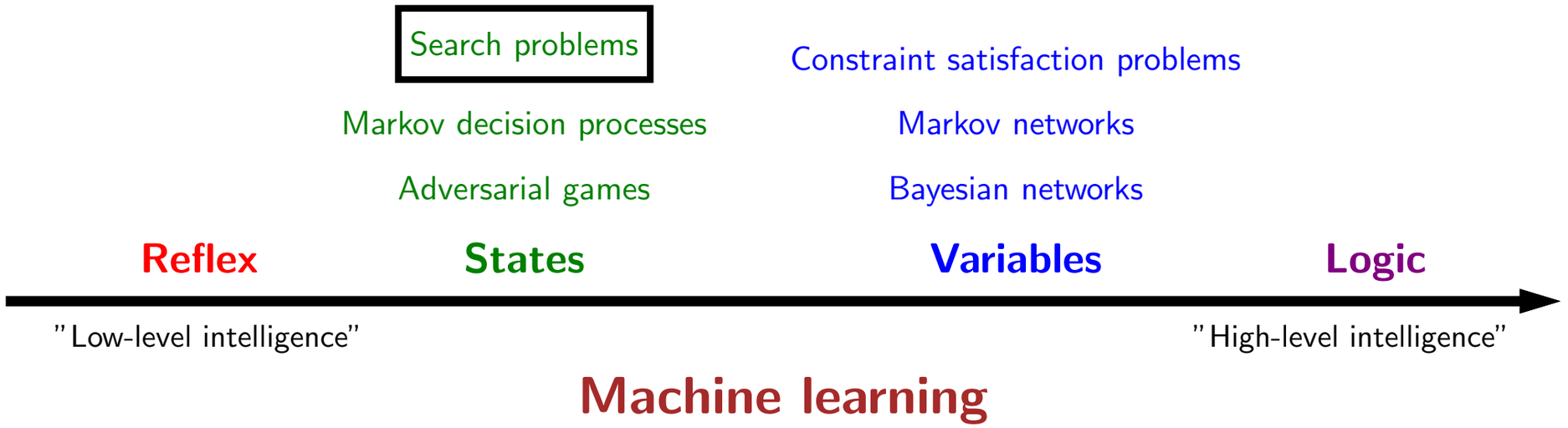
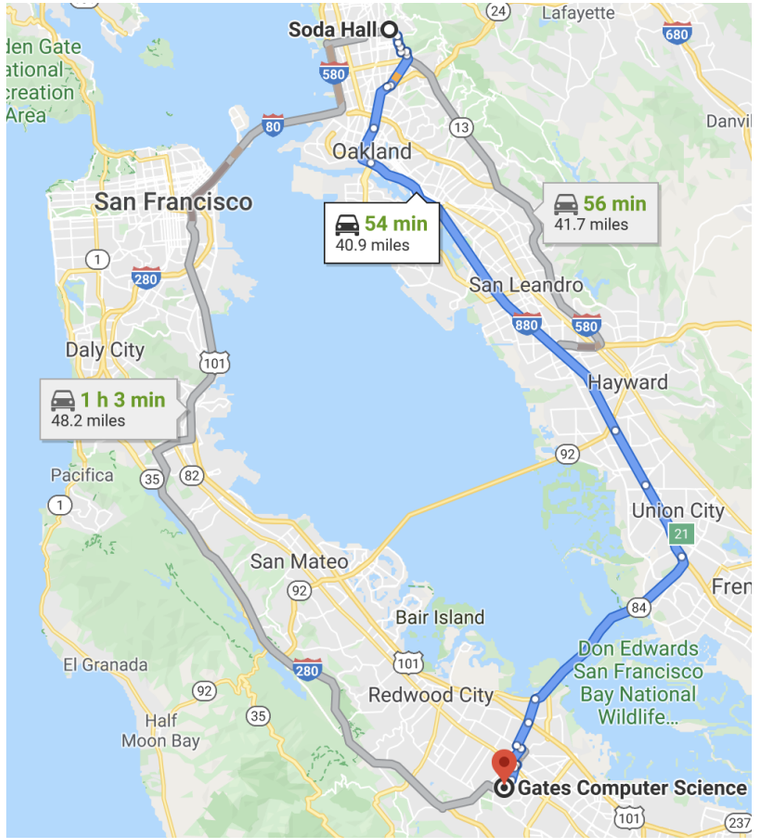
Search: basics and dynamic programming

Course plan


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

Actions: acceleration and throttle

- In robot motion planning, the goal is get a robot to move from one position/pose to another. Some of the most popular search algorithms like A star are developed for some of the first intelligent robots (Shakey 1983)

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, energy, safety, or expressiveness.

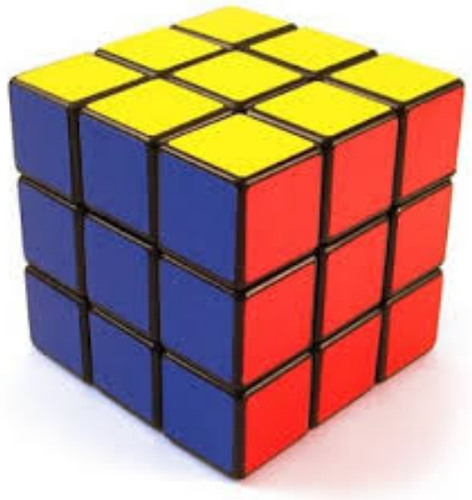
Application: multi-robot systems

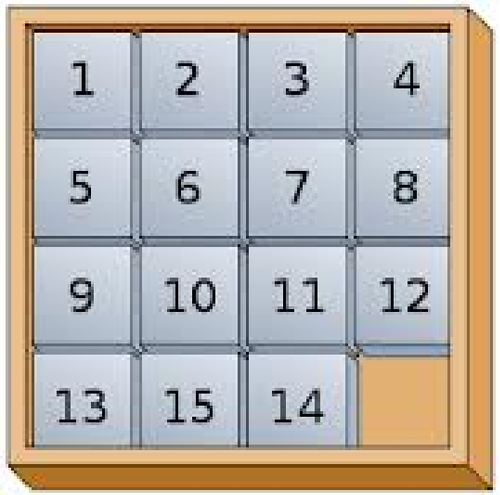
Objective: fastest? most energy efficient?

