Constraint Satisfaction Problems
What is Search For?

- Assumptions about the world: a single agent, deterministic actions, fully observed state, discrete state space

- Planning: sequences of actions
  - The path to the goal is the important thing
  - Paths have various costs, depths
  - Heuristics give problem-specific guidance

- Identification: assignments to variables
  - The goal itself is important, not the path
  - All paths at the same depth (for some formulations)
  - CSPs are a specialized class of identification problems
Constraint Satisfaction Problems

- **Standard search problems:**
  - State is a “black box”: arbitrary data structure
  - Goal test can be any function over states
  - Successor function can also be anything

- **Constraint satisfaction problems (CSPs):**
  - A special subset of search problems
  - State is defined by variables $X_i$ 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

- Simple example of a formal representation language

- Allows useful general-purpose algorithms with more power than standard search algorithms
Example: Map Coloring

- **Variables:** WA, NT, Q, NSW, V, SA, T
- **Domains:** $D = \{\text{red, green, blue}\}$
- **Constraints:** adjacent regions must have different colors
  
  Implicit: WA $\neq$ NT
  
  Explicit: $(WA, NT) \in \{(\text{red, green}), (\text{red, blue}), \ldots\}$

- **Solutions are assignments satisfying all constraints, e.g.:**
  
  \{WA=\text{red}, NT=\text{green}, Q=\text{red}, NSW=\text{green}, V=\text{red}, SA=\text{blue}, T=\text{green}\}
Example: N-Queens

- **Formulation 1:**
  - Variables: \( X_{ij} \)
  - Domains: \( \{0, 1\} \)
  - Constraints

\[
\forall i, j, k \quad (X_{ij}, X_{ik}) \in \{(0, 0), (0, 1), (1, 0)\}
\]
\[
\forall i, j, k \quad (X_{ij}, X_{kj}) \in \{(0, 0), (0, 1), (1, 0)\}
\]
\[
\forall i, j, k \quad (X_{ij}, X_{i+k, j+k}) \in \{(0, 0), (0, 1), (1, 0)\}
\]
\[
\forall i, j, k \quad (X_{ij}, X_{i+k, j-k}) \in \{(0, 0), (0, 1), (1, 0)\}
\]
\[
\sum_{i,j} X_{ij} = N
\]
Example: N-Queens

- **Formulation 2:**
  - **Variables:** $Q_k$
  - **Domains:** $\{1, 2, 3, \ldots N\}$
  - **Constraints:**
    - Implicit: $\forall i, j$ non-threatening($Q_i, Q_j$)
    - Explicit: $(Q_1, Q_2) \in \{(1, 3), (1, 4), \ldots\}$
Constraint Graphs

- Binary CSP: each constraint relates (at most) two variables
- Binary constraint graph: nodes are variables, arcs show constraints
- General-purpose CSP algorithms use the graph structure to speed up search. E.g., Tasmania is an independent subproblem!
Example: Cryptarithmetic

- **Variables:**
  
  \[ F \ T \ U \ W \ R \ O \ X_1 \ X_2 \ X_3 \]

- **Domains:**
  
  \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}

- **Constraints:**
  
  \[ \text{alldiff}(F, T, U, W, R, O) \]

  \[ O + O = R + 10 \cdot X_1 \]

  \[ \ldots \]
Example: Sudoku

- Variables:
  - Each (open) square
- Domains:
  - \{1,2,...,9\}
- Constraints:
  - 9-way alldiff for each column
  - 9-way alldiff for each row
  - 9-way alldiff for each region
  - (or can have a bunch of pairwise inequality constraints)
Example: The Waltz Algorithm

- The Waltz algorithm is for interpreting line drawings of solid polyhedra as 3D objects
- An early example of an AI computation posed as a CSP

Approach:
- Each intersection is a variable
- Adjacent intersections impose constraints on each other
- Solutions are physically realizable 3D interpretations

*corner = convex or concave?
Varieties of CSPs

- **Discrete Variables**
  - Finite domains
    - Size $d$ means $O(d^n)$ complete assignments
    - E.g., Boolean CSPs, including Boolean satisfiability (NP-complete)
  - Infinite domains (integers, strings, etc.)
    - E.g., job scheduling, variables are start/end times for each job
    - Linear constraints solvable, nonlinear undecidable

- **Continuous variables**
  - E.g., start/end times for Hubble Telescope observations
  - Linear constraints solvable in polynomial time by LP methods
Varieties of Constraints

- Varieties of Constraints
  - Unary constraints involve a single variable (equivalent to reducing domains), e.g.:
    \[ SA \neq \text{green} \]
  - Binary constraints involve pairs of variables, e.g.:
    \[ SA \neq WA \]
  - Higher-order constraints involve 3 or more variables: e.g., cryptarithmetic column constraints

- Preferences (soft constraints):
  - E.g., red is better than green
  - Often representable by a cost for each variable assignment
  - Gives constrained optimization problems
  - (We’ll ignore these until we get to Bayes’ nets)
Real-World CSPs

- Scheduling problems: e.g., when can we all meet?
- Timetabling problems: e.g., which class is offered when and where?
- Assignment problems: e.g., who teaches what class
- Hardware configuration
- Transportation scheduling
- Factory scheduling
- Circuit layout
- Fault diagnosis
- ... lots more!

- Many real-world problems involve real-valued variables...
Standard Search Formulation

- Standard search formulation of CSPs

- States defined by the values assigned so far (partial assignments)
  - Initial state: the empty assignment, {}
  - Successor function: assign a value to an unassigned variable
  - Goal test: the current assignment is complete and satisfies all constraints

- We’ll start with the straightforward, naïve approach, then improve it
Search Methods

- What would BFS do?
  *all solutions are at leaf; root = empty, 1-level = only one assignments, ....

