CE213 Artificial Intelligence – Lecture 2

Problem Solving by State Space Search

Basic idea for general problem solver:

"generate-and-evaluate"

How do we get a computer to solve problems?

How to represent the solution to a problem?

How to generate potential solutions?

How to evaluate them?

The Corn, Goose and Fox Problem

A farmer is taking a sack of corn, a goose, and a fox to market.
He has to cross a river under the following constraints:

There is a rowing boat but it is only large enough to carry the farmer (*rower*) *and any one of the three things he is taking.*

Unfortunately there are two problems:

The goose will eat the corn if the farmer is not present.

The fox will eat the goose if the farmer is not present.

(*The goose or fox will not run away even when the farmer is not with them* ☺)

How does the farmer get everything across the river?

Solving the Corn, Goose and Fox Problem

We begin by abstracting the **essentials** of the problem:

Clearly, any solution consists of a series of **moves / actions**, in which the farmer ferries either the corn, the goose, the fox, or just himself across the river.

So at any stage there are, at most, four things (moves) the farmer can do:

Take the corn across:	*c*
Take the goose across:	*g*
Take the fox across:	*f*
Go across alone:	*n*

We also need to describe the **situations** resulted from these moves:

- We need to know which animals are on which bank.

  In this problem we can **ignore** what happens while the boat is crossing, and just consider the situations before and after each move.

- We need to keep track of the positions of *four* entities: the corn, the goose, the fox, and the farmer.

  The farmer and the boat have to stay together.

  Each must be on the left bank (*L*) or the right bank (*R*).

So we can represent any situation by a 4-tuple (a state representation):

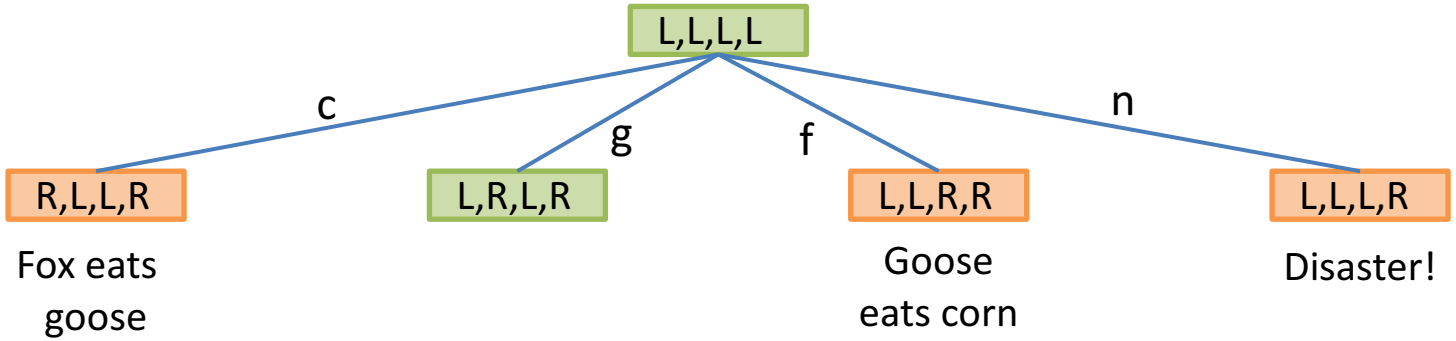
[*Corn place, Goose place, Fox place, Farmer place*]

e.g., [*L*,*L*,*L*,*R*] indicates that the corn, the goose and the fox are all on the left bank, but the farmer is on the right bank.

