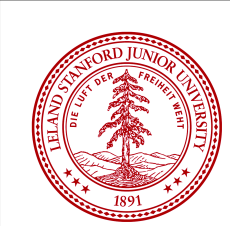


Lecture 5: Search I

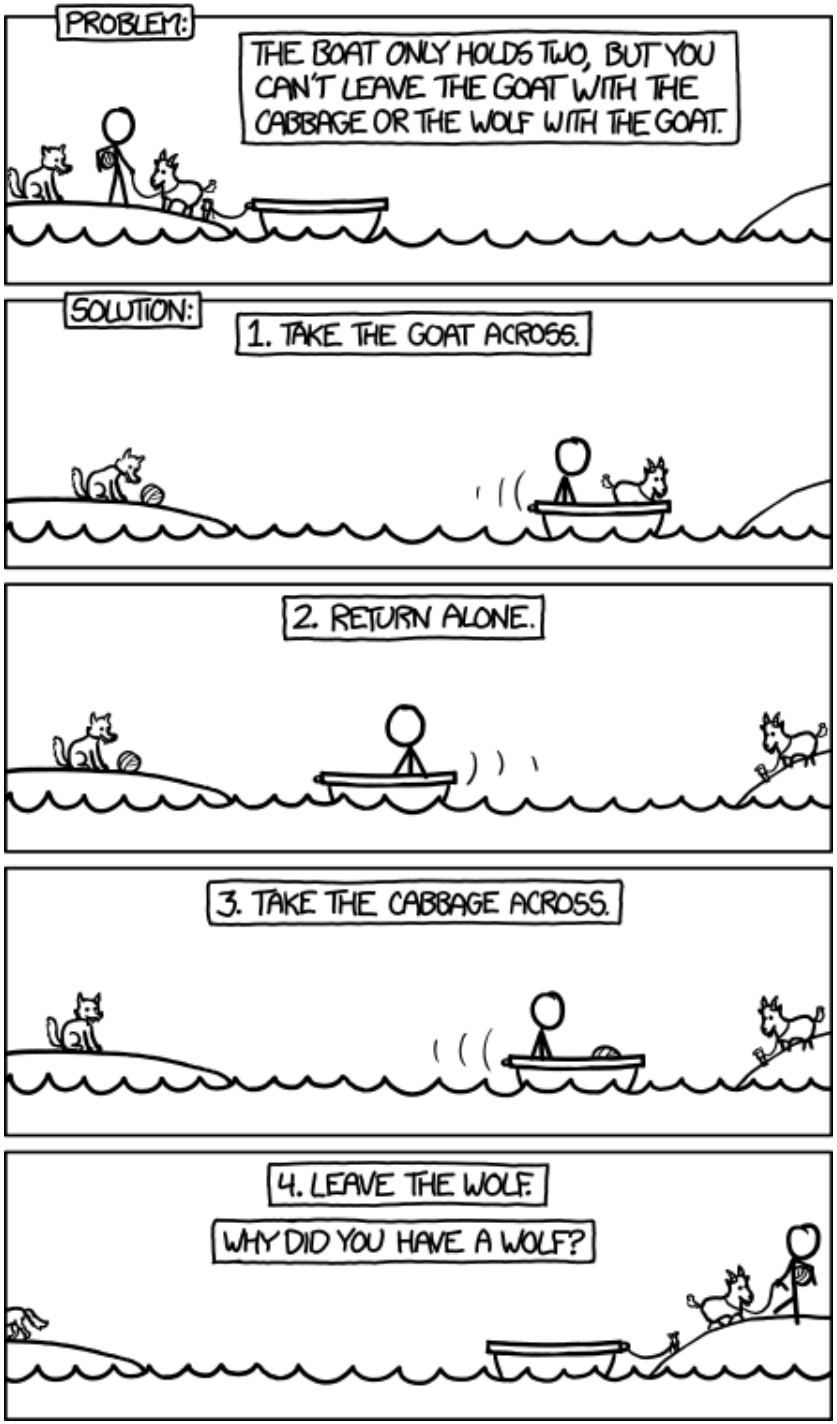
cs221.stanford.edu/q

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?

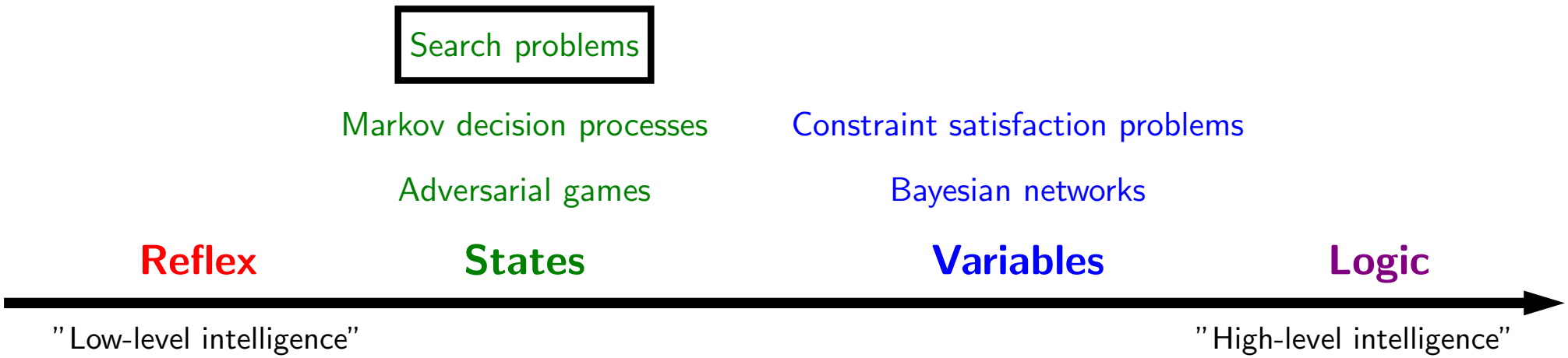
4
5
6
7
no solution

- 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, observing the consequences. If nothing got eaten, you might continue with the next action. Otherwise, you undo that move and try something else.
- How can we get a machine to do 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?

