1 Artificial Intelligence

- An **agent** is an entity that can perceive and act. This course is about designing rational agents.
- Rational behavior: doing the right thing.
- Environment Types: Fully observable; Deterministic; Episodic; Static, Discrete; Single-agent. The counter part: partially observable; stochastic; sequential; dynamic; continuous; multi-agent.
- An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators.
- Being rational means maximizing your **expected utility**. And a better title for this course would be **Computational Rationality**.
- **Rational:** maximally achieving pre-defined goals.
- **Rationality:** only concerns what decisions are made (not the thought process behind them)
- **Utility:** Goals are expressed in the terms of the utility of outcomes. And CS188 thinks that being rational means maximizing your expected utility.
- **Automation:** Applied AI involves many kinds of automation. Scheduling, route planning, medical diagnosis, web search engines, spam classifiers, automated help desks, fraud detection, product recommendations. "Did any one of these remind you of subtopics and projects in Machine Learning?"
- **Agent:** An agent is an entity that perceives and acts. A rational agent selects actions that maximize its *expected* utility.
- Making decision, reasoning under Uncertainty, and their applications.

2 Problem Solving

- A search problem consists of
  - a state space
  - a successor function (namely **update function** in data mining algorithm series, more namely **recursion** in bullshit technology.)
  - a start state (**initial value**), goal test (**terminating value**) and path cost function (**we say weights in Graph Theory**)
  - Does any one of the above reminds you of **recursion**?
- Problems are often modelled as a state space, a set of states that a problem can be in. The set of states forms a graph where two states are **connected** if there is an operation that can be performed to transform the first state into the second.
- A solution is a sequence of actions (a plan) which transforms the start state to a goal state.
- **State space graph**: A mathematical representation of a search problem.
- **Search Trees**
  - This is a "what-if" tree of plans and outcomes
  - For most problems, we can never actually build the whole tree
- **General Tree Search** Frontier; Expansion; Exploration Strategy.
- **States vs. Nodes** Nodes in state space graphs are problem states; Nodes in search trees are plans. The same problem state may be achieved by multiple search tree nodes.
- **Graph Search** Graph Search still produces a search tree; Graph search is almost always better than tree search.
- DFS graph search needs to store "explored set", which is $O(b^m)$. However, **DFS is optimal** when the search tree is finite, all action costs are identical and all solutions have the same length. However limitating this may sound, there is an important class of problems that satisfies these conditions: the CSPs (constraint satisfaction problems). Maybe all the examples you thought about fall in this (rather common) category.
- The breadth first search and iterative deepening are conceptually and computationally the same. The only difference is the "space" (we call them **memory**) would be partially saved by iterative deepening search.
- **Heuristics** estimate of how close a node is to a goal; Designed for a particular search problem.
- **A star search** Uniform-cost orders by **path cost**, or backward cost $g(n)$; Best-first orders by **distance** to goal, or forward cost $h(n)$. A* Search orders by the sum: $f(n) = g(n) + h(n)$. The distance is an estimated one.
- When A* terminates its search, it has, by definition, found a path whose actual cost is lower than the estimated cost of any path through any node on the frontier. But since those estimates are optimistic, A* can safely ignore those nodes.
- In general, most natural admissible heuristics tend to be consistent, especially if from relaxed problems.
- Types of agents: reflex agents, planning agents (optimal vs. complete planning).

2.1 Uninformed Search

- **State Space:** A state space is also an abstraction of the world. A successor function **models** how your world works (namely, evolves and response to your actions).
- **Search Problems:** They are just models, aka, no more than models in the mathematical sense.
- **World State:** Includes every last detail of the environment.

- **Search State:** Keeps only the details needed for planning (namely, abstraction). Because only with abstraction can we solve problems smoothly.
- **Search Trees:** For most problems, we can never actually build the whole tree. (So we **ignore** most of the tree.)
- **Complete:** Guaranteed to find a solution if one exists? **Optimal:** Guaranteed to find the least cost path?
- **DFS vs BFS:** When will one outperform the other?
- **Uniform Cost Search:** Expand a cheapest node first. Thus fringe is a **priority queue**. (priority, cumulative cost, namely, add them up!) Therefore it's complete and optimal! But it explores options in every "direction". And this algorithm shows **no information** about goal location.
- Search operates over **models** (namely, abstractions) of the world. Planning is all "in simulation", therefore your search is only as good as your model is.

