Lecture 1, Part 2: Search I

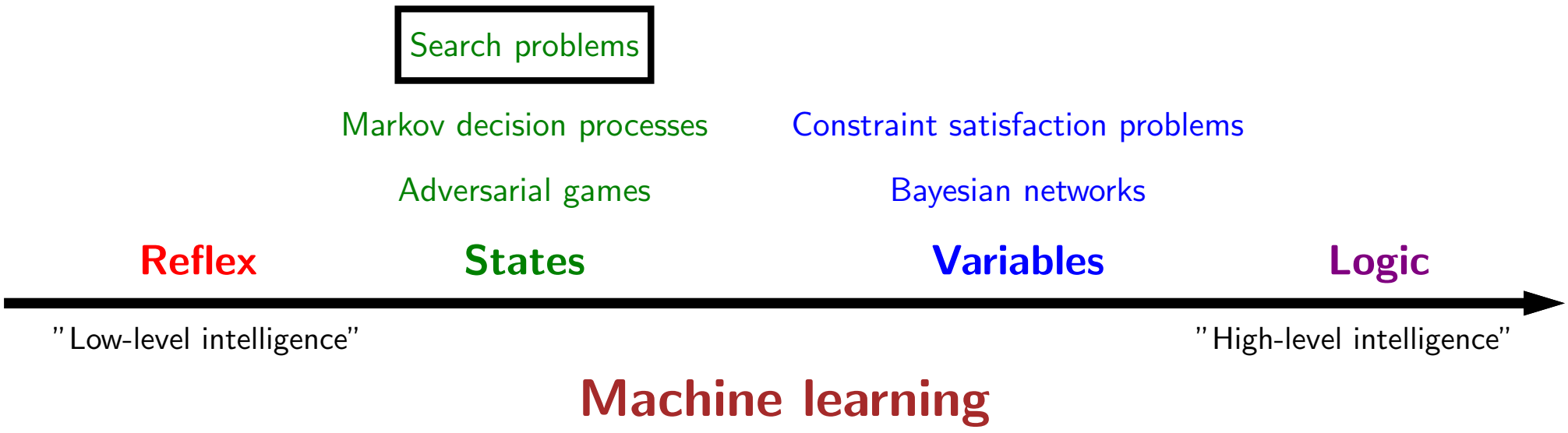
Question

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

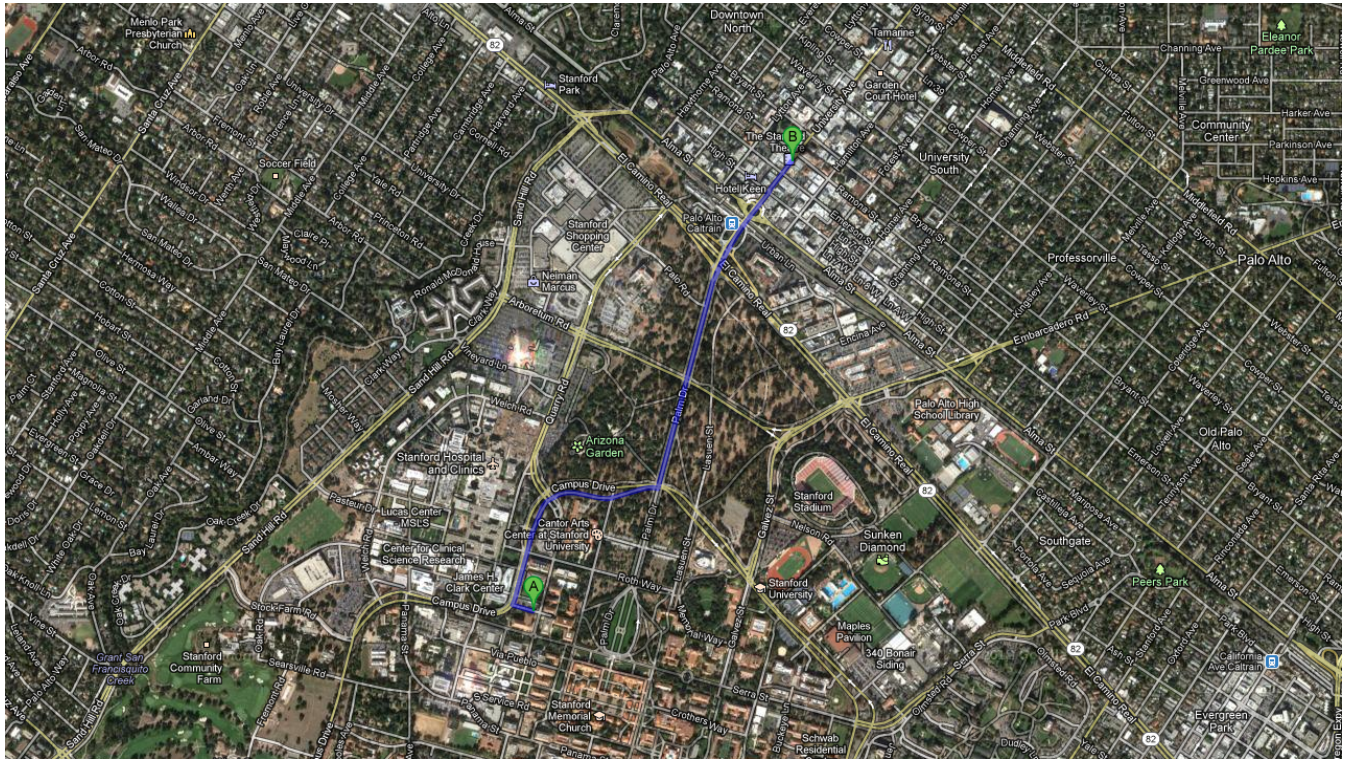
- When you solve this problem, try to think about how you did it. You probably simulated the scenario in your head, trying to send the farmer over with the goat and observing the consequences. If nothing got eaten, you might continue with the next action. Otherwise, you undo that move and try something 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.

