• 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.

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

In robot motion planning, the goal is to 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.

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: route finding

Objective: shortest? fastest? most scenic?
Actions: go straight, turn left, turn right

Application: robot motion planning

Objective: fastest? most energy efficient? safest?
Actions: translate and rotate joints

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)

Beyond reflex

Classifier (reflex-based models):

\[
\begin{align*}
\hat{x} & \rightarrow f & \text{single action } y \in \{-1, +1\}
\end{align*}
\]

Search problem (state-based models):

\[
\begin{align*}
\hat{x} & \rightarrow f & \text{action sequence } (a_1, a_2, a_3, a_4, \ldots)
\end{align*}
\]

Key: need to consider future consequences of an action!

Paradigm

Modeling

Inference

Learning

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.
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. The next lecture will cover learning.

Farmers, cabbage, goat, wolf

**Actions:**
- \( F \rightarrow \leftarrow \)
- \( FC \rightarrow \leftarrow \)
- \( FG \rightarrow \leftarrow \)
- \( FW \rightarrow \leftarrow \)

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

**Roadmap**
- Tree search
- Dynamic programming
- Uniform cost search

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 \( FG \rightarrow \) means that the farmer will take the goat across to the right bank; \( F \leftarrow \) means that the farmer is coming back to the left bank alone.

**Search problem**

- 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 as $Cost(s, a)$. 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.

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.

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

```python
def backtrackingSearch(s, path):
    if isEnd(s):
        update minimum cost path
        return minimum cost path
    for each action $a$ in Actions(s):
        Extend path with $Succ(s, a)$ and $Cost(s, a)$
        Call backtrackingSearch($Succ(s, a)$, path)
        Return minimum cost path
```

[semi-live solution: backtrackingSearch]
Depth-first search

**Assumption: zero action costs**

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

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

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

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

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 $D$)

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)

- 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 (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?

- 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 $D$

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Action costs</th>
<th>Space</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFS</td>
<td>zero</td>
<td>$O(D)$</td>
<td>$O(bD)$</td>
</tr>
<tr>
<td>BFS</td>
<td>constant $\geq 0$</td>
<td>$O(b^d)$</td>
<td>$O(b^d)$</td>
</tr>
<tr>
<td>DFS-ID</td>
<td>constant $\geq 0$</td>
<td>$O(d)$</td>
<td>$O(b^d)$</td>
</tr>
<tr>
<td>Backtracking</td>
<td>any</td>
<td>$O(D)$</td>
<td>$O(b^D)$</td>
</tr>
</tbody>
</table>

- 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).

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).

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.

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.

Algorithm: dynamic programming

```python
def DynamicProgramming(s):
    If already computed for s, return cached answer.
    If IsEnd(s): return solution
    For each action $a \in \text{Actions}(s)$:
        [semi-live solution: Dynamic Programming]
```

Assumption: acyclicity

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

Key idea: state

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

<table>
<thead>
<tr>
<th>past actions (all cities)</th>
<th>1 3 4 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>state (current city)</td>
<td>1 3 4 6</td>
</tr>
</tbody>
</table>
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 whether the previous city was an odd numbered city to enforce this constraint.

Note that in doing so, we have \( 2n \) states rather than \( n^2 \) states, which is a substantial savings. So the lesson is to pay attention to what information you actually need in the state.

Question

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

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 (\( \min(\text{number of odd cities visited}, 3) \), current city). Thus, our state space is \( O(n) \) rather than \( O(n^2) \).

We can visualize what augmenting the state does to the state graph. Effectively, we are copying each node 4 times, and the edges are redirected to move between these copies.

Note that some states such as \((2, 1)\) aren’t reachable (if you’re in city 1, it’s impossible to have visited 2 odd cities already); the algorithm will not touch those states and that’s perfectly okay.
Question
Objective: travel from city 1 to city $n$, visiting more odd than even cities. What is the minimal state?

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

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

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...

Roadmap
- **Tree search**
- **Dynamic programming**
- **Uniform cost search**

- 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.

- Recall that we used dynamic programming to compute the future cost of each state, 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 is always a clear order in which to compute all the past costs. If the past costs of all the predecessors of a state $s$ are computed, then we could compute the past cost of $s$ by taking the minimum.
- Note that 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: Cost(s, a) ≥ 0.

UCS in action:

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

Uniform cost search example

Example: UCS example

Start state: A, end state: D

Minimum cost path:
A → B → C → D with cost 3
### Uniform cost search (UCS)

**Algorithm: uniform cost search** [Dijkstra, 1956]

1. Add $s_{\text{start}}$ to frontier (priority queue)
2. Repeat until frontier is empty:
   - Remove $s$ with smallest priority $p$ from frontier
   - If IsEnd($s$): return solution
   - Add $s$ to explored
   1. For each action $a \in \text{Actions}(s)$:
      - Get successor $s' \leftarrow \text{Succ}(s, a)$
      - If $s'$ already in explored: continue
      - Update frontier with $s'$ and priority $p + \text{Cost}(s, a)$

[semi-live solution: Uniform Cost Search]

### 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:**

- 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 $p_u$.
- 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.
  1. First, by definition of 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 PastCost($t$) + Cost($t, a$) = $p_t + \text{Cost}(t, a)$, where the last equality is by the inductive hypothesis.
  2. Second, we have $p_t + \text{Cost}(t, a) \geq p_u$ since we updated the frontier based on $(t, a)$.
  3. 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

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Cycles?</th>
<th>Action costs</th>
<th>Time/space</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td>no</td>
<td>any</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>UCS</td>
<td>yes</td>
<td>$\geq 0$</td>
<td>$O(n \log n)$</td>
</tr>
</tbody>
</table>

**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 $N$)
Summary

- **Tree search**: memory efficient, suitable for huge state spaces but exponential worst-case running time

- **State**: summary of past actions sufficient to choose future actions optimally

- **Graph search**: dynamic programming and uniform cost search construct optimal paths (exponential savings!)

- **Next time**: learning action costs, searching faster with A*