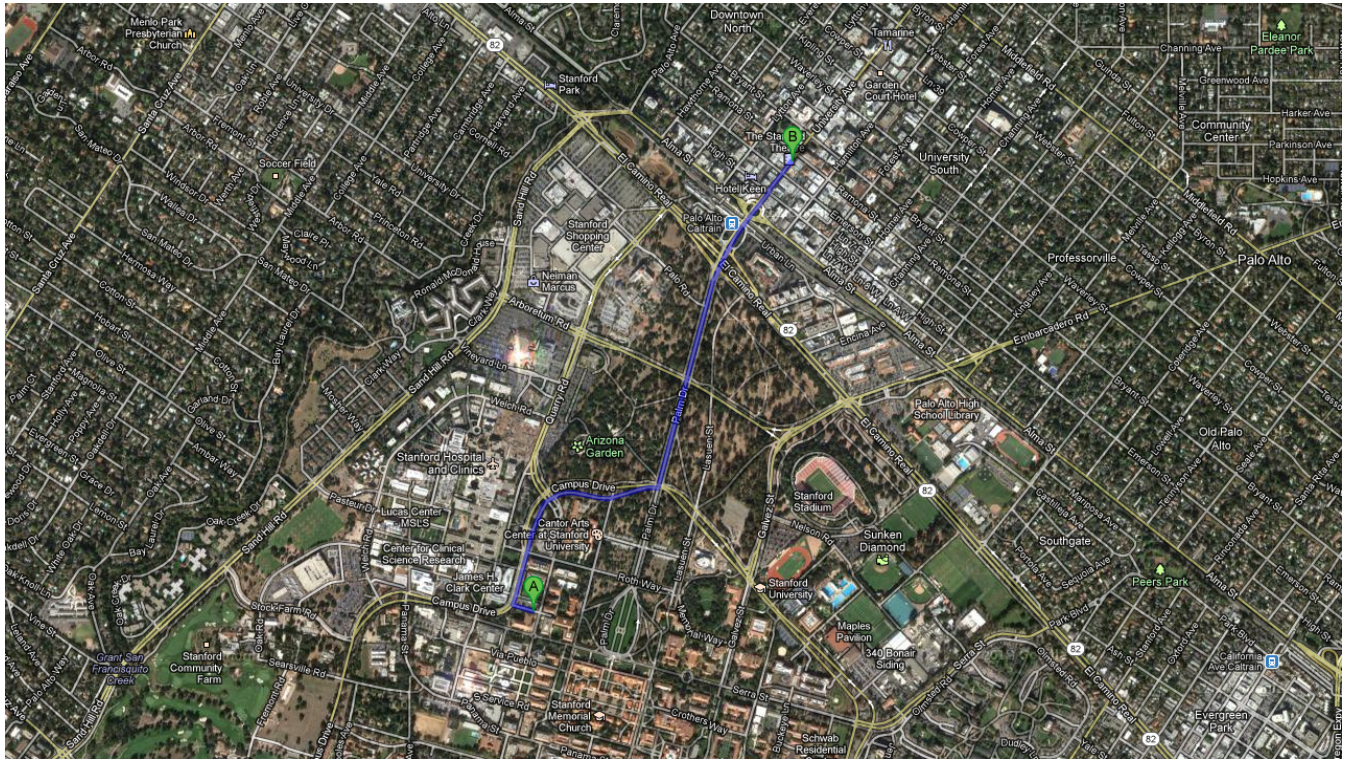
- This example points out that sometimes you can do better if you change the model (perhaps the value of having a wolf is zero) instead of focusing on the algorithm.

Course plan



Machine learning

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 x 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).

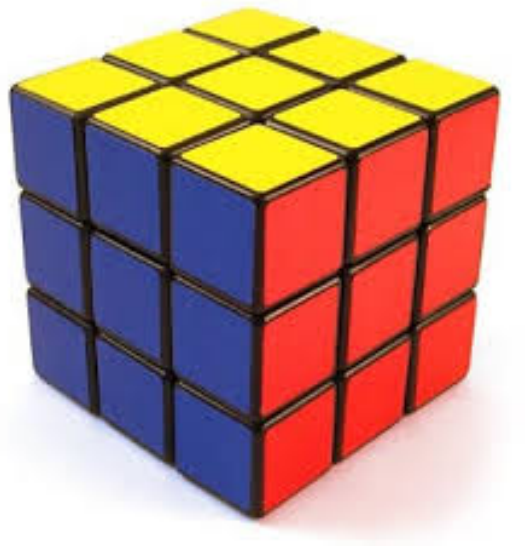
Application: robot motion planning

Objective: fastest? most energy efficient? safest?

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

Linear classifier (reflex-based models):

Search problem (state-based models):

Key: model effects of entire action sequence!

