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:

```
(defn lmg1 [n]
  (list (+ n 3) (* n 2) (- n 7)))

user=> (breadth-search 5 35 lmg1)
(5 8 16 32 35)

user=> (time (breadth-search 5 1357 lmg1))
"Elapsed time: 847.73328 msecs"
(5 8 11 22 44 88 176 169 338 341 682 1364 1357)
```

Notice a couple of things...

- breadth-search takes 3 arguments: a start state, a goal state & a LMG
- functions get passed as arguments just like any other data

example-2 – route finding

For the next problem we will use maps – symbolically indexed collections.

```
(def english
  '{one 1, two 2, three 3, four 4})

user=> (english 'two)     ; using a map as a fn
2

user=> ('two english)     ; using a symbol as a fn
2
```

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 Lisps & some other functional languages. Lisps talks about variables with lexical and dynamic scope which are roughly equivalent to local and global variables in other languages.

Clojure makes significant use of immutable variables (ie: their values cannot be changed) this limits problems with multiple concurrent access (and other problems!). It is possible to re-def variables but it is bad practice.

note that maps can be nested

```
(def 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}
     }})

user=> ('zimbabwe ('africa country))
{capital harare, population 11}

user=> ('population ('zimbabwe ('africa country)))
11
```

As we progress with the module we will use keyword (prefixed ":" and in a global name-space) for map keys, so our country data structure would be...

```
(def country2
  '{: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}
     }})
```

Keywords in place of quoted symbols in the examples above...

```
user=> (:population (:zimbabwe (:africa country2)))
11
```

...but they are handled a little differently when used as functions. This means that we can thread them more easily than quoted symbols...

```
user=> (-> country2 :africa :zimbabwe :population)
11
```

```
user=> (-> country 'africa 'zimbabwe 'population)
(clojure.core/-> (clojure.core/-> country (quote africa)) (quote
zimbabwe))
user=> (-> country ('africa) ('zimbabwe) ('population))
11
```

Can you explain this? [2 CPs]

More with maps...

```
(def english '{one 1, two 2, three 3, four 4})

user=> (english 'two)
2

; using numbers as an internal representation...

(def hindi '{1 ek, 2 do, 3 tin, 4 char})

user=> (hindi 3)
tin

; writing an english -> hindi translator

(defn english->hindi [sym]
  (hindi (english sym)))

user=> (english->hindi 'three)
tin
```

but... it would be nicer to have all maps the same structure, ie: {word => internal-rep}

```
 (def english '{one 1, two 2, three 3, four 4})
(def hindi   '{ek 1,   do 2, tin 3,   char 4})
(def tamil   '{ondo 1, rendu 2, mundru 3, naalu 4})
```

which requires a "helper fn" to flip map entries when necessary. This is one of those cases where we need use new language features before we are ready to fully analyse their semantics (sorry but 3 CPs for the first person to give a good explanation of flip-map).

```
 (defn flip-map [m]
  (into {} (map (fn [[k v]] {v k}) (seq m))))

user=> (flip-map tamil)
{1 ondo, 2 rendu, 3 mundru, 4 naalu}

(defn translate [src dst sym]
  ((flip-map dst) (src sym)))

; what is (flip-map X)

user=> (translate hindi tamil 'char)
naalu
```

```
user=> (translate hindi tamil 'mango)
nil
```

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 a map. We can then use this to build a LMG...

```
(def 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)
    })

user=> (flights 'delhi)
(calcutta mumbai chennai)
user=> (flights 'amsterdam)
(dublin teesside joburg delhi dubai)
```

Notice that the call `(-> flights airport)` works a bit like an LMG – it takes one state and generates successor states. This means we can use it as an LMG...

```
user=> (breadth-search 'teesside 'colombo flights)
(teesside amsterdam dubai colombo)
```