Searching for a solution

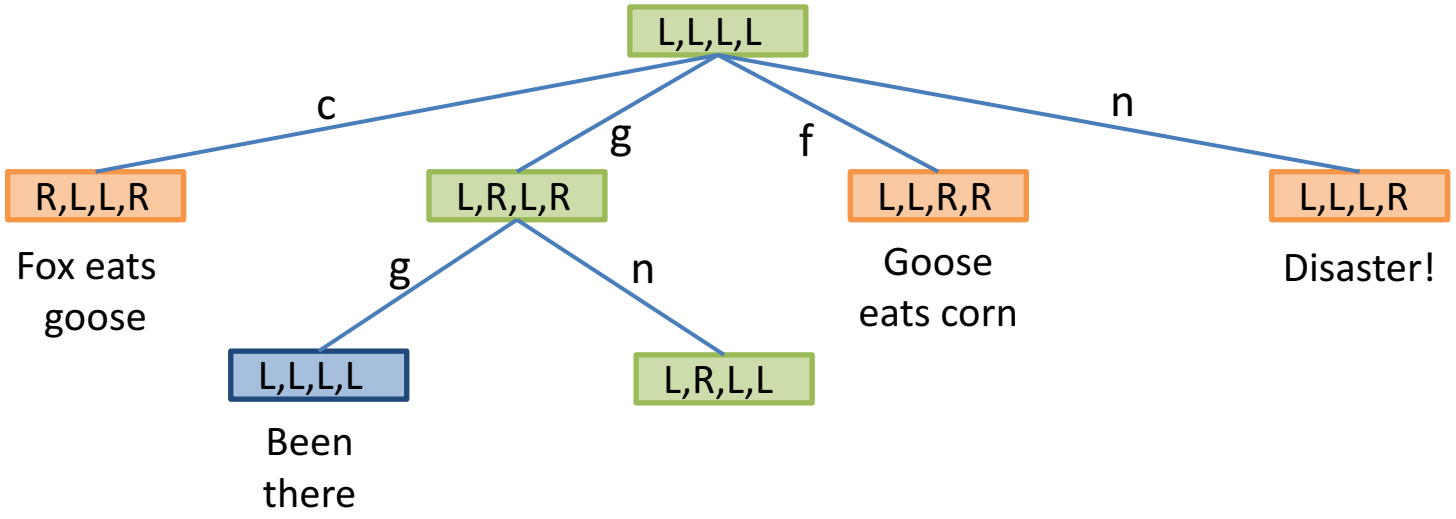
Initially, at [L,L,L,L] there are four possible moves:

Clearly, only one is a viable choice for the first move, that is *g*.



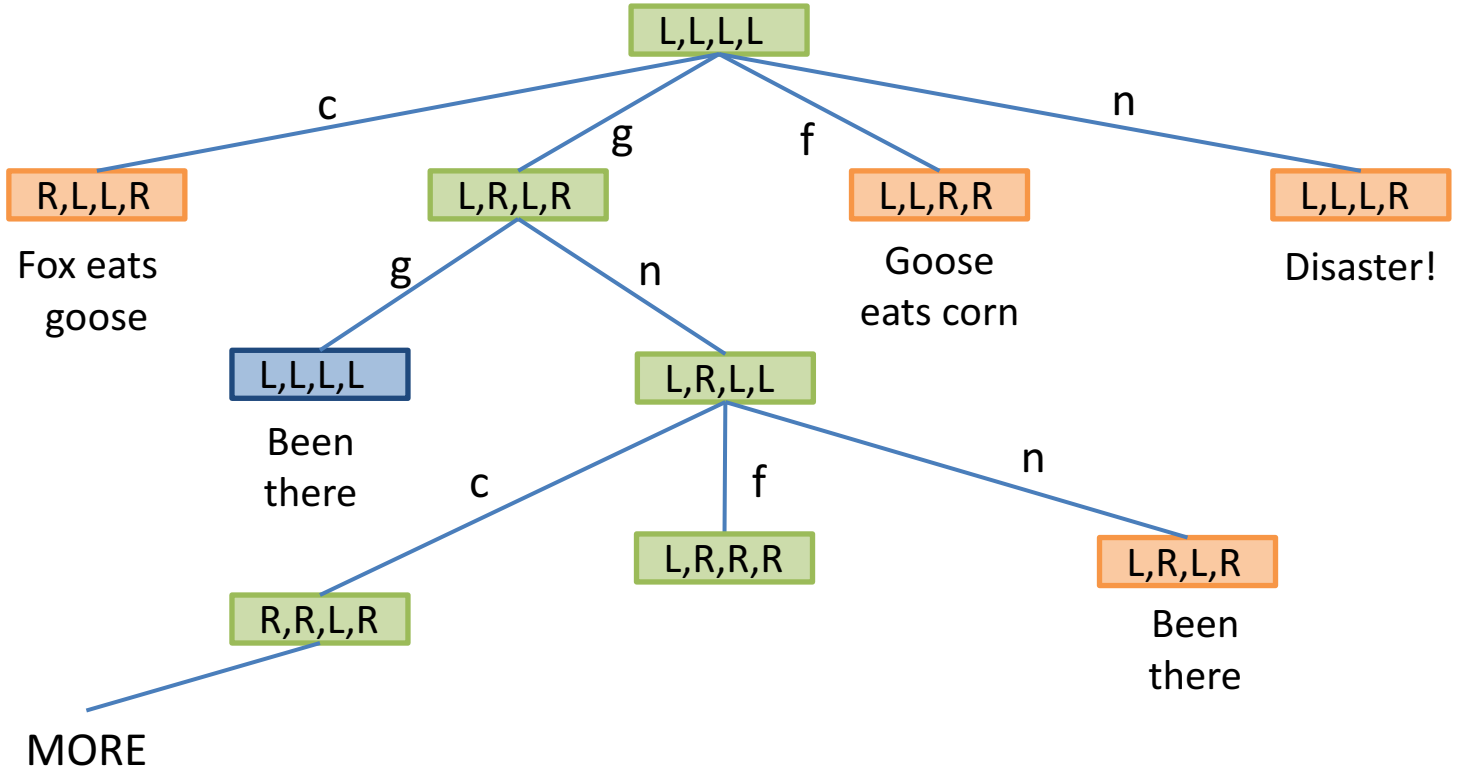
(Note: Initial state could be [R,R,R,R], and search process would be similar)

For the second move, at [L,R,L,R] there are only two possibilities: *g* or *n*. Only one of them is worth doing.