- What would DFS do?
  *might be ok, but only at the end, found deadend and backtrack

- What problems does naïve search have?
  *should check constraints on the way
Backtracking search is the basic uninformed algorithm for solving CSPs

Idea 1: One variable at a time
- Variable assignments are commutative, so fix ordering
  - I.e., [WA = red then NT = green] same as [NT = green then WA = red]
  - Only need to consider assignments to a single variable at each step

Idea 2: Check constraints as you go
- I.e. consider only values which do not conflict with previous assignments
- Might have to do some computation to check the constraints
  - “Incremental goal test”

Depth-first search with these two improvements is called *backtracking search* (not the best name)

Can solve n-queens for \( n \approx 25 \)
Backtracking Example

(ordering/one assign)

(fail remove, eg (red, red))
Backtracking Search

function \textsc{Backtracking-Search}(csp) \textbf{returns} solution/failure
\hspace*{1em} return \textsc{Recursive-Backtracking}({\{} , csp) \\

function \textsc{Recursive-Backtracking}(assignment, csp) \textbf{returns} soln/failure
\hspace*{1em} if assignment is complete then return assignment \\
\hspace*{1em} var ← \textsc{Select-Unassigned-Variable}(\text{VARIABLES}[csp], assignment, csp) \\
\hspace*{1em} for each value in \textsc{Order-Domain-Values}(var, assignment, csp) do \\
\hspace*{1em} \hspace*{1em} if value is consistent with assignment given \text{CONSTRAINTS}[csp] then \\
\hspace*{1em} \hspace*{1em} \hspace*{1em} add \{var = value\} to assignment \\
\hspace*{1em} \hspace*{1em} \hspace*{1em} result ← \textsc{Recursive-Backtracking}(assignment, csp) \\
\hspace*{1em} \hspace*{1em} \hspace*{1em} if result \neq \text{failure} then return result \\
\hspace*{1em} \hspace*{1em} remove \{var = value\} from assignment \\
\hspace*{1em} return failure

- Backtracking = DFS + variable-ordering + fail-on-violation
- What are the choice points?

*variable and value choice
Improving Backtracking

- General-purpose ideas give huge gains in speed

- Ordering:
  - Which variable should be assigned next?
  - In what order should its values be tried?

- Filtering: Can we detect inevitable failure early?

- Structure: Can we exploit the problem structure?
Filtering: Forward Checking

- Filtering: Keep track of domains for unassigned variables and cross off bad options.
- Forward checking: Cross off values that violate a constraint when added to the existing assignment.
Filtering: Constraint Propagation

- Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures:

- NT and SA cannot both be blue!
- Why didn’t we detect this yet?
- *Constraint propagation*: reason from constraint to constraint
An arc $X \rightarrow Y$ is consistent iff for every $x$ in the tail there is some $y$ in the head which could be assigned without violating a constraint.

Forward checking: Enforcing consistency of arcs pointing to each new assignment.

*Delete from the tail!*

*every tail*
A simple form of propagation makes sure all arcs are consistent:

Important: If X loses a value, neighbors of X need to be rechecked!

Arc consistency detects failure earlier than forward checking

Can be run as a preprocessor or after each assignment

What’s the downside of enforcing arc consistency? *slow

Remember: *Delete from the tail!*
Enforcing Arc Consistency in a CSP

function AC-3(csp) returns the CSP, possibly with reduced domains
inputs: csp, a binary CSP with variables \{X_1, X_2, \ldots, X_n\}
local variables: queue, a queue of arcs, initially all the arcs in csp

while queue is not empty do
    \((X_i, X_j)\) ← REMOVE-FIRST(queue)
    if REMOVE-INCONSISTENT-VALUES(X_i, X_j) then
        for each \(X_k\) in Neighbors[X_i] do
            add \((X_k, X_i)\) to queue

function REMOVE-INCONSISTENT-VALUES(X_i, X_j) returns true iff succeeds
removed ← false
for each \(x\) in DOMAIN[X_i] do
    if no value \(y\) in DOMAIN[X_j] allows \((x, y)\) to satisfy the constraint \(X_i \leftarrow X_j\)
        then delete \(x\) from DOMAIN[X_i]; removed ← true
return removed

- Runtime: \(O(n^2d^3)\), can be reduced to \(O(n^2d^2)\)
- ... but detecting all possible future problems is NP-hard – why?
Limitations of Arc Consistency

- After enforcing arc consistency:
  - Can have one solution left
  - Can have multiple solutions left
  - Can have no solutions left (and not know it)

- Arc consistency still runs inside a backtracking search!
Ordering: Minimum Remaining Values

- **Variable Ordering: Minimum remaining values (MRV):**
  - Choose the variable with the fewest legal left values in its domain

- Why min rather than max?
- Also called “most constrained variable”
- “Fail-fast” ordering
Ordering: Least Constraining Value

- Value Ordering: Least Constraining Value
  - Given a choice of variable, choose the *least constraining value*
  - I.e., the one that rules out the fewest values in the remaining variables
  - Note that it may take some computation to determine this! (E.g., rerunning filtering)

- Why least rather than most?

- Combining these ordering ideas makes 1000 queens feasible

*easiest value – don’t need to use all the values*
Constraint Satisfaction Problems II
Reminder: CSPs

CSPs:
- Variables
- Domains
- Constraints
  - Implicit (provide code to compute)
  - Explicit (provide a list of the legal tuples)
  - Unary / Binary / N-ary

Goals:
- Here: find any solution
- Also: find all, find best, etc.
Improving Backtracking

- General-purpose ideas give huge gains in speed
  - ... but it’s all still NP-hard

- Filtering: Can we detect inevitable failure early?

- Ordering:
  - Which variable should be assigned next? (MRV)
  - In what order should its values be tried? (LCV)

- Structure: Can we exploit the problem structure?
Limitations of Arc Consistency

- After enforcing arc consistency:
  - Can have one solution left
  - Can have multiple solutions left
  - Can have no solutions left (and not know it)

- Arc consistency still runs inside a backtracking search!
**K-Consistency**

- Increasing degrees of consistency
  - 1-Consistency (Node Consistency): Each single node’s domain has a value which meets that node’s unary constraints
  - 2-Consistency (Arc Consistency): For each pair of nodes, any consistent assignment to one can be extended to the other
  - K-Consistency: For each k nodes, any consistent assignment to k-1 can be extended to the k\(^{th}\) node.

- Higher k more expensive to compute

- (You need to know the k=2 case: arc consistency)
Strong K-Consistency

- Strong k-consistency: also k-1, k-2, ... 1 consistent
- Claim: strong n-consistency means we can solve without backtracking!
- Why?
  - Choose any assignment to any variable
  - Choose a new variable
  - By 2-consistency, there is a choice consistent with the first
  - Choose a new variable
  - By 3-consistency, there is a choice consistent with the first 2
  - ...

