COMP 210, FALL 2000
Lecture 25: Graphs, Paths, and Search

Reminders:

1. No homework due Wednesday. Next homework will be available this Wednesday.
2. Go to lab this week. It will discuss several things that are critical for the Missionaries and Cannibals homework.

Review

1. We revisited Wednesday's lecture and got it right. Then we went forward and talked about termination conditions. A critical point was that every generative recursion program must contain a termination argument after the Contract and Purpose. We saw several examples of termination arguments.

Another Kind of Problem–-a graph problem

JetSet Air (remember them from lecture six?) was so successful with their computerized system for keeping maintenance records that they want to develop a similar system to manage information about the various routes that they fly. For each city that JetSet serves, it needs, at a minimum, a way of determining the cities that a passenger can reach with a direct flight. That is, for a city *a*, what cities are destinations on flights originating from *a*? Since COMP 210 did such a good job on maintenance, they've asked us to design the data structures and to develop the programs. How will we represent this information?

```
A city is a symbol.

;; The information for a city can be represented as a structure
;;  (make-city-info name dests)
;; where c is a city (symbol) and dests is a list of symbole
(define-struct city (name dests))

;; A route-map is a list of city-info

(define routes
  (list (make-city-info  'Houston   (list 'Dallas 'NewOrleans))
        (make-city-info 'Dallas   (list  'LittleRock  'Memphis))
        (make-city-info  'NewOrleans (list 'Memphis))
        (make-city-info 'Memphis  (list 'Nashville)) ))
```

As a first program, we need a program **find-flights** that consumes a route-map and returns a sequence of cities (not necessarily the shortest sequence) by which we can fly from a starting city to a final city. If no such sequence exists, the program should return **false** (?)

```
;; find-flights: city city route-map → (list of city) or false
;; Purpose: create a path of flights from start to finish or return false
(define (find-flights start finish  rm) …)
```

Examples:

```
(find-flights  ‘Houston ‘Houston routes)
= (list ‘Houston)

(find-flights ‘Houston ‘Dallas)
= (list ‘Houston ‘Dallas)

(find-flights ‘Dallas ‘Nashville)
= (list ‘Dallas ‘LittleRock ‘Memphis ‘Nashville)
```

How would we write find-flights? If there is a direct flight from start to finish, the route is trivial, as is the program. All we need to do is to walk the list-of-city in the city-info structure for **start** and find **finish**. What if there is no direct flight? The list-of-city in the city-info structure for start gives us all the cities that we can reach in one flight (one hop). We can look through the city-info for the final city–-trying to find a two-hop solution. If that fails, we can look through those two-hop cities for a three-hop flight, and …

Based on this description, we should be able to write **find-flight**. Is this a problem for structural recursion, or for generative recursion? Hint: the answer is **generative**. If there is no direct flight between start and finish, we generate new problems–-flying from the cities that are reachable to finish. These new problems are based on our understanding of how to search for a path through the route map. (To be sure, they rely on information in the route map, but we don’t run over the whole route map in some structurally determined order. Instead, we search outward from start, looking for finish.)

Since the program needs generative recursion, we need to answer the questions that derive from the generative recursion template.

1. What is the trivial case? When start = finish.
2. What is the solution to the trivial case? A list containing start.
3. How do we generate new problems?

   Find all the cities that are destinations from start, and look for a route from one of those cities to finish. (Recur on same route map and finish, new start).
4. How do we combine the solutions?

   If we find at least one route, we keep one and add the starting city to that route. Otherwise, we return false.

Let's fill in the code…

```
;; find-flights: city city route-map → (list of city) or false
;; Purpose: create a path of flights from start to finish or return false
(define (find-flights start finish  rm)
  (cond
    [(symbol=? start finish) (list start)]
    [(else
         (local [(define possible-route
                      (find-flights-for-list (direct-cities start rm) finish rm))]
                  (cond
                        [(boolean? possible-route)  false]
                        [else  (cons start possible-route)])) ] ))


;; direct-cities: city route-map → list-of-city
;; Purpose: return a list of all cities in the route map with direct flights from
;;         the city given as an argument
(define (direct-cities  from-city  rm)
  (local [(define  from-city-info
                  (filter (lambda (c ) (symbol=? (city-info-name c) from-city)) rm))]
          (cond
             [(empty? from-city-info)  empty]
             [else (city-info-dests (first (from-city-info))])))

;; find-flights-for-list: list-of-city city route-map → list-of-city or false
;; Purpose: finds a flight route from some city in the input list to the destination,
;;         or returns false if no such route can be found.
(define (find-flights-for-list aloc finish rm)
  (cond
     [(empty? aloc)  false]
     [else
        (local [(define possible-route
                   (find-flights (first aloc) finish rm))]
             (cond
                  [(boolean? possible-route)
                   (find-flights-for-list (rest aloc) finish rm)]
                  [else  possible-route]))]))
```

How does this program work? It employs a common algorithmic technique called *backtracking*. It tries a potential solution. If that solution does not work, we go back and try another possible solution, and another, and another, until one of two things happens. Either we find a solution, or we exhaust the possibilities.

What's the termination argument for find-flights? Each recursive call looks for a route that uses fewer flights. Eventually, each path must end in the finish city, or the city has no outbound flights. [This is an oddity of the way we have formulated the route map, but bear with me for a day or two.]

This program works fine on our initial route map. (*Time permitting, work an example from page 1.*) What if we add a flight from ‘Dallas to ‘Houston?

```
(define new-routes
  (list (make-city-info  ‘Houston   (list ‘Dallas ‘NewOrleans))
        (make-city-info ‘Dallas   (list  ‘Houston ‘LittleRock  ‘Memphis))
        (make-city-info  ‘NewOrleans (list ‘Memphis))
        (make-city-info ‘Memphis  (list ‘Nashville)) ))
```

What happens when we try

```
(find-flights ‘Houston ‘Memphis new-routes)  ?
```

Let’s write down the series of calls that occur.

```
(find-flights ‘Houston ‘Memphis new-routes)
    (find-flights-for-list (list ‘Dallas ‘NewOrleans) ‘Memphis new-routes)
    (find-flights ‘Dallas ‘Memphis new-routes)
    (find-flights (list ‘Houston ‘LittleRock ‘Memphis) ‘Memphis new-routes)
    (find-flights ‘Houston ‘Memphis new-routes)
    … and so on for quite a while …
```

We ended up with a non-terminating evaluation (or an infinite loop). What happened? First, our termination condition is wrong. It assumes that the route-map has no cycles–ways that we can fly from a to b and from b to a. In our new route-map, we have a cycle (‘Houston to ‘Dallas and ‘Dallas to ‘Houston). This clearly causes a major problem with the program. Thus, our original program only works on route-maps that have no cycles (or loops, or strongly-connected components, or …)

Why does it break when it confronts a cycle? Because it has no recollection as to which cities it has already tried. Each recursive call is independent of all the others. If the program is to operate correctly on route-maps (or *s*) that have cycles (called *cyclic graphs*), it will need to remember all of the cities that it has already tried (or *visited*)

Next class, we’ll see how to do that.