Theta_n$
  - **Dominant Strategy** is a strategy works against all other strategies, which in turn means that an agent can adopt it without regard for the other strategies.
  - A direct mechanism is **incentive compatiable** if it has an equilibrium s^* where $s^*(\theta_i) = \theta_i$ for all θ_i for all i. **Incentive compatible means that you have a direct mechanism where there is an equilibrium such that all agents truthfully reveal their values/private information.**
  - A mechanism where agents have a dominant strategy is called a **strategy-proof** mechanism (or dominant strategy equilibrium).
  - **Truth-revealing** or **truthful** is strategy involves the agent revealing their true value.
  - **Revelation principle** states that any mechanism can be transformed into an equivalent truth-revealing mechanism.
  - **Theorem**: Suppose there exists a mechanism M that implements social choice function f in dominant strategies. Then there is a **direct strategy-proof mechanism** M' which also implements f.

- **Gibbard-Scatterthwaite Theorem**: Assume that O if finite and $|O| > 2$, each o in O can be achieved by Social choice function 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**

- **Median-Voter rule** is strategy proof for single-peaked preferences.

- **Quasilinear Preferences**:
  - **Outcome** $o = (x, t_1, \ldots, t_n)$, where x is a project choice, $t_i \in R$ are transfers (money)
  - Utility functions: $u_i(o, \theta_i) = v_i(x, \theta_i) - t_i$
  - **Quasilinear mechanism** $M = (S_1, \ldots, S_n, g())$ where $g() = x(), t_1, \ldots, t_n)$

- **Vickrey-Clarke-Groves**

- Goal: want to maximize the global utility $V = \sum_i v_i$ without tragedy of commons
- **Tragedy of the commons**: if nobody has to pay for using a common resource, then it tends to be exploited in a way that leads to a lower total utility for all agents.
- **VCG** is a dominant strategy for each agent to report its true utility and achieves an efficient allocation of the goods. Each agent pays a tax equivalent to the loss in global utility that occurs because of the agent's presence in the game.
- **Choice rule** $x^*(\theta) = argmax_x \sum_i v_i(x, \theta_i)$
- **Transfer rules** $t_i(\theta) = h_i(\theta_{-i}) - \sum_{j \neq i} v_j(x^*(\theta), \theta_j)$, where h_i is some function based on reports of everyone else and $\sum$ is the sum of how much everyone else values this outcome.

Machine Learning

- **Machine Learning**: A computer program is said to **learn** from **experience** E with respect to some class of **tasks** T and **performance measures** P if its performance at tasks in T, as measured by P, improves with experience E.
- Type of Learning
  - **Supervised Learning**: learn a function from examples of its **inputs and outputs** (SVM, decision tree, Neural Nets)
  - **Unsupervised Learning**: learn **patterns** in the input when no specific output is given (Clustering, KNN)
  - **Reinforcement learning**: Agents learn from **feedback**, rewards or punishments (TD-Learning, Q-learning)
  - **Simi-Supervised Learning**
  - **Active Learning**
  - **Transfer Learning**
  - **Apprenticeship Learning**
- Representation
  - Linear weighted polynomials
  - Propositional logic
  - First order logic
  - Bayes nets
  - ...
- **Supervised Learning: Hypothesis Learning and Representation**
  - Any hypothesis found to approximate the target function well over a **sufficiently large set of training examples** will also approximate the target function well over any **unobserved** examples
  - Prefer the simplest hypothesis consistent with the data
  - Possibility of finding a single consistent hypothesis depends on the hypothesis space
- **Decision Trees**

  - Classify instances by sorting them down the tree from root to leaf. Start at root, test attribute by each node until reach a leaf
  - **Fully expressive within the class of propositional language**
  - Learning Decision Tree (DTL) input: examples, attributes

```
1. if examples have the same classes, return leaf
2. if attributes is empty then return MODE(example)
3. best = ChooseAttribute(attributes, examples)
4. tree = new decision tree with root test best
5. for each $v_i$ of best:
    1. examples = {elements of examples with $best = v_i$}
    2. subtree = DTL(examples, attributes - best, MODE(examples))
    3. add a branch to tree with label $v_i$ and subtree *subtree*
6. return tree
```

- **Information Theory**
  - **Entropy**: $I(P(v_1), \ldots, P(v_i)) = \sum_{i=1}^{n} -P(v_i) log_2 P(v_i)$ a measure of uncertainty where approaching 0 is certainty and approaching 1 is uncertainty
    - For training set containing p positive and n negative: $I(\frac{p}{p+n}, \frac{n}{p+n}) = -\frac{p}{p+n} log_2 \frac{p}{p+n} - \frac{n}{p+n} log_2 \frac{n}{p+n}$
  - **Remainder**: $remainder(A) = \sum_{i=1}^{v} \frac{p_i+n_i}{p+n} I(\frac{p_i}{p_i+n_i}, \frac{n_i}{p_i+n_i})$ weighted average of the entrophy left after an attribute test.
  - **Information Gain(IG)** $IG(A) = I(\frac{p}{p+n}, \frac{n}{p+n}) - remainder(A)$ or reduction in entrophy from the attribute test
- **Assessing Performance**
  - good if the hypothesis predict **unseen examples** well.
  - Devide all data into two disjoint sets: **training set** and **test set**
  - **NO PEEKING AT THET TEST SET**
  - **Overfitting**: Finding patterns in the data where there is no actual pattern
    - Given a hypothesis space H, a hypothesis h in H is said to overfit the training data if there exists some alternative hypothesis h' in H such that **h has smaller error** than h' on the training examples, but **h' has smaller error** than h over the entire distribution of instances. $error_{Training}(h) < error_{Training}(h')$ and $error_{Test}(h') < error_{Te}(h)$
  - Avoid Overfitting: **Pruning**
    - Assume there is **no pattern** in the data (null hypothesis)
    - Compute probability that a sample size p+n would **exhibit observed deviation**
  - **Cross Validation**