- Lots of middle ground between arc consistency and n-consistency! (e.g. k=3, called path consistency)

*$x_i\rightarrow x_m\rightarrow x_j$

\(x_i,j \text{ path consistent with } x_m\)
Problem Structure

- Extreme case: independent subproblems
  - Example: Tasmania and mainland do not interact

- Independent subproblems are identifiable as connected components of constraint graph

- Suppose a graph of $n$ variables can be broken into subproblems of only $c$ variables:
  - Worst-case solution cost is $O((n/c)(d^c))$, linear in $n$
  - E.g., $n = 80$, $d = 2$, $c = 20$
  - $2^{80} = 4$ billion years at 10 million nodes/sec
  - $(4)(2^{20}) = 0.4$ seconds at 10 million nodes/sec
Tree-Structured CSPs

- Theorem: if the constraint graph has no loops, the CSP can be solved in $O(n d^2)$ time
  - Compare to general CSPs, where worst-case time is $O(d^n)$

- This property also applies to probabilistic reasoning (later): an example of the relation between syntactic restrictions and the complexity of reasoning
Tree-Structured CSPs

- Algorithm for tree-structured CSPs:
  - Order: Choose a root variable, order variables so that parents precede children
  - Remove backward: For $i = n : 2$, apply RemoveInconsistent($\text{Parent}(X_i), X_i$)
  - Assign forward: For $i = 1 : n$, assign $X_i$ consistently with $\text{Parent}(X_i)$

- Runtime: $O(n d^2)$ (why?)
Tree-Structured CSPs

- **Claim 1**: After backward pass, all root-to-leaf arcs are consistent
  - **Proof**: Each $X \rightarrow Y$ was made consistent at one point and $Y$’s domain could not have been reduced thereafter (because $Y$’s children were processed before $Y$)

- **Claim 2**: If root-to-leaf arcs are consistent, forward assignment will not backtrack
  - **Proof**: Induction on position

- Why doesn’t this algorithm work with cycles in the constraint graph?

*arc consistency for only two vars; need higher level consistency

- **Note**: we’ll see this basic idea again with Bayes’ nets
Improving Structure
Nearly Tree-Structured CSPs

- Conditioning: instantiate a variable, prune its neighbors' domains
- Cutset conditioning: instantiate (in all ways) a set of variables such that the remaining constraint graph is a tree
- Cutset size $c$ gives runtime $O\left( (d^c) (n-c) d^2 \right)$, very fast for small $c$
Cutset Conditioning

- Choose a cutset
- Instantiate the cutset (all possible ways)
- Compute residual CSP for each assignment
- Solve the residual CSPs (tree structured)
Cutset Quiz

- Find the smallest cutset for the graph below.

*A,B*
Tree Decomposition*

- Idea: create a tree-structured graph of mega-variables
- Each mega-variable encodes part of the original CSP
- Subproblems overlap to ensure consistent solutions

\[
\begin{align*}
\{&(WA=r, SA=g, NT=b), \\
&(WA=b, SA=r, NT=g), \\
&\ldots\} \\
\{&(NT=r, SA=g, Q=b), \\
&(NT=b, SA=g, Q=r), \\
&\ldots\} \\
&\text{Agree: } (M1, M2) \in \\
&\{((WA=g, SA=g, NT=g), (NT=g, SA=g, Q=g)), \ldots\}
\end{align*}
\]
Iterative Improvement
Iterative Algorithms for CSPs

- Local search methods typically work with “complete” states, i.e., all variables assigned

- To apply to CSPs:
  - Take an assignment with unsatisfied constraints
  - Operators reassign variable values
  - No fringe! Live on the edge.

- Algorithm: While not solved,
  - Variable selection: randomly select any conflicted variable
  - Value selection: min-conflicts heuristic:
    - Choose a value that violates the fewest constraints
    - I.e., hill climb with \( h(n) = \text{total number of violated constraints} \)
Example: 4-Queens

- States: 4 queens in 4 columns \((4^4 = 256\) states\)
- Operators: move queen in column
- Goal test: no attacks
- Evaluation: \(c(n) = \) number of attacks
Performance of Min-Conflicts

- Given random initial state, can solve n-queens in almost constant time for arbitrary n with high probability (e.g., n = 10,000,000)!

- The same appears to be true for any randomly-generated CSP except in a narrow range of the ratio

$$R = \frac{\text{number of constraints}}{\text{number of variables}}$$

![Diagram showing CPU time vs. critical ratio with a zone labeled "not over-constrained, not under-constrained" and another labeled "hard problems you are here".](image)
Local Search

- Tree search keeps unexplored alternatives on the fringe (ensures completeness)

- Local search: improve a single option until you can’t make it better (no fringe!)

- New successor function: local changes

- Generally much faster and more memory efficient (but incomplete and suboptimal)
Hill Climbing

- **Simple, general idea:**
  - Start wherever
  - Repeat: move to the best neighboring state
  - If no neighbors better than current, quit

- **What’s bad about this approach?**
  - Complete?
  - Optimal?

- **What’s good about it?**
Hill Climbing Quiz

Starting from X, where do you end up?
Starting from Y, where do you end up?
Starting from Z, where do you end up?
Simulated Annealing

- Idea: Escape local maxima by allowing downhill moves
  - But make them rarer as time goes on

```plaintext
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
inputs: problem, a problem
        schedule, a mapping from time to “temperature”
local variables: current, a node
                next, a node
                T, a “temperature” controlling prob. of downward steps

current ← MAKE-NODE(INITIAL-STATE[problem])
for t ← 1 to ∞ do
    T ← schedule[t]
    if T = 0 then return current
    next ← a randomly selected successor of current
    ΔE ← VALUE[next] − VALUE[current]
    if ΔE > 0 then current ← next
    else current ← next only with probability e^{ΔE/T}
```
Simulated Annealing

- Theoretical guarantee:
  - Stationary distribution: \( p(x) \propto e^{\frac{E(x)}{kT}} \)
  - If T decreased slowly enough, will converge to optimal state!