2.2 Informed Search

- **Informed Search:** Inject information of where the goal might be. Key idea: Heuristics.
- **Successor Function:** If I do this, what will happen, in my model.
- **Search Heuristics:** Something tells you that whether you are getting close to the goal, or not. It's a function that *estimates* how close a state is to a goal. It's designed for a particular search. Examples: Manhattan distance, euclidean distance. (They are not perfect, but they are *at least* something.)
- **Greedy Search:** A common case: best-first takes you straight to the (wrong) goal.
- **A* Search:** Revised. Combine both UCS and Greedy, namely, tortoise and rabbit. Uniform-cost orders by path cost, or backward cost. Greedy orders by goal proximity, or forward cost.
- **A* Search:** Stop when you **dequeue** a goal from the fringe. Lesson: We need estimates to be less than actual costs.
- **Admissibility:** Admissible (optimistic) heuristics slow down bad plans but never out-weight true costs. Inadmissible is just a fancy name for, **pessimistic**, it traps good plans on the fringe.
- A heuristic **h** is **admissible** (optimistic) if:

$$0 \leq h(n) \leq h^*(n) \tag{1}$$

  where $h^*(n)$ is a true cost to a nearest goal. Thus coming up with admissible heuristics is most of what's involved in using A^* in practice. A^* is not problem specific, but your heuristic is.
- **Crating Admissible Heuristics:** Most of the work in solving hard search problems optimally is in coming up with admissible heuristics. Often, admissible heuristics are solutions to **relaxed problems**, where new actions are available.
- **Graph Search:** For tree search, if it fails to detect repeated states can cause exponentially more work. Idea: **never expand** a state twice.
- Important: (in python's idea) store the closed set as a set, not a list. In Lisp's concept, make it a hash table (it is verified, just use hash table in Lisp).
- **Consistency of Heuristics:** real cost should be larger or equal than cost implied by heuristic. (Namely, please be **Conservative**, aka guess "smally" rather than "biggerly".) Implication: f value along a path never decreases.
- **Optimality:** For tree search, requires heuristic admissible; for graph search, requires consistent. And consistency implies admissibility.
- **Heuristics:** The design of this number (function) is key, often use **relaxed problems**.

2.3 Constraint Satisfaction Problems

- **Search:** a single agent, deterministic actions, fully observed state, discrete state space.
- **Planning:** a sequence of actions. The **path** to the goal is the important thing.
- **Identification:** assignments to variables. The goal itself is important, not the path.
- **CSP:** a special subset of search problems. State is defined by **variable** X_i with values from a domain D (sometimes D depends on i). Goal test is a set of constraints specifying allowable combinations of values for subsets of variables.
- CSP allows useful general-purpose algorithms with more power than standard search algorithms. (Namely, add more "rules", walk through (traverse) less paths.)
- **CSP Varieties:** Discrete variables; continuous variables.
- **Varieties of Constraints:** Unary; Binary; Higher-order constraints. Or Preferences (soft constraints).
- **Backtrack Search:** The basic uninformed algorithm for solving CSPs. Namely, recursion. One variable at a time; check constraints as you go (Online shit? Incremental goal test). So backtracking is equal to DFS add variable ordering and add fail-on-violation.
- **Improve Backtracking:** Ordering; Filtering; Structure.
- **Filtering:** Keep track of domains for unassigned variables and cross off bad options. Namely, build a mathematical filter. Namely, ask (cond, else) when doing forward checking.
- **Forward Checking:** Enforcing consistency of arcs pointing to each new assignment.
- **Arc Consistency:** It still runs inside a backtrack search.
- **Ordering:** Minimum Remaining Values. Variable ordering, **always** choose the variable with the **fewest** legal left values in its domain, given a choice of variables.