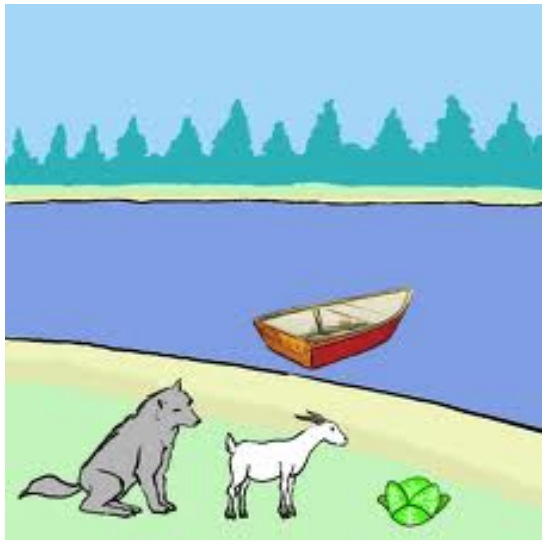
- Last week, we finished our tour of machine learning of **reflex-based models** (e.g., linear predictors and neural networks) that output either a $+1$ or -1 (for binary classification) or a real number (for regression).
- While reflex-based models were appropriate for some applications such as sentiment 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 taking 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 to take 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.
- There are many real-world examples of this paradigm, which we will describe next. 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.

Roadmap

Tree search

Dynamic programming

Uniform cost search



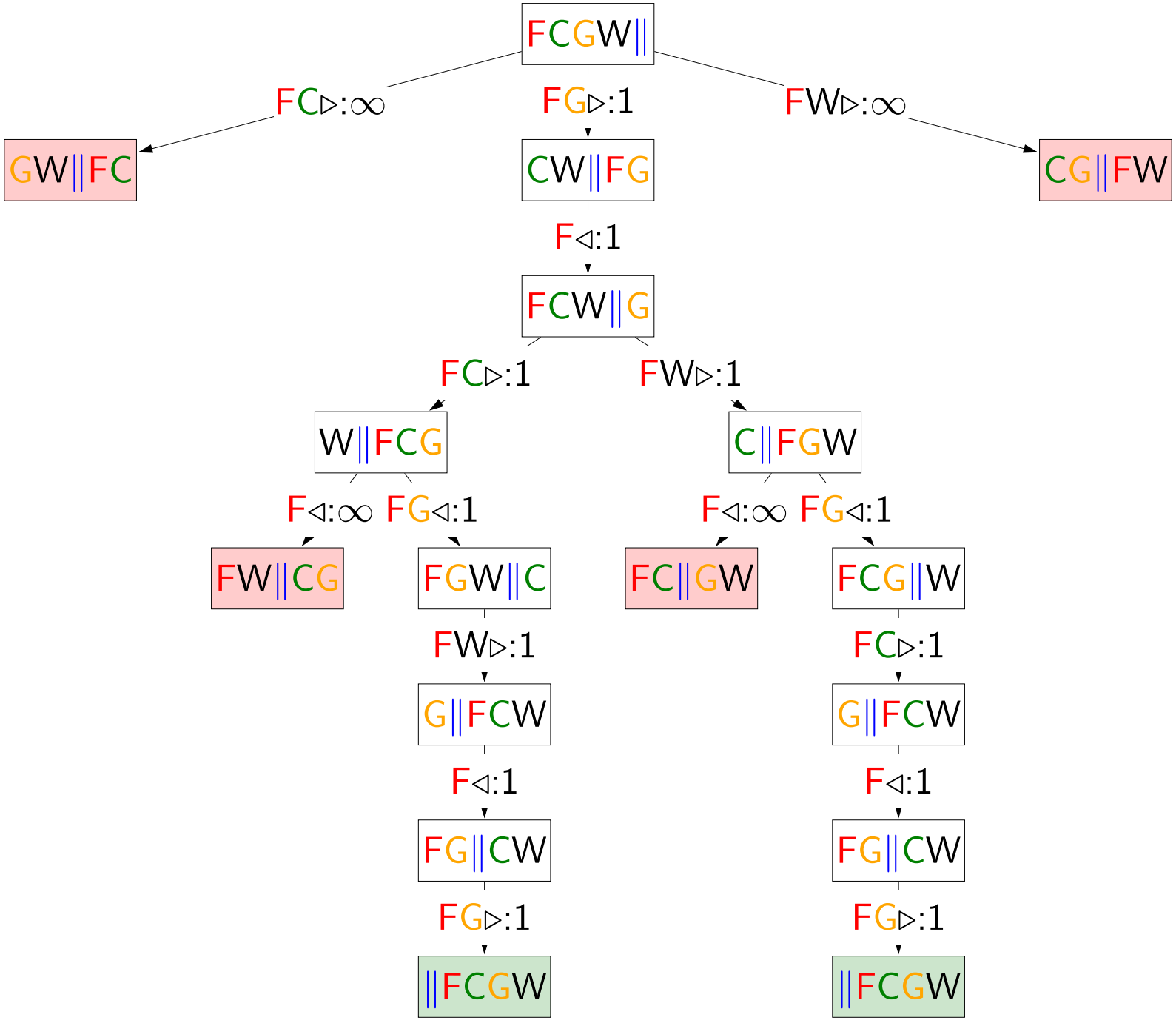
Farmer Cabbage Goat Wolf

Actions:

