Project 6 (WordNet) Checklist

Prologue

Project goal: find the shortest common ancestor of a digraph in WordNet, a semantic lexicon for the English language that computational linguists and cognitive scientists use extensively

Files:

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

Exercises

Exercise 1. (*Graph Properties*) The *eccentricity* of a vertex v is the length of the shortest path from that vertex to the furthest vertex from v. The *diameter* of a graph is the maximum eccentricity of any vertex. The *radius* of a graph is the smallest eccentricity of any vertex. A *center* is a vertex whose eccentricity is the radius. Implement a data type `GraphProperties` that supports the following API to calculate the aforementioned graph properties:

GraphProperties	
`GraphProperties(Graph G)`	calculate graph properties for the undirected graph G
`int eccentricity(int v)`	eccentricity of vertex v
`int diameter()`	diameter of G
`int radius()`	radius of G
`Iterable<Integer> centers()`	centers of G

>_ ~/workspace/project6

```
$ java edu.umb.cs210.p6.GraphProperties data/tinyG.txt
Diameter = 7
Radius   = 4
Centers  = 0 4 6
```

Exercises

GraphProperties.java

```java
package edu.umb.cs210.p6;

import dsa.BreadthFirstPaths;
import dsa.Graph;
import dsa.LinkedQueue;
import stdlib.In;
import stdlib.StdOut;
import stdlib.StdStats;

public class GraphProperties {
    private int[] eccentricities;
    private int diameter;
    private int radius;
    private LinkedQueue<Integer> centers;

    // Calculate graph properties for the graph G.
    public GraphProperties(Graph G) {
// *******YOU DO NOT NEED TO CHECK THIS CORNER CASE:
//      throw new IllegalArgumentException("G is not connected");
// ****** Ignore the corner case requirement for this problem ***************

        ...
    }

    // Eccentricity of v.
    public int eccentricity(int v) {
        ...
    }

    // Diameter of G.
    public int diameter() {
        ...
    }

    // Radius of G.
    public int radius() {
```

Exercises

GraphProperties.java

```java
        ...
    }

    // Centers of G.
    public Iterable<Integer> centers() {
        ...
    }

    // Test client. [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);
        StdOut.println("Diameter = " + gp.diameter());
        StdOut.println("Radius   = " + gp.radius());
        StringBuilder centers = new StringBuilder();
        for (int v : gp.centers()) centers.append(v).append(" ");
        StdOut.println("Centers  = " + centers.toString());
    }
}
```

Exercises

Exercise 2. (*Degrees*) The *indegree* of a vertex in a digraph is the number of directed edges that point to that vertex. The *outdegree* of a vertex in a digraph is the number of directed edges that emanate from that vertex. No vertex is reachable from a vertex of outdegree 0, which is called a *sink*; a vertex of indegree 0, which is called a *source*, is not reachable from any other vertex. A digraph where self-loops are allowed and every vertex has outdegree 1 is called a map (a function from the set of integers from 0 to $V-1$ onto itself). Implement a data type `Degrees` that implements the following API to calculate the aforementioned properties of a digraph:

Degrees

`Degrees(Digraph G)`	construct a `Degrees` object from a digraph G
`Iterable<Integer> sources()`	sources of G
`Iterable<Integer> sinks()`	sinks of G
`boolean isMap()`	is G a map?

~/workspace/project6

```
$ java edu.umb.cs210.p6.Degrees data/tinyDG.txt
Sources = 7
Sinks   = 1
Is Map  = false
```

Exercises

Degrees.java

```java
package edu.umb.cs210.p6;

import dsa.DiGraph;
import dsa.LinkedQueue;
import stdlib.In;
import stdlib.StdOut;

public class Degrees {
    private DiGraph G;
    private int[] outdegree;
    private int[] indegree;

    // Construct a Degrees object from a digraph G.
    public Degrees(DiGraph G) {
        setDegrees(G);
        ...
    }

    // Sources of G.
    public Iterable<Integer> sources() {
        ...
    }

    // Sinks of G.
    public Iterable<Integer> sinks() {
        ...
    }

    // Is G a map?
    public boolean isMap() {
        ...
    }

    // helper method calculates the in and out degrees of each vertex
    private void setDegrees(DiGraph g) {
        outdegree = new int[g.V()];
```

Exercises

Degrees.java

```java
        indegree = new int[g.V()];
        for (int from = 0; from < g.V(); from++) {
            for (int to : g.adj(from)) {
                outdegree[from]++;
                indegree[to]++;
            }
        }
    }

    // Test client. [DO NOT EDIT]
    public static void main(String[] args) {
        In in = new In(args[0]);
        DiGraph G = new DiGraph(in);
        Degrees degrees = new Degrees(G);

        StringBuilder sources = new StringBuilder();
        StringBuilder   sinks = new StringBuilder();
        for (int v : degrees.sources()) sources.append(v).append(" ");
        for (int v : degrees.sinks())     sinks.append(v).append(" ");

        StdOut.println("Sources = " + sources.toString());
        StdOut.println("Sinks   = " + sinks.toString());
        StdOut.println("Is Map  = " + degrees.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`
  - `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 the nouns as an iterable object
- `boolean isNoun(String word)`
  - Return `true` if the given word is a synset noun, and `false` otherwise
- `String sca(String noun1, String noun2)`
  - Return the shortest common ancestor of the given nouns, computed using `sca`
- `int distance(String noun1, String noun2)`
  - Return the length of the shortest ancestral path between the given nouns, computed using `sca`

Problems

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

Hints:

- Instance variable
  - A rooted DAG, `Digraph G`
- `ShortestCommonAncestor(Digraph G)`
  - Initialize instance variable appropriately
- `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 ancestor(int v, int w)`
  - Return the shortest common ancestor of vertices `v` and `w`; to compute this, enumerate the vertices in `distFrom(v)`, and find a vertex `x` that is also in `distFrom(w)` and yields the minimum value for `dist(v, x) + dist(x, w)`

Problems

- `int length(int v, int w)`
  - Return the length of the shortest ancestral path between `v` and `w`; use `int length(int v, int w)` and `int ancestor(int v, int w)` to implement this method
- `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 `int length(int v, int w)` and `int ancestor(int v, int w)` 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 `int[] triad((Iterable<Integer> A, Iterable<Integer> B)` and `SeparateChainingHashST<Integer, Integer> distFrom(int v)` to implement this method
- `int ancestor(Iterable<Integer> A, Iterable<Integer> B)`
  - Return a shortest common ancestor of vertex subsets `A` and `B`; use `int[] triad((Iterable<Integer> A, Iterable<Integer> B)` to implement this method

Problems

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

Hints:

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

Problems

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

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

```
>_ ~/workspace/project6

$ more 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 files `outcast*.txt`, each containing a list of nouns, can be used as inputs for the test client in `Outcast`

```
>_ ~/workspace/project6

$ more outcast5.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 <program>
```

  where `<program>` is the fully-qualified name of the program

- ⇝ Make sure your code is adequately commented, is not sloppy, and meets any project-specific requirements, such as corner cases and running time
- ⇝ 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. `Degrees.java`
3. `WordNet.java`
4. `ShortestCommonAncestor.java`
5. `Outcast.java`
6. `report.txt`