Legal Move Generation & Search – a practical guide

Search routines are used to find paths between start and goal states. Search routines use legal move generators (LMGs) to generate possible successor states (ie: those states that can be reached in one move from some current state).

example-1 – a numbers puzzle

With this puzzle you have to work out how to get from one number to another by using a small number of simple (mathematical rules). For example... how can you get from 5 to 159 with the following three rules...

1. you can multiply by 3
2. add 7
3. subtract 2

You can find a search routine on the agent-domain web site but need to write your own LMG. LMGs are written as functions which take one state as their arg and produces a list of successor states, eg:

```
(defun lmg1 (n)
  (list (* n 3)
        (+ n 7)
        (- n 2)
        ))

> (lmg1 5)  ==>  (15 12 3)
> (lmg1 123)  ==>  (369 130 121)

> (breadth-search 5 159 #'lmg1)
==> (159 53 46 39 13 15 5)
```

Notice a couple of things...

- breadth-search takes 3 arguments: a start state, a goal state & a LMG
- it returns the path in reverse (ie: the route backwards from the goal – this is not a feature of search but just the way breadth-search happens to work)
- to pass a function as an argument (the LMG in this case) we need a bit of extra syntax #' which we read as "the function named..."

example-2 – route finding

Before we get on to the example problem we need a couple of new features of Lisp. The **utils.cl** file defines a few functions for handling sets and association lists, the function called "**->**" (pronounced "lookup") retrieves values from an association list. An association list is a symbolically indexed collection

```
(defvar english '((one 1)(two 2)(three 3)(four 4)))

> (-> english 'two)
(2)
> (-> english 'four)
(4)
```

S.C.Lynch@tees.ac.uk
www.agent-domain.org

a note about global variables

The terms used to discuss variables & assignment is often a little different in Lisp & some other functional languages. Lisp talks about variables with lexical and dynamic scope which are roughly equivalent to local and global variables in other languages. We can declare (and assign an initial value to) a global variable in Lisp using ***defvar*** or ***defparameter***. It is probably better to use ***defparameter*** because you can edit and recompile and initialisation value & it will change – you cannot do this with ***defvar*** declarations which can only be changed using ***setf*** (a kind of assignment form). In the Lisp notes we supply in this collection of documents we will use both ***defvar*** and ***defparameter*** from time to time.

By convention Lisp programmers put ***** at the start & end of global variable names so would spell a global variable ****towns**** rather than ***towns***. We will do this most of the time.

a dot (full-stop, period) is used to construct a pair (think about linked lists)

```
(defvar *hindi* '((ek . 1)(do . 2)(tin . 3)(char . 4))))

> (-> *hindi* 'tin)
3
```

association lists can be nested

```
(defvar *country*
'((Africa
   (Botswana (Capital . Gaborone) (Population . 1.5))
   (Zimbabwe (Capital . Harare)   (Population . 11)))
  (Asia
   (India   (Capital . New-Delhi) (Population . 980))
   (Sri-Lanka (Capital . Colombo) (Population . 15))
  )))

> (-> *country* 'Africa)
((Botswana (Capital . Gaborone) (Population . 1.5))
 (Zimbabwe (Capital . Harare) (Population . 11)))

> (-> *country* 'Africa 'Botswana)
((Capital . Gaborone) (Population . 1.5))

> (-> *country* 'Africa 'Botswana 'Capital)
Gaborone
```

Back to the route finding problem. We want to find a route of interconnecting flights from one airport to another given some representation of which airports have direct connecting flights to which other airports. We can conveniently specify direct connections using a data structure which is an association list (even though it might not seem like an association l,ist when you first look at it!). We can then use this to build a LMG.

The association list...

```
(defvar *flights*)
(setf *flights*
  '((teesside  amsterdam dublin heathrow)
    (amsterdam dublin teesside joburg delhi dubai)
    (delhi     calcutta mumbai chennai)
    (calcutta  mumbai kathmandu)
    (mumbai    chennai delhi dubai)
    (chennai   colombo)
    (dubai     delhi colombo joburg)
    ))

> (-> *flights* 'delhi)
(calcutta mumbai chennai)

> (-> *flights* 'amsterdam)
(dublin teesside joburg delhi dubai)
```

Notice that the call to `(-> *flights* airport)` works a bit like an LMG – it takes one state and generates successor states. Wrapping this call to lookup as a function definition provides a LMG...

```
(defun connections (airport)
  (-> *flights* airport))

> (connections 'delhi)
(calcutta mumbai chennai)

> (breadth-search 'teesside 'colombo #'connections)
(teesside amsterdam dubai colombo)
```