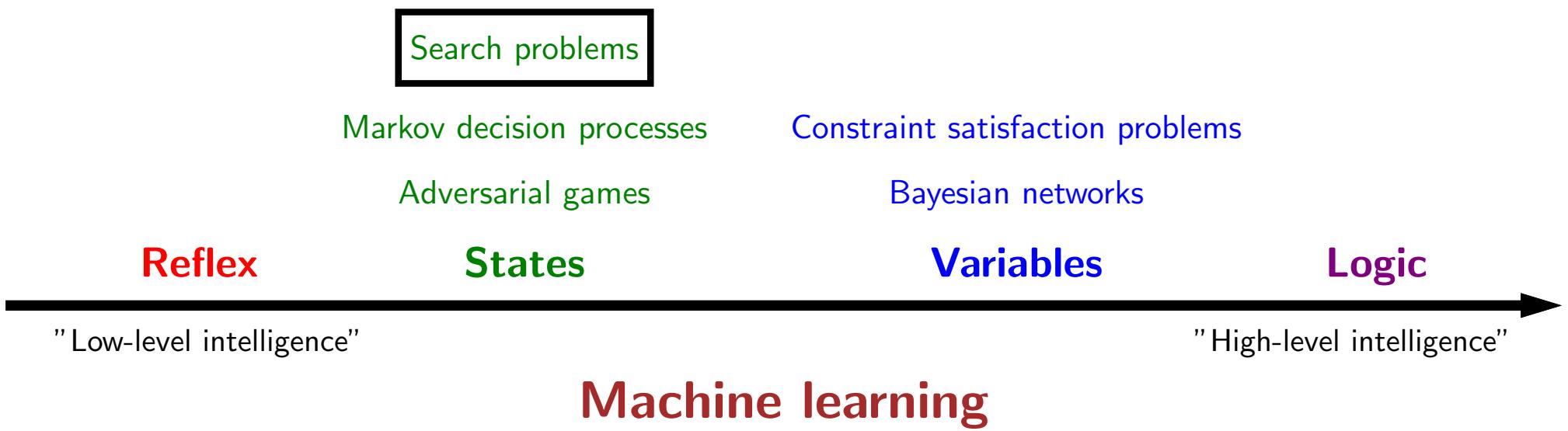




Search I



Course plan



Paradigm

Modeling

Inference

Learning



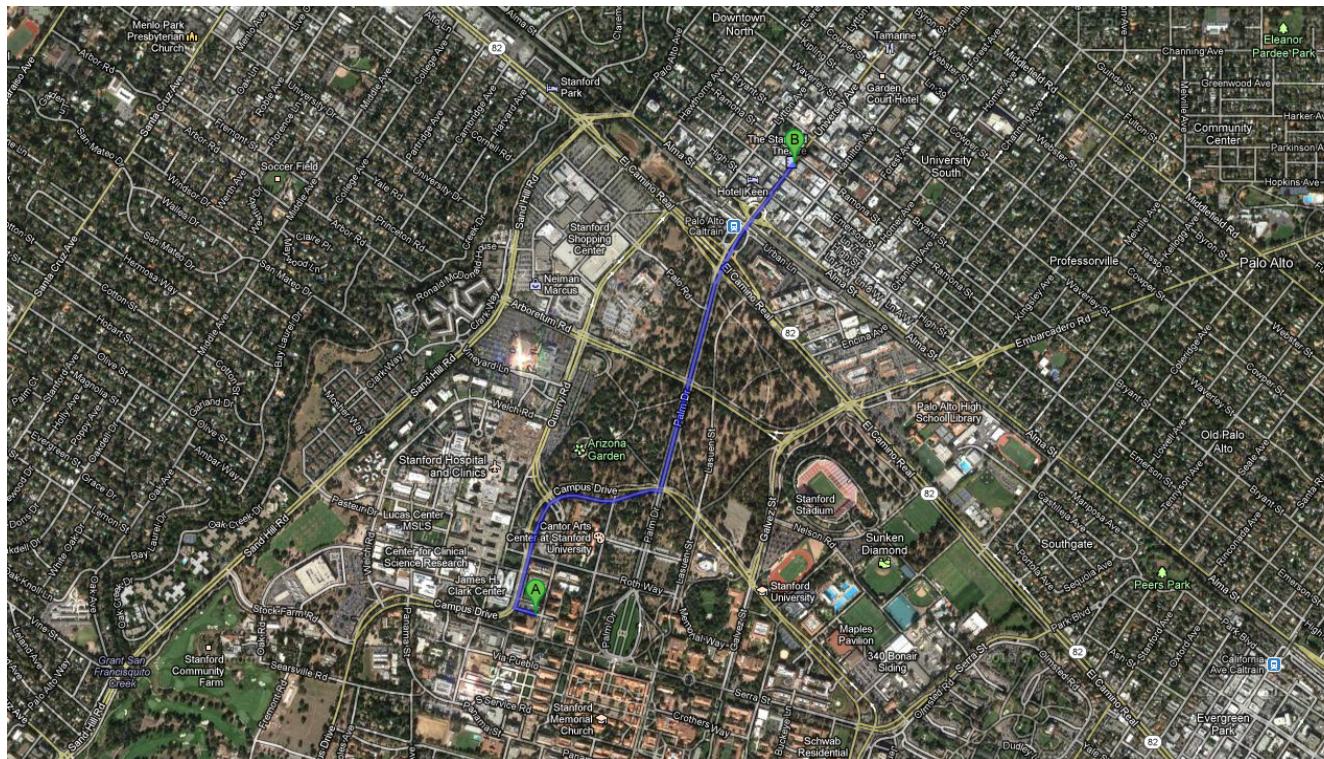
Roadmap

Tree search

Dynamic programming

Uniform cost search

Application: route finding

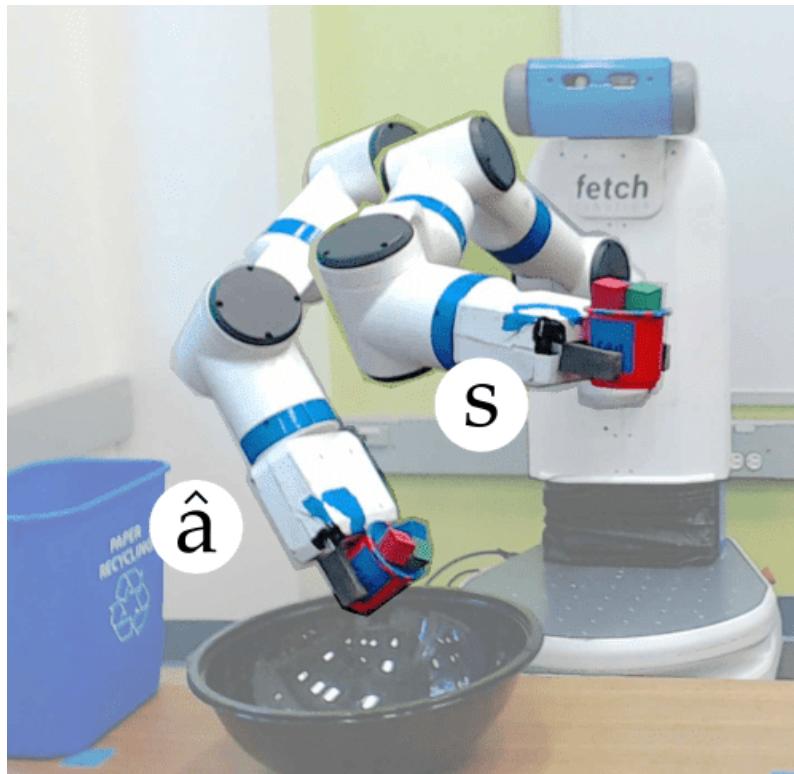


Objective: shortest? fastest? most scenic?

Actions: go straight, turn left, turn right

- Route finding is perhaps the most canonical example of a search problem. We are given as the input a map, a source point and a destination point. The goal is to output a sequence of actions (e.g., go straight, turn left, or turn right) that will take us from the source to the destination.
- We might evaluate action sequences based on an objective (distance, time, or pleasantness).

Application: robot motion planning

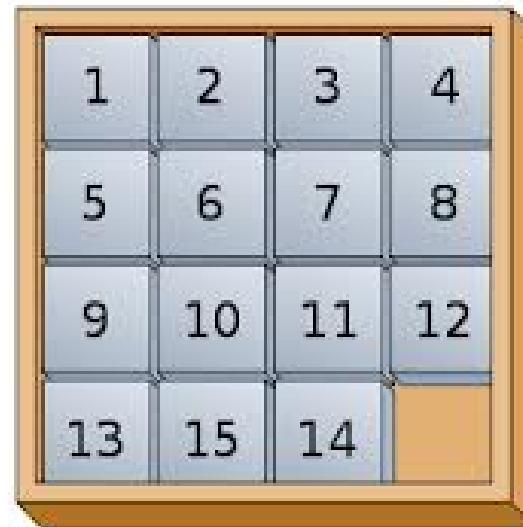
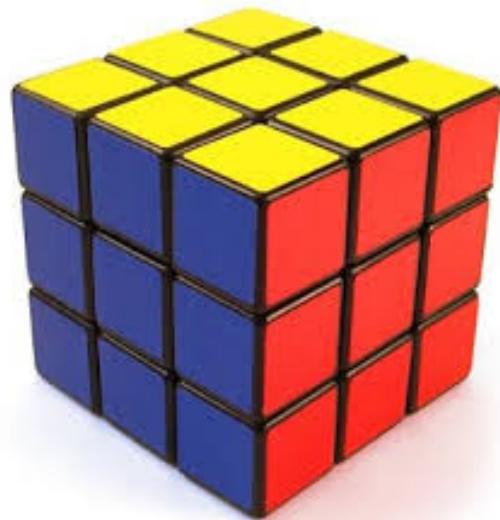


Objective: fastest? most energy efficient? safest? most expressive?

Actions: translate and rotate joints

- In robot motion planning, the goal is get a robot to move from one position/pose to another. The desired output trajectory consists of individual actions, each action corresponding to moving or rotating the joints by a small amount.
- Again, we might evaluate action sequences based on various resources like time or energy.

Application: solving puzzles



Objective: reach a certain configuration

Actions: move pieces (e.g., Move12Down)

- In solving various puzzles, the output solution can be represented by a sequence of individual actions. In the Rubik's cube, an action is rotating one slice of the cube. In the 15-puzzle, an action is moving one square to an adjacent free square.
- In puzzles, even finding one solution might be an accomplishment. The more ambitious might want to find the best solution (say, minimize the number of moves).

Application: machine translation

la maison bleue



the blue house

Objective: fluent English and preserves meaning

Actions: append single words (e.g., the)

- In machine translation, the goal is to output a sentence that's the translation of the given input sentence. The output sentence can be built out of actions, each action appending a word or a phrase to the current output.

Beyond reflex

Classifier (reflex-based models):



Search problem (state-based models):



Key: need to consider future consequences of an action!

- Last week, we finished our tour of machine learning of **reflex-based models** (e.g., linear predictors and neural networks) that output either a $+1$ or -1 (for binary classification) or a real number (for regression).
- While reflex-based models were appropriate for some applications such as sentiment classification or spam filtering, the applications we will look at today, such as solving puzzles, demand more.
- To tackle these new problems, we will introduce **search problems**, our first instance of a **state-based model**.
- In a search problem, in a sense, we are still building a predictor f which takes an input x , but f will now return an entire **action sequence**, not just a single action. Of course you should object: can't I just apply a reflex model iteratively to generate a sequence? While that is true, the search problems that we're trying to solve importantly require reasoning about the consequences of the entire action sequence, and cannot be tackled by myopically predicting one action at a time.
- Tangent: Of course, saying "cannot" is a bit strong, since sometimes a search problem can be solved by a reflex-based model. You could have a massive lookup table that told you what the best action was for any given situation. It is interesting to think of this as a time/memory tradeoff where reflex-based models are performing an implicit kind of caching. Going on a further tangent, one can even imagine **compiling** a state-based model into a reflex-based model; if you're walking around Stanford for the first time, you might have to really plan things out, but eventually it kind of becomes reflex.
- We have looked at many real-world examples of this paradigm. For each example, the key is to decompose the output solution into a sequence of primitive actions. In addition, we need to think about how to evaluate different possible outputs.



Question

A **farmer** wants to get his **cabbage**, **goat**, and **wolf** across a river. He has a boat that only holds two. He cannot leave the cabbage and goat alone or the goat and wolf alone. How many river crossings does he need?

4

5

6

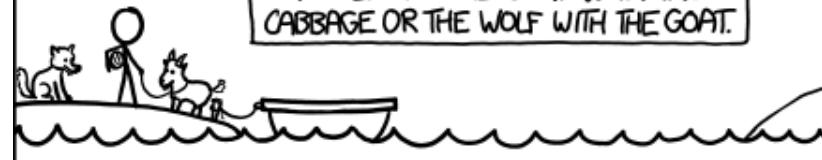
7

no solution

- When you solve this problem, try to think about how you did it. You probably simulated the scenario in your head, trying to send the farmer over with the goat and observing the consequences. If nothing got eaten, you might continue with the next action. Otherwise, you undo that move and try something else.
- But the point is not for you to be able to solve this one problem manually. The real question is: How can we get a machine to do solve all problems like this automatically? One of the things we need is a systematic approach that considers all the possibilities. We will see that **search problems** define the possibilities, and **search algorithms** explore these possibilities.

PROBLEM:

THE BOAT ONLY HOLDS TWO, BUT YOU CAN'T LEAVE THE GOAT WITH THE CABBAGE OR THE WOLF WITH THE GOAT.



SOLUTION:

1. TAKE THE GOAT ACROSS.



2. RETURN ALONE.

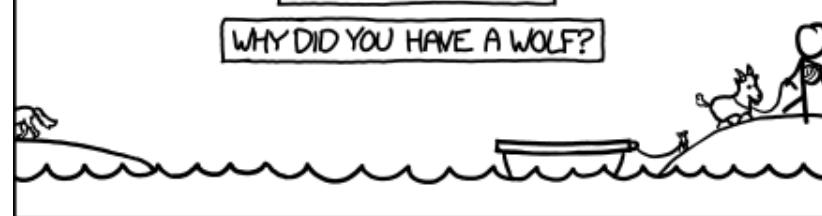


3. TAKE THE CABBAGE ACROSS.

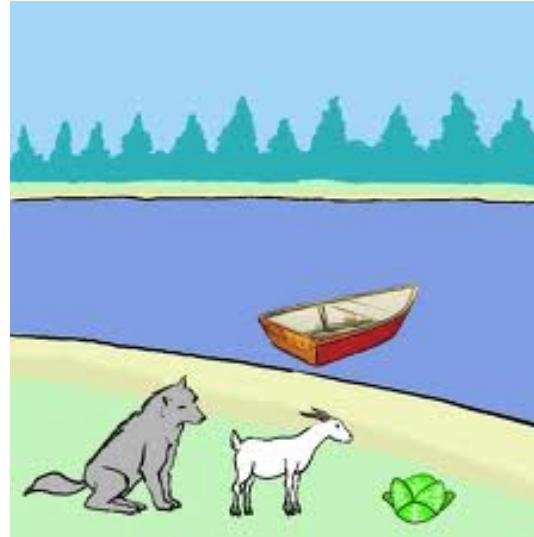


4. LEAVE THE WOLF.

WHY DID YOU HAVE A WOLF?



- This example, taken from *xkcd*, points out the cautionary tale that sometimes you can do better if you change the model (perhaps the value of having a wolf is zero) instead of focusing on the algorithm.



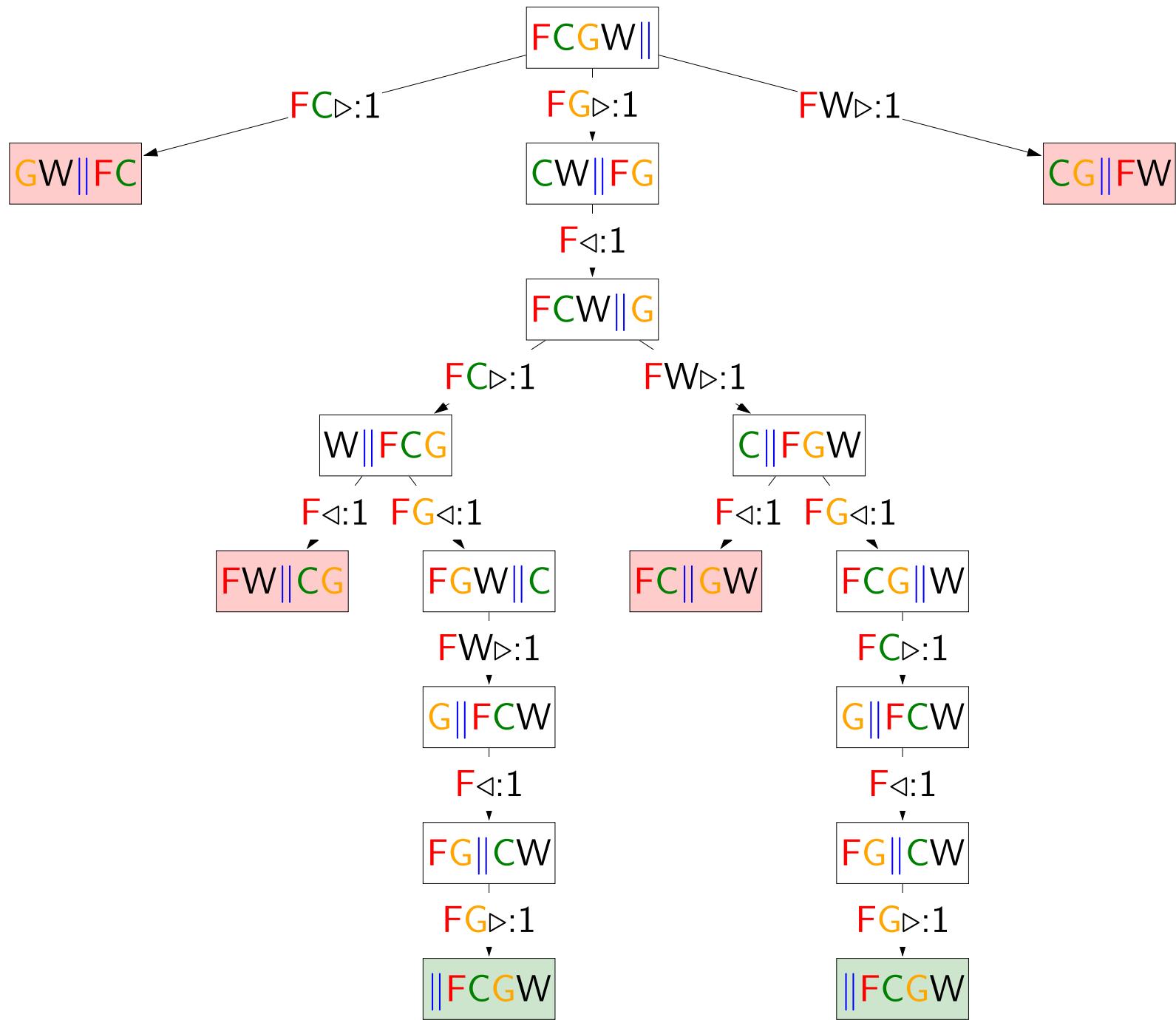
Farmer Cabbage Goat Wolf

Actions:

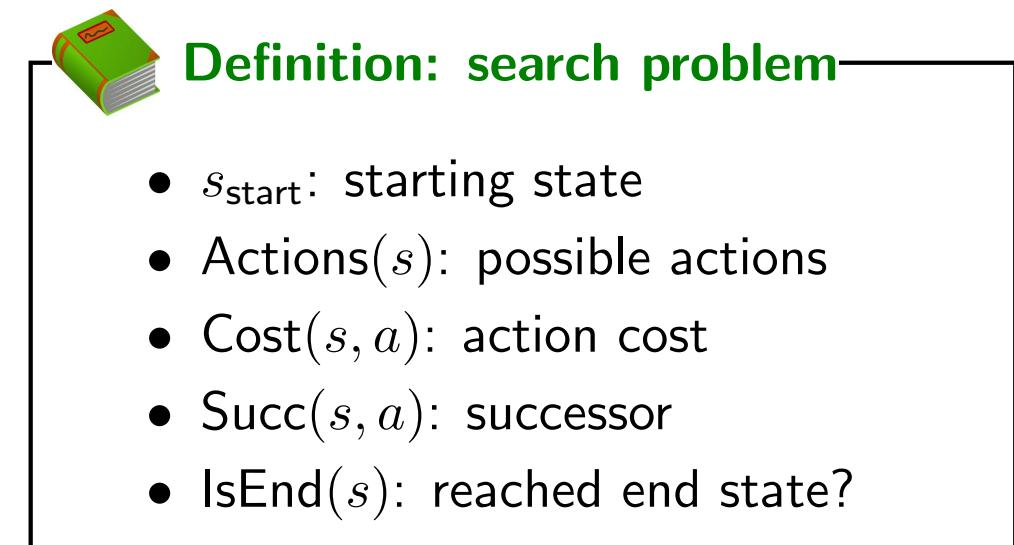
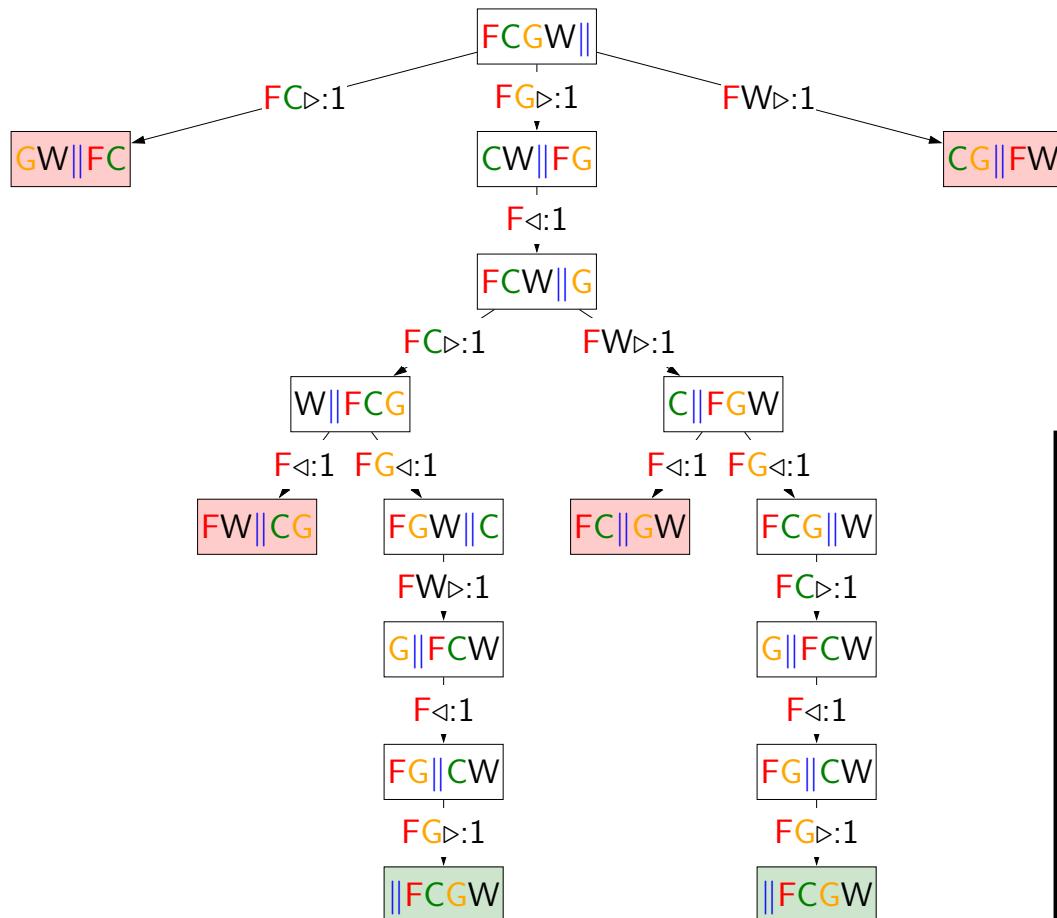
$F \triangleright$	$F \triangleleft$
$FC \triangleright$	$FC \triangleleft$
$FG \triangleright$	$FG \triangleleft$
$FW \triangleright$	$FW \triangleleft$

Approach: build a **search tree** ("what if?")

- We first start with our boat crossing puzzle. While you can possibly solve it in more clever ways, let us approach it in a very brain-dead, simple way, which allows us to introduce the notation for search problems.
- For this problem, we have eight possible actions, which will be denoted by a concise set of symbols. For example, the action $\text{FG}\triangleright$ means that the farmer will take the goat across to the right bank; $\text{F}\triangleleft$ means that the farmer is coming back to the left bank alone.



Search problem



- We will build what we will call a **search tree**. The root of the tree is the start state s_{start} , and the leaves are the end states ($\text{IsEnd}(s)$ is true). Each edge leaving a node s corresponds to a possible action $a \in \text{Actions}(s)$ that could be performed in state s . The edge is labeled with the action and its cost, written $a : \text{Cost}(s, a)$. The action leads deterministically to the successor state $\text{Succ}(s, a)$, represented by the child node.
- In summary, each root-to-leaf path represents a possible action sequence, and the sum of the costs of the edges is the cost of that path. The goal is to find the root-to-leaf path that ends in a valid end state with minimum cost.
- Note that in code, we usually do not build the search tree as a concrete data structure. The search tree is used merely to visualize the computation of the search algorithms and study the structure of the search problem.
- For the boat crossing example, we have assumed each action (a safe river crossing) costs 1 unit of time. We disallow actions that return us to an earlier configuration. The green nodes are the end states. The red nodes are not end states but have no successors (they result in the demise of some animal or vegetable). From this search tree, we see that there are exactly two solutions, each of which has a total cost of 7 steps.



Transportation example



Example: transportation

Street with blocks numbered 1 to n .

Walking from s to $s + 1$ takes 1 minute.

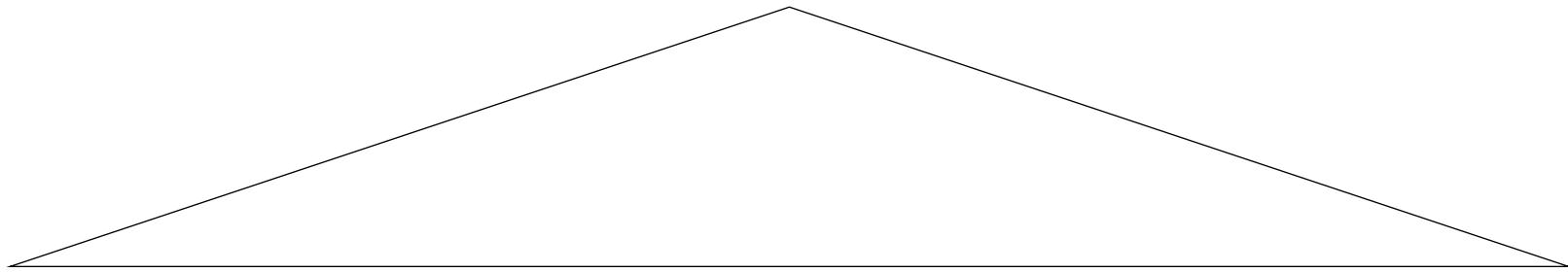
Taking a magic tram from s to $2s$ takes 2 minutes.

How to travel from 1 to n in the least time?

[semi-live solution: `TransportationProblem`]

- Let's consider another problem and practice modeling it as a search problem. Recall that this means specifying precisely what the states, actions, goals, costs, and successors are.
- To avoid the ambiguity of natural language, we will do this directly in code, where we define a `SearchProblem` class and implement the methods: `startState`, `isEnd` and `succAndCost`.

Backtracking search



[whiteboard: search tree]

If b actions per state, maximum depth is D actions:

- Memory: $O(D)$ (**small**)
- Time: $O(b^D)$ (**huge**) [$2^{50} = 1125899906842624$]

- Now let's put modeling aside and suppose we are handed a search problem. How do we construct an algorithm for finding a **minimum cost path** (not necessarily unique)?
- We will start with **backtracking search**, the simplest algorithm which just tries all paths. The algorithm is called recursively on the current state s and the path leading up to that state. If we have reached a goal, then we can update the minimum cost path with the current path. Otherwise, we consider all possible actions a from state s , and recursively search each of the possibilities.
- Graphically, backtracking search performs a depth-first traversal of the search tree. What is the time and memory complexity of this algorithm?
- To get a simple characterization, assume that the search tree has maximum depth D (each path consists of D actions/edges) and that there are b available actions per state (the **branching factor** is b).
- It is easy to see that backtracking search only requires $O(D)$ memory (to maintain the stack for the recurrence), which is as good as it gets.
- However, the running time is proportional to the number of nodes in the tree, since the algorithm needs to check each of them. The number of nodes is $1 + b + b^2 + \dots + b^D = \frac{b^{D+1} - 1}{b - 1} = O(b^D)$. Note that the total number of nodes in the search tree is on the same order as the number of leaves, so the cost is always dominated by the last level.
- In general, there might not be a finite upper bound on the depth of a search tree. In this case, there are two options: (i) we can simply cap the maximum depth and give up after a certain point or (ii) we can disallow visits to the same state.
- It is worth mentioning that the greedy algorithm that repeatedly chooses the lowest action myopically won't work. Can you come up with an example?

Backtracking search



Algorithm: backtracking search

```
def backtrackingSearch( $s$ , path):
    If IsEnd( $s$ ): update minimum cost path
    For each action  $a \in \text{Actions}(s)$ :
        Extend path with  $\text{Succ}(s, a)$  and  $\text{Cost}(s, a)$ 
        Call backtrackingSearch( $\text{Succ}(s, a)$ , path)
    Return minimum cost path
```

[semi-live solution: `backtrackingSearch`]

Depth-first search



Assumption: zero action costs

Assume action costs $\text{Cost}(s, a) = 0$.

Idea: Backtracking search + stop when find the first end state.

If b actions per state, maximum depth is D actions:

- Space: still $O(D)$
- Time: still $O(b^D)$ worst case, but could be much better if solutions are easy to find

- Backtracking search will always work (i.e., find a minimum cost path), but there are cases where we can do it faster. But in order to do that, we need some additional assumptions — there is no free lunch.
- Suppose we make the assumption that all the action costs are zero. In other words, all we care about is finding a valid action sequence that reaches the goal. Any such sequence will have the minimum cost: zero.
- In this case, we can just modify backtracking search to not keep track of costs and then stop searching as soon as we reach a goal. The resulting algorithm is **depth-first search** (DFS), which should be familiar to you. The worst time and space complexity are of the same order as backtracking search. In particular, if there is no path to an end state, then we have to search the entire tree.
- However, if there are many ways to reach the end state, then we can stop much earlier without exhausting the search tree. So DFS is great when there are an abundance of solutions.

Breadth-first search



Assumption: constant action costs

Assume action costs $\text{Cost}(s, a) = c$ for some $c \geq 0$.

Idea: explore all nodes in order of increasing depth.

Legend: b actions per state, solution has d actions

- Space: now $O(b^d)$ (a lot worse!)
- Time: $O(b^d)$ (better, depends on d , not D)

- **Breadth-first search** (BFS), which should also be familiar, makes a less stringent assumption, that all the action costs are the same non-negative number. This effectively means that all the paths of a given length have the same cost.
- BFS maintains a queue of states to be explored. It pops a state off the queue, then pushes its successors back on the queue.
- BFS will search all the paths consisting of one edge, two edges, three edges, etc., until it finds a path that reaches a end state. So if the solution has d actions, then we only need to explore $O(b^d)$ nodes, thus taking that much time.
- However, a potential show-stopper is that BFS also requires $O(b^d)$ space since the queue must contain all the nodes of a given level of the search tree. Can we do better?

DFS with iterative deepening



Assumption: constant action costs

Assume action costs $\text{Cost}(s, a) = c$ for some $c \geq 0$.

Idea:

- Modify DFS to stop at a maximum depth.
- Call DFS for maximum depths 1, 2,

DFS on d asks: is there a solution with d actions?

Legend: b actions per state, solution size d

- Space: $O(d)$ (saved!)
- Time: $O(b^d)$ (same as BFS)

- Yes, we can do better with a trick called **iterative deepening**. The idea is to modify DFS to make it stop after reaching a certain depth. Therefore, we can invoke this modified DFS to find whether a valid path exists with at most d edges, which as discussed earlier takes $O(d)$ space and $O(b^d)$ time.
- Now the trick is simply to invoke this modified DFS with cutoff depths of $1, 2, 3, \dots$ until we find a solution or give up. This algorithm is called DFS with iterative deepening (DFS-ID). In this manner, we are guaranteed optimality when all action costs are equal (like BFS), but we enjoy the parsimonious space requirements of DFS.
- One might worry that we are doing a lot of work, searching some nodes many times. However, keep in mind that both the number of leaves and the number of nodes in a search tree is $O(b^d)$ so asymptotically DFS with iterative deepening is the same time complexity as BFS.



Tree search algorithms

Legend: b actions/state, solution depth d , maximum depth D

Algorithm	Action costs	Space	Time
Backtracking	any	$O(D)$	$O(b^D)$
DFS	zero	$O(D)$	$O(b^D)$
BFS	constant ≥ 0	$O(b^d)$	$O(b^d)$
DFS-ID	constant ≥ 0	$O(d)$	$O(b^d)$

- Always exponential time
- Avoid exponential space with DFS-ID

- Here is a summary of all the tree search algorithms, the assumptions on the action costs, and the space and time complexities.
- The take-away is that we can't avoid the exponential time complexity, but we can certainly have linear space complexity. Space is in some sense the more critical dimension in search problems. Memory cannot magically grow, whereas time "grows" just by running an algorithm for a longer period of time, or even by parallelizing it across multiple machines (e.g., where each processor gets its own subtree to search).



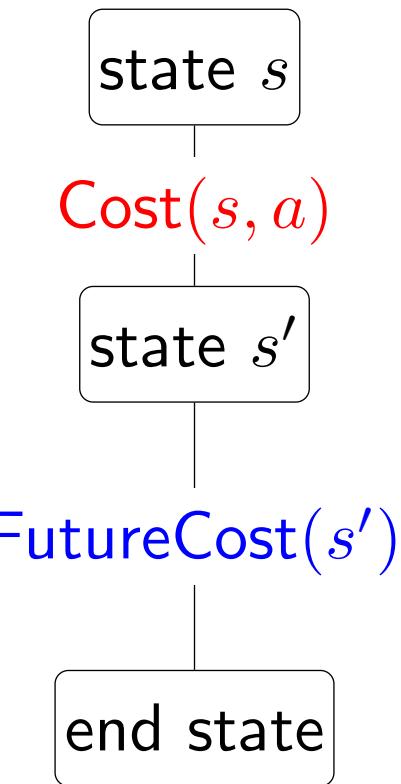
Roadmap

Tree search

Dynamic programming

Uniform cost search

Dynamic programming



Minimum cost path from state s to a end state:

$$\text{FutureCost}(s) = \begin{cases} 0 & \text{if } \text{IsEnd}(s) \\ \min_{a \in \text{Actions}(s)} [\text{Cost}(s, a) + \text{FutureCost}(\text{Succ}(s, a))] & \text{otherwise} \end{cases}$$

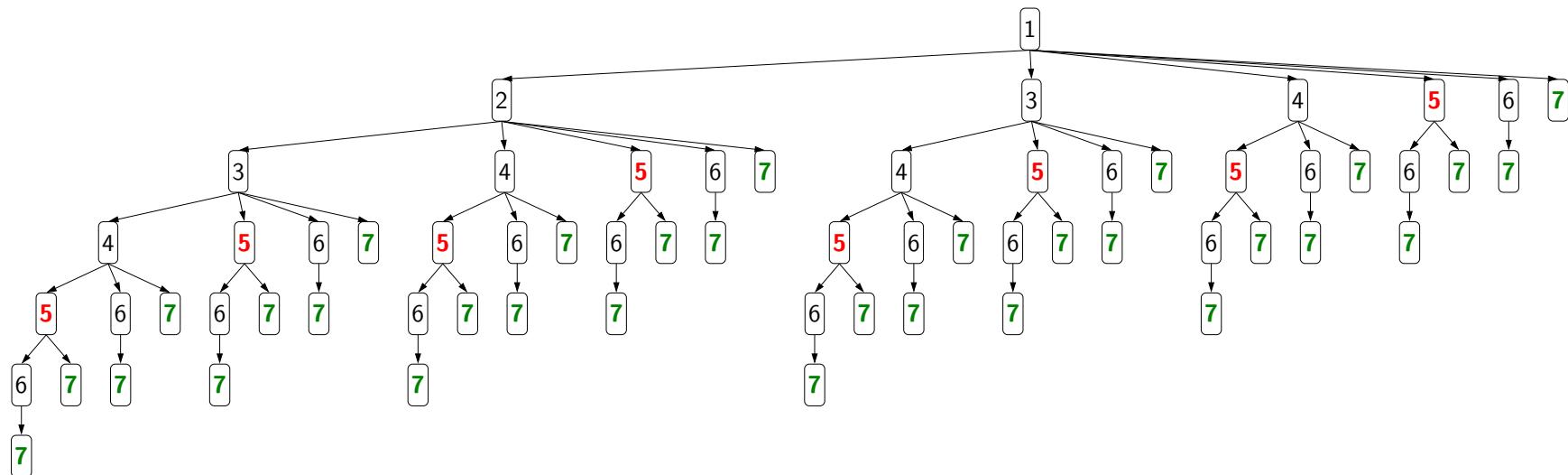
- Now let's see if we can avoid the exponential running time of tree search. Our first algorithm will be dynamic programming. We have already seen dynamic programming in specific contexts. Now we will use the search problem abstraction to define a single dynamic program for all search problems.
- First, let us try to think about the minimum cost path in the search tree recursively. Define $\text{FutureCost}(s)$ as the cost of the minimum cost path from s to some end state. The minimum cost path starting with a state s to an end state must take a first action a , which results in another state s' , from which we better take a minimum cost path to the end state.
- Written in symbols, we have a nice recurrence. Throughout this course, we will see many recurrences of this form. The basic form is a base case (when s is an end state) and an inductive case, which consists of taking the minimum over all possible actions a from s , taking an initial step resulting in an **immediate** action cost $\text{Cost}(s, a)$ and a **future** cost.

Motivating task



Example: route finding

Find the minimum cost path from city 1 to city n , only moving forward. It costs c_{ij} to go from i to j .

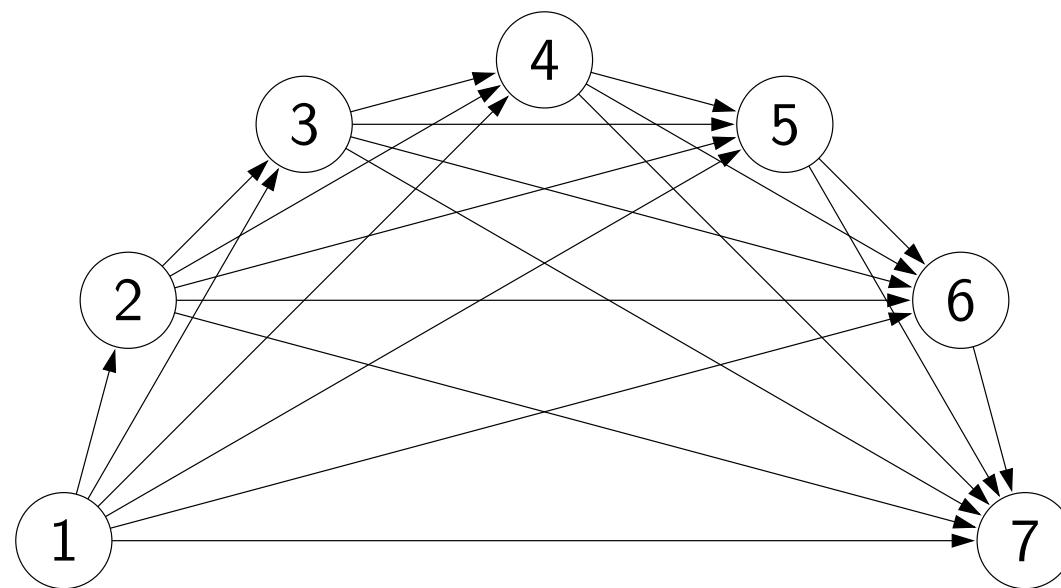


Observation: future costs only depend on current city

- Now let us see if we can avoid the exponential time. If we consider the simple route finding problem of traveling from city 1 to city n , the search tree grows exponentially with n .
- However, upon closer inspection, we note that this search tree has a lot of repeated structures. Moreover (and this is important), the future costs (the minimum cost of reaching a end state) of a state only depends on the current city! So therefore, all the subtrees rooted at city 5, for example, have the same minimum cost!
- If we can just do that computation once, then we will have saved big time. This is the central idea of **dynamic programming**.
- We've already reviewed dynamic programming in the first lecture. The purpose here is to construct one generic dynamic programming solution that will work on any search problem. Again, this highlights the useful division between modeling (defining the search problem) and algorithms (performing the actual search).

Dynamic programming

State: ~~past sequence of actions~~ current city



Exponential saving in time and space!

- Let us collapse all the nodes that have the same city into one. We no longer have a tree, but a directed acyclic graph with only n nodes rather than exponential in n nodes.
- Note that dynamic programming is only useful if we can define a search problem where the number of states is small enough to fit in memory.

Dynamic programming



Algorithm: dynamic programming

```
def DynamicProgramming( $s$ ):  
    If already computed for  $s$ , return cached answer.  
    If IsEnd( $s$ ): return solution  
    For each action  $a \in \text{Actions}(s)$ : ...
```

[semi-live solution: Dynamic Programming]



Assumption: acyclicity

The state graph defined by $\text{Actions}(s)$ and $\text{Succ}(s, a)$ is acyclic.

- The dynamic programming algorithm is exactly backtracking search with one twist. At the beginning of the function, we check to see if we've already computed the future cost for s . If we have, then we simply return it (which takes constant time if we use a hash map). Otherwise, we compute it and save it in the cache so we don't have to recompute it again. In this way, for every state, we are only computing its value once.
- For this particular example, the running time is $O(n^2)$, the number of edges.
- One important point is that the graph must be acyclic for dynamic programming to work. If there are cycles, the computation of a future cost for s might depend on s' which might depend on s . We will infinite loop in this case. To deal with cycles, we need uniform cost search, which we will describe later.

Dynamic programming



Key idea: state

A **state** is a summary of all the past actions sufficient to choose future actions **optimally**.

past actions (all cities) 1 3 4 6

state (current city) 1 3 4 6

- So far, we have only considered the example where the cost only depends on the current city. But let's try to capture exactly what's going on more generally.
- This is perhaps the most important idea of this lecture: **state**. A state is a summary of all the past actions sufficient to choose future actions optimally.
- What state is really about is forgetting the past. We can't forget everything because the action costs in the future might depend on what we did on the past. The more we forget, the fewer states we have, and the more efficient our algorithm. So the name of the game is to find the minimal set of states that suffice. It's a fun game.

Handling additional constraints

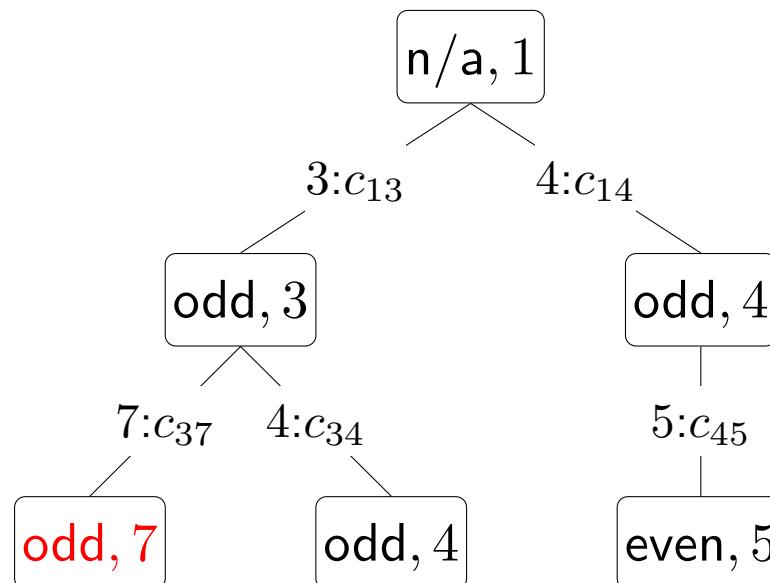


Example: route finding

Find the minimum cost path from city 1 to city n , only moving forward. It costs c_{ij} to go from i to j .

Constraint: Can't visit three odd cities in a row.

State: (whether previous city was odd, current city)



- Let's add a constraint that says we can't visit three odd cities in a row. If we only keep track of the current city, and we try to move to a next city, we cannot enforce this constraint because we don't know what the previous city was. So let's add the previous city into the state.
- This will work, but we can actually make the state smaller. We only need to keep track of whether the previous city was an odd numbered city to enforce this constraint.
- Note that in doing so, we have $2n$ states rather than n^2 states, which is a substantial savings. So the lesson is to pay attention to what information you actually need in the state.

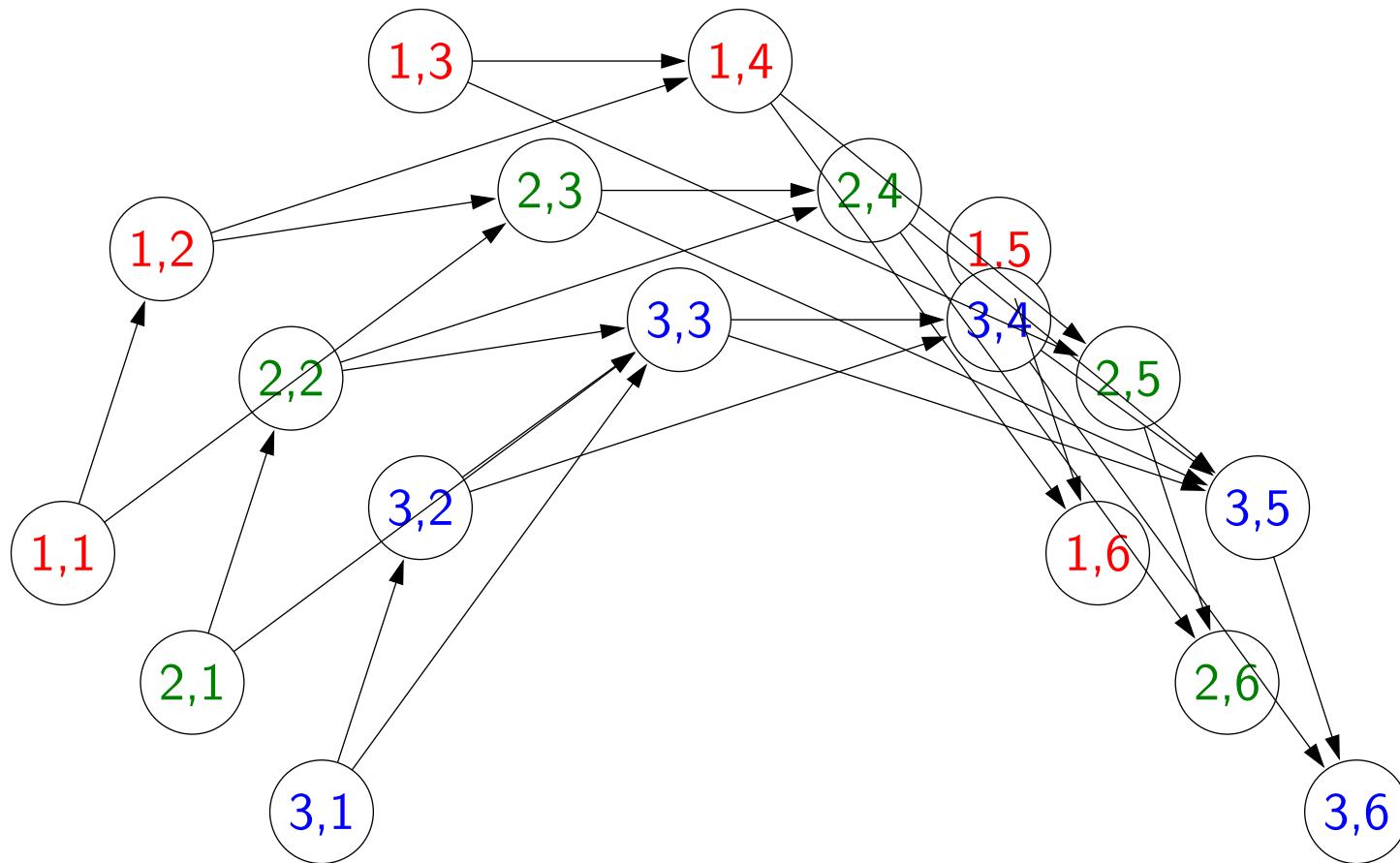


Question

Objective: travel from city 1 to city n , visiting at least 3 odd cities.
What is the minimal state?

