Search: modeling

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?

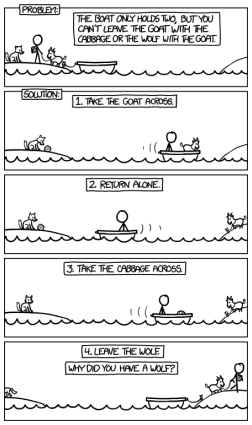
- 4

- 5

- 6

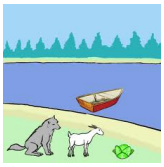
- 7

- 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.

- 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.



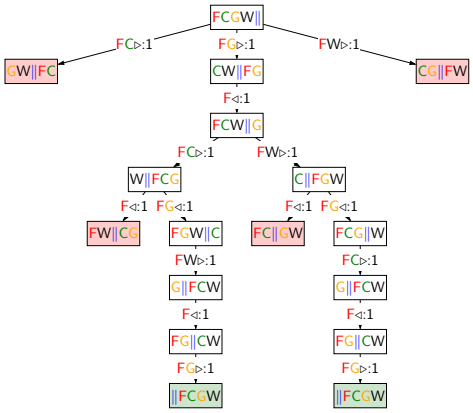
Farmer Cabbage Goat Wolf

Actions:

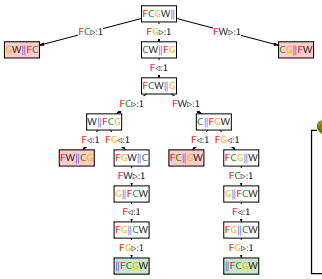
F▷	F◁
FC▷	FC◁
FG▷	FG◁
FW▷	FW◁

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▷ means that the farmer will take the goat across to the right bank; F◁ means that the farmer is coming back to the left bank alone.



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?

[semi-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`.