

## Lab 3 Property Lists and Search

### 1. Property List exercise

The search programs that are in your book use property lists. Here are some exercises that might help you understand property lists.

In Lisp, a symbol can have any number of properties, each of which can have its own value (or property and values). For example, a chair might have a color, weight, height.

To put a property on an item, you use the following notation:

```
-> (setf (get 'chair3 'color) 'blue)  
blue
```

To retrieve a property from an item you use:

```
-> (get 'chair3 'color)  
blue
```

A symbol can have any number of properties. For example, we can specify another property of chair3:

```
-> (setf (get 'chair3 'owner) 'john)
```

The setf function is used, because property items tend to be GLOBAL....So you are always able to get the property of an item.

### 2. Search exercises: There are items that need to be turned in at the end of the Lab

This part of the lab is to help you understand searching.

As previously explained in class, you have several objectives in doing a search:

1. Finding the goal
2. Finding the “best” path to the goal (usually defined as the fewest nodes)
3. Finding the “best” path quickly (usually defined as the number of nodes that needed to be looked at (i.e., expanded)).

So, in addition to finding out how the search algorithms actually work, we also want to understand their “properties” in terms how “good” the search is and how “fast” the search is.

Let’s start with the depth-first. You can find the code (as it appears in the book) on the class web site: [zeus.csci.unt.edu/swigger/csci3210](http://zeus.csci.unt.edu/swigger/csci3210).

**A. Very briefly, the program begins** by placing a property on each of the nodes in the tree so that you can find the children. I have made the search space similar for all the programs so that you can see the differences.

```
(defun addbranches (location branches)
  (setf (get location 'branch) branches))
```

```
(addbranches 'denton '(acity bcity))
(addbranches 'acity '(city dcity))
(addbranches 'bcity '(ecity))
(addbranches 'city '(fcity))
(addbranches 'ecity '(gcity hcity icity))
(addbranches 'gcity '(jcity kcity))
(addbranches 'hcity '(lcity mcity))
(addbranches 'icity '(ncity ocity))
```

For example, if you type:

```
(get 'denton 'branch)
```

Should return -> (acity bcity)

**B. The matcher simply looks at an element** and determines if it is the same:

```
(defun match (element pattern)
  (equal element pattern))
```

So, if you type: (match 'acity 'acity) -> you get t  
(match 'acity 'bcity) -> you get nil

**C. We need a function that will go get the children.** This program does that in the function *morepaths*. But it needs to get the children and then ATTACH THEM TO the list of existing PATHS. (example: the children of 'acity are 'city and 'dcity). They need to be attached to a list SEPARATELY.

```
(defun morepaths (path)
  (mapcar #'(lambda (nextpath) (cons nextpath path))
    (get (car path) 'branch)))
```

Call this function and see what happens. For example, assume that your current path is: ' (acity denton). Type: (**morepaths ' (acity denton)**)

You get the children of acity, but as separate lists. This is what the map function does. It maps the function (lambda (nextpath) (cons nextpath path)) onto the list that it gets from (get (car path) 'branch). In this example, when we take the car of path, we go and find

(get 'acity 'branch), it returns the list '(city dcity). So, the map function applies the cons function to each of the elements in this list (i.e., city dcity) to path one at a time.

**D. Finally, we have the depth-first function.** This is the driver program.

```
(defun depth-first (tree pattern)
  (let* ((paths (list (list tree)))
        (current paths))
    (loop
      (setq current (car paths))
      (cond ((null paths) (return nil))
            ((match (car current) pattern)
             (return (reverse current)))
            (t (setq paths (append (morepaths current) (cdr paths))))))))
```

You can call it with

```
(depth-first 'denton 'ocity) and it will return the path for all the items.
```

After you run the program, modify it by adding print statements so that you can see what is happening. You want to put the print statements just after the loop (see below):

```
(loop
  (setq current (car paths))
  (print 'paths) (princ paths)
  (print 'current) (princ current)
  (cond ((null paths) (return nil)))
```

Now run this program. You should see the paths expanded in a depth first order. For example....Denton's children are: acity and bcity....The program adds these to Denton, then goes after acity's children...acity's children are city and dcity...etc.... If you don't understand this, ask me.

**E. We need to see the path, but we also want to know how many nodes needed to be expanded** (so that we can compare our results – Later we will add 'cost,' but the number of nodes that were expanded helps us identify which search is most efficient).

Begin by adding (and initializing) a temporary variable for a counter (in the let statement area). Then look at the depth-first code and find a place where you can add 1 to the counter every time the program gets ready to expand a node.

Finally, you will need to return this value, along with the path. (Mmmm... how do we return two values?) Hint: you can *cons* the counter to whatever you're returning – that should do it. It will appear as a number at the beginning of the list of paths.

**F. Modify Depth so that it runs a Breadth First Search.** (This takes a change to 1 statement)

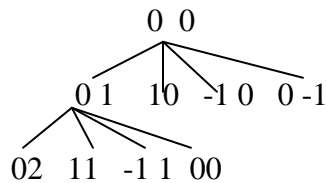
- Now, compare the results to depth?
- Does it return the same path?
- Did it expand the same number?

**G. A new search problem** – Here is a grid for a search program that is looking for the gold. A **G** in the square indicates the gold. You start at the origin  $\langle 0,0 \rangle$ . You go until you find the gold. Just like above, your program should return the solution path. S

		<b>G</b>

We need to alter the search program and adapt it to a new data structure. If you start with  $(0\ 0)$  and your goal is to get to  $(2\ 2)$ , you can still use your program to expand the nodes. Instead of property lists, however, we might want to use lists...and generate the nodes that need to be expanded.

Think of the board like a tree:



To generate the lists, you will need to write a function that generates the children. The children will be  $+1$  and  $-1$  on each side. So, given a list, you add 1 or subtract 1 from each side. (*generate-list* '(0 0) ) should return ((0 1)(1 0) (-1 0) (0 -1))

To 'collect' these lists you will probably need to create a list of a list.

This function obviously returns lists that are out-of-bounds. So you will need a second function that takes that list and returns only legal lists. The function should check for negatives and out-of-bounds. This is like a filter function, only your filter will check for negatives and greater than 2 items.

These two functions can be used in the depth-first search. You can try and work on this problem over the weekend – IF YOU DO, YOU WANT TO MAKE SURE THAT YOU ALSO LOOK FOR DUPLICATE NODES. Otherwise, you can get into an infinite loop.

## Turn in at end of lab: lab3

1. Breadth-first search that includes the print statements and your node counter.
2. A function that generates the nodes for the grid
3. A function that filters out negatives and out-of-bounds numbers

The debug problems that I couldn't access on Friday:

```
(defun funny (x)
  (do ((f x (cdr x))
      (b (reverse x) (cdr x))
      (new1 nil (append (list (car f)) new1))
      (new2 nil (append (list (car b)) new2)))
      ((null f) (append new1 new2))))
(funny '(a b c d))
      (d c b a a b c d)
```

```
(defun flip (switches)
  (mapcar #'(lambda (switch)
    (if (equal switch 'on)
        (setf switch 'off))
        (if (equal switch 'off)
            (setf switch 'on)
            ;;else
            (setf switch switch))))
    switches))
```

```
(flip '(on of on)) => (OFF ON OFF)
```

```
(flip '(a b c on)) => (A B C OFF)
```