CSCE476/876 **Spring 2006**

Homework 5

Assigned on: Friday, February 24, 2006.

Due: Friday, March 10, 2006.

Programming assignment should be submitted with `handin`.

---

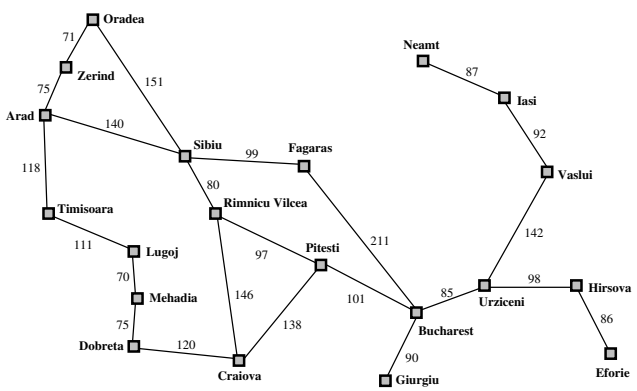
1. Data structures in Common Lisp (Total 50 points)
2. Implementing Search in Common Lisp (Total 45 points)
3. Documentation and organization of code. (Total 5 points)
   In particular, you must specify your sources and acknowledge any help you may have received.

Do not hesitate to seek help during recitation and office hours.

---

1 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.



Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Dobreta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

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

Indications (follow illustration in Figure 2):

- Create a data structure for a city using `defstruct`.
- Include an attribute `name` to store the name of the city.

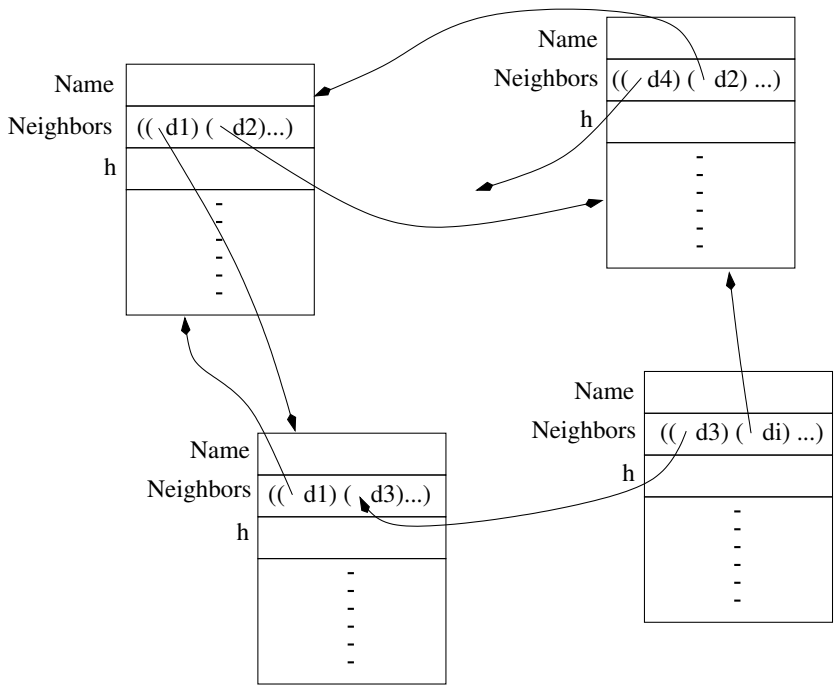


Figure 2: Data structures.

- Include an attribute `neighbors` to store the neighboring cities.
- Include an attribute `h` that provides the value of the straight-line distance to Bucharest.
- Create a global variable that stores all the cities. Use `defvar` to declare the global variables. Implement it this in **two different ways**: a list `*all-cities-list*` and a hash-table `*all-cities-htable*` [1]. Use the name of a city as key and the structure as value. For sake of clarity, you are *not* asked to implement a hash-table (which you probably did in CSCE310) but to use a hash-table in Lisp.
- 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).