State graph

State: $(\min(\text{number of odd cities visited}, 3), \text{current city})$



- Our first thought might be to remember how many odd cities we have visited so far (and the current city).
- But if we're more clever, we can notice that once the number of odd cities is 3, we don't need to keep track of whether that number goes up to 4 or 5, etc. So the state we actually need to keep is $(\min(\text{number of odd cities visited}, 3), \text{current city})$. Thus, our state space is $O(n)$ rather than $O(n^2)$.
- We can visualize what augmenting the state does to the state graph. Effectively, we are copying each node 4 times, and the edges are redirected to move between these copies.
- Note that some states such as $(2, 1)$ aren't reachable (if you're in city 1, it's impossible to have visited 2 odd cities already); the algorithm will not touch those states and that's perfectly okay.



Question

Objective: travel from city 1 to city n , visiting more odd than even cities.
What is the minimal state?

- An initial guess might be to keep track of the number of even cities and the number of odd cities visited.
- But we can do better. We have to just keep track of the number of odd cities minus the number of even cities and the current city. We can write this more formally as $(n_1 - n_2, \text{current city})$, where n_1 is the number of odd cities visited so far and n_2 is the number of even cities visited so far.



Summary

- **State:** summary of past actions sufficient to choose future actions optimally
- **Dynamic programming:** backtracking search with **memoization**
 - potentially exponential savings

Dynamic programming only works for acyclic graphs...what if there are cycles?



Roadmap

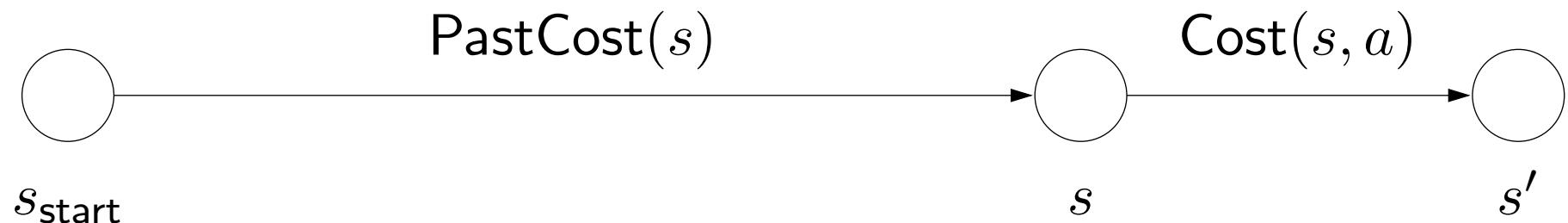
Tree search

Dynamic programming

Uniform cost search

Ordering the states

Observation: prefixes of optimal path are optimal



Key: if graph is acyclic, dynamic programming makes sure we compute $\text{PastCost}(s)$ before $\text{PastCost}(s')$

If graph is cyclic, then we need another mechanism to order states...

- Recall that we used dynamic programming to compute the future cost of each state s , the cost of the minimum cost path from s to a end state.
- We can analogously define $\text{PastCost}(s)$, the cost of the minimum cost path from the start state to s . If instead of having access to the successors via $\text{Succ}(s, a)$, we had access to predecessors (think of reversing the edges in the state graph), then we could define a dynamic program to compute all the $\text{PastCost}(s)$.
- Dynamic programming relies on the absence of cycles, so that there is always a clear order in which to compute all the past costs. If the past costs of all the predecessors of a state s are computed, then we could compute the past cost of s by taking the minimum.
- Note that $\text{PastCost}(s)$ will always be computed before $\text{PastCost}(s')$ if there is an edge from s to s' . In essence, the past costs will be computed according to a topological ordering of the nodes.
- However, when there are cycles, no topological ordering exists, so we need another way to order the states.

Uniform cost search (UCS)



Key idea: state ordering

UCS enumerates states in order of increasing past cost.



Assumption: non-negativity

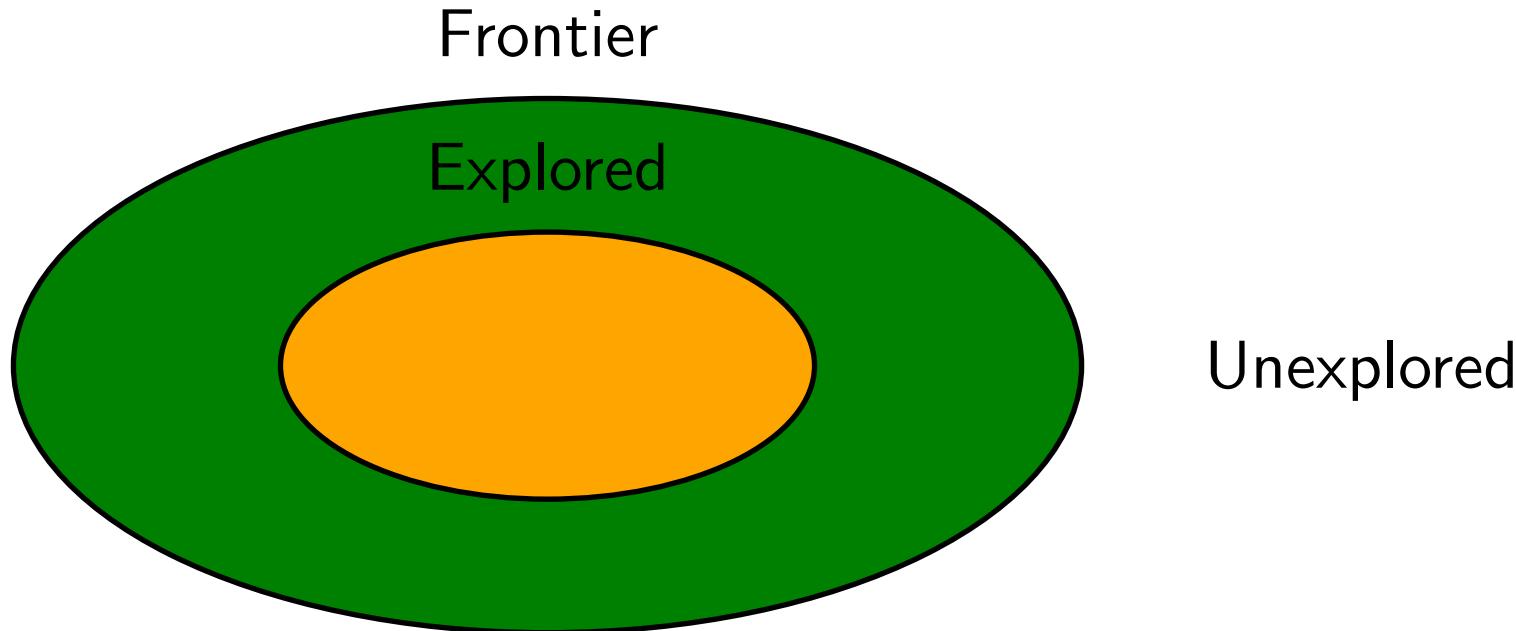
All action costs are non-negative: $\text{Cost}(s, a) \geq 0$.

UCS in action:



- The key idea that uniform cost search (UCS) uses is to compute the past costs in order of increasing past cost. To make this efficient, we need to make an important assumption that all action costs are non-negative.
- This assumption is reasonable in many cases, but doesn't allow us to handle cases where actions have payoff. To handle negative costs (positive payoffs), we need the Bellman-Ford algorithm. When we talk about value iteration for MDPs, we will see a form of this algorithm.
- Note: those of you who have studied algorithms should immediately recognize UCS as Dijkstra's algorithm. Logically, the two are indeed equivalent. There is an important implementation difference: UCS takes as input a **search problem**, which implicitly defines a large and even infinite graph, whereas Dijkstra's algorithm (in the typical exposition) takes as input a fully concrete graph. The implicitness is important in practice because we might be working with an enormous graph (a detailed map of world) but only need to find the path between two close by points (Stanford to Palo Alto).
- Another difference is that Dijkstra's algorithm is usually thought of as finding the shortest path from the start state to every other node, whereas UCS is explicitly about finding the shortest path to an end state. This difference is sharpened when we look at the A* algorithm next time, where knowing that we're trying to get to the goal can yield a much faster algorithm. The name uniform cost search refers to the fact that we are exploring states of the same past cost uniformly (the video makes this visually clear); in contrast, A* will explore states which are biased towards the end state.

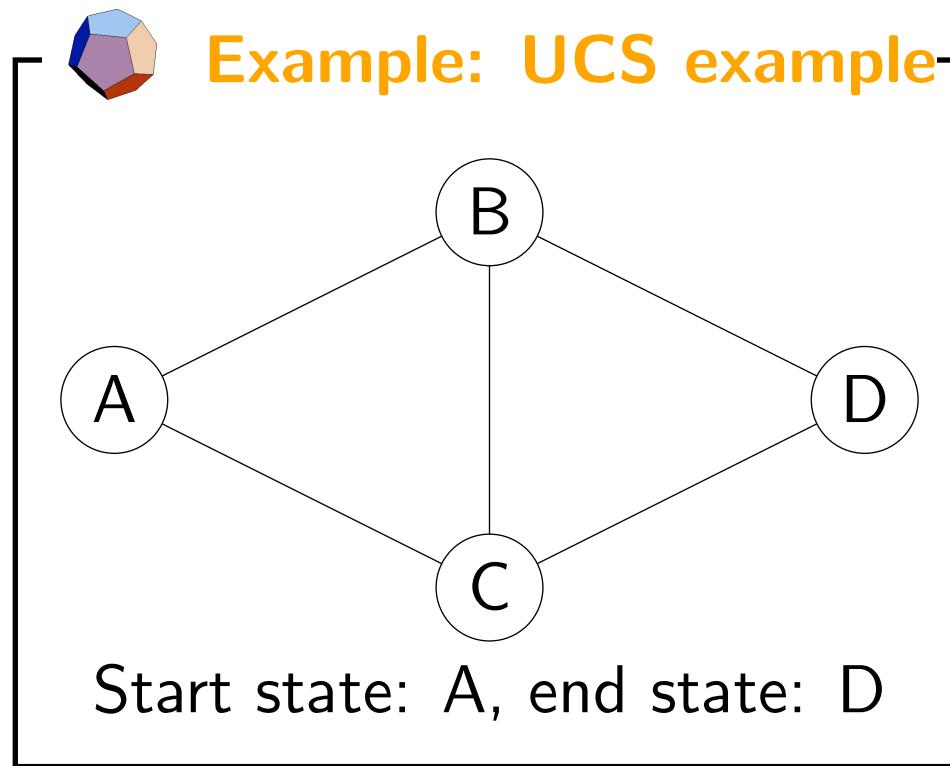
High-level strategy



- **Explored:** states we've found the optimal path to
- **Frontier:** states we've seen, still figuring out how to get there cheaply
- **Unexplored:** states we haven't seen

- The general strategy of UCS is to maintain three sets of nodes: explored, frontier, and unexplored. Throughout the course of the algorithm, we will move states from unexplored to frontier, and from frontier to explored.
- The key invariant is that we have computed the minimum cost paths to all the nodes in the explored set. So when the end state moves into the explored set, then we are done.

Uniform cost search example



[whiteboard]

Minimum cost path:

$A \rightarrow B \rightarrow C \rightarrow D$ with cost 3

- Before we present the full algorithm, let's walk through a concrete example.
- Initially, we put A on the frontier. We then take A off the frontier and mark it as explored. We add B and C to the frontier with past costs 1 and 100, respectively.
- Next, we remove from the frontier the state with the minimum past cost (priority), which is B. We mark B as explored and consider successors A, C, D. We ignore A since it's already explored. The past cost of C gets updated from 100 to 2. We add D to the frontier with initial past cost 101.
- Next, we remove C from the frontier; its successors are A, B, D. A and B are already explored, so we only update D's past cost from 101 to 3.
- Finally, we pop D off the frontier, find that it's a end state, and terminate the search.

Uniform cost search (UCS)



Algorithm: uniform cost search [Dijkstra, 1956]

Add s_{start} to **frontier** (priority queue)

Repeat until frontier is empty:

 Remove s with smallest priority p from frontier

 If $\text{IsEnd}(s)$: return solution

 Add s to **explored**

 For each action $a \in \text{Actions}(s)$:

 Get successor $s' \leftarrow \text{Succ}(s, a)$

 If s' already in explored: continue

 Update **frontier** with s' and priority $p + \text{Cost}(s, a)$

[semi-live solution: Uniform Cost Search]

- Implementation note: we use `util.PriorityQueue` which supports `removeMin` and `update`. Note that `frontier.update(state, pastCost)` returns whether `pastCost` improves the existing estimate of the past cost of state.

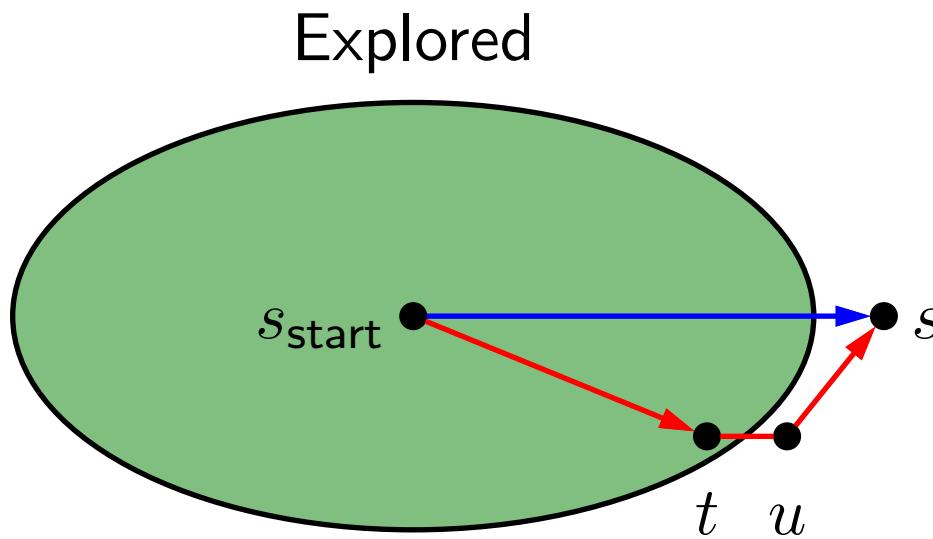
Analysis of uniform cost search



Theorem: correctness

When a state s is popped from the frontier and moved to explored, its priority is $\text{PastCost}(s)$, the minimum cost to s .

Proof:



- Let p_s be the priority of s when s is popped off the frontier. Since all costs are non-negative, p_s increases over the course of the algorithm.
- Suppose we pop s off the frontier. Let the blue path denote the path with cost p_s .
- Consider any alternative red path from the start state to s . The red path must leave the explored region at some point; let t and $u = \text{Succ}(t, a)$ be the first pair of states straddling the boundary. We want to show that the red path cannot be cheaper than the blue path via a string of inequalities.
- First, by definition of $\text{PastCost}(t)$ and non-negativity of edge costs, the cost of the red path is at least the cost of the part leading to u , which is $\text{PastCost}(t) + \text{Cost}(t, a) = p_t + \text{Cost}(t, a)$, where the last equality is by the inductive hypothesis.
- Second, we have $p_t + \text{Cost}(t, a) \geq p_u$ since we updated the frontier based on (t, a) .
- Third, we have that $p_u \geq p_s$ because s was the minimum cost state on the frontier.
- Note that p_s is the cost of the blue path.

DP versus UCS

N total states, n of which are closer than end state

Algorithm	Cycles?	Action costs	Time/space
DP	no	any	$O(N)$
UCS	yes	≥ 0	$O(n \log n)$

Note: UCS potentially explores fewer states, but requires more overhead to maintain the priority queue

Note: assume number of actions per state is constant (independent of n and N)

- DP and UCS have complementary strengths and weaknesses; neither dominates the other.
- DP can handle negative action costs, but is restricted to acyclic graphs. It also explores all N reachable states from s_{start} , which is inefficient. This is unavoidable due to negative action costs.
- UCS can handle cyclic graphs, but is restricted to non-negative action costs. An advantage is that it only needs to explore n states, where n is the number of states which are cheaper to get to than any end state. However, there is an overhead with maintaining the priority queue.
- One might find it unsatisfying that UCS can only deal with non-negative action costs. Can we just add a large positive constant to each action cost to make them all non-negative? It turns out this doesn't work because it penalizes longer paths more than shorter paths, so we would end up solving a different problem.



Summary

- Tree search: memory efficient, suitable for huge state spaces but exponential worst-case running time
- State: summary of past actions sufficient to choose future actions optimally
- Graph search: dynamic programming and uniform cost search construct optimal paths (exponential savings!)
- Next time: learning action costs, searching faster with A*

- We started out with the idea of a search problem, an abstraction that provides a clean interface between modeling and algorithms.
- Tree search algorithms are the simplest: just try exploring all possible states and actions. With backtracking search and DFS with iterative deepening, we can scale up to huge state spaces since the memory usage only depends on the number of actions in the solution path. Of course, these algorithms necessarily take exponential time in the worst case.
- To do better, we need to think more about bookkeeping. The most important concept from this lecture is the idea of a **state**, which contains all the information about the past to act optimally in the future. We saw several examples of traveling between cities under various constraints, where coming up with the proper minimal state required a deep understanding of search.
- With an appropriately defined state, we can apply either dynamic programming or UCS, which have complementary strengths. The former handles negative action costs and the latter handles cycles. Both require space proportional to the number of states, so we need to make sure that we did a good job with the modeling of the state.



Search II





Question

Suppose we want to travel from city 1 to city n (going only forward) and back to city 1 (only going backward). It costs $c_{ij} \geq 0$ to go from i to j . Which of the following algorithms can be used to find the minimum cost path (select all that apply)?

depth-first search

breadth-first search

dynamic programming

uniform cost search

- Let's first start by figuring out what the search problem actually is. Any action sequence needs to satisfy the constraint that we move forward to n and then move backwards to 1. So we need to keep track of the current city i as well as the direction in the state.
- We can write down the details, but all that matters for this question is that the graph is acyclic (note that the graph implied by c_{ij} over cities is not acyclic, but keeping track of directionality makes it acyclic). Also, all edge costs are non-negative.
- Now, let's think about which algorithms will work. Recall the various assumptions of the algorithms. DFS won't work because it assumes all edge costs are zero. BFS also won't work because it assumes all edge costs are the same. Dynamic programming will work because the graph is acyclic. Uniform cost search will also work because all the edge costs are non-negative.



Key idea: state

A **state** is a summary of all the past actions sufficient to choose future actions **optimally**.

past actions (all cities) 1 3 4 6 5 3

state (current city) 1 3 ← 6 5 3

Review



Definition: search problem

- s_{start} : starting state
- $\text{Actions}(s)$: possible actions
- $\text{Cost}(s, a)$: action cost
- $\text{Succ}(s, a)$: successor
- $\text{IsEnd}(s)$: reached end state?

Objective: find the minimum cost path from s_{start} to an s satisfying $\text{IsEnd}(s)$.

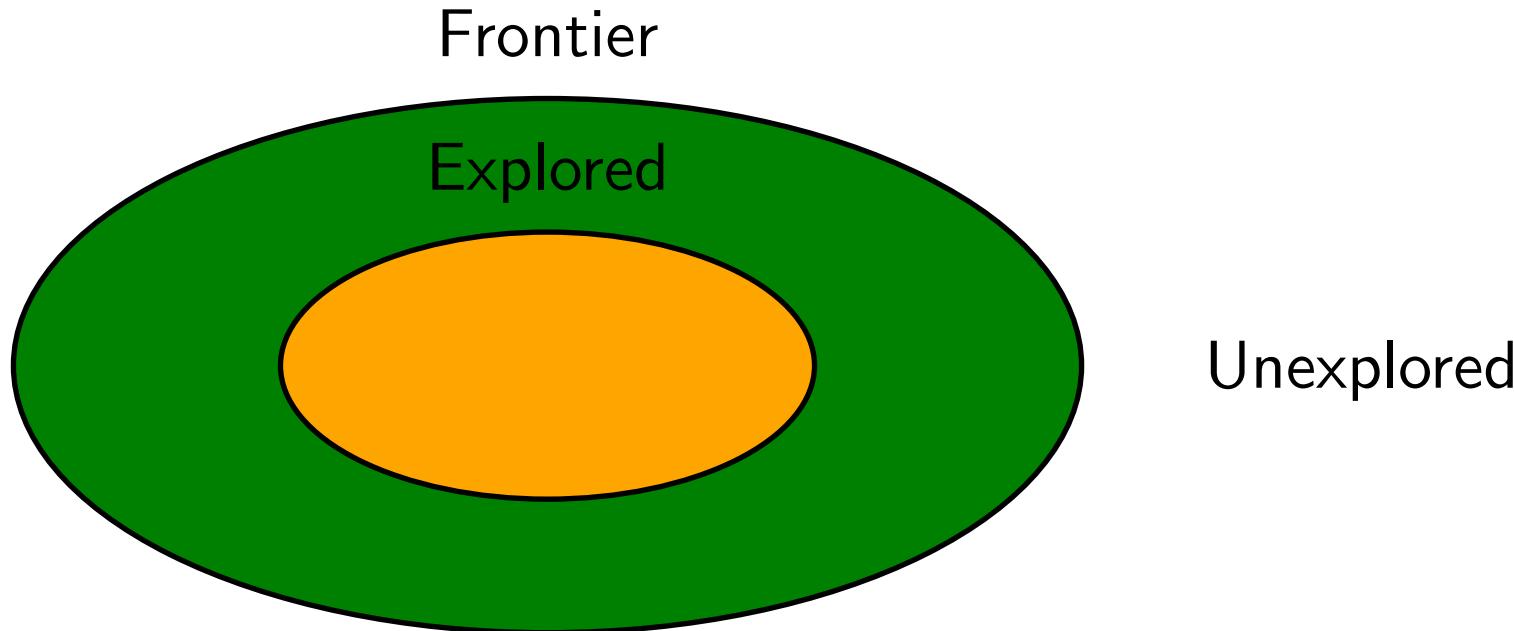
Paradigm

Modeling

Inference

Learning

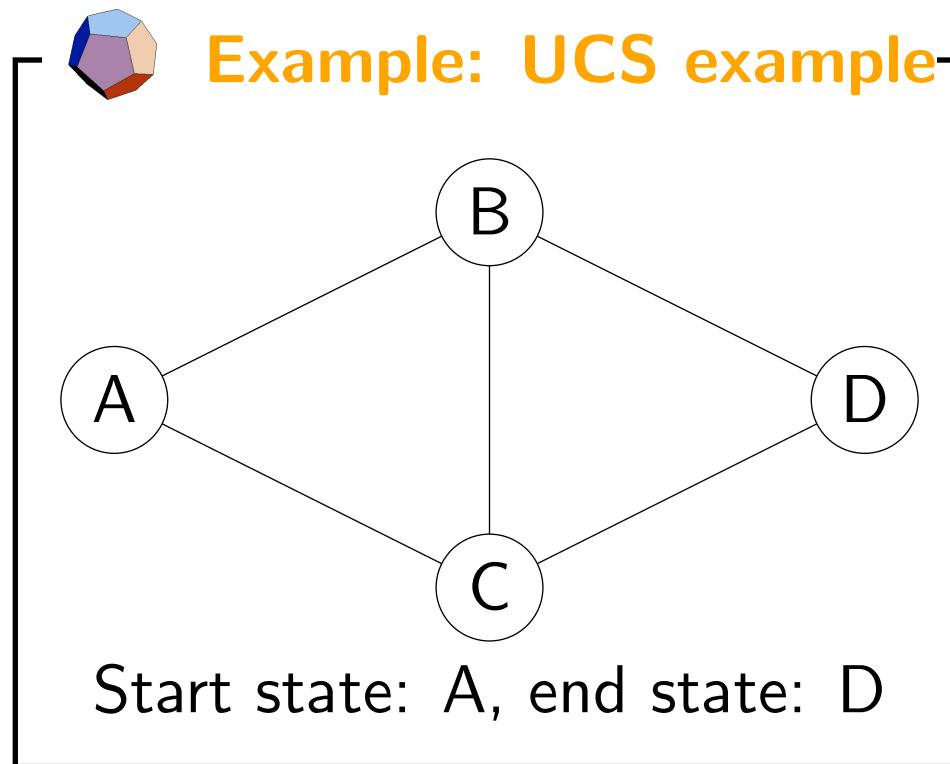
High-level strategy



- **Explored:** states we've found the optimal path to
- **Frontier:** states we've seen, still figuring out how to get there cheaply
- **Unexplored:** states we haven't seen

- The general strategy of UCS is to maintain three sets of nodes: explored, frontier, and unexplored. Throughout the course of the algorithm, we will move states from unexplored to frontier, and from frontier to explored.
- The key invariant is that we have computed the minimum cost paths to all the nodes in the explored set. So when the end state moves into the explored set, then we are done.

Uniform cost search example



[whiteboard]

Minimum cost path:

$A \rightarrow B \rightarrow C \rightarrow D$ with cost 3

- Before we present the full algorithm, let's walk through a concrete example.
- Initially, we put A on the frontier. We then take A off the frontier and mark it as explored. We add B and C to the frontier with past costs 1 and 100, respectively.
- Next, we remove from the frontier the state with the minimum past cost (priority), which is B. We mark B as explored and consider successors A, C, D. We ignore A since it's already explored. The past cost of C gets updated from 100 to 2. We add D to the frontier with initial past cost 101.
- Next, we remove C from the frontier; its successors are A, B, D. A and B are already explored, so we only update D's past cost from 101 to 3.
- Finally, we pop D off the frontier, find that it's a end state, and terminate the search.

DP versus UCS

N total states, n of which are closer than end state

Algorithm	Cycles?	Action costs	Time/space
DP	no	any	$O(N)$
UCS	yes	≥ 0	$O(n \log n)$

Note: UCS potentially explores fewer states, but requires more overhead to maintain the priority queue

Note: assume number of actions per state is constant (independent of n and N)

- DP and UCS have complementary strengths and weaknesses; neither dominates the other.
- DP can handle negative action costs, but is restricted to acyclic graphs. It also explores all N reachable states from s_{start} , which is inefficient. This is unavoidable due to negative action costs.
- UCS can handle cyclic graphs, but is restricted to non-negative action costs. An advantage is that it only needs to explore n states, where n is the number of states which are cheaper to get to than any end state. However, there is an overhead with maintaining the priority queue.
- One might find it unsatisfying that UCS can only deal with non-negative action costs. Can we just add a large positive constant to each action cost to make them all non-negative? It turns out this doesn't work because it penalizes longer paths more than shorter paths, so we would end up solving a different problem.

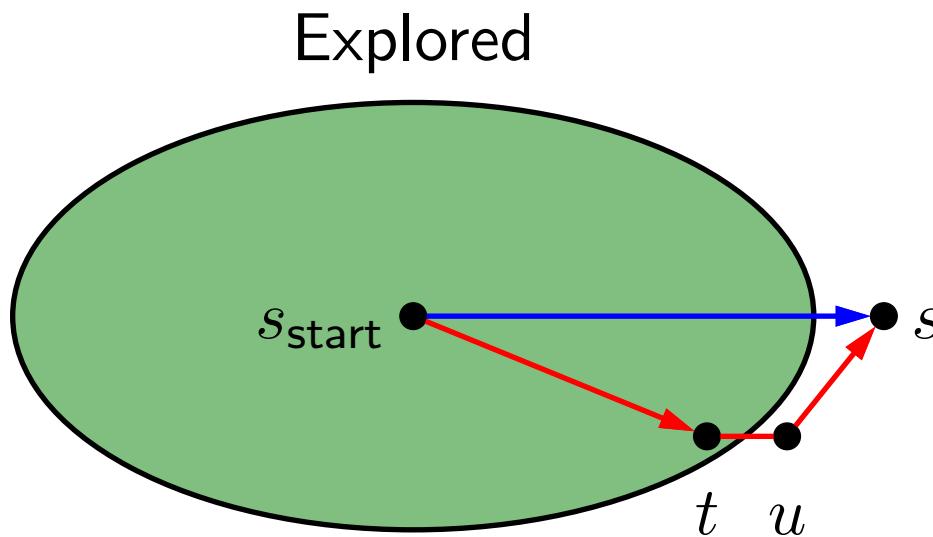
Analysis of uniform cost search



Theorem: correctness

When a state s is popped from the frontier and moved to explored, its priority is $\text{PastCost}(s)$, the minimum cost to s .

Proof:



- Let p_s be the priority of s when s is popped off the frontier. Since all costs are non-negative, p_s increases over the course of the algorithm.
- Suppose we pop s off the frontier. Let the blue path denote the path with cost p_s .
- Consider any alternative red path from the start state to s . The red path must leave the explored region at some point; let t and $u = \text{Succ}(t, a)$ be the first pair of states straddling the boundary. We want to show that the red path cannot be cheaper than the blue path via a string of inequalities.
- First, by definition of $\text{PastCost}(t)$ and non-negativity of edge costs, the cost of the red path is at least the cost of the part leading to u , which is $\text{PastCost}(t) + \text{Cost}(t, a) = p_t + \text{Cost}(t, a)$, where the last equality is by the inductive hypothesis.
- Second, we have $p_t + \text{Cost}(t, a) \geq p_u$ since we updated the frontier based on (t, a) .
- Third, we have that $p_u \geq p_s$ because s was the minimum cost state on the frontier.
- Note that p_s is the cost of the blue path.

Paradigm

Modeling

Inference

Learning



Roadmap

Learning costs

A* search

Relaxation



Search

Transportation example

Start state: 1

Walk action: from s to $s + 1$ (cost: 1)

Tram action: from s to $2s$ (cost: 2)

End state: n



search algorithm

walk walk tram tram tram walk tram tram

(minimum cost path)

- Recall the magic tram example from the last lecture. Given a search problem (specification of the start state, end test, actions, successors, and costs), we can use a search algorithm (DP or UCS) to yield a solution, which is a sequence of actions of minimum cost reaching an end state from the start state.



Learning

Transportation example

Start state: 1

Walk action: from s to $s + 1$ (cost: ?)

Tram action: from s to $2s$ (cost: ?)

End state: n

walk walk tram tram tram walk tram tram



learning algorithm

walk cost: 1, tram cost: 2

- Now suppose we don't know what the costs are, but we observe someone getting from 1 to n via some sequence of walking and tram-taking. Can we figure out what the costs are? This is the goal of learning.

Learning as an inverse problem

Forward problem (search):

$$\text{Cost}(s, a) \longrightarrow (a_1, \dots, a_k)$$

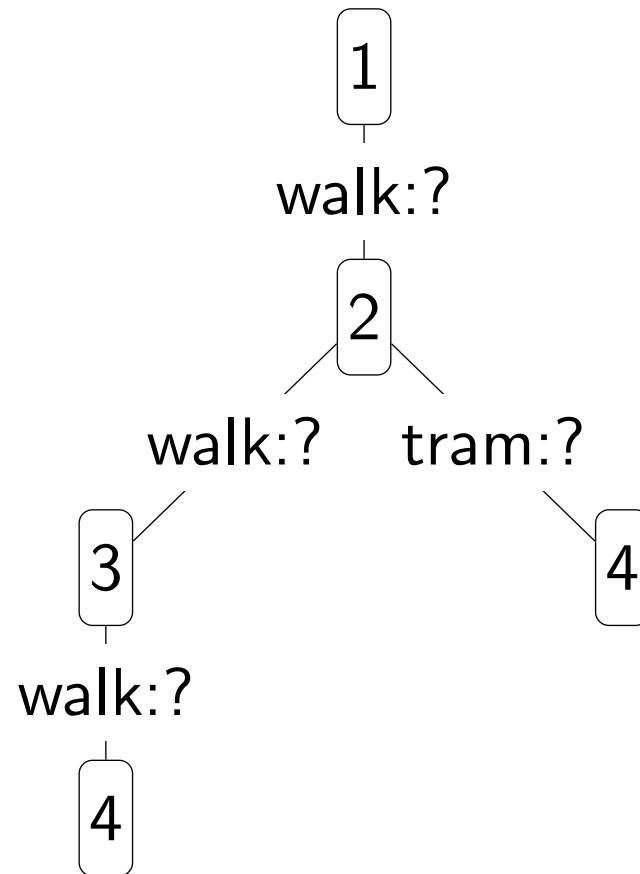
Inverse problem (learning):

$$(a_1, \dots, a_k) \longrightarrow \text{Cost}(s, a)$$

- More generally, so far we have thought about search as a "forward" problem: given costs, finding the optimal sequence of actions.
- Learning concerns the "inverse" problem: given the desired sequence of actions, reverse engineer the costs.

Prediction (inference) problem

Input x : search problem without costs



Output y : solution path

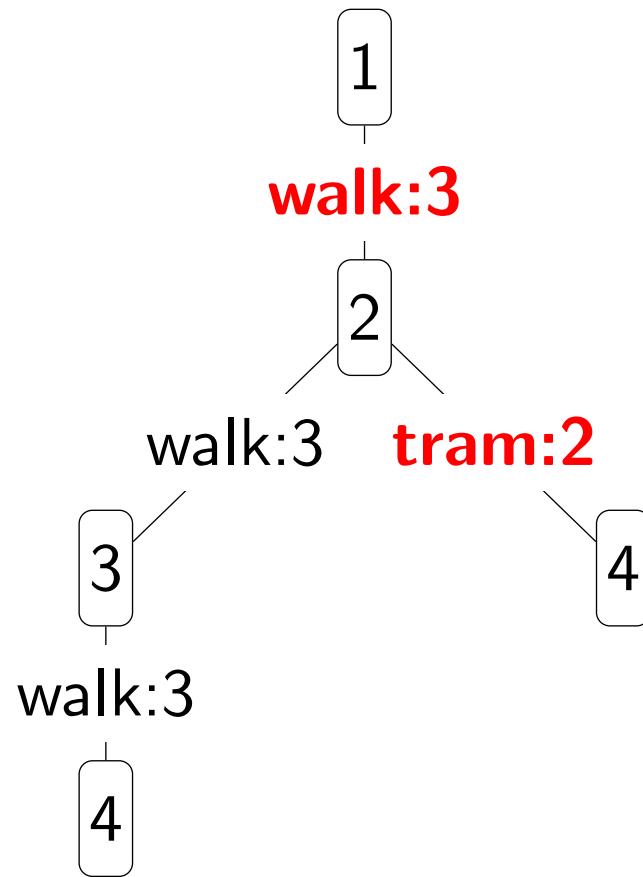
walk walk walk

- Let's cast the problem as predicting an output y given an input x . Here, the input x is the search problem (visualized as a search tree) without the costs provided. The output y is the desired solution path. The question is what the costs should be set to so that y is actually the minimum cost path of the resulting search problem.

Tweaking costs

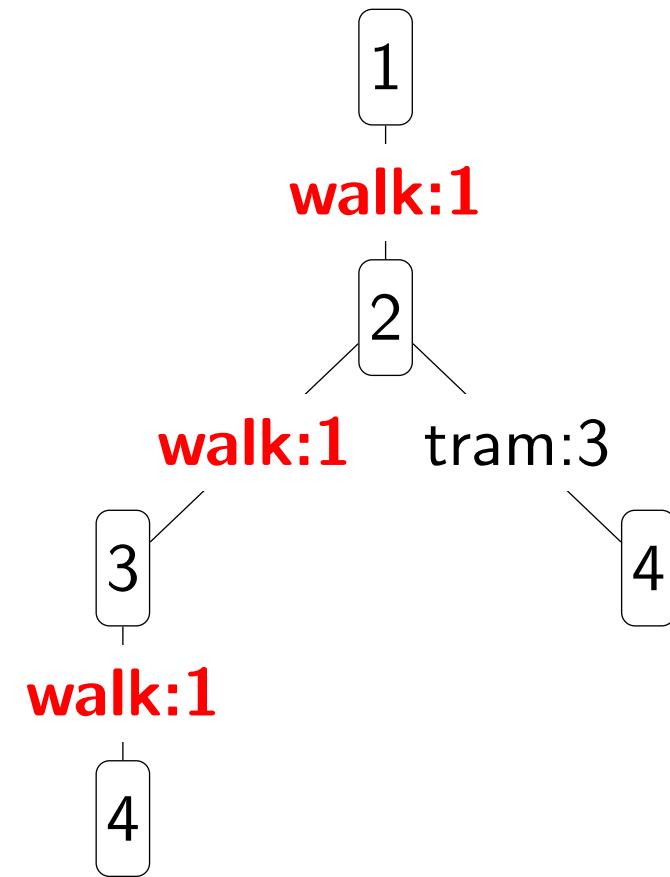
Costs: {walk:3, tram:2}

Minimum cost path:



Costs: {walk:1, tram:3}

Minimum cost path:



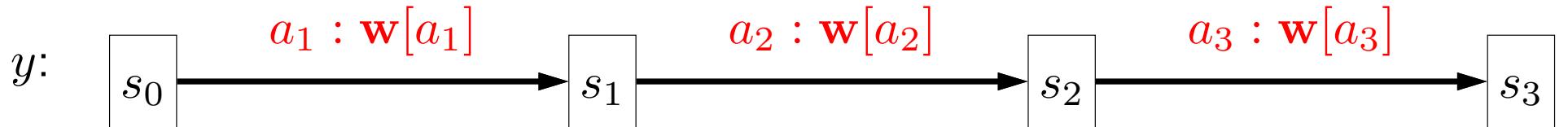
- Suppose the walk cost is 3 and the tram cost is 2. Then, we would obviously predict the [walk, tram] path, which has lower cost.
- But this is not our desired output, because we actually saw the person walk all the way from 1 to 4. How can we update the action costs so that the minimum cost path is walking?
- Intuitively, we want the tram cost to be more and the walk cost to be less. Specifically, let's increase the cost of every action on the predicted path and decrease the cost of every action on the true path. Now, the predicted path coincides with the true observed path. Is this a good strategy in general?

Modeling costs (simplified)

Assume costs depend only on the action:

$$\text{Cost}(s, a) = \mathbf{w}[a]$$

Candidate output path:



Path cost:

$$\text{Cost}(y) = \mathbf{w}[a_1] + \mathbf{w}[a_2] + \mathbf{w}[a_3]$$

- For each action a , we define a weight $w[a]$ representing the cost of action a . Without loss of generality, let us assume that the cost of the action does not depend on the state s .
- Then the cost of a path y is simply the sum of the weights of the actions on the path. Every path has some cost, and recall that the search algorithm will return the minimum cost path.

Learning algorithm



Algorithm: Structured Perceptron (simplified)

- For each action: $\mathbf{w}[a] \leftarrow 0$
- For each iteration $t = 1, \dots, T$:
 - For each training example $(x, y) \in \mathcal{D}_{\text{train}}$:
 - Compute the minimum cost path y' given \mathbf{w}
 - For each action $a \in y$: $\mathbf{w}[a] \leftarrow \mathbf{w}[a] - 1$
 - For each action $a \in y'$: $\mathbf{w}[a] \leftarrow \mathbf{w}[a] + 1$
- Try to decrease cost of true y (from training data)
- Try to increase cost of predicted y' (from search)

[semi-live solution]

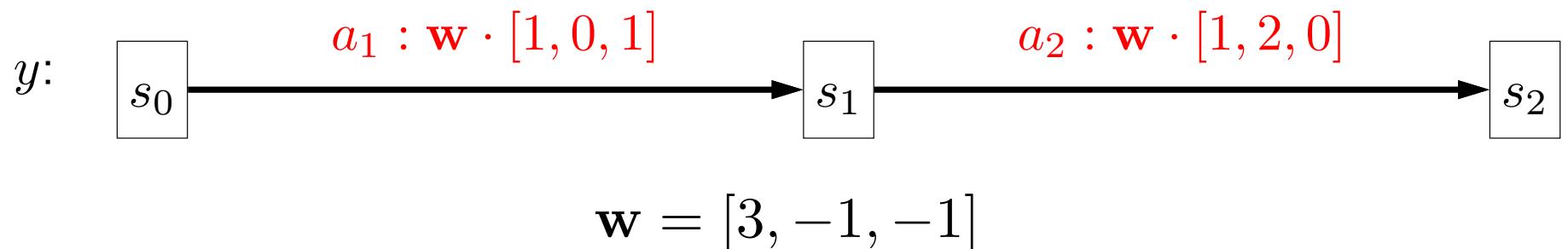
- We are now in position to state the (simplified version of) **structured Perceptron** algorithm.
- Advanced: the Perceptron algorithm performs stochastic gradient descent (SGD) on a modified hinge loss with a constant step size of $\eta = 1$. The modified hinge loss is $\text{Loss}(x, y, \mathbf{w}) = \max\{-(\mathbf{w} \cdot \phi(x))y, 0\}$, where the margin of 1 has been replaced with a zero. The structured Perceptron is a generalization of the Perceptron algorithm, which is stochastic gradient descent on $\text{Loss}(x, y, \mathbf{w}) = \max_{y'} \{\sum_{a \in y} \mathbf{w}[a] - \sum_{a \in y'} \mathbf{w}[a]\}$ (note the relationship to the multiclass hinge loss). Even if you don't really understand the loss function, you can still understand the algorithm, since it is very intuitive.
- We iterate over the training examples. Each (x, y) is a tuple where x is a search problem without costs and y is the true minimum-cost path. Given the current weights w (action costs), we run a search algorithm to find the minimum-cost path y' according to those weights. Then we update the weights to favor actions that appear in the correct output y (by reducing their costs) and disfavor actions that appear in the predicted output y' (by increasing their costs). Note that if we are not making a mistake (that is, if $y = y'$), then there is no update.
- Collins (2002) proved (based on the proof of the original Perceptron algorithm) that if there exists a weight vector that will make zero mistakes on the training data, then the Perceptron algorithm will converge to one of those weight vectors in a finite number of iterations.

Generalization to features (skip)

Costs are parametrized by feature vector:

$$\text{Cost}(s, a) = \mathbf{w} \cdot \phi(s, a)$$

Example:



Path cost:

$$\text{Cost}(y) = 2 + 1 = 3$$

- So far, the cost of an action a is simply $\mathbf{w}[a]$. We can generalize this to allow the cost to be a general dot product $\mathbf{w} \cdot \phi(s, a)$, which (i) allows the features to depend on both the state and the action and (ii) allows multiple features per edge. For example, we can have different costs for walking and tram-taking depending on which part of the city we are in.
- We can equivalently write the cost of an entire output y as $\mathbf{w} \cdot \phi(y)$, where $\phi(y) = \phi(s_0, a_1) + \phi(s_1, a_2)$ is the sum of the feature vectors over all actions.

Learning algorithm (skip)



Algorithm: Structured Perceptron [Collins, 2002]

- For each action: $\mathbf{w} \leftarrow 0$
- For each iteration $t = 1, \dots, T$:
 - For each training example $(x, y) \in \mathcal{D}_{\text{train}}$:
 - Compute the minimum cost path y' given \mathbf{w}
 - $\mathbf{w} \leftarrow \mathbf{w} - \phi(y) + \phi(y')$
 - Try to decrease cost of true y (from training data)
 - Try to increase cost of predicted y' (from search)

Applications

- Part-of-speech tagging

Fruit flies like a banana.  Noun Noun Verb Det Noun

- Machine translation

la maison bleue  *the blue house*

- The structured Perceptron was first used for natural language processing tasks. Given it's simplicity, the Perceptron works reasonably well. With a few minor tweaks, you get state-of-the-art algorithms for structured prediction, which can be applied to many tasks such as machine translation, gene prediction, information extraction, etc.
- On a historical note, the structured Perceptron merges two relatively classic communities. The first is search algorithms (uniform cost search was developed by Dijkstra in 1956). The second is machine learning (Perceptron was developed by Rosenblatt in 1957). It was only over 40 years later that the two met.



Roadmap

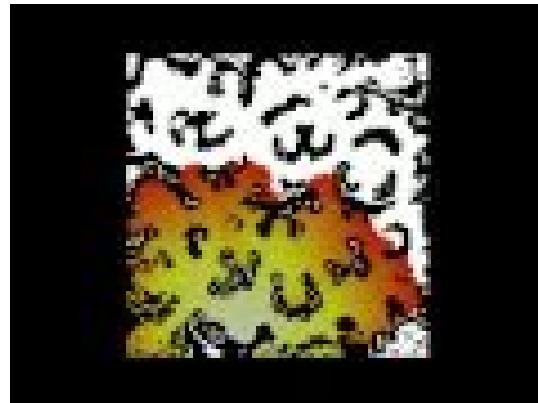
Learning costs

A* search

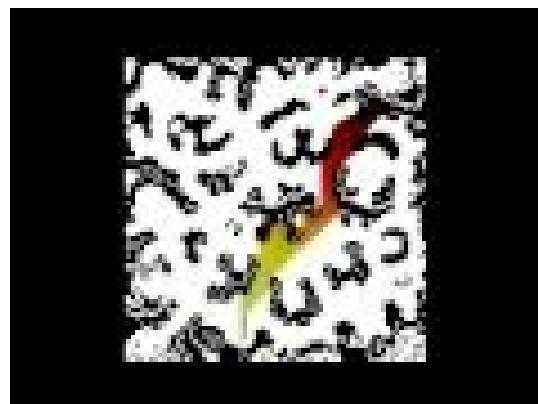
Relaxation

A* algorithm

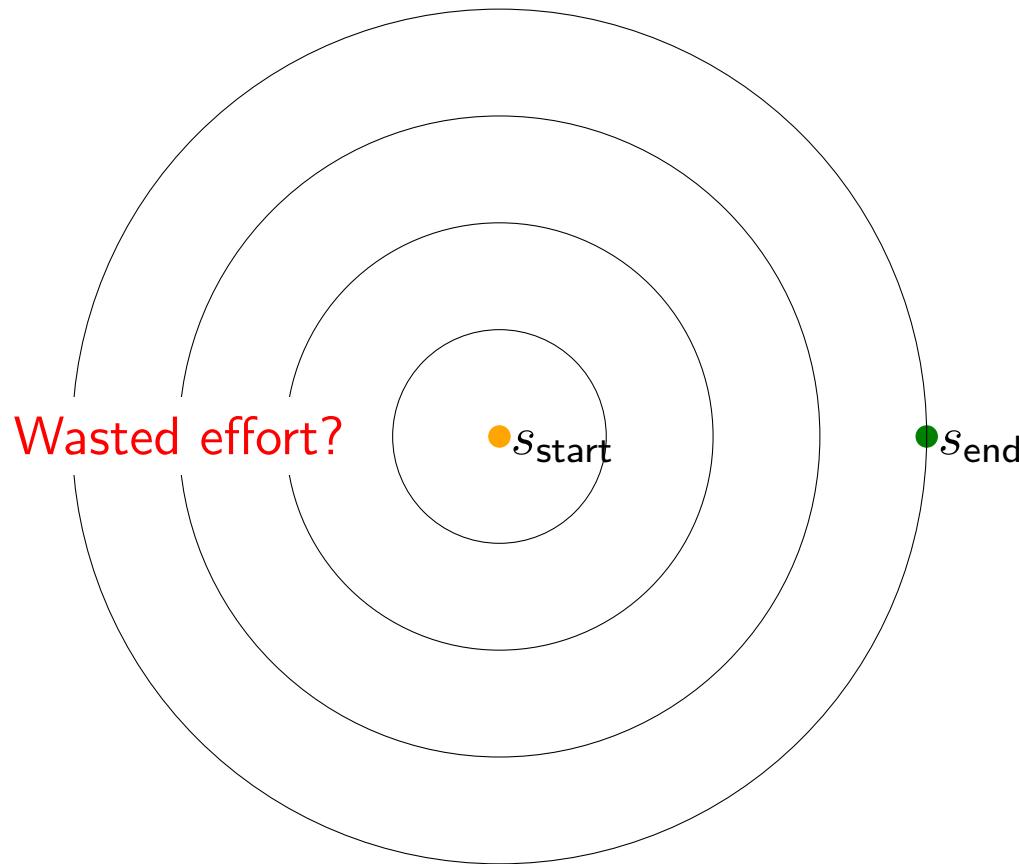
UCS in action:



A* in action:



Can uniform cost search be improved?



