Assignment #3: Searching for Paths  
CSCI E-220 Artificial Intelligence  
Due: Thursday, September 22, 2011

1. Introduction

The purpose of this exercise is twofold: to write some search algorithms, and to demonstrate the relative virtues of the basic search techniques discussed in lecture. After writing your routines you will run experiments using the search techniques, examining their efficiency and efficacy at solving problems in the domain of traveling around Western Europe.

The problem you are given is to find a path between two European cities, using only the roads provided on the map given in this handout. For instance, we may want you to find a path from Paris to Vienna. One such path would be Paris to Genoa to Rome to Munich to Prague to Vienna, with a total length of 629 + 328 + 582 + 174 + 185 = 1898 kilometers.

2. The data and data-access functions

At the end of this handout is a rough map, showing road distances between selected European cities. The data in the map has been coded in lisp-readable format in the file ~lib220/asst3/cities.l. The variable CITYLIST contains a list of cities. The variable CONNECTIVITY is a list of all connected city pairs and the road distance (in kilometers) between them. The variable POSITION contains a list of the cities and their positions on the map (using an arbitrary grid for which the conversion factor to kilometers is 0.1393. You won't need to use the number explicitly, but it shows up in a function we provide.)

The cities.l file also contains three data-access functions. The function NEIGHBORS returns a list of all the neighbors of a given city; CONNECTIONS returns a list of dotted pairs of neighboring cities and their associated road distances (in kilometers); and AIR-DISTANCE returns the air-distance (in kilometers) between two given cities. Read through the file carefully, then load it into clisp and run each of the functions on some input to make sure you understand how it works. Remember that a dotted pair is an s-expression whose car and cdr are both atoms.

3. The search functions you will write

You must write a total of four separate search programs: BREADTH-FIRST, DEPTH-FIRST, LEAST-COST, and ASTAR. However, if you plan your code well, you may find that the four are nearly identical in most respects. Expert LISP coders may be able to write a single basic search function with four top-level function calls to it.

For your A* algorithm, use the air distances between two cities as the heuristic estimate for the remaining cost.
4. Details

Your program must begin by loading the file ~lib220/asst3/cities.l. Subsequently, your four top-level functions must be callable as follows:

(BREADTH-FIRST city1 city2)  (LEAST-COST city1 city2)
(DEPTH-FIRST city1 city2)     (ASTAR city1 city2)

Each function must return a path from city1 to city2 as a list of city names, and must also include two cost values somewhere in the returned data structure (e.g. as the first or last elements of the list): the total number of nodes expanded during this search, and the total kilometer distance along the traveled roads from start to finish.

Example:  (BREADTH-FIRST 'PARIS 'BERN) might return (5 722 PARIS BRUSSELS BERN). Alternatively it might return (7 931 PARIS GENOA BERN). When there is more than one possible solution your algorithm may return any of them; that is, I don't care in which order you expand nodes when they are at the same depth.

For ASTAR (and, therefore, for LEAST-COST), you may find it easiest for your recursive algorithm to include your heuristic estimate as part of the kilometer-distance cost numbers that you return; this is fine as long as the heuristic estimate from the final goal is zero (as it must be if your algorithm is to be admissible) because the final estimate for the total distance -- which is the only value humans will see -- will be correct, even though some of the intermediate estimates may not be.

The BREADTH-FIRST and DEPTH-FIRST algorithms are relatively straightforward to write, but do require some thought as to how to return an appropriate value. (That is, it's easy to traverse a tree and get to the destination, but making sure that you've kept track of what to return is not quite so easy). Please use recursion as much as possible, and think about how you solved Problem 5 of Assignment 2.

For the LEAST-COST algorithm you will need to use CONNECTIONS to get the distances between cities. For ASTAR you will also need to use AIR-DISTANCE as your heuristic estimate of remaining distance. Since LEAST-COST is the same as ASTAR except with a zero-valued heuristic guess, you can write essentially the same program for both if you begin with ASTAR and then make one tiny modification for LEAST-COST.

