Project 6 (WordNet) Checklist

Prologue

Project goal: find the shortest common ancestor of a digraph in WordNet, a semantic lexicon for the English language

Files:

- `project6.pdf` (project writeup)
- `project6_checklist.pdf` (checklist)
- `project6.zip` (starter files for the exercises/problems, `report.txt` file for the project report, and test data files)

Exercises

Exercise 1. (*Graph Properties*) Consider an undirected graph G with V vertices and E edges.

- The *degree distribution* of G is a function mapping each degree value in G to the number of vertices with that value.
- The *average degree* of G is $\frac{2E}{V}$.
- The *average path length* of G is the average length of all the paths in G.
- The local clustering coefficient C_i for a vertex v_i is the number of edges that actually exist between the vertices in its neighbourhood divided by the number of edges that could possibly exist between them, which is $\frac{V(V-1)}{2}$. The *global clustering coefficient* of G is $\frac{1}{V}\sum_i^V C_i$.

Exercises

Implement a data type called `GraphProperties` with the following API to compute the aforementioned graph properties:

GraphProperties	
`GraphProperties(Graph G)`	computes graph properties for the undirected graph `G`
`RedBlackBinarySearchTreeST<Integer, Integer> degreeDistribution()`	returns the degree distribution of the graph
`double averageDegree()`	returns the average degree of the graph
`double averagePathLength()`	returns the average path length of the graph
`double clusteringCoefficient()`	returns the global clustering coefficient of the graph

```
>_ ~/workspace/project6

$ java GraphProperties data/tinyG.txt
Degree distribution:
  1: 3
  2: 4
  3: 5
  4: 1
Average degree         =   2.308
Average path length    =   3.090
Clustering coefficient =   0.256
```

Exercises

GraphProperties.java

```java
import dsa.BFSPaths;
import dsa.Graph;
import dsa.RedBlackBinarySearchTreeST;
import stdlib.In;
import stdlib.StdOut;

public class GraphProperties {
    private RedBlackBinarySearchTreeST<Integer, Integer> st; // degree -> frequency
    private double avgDegree;                                  // average degree of the graph
    private double avgPathLength;                              // average path length of the graph
    private double clusteringCoefficient;                      // clustering coefficient of the graph

    // Computes graph properties for the undirected graph G.
    public GraphProperties(Graph G) {
        ...
    }

    // Returns the degree distribution of the graph (a symbol table mapping each degree value to
    // the number of vertices with that value).
    public RedBlackBinarySearchTreeST<Integer, Integer> degreeDistribution() {
        ...
    }

    // Returns the average degree of the graph.
    public double averageDegree() {
        ...
    }

    // Returns the average path length of the graph.
    public double averagePathLength() {
        ...
    }

    // Returns the global clustering coefficient of the graph.
    public double clusteringCoefficient() {
```

Exercises

GraphProperties.java

```java
        ...
    }

    // Returns true if G has an edge between vertices v and w, and false otherwise.
    private static boolean hasEdge(Graph G, int v, int w) {
        for (int u : G.adj(v)) {
            if (u == w) {
                return true;
            }
        }
        return false;
    }

    // Unit tests the data type. [DO NOT EDIT]
    public static void main(String[] args) {
        In in = new In(args[0]);
        Graph G = new Graph(in);
        GraphProperties gp = new GraphProperties(G);
        RedBlackBinarySearchTreeST<Integer, Integer> st = gp.degreeDistribution();
        StdOut.println("Degree distribution:");
        for (int degree : st.keys()) {
            StdOut.println("  " + degree + ": " + st.get(degree));
        }
        StdOut.printf("Average degree         = %7.3f\n", gp.averageDegree());
        StdOut.printf("Average path length    = %7.3f\n", gp.averagePathLength());
        StdOut.printf("Clustering coefficient = %7.3f\n", gp.clusteringCoefficient());
    }
}
```

Exercises

Exercise 2. (*DiGraph Properties*) Consider a digraph G with V vertices.