- What the hell is CSP? Variables; Domains; Constraints—Implicit, Explicit, Unary/Binary/N-ary. Goals: find any solution; find all; find best, etc.
- **K-Consistency:** For each k nodes, any consistent assignment to $k-1$ can be extended to the k^{th} node.
- Suppose a graph of n variables can be broken into subproblems of only c variables. Example: $n = 80, d = 2, c = 20$. But this "crap" is somehow impractical.
- **Tree-Structured CSPS:** Theorem, if the constraint graph has no loops, the CSP can be solved in $O(nd^2)$ time. For general CSPs, worst case is $O(d^n)$. This also applies to probabilistic reasoning: an example of the relation between syntactic restrictions and the complexity of reasoning.
- **Nearly Tree-Structured CSPs:** Cutset conditioning: instantiate (in all ways) a set of variables such that the remaining constraint graph is a **tree**.
- *Sorry this is in ai class, everything is hard.—CS188*
- **Tree Decomposition:** Create a tree-structured graph of mega-variables. Each mega-variables encodes part of the original CSP.
- CSPs are a special kind of search problems where states are partial assignments and goal test is defined by constraints. The basic solution is backtrack search.
- **Local Search:** (yet another fancy name of EM algorithm.) It improves a single option until you can't make it better. (**No fringe!**)
- Generally local search is much faster and more memory efficient. But it is also **incomplete and suboptimal**.
- **Hill Climbing:** Simple general idea—Start wherever, repeat: move to the best neighboring state; if no neighbors better than current, quit.
- **Simulated Annealing:** Idea, escape local maxima by allowing downhill moves.
- The more downhill steps you need to escape a local optimum, the less likely you are to ever make them all in a row. Therefore people think hard about ridge operators which let you jump around the space in better ways.
- **Genetic Algorithms:** It uses a natural selection metaphor—keep best N hypotheses at each step based on a fitness function; Also have pairwise crossover operators, with optional mutation to give variety.

2.4 Adversarial Search

- **Meaning:** How to decide how to act, when there is an adversary in "your world (model, abstraction, etc.)".
- Monte Carlo methods are just a fancy name for **randomized** methods.
- **Pacman:** Behavior from **Computation**.
- **Axes:** Deterministic or stochastic? One, two or more players? Zero sum? Perfect information (can you see the state)?
- For this course, we want algorithms for calculating a **strategy (policy)** which recommends a **move** from each state.
- Different from search: we do not **control** our opponent. We need to give out **policies**.
- One possible formalization is: States, Players, Actions, Transition Function $S \times A \to S$, Terminal Test $S \to \{t, f\}$, Terminal Utilities $S \times P \to R$.
- Players usually take turns; Actions may depend on player/state; Terminal utilities tells us how much it's worth to **each of the players**.
- **Zero-Sum Games:** Let us think of a single value that one maximizes and the other minimizes.
- **General Games:** Agents have independent utilities. Cooperation, indifference, competition, and more are all possible.
- **Value** of a state: The **best** achievable outcome (utility) from that state.
- **Minimax Values:** States Under Opponent's Control: $V(s') = min\ V(s)$ States Under Agent's Control: **Maximize** out of all possible "worst" results your **opponent** offers. In choosing universities and advisors, pick out the "tallest" guy from the "small man".
- In other words, life is much much worse when there is an (or more than one) adversary. I want the "global maximum", but the adversary just **won't let it happen**.
- **Minimax Search:** A state-space search tree; players alternate turns; compute each node's **minimax value**, namely the best achievable utility against a rational (optimal) adversary.
- Ask this question to yourself: do we **really** need to explore the whole tree?
- **Resource Limits:** In realistic games, cannot search to leaves. Solution: **Depth-Limited Search**. Search only to a limited depth in the tree, and replace terminal utilities with an evaluation function for non-terminal positions.
- **Depth Matters:** An important example of the tradeoff between complexity of features and complexity of computation.
- **Evaluation Functions:** In practice, typically weighted linear sum of features.
- **Game Tree Pruning:** Look at the trees that do not have to be minimized.
- **Alpha-Beta Pruning:** Key idea it symmetric. To sum up, it's already **bad enough** that it won't be played. α MAX's best option on the path to root. β MIN's best option on path to root. Tip: You have to be right for the children of the route. Therefore good child ordering improves effectiveness of pruning.