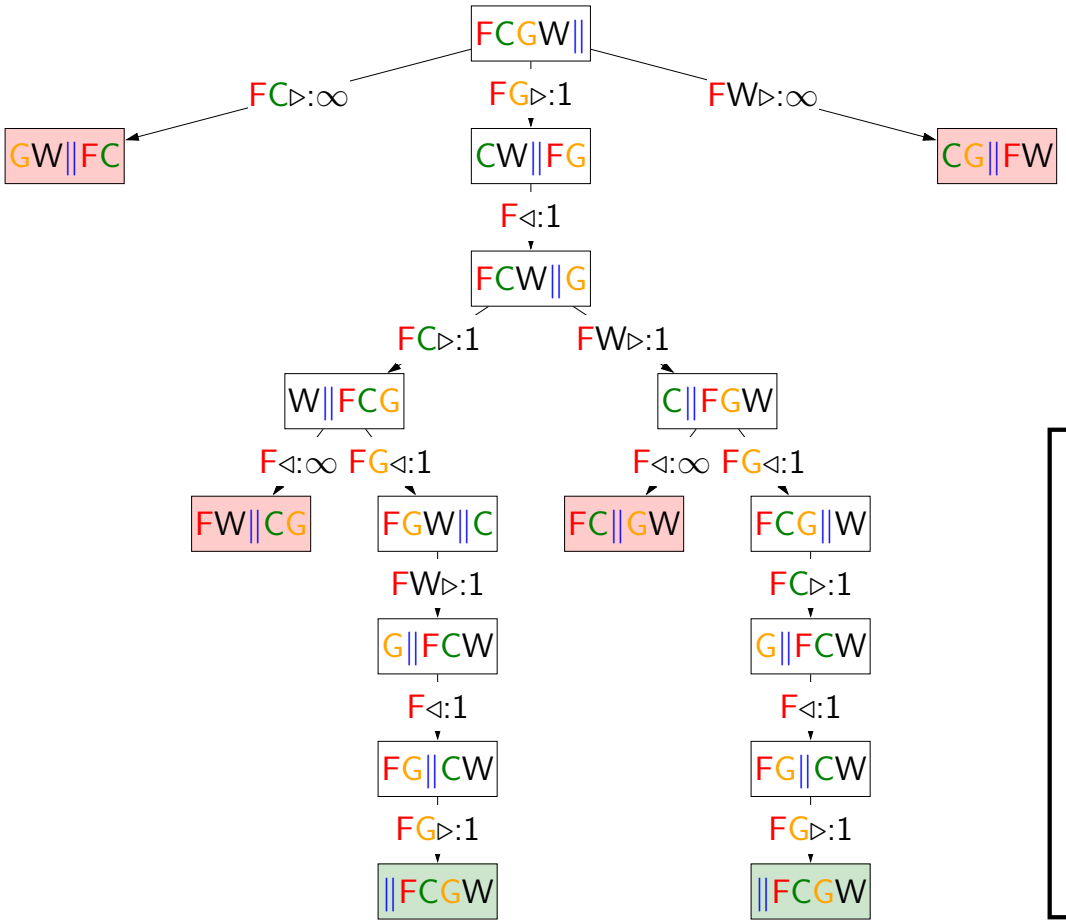
F▷	F◁
FC▷	FC◁
FG▷	FG◁
FW▷	FW◁

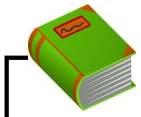
Approach: build a **search tree** ("what if?")

- 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 FG$\triangleright$ means that the farmer will take the goat across to the right bank; F$\triangleleft$ means that the farmer is coming back to the left bank alone.


FCGW||
FC▷:∞
FG▷:1
FW▷:∞
GW||FC
CW||FG
CG||FW
F◁:1
FCW||G
FC▷:1
FW▷:1
W||FCG
C||FGW
F◁:∞
FG◁:1
F◁:∞
FG◁:1
FW||CG
FGW||C
FC||GW
FCG||W
FW▷:1
FC▷:1
G||FCW
G||FCW
F◁:1
F◁:1
FG||CW
FG||CW
FG▷:1
FG▷:1
||FCGW
||FCGW

Search problem





Definition: search problem

- s_{start}: starting state
- Actions(s): possible actions
- Cost(s, a): action cost
- Succ(s, a): successor
- IsGoal(s): found solution?

- 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 goal states (IsGoal(s) is true). Each edge leaving a node s corresponds to a possible action $a \in$ Actions(s) that could be performed in state s. The edge is labeled with the action and its cost, written a : Cost(s, a). The action leads deterministically to the successor state 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 has the 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. Invalid actions (ones that result in an eating event) cost ∞ and the successor is marked in red. We disallow actions that return us to an earlier configuration. The green nodes are the goal states. 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?

[live solution]

- Let's consider take 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`, `isGoal` and `succAndCost`.

Backtracking search

Algorithm: backtracking search

```
def BacktrackingSearch(s, path):
    If IsGoal(s): update minimum cost path
    For each action a ∈ Actions(s):
        Extend path with Succ(s, a) and Cost(s, a)
        Call BacktrackingSearch(Succ(s, a), path)
```

[whiteboard: search tree]

[live solution]

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 `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 $1 + b + b^2 + \cdots + b^D = \frac{b^{D+1}-1}{b-1} = O(b^D)$. 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.

Depth-first search

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

Assumption: zero action costs

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

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 the goal state, then we have to search the entire tree.
- However, if there are many ways to reach the goal 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

Idea: explore all nodes in order of increasing depth.

