ALGORITHMS THAT BACKTRACK

COMP 210 – 04 NOV 2005
(P)REVIEW

- All the way back to Lecture 14 (9/28)
- Descendant family trees
blue-eyed-descendant?

Data definition:

(define-struct parent (children name date eyes))

;; A parent is (always!) a structure:
;; (make-parent loc n d e)
;; where loc is a list of children, n and e
;; are symbols, and d is a number.

;; A list-of-children is either
;; 1. empty or
;; 2. (cons p loc) where p is a parent and loc is
;;    a list of children.
blue-eyed-descendant?

;; blue-eyed-descendant? : parent -> boolean
;; to determine whether a-parent any of the descendants (children, ;; grandchildren, and so on) have 'blue in the eyes field
(define (blue-eyed-descendant? a-parent)
  (cond
   [(symbol=? (parent-eyes a-parent) 'blue) true]
   [else (blue-eyed-children? (parent-children a-parent))])))

;; blue-eyed-children? : list-of-children -> boolean
;; to determine whether any of the structures in alloc is blue-eyed ;; or has any blue-eyed descendant
(define (blue-eyed-children? alloc)
  (cond
   [(empty? alloc) false]
   [(blue-eyed-descendant? (first alloc)) true]
   [else (blue-eyed-children? (rest alloc))])))
The book devotes a whole section to this.

A common technique when searching trees:
1. Go down one branch
2. If you don’t find the answer, go down the next branch

This applies to a more general class of tree-like structures, called
GRAPHS

[YOU’LL SEE THESE AGAIN AND AGAIN]
DIRECTED GRAPHS, FORMALLY

- A directed graph $G = \{ V, E \}$
  - $V$: a set of vertices
  - $E$: a set of edges

- An edge is a pair of vertices $\{ V_1, V_2 \}$
  - The edge connects $V_1$ to $V_2$

- We interpret these sets as a picture in which vertices are connected to one another by edges
Trees are graphs, too

With the added restriction that each vertex may have exactly one edge leading to it

We call the number of inbound edges “in-degree”, so trees are directed graphs of in-degree 1
EXAMPLES OF DIRECTED GRAPHS

- The Web: a page has potentially many links to another page
- The Internet: computers connected to other computers (it seems like it might be undirected, but consider a firewall: things can go out, but not back in)
- Downtown Houston: one-way streets, and some streets don’t connect
- Facebook, MySpace, Friendster, Orkut, etc. (linking people to each other, in a DIRECTED fashion)
REPRESENTING GRAPHS

- We choose Scheme lists
- A node ("vertex") is a symbol, like 'A
- A graph is
  - a list of
  - (list node (listof nodes))
  - We call this an "associative list"
- The (listof nodes) represents the nodes reachable from that node
(define Graph
  (list
    (list 'A (list 'B 'C))
    (list 'B (list 'C))
    (list 'C empty)
    (list 'D empty))))
PROBLEM: ROUTE SEARCH

- We want to find a route from one node to another.
  - (Maybe this is a maze in which you have a starting point, a number of one-way paths, and a goal.)
EXAMPLE

- LIFE -

♥♥♥♥♥

YOU

TREASURE

A
THIS TIME, IN SCHEME

(define Graph
  ’[(A (C F))
    (B (E))
    (C (D B))
    (D (F))
    (E ())
    (F (I))
    (G ())
    (H (G))
    (I (H))]])
OUR GOAL: find-route

;; find-route : node node graph -> [node]
;; find a path from a to b in graph g
(define (find-route a b g) ...)

;; examples
(find-route 'A 'A Graph)
=> (list 'A)
(find-route 'A 'B Graph)
=> (list 'A 'C 'B)
(find-route ‘D ‘A G)
=> ?

We need to expand our function’s return type slightly to encode this
UPDATED: find-route

;; find-route : node node graph -> [node] or false
;; find a path from a to b in graph g
;; if no path exists, returns false
(define (find-route a b g) ...)
SOLVING A RECURSIVE PROBLEM

1. What’s the trivial problem (the one we know how to solve right away)?
2. What’s the trivial problem’s solution?
3. How do we break a non-trivial problem up into smaller problems?
4. How do we combine the results?
1. The trivial problem:
   if (symbol=? a b), we’re done.

2. The path in this case is
   (list b).

3. Otherwise,
   inspect each neighbor of a and see if there exists
   a path to b from it.

4. If we do find a path from a neighbor,
   prepend our current node (cons a path) and return.
FIRST ATTEMPT: find-route

;; find-route : node node graph -> [node] or false

(define (find-route a b g)
  (cond
    [(symbol=? a b) (list b)]
    [else ... now what?])

find-route (2)

;; find-route : node node graph -> [node] or false
(define (find-route a b g)
  (cond
    [(symbol=? a b) (list b)]
    [else (local
            [(define possible-route
               (find-route/list (neighbors a g) b g))]
            (cond
              [(cons? possible-route)
               (cons a possible-route)]
              [else false]))])))

TODO: write find-route/list and neighbors
We said that, given a list of nodes, it should find a path (if it exists) from any of them.

This is just like (blue-eyed-children?), remember?

We had (blue-eyed-descendant?) for one ftn, but needed a helper to look through a list of children.

;;; blue-eyed-descendant? : parent -> boolean
(define (blue-eyed-descendant? a-parent) ...)

