

## COMP 210, FALL 2000

### Lecture 25: Graphs, Paths, and Search

#### Reminders:

1. No homework due Wednesday. Next homework will be available this Wednesday.

#### Review

1. We 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.
2. We started looking at the route discovery problem for JetSet air. It is an example of a backtracking problem that will require an *accumulator*. (This lecture's subject)

#### Another Kind of Problem--a graph problem

See lecture notes for Lecture 24 for the data definitions and the code.

How does **find-flights** 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.

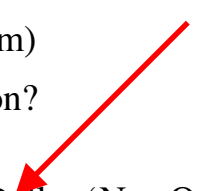
*What's the termination argument for find-flights?* Each recursive call looks for a route that uses fewer flights. Eventually, each path must end in the finish city, or the city has no outbound flights. [This is an oddity of the way we have formulated the route map, but bear with me for a day or two.]

This program works fine on our initial route map.

```
Work (find-flights 'Houston 'Memphis rm)
```

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*)

One way to handle this problem is to add a new parameter to find-flights that stores the cities already visited (as a list, naturally). Then, find-flights can check the list of already visited cities to avoid redoing work (and hitting a case that causes an infinite recursion).

That modification looks like -- *see the slide* --

What should the initial argument passed to **visited** be? It *must* be **empty**. Passing it other values can cause the program to malfunction. For example, if you started it with a list of all the cities in the route map, it would never find any routes except the trivial ones.

Using this on `routes2`, we find that it terminates without hitting the infinite recursion—our addition of memory allowed it to prune its search when it started to run over parts of the graph that it had already visited.

This also produces a much simpler termination condition. Because it knows about its own history, `find-flights` will only search outward from a given city once. Thus, as long as the route map is finite, the search will terminate.

[Simple, beautiful termination argument!]

So what is this parameter **visited**? In COMP 210, we call this kind of parameter an *accumulator*. It accumulates information over the course of the computation and lets the program have access to that information. In effect, it provides the function with a record of where it has been (and, perhaps, the results of some of those earlier computations). The next several classes will look at aspects of designing programs with accumulators. You should also read the material on accumulators in the book.

There is one distasteful aspect of the way that we used **visited** to fix **find-flights**. To cure what was, in essence, a design flaw in the program, we changed its contract by adding an extra parameter. Can we avoid this problem? Certainly. We can make the new version of `find-flights` be a helper function with a new name, such as **fixed-find-flights**, and rewrite **find-flights** so that it simply invokes **fixed-find-flights** with the extra argument. This arrangement has the advantage that it allows us to ensure the correct initial value for the parameter **visited**.

This is a great example of a place where we can use `local` to hide the entire mess. We can rewrite **find-flights** so that it defines **fixed-find-flights**, **direct-cities**, and **find-flights-for-list** inside a `local`. What parameters can we elide at that point? (At least the route map and the destination city!)

### Another Example

Let's write a program **reverse** that consumes a list (of alpha) and produces a list (of alpha) that has the same elements in the reverse order. That is, the first element of the input becomes the last element of the output. Again, we'll start with a version based on structural recursion.

```
;; reverse: list-of-alpha → list-of-alpha
;; Purpose: constructs the reverse of a list of items
(define (reverse aloa)
  (cond
    [(empty? aloa) empty]
    [(cons? aloa)
     (make-last-item (first aloa) (reverse (rest aloa)))])))
```

```

;; make-last-item: alpha list-of-alpha → list-of-alpha
;; Purpose: adds an element to the end of a list
(define (make-last-item an-elt aloe)
  (cond
    [(empty? aloe) (list an-elt)]
    [(cons? aloe) (cons (first aloe)
                        (make-last-item an-elt (rest aloe)))]))

```

What happens on a call to reverse?

```

(reverse (list 1 2 3))
= (make-last-item 1 (reverse (list 2 3)))
= (make-last-item 1 (make-last-item 2 (reverse (list 3))))
= (make-last-item 1 (make-last-item 2 (make-last-item 3 (reverse
empty))))
= (make-last-item 1 (make-last-item 2 (make-last-item 2 empty)))
... and then it starts returning...

```

Again, to process all of these nested calls to make-last item, we will end up traversing the end of the original list many times. This begins to look like the last example, right down to the fact that it seems to waste a lot of computation.

Can we use an accumulator to simplify the program? Compare the structural version of available-days to the structural version of reverse. Notice that they both pass the value returned by a recursive call to another recursive procedure. This is precisely what gives rise to the kind of quadratic behavior that we observed when we hand evaluated the examples. It gives rise to a simple rule for when to consider using an accumulator.

**Consider using an accumulator if the program processes the return value of a recursive call with another recursive call**

Next lecture, we'll look at a process for transforming a program based on structural recursion into one that uses an accumulator, *provided that the original program fits our rule.*