Actions: acceleration and steering of all robots

- Instead of planning for one agent, we can plan for a fleet of agents. For example, a group of robots need to coordinate in a warehouse to move objects from one shelf to another.

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.

Paradigm

Modeling

Inference **Learning**

- Recall the modeling-inference-learning paradigm. For reflex-based classifiers, modeling consisted of choosing the features and the neural network architecture; inference was trivial forward computation of the output given the input; and learning involved using stochastic gradient descent on the gradient of the loss function, which might involve backpropagation.
- Today, we will focus on the modeling and inference part of search problems.

Roadmap

Modeling

Modeling Search Problems

Learning

Structured Perceptron

Algorithms

Tree Search

Dynamic Programming

Uniform Cost Search

Programming and Correctness of UCS

A*

A* Relaxations

- Here are the rest of the modules under the search unit.
- We will start by talking about how we can define search problems, We will define states, successor, and cost functions.
- Next, we will introduce different types of search problems, starting with backtracking search and following with some of the other search algorithms such as DFS, BFS, and DFS-ID.
- Then we will continue discussing dynamic programming and uniform cost search. We will prove that uniform cost search is correct. Then we introduce the A* algorithm which tries to speed up UCS. Through this we introduce the concept of heuristics and their properties such as consistency, efficiency, and admissibility. Finally, we go over the idea of relaxations and how that allows us to come up with heuristics.
- While we will not cover learning cost functions, you are welcome to study them yourself. Notable algorithms include the structured perceptron algorithm.

Roadmap

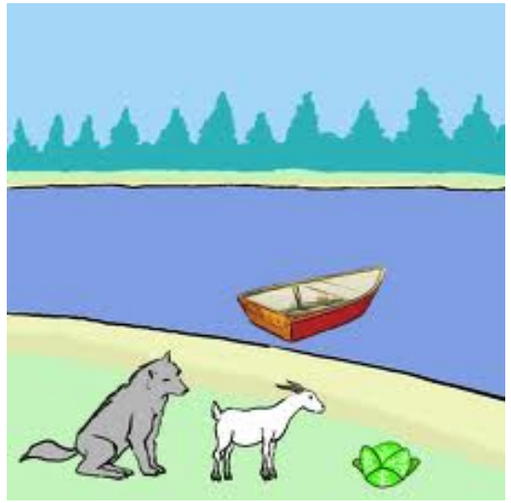
Topics in the lecture:

Modeling (c3.1-3)

Tree search (c3.4)

Dynamic programming

Uniform cost search (c3.4)



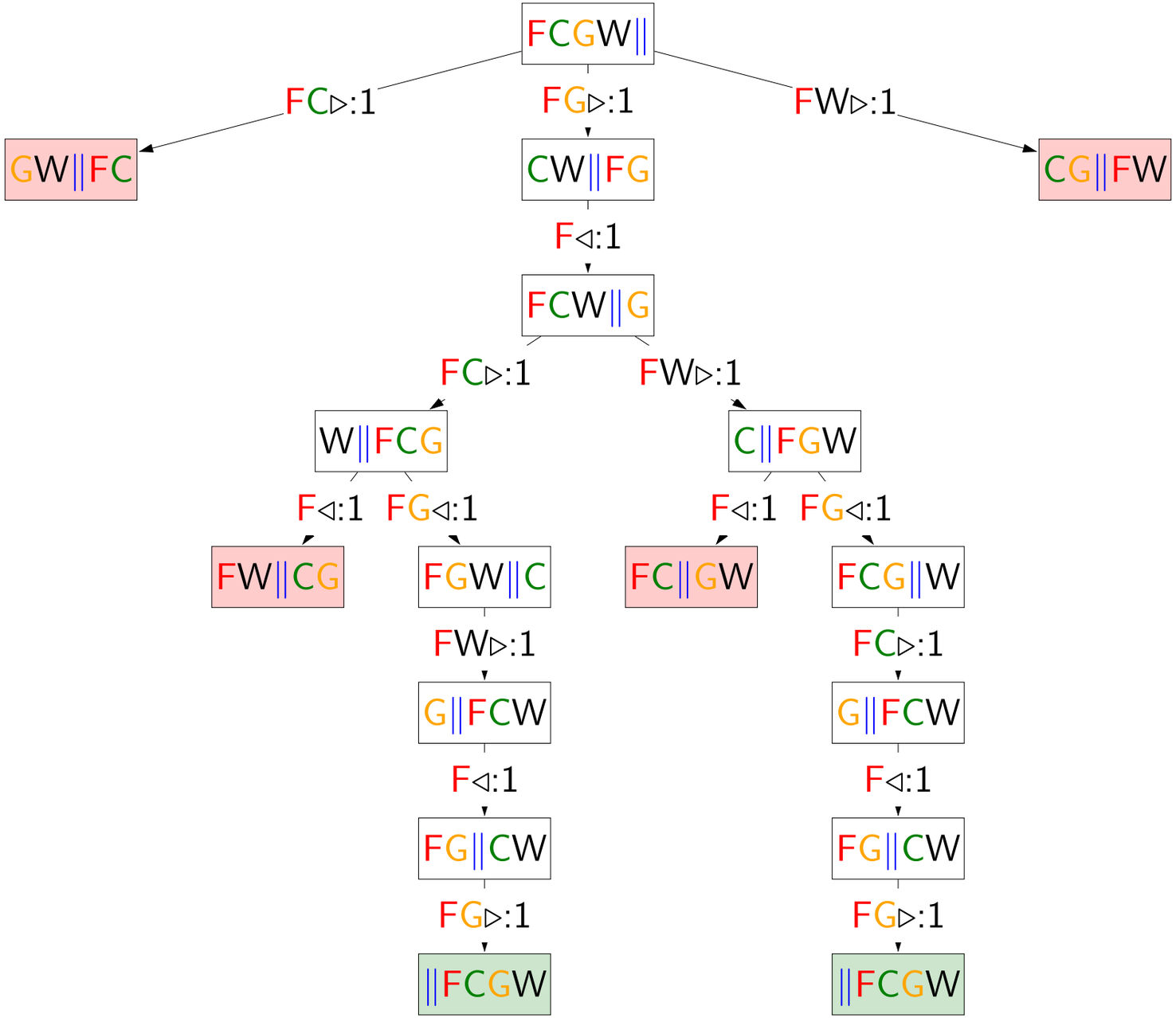
Farmer Cabbage Goat Wolf

Actions:

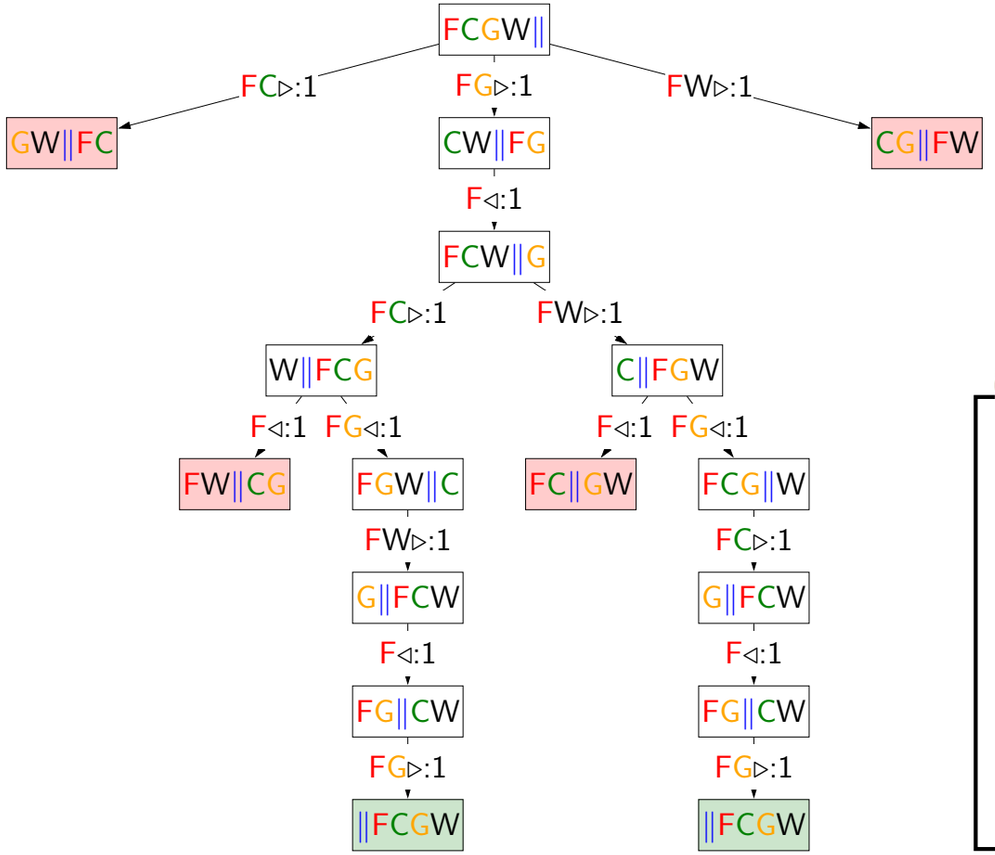
F▷	F◁
FC▷	FC◁
FG▷	FG◁
FW▷	FW◁

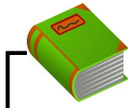
Approach: build a **search tree** ("what if?")

- 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?

What are the states, actions, goals, and costs?

- 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.
- What are the states? The block that you are on is a natural state
- What about actions? Theres two things we can do, walk or take the tram
- Goals? We say, how do we travel from 1 to n and this means getting to state n is the goal
- Costs? Walking costs 1 and tram costs 2.

Roadmap

Topics in the lecture:

Modeling (c3.1-3)

Tree search (c3.4)

Dynamic programming

Uniform cost search (c3.4)

Backtracking / exhaustive search

Algorithm: backtracking search

def backtrackingSearch(s):

    If IsEnd(s): return 0

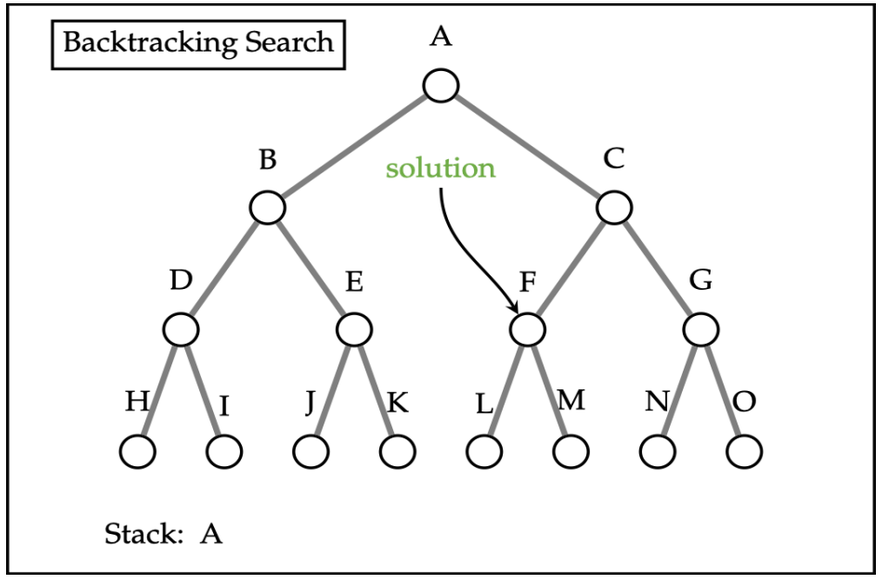
    If Actions(s) is empty: return ∞

    Set CostList $= []$

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

        Add backtrackingSearch(Succ(s, a))+Cost(s, a) to CostList

    Return minimum cost in CostList



- 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.