**Special Runtime Note:** If you find yourself running into stack overflow problems, or if your execution speed is very slow, try invoking clisp with the -C flag:

```
% clisp -C
```

This has the effect of compiling code you load using (load 'filename.l) and not only speeds up execution but also modifies the way clisp allocates the stack memory, so that you should not run out of stack space. On the other hand, you can no longer debug your code using the LISP debugger, so don't do this until you're done programming.
5. Your assignment (what you must hand in)

(a) Write the code for the four algorithms, as described above.

(b) Run your search programs on the test case Paris to Vienna using all four search methods, and compare their results. Make a table of the results. How does each algorithm do in terms of the two measures we've asked you to return?

(c) Try to find city pairs for which breadth-first search expands fewer nodes than depth-first, and vice versa. Also compare the efficiency of least-cost versus A* search on various examples. Comment on the results of your tests. What is it about each of the city pairs that makes an algorithm better or worse than another at solving that particular example? Can you find a city-pair for which depth-first search needs to expand many thousands of nodes? What about breadth-first search—even can you find a case that takes the largest possible number of expansions? What about one in which depth-first search works very quickly, but returns an extremely roundabout solution in terms of total kilometer distance?

Include the relevant parts of a runtime session showing the results of running the four search algorithms on the city pairs you selected, and answering all the questions in this section.

(d) Hand in the lisp code electronically using the Unix submit function. Hand in a printed version of the lisp code, of your runtime session results, and of your written commentary.

6. Optional extension activity

(1) - 5%. Experiment with alternative heuristics for the A* algorithm, including heuristics that are not guaranteed to be underestimates. Can you achieve greater search efficiency (fewer nodes expanded) by using 120% of the air distance as your heuristic? Do you then often miss the best answer? What if your heuristic looks just at the x-coordinate or y-coordinate? Are there any other heuristics you could try? Compare their efficiencies against each other. Remember that any underestimating heuristic will make the A* algorithm admissible in the formal sense, so that you should expect to get the same final result -- the only difference should be in the number of nodes examined. With over-estimating heuristics you no longer have an admissible algorithm; do you continue to get the same results anyhow? Why?

(2) – 10%. Write a search program that uses bidirectional search (starting from both ends and proceeding towards a midpoint where a city is on both trees). Include the routines needed to determine the efficiency of this algorithm as compared to the other algorithms, and comment. You will need to use ideas from the text that we did not go over in detail in class.
7. Programming tips

When you are writing LISP code, KEEP EVERY FUNCTION SHORT! You will need to write a lot of helper functions, but your code will be easier to read – and easier to write, for that matter, assuming you are a top-down programmer. As you write your top-level functions, anytime you run up against a task that seems difficult, simply encapsulate it in a function call, with appropriate arguments, that you will write later.

When you need to create a data structure to store more information than is being provided by the user, your top-level function should call a trivial auxiliary function with appropriately-structured input. This is a LISP technique not available in every programming language and takes some getting used to. See my example, below.

When you are trying to keep track of a variable that doesn’t seem to fit into the recursive framework (for example, in this case, the total number of nodes you have expanded, no matter where in the tree you happened to find them), you should think about using a global variable. **That is one of the few times a global variable is appropriate.** Another appropriate use of globals is when a variable is really a constant, and it would be a pain to pass it around from function to function ad infinitum. You must DECLARE all globals, as in the example below.

For this program, the main routine (whether written recursively or iteratively) must keep track of a LIST of LISTS, representing the various possible partial paths we are considering.

Here is some code (which you may borrow freely, or ignore completely if you have a better idea) to demonstrate how I would begin the breadth-first search task, in light of the advice above.

```lisp
(defun breadth-first (start goal)
  (declare (special goal))
  (let ((result nil) (count 0))
    (declare (special count))
    (setq result (breadth-aux (list (list 0 start))))
    (cons count result)))
```

In the above function, I declare “goal” to be special, so that I don’t need to pass it explicitly through the recursive function – it is never going to change, so it doesn’t hurt to treat it like a constant. I define a local variable “count” with initial value 0, and also declare it to be “special”, so that it will be available globally to any function called during the execution of this program.

Finally, I delegate the bulk of the work to a helper function “breadth-aux”, to which I pass the start state of my search: a list consisting of the single list (0 START), which is a valid state showing 0 kilometers traveled, and a partial path beginning at the start node and going nowhere. (We don’t include COUNT here because we’ll stick it on at the very end – it’s not a value that gets calculated recursively.)