Assumption: constant action costs

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

Legend: b actions per state, solution has d actions

- Space: now $O(b^d)$ (a lot worse!)
- Time: $O(b^d)$ (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 goal 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

Idea:

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

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

Assumption: constant action costs

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

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 $1, 2, 3, \ldots$ 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**
DFS	zero	$O(D)$	$O(b^D)$
BFS	constant	$O(b^d)$	$O(b^d)$
DFS-ID	constant	$O(d)$	$O(b^d)$
Backtracking	any	$O(D)$	$O(b^D)$

- 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. Memory cannot magically grow, whereas time grows just by running things longer or parallelizing across multiple machines.

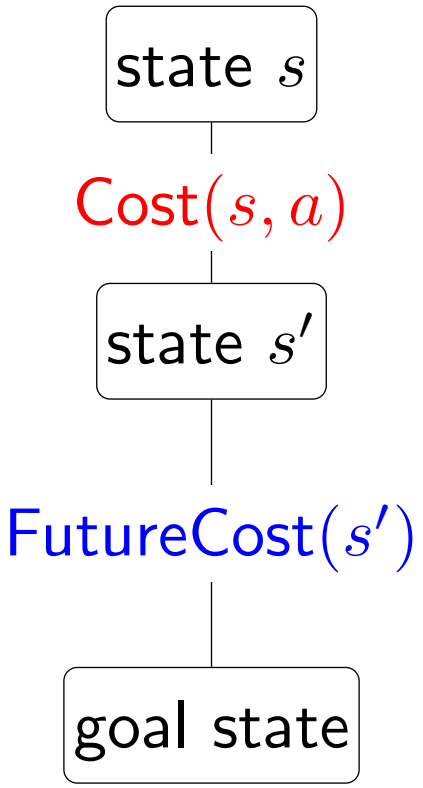
Roadmap

Tree search

Dynamic programming

Uniform cost search

Dynamic programming



Minimum cost path from state s to a goal state:

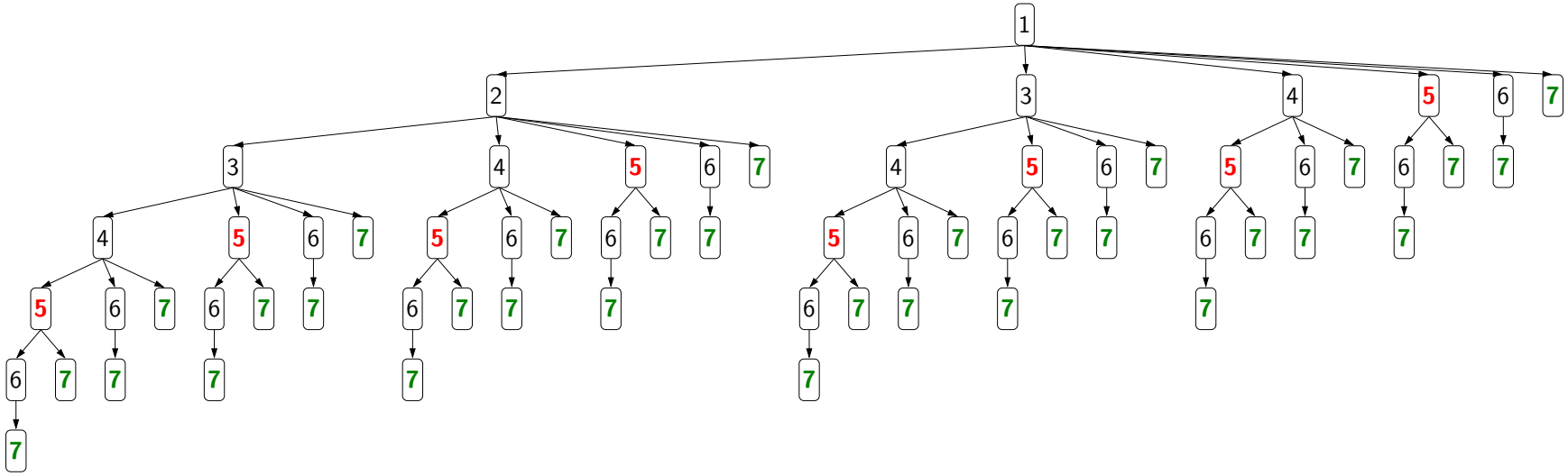
$$\text{FutureCost}(s) = \begin{cases} 0 & \text{if IsGoal}(s) \\ \min_{a \in \text{Actions}(s)} [\text{Cost}(s, a) + \text{FutureCost}(\text{Succ}(s, a))] & \text{otherwise} \end{cases}$$

- 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 FutureCost(s) as the cost of the minimum cost path from s to a goal state. The minimum cost path starting with a state s to a goal state must take a first action a, which results in another state s', from which we better take a minimum cost path to a goal 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 a goal 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 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.