3 Uncertain Knowledge and Reasoning

3.1 Expectimax and Utilities

- Uncertain outcomes controlled by **chance**, not an adversary!
- Values should now reflect average-case (expectimax) outcomes, not worst-case (minimax) outcomes. (Explicit randomness, Unpredicable opponents, Actions can fail).
- **Expectimax search:** compute the average score under optimal play. Key idea: Calculate their **expected utilities**.
- For average-case expectimax reasoning, we need **magnitudes** to be meaningful. Not only order matters, magnitudes as well.
- As we get more evidence, probabilities may change.
- **Random variable** represents an event whose outcome is unknown.
- **Probability distribution** is an assignment of **weights** to outcomes. Note, in functional programming, we can also say that it's a **mapping** of weights to outcomes.
- The **expected value** of a function of a random variable is the **average**, weighted by the probability distribution over outcomes.
- Having a probabilistic belief about another agent's action **does not** mean that the agent is flipping any coins!
- Based on **how we think** the world works, what computation we should do. Are our opponents adversarial or by chance?
- **Minimax** generalization:
  - Terminals have utility tuples (namely, lists)
  - Node values are also utility tuples
  - Each player maximizes its own component
  - Can give rise to **cooperation** and **competition** dynamically.
- **Vampire Bunnies**
- Worst case reasoning only works to the extent that your model is sufficiently simple. Namely, **minimax** is just a **binary** "crap", though the world can be "simulated" based on asking **yes or nos**, it is still too young too simple, sometimes naive, naive¡'Yo you may be hit by a METEOR!!"
- Tip: A rational agent should choose the action that maximizes its expected utility, **given its knowledge**.
- **Utilities** are functions from outcomes (states of the world) to real numbers that describe an **agent's preferences**.
- We hard-wire utilities and let behaviors emerge. The search procedure should do that for us. Behavior is complicated and **context** dependent.
- Utilities can also be regarded as a reflection of **uncertain outcomes**. But win or lose, you play it.
- An agent with **intransitive preferences** can be induced to give away all of its money. (Loop forever.)
- Utility scales: normalized utilities, micromorts, quality-adjusted life years.
- People would pay to reduce their risks.

3.2 Markov Decision Processes

- **MDP** A way of formalizing the idea of non-deterministic search, which is the search when your actions' outcomes are **uncertain**.
- **Noisy Movement** actions do not always go as planned.
- An MDP is defined by: a set of states, a set of actions, a transition function, a reward function, a start state, and maybe a terminal state. Therefore, one way to solve the MDPs is with expectimax search. Namely, it's yet another **fancy search**, but our **testing goal** has changed.
- "Markov" means action outcomes depend only on the current state (namely, to simplify the calculation, we **do need to** make some assumptions.) This is just like search, where the **successor function** could only depend on the current state (not the history). Well, if you do want to depend on the history, GIFF more powerful computers and be a master of, you named it, **statistics and matrix**.
- In deterministic single-agent search problems, we wanted an optimal plan, or sequence of actions, from start to a goal. (Actually same "greedy ideas" in non-deterministic problems, but in situations like this, we are forced to enjoy the **uncertainty**, which comes from, well, you named it, mother nature or rather, quantum mechanics.)
- An optimal **policy** is one that maximizes expected utility **if followed**. When we say if, you know we are talking about the fucking **uncertainty**.
- An **explicit** policy defines a reflex agent.
- **Reflex** an action that is performed as a response to a stimulus and without conscious thought.
- Expectimax did **not** compute entire policies, solutions, ideas, paths whatsoever. It computed the action for a single state only. But **IT WORKS**.
- What **Markov** did is just to remove the redundancy so that we can use **minimal** information to **render** the things down.
- **Your models are never going to be perfect.**
- Each MDP state projects an expectimax-like search tree.
- **Utilities of Sequences** Ask questions: more or less; now or later (mind the **discounting**).
- **Discounting** values of rewards (may and usually so) decay exponentially. It helps our algorithms **converge**!!!

