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

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 |

**Course plan**

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

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.
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).

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.

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).

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.

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)

Application: machine translation

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.

• 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.

• 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.

Anticipating future costs

Search problem (state-based models):

\[ x \xrightarrow{f} \text{action sequence } (a_1, a_2, a_3, a_4, \ldots) \]

Key: need to consider future consequences of an action!

Paradigm

Modeling

Inference

Learning

Roadmap

Tree search

Dynamic programming

• 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.

• 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.
Farmer Cabbage Goat Wolf

Actions:
F ⊥
FC ⊥
FG ⊥
FW ⊥
F ◁
FC ◁
FG ◁
FW ◁

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

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?

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**

- **Algorithm:** backtracking search

```python
def backtrackingSearch(s):
    if isEnd(s):
        return empty path
    for each action a ∈ Actions(s):
        Add Cost(s, a) to cost of path found in recursive call
        Recursively search the successors of s.
        Return minimum cost path
```

**Depth-first search**

**Assumption:** zero action costs

Assume action costs 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)

---

**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>$O(b^D)$</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

---

*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.*
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 an end state) and an inductive case, which consists of taking the minimum over all possible actions a from s, taking an initial step resulting in an immediate action cost Cost(s, a) and a future cost.

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 k, 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.

Next time
• What to do if there are cycles and variable edge costs?
• What about complex constraints? (can take tram at most twice)

Acyclicity

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

• Backtracking search, DP: infinite loop!

• DFS and BFS: remember which states you visited already, don’t revisit them

Dynamic programming

Algorithm: dynamic programming

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

[Live solution]

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

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.