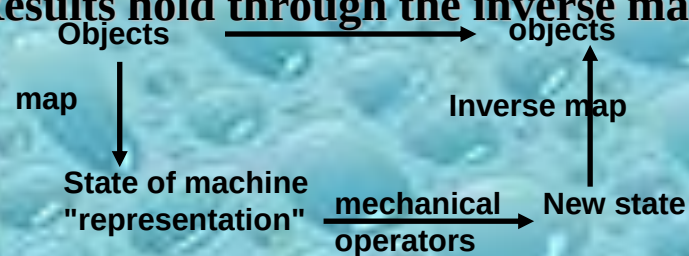
(General) Problem Solving

Many problems can be solved by *searching* a "problem space" of *representations*

Law of Representation

- **Objects of Interest are MAPPED into bits/symbolic form, held as bits or state**
- **mechanical operations are performed *on the representations***
- **Results hold through the inverse map**

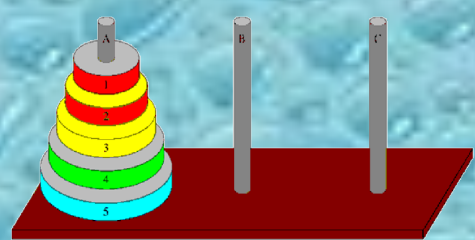


Can a computer "solve" problems?

- Puzzles are an area where human intelligence seems necessary
  - Few animals can solve Rubiks Cube!
- Many kinds
  - Towers of Hanoi
  - Sliding Block Puzzles
  - Combination Puzzles
  - Interlocking Puzzles
  - Jigsaw Puzzles
  - Cryptographic Puzzles
  - etc.

Towers of Hanoi

- Three posts with n disks
- Goal is to move disks from one post to another without ever putting larger on smaller
- Takes 2^n-1 steps



Water Jug Problem

- **Two friends who have an eight-quart jug of water wish to share it evenly. They also have two empty jars, one holding five quarts, the other three. How can they each measure exactly 4 quarts of water *since pouring 4 qts from 8->5 is not a legal move?***

Coin Puzzle

- infinite family of puzzles
  - what 5 coins add up to 42c?
  - what 6 coins add up to 88c?
- many puzzles are insoluble
  - but its trivial to create solvable puzzles!
- humans use heuristics
  - try quarters first
  - take pennies to get to multiple of 5
  - etc

N Queens

- **Problem: Place N queens on an N by N chessboard without them attacking each other**

How big are these problems? Roughly how many configurations?

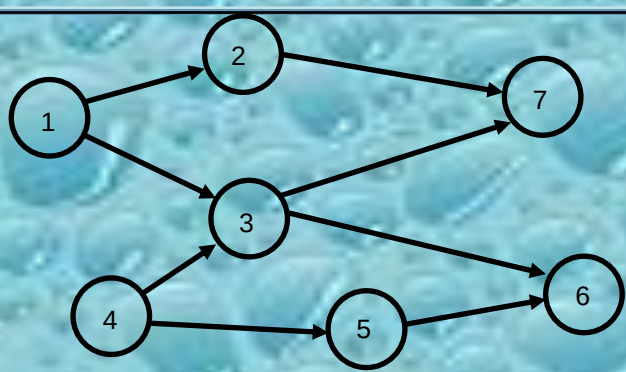
Water Jug	20
Hanoi	n^3
Coin Puzzle	polynomial in c coins
TicTacToe	3^9
N-queens	n! placements
Game	Pieces ^{positions}

We often cannot store the whole thing!

Problem Space Approach

- **Formalize Task as**
  - **A set of discrete states**
  - **Operations which make legal transitions between the states**
  - **Initial and Goal States**
    - **–or goalp function**
    - **–Solving the problem means finding a path from an initial state to a goal state**
    - **–REPRESENTATION PLUS SEARCH**