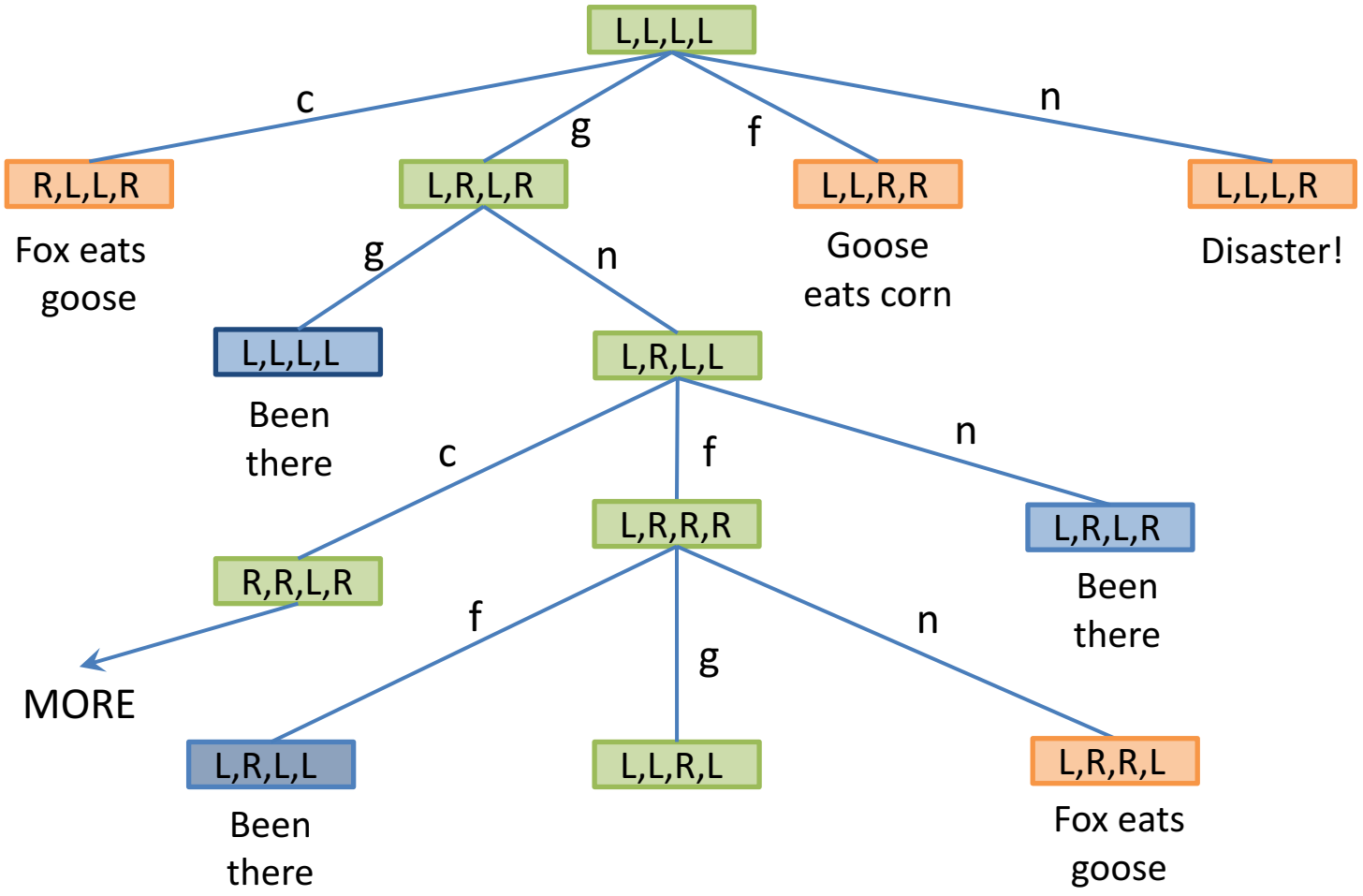


The structure we are developing is called a **search tree**.

For the third move, at [L,R,L,L] there are three possibilities: *c*, *f*, or *n*. Two of them are viable. We will follow up one of them (by random choice).

