CS310: Advanced Data Structures and Algorithms

Fall 2021 Programming Assignment 2

Due: Friday, November 12 2021 before midnight

Goals

This assignment aims to help you:

- Practice graphs
- Try greedy algorithms

In this assignment we begin our systematic study of algorithms and algorithm patterns.

Reading

- Graphs (K&T, chapter 3, S&W Chapter 4.1-4.2)
- Greedy algorithms (K&T chapter 4, S&W Chapter 4.4)
- Recursion (from CS210)

Advice: Before writing the code try to run a small example on paper. Think what you would do if you were given the set of instructions or hints and had to do it without a computer. Then start programming. It is always advisable, but especially important in this programming assignment. This assignment is about 80% reading, 20% coding...

Questions

1. **Graphs** (Loosely based on the "small world phenomenon" exercise from S&W, see here:

   http://www.cs.princeton.edu/courses/archive/spring03/cs226/assignments/bacon.html.

   Briefly, the assignment asks you to read in a file containing information about films and actors, and find the Bacon number (or whoever is the center of the universe) of an actor given as a parameter. Your task is to write a class named `DegreesOfSeparationBFS` that builds a shortest paths graph from a center, say Kevin Bacon (or any other actor) based on a `SymbolGraph` read from a file. Then it can take an actor's name and produces this actor's Bacon number and the shortest path to Kevin Bacon (or whoever the center is). The name format is "Last, First" . So for an input of:

   ```
   "Kidman, Nicole"
   ```

   It has to calculate Nicole Kidman's Bacon number (2) and the path in the graph:

   ```
   Kidman, Nicole -> My Life (1993 I) with de Sosa, Ruth -> Planes, Trains & Automobiles (1987) with Bacon, Kevin
   ```

   You may use the `DegreesOfSeparation` class for ideas (see S&W code).

   Make sure that you understand the `SymbolGraph` class very well before you start coding. An illustration appears in the Undirected Graphs class notes.

   Notice that the original assignment asks you to calculate a histogram, but the part that writes the histogram is already implemented by S&W:

https://algs4.cs.princeton.edu/41graph/BaconHistogram.java.html.

Therefore, it is not going to be tested as part of my test suite, but you should print out the histogram to your `memo.txt` file (see below). The original assignment has you read the actor names from the standard input but this is not what you have to do here.

Specific Instructions:

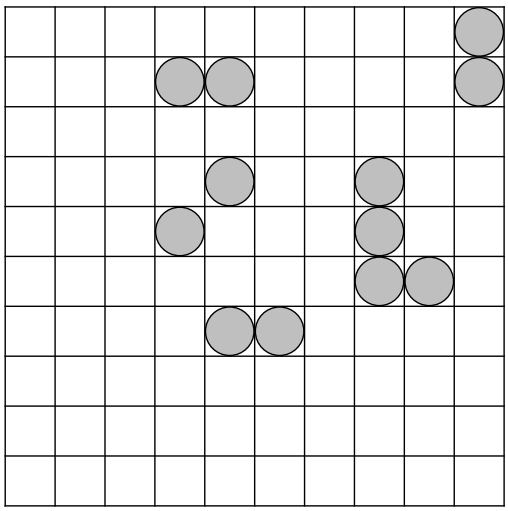
- The `DegreesOfSeparationBFS` class has two class variables: a **SymbolGraph** and a `BreadthFirstPaths`, with the appropriate getters.
- You should be able to instantiate it, so the constructor has to be public. The constructor gets three `Strings`: a File name, a delimiter and a source. For example: `DegreesOfSeparationBFS("movies.txt", "/" ,"Bacon, Kevin")`. Notice the name format is Last, First. The constructor builds the symbol graph and calculates the distance.
- In addition to the getters, the class should implement the following functions (at least. You can add more as you see fit):
  - `int baconNumber(String sink)`. This function gets a `String` in the format "Last, First" (for example "Kidman, Nicole") and calculates the actor's beacon number. In the case of Nicole Kidman it should return 2. If the actor is not in the database **or** there is no path from the center to it, it should return -1.
  - `Stack<Integer> graphPath(String sink)`. This function calculates the path itself. Normally I would advise you to do both in one function (the Bacon number and the path) but it makes my grading easier... Notice that the `Stack<Integer>` holds the indices of the vertices on the path. Look at the `algs4` implementation of paths.
  - I suggest you also add functions for printing the path for debugging purposes but I won't test them.
- The command line parameters should be at least the following three: The input movie file name, the separator and the "center" (Kevin Bacon in the example but you should not hard code it. The center can be any actor). Additional optional command line parameters will be actor names whose Bacon number we want to test. So, for example, you should run your function as follows: `DegreesOfSeparationBFS movies.txt "/" "Bacon, Kevin", "Kidman, Nicole", "Nicholson, Jack"`

2. **Greedy recursive search:** Consider an $N \times N$ grid in which some squares are occupied (see figure below). The squares belong to the same group if they share a common edge. In the figure there is one group of four occupied squares, three groups of two squares, and two individual occupied squares. Assume the grid is represented by a 2 dimensional array. Write a program that:

   (a) Compute the size of a group when a square in the group is given.