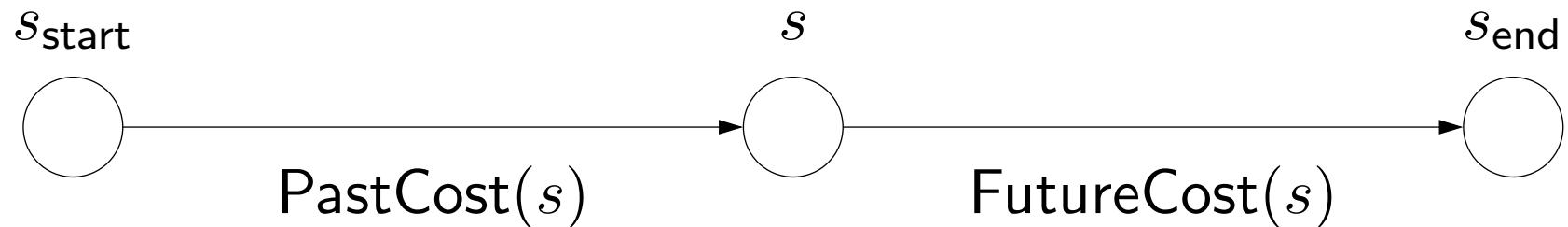
Problem: UCS orders states by cost from s_{start} to s

Goal: take into account cost from s to s_{end}

- Now our goal is to make UCS faster. If we look at the UCS algorithm, we see that it explores states based on how far they are away from the start state. As a result, it will explore many states which are close to the start state, but in the opposite direction of the end state.
- Intuitively, we'd like to bias UCS towards exploring states which are closer to the end state, and that's exactly what A* does.

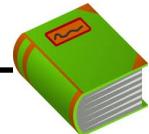
Exploring states

UCS: explore states in order of $\text{PastCost}(s)$



Ideal: explore in order of $\text{PastCost}(s) + \text{FutureCost}(s)$

A*: explore in order of $\text{PastCost}(s) + h(s)$



Definition: Heuristic function

A heuristic $h(s)$ is any estimate of $\text{FutureCost}(s)$.

- First, some terminology: $\text{PastCost}(s)$ is the minimum cost from the start state to s , and $\text{FutureCost}(s)$ is the minimum cost from s to an end state. Without loss of generality, we can just assume we have one end state. (If we have multiple ones, create a new official goal state which is the successor of all the original end states.)
- Recall that UCS explores states in order of $\text{PastCost}(s)$. It'd be nice if we could explore states in order of $\text{PastCost}(s) + \text{FutureCost}(s)$, which would definitely take the end state into account, but computing $\text{FutureCost}(s)$ would be as expensive as solving the original problem.
- A* relies on a **heuristic** $h(s)$, which is an estimate of $\text{FutureCost}(s)$. For A* to work, $h(s)$ must satisfy some conditions, but for now, just think of $h(s)$ as an approximation. We will soon show that A* will explore states in order of $\text{PastCost}(s) + h(s)$. This is nice, because now states which are estimated (by $h(s)$) to be really far away from the end state will be explored later, even if their $\text{PastCost}(s)$ is small.

A* search



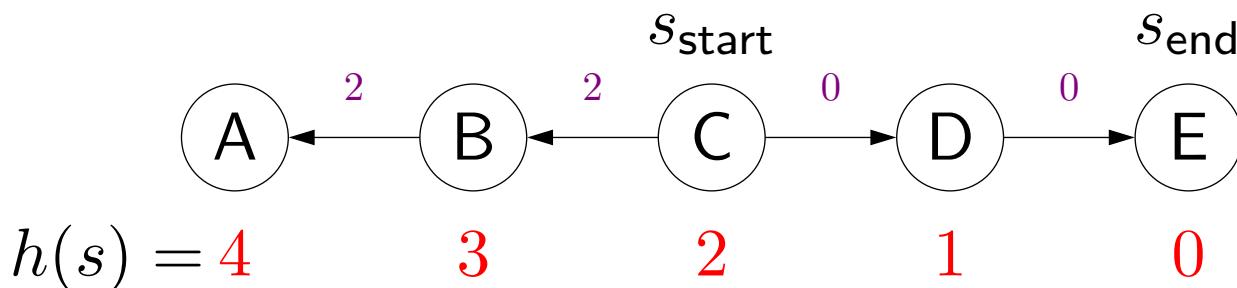
Algorithm: A* search [Hart/Nilsson/Raphael, 1968]

Run uniform cost search with **modified edge costs**:

$$\text{Cost}'(s, a) = \text{Cost}(s, a) + h(\text{Succ}(s, a)) - h(s)$$

Intuition: add a penalty for how much action a takes us away from the end state

Example:



$$\text{Cost}'(C, B) = \text{Cost}(C, B) + h(B) - h(C) = 1 + (3 - 2) = 2$$

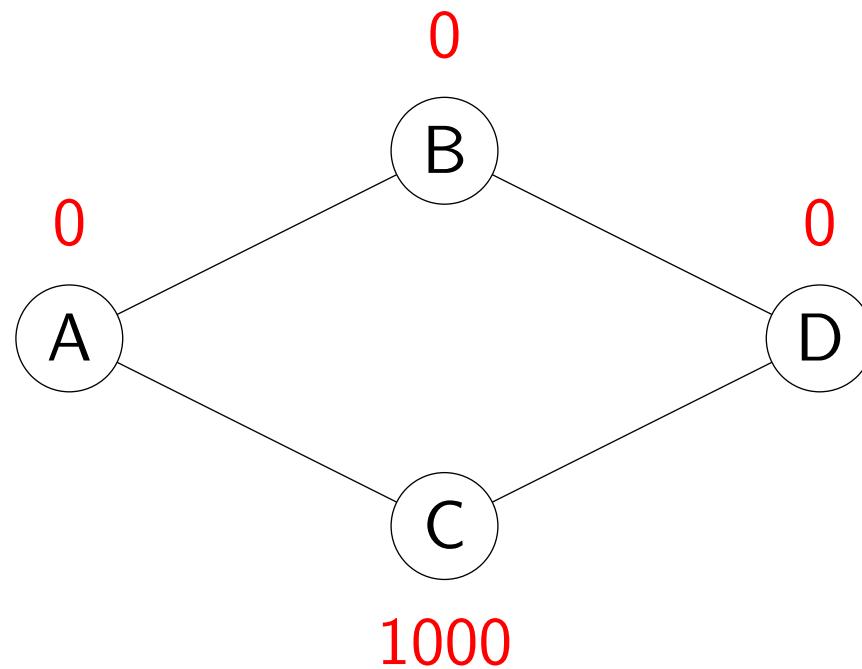
- Here is the full A* algorithm: just run UCS with modified edge costs.
- You might feel tricked because we promised you a shiny new algorithm, but actually, you just got a refurbished version of UCS. (This is a slightly unorthodox presentation of A*. The normal presentation is modifying UCS to prioritize by $\text{PastCost}(s) + h(s)$ rather than $\text{PastCost}(s)$.) But I think the modified edge costs view shows a deeper connection to UCS, and we don't even have to modify the UCS code at all.
- How should we think of these modified edge costs? It's the same edge cost $\text{Cost}(s, a)$ plus an additional term. This term is difference between the estimated future cost of the new state $\text{Succ}(s, a)$ and that of the current state s . In other words, we're measuring how much farther from the end state does action a take us. If this difference is positive, then we're penalizing the action a more. If this difference is negative, then we're favoring this action a .
- Let's look at a small example. All edge costs are 1. Let's suppose we define $h(s)$ to be the actual $\text{FutureCost}(s)$, the minimum cost to the end state. In general, this is not the case, but let's see what happens in the best case. The modified edge costs are 2 for actions moving away from the end state and 0 for actions moving towards the end state.
- In this case, UCS with original edge costs 1 will explore all the nodes. However, A* (UCS with modified edge costs) will explore only the three nodes on the path to the end state.

An example heuristic

Will any heuristic work?

No.

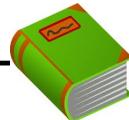
Counterexample:



Doesn't work because of **negative modified edge costs!**

- So far, we've just said that $h(s)$ is just an approximation of $\text{FutureCost}(s)$. But can it be any approximation?
- The answer is no, as the counterexample clearly shows. The modified edge costs would be 1 (A to B), 1002 (A to C), 5 (B to D), and -999 (C to D). UCS would go to B first and then to D, finding a cost 6 path rather than the optimal cost 3 path through C.
- If our heuristic is lying to us (bad approximation of future costs), then running A* (UCS on modified costs) could lead to a suboptimal solution. Note that the reason this heuristic doesn't work is the same reason UCS doesn't work when there are negative action costs.

Consistent heuristics

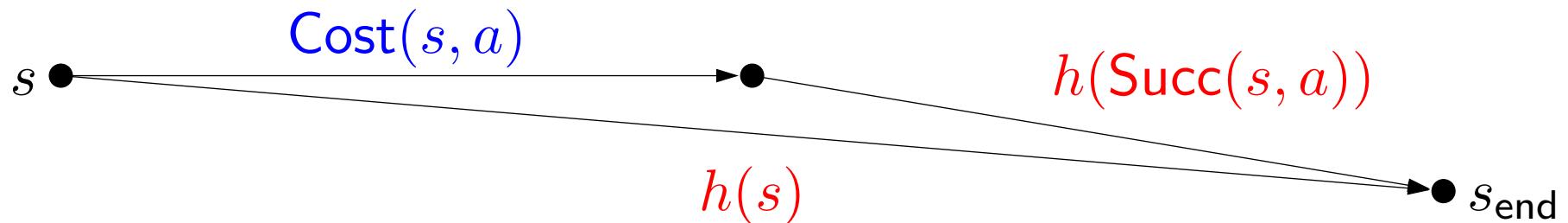


Definition: consistency

A heuristic h is **consistent** if

- $\text{Cost}'(s, a) = \text{Cost}(s, a) + h(\text{Succ}(s, a)) - h(s) \geq 0$
- $h(s_{\text{end}}) = 0$.

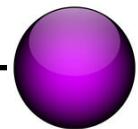
Condition 1: needed for UCS to work (triangle inequality).



Condition 2: $\text{FutureCost}(s_{\text{end}}) = 0$ so match it.

- We need $h(s)$ to be **consistent**, which means two things. First, the modified edge costs are non-negative (this is the main property). This is important for UCS to find the minimum cost path (remember that UCS only works when all the edge costs are non-negative).
- Second, $h(s_{\text{end}}) = 0$, which is just saying: be reasonable. The minimum cost from the end state to the end state is trivially 0, so just use 0.
- We will come back later to the issue of getting a hold of a consistent heuristic, but for now, let's assume we have one and see what we can do with it.

Correctness of A*



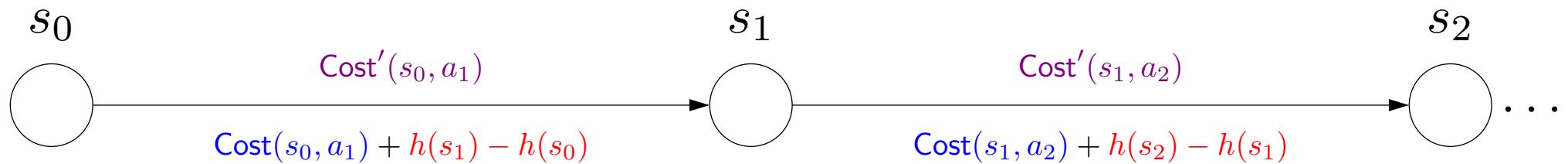
Proposition: correctness

If h is consistent, A* returns the minimum cost path.

- The main theoretical result for A* is that if we use any consistent heuristic, then we will be guaranteed to find the minimum cost path.

Proof of A* correctness

- Consider any path $[s_0, a_1, s_1, \dots, a_L, s_L]$:



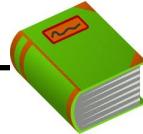
- Key identity:

$$\underbrace{\sum_{i=1}^L \text{Cost}'(s_{i-1}, a_i)}_{\text{modified path cost}} = \underbrace{\sum_{i=1}^L \text{Cost}(s_{i-1}, a_i)}_{\text{original path cost}} + \underbrace{h(s_L) - h(s_0)}_{\text{constant}}$$

- Therefore, A* (finding the minimum cost path using modified costs) solves the original problem (even though edge costs are all different!)

- To show the correctness of A*, let's take any path of length L from $s_0 = s_{\text{start}}$ to $s_L = s_{\text{end}}$. Let us compute the modified path cost by just adding up the modified edge costs. Just to simplify notation, let $c_i = \text{Cost}(s_{i-1}, a_i)$ and $h_i = h(s_i)$. The modified path cost is $(c_1 + h_1 - h_0) + (c_2 + h_2 - h_1) + \cdots + (c_L + h_L - h_{L-1})$. Notice that most of the h_i 's actually cancel out (this is known as **telescoping sums**).
- We end up with $\sum_{i=1}^L c_i$, which is the original path cost plus $h_L - h_0$. First, notice that $h_L = 0$ because s_L is an end state and by the second condition of consistency, $h(s_L) = 0$. Second, h_0 is just a constant (in that it doesn't depend on the path at all), since all paths must start with the start state.
- Therefore, the modified path cost is equal to the original path cost plus a constant. A*, which is running UCS on the modified edge costs, is equivalent to running UCS on the original edge costs, which minimizes the original path cost.
- This is kind of remarkable: all the edge costs are modified in A*, but yet the final path cost is the same (up to a constant)!

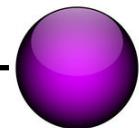
Admissibility



Definition: admissibility

A heuristic $h(s)$ is admissible if
$$h(s) \leq \text{FutureCost}(s)$$

Intuition: admissible heuristics are optimistic



Theorem: consistency implies admissibility

If a heuristic $h(s)$ is **consistent**, then $h(s)$ is **admissible**.

Proof: use induction on $\text{FutureCost}(s)$

- So far, we've just assumed that $\text{FutureCost}(s)$ is the best possible heuristic (ignoring for the moment that it's impractical to compute). Let's actually prove this now.
- To do this, we just have to show that any consistent heuristic $h(s)$ satisfies $h(s) \leq \text{FutureCost}(s)$ (since by the previous theorem, the larger the heuristic, the better). In fact, this property has a special name: we say that $h(s)$ is **admissible**. In other words, an admissible heuristic $h(s)$ **underestimates** the future cost: it is optimistic.
- The proof proceeds by induction on increasing $\text{FutureCost}(s)$. In the base case, we have $0 = h(s_{\text{end}}) \leq \text{FutureCost}(s_{\text{end}}) = 0$ by the second condition of consistency.
- In the inductive case, let s be a state and let a be an optimal action leading to $s' = \text{Succ}(s, a)$ that achieves the minimum cost path to the end state; in other words, $\text{FutureCost}(s) = \text{Cost}(s, a) + \text{FutureCost}(s')$. Since $\text{Cost}(s, a) \geq 0$, we have that $\text{FutureCost}(s') \leq \text{FutureCost}(s)$, so by the inductive hypothesis, $h(s') \leq \text{FutureCost}(s')$. To show the same holds for s , consider: $h(s) \leq \text{Cost}(s, a) + h(s') \leq \text{Cost}(s, a) + \text{FutureCost}(s') = \text{FutureCost}(s)$, where the first inequality follows by consistency of $h(s)$, the second inequality follows by the inductive hypothesis, and the third equality follows because a was chosen to be the optimal action. Therefore, we conclude that $h(s) \leq \text{FutureCost}(s)$.
- Aside: People often talk about admissible heuristics. Using A* with an admissible heuristic is only guaranteed to find the minimum cost path for tree search algorithms, where we don't use an explored list. However, the UCS and A* algorithms we are considering in this class are graph search algorithms, which require consistent heuristics, not just admissible heuristics, to find the minimum cost path. There are some admissible heuristics which are not consistent, but most natural ones are consistent.



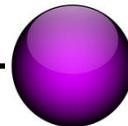
Roadmap

Learning costs

A* search

Relaxation

Efficiency of A*



Theorem: efficiency of A*

A* explores all states s satisfying

$$\text{PastCost}(s) \leq \text{PastCost}(s_{\text{end}}) - h(s)$$

Interpretation: the larger $h(s)$, the better

Proof: A* explores all s such that

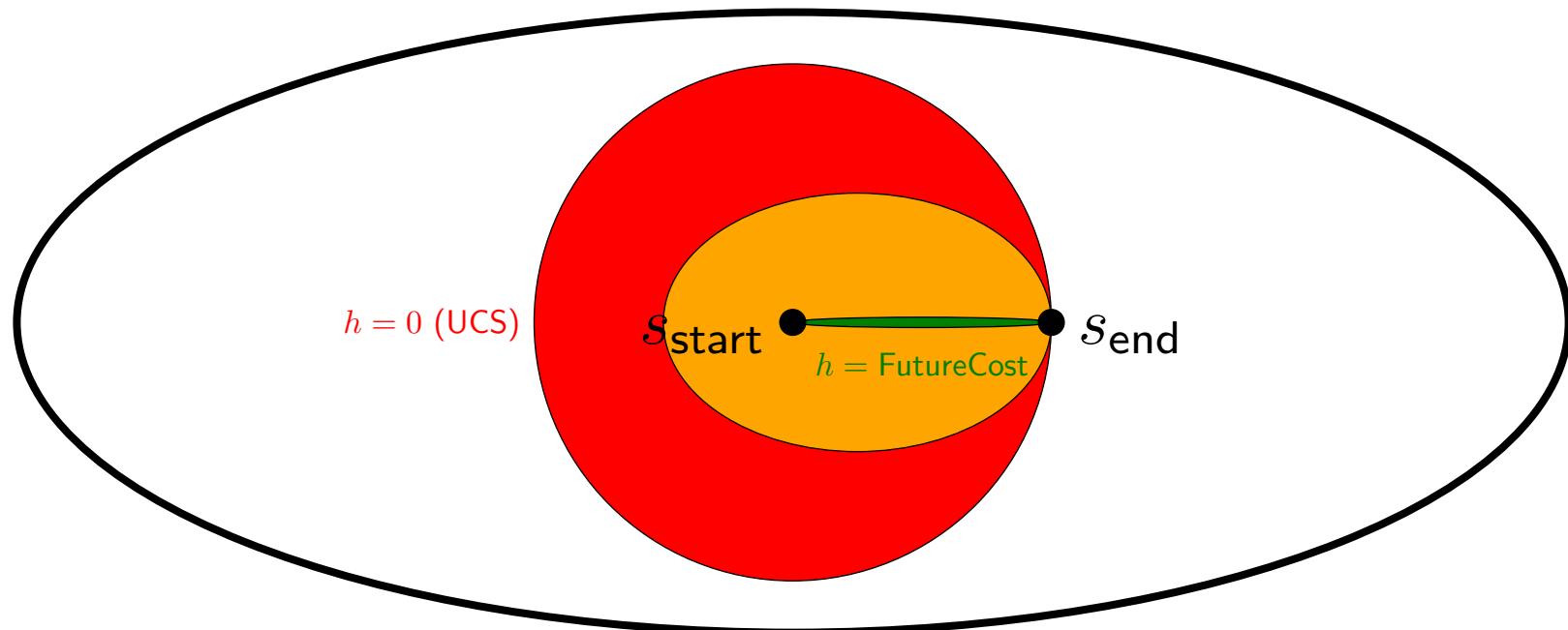
$$\text{PastCost}(s) + h(s)$$

\leq

$$\text{PastCost}(s_{\text{end}})$$

- We've proven that A* is correct (finds the minimum cost path) for any consistent heuristic h . But for A* to be interesting, we need to show that it's more efficient than UCS (on the original edge costs). We will measure speed in terms of the number of states which are explored prior to exploring an end state.
- Our second theorem is about the efficiency of A*: recall that UCS explores states in order of past cost, so that it will explore every state whose past cost is less than the past cost of the end state.
- A* explores all states for which $\text{PastCost}'(s) = \text{PastCost}(s) + h(s) - h(s_{\text{start}})$ is less than $\text{PastCost}'(s_{\text{end}}) = \text{PastCost}(s_{\text{end}}) + h(s_{\text{end}}) - h(s_{\text{start}})$, or equivalently $\text{PastCost}(s) + h(s) \leq \text{PastCost}(s_{\text{end}})$ since $h(s_{\text{end}}) = 0$.
- From here, it's clear that we want $h(s)$ to be as large as possible so we can push as many states over the $\text{PastCost}(s_{\text{end}})$ threshold, so that we don't have to explore them. Of course, we still need h to be consistent to maintain correctness.
- For example, suppose $\text{PastCost}(s_1) = 1$ and $h(s_1) = 1$ and $\text{PastCost}(s_{\text{end}}) = 2$. Then we would have to explore s_1 ($1 + 1 \leq 2$). But if we were able to come up with a better heuristic where $h(s_1) = 2$, then we wouldn't have to explore s_1 ($1 + 2 > 2$).

Amount explored



- If $h(s) = 0$, then A* is same as UCS.
- If $h(s) = \text{FutureCost}(s)$, then A* only explores nodes on a minimum cost path.
- Usually $h(s)$ is somewhere in between.

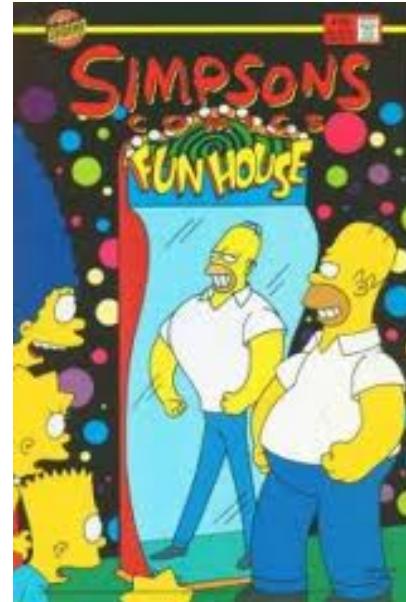
- In this diagram, each ellipse corresponds to the set of states which are explored by A* with various heuristics. In general, any heuristic we come up with will be between the trivial heuristic $h(s) = 0$ which corresponds to UCS and the oracle heuristic $h(s) = \text{FutureCost}(s)$ which is unattainable.

A* search



Key idea: distortion

A* distorts edge costs to favor end states.



- What exactly is A* doing to the edge costs? Intuitively, it's biasing us towards the end state.

How do we get good heuristics? Just relax...



Relaxation

Intuition: ideally, use $h(s) = \text{FutureCost}(s)$, but that's as hard as solving the original problem.



Key idea: relaxation

Constraints make life hard. Get rid of them.

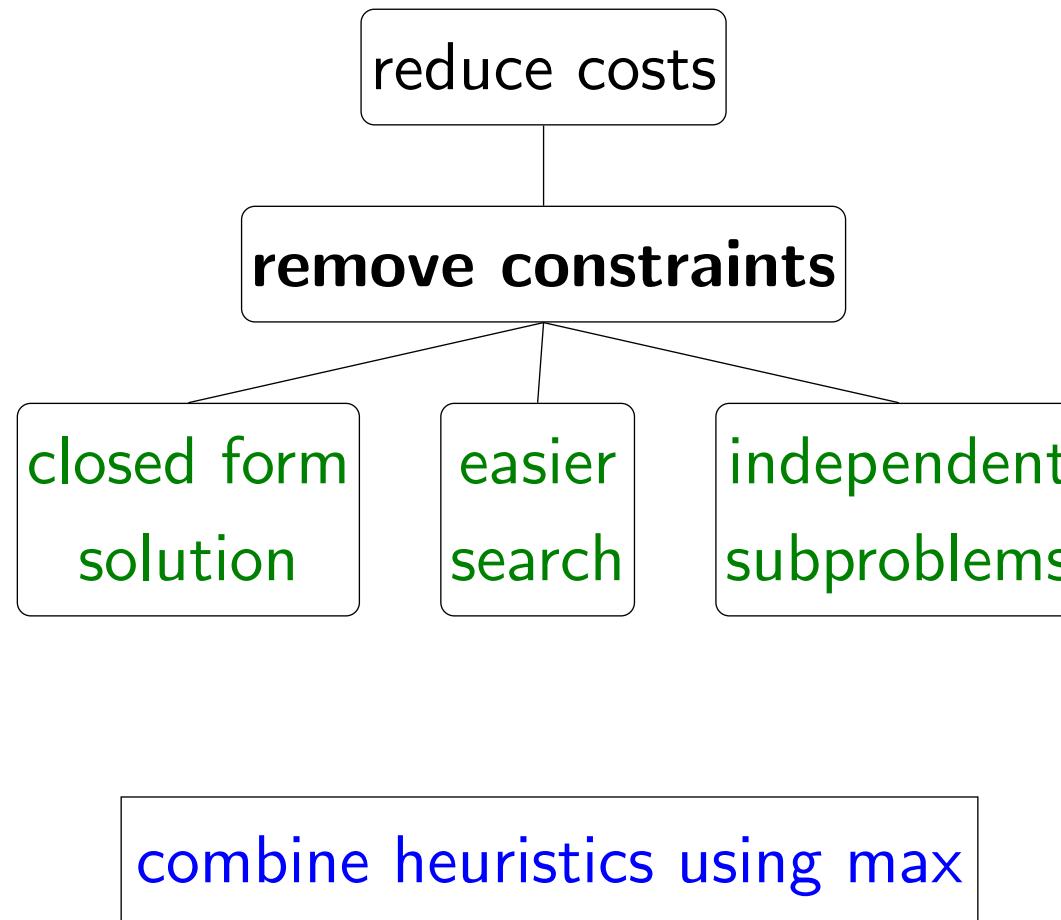
But this is just for the heuristic!



- So far, given a heuristic $h(s)$, we can run A* using it and get a savings which depends on how large $h(s)$ is. However, we've only seen two heuristics: $h(s) = 0$ and $h(s) = \text{FutureCost}(s)$. The first does nothing (gives you back UCS), and the second is hard to compute.
- What we'd like to do is to come up with a general principle for coming up with heuristics. The idea is that of a **relaxation**: instead of computing $\text{FutureCost}(s)$ on the original problem, let us compute $\text{FutureCost}(s)$ on an easier problem, where the notion of easy will be made more formal shortly.
- Note that coming up with good heuristics is about **modeling**, not algorithms. We have to think carefully about our problem domain and see what kind of structure we can exploit in it.



Relaxation overview

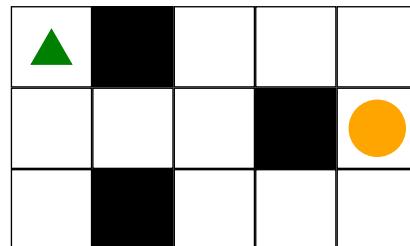


Closed form solution

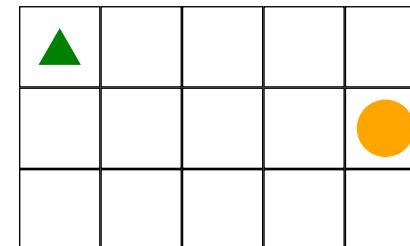


Example: knock down walls

Goal: move from triangle to circle



Hard



Easy

Heuristic:

$$h(s) = \text{ManhattanDistance}(s, (2, 5))$$

$$\text{e.g., } h((1, 1)) = 5$$

- Here's a simple example. Suppose states are positions (r, c) on the grid. Possible actions are moving up, down, left, or right, provided they don't move you into a wall or off the grid; and all edge costs are 1. The start state is at the triangle at $(1, 1)$, and the end state is the circle at position $(2, 5)$.
- With an arbitrary configuration of walls, we can't compute $\text{FutureCost}(s)$ except by doing search. However, if we just **relaxed** the original problem by removing the walls, then we can compute $\text{FutureCost}(s)$ in **closed form**: it's just the Manhattan distance between s and s_{end} . Specifically, $\text{ManhattanDistance}((r_1, c_1), (r_2, c_2)) = |r_1 - r_2| + |c_1 - c_2|$.



Easier search



Example: original problem

Start state: 1

Walk action: from s to $s + 1$ (cost: 1)

Tram action: from s to $2s$ (cost: 2)

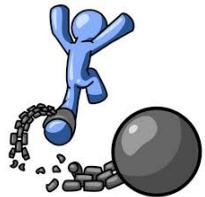
End state: n

Constraint: can't have more tram actions than walk actions.

State: (location, **#walk - #tram**)

Number of states goes from $O(n)$ to $O(n^2)$!

- Let's revisit our magic tram example. Suppose now that a decree comes from above that says you can't have take the tram more times than you walk. This makes our lives considerably worse, since if we wanted to respect this constraint, we have to keep track of additional information (augment the state).
- In particular, we need to keep track of the number of walk actions that we've taken so far minus the number of tram actions we've taken so far, and enforce that this number does not go negative. Now the number of states we have is much larger and thus, search becomes a lot slower.



Easier search



Example: relaxed problem

Start state: 1

Walk action: from s to $s + 1$ (cost: 1)

Tram action: from s to $2s$ (cost: 2)

End state: n

~~Constraint: can't have more tram actions than walk actions.~~

Original state: (location, ~~#walk - #tram~~)

Relaxed state: location

- What if we just ignore that constraint and solve the original problem? That would be much easier/faster.
But how do we construct a consistent heuristic from the solution from the relaxed problem?

Easier search

- Compute relaxed $\text{FutureCost}_{\text{rel}}(\text{location})$ for **each** location $(1, \dots, n)$ using dynamic programming or UCS



Example: reversed relaxed problem

Start state: n

Walk action: from s to $s - 1$ (cost: 1)

Tram action: from s to $s/2$ (cost: 2)

End state: 1

Modify UCS to compute all past costs in reversed relaxed problem
(equivalent to future costs in relaxed problem!)

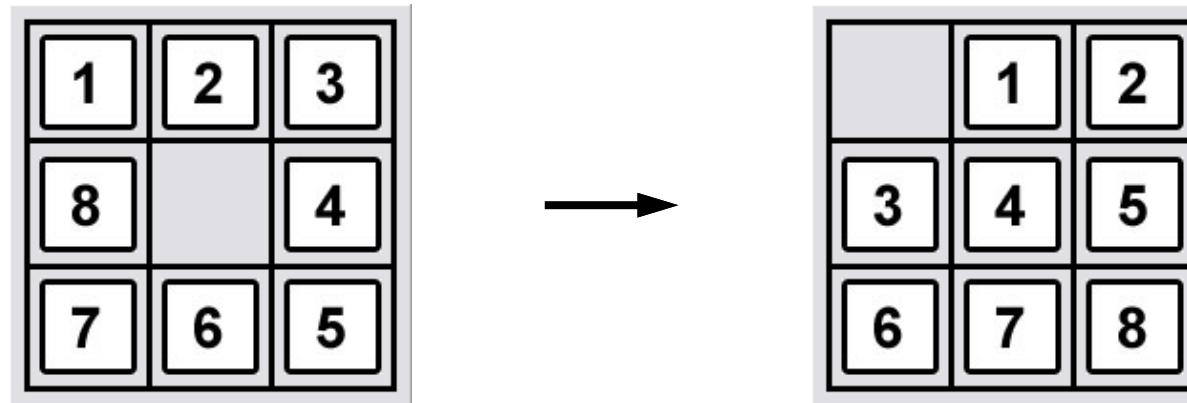
- Define heuristic for original problem:

$$h((\text{location}, \#\text{walk}-\#\text{tram})) = \text{FutureCost}_{\text{rel}}(\text{location})$$

- We want to now construct a heuristic $h(s)$ based on the future costs under the relaxed problem.
- For this, we need the future costs for all the relaxed states. One straightforward way to do this is by using dynamic programming. However, if we have cycles, then we need to use uniform cost search.
- But recall that UCS only computes the past costs of all states up until the end. So we need to make two changes. First, we simply don't stop at the end, but keep on going until we've explored all the states. Second, we define a **reversed relaxed problem** (where all the edges are just reversed), and call UCS on that. UCS will return past costs in the reversed relaxed problem which correspond exactly to future costs in the relaxed problem.
- Finally, we need to construct the actual heuristic. We have to be a bit careful because the state spaces of the relaxed and original problems are different. For this, we set the heuristic $h(s)$ to the future cost of the relaxed version of s .
- Note that the minimum cost returned by A* (UCS on the modified problem) is the true minimum cost minus the value of the heuristic at the start state.

Independent subproblems

[8 puzzle]



Original problem: tiles **cannot** overlap (constraint)

Relaxed problem: tiles **can** overlap (no constraint)

Relaxed solution: 8 indep. problems, each in closed form



Key idea: independence

Relax original problem into independent subproblems.

- So far, we've seen that some heuristics $h(s)$ can be computed in closed form and others can be computed by doing a cheaper search. But there's another way to define heuristics $h(s)$ which are efficient to compute.
- In the 8-puzzle, the goal is to slide the tiles around to produce the desired configuration, but with the constraint that no two tiles can occupy the same position. However, we can throw that constraint out the window to get a relaxed problem. Now, the new problem is really easy, because the tiles can now move **independently**. So we've taken one giant problem and turned it into 8 smaller problems. Each of the smaller problems can now be solved separately (in this case, in closed form, but in other cases, we can also just do search).
- It's worth remembering that all of these relaxed problems are simply used to get the value of the heuristic $h(s)$ to guide the full search. The actual solutions to these relaxed problems are not used.

General framework

Removing constraints

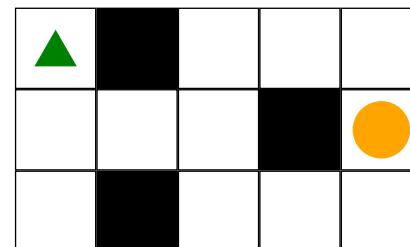
(knock down walls, walk/tram freely, overlap pieces)



Reducing edge costs

(from ∞ to some finite cost)

Example:



Original: $\text{Cost}((1, 1), \text{East}) = \infty$

Relaxed: $\text{Cost}_{\text{rel}}((1, 1), \text{East}) = 1$

- We have seen three instances where removing constraints yields simpler solutions, either via closed form, easier search, or independent subproblems. But we haven't formally proved that the heuristics you get are consistent!
- Now we will analyze all three cases in a unified framework. Removing constraints can be thought of as adding edges (you can go between pairs of states that you weren't able to before). Adding edges is equivalent to reducing the edge cost from infinity to something finite (the resulting edge cost).



MDPs I





Question

How would you get to Mountain View on Friday night in the least amount of time?

bike

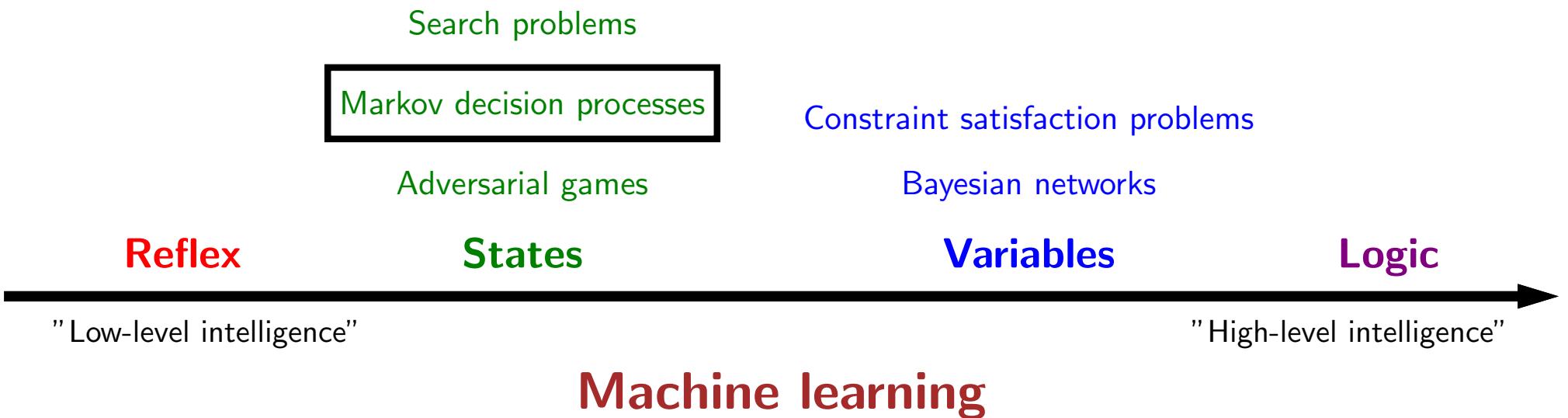
drive

Caltrain

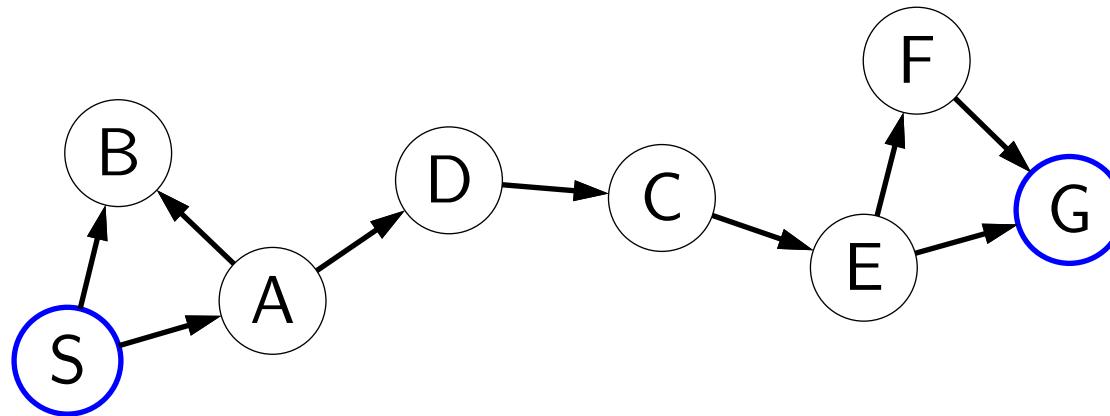
Uber/Lyft

fly

Course plan



So far: search problems

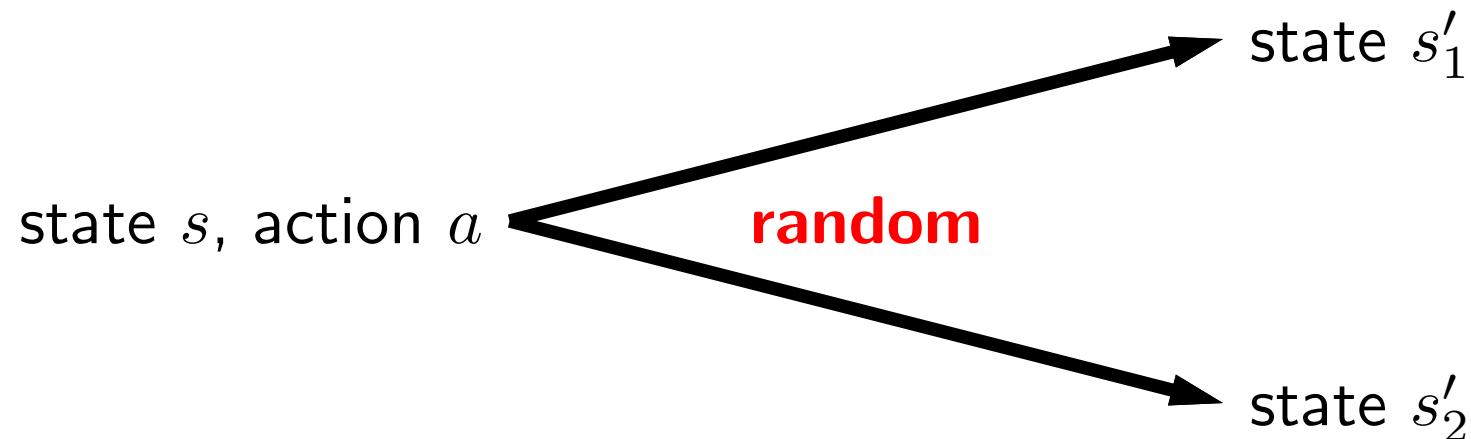


deterministic
state s , action a $\xrightarrow{\hspace{1cm}}$ state $\text{Succ}(s, a)$



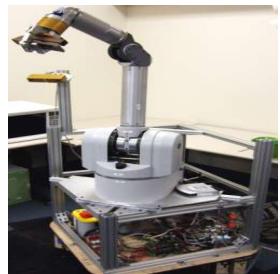
- Last week, we looked at search problems, a powerful paradigm that can be used to solve a diverse range of problems ranging from word segmentation to package delivery to route finding. The key was to cast whatever problem we were interested in solving into the problem of finding the minimum cost path in a graph.
- However, search problems assume that taking an action a from a state s results **deterministically** in a unique successor state $\text{Succ}(s, a)$.

Uncertainty in the real world



- In the real world, the deterministic successor assumption is often unrealistic, for there is **randomness**: taking an action might lead to any one of many possible states.
- One deep question here is how we can even hope to act optimally in the face of randomness? Certainly we can't just have a single deterministic plan, and talking about a minimum cost path doesn't make sense.
- Today, we will develop tools to tackle this more challenging setting. We will fortunately still be able to reuse many of the intuitions about search problems, in particular the notion of a state.

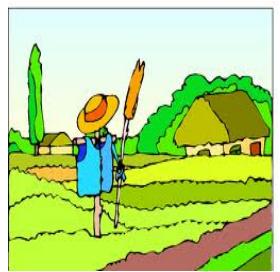
Applications



Robotics: decide where to move, but actuators can fail, hit unseen obstacles, etc.



Resource allocation: decide what to produce, don't know the customer demand for various products



Agriculture: decide what to plant, but don't know weather and thus crop yield

- Randomness shows up in many places. They could be caused by limitations of the sensors and actuators of the robot (which we can control to some extent). Or they could be caused by market forces or nature, which we have no control over.
- We'll see that all of these sources of randomness can be handled in the same mathematical framework.

Volcano crossing



Run (or press ctrl-enter)

		-50	20
		-50	
2			

- Let us consider an example. You are exploring a South Pacific island, which is modeled as a 3x4 grid of states. From each state, you can take one of four actions to move to an adjacent state: north (N), east (E), south (S), or west (W). If you try to move off the grid, you remain in the same state. You start at (2,1). If you end up in either of the green or red squares, your journey ends, either in a lava lake (reward of -50) or in a safe area with either no view (2) or a fabulous view of the island (20). What do you do?
- If we have a deterministic search problem, then the obvious thing will be to go for the fabulous view, which yields a reward of 20. You can set `numIters` to 10 and press Run. Each state is labeled with the maximum expected utility (sum of rewards) one can get from that state (analogue of FutureCost in a search problem). We will define this quantity formally later. For now, look at the arrows, which represent the best action to take from each cell. Note that in some cases, there is a tie for the best, where some of the actions seem to be moving in the wrong direction. This is because there is no penalty for moving around indefinitely. If you change `moveReward` to -0.1, then you'll see the arrows point in the right direction.
- In reality, we are dealing with treacherous terrain, and there is on each action a probability `slipProb` of slipping, which results in moving in a random direction. Try setting `slipProb` to various values. For small values (e.g., 0.1), the optimal action is to still go for the fabulous view. For large values (e.g., 0.3), then it's better to go for the safe and boring 2. Play around with the other reward values to get intuition for the problem.
- Important: note that we are only specifying the dynamics of the world, not directly specifying the best action to take. The best actions are computed automatically from the algorithms we'll see shortly.



Roadmap

Markov decision process

Policy evaluation

Value iteration

Dice game



Example: dice game

For each round $r = 1, 2, \dots$

- You choose **stay** or **quit**.
- If **quit**, you get \$10 and we end the game.
- If **stay**, you get \$4 and then I roll a 6-sided dice.
 - If the dice results in 1 or 2, we end the game.
 - Otherwise, continue to the next round.

Start

Stay

Quit

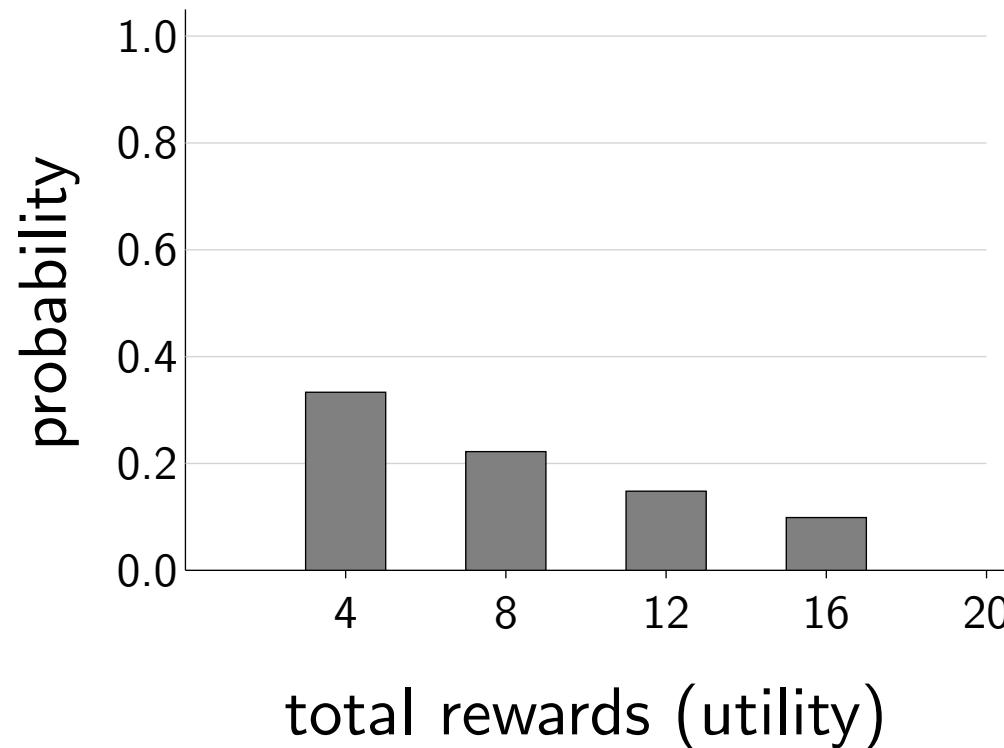
Dice:

Rewards:
 0

- We'll see more volcanoes later, but let's start with a much simpler example: a dice game. What is the best strategy for this game?