Course plan


Search problems
Markov decision processes
Constraint satisfaction problems
Adversarial games
Bayesian networks
Reflex
States
Variables
Logic
"Low-level intelligence"
"High-level intelligence"
Machine learning

- So far, we have worked with only the simplest types of models — reflex models. We used these as a starting point to explore machine learning. Now we will proceed to the first type of state-based models, search problems.

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).

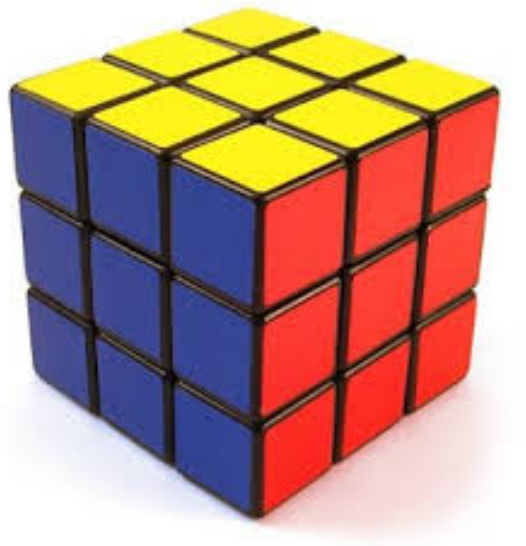
Application: robot motion planning

Objective: fastest? most energy efficient? safest?

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: generate fluent English sentence with same 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.

Anticipating future costs

Search problem (state-based models):

Key: need to consider future consequences of an action!

- While reflex-based models were appropriate for some applications, 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.

Paradigm

Modeling

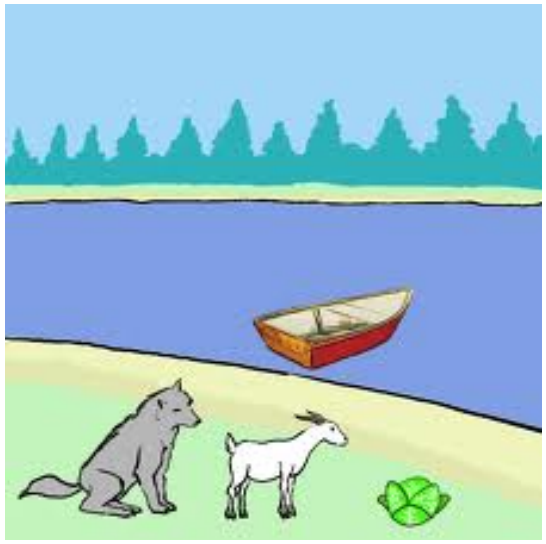
Inference **Learning**

- Recall the modeling-inference-learning paradigm. Today and Thursday, we will focus on the modeling and inference part of search problems. Next Tuesday, we will cover learning.