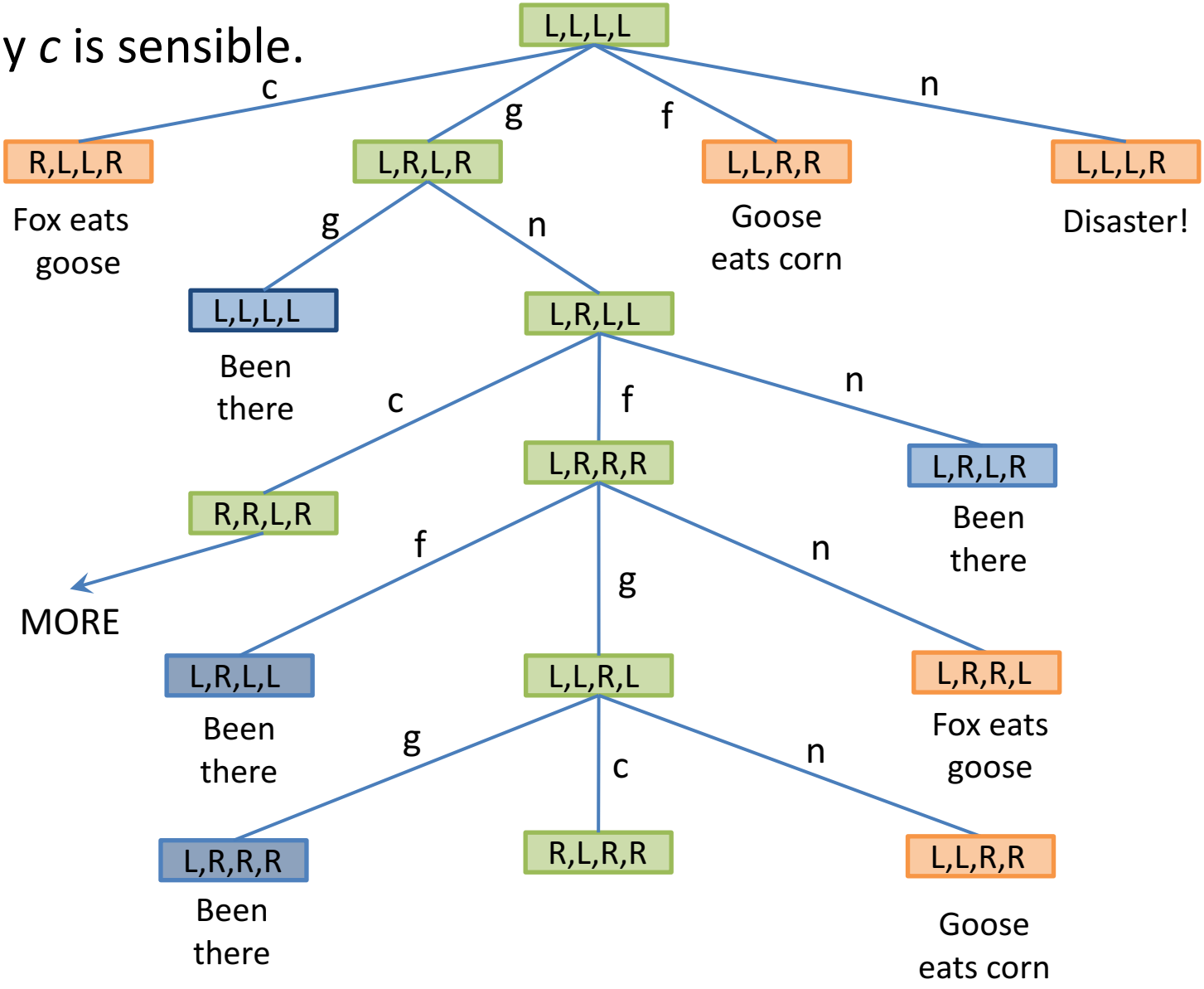


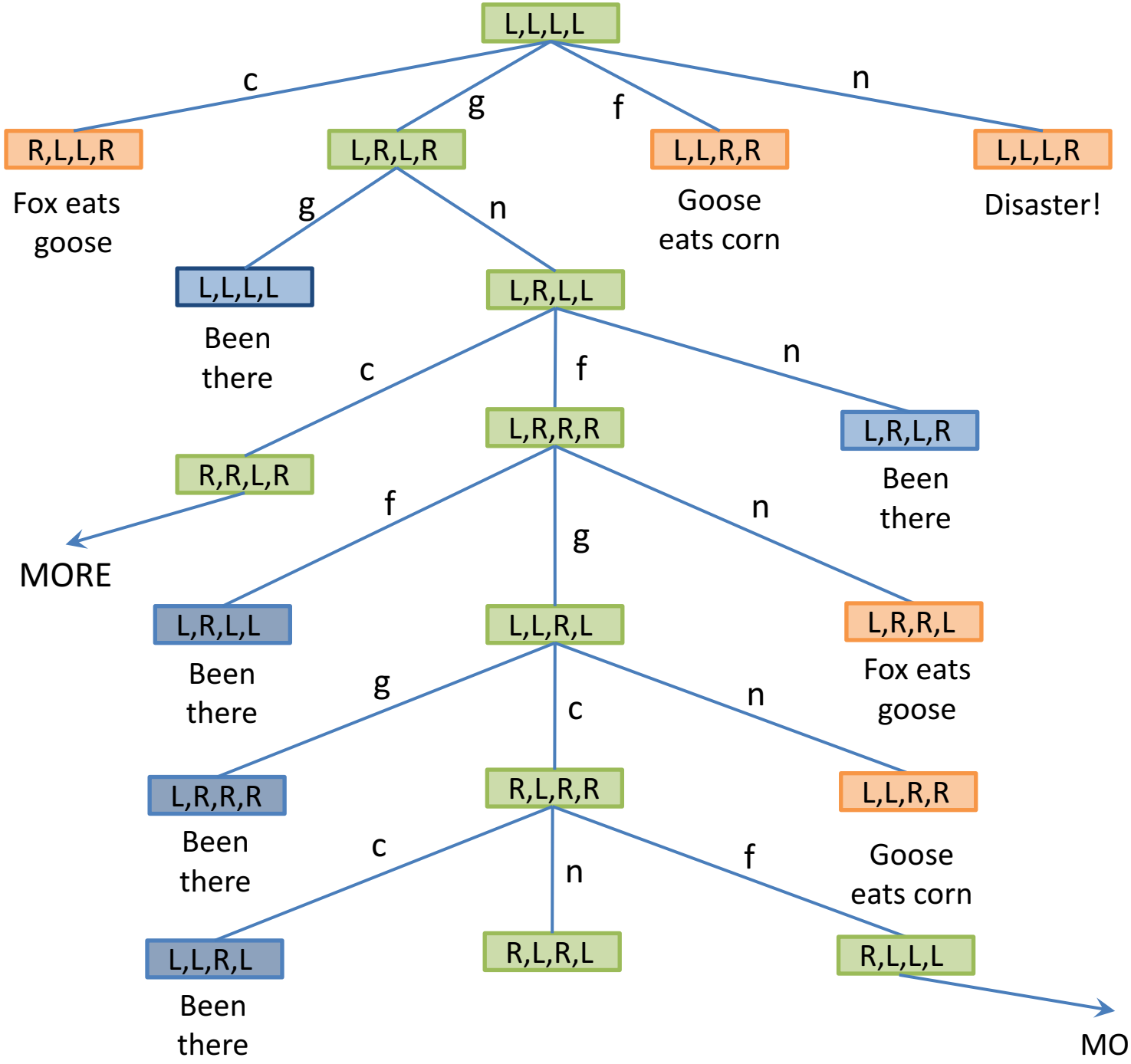
For the fourth move, at [L,R,R,R] there are three possibilities: *f*, *g*, or *n*. Only *g* is sensible.