- G is a *directed acyclic graph (DAG)* if it does not contain any directed cycles.
- G is a *map* if every vertex has an outdegree of 1.
- A vertex v is a *source* if its indegree is 0.
- A vertex v is a *sink* if its outdegree is 0.

Implement a data type called `DiGraphProperties` with the following API to compute the aforementioned digraph properties:

DiGraphProperties

`DiGraphProperties(DiGraph G)`	computes graph properties for the digraph `G`
`boolean isDAG()`	returns `true` if the digraph is a DAG, and `false` otherwise
`boolean isMap()`	returns `true` if the digraph is a map, and `false` otherwise
`Iterable<Integer> sources()`	returns all the sources in the digraph
`Iterable<Integer> sinks()`	returns all the sinks in the digraph

>_ ~/workspace/project6

```
$ java DiGraphProperties data/tinyDG.txt
Sources: 7
Sinks: 1
Is DAG? false
Is Map? false
```

Exercises

DiGraphProperties.java

```java
import dsa.DiCycle;
import dsa.DiGraph;
import dsa.LinkedBag;
import stdlib.In;
import stdlib.StdOut;

public class DiGraphProperties {
    private boolean isDAG;                // is the digraph a DAG?
    private boolean isMap;                // is the digraph a map?
    private LinkedBag<Integer> sources; // the sources in the digraph
    private LinkedBag<Integer> sinks;   // the sinks in the digraph

    // Computes graph properties for the digraph G.
    public DiGraphProperties(DiGraph G) {
        ...
    }

    // Returns true if the digraph is a directed acyclic graph (DAG), and false otherwise.
    public boolean isDAG() {
        ...
    }

    // Returns true if the digraph is a map, and false otherwise.
    public boolean isMap() {
        ...
    }

    // Returns all the sources (ie, vertices without any incoming edges) in the digraph.
    public Iterable<Integer> sources() {
        ...
    }

    // Returns all the sinks (ie, vertices without any outgoing edges) in the digraph.
    public Iterable<Integer> sinks() {
        ...
```

Exercises

DiGraphProperties.java

```java
    }

    // Unit tests the data type. [DO NOT EDIT]
    public static void main(String[] args) {
        In in = new In(args[0]);
        DiGraph G = new DiGraph(in);
        DiGraphProperties gp = new DiGraphProperties(G);
        StdOut.print("Sources: ");
        for (int v : gp.sources()) {
            StdOut.print(v + " ");
        }
        StdOut.println();
        StdOut.print("Sinks: ");
        for (int v : gp.sinks()) {
            StdOut.print(v + " ");
        }
        StdOut.println();
        StdOut.println("Is DAG? " + gp.isDAG());
        StdOut.println("Is Map? " + gp.isMap());
    }
}
```

Problems

The guidelines for the project problems that follow will be of help only if you have read the description of the project and have a general understanding of the problems involved. It is assumed that you have done the reading.

Problems

Problem 1. (*`WordNet` Data Type*)

Hints:

- ⇝ Instance variables
  - ⇝ A symbol table that maps a synset noun to a set of synset IDs (a synset noun can belong to multiple synsets), `RedBlackBST<String, SET<Integer>> st`
  - ⇝ A symbol table that maps a synset ID to the corresponding synset string, `RedBlackBST<Integer, String> rst`
  - ⇝ For shortest common ancestor computations, `ShortestCommonAncestor sca`
- ⇝ `WordNet(String synsets, String hypernyms)`
  - ⇝ Initialize instance variables `st` and `rst` appropriately using the synset file
  - ⇝ Construct a `DiGraph` object `G` (representing a rooted DAG) with V vertices (equal to the number of entries in the synset file), and add edges to it, read in from the hypernyms file
  - ⇝ Initialize `sca` using `G`

Problems

- `Iterable<String> nouns()`
  - Return all WordNet nouns
- `boolean isNoun(String word)`
  - Return `true` if the given word is a synset noun, and `false` otherwise
- `String sca(String noun1, String noun2)`
  - Use `sca` to compute and return a synset that is a shortest common ancestor of the given nouns
- `int distance(String noun1, String noun2)`
  - Use `sca` to compute and return the length of the shortest ancestral path between the given nouns

