T-301-REIR, REIKNIRIT
HAUST 2018

S4 - WORDNET

Assignment grading.

- Correctness of `SAP.java` (Mooshak score): 35%
- Correctness of `WordNet.java` (Mooshak score): 25%
- Correctness of the `Outcast.java` (Mooshak score): 20%
- Report: 20%
- Bonus (performance improvement): 5%

Handin Submit `WordNet.java`, `SAP.java` and `Outcast.java` on Mooshak. Submit report in Canvas.

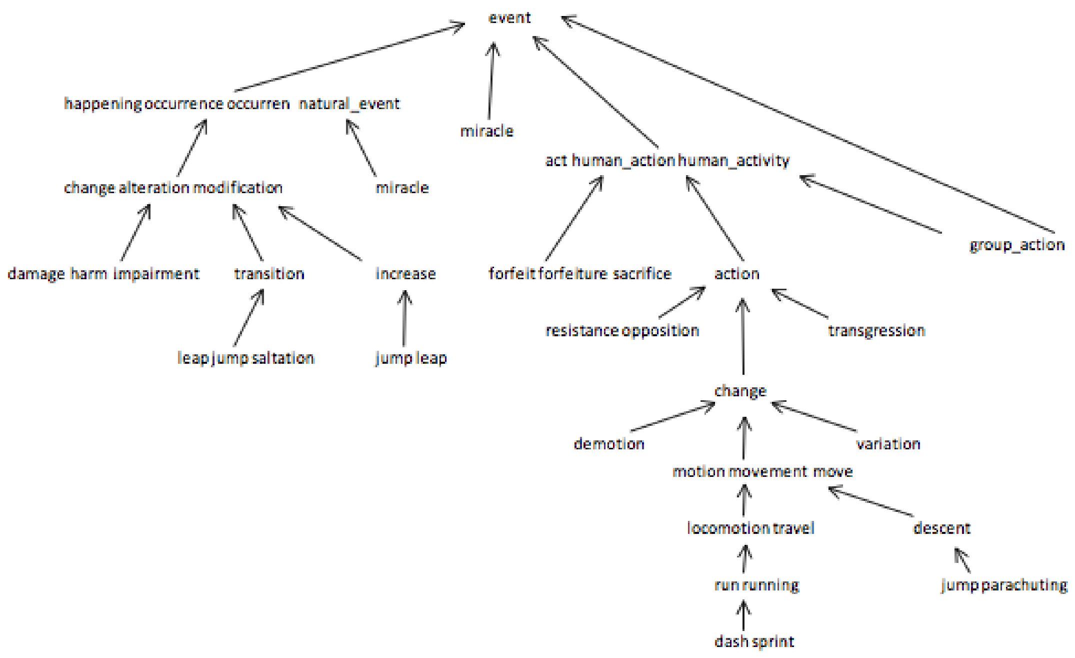
Note You can submit 12 times to each of the four problems on Mooshak. (When working in pairs, this counts the submissions of both parties.) Each additional submission reduces the grade by 0.05.

Introduction

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* („undirheiti“, more specific synset) to a *hypernym* („yfirheiti“, 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 is to build the wordnet digraph: each vertex v is an integer that represents a synset, and each directed edge $v \to w$ represents that w is a hypernym of v. The wordnet digraph is a *rooted DAG*: it is acyclic and has one vertex 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.

Date: 9. október.



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 synset id; subsequent fields are the id numbers of the synset's hypernyms. For example, the following line

```
164,21012,56099
```

  means that the 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 `synsets.txt`.

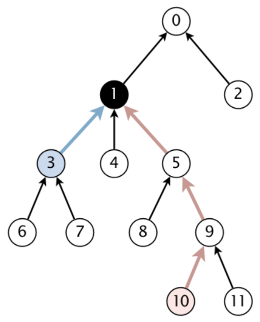
```
164,Actifed,trade name for a drug containing an antihistamine and a
   decongestant...
21012,antihistamine,a medicine used to treat allergies...
```

```
56099,nasal_decongestant,a decongestant that provides temporary relief
    of nasal...
```

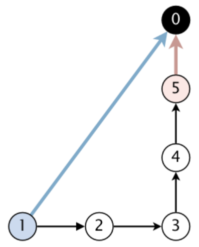
Shortest ancestral path

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. We refer to the common ancestor in a shortest ancestral path as a *shortest common ancestor*. Note that a shortest common ancestor always exists because the root is an ancestor of every vertex

For example, in the digraph at left (`digraph1.txt`), the shortest ancestral path between `3` and `10` has length `4` (with common ancestor `1`). In the digraph at right (`digraph2.txt`), 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`).