        - Split the training set into two parts, one for training and one for choosing the hypothesis with highest accuracy
        - **K-fold cross validation**: run k experiments, each time puttting aside 1/k of the data to test on
        - **Leave-one-out cross validation** n = k
    - **Ensemble learning** using many hypothesis from hypothesis space and combime their predictions
        - Elections/Committees
    - **Bagging**: Train different model with same learning algorithm but different set of training example, and combime the predictions.
    - **Boosting**: Increased the chance to be trained on the misclassified examples.
        - No need to learn a perfect hypothesis
        - boost any weak learning algorithm
        - Easy to program
        - Good generalization

Artificial Neural Networks

- Goal: Design methods for learning **arbitrary** relationships and ensure the mothods are **efficient** and do not **overfit**
- **A neuron**:
    - Dendrities: Receive inputs
    - Soma: Controls activity of the neuron
    - Axon: Sends output to other cells
    - Synapse: Links between neurons
- **Artificial Neuron**: Link, Weight, Input, Activation Fun, Output
- **Artificial Neural Nets**:
    - Collection of simple artificial neurons
    - **Weights** $W_{i,j}$ denote strength of connection from i to j
    - Input function: $in_i = \sum_j W_{i,j} * a_j$
    - **Activation** Function: $a_i = g(in_i)$
        - Activation function should be **non-linear**
        - Common Activation Functions
            - **Rectified Linear Unit (ReLU)**: $g(x) = max\{0.x\}$
            - **Sigmoid** Functions: $g(x) = \frac{1}{1+e^x}$
            - **Hyperbolic Tangent**: $g(x) = tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$
            - **Threshold Function**: $g(x) = 1$ if $x \geq b$ else 0

- **Logic Gate**: It is possible to construct a universal set of logical gates using the neurons described
- Structure
  - **Feed-forward ANN**: Direct **acyclic** graph, no internal state
  - **Recurrant ANN**: Directed cyclic graph, dynamical system with an internal state. Able to **remember** information for future use.
- **Perceptron**: Single layer feed-forward network. (For detail, see CS489 Note)
  - Training: $w_j = w_j + \alpha(l(y_i) \cdot g'(I) \cdot x_{i,j}) \forall j$. whre α is learning rate and $l(y)$ is loss function and I is prediction
- **Multilayer Neural Networks**: Any continuous function can be learned by an ANN with just one hidden layer
  - Training using **Gradient Descent** $\Delta_i = E \cdot g'(I)\, W_{j,i} = W_{j,i} + \alpha\Delta_i a_j$
  - **Back Propagation**
    - Idea: Each hidden layer caused some of the error in the output layer, amount of error caused is proportionate to the connection strength
      - Compute Deltas and weight change for output layer, update the weights, repeat untill all hidden layers updated
  - **Deep Learning** refers to neural networks with more than one hidden layers. Single hidden layer tends to overfit
  - When to use ANNs:
    - have high dimensional or real-valed inputs with noise
    - Form of target function is unknown (no model)
    - Not important for human to be able to understand the mapping
  - Drawbacks of ANNs:
    - Unclear how to interpret weights
    - How deep/ how many neurons: 玄学
    - Tendency to overfit

Statistical Learning

- Goal: Build models of the world directly from data. Focus on learning models for probabilitic models.
- **Bayesian Learning**
  - Prior: $P(H)$
  - Likelihood: $P(d|H)$
  - Evidence: $d = (d_1, d_2, \ldots, d_n)$
  - Compute the probability of each hypothesis given the data
  - $P(x|d) = \sum_i P(x|d, h_i)P(h_i|d) = \sum_i P(x|h_i)P(h_i|d)$

  - Predictions are **weighted averages** over the predictions of all the individual hypothesis.
  - Problem: **Intractable** if hypothesis space is large
- **Maximum a posteriori (MAP)**
  - Make prediction on the **most probable** hypothesis h_{MAP}
  - $h_{MAP} = argmax_{h_i} P(h_i|d)$ $P(x|d) = P(x|h_{MAP})$
  - Less accurate than Bayesian prediction
  - MAP and Bayesian predictions converge as data increases
  - **No overfitting**
  - Find h_{MAP} may be intractable
- **Maximum Likelihood (ML)**
  - Idea: simplify MAP by assuming **uniform prior**
  - Subject to **overfitting**
  - $h_{ML} = argmax_j \sum_i logP(d_i|h_j)$
  -