Rewards

If follow policy "stay":



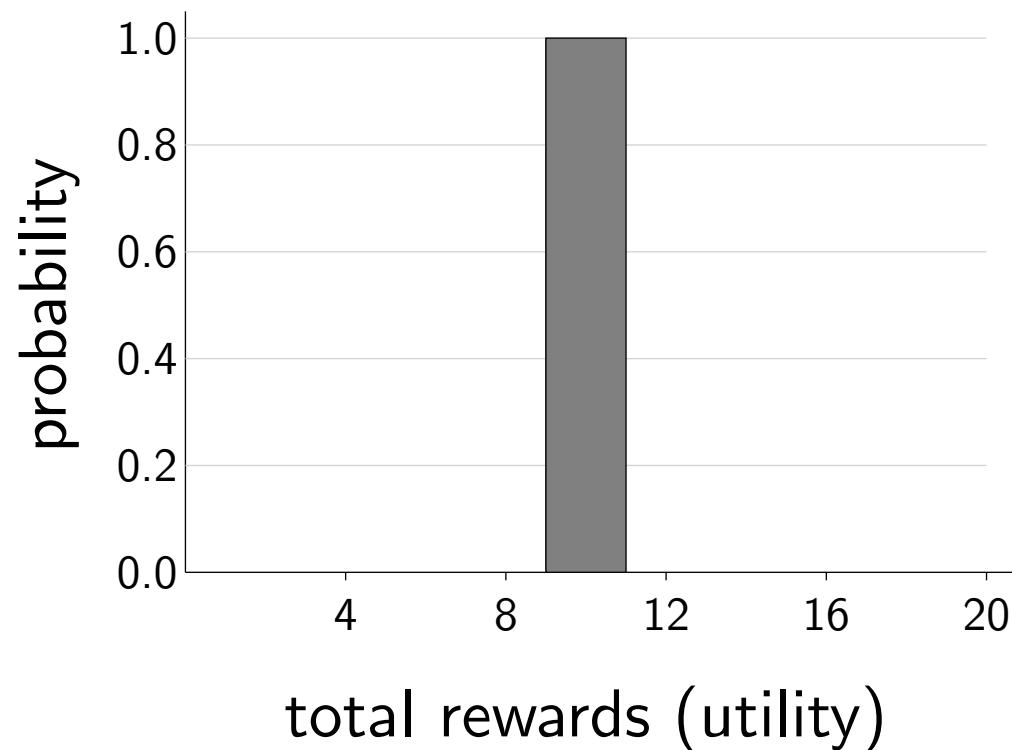
Expected utility:

$$\frac{1}{3}(4) + \frac{2}{3} \cdot \frac{1}{3}(8) + \frac{2}{3} \cdot \frac{2}{3} \cdot \frac{1}{3}(12) + \dots = 12$$

- Let's suppose you always stay. Note that each outcome of the game will result in a different sequence of rewards, resulting in a **utility**, which is in this case just the sum of the rewards.
- We are interested in the **expected** utility, which you can compute to be 12.

Rewards

If follow policy "quit":



Expected utility:

$$1(10) = 10$$

- If you quit, then you'll get a reward of 10 deterministically. Therefore, in expectation, the "stay" strategy is preferred, even though sometimes you'll get less than 10.

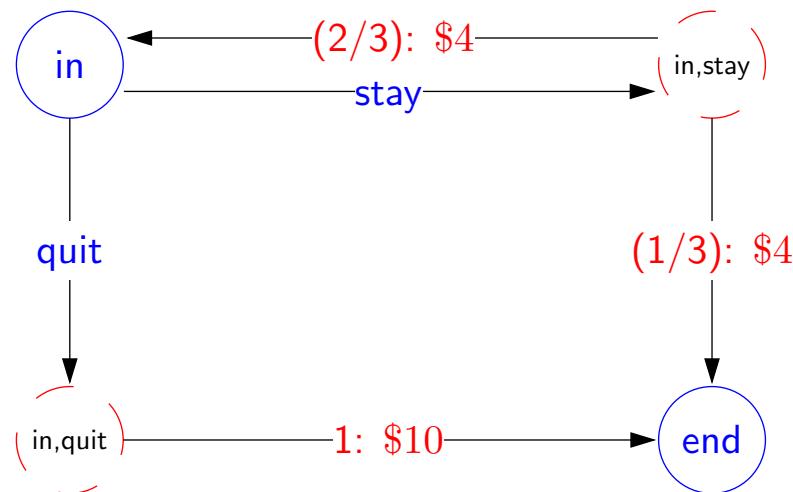
MDP for dice game



Example: dice game

For each round $r = 1, 2, \dots$

- You choose **stay** or **quit**.
- If **quit**, you get \$10 and we end the game.
- If **stay**, you get \$4 and then I roll a 6-sided dice.
 - If the dice results in 1 or 2, we end the game.
 - Otherwise, continue to the next round.



- While we already solved this game directly, we'd like to develop a more general framework for thinking about not just this game, but also other problems such as the volcano crossing example. To that end, let us formalize the dice game as a **Markov decision process** (MDP).
- An MDP can be represented as a graph. The nodes in this graph include both **states** and **chance nodes**. Edges coming out of states are the possible actions from that state, which lead to chance nodes. Edges coming out of a chance nodes are the possible random outcomes of that action, which end up back in states. Our convention is to label these chance-to-state edges with the probability of a particular **transition** and the associated reward for traversing that edge.

Markov decision process



Definition: Markov decision process

States: the set of states

$s_{\text{start}} \in \text{States}$: starting state

$\text{Actions}(s)$: possible actions from state s

$T(s, a, s')$: probability of s' if take action a in state s

$\text{Reward}(s, a, s')$: reward for the transition (s, a, s')

$\text{IsEnd}(s)$: whether at end of game

$0 \leq \gamma \leq 1$: discount factor (default: 1)

- A **Markov decision process** has a set of states States , a starting state s_{start} , and the set of actions $\text{Actions}(s)$ from each state s .
- It also has a **transition distribution** T , which specifies for each state s and action a , a distribution over possible successor states s' . Specifically, we have that $\sum_{s'} T(s, a, s') = 1$ because T is a probability distribution (more on this later).
- Associated with each transition (s, a, s') is a reward, which could be either positive or negative.
- If we arrive in a state s for which $\text{IsEnd}(s)$ is true, then the game is over.
- Finally, the discount factor γ is a quantity which specifies how much we value the future and will be discussed later.

Search problems



Definition: search problem

States: the set of states

$s_{\text{start}} \in \text{States}$: starting state

$\text{Actions}(s)$: possible actions from state s

$\text{Succ}(s, a)$: where we end up if take action a in state s

$\text{Cost}(s, a)$: cost for taking action a in state s

$\text{IsEnd}(s)$: whether at end

- $\text{Succ}(s, a) \Rightarrow T(s, a, s')$
- $\text{Cost}(s, a) \Rightarrow \text{Reward}(s, a, s')$

- MDPs share many similarities with search problems, but there are differences (one main difference and one minor one).
- The main difference is the move from a deterministic successor function $\text{Succ}(s, a)$ to transition probabilities over s' . We can think of the successor function $\text{Succ}(s, a)$ as a special case of transition probabilities:

$$T(s, a, s') = \begin{cases} 1 & \text{if } s' = \text{Succ}(s, a) \\ 0 & \text{otherwise} \end{cases}.$$

- A minor difference is that we've gone from minimizing costs to maximizing rewards. The two are really equivalent: you can negate one to get the other.

Transitions



Definition: transition probabilities

The **transition probabilities** $T(s, a, s')$ specify the probability of ending up in state s' if taken action a in state s .



Example: transition probabilities

s	a	s'	$T(s, a, s')$
in	quit	end	1
in	stay	in	2/3
in	stay	end	1/3

- Just to dwell on the major difference, transition probabilities, a bit more: for each state s and action a , the transition probabilities specifies a distribution over successor states s' .

Probabilities sum to one



Example: transition probabilities

s	a	s'	$T(s, a, s')$
in	quit	end	1
in	stay	in	2/3
in	stay	end	1/3

For each state s and action a :

$$\sum_{s' \in \text{States}} T(s, a, s') = 1$$

Successors: s' such that $T(s, a, s') > 0$

- This means that for each given s and a , if we sum the transition probability $T(s, a, s')$ over all possible successor states s' , we get 1.
- If a transition to a particular s' is not possible, then $T(s, a, s') = 0$. We refer to the s' for which $T(s, a, s') > 0$ as the successors.
- Generally, the number of successors of a given (s, a) is much smaller than the total number of states. For instance, in a search problem, each (s, a) has exactly one successor.



Transportation example



Example: transportation

Street with blocks numbered 1 to n .

Walking from s to $s + 1$ takes 1 minute.

Taking a magic tram from s to $2s$ takes 2 minutes.

How to travel from 1 to n in the least time?

Tram fails with probability 0.5.

[semi-live solution]

- Let us revisit the transportation example. As we all know, magic trams aren't the most reliable forms of transportation, so let us assume that with probability $\frac{1}{2}$, it actually does as advertised, and with probability $\frac{1}{2}$ it just leaves you in the same state.

What is a solution?

Search problem: path (sequence of actions)

MDP:



Definition: policy

A **policy** π is a mapping from each state $s \in \text{States}$ to an action $a \in \text{Actions}(s)$.



Example: volcano crossing

s	$\pi(s)$
(1,1)	S
(2,1)	E
(3,1)	N
...	...

- So we now know what an MDP is. What do we do with one? For search problems, we were trying to find the minimum cost **path**.
- However, fixed paths won't suffice for MDPs, because we don't know which states the random dice rolls are going to take us.
- Therefore, we define a **policy**, which specifies an action for every single state, not just the states along a path. This way, we have all our bases covered, and know what action to take no matter where we are.
- One might wonder if we ever need to take different actions from a given state. The answer is no, since like as in a search problem, the state contains all the information that we need to act optimally for the future. In more formal speak, the transitions and rewards satisfy the **Markov property**. Every time we end up in a state, we are faced with the exact same problem and therefore should take the same optimal action.



Roadmap

Markov decision process

Policy evaluation

Value iteration

Evaluating a policy



Definition: utility

Following a policy yields a **random path**.

The **utility** of a policy is the (discounted) sum of the rewards on the path (this is a random quantity).

Path	Utility
[in; stay, 4, end]	4
[in; stay, 4, in; stay, 4, in; stay, 4, end]	12
[in; stay, 4, in; stay, 4, end]	8
[in; stay, 4, in; stay, 4, in; stay, 4, in; stay, 4, end]	16
...	...



Definition: value (expected utility)

The **value** of a policy is the **expected utility**.

- Now that we've defined an MDP (the input) and a policy (the output), let's turn to defining the evaluation metric for a policy — there are many of them, which one should we choose?
- Recall that we'd like to maximize the total rewards (utility), but this is a random quantity, so we can't quite do that. Instead, we will instead maximize the **expected utility**, which we will refer to as **value** (of a policy).

Evaluating a policy: volcano crossing

Run

(or press ctrl-enter)

a	r	s	
			(2,1)
E	-0.1	(2,2)	
S	-0.1	(3,2)	
E	-0.1	(3,3)	
E	-50.1	(2,3)	

2.4	-0.5	-50	40
3.7 →	5	-50	31
2	12.6 →	16.3 →	26.2

Value: 3.73

Utility: -36.79

- To get an intuitive feel for the relationship between a value and utility, consider the volcano example. If you press Run multiple times, you will get random paths shown on the right leading to different utilities. Note that there is considerable variation in what happens.
- The expectation of this utility is the **value**.
- You can run multiple simulations by increasing numEpisodes. If you set numEpisodes to 1000, then you'll see the average utility converging to the value.

Discounting



Definition: utility

Path: $s_0, a_1 r_1 s_1, a_2 r_2 s_2, \dots$ (action, reward, new state).

The **utility** with discount γ is

$$u_1 = r_1 + \gamma r_2 + \gamma^2 r_3 + \gamma^3 r_4 + \dots$$

Discount $\gamma = 1$ (save for the future):

[stay, stay, stay, stay]: $4 + 4 + 4 + 4 = 16$

Discount $\gamma = 0$ (live in the moment):

[stay, stay, stay, stay]: $4 + 0 \cdot (4 + \dots) = 4$

Discount $\gamma = 0.5$ (balanced life):

[stay, stay, stay, stay]: $4 + \frac{1}{2} \cdot 4 + \frac{1}{4} \cdot 4 + \frac{1}{8} \cdot 4 = 7.5$

- There is an additional aspect to utility: **discounting**, which captures the fact that a reward today might be worth more than the same reward tomorrow. If the discount γ is small, then we favor the present more and downweight future rewards more.
- Note that the discounting parameter is applied exponentially to future rewards, so the distant future is always going to have a fairly small contribution to the utility (unless $\gamma = 1$).
- The terminology, though standard, is slightly confusing: a larger value of the discount parameter γ actually means that the future is discounted less.

Policy evaluation



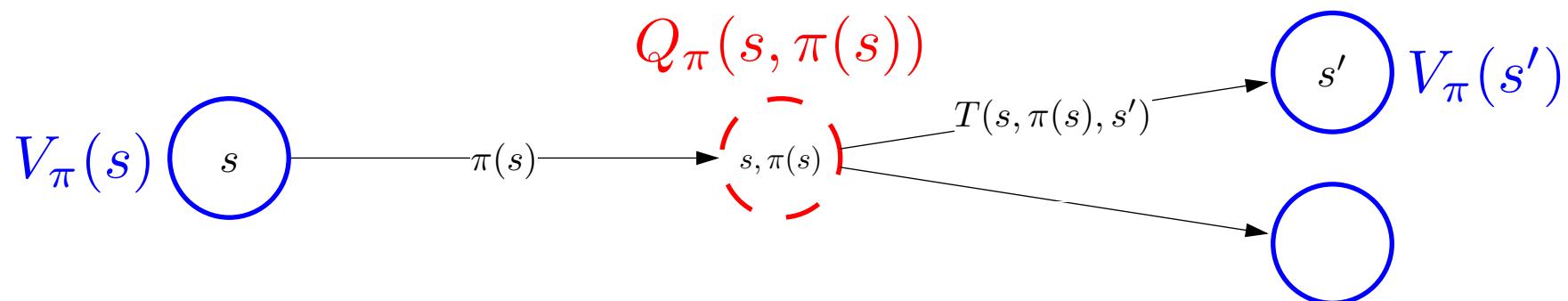
Definition: value of a policy

Let $V_\pi(s)$ be the expected utility received by following policy π from state s .



Definition: Q-value of a policy

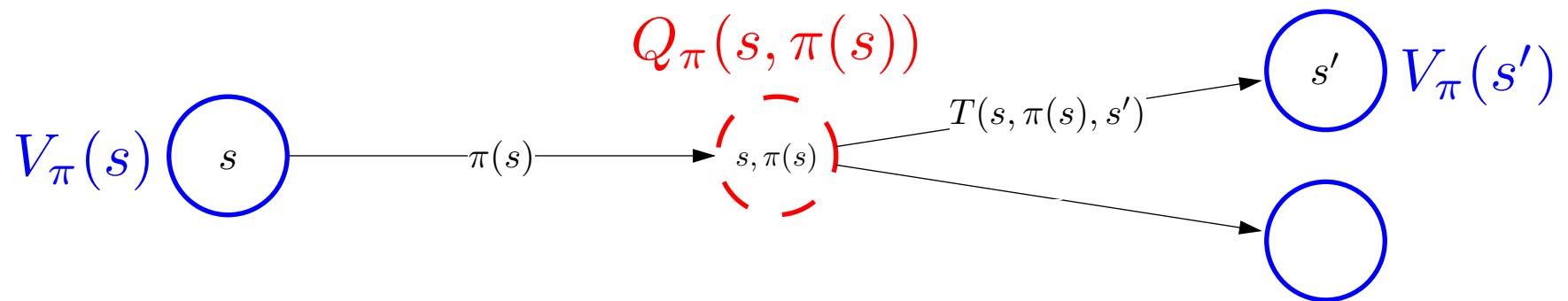
Let $Q_\pi(s, a)$ be the expected utility of taking action a from state s , and then following policy π .



- Associated with any policy π are two important quantities, the value of the policy $V_\pi(s)$ and the Q-value of a policy $Q_\pi(s, a)$.
- In terms of the MDP graph, one can think of the value $V_\pi(s)$ as labeling the state nodes, and the Q-value $Q_\pi(s, a)$ as labeling the chance nodes.
- This label refers to the expected utility if we were to start at that node and continue the dynamics of the game.

Policy evaluation

Plan: define recurrences relating value and Q-value

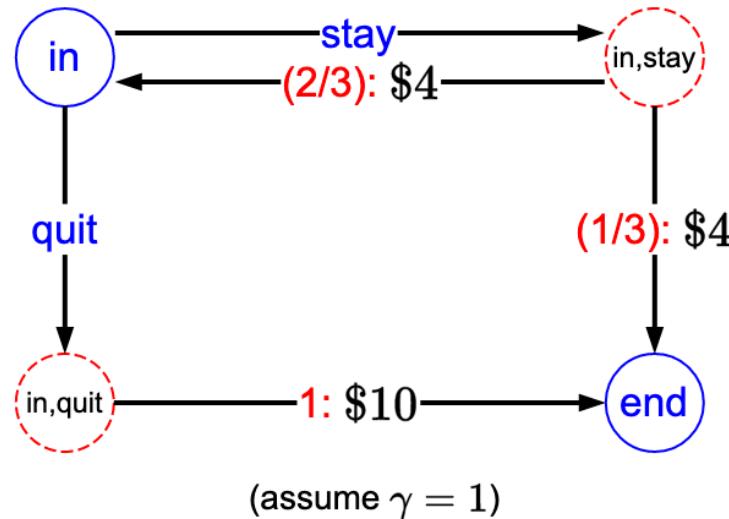


$$V_{\pi}(s) = \begin{cases} 0 & \text{if } \text{IsEnd}(s) \\ Q_{\pi}(s, \pi(s)) & \text{otherwise.} \end{cases}$$

$$Q_{\pi}(s, a) = \sum_{s'} T(s, a, s')[\text{Reward}(s, a, s') + \gamma V_{\pi}(s')]$$

- We will now write down some equations relating value and Q-value. Our eventual goal is to get to an algorithm for computing these values, but as we will see, writing down the relationships gets us most of the way there, just as writing down the recurrence for FutureCost directly lead to a dynamic programming algorithm for acyclic search problems.
- First, we get $V_\pi(s)$, the value of a state s , by just following the action edge specified by the policy and taking the Q-value $Q_\pi(s, \pi(s))$. (There's also a base case where $\text{IsEnd}(s)$.)
- Second, we get $Q_\pi(s, a)$ by considering all possible transitions to successor states s' and taking the expectation over the immediate reward $\text{Reward}(s, a, s')$ plus the discounted future reward $\gamma V_\pi(s')$.
- While we've defined the recurrence for the expected utility directly, we can derive the recurrence by applying the law of total expectation and invoking the Markov property. To do this, we need to set up some random variables: Let s_0 be the initial state, a_1 be the action that we take, r_1 be the reward we obtain, and s_1 be the state we end up in. Also define $u_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots$ to be the utility of following policy π from time step t . Then $V_\pi(s) = \mathbb{E}[u_1 \mid s_0 = s]$, which (assuming s is not an end state) in turn equals $\sum_{s'} \mathbb{P}[s_1 = s' \mid s_0 = s, a_1 = \pi(s)] \mathbb{E}[u_1 \mid s_1 = s', s_0 = s, a_1 = \pi(s)]$. Note that $\mathbb{P}[s_1 = s' \mid s_0 = s, a_1 = \pi(s)] = T(s, \pi(s), s')$. Using the fact that $u_1 = r_1 + \gamma u_2$ and taking expectations, we get that $\mathbb{E}[u \mid s_1 = s', s_0 = s, a_1 = \pi(s)] = \text{Reward}(s, \pi(s), s') + \gamma V_\pi(s')$. The rest follows from algebra.

Dice game



Let π be the "stay" policy: $\pi(\text{in}) = \text{stay}$.

$$V_\pi(\text{end}) = 0$$

$$V_\pi(\text{in}) = \frac{1}{3} (4 + V_\pi(\text{end})) + \frac{2}{3} (4 + V_\pi(\text{in}))$$

In this case, can solve in closed form:

$$V_\pi(\text{in}) = 12$$

- As an example, let's compute the values of the nodes in the dice game for the policy "stay".
- Note that the recurrence involves both $V_\pi(\text{in})$ on the left-hand side and the right-hand side. At least in this simple example, we can solve this recurrence easily to get the value.

Policy evaluation



Key idea: iterative algorithm

Start with arbitrary policy values and repeatedly apply recurrences to converge to true values.



Algorithm: policy evaluation

Initialize $V_{\pi}^{(0)}(s) \leftarrow 0$ for all states s .

For iteration $t = 1, \dots, t_{PE}$:

For each state s :

$$V_{\pi}^{(t)}(s) \leftarrow \underbrace{\sum_{s'} T(s, \pi(s), s')[\text{Reward}(s, \pi(s), s') + \gamma V_{\pi}^{(t-1)}(s')]}_{Q^{(t-1)}(s, \pi(s))}$$

- But for a much larger MDP with 100000 states, how do we efficiently compute the value of a policy?
- One option is the following: observe that the recurrences define a system of linear equations, where the variables are $V_\pi(s)$ for each state s and there is an equation for each state. So we could solve the system of linear equations by computing a matrix inverse. However, inverting a 100000×100000 matrix is expensive in general.
- There is an even simpler approach called **policy evaluation**. We've already seen examples of iterative algorithms in machine learning: the basic idea is to start with something crude, and refine it over time.
- Policy iteration starts with a vector of all zeros for the initial values $V_\pi^{(0)}$. Each iteration, we loop over all the states and apply the two recurrences that we had before. The equations look hairier because of the superscript (t) , which simply denotes the value of at iteration t of the algorithm.

Policy evaluation computation

$$V_{\pi}^{(t)}(s)$$

iteration t

state s	0	-0.1	-0.2	0.7	1.1	1.6	1.9	2.2	2.4	2.6
0	-0.1	1.8	1.8	2.2	2.4	2.7	2.8	3	3.1	
0	4	4	4	4	4	4	4	4	4	4
0	-0.1	1.8	1.8	2.2	2.4	2.7	2.8	3	3.1	
0	-0.1	-0.2	0.7	1.1	1.6	1.9	2.2	2.4	2.6	

- We can visualize the computation of policy evaluation on a grid, where column t denotes all the values $V_{\pi}^{(t)}(s)$ for a given iteration t . The algorithm initializes the first column with 0 and then proceeds to update each subsequent column given the previous column.
- For those who are curious, the diagram shows policy evaluation on an MDP over 5 states where state 3 is a terminal state that delivers a reward of 4, and where there is a single action, MOVE, which transitions to an adjacent state (with wrap-around) with equal probability.

Policy evaluation implementation

How many iterations (t_{PE})? Repeat until values don't change much:

$$\max_{s \in \text{States}} |V_\pi^{(t)}(s) - V_\pi^{(t-1)}(s)| \leq \epsilon$$

Don't store $V_\pi^{(t)}$ for each iteration t , need only last two:

$$V_\pi^{(t)} \text{ and } V_\pi^{(t-1)}$$

- Some implementation notes: a good strategy for determining how many iterations to run policy evaluation is based on how accurate the result is. Rather than set some fixed number of iterations (e.g, 100), we instead set an error tolerance (e.g., $\epsilon = 0.01$), and iterate until the maximum change between values of any state s from one iteration (t) to the previous ($t - 1$) is at most ϵ .
- The second note is that while the algorithm is stated as computing $V_{\pi}^{(t)}$ for each iteration t , we actually only need to keep track of the last two values. This is important for saving memory.

Complexity



Algorithm: policy evaluation

Initialize $V_{\pi}^{(0)}(s) \leftarrow 0$ for all states s .

For iteration $t = 1, \dots, t_{\text{PE}}$:

For each state s :

$$V_{\pi}^{(t)}(s) \leftarrow \underbrace{\sum_{s'} T(s, \pi(s), s')[\text{Reward}(s, \pi(s), s') + \gamma V_{\pi}^{(t-1)}(s')]}_{Q^{(t-1)}(s, \pi(s))}$$

MDP complexity

S states

A actions per state

S' successors (number of s' with $T(s, a, s') > 0$)

Time: $O(t_{\text{PE}} S S')$

- Computing the running time of policy evaluation is straightforward: for each of the t_{PE} iterations, we need to enumerate through each of the S states, and for each one of those, loop over the successors S' . Note that we don't have a dependence on the number of actions A because we have a fixed policy $\pi(s)$ and we only need to look at the action specified by the policy.
- Advanced: Here, we have to iterate t_{PE} time steps to reach a target level of error ϵ . It turns out that t_{PE} doesn't actually have to be very large for very small errors. Specifically, the error decreases exponentially fast as we increase the number of iterations. In other words, to cut the error in half, we only have to run a constant number of more iterations.
- Advanced: For acyclic graphs (for example, the MDP for Blackjack), we just need to do one iteration (not t_{PE}) provided that we process the nodes in reverse topological order of the graph. This is the same setup as we had for dynamic programming in search problems, only the equations are different.

Policy evaluation on dice game

Let π be the "stay" policy: $\pi(\text{in}) = \text{stay}$.

$$V_{\pi}^{(t)}(\text{end}) = 0$$

$$V_{\pi}^{(t)}(\text{in}) = \frac{1}{3}(4 + V_{\pi}^{(t-1)}(\text{end})) + \frac{2}{3}(4 + V_{\pi}^{(t-1)}(\text{in}))$$

s	end	in	$(t = 100 \text{ iterations})$
$V_{\pi}^{(t)}$	0.00	12.00	

Converges to $V_{\pi}(\text{in}) = 12$.

- Let us run policy evaluation on the dice game. The value converges very quickly to the correct answer.



Summary so far

- MDP: graph with states, chance nodes, transition probabilities, rewards
- Policy: mapping from state to action (solution to MDP)
- Value of policy: expected utility over random paths
- Policy evaluation: iterative algorithm to compute value of policy

- Let's summarize: we have defined an MDP, which we should think of a graph where the nodes are states and chance nodes. Because of randomness, solving an MDP means generating policies, not just paths. A policy is evaluated based on its value: the expected utility obtained over random paths. Finally, we saw that policy evaluation provides a simple way to compute the value of a policy.



Roadmap

Markov decision process

Policy evaluation

Value iteration

- If we are given a policy π , we now know how to compute its value $V_\pi(s_{\text{start}})$. So now, we could just enumerate all the policies, compute the value of each one, and take the best policy, but the number of policies is exponential in the number of states (A^S to be exact), so we need something a bit more clever.
- We will now introduce value iteration, which is an algorithm for finding the best policy. While evaluating a given policy and finding the best policy might seem very different, it turns out that value iteration will look a lot like policy evaluation.

Optimal value and policy

Goal: try to get directly at maximum expected utility

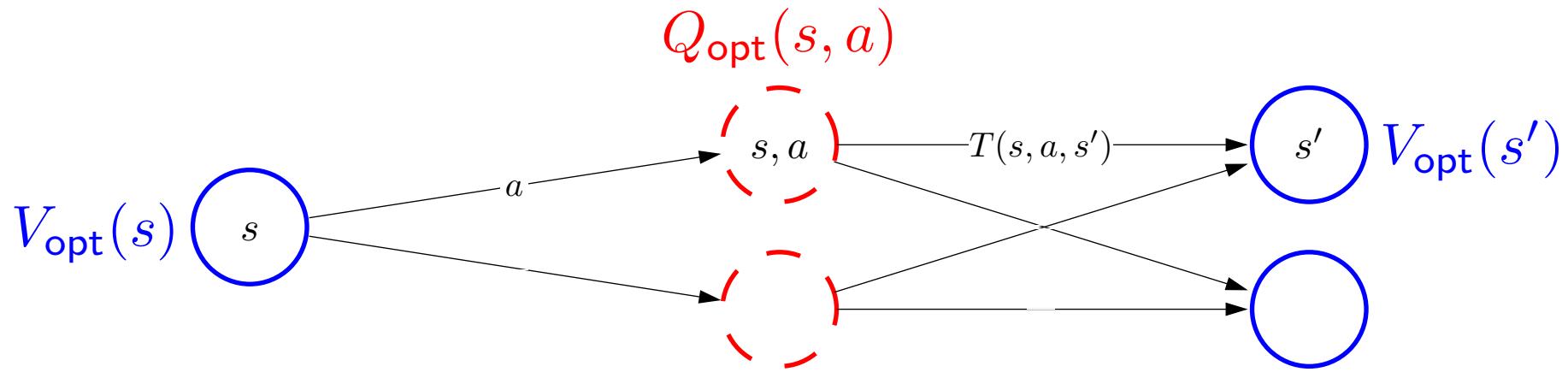


Definition: optimal value

The **optimal value** $V_{\text{opt}}(s)$ is the maximum value attained by any policy.

- We will write down a bunch of recurrences which look exactly like policy evaluation, but instead of having V_π and Q_π with respect to a fixed policy π , we will have V_{opt} and Q_{opt} , which are with respect to the optimal policy.

Optimal values and Q-values



Optimal value if take action a in state s :

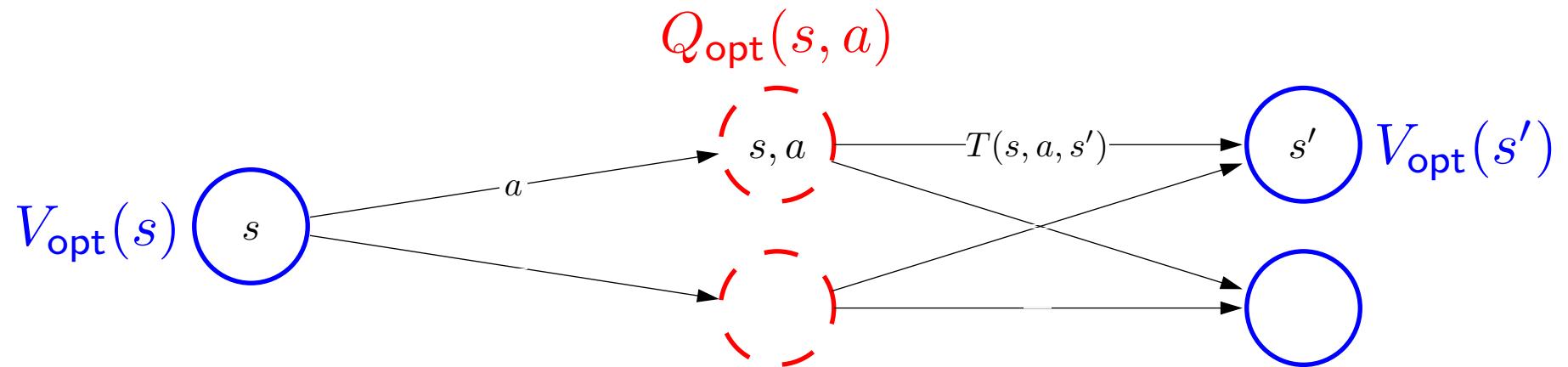
$$Q_{\text{opt}}(s, a) = \sum_{s'} T(s, a, s') [\text{Reward}(s, a, s') + \gamma V_{\text{opt}}(s')].$$

Optimal value from state s :

$$V_{\text{opt}}(s) = \begin{cases} 0 & \text{if } \text{IsEnd}(s) \\ \max_{a \in \text{Actions}(s)} Q_{\text{opt}}(s, a) & \text{otherwise.} \end{cases}$$

- The recurrences for V_{opt} and Q_{opt} are identical to the ones for policy evaluation with one difference: in computing V_{opt} , instead of taking the action from the fixed policy π , we take the best action, the one that results in the largest $Q_{\text{opt}}(s, a)$.

Optimal policies



Given Q_{opt} , read off the optimal policy:

$$\pi_{\text{opt}}(s) = \arg \max_{a \in \text{Actions}(s)} Q_{\text{opt}}(s, a)$$

- So far, we have focused on computing the value of the optimal policy, but what is the actual policy? It turns out that this is pretty easy to compute.
- Suppose you're at a state s . $Q_{\text{opt}}(s, a)$ tells you the value of taking action a from state s . So the optimal action is simply to take the action a with the largest value of $Q_{\text{opt}}(s, a)$.

Value iteration



Algorithm: value iteration [Bellman, 1957]

Initialize $V_{\text{opt}}^{(0)}(s) \leftarrow 0$ for all states s .

For iteration $t = 1, \dots, t_{\text{VI}}$:

For each state s :

$$V_{\text{opt}}^{(t)}(s) \leftarrow \max_{a \in \text{Actions}(s)} \underbrace{\sum_{s'} T(s, a, s') [\text{Reward}(s, a, s') + \gamma V_{\text{opt}}^{(t-1)}(s')]}_{Q_{\text{opt}}^{(t-1)}(s, a)}$$

Time: $O(t_{\text{VI}} S A S')$

[semi-live solution]

- By now, you should be able to go from recurrences to algorithms easily. Following the recipe, we simply iterate some number of iterations, go through each state s and then replace the equality in the recurrence with the assignment operator.
- Value iteration is also guaranteed to converge to the optimal value.
- What about the optimal policy? We get it as a byproduct. The optimal value $V_{\text{opt}}(s)$ is computed by taking a max over actions. If we take the argmax, then we get the optimal policy $\pi_{\text{opt}}(s)$.

Value iteration: dice game

s	end	in
$V_{\text{opt}}^{(t)}$	0.00	12.00 ($t = 100$ iterations)
$\pi_{\text{opt}}(s)$	-	stay

- Let us demonstrate value iteration on the dice game. Initially, the optimal policy is "quit", but as we run value iteration longer, it switches to "stay".

Value iteration: volcano crossing

Run

(or press ctrl-enter)

		-50	20
		-50	
2			

- As another example, consider the volcano crossing. Initially, the optimal policy and value correspond to going to the safe and boring 2. But as you increase numIters, notice how the value of the far away 20 propagates across the grid to the starting point.
- To see this propagation even more clearly, set slipProb to 0.



Roadmap

Learning costs

A* search

Relaxation

How do we get good heuristics? Just relax...



Relaxation

Intuition: ideally, use $h(s) = \text{FutureCost}(s)$, but that's as hard as solving the original problem.



Key idea: relaxation

Constraints make life hard. Get rid of them.

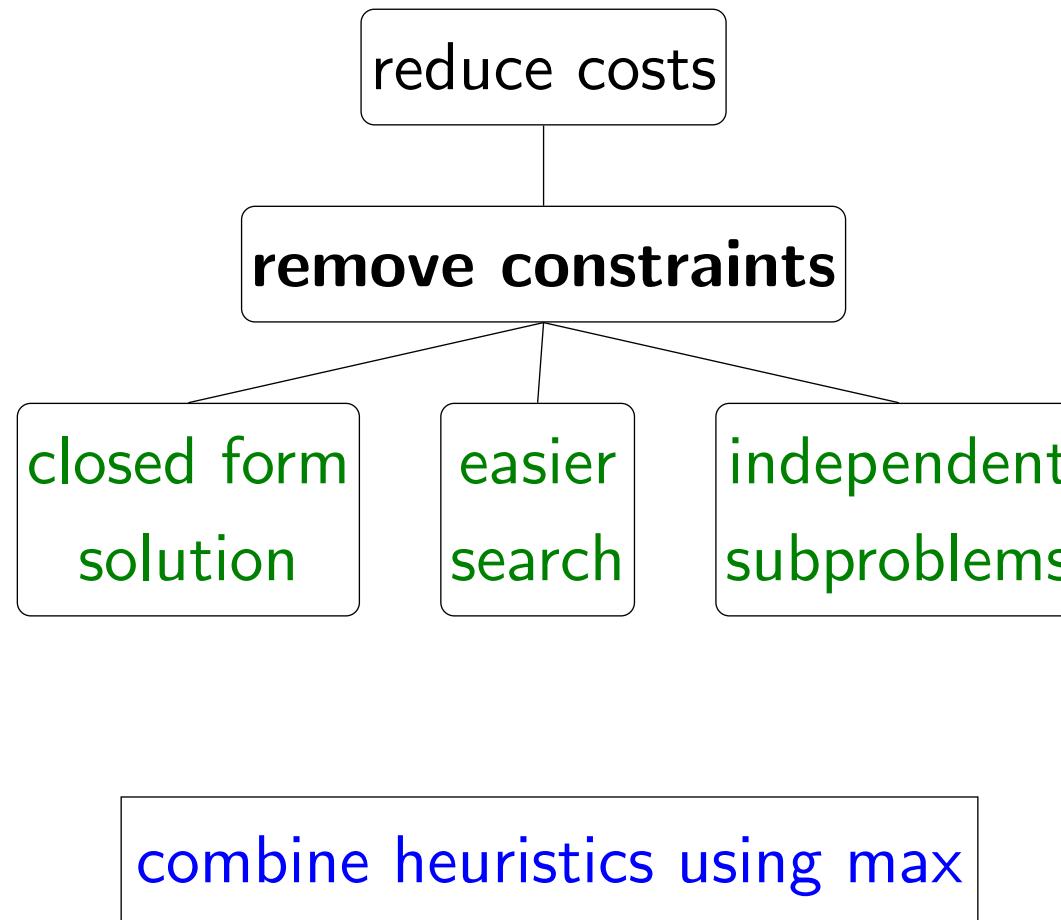
But this is just for the heuristic!



- So far, given a heuristic $h(s)$, we can run A* using it and get a savings which depends on how large $h(s)$ is. However, we've only seen two heuristics: $h(s) = 0$ and $h(s) = \text{FutureCost}(s)$. The first does nothing (gives you back UCS), and the second is hard to compute.
- What we'd like to do is to come up with a general principle for coming up with heuristics. The idea is that of a **relaxation**: instead of computing $\text{FutureCost}(s)$ on the original problem, let us compute $\text{FutureCost}(s)$ on an easier problem, where the notion of easy will be made more formal shortly.
- Note that coming up with good heuristics is about **modeling**, not algorithms. We have to think carefully about our problem domain and see what kind of structure we can exploit in it.



Relaxation overview

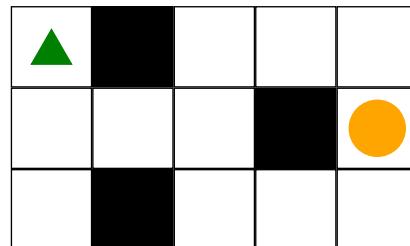


Closed form solution

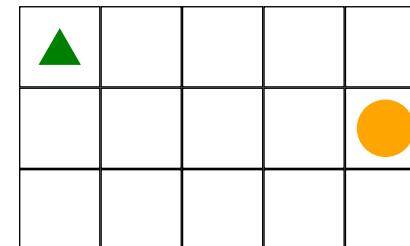


Example: knock down walls

Goal: move from triangle to circle



Hard



Easy

Heuristic:

$$h(s) = \text{ManhattanDistance}(s, (2, 5))$$

$$\text{e.g., } h((1, 1)) = 5$$

- Here's a simple example. Suppose states are positions (r, c) on the grid. Possible actions are moving up, down, left, or right, provided they don't move you into a wall or off the grid; and all edge costs are 1. The start state is at the triangle at $(1, 1)$, and the end state is the circle at position $(2, 5)$.
- With an arbitrary configuration of walls, we can't compute $\text{FutureCost}(s)$ except by doing search. However, if we just **relaxed** the original problem by removing the walls, then we can compute $\text{FutureCost}(s)$ in **closed form**: it's just the Manhattan distance between s and s_{end} . Specifically, $\text{ManhattanDistance}((r_1, c_1), (r_2, c_2)) = |r_1 - r_2| + |c_1 - c_2|$.



Easier search



Example: original problem

Start state: 1

Walk action: from s to $s + 1$ (cost: 1)

Tram action: from s to $2s$ (cost: 2)

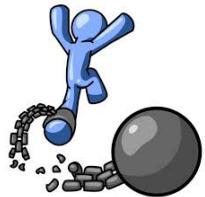
End state: n

Constraint: can't have more tram actions than walk actions.

State: (location, **#walk - #tram**)

Number of states goes from $O(n)$ to $O(n^2)$!

- Let's revisit our magic tram example. Suppose now that a decree comes from above that says you can't have take the tram more times than you walk. This makes our lives considerably worse, since if we wanted to respect this constraint, we have to keep track of additional information (augment the state).
- In particular, we need to keep track of the number of walk actions that we've taken so far minus the number of tram actions we've taken so far, and enforce that this number does not go negative. Now the number of states we have is much larger and thus, search becomes a lot slower.



Easier search



Example: relaxed problem

Start state: 1

Walk action: from s to $s + 1$ (cost: 1)

Tram action: from s to $2s$ (cost: 2)

End state: n

~~Constraint: can't have more tram actions than walk actions.~~

Original state: (location, ~~#walk - #tram~~)

Relaxed state: location

- What if we just ignore that constraint and solve the original problem? That would be much easier/faster.
But how do we construct a consistent heuristic from the solution from the relaxed problem?

Easier search

- Compute relaxed $\text{FutureCost}_{\text{rel}}(\text{location})$ for **each** location $(1, \dots, n)$ using dynamic programming or UCS



Example: reversed relaxed problem

Start state: n

Walk action: from s to $s - 1$ (cost: 1)

Tram action: from s to $s/2$ (cost: 2)

End state: 1

Modify UCS to compute all past costs in reversed relaxed problem
(equivalent to future costs in relaxed problem!)

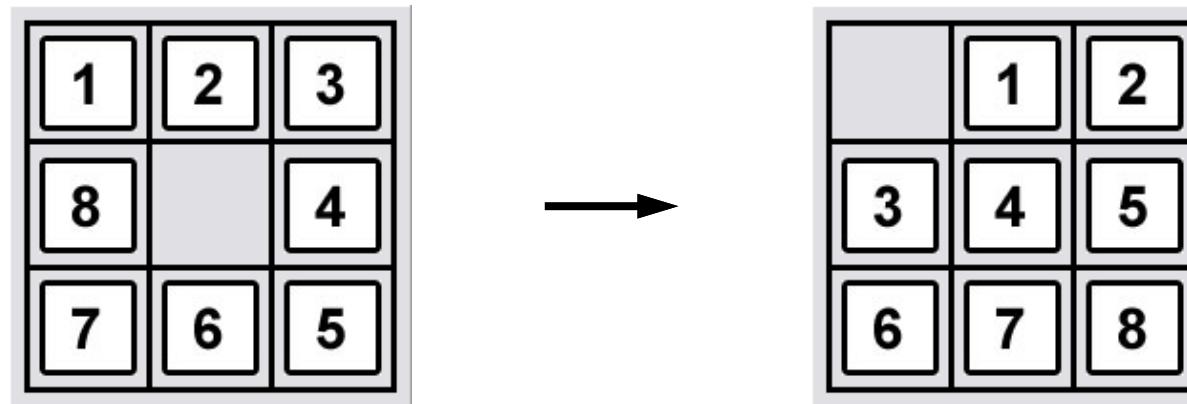
- Define heuristic for original problem:

$$h((\text{location}, \#\text{walk}-\#\text{tram})) = \text{FutureCost}_{\text{rel}}(\text{location})$$

- We want to now construct a heuristic $h(s)$ based on the future costs under the relaxed problem.
- For this, we need the future costs for all the relaxed states. One straightforward way to do this is by using dynamic programming. However, if we have cycles, then we need to use uniform cost search.
- But recall that UCS only computes the past costs of all states up until the end. So we need to make two changes. First, we simply don't stop at the end, but keep on going until we've explored all the states. Second, we define a **reversed relaxed problem** (where all the edges are just reversed), and call UCS on that. UCS will return past costs in the reversed relaxed problem which correspond exactly to future costs in the relaxed problem.
- Finally, we need to construct the actual heuristic. We have to be a bit careful because the state spaces of the relaxed and original problems are different. For this, we set the heuristic $h(s)$ to the future cost of the relaxed version of s .
- Note that the minimum cost returned by A* (UCS on the modified problem) is the true minimum cost minus the value of the heuristic at the start state.

Independent subproblems

[8 puzzle]



Original problem: tiles **cannot** overlap (constraint)

Relaxed problem: tiles **can** overlap (no constraint)

Relaxed solution: 8 indep. problems, each in closed form



Key idea: independence

Relax original problem into independent subproblems.

- So far, we've seen that some heuristics $h(s)$ can be computed in closed form and others can be computed by doing a cheaper search. But there's another way to define heuristics $h(s)$ which are efficient to compute.
- In the 8-puzzle, the goal is to slide the tiles around to produce the desired configuration, but with the constraint that no two tiles can occupy the same position. However, we can throw that constraint out the window to get a relaxed problem. Now, the new problem is really easy, because the tiles can now move **independently**. So we've taken one giant problem and turned it into 8 smaller problems. Each of the smaller problems can now be solved separately (in this case, in closed form, but in other cases, we can also just do search).
- It's worth remembering that all of these relaxed problems are simply used to get the value of the heuristic $h(s)$ to guide the full search. The actual solutions to these relaxed problems are not used.

General framework

Removing constraints

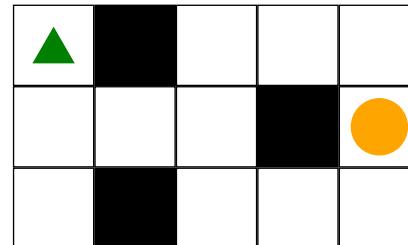
(knock down walls, walk/tram freely, overlap pieces)



Reducing edge costs

(from ∞ to some finite cost)

Example:



Original: $\text{Cost}((1, 1), \text{East}) = \infty$

Relaxed: $\text{Cost}_{\text{rel}}((1, 1), \text{East}) = 1$

General framework



Definition: relaxed search problem

A **relaxation** P_{rel} of a search problem P has costs that satisfy:

$$\text{Cost}_{\text{rel}}(s, a) \leq \text{Cost}(s, a).$$

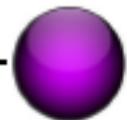


Definition: relaxed heuristic

Given a relaxed search problem P_{rel} , define the **relaxed heuristic** $h(s) = \text{FutureCost}_{\text{rel}}(s)$, the minimum cost from s to an end state using $\text{Cost}_{\text{rel}}(s, a)$.

- More formally, we define a relaxed search problem as one where the relaxed edge costs are no larger than the original edge costs.
- The relaxed heuristic is simply the future cost of the relaxed search problem, which by design should be efficiently computable.

General framework



Theorem: consistency of relaxed heuristics

Suppose $h(s) = \text{FutureCost}_{\text{rel}}(s)$ for some relaxed problem P_{rel} .

Then $h(s)$ is a consistent heuristic.

Proof:

$$h(s) \leq \text{Cost}_{\text{rel}}(s, a) + h(\text{Succ}(s, a)) \quad [\text{triangle inequality}]$$

$$\leq \text{Cost}(s, a) + h(\text{Succ}(s, a)) \quad [\text{relaxation}]$$

- We now need to check that our defined heuristic $h(s)$ is actually consistent, so that using it actually will yield the minimum cost path of our original problem (not the relaxed problem, which is just a means towards an end).
- Checking consistency is actually quite easy. The first inequality follows because $h(s) = \text{FutureCost}_{\text{rel}}(s)$, and all future costs correspond to the minimum cost paths. So taking action a from state s better be no better than taking the best action from state s (this is all in the search problem P_{rel}).
- The second inequality follows just by the definition of a relaxed search problem.
- The significance of this theorem is that we only need to think about coming up with relaxations rather than worrying directly about checking consistency.

Tradeoff

Efficiency:

$h(s) = \text{FutureCost}_{\text{rel}}(s)$ must be easy to compute

Closed form, easier search, independent subproblems

Tightness:

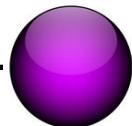
heuristic $h(s)$ should be close to $\text{FutureCost}(s)$

Don't remove too many constraints

- How should one go about designing a heuristic?
- First, the heuristic should be easy to compute. As the main point of A* is to make things more efficient, if the heuristic is as hard as to compute as the original search problem, we've gained nothing (an extreme case is no relaxation at all, in which case $h(s) = \text{FutureCost}(s)$).
- Second, the heuristic should tell us some information about where the goal is. In the extreme case, we relax all the way and we have $h(s) = 0$, which corresponds to running UCS. (Perhaps it is reassuring that we never perform worse than UCS.)
- So the art of designing heuristics is to balance informativeness with computational efficiency.

Max of two heuristics

How do we combine two heuristics?



Proposition: max heuristic

Suppose $h_1(s)$ and $h_2(s)$ are consistent.

Then $h(s) = \max\{h_1(s), h_2(s)\}$ is consistent.