Backtracking / Exhaustive search

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

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

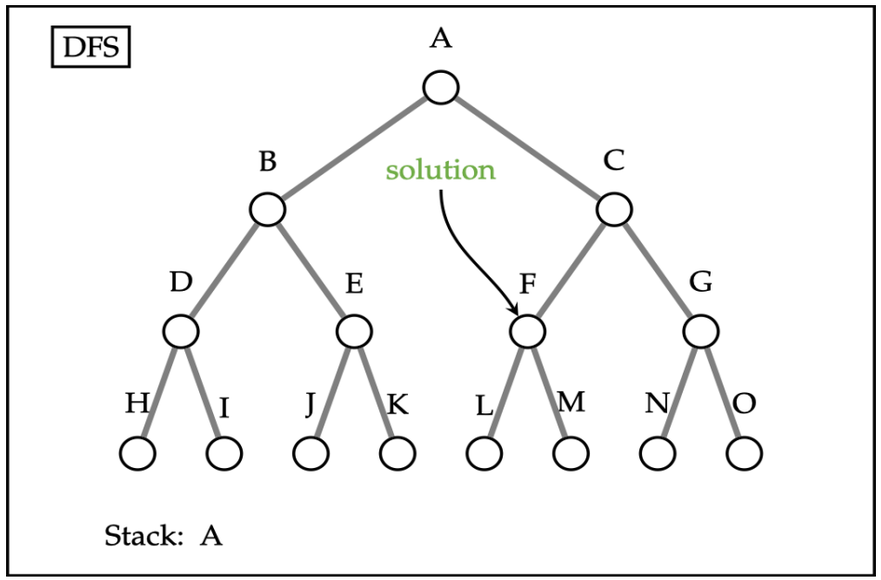
- What is the time and memory complexity of this algorithm?
- To get a simple characterization, assume that the search tree has maximum depth D (each path consists of D actions/edges) and that there are b available actions per state (the **branching factor** is b).
- It is easy to see that backtracking 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?

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.



- 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.

Depth-first search

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

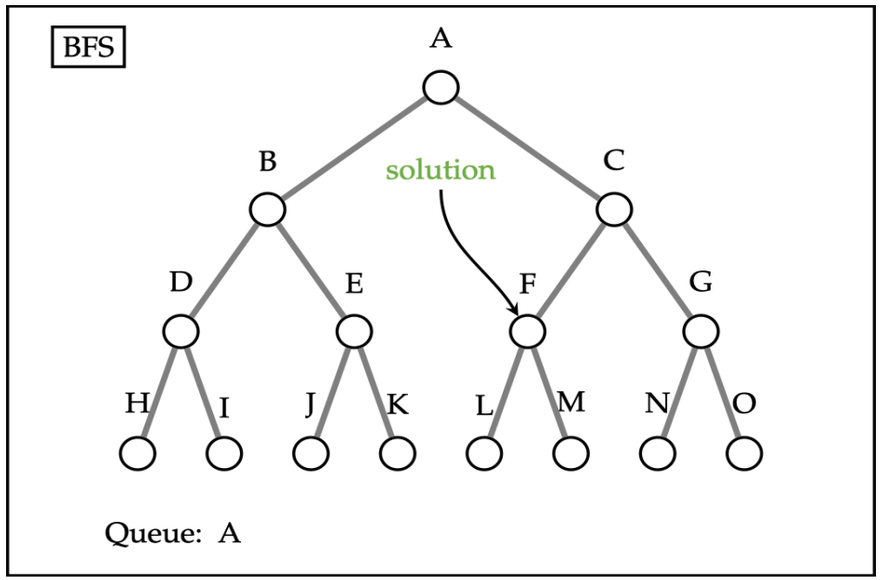
- The worst time and space complexity are of the same order as backtracking search. In particular, if there is no path to an end state, then we have to search the entire tree.
- However, if there are many ways to reach the end state, then we can stop much earlier without exhausting the search tree. So 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.



- **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.

Breadth-first search

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)

- 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?

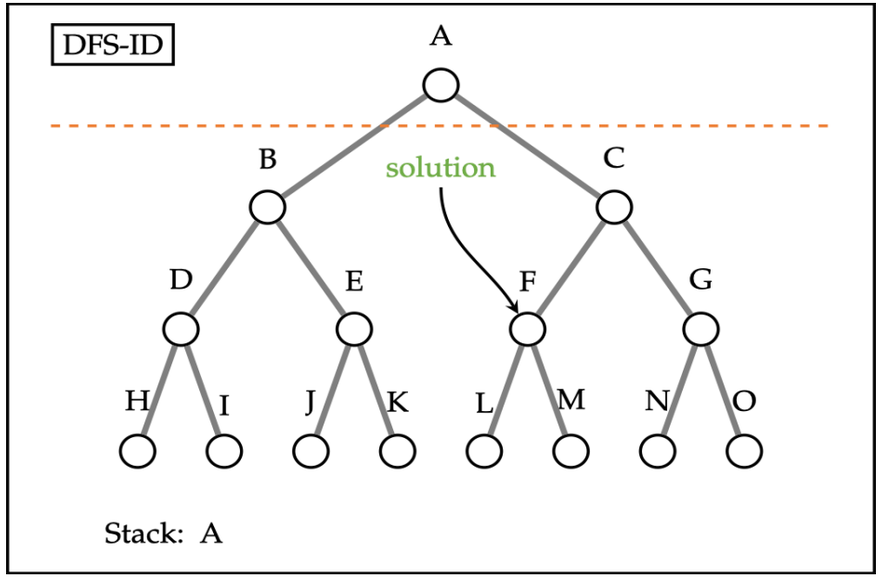
DFS with iterative deepening

Assumption: constant action costs

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

Idea: let DFS stop at a maximum depth; call DFS for maximum depths $1, 2, \ldots$.

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



- 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).

