Exploring Networks

This tutorial works through the solution to a common kind of problems – navigating a network where the links are described by a set of tuples. To start off with we will use a simple structure as follows where the tuple (a b) means that there is a link from a to b (but not necessarily that there is a link from b to a).

Check the following example of a map...

```
(defparameter map
  '((newcastle middlesbrough)
    (middlesbrough saltburn)
    (middlesbrough newcastle)
    (newcastle durham)
    (saltburn marske)
    (morpeth newcastle)
    ))
```

It would be possible to do this using a search mechanism in which case all we would have to do is build a legal move generator. For the map data above a legal move generator could be specified as follows...

```
(defun lmg1 (place)
  ;; assumes map is special/global
  (foreach ( `(,place ?goes-to) map)
           #?goes-to))

> (lmg1 'middlesbrough)  →  (saltburn newcastle)
> (lmg1 'newcastle)  →  (middlesbrough durham)

> (breadth-search 'morpeth 'marske #'lmg1)
(marske saltburn middlesbrough newcastle morpeth)
```

An alternative approach (ie: not using search) is to write a function which does the network exploration itself. An important difference between networks and trees is that networks can contain circular paths, the following sets of tuples show various types of circularity...

1. (a a) - a node links to itself
2. (a b) (b a) - two nodes link in both directions
3. (a b) (b c) (c a) - circularity occurs over 3 or more nodes

If our algorithm allows itself to cycle round a circular path it may never stop, so we need to avoid this. One (neat) way of avoiding circular paths is to keep track of any places/nodes that have been visited & check that they are not visited again. The following function does this in two ways...

1. it keeps track of everywhere it has visited by placing visited nodes in a variable called path
2. it checks to see if a node is in the previously visited path before going to that node again (& only goes to it if it has not been visited)

Notes:

(i) when the function starts the only node that has been visited is the start node

(ii) the function returns the path

(iii) the function does not necessarily return the shortest path that exists between start & goal nodes

(iv) the greyed section of the cond statement is optional. It may increase efficiency in some cases

```
(defun route (data start goal &optional (path (list start)))
  (cond ((eq start goal)
         path
         )
        ((member `(,start ,goal) data :test #'equal)     ;
         (cons goal path)                                ;
         )                                               ;
        (t
         (foreach ( `(,start ?next) data)
                  (unless (member #?next path)
                    (route data #?next goal (cons #?next path))))
         )))

> (route map 'newcastle 'marske)
(((marske saltburn middlesbrough newcastle)))

> (route map 'morpeth 'marske)
((((marske saltburn middlesbrough newcastle morpeth))))
```

Notice the level of nesting seen in the results. This can be corrected by using ***foreach@*** in place of ***foreach*** which limits the amount of nesting (check matcher documentation for details). The next solution looses the optional cond statement & uses ***if*** instead...

```
(defun route (data start goal &optional (path (list start)))
  (if (eq start goal)
      path
    (foreach@ ( `(,start ?next) data)
         (unless (member #?next path)
           (route data #?next goal (cons #?next path))))
    ))

> (route map 'newcastle 'marske)
(marske saltburn middlesbrough newcastle)
```

Based on this function we will add some detail to our tuples (i) a tuple relation name and (ii) a measure of distance...

```
(defparameter map
  '((distance newcastle middlesbrough 35)
    (distance middlesbrough saltburn 10)
    (distance middlesbrough newcastle 35)
    (distance newcastle durham 15)
    (distance saltburn marske 5)
    (distance morpeth newcastle 15)
    ))
```

To deal with the new tuple structure we need a different pattern in the foreach form, something like...

```
`(distance ,start ?next ?d)
```

The function below uses this pattern & accumulates the distance travelled in an optional arg called "miles". This distance is returned along with the path when a route is found.

```
(defun route (data start goal
                  &optional (path (list start)) (miles 0))
  (if (eq start goal)
      (cons miles path)
    (foreach@ ( `(distance ,start ?next ?d) data)
        (unless (member #?next path)
          (route data #?next goal
                 (cons #?next path) (+ #?d miles))))
         ))

> (route map 'newcastle 'marske)
(50 marske saltburn middlesbrough newcastle)

> (route map 'newcastle 'lilongwe)
nil
```

In the function above, the distance is held separately from the path and only put onto the path when the result is returned. In the session when we discussed these functions an alternative solution was suggested where the distance is held as part of the path (it is always the first item in the path list). In this function the path starts off being set as a 0 distance and the start state (see the specification of the optional path argument). This distance is then modified at each recursive call of the function.

```
(defun route (data start goal &optional (path (list 0 start)))
  (if (eq start goal)
      path
    (foreach@ ( `(distance ,start ?next ?d) data)
        (unless (member #?next path)
          (route data #?next goal
                 (cons (+ (first path) #?d)
                       (cons #?next (rest path)))))
             )))

> (route map 'newcastle 'marske)
(50 marske saltburn middlesbrough newcastle)

> (route map 'newcastle 'lilongwe)
nil
```