Assignment 2: PageRank on Graphs

ver 1.04

Mark Dras

May 23, 2020

1 Introduction

(Note that these specifications may evolve with corrections or clarifications: a changelog will list changes in Section 4.)

You were all introduced to the idea of PageRank — the approach to ranking pages on the Web used by Google — in the previous assignment: given some data structure consisting of vertices / nodes and edges connecting them, the vertices are given a score measuring their importance in some sense which is based on the configuration of edges.

In this assignment, you will be applying the basic ideas behind PageRank to a graph, which is what it's applied to in practice. Typically, PageRank is applied using matrices and operations over them. In this assignment, we're going to take an approach where we imagine there's a hypothetical web surfer who is visiting web pages, who chooses links at random to decide on the next page to visit: this is in fact how it's characterised by the inventors of the PageRank algorithm.[1] This is slower than the matrix approach, but it gives you a feel for what's happening in the graph during PageRank.[2]

If you want to read more deeply about PageRank (although this isn't necessary for the assignment), you'll find that most treatments focus on the matrix representation: having said this, the Wikipedia page[3] is reasonable, as is one by the American Mathematical Society.[4]

What you'll be doing in this assignment is adding methods as specified below to the graph code supplied for the lectures and workshops. You'll be working with directed weighted graphs: the weight on an edge (u, v) represents, for example, the number of links from page u to page v. A graph might then look like in Figure 1. In the graph, the link (edge) from page (vertex) 1 to page (vertex) 3 has weight 2. I'll be referring to this graph below as a running example.

2 Your Tasks

For your tasks, you'll be adding attributes and methods to existing classes given in the code bundle accompanying these specs. Where it's given, **you should use exactly the method stub provided** for implementing your tasks. Don't change the names or the parameters.

You'll see that the supplied classes are similar to the ones provided for week 9 and 10 lecture and practical material, but have a few differences. The main ones are as follows.

[1] `http://ilpubs.stanford.edu:8090/422/`, Sec 2.5

[2] Plus there's no solution you can just download from the Web.

[3] `http://en.wikipedia.org/wiki/PageRank`

[4] `http://www.ams.org/samplings/feature-column/fcarc-pagerank`

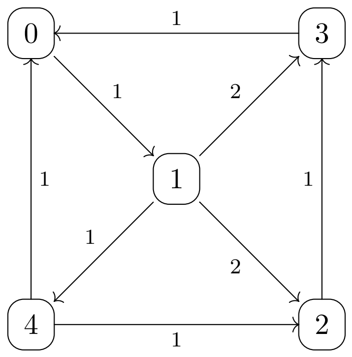

Figure 1: A small sample graph

- The actual model of a websurfer uses true randomness in deciding on links to follow. Because we want your results to be exactly reproducible, and able to pass JUnit tests more easily, we're using pseudo-random numbers. We're using Java's `Random` class[5] with a given seed: this seed is the starting value, and whenever a number is generated from the random number generator, it will always follow the same sequence when started with a particular seed. For this assignment, this is wrapped inside a class `PseudoRand`. The constructor expects an integer value to be used as a seed; it's invoked as follows:

  ```
  PseudoRand p = new PseudoRand(1);
  ```

  You can see what it does with e.g.:

  ```
  for (int i = 0; i < 5; i++)
      System.out.println(p.genPseudoRandDouble());
  ```

- As a consequence, each Vertex is initialised with a seed value, in addition to its integer ID. The seed value will be used in initialising an instance of `PseudoRand` as part of finding the next link to transition to in the process of websurfing.

- As a further consequence, the input files representing graphs differ from the lecture ones, as the seed for each vertex is specified there. The format for a graph input file is then as follows:

  ```
  N
  v1 s1
  v2 s2
  ...
  vN sN
  vi vj wij
  ...
  ```

  N is the number of vertices in the graph; the next N lines list each vertex v_i and its seed s_i; the lines after that give the edges in the graph, for each edge (v_i, v_j) with weight w_{ij}.