;;; blue-eyed-children? : list-of-children -> boolean
(define (blue-eyed-children? aloc) ...)
find-route/list (2)

;; find-route/list : [node] node graph -> [node] or false
;; finds the route in g, if it exists, from some node in l
;; to b; if no path exists, returns false
(define (find-route/list l b g)
  (cond
   [(empty? l) false]
   [else ... (find-route (first l) b g) ... ...
   ... (find-route/list (rest l) b g) ... ]))
(define (find-route/list l b g)
  (cond
   [(empty? l) false]
   [else (local
          [(define possible-route
             (find-route (first l) b g))]
            (cond
             [(cons? possible-route) possible-route]
             [else (find-route/list (rest l) b g)]))])))
(define (neighbors n g)
  (cond
    [(empty? g) (error 'neighbors "Not in graph!")]
    [else (cond
      [(symbol=? n (first (first g)))
        (second (first g))]
      [else (neighbors n (rest g)))])))
TIME EXTENDED!

➤ Seriously, we have time left over?

\[
\text{(cond}
\,
\begin{align*}
&\text{[(find-routes-in-cyclic-graphs?) (go)]} \\
&\text{[(learn-about-associative-lists?) (go)]}
\end{align*}
\]

\]
ASSOCIATIVE LISTS

▶ These things are fun
▶ Use them to organize data by “name”
▶ Type: [(list X ?)]
▶ Example:

   (define too-many-dans (list
       (list ‘dsandler “Dan Sandler”)
       (list ‘dlsmith “Dan Smith”)
       (list ‘danvk “Dan Vanderkam”)))
FUNCTIONS FOR ASSOCIATIVE LISTS

You could write your own, like (neighbors), but Scheme gives us the most abstract one:

;; assf : (X -> boolean) [(list X ...)] -> ?
;; (an unfortunate name)
;; if there exists a (list x ...) in the associative list
;; al return the second of the list; otherwise false
(define (assf func al)
  (cond
   [(empty? al) false]
   [else (cond
            [((func (first (first al))) (second (first al))]  
                [else (assf func (rest al)))])))))
(assf (lambda (x) (symbol=? x 'dsandler))
    too-many-dans)
⇒ "Dan Sandler"

(assf (lambda (x) (symbol=? x 'dwallach))
    too-many-dans)
⇒ false

There are others, too
- ...shorthands for “look in the assoc. for something ‘equal’ to x”
- To define these requires knowledge of Scheme’s weird equivalence functions
- (Of these, you’ve probably already seen equal? … it gets weirder from there)
BACK TO GRAPHS

How would we write (neighbors) with assf?

; neighbors : node graph -> [node]
(define (neighbors n g)
  (assf (lambda (x) (symbol=? x n)) g))

(Easy!)
ONE LAST NOTE

Prof. Taha points out: “If you know the entire graph ahead of time, why not just write that into the function?”

(define (graph1-neighbors n)
  (cond
   [(symbol=? n ‘A) (B C)]
   [(symbol=? n ‘B) (C)]
   [(symbol=? n ‘C) ()]
   [(symbol=? n ‘D) ()]
   [else (error …)])
)

Each new (?-neighbors) function you write represents a different graph.

Our graph data definition becomes a function. Crazy!
GRAPHS WITH CYCLES

➢ We’re time-travelling to next week’s lectures, now

➢ If we ran (find-route) on a cyclic directed graph, what might happen?
  ➢ Try it.

➢ How does this violate the recursive algorithm design?
  ➢ Problem doesn’t necessarily get smaller at every step!
I DON’T NEED TO WALK AROUND IN CIRCLES

If only we had some way to remember which nodes we’ve already seen...

Maybe we can pass that information from function call to function call.

We call this kind of recursion “accumulation”—we’re accumulating data as we dig deeper into the problem, as well as potentially creating data on our way back “out”
ACCUMULATION: A CRASH COURSE

- Old-school:
  ; sum: [num] -> num
  (define (sum l)
    (cond
      [(empty? l) 0]
      [else
        (+ (first l)
          (sum (rest l)))]))

  (sum (list 1 2 3 4)) => 10

- New-school:
  ; asum: [num] num -> num
  (define (asum l a)
    (cond
      [(empty? l) a]
      [else
        (+ (first l)
          (asum (rest l) a)]]))

  (asum (list 1 2 3 4) 0) => 10
ACCUMULATING A LIST OF "SEEN" NODES

```
(define (route2 a b g seen)
  (cond
    [(symbol=? a b) (list a)]
    [(in-list? a seen) false]
    [else (local
      [(define possible-route
          (route2/list (neighbors a g) b g (cons a seen)))
        (cond
          [(cons? possible-route) (cons a possible-route)]
          [else false]))])))

(define (route2/list l b g seen)
  (cond
    [(empty? l) false]
    [else (local
      [(define possible-route (route2 (first l) b g seen))
        (cond
          [(cons? possible-route) possible-route]
          [else (route2/list (rest l) b g seen)))]))]
```
TESTING OUR NEW FUNCTION

(define G
  '[(A (B C D))
    (B (C D))
    (C (D))
    (D (E G))
    (E (A))
    (F ())
    (G ())])

> (route 'E 'G G)

…

user break

> (route2 'E 'G G empty)

(list 'E 'A 'B 'C 'D 'G)
= FIN =