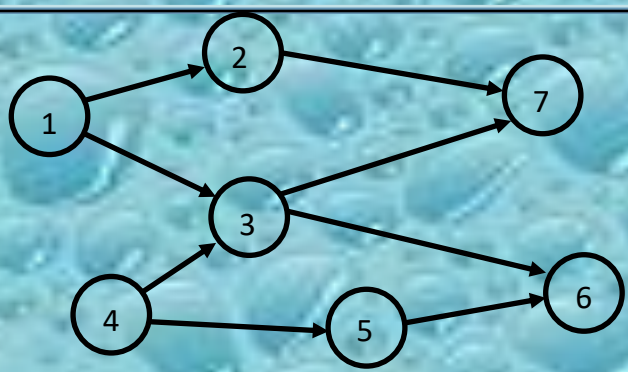
A Graph is a set {V, E}



V = { 1, 2, 3, 4, 5, 6, 7}

E = { (1 2), (1 3), (2 7), (3 7),
(4 3), (4 5), (5 6), (3 6)}

A Problem Space is a graph..



Vertices are representations of problem states
Edges are transitions corresponding to "Moves", calculated by logical rules or code

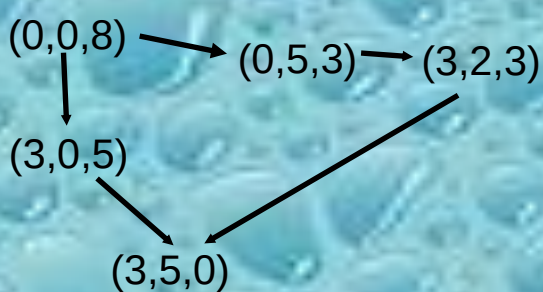
Usually, we don't enumerate the whole space.

Water Jug Problem

capacity is (3,5,8)

state is (x y z) oz in each jug ALL < capacity

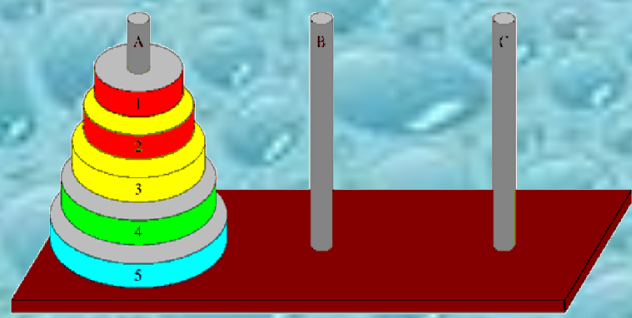
You pour from one to another the max you can



Goal is (0 4 4)

Towers of Hanoi

- Three posts with n disks
- Goal is to move all disks from one post to another without ever putting larger on smaller
- Representations can be in 3 lists of symbols
  - (1 2 3) nil nil
  - (2 3) (1) nil
  - (3) (1) (2)
  - (3) nil (1 2)
  - nil (3) (1 2)
  - (1) (3) (2)
  - (1) (2 3) nil
  - nil (1 2 3) nil



Hanoi as a problem space

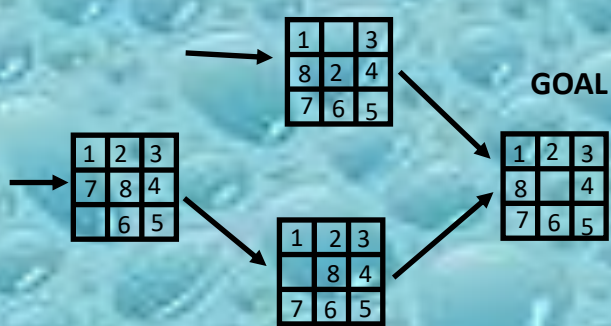
```
(defun hlegal (post1 post2)
 (and post1 (or (null post2)(< (car post1)(car post2)))))

(defun hnext (state &aux ans)
 (let ((one (car state))
       (two (cadr state))
       (thr (caddr state)))
  (if (hlegal one two)
       (push (list (cdr one)(cons (car one) two) thr) ans))
  (if (hlegal one thr)
       (push (list (cdr one) two (cons (car one) thr)) ans))
  (if (hlegal two one)
       (push (list (cons (car two) one) (cdr two) thr) ans))
  (if (hlegal two thr)
       (push (list one (cdr two)(cons (car two) thr)) ans))
  (if (hlegal thr one)
       (push (list (cons (car thr) one) two (cdr thr)) ans))
  (if (hlegal thr two)
       (push (list one (cons (car thr) two) (cdr thr)) ans))
  ans))
```

N Queens

- Problem: Place N queens on an N by N chessboard without them attacking each other
- Representation can be
  - array of N xN with 1-N integers for queens
  - list of (row.col) pairs where queens are
  - bit array with 1's where queens are placed
  - Vector of N columns (implicit row order)

8-Puzzle Example

- Each Vertex is an arrangement of tiles
- Edges correspond to simple movements
- Goal is tiles in particular order



How to represent the Problem?