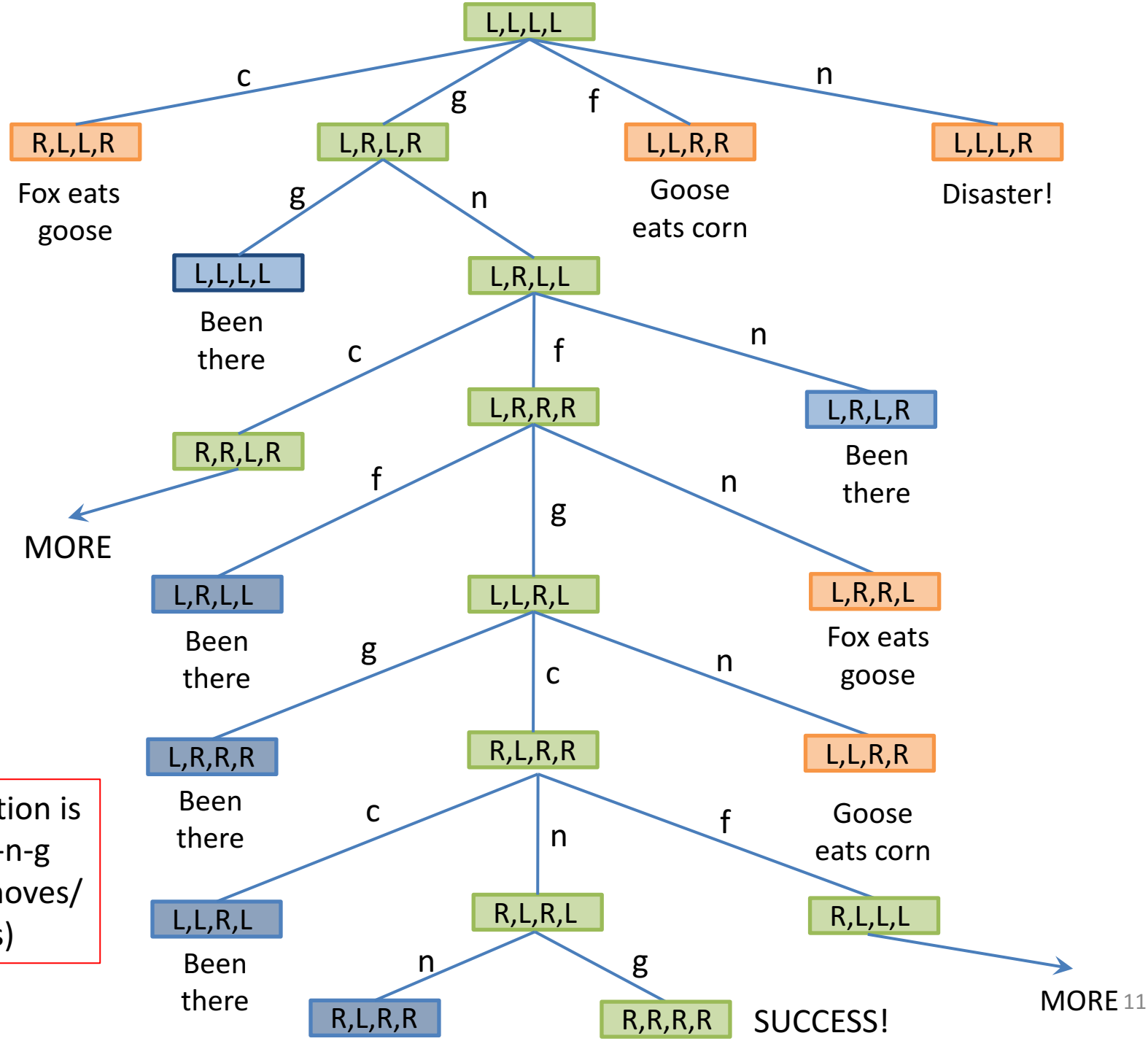


For the fifth move, at [L,L,R,L] there are three possibilities: *g*, *c*, or *n*.

Only *c* is sensible.




L,L,L,L
c
g
f
n
R,L,L,R
Fox eats goose
L,R,L,R
L,L,R,R
Goose eats corn
L,L,L,R
Disaster!
g
n
L,L,L,L
Been there
L,R,L,L
c
f
n
R,R,L,R