The last two lines, which store the result of breadth-aux and then stick the total count to the front of the list, could have been combined into one line if not for the fact that breadth-aux is a function with side effects – specifically, it modifies the value of the count variable. Thus, we
must be sure that breadth-aux has completed its work before we create a list including the count value. This example shows one of the few times that using setq is appropriate.

```
(defun breadth-aux (state-list)
    (if (is-goal-state (car state-list))
        (car state-list)
        (breadth-aux (append (cdr state-list)
                               (expand (car state-list))))))
```

Breadth-aux is the main recursive routine, and it’s also really simple. You’ll notice that I’ve farmed some of the work out to two more functions that I haven’t yet written (you’ll do that) called `is-goal-state` and `expand`. One thing I have NOT shown you is how to modify the values of count or of the kilometers traveled so far. You’ll do that as part of the `EXPAND` function.

**It is CRUCIAL that you understand how the above functions work and why I wrote them the way I did. If you don’t, stop and ask me or the T.A. for help before proceeding!**

If you’ve fully understood breadth-aux, and you remember the difference in the formal algorithms for breadth and depth first search, you’ll see that programming DEPTH-FIRST is trivially different.

(Side note: Those of you who have studied the theory of algorithms and are familiar with the efficiency problems of tail-recursion may feel uncomfortable writing the program in the manner I suggested, and might prefer to write this function iteratively, to save time and stack space. Either method is acceptable for this assignment.)

Programming LEAST-COST and ASTAR* is not much more complicated, but you will want to use the built-in `SORT` function. Here's an example of a `SORT` call in the context of this assignment. The call below assumes that `FOO` is a list of lists such as `'((3 a b) (5 a c d) (1 a e))`, and that the numeric variable on which you wish to sort these lists is in the first position of each sublist as in the example of this paragraph. The `LAMBDA` call is a method of defining a temporary function in-line that you will never use again (see Graham pages 25-27 for a description of `LAMBDA` and the ‘#’ notation).

```
(sort foo #'(lambda (x y) (< (car x) (car y))))
```

Note that `SORT` is a destructive function, meaning that it actually changes the value of the atom `FOO` when it is called. So, you don’t need to write `(setq foo (sort foo ...))`. You could make the sort easier to read by defining a new function “car-less-than” instead of using `LAMBDA`.

To load the `cities.l` file, call `(load "cities.l")` at the beginning of your own solution, assuming that you have copied `cities.l` to your own directory. You can also load it directly from the course library account: `(load "~lib220/asst3/cities.l")`

Model your own programming and commenting style after the style of `cities.l`.
 Utility Functions

Call: (NEIGHBORS city)
Argument: The name of a city
Value: Returns a list of the neighboring cities

(defun neighbors (city)
  (mapcar 'car (connections city)))

Call: (CONNECTIONS city)
Argument: The name of a city
Value: Returns a list of dotted pairs, each of which is the name of a
  neighboring city and its associated road distance from the input city.
Note: CONNECT-AUX is run once (see below) to establish property lists
  with the connection information. CONNECTIONS is then very efficient.
Functions CITY1, CITY2 and DIST provide a readable interface with
  the raw data. See the explanation of the data format, below.

(defun connections (city)
  (get city 'connections))

(defun connect-aux (city clist)
  (cond
    ((null clist) nil)
    ((equal city (city1 (car clist)))
     (cons (cons (city2 (car clist)) (dist (car clist)))
          (connect-aux city (cdr clist))))
    ((equal city (city2 (car clist)))
     (cons (cons (city1 (car clist)) (dist (car clist)))
          (connect-aux city (cdr clist))))
    (t (connect-aux city (cdr clist))))

(defun city1 (cinfo) (car cinfo))
(defun city2 (cinfo) (cadr cinfo))
(defun dist (cinfo) (caddr cinfo))