Proof: exercise

- In many situations, you'll be able to come up with two heuristics which are both reasonable, but no one dominates the other. Which one should you choose? Fortunately, you don't have to choose because you can use both of them!
- The key point is the max of two consistent heuristics is consistent. Why max? Remember that we want heuristic values to be larger. And furthermore, we can prove that taking the max leads to a consistent heuristic.
- Stepping back a bit, there is a deeper takeaway with A* and relaxation here. The idea is that when you are confronted with a difficult problem, it is wise to start by solving easier versions of the problem (being an optimist). The result of solving these easier problems can then be a useful guide in solving the original problem.
- For example, if the relaxed problem turns out to have no solution, then you don't even have to bother solving the original problem, because a solution can't possibly exist (you've been optimistic by using the relaxation).



Summary

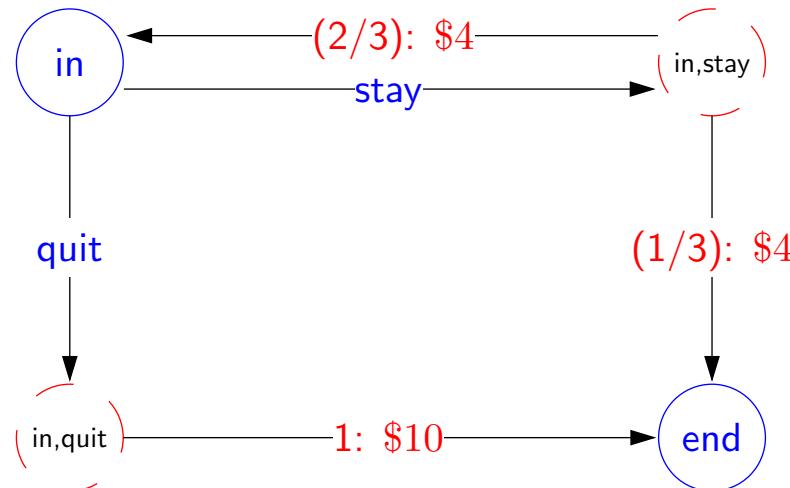
- Structured Perceptron (reverse engineering): learn cost functions (search + learning)
- A*: add in heuristic estimate of future costs
- Relaxation (breaking the rules): framework for producing consistent heuristics
- Next time: when actions have unknown consequences...



MDPs II



Review: MDPs



Definition: Markov decision process

States: the set of states

$s_{\text{start}} \in \text{States}$: starting state

$\text{Actions}(s)$: possible actions from state s

$T(s, a, s')$: probability of s' if take action a in state s

$\text{Reward}(s, a, s')$: reward for the transition (s, a, s')

$\text{IsEnd}(s)$: whether at end of game

$0 \leq \gamma \leq 1$: discount factor (default: 1)

- Last time, we talked about MDPs, which we can think of as graphs, where each node is either a state s or a chance node (s, a) . Actions take us from states to chance nodes. This movement is something we can control. Transitions take us from chance nodes to states. This movement is random, and the various likelihoods are governed by transition probabilities.

Review: MDPs

- Following a **policy** π produces a path (**episode**)

$s_0; a_1, r_1, s_1; a_2, r_2, s_2; a_3, r_3, s_3; \dots; a_n, r_n, s_n$

- **Value** function $V_\pi(s)$: expected utility if follow π from state s

$$V_\pi(s) = \begin{cases} 0 & \text{if } \text{IsEnd}(s) \\ Q_\pi(s, \pi(s)) & \text{otherwise.} \end{cases}$$

- **Q-value** function $Q_\pi(s, a)$: expected utility if first take action a from state s and then follow π

$$Q_\pi(s, a) = \sum_{s'} T(s, a, s')[\text{Reward}(s, a, s') + \gamma V_\pi(s')]$$

- Given a policy π and an MDP, we can run the policy on the MDP yielding a sequence of states, action, rewards $s_0; a_1, r_1, s_1; a_2, r_2, s_2; \dots$. Formally, for each time step t , $a_t = \pi(s_{t-1})$, and s_t is sampled with probability $T(s_{t-1}, a_t, s_t)$. We call such a sequence an **episode** (a path in the MDP graph). This will be a central notion in this lecture.
- Each episode (path) is associated with a **utility**, which is the discounted sum of rewards: $u_1 = r_1 + \gamma r_2 + \gamma^2 r_3 + \dots$. It's important to remember that the utility u_1 is a **random variable** which depends on how the transitions were sampled.
- The value of the policy (from state s_0) is $V_\pi(s_0) = \mathbb{E}[u_1]$, the expected utility. In the last lecture, we worked with the values directly without worrying about the underlying random variables (but that will soon no longer be the case). In particular, we defined recurrences relating the value $V_\pi(s)$ and Q-value $Q_\pi(s, a)$, which represents the expected utility from starting at the corresponding nodes in the MDP graph.
- Given these mathematical recurrences, we produced algorithms: policy evaluation computes the value of a policy, and value iteration computes the optimal policy.

Unknown transitions and rewards



Definition: Markov decision process

States: the set of states

$s_{\text{start}} \in \text{States}$: starting state

$\text{Actions}(s)$: possible actions from state s

$\text{IsEnd}(s)$: whether at end of game

$0 \leq \gamma \leq 1$: discount factor (default: 1)

reinforcement learning!

- In this lecture, we assume that we have an MDP where we neither know the transitions nor the reward functions. We are still trying to maximize expected utility, but we are in a much more difficult setting called **reinforcement learning**.

Mystery game



Example: mystery buttons

For each round $r = 1, 2, \dots$

- You choose A or B.
- You move to a new state and get some rewards.

Start

A

B

State: 5,0

Rewards: 0

- To put yourselves in the shoes of a reinforcement learner, try playing the game. You can either push the A button or the B button. Each of the two actions will take you to a new state and give you some reward.
- This simple game illustrates some of the challenges of reinforcement learning: we should take good actions to get rewards, but in order to know which actions are good, we need to explore and try different actions.

From MDPs to reinforcement learning



visit wpclipart for original image
www.wpclipart.com/people/thinkers/thinker.html

Markov decision process (offline)

- Have mental model of how the world works.
- Find policy to collect maximum rewards.

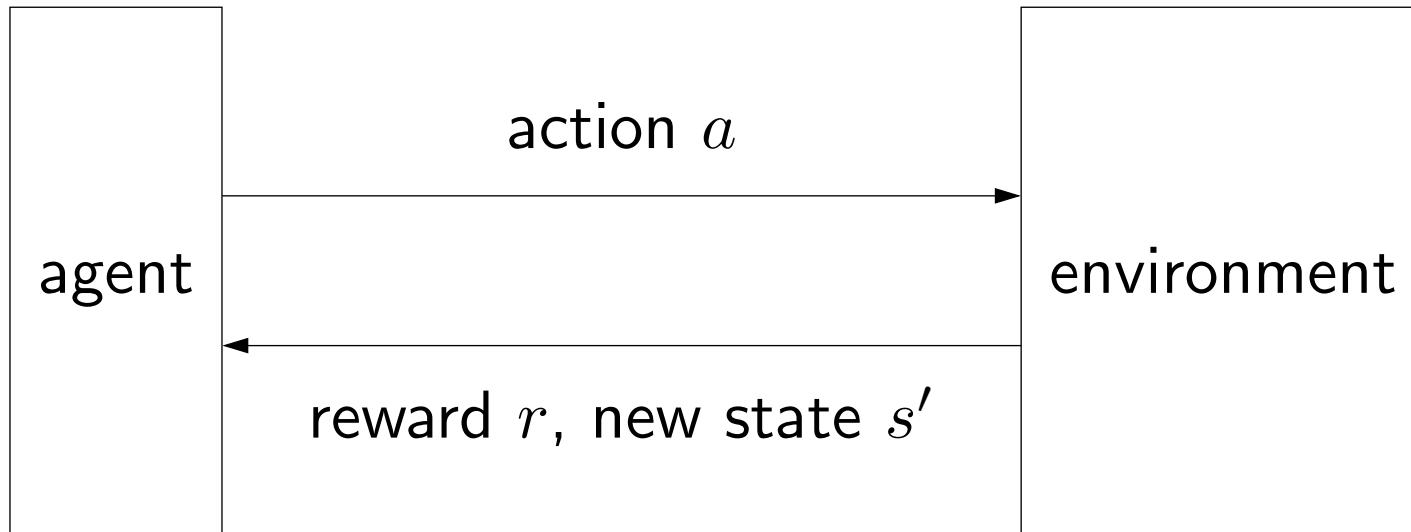


Reinforcement learning (online)

- Don't know how the world works.
- Perform actions in the world to find out and collect rewards.

- An important distinction between solving MDPs (what we did before) and reinforcement learning (what we will do now) is that the former is **offline** and the latter is **online**.
- In the former case, you have a mental model of how the world works. You go lock yourself in a room, think really hard, come up with a policy. Then you come out and use it to act in the real world.
- In the latter case, you don't know how the world works, but you only have one life, so you just have to go out into the real world and learn how it works from experiencing it and trying to take actions that yield high rewards.
- At some level, reinforcement learning is really the way humans work: we go through life, taking various actions, getting feedback. We get rewarded for doing well and learn along the way.

Reinforcement learning framework



Algorithm: reinforcement learning template

For $t = 1, 2, 3, \dots$

Choose action $a_t = \pi_{\text{act}}(s_{t-1})$ (**how?**)

Receive reward r_t and observe new state s_t

Update parameters (**how?**)

- To make the framework clearer, we can think of an **agent** (the reinforcement learning algorithm) that repeatedly chooses an action a_t to perform in the environment, and receives some reward r_t , and information about the new state s_t .
- There are two questions here: how to choose actions (what is π_{act}) and how to update the parameters. We will first talk about updating parameters (the learning part), and then come back to action selection later.

Volcano crossing



Run (or press ctrl-enter)

		-50	20
		-50	
2			

Utility: 2

a r s
(2,1)
W 0 (2,1)
W 0 (2,1)
N 0 (1,1)
W 0 (1,1)
N 0 (1,1)
E 0 (1,2)
S 0 (2,2)
W 0 (2,1)
N 0 (2,2)
N 0 (3,2)
S 0 (3,2)
W 2 (3,1)

- Recall the volcano crossing example from the previous lecture. Each square is a state. From each state, you can take one of four actions to move to an adjacent state: north (N), east (E), south (S), or west (W). If you try to move off the grid, you remain in the same state. The starting state is (2,1), and the end states are the four marked with red or green rewards. Transitions from (s, a) lead where you expect with probability `1-slipProb` and to a random adjacent square with probability `slipProb`.
- If we solve the MDP using value iteration (by setting `numIters` to 10), we will find the best policy (which is to head for the 20). Of course, we can't solve the MDP if we don't know the transitions or rewards.
- If you set `numIters` to zero, we start off with a random policy. Try pressing the Run button to generate fresh episodes. How can we learn from this data and improve our policy?



Roadmap

Reinforcement learning

Monte Carlo methods

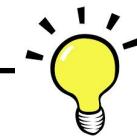
Bootstrapping methods

Covering the unknown

Summary

Model-based Monte Carlo

Data: $s_0; a_1, r_1, s_1; a_2, r_2, s_2; a_3, r_3, s_3; \dots; a_n, r_n, s_n$



Key idea: model-based learning

Estimate the MDP: $T(s, a, s')$ and $\text{Reward}(s, a, s')$

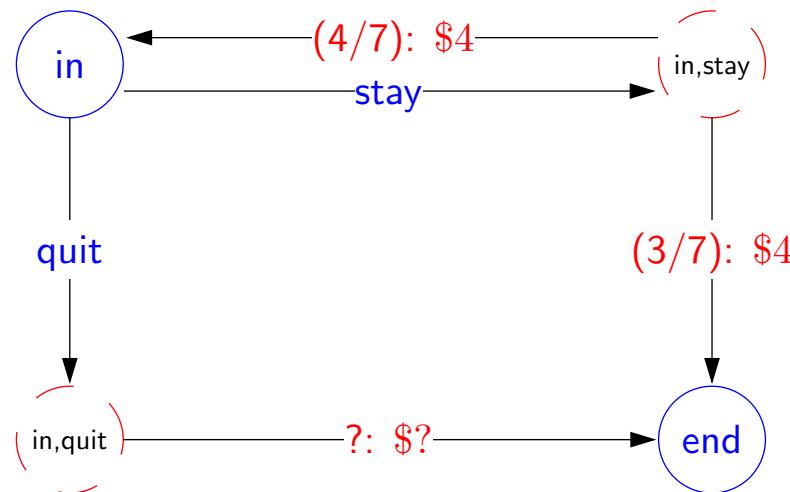
Transitions:

$$\hat{T}(s, a, s') = \frac{\# \text{ times } (s, a, s') \text{ occurs}}{\# \text{ times } (s, a) \text{ occurs}}$$

Rewards:

$$\widehat{\text{Reward}}(s, a, s') = r \text{ in } (s, a, r, s')$$

Model-based Monte Carlo



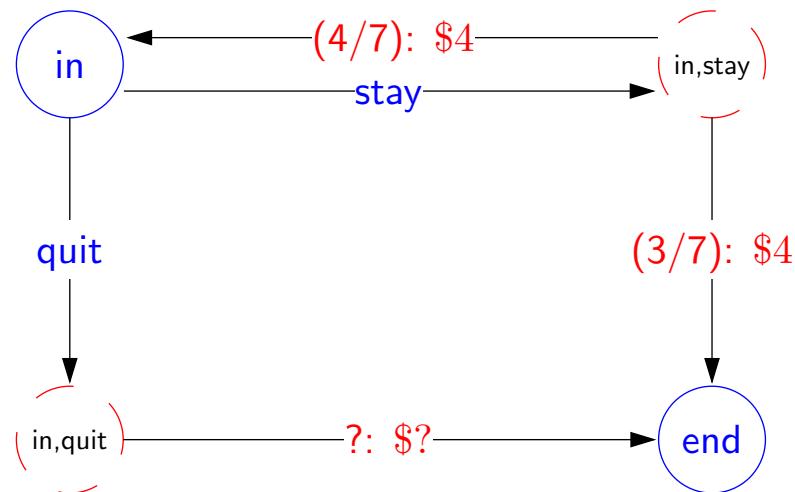
Data (following policy $\pi(s) = \text{stay}$):

[in; stay, 4, end]

- Estimates converge to true values (under certain conditions)
- With estimated MDP $(\hat{T}, \widehat{\text{Reward}})$, compute policy using value iteration

- The first idea is called **model-based** Monte Carlo, where we try to estimate the model (transitions and rewards) using Monte Carlo simulation.
- Monte Carlo is a standard way to estimate the expectation of a random variable by taking an average over samples of that random variable.
- Here, the data used to estimate the model is the sequence of states, actions, and rewards in the episode. Note that the samples being averaged are not independent (because they come from the same episode), but they do come from a Markov chain, so it can be shown that these estimates converge to the expectations by the ergodic theorem (a generalization of the law of large numbers for Markov chains).
- But there is one important caveat...

Problem



Problem: won't even see (s, a) if $a \neq \pi(s)$ ($a = \text{quit}$)



Key idea: exploration

To do reinforcement learning, need to explore the state space.

Solution: need π to **explore** explicitly (more on this later)

- So far, our policies have been deterministic, mapping s always to $\pi(s)$. However, if we use such a policy to generate our data, there are certain (s, a) pairs that we will never see and therefore never be able to estimate their Q-value and never know what the effect of those actions are.
- This problem points at the most important characteristic of reinforcement learning, which is the need for **exploration**. This distinguishes reinforcement learning from supervised learning, because now we actually have to act to get data, rather than just having data poured over us.
- To close off this point, we remark that if π is a non-deterministic policy which allows us to explore each state and action infinitely often (possibly over multiple episodes), then the estimates of the transitions and rewards will converge.
- Once we get an estimate for the transitions and rewards, we can simply plug them into our MDP and solve it using standard value or policy iteration to produce a policy.
- Notation: we put hats on quantities that are estimated from data $(\hat{Q}_{\text{opt}}, \hat{T})$ to distinguish from the true quantities (Q_{opt}, T) .

From model-based to model-free

$$\hat{Q}_{\text{opt}}(s, a) = \sum_{s'} \hat{T}(s, a, s') [\widehat{\text{Reward}}(s, a, s') + \gamma \hat{V}_{\text{opt}}(s')]$$

All that matters for prediction is (estimate of) $Q_{\text{opt}}(s, a)$.



Key idea: model-free learning

Try to estimate $Q_{\text{opt}}(s, a)$ directly.

- Taking a step back, if our goal is to just find good policies, all we need is to get a good estimate of \hat{Q}_{opt} . From that perspective, estimating the model (transitions and rewards) was just a means towards an end. Why not just cut to the chase and estimate \hat{Q}_{opt} directly? This is called **model-free** learning, where we don't explicitly estimate the transitions and rewards. Model-free learning is the equivalent of learning the expected utility at chance nodes instead of costs as edge weights like we were doing in the previous slide.

Model-free Monte Carlo

Data (following policy π):

$$s_0; a_1, r_1, s_1; a_2, r_2, s_2; a_3, r_3, s_3; \dots; a_n, r_n, s_n$$

Recall:

$Q_\pi(s, a)$ is expected utility starting at s , first taking action a , and then following policy π

Utility:

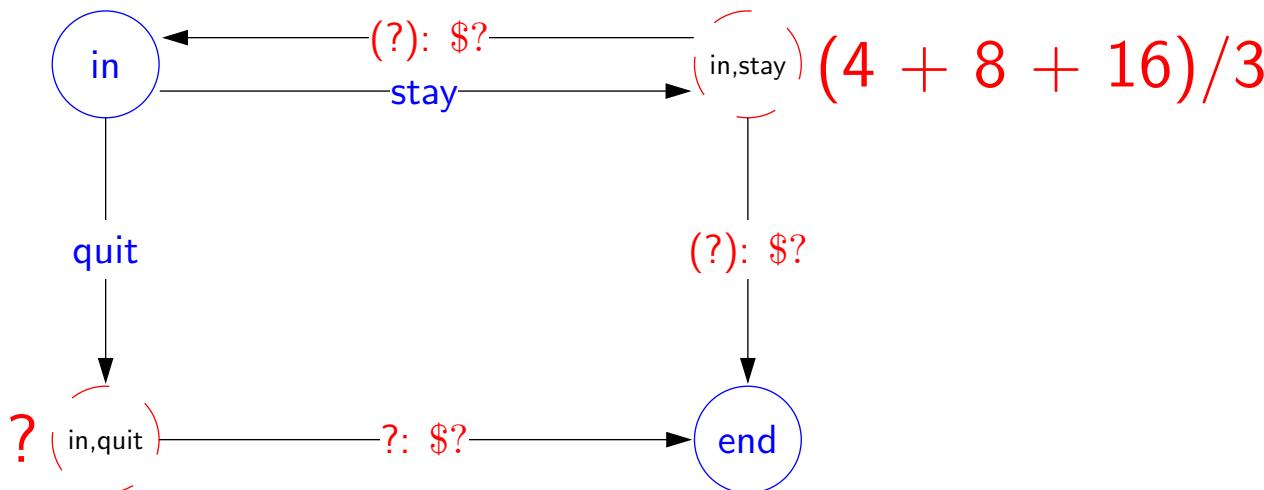
$$u_t = r_t + \gamma \cdot r_{t+1} + \gamma^2 \cdot r_{t+2} + \dots$$

Estimate:

$$\hat{Q}_\pi(s, a) = \text{average of } u_t \text{ where } s_{t-1} = s, a_t = a$$

(and s, a doesn't occur in s_0, \dots, s_{t-2})

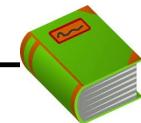
Model-free Monte Carlo



Data (following policy $\pi(s) = \text{stay}$):

[in; stay, 4, in; stay, 4, in; stay, 4, in; stay, 4, end]

Note: we are estimating Q_π now, not Q_{opt}



Definition: on-policy versus off-policy

On-policy: estimate the value of data-generating policy

Off-policy: estimate the value of another policy

- Recall that $Q_\pi(s, a)$ is the expected utility starting at s , taking action a , and the following π .
- In terms of the data, define u_t to be the discounted sum of rewards starting with r_t .
- Observe that $Q_\pi(s_{t-1}, a_t) = \mathbb{E}[u_t]$; that is, if we're at state s_{t-1} and take action a_t , the average value of u_t is $Q_\pi(s_{t-1}, a_t)$.
- But that particular state and action pair (s, a) will probably show up many times. If we take the average of u_t over all the times that $s_{t-1} = s$ and $a_t = a$, then we obtain our Monte Carlo estimate $\hat{Q}_\pi(s, a)$. Note that nowhere do we need to talk about transitions or immediate rewards; the only thing that matters is total rewards resulting from (s, a) pairs.
- One technical note is that for simplicity, we only consider $s_{t-1} = s, a_t = a$ for which the (s, a) doesn't show up later. This is not necessary for the algorithm to work, but it is easier to analyze and think about.
- Model-free Monte Carlo depends strongly on the policy π that is followed; after all it's computing Q_π . Because the value being computed is dependent on the policy used to generate the data, we call this an **on-policy** algorithm. In contrast, model-based Monte Carlo is **off-policy**, because the model we estimated did not depend on the exact policy (as long as it was able to explore all (s, a) pairs).

Model-free Monte Carlo (equivalences)

Data (following policy π):

$$s_0; a_1, r_1, s_1; a_2, r_2, s_2; a_3, r_3, s_3; \dots; a_n, r_n, s_n$$

Original formulation

$$\hat{Q}_\pi(s, a) = \text{average of } u_t \text{ where } s_{t-1} = s, a_t = a$$

Equivalent formulation (convex combination)

On each (s, a, u) :

$$\eta = \frac{1}{1 + (\# \text{ updates to } (s, a))}$$

$$\hat{Q}_\pi(s, a) \leftarrow (1 - \eta)\hat{Q}_\pi(s, a) + \eta u$$

[whiteboard: u_1, u_2, u_3]

- Over the next few slides, we will interpret model-free Monte Carlo in several ways. This is the same algorithm, just viewed from different perspectives. This will give us some more intuition and allow us to develop other algorithms later.
- The first interpretation is thinking in terms of **interpolation**. Instead of thinking of averaging as a batch operation that takes a list of numbers (realizations of u_t) and computes the mean, we can view it as an iterative procedure for building the mean as new numbers are coming in.
- In particular, it's easy to work out for a small example that averaging is equivalent to just interpolating between the old value $\hat{Q}_\pi(s, a)$ (current estimate) and the new value u (data). The interpolation ratio η is set carefully so that u contributes exactly the right amount to the average.
- Advanced: in practice, we would constantly improve the policy π constantly over time. In this case, we might want to set η to something that doesn't decay as quickly (for example, $\eta = 1/\sqrt{\#\text{ updates to } (s, a)}$). This rate implies that a new example contributes more than an old example, which is desirable so that $\hat{Q}_\pi(s, a)$ reflects the more recent policy rather than the old policy.

Model-free Monte Carlo (equivalences)

─ Equivalent formulation (convex combination) ─

On each (s, a, u) :

$$\hat{Q}_\pi(s, a) \leftarrow (1 - \eta) \hat{Q}_\pi(s, a) + \eta u$$

─ Equivalent formulation (stochastic gradient) ─

On each (s, a, u) :

$$\hat{Q}_\pi(s, a) \leftarrow \hat{Q}_\pi(s, a) - \eta [\underbrace{\hat{Q}_\pi(s, a)}_{\text{prediction}} - \underbrace{u}_{\text{target}}]$$

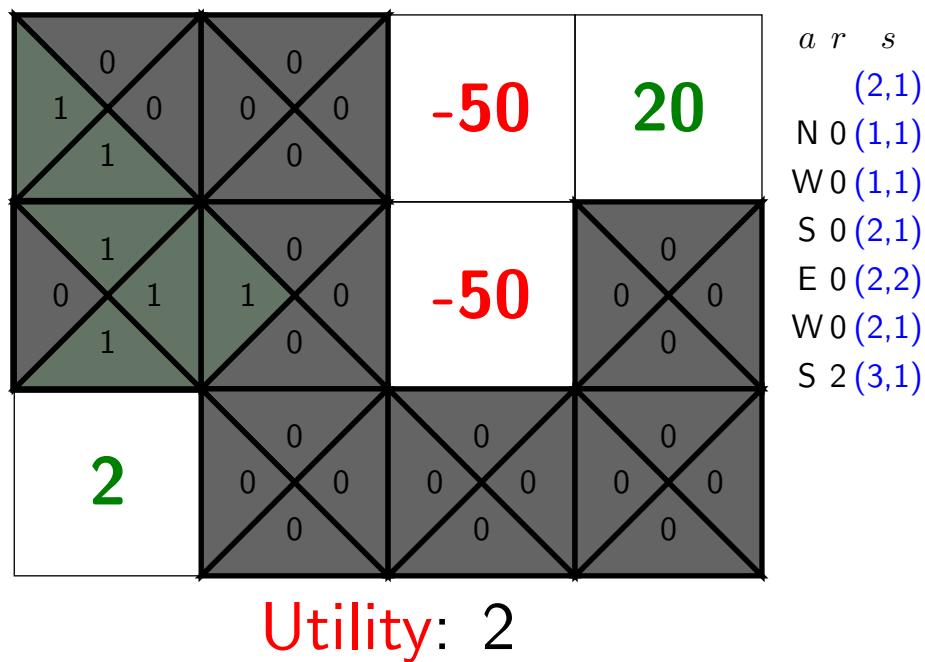
Implied objective: least squares regression

$$(\hat{Q}_\pi(s, a) - u)^2$$

- The second equivalent formulation is making the update look like a stochastic gradient update. Indeed, if we think about each (s, a, u) triple as an example (where (s, a) is the input and u is the output), then the model-free Monte Carlo is just performing stochastic gradient descent on a least squares regression problem, where the weight vector is \hat{Q}_π (which has dimensionality SA) and there is one feature template " (s, a) equals ___".
- The stochastic gradient descent view will become particularly relevant when we use non-trivial features on (s, a) .

Volcanic model-free Monte Carlo

Run (or press ctrl-enter)



- Let's run model-free Monte Carlo on the volcano crossing example. `slipProb` is zero to make things simpler. We are showing the Q-values: for each state, we have four values, one for each action.
- Here, our exploration policy is one that chooses an action uniformly at random.
- Try pressing "Run" multiple times to understand how the Q-values are set.
- Then try increasing `numEpisodes`, and seeing how the Q-values of this policy become more accurate.
- You will notice that a random policy has a very hard time reaching the 20.



Roadmap

Reinforcement learning

Monte Carlo methods

Bootstrapping methods

Covering the unknown

Summary

Using the utility

Data (following policy $\pi(s) = \text{stay}$):

[in; stay , 4, end]	$u = 4$
[in; stay , 4, in; stay , 4, end]	$u = 8$
[in; stay , 4, in; stay , 4, in; stay , 4, end]	$u = 12$
[in; stay , 4, in; stay , 4, in; stay , 4, in; stay , 4, end]	$u = 16$



Algorithm: model-free Monte Carlo

On each (s, a, u) :

$$\hat{Q}_\pi(s, a) \leftarrow (1 - \eta)\hat{Q}_\pi(s, a) + \eta \underbrace{u}_{\text{data}}$$

Using the reward + Q-value

Current estimate: $\hat{Q}_\pi(s, \text{stay}) = 11$

Data (following policy $\pi(s) = \text{stay}$):

[in; stay, 4, end] $4 + 0$

[in; stay, 4, in; stay, 4, end] $4 + 11$

[in; stay, 4, in; stay, 4, in; stay, 4, end] $4 + 11$

[in; stay, 4, in; stay, 4, in; stay, 4, in; stay, 4, end] $4 + 11$



Algorithm: SARSA

On each (s, a, r, s', a') :

$$\hat{Q}_\pi(s, a) \leftarrow (1 - \eta)\hat{Q}_\pi(s, a) + \eta [\underbrace{r}_{\text{data}} + \gamma \underbrace{\hat{Q}_\pi(s', a')}_{\text{estimate}}]$$

- Broadly speaking, reinforcement learning algorithms interpolate between new data (which specifies the **target** value) and the old estimate of the value (the **prediction**).
- Model-free Monte Carlo's target was u , the discounted sum of rewards after taking an action. However, u itself is just an estimate of $Q_\pi(s, a)$. If the episode is long, u will be a pretty lousy estimate. This is because u only corresponds to one episode out of a mind-blowing exponential (in the episode length) number of possible episodes, so as the episode lengthens, it becomes an increasingly less representative sample of what could happen. Can we produce better estimate of $Q_\pi(s, a)$?
- An alternative to model-free Monte Carlo is SARSA, whose target is $r + \gamma \hat{Q}_\pi(s', a')$. Importantly, SARSA's target is a combination of the data (the first step) and the estimate (for the rest of the steps). In contrast, model-free Monte Carlo's u is taken purely from the data.

Model-free Monte Carlo versus SARSA



Key idea: bootstrapping

SARSA uses estimate $\hat{Q}_\pi(s, a)$ instead of just raw data u .

u

based on one path

unbiased

large variance

wait until end to update

$r + \hat{Q}_\pi(s', a')$

based on estimate

biased

small variance

can update immediately

- The main advantage that SARSA offers over model-free Monte Carlo is that we don't have to wait until the end of the episode to update the Q-value.
- If the estimates are already pretty good, then SARSA will be more reliable since u is based on only one path whereas $\hat{Q}_\pi(s', a')$ is based on all the ones that the learner has seen before.
- Advanced: We can actually interpolate between model-free Monte Carlo (all rewards) and SARSA (one reward). For example, we could update towards $r_t + \gamma r_{t+1} + \gamma^2 \hat{Q}_\pi(s_{t+1}, a_{t+2})$ (two rewards). We can even combine all of these updates, which results in an algorithm called SARSA(λ), where λ determines the relative weighting of these targets. See the Sutton/Barto reinforcement learning book (chapter 7) for an excellent introduction.
- Advanced: There is also a version of these algorithms that estimates the value function V_π instead of Q_π . Value functions aren't enough to choose actions unless you actually know the transitions and rewards. Nonetheless, these are useful in game playing where we actually know the transition and rewards, but the state space is just too large to compute the value function exactly.



Question

Which of the following algorithms allows you to estimate $Q_{\text{opt}}(s, a)$ (select all that apply)?

model-based Monte Carlo

model-free Monte Carlo

SARSA

- Model-based Monte Carlo estimates the transitions and rewards, which fully specifies the MDP. With the MDP, you can estimate anything you want, including computing $Q_{\text{opt}}(s, a)$
- Model-free Monte Carlo and SARSA are on-policy algorithms, so they only give you $\hat{Q}_{\pi}(s, a)$, which is specific to a policy π . These will not provide direct estimates of $Q_{\text{opt}}(s, a)$.

Q-learning

Problem: model-free Monte Carlo and SARSA only estimate Q_π , but want Q_{opt} to act optimally

Output	MDP	reinforcement learning
Q_π	policy evaluation	model-free Monte Carlo, SARSA
Q_{opt}	value iteration	Q-learning

- Recall our goal is to get an optimal policy, which means estimating Q_{opt} .
- The situation is as follows: Our two methods (model-free Monte Carlo and SARSA) are model-free, but only produce estimates Q_{π} . We have one algorithm, model-based Monte Carlo, which can be used to produce estimates of Q_{opt} , but is model-based. Can we get an estimate of Q_{opt} in a model-free manner?
- The answer is yes, and Q-learning is an **off-policy** algorithm that accomplishes this.
- One can draw an analogy between reinforcement learning algorithms and the classic MDP algorithms. MDP algorithms are offline, RL algorithms are online. In both cases, algorithms either output the Q-values for a fixed policy or the optimal Q-values.

Q-learning

MDP recurrence:

$$Q_{\text{opt}}(s, a) = \sum_{s'} T(s, a, s')[\text{Reward}(s, a, s') + \gamma V_{\text{opt}}(s')]$$



Algorithm: Q-learning [Watkins/Dayan, 1992]

On each (s, a, r, s') :

$$\hat{Q}_{\text{opt}}(s, a) \leftarrow (1 - \eta) \underbrace{\hat{Q}_{\text{opt}}(s, a)}_{\text{prediction}} + \eta \underbrace{(r + \gamma \hat{V}_{\text{opt}}(s'))}_{\text{target}}$$

$$\text{Recall: } \hat{V}_{\text{opt}}(s') = \max_{a' \in \text{Actions}(s')} \hat{Q}_{\text{opt}}(s', a')$$

- To derive Q-learning, it is instructive to look back at the MDP recurrence for Q_{opt} . There are several changes that take us from the MDP recurrence to Q-learning. First, we don't have an expectation over s' , but only have one sample s' .
- Second, because of this, we don't want to just replace $\hat{Q}_{\text{opt}}(s, a)$ with the target value, but want to interpolate between the old value (prediction) and the new value (target).
- Third, we replace the actual reward $\text{Reward}(s, a, s')$ with the observed reward r (when the reward function is deterministic, the two are the same).
- Finally, we replace $V_{\text{opt}}(s')$ with our current estimate $\hat{V}_{\text{opt}}(s')$.
- Importantly, the estimated optimal value $\hat{V}_{\text{opt}}(s')$ involves a maximum over actions rather than taking the action of the policy. This max over a' rather than taking the a' based on the current policy is the principle difference between Q-learning and SARSA.

SARSA versus Q-learning



Algorithm: SARSA

On each (s, a, r, s', a') :

$$\hat{Q}_\pi(s, a) \leftarrow (1 - \eta)\hat{Q}_\pi(s, a) + \eta(r + \gamma\hat{Q}_\pi(s', a'))$$



Algorithm: Q-learning [Watkins/Dayan, 1992]

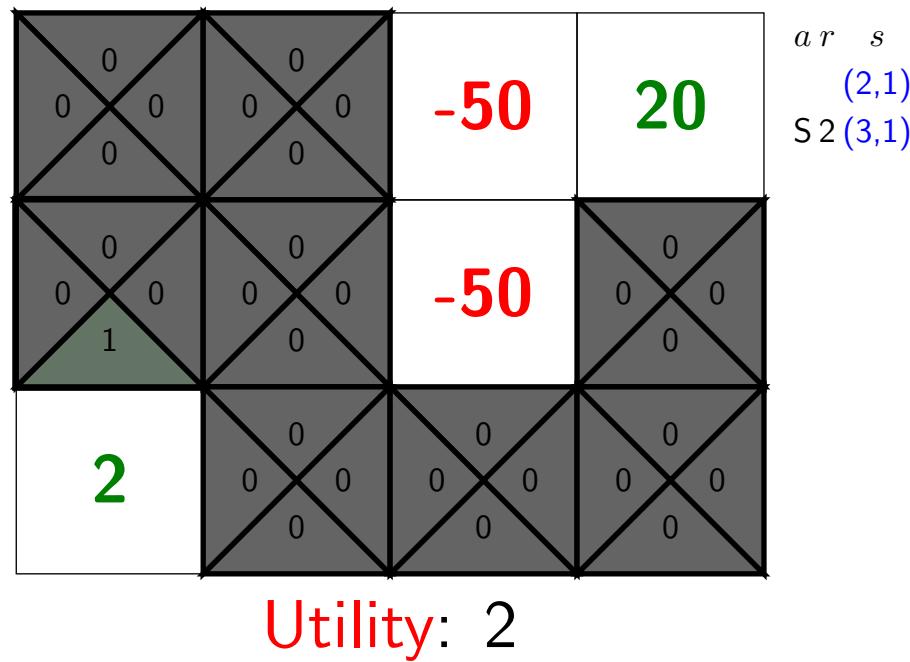
On each (s, a, r, s') :

$$\hat{Q}_{\text{opt}}(s, a) \leftarrow (1 - \eta)\hat{Q}_{\text{opt}}(s, a) + \eta(r + \gamma \max_{a' \in \text{Actions}(s')} \hat{Q}_{\text{opt}}(s', a'))$$

Volcanic SARSA and Q-learning

Run

(or press ctrl-enter)



- Let us try SARSA and Q-learning on the volcanic example.
- If you increase `numEpisodes` to 1000, SARSA will behave very much like model-free Monte Carlo, computing the value of the random policy.
- However, note that Q-learning is computing an estimate of $Q_{\text{opt}}(s, a)$, so the resulting Q-values will be very different. The average utility will not change since we are still following and being evaluated on the same random policy. This is an important point for **off-policy** methods: the online performance (average utility) is generally a lot worse and not representative of what the model has learned, which is captured in the estimated Q-values.



Roadmap

Reinforcement learning

Monte Carlo methods

Bootstrapping methods

Covering the unknown

Summary



Exploration



Algorithm: reinforcement learning template

For $t = 1, 2, 3, \dots$

Choose action $a_t = \pi_{\text{act}}(s_{t-1})$ (**how?**)

Receive reward r_t and observe new state s_t

Update parameters (**how?**)

$s_0; a_1, r_1, s_1; a_2, r_2, s_2; a_3, r_3, s_3; \dots; a_n, r_n, s_n$

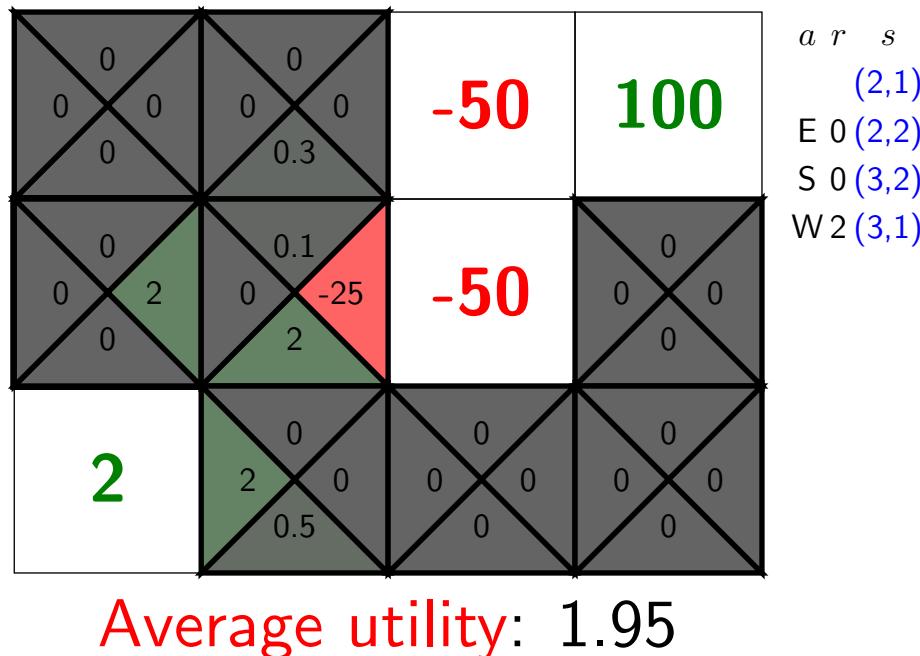
Which **exploration policy** π_{act} to use?

- We have so far given many algorithms for updating parameters (i.e., $\hat{Q}_\pi(s, a)$ or $\hat{Q}_{\text{opt}}(s, a)$). If we were doing supervised learning, we'd be done, but in reinforcement learning, we need to actually determine our **exploration policy** π_{act} to collect data for learning. Recall that we need to somehow make sure we get information about each (s, a) .
- We will discuss two complementary ways to get this information: (i) explicitly explore (s, a) or (ii) explore (s, a) implicitly by actually exploring (s', a') with similar features and generalizing.
- These two ideas apply to many RL algorithms, but let us specialize to Q-learning.

No exploration, all exploitation

Attempt 1: Set $\pi_{\text{act}}(s) = \arg \max_{a \in \text{Actions}(s)} \hat{Q}_{\text{opt}}(s, a)$

Run (or press ctrl-enter)



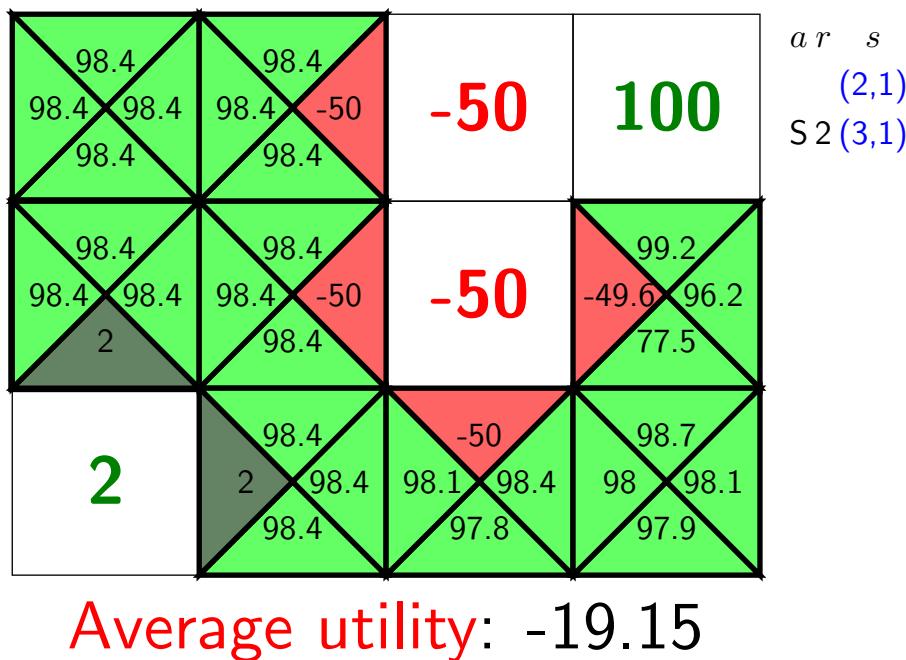
Problem: $\hat{Q}_{\text{opt}}(s, a)$ estimates are inaccurate, **too greedy!**

- The naive solution is to explore using the optimal policy according to the estimated Q-value $\hat{Q}_{\text{opt}}(s, a)$.
- But this fails horribly. In the example, once the agent discovers that there is a reward of 2 to be gotten by going south that becomes its optimal policy and it will not try any other action. The problem is that the agent is being too greedy.
- In the demo, if multiple actions have the same maximum Q-value, we choose randomly. Try clicking "Run" a few times, and you'll end up with minor variations.
- Even if you increase numEpisodes to 10000, nothing new gets learned.

No exploitation, all exploration

Attempt 2: Set $\pi_{\text{act}}(s) = \text{random from Actions}(s)$

Run (or press ctrl-enter)



Problem: average utility is low because exploration is **not guided**

- We can go to the other extreme and use an exploration policy that always chooses a random action. It will do a much better job of exploration, but it doesn't exploit what it learns and ends up with a very low utility.
- It is interesting to note that the value (average over utilities across all the episodes) can be quite small and yet the Q-values can be quite accurate. Recall that this is possible because Q-learning is an off-policy algorithm.

Exploration/exploitation tradeoff



Key idea: balance

Need to balance **exploration** and **exploitation**.



Examples from life: restaurants, routes, research

Epsilon-greedy

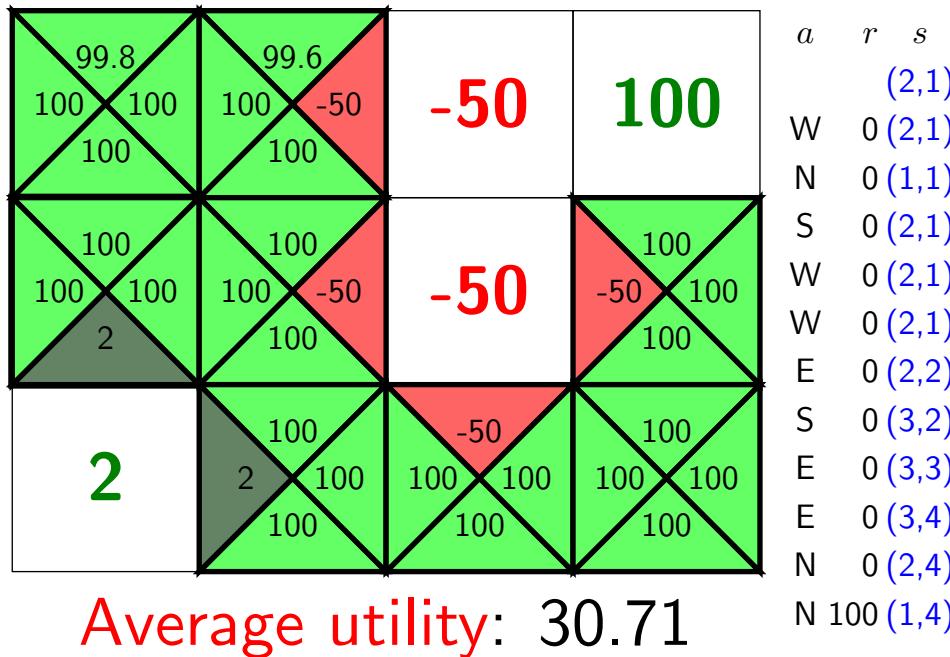


Algorithm: epsilon-greedy policy

$$\pi_{\text{act}}(s) = \begin{cases} \arg \max_{a \in \text{Actions}} \hat{Q}_{\text{opt}}(s, a) & \text{probability } 1 - \epsilon, \\ \text{random from Actions}(s) & \text{probability } \epsilon. \end{cases}$$

Run

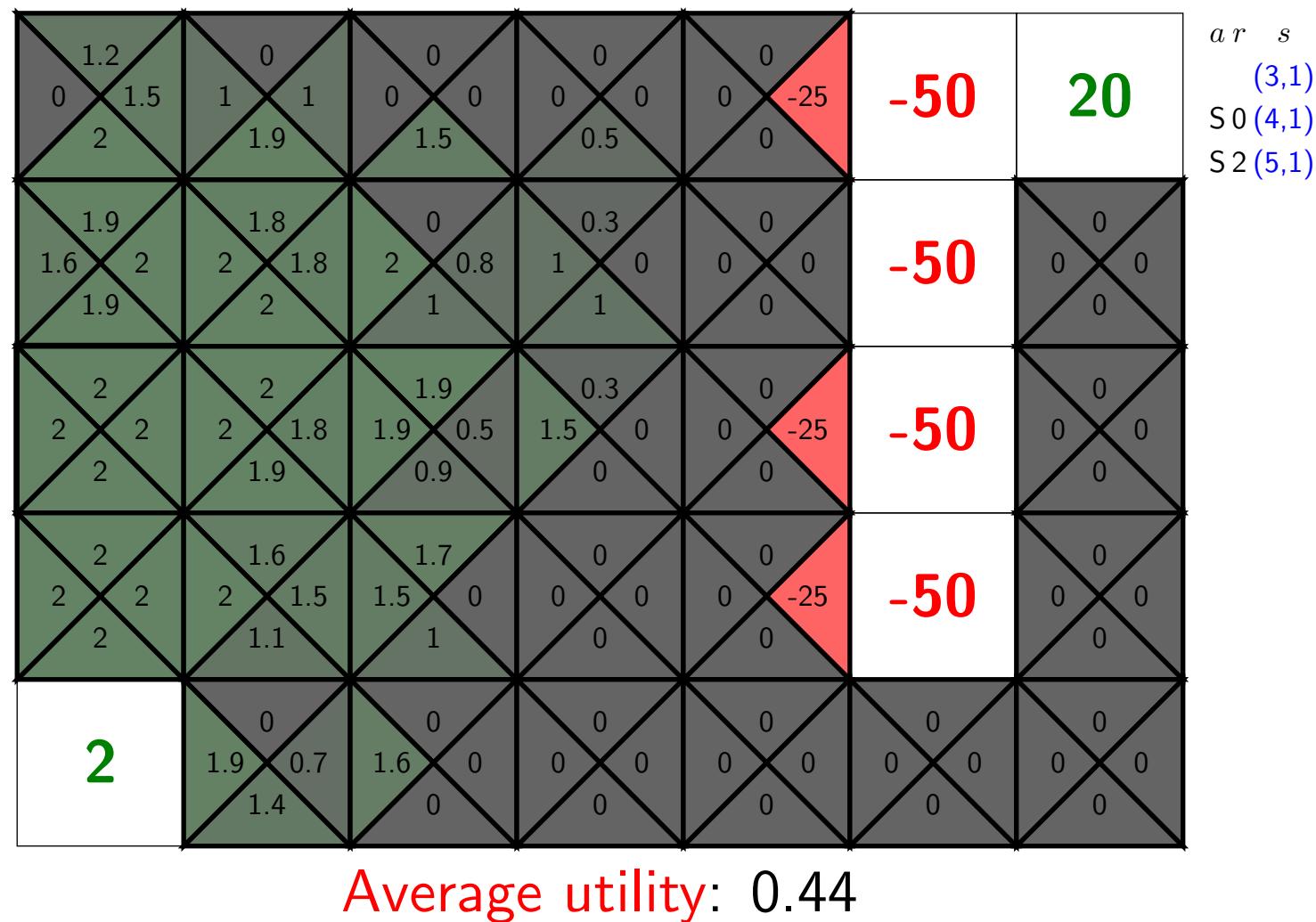
(or press ctrl-enter)



- The natural thing to do when you have two extremes is to interpolate between the two. The result is the **epsilon-greedy** algorithm which explores with probability ϵ and exploits with probability $1 - \epsilon$.
- It is natural to let ϵ decrease over time. When you're young, you want to explore a lot ($\epsilon = 1$). After a certain point, when you feel like you've seen all there is to see, then you start exploiting ($\epsilon = 0$).
- For example, we let $\epsilon = 1$ for the first third of the episodes, $\epsilon = 0.5$ for the second third, and $\epsilon = 0$ for the final third. This is not the optimal schedule. Try playing around with other schedules to see if you can do better.

