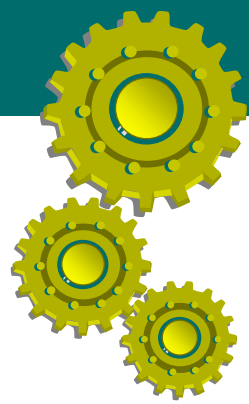
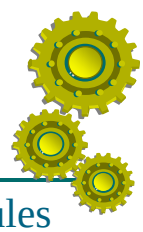


Basic Search on problem spaces

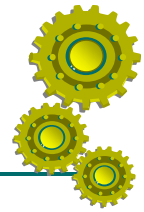
Search

Adds organization to the search tree



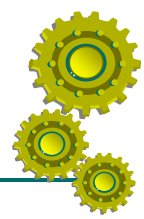
- "Expanding" a node means applying all rules to generate new nodes to explore.
- Testing for success means validating that you found a solution
- Backtracking means undoing moves when you reach a dead end
  - we let the recursive program stack handle dead-ends.

Breadth and Depth



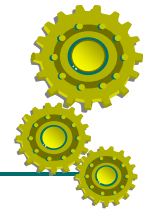
- Breadth-first Search
  - Expand the current state
  - Use a QUEUE for States-to-Explore
- Depth-First Search
  - Expand the current state
  - Uses a STACK for states to explore

Functional arguments (funargs, pass using #')



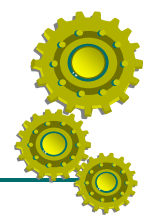
- STATES a list of graph nodes to explore
- #'NEXTF - expands a node (state) to all legal next states
- #'GOALP tests for solution

Depth First Search



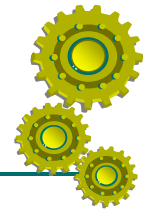
```
(defun dfs (nodes goalp nextf)
 (cond ((null nodes) nil)
          ;; Return the first node if it is a goal node
       ((funcall goalp (first nodes)) (first nodes))
          ;; Push the children on the front of the stack
       (t (dfs (append (funcall nextf (first nodes))
                       (rest nodes));use a stack!
               goalp
               nextf))))
```

Breadth-first



```
(defun bfs (nodes goalp nextf)
 (cond ((null nodes) nil) ;unsolveable
          ;; Return the first node if it is a goal node
           ((funcall goalp (first nodes)) (first nodes))
         ;; Put the children in the back of the queue
           (t (bfs (append (rest nodes)
                           (funcall nextf (first nodes)))
                   goalp
                   nextf))))
```

Test problems: Coingames

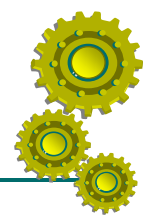


- I have 40 cents made of 3 coins.
- Which nickels, dimes, pennies, quarters?

- State : (total coins)

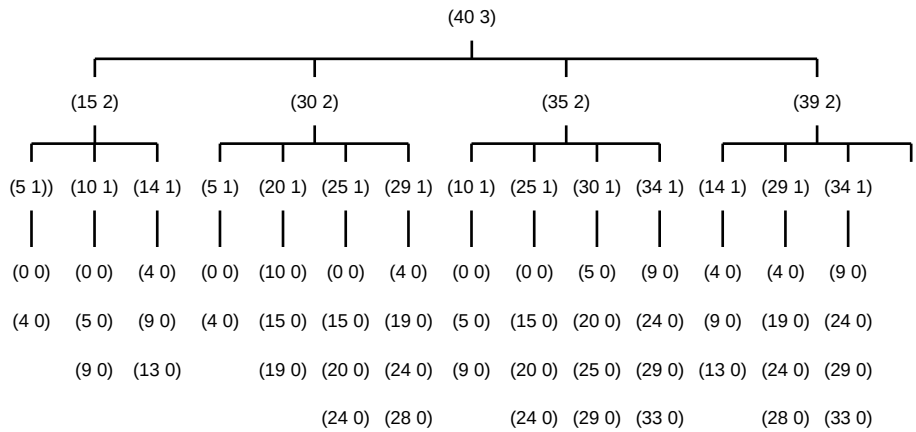
- Goal: (0 0)
- (defun coingoal (x) (and (zerop (car x))
- (zerop (cadr x)))
- How to proceed (40 3) (35 2)(25 1)(0 0)?