- I've written a method in class `Graph`, `readWeightedFromFileWSeedAndSetDirected(String fInName)`, that reads in this kind of graph, so you don't need to do anything special to handle this; it replaces both the reading and setting methods from the lecture classes. You can see how it works in the `main()` method of class `Graph`.

[5] `https://docs.oracle.com/javase/8/docs/api/java/util/Random.html`

2.1 Pass Level

To achieve at least a Pass (50–64%) for the assignment, you should do all of the following.

For this section, you can test your code on the data contained in `tiny-weight.txt`, which contains the graph from the example.

T1 Add two attributes to the class `Vertex`: `visits`, which will allow you to keep track of the number of visits to a vertex; and `weight`, which will allow you to represent a normalised score for the vertex. When a vertex is created, these values should both be set to zero. The actual setting of `weight` is described in Task **T7**. (Note that this is a weight for a vertex, and is different from the weights of edges described above.)

Now write get and set methods for each variable, including at least the following:

```
public Integer getVisits() {
// PRE: -
// POST: returns the value of attribute visits for vertex v
}

public Double getWeight() {
// PRE: -
// POST: returns the value of attribute weight for vertex v
}
```

T2 Also in class `Vertex`, write a method that will return a pseudorandom number in the range $[0, 1)$ by calling the pseudorandom number generator from class `PseudoRand`.

```
public Double getPseudoRandomDouble() {
// PRE: -
// POST: returns a pseudorandom double value
}
```

T3 Continuing in class `Vertex`, write a method that will select a vertex at (pseudo)random from the ordered list of adjacent vertices, according to the probability distribution induced by the weights on the edges. For example, vertex 1 in the example graph has edges to vertices 2, 3 and 4 with weights 2, 2 and 1 respectively. So there should be a 40% chance of choosing vertex 2, 40% of choosing vertex 3, and 20% of choosing vertex 4.[6]

For this particular example, you can make a choice by generating a random number in the range $[0, 5)$ (that is, including 0 but not including 5). If the random number is in the range $[0, 2)$, choose vertex 2; if it's in the range $[2, 4)$, choose vertex 3; and if it's in the range $[4, 5)$, choose vertex 4. You should use an instance of class `PseudoRand` to do this.

The method should then have the following form:

```
public Integer getPseudoRandomLink() {
// PRE: vertex v has non-empty adjacency list
// POST: returns a vertex ID randomly selected from adjacent vertices,
//         according to distribution of edge weights
//       returns vertex ID v if empty adjacency list
}
```

[6] If you don't order the vertices according to their natural ordering, as in this example, you'll get different results for the pseudo-randomly chosen link, and may not match some of the JUnit tests.

T4 In class `GraphApplic`, you'll be using `getPseudoRandomLink()` to jump from one page (vertex) to the next. You'll be writing a method that, starting at a given vertex, surfs the graph by (for the most part) choosing at each vertex a random adjacent vertex u, then visiting u and choosing again there. At each vertex that you jump **from**,[7] the vertex attribute `visits` should be incremented by 1. E.g. in the sample graph we might start at vertex 0, then jump to vertex 1 (no choice), then to vertex 3.

The method should then have the following form:[8]

```
public Integer surfNoJump(Integer v, Integer n) {
// PRE: v is vertex to start surf; n >= 0
// POST: surfs the graph randomly for n moves,
//       choosing adjacent vertex according to distribution of edge weights
//       modifies # visits in vertices
//       returns last visited vertex at end of surfing
}
```

If you have modified the appropriate `print()` method(s) to print out vertex visits (in parentheses after the vertex ID), the call

```
g.surfNoJump(g.getFirstVertex(), 1000);
```

will produce a pattern of visits as follows (assuming that you're using the specification of the graph from file `tiny-weight.txt`, which specifies a particular seed for each vertex):

```
Number of vertices is 5
Number of edges is 8
vertex 0 (272) : 1
vertex 1 (272) : 2 3 4
vertex 2 (152) : 3
vertex 3 (253) : 0
vertex 4 (51) : 0 2
```

T5 This task extends **T4**.