Generalization

Problem: large state spaces, hard to explore



- Now we turn to another problem with vanilla Q-learning.
- In real applications, there can be millions of states, in which there's no hope for epsilon-greedy to explore everything in a reasonable amount of time.

Q-learning

Stochastic gradient update:

$$\hat{Q}_{\text{opt}}(s, a) \leftarrow \hat{Q}_{\text{opt}}(s, a) - \eta \underbrace{\hat{Q}_{\text{opt}}(s, a)}_{\text{prediction}} - \underbrace{(r + \gamma \hat{V}_{\text{opt}}(s'))}_{\text{target}}$$

This is **rote learning**: every $\hat{Q}_{\text{opt}}(s, a)$ has a different value

Problem: doesn't generalize to unseen states/actions

- If we revisit the Q-learning algorithm, and think about it through the lens of machine learning, you'll find that we've just been memorizing Q-values for each (s, a) , treating each pair independently.
- In other words, we haven't been generalizing, which is actually one of the most important aspects of learning!

Function approximation



Key idea: linear regression model

Define **features** $\phi(s, a)$ and **weights** \mathbf{w} :

$$\hat{Q}_{\text{opt}}(s, a; \mathbf{w}) = \mathbf{w} \cdot \phi(s, a)$$



Example: features for volcano crossing

$$\phi_1(s, a) = \mathbf{1}[a = W] \quad \phi_7(s, a) = \mathbf{1}[s = (5, *)]$$

$$\phi_2(s, a) = \mathbf{1}[a = E] \quad \phi_8(s, a) = \mathbf{1}[s = (*, 6)]$$

...

...

- **Function approximation** fixes this by parameterizing \hat{Q}_{opt} by a weight vector and a feature vector, as we did in linear regression.
- Recall that features are supposed to be properties of the state-action (s, a) pair that are indicative of the quality of taking action a in state s .
- The ramification is that all the states that have similar features will have similar Q-values. For example, suppose ϕ included the feature $\mathbf{1}[s = (*, 4)]$. If we were in state $(1, 4)$, took action E, and managed to get high rewards, then Q-learning with function approximation will propagate this positive signal to all positions in column 4 taking any action.
- In our example, we defined features on actions (to capture that moving east is generally good) and features on states (to capture the fact that the 6th column is best avoided, and the 5th row is generally a good place to travel to).

Function approximation



Algorithm: Q-learning with function approximation

On each (s, a, r, s') :

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \underbrace{\hat{Q}_{\text{opt}}(s, a; \mathbf{w})}_{\text{prediction}} - \underbrace{(r + \gamma \hat{V}_{\text{opt}}(s'))}_{\text{target}} \phi(s, a)$$

Implied objective function:

$$(\underbrace{\hat{Q}_{\text{opt}}(s, a; \mathbf{w})}_{\text{prediction}} - \underbrace{(r + \gamma \hat{V}_{\text{opt}}(s'))}_{\text{target}})^2$$

- We now turn our linear regression into an algorithm. Here, it is useful to adopt the stochastic gradient view of RL algorithms, which we developed a while back.
- We just have to write down the least squares objective and then compute the gradient with respect to w now instead of \hat{Q}_{opt} . The chain rule takes care of the rest.

Covering the unknown



Epsilon-greedy: balance the exploration/exploitation tradeoff

Function approximation: can generalize to unseen states



Summary so far

- Online setting: learn and take actions in the real world!
- Exploration/exploitation tradeoff
- Monte Carlo: estimate transitions, rewards, Q-values from data
- Bootstrapping: update towards target that depends on estimate rather than just raw data

- This concludes the technical part of reinforcement learning.
- The first part is to understand the setup: we are taking good actions in the world both to get rewards under our current model, but also to collect information about the world so we can learn a better model. This exposes the fundamental exploration/exploitation tradeoff, which is the hallmark of reinforcement learning.
- We looked at several algorithms: model-based Monte Carlo, model-free Monte Carlo, SARSA, and Q-learning. There were two complementary ideas here: using Monte Carlo approximation (approximating an expectation with a sample) and bootstrapping (using the model predictions to update itself).



Roadmap

Reinforcement learning

Monte Carlo methods

Bootstrapping methods

Covering the unknown

Summary

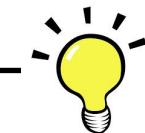
Challenges in reinforcement learning

Binary classification (sentiment classification, SVMs):

- Stateless, full feedback

Reinforcement learning (flying helicopters, Q-learning):

- Stateful, partial feedback



Key idea: partial feedback

Only learn about actions you take.



Key idea: state

Rewards depend on previous actions \Rightarrow can have delayed rewards.

States and information

	stateless	state
full feedback	supervised learning (binary classification)	supervised learning (structured prediction)
partial feedback	multi-armed bandits	reinforcement learning

- If we compare simple supervised learning (e.g., binary classification) and reinforcement learning, we see that there are two main differences that make learning harder.
- First, reinforcement learning requires the modeling of state. State means that the rewards across time steps are related. This results in **delayed rewards**, where we take an action and don't see the ramifications of it until much later.
- Second, reinforcement learning requires dealing with partial feedback (rewards). This means that we have to actively explore to acquire the necessary feedback.
- There are two problems that move towards reinforcement learning, each on a different axis. Structured prediction introduces the notion of state, but the problem is made easier by the fact that we have full feedback, which means that for every situation, we know which action sequence is the correct one; there is no need for exploration; we just have to update our weights to favor that correct path.
- Multi-armed bandits require dealing with partial feedback, but do not have the complexities of state. One can think of a multi-armed bandit problem as an MDP with unknown random rewards and one state. Exploration is necessary, but there is no temporal depth to the problem.

Deep reinforcement learning

just use a neural network for $\hat{Q}_{\text{opt}}(s, a)$

Playing Atari [Google DeepMind, 2013]:

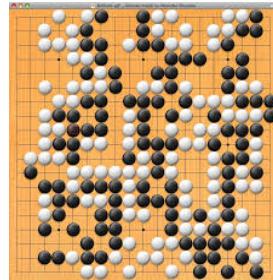


- last 4 frames (images) \Rightarrow 3-layer NN \Rightarrow keystroke
- ϵ -greedy, train over 10M frames with 1M replay memory
- Human-level performance on some games (breakout), less good on others (space invaders)

- Recently, there has been a surge of interest in reinforcement learning due to the success of neural networks. If one is performing reinforcement learning in a simulator, one can actually generate tons of data, which is suitable for rich functions such as neural networks.
- A recent success story is DeepMind, who successfully trained a neural network to represent the \hat{Q}_{opt} function for playing Atari games. The impressive part was the lack of prior knowledge involved: the neural network simply took as input the raw image and outputted keystrokes.

Deep reinforcement learning

- Policy gradient: train a policy $\pi(a | s)$ (say, a neural network) to directly maximize expected reward
- Google DeepMind's AlphaGo (2016), AlphaZero (2017)



- Andrej Karpathy's blog post

<http://karpathy.github.io/2016/05/31/rl>

- One other major class of algorithms we will not cover in this class is **policy gradient**. Whereas Q-learning attempts to estimate the value of the optimal policy, policy gradient methods optimize the policy to maximize expected reward, which is what we care about. Recall that when we went from model-based methods (which estimated the transition and reward functions) to model-free methods (which estimated the Q function), we moved closer to the thing that we want. Policy gradient methods take this farther and just focus on the only object that really matters at the end of the day, which is the policy that an agent follows.
- Policy gradient methods have been quite successful. For example, it was one of the components of AlphaGo, Google DeepMind's program that beat the world champion at Go. One can also combine value-based methods with policy-based methods in actor-critic methods to get the best of both worlds.
- There is a lot more to say about deep reinforcement learning. If you wish to learn more, Andrej Karpathy's blog post offers a nice introduction.

Applications



Autonomous helicopters: control helicopter to do maneuvers in the air



Backgammon: TD-Gammon plays 1-2 million games against itself, human-level performance



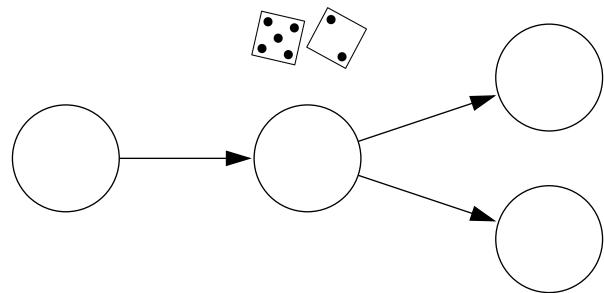
Elevator scheduling; send which elevators to which floors to maximize throughput of building



Managing datacenters; actions: bring up and shut down machine to minimize time/cost

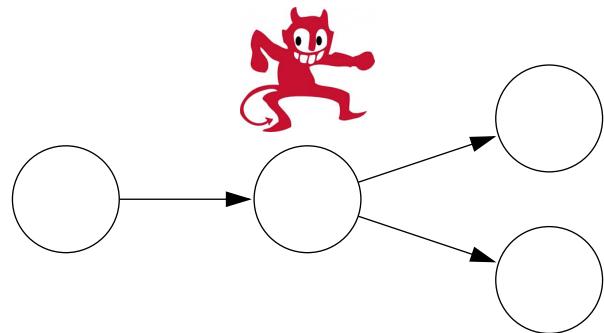
- There are many other applications of RL, which range from robotics to game playing to other infrastructural tasks. One could say that RL is so general that anything can be cast as an RL problem.
- For a while, RL only worked for small toy problems or settings where there were a lot of prior knowledge / constraints. Deep RL — the use of powerful neural networks with increased compute — has vastly expanded the realm of problems which are solvable by RL.

Markov decision processes: against nature (e.g., Blackjack)



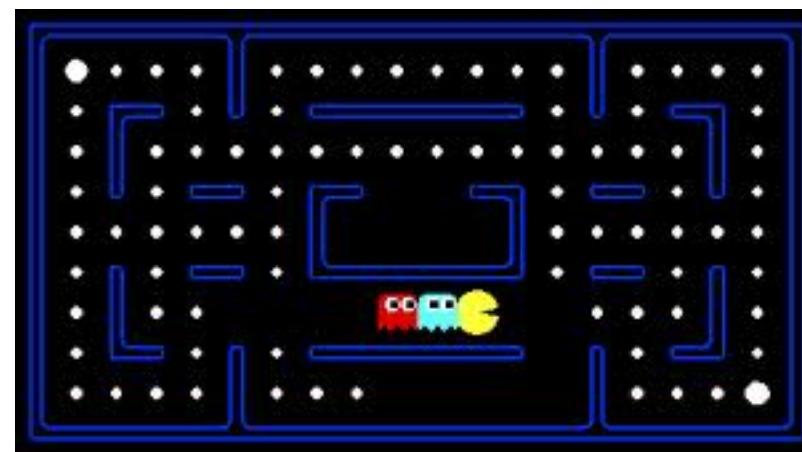
Next time...

Adversarial games: against opponent (e.g., chess)

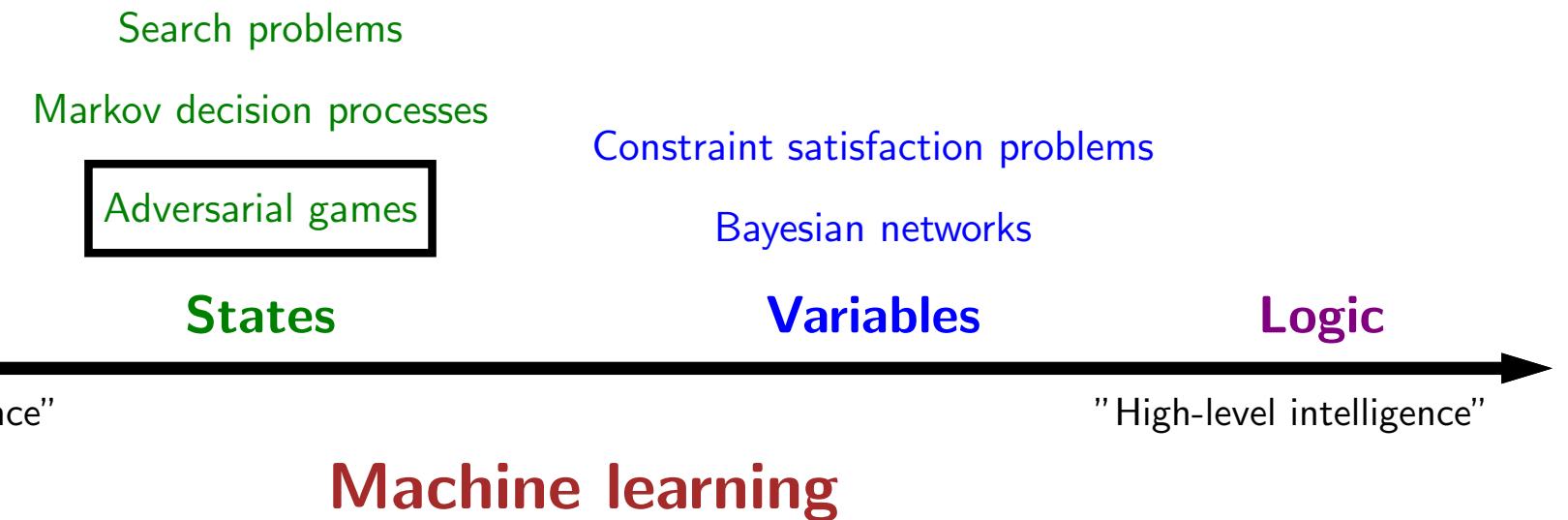




Games I

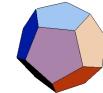


Course plan



- This lecture will be about games, which have been one of the main testbeds for developing AI programs since the early days of AI. Games are distinguished from the other tasks that we've considered so far in this class in that they make explicit the presence of other agents, whose utility is not generally aligned with ours. Thus, the optimal strategy (policy) for us will depend on the strategies of these agents. Moreover, their strategies are often unknown and adversarial. How do we reason about this?

A simple game



Example: game 1

You choose one of the three bins.

I choose a number from that bin.

Your goal is to maximize the chosen number.

A

-50 50

B

1 3

C

-5 15

- Which bin should you pick? Depends on your mental model of the other player (me).
- If you think I'm working with you (unlikely), then you should pick A in hopes of getting 50. If you think I'm against you (likely), then you should pick B as to guard against the worst case (get 1). If you think I'm just acting uniformly at random, then you should pick C so that on average things are reasonable (get 5 in expectation).



Roadmap

Games, expectimax

Minimax, expectiminimax

Evaluation functions

Alpha-beta pruning

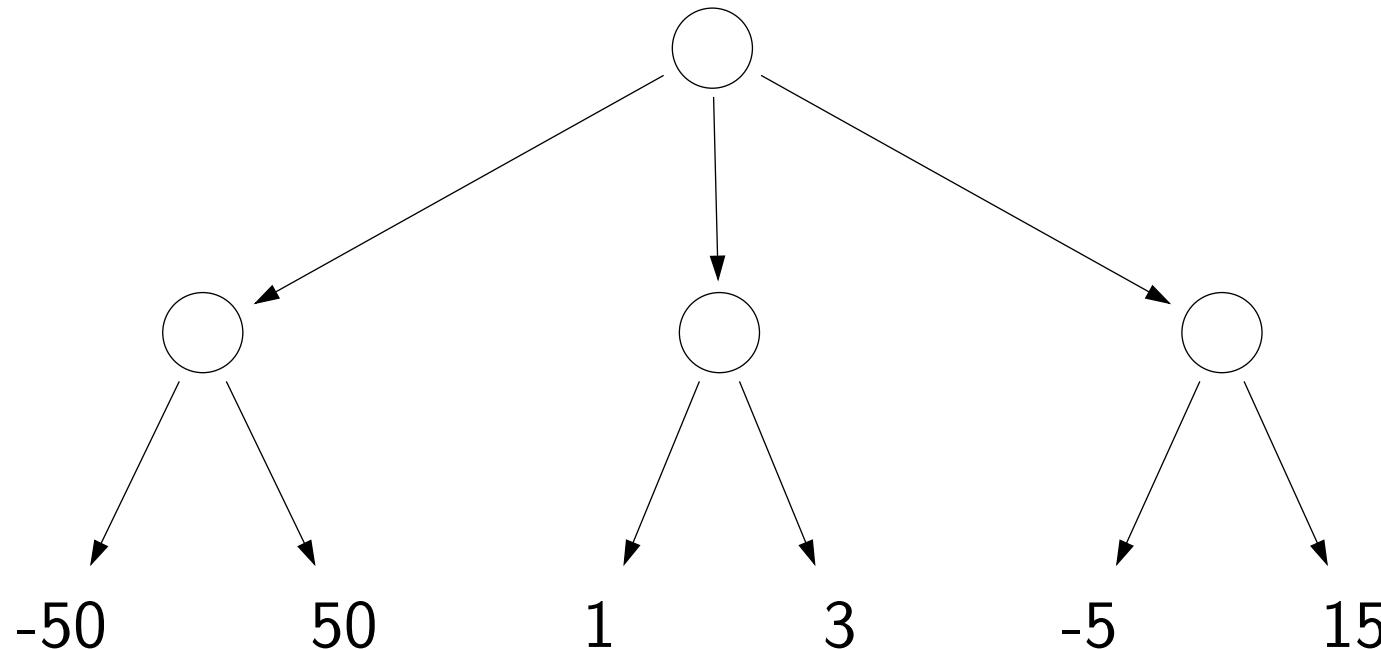
Game tree



Key idea: game tree

Each node is a decision point for a player.

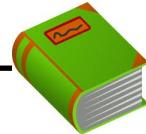
Each root-to-leaf path is a possible outcome of the game.



- Just as in search problems, we will use a tree to describe the possibilities of the game. This tree is known as a **game tree**.
- Note: We could also think of a game graph to capture the fact that there are multiple ways to arrive at the same game state. However, all our algorithms will operate on the tree rather than the graph since games generally have enormous state spaces, and we will have to resort to algorithms similar to backtracking search for search problems.

Two-player zero-sum games

Players = {agent, opp}



Definition: two-player zero-sum game

s_{start} : starting state

$\text{Actions}(s)$: possible actions from state s

$\text{Succ}(s, a)$: resulting state if choose action a in state s

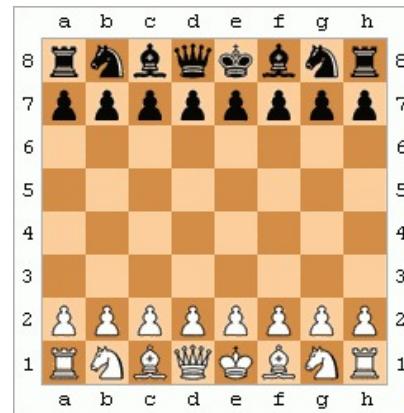
$\text{IsEnd}(s)$: whether s is an end state (game over)

$\text{Utility}(s)$: agent's utility for end state s

$\text{Player}(s) \in \text{Players}$: player who controls state s

- In this lecture, we will specialize to **two-player zero-sum** games, such as chess. To be more precise, we will consider games in which people take turns (unlike rock-paper-scissors) and where the state of the game is fully-observed (unlike poker, where you don't know the other players' hands). By default, we will use the term **game** to refer to this restricted form.
- We will assume the two players are named agent (this is your program) and opp (the opponent). Zero-sum means that the utility of the agent is negative the utility of the opponent (equivalently, the sum of the two utilities is zero).
- Following our approach to search problems and MDPs, we start by formalizing a game. Since games are a type of state-based model, much of the skeleton is the same: we have a start state, actions from each state, a deterministic successor state for each state-action pair, and a test on whether a state is at the end.
- The main difference is that each state has a designated Player(s), which specifies whose turn it is. A player p only gets to choose the action for the states s such that $\text{Player}(s) = p$.
- Another difference is that instead of having edge costs in search problems or rewards in MDPs, we will instead have a utility function $\text{Utility}(s)$ defined only at the end states. We could have used edge costs and rewards for games (in fact, that's strictly more general), but having all the utility at the end states emphasizes the all-or-nothing aspect of most games. You don't get utility for capturing pieces in chess; you only get utility if you win the game. This ultra-delayed utility makes games hard.

Example: chess



Players = {white, black}

State s : (position of all pieces, whose turn it is)

Actions(s): legal chess moves that Player(s) can make

IsEnd(s): whether s is checkmate or draw

Utility(s): $+\infty$ if white wins, 0 if draw, $-\infty$ if black wins

- Chess is a canonical example of a two-player zero-sum game. In chess, the state must represent the position of all pieces, and importantly, whose turn it is (white or black).
- Here, we are assuming that white is the agent and black is the opponent. White moves first and is trying to maximize the utility, whereas black is trying to minimize the utility.
- In most games that we'll consider, the utility is degenerate in that it will be $+\infty$, $-\infty$, or 0.

Characteristics of games

- All the utility is at the end state



- Different players in control at different states



- There are two important characteristics of games which make them hard.
- The first is that the utility is only at the end state. In typical search problems and MDPs that we might encounter, there are costs and rewards associated with each edge. These intermediate quantities make the problem easier to solve. In games, even if there are cues that indicate how well one is doing (number of pieces, score), technically all that matters is what happens at the end. In chess, it doesn't matter how many pieces you capture, your goal is just to checkmate the opponent's king.
- The second is the recognition that there are other people in the world! In search problems, you (the agent) controlled all actions. In MDPs, we already hinted at the loss of control where nature controlled the chance nodes, but we assumed we knew what distribution nature was using to transition. Now, we have another player that controls certain states, who is probably out to get us.

The halving game



Problem: halving game

Start with a number N .

Players take turns either decrementing N or replacing it with $\lfloor \frac{N}{2} \rfloor$.

The player that is left with 0 wins.

[semi-live solution: HalvingGame]

Policies

Deterministic policies: $\pi_p(s) \in \text{Actions}(s)$

action that player p takes in state s

Stochastic policies $\pi_p(s, a) \in [0, 1]$:

probability of player p taking action a in state s

[semi-live solution: `humanPolicy`]

- Following our presentation of MDPs, we revisit the notion of a **policy**. Instead of having a single policy π , we have a policy π_p for each player $p \in \text{Players}$. We require that π_p only be defined when it's p 's turn; that is, for states s such that $\text{Player}(s) = p$.
- It will be convenient to allow policies to be stochastic. In this case, we will use $\pi_p(s, a)$ to denote the probability of player p choosing action a in state s .
- We can think of an MDP as a game between the agent and nature. The states of the game are all MDP states s and all chance nodes (s, a) . It's the agent's turn on the MDP states s , and the agent acts according to π_{agent} . It's nature's turn on the chance nodes. Here, the actions are successor states s' , and nature chooses s' with probability given by the transition probabilities of the MDP: $\pi_{\text{nature}}((s, a), s') = T(s, a, s')$.

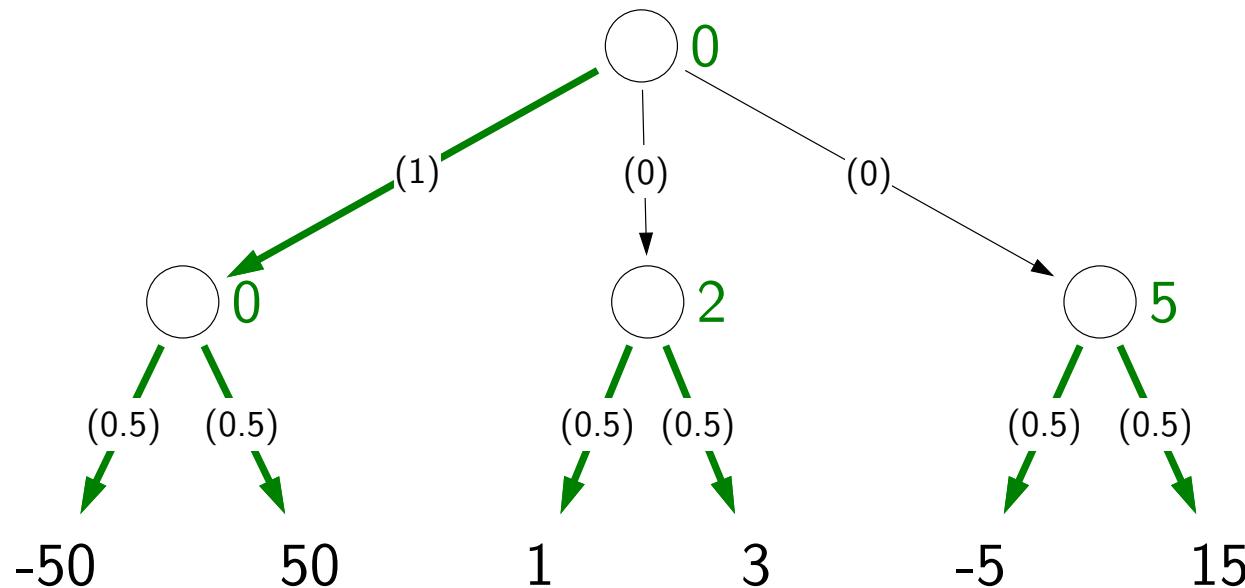
Game evaluation example



Example: game evaluation

$$\pi_{\text{agent}}(s) = A$$

$$\pi_{\text{opp}}(s, a) = \frac{1}{2} \text{ for } a \in \text{Actions}(s)$$

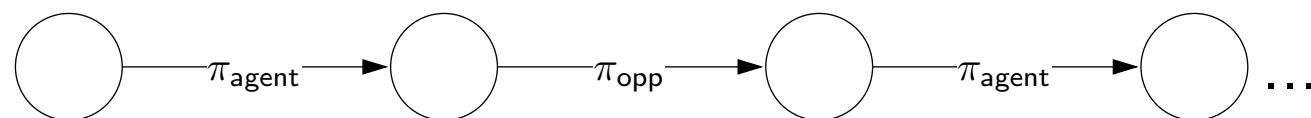


$$V_{\text{eval}}(s_{\text{start}}) = 0$$

- Given two policies π_{agent} and π_{opp} , what is the (agent's) expected utility? That is, if the agent and the opponent were to play their (possibly stochastic) policies a large number of times, what would be the average utility? Remember, since we are working with zero-sum games, the opponent's utility is the negative of the agent's utility.
- Given the game tree, we can recursively compute the value (expected utility) of each node in the tree. The value of a node is the weighted average of the values of the children where the weights are given by the probabilities of taking various actions given by the policy at that node.

Game evaluation recurrence

Analogy: recurrence for policy evaluation in MDPs



Value of the game:

$$V_{\text{eval}}(s) = \begin{cases} \text{Utility}(s) & \text{IsEnd}(s) \\ \sum_{a \in \text{Actions}(s)} \pi_{\text{agent}}(s, a) V_{\text{eval}}(\text{Succ}(s, a)) & \text{Player}(s) = \text{agent} \\ \sum_{a \in \text{Actions}(s)} \pi_{\text{opp}}(s, a) V_{\text{eval}}(\text{Succ}(s, a)) & \text{Player}(s) = \text{opp} \end{cases}$$

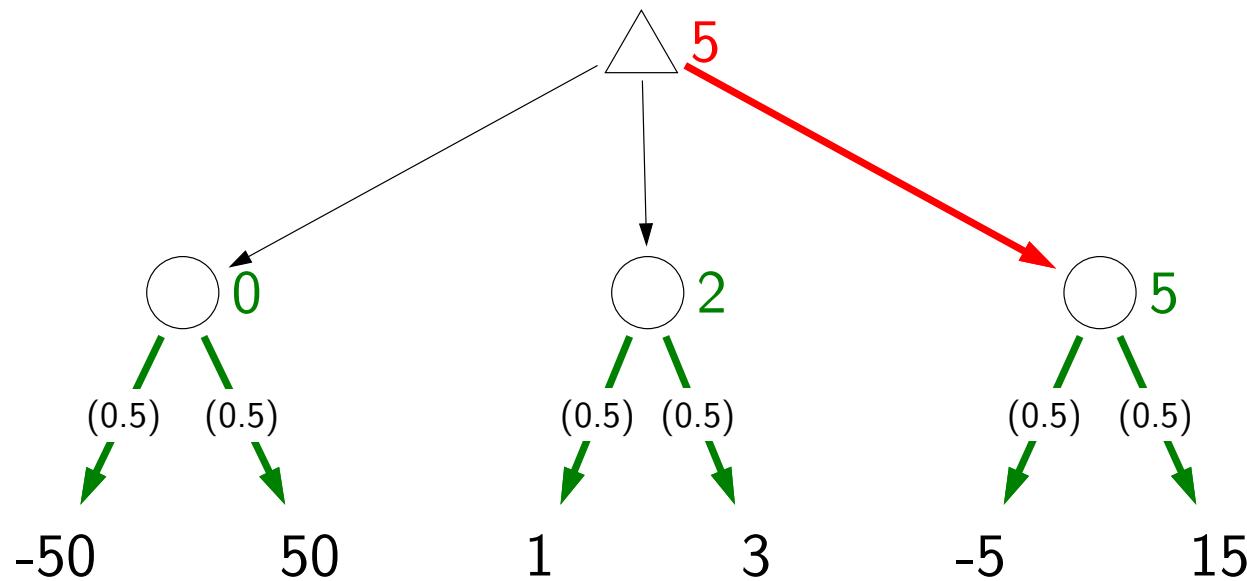
- More generally, we can write down a recurrence for $V_{\text{eval}}(s)$, which is the **value** (expected utility) of the game at state s .
- There are three cases: If the game is over ($\text{IsEnd}(s)$), then the value is just the utility $\text{Utility}(s)$. If it's the agent's turn, then we compute the expectation over the value of the successor resulting from the agent choosing an action according to $\pi_{\text{agent}}(s, a)$. If it's the opponent's turn, we compute the expectation with respect to π_{opp} instead.

Expectimax example



Example: expectimax

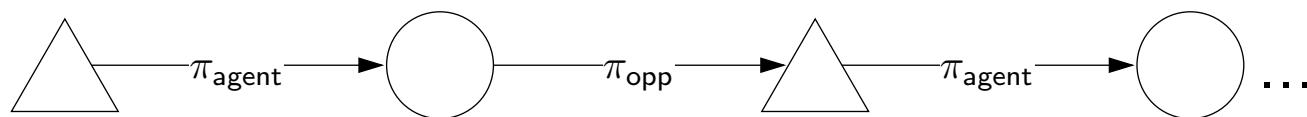
$$\pi_{\text{opp}}(s, a) = \frac{1}{2} \text{ for } a \in \text{Actions}(s)$$



- Game evaluation just gave us the value of the game with two fixed policies π_{agent} and π_{opp} . But we are not handed a policy π_{agent} ; we are trying to find the best policy. Expectimax gives us exactly that.
- In the game tree, we will now use an upward-pointing triangle to denote states where the player is maximizing over actions (we call them **max nodes**).
- At max nodes, instead of averaging with respect to a policy, we take the max of the values of the children.
- This computation produces the **expectimax value** $V_{\text{exptmax}}(s)$ for a state s , which is the maximum expected utility of any agent policy when playing with respect to a fixed and known opponent policy π_{opp} .

Expectimax recurrence

Analogy: recurrence for value iteration in MDPs



$$V_{\text{exptmax}}(s) = \begin{cases} \text{Utility}(s) & \text{IsEnd}(s) \\ \max_{a \in \text{Actions}(s)} V_{\text{exptmax}}(\text{Succ}(s, a)) & \text{Player}(s) = \text{agent} \\ \sum_{a \in \text{Actions}(s)} \pi_{\text{opp}}(s, a) V_{\text{exptmax}}(\text{Succ}(s, a)) & \text{Player}(s) = \text{opp} \end{cases}$$

- The recurrence for the expectimax value V_{exptmax} is exactly the same as the one for the game value V_{eval} , except that we maximize over the agent's actions rather than following a fixed agent policy (which we don't know now).
- Where game evaluation was the analogue of policy evaluation for MDPs, expectimax is the analogue of value iteration.



Roadmap

Games, expectimax

Minimax, expectiminimax

Evaluation functions

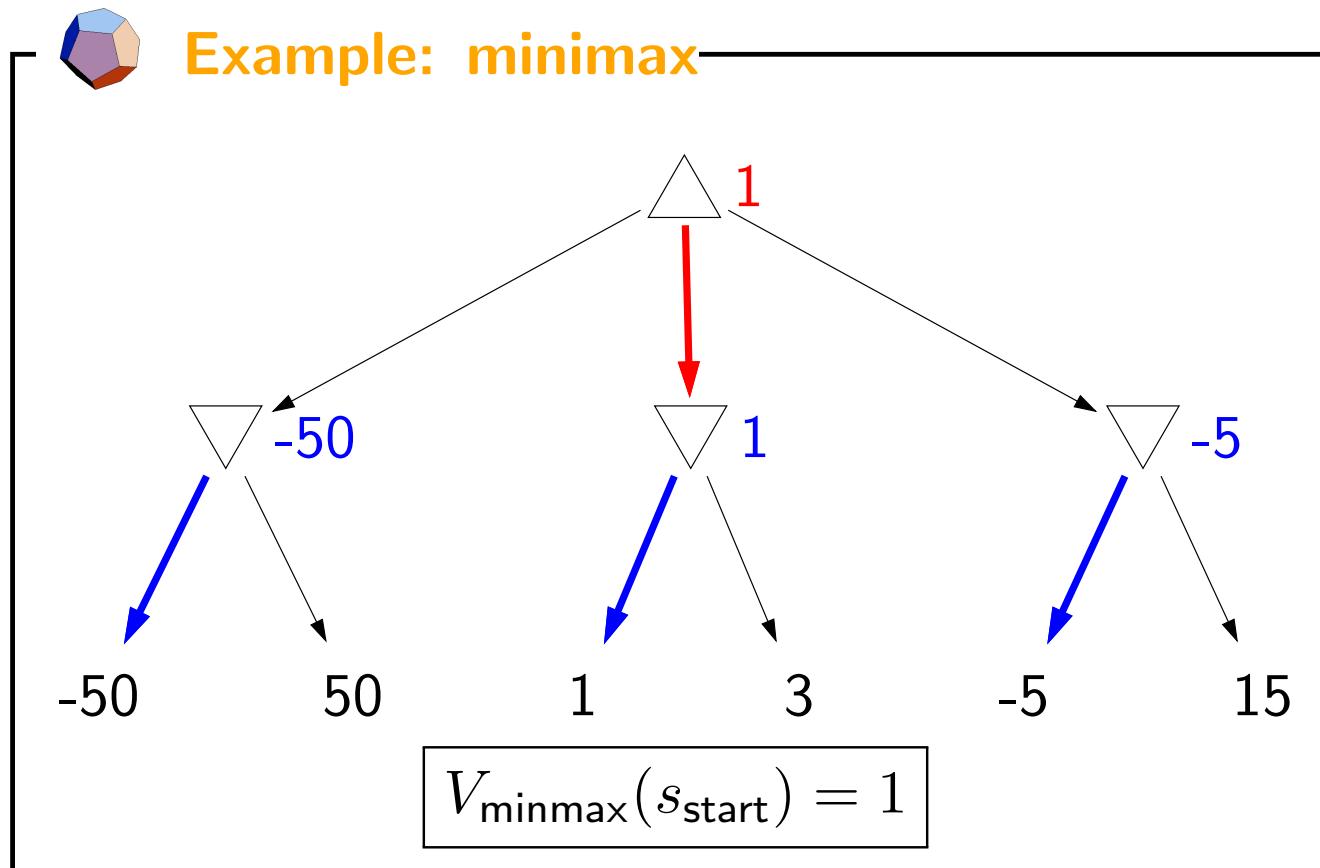
Alpha-beta pruning

Problem: don't know opponent's policy

Approach: assume the worst case



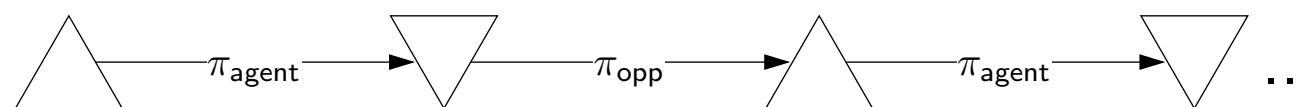
Minimax example



- If we could perform some mind-reading and discover the opponent's policy, then we could maximally exploit it. However, in practice, we don't know the opponent's policy. So our solution is to assume the **worst case**, that is, the opponent is doing everything to minimize the agent's utility.
- In the game tree, we use an upside-down triangle to represent **min nodes**, in which the player minimizes the value over possible actions.
- Note that the policy for the agent changes from choosing the rightmost action (expectimax) to the middle action. Why is this?

Minimax recurrence

No analogy in MDPs:



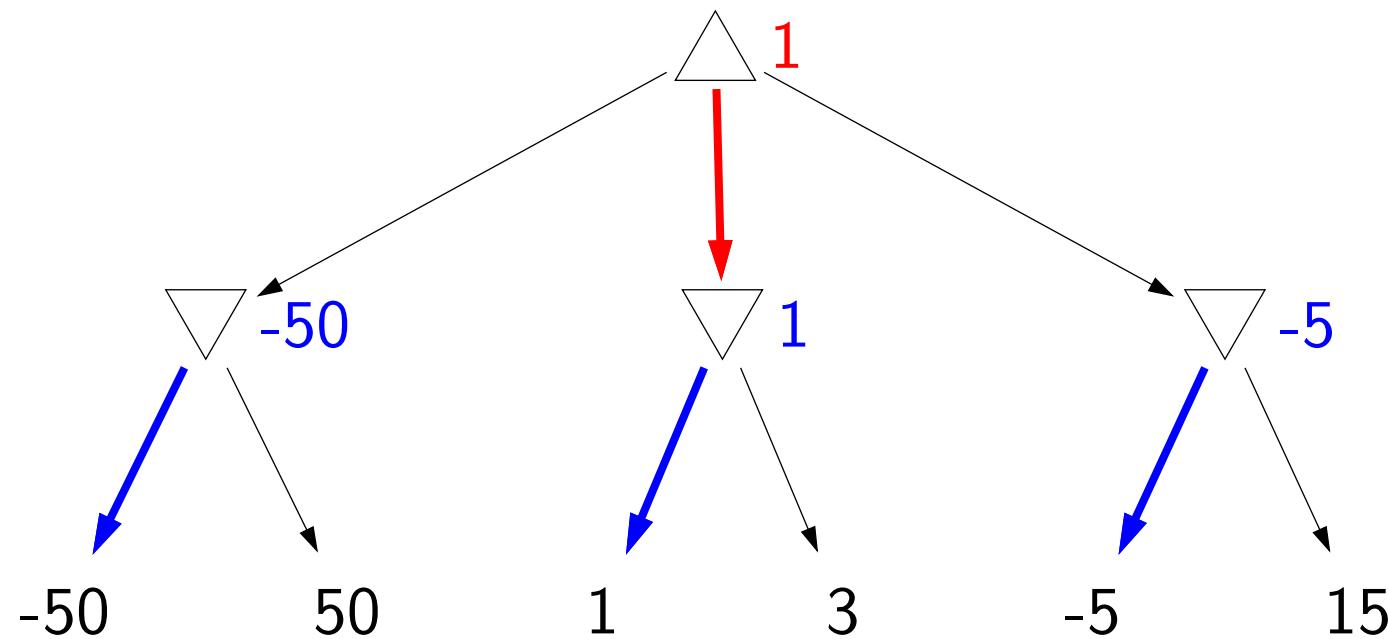
$$V_{\text{minmax}}(s) = \begin{cases} \text{Utility}(s) & \text{IsEnd}(s) \\ \max_{a \in \text{Actions}(s)} V_{\text{minmax}}(\text{Succ}(s, a)) & \text{Player}(s) = \text{agent} \\ \min_{a \in \text{Actions}(s)} V_{\text{minmax}}(\text{Succ}(s, a)) & \text{Player}(s) = \text{opp} \end{cases}$$

- The general recurrence for the minimax value is the same as expectimax, except that the expectation over the opponent's policy is replaced with a minimum over the opponent's possible actions. Note that the minimax value does not depend on any policies at all: it's just the agent and opponent playing optimally with respect to each other.

Extracting minimax policies

$$\pi_{\max}(s) = \arg \max_{a \in \text{Actions}(s)} V_{\min\max}(\text{Succ}(s, a))$$

$$\pi_{\min}(s) = \arg \min_{a \in \text{Actions}(s)} V_{\min\max}(\text{Succ}(s, a))$$



- Having computed the minimax value V_{minmax} , we can extract the minimax policies π_{max} and π_{min} by just taking the action that leads to the state with the maximum (or minimum) value.
- In general, having a value function tells you which states are good, from which it's easy to set the policy to move to those states (provided you know the transition structure, which we assume we know here).

The halving game



Problem: halving game

Start with a number N .

Players take turns either decrementing N or replacing it with $\lfloor \frac{N}{2} \rfloor$.

The player that is left with 0 wins.

[semi-live solution: `minimaxPolicy`]

Face off

Recurrences produces policies:

$$\begin{aligned} V_{\text{exptmax}} &\Rightarrow \pi_{\text{exptmax}(7)}, \pi_7 \text{ (some opponent)} \\ V_{\text{minmax}} &\Rightarrow \pi_{\text{max}}, \pi_{\text{min}} \end{aligned}$$

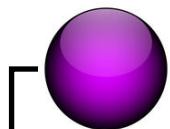
Play policies against each other:

	π_{min}	π_7
π_{max}	$V(\pi_{\text{max}}, \pi_{\text{min}})$	$V(\pi_{\text{max}}, \pi_7)$
$\pi_{\text{exptmax}(7)}$	$V(\pi_{\text{exptmax}(7)}, \pi_{\text{min}})$	$V(\pi_{\text{exptmax}(7)}, \pi_7)$

What's the relationship between these values?

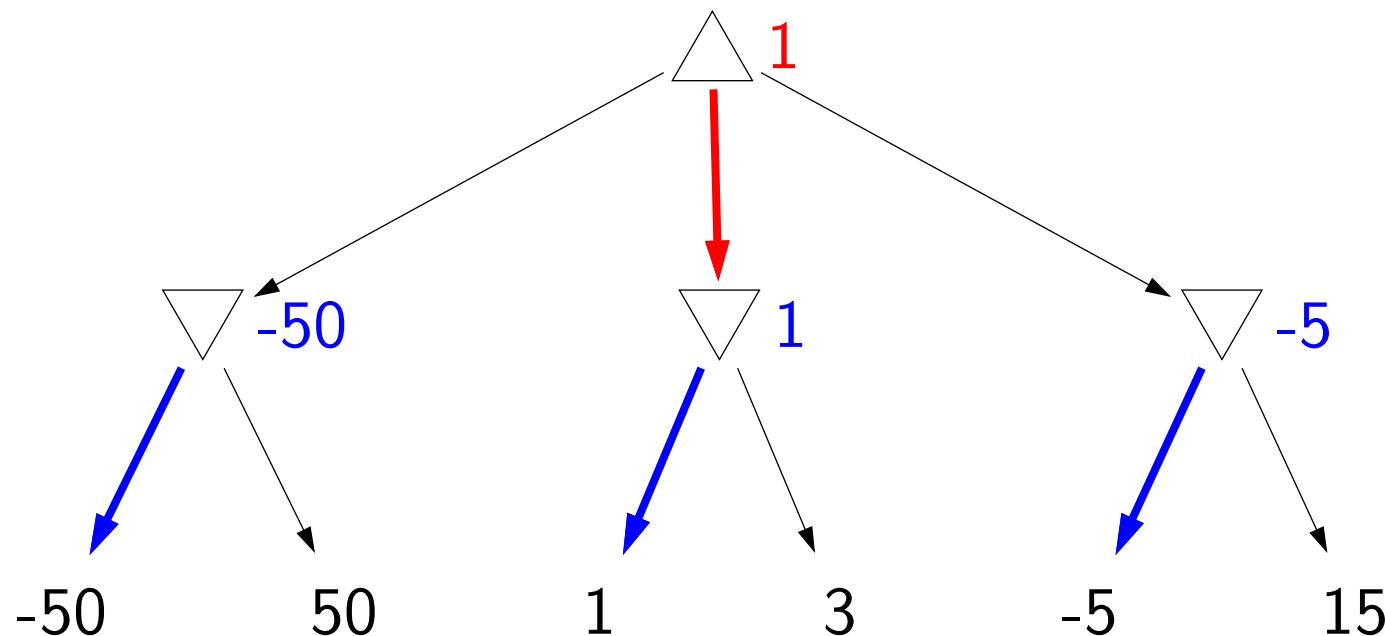
- So far, we have seen how expectimax and minimax recurrences produce policies.
- The expectimax recurrence computes the best policy $\pi_{\text{exptmax}(7)}$ against a fixed opponent policy (say π_7 for concreteness).
- The minimax recurrence computes the best policy π_{max} against the best opponent policy π_{min} .
- Now, whenever we take an agent policy π_{agent} and an opponent policy π_{opp} , we can play them against each other, which produces an expected utility via game evaluation, which we denote as $V(\pi_{\text{agent}}, \pi_{\text{opp}})$.
- How do the four game values of different combination of policies relate to each other?