Roadmap

Tree search

Dynamic programming



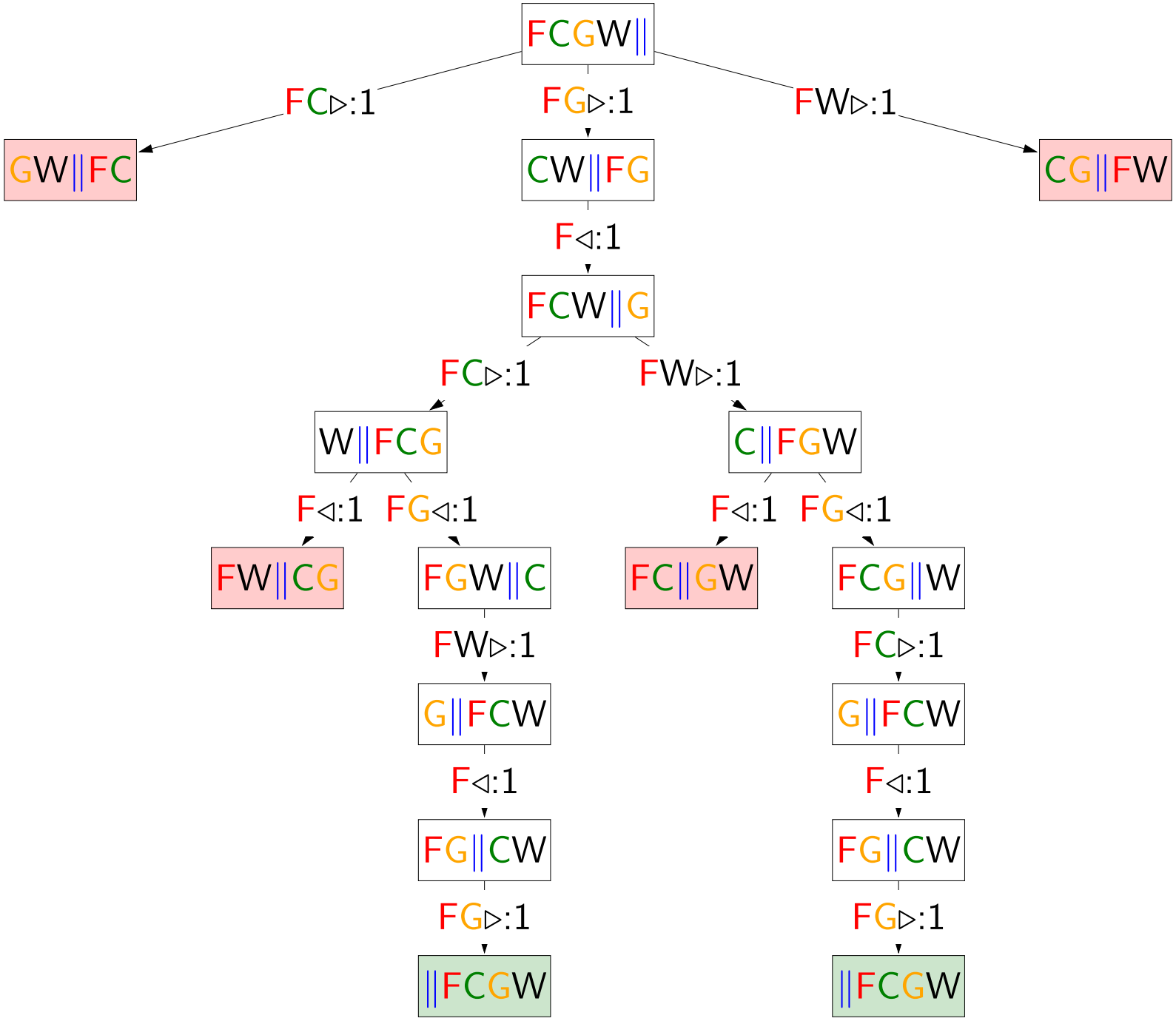
Farmer Cabbage Goat Wolf

Actions:

