Assignment 1

CSG120, Fall 2003
Due: Thursday, Sept. 25

Part I. General depth-limited and iterative deepening search functions

1. Here you will write 2 variants of a general function that performs depth-limited depth-first (DLDF) search in a graph. Each should accept application-specific arguments as in the code found in

`www.ccs.neu.edu/home/rjw/csg120/programs/search/blind-search.lisp`

Also, they should simply return a list of states on the path found. Implement your functions using recursion on the successors.

a. First, write a version that performs no repeated-state checking at all. Make up a simple graph on which this program will succeed at finding a solution and run your program on it and capture the result in a dribble file. Pass it a suitably high depth limit, as appropriate. Its signature should be

```
(find-path-dldf <start-state> <goal-test> <successors> <depth-limit>)
```

where the actual parameters passed in for the second and third arguments are application-specific functions.

b. Now create a version of your program that avoids creating paths with loops but does not do repeated-state checking at a more global level. This program should consist of a very minor modification to your first program.

This function's signature should be

```
(find-path-dldf-no-loops <start-state> <goal-test> <successors>
                         <equal-states> <depth-limit>)
```

Why does this version need an additional argument?

Run a test of your program that shows that it works properly on the graph used as the example in

`www.ccs.neu.edu/home/rjw/csg120/programs/search/blind-search.lisp`

and capture this in a dribble file.

c. It is possible to create another variant of this program that uses a global closed list to avoid expanding repeated states anywhere in the search tree. However, doing this in a simple-minded way (i.e., by simply placing any expanded node on the closed list and refusing to expand any node found to already appear on the closed list) will not work. Explain. (Extra credit: Can you think of a fix for this problem? Describe it; you need not implement it.)

2. Now you will create 2 variants of an iterative deepening depth-first (IDDF) program that make use of the two depth-limited depth-first search functions from Problem 1.

a. Use find-path-dldf (from Problem 1a). The signature of this iterative deepening function should be

```
(find-path-iddf <start-state> <goal-test> <successors>)
```

Capture dribble output showing that it works on the graph used in the example in

`www.ccs.neu.edu/home/rjw/csg120/programs/search/blind-search.lisp`

b. Use find-path-dldf-no-loops (from Problem 1b). The signature of this iterative deepening function should be

```
(find-path-iddf-no-loops <start-state> <goal-test> <successors>
                         <equal-states>)
```

Capture dribble output showing that it works on the graph used in the same example used above.

Part II. Application to water-jug puzzles

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 part of the assignment you will write application-specific code and run it together with your search programs from Part I to solve the following water-jug puzzle:

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:

3. Write a function in Lisp that computes a list of successors 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.

4. Run each of your iterative-deepening depth-first programs from Part I using appropriate arguments to find solutions to this water-jug puzzle. Capture all output from these runs in a dribble file.

Comment on the relative speed of the 2 versions of your IDDF program on this problem.

What you should turn in: Hardcopy of all source files and dribble files, and answers to the questions (1b, 1c, and 4).