- Is this an interesting guarantee?
  - Sounds like magic, but reality is reality:
    - The more downhill steps you need to escape a local optimum, the less likely you are to ever make them all in a row
    - People think hard about ridge operators which let you jump around the space in better ways
Genetic algorithms use a natural selection metaphor

- Keep best N hypotheses at each step (selection) based on a fitness function
- Also have pairwise crossover operators, with optional mutation to give variety

- Possibly the most misunderstood, misapplied (and even maligned) technique around
Example: N-Queens

- Why does crossover make sense here?
- When wouldn’t it make sense?
- What would mutation be? *local change
- What would a good fitness function be?

*global scale, not local nudging
Adversarial Search
Game Playing State-of-the-Art

- **Checkers:** 1950: First computer player. 1994: First computer champion: Chinook ended 40-year-reign of human champion Marion Tinsley using complete 8-piece endgame. 2007: Checkers solved!

- **Chess:** 1997: Deep Blue defeats human champion Gary Kasparov in a six-game match. Deep Blue examined 200M positions per second, used very sophisticated evaluation and undisclosed methods for extending some lines of search up to 40 ply. Current programs are even better, if less historic.

- **Go:** 2016: Alpha Go defeats human champion. Uses Monte Carlo Tree Search, learned evaluation function.

- **Pacman**
Many different kinds of games!

Axes:
- Deterministic or stochastic?
- One, two, or more players?
- Zero sum?
- Perfect information (can you see the state)?

Want algorithms for calculating a strategy (policy) which recommends a move from each state
Deterministic Games

- Many possible formalizations, one is:
  - States: \( S \) (start at \( s_0 \))
  - Players: \( P=\{1...N\} \) (usually take turns)
  - Actions: \( A \) (may depend on player / state)
  - Transition Function: \( S \times A \rightarrow S \)
  - Terminal Test: \( S \rightarrow \{t,f\} \)
  - Terminal Utilities: \( S \times P \rightarrow R \)

- Solution for a player is a policy: \( S \rightarrow A \)
Zero-Sum Games

- Agents have opposite utilities (values on outcomes)
- Lets us think of a single value that one maximizes and the other minimizes
- Adversarial, pure competition

General Games

- Agents have independent utilities (values on outcomes)
- Cooperation, indifference, competition, and more are all possible
- More later on non-zero-sum games
Value of a state: The best achievable outcome (utility) from that state.

Non-Terminal States:
\[ V(s) = \max_{s' \in \text{children}(s)} V(s') \]

Terminal States:
\[ V(s) = \text{known} \]
Adversarial Game Trees – two agents
Minimax Values

States Under Agent’s Control:

\[ V(s) = \max_{s' \in \text{successors}(s)} V(s') \]

States Under Opponent’s Control:

\[ V(s') = \min_{s \in \text{successors}(s')} V(s) \]

Terminal States:

\[ V(s) = \text{known} \]
Tic-Tac-Toe Game Tree

*If two perfect players → value 0 (game is solved)
Deterministic, zero-sum games:
- Tic-tac-toe, chess, checkers
- One player maximizes result
- The other minimizes result

Minimax search:
- A state-space search tree
- Players alternate turns
- Compute each node’s minimax value: the best achievable utility against a rational (optimal) adversary
Minimax Implementation

```
def max-value(state):
    initialize v = -\infty
    for each successor of state:
        v = max(v, min-value(successor))
    return v
```

```
def min-value(state):
    initialize v = +\infty
    for each successor of state:
        v = min(v, max-value(successor))
    return v
```

\[
V(s) = \max_{s' \in \text{successors}(s)} V(s')
\]

\[
V(s') = \min_{s \in \text{successors}(s')} V(s)
\]
def value(state):
    if the state is a terminal state: return the state’s utility
    if the next agent is MAX: return max-value(state)
    if the next agent is MIN: return min-value(state)

def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor))
    return v

def min-value(state):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor))
    return v
Minimax Example
Minimax Properties

Optimal against a perfect player. Otherwise?
Minimax Efficiency

- How efficient is minimax?
  - Just like (exhaustive) DFS
  - Time: $O(b^m)$
  - Space: $O(bm)$

- Example: For chess, $b \approx 35$, $m \approx 100$
  - Exact solution is completely infeasible
  - But, do we need to explore the whole tree?
Resource Limits
Minimax Pruning
Alpha-Beta Pruning

- General configuration (MIN version)
  - We’re computing the MIN-VALUE at some node $n$
  - We’re looping over $n$’s children
  - $n$’s estimate of the children’s min is dropping
  - Who cares about $n$’s value? MAX
  - Let $a$ be the best value that MAX can get at any choice point along the current path from the root
  - If $n$ becomes worse than $a$, MAX will avoid it, so we can stop considering $n$’s other children (it’s already bad enough that it won’t be played)

- MAX version is symmetric
def min-value(state, α, β):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor, α, β))
        if v ≤ α return v
    β = min(β, v)
    return v

def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        if v ≥ β return v
    α = max(α, v)
    return v
Alpha-Beta Pruning Properties

- This pruning has **no effect** on minimax value computed for the root!

- Values of intermediate nodes might be wrong
  - Important: children of the root may have the wrong value
  - So the most naïve version won’t let you do action selection

- Good child ordering improves effectiveness of pruning

- With “perfect ordering”:
  - Time complexity drops to $O(b^{m/2})$
  - Doubles solvable depth!
  - Full search of, e.g. chess, is still hopeless...

- This is a simple example of **metareasoning** (computing about what to compute)

*computing alpha, beta makes not computing certain children*
Alpha-Beta Quiz

*Prune f*
Alpha-Beta Quiz 2

*Prune g, l*
Resource Limits

- **Problem:** In realistic games, cannot search to leaves!
- **Solution:** Depth-limited search
  - Instead, search only to a limited depth in the tree
  - Replace terminal utilities with an evaluation function for non-terminal positions
- **Example:**
  - Suppose we have 100 seconds, can explore 10K nodes/sec
  - So can check 1M nodes per move
  - $\alpha-\beta$ reaches about depth 8 – decent chess program
- **Guarantee of optimal play is gone**
- **More plies makes a BIG difference**
- **Use iterative deepening for an anytime algorithm**
Evaluation Functions

- Evaluation functions score non-terminals in depth-limited search

- Ideal function: returns the actual minimax value of the position
- In practice: typically weighted linear sum of features:
  \[
  \text{Eval}(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)
  \]
- e.g. \( f_1(s) = (\text{num white queens} - \text{num black queens}) \), etc.
Depth Matters

- Evaluation functions are always imperfect
- The deeper in the tree the evaluation function is buried, the less the quality of the evaluation function matters
- An important example of the tradeoff between complexity of features and complexity of computation
Synergies between Evaluation Function and Alpha-Beta?

