Administrative Notes

Exam

- Due today at 5 PM
- Hand in to my office, DH 2065

Homework

- Available this afternoon

Graph Problems

Remember JetSet Air?

- Need software for route management
- First problem:
  - → For each city, where do the direct flights go?
  - → This is a data-structures problem

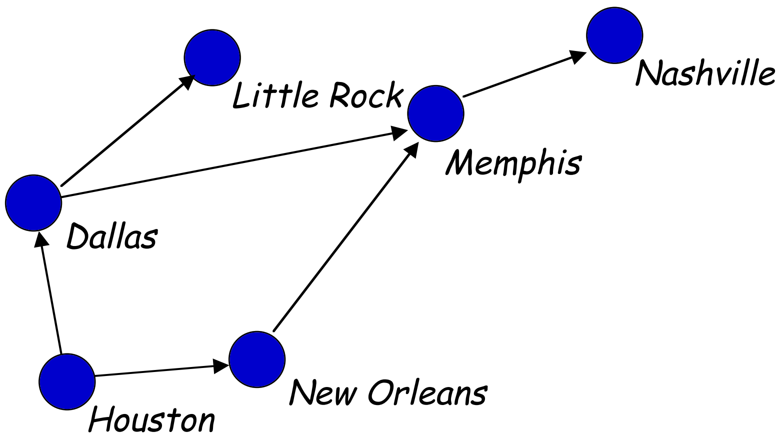
```
;; a city is a symbol

;; The information for a city is a structure
;;  (make-city-info  name  dests)
;; where name is a city and dests is a list of cities
(define-struct city-info (name dests))

;; a route-map is a list of city-info
;; We will use Scheme's built-in implementation of lists
```

Graph Problems

```
;; Example Route Map
(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))
   ))
```



This encodes the answers for direct flights

Graph Problems

Develop a program find-flights

```
;; find-flights:  city  city  route-map -> list of city
;; Purpose: create a path of flights from start to finish.
;;           Returns an empty list if no route is found.
(define (find-flights start finish  rm)  …  )

;; Examples
(find-flights  'Houston  'Houston  routes) ⇒ (list 'Houston)

(find-flights  'Houston  'Dallas routes) ⇒  (list  'Houston  'Dallas)

(find-flights  'Dallas  'Nashville routes)
            ⇒ (list 'Dallas 'Memphis 'Nashville)
```

How do we fill in the body of find-flights ?

Find-flights

Direct flights are easy

1. Look through the route map for start city
2. Walk its dests list looking for finish city
3. Report success or failure

What about multi-hop routes?

1. List of city gives us direct flights
2. Look through dests list of those cities for two-hop sol'ns
3. Look through their dests list for three-hop sol'ns
4. Look through …

This begins to look like generative recursion …

Generative Recursion

List of helpful questions

1. What is the trivial case? start = finish
   - → How do we identify it? start = finish
   - → How do we solve it? (list start) or (list finish)
2. How do we generate subproblems?
   - → How many should we generate?
   - → How do we generate them?

   Find direct-cities of start, look for route from one of those to finish (recur)
3. Does the subproblem solution solve the original problem? no
4. Must we combine solutions from multiple subproblems? yes
   - → How do we combine them? add them to the list

Graph Problems

Subtask1 : Develop direct cities

```
;; direct-cities : city route-map -> list of city
;; Purpose: returns the list of all cities that can be reached from the
;;           argument city by ac direct flight, according to the route map
;;           Returns empty if no such flights exist.
(define (direct-cities from-city rm)
    (local [(define from-city-dests
                 (filter (lambda (city) (symbol=? (city-info-name city) from-city))
                         rm))]
        (cond
           [(empty? from-city-dests) empty]
           [else (city-info-dests (first from-city-dests))])
     ))
```

Find-flights

Subtask 2: Develop find-flights

```
;; find-flights: city city route-map -> list of city
;; Purpose: ...
(define (find-flights start finish rm)
    (cond
       [(symbol=? start finish) (list start)]     ;; trivial case from questions
       [else
           (local [(define possible-route
                         (find-flights-for-list (direct-cities start rm) finish rm))]
                   (cond
                      [(empty? possible-route)  empty]
                      [else  (cons start possible-route)]
                   )
            )
       ]
    ))
```

Find-flights

And, find-flights-for-list

```
;; find-flights-for-list: list-of-city city route-map -> list of city
;; Purpose: finds a route from some city in the argument list to the
;;           city given as the singleton argument, using the route map
(define (find-flights-for-list aloc finish rm)
   (cond
     [(empty? aloc)   empty]
     [else
      (local [(define one-route (find-flights (first aloc) finish rm))]
             (cond
               [(empty? one-route) (find-flights-for-list (rest aloc) finish rm)]
               [else  one-route] )) ]
   ))
```

Does this work? To DrScheme !

Find-flights

Termination argument?

- Direct-cities uses filter, but no other recursion
- Find-flights either:
  - → Discovers that start = finish and returns (list start), or
  - → Invokes find-flights-for-list on start's neighbors in rm
- Find-flights-for-list
  - → Follows the list template
  - → Uses find-flights to check (first aloc)
  - → Recurs on find-flights-for-list

This is the recursion that we must consider

Find-flights

Termination argument

Find-flights-for-list takes a destination list for a single city and tries to locate a flight to finish from one of those cities. That destination list is finite, so it invokes find-flight a limited number of times. It stops when it hits the end of the list or when it finds a route.

Find-flight checks the trivial case, then uses find-flights-for-list on (direct-cities start rm).

Because the route-map is finite, this process eventually halts—either it finds a route, or it runs out of flights to check.

This argument relies on a peculiar property of our route-map: it contains no cycles.

(How do those planes get from Little Rock back to Houston?)

Find-flights

What happens if we add a cycle?

- Add a Dallas to Houston flight
- Now, (find-flights 'Houston 'Nashville new-routes) recurs indefinitely. (almost always a bad thing)

What's the real problem?

- Find-flights and find-flights-for-list have no history
  - → Those who ignore the past are doomed to repeat it
- Need to give them some institutional memory
  - → Add a parameter that contains cities already tested

Find-flights, take 2

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

Find-flights, take 2

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

Find-flights, take 2

Terminaton argument

This version of find-flights has the same pattern of recurrences as the original, except that it cannot recur on a city that is in the visited list.

Thus, it searches outward from each city exactly once. Since the number of cities is finite (and small), this terminates. It cannot recur indefinitely.

Rather than relying on the absence of cycles, it relies on the finite nature of the route map—much more sensible.

So, what is "visited"?

- We used "visited" to accumulate information
  - → Gathered over course of computation
  - → Used to ensure correct behavior
- We call such a parameter an *accumulator*

Next several classes

- We will see more examples of accumulators
  - → Keeping track of the past
  - → Preserving results from the past