   (b) Computes the number of given groups.

   (c) Lists all groups.



Directions:

- For the class setup, use the attached `Grid.java` as a skeleton and follow the setup. The attached file contains a main, the `Grid`, and the `Spot` class. You will have to write only the `groupSize` method, a `calcAllGroups` method and as many helper functions as you see fit.
- I recommend that you also create a function that prints all the groups, for your own debugging purposes (I will not test it).
- Here is one idea: set up a `Set` of Spots to hold spots you've found so far in the cluster. From the current spot-position, greedily add as many direct neighbor spots as possible. For each neighbor spot that is actually newly-found, call recursively from that position. You need a recursion helper here to fill the set. Once the set is filled with spots in the group, the top-level method just returns its size.
- While you may use the S&W code, it's probably best if you just use standard java.
- **Usage:** Your main function should be able to handle two options:
  - `java pa2.Grid 3 7` for example, to search from (3,7), the top occupied square of the L-shaped group. This case should just print out 4.
  - `java pa2.Grid -all`. In this case the program should print all groups, separated by newlines.

  This means you have to prepare for the number of command line arguments to be either one or two, whereas in the former case only the string "-all" is acceptable, and in the latter only two numbers are acceptable. Anything else should print an error message and exit. This is a very common thing in command line tools – the ability to deal with a number of options of variable lengths.
- You should write a separate function that prints everything (I did it in one function, see skeleton) and add a variable to the class. You can write more than one, as long as the data is stored in the `Grid` class. To print all the groups you can take advantage of the `Spot` class `toString` function.

3. Dijkstra's algorithm modification: Modify Dijkstra's algorithm (code is available at the `algs4` library, `DijkstraSP.java`, **Which should be your starting point**), to implement a "tie breaker". When relaxing the edge, i.e., comparing `distTo[w]` to `distTo[v] + e.weight()`, if $distTo[w] > distTo[v] + e.weight()$ then proceed as usual. However, if there are two paths with equal weights leading to a vertex, the one with the *smaller number of edges* will be selected. The big-O runtime should stay the same! So you should also keep track of the number of edges along a path. Before you code, make sure you understand Dijkstra's algorithm or at least its outline. Think about all you have to change in the algorithm to make it work in the same big-O (not too much!). Name your class `DijkstraTieSP.java` .

   Test it on the attached file, `tinyEDW2.txt`, which is a modified version of S&W's `tinyEDW.txt` , containing a case where a tie breaker is needed.

Delivery

Your source files should be under the pa2 package. As before - I recommend to have three directories: `src`, `lib` and `classes` or `bin`, just as in `pa1` – so that there must be a `pa2` subdirectory under src and classes. For compilation and running instructions, see `pa1`. The `algs4.jar` You should include a `memo.txt` in your submission. In particular, the classes should be named as follows:

- `DegreesOfSeparationBFS.java`: usage (when in the `classes` directory): `java -cp .:../lib/algs4.jar pa2.DegreesOfSeparationBFS movies.txt "/" "Bacon, Kevin", "Kidman, Nicole"` (notice the quotes). The `movies.txt` file is an example of an input file - don't hardcode it! Notice that there may be more than four parameters.
- `Grid.java`, usage example: `java pa2.Grid 3 7` or `java pa2.Grid -all`
- `DijkstraTieSP.java`, usage example: `java -cp .:../lib/algs4.jar pa2.DijkstraTieSP tinyEDW2.txt 0`
- In classes that use algs4, don't forget the `-cp .:../lib/algs4.jar` during compilation and runtime.
- In your `memo.txt` file state the following:

- Whether you used late days and if so – how many
- Print out the histogram for the file `movies.txt` with Kevin Bacon in the center into your `memo.txt` file.
- In question 3 – what is the difference between the paths returned by the original implementation of Dijkstra's algorithm and the one you implemented? Specifically, run the `algs4 DijktsraSP` implementation and save the output to the `memo.txt` file using the `tinyEWD2.txt` file given as a handout, starting from 0. Then run your version, save it to the `memo.txt` file. Explain the difference briefly.