DFS with iterative deepening

Legend: b actions per state, solution size d

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

- 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

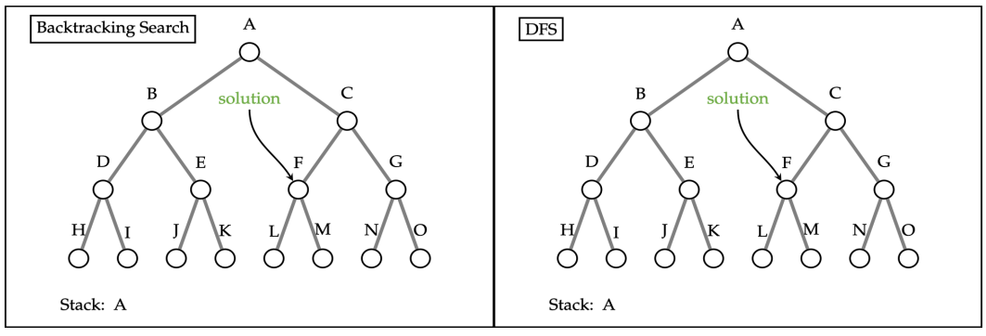
Legend: b actions/state, solution depth d, maximum depth D

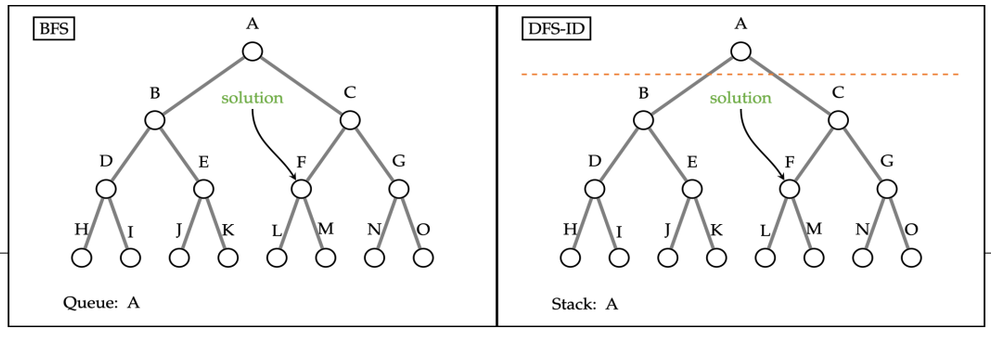
Algorithm	**Action costs**	**Space**	**Time**
Backtracking	any	$O(D)$	$O(b^D)$
DFS	zero	$O(D)$	$O(b^D)$
BFS	constant ≥ 0	$O(b^d)$	$O(b^d)$
DFS-ID	constant ≥ 0	$O(d)$	$O(b^d)$

- 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).

Tree Search Review





Roadmap

Topics in the lecture:

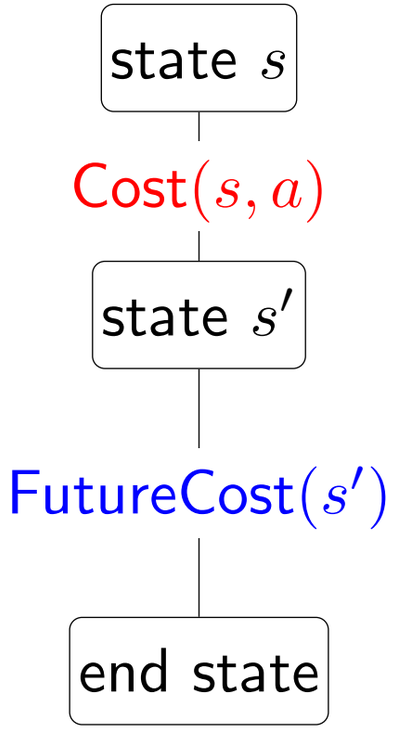
Modeling (c3.1-3)

Tree search (c3.4)

Dynamic programming

Uniform cost search (c3.4)

Dynamic programming



Minimum cost path from state s to a end state:

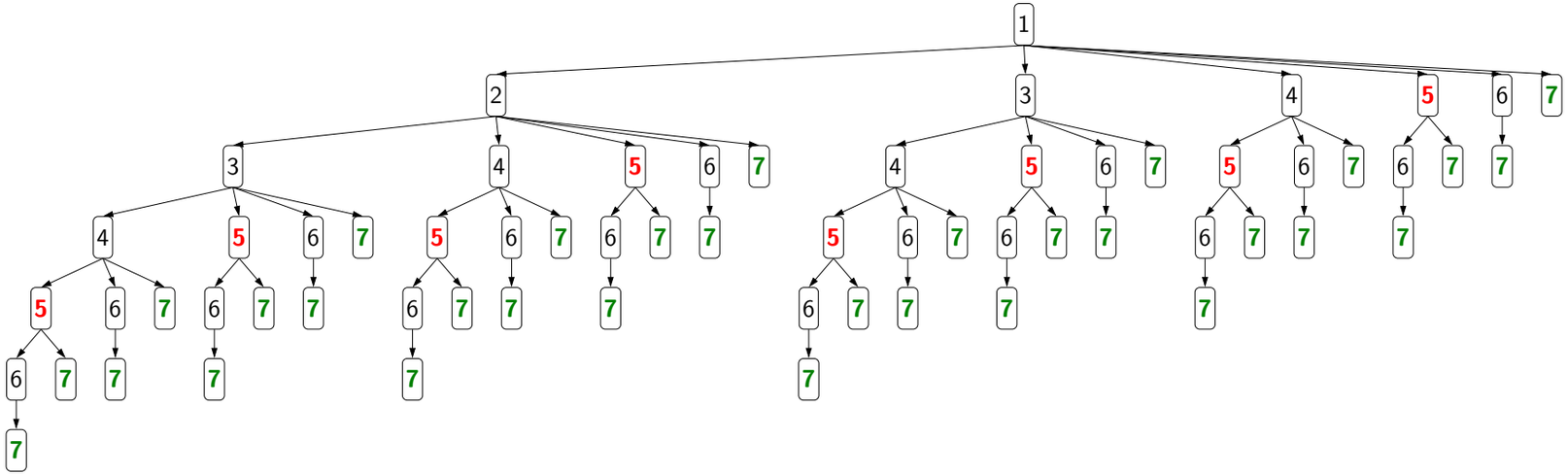
$$\text{FutureCost}(s) = \begin{cases} 0 & \text{if IsEnd}(s) \\ \min_{a \in \text{Actions}(s)} [\text{Cost}(s, a) + \text{FutureCost}(\text{Succ}(s, a))] & \text{otherwise} \end{cases}$$