Instead of only following links, some proportion of the time (say 10%), you don't choose an adjacent vertex: you just choose a vertex uniformly (pseudo)randomly from the graph. You can imagine that your web surfer got bored with following links, and clicked Google's *I'm Feeling Lucky* button.[9] So say you've arrived at vertex 3; you flip a (biased) coin, and decide to jump to a (pseudo)randomly chosen vertex, rather than following the (only) link to vertex 0. You consider vertices 0, 1, 2, 3, 4 and choose uniformly randomly from among them: you then jump to (say) vertex 2. Again, you should use an instance of class `PseudoRand` to generate this value.

The method should then have the following form:

```
public Integer surfWithJump(Integer v, Integer n, Double jumpThreshold) {
// PRE: v is vertex to start surf; n >= 0; 0 <= jumpThreshold <= 1.0
// POST: surfs the graph randomly for n moves,
//       choosing adjacent vertex according to distribution of edge weights
//         if random number is below jumpThreshold,
```

[7]That is, the finishing point vertex for the surf isn't included.

[8]What happens for vertices that have no edges to other vertices? We'll fix that in the next section. For the Pass level, there will be no graphs that have vertices like this.

[9]If you're using the matrix approach, this guarantees the matrix will be ergodic, which is necessary for some of the matrix calculations.

```
//       choosing any vertex uniformly randomly otherwise;
//       modifies # visits in vertices
//       returns last visited vertex at end of surfing
}
```

Note that you should use each random number only once. For example, you should use one random number for testing against the threshold for jumping, and then another random number for choosing a vertex.

If you have modified the appropriate `print()` method(s) to print out vertex visits (in parentheses after the vertex ID), the call

```
g.surfWithJump(g.getFirstVertex(), 10000, 0.9);
```

will produce a patterns of visits something like the following (assuming that you're using the specification of the graph from file `tiny-weight.txt`, which specifies a particular seed for each vertex):

```
Number of vertices is 5
Number of edges is 8
vertex 0 (2719) : 1
vertex 1 (2661) : 2 3 4
vertex 2 (1470) : 3
vertex 3 (2481) : 0
vertex 4 (669) : 0 2
```

T6 As with the previous assignment, you'll be using this score to rank pages (vertices). Write the following method in class `GraphApplic`:

```
public ArrayList<Integer> topN(Integer N) {
// PRE: none
// POST: returns N vertices with highest number of visits, in order;
//       returns all vertices if <N in the graph;
//       returns vertices ranked 1,..,N,N+1,..,N+k if these k have the
//         same number of visits as vertex N
}
```

For the example above in **T5**., after `surfWithJump()`, the call

```
g.topN(2);
```

would return an ArrayList with 0 in the first slot, and 1 in the second.

2.2 Credit Level

To achieve at least a Credit (65–74%) for the assignment, you should do the following. You should also have completed all the Pass-level tasks.

For this section, you should also test your code on the data in `medium-weight.txt`. This contains 50 vertices, so the various methods will take longer (sometimes a lot longer) on it.

T7 Write the following method in class `GraphApplic`:

```
public void setVertexWeights () {
// PRE: -
// POST: set weights of each vertex v to be v.visits divided by
//         the total of visits for all vertices
}
```

These weights correspond to the PageRank for a vertex.

Assuming a further expansion of the appropriate `print()` to give both `visits` and `weight` in parentheses after each vertex, for the example in **T5** (where the total number of visits is 10000) you would have the following:

```
Number of vertices is 5
Number of edges is 8
vertex 0 (2719,0.2719) : 1
vertex 1 (2661,0.2661) : 2 3 4
vertex 2 (1470,0.1470) : 3
vertex 3 (2481,0.2481) : 0
vertex 4 (669,0.0669) : 0 2
```

T8 Now we'll fix `surfWithJump` from above. For a vertex v which has no out-edges (a SINK), choose the next vertex to visit uniformly randomly over all vertices in the graph (including v). (This is the same process as for the jump.)