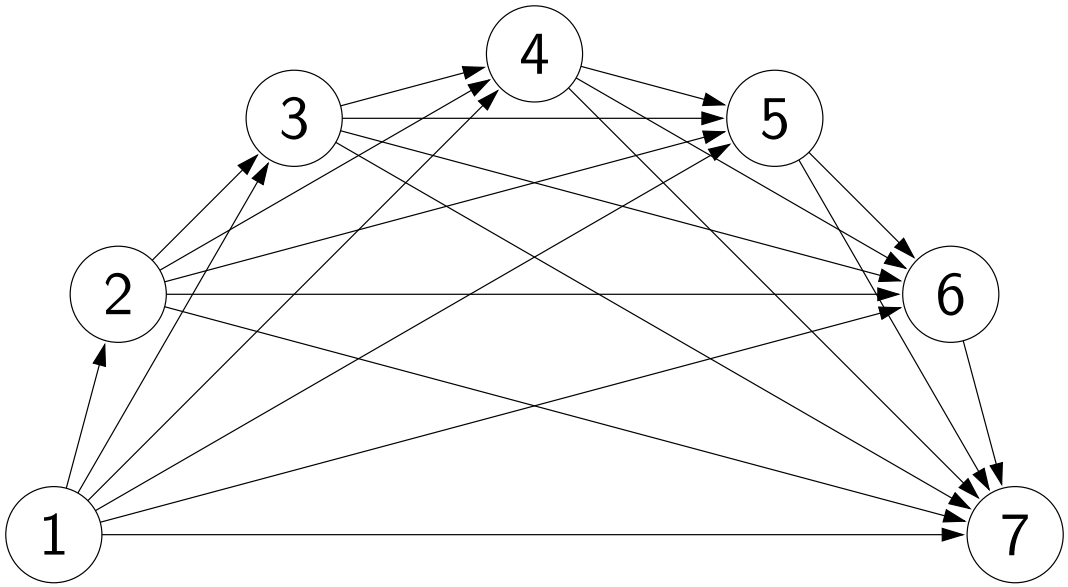


Observation: future costs only depend on current city

- 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 goal 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**.

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. This provides us with no longer a tree, but a directed acyclic graph with only n nodes rather than exponential in n nodes.

Dynamic programming

Algorithm: dynamic programming

def DynamicProgramming(s):

    If already computed for s, return cached answer.

    If IsGoal(s): return solution

    For each action $a \in$ Actions(s): ...

[live solution]

Assumption: acyclicity

The state graph defined by Actions(s) and 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, using 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**.

- 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.

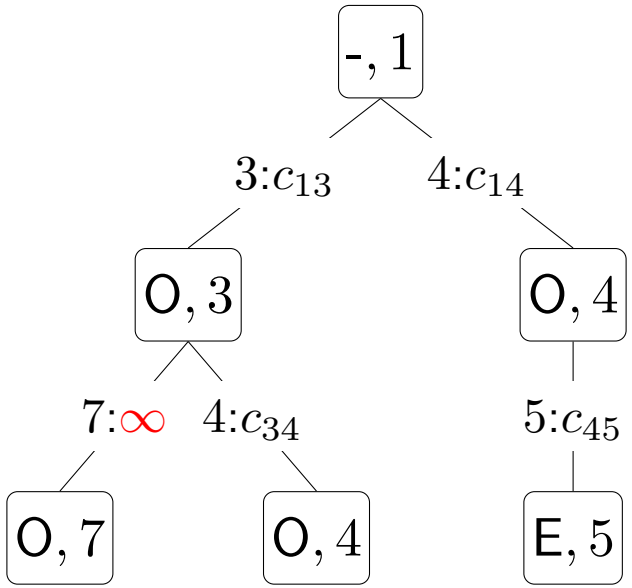
Handling additional constraints

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.

