CS3460: Data Structures

Lab 08: Graphs

Total Points: 20

Problems

Notes on Grading: Unless otherwise stated, all programs will receive input via `System.in` and will output solutions via `System.out`.

To simplify the grading process, all grading will be automated. When applicable, you will be provided with sample input/output files for testing. You can ensure that your program will receive full marks by testing it with these provided files.

```
$ java YourProgram < input.txt > output.txt
$ diff output.txt correct.txt
```

The first line executes the Java program, redirecting input from a file `input.txt` and writing the output to a file `output.txt`. The second line compares your program's output (now stored in `output.txt`) with the correct answer (stored in `correct.txt`). If these files match exactly, the `diff` program will print nothing. Otherwise, it will list the differences.

Submission: Please submit any `.java` files necessary to compile/run your project. Please do not include any other files (such as input/output files).

1. **Water Jugs (10 points):** You managed to get the wolf, the goat, and the cabbage across the river safely, and all that depth-first searching has worked up quite an appetite! It's time for dinner, and you have a head of cabbage to boil.

   You have two water jugs, which have integer sizes $0 < A \leq B \leq 1000$. In order to boil the cabbage for dinner, you need to measure out exactly C units of water. Please write a program named `Jugs.java` that, given A, B, and C, computes whether the task is possible or impossible.

```
Enter A: 3
Enter B: 4
Enter C: 5
Possible!

A: Fill Jug 1 [a = 3, b = 0]
B: Fill Jug 2 [a = 3, b = 4]
C: Empty Jug 1 [a = 0, b = 4]
F: Pour Jug 2 -> Jug 1 [a = 3, b = 1]
C: Empty Jug 1 [a = 0, b = 1]
F: Pour Jug 2 -> Jug 1 [a = 1, b = 0]
B: Fill Jug 2 [a = 1, b = 4]
```

   Note that the above solution is not optimal, because it performs a few redundant operations (fills the first jug, then fills the second, then empties the first one, wasting a fill and empty action). You may not come up with the same solution as mine, but as long as your solution is *valid*, it is correct.

```
Enter A: 3
Enter B: 6
Enter C: 5
Impossible!
```

   To solve this problem, perform a depth-first search through a graph where your state is represented by a pair of integers (a, b), meaning that Jug 1 contains a units of water and Jug 2 contains b units of water. You will start from the state $(0, 0)$ where both jugs are empty, and any state (a, b) where $a + b = C$ is a valid goal state.

   At any state, there are six possible transitions:

(a) Fill Jug 1 to A

(b) Fill Jug 2 to B

(c) Empty Jug 1 to 0

(d) Empty Jug 2 to 0

(e) Pour Jug 1 into Jug 2 (until 2 is full or 1 is empty)

(f) Pour Jug 2 into Jug 1 (until 1 is full or 2 is empty)

If your program finds a solution, it should print out a step-by-step transcript of the solution as shown above. Your solution does not have to be optimal. **The first character on each line must be the letter corresponding to each action type.** You can use a two-dimensional array to represent your set of visited nodes. You will need "back pointers" to print the solution out. When a new state (x, y) is visited, the previous state should be recorded, with a string attached to describe the transition from the previous state to (x, y).

2. **Word Ladder (10 points):** Consider a word ladder that can be formed between two 4-letter words.

lose → lost → lest → best → beat

Each word is formed by changing a single letter in the previous word. In this problem, we would like to find the shortest path that transitions the starting word to the goal word. All valid words must belong to the provided dictionary (file `dictionary4.txt`) and are made up of lowercase letters in the range a-z.

The solution is to perform a breadth-first search through the implicit graph, where each node corresponds to a word, and edges are formed by words that share all but one letter in common. To solve this problem effectively, we will need two data structures, a `HashMap` and a `Queue` (for this problem, you may use the classes provided in the Java standard library).

The `HashMap` will be used for two things: marking words as "visited", and storing back pointers. Both of which can be done with one `HashMap` by associating each entry in the dictionary with an empty string (or null) when it hasn't yet been visited, and updating that value to the predecessor for the breadth-first search when it is visited. Therefore, if a key is matched with a non-empty value, it has been visited.

The `Queue` can be used for a standard FIFO work queue. Keep in mind that `Queue` in Java is an interface that is implemented by `LinkedList`, among other classes.

Please write a program named `WordLadder.java` that loads all of the words from the provided dictionary into a `HashMap` (associated with an empty string or null), asks the user for a start and end word, and then prints whether it's possible, and if so, the shortest sequence between the two words.

```
Enter the start word: lose
Enter the end word: beat
Possible!
lose
lost
lest
best
beat
```

```
Enter the start word: quiz
Enter the end word: exam
Impossible!
```