- **Alpha-Beta**: amount of pruning depends on expansion ordering
  - Evaluation function can provide guidance to expand most promising nodes first (which later makes it more likely there is already a good alternative on the path to the root)
  - (somewhat similar to role of A* heuristic, CSPs filtering)

- **Alpha-Beta**: (similar for roles of min-max swapped)
  - Value at a min-node will only keep going down
  - Once value of min-node lower than better option for max along path to root, can prune
  - Hence: IF evaluation function provides upper-bound on value at min-node, and upper-bound already lower than better option for max along path to root THEN can prune
Uncertainty and Utilities
Worst-Case vs. Average Case

Idea: Uncertain outcomes controlled by chance, not an adversary!
Expectimax Search

- Why wouldn’t we know what the result of an action will be?
  - Explicit randomness: rolling dice
  - Unpredictable opponents: the ghosts respond randomly
  - Actions can fail: when moving a robot, wheels might slip

- Values should now reflect average-case (expectimax) outcomes, not worst-case (minimax) outcomes

- Expectimax search: compute the average score under optimal play
  - Max nodes as in minimax search
  - Chance nodes are like min nodes but the outcome is uncertain
  - Calculate their expected utilities
  - I.e. take weighted average (expectation) of children

- Later, we’ll learn how to formalize the underlying uncertain-result problems as Markov Decision Processes
def value(state):
    if the state is a terminal state: return the state’s utility
    if the next agent is MAX: return max-value(state)
    if the next agent is EXP: return exp-value(state)

def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor))
    return v

def exp-value(state):
    initialize v = 0
    for each successor of state:
        p = probability(successor)
        v += p * value(successor)
    return v
def exp-value(state):
    initialize v = 0
    for each successor of state:
        p = probability(successor)
        v += p * value(successor)
    return v

v = (1/2) (8) + (1/3) (24) + (1/6) (-12) = 10
Expectimax Example
Expectimax Pruning?

*Nop
Depth-Limited Expectimax

Estimate of true expectimax value (which would require a lot of work to compute)
In expectimax search, we have a probabilistic model of how the opponent (or environment) will behave in any state:

- Model could be a simple uniform distribution (roll a die)
- Model could be sophisticated and require a great deal of computation
- We have a chance node for any outcome out of our control: opponent or environment
- The model might say that adversarial actions are likely!

For now, assume each chance node magically comes along with probabilities that specify the distribution over its outcomes.
Quiz: Informed Probabilities

- Let’s say you know that your opponent is actually running a depth 2 minimax, using the result 80% of the time, and moving randomly otherwise
- Question: What tree search should you use?

Answer: Expectimax!

- To figure out EACH chance node’s probabilities, you have to run a simulation of your opponent
- This kind of thing gets very slow very quickly
- Even worse if you have to simulate your opponent simulating you...
- ... except for minimax, which has the nice property that it all collapses into one game tree
The Dangers of Optimism and Pessimism

**Dangerous Optimism**
Assuming chance when the world is adversarial

**Dangerous Pessimism**
Assuming the worst case when it’s not likely
Assumptions vs. Reality

Pacman used depth 4 search with an eval function that avoids trouble
Ghost used depth 2 search with an eval function that seeks Pacman

Results from playing 5 games

<table>
<thead>
<tr>
<th></th>
<th>Adversarial Ghost</th>
<th>Random Ghost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimax Pacman</td>
<td>Won 5/5</td>
<td>Won 5/5</td>
</tr>
<tr>
<td></td>
<td>Avg. Score: 483</td>
<td>Avg. Score: 493</td>
</tr>
<tr>
<td>Expectimax Pacman</td>
<td>Won 1/5</td>
<td>Won 5/5</td>
</tr>
<tr>
<td></td>
<td>Avg. Score: -303</td>
<td>Avg. Score: 503</td>
</tr>
</tbody>
</table>

*easy environment*
Mixed Layer Types

- E.g. Backgammon
- Expectiminimax
  - Environment is an extra “random agent” player that moves after each min/max agent
  - Each node computes the appropriate combination of its children
Example: Backgammon

- Dice rolls increase $b$: 21 possible rolls with 2 dice
  - Backgammon $\approx$ 20 legal moves
  - Depth 2 = $20 \times (21 \times 20)^3 = 1.2 \times 10^9$

- As depth increases, probability of reaching a given search node shrinks
  - So usefulness of search is diminished
  - So limiting depth is less damaging
  - But pruning is trickier...

- Historic AI: TDGammon uses depth-2 search + very good evaluation function + reinforcement learning: world-champion level play

- 1st AI world champion in any game!
Multi-Agent Utilities

- What if the game is not zero-sum, or has multiple players?

- Generalization of minimax:
  - Terminals have utility tuples
  - Node values are also utility tuples
  - Each player maximizes its own component
  - Can give rise to cooperation and competition dynamically...

```
1,6,6  7,1,2  6,1,2  7,2,1  5,1,7  1,5,2  7,7,1  5,2,5
```

![Diagram of utility tuples and game tree]
Utilities → decision making agent
Maximum Expected Utility

- Why should we average utilities? Why not minimax?

- Principle of maximum expected utility:
  - A rational agent should choose the action that maximizes its expected utility, given its knowledge

- Questions:
  - Where do utilities come from? *user gives vs. self-decide?*
  - How do we know such utilities even exist?
  - How do we know that averaging even makes sense?
  - What if our behavior (preferences) can’t be described by utilities?
What Utilities to Use?

- For worst-case minimax reasoning, terminal function scale doesn’t matter
  - We just want better states to have higher evaluations (get the ordering right)
  - We call this insensitivity to monotonic transformations

- For average-case expectimax reasoning, we need magnitudes to be meaningful
Utilities are functions from outcomes (states of the world) to real numbers that describe an agent’s preferences.

