CS 540: Introduction to Artificial Intelligence
Homework Assignment # 3

Assigned: 2/19
Due: 2/26 before class

Hand in your homework:

If a homework has programming questions, please hand in the Java program. If a homework has written questions, please hand in a PDF file. Regardless, please zip all your files into hwX.zip where X is the homework number. Go to UW Canvas, choose your CS540 course, choose Assignment, click on Homework X: this is where you submit your zip file.

Late Policy:

All assignments are due at the beginning of class on the due date. One (1) day late, defined as a 24-hour period from the deadline (weekday or weekend), will result in 10% of the total points for the assignment deducted. So, for example, if a 100-point assignment is due on a Wednesday 9:30 a.m., and it is handed in between Wednesday 9:30 a.m. and Thursday 9:30 a.m., 10 points will be deducted. Two (2) days late, 25% off; three (3) days late, 50% off. No homework can be turned in more than three (3) days late. Written questions and program submission have the same deadline.
Assignment grading questions must be raised with the instructor within one week after the assignment is returned.

Collaboration Policy:

You are to complete this assignment individually. However, you are encouraged to discuss the general algorithms and ideas with classmates, TAs, and instructor in order to help you answer the questions. You are also welcome to give each other examples that are not on the assignment in order to demonstrate how to solve problems. But we require you to:

- not explicitly tell each other the answers
- not to copy answers or code fragments from anyone or anywhere
- not to allow your answers to be copied
- not to get any code on the Web

In those cases where you work with one or more other people on the general discussion of the assignment and surrounding topics, we suggest that you specifically record on the assignment the names of the people you were in discussion with.

Question 1: A Water Jug Problem [60 points]

This is a programming question. The solution to the programming problem should be coded in Java, and you are required to use only built-in libraries to complete this homework. Please submit a single zip file named hw3.zip, which should contain a source code file named WaterJug.java with no package statements, and make sure your program is runnable from command line on a department Linux machine. We provide a skeleton WaterJug.java code that you can optionally use, or you can write your own.

The goal of this assignment is to become familiar with uninformed search – breadth-first search (BFS), depth-first search (DFS) and iterative deepening search (IDDFS). The assignment tests your understanding of AI concepts, and your ability to turn conceptual understanding into a computer program. All concepts needed for this homework have been discussed in class, but there may not be existing pseudo-code for you to directly follow. We ask you to implement your own stack for BFS/DFS/IDDFS as we did in class, rather than writing a recursive program.

In this question, you will implement uninformed search for a water jug problem.

To illustrate, suppose you are given two containers: a 2-gallon water jug and a 1-gallon jug. Neither of them have any measuring marks on them at all. Initially both jugs are empty. You have 3 types of action available to you: 1) fill a jug up completely full (i.e. f1, f2) 2) empty a jug completely (i.e. e1, e2) 3) pour as much water as possible from one jug into the other (i.e. p12 (pour water from jug1 into jug2), p21 (pour water from jug2 into jug1)). The goal is you must end up with a jug having exactly 1 gallons of water. Formally:

- State space $S = \{0, 1, 2\} \times \{0, 1\}$
- Initial state $s_0 = (0, 0)$
- Goal states $G = \{(0, 1), (1, 0), (1, 1), (2, 1)\}$
- Actions $A = \{f1, f2, e1, e2, p12, p21\}$
- Successor function is given by the table below

	f1	**f2**	**e1**	**e2**	**p12**	**p21**
(0,0)	(2,0)	(0,1)	-	-	-	-
(1,0)	(2,0)	(1,1)	(0,0)	-	(0,1)	-
(2,0)	-	(2,1)	(0,0)	-	(1,1)	-
(0,1)	(2,1)	-	-	(0,0)	-	(1,0)
(1,1)	(2,1)	-	(0,1)	(1,0)	-	(2,0)
(2,1)	-	-	(0,1)	(2,0)	-	-

- $Cost = 1$ for all arcs
- The search problem: find a solution path from a state in s_0 to a state in G

There are an infinite number of solutions. If a search algorithm is optimal, a solution to the water jug is a path from the initial state to a goal state with the smallest path cost. Possible solutions of the example above:

- ((0,0), (0,1))

- ((0,0), (2,0), (1,1))
- ((0,0), (2,0), (0,0), (2,0), (1,1))

Write a program WaterJug.java with the following command-line format:

```
$java WaterJug FLAG cap_jug1 cap_jug2 curr_jug1 curr_jug2 goal
```

where **FLAG** is an integer that specifies the output of the program (see below). **cap_jug1** and **cap_jug2** specify each jug's capacity. **curr_jug1** and **curr_jug2** specify how much water each jug is initially filled and cannot exceed its capacity. **goal** specifies how much water you must end up with in either jug1 or jug2, or both jugs. These take values in integer 0-9 for command-line argument. For example, given the earlier example and FLAG=100, the command line would be

```
$java WaterJug 100 2 1 0 0 1
```

(Part a, 5 points) When FLAG=100, print out the successor states of an initial state, in the order they are pushed into the stack (see below). Each successor state should be printed as jug1 and jug2 state digits back-to-back on a single line. For example,

```
$java WaterJug 100 2 1 0 0 1
01
20

$java WaterJug 100 5 7 3 1 2
01
04
30
37
40
51
```

Important: We ask you to implement the following order among successors. If we view a state as a 2-digit integer, then there is a natural order among states. Whenever you push successors into the stack, push them from small 2-digit to large 2-digit. This order will be used throughout this program, so that the output is well-defined.