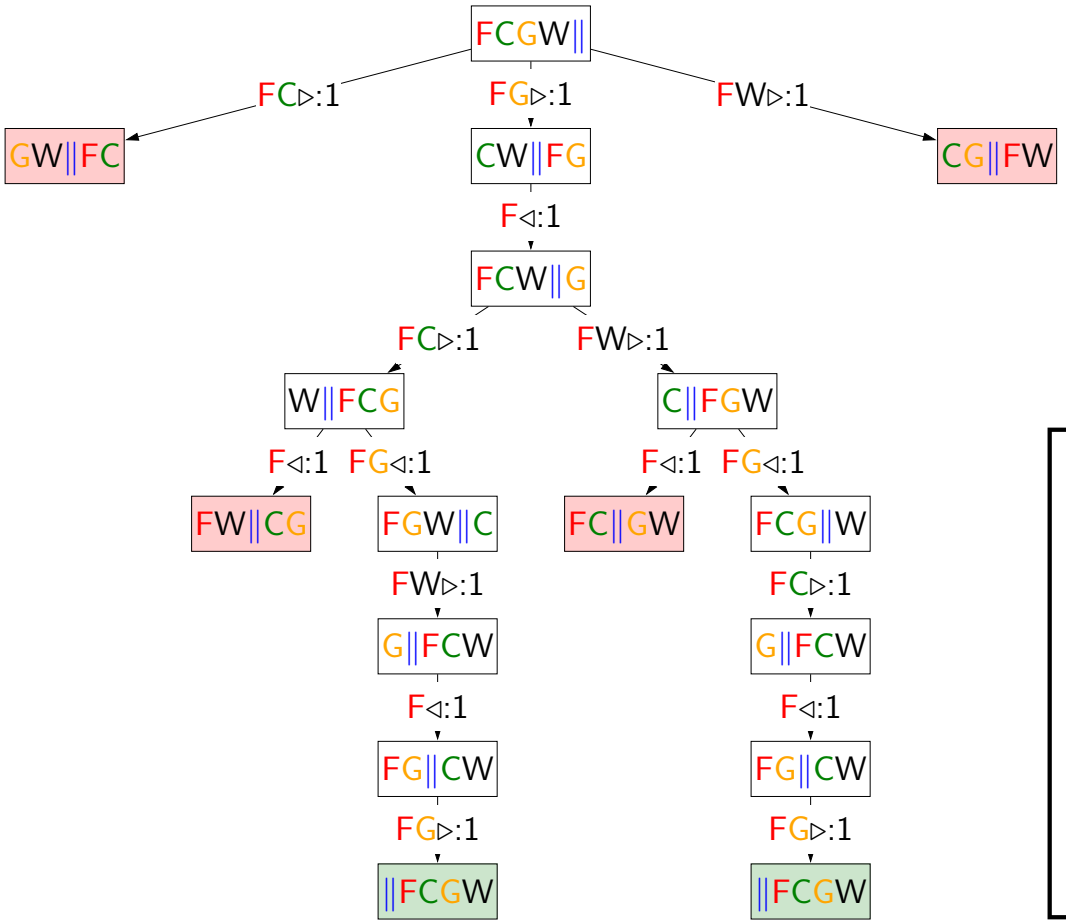
F▷	F◁
FC▷	FC◁
FG▷	FG◁
FW▷	FW◁

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▷:1
FG▷:1
FW▷:1
GW||FC
CW||FG
CG||FW
F◁:1
FCW||G
FC▷:1
FW▷:1
W||FCG
C||FGW
F◁:1
FG◁:1
F◁:1
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
- IsEnd(s): reached end state?

- 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 (IsEnd(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 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.

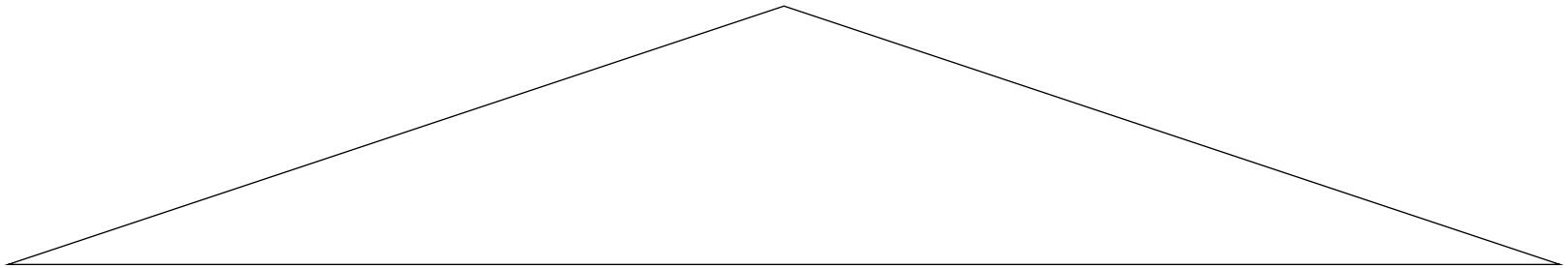
Taking a magic tram from s to $2s$ takes 2 minutes.