- **Infinite Utilities** Finite horizon, discounting or absorbing state (like "overheated" for racing). In general we will have **discounts**.
- **Policy** Mapping from actions to states. **Utility** sum of (discounted) rewards.
- **Values of States** fundamental operation: compute the (**expectimax**) value of a state.
  - Expected utility under optimal action.
  - Average sum of (discounted) rewards.
  - This is **just** what expectimax computed.
- Recursive definition of values:
  - $V^*(s) = \max_a * Q^*(s,a)$
  - The value of a state is the **max** over all the **actions** available to you.
  - $Q^*(s,a) = \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$
  - We are going to get a reward in the next time step, and a **future** reward. They are going to be **weighted** by the **relative probabilities** that come from our **transition function**. Mind the future shit, so we need to add a **discount** on the future, because it's not what we get NOW.
- States are repeated. Tree goes on forever. Note: deep parts of the tree eventually **does not** matter if $\gamma < 1$.
- **Time-Limited Values** $V_k(s)$ is the **optimal** value of s if the game ends in k more time steps. Equivalently, it's what a **depth-k** expectimax would give from s. (Namely, life is short, PLEASE do not loop/recur forever, please...)
- Because we are just truncating, we are just ending the game—we **do not** need an evaluation function. Because **IT JUST STOPS**.
- So it is a **trade-off** of how many states you have, how connected they are and how deeply you want to go into the tree.
- Basic idea: approximations get refined towards optimal values. (When we say refine, we can also say, you guessed it, **filtering**.)
- **Convergence**
  - If the tree has maximum depth M, then V_M holds the actual untruncated values. **I have done the EM search** and any further iteration/recursion will do nothing (Namely, their **reward**, return, gains, utilities are considered, regarded, or set 0, so it's "mathematically" dead, then we do **not** need to, you name it, do calculations **any more**).
  - If the discount is **less than** 1.
  - The last layer is at best all $R_{\max}$.
  - The last layer is at worst $R_{\min}$.
  - But everything is **discounted** by γ^k that far out.
  - So V_k and V_{k+1} are **at most** $\gamma^k \max|R|$ different.
  - So as k increases, the values converge because the **differences** are **smaller than smaller**. After all these "smalling" processes, it will die, or at least looks like "dead". That word means there is **no significant** further growth.

3.3 Markov Decision Processes Continued

- Oh! That's their probability of landing a s'.
- **Policy** A map (surely it's with Python and Lisp) of states to actions.
- Your transition function tells you what the **likelihoods** are.
- **Rewards** are instantaneous and **values** are accumulative. It's an very important distinction.
- **How to be optimal:** Take correct first action; Keep being optimal. It's just some kind of recursive procedures, or some "iterative dynamic programming". Holy Jesus Lord, **they are actually fuckingly the same thing, will you be able to understand**? This of course is in terms of **implementation**. And isn't this concept called "greedy" in algorithm book?
- Bellman equations **characterize** the optimal values; the **fuck yourself** algorithm "computes" them. Because of this, there is now a **real recursion**. It bottoms out at 0. And suddenly we have a notion of times-attached.
- Expectimax trees max over all actions to compute the **optimal** values.
- Turn recursive Bellman equations into **updates**. Namely, it's just the fucking value iteration.
- Without the maxes, the Bellman equations are just a **linear system**.
- $\pi^*(s) = argmax_a \Sigma T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$ This is called **policy extraction**, since it gets the policy implied by the values. This is also called **reconstruct**.
- $\pi^*(s) = argmax_a Q(s,a)$ Computing actions from **Q-Values** is completely trivial. And it is much easier to select from q-values than values.
- What value iteration does, is essentially mimics Bellman Equation. Problem: It's slow—$O(S^2A)$ per iteration. The "max" at each state rarely changes. And more importantly, the **policy** often converges **long before** the values.
- **Policy Iteration (Or Recursion)** Evaluation: for fixed current policy π, find values with policy evaluation. Improvement: for fixed values, get a better policy by **argmax**.
- Another view (which is more practical) is thinking we are doing **value iteration** but on most rounds we just go with the **last action** that optimize the state, rather than considering them all.
- They differ only in whether we plug in a fixed policy or max over actions. Namely, these all look (and essentially are) the same.