Where do utilities come from?
- In a game, may be simple (+1/-1)
- Utilities summarize the agent’s goals
- Theorem: any “rational” preferences can be summarized as a utility function

We hard-wire utilities and let behaviors emerge
- Why don’t we let agents pick utilities?
- Why don’t we prescribe behaviors?
Utilities: Uncertain Outcomes

Getting ice cream

Get Single

Get Double

Oops

Whew!
Preferences

- An agent must have preferences among:
  - Prizes: $A, B$, etc.
  - Lotteries: situations with uncertain prizes

  \[ L = [p, A; (1 - p), B] \]

- Notation:
  - Preference: $A > B$
  - Indifference:
We want some constraints on preferences before we call them rational, such as:

Axiom of Transitivity: \((A \succ B) \land (B \succ C) \Rightarrow (A \succ C)\)

For example: an agent with intransitive preferences can be induced to give away all of its money

- If \(B > C\), then an agent with \(C\) would pay (say) 1 cent to get \(B\)
- If \(A > B\), then an agent with \(B\) would pay (say) 1 cent to get \(A\)
- If \(C > A\), then an agent with \(A\) would pay (say) 1 cent to get \(C\)
Rational Preferences

The Axioms of Rationality

- **Orderability**
  $$(A \succ B) \lor (B \succ A) \lor (A \sim B)$$

- **Transitivity**
  $$(A \succ B) \land (B \succ C) \Rightarrow (A \succ C)$$

- **Continuity**
  $$A \succ B \succ C \Rightarrow \exists p \ [p, A ; 1 - p, C] \sim B$$

- **Substitutability**
  *holds if A,B switched
  $$A \sim B \Rightarrow [p, A ; 1 - p, C] \sim [p, B ; 1 - p, C]$$

- **Monotonicity**
  $$A \succ B \Rightarrow (p \geq q \Leftrightarrow [p, A ; 1 - p, B] \geq [q, A ; 1 - q, B])$$

Theorem: Rational preferences imply behavior describable as maximization of expected utility
Theorem [Ramsey, 1931; von Neumann & Morgenstern, 1944]
- Given any preferences satisfying these constraints, there exists a real-valued function $U$ such that:

\[ U(A) \geq U(B) \iff A \succeq B \]

\[ U([p_1, S_1; \ldots ; p_n, S_n]) = \sum_i p_i U(S_i) \]

- I.e. values assigned by $U$ preserve preferences of both prizes and lotteries!

Maximum expected utility (MEU) principle:
- Choose the action that maximizes expected utility
- Note: an agent can be entirely rational (consistent with MEU) without ever representing or manipulating utilities and probabilities
- E.g., a lookup table for perfect tic-tac-toe, a reflex vacuum cleaner
Example: Human Rationality?

- Famous example of Allais (1953)
  - A: [0.8, $4k; 0.2, $0]
  - B: [1.0, $3k; 0.0, $0]
  - C: [0.2, $4k; 0.8, $0]
  - D: [0.25, $3k; 0.75, $0]

- Most people prefer B > A, C > D

- But if U($0) = 0, then
  - B > A ⇒ U($3k) > 0.8 U($4k)
  - C > D ⇒ 0.8 U($4k) > U($3k)\cdot4 \text{ times, so people irrational}
Markov Decision Processes
Non-Deterministic Search
Example: Grid World

- A maze-like problem
  - The agent lives in a grid
  - Walls block the agent’s path

- Noisy movement: actions do not always go as planned
  - 80% of the time, the action North takes the agent North (if there is no wall there)
  - 10% of the time, North takes the agent West; 10% East
  - If there is a wall in the direction the agent would have been taken, the agent stays put

- The agent receives rewards each time step
  - Small “living” reward each step (can be negative)
  - Big rewards come at the end (good or bad)

- Goal: maximize sum of rewards
Grid World Actions

Deterministic Grid World

Stochastic Grid World
An MDP is defined by:
- A set of states $s \in S$
- A set of actions $a \in A$
- A transition function $T(s, a, s')$
  - Probability that $a$ from $s$ leads to $s'$, i.e., $P(s'| s, a)$
  - Also called the model or the dynamics
- A reward function $R(s, a, s')$
  - Sometimes just $R(s)$ or $R(s')$
- A start state
- Maybe a terminal state

MDPs are non-deterministic search problems
- One way to solve them is with expectimax search
- We’ll have a new tool soon
What is Markov about MDPs?

- “Markov” generally means that given the present state, the future and the past are independent.

- For Markov decision processes, “Markov” means action outcomes depend only on the current state:

  \[
  P(S_{t+1} = s'|S_t = s_t, A_t = a_t, S_{t-1} = s_{t-1}, A_{t-1}, \ldots S_0 = s_0) = P(S_{t+1} = s'|S_t = s_t, A_t = a_t)
  \]

- This is just like search, where the successor function could only depend on the current state (not the history).
Policies

- **In deterministic single-agent search problems, we wanted an optimal plan, or sequence of actions, from start to a goal**

- **For MDPs, we want an optimal policy** $\pi^*: S \rightarrow A$
  - A policy $\pi$ gives an action for each state
  - An optimal policy is one that maximizes expected utility if followed
  - An explicit policy defines a reflex agent

- **Expectimax didn’t compute entire policies**
  - It computed the action for a single state only

Optimal policy when $R(s, a, s') = -0.03$ for all non-terminals $s$
Optimal Policies

\[ R(s) = -2.0 \]

\[ R(s) = -0.4 \]

\[ R(s) = -0.03 \]

\[ R(s) = -0.01 \]
Example: Racing

- A robot car wants to travel far, quickly
- Three states: Cool, Warm, Overheated
- Two actions: Slow, Fast
- Going faster gets double reward
Each MDP state projects an expectimax-like search tree

\[ T(s,a,s') = P(s'|s,a) \]

\[ R(s,a,s') \]

(s, a) is a q-state

(s, a, s') called a transition
Utilities of Sequences

- What preferences should an agent have over reward sequences?