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

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

    If IsEnd(s): return empty path

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

        Call backtrackingSearch(Succ(s, a))

        Add Cost(s, a) to cost of path found in recursive call

    Return minimum cost path

[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 $1, 2, \ldots$.

  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 $1, 2, 3, \ldots$ 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 ≥ 0	$O(b^d)$	$O(b^d)$
DFS-ID	constant ≥ 0	$O(d)$	$O(b^d)$
Backtracking	any	$O(D)$	$O(b^D)$

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

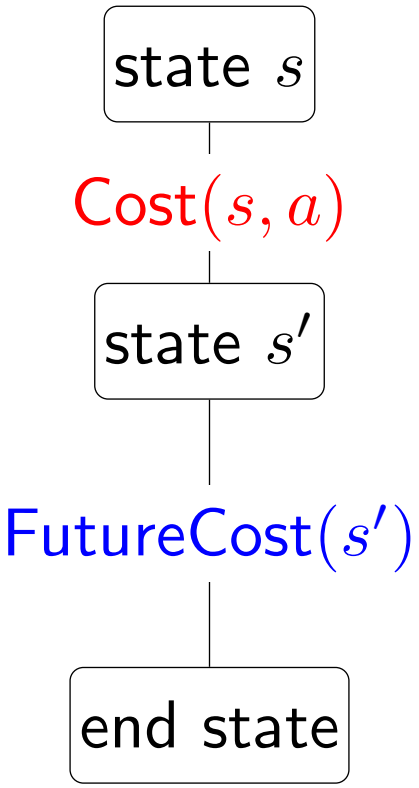
- Here is a summary of all the tree search algorithms, the assumptions on the action costs, and the space and time complexities.
- The take-away is that we can't avoid the exponential time complexity, but we can certainly have linear space complexity. Space is in some sense the more critical dimension 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

Future cost



Minimum cost path from state s to a end state:

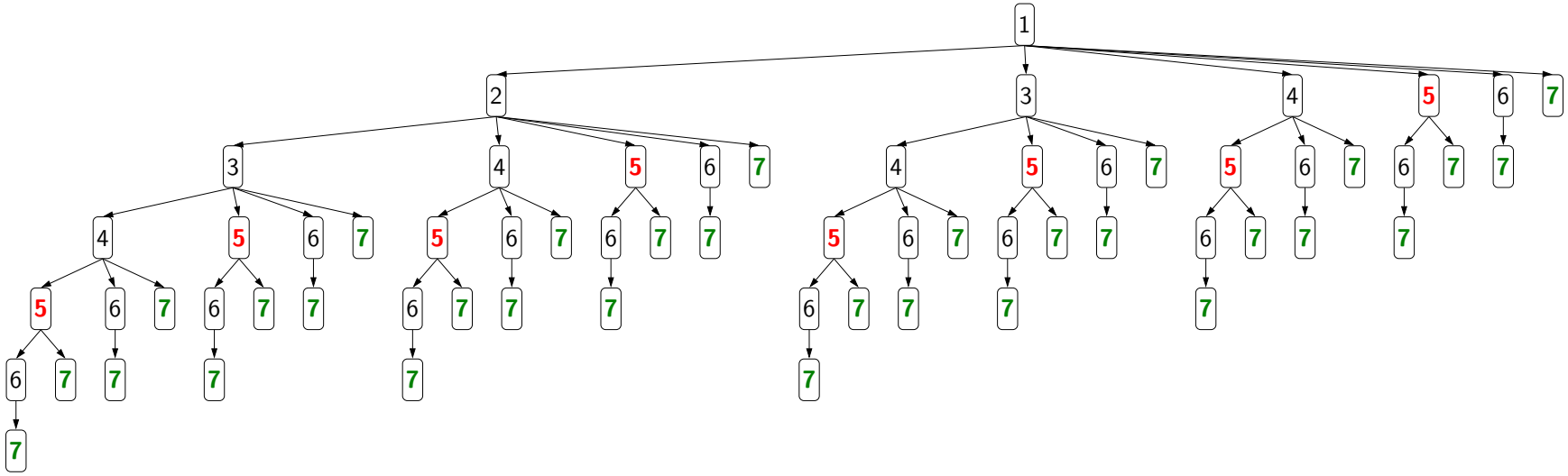
$$\text{FutureCost}(s) = \begin{cases} 0 & \text{if IsEnd}(s) \\ \min_{a \in \text{Actions}(s)} [\text{Cost}(s, a) + \text{FutureCost}(\text{Succ}(s, a))] & \text{otherwise} \end{cases}$$