- **Each state is 3 by 3 matrix**
  - **Numbers 1-8 represent tiles**
  - **Number 0 represents empty space**
- **(Or just a list of 9 numbers 0-8)**
- **Transitions correspond to legal moves of the puzzle (Law of representation)**

How to compute the transitions?

- **Use Logical Rules, encoded as computer programs which act upon the chosen representation to construct new vertices:**

1) If the hole is in the middle, move in a side (4 rules)
   if a[2,2]= 0 then pick [1,2] [3,2] [2,1] or [3,1]
2) If the hole is in corner, move a side (8 rules)
   if a[1,1]=0, pick [1,2] or [2,1]
3) if the hole is in side, move middle or corner (12 rules)
   if a[1,2]= 0 then pick [1,1] [2,2] or [1,3]

Setup for 8-puzzle

- (defparameter *goal* '(1 2 3 8 0 4 7 6 5))

- (defparameter *moves*
- '((d r)(d r l)(d l)
- (u d r)(u d l r)(u d l)
- (u r)(u r l)(u l)))
- ;;;effect of left right up and down on a tile index
- (defparameter *delta*
- '((l . -1)(r . 1)(u . -3)(d . 3)))

find all moves from a state

- (defun move (state m)
- (let* ((i (index 0 state)) ;find the 0
- (j (+ i (cdr (assoc m *delta*)))))
- (loop for k from 0 to (- (length state) 1) collect
- (cond ((eq k i)(nth j state))
- ((eq k j)(nth i state))
- (t (nth k state))))))

- (defun allmoves (state)
- (loop for m in (nth (index 0 state) *moves*) collect (move state m)))

utility fns

- (defun index (item list)
- (cond ((null list) -1)
- ((eq (car list) item) 0)
- (t (+ 1 (index item (cdr list))))))

- (defun pick (items) (nth (random (length items)) items))

Relation between Representation and Rules

- **Compact representation, e.g. bitcoded enumeration of only possible states, means:**
  - **Larger subgraph can be stored in memory**
  - **But rules of transition get more complex (and slower)**
- **Using a more expansive state representation means**
  - **Less graph can be stored in memory**
  - **often more efficient rule application**
    - ***Moral: Transition rules are expressed with respect to state representation***

Sometimes Mathematics can Help

Eg: gray-coding the 8 puzzle reduces 3 kinds of rules to one rule (to ring them all)

- **Represent state as nine 4-bit codes according to the following array:**

6 0110	7 0111	3 0011
14 1110	15 1111	11 1011
12 1100	13 1101	9 1001

- **Transition Rule: swap the hole with any tile whose label differs by single bit**

GOAL STATE

Tile		1	2	3	4	5	6	7	8
Label	15	6	7	3	11	9	13	12	14

Math Trick

- x = |a – b|
- Is x a power of 2?
- (zerop (logand x (- x 1))
  - Why? 2^k = 100000 and 2^k-1 = 011111
  - Bitwise “AND” then together -> 0

using math trick for 8 puzzle

- (defvar *goal* '(15 6 7 3 11 9 13 12 14))
- (defun pow2 (x) (if (>= x 1) (zerop (logand x (- x 1)))))
- (defun swap (board t1 t2)
- (sublis (list (cons t1 t2)(cons t2 t1)) board))
- (defun allmoves (state)
- (loop for tile in (cdr state) with hole = (car state)
- when (pow2 (abs (- hole tile)))
- collect (swap state hole tile)))

Weak Methods

- **General Purpose Methods**
  - **No Domain Knowledge**
  - **But sometimes it can be heuristically added**
- **Often weak methods arise through human introspection on our own problem-solving techniques.**
- **Brute force/enumeration**
- **Random Generate & Test**
- **Hill Climbing**

So Whats a Strong Method?