L,R,R,R
L,R,L,R
Been there
MORE
f
g
n
L,R,L,L
Been there
L,L,R,L
L,R,R,L
Fox eats goose
g
c
n
L,R,R,R
Been there
R,L,R,R
L,L,R,R
Goose eats corn
c
n
f
L,L,R,L
Been there
R,L,R,L
R,L,L,L
MORE



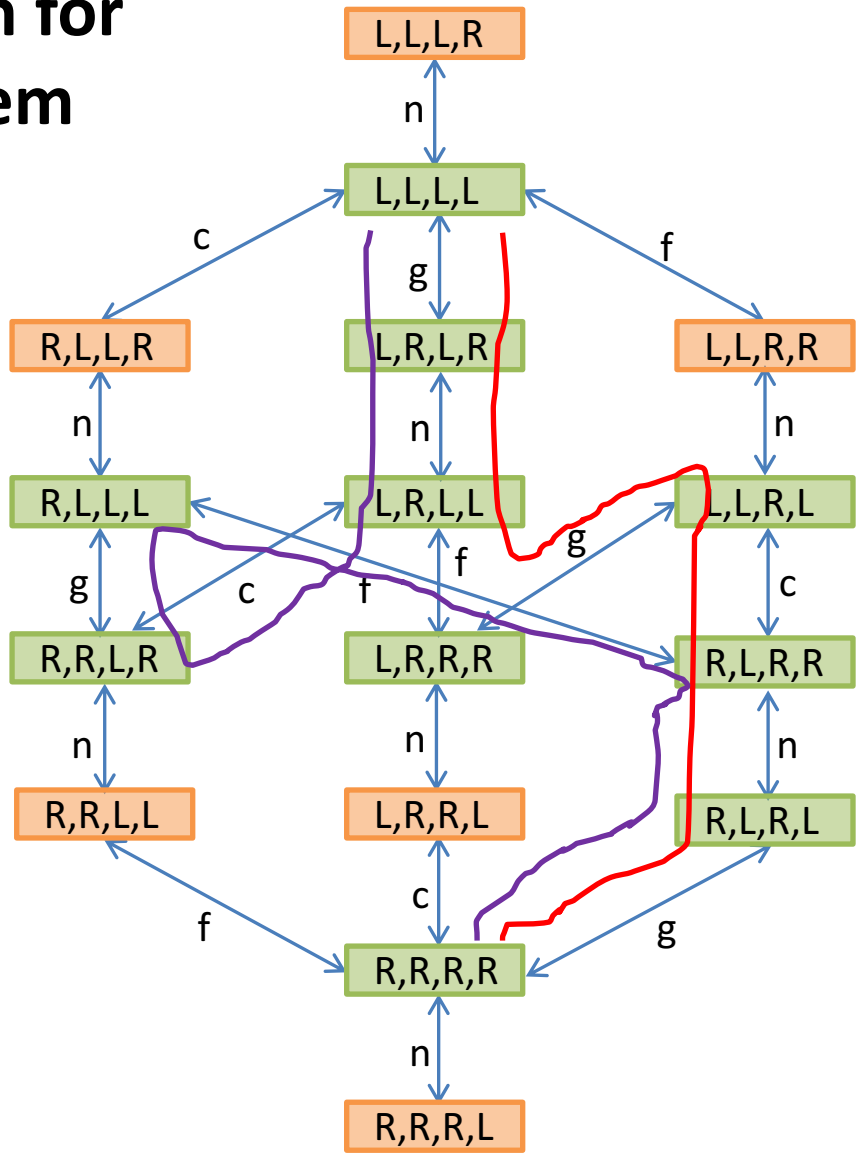
The solution is g-n-f-g-c-n-g (seven moves/ crossings)