Call: (AIR-DISTANCE city1 city2)
Arguments: The names of two cities
Returns: an approximation of the air distance between city1 and
  city2, based on their map coordinates as provided in the raw data.
Notes: The factor of 0.1393 is the approximate conversion factor
  between thousandths-of-a-map-inch and kilometers, so that the raw
  air-distance data can be equated with the raw road-kilometrage data.
The functions XCOORD and YCOORD are provided for readability. The
  functions SQUARE is provided both for efficiency and readability.
The (+ 0 ...) around the function is to prevent ROUND from returning
  more than one value (a weirdness of CommonLISP).
(defun air-distance (city1 city2)
  (+ 0
      (round
       (* 0.1393
          (sqrt (+ (square (- (xcoord city1) (xcoord city2)))
                  (square (- (ycoord city1) (ycoord city2))))))))))

(defun square (x) (* x x))

(defun xcoord (city) (car (cadr (assoc city POSITION))))
(defun ycoord (city) (cdr (cadr (assoc city POSITION))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                Data                             ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; CITYLIST is a global variable containing the names of all the cities.

(setq CITYLIST
  '(Copenhagen Hamburg Berlin Warsaw Amsterdam Brussels Prague
    Paris Bern Munich Vienna Budapest Belgrade Trieste Genoa
    Rome Madrid Naples Lisbon))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; CONNECTIVITY is a list of triples. Each triple is a list in the form
;; (city1 city2 dist) where "city1" and "city2" are connected by a direct
;; road of length "dist" kilometers. This data is taken from the map.

(setq CONNECTIVITY
  '((Copenhagen Hamburg 180)
    (Hamburg Amsterdam 338)
    (Hamburg Berlin 182)
    (Berlin Bern 628)
    (Berlin Prague 219)
    (Berlin Warsaw 345)
    (Warsaw Prague 479)
    (Warsaw Vienna 464)
    (Warsaw Budapest 394)
    (Amsterdam Munich 526)
    (Amsterdam Bern 558)
    (Amsterdam Brussels 164)
    (Brussels Bern 497)
    (Brussels Genoa 740)
    (Brussels Paris 225)
    (Prague Vienna 185)
    (Prague Munich 174)
    (Paris Genoa 629)
    (Paris Madrid 805)
    (Bern Munich 311)
    (Bern Trieste 489)
    (Bern Genoa 304)
    (Bern Madrid 1104)
    (Munich Vienna 280)
    (Munich Rome 582)
    (Vienna Budapest 155)
    (Vienna Trieste 317)
(Vienna Belgrade 501)
(Budapest Trieste 384)
(Budapest Belgrade 263)
(Belgrade Trieste 403)
(Trieste Genoa 361)
(Trieste Rome 442)
(Genoa Madrid 951)
(Genoa Rome 328)
(Rome Naples 134)
(Madrid Lisbon 339))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; When this file is loaded, property lists are constructed automatically
;; so that future lookups of the information can be performed efficiently.
;; The function PUTPROP is defined for readability.

(defun putprop (atom property value)
  (setf (get atom property) value))
(dolist (city CITYLIST)
  (putprop city 'connections (connect-aux city CONNECTIVITY)))
(princ "Connection information has been loaded.")
(terpri)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; POSITION is a list of cities and their map positions. Each element
;; of POSITION is a two-element list. The first element is the name of
;; a city and the second element is a dotted pair, in the form
;; (xcoord . ycoord), of the city. The units are roughly thousandths-of-
;; an-inch, based on actual position on the printed 8"-by-11" map.

(setq POSITION
  '((Copenhagen (4937 . 9500))
    (Hamburg (5562 . 8437))
    (Berlin (4500 . 8125))
    (Warsaw (2437 . 8250))
    (Amsterdam (6875 . 7875))
    (Brussels (7062 . 7125))
    (Prague (4125 . 7000))
    (Paris (7625 . 6250))
    (Bern (6125 . 5875))
    (Munich (4875 . 6000))
    (Vienna (3437 . 6250))
    (Budapest (2625 . 6062))
    (Belgrade (1937 . 4937))
    (Trieste (3937 . 5187))
    (Genoa (5625 . 4375))
    (Rome (4312 . 3250))
    (Madrid (9875 . 2750))
    (Naples (3625 . 2875))
    (Lisbon (11625 . 2125)))))
Figure 3.7
Approximate European road distances