CSCE476/876 **Spring 2001**

Homework 3

Assigned on: Friday, February 23^{st}, 2001

Question	Due date	Grading
Question 1	Pen and paper part should be turned in class, on Wed Feb 28^{th}, 2001	8 points
Question 2	A *working and fully documented* function should be turned in with `handin` on Fri Mar 9^{th}, 2001.	12 points
Question 3	A first *sketch* with data structures and main functions should be turned in with `handin` on Fri Mar 9^{th}, 2001. Search need *not* be in working conditions.	30 points
Question 3	Result table and bug-free code submitted with handin on Mon Mar. 19^{th}, 2001.	50 points
A 15-point *bonus* will be awarded when Question 3 is turned in a working condition and fully documented by Fri Mar 9^{th}, 2001.		
In addition to office hours, help sessions will be scheduled upon request. Check Syllabus for respective turn-in modalities, hours, etc.		

1 Evaluation function

Adapted from AIMA, Exercise 4.1.

With $g(n)$ being the path length,

1. Suppose that we run a greedy search algorithm with $h(n) = -g(n)$. What sort of search will the greedy search emulate?
   Explain. *(4 points)*

2. Suppose that we run a search algorithm with $h(n) = g(n)$. What sort of search will the greedy search emulate?
   Explain. *(4 points)*

2 A special set-difference

Amy Beckwith, a graduate student at CSE, needed to implement the following function for her experiments.

The function should remove some specified elements of a set from a partition of the set. It takes two lists, a list of atoms `lost` and a partition of the set `partition` (with `lost` $\subseteq \bigcup_{S \in \texttt{partition}} S$). It returns a new list obtained by removing, in the elements of `partition`, the elements of `lost`.

Example: Given,

`(setf lost '(a b c d)`, and

`(setf partition '((a b f) (c) (h d j k))`,

`(amys-set-difference partition lost)` should return `((f) (h j k))`.

You are asked to implement the function `amys-set-difference` in two manners:

1. iteratively, and *(6 points)*
2. recursively. *(6 points)*

Hint: check the definition of `remove-if`, which may be useful at some point.

3 Implementing Search in Common Lisp

You are asked to implement one (1) uninformed and one (1) informed search strategy of your choice for the 'Romanian Holiday' problem. Needless to say, you should first get your Homework 2 working. Write two functions `Uniformed-Search` and `Informed-search` that take as input the name of tow cities on the map and return:

1. The number of nodes generated/visited by the search process,
2. The path between the two cities, and
3. The cost of the path found (even when the function $g(n)$ is not used to choose the node to expand).

Hint: use the Lisp function `values`. You may choose to write one search function and give it the strategy as an argument.

3.1 Results to report

In addition to your code, report the results of your two functions applied to *each* city in Romania (as first argument) and Bucharest (as second argument) as indicated in the following table:

Search Strategy I: *specify*			
City name	**#nodes visited**	**Path to Bucharest**	**Total cost of path**
Arad Bucharest ⋮ Vaslui Zerind			

Search Strategy II: *specify*			
City name	**#nodes visited**	**Path to Bucharest**	**Total cost of path**
Arad Bucharest ⋮ Vaslui Zerind			

3.2 Some indications

You may find the following useful:

1. Modify the data structure of a city that you implemented in Homework2 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 to use the same function for both search strategies you adopt.

7. Implement the search strategies either recursively or iteratively, as you prefer.

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).