Assignment 1

CSU520, Spring 2008
Due: Wednesday, Jan. 23

Suppose that you have two containers for holding water, a 5-gallon jug and an 11-gallon jug, and you have access to a water faucet for filling either jug to the top and a drain for emptying either jug. Neither of the jugs has any markings to indicate how many gallons of water it contains when it is neither entirely full nor entirely empty. Thus, for example, if you want exactly 4 gallons of water, it is not possible to simply place the empty 5-gallon jug under the faucet and fill it to exactly 4 gallons. Instead, you must perform a succession of six possible moves as listed below. For the sake of generality, these moves are described for any two sizes of jug.

The possible moves are:

1. Pour from the first jug into the second jug until either the second jug is full or the first jug is empty, whichever occurs first.
2. Pour from the second jug into the first jug until either the first jug is full or the second jug is empty, whichever occurs first.
3. Fill up the first jug from the faucet.
4. Fill up the second jug from the faucet.
5. Empty the first jug down the drain.
6. Empty the second jug down the drain.

For example, to obtain exactly 6 gallons of water in the 11-gallon jug (starting with both jugs empty) you could first fill the 11-gallon jug from the faucet (move 4), then pour its contents into the 5-gallon jug (move 2), leaving the desired 6 gallons in the 11-gallon jug. Similarly, if you wanted 4 gallons of water in the 5-gallon jug (starting with both jugs empty), you could first fill the 5-gallon jug from the faucet (move 3), then pour its contents into the 11-gallon jug (move 1), then fill the 5-gallon jug again (move 3), then pour its contents into the 11-gallon jug (move 1), then fill the 5-gallon jug again (move 3), then pour its contents into the 11-gallon jug, which would leave 4 gallons in the 5-gallon jug.

To create a Lisp program to search for solutions to such puzzles, let the list of two numbers (x y) represent x gallons in the first jug and y gallons in the second jug. In this notation, the sequence of states visited in the first example above is

```
(0 0) -> (0 11) -> (5 6)
```

while the sequence of states visited in the second example is

```
(0 0) -> (5 0) -> (0 5) -> (5 5) -> (0 10) -> (5 10) -> (4 11)
```

For this assignment, you will write application-specific code as well as a specific search program and combine them to solve the following water-jug puzzle instance:

Starting with an empty 5-gallon jug and an empty 11-gallon jug, how can we end up with exactly 3 gallons of water in the 11-gallon jug and with the 5-gallon jug empty?

Specifically, do the following:

1. Write a function in Lisp that computes a list of successor states for any state in this puzzle, using this representation of states. Run this function on some samples to show that it works correctly when one jug holds 5 gallons and the other jug holds 11 gallons. Capture the test output from this function in a dribble file.

Your code must use global parameter names for the jug sizes, so that it can be easily modified to work on puzzles involving, say, 9-gallon and 13-gallon jugs. To do this, your code should begin with the following declarations

```
(defparameter *jug-1* 5)
(defparameter *jug-2* 11)
```

Only these parameter names, not the values 5 and 11, should appear in the remaining code. Comment your code well, and use good Lisp programming style.

2. Download the depth-limited depth-first search program

`www.ccs.neu.edu/home/rjw/csu520/programs/search/depth-limited-dfs-no-loops.lisp`

and use it as the basis for defining a Lisp function that performs iterative-deepening depth-first search. This function should have the signature

```
(find-path-iddfs <start-state> <goal-test> <successors>
                 <equal-states> <max-depth-limit>)
```

The last argument is used only to limit the overall search depth in problems where the search might take too long (or never terminate). You may make it an optional argument with a suitably high default value if you wish.

3. Run the function `find-path-iddfs` with suitable arguments, including the successors function you wrote for Problem 1, to solve the water-jug problem instance specified above. When doing this, evaluate

```
(pprint (find-path-iddf ...))
```

so that the entire list of states returned as the solution will be displayed, and capture the output in a dribble file.

What you should turn in: Hardcopy of all source files and dribble files.

Extra Credit. The function `find-path-dldfs` I have provided for you that performs depth-limited depth-first search only checks for repeated states by avoiding paths with loops, but it does not do more global repeated-state checking. This means that some nodes may get expanded multiple times in the search as long as they are not being considered on the same path. Consider this alternative strategy: Suppose every expanded node is placed on the closed list and never expanded again, as in the general search algorithm discussed in class. (In the context of iterative deepening, this closed list would, of course, be cleared every time the depth limit is increased.) Explain why this will not necessarily work properly when performing depth-limited search. Then describe a slightly more complex strategy that can be used to prevent this difficulty while avoiding expanding any nodes that should not be expanded again.