1.1 Tasks

1. (10 points) Design, implement and test your map.
2. (5 points) Write a function `all-cities-from-list` that takes a global variable, `*all-cities-list*`, and returns a list of all *names* of cities on the map.

[1] Check documentation on hash-tables in `http://www.cs.cmu.edu/afs/cs.cmu.edu/project/ai-repository/ai/html/cltl/cltl2.html` and `http://www.lisp.org/HyperSpec/Body/fun_make-hash-table.html`.

3. (5 points) Write a and `all-cities-from-htable` that takes a global variable, `*all-cities-htable*` and returns a list of all the *structures* of cities on the map.

4. (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).

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

6. (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 the structures of the neighbors of `my-city` and their distance to `my-city`.

7. (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.

Note that the global variables should always be passed as arguments to these functions (becoz it is cleaner).

2 Implementing Search in Common Lisp (Total 45 points)

You are asked to implement as a Tree-Search the search strategies below. Do not implement a Graph-Search, it will not be accepted.

- Any uninformed search strategy of your choice, 15 points
- A Greedy search strategy, and 15 points
- An A* search strategy. 15 points

for the 'Romanian Holiday' problem. Needless to say, you should first get Section 1 to work. Write `Search` that take as input the name of <u>any city</u> on the map, the name of a search strategy, and returns:

1. The path to Bucharest,
2. The number of nodes generated/visited by the search process,
3. The cost of the path found (even when the function $g(n)$ is not used to choose the node to expand),
4. The running time spent by Lisp on the search. You may use the function `get-internal-run-time` before and after running the search and print the difference or use the function `time` and report the value of `cpu time (non-gc)` as printed on the *standard-output* (i.e., the emacs buffer).

Hints:

- You may want to use the Lisp function `values` and `multiple-value-bind`.
- You may choose to write one search function and give it the strategy as an argument.

2.1 Results to report

In addition to your code, report the results of your two functions applied to *each* city in Romania as indicated in the following table:

Uninformed search of your choice				
City name	**#nodes visited**	**Path to Bucharest**	**Total cost of path**	**CPU time**
Arad Bucharest ⋮ Vaslui Zerind				
Greedy Search				
City name	**#nodes visited**	**Path to Bucharest**	**Total cost of path**	**CPU time**
Arad Bucharest ⋮ Vaslui Zerind				
A* Search				
City name	**#nodes visited**	**Path to Bucharest**	**Total cost of path**	**CPU time**
Arad Bucharest ⋮ Vaslui Zerind				

2.2 Some indications

Follow the requirements below:

1. Modify the data structure of a city that you implemented in Section 1 to add one more field `visited`, initialized to `Nil`. Use this attribute for loop control during search: when a city is visited, set this field to `T`.

2. Create a new data structure (e.g., `defstruct`) to represent a node in the search tree. The structure should have attributes that *point* to the structures of its parent (when applicable), its children (list), the city it represents. Other attributes may be necessary, such as path value at the node.

3. Implement a function `expand-node` that takes a node in the search tree and generates its children, which should correspond to cities not *yet* visited. It needs to generate one node data-structure per child.

4. Implement a function `evaluate-node` that takes a node and a search strategy and returns the value of the node (e.g., $g(n), h(n)$ or $f(n)$).

5. Implement a function that takes a fringe (i.e., a list of nodes to be expanded) and returns the node to expand. As a refinement, you can provide the name of the search strategy as an optional second argument (check `:key` in the list of arguments of a function).

6. If you separate the implementation search strategy from the evaluation functions cleverly enough, you may be able to use the same search function for all search strategies you implement.

7. Implement the search strategies iteratively, not recursively.

8. Declare a global variable `*nnv*` for storing the *number of nodes visited*. The search function should set up its value and the function `expand-node` should increment this value at every expansion (technically, every instantiation of a search-node structure).

9. Load the information about the cities from the file `cities.in` which is on the course website.