Project 6 – WordNet

Grading Standards

Standards-based Grading (percentage of indicated max points):
Outstanding – 100%, Excellent – 95%, Acceptable – 87%, Unacceptable – 75%, No Basis – 0%

1. I can produce a complete and professional project plan. (20 pts)
2. I can produce an externally to correct solution to WordNet as demonstrated by test cases passed. (30 pts – 10 pts per required API)
3. I can produce internally correct solutions to by following the APIs, correctly applying the concepts covered in Unit 6, not using content beyond what we have discussed in class, and adhering to the library restriction in the deliverables section of this document. (20 pts)
4. I can produce solutions that use good style as laid out by the guidelines in this document. (10 pts)
5. I can produce a complete and professional project reflection. (20 pts)

Project 6 – WordNet

Objective

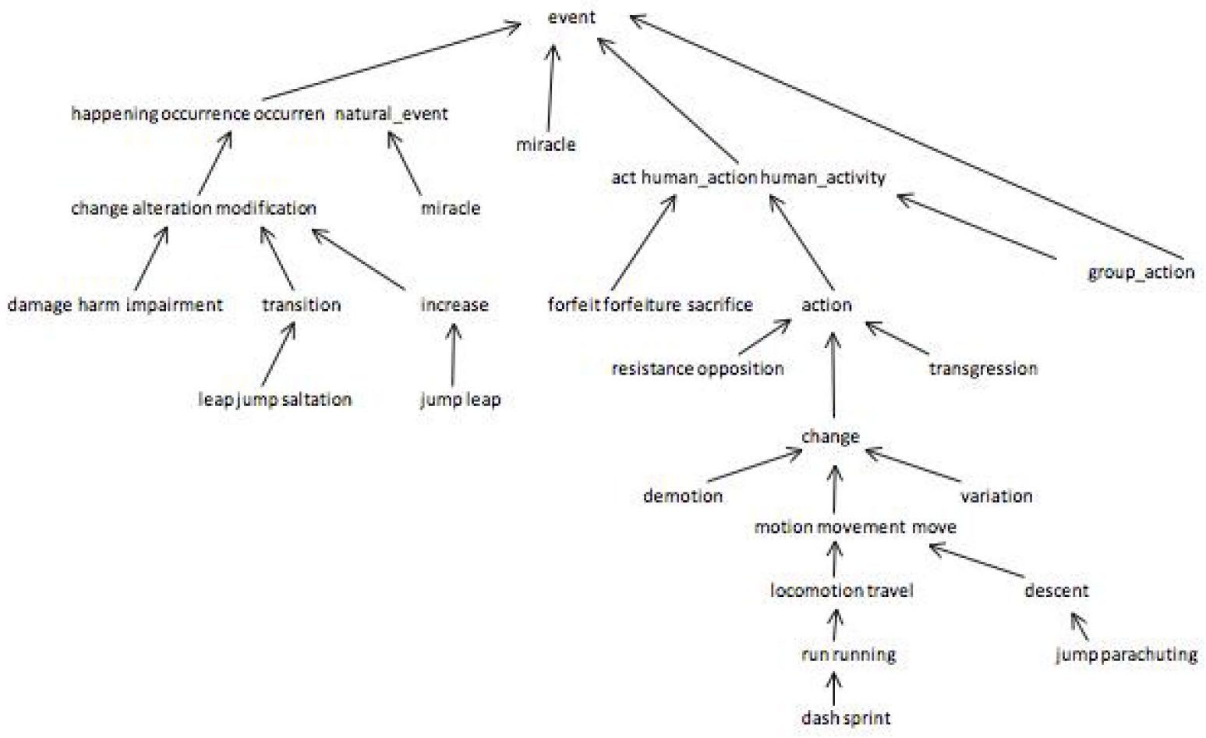
In this assignment, you will implement three data types: WordNet, SAP, and Outcast. You should implement the SAP data type first. WordNet depends on SAP and Outcast depends on WordNet. This project will have two checkpoints – one for SAP and one for WordNet and Outcast. All three classes have a pre-populated main method to be used for testing. You may change the input files for these to test different scenarios.

Background

WordNet is a semantic lexicon for the English language that is used extensively by computational linguists and cognitive scientists; for example, it was a key component in IBM's Watson. WordNet groups words into sets of synonyms called *synsets* and describes semantic relationships between them. One such relationship is the *is-a* relationship, which connects a *hyponym* (more specific synset) to a *hypernym* (more general synset). For example, a *plant organ* is a hypernym of *carrot* and *plant organ* is a hypernym of *plant root*.