- More or less? \([1, 2, 2]\) or \([2, 3, 4]\)

- Now or later? \([0, 0, 1]\) or \([1, 0, 0]\)
Discounting

- It’s reasonable to maximize the sum of rewards
- It’s also reasonable to prefer rewards now to rewards later
- One solution: values of rewards decay exponentially

- Worth Now: \(1\)
- Worth Next Step: \(\gamma\)
- Worth In Two Steps: \(\gamma^2\)
Discounting

- **How to discount?**
  - Each time we descend a level, we multiply in the discount once

- **Why discount?**
  - Sooner rewards probably do have higher utility than later rewards
  - Also helps our algorithms converge

- **Example: discount of 0.5**
  - $U([1,2,3]) = 1*1 + 0.5*2 + 0.25*3$
  - $U([1,2,3]) < U([3,2,1])$
Stationary Preferences

- Theorem: if we assume stationary preferences:
  
  \[ [a_1, a_2, \ldots] \succ [b_1, b_2, \ldots] \]
  
  \[ \iff \]
  
  \[ [r, a_1, a_2, \ldots] \succ [r, b_1, b_2, \ldots] \]

- Then: there are only two ways to define utilities
  
  - Additive utility: \[ U([r_0, r_1, r_2, \ldots]) = r_0 + r_1 + r_2 + \cdots \]
  
  - Discounted utility: \[ U([r_0, r_1, r_2, \ldots]) = r_0 + \gamma r_1 + \gamma^2 r_2 \cdots \]
Quiz: Discounting

- **Given:**
  - States: a, b, c, d, e
  - Actions: East, West, and Exit (only available in exit states a, e)
  - Transitions: deterministic

- **Quiz 1:** For $\gamma = 1$, what is the optimal policy?

- **Quiz 2:** For $\gamma = 0.1$, what is the optimal policy?

- **Quiz 3:** For which $\gamma$ are West and East equally good when in state d?
Infinite Utilities?!

- **Problem:** What if the game lasts forever? Do we get infinite rewards?

- **Solutions:**
  - **Finite horizon:** (similar to depth-limited search)
    - Terminate episodes after a fixed $T$ steps (e.g. life)
    - Gives nonstationary policies ($\pi$ depends on time left)
  - **Discounting:** use $0 < \gamma < 1$
    
    $$ U([r_0, \ldots r_\infty]) = \sum_{t=0}^{\infty} \gamma^t r_t \leq R_{\text{max}}/(1 - \gamma) $$
    
    - Smaller $\gamma$ means smaller “horizon” – shorter term focus
  - **Absorbing state:** guarantee that for every policy, a terminal state will eventually be reached (like “overheated” for racing)

*yes, so cannot decide action*
Recap: Defining MDPs

- **Markov decision processes:**
  - Set of states $S$
  - Start state $s_0$
  - Set of actions $A$
  - Transitions $P(s'|s,a)$ (or $T(s,a,s')$)
  - Rewards $R(s,a,s')$ (and discount $\gamma$)

- **MDP quantities so far:**
  - Policy = Choice of action for each state
  - Utility = sum of (discounted) rewards
Solving MDPs
Optimal Quantities

- The value (utility) of a state $s$:
  \[ V^*(s) = \text{expected utility starting in } s \text{ and acting optimally} \]

- The value (utility) of a q-state $(s,a)$:
  \[ Q^*(s,a) = \text{expected utility starting out having taken action } a \text{ from state } s \text{ and (thereafter) acting optimally} \]

- The optimal policy:
  \[ \pi^*(s) = \text{optimal action from state } s \]
Snapshot of Demo – Gridworld V Values

VALUES AFTER 100 ITERATIONS

0.64  0.74  0.85  1.00

0.57  0.57  -1.00

0.49  0.43  0.48  0.28

Noise = 0.2
Discount = 0.9
Living reward = 0

*action noise
Snapshot of Demo – Gridworld Q Values

Noise = 0.2
Discount = 0.9
Living reward = 0

Q-VALUES AFTER 100 ITERATIONS
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 value:
\[
V^*(s) = \max_a Q^*(s, a)
\]
\[
Q^*(s, a) = \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V^*(s') \right]
\]
\[
V^*(s) = \max_a \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V^*(s') \right]
\]
Racing Search Tree

- We’re doing way too much work with expectimax!

- Problem: States are repeated
  - Idea: Only compute needed quantities once

- Problem: Tree goes on forever
  - Idea: Do a depth-limited computation, but with increasing depths until change is small
  - Note: deep parts of the tree eventually don’t matter if $\gamma < 1$
Key idea: time-limited values

Define $V_k(s)$ to be 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 $s$
Value Iteration

- Start with $V_0(s) = 0$: no time steps left means an expected reward sum of zero
- Given vector of $V_k(s)$ values, do one ply of expectimax from each state:
  \[
  V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V_k(s') \right]
  \]
- Repeat until convergence
- Complexity of each iteration: $O(S^2A)$
- Theorem: will converge to unique optimal values:
  - Basic idea: approximations get refined towards optimal values
  - Policy may converge long before values do
Example: Value Iteration

\[ V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V_k(s') \right] \]
Markov Decision Processes II
Recap: MDPs

- **Markov decision processes:**
  - States $S$
  - Actions $A$
  - Transitions $P(s'|s,a)$ (or $T(s,a,s')$)
  - Rewards $R(s,a,s')$ (and discount $\gamma$)
  - Start state $s_0$

- **Quantities:**
  - Policy = map of states to actions
  - Utility = sum of discounted rewards
  - Values = expected future utility from a state (max node)
  - Q-Values = expected future utility from a q-state (chance node)
The value (utility) of a state $s$:

$V^*(s) =$ expected utility starting in $s$ and acting optimally

The value (utility) of a q-state $(s,a)$:

$Q^*(s,a) =$ expected utility starting out having taken action $a$ from state $s$ and (thereafter) acting optimally

The optimal policy:

$\pi^*(s) =$ optimal action from state $s$
The Bellman Equations

How to be optimal:

Step 1: Take correct first action
Step 2: Keep being optimal
The Bellman Equations

- Definition of “optimal utility” via expectimax recurrence gives a simple one-step lookahead relationship amongst optimal utility values

\[ V^*(s) = \max_a Q^*(s, a) \]

\[ Q^*(s, a) = \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V^*(s') \right] \]

\[ V^*(s) = \max_a \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V^*(s') \right] \]

- These are the Bellman equations, and they characterize optimal values in a way we’ll use over and over
Bellman equations characterize the optimal values:

\[ V^*(s) = \max_a \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V^*(s') \right] \]

Value iteration computes them:

\[ V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V_k(s') \right] \]

Value iteration is just a fixed point solution method

- ... though the \( V_k \) vectors are also interpretable as time-limited values
How do we know the $V_k$ vectors are going to converge?