This is a family of puzzles



- (defun randcoin (x &aux q d n p)
- (setf q (random x)
- d (random x)
- n (random x)
- p (random x))
- (list (+ (* 25 q)(* 10 d)(* 5 n) p)
- (+ q d n p)))

Coin Game Tree

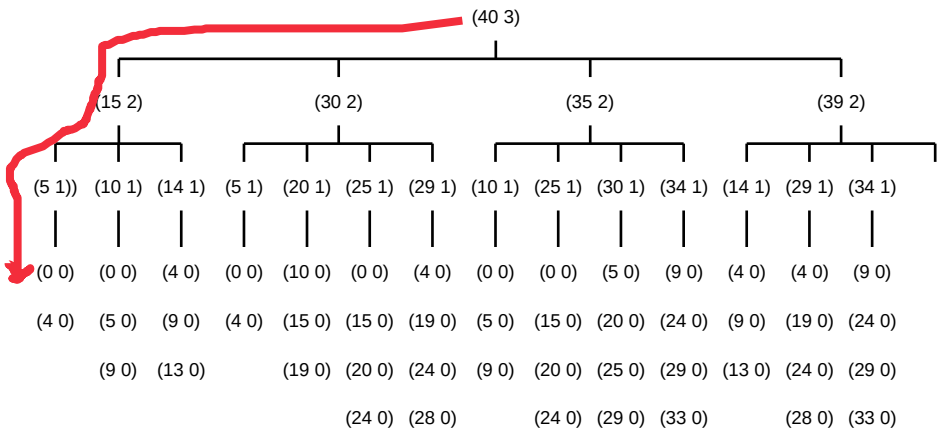


Expand the coin state

(40 3)->((15 2)(30 2)(35 2)(39 2))

```
(defun coinnext (x)
    (if (and (> (car x) 0)(> (cadr x) 0))
      (loop for i in '(25 10 5 1)  ;hidden domain knowledg
        when (<= i (car x))
        collect (list (- (car x) i)
                      (- (cadr x) 1)))))
```

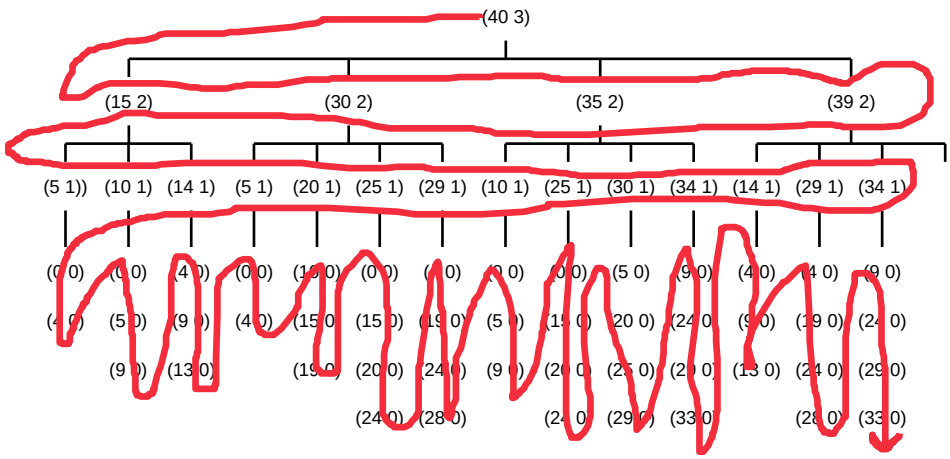
Coin Game Tree: DFS



Features of depth first search

- Sometimes very fast
- uses minimal amount of space
  - one expansion of tree for each level
- May never terminate on infinite problems with loopy problem state graphs
  - Coin Problem is "dissipative"
  - However, many paths hit same states

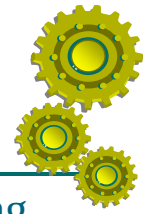
Coin Game Tree: BFS



Features of BFS

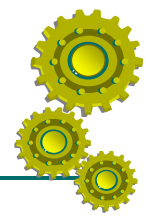
- guaranteed to find a goal.
- shortest path
- Takes an exponentially increasing amount of space as trees get deeper!