Problems

Problem 2. (`ShortestCommonAncestor` *Data Type*)

Hints:

- Instance variable
  - A rooted DAG, `DiGraph G`
- `ShortestCommonAncestor(DiGraph G)`
  - Initialize instance variable appropriately
- `private SeparateChainingHashST<Integer, Integer> distFrom(int v)`
  - Return a map of vertices reachable from `v` and their respective shortest distances from `v`, computed using BFS starting at `v`
- `int length(int v, int w)`
  - Return the length of the shortest ancestral path between `v` and `w`; use `ancestor(int v, int w)` and `distFrom(int v)` methods to implement this method
- `int ancestor(int v, int w)`
  - Return the shortest common ancestor of vertices `v` and `w`; to compute this, enumerate the vertices in `distFrom(v)` to find a vertex `x` that is also in `distFrom(w)` and has the minimum value for `distFrom(v)[x] + distFrom(w)[x]`

Problems

- private int[] triad(Iterable<Integer> A, Iterable<Integer> B)
  - Return a 3-element array consisting of a shortest common ancestor a of vertex subsets A and B, a vertex v from A, and a vertex w from B such that the path v-a-w is the shortest ancestral path of A and B; use length(int v, int w) and ancestor(int v, int w) methods to implement this method
- int length(Iterable<Integer> A, Iterable<Integer> B)
  - Return the length of the shortest ancestral path of vertex subsets A and B; use triad((Iterable<Integer> A, Iterable<Integer> B) and distFrom(int v) methods to implement this method
- int ancestor(Iterable<Integer> A, Iterable<Integer> B)
  - Return a shortest common ancestor of vertex subsets A and B; use triad((Iterable<Integer> A, Iterable<Integer> B) to implement this method

Problems

Problem 3. (*`Outcast` Data Type*)

Hints:

- ⇝ Instance variable
  - ⇝ The WordNet semantic lexicon, `WordNet wordnet`
- ⇝ `Outcast(WordNet wordnet)`
  - ⇝ Initialize instance variable appropriately
- ⇝ `String outcast(String[] nouns)`
  - ⇝ Compute the sum of the distances (using `wordnet`) between each noun in `nouns` and every other, and return the noun with the largest distance

Problems

The `data` directory has a number of sample input files for testing

- See project writeup for the format of the synset (`synset*.txt`) and hypernym (`hypernym*.txt`) files
- The `digraph*.txt` files representing digraphs can be used as inputs for `ShortestCommonAncestor`

```
>_ ~/workspace/project6

$ cat data/digraph1.txt
12
11
 6  3
 7  3
 3  1
 4  1
 5  1
 8  5
 9  5
10  9
11  9
 1  0
 2  0
```

- The `outcast*.txt` files, each containing a list of nouns, can be used as inputs for `Outcast`

```
>_ ~/workspace/project6

$ cat data/outcast5a.txt
horse
zebra
cat
bear
table
```

Epilogue

Use the template file `report.txt` to write your report for the project

Your report must include:

- Time (in hours) spent on the project
- Difficulty level (1: very easy; 5: very difficult) of the project
- A short description of how you approached each problem, issues you encountered, and how you resolved those issues
- Acknowledgement of any help you received
- Other comments (what you learned from the project, whether or not you enjoyed working on it, etc.)

Epilogue

Before you submit your files:

- ⇝ Make sure your programs meet the style requirements by running the following command on the terminal

```
>_ ~/workspace/project6

$ check_style src/*.java
```

- ⇝ Make sure your code is adequately commented, is not sloppy, and meets any project-specific requirements, such as corner cases and time complexities
- ⇝ Make sure your report uses the given template, isn't too verbose, doesn't contain lines that exceed 80 characters, and doesn't contain spelling mistakes

Epilogue

Files to submit:

1. `GraphProperties.java`
2. `DiGraphProperties.java`
3. `WordNet.java`
4. `ShortestCommonAncestor.java`
5. `Outcast.java`
6. `report.txt`