CSCE476/876 **Spring 2002**

Homework 5

Assigned on: Monday March 11, 2002.

Due: Wednesday March 27 and Monday April 1st, 2002. According to schedule listed below.

Question	Due date	Grading
Question 1, 2 and 3	Pen and paper part should be turned in class, on Wednesday March 27^{th}, 2002	10 points, each 30 points, total
Question 4	A first *sketch* with data structures and main functions should be turned in with `handin` on Wednesday March 27^{th}, 2002. Search need *not* be in working conditions.	30 points
Question 4	Result table and bug-free code submitted with handin on Mon April 1^{st}, 2002.	40 points
In addition to office hours, help sessions will be scheduled upon request. Check Syllabus for respective turn-in modalities, hours, etc.		

Pen and paper part should be returned in class
Programming assignment should be submitted with `handin`

---

1 Scheduling problem as a CSP

Courtesy of Rina Dechter, UC-Irvine.

Consider the problem of scheduling five tasks: T1, T2, T3, T4 and T5, each of which takes one hour to complete. The tasks may start at 1:00, 2:00 or 3:00. Any number of tasks can be executed simultaneously provided the following restrictions are satisfied. T1 must start after T3. T3 must start before T4 and after T5. T2 cannot execute at the same time as T1 or T4. And T4 cannot start at 2:00.

1. Formulate the problem as a CSP by stating: the variables, their values and the constraints (hint: focus on the start time of a task.)
2. Draw the constraint graph.
3. Apply arc-consistency to each constraint in the CSP until no values can be ruled out (the CSP becomes arc-consistent).

2 Consistency checking

Consider the 4-Queens problem where each queen is associated with a row and can be assigned to any column in the row.

1. Express the domains of the variables.

2. Consider the binary constraints between every two queens and make explicit the constraint definition.

3. Draw the constraint graph.

4. Arc-consistency of a binary constraint C_{Q_i,Q_j} between variables Q_i and Q_j ensures that every value for the variable Q_i has a support (at least one consistent value) in the domain of the variable Q_j, and vice versa. Run manually arc-consistency on the 4-Queens problem. Can you remove any value? At the end of the operation, the CSP is said to be arc-consistent.

5. Arc-consistency is also called 2-consistency because it considers all combinations of two variables at the same time. Let's consider all combinations of 3 variables at the same time and let's check whether or not every value in the domain of a given variable has a support in the domain of the two other variables. If it does not, the value can be removed. Can you remove any value? (Alternatively stated, a CSP is said to be 3-consistent, if for every combination of consistent values for two variables, one can find a value in the domain of any third variable such that the constraints between the three variables are satisfied).

6. What can you conclude about the effectiveness of consistency algorithms for the 4-Queens problem?

3 Latin square

Adapted from of Daphne Koller, Stanford University.

A Latin Square is a $N \times N$ array filled with colors, in which no color appears more than once in any row or column. Finding a solution to a 4×4 Latin Square can be formulated as a CSP, with a variable for each cell in the array, each having a domain of $\{r, g, b, y\}$, and a set of constraints asserting that any pair of cells appearing in the same row must have different colors, and that any pair of cells appearing in the same column must have different colors.

At this point in the search, seven of the cells have been instantiated (displayed in **boldface** in Figure 1), and the initial domains of the remaining cells are shown. The next cell to instantiate has the domain values in *italics*.

Re-execute (from scratch) the 7 first assignments in the specified order and, at *<u>each</u>* assignment, draw the Latin Square while filtering the domains of the relative future variables. Do this process for the two following look-ahead strategies:

1 **r**	2 **g**	3 **b**	4 **y**
5 **g**	6 **y**	*r g b y*	r g b y
7 **b**	r g b y	r g b y	r g b y
r g b y	r g b y	r g b y	r g b y

Figure 1: *Current state of search.*

- the partial look-ahead strategy, forward checking (FC)
- the full look-ahead strategy, maintaining arc consistency.

Indicate eliminated values by crossing them out or just erasing them.

4 Implementing Search in Common Lisp

You are asked to implement:

- A depth-first search strategy,
- A Greedy search strategy, and
- An A* search strategy

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.

4.1 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:

Depth-First Search			
City name	**#nodes visited**	**Path to Bucharest**	**Total cost of path**
Arad Bucharest ⋮ Vaslui Zerind			

Greedy Search			
City name	**#nodes visited**	**Path to Bucharest**	**Total cost of path**
Arad Bucharest ⋮ Vaslui Zerind			

A^* Search			
City name	**#nodes visited**	**Path to Bucharest**	**Total cost of path**
Arad Bucharest ⋮ Vaslui Zerind			

4.2 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 all search strategies you implement.

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).