Using Program State



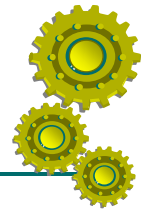
- Instead of explicitly calculating and storing the tree or graph, a program can use its parameters and call/return stack to keep track of search.

implicit dfs code



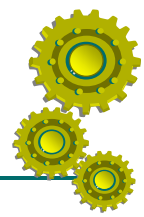
```
(defun solve-coin (total count)
        (cond ((and (zerop total)(zerop count)) T)
              ((or (<= total 0)(<= count 0)) nil)
              (t (loop for i in '(25 10 5 1) thereis
                    (solve-coin  (- total i) (- count 1))))))
```

Paths versus States



- Sometimes you just need to find the state
  - Unfortunately (0 0) says "there is a solution" but doesn't say the solution!
- Other times you need the path of moves to the final state
  - Have to have a representation which can link current state to previous states, back to initial condition

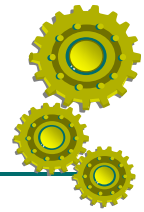
keep sofar in DFS/BFS state (amount coins sofar)



Now to start with (40 3 nil)!!!

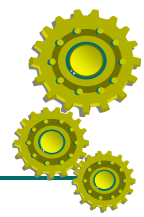
```
(defun coinnextpath (x)
    (if (and (> (car x) 0)(> (cadr x) 0))
      (loop for i in '(25 10 5 1)  ;note heuristic knowledge
        when (<= i (car x))
        collect (list (- (car x) i)
                      (- (cadr x) 1)
                      (cons i (caddr x))))))  ;keep track of path
```

implicit dfs code with path



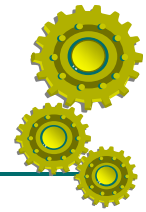
```
(defun solve-coin (total count &optional sofar)
      ;;;sofar is a list of the coins used so far.
   (cond ((and (zerop total)(zerop count)) sofar)
      ((or (<= total 0)(<= count 0)) nil)
      (t (loop for i in '(25 10 5 1) thereis
        (solve-coin  (- total i) (- count 1) (cons i sofar))))))
```

8 queens problem



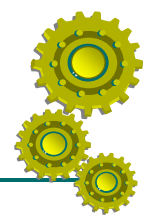
```
• (defun openq (p list) ; test if proposed column p is clear
•   (and (not (member p list))
•        (loop for i in list as j from 1
•              never (eql (abs (- p i)) j))))

• (defun qnext (sofar)
•   (loop for i from 1 to 8
•         when (openq i sofar)
•         collect (cons i sofar)))

• (defun qgoal (sofar)
•   (= 8 (length sofar)))
```

implicit dfs for n-queens



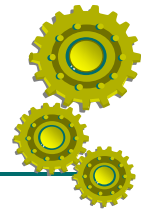
```
(defun oneq (n &OPTIONAL list)
  (if (eql (length list) n) list
      (loop for i from 1 to n
            thereis (and (openq i list)
                         (oneq n (cons i list))))))

;prints all solutions (92 for size 8)
(defun allq (n &OPTIONAL list)
  (if (eql (length list) n) (print list)
      (loop for i from 1 to n
            do (and (openq i list)
                    (allq n (cons i list))))))
```

Water Jug problem



```
(defparameter *jugs* '(3 5 8))

(defun nextjug (config)
  (loop for (x y) in '((0 1)(0 2)(1 0)(1 2)(2 0)(2 1))
        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))
              ((= i dest)(+ x amount))
              (t x))))

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

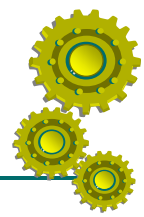
only 16 states!



- (defparameter *jughash* (make-hash-table :test #'equal))

- (defun jugexplore (nodes nextf)
- (cond ((null nodes) nil)
- ((gethash (car nodes) *jughash*) (jugexplore (cdr nodes) nextf))
- (t (setf (gethash (car nodes) *jughash*) (funcall nextf (car nodes)))
- (jugexplore (append (rest nodes)(funcall nextf (car nodes))) nextf))))

Towers of Hanoi



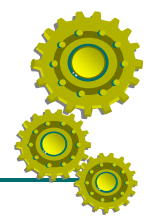
- (defun hgoal (state)
- (equal (cadr state) '(1 2 3)))

- (defun hnext (state &aux ans)
- (let ((one (car state))
- (two (cadr state))
- (thr (caddr state)))
- (and one (or (null two)(< (car one)(car two)))
- (push (list (cdr one)(cons (car one) two) thr) ans))
- (and one (or (null thr)(< (car one)(car thr)))
- (push (list (cdr one) two (cons (car one) thr)) ans))
- (and two (or (null one) (< (car two)(car one)))
- (push (list (cons (car two) one) (cdr two) thr) ans))
- (and two (or (null thr) (< (car two)(car thr)))
- (push (list one (cdr two)(cons (car two) thr)) ans))
- (and thr (or (null one) (< (car thr)(car one)))
- (push (list (cons (car thr) one) two (cdr thr)) ans))
- (and thr (or (null two) (< (car thr)(car two)))
- (push (list one (cons (car thr) two) (cdr thr)) ans))
- ans))

recursive DFS with cycle check

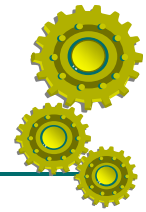
```
(defun dfs1 (state goalf nextf)
  (cond ((funcall goalf state) state)
        ((null state) nil)
        (t (loop for newstate in (funcall nextf state)
                 thereis (dfs1 newstate goalf nextf)))))