T9 It's an interesting fact that carrying out this surfing process as described above will eventually lead the weights in **T7** to stabilize and converge on particular values.[10] In this part of the assignment, you'll be surfing the web until the weights generated by the surfing converge. (The starting point for the surfing here isn't important.)[11] This could go on and on, with small changes continuing at the third or fourth or seventeenth decimal place; so we'll stop the process once there's no change at a given number of decimal places.

Write the following method in class `GraphApplic`:

```
public void convergenceWeights(Integer dp, Double jumpThreshold) {
// PRE: dp >= 0 representing number of decimal places,
//      0 <= jumpThreshold <= 1.0
// POST: web is surfed until all weights are constant to dp decimal places,
//       for at least one iteration
}
```

Suppose you ran the example from **T5** for another 10000 iterations, and got the following.

```
Number of vertices is 5
Number of edges is 8
Node 0 (5456,0.2728) : 1
Node 1 (5310,0.2655) : 2 3 4
Node 2 (2939,0.14695) : 3
Node 3 (4949,0.24745) : 0
Node 4 (1346,0.0673) : 0 2
```

This wouldn't have converged to 3 decimal places, as e.g. vertex 0 is different at 3 decimal places from the previous.

Note that increasing the `dp` parameter for `convergenceWeights()` will make the method take dramatically longer.

[10]The random process that the surfer is following is called a MARKOV CHAIN. The phenomenon whereby the surfing converges on a particular value is called MIXING.

[11]As this description suggests, you can call your previously written functions for surfing the graph. If you do, exactly how you handle the number of iterations to try before testing for convergence is up to you.

2.3 (High) Distinction Level

To achieve at least a Distinction (75–100%) for the assignment, you should do the following. You should also have completed all the Credit-level tasks.

T10 The HITTING TIME for a page is the expected number of moves between visits to that page.

Write the following method in class `GraphApplic`:

```
public Integer surfWithJumpUntilHit(Integer v, Integer n, Double jumpThreshold) {
// PRE: v is vertex to start surf; n >= 0; 0 <= jumpThreshold <= 1.0
// POST: surfs the graph randomly until visit v for second time (maximum n moves),
//       choosing adjacent vertex according to distribution of edge
//         weights if random number is below jumpThreshold,
//       choosing any vertex uniformly randomly otherwise;
//       modifies # visits in vertices
//       returns number of vertices visited
}
```

Suppose we call the method as follows:

`g.surfWithJumpUntilHit(g.getFirstVertexID(), 10000, 0.9);`

Suppose we then moved successively to vertices 1, 2, 3 and then back to 0. The number of moves — and the value returned by the method — would then be 4.

T11 To get an accurate estimate for hitting time, given the underlying random process, we would need to average this over a number of iterations. That is, call `surfWithJumpUntilHit()` with a given starting vertex, storing the number of moves between visits; then repeat this for a given number of iterations, and take the mean.

Write the following method in class `GraphApplic`:

```
public Double averageHittingTime(Integer v, Integer n, Integer maxHits, Double jumpThreshold) {
// PRE: n >= 1 is number of iterations for averaging; maxHits >= 0; 0 <= jumpThreshold <= 1.0
// POST: returns average number of vertices visited until hitting, across n iterations,
//         (not including those which stopped because they reached maxHits)
//       returns 0 if all iterations reached maxVisits
}
```

Excluding cases where you reach maxHits means that your average might be over fewer than n iterations: don't do replacement iterations for these.

You can check your answer here because, for a sufficiently accurate estimate, it should be equal to the reciprocal of the weight of the vertex.

Calling the method as follows:

`g.averageHittingTime(g.getFirstVertexID(), 10000, 1000000, 0.9);`

I get 3.637.

T12 The COVER TIME is the time required for the random surfer to visit every page at least once, starting from a random page.