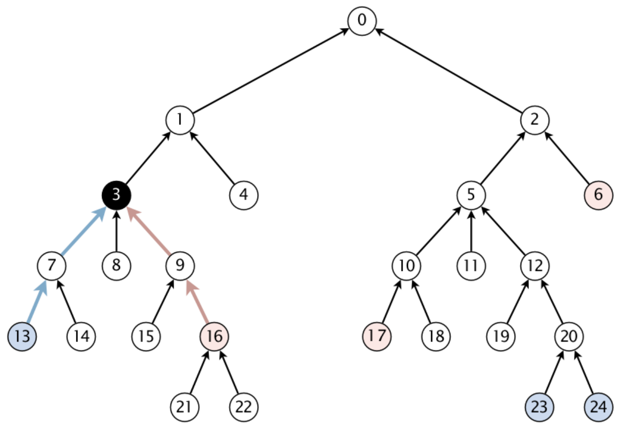


v = 3, w = 10
shortest ancestral path: 3–1–5–9–10
associated length: 4
shortest common ancestor: 1



v = 1, w = 5
ancestral path: 1–2–3–4–5
shortest ancestral path: 1–0–5
associated length: 2
shortest common ancestor: 0

We generalize the notion of shortest common ancestor to subsets of vertices. A shortest ancestral path of two subsets of vertices A and B is a shortest ancestral path over all pairs of vertices v and w, with v in A and w in B.



A = { 13, 23, 24 }, B = { 6, 16, 17 }
ancestral path: 13–7–3–1–0–2–6
ancestral path: 23–20–12–5–10–17
ancestral path: 23–20–12–5–2–6

shortest ancestral path: 13–7–3–9–16
associated length: 4
shortest common ancestor: 3

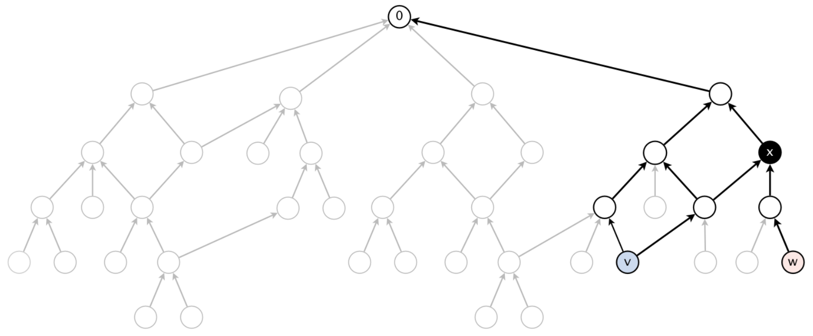
SAP Data type Implement an immutable data type SAP with the following API:

```
// 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 shortest common common ancestor of v and w; -1 if no such path
public int ancestor(int v, int w)

// length of shortest ancestral path of vertex subsets A and B; -1 if no such path
public int length(Iterable<Integer> A, Iterable<Integer> B)

// a shortest common ancestor of vertex subsets A and B; -1 if no such path
public int ancestor(Iterable<Integer> A, Iterable<Integer> B)

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

Your implementation must verify that the given digraph is actually a rooted DAG. All methods should throw a `java.lang.IndexOutOfBoundsException` if one (or more) of the input arguments is not between `0` and `G.V()-1`. You may assume that the iterable arguments contain at least one integer. The constructor should throw a `java.lang.IllegalArgumentException` if the digraph is not a rooted DAG.

Performance requirements All methods (and the constructor) should take time at most proportional to $E + V$ in the worst case, where E and V are the number of edges and vertices in the digraph, respectively. Your data type should use space proportional to $E + V$.

For extra credit, in addition, the methods length() and ancestor() should take time proportional to the number of vertices and edges reachable from the argument vertices (or better), For example, to compute the shortest common ancestor of v and w in the digraph below, your algorithm can examine only the highlighted vertices and edges and it cannot initialize any vertex-indexed arrays.



Test client A natural `main()` test client reads a digraph input file, construct the digraph, and use that digraph to test that all SAP methods work as expected. The following test client reads in vertex pairs from standard input, and prints out the length of the shortest ancestral path between the two vertices along with a shortest common ancestor: On the left below you can see the contents of `digraph1.txt` and on the right are a few samples of expected values if your program reads in that input file:

```
12                          v = 3, w = 10
11                          length = 4, ancestor = 1
```

```
 6  3
 7  3                        v = 8, w = 11
 3  1                        length = 3, ancestor = 5
 4  1
 5  1                        v = 6, w = 2
 8  5                        length = 4, ancestor = 0
 9  5
10  9                        v = {3, 8, 6, 1}, w = {10, 2}
11  9                        length = 2, ancestor = 0
 1  0
 2  0
```

Obs: *[Updated 9 Oct 2017]* Mooshak will not test `main()` functions. Its contents is therefore immaterial to correctness, and it will not be graded.

WordNet data type

Implement an immutable data type WordNet with the following API:

```
// 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 a shortest common ancestor
// of nounA and nounB
public String sap(String nounA, String nounB)

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

The constructor should throw a `java.lang.IllegalArgumentException` if the input does not correspond to a rooted DAG. The `distance()` and `sap()` methods should throw a `java.lang.IllegalArgumentException` unless both of the noun arguments are WordNet nouns.

Your data type should use space linear in the input size (size of synsets and hypernyms files). The constructor should take time linearithmic (or better) in the input size. The method `isNoun()` should run in time logarithmic (or better) in the number of nouns. The methods `distance()` and `sap()` should run in time linear in the size of the WordNet digraph.

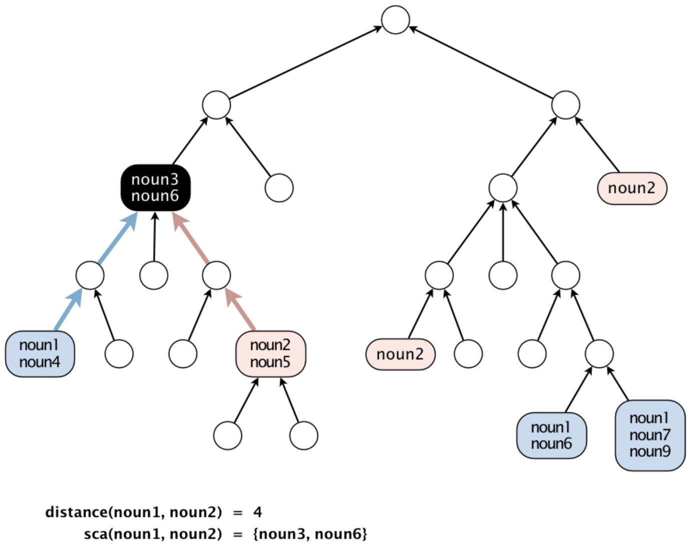
Measuring semantic relatedness of two nouns

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). However, not most of us agree that *George Bush* and *Eric Arthur Blair* are related concepts. 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 x and y as follows:

- A = set of synsets in which x appears
- B = set of synsets in which y appears
- $distance(x, y)$ = length of shortest ancestral path of subsets A and B
- $sap(x, y)$ = a shortest common ancestor of subsets A and 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 $x_1, x_2, ..., x_n$, 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:

- $d_i = distance(x_i, x_1) + distance(x_i, x_2) + \ldots + distance(x_i, x_n)$

and return a noun x_t for which d_t is maximum. Note that because $distance(x_i, x_i) = 0$, it will not contribute to the sum.

Implement an immutable data type Outcast with the following API:

```
// constructor takes a WordNet object
public Outcast(WordNet wordnet)

// given an array of WordNet nouns, return an outcast
public String outcast(String[] nouns)
```

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

The following test client takes from the command line the name of a synset file, the name of a hypernym file, followed by the names of outcast files, and prints out an outcast in each file:

```java
public static void main(String[] args) {
    WordNet wordnet = new WordNet(args[0], args[1]);
    Outcast outcast = new Outcast(wordnet);
    for (int t = 2; t < args.length; t++) {
        String[] nouns = In.readStrings(args[t]);
        StdOut.println(args[t] + ": " + outcast.outcast(nouns));
    }
}
```

Here is a sample execution:

```
% more outcast5.txt
horse zebra cat bear table

% more outcast8.txt
water soda bed orange_juice milk apple_juice tea coffee

% more outcast11.txt
apple pear peach banana lime lemon blueberry strawberry mango watermelon potato


% java Outcast synsets.txt hypernyms.txt outcast5.txt outcast8.txt outcast11.txt
outcast5.txt: table
outcast8.txt: bed
outcast11.txt: potato
```

School of Computer Science, Reykjavík University, Menntavegi 1, 101 Reykjavík

E-mail address: mmh@ru.is