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

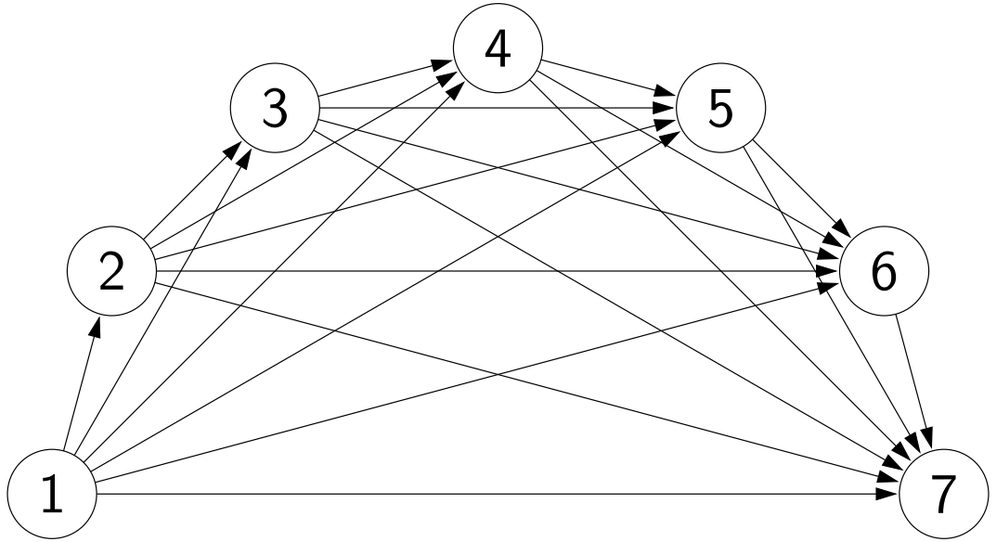


Observation: future costs only depend on current city

- 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**.
- You may have already seen dynamic programming earlier in your CS lives. 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.

Dynamic programming

Algorithm: search with dynamic programming

def DynamicProgrammingSearch(s):

    If already computed for s, return cached answer.

    If IsEnd(s): return solution

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

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 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.

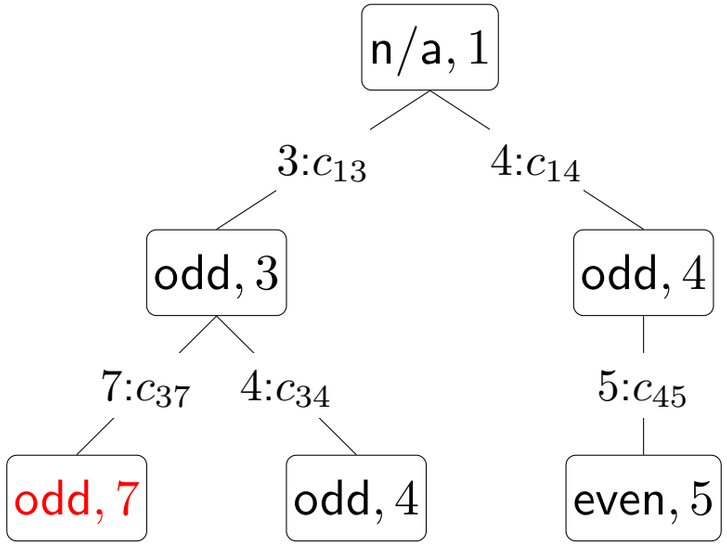
Handling additional constraints

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)