The WordNet Digraph

Your first task it to build the wordnet digraph. Each vertex v is an integer that represents a synset, and each directed edge v->w represents that w is a hypernym of v. The wordnet digraph is a *rooted DAG*: it is acyclic and has one vertex – the root – that is an ancestor of every other vertex. However, it is not necessarily a tree because a synset can have more than one hypernym. A small subgraph of the wordnet digraph is illustrated below.



The WordNet Input File Formats

We now describe the two data files that you will use to create the wordnet digraph. The files are in *CSV format*: each line contains a sequence of fields, separated by commas.

- **List of noun synsets**. The file `synsets.txt` lists all the (noun) synsets in WordNet. The first field is the *synset id* (an integer), the second field is the synonym set (or *synset*), and the third field is its dictionary definition (or *gloss*). For example, the line:

  ```
  36, AND_circuit AND_gate, a circuit in a computer that fires only when all of its inputs fire
  ```

  means that the synset `{AND_circuit, AND_gate}` has an id number of 36 and it's gloss is `a circuit in a computer that fires only when all of its inputs fire.` The individual nouns that comprise a synset are separated by spaces (and a synset element is not permitted to contain a space). The S synset ids are numbered 0 through S-1; the id numbers will appear consecutively in the synset file.
- **List of hypernyms.** The file `hypernyms.txt` contains the hypernym relationships: The first field is a sysnset id; sybsequent fields are the id numbers of the sysnset's hypernyms. For example the following line: `164, 21012,56099` means that the synset `164` (`"Actifed"`) has two hypernyms: `21012` (`"antihistamine"`) and `56099` (`"nasal_decongestant"`), representing that Actifed is both an antihistamine and a nasal decongestant. The synsets are obtained from the corresponding lines in the file `sysnsets.txt`.

Project 6 – WordNet

WordNet Data Type

Implement an immutable data type WordNet with the following API:

```
public class WordNet {

   // constructor takes the name of the two input files
   public WordNet(String synsets, String hypernyms)

   // returns all WordNet nouns
   public Iterable<String> nouns()

   // is the word a WordNet noun?
   public boolean isNoun(String word)

   // distance between nounA and nounB (defined below)
   public int distance(String nounA, String nounB)

   // a synset (second field of synsets.txt) that is the common ancestor of nounA and nounB
   // in a shortest ancestral path (defined below)
   public String sap(String nounA, String nounB)

   // do unit testing of this class
   public static void main(String[] args)
}
```

Corner Cases:

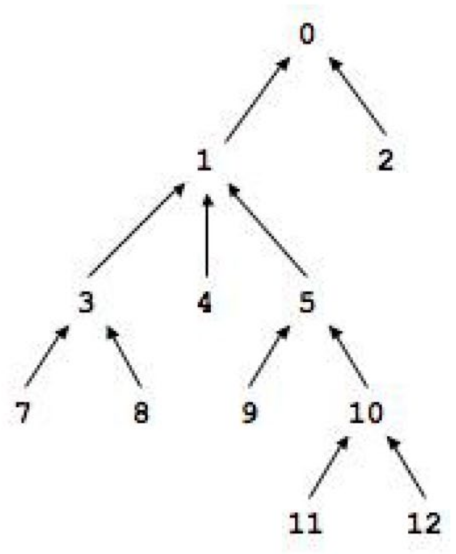
- All methods and the constructor should throw a `NullPointerException` if any argument is null.
- The constructor should throw an `IllegalArgumentException` if the input does not correspond to a rooted DAG.
- The `distance` and `sap` methods should throw an `IllegalArgumentException` unless both noun arguments are WordNet nouns.