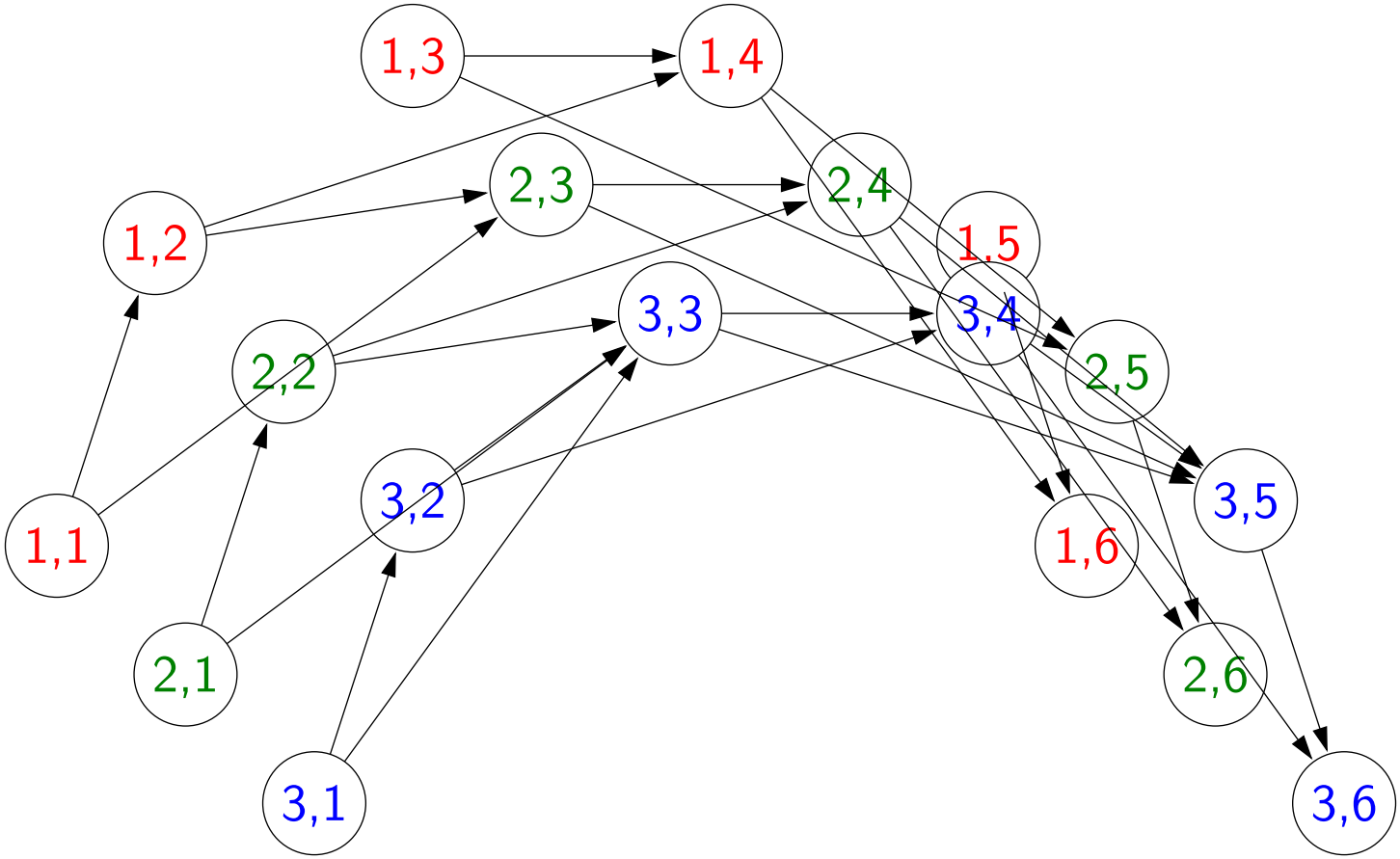
cs221.stanford.edu/q

Question

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

State graph

State: (min(number of odd cities visited, 3), 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.

cs221.stanford.edu/q

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** — exponential savings

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

Roadmap

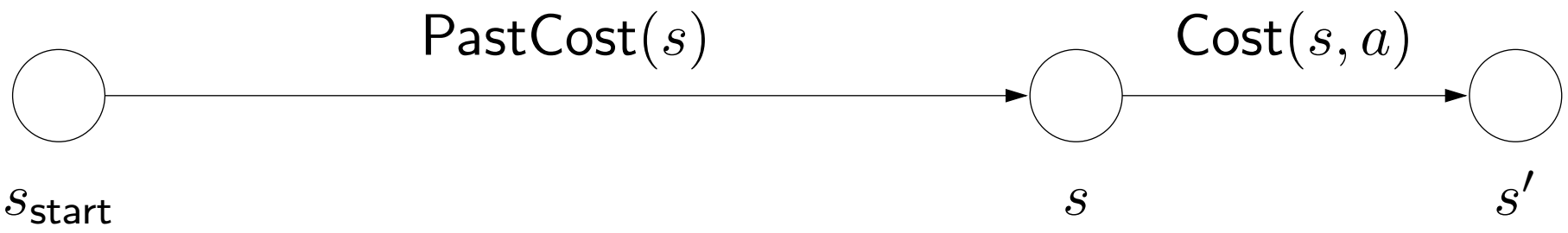
Tree search

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 PastCost(s) before 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 goal state.
- We can analogously define PastCost(s), the cost of the minimum cost path from the start state to s. If instead of having access to the successors via 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 PastCost(s).
- Dynamic programming relies on the absence of cycles, so that there was 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 PastCost(s) will always be computed before 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.