- That wasn't planning! It was learning. There was an MDP, but you couldn't solve it with just **purely precomputation**. (It's all about the computation, but orders, namely, prior and posterior do matter.)
- Important ideas in reinforcement learning (in control theory bitches, they call the **crap negative/positive feedback loop**):
  - Exploration: you **have to** (as is with my situation, I have to learn, to prove to others, and to be strong) try unknown actions to get information.
  - Exploitation: eventually, you have to **use what you know**
  - Regret: even if you learn intelligently, you **make mistakes**.
  - **Sampling** because of chance, you have to try things **repeatedly**.
  - **Difficulty** learning can be much harder than solving a known MDP. (Comment: maybe that's why the Machine Learning SIGs are recruiting more and more, say, members/noobs?)
- Your **lack of knowledge** then triggers a much more difficult reasoning problem about **HOW** you should act. (Math buddies tells you WHAT. Programmers do the HOW, and a few true geniuses learn/are forced to ask WHY.) "Why OSX is more fancy? Why Windows has such a large market share, etc."
- **How you solve an MDP when you don't know which MDP you are solving?** Eggs first? Or Chicken first?

4 Learning

4.1 Reinforcement Learning 1

- Basic idea:
  - Receive feedback in the form of **rewards**
  - Agent's utility is defined by the **reward function**
  - Must (learn to) act so as to **maximize expected rewards**
  - Rule of thumb: All learning is based on what? **Observed samples of outcomes**. Namely, weighing the elephant. When you take an action, you see what happens. But you **DO NOT** see everything that might happen.
- New twist: don't know Transition function or reward function. Must actually **try actions and states** out to learn.
- Offline (MDPs). Online (Reinforcement Learning). (And then pretend that it is true.)
- **Model-Based Idea** Learn an approximate model based on experiences (namely, empirical crap). Solve for values **as if** the learned model **were** correct. (WHAT IF it is not correct? Then go fuck yourself.) This is, indeed how the state-of-the-art data mining, machine learning, and maybe cock sucking works.
- Model-Based Learning: Learn the empirical MDP model, then solve the **learned** MDP.
- $E[A] = \sum_a P(a) \cdot a$ This is 100% valid when we **know** $P(A)$. Without $P(A)$, we can **try to** collect samples $[a_1, a_2, \ldots, a_N]$.
  - For Model Based, we have $\hat{P}(a) = \frac{num(a)}{N}$ Eventually you learn the right model.
  - For Model Free, we just do $E[A] \simeq \frac{1}{N} \sum_i a_i$ Samples appear with the **right frequencies**.
- Passive Reinforcement Learning: Goal is to learn the **state values** by inputting/carrying out a fixed policy (namely, action/signal).
- We are missing something! But in the end, it's an average. It's an **average** of a bunch of things, each thing is a **one-step reward** plus a **discounted future** from **previous computation**. (Aren't math and natural language description sucking? Indeed they are.)
- **Sample-Based Policy Evaluation** Take samples of outcomes s' (by doing the ACTION!) and **average**.
- Big Idea: Learn from every experience. Namely: Update $V(s)$ each time we experience a transition (s, a, s', r). How? $V^\pi \leftarrow (1-\alpha)V^\pi(s) + (\alpha) * sample$ The fucking α is called learning rate.
- You can think of that as an **error**. $V^\pi(s) \leftarrow V^\pi(s) + \alpha(sample - V^\pi(s))$ Adjust your estimate in the direction of error by some small step size of α. In communication, they call the crap **Affine Projection** algorithm.
- Running interpolation update: $\bar{x}_n = (1-\alpha) \cdot \bar{x}_{n-1} + \alpha \cdot x_n$
- **Active Reinforcement Learning** learn the optimal policy/values
- But Q-Values are **more useful** (how the heck do you know that it is more useful? By trials and errors indeed.)
- $Q_{k+1}(s, a) \leftarrow \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma \max_{a'} Q_k(s', a')]$ This crap is called sample-based Q-value recursion/iteration. Key idea: instead of looking at **two consecutive** values, let us look at **two consecutive** Q-values. (Namely, because they are "more useful". Practically, they are relatively easy to, you guessed it, COMPUTE.)