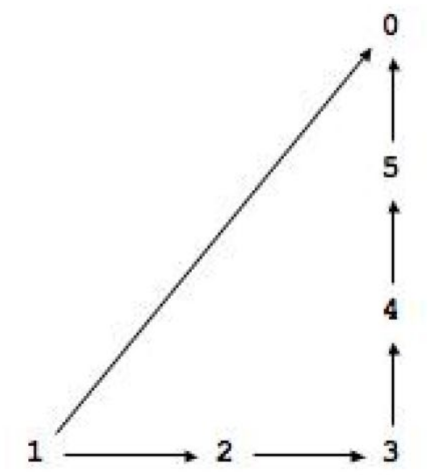
Performance Requirements:

- Your data type should use O(N) space where N is the input size (size of synsets and hypernyms files).
- The constructor should run in O(N) or better time where N is the input size (size of synsets and hypernyms files).
- The method `isNoun` should run in O(lg N) or better time where N is the number of nouns.
- The methods `distance` and `sap` should run in O(N) where N is the size of the WordNet digraph.
- For the analysis, assume that the number of nouns per synset is bounded by a constant.

Shortest Ancestral Path

An ancestral path between two vertices v and w in a digraph is a directed path from v to a common ancestor x, together with a directed path from w to the same ancestor x. A shortest ancestral path is an ancestral path of minimum total length. For example, in the digraph at left, the shortest ancestral path between 3 and 11 has length 4 (with common ancestor 1). In the digraph at right, one ancestral path between 1 and 5 has length 4 (with common ancestor 5), but the shortest ancestral path has length 2 (with common ancestor 0).





Project 6 – WordNet

SAP Data Type

Implement an immutable data type SAP with the following API:

```
public class SAP {

   // constructor takes a digraph (not necessarily a DAG)
   public SAP(Digraph G)

   // length of shortest ancestral path between v and w; -1 if no such path
   public int length(int v, int w)

   // a common ancestor of v and w that participates in a shortest ancestral path; -1 if no such path
   public int ancestor(int v, int w)

    // length of shortest ancestral path between any vertex in v and any vertex in w; -1 if no such path
    public int length(Iterable<Integer> v, Iterable<Integer> w)

    // a common ancestor that participates in shortest ancestral path; -1 if no such path
    public int ancestor(Iterable<Integer> v, Iterable<Integer> w)

    // do unit testing of this class
    public static void main(String[] args)
 }
```

Corner Cases:

- All methods should throw a `NullPointerException` if any argument is null.
- All methods should throw an `IndexOutOfBoundsException` if any argument vertex is invalid – not between 0 and V -1 where V is the number of vertices in the Graph G.

Performance Requirements:

- All methods (and the constructor) should take O(E + V) time where E and V are the number of edges and vertices in the digraph, respectively.
- Your data type should use space proportional to E + V.

Project 6 – WordNet

Measuring the Semantic Relatedness of Two Nouns

Semantic relatedness refers to the degree to which two concepts are related. Measuring semantic relatedness is a challenging problem. For example, most of us agree that *George Bush* and *John Kennedy* (two U.S. presidents) are more related than are *George Bush* and *chimpanzee* (two primates). It might not be clear whether *George W. Bush* and *Eric Arthur Blair* are more related than two arbitrary people. But if one is aware that *George Bush* and *Eric Arthur Blair* (aka George Orwell) are both communicators, then it becomes clear that the two concepts might be related.

We define the semantic relatedness of two wordnet nouns *A* and *B* as follows:

- *distance(A, B)* = the minimum length of any ancestral path between any synset *v* of *A* and any synset *w* of *B.*

This is the notion of distance that you will use to implement the `distance` and `sap` methods in the `WordNet` data type.

Outcast Detection

Given a list of wordnet nouns, A_1, A_2, …, A_n, which noun is the least related to the others? To identify *an outcast*, compute the sum of the distances between each noun and every other one:

$$d_i = dist(A_i, A_1) + \ dist(A_i, A_2) + \cdots + \ dist(A_i, A_n)$$

and return a noun A_t for which d_t is the maximum.

Implement an immutable data type `Outcast` with the following API:

```
public class Outcast {
   public Outcast(WordNet wordnet)          // constructor takes a WordNet object
   public String outcast(String[] nouns)    // given an array of WordNet nouns, return an outcast
   public static void main(String[] args)   // provided test client in starter code
```

Assume that the argument to `outcast` contains only valid wordnet nouns (and that it contains at least two such nouns).

