

# Tutorial 2:

## Problem Representation and Uninformed Search

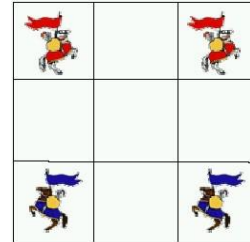
### Question 1

#### Exchanging the Knights

**States:** 2 blue knights (B) and 2 red knights (R) located on a 3x3 board, can only be located on the edges, can't have more than 1 knight in one square, blank cells (0)

**Initial state:**

R	0	R
0	0	0
B	0	B



**Successor function:** 8 actions e.g. move down one row and right 2 columns; move up one row and right 2 columns

**Goal test:**

B	0	B
0	0	0
R	0	R

**Path cost:** each step costs 1, so the path cost is the number of steps in the path from the initial to the goal state

#### Towers of Hanoi

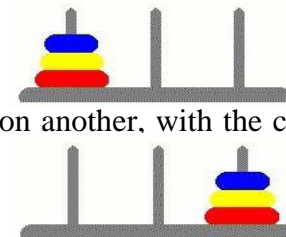
**States:** 3 pegs (1,2,3), 3 disks (1,2,3), combinations of the disks on the pegs, with the condition that larger disks cannot be on top of smaller disks

**Initial state:** all 3 disks on the first peg

**Successor function:** Take the top disk of one peg and place it on another, with the condition that there is not a smaller disk already on the new peg

**Goal test:** all 3 disks on the third peg

**Path cost:** each step costs 1, so the path cost is the number of steps in the path from the initial to the goal state



### Question 2

A representation should be as formal as possible; the more formal you make it, the fewer assumptions are built in. The goal is to have no assumptions.

State transitions should be written down in the same terms as the representation you use. The set of state transitions of a system should clearly indicate all of the possible changes that can happen to the \*state representation\*.

There are different ways in which to represent the constraints in a problem. Each has its own benefits and issues:

- Check the constraints at the transition level: each transition has its own constraint check. This is flexible, but time consuming, especially in simple problems with many transitions, but few constraints.
- Check the constraints at the system level: the system has a set of constraints that are checked as transition pre/post conditions. This is less flexible than transition-level constraints, but also simpler and less cumbersome. One significant downside is that this can make your problem representation less formal, causing some sloppiness in the representation.
- Don't check the constraints, but make invalid states infinitely bad: this option is only available when using a heuristic search algorithm, and may be less or more efficient, depending upon the problem.

### Exchanging the Knights

There are several elegant solutions to this problem.

The best is probably a linked-list representation such that, with board coordinates labelled like this:

```
0 1 2
3 X 4
5 6 7
```

The linked list represents (in order) {0,4,5,1,7,3,2,6}. Transitions involve moving a knight (a symbol in the list) to a neighbouring empty node. Maintaining the constraints under this system is *very* easy, but the system isn't extensible.

An extensible option is to lay out your board like this (matrix):

```
0,0 0,1 0,2
1,0 1,1 1,2
2,0 2,1 2,2
```

and make the knight moves look something like:

$$\{x, y\} = \{x+1, y+2\}$$

$$\{x, y\} = \{x+2, y+1\}$$

etc.

Another non-extensible option is to lay out your board like this:

```
0 1 2
7 X 3
6 5 4
```

and make the transition rules:

$$x = (x+3) \bmod 8$$

$$x = (x+5) \bmod 8$$

## Towers of Hanoi

One option is to represent the 3 poles as 3 stacks

$P[0], P[1], P[2]$

Each stack is a LIFO queue with two operations:  $Push(S,i)/Pop(S)$

An alternative is a 3x3 array

$Disks[P_n][D_n]$

But you have to define the two stack operations on this array

The 3-stack representation is probably the nicest, as the problem translates very naturally to stacks. Having a stack makes a lot of implicit assumptions explicit (such as: you can't move a disc from the bottom of a peg).

Time/space complexity wise, stacks are also good.

There is only really one transition in this system (pop a disc and push it somewhere else), but you need to express the fact that the pegs are different. Also, you need to make sure the constraint is maintained.