Case 1: If the tree has maximum depth $M$, then $V_M$ holds the actual untruncated values

Case 2: If the discount is less than 1

Sketch: For any state $V_k$ and $V_{k+1}$ can be viewed as depth $k+1$ expectimax results in nearly identical search trees

The difference is that on the bottom layer, $V_{k+1}$ has actual rewards while $V_k$ has zeros

That last layer is at best all $R_{\text{MAX}}$

It is at worst $R_{\text{MIN}}$

But everything is discounted by $\gamma^k$ 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
Policy Evaluation
Fixed Policies

- Expectimax trees max over all actions to compute the optimal values.
- If we fixed some policy $\pi(s)$, then the tree would be simpler – only one action per state. 
  - ... though the tree’s value would depend on which policy we fixed.

Do the optimal action

Do what $\pi$ says to do
Utilities for a Fixed Policy

- Another basic operation: compute the utility of a state $s$ under a fixed (generally non-optimal) policy.

- Define the utility of a state $s$, under a fixed policy $\pi$:
  
  $V^\pi(s) = \text{expected total discounted rewards starting in } s \text{ and following } \pi$

- Recursive relation (one-step look-ahead / Bellman equation):

  $$V^\pi(s) = \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V^\pi(s')]$$
Example: Policy Evaluation

Always Go Right

Always Go Forward
Policy Evaluation

How do we calculate the V’s for a fixed policy $\pi$?

Idea 1: Turn recursive Bellman equations into updates (like value iteration)

$$V_0^\pi(s) = 0$$

$$V_{k+1}^\pi(s) \leftarrow \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V_k^\pi(s')]$$

Efficiency: $O(S^2)$ per iteration

Idea 2: Without the maxes, the Bellman equations are just a linear system

- Solve with Matlab (or your favorite linear system solver)
Policy Extraction

*Input policy output value vs. input value output policy (action)
Let’s imagine we have the optimal values $V^*(s)$

How should we act?
- It’s not obvious!

We need to do a mini-expectimax (one step)

$$\pi^*(s) = \arg\max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V^*(s')]$$

This is called policy extraction, since it gets the policy implied by the values.
Computing Actions from Q-Values

- Let’s imagine we have the optimal q-values:

- How should we act?
  - Completely trivial to decide!

\[ \pi^*(s) = \arg \max_a Q^*(s, a) \]

- Important lesson: actions are easier to select from q-values than values!
Policy Iteration
Problems with Value Iteration

- Value iteration repeats the Bellman updates:

\[ V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V_k(s') \right] \]

- Problem 1: It’s slow – \(O(S^2A)\) per iteration

- Problem 2: The “max” at each state rarely changes

- Problem 3: The policy often converges long before the values
$k=0$

VALUES AFTER 0 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0
k=5

VALUES AFTER 5 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0
k=100

VALUES AFTER 100 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0
Policy Iteration

- **Alternative approach for optimal values:**
  - **Step 1: Policy evaluation:** calculate utilities for some fixed policy (not optimal utilities!) until convergence
  - **Step 2: Policy improvement:** update policy using one-step look-ahead with resulting converged (but not optimal!) utilities as future values
  - Repeat steps until policy converges

- **This is policy iteration**
  - It’s still optimal!
  - Can converge (much) faster under some conditions
Policy Iteration

- Evaluation: For fixed current policy $\pi$, find values with policy evaluation:
  - Iterate until values converge:
    \[
    V_{k+1}^{\pi_i}(s) \leftarrow \sum_{s'} T(s, \pi_i(s), s') \left[ R(s, \pi_i(s), s') + \gamma V_k^{\pi_i}(s') \right]
    \]

- Improvement: For fixed values, get a better policy using policy extraction
  - One-step look-ahead:
    \[
    \pi_{i+1}(s) = \arg \max_a \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V^{\pi_i}(s') \right]
    \]
Both value iteration and policy iteration compute the same thing (all optimal values)

In value iteration:
- Every iteration updates both the values and (implicitly) the policy
- We don’t track the policy, but taking the max over actions implicitly recomputes it

In policy iteration:
- We do several passes that update utilities with fixed policy (each pass is fast because we consider only one action, not all of them)
- After the policy is evaluated, a new policy is chosen (slow like a value iteration pass)
- The new policy will be better (or we’re done)

Both are dynamic programs for solving MDPs
Summary: MDP Algorithms

- **So you want to...**
  - Compute optimal values: use value iteration or policy iteration
  - Compute values for a particular policy: use policy evaluation
  - Turn your values into a policy: use policy extraction (one-step lookahead)

- **These all look the same!**
  - They basically are – they are all variations of Bellman updates
  - They all use one-step lookahead expectimax fragments
  - They differ only in whether we plug in a fixed policy or max over actions
RL: Double Bandits
Double-Bandit MDP

- Actions: Blue, Red
- States: Win, Lose

No discount
100 time steps
Both states have the same value
Offline Planning

- Solving MDPs is offline planning
  - You determine all quantities through computation
  - You need to know the details of the MDP
  - You do not actually play the game!

<table>
<thead>
<tr>
<th>Action</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Play Red</td>
<td>150</td>
</tr>
<tr>
<td>Play Blue</td>
<td>100</td>
</tr>
</tbody>
</table>

No discount
100 time steps
Both states have the same value
Let’s Play!

$2 \quad $2 \quad $0 \quad $2 \quad $2

$2 \quad $2 \quad $0 \quad $0 \quad $0
Online Planning

- Rules changed! Red’s win chance is different.
Let’s Play!

$0 \quad $0 \quad $0 \quad $2 \quad $0

$2 \quad $0 \quad $0 \quad $0 \quad $0$
What Just Happened?

- That wasn’t planning, it was learning!
  - Specifically, reinforcement learning
  - There was an MDP, but you couldn’t solve it with just computation
  - You needed to actually act to figure it out

- Important ideas in reinforcement learning that came up
  - Exploration: you have to 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