Algorithms Library File

When you import your starter code from GitHub you will notice that it has a reference library called algs4.jar. This library contains many useful classes. You can look at the code for each of these classes in Eclipse by dropping down Referenced Libraries -> algs4.jar -> (default package) and double clicking on any of the .class files. There are four classes in this library that may be of particular use to this project.

```
public class Digraph {
      public Digraph(int V)                //empty Digraph with V Vertices
      public Digraph(In in)                //Digraph from an input stream
      public Digraph(Digraph G)            //create a copy of G
      public int V()                       //number of vertices
      public int E()                       //number of edges
      public void addEdge(int v, int w)    //adds directed edge v->w to the digraph
      public Iterable<Integer> adj(int v) //returns the vertices adjacent to vertex v
      public Digraph reverse()             //returns the reverse of current Digraph
      public String toString()             //String representation of the graph
}

public class BreadthFirstDirectedPath {
      // Computes shortest path from s to every other vertex in the graph
      public BreadthFirstDirectedPath(Digraph G, int s)

      // Computes shortest path from any one of the source vertices in sources to every
      // other vertex in the graph
      public BreadthFirstDirectedPath(Digraph G, Iterable<Integer> sources)

      // Is there a directed path from source to vertex v?
      public boolean hasPathTo(int v)

      // Returns the number of edges in a shortest path from the source to vertex v
      public int distTo(int v)

      // Returns a shortest path from s to v
      public Iterable<Integer> pathTo(int v)
}

public class Topological {
      // Determines whether the digraph G has a topological order and finds it if it does
      public Topological(Digraph G)

      // Determines whether the edge-weighted digraph G has a topological order and finds
      // it if it does
      public Topological(EdgeWeightedDigraph G)

      // Returns a topological order if the digraph has one
      public Iterable<Integer> order()

      // Returns if the digraph has a topological order
      public boolean hasOrder()
}
```

```
public class DirectedDFS {
      // Computes the vertices in digraph G that are reachable from source vertex s
      public DirectedDFS(Digraph G, int s)

      // Computes the vertices in digraph G that are connected to any of the source
      // vertices in sources
      public DirectedDFS(Digraph G, Iterable<Integer> sources)

      // Is there a directed path from the source vertex (or vertices) and vertex v?
      public boolean marked(int v)

      // Returns the number of vertices reachable from the source vertex (or vertices)
      public int count()
}
```

Project Deliverables

- Implement and commit SAP.java for Checkpoint 1
- Implement and commit WordNet.java and Outcast.java for Final checkpoint.
- You may not call any library functions other than those in java.lang, java.util, and algs4.jar

Style Guidelines

You are required to:

- Use conventional indentation and whitespace.
- Avoid long lines over 100 characters in length.
- Give meaningful names to methods and variables in your code.
- Follow Java's naming standards about the format of ClassNames, methodAndVariableNames, and CONSTANT_NAMES.
- Include a blank line followed by meaningful comments at the start of each method other than main, describing its behavior. This may be javadoc or regular block comments.
- Include short, meaningful comments for any "tricky" parts of your methods for the reader. You do not need to comment every line of code – but explain how your logic works where appropriate. Javadoc is not appropriate for these comments.
- Include and complete the following at the top of all your submitted .java files (except the ones marked as "DO NOT MODIFY"):

```
/**
 * Name:
 * Mrs. Kankelborg
 * Period
 * Project 6 WordNet
 * Revision History:
 *
 * Class Description: (in your own words)
 */
```

Optional Optimizations

There are a few things you can do to speed up a sequences of SAP computations on the same digraph. Do not attempt to do this or any of your own invention without thoroughly testing your code. These are meant as optional extensions after you have committed and submitted a solution that passes all of the test cases on TeslaStemCS.org. There are no points associated with these, just an exercise for the curious and the interested.

- The bottleneck operation is re-initializing arrays of length V to perform the BFS computations. This must be done once for the first BFS computation, but if you keep track of which array entries change, you can reuse the same array from computation to computation (re-initializing only those entries that changed in the previous computation). This leads to a dramatic savings when only a small number of entries change (and this is the typical case for the wordnet digraph). Note that if you have any other loops that iterates through all of the vertices, then you must eliminate those to achieve a sublinear running time. (An alternative is to replace the arrays with symbol tables, where, in constant time, the constructor initializes the value associated with every key to be null.)
- If you run the two breadth-first searches from v and w in lockstep (alternating back and forth between exploring vertices in each of the two searches), then you can terminate the BFS from v (or w) as soon as the distance exceeds the length of the best ancestral path found so far.
- Implement a software cache of recently computed `length()` and `ancestor()` queries.

Special thanks to Alina Ene and Kevin Wayne who created this assignment. Copyright © 2006