State Space Diagram/Graph for Corn, Goose and Fox Problem

unviable

viable

There are 16 states in total, including 6 unviable states.
There are 2 possible optimal solutions, each needing 7 crossings, and some non-optimal solutions (with more than 7 crossings).



How we solved the problem

Abstraction

We identified the **essential elements** of the problem.

Problem/Knowledge Representation

We developed a simple way of representing any **situation / state** that could arise in the course of solving the problem:

In this case, a tuple indicating the location of each entity.

We also represented the **moves / actions** that could be made in solving the problem:

In this case four such moves that might be applicable in any situation.

Our representation was incomplete in that we did not specify what the **conditions and consequences** of these moves would be.

A representation suitable for a computer would have to include this.

Search

We used a straightforward search method through the **possible moves** to find a solution to the problem (there are many advanced search strategies).

PROBLEM/KNOWLEDGE REPRESENTATION

+

SEARCH

=

ARTIFICIAL INTELLIGENCE

(TRADITIONAL)

State Space Representation

State space representation of problems has five key components:

1. A ***State Space***, ***S***

   S is a set of states. There is no limit to the number of states.

   Each state is a distinct situation that could arise during attempts to solve the problem.

      It is usually OK to also include states that could never arise.

2. An ***Initial State***, $s_i \in S$

   The state from which the problem solver begins

3. A Set of ***Goal States***, $G \in S$

   Note: This set may have only one member.

State Space Representation (2)

4. A Set of **Operators**, ***O***

   Members of this set are the actions that can be taken or operations that can be applied in solving the problem. In principle, one operator only is ok.

5. A **Transition Function**: $T : \boldsymbol{S} \times \boldsymbol{O} \rightarrow \boldsymbol{S}$

   This denotes the effect of the actions available for solving the problem.

   It defines the state transition produced by applying any operator ***o*** in any state ***s***. The result is also a state.

   The transition function is often specified by defining the transformation produced by each operator individually (pre- and post- conditions).

   (see examples later)

Representing Operators and Transition Function

There are many ways for representing operators.

One of the most useful is to define each operator in terms of its pre-conditions and post-conditions.

Pre-conditions

Facts that must be true immediately before the operator is to be applied.

The operator cannot be applied if its pre-conditions are not satisfied.

Post-conditions

Facts that must be true immediately after the operator has been applied.

Thus the post-conditions describe the effects of the operator.

We could informally represent the operators in the Corn, Goose and Fox problem:

***Operator c:* (*g, f, and n could be* defined *similarly*)**

Pre-conditions: Boat/farmer on the same bank as corn.

Post-conditions: Corn is on opposite bank. Boat/farmer on opposite bank.

The Three Jugs Problem

You have three jugs which can hold exactly 8, 5, and 3 litres of wine, respectively.

The 8 litre jug is full to the brim with wine, the other two jugs are empty.

You must divide the wine into two equal amounts of 4 litres each accurately.

It is assumed that you can pour (decant) without spilling, but you cannot use any other vessels.

There is no calibration on the jugs.