- A strong method is based on knowledge of the domain, and can make for much faster algorithms.
- Example Hanoi:
- To move n discs from peg A to peg C:

1. move n−1 discs from A to B. This leaves disc n alone on peg A
2. move disc n from A to C
3. move n−1 discs from B to C so they sit on disc n

strong method for hanoi

```
(defun hanoi (n source target other)
 (if (= 1 n) (print (list 'move source target))
  (progn (hanoi (- n 1) source other target)
         (hanoi 1 source target other)
         (hanoi (- n 1) other target source))))
```

The Random Search Algorithm

- While Current State is not Goalstate:
  - 1) Pick one rule that applies
  - 2) Move to next state

  –What's Wrong with This?

Water Jug random Walk

```
(defparameter *jugs* '(3 5 8)) ;;;size of containers to pour

(defun nextjug (config)
  (loop for (x y) in '((0 1)(0 2)(1 0)(1 2)(2 0)(2 1)) ;source-dest pairs
        when (and (not (zerop (nth x config))) ; source has content
                  (< (nth y config)(nth y *jugs*)));goal below capacity
      collect (movewater config x y (min (- (nth y *jugs*)(nth y config))
                                         (nth x config)))))

(defun movewater (config source dest amount)
  (loop for i from 0 to 2 as x in config collect
        (cond ((= i source)(- x amount))  ;subtract from source
              ((= i dest)(+ x amount))    ;add to destination
              (t x))))                    ;leave third jug alone
```

Stupid approach to WaterJug continued

```
(defun juggoal (config)(equal config '(0 4 4)))

(defun randomwalk (config goalp)  ;use #'juggoal
 (cond
   ((funcall goalp config) config)
   (t (randomwalk (pick (nextjug config)) goalp))))

(defun pick (items) (nth (random (length items))
 items)) ;;uniform pick one thing from a list
```

can we randomwalk the coin puzzle?

- The coinpuzzle is DISSIPATIVE
  - each step adds a coin and lowers total-to-go
- Random walk needs to not overshoot

Brute Force Enumeration

- It Really Works
- Shunned by AI historically because as problems scale, computers become inadequate.
  - But for Moore's Law!
  - But it works for cracking passwords!!!
- Examples:
  - Finding assignment of variables for logical statement by enumerating the whole truth table.
  - Calculating all permutations or rotations, and searching through them with a for loop

Brute Forcing the N queens

- Representation is the 8 columns in a list
  - Row number is implicit
- Every permutation of 1-8 is possible solution
  - N!
- Lets have a “legal solution test” and loop through all n! permutations.

Brute Forcing the N queens

```
(defun permute (lst)
  (if (= 1 (length lst)) (list lst)
      (loop for x in lst append
        (loop for y in (permute (remove x lst :count 1))
          collect (cons x y)))))
(defun iota (n) (loop for i from 1 to n collect i))
(defun testperm (p)
  (loop for tail on p always
        (openq (car tail)(cdr tail))))
(defun bruteq (n) (loop for perm in (permute (iota n))
   when (testperm perm) collect perm))
```

testing for queen attack

```
(defun openq (p list) ; test if proposed column p is clear
  (and (not (member p list))
       (loop for i in list as j from 1
           never (eql (abs (- p i)) j))))

(defun printq (board)
  (loop for i from 1 as col in board do
       (loop for j from 1 to (length board) do
            (if (= j col)
                (format t " Q")
                (format t " ")))
       (terpri)))
```

Generate & Test (G&T)

- Generate a random guess (or path)
- Test it.

- GT can be faster than Brute Force but not guaranteed.
  - Factor by guessing and testing divisibility??
  - If solutions are very sparse in the space???

Next time: Organized Search of a problem space

- Instead of random walking or trying to hillclimb, keep track of where you've been.
- Instead of envisioning the graph, just keep dynamic parts in a (queue, stack, program stack).
- Search is still "weak" because
  - It knows not what the graph is about
  - It is broadly applicable
  - But can add domain knowledge to guide it.