(defun dfs2 (state goalf nextf &optional (sofar nil))
  (cond ((funcall goalf state) (cons state sofar))
        ((null state) nil)
        ((member state sofar :test #'equal) nil)
        (t (loop for newstate in (funcall nextf state)
                 thereis (dfs2 newstate goalf nextf (cons state sofar))))))
```

BFS and DFS are Weak



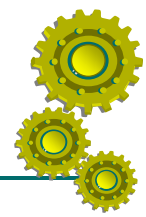
- They work without domain knowledge
  - But quarters-first is domain knowledge!
- What else can we add to PRUNE the tree?
  - If Sum/Coins < 1 or > 25 its impossible
  - First, take pennies to round to 5x
  - etc.
- Adding Domain Knowledge to a weak method makes it stronger.
- Often this is called "Heuristics"

Topics to consider for Effective Heuristic Search



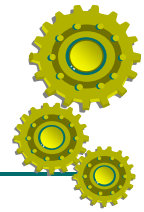
- Search Direction
- Topology of Problem Space

Search Direction



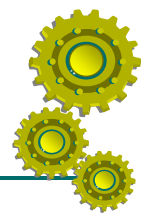
- Backward (Goal-Directed)
  - Start at Goal, look for path to current
- Forward
  - Start at current state, look for goal

Factors in Choosing direction



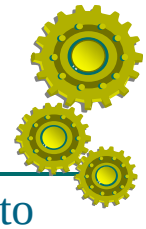
- Density of the Target
  - move from smaller set to larger set
- (Asymmetric?) Branching Factor
  - keep search as small as possible
- Auditability
  - Does reasoning need to be human-like?

Topology of Space



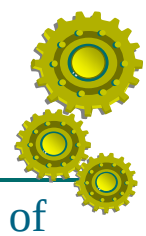
- Is the problem space more like a tree or a graph?
  - Cyclical versus Dissipative Problems
  - dissipative problems "use up" resources as you go
    - Thus DFS is more appropriate then BFS
- Do you really need to keep track of states to avoid cycles?
- What is the relative cost of cycle-checking, duplication of effort, and space?
  - keeping track of where you've been can be slower than revisiting the same state!
    - can use a hash table to speed up cycle check

Heuristic computes "goodness" for any state of the problem



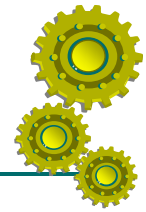
- can use decreasing or increasing distance to goal
- For the 8-puzzle:
  - Guess: counting tiles in position
  - ➢ Need to be as "smooth" as possible across the space!
  - Guess: Sum of "City Block" distance from goal for every tile.

Heuristic Functions
"Domain Rules of Thumb"



- Quick to Compute, Approximate measure of goodness, used to guide search methods.
  - ➢ Checkers: My Pieces/Your Pieces
  - ➢ 8-Puzzle: How many tiles in place?
  - ➢ Hanoi: How many disks on right peg?

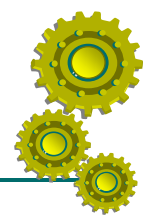
Heuristic Search



- Is there an easy function which (approximately) evaluates states with respect to their "goodness", or nearness to goal?

- Why Approximate?
  - Because an exact measure means either problem is trivial, or function isn't easy!
- If so, can we use this information to speed up search?
  - Theoretically, No; Practically, Yes!

Best first: Use a HEAP!



- DFS expands states onto a stack
- BFS expands states onto a queue
- Bestfs expands states in sorted order
  - usually using a heap...