(**Is this more difficult than the corn, goose and fox problem?**)

Formalising the Three Jugs Problem

The State Space

We denote any possible state as a 3-tuple $\langle x,y,z \rangle$ where

x is the amount of wine in the 8 litre jug

y is the amount of wine in the 5 litre jug

z is the amount of wine in the 3 litre jug

x, y, z are integers.

Hence $0 \leq x \leq 8$, $0 \leq y \leq 5$, and $0 \leq z \leq 3$.

Initial State

$\langle 8,0,0 \rangle$

Goal State

$\langle 4,4,0 \rangle$

Formalising the Three Jugs Problem (2)

Set of Operators

Decant_X_to_Y, *Decant_X_to_Z*, *Decant_Y_to_X*
Decant_Y_to_Z, *Decant_Z_to_X*, *Decant_Z_to_Y*

Transition Function

Operator: *Decant_X_to_Y*

Pre-conditions: $\langle x,y,z \rangle$, $x > 0$ and y<5

Post-conditions: $\langle x',y',z' \rangle$

IF $x < (5 - y)$ THEN $x' = 0$, $y' = y + x$, $z' = z$.

ELSE $x' = x - (5\text{-}y)$, $y' = 5$, $z' = z$.

x denotes value of X before operation; x' its value after the operation.

Five other operations could be defined in a similar way (see Java code).

Representing the Three Jugs Problem in Java

```
public class ThreeJugState{

    private static int[] capacity = new int[3]; // Holds the capacities of each jug
    private int[] content = new int[3]; // Hold the current contents of each jug
```

Representing the Three Jugs Problem in Java (2)

```
/** Constructors ------------------------------------------------------------- */

    /**
     * Constructor that creates a new state leaving capacities of each jug unchanged.
     * x, y, and z are the contents of each jug
     */
    public ThreeJugState(int x, int y, int z) {
        content[0] = x;
        content[1] = y;
        content[2] = z;
    }

    /**
     * Constructor that creates a new state and resets the capacities.
     * (Default values of capacities are zero so this constructor should be used
     * when a new problem is begun).
     * x, y, and z are the contents of each jug
     */
    public ThreeJugState(int capX, int capY, int capZ, int x, int y, int z) {
        this(x, y, z);
        capacity[0] = capX;
        capacity[1] = capY;
        capacity[2] = capZ;
    }
```

Representing the Three Jugs Problem in Java (3)

```
public ThreeJugState decant(int source, int dest) {
   if (source == dest) { // Source and dest are same jug
       System.out.println("WARNING: Attempt to decant from and to same jug: " + source);
       return this;
   } else if (content[source] == 0) { // Nothing to pour
       return this;
   } else if (content[dest] == capacity[dest]) { // Destination already full
       return this;
   } else {
       int remainingVolume = capacity[dest] - content[dest];
       int newDestContent, newSourceContent;
       if (remainingVolume >= content[source]) {  // Empty source into destination
           newSourceContent = 0;
           newDestContent = content[dest] + content[source];
       } else {        // Fill up destination from source leaving remainder in source
           newDestContent = capacity[dest];
           newSourceContent = content[source] - remainingVolume;
       }
       ThreeJugState newState = this.cloneState();
       newState.setContent(source, newSourceContent);
       newState.setContent(dest, newDestContent);
       return newState;
   }
}
```

Representing the Three Jugs Problem in Java (4)

```
    /* Setting and Getting */

     public int getContent(int j) { return content[j]; }

     public void setContent(int j, int vol) { content[j] = vol; }

     public int getCapacity(int j) { return capacity[j]; }

     public void setCapacity(int j, int vol) { capacity[j] = vol; }


     /**
      * cloneState creates a new state identical to current state
      *
      * @return a new state whose content values are same as those of current state.
      */
     public ThreeJugState cloneState() {
         return new ThreeJugState(content[0], content[1], content[2]);
     }

}
```

The 8-Puzzle Problem

The 8-puzzle consists of a 3 by 3 grid. On each grid square there is a title, except for one square that remains empty. The 8 tiles are numbered from 1 to 8 respectively, so that each tile can be uniquely identified. A tile next to the empty square can be moved into the empty space, leaving its previous square empty in turn. The objective of the 8-puzzle problem is to find a solution, a sequence of tile moves, which leads from any initial grid configuration to a given goal grid configuration, e.g.,

8	7	6
5	4	3
2	1	

Initial State

1	2	3
4	5	6
7	8	

Goal State

Your assignment will be to write a Java program to solve this problem.

Summary

Toy problems

- Corn, Goose and Fox
- Three Jugs

Has this lecture helped improve your problem solving skills?

Finding a solution

- Abstracting the essential features of a problem
- Systematically searching for a solution

State space representation (Problem formalisation)

- State space
- Initial and goal states
- Operators
- Transition function

How to search effectively and efficiently?