Minimax property 1



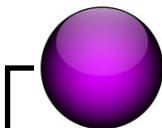
Proposition: best against minimax opponent

$$V(\pi_{\max}, \pi_{\min}) \geq V(\pi_{\text{agent}}, \pi_{\min}) \text{ for all } \pi_{\text{agent}}$$



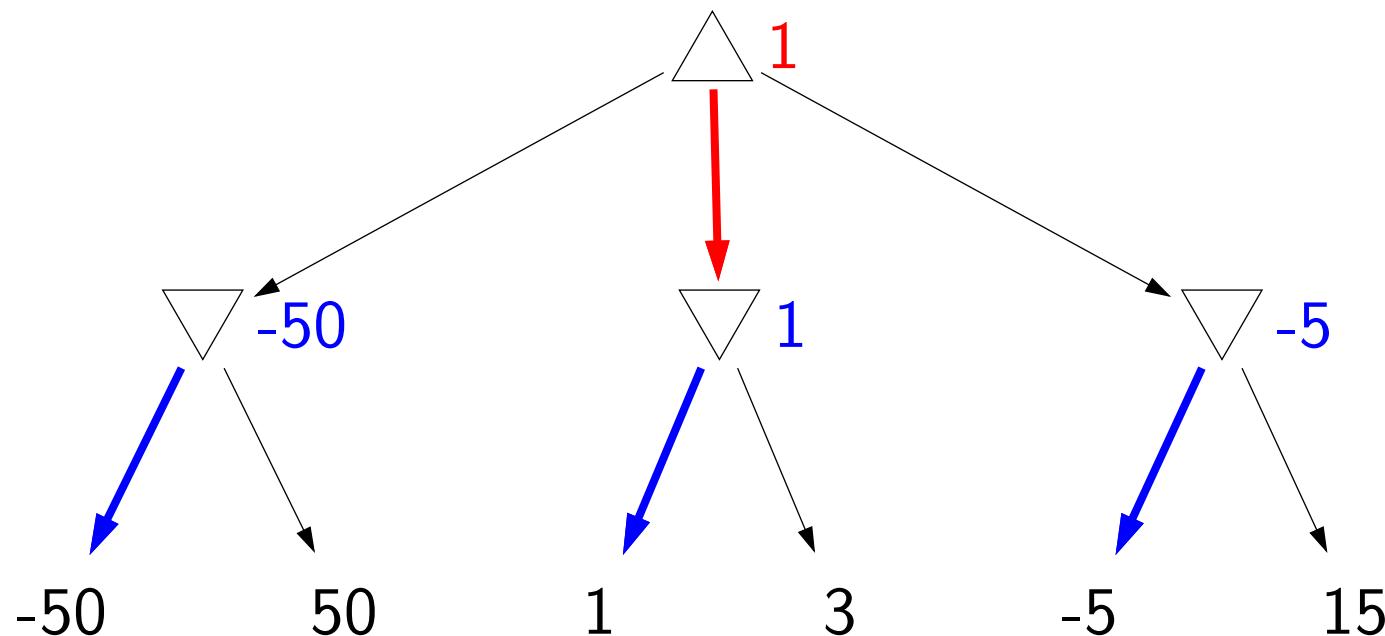
- Recall that π_{\max} and π_{\min} are the minimax agent and opponent policies, respectively. The first property is if the agent were to change her policy to any π_{agent} , then the agent would be no better off (and in general, worse off).
- From the example, it's intuitive that this property should hold. To prove it, we can perform induction starting from the leaves of the game tree, and show that the minimax value of each node is the highest over all possible policies.

Minimax property 2



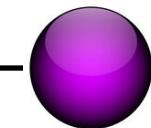
Proposition: lower bound against any opponent

$$V(\pi_{\max}, \pi_{\min}) \leq V(\pi_{\max}, \pi_{\text{opp}}) \text{ for all } \pi_{\text{opp}}$$



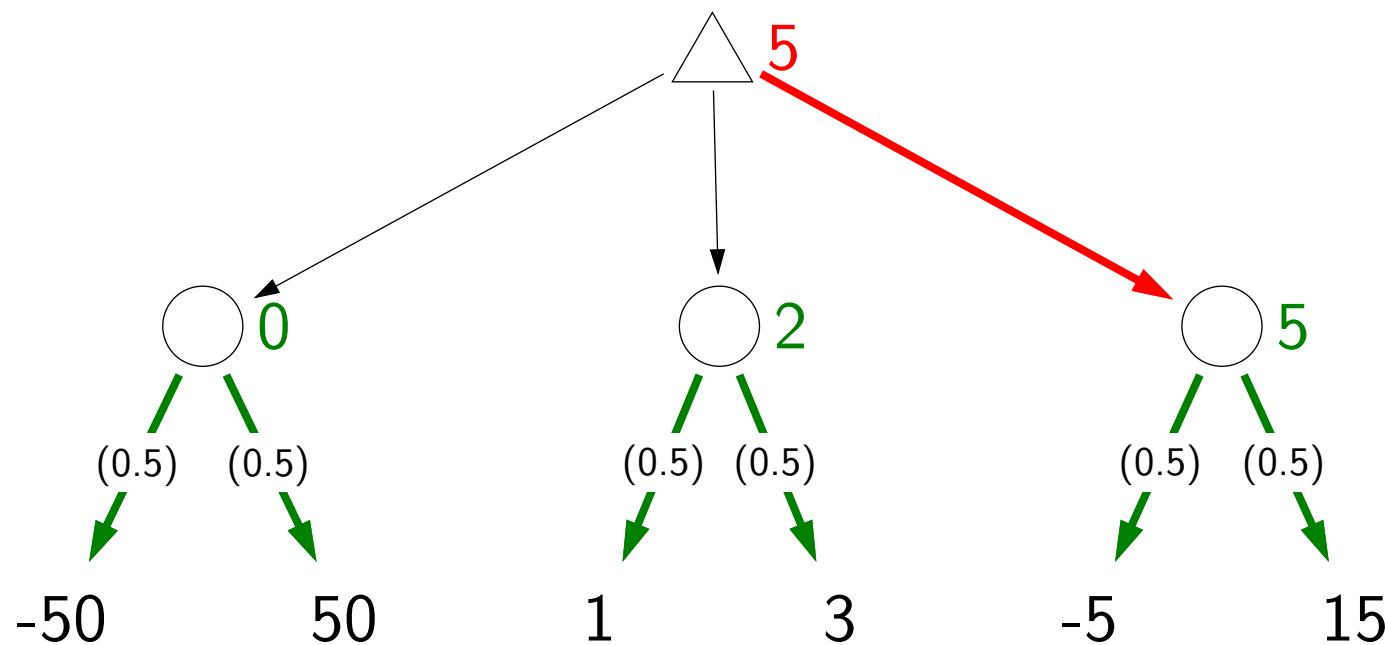
- The second property is the analogous statement for the opponent: if the opponent changes his policy from π_{\min} to π_{opp} , then he will be no better off (the value of the game can only increase).
- From the point of view of the agent, this can be interpreted as guarding against the worst case. In other words, if we get a minimax value of 1, that means no matter what the opponent does, the agent is guaranteed at least a value of 1. As a simple example, if the minimax value is $+\infty$, then the agent is guaranteed to win, provided it follows the minimax policy.

Minimax property 3



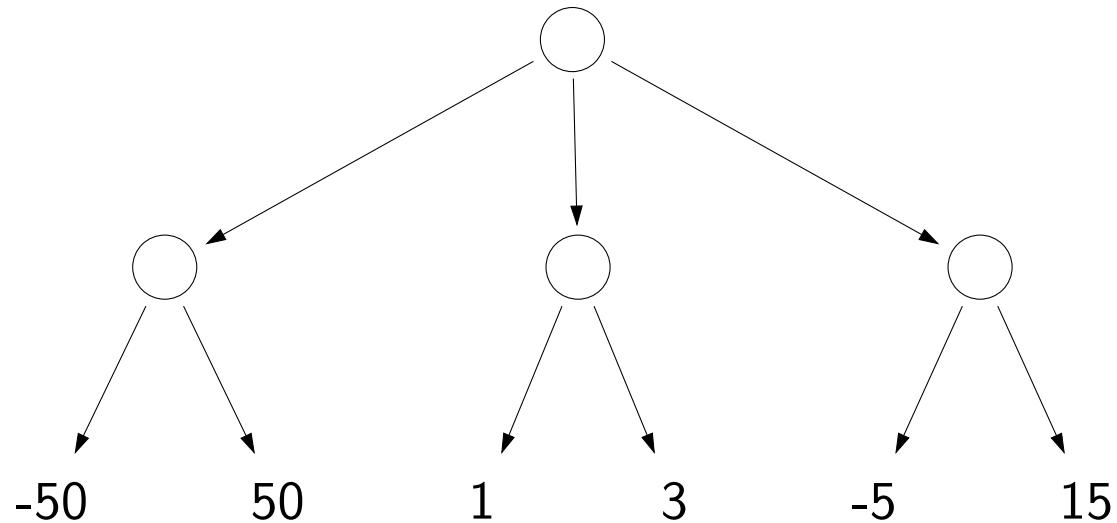
Proposition: not optimal if opponent is known

$$V(\pi_{\max}, \pi_7) \leq V(\pi_{\text{exptmax}(7)}, \pi_7) \text{ for opponent } \pi_7$$



- However, following the minimax policy might not be optimal for the agent if the opponent is known to be not playing the adversarial (minimax) policy.
- Consider the running example where the agent chooses A, B, or C and the opponent chooses a bin. Suppose the agent is playing π_{\max} , but the opponent is playing a stochastic policy π_7 corresponding to choosing an action uniformly at random.
- Then the game value here would be 2 (which is larger than the minimax value 1, as guaranteed by property 2). However, if we followed the expectimax $\pi_{\text{exptmax}(7)}$, then we would have gotten a value of 5, which is even higher.

Relationship between game values



$$\begin{array}{ccc} \pi_{\min} & & \pi_7 \\ V(\pi_{\max}, \pi_{\min}) & \leq & V(\pi_{\max}, \pi_7) \\ \pi_{\max} & & \\ 1 & & 2 \\ \vee & & \wedge \\ \end{array}$$

$$\begin{array}{cc} \pi_{\text{exptmax}(7)} & V(\pi_{\text{exptmax}(7)}, \pi_{\min}) \\ -5 & 5 \end{array}$$

- Putting the three properties together, we obtain a chain of inequalities that allows us to relate all four game values.
- We can also compute these values concretely for the running example.

A modified game



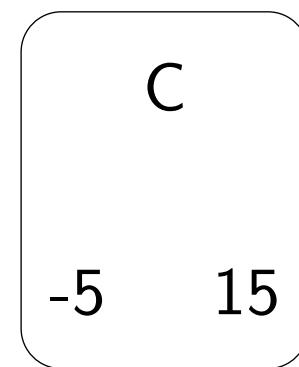
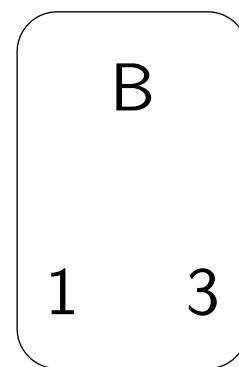
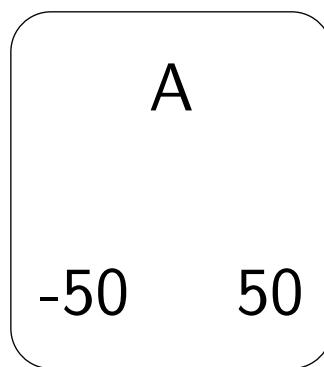
Example: game 2

You choose one of the three bins.

Flip a coin; if heads, then move one bin to the left (with wrap around).

I choose a number from that bin.

Your goal is to maximize the chosen number.



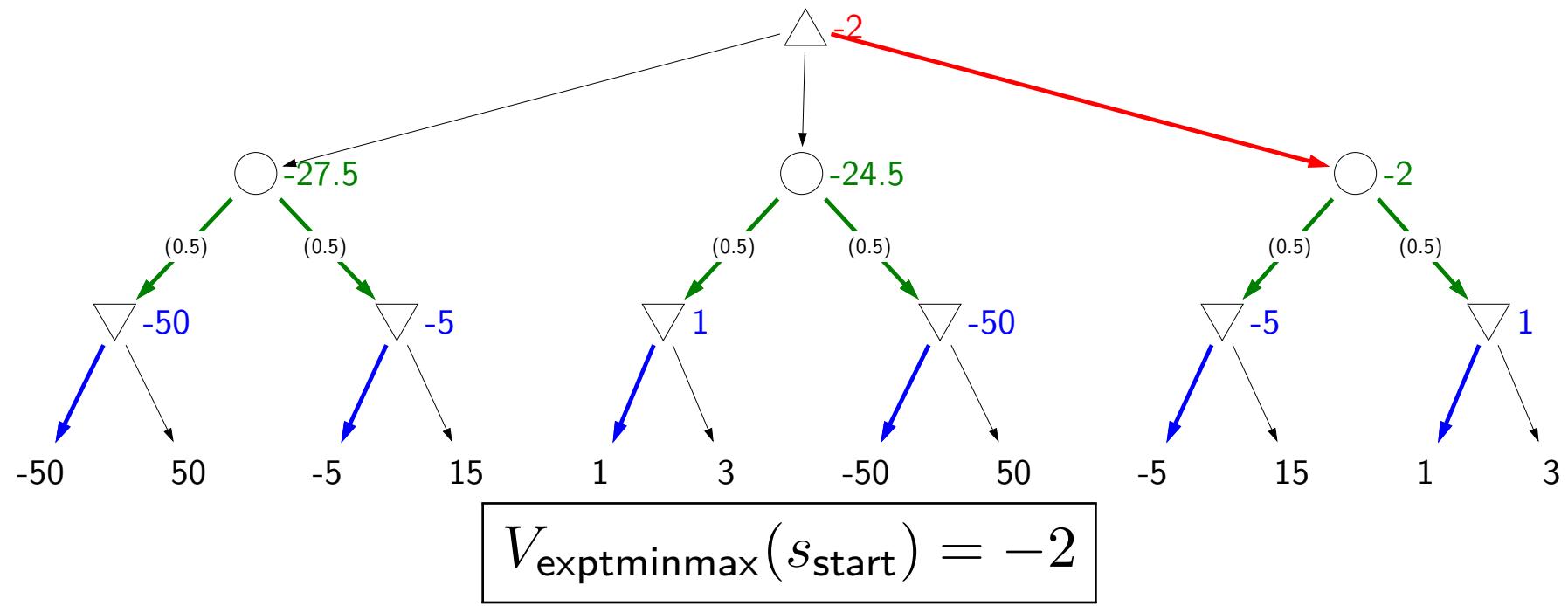
- Now let us consider games that have an element of chance that does not come from the agent or the opponent. Or in the simple modified game, the agent picks, a coin is flipped, and then the opponent picks.
- It turns out that handling games of chance is just a straightforward extension of the game framework that we have already.

Expectiminimax example



Example: expectiminimax

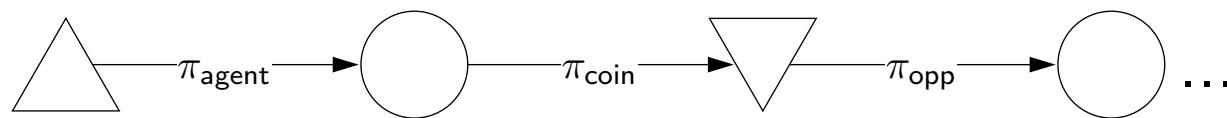
$$\pi_{\text{coin}}(s, a) = \frac{1}{2} \text{ for } a \in \{0, 1\}$$



- In the example, notice that the minimax optimal policy has shifted from the middle action to the rightmost action, which guards against the effects of the randomness. The agent really wants to avoid ending up on A, in which case the opponent could deliver a deadly -50 utility.

Expectiminimax recurrence

Players = {agent, opp, coin}



$$V_{\text{exptminmax}}(s) = \begin{cases} \text{Utility}(s) & \text{IsEnd}(s) \\ \max_{a \in \text{Actions}(s)} V_{\text{exptminmax}}(\text{Succ}(s, a)) & \text{Player}(s) = \text{agent} \\ \min_{a \in \text{Actions}(s)} V_{\text{exptminmax}}(\text{Succ}(s, a)) & \text{Player}(s) = \text{opp} \\ \sum_{a \in \text{Actions}(s)} \pi_{\text{coin}}(s, a) V_{\text{exptminmax}}(\text{Succ}(s, a)) & \text{Player}(s) = \text{coin} \end{cases}$$

- The resulting game is modeled using **expectiminimax**, where we introduce a third player (called coin), which always follows a known stochastic policy. We are using the term *coin* as just a metaphor for any sort of natural randomness.
- To handle coin, we simply add a line into our recurrence that sums over actions when it's coin's turn.



Summary so far

Primitives: **max** nodes, **chance** nodes, **min** nodes

Composition: alternate nodes according to model of game

Value function $V_{\dots}(s)$: recurrence for expected utility

Scenarios to think about:

- What if you are playing against multiple opponents?
- What if you and your partner have to take turns (table tennis)?
- Some actions allow you to take an extra turn?

- In summary, so far, we've shown how to model a number of games using game trees, where each node of the game tree is either a max, chance, or min node depending on whose turn it is at that node and what we believe about that player's policy.
- Using these primitives, one can model more complex turn-taking games involving multiple players with heterogeneous strategies and where the turn-taking doesn't have to strictly alternate. The only restriction is that there are two parties: one that seeks to maximize utility and the other that seeks to minimize utility, along with other players who have known fixed policies (like coin).



Roadmap

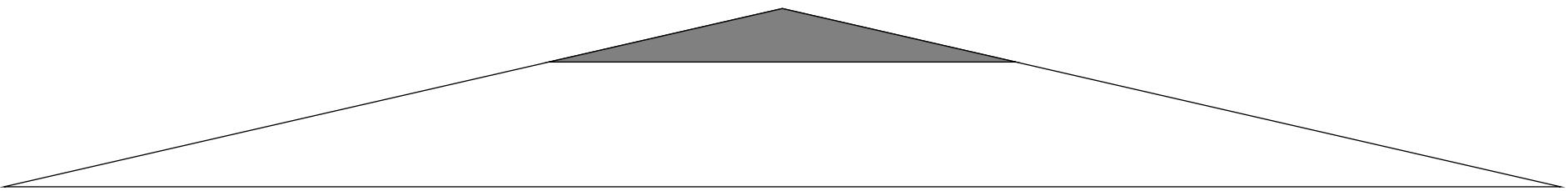
Games, expectimax

Minimax, expectiminimax

Evaluation functions

Alpha-beta pruning

Depth-limited search



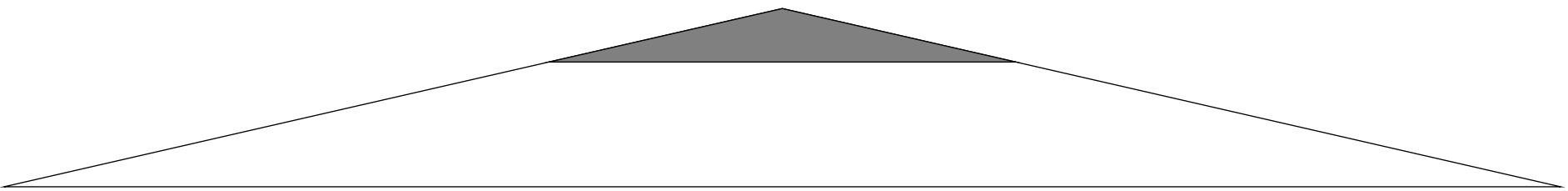
Limited depth tree search (stop at maximum depth d_{\max}):

$$V_{\min\max}(s, \textcolor{red}{d}) = \begin{cases} \text{Utility}(s) & \text{IsEnd}(s) \\ \text{Eval}(s) & d = 0 \\ \max_{a \in \text{Actions}(s)} V_{\min\max}(\text{Succ}(s, a), \textcolor{red}{d}) & \text{Player}(s) = \text{agent} \\ \min_{a \in \text{Actions}(s)} V_{\min\max}(\text{Succ}(s, a), \textcolor{red}{d - 1}) & \text{Player}(s) = \text{opp} \end{cases}$$

Use: at state s , call $V_{\min\max}(s, d_{\max})$

Convention: decrement depth at last player's turn

Evaluation functions



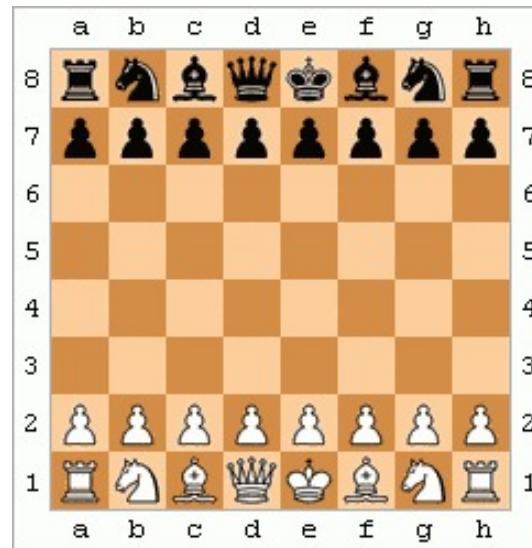
Definition: Evaluation function

An evaluation function $\text{Eval}(s)$ is a (possibly very weak) estimate of the value $V_{\text{minmax}}(s)$.

Analogy: FutureCost(s) in search problems

- The first idea on how to speed up minimax is to search only the tip of the game tree, that is down to depth d_{\max} , which is much smaller than the total depth of the tree D (for example, d_{\max} might be 4 and $D = 50$).
- We modify our minimax recurrence from before by adding an argument d , which is the maximum depth that we are willing to descend from state s . If $d = 0$, then we don't do any more search, but fall back to an **evaluation function** $\text{Eval}(s)$, which is supposed to approximate the value of $V_{\min\max}(s)$ (just like the heuristic $h(s)$ approximated $\text{FutureCost}(s)$ in A* search).
- If $d > 0$, we recurse, decrementing the allowable depth by one at only min nodes, not the max nodes. This is because we are keeping track of the depth rather than the number of plies.

Evaluation functions



Example: chess

$\text{Eval}(s) = \text{material} + \text{mobility} + \text{king-safety} + \text{center-control}$

$$\begin{aligned}\text{material} &= 10^{100}(K - K') + 9(Q - Q') + 5(R - R') + \\ &\quad 3(B - B' + N - N') + 1(P - P')\end{aligned}$$

$$\text{mobility} = 0.1(\text{num-legal-moves} - \text{num-legal-moves}')$$

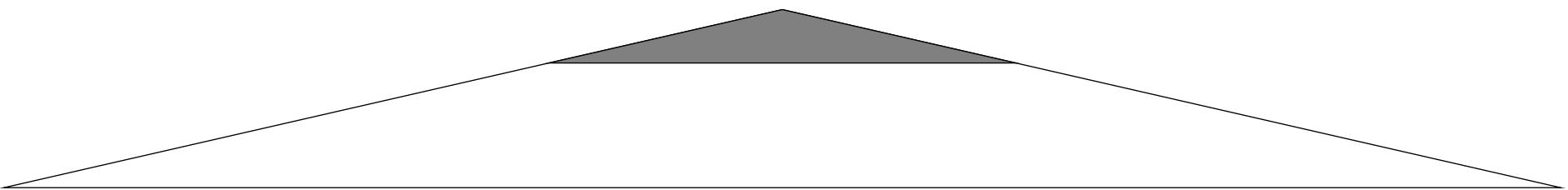
...

- Now what is this mysterious evaluation function $\text{Eval}(s)$ that serves as a substitute for the horrendously hard $V_{\min\max}$ that we can't compute?
- Just as in A*, there is no free lunch, and we have to use domain knowledge about the game. Let's take chess for example. While we don't know who's going to win, there are some features of the game that are likely indicators. For example, having more pieces is good (material), being able to move them is good (mobility), keeping the king safe is good, and being able to control the center of the board is also good. We can then construct an evaluation function which is a weighted combination of the different properties.
- For example, $K - K'$ is the difference in the number of kings that the agent has over the number that the opponent has (losing kings is really bad since you lose then), $Q - Q'$ is the difference in queens, $R - R'$ is the difference in rooks, $B - B'$ is the difference in bishops, $N - N'$ is the difference in knights, and $P - P'$ is the difference in pawns.



Summary: evaluation functions

Depth-limited exhaustive search: $O(b^{2d})$ time



- $\text{Eval}(s)$ attempts to estimate $V_{\text{minmax}}(s)$ using domain knowledge
- No guarantees (unlike A*) on the error from approximation

- To summarize, this section has been about how to make naive exhaustive search over the game tree to compute the minimax value of a game faster.
- The methods so far have been focused on taking shortcuts: only searching up to depth d and relying on an **evaluation function**, and using a cheaper mechanism for estimating the value at a node rather than search its entire subtree.



Roadmap

Games, expectimax

Minimax, expectiminimax

Evaluation functions

Alpha-beta pruning

Pruning principle

Choose A or B with maximum value:

A: [3, **5**]

B: [**5**, 100]



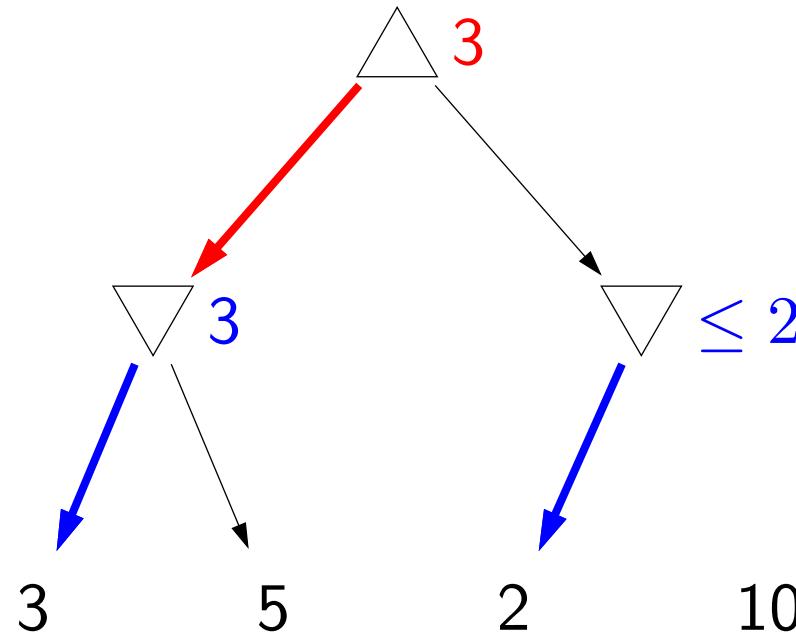
Key idea: branch and bound

Maintain lower and upper bounds on values.

If intervals don't overlap non-trivially, then can choose optimally without further work.

- We continue on our quest to make minimax run faster based on **pruning**. Unlike evaluation functions, these are general purpose and have theoretical guarantees.
- The core idea of pruning is based on the branch and bound principle. As we are searching (branching), we keep lower and upper bounds on each value we're trying to compute. If we ever get into a situation where we are choosing between two options A and B whose intervals don't overlap or just meet at a single point (in other words, they do not **overlap non-trivially**), then we can choose the interval containing larger values (B in the example). The significance of this observation is that we don't have to do extra work to figure out the precise value of A.

Pruning game trees



Once see 2, we know that value of right node must be ≤ 2

Root computes $\max(3, \leq 2) = 3$

Since branch doesn't affect root value, can safely prune

- In the context of minimax search, we note that the root node is a max over its children.
- Once we see the left child, we know that the root value must be at least 3.
- Once we get the 2 on the right, we know the right child has to be at most 2.
- Since those two intervals are non-overlapping, we can prune the rest of the right subtree and not explore it.

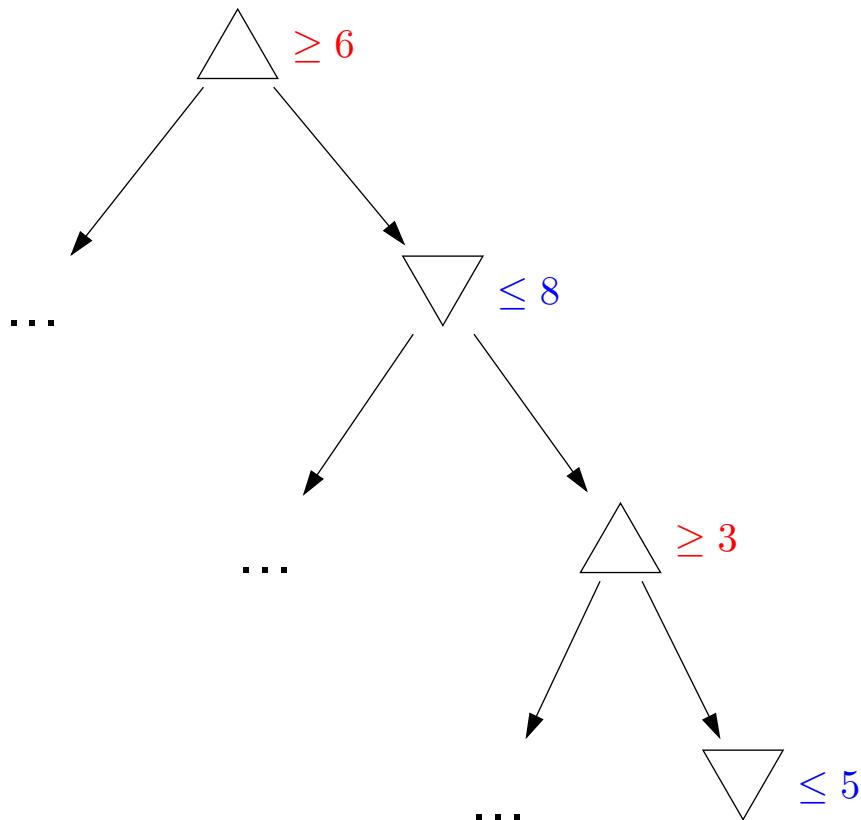
Alpha-beta pruning



Key idea: optimal path

The optimal path is path that minimax policies take.

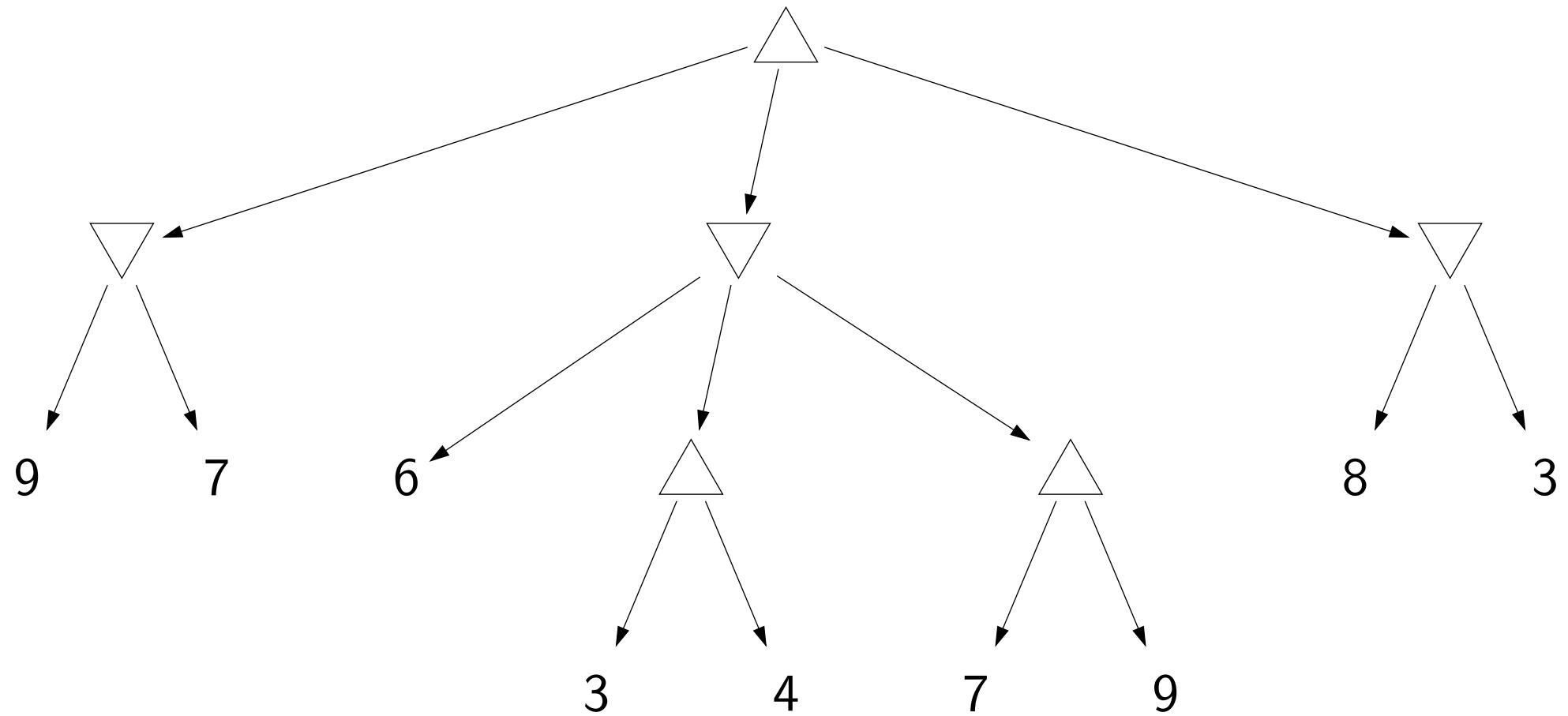
Values of all nodes on path are the same.



- a_s : lower bound on value of max node s
- b_s : upper bound on value of min node s
- Prune a node if its interval doesn't have non-trivial overlap with every ancestor (store $\alpha_s = \max_{s' \preceq s} a_s$ and $\beta_s = \min_{s' \preceq s} b_s$)

- In general, let's think about the minimax values in the game tree. The value of a node is equal to the utility of at least one of its leaf nodes (because all the values are just propagated from the leaves with min and max applied to them). Call the first path (ordering by children left-to-right) that leads to the first such leaf node the **optimal path**. An important observation is that the values of all nodes on the optimal path are the same (equal to the minimax value of the root).
- Since we are interested in computing the value of the root node, if we can certify that a node is not on the optimal path, then we can prune it and its subtree.
- To do this, during the depth-first exhaustive search of the game tree, we think about maintaining a lower bound ($\geq a_s$) for all the max nodes s and an upper bound ($\leq b_s$) for all the min nodes s .
- If the interval of the current node does not non-trivially overlap the interval of every one of its ancestors, then we can prune the current node. In the example, we've determined the root's node must be ≥ 6 . Once we get to the node on at ply 4 and determine that node is ≤ 5 , we can prune the rest of its children since it is impossible that this node will be on the optimal path (≤ 5 and ≥ 6 are incompatible). Remember that all the nodes on the optimal path have the same value.
- Implementation note: for each max node s , rather than keeping a_s , we keep α_s , which is the maximum value of $a_{s'}$ over s and all its max node ancestors. Similarly, for each min node s , rather than keeping b_s , we keep β_s , which is the minimum value of $b_{s'}$ over s and all its min node ancestors. That way, at any given node, we can check interval overlap in constant time regardless of how deep we are in the tree.

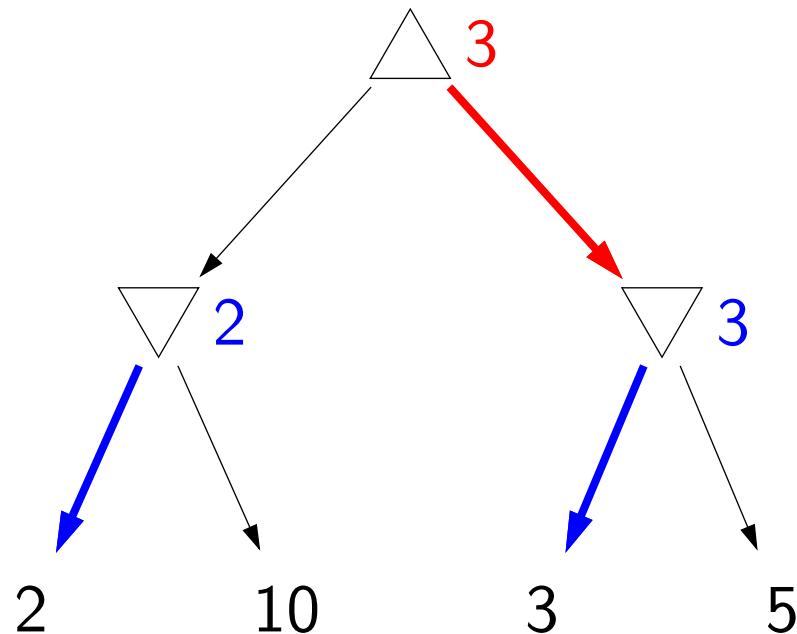
Alpha-beta pruning example



Move ordering

Pruning depends on order of actions.

Can't prune the 5 node:



- We have so far shown that alpha-beta pruning correctly computes the minimax value at the root, and seems to save some work by pruning subtrees. But how much of a savings do we get?
- The answer is that it depends on the order in which we explore the children. This simple example shows that with one ordering, we can prune the final leaf, but in the second, we can't.

Move ordering

Which ordering to choose?

- Worst ordering: $O(b^{2 \cdot d})$ time
- Best ordering: $O(b^{2 \cdot 0.5d})$ time
- Random ordering: $O(b^{2 \cdot 0.75d})$ time

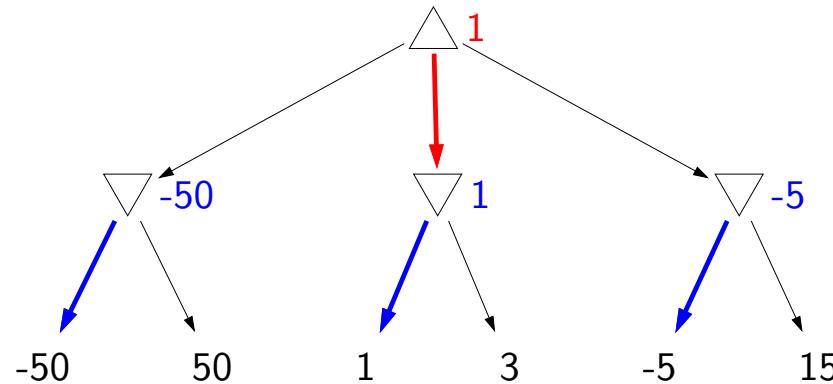
In practice, can use evaluation function $\text{Eval}(s)$:

- Max nodes: order successors by decreasing $\text{Eval}(s)$
- Min nodes: order successors by increasing $\text{Eval}(s)$

- In the worst case, we don't get any savings.
- If we use the best possible ordering, then we save half the exponent, which is *significant*. This means that if we could search to depth 10 before, we can now search to depth 20, which is truly remarkable given that the time increases exponentially with the depth.
- In practice, of course we don't know the best ordering. But interestingly, if we just use a random ordering, that allows us to search 33 percent deeper.
- We could also use a heuristic ordering based on a simple evaluation function. Intuitively, we want to search children that are going to give us the largest lower bound for max nodes and the smallest upper bound for min nodes.



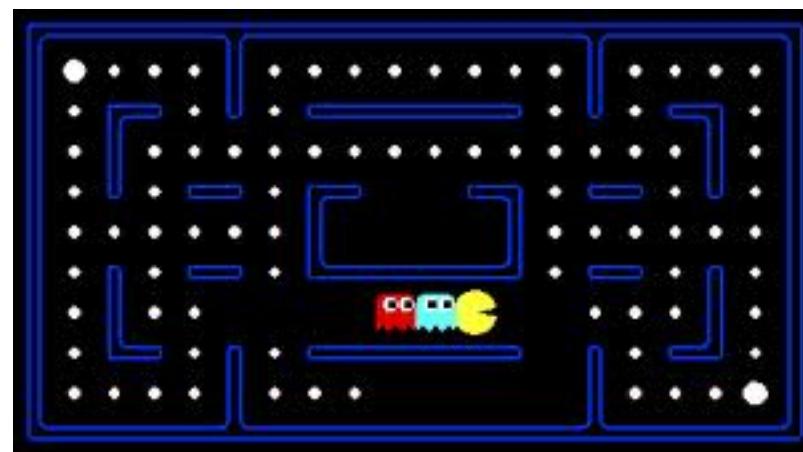
Summary



- Game trees: model opponents, randomness
- Minimax: find optimal policy against an adversary
- Evaluation functions: domain-specific, approximate
- Alpha-beta pruning: domain-general, exact



Games II





Question

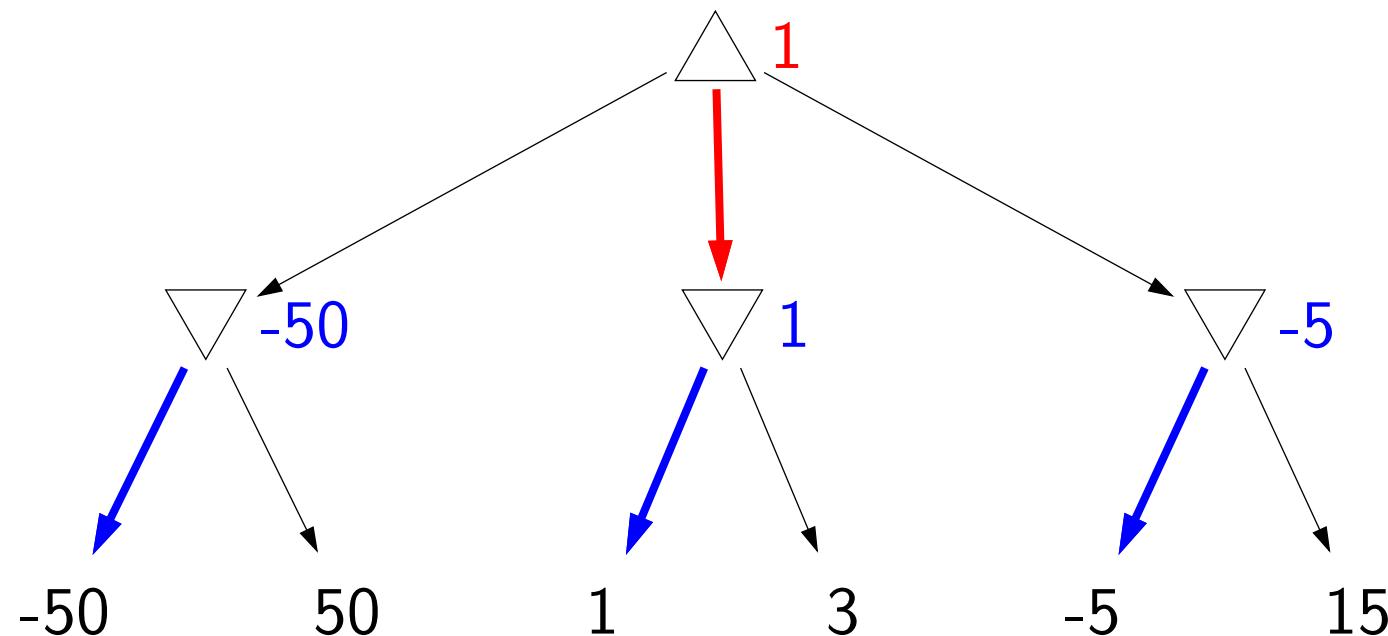
For a simultaneous two-player zero-sum game (like rock-paper-scissors), can you still be optimal if you reveal your strategy?

yes

no

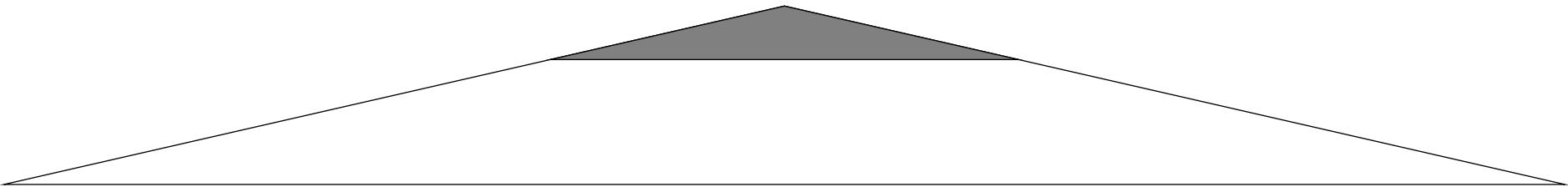
Review: minimax

agent (max) versus opponent (min)



- Recall that the central object of study is the game tree. Game play starts at the root (starting state) and descends to a leaf (end state), where at each node s (state), the player whose turn it is ($\text{Player}(s)$) chooses an action $a \in \text{Actions}(s)$, which leads to one of the children $\text{Succ}(s, a)$.
- The **minimax principle** provides one way for the agent (your computer program) to compute a pair of minimax policies for both the agent and the opponent ($\pi_{\text{agent}}^*, \pi_{\text{opp}}^*$).
- For each node s , we have the minimax value of the game $V_{\text{minmax}}(s)$, representing the expected utility if both the agent and the opponent play optimally. Each node where it's the agent's turn is a max node (right-side up triangle), and its value is the maximum over the children's values. Each node where it's the opponent's turn is a min node (upside-down triangle), and its value is the minimum over the children's values.
- Important properties of the minimax policies: The agent can only decrease the game value (do worse) by changing his/her strategy, and the opponent can only increase the game value (do worse) by changing his/her strategy.

Review: depth-limited search



$$V_{\text{minmax}}(s, d) = \begin{cases} \text{Utility}(s) & \text{IsEnd}(s) \\ \text{Eval}(s) & d = 0 \\ \max_{a \in \text{Actions}(s)} V_{\text{minmax}}(\text{Succ}(s, a), d) & \text{Player}(s) = \text{agent} \\ \min_{a \in \text{Actions}(s)} V_{\text{minmax}}(\text{Succ}(s, a), d - 1) & \text{Player}(s) = \text{opp} \end{cases}$$

Use: at state s , choose action resulting in $V_{\text{minmax}}(s, d_{\text{max}})$

- In order to approximately compute the minimax value, we used a **depth-limited search**, where we compute $V_{\text{minmax}}(s, d_{\text{max}})$, the approximate value of s if we are only allowed to search to at most depth d_{max} .
- Each time we hit $d = 0$, we invoke an evaluation function $\text{Eval}(s)$, which provides a fast reflex way to assess the value of the game at state s .

Evaluation function

Old: hand-crafted



Example: chess

$\text{Eval}(s) = \text{material} + \text{mobility} + \text{king-safety} + \text{center-control}$

$$\begin{aligned}\text{material} &= 10^{100}(K - K') + 9(Q - Q') + 5(R - R') + \\ &\quad 3(B - B' + N - N') + 1(P - P')\end{aligned}$$

$$\text{mobility} = 0.1(\text{num-legal-moves} - \text{num-legal-moves}')$$

...

New: learn from data

$$\text{Eval}(s) = V(s; \mathbf{w})$$

- Having a good evaluation function is one of the most important components of game playing. So far we've shown how one can manually specify the evaluation function by hand. However, this can be quite tedious, and moreover, how does one figure out to weigh the different factors? In this lecture, we will consider a method for learning this evaluation function automatically from data.
- The three ingredients in any machine learning approach are to determine the (i) model family (in this case, what is $V(s; \mathbf{w})$?), (ii) where the data comes from, and (iii) the actual learning algorithm. We will go through each of these in turn.