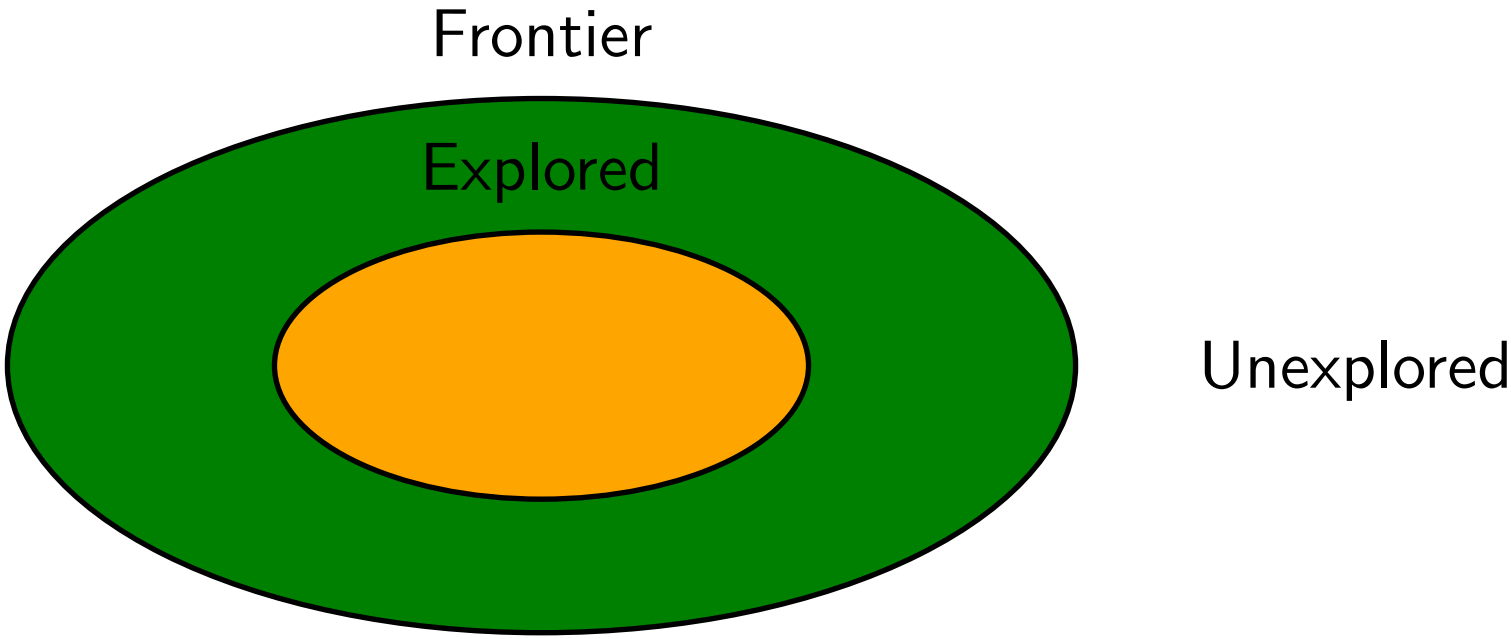
Assumption: non-negativity

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 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 a goal 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 goal state.

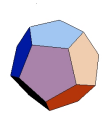
High-level strategy



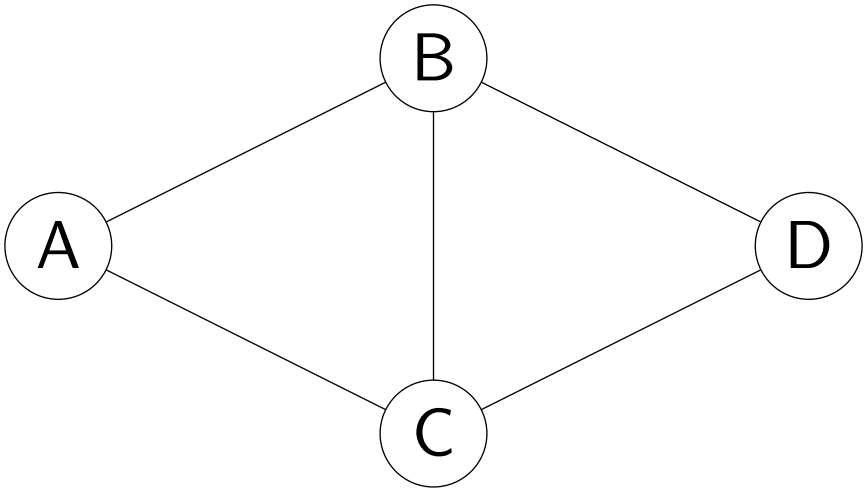
- Explored: states we've found the optimal path to
- Frontier: states we've seen, still figuring out how to get there cheaply
- Unexplored: states we haven't seen

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

Uniform cost search example



Example: UCS example



Start state: A, goal state: D

[whiteboard]

Best path:

$$A \rightarrow B \rightarrow C \rightarrow D \text{ with cost } 3$$

- 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 goal 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 IsGoal(s): return solution
- Add s to **explored**
- For each action $a \in$ Actions(s):
  - Get successor $s' \leftarrow$ Succ(s, a)
  - If s' already in explored: continue
  - Update **frontier** with s' and priority $p +$ Cost(s, a)

[live solution]

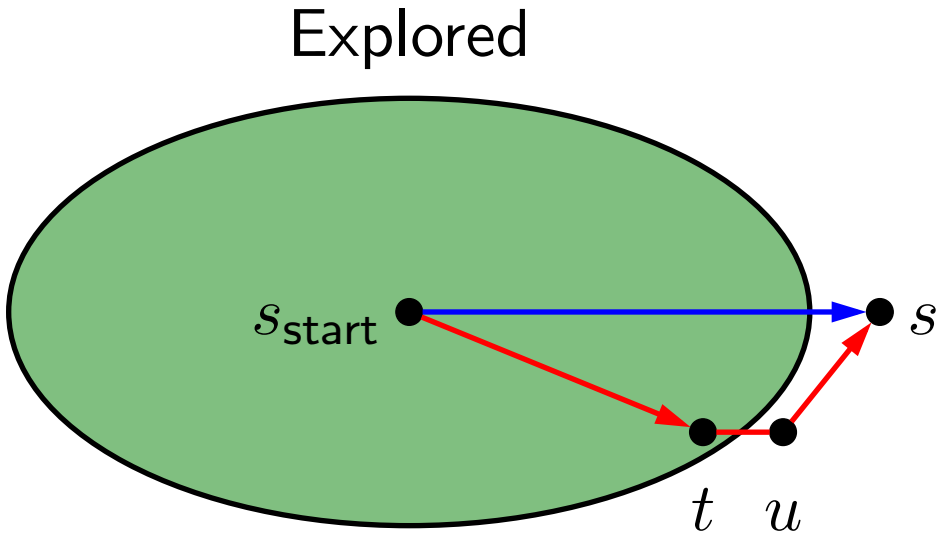
- Implementation note: we use `util.PriorityQueue` which supports `removeMin` and `update`. Note that `frontier.update(state, pastCost)` returns whether `pastCost` improves the existing estimate of the past cost of `state`.

Analysis of uniform cost search

Theorem: correctness

When a state s is popped off the frontier, its priority is 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 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 part 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 goal 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

- 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 goal state. However, there is an overhead with maintaining the priority queue.
- Note: in these big-O calculations, we are assuming the number of actions per state is a constant (independent of n and N).

cs221.stanford.edu/q

Question

What is the most interesting thing you learned from lecture today?

Summary

- Tree search: memory efficient, suitable for huge state spaces (to the extent anything works)
- 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, we need to make sure that we did a good job with the modeling of the state.