## Missionaries and Cannibals

The canonical solution to this is to take advantage of the fact that the right side of the bank always contains 3 minus the number of [missionaries|cannibals] on the left bank. This reduces the necessary information to the number of missionaries on the left bank, the number of cannibals on the left bank and the position of the boat: MCB

Transitions are obvious, with the problem that there is no particularly elegant way to write down the restrictions.

Notes on advantages and disadvantages:

- Is it extensible?
  - Array representation -> less extensible (why?)
  - if the board size changes,...
- Space complexity:
  - The smaller the size of state representation, the better:
  - $M$  = maximum number of nodes in memory.
  - Space requirement =  $M \times \text{Size}(\text{state})$
- Time complexity
  - Less actions ( $A$ ) and less conditions ( $C$ ), the better.
  - $N$  = total number of nodes expanded
  - Time complexity =  $N \times (A \times C)$
  - Linked list representation -> less complex in terms of both space and time.

### Question 3

When deciding on which search method would be better for your problem, there are several factors you will want to consider:

- Do I care about finding the most efficient solution, or just any solution?
- What is the branching factor of my problem?
- How deep is a solution likely to be found?
- How much of a problem are repeated states likely to be?

Obviously, you would always like to have repeated state detection, because nobody likes the idea of their search being stuck in an infinite loop, but there may be situations where the space complexity required by repeated state detection is unacceptable.

#### **Breadth-first:**

Suppose:  $b$  = EBF,  $d$  = solution depth

Optimal (if step cost=1) and complete

Exponential space/time complexity:  $O(b^d)$

#### **Depth-first:**

Suppose:  $b$  = EBF,  $d$  = solution depth,  $m$ =max depth

Not optimal nor complete (if repeated states are not checked)

Linear space complexity:  $O(bm)$

Exponential time complexity:  $O(b^m)$  (bad if  $m > d$ )

### Question 4

Pseudo-code guide lines:

- Define function name and description.
- Define input/output parameters and descriptions.
- Define Subroutine functions names and descriptions.
- States lists of actions to perform.
- Use indentation to define blocks corresponding to flow chart boxes.

Loop control:

- for each  $a$  in ListA
- loop do
- while  $a > 0$  do

Condition:

- if Empty?(L) then return failure

## Exchanging the Knights

Assuming linked cell representation

### Goal-test

```
function Goal-test(state) returns succeeds or failure
    if state = [B,0,R,0,R,0,B,0] return succeeds
    else return failure
```

### Successor Function

```
function Expand(node) return nodes
    T = []
    for i = 0 to 7:
        s = State(node)
        if empty(s[i+1 mod 8]) and not empty(s[i])
            then s[i+1 mod 8] =s[i]; add(Make-Node(s),T)
        s = State(node)
        if empty(s[i+7 mod 8]) and not empty(s[i])
            then s[i+7 mod 8] =s[i]; add(Make-Node(s),T)
    return T
```

## Towers of Hanoi

Assuming stack representation

### Goal-test

```
function Goal-test(state) returns succeeds or failure
    if state[0]=[] and state[1]=[] and state[2]=[1,2,3] return succeeds
    else return failure
```

### Successor Function

```
function Expand(node) return nodes
    T = []
    for m = 0 to 2:
        for n = 0 to 2:
            P = State(node)
            if n≠m and P[m] is not empty and (P[n] is empty or
            topDisk(P[n]>topDisk(P[m]))
            then Push(P[n],Pop(P[m])); add(Make-Node(P),T)
    return T
```

## **Update function**

### **Breadth first**

```
if (not-empty(L))
    if (goal-test(L(0))
        return path
    else
        expanded_nodes = successor-function(L(0))
        // remove n from start of L, put expanded nodes at end of L
        L = [L(1:end) expanded_nodes]
```

### **Depth first**

```
if (not-empty(L))
    if (goal-test(L(0))
        return path
    else
        expanded_nodes = successor-function(L(0))
        // remove n from start of L, put expanded nodes at start of L
        L = [expanded_nodes L(1:end)]
```