Roadmap

TD learning

Simultaneous games

Non-zero-sum games

State-of-the-art

Model for evaluation functions

Linear:

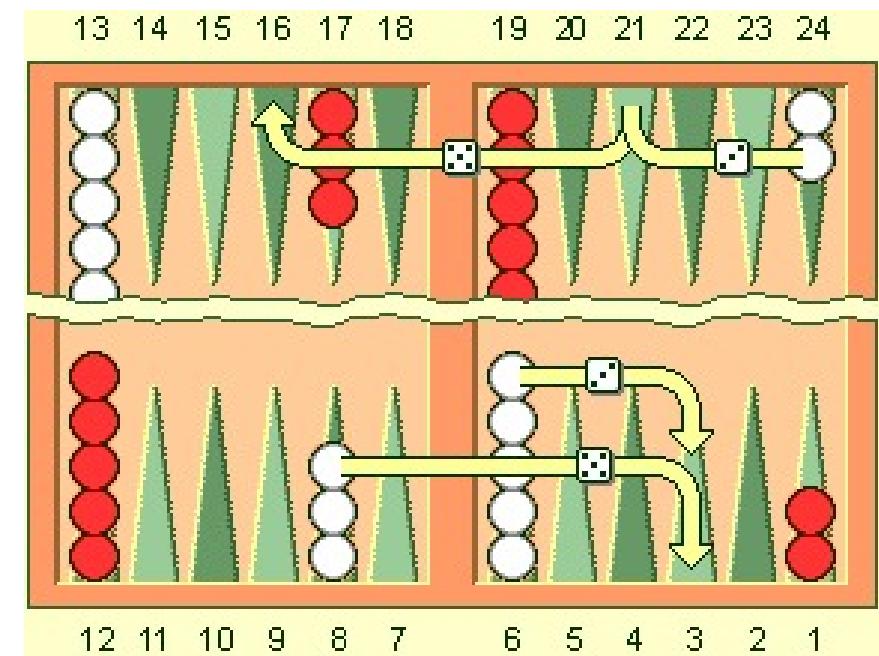
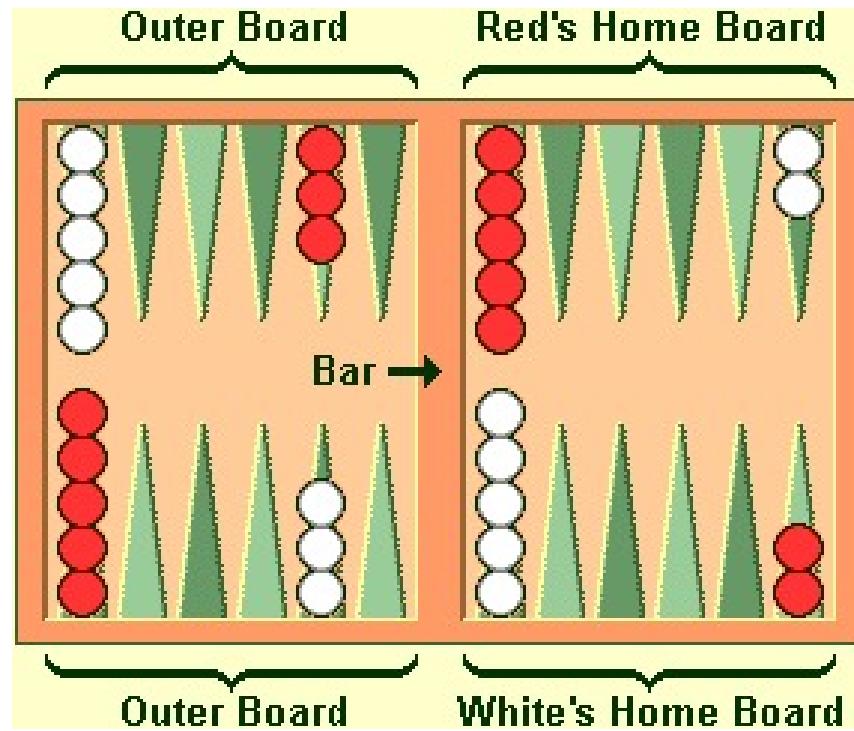
$$V(s; \mathbf{w}) = \mathbf{w} \cdot \phi(s)$$

Neural network:

$$V(s; \mathbf{w}, \mathbf{v}_{1:k}) = \sum_{j=1}^k w_j \sigma(\mathbf{v}_j \cdot \phi(s))$$

- When we looked at Q-learning, we considered linear evaluation functions (remember, linear in the weights w). This is the simplest case, but it might not be suitable in some cases.
- But the evaluation function can really be any parametrized function. For example, the original TD-Gammon program used a neural network, which allows us to represent more expressive functions that capture the non-linear interactions between different features.
- Any model that you could use for regression in supervised learning you could also use here.

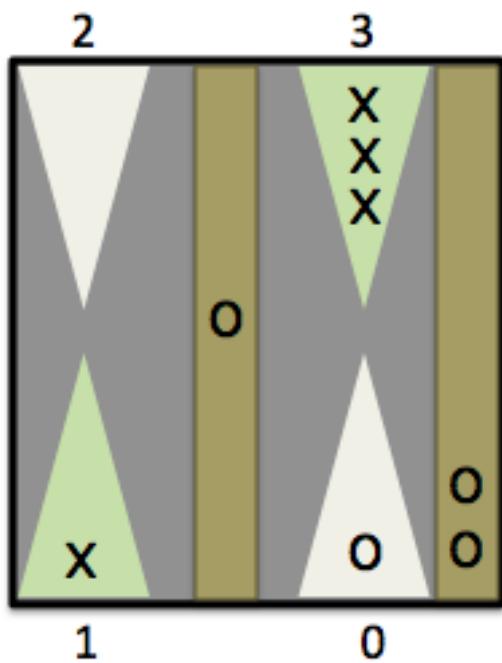
Example: Backgammon



- As an example, let's consider the classic game of backgammon. Backgammon is a two-player game of strategy and chance in which the objective is to be the first to remove all your pieces from the board.
- The simplified version is that on your turn, you roll two dice, and choose two of your pieces to move forward that many positions.
- You cannot land on a position containing more than one opponent piece. If you land on exactly one opponent piece, then that piece goes on the bar and has start over from the beginning. (See the Wikipedia article for the full rules.).

Features for Backgammon

state s



Features $\phi(s)$:

$[(\# \text{ o in column 0}) = 1] : 1$

$[(\# \text{ o on bar})] : 1$

$[(\text{fraction o removed})] : \frac{1}{2}$

$[(\# \text{ x in column 1}) = 1] : 1$

$[(\# \text{ x in column 3}) = 3] : 1$

$[(\text{is it o's turn})] : 1$

- As an example, we can define the following features for Backgammon, which are inspired by the ones used by TD-Gammon.
- Note that the features are pretty generic; there is no explicit modeling of strategies such as trying to avoid having singleton pieces (because it could get clobbered) or preferences for how the pieces are distributed across the board.
- On the other hand, the features are mostly **indicator** features, which is a common trick to allow for more expressive functions using the machinery of linear regression. For example, instead of having one feature whose value is the number of pieces in a particular column, we can have multiple features for indicating whether the number of pieces is over some threshold.

Generating data

Generate using policies based on current $V(s; \mathbf{w})$:

$$\pi_{\text{agent}}(s; \mathbf{w}) = \arg \max_{a \in \text{Actions}(s)} V(\text{Succ}(s, a); \mathbf{w})$$

$$\pi_{\text{opp}}(s; \mathbf{w}) = \arg \min_{a \in \text{Actions}(s)} V(\text{Succ}(s, a); \mathbf{w})$$

Note: don't need to randomize (ϵ -greedy) because game is already stochastic (backgammon has dice) and there's function approximation

Generate episode:

$$s_0; a_1, r_1, s_1; a_2, r_2, s_2; a_3, r_3, s_3; \dots; a_n, r_n, s_n$$

- The second ingredient of doing learning is generating the data. As in reinforcement learning, we will generate a sequence of states, actions, and rewards by simulation — that is, by playing the game.
- In order to play the game, we need two exploration policies: one for the agent, one for the opponent. The policy of the dice is fixed to be uniform over $\{1, \dots, 6\}$ as expected.
- A natural policy to use is one that uses our current estimate of the value $V(s; \mathbf{w})$. Specifically, the agent's policy will consider all possible actions from a state, use the value function to evaluate how good each of the successor states are, and then choose the action leading to the highest value. Generically, we would include $\text{Reward}(s, a, \text{Succ}(s, a))$, but in games, all the reward is at the end, so $r_t = 0$ for $t < n$ and $r_n = \text{Utility}(s_n)$. Symmetrically, the opponent's policy will choose the action that leads to the lowest possible value.
- Given this choice of π_{agent} and π_{opp} , we generate the actions $a_t = \pi_{\text{Player}(s_{t-1})}(s_{t-1})$, successors $s_t = \text{Succ}(s_{t-1}, a_t)$, and rewards $r_t = \text{Reward}(s_{t-1}, a_t, s_t)$.
- In reinforcement learning, we saw that using an exploration policy based on just the current value function is a bad idea, because we can get stuck exploiting local optima and not exploring. In the specific case of Backgammon, using deterministic exploration policies for the agent and opponent turns out to be fine, because the randomness from the dice naturally provides exploration.

Learning algorithm

Episode:

$$s_0; a_1, r_1, s_1; a_2, r_2, s_2, a_3, r_3, s_3; \dots, a_n, r_n, s_n$$

A small piece of experience:

$$(s, a, r, s')$$

Prediction:

$$V(s; \mathbf{w})$$

Target:

$$r + \gamma V(s'; \mathbf{w})$$

- With a model family $V(s; \mathbf{w})$ and data $s_0, a_1, r_1, s_1, \dots$ in hand, let's turn to the learning algorithm.
- A general principle in learning is to figure out the **prediction** and the **target**. The prediction is just the value of the current function at the current state s , and the target uses the data by looking at the immediate reward r plus the value of the function applied to the successor state s' (discounted by γ). This is analogous to the SARSA update for Q-values, where our target actually depends on a one-step lookahead prediction.

General framework

Objective function:

$$\frac{1}{2}(\text{prediction}(\mathbf{w}) - \text{target})^2$$

Gradient:

$$(\text{prediction}(\mathbf{w}) - \text{target})\nabla_{\mathbf{w}}\text{prediction}(\mathbf{w})$$

Update:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \underbrace{(\text{prediction}(\mathbf{w}) - \text{target})\nabla_{\mathbf{w}}\text{prediction}(\mathbf{w})}_{\text{gradient}}$$

- Having identified a prediction and target, the next step is to figure out how to update the weights. The general strategy is to set up an objective function that encourages the prediction and target to be close (by penalizing their squared distance).
- Then we just take the gradient with respect to the weights w .
- Note that even though technically the target also depends on the weights w , we treat this as constant for this derivation. The resulting learning algorithm by no means finds the global minimum of this objective function. We are simply using the objective function to motivate the update rule.

Temporal difference (TD) learning



Algorithm: TD learning

On each (s, a, r, s') :

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \underbrace{V(s; \mathbf{w})}_{\text{prediction}} - \underbrace{(r + \gamma V(s'; \mathbf{w}))}_{\text{target}} \nabla_{\mathbf{w}} V(s; \mathbf{w})$$

For linear functions:

$$V(s; \mathbf{w}) = \mathbf{w} \cdot \phi(s)$$

$$\nabla_{\mathbf{w}} V(s; \mathbf{w}) = \phi(s)$$

- Plugging in the prediction and the target in our setting yields the TD learning algorithm. For linear functions, recall that the gradient is just the feature vector.

Example of TD learning

Step size $\eta = 0.5$, discount $\gamma = 1$, reward is end utility



Example: TD learning

S1	r:0	S4	r:0	S8	r:1	S9
$\phi: \binom{0}{1}$		$\phi: \binom{1}{0}$		$\phi: \binom{1}{2}$		$\phi: \binom{1}{0}$
$w: \binom{0}{0}$	p:0	$w: \binom{0}{0}$	p:0	$w: \binom{0}{0}$	p:0	$w: \binom{0.5}{1}$
t:0		t:0			t:1	
p-t:0		p-t:0			p-t:-1	
S1	r:0	S2	r:0	S6	r:0	S10
$\phi: \binom{0}{1}$		$\phi: \binom{1}{0}$		$\phi: \binom{0}{0}$		$\phi: \binom{1}{0}$
$w: \binom{0.5}{1}$	p:1	$w: \binom{0.5}{0.75}$	p:0.5	$w: \binom{0.25}{0.75}$	p:0	$w: \binom{0.25}{0.75}$
t:0.5		t:0			t:0.25	
p-t:0.5		p-t:0.5			p-t:-0.25	

- Here's an example of TD learning in action. We have two episodes: [S1, 0, S4, 0, S8, 1, S9] and [S1, 0, S2, 0, S6, 0, S10].
- In games, all the reward comes at the end and the discount is 1. We have omitted the action because TD learning doesn't depend on the action.
- Under each state, we have written its feature vector, and the weight vector before updating on that state. Note that no updates are made until the first non-zero reward. Our prediction is 0, and the target is $1 + 0$, so we subtract $-0.5[1, 2]$ from the weights to get $[0.5, 1]$.
- In the second row, we have our second episode, and now notice that even though all the rewards are zero, we are still making updates to the weight vectors since the prediction and targets computed based on adjacent states are different.

Comparison



Algorithm: TD learning

On each (s, a, r, s') :

$$\mathbf{w} \leftarrow \mathbf{w} - \eta [\underbrace{\hat{V}_\pi(s; \mathbf{w})}_{\text{prediction}} - \underbrace{(r + \gamma \hat{V}_\pi(s'; \mathbf{w}))}_{\text{target}}] \nabla_{\mathbf{w}} \hat{V}_\pi(s; \mathbf{w})$$



Algorithm: Q-learning

On each (s, a, r, s') :

$$\mathbf{w} \leftarrow \mathbf{w} - \eta [\underbrace{\hat{Q}_{\text{opt}}(s, a; \mathbf{w})}_{\text{prediction}} - \underbrace{(r + \gamma \max_{a' \in \text{Actions}(s)} \hat{Q}_{\text{opt}}(s', a'; \mathbf{w}))}_{\text{target}}] \nabla_{\mathbf{w}} \hat{Q}_{\text{opt}}(s, a; \mathbf{w})$$

Comparison

Q-learning:

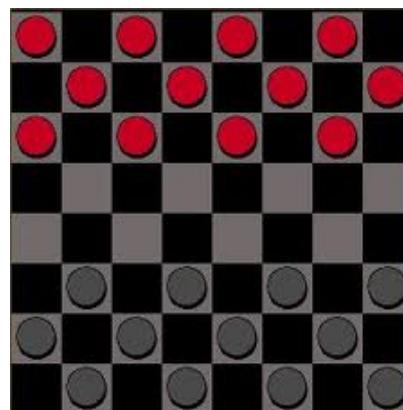
- Operate on $\hat{Q}_{\text{opt}}(s, a; \mathbf{w})$
- Off-policy: value is based on estimate of optimal policy
- To use, don't need to know MDP transitions $T(s, a, s')$

TD learning:

- Operate on $\hat{V}_\pi(s; \mathbf{w})$
- On-policy: value is based on exploration policy (usually based on \hat{V}_π)
- To use, need to know rules of the game $\text{Succ}(s, a)$

- TD learning is very similar to Q-learning. Both algorithms learn from the same data and are based on gradient-based weight updates.
- The main difference is that Q-learning learns the Q-value, which measures how good an action is to take in a state, whereas TD learning learns the value function, which measures how good it is to be in a state.
- Q-learning is an off-policy algorithm, which means that it tries to compute Q_{opt} , associated with the optimal policy (not Q_π), whereas TD learning is on-policy, which means that it tries to compute V_π , the value associated with a fixed policy π . Note that the action a does not show up in the TD updates because a is given by the fixed policy π . Of course, we usually are trying to optimize the policy, so we would set π to be the current guess of optimal policy $\pi(s) = \arg \max_{a \in \text{Actions}(s)} V(\text{Succ}(s, a); \mathbf{w})$.
- When we don't know the transition probabilities and in particular the successors, the value function isn't enough, because we don't know what effect our actions will have. However, in the game playing setting, we do know the transitions (the rules of the game), so using the value function is sufficient.

Learning to play checkers

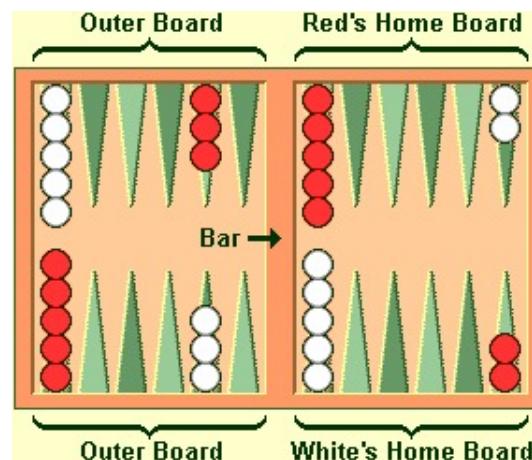


Arthur Samuel's checkers program [1959]:

- Learned by playing itself repeatedly (self-play)
- Smart features, linear evaluation function, use intermediate rewards
- Used alpha-beta pruning + search heuristics
- Reach human amateur level of play
- IBM 701: 9K of memory!

- The idea of using machine learning for game playing goes as far back as Arthur Samuel's checkers program. Many of the ideas (using features, alpha-beta pruning) were employed, resulting in a program that reached a human amateur level of play. Not bad for 1959!

Learning to play Backgammon

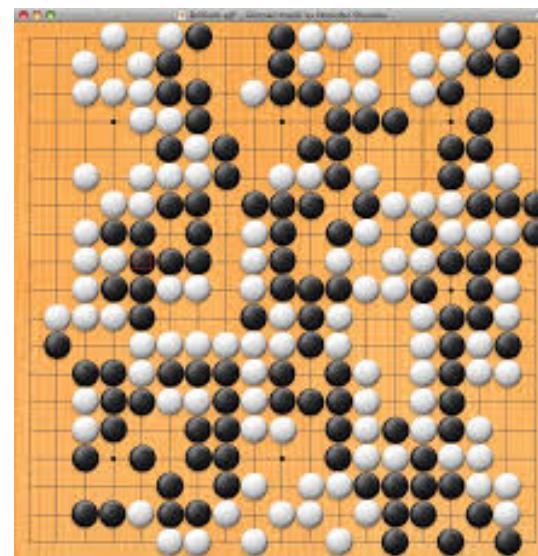


Gerald Tesauro's TD-Gammon [1992]:

- Learned weights by playing itself repeatedly (1 million times)
- Dumb features, neural network, no intermediate rewards
- Reached human expert level of play, provided new insights into opening

- Tesauro refined some of the ideas from Samuel with his famous TD-Gammon program provided the next advance, using a variant of TD learning called $\text{TD}(\lambda)$. It had dumber features, but a more expressive evaluation function (neural network), and was able to reach an expert level of play.

Learning to play Go



AlphaGo Zero [2017]:

- Learned by self play (4.9 million games)
- Dumb features (stone positions), neural network, no intermediate rewards, Monte Carlo Tree Search
- Beat AlphaGo, which beat Le Sedol in 2016
- Provided new insights into the game

- Very recently, self-play reinforcement learning has been applied to the game of Go. AlphaGo Zero uses a single neural network to predict winning probability and actions to be taken, using raw board positions as inputs. Starting from random weights, the network is trained to gradually improve its predictions and match the results of an approximate (Monte Carlo) tree search algorithm.



Summary so far

- Parametrize evaluation functions using features
- TD learning: learn an evaluation function

$$(\text{prediction}(\mathbf{w}) - \text{target})^2$$

Up next:

Turn-based



Simultaneous

Zero-sum



Non-zero-sum



Roadmap

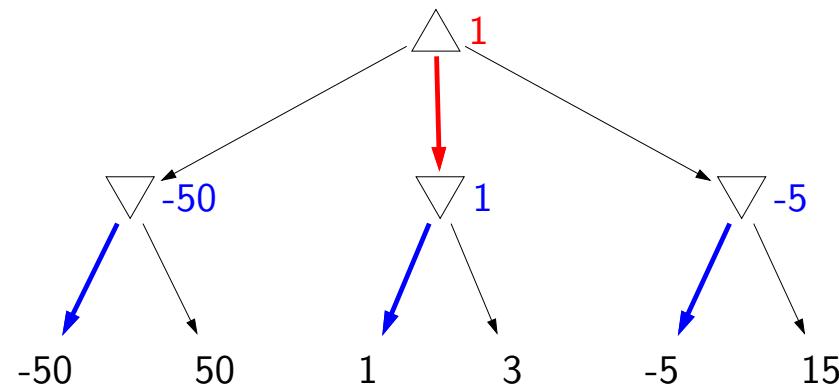
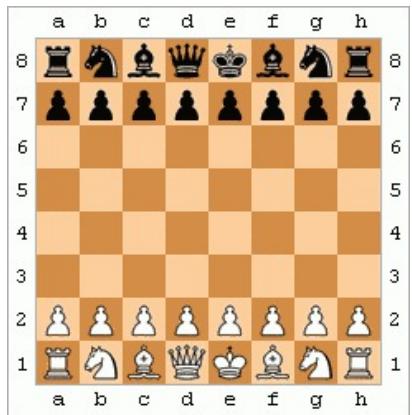
TD learning

Simultaneous games

Non-zero-sum games

State-of-the-art

Turn-based games:



Simultaneous games:



?

- Game trees were our primary tool to model turn-based games. However, in simultaneous games, there is no ordering on the player's moves, so we need to develop new tools to model these games. Later, we will see that game trees will still be valuable in understanding simultaneous games.



Two-finger Morra



Example: two-finger Morra

Players A and B each show 1 or 2 fingers.

If both show 1, B gives A 2 dollars.

If both show 2, B gives A 4 dollars.

Otherwise, A gives B 3 dollars.

[play with a partner]



Question

What was the outcome?

player A chose 1, player B chose 1

player A chose 1, player B chose 2

player A chose 2, player B chose 1

player A chose 2, player B chose 2



Payoff matrix



Definition: single-move simultaneous game

Players = {A, B}

Actions: possible actions

$V(a, b)$: **A's utility** if A chooses action a , B chooses b
(let V be **payoff matrix**)

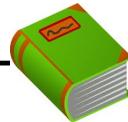


Example: two-finger Morra payoff matrix

A \ B	1 finger	2 fingers
1 finger	2	-3
2 fingers	-3	4

- In this lecture, we will consider only single move games. There are two players, A and B who both select from one of the available actions. The value or utility of the game is captured by a payoff matrix V whose dimensionality is $|\text{Actions}| \times |\text{Actions}|$. We will be analyzing everything from A's perspective, so entry $V(a, b)$ is the utility that A gets if he/she chooses action a and player B chooses b .

Strategies (policies)



Definition: pure strategy

A pure strategy is a single action:

$$a \in \text{Actions}$$



Definition: mixed strategy

A mixed strategy is a probability distribution

$$0 \leq \pi(a) \leq 1 \text{ for } a \in \text{Actions}$$



Example: two-finger Morra strategies

Always 1: $\pi = [1, 0]$

Always 2: $\pi = [0, 1]$

Uniformly random: $\pi = [\frac{1}{2}, \frac{1}{2}]$

- Each player has a **strategy** (or a policy). A pure strategy (deterministic policy) is just a single action. Note that there's no notion of state since we are only considering single-move games.
- More generally, we will consider **mixed strategies** (randomized policy), which is a probability distribution over actions. We will represent a mixed strategy π by the vector of probabilities.

Game evaluation



Definition: game evaluation

The **value** of the game if player A follows π_A and player B follows π_B is

$$V(\pi_A, \pi_B) = \sum_{a,b} \pi_A(a)\pi_B(b)V(a, b)$$



Example: two-finger Morra

Player A always chooses 1: $\pi_A = [1, 0]$

Player B picks randomly: $\pi_B = [\frac{1}{2}, \frac{1}{2}]$

Value:

$$-\frac{1}{2}$$

[whiteboard: matrix]

- Given a game (payoff matrix) and the strategies for the two players, we can define the value of the game.
- For pure strategies, the value of the game by definition is just reading out the appropriate entry from the payoff matrix.
- For mixed strategies, the value of the game (that is, the expected utility for player A) is gotten by summing over the possible actions that the players choose: $V(\pi_A, \pi_B) = \sum_{a \in \text{Actions}} \sum_{b \in \text{Actions}} \pi_A(a)\pi_B(b)V(a, b)$. We can also write this expression concisely using matrix-vector multiplications: $\pi_A^\top V \pi_B$.

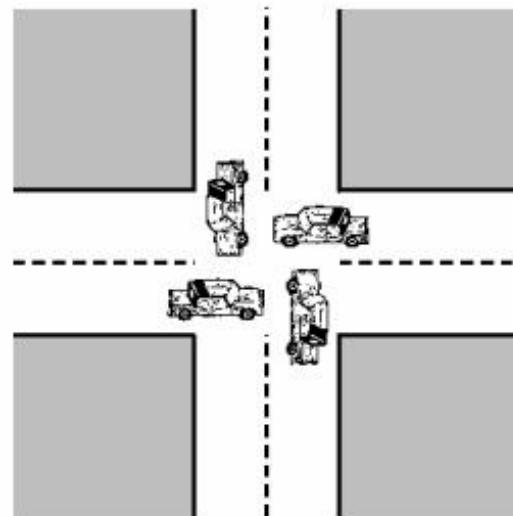
How to optimize?

Game value:

$$V(\pi_A, \pi_B)$$

Challenge: player A wants to maximize, player B wants to minimize...

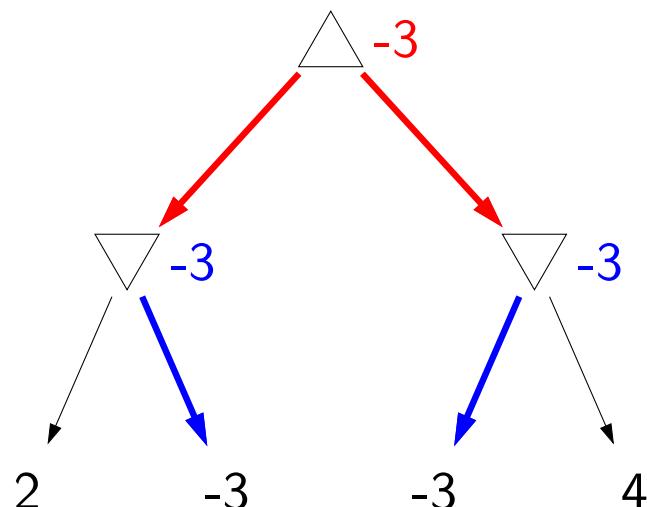
simultaneously



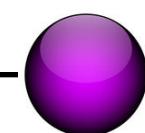
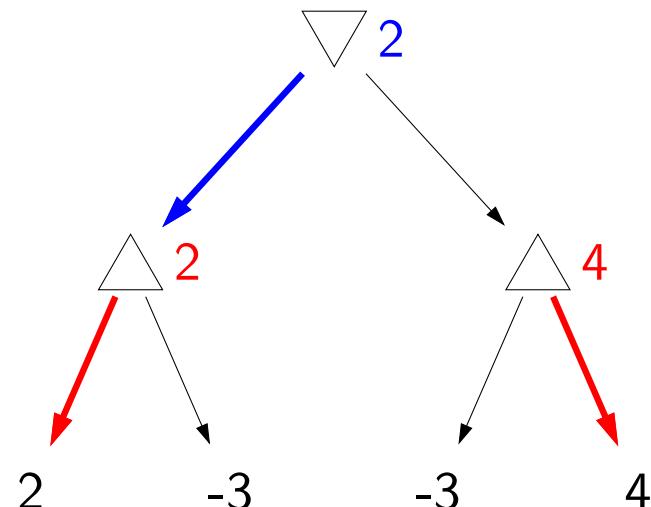
- Having established the values of fixed policies, let's try to optimize the policies themselves. Here, we run into a predicament: player A wants to maximize V but player B wants to minimize V **simultaneously**.
- Unlike turn-based games, we can't just consider one at a time. But let's consider the turn-based variant anyway to see where it leads us.

Pure strategies: who goes first?

Player A goes first:



Player B goes first:



Proposition: going second is no worse

$$\max_a \min_b V(a, b) \leq \min_b \max_a V(a, b)$$

- Let us first consider pure strategies, where each player just chooses one action. The game can be modeled by using the standard minimax game trees that we're used to.
- The main point is that if player A goes first, he gets -3 , but if he goes second, he gets 2 . In general, it's at least as good to go second, and often it is strictly better. This is intuitive, because seeing what the first player does gives more information.

Mixed strategies



Example: two-finger Morra

Player A reveals: $\pi_A = [\frac{1}{2}, \frac{1}{2}]$

Value $V(\pi_A, \pi_B) = \pi_B(1)(-\frac{1}{2}) + \pi_B(2)(+\frac{1}{2})$

Optimal strategy for player B is $\pi_B = [1, 0]$ (**pure!**)



Proposition: second player can play pure strategy

For any fixed mixed strategy π_A :

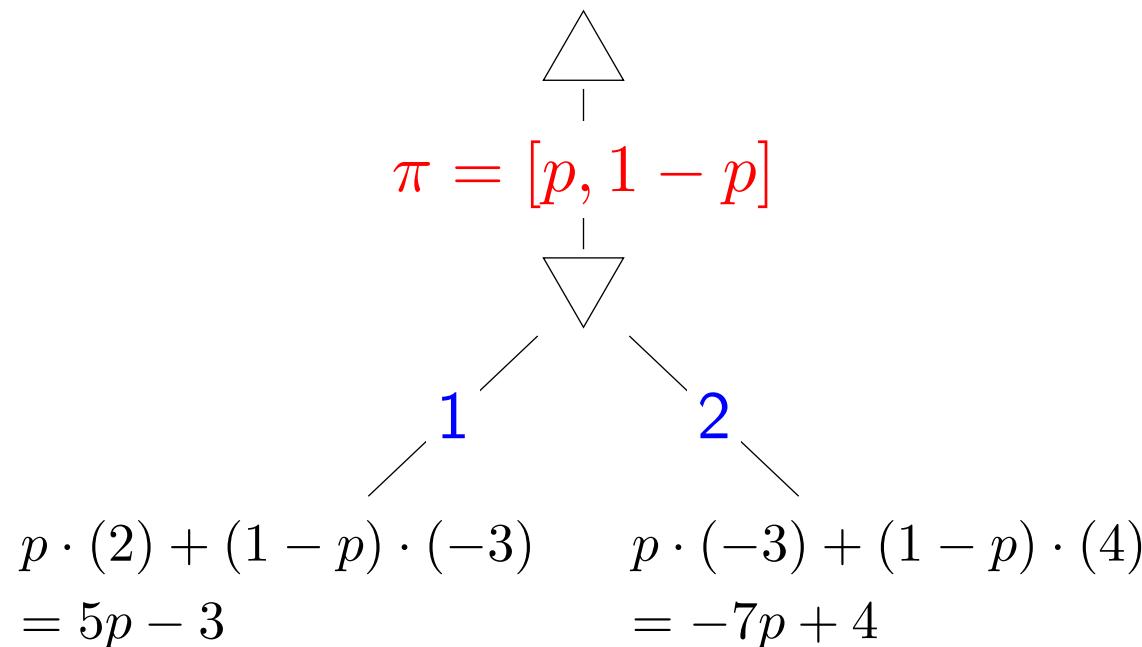
$$\min_{\pi_B} V(\pi_A, \pi_B)$$

can be attained by a pure strategy.

- Now let us consider mixed strategies. First, let's be clear on what playing a mixed strategy means. If player A chooses a mixed strategy, he reveals to player B the full probability distribution over actions, but importantly not a particular action (because that would be the same as choosing a pure strategy).
- As a warmup, suppose that player A reveals $\pi_A = [\frac{1}{2}, \frac{1}{2}]$. If we plug this strategy into the definition for the value of the game, we will find that the value is a convex combination between $\frac{1}{2}(2) + \frac{1}{2}(-3) = -\frac{1}{2}$ and $\frac{1}{2}(-3) + \frac{1}{2}(4) = \frac{1}{2}$. The value of π_B that minimizes this value is $[1, 0]$. The important part is that this is a **pure strategy**.
- It turns out that no matter what the payoff matrix V is, as soon as π_A is fixed, then the optimal choice for π_B is a pure strategy. This is useful because it will allow us to analyze games with mixed strategies more easily.

Mixed strategies

Player A first reveals his/her mixed strategy



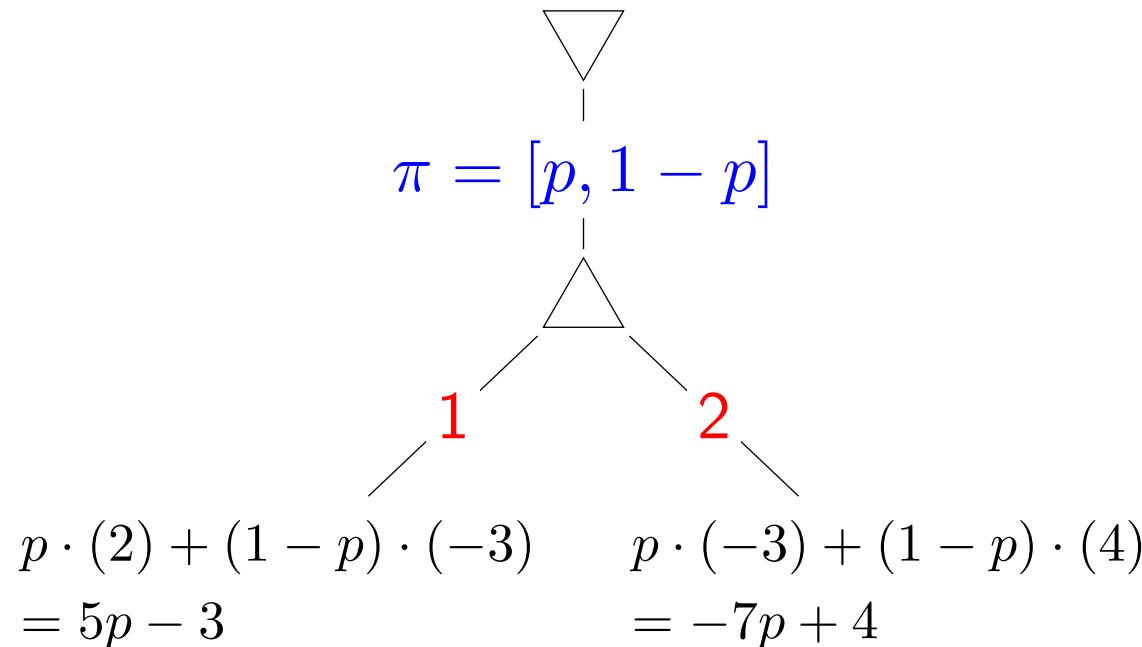
Minimax value of game:

$$\max_{0 \leq p \leq 1} \min\{5p - 3, -7p + 4\} = \boxed{-\frac{1}{12}} \text{ (with } p = \frac{7}{12})$$

- Now let us try to draw the minimax game tree where the player A first chooses a mixed strategy, and then player B chooses a pure strategy.
- There are an uncountably infinite number of mixed strategies for player A, but we can summarize all of these actions by writing a single action template $\pi = [p, 1 - p]$.
- Given player A's action, we can compute the value if player B either chooses 1 or 2. For example, if player B chooses 1, then the value of the game is $5p - 3$ (with probability p , player A chooses 1 and the value is 2; with probability $1 - p$ the value is -3). If player B chooses action 2, then the value of the game is $-7p + 4$.
- The value of the min node is $F(p) = \min\{5p - 3, -7p + 4\}$. The value of the max node (and thus the minimax value of the game) is $\max_{0 \leq 1 \leq p} F(p)$.
- What is the best strategy for player A then? We just have to find the p that maximizes $F(p)$, which is the minimum over two linear functions of p . If we plot this function, we will see that the maximum of $F(p)$ is attained when $5p - 3 = -7p + 4$, which is when $p = \frac{7}{12}$. Plugging that value of p back in yields $F(p) = -\frac{1}{12}$, the minimax value of the game if player A goes first and is allowed to choose a mixed strategy.
- Note that if player A decides on $p = \frac{7}{12}$, it doesn't matter whether player B chooses 1 or 2; the payoff will be the same: $-\frac{1}{12}$. This also means that whatever mixed strategy (over 1 and 2) player B plays, the payoff would also be $-\frac{1}{12}$.

Mixed strategies

Player B first reveals his/her mixed strategy



Minimax value of game:

$$\min_{p \in [0,1]} \max \{5p - 3, -7p + 4\} = \boxed{-\frac{1}{12}} \text{ (with } p = \frac{7}{12})$$

- Now let us consider the case where player B chooses a mixed strategy $\pi = [p, 1 - p]$ first. If we perform the analogous calculations, we'll find that we get that the minimax value of the game is exactly the same ($-\frac{1}{12}$)!
- Recall that for pure strategies, there was a gap between going first and going second, but here, we see that for mixed strategies, there is no such gap, at least in this example.
- Here, we have been computed minimax values in the conceptually same manner as we were doing it for turn-based games. The only difference is that our actions are mixed strategies (represented by a probability distribution) rather than discrete choices. We therefore introduce a variable (e.g., p) to represent the actual distribution, and any game value that we compute below that variable is a function of p rather than a specific number.

General theorem



Theorem: minimax theorem [von Neumann, 1928]

For every simultaneous two-player zero-sum game with a finite number of actions:

$$\max_{\pi_A} \min_{\pi_B} V(\pi_A, \pi_B) = \min_{\pi_B} \max_{\pi_A} V(\pi_A, \pi_B),$$

where π_A, π_B range over **mixed strategies**.

Upshot: revealing your optimal mixed strategy doesn't hurt you!

Proof: linear programming duality

Algorithm: compute policies using linear programming

- It turns out that having no gap is not a coincidence, and is actually one of the most celebrated mathematical results: the von Neumann minimax theorem. The theorem states that for any simultaneous two-player zero-sum game with a finite set of actions (like the ones we've been considering), we can just swap the min and the max: it doesn't matter which player reveals his/her strategy first, as long as their strategy is optimal. This is significant because we were stressing out about how to analyze the game when two players play simultaneously, but now we find that both orderings of the players yield the same answer. It is important to remember that this statement is true only for mixed strategies, not for pure strategies.
- This theorem can be proved using linear programming duality, and policies can be computed also using linear programming. The sketch of the idea is as follows: recall that the optimal strategy for the second player is always deterministic, which means that the $\max_{\pi_A} \min_{\pi_B} \dots$ turns into $\max_{\pi_A} \min_b \dots$. The min is now over n actions, and can be rewritten as n linear constraints, yielding a linear program.
- As an aside, recall that we also had a minimax result for turn-based games, where the max and the min were over agent and opponent policies, which map states to actions. In that case, optimal policies were always deterministic because at each state, there is only one player choosing.



Summary

- **Challenge:** deal with simultaneous min/max moves
- **Pure strategies:** going second is better
- **Mixed strategies:** doesn't matter (von Neumann's minimax theorem)



Roadmap

TD learning

Simultaneous games

Non-zero-sum games

State-of-the-art

Utility functions

Competitive games: minimax (linear programming)



Collaborative games: pure maximization (plain search)



Real life: ?

- So far, we have focused on competitive games, where the utility of one player is the exact opposite of the utility of the other. The minimax principle is the appropriate tool for modeling these scenarios.
- On the other extreme, we have collaborative games, where the two players have the same utility function. This case is less interesting, because we are just doing pure maximization (e.g., finding the largest element in the payoff matrix or performing search).
- In many practical real life scenarios, games are somewhere in between pure competition and pure collaboration. This is where things get interesting...

Prisoner's dilemma



Example: Prisoner's dilemma

Prosecutor asks A and B individually if each will testify against the other.

If both testify, then both are sentenced to 5 years in jail.

If both refuse, then both are sentenced to 1 year in jail.

If only one testifies, then he/she gets out for free; the other gets a 10-year sentence.

[play with a partner]



Question

What was the outcome?

player A testified, player B testified

player A refused, player B testified

player A testified, player B refused

player A refused, player B refused

Prisoner's dilemma



Example: payoff matrix

B \ A	testify	refuse
testify	$A = -5, B = -5$	$A = -10, B = 0$
refuse	$A = 0, B = -10$	$A = -1, B = -1$



Definition: payoff matrix

Let $V_p(\pi_A, \pi_B)$ be the utility for player p .

- In the prisoner's dilemma, the players get both penalized only a little bit if they both refuse to testify, but if one of them defects, then the other will get penalized a huge amount. So in practice, what tends to happen is that both will testify and both get sentenced to 5 years, which is clearly worse than if they both had cooperated.

Nash equilibrium

Can't apply von Neumann's minimax theorem (not zero-sum), but get something weaker:



Definition: Nash equilibrium

A **Nash equilibrium** is (π_A^*, π_B^*) such that no player has an incentive to change his/her strategy:

$$V_A(\pi_A^*, \pi_B^*) \geq V_A(\pi_A, \pi_B^*) \text{ for all } \pi_A$$

$$V_B(\pi_A^*, \pi_B^*) \geq V_B(\pi_A^*, \pi_B) \text{ for all } \pi_B$$



Theorem: Nash's existence theorem [1950]

In any finite-player game with finite number of actions, there exists **at least one** Nash equilibrium.

- Since we no longer have a zero-sum game, we cannot apply the minimax theorem, but we can still get a weaker result.
- A Nash equilibrium is kind of a state point, where no player has an incentive to change his/her policy unilaterally. Another major result in game theory is Nash's existence theorem, which states that any game with a finite number of players (importantly, not necessarily zero-sum) has at least one Nash equilibrium (a stable point). It turns out that finding one is hard, but we can be sure that one exists.

Examples of Nash equilibria



Example: Two-finger Morra

Nash equilibrium: A and B both play $\pi = [\frac{7}{12}, \frac{5}{12}]$.



Example: Collaborative two-finger Morra

Two Nash equilibria:

- A and B both play 1 (value is 2).
- A and B both play 2 (value is 4).



Example: Prisoner's dilemma

Nash equilibrium: A and B both testify.

- Here are three examples of Nash equilibria. The minimax strategies for zero-sum are also equilibria (and they are global optima).
- For purely collaborative games, the equilibria are simply the entries of the payoff matrix for which no other entry in the row or column are larger. There are often multiple local optima here.
- In the Prisoner's dilemma, the Nash equilibrium is when both players testify. This is of course not the highest possible reward, but it is stable in the sense that neither player would want to change his/her strategy. If both players had refused, then one of the players could testify to improve his/her payoff (from -1 to 0).



Summary so far

Simultaneous zero-sum games:

- von Neumann's minimax theorem
- Multiple minimax strategies, single game value

Simultaneous non-zero-sum games:

- Nash's existence theorem
- Multiple Nash equilibria, multiple game values

Huge literature in game theory / economics

- For simultaneous zero-sum games, all minimax strategies have the same game value (and thus it makes sense to talk about the value of a game). For non-zero-sum games, different Nash equilibria could have different game values (for example, consider the collaborative version of two-finger Morra).



Roadmap

TD learning

Simultaneous games

Non-zero-sum games

State-of-the-art



State-of-the-art: chess

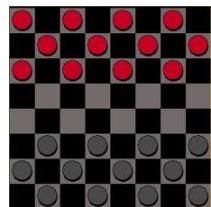
1997: IBM's Deep Blue defeated world champion Gary Kasparov

Fast computers:

- Alpha-beta search over 30 billion positions, depth 14
- Singular extensions up to depth 20

Domain knowledge:

- Evaluation function: 8000 features
- 4000 "opening book" moves, all endgames with 5 pieces
- 700,000 grandmaster games
- Null move heuristic: opponent gets to move twice



State-of-the-art: checkers

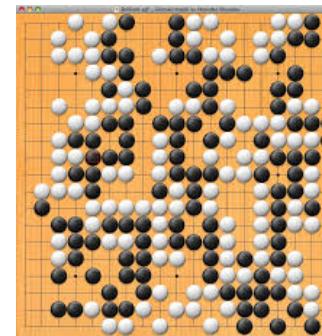
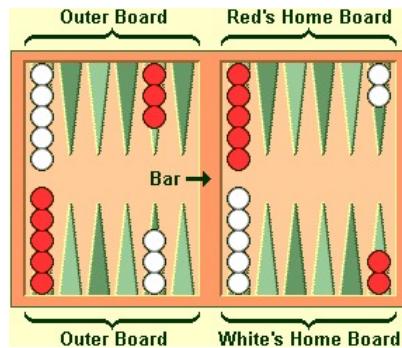
1990: Jonathan Schaeffer's **Chinook** defeated human champion; ran on standard PC

Closure:

- 2007: Checkers solved in the minimax sense (outcome is draw), but doesn't mean you can't win
- Alpha-beta search + 39 trillion endgame positions

Backgammon and Go

Alpha-beta search isn't enough...



Challenge: large branching factor

- Backgammon: randomness from dice (can't prune!)
- Go: large board size (361 positions)

Solution: learning

- For games such as checkers and chess with a manageable branching factor, one can rely heavily on minimax search along with alpha-beta pruning and a lot of computation power. A good amount of domain knowledge can be employed as to attain or surpass human-level performance.
- However, games such as Backgammon and Go require more due to the large branching factor. Backgammon does not intrinsically have a larger branching factor, but much of this branching is due to the randomness from the dice, which cannot be pruned (it doesn't make sense to talk about the most promising dice move).
- As a result, programs for these games have relied a lot on TD learning to produce good evaluation functions without searching the entire space.

AlphaGo



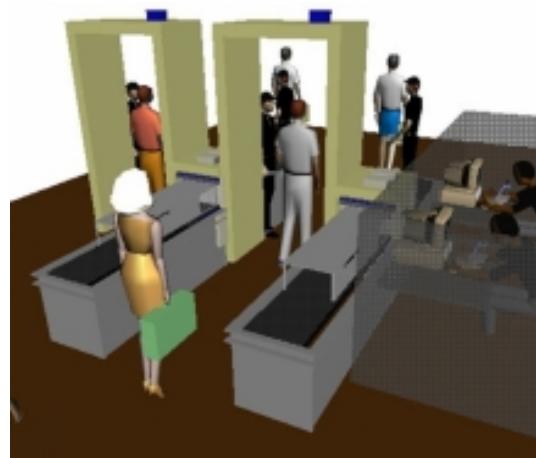
- Supervised learning: on human games
- Reinforcement learning: on self-play games
- Evaluation function: convolutional neural network (value network)
- Policy: convolutional neural network (policy network)
- Monte Carlo Tree Search: search / lookahead

Section: AlphaGo Zero

- The most recent visible advance in game playing was March 2016, when Google DeepMind's AlphaGo program defeated Le Sedol, one of the best professional Go players 4-1.
- AlphaGo took the best ideas from game playing and machine learning. DeepMind executed these ideas well with lots of computational resources, but these ideas should already be familiar to you.
- The learning algorithm consisted of two phases: a supervised learning phase, where a policy was trained on games played by humans (30 million positions) from the KGS Go server; and a reinforcement learning phase, where the algorithm played itself in attempt to improve, similar to what we say with Backgammon.
- The model consists of two pieces: a value network, which is used to evaluate board positions (the evaluation function); and a policy network, which predicts which move to make from any given board position (the policy). Both are based on convolutional neural networks, which we'll discuss later in the class.
- Finally, the policy network is not used directly to select a move, but rather to guide the search over possible moves in an algorithm similar to Monte Carlo Tree Search.

Other games

Security games: allocate limited resources to protect a valuable target.
Used by TSA security, Coast Guard, protect wildlife against poachers, etc.



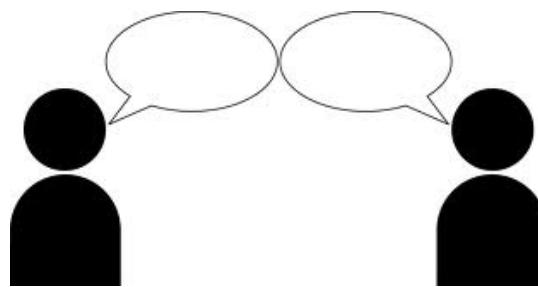
- The techniques that we've developed for game playing go far beyond recreational uses. Whenever there are multiple parties involved with conflicting interests, game theory can be employed to model the situation.
- For example, in a security game a defender needs to protect a valuable target from a malicious attacker. Game theory can be used to model these scenarios and devise optimal (randomized) strategies. Some of these techniques are used by TSA security at airports, to schedule patrol routes by the Coast Guard, and even to protect wildlife from poachers.

Other games

Resource allocation: users share a resource (e.g., network bandwidth); selfish interests leads to volunteer's dilemma



Language: people have speaking and listening strategies, mostly collaborative, applied to dialog systems



- For example, in resource allocation, we might have n people wanting to access some Internet resource. If all of them access the resource, then all of them suffer because of congestion. Suppose that if $n - 1$ connect, then those people can access the resource and are happy, but the one person left out suffers. Who should volunteer to step out (this is the volunteer's dilemma)?
- Another interesting application is modeling communication. There are two players, the speaker and the listener, and the speaker's actions are to choose what words to use to convey a message. Usually, it's a collaborative game where utility is high when communication is successful and efficient. These game-theoretic techniques have been applied to building dialog systems.



Summary

- Main challenge: not just one objective
- Minimax principle: guard against adversary in turn-based games
- Simultaneous non-zero-sum games: mixed strategies, Nash equilibria
- Strategy: **search game tree + learned evaluation function**

- Games are an extraordinary rich topic of study, and we have only seen the tip of the iceberg. Beyond simultaneous non-zero-sum games, which are already complex, there are also games involving partial information (e.g., poker).
- But even if we just focus on two-player zero-sum games, things are quite interesting. To build a good game-playing agent involves integrating the two main thrusts of AI: search and learning, which are really symbiotic. We can't possibly search an exponentially large number of possible futures, which means we fall back to an evaluation function. But in order to learn an evaluation function, we need to search over enough possible futures to build an accurate model of the likely outcome of the game.