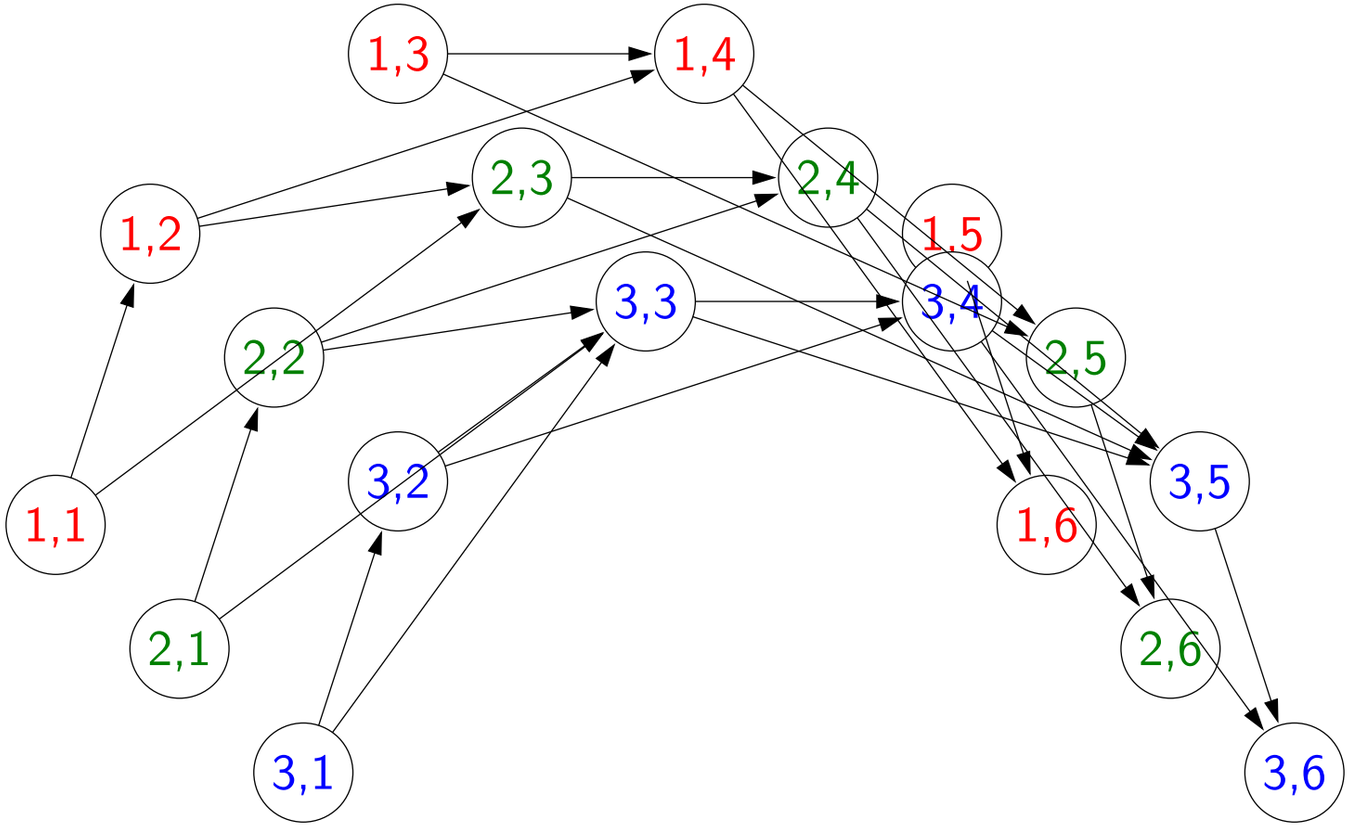


- Let's add a constraint that says we can't visit three odd cities in a row. If we only keep track of the current city, and we try to move to a next city, we cannot enforce this constraint because we don't know what the previous city was. So let's add the previous city into the state.
- This will work, 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.

State graph

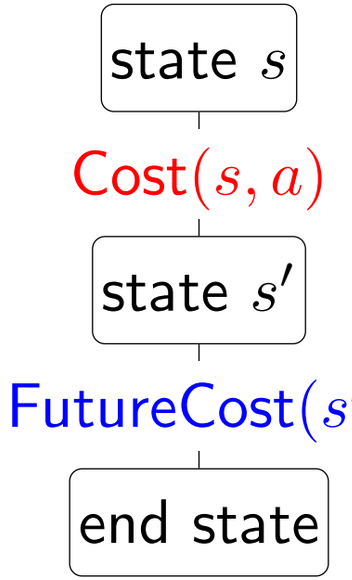
Whats the minimal state if we visit 3 odd cities when going from city 1 to n?

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.

Summary and Review



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

Key idea: state

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

Roadmap

Topics in the lecture:

Modeling (c3.1-3)

Tree search (c3.4)

Dynamic programming

Uniform cost search (c3.4)

From depth to breadth first search

Our earlier DP algorithm: improved exhaustive search

- Go down the tree by taking actions
- Use FutureCost to re-use computation

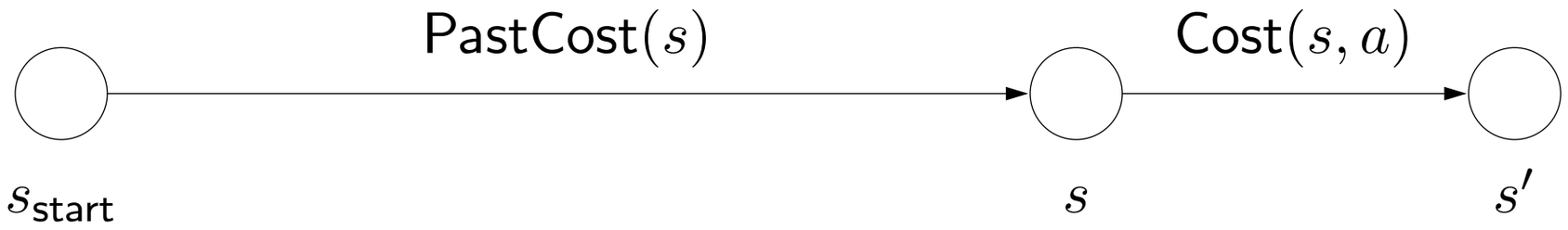
Up next: improved breadth first search

- Expand states close to the start (breadth)
- Use PastCost to re-use computation

- Our dynamic programming solution from earlier was an improvement to exhaustive and depth first search
- We would go down the tree and take actions, adding up costs as we went
- Dynamic programming was used explicitly to re-use computations for subtrees
- Now we want to consider an alternative based on breadth first search
- The idea is going to be similar - we are going to go shallow to deep
- We are going to try to take actions to explore states with lowest cost and work outwards
- Expanding states in order of increasing cost seems almost as hard as our end goal of finding the minimum cost path
- The key thing is that we can do this efficiently using what we will call PastCost that Ill define next

Ordering the states

Observation: prefixes of optimal path are optimal



Key: PastCost(s) is the optimal cost to get to state s and this follows the same dynamic programming recurrence from earlier

- 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 PastCost(s), the cost of the minimum cost path from the start state to s.
- Is there an efficient way to make use of PastCost in our search algorithm the same way we used FutureCost?

Uniform cost search (UCS)

Key idea: state ordering

UCS enumerates states in order of increasing past cost.