- Now let's see if we can avoid the exponential running time of tree search with dynamic programming. 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 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 a 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 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.

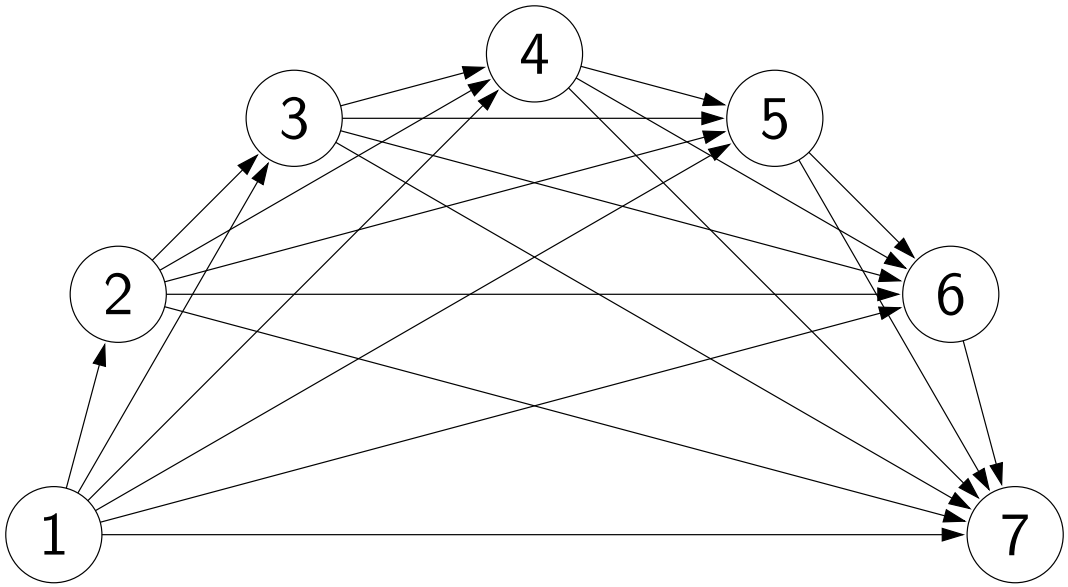


Observation: future costs only depend on current city

- 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: summary of past needed to make optimal decision



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 $a \in$ Actions(s): ...

[live solution]

Trade-off: Can be much faster, but needs $O(N)$ memory (N is number of states)

Acyclicity

Assumption: acyclicity

The state graph defined by Actions(s) and Succ(s, a) is acyclic.

- Backtracking search, DP: infinite loop!

- DFS and BFS: remember which states you visited already, don't revisit them

- 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.

Next time

- What to do if there are cycles and variable edge costs?

- What about complex constraints? (can take tram at most twice)

Summary

- State: summary of past actions sufficient to choose future actions optimally
- Tree search: memory efficient, suitable for huge state spaces but exponential worst-case running time
- Dynamic programming: backtracking search with **memoization** — potentially exponential savings

- We started out with the idea of a search problem, an abstraction that provides a clean interface between modeling and algorithms. Central to this is the concept of a **state**, which contains all information about the past needed to make optimal decisions in the future.
- We then looked at algorithms to find minimum-cost paths to an end state. 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.
- Finally, we saw how to improve these algorithms with some smart bookkeeping. This led us to dynamic programming, which handles arbitrary action costs, but assumes the problem is acyclic.