(Part b, 5 points) When FLAG=200, verify if each of the successor states is a goal node. Recall this is true only when either jug1 or jug2, or both have exactly ***goal*** gallon. Print each successor state following by either true or false, separated by a whitespace.

```
$java WaterJug 200 2 1 0 0 1
01 true
20 false

$java WaterJug 200 5 7 3 1 2
01 false
```

```
04 false
30 false
37 false
40 false
51 false
```

(Part c, 15 points) When FLAG=300, perform a breadth-first search till a goal state is reached. You will need to implement BFS using OPEN and CLOSED lists to keep track of progress through the state space. On the first line, print the initial state. During each iteration step, take out a state from the fringe, verify whether it is a goal node. If so, print it, followed by the word "Goal" and terminate. Else, check whether it is already expanded. If yes, discard it. Otherwise, expand the state, add its successors to the fringe, and mark that state as already-expanded. At the end of each iteration step, print the expanded state, OPEN list, and CLOSED list with the following format:

$s_{expanded}$ $[o_1, o_2, ..., o_m]$ $[c_1, c_2, ..., c_n]$

The last line, you will print the word "Path", followed by a sequence of states from the initial to the goal (all space-separated). For example,

```
$java WaterJug 300 4 3 0 0 2
00
00 [03,40] [00]
03 [40,30,43] [00,03]
40 [30,43,13] [00,03,40]
30 [43,13,33] [00,03,40,30]
43 [13,33] [00,03,40,30,43]
13 [33,10] [00,03,40,30,43,13]
33 [10,42] [00,03,40,30,43,13,33]
10 [42,01] [00,03,40,30,43,13,33,10]
42 Goal
Path 00 03 30 33 42
```

(Part d, 15 points) When FLAG=400, perform a depth-first search till a goal state is reached. Again, you will need to implement DFS using OPEN and CLOSED lists to keep track of progress through the state space. Print results like in part c. For example,

```
$java WaterJug 400 4 3 0 0 2
00
00 [03,40] [00]
40 [03,13,43] [00,40]
43 [03,13] [00,40,43]
13 [03,10] [00,40,43,13]
10 [03,01] [00,40,43,13,10]
01 [03,41] [00,40,43,13,10,01]
41 [03,23] [00,40,43,13,10,01,41]
23 Goal
Path 00 40 13 10 01 41 23
```

(Part e, 20 points) When FLAG=5XX, perform a depth-limited depth-first search with cutoff depth XX (i.e. this is one outer-loop of iterative deepening). For example, if FLAG=500, the cutoff depth is 0. In DFS, you will push the initial state in the stack, pop it out, do a goal test, but will NOT expand it. If FLAG=501, the cutoff depth is one. In DFS, you will expand the initial state (i.e. put its successors into the stack in the order we specified in Part a). You will pop each successor out, perform goal-check (and terminate the program if goal-check succeeds). But you will not expand any of these successors.

XX can be 00 to 99. If depth-limited DFS finds a goal before the cutoff, it should stop.

Print results similar to part d with a current depth parameter prefixed. For example,

```
$java WaterJug 555 4 3 0 0 2
0:00
0:00 [] [00]
1:00
1:00 [03,40] [00]
1:40 [03] [00,40]
1:03 [] [00,40,03]
2:00
2:00 [03,40] [00]
2:40 [03,13,43] [00,40]
2:43 [03,13] [00,40,43]
2:13 [03] [00,40,43,13]
2:03 [30] [00,40,43,13,03]
2:30 [] [00,40,43,13,03,30]
3:00
3:00 [03,40] [00]
3:40 [03,13,43] [00,40]
3:43 [03,13] [00,40,43]
3:13 [03,10] [00,40,43,13]
3:10 [03] [00,40,43,13,10]
3:03 [30] [00,40,43,13,10,03]
3:30 [33] [00,40,43,13,10,03,30]
3:33 [] [00,40,43,13,10,03,30,33]
4:00
4:00 [03,40] [00]
4:40 [03,13,43] [00,40]
4:43 [03,13] [00,40,43]
4:13 [03,10] [00,40,43,13]
4:10 [03,01] [00,40,43,13,10]
4:01 [03] [00,40,43,13,10,01]
4:03 [30] [00,40,43,13,10,01,03]
4:30 [33] [00,40,43,13,10,01,03,30]
4:33 [42] [00,40,43,13,10,01,03,30,33]
4:42 Goal
Path 00 03 30 33 42
```