Write the following method in class `GraphApplic`:

```
public Integer surfWithJumpUntilCover(Integer v, Integer n, Double jumpThreshold) {
// PRE: v is vertex to start surf; n >= 0; 0 <= jumpThreshold <= 1.0
// POST: surfs the graph randomly until all vertices visited (with maximum n vertices visited),
//       choosing adjacent vertex according to distribution of edge
//         weights if random number is below jumpThreshold,
//       choosing any vertex uniformly randomly otherwise;
//       modifies # visits in vertices
//       returns number of vertices visited
}
```

T13 Write the following method in class `GraphApplic`:

```
public Double averageCoverTime(Integer n, Integer maxVisits, Double jumpThreshold) {
// PRE: n >= 1 is number of iterations for averaging; maxVisits >= 0; 0 <= jumpThreshold <= 1.0
// POST: returns average number of vertices visited until cover, across n iterations,
//         (not including those which stopped because they reached maxVisits)
//         randomly selecting start vertex each iteration
//       returns 0 if all iterations reached maxVisits
}
```

This averaging process is similar to `averageHittingTime()`, except that when calling `surfWithJumpUntilCover()`, you should do so **with a random vertex**.

T14 It's possible to change the weight of a vertex by changing the structure of the graph. You can think of this as adding links from your page to others. You'll notice that if you add links from vertex 23 to all others (with weight 1.0) in `medium-weight.txt`, the weight for vertex 23 drops. It's possible to choose to add links in such a way that the weight (PageRank) of a given vertex increases. (This is what happens in Search Engine Optimisation.)

Your goal in this task is test adding edges to a graph to see how best to improve the weight of a given vertex. For example, if the vertex of interest is 23, you will try adding edge (23,0), assuming it doesn't already exist, and (surfing until convergence) record what the weight of vertex 23 is; then do the same for edge (23,1) (without the previously added edge (23,0)), again assuming it doesn't exist, and see if the resulting weight for vertex 23 is better than the previously recorded one; and so on.

Consequently, write the following method in class `GraphApplic`:

```
public Integer boostVertex(Integer v, Integer dp) {
// PRE: v is a vertex in the graph
// POST: returns a vertex v2 (!= v) such that when edge (v,v2,1.0) is added to the graph,
//         the weight of v is larger than when edge (v,v3,1.0) is added to the graph
//         (for any v3), when the graph is surfed to convergence
//       if there is no such vertex v2 (i.e. v is already connected to all other vertices),
//         return v
//       edges are only added if they do not already exist in the graph
}
```

(Note that running this on `medium-weight` can be quite slow.)

You'll probably find it helpful to add methods to class `Graph` for adding edges to, and deleting edges from, a graph.

3 What To Hand In

In the submission page on iLearn for this assignment you must include the following:

Submit a zip file consisting of all the Java class code (i.e. the `.java` files) in the package from the original assignment code bundle. Follow the instructions in iLearn on how to create the zipfile for submission.

Your file must leave unchanged the specification of already implemented functions, and include your implementations of your selection of method stubs outlined above.

Do not change the names of the method stubs because the auto-tester assumes the names given. Do not change the package statement. You may however include additional auxiliary methods if you need them.

Please note that we are unable to check individual submissions and so it is very important to abide by the above submission instructions.

4 Changelog

- **6/5/20**: Assignment draft released.
- **11/5/20**:
  - Clarified in **T3** that the vertices should be ordered when pseudo-randomly choosing which one to jump to.
  - Fixed description around `surfNoJump()` in **T4** to align with with JUnit test, so that the start vertex of the surf but not the end vertex is included.
  - Fixed `convergenceWeights()` in **T9** to align with JUnit test (threshold parameter was missing in specs, present in JUnit).
- **16/5/20**:
  - Clarified one-time use of pseudo-random numbers.
  - Clarified maxHits in **T11**.
- **19/5/20**:
  - Corrected `getVisits(), getWeight()` method definition to match code framework and JUnit tests.
- **22/5/20**:
  - Corrected `getPseudoRandomLink()` method definition to match code framework and JUnit tests.