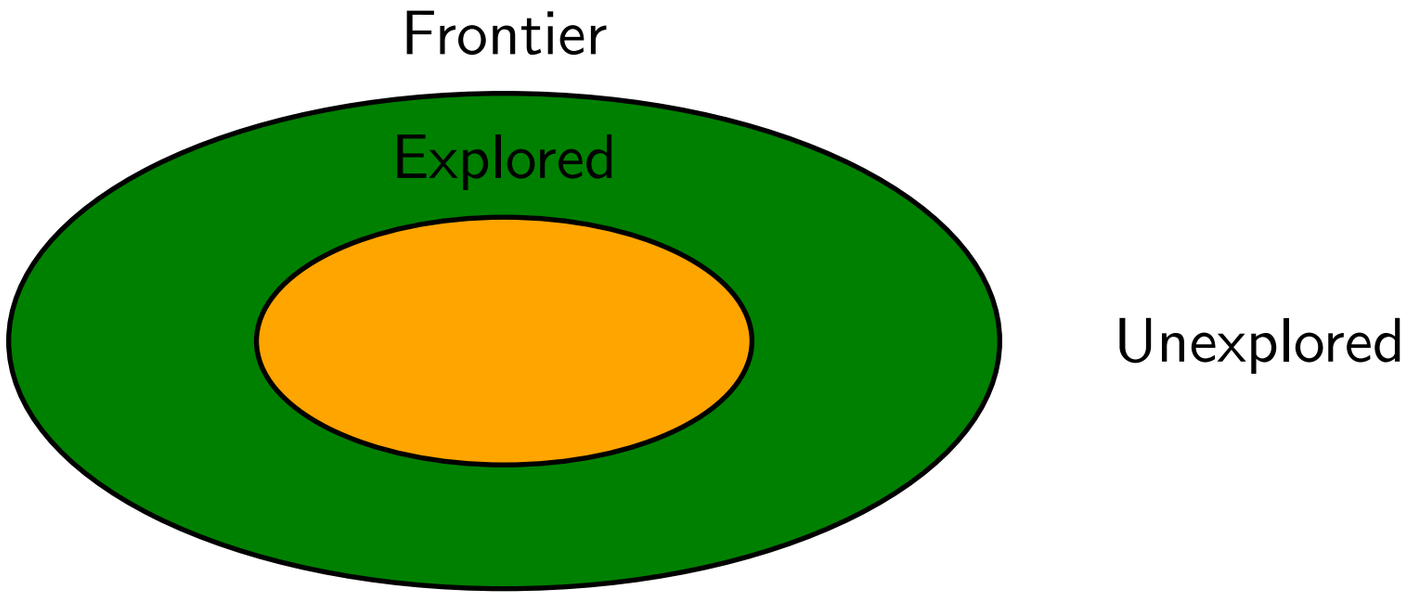
Assumption: non-negativity

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.

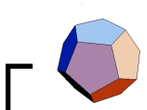
High-level strategy



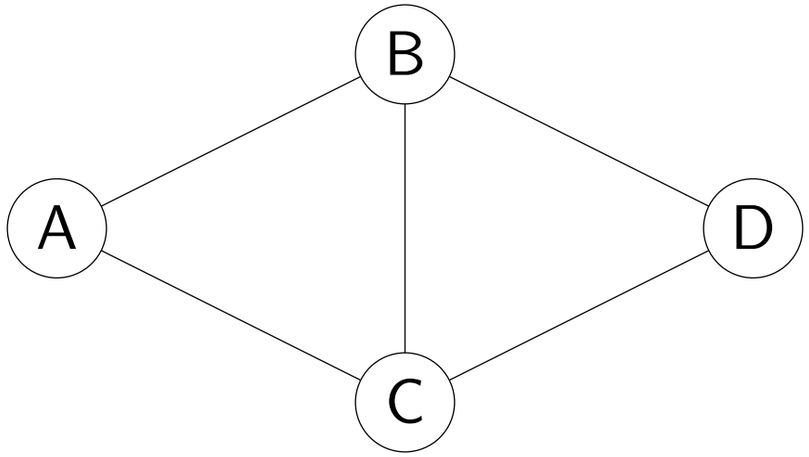
- Explored: states we've found the optimal path to
- Frontier: states we've seen, still figuring out how to get there cheaply
- Unexplored: states we haven't seen

- 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



Example: UCS example



Start state: A, end state: D

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)

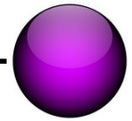
Algorithm: uniform cost search [Dijkstra, 1956]

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

Repeat until frontier is empty:

- Remove s with smallest priority p from frontier
- If IsEnd(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)

Analysis of uniform cost search



Theorem: correctness

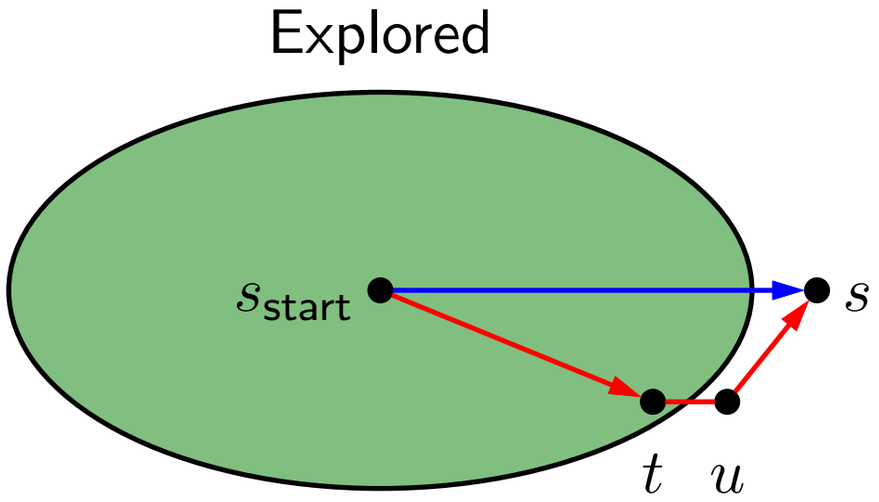
When a state s is popped from the frontier and moved to explored, its priority is PastCost(s), the minimum cost to s.

Proof:

Easy part: priority value p_s must be a valid path cost

Harder part: priority value p_s is the minimum path cost

Can there be a path start $\rightarrow t \rightarrow u \rightarrow s$ with lower cost?



Use Induction: priority value is PastCost in explored set

Inductive step: PastCost(t) + Cost$(t, a) = p_t$ + Cost(t, a)

Since u is in the frontier, p_t + Cost$(t, a) \geq p_u$

Since p_s is top of the queue, $p_u \geq p_s$

- 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: 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.