CSCE476/876 **Spring 2001**

Homework 2

Assigned on: Monday, February 5^{th}, 2001
Due on: Wednesday February 21^{th}, 2001

Pen and paper part should be returned in class
Programming assignment should be submitted with handin
Check Syllabus for respective turn-in modalities, hours, etc.

1 AIMA, Exercise 2.4, Page 50. (5 points)

2 AIMA, Exercise 3.2, Page 87. (5 points)

3 AIMA, Exercise 3.9, Page 89. (10 points)

4 Basic search strategies (Total 12 points)

Assume you have the following search space:

State	Next-state	Cost
A	B	4
A	C	1
B	D	3
B	E	8
C	C	0
C	D	2
C	F	6
D	C	2
D	E	4
E	G	2
F	G	8

1. Draw the state space of this problem as a directed graph in which the states are represented as nodes and actions/operators as directed edges labeled by their cost. (2 points)
2. Assume that the initial state is A and the goal state is G. Use each of the following search strategies to find a path from the initial state to the goal state.
   - Breadth-first search (2.5 points)
   - Depth-first search (2.5 points)

- Uniform cost search (2.5 points)
- Iterative deepening search (2.5 points)

At each step of a search algorithm, report:

- which node is being expanded, and
- the content of the fringe.

Also, report the solution path found.

5 Conditional instruction (Total 8 points)

Courtesy of B. Faltings, EPFL

`cond` can be thought as a generalization of a sequence of the IF.. THEN.. ELSE statements: `cond` allows us to express a set of alternatives

1. (5 points) Write a function `traffic-light` to represent the behavior of a driver. The function takes two arguments: the first one is the state of the traffic light (red, yellow, green) and the second parameter is a Boolean stating whether there is a camera or not at the traffic light (common in Europe!). This function must return the atom:

   - `continue`, (1) if the light is green, or (2) if the light is yellow and there is no camera.
   - `brake`, (1) if the light is red, or (2) if the light is yellow and there is a camera.

   Warning: traffic-light must not use `or`.

2. (3 points) Write a second function `traffic-light-or` using `or`.

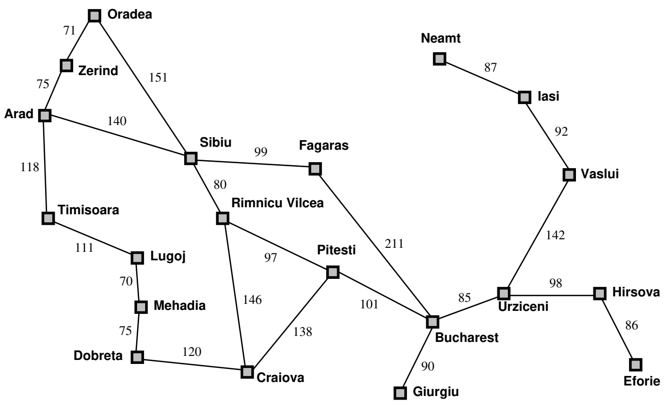
6 Exify (10 points)

Write a **<u>recursive</u>** function `exify` that takes a list as input and returns a list in which all non-nil elements are replaced by the atom `X`.
Test it first on: `(exify '(1 hello 3 foo 0 nil bar))`.
It should return: `(X X X X X NIL X)`.
Then test it on: `(exify '(1 (hello (3 nil (foo)) 0 (nil)) (((bar))))`.
It should return: `(X (X (X NIL (X)) X (NIL)) (((X))))`.

7 Data structures in Common Lisp (Total 50 points)

Using `defstruct` (see LWH, Chapter 13), create data structures in Common Lisp to represent the map of Romania. Include the information about the distances between two cities linked by a road as well as the distance from any given city to Bucharest as indicated in Figure 1.

Indications (follow illustration in Figure 2):



Straight−line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Figure 1: *Map of Romania with road distances in kilometers and straight-line distances to Bucharest. AIMA page 95.*

- Create a data structure for a city using `defstruct`.
- Include an attribute to store the name of the city.
- Include an attribute to store the neighboring cities.
- Include an attribute that provides the value of the straight-line distance to Bucharest.
- Create a global variable that stores all the cities. Implement it this in **two different ways**: a list `*all-cities-list*` and a hash-table `*all-cities-htable*` (check definition of a hash-table in `http://www.cs.cmu.edu/afs/cs.cmu.edu/project/ai-repository/ai/html/cltl/cltl2.html`).
- After creating structures for all the cities, loop through them again in order to include, in the relevant attribute of a city, a reference its neighboring cities. Store these neighbors as an association list of the structure of a neighbor and the distance between the two (see LWH, page 31).

Tasks:

1. (10 points) Design, implement and test your map.
2. (10 points) Write two functions `all-cities-from-list` and `all-cities-from-htable` that take a global variable, `*all-cities-list*` and `*all-cities-htable*`, respectively, and return a list of all cities on the map.
3. (5 point) Write two functions `get-city-from-list` and `get-city-from-htable` that take the name of a city as input and return the corresponding structure (by accessing a global variable, `*all-cities-list*` and `*all-cities-htable*`, respectively).

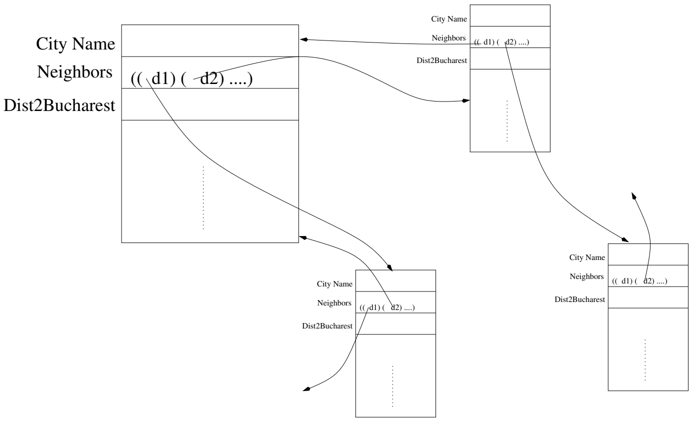


Figure 2: *Data structures.*

4. (5 points) Write two functions `neighbors-using-list` and `neighbors-using-htable` that take the name of a city as input and return a list of its direct neighbors. `neighbors-using-list` and `neighbors-using-htable` should use `get-city-from-list` and `get-city-from-htable`, respectively.

5. (10 points) Using `*all-cities-htable*`, write a function `neighbors-within-d` that takes the name of a city `my-city` and a number `distance`, then returns, for all direct neighbors within `distance` from `my-city` ($\leq$), an association list of all neighbors of `my-city` and their distance to `my-city`.

6. (10 points) Using `*all-cities-htable*`, write a function `neighbors-p` that takes the name of two cities `city-1` and `city-2`, and returns the distance between them if they are directly connected or nil if they are not.