Discussion 12: Project 6

2 Exercises and 3 Problems

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

- $\rightsquigarrow$ The *degree distribution* of G is a function mapping each degree value in G to the number of vertices with that value.
- $\rightsquigarrow$ The *average degree* of G is $\frac{2E}{V}$.
- $\rightsquigarrow$ The *average path length* of G is the average length of all the paths in G.
- $\rightsquigarrow$ 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 $\frac{1}{V}\sum_i^V C_i$.

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
```

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`

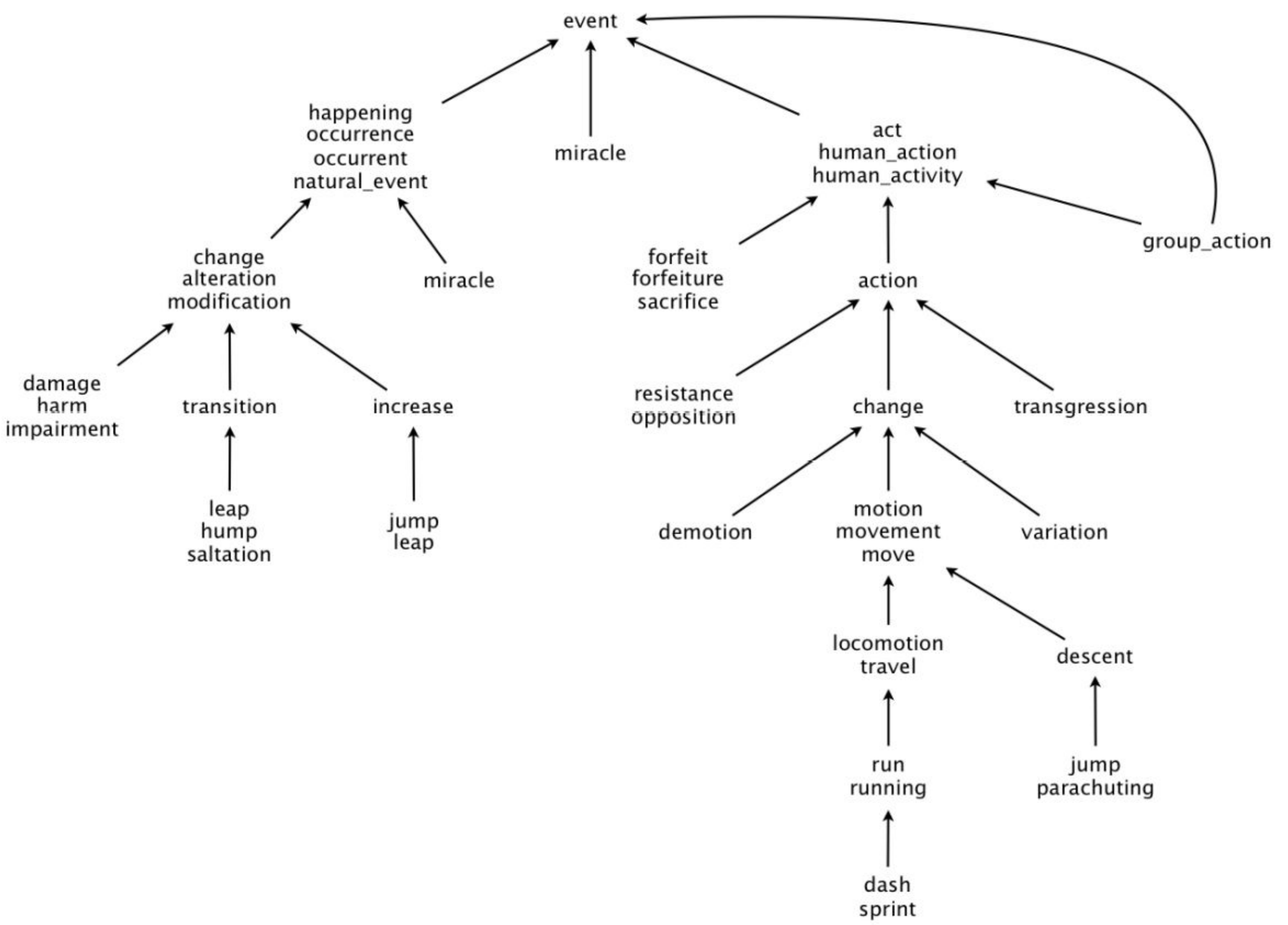
`DiGraphProperties(DiGraph G)`	computes graph properties for the digraph `G`
`boolean isDAG()`	returns `true` if the digraph is a DAG, and `false` otherwise
`boolean isMap()`	returns `true` if the digraph is a map, and `false` otherwise
`Iterable<Integer> sources()`	returns all the sources in the digraph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

WordNet groups words into sets of synonyms called synsets. For example, {*AND circuit, AND gate*} is a synset that represents a logical gate that fires only when all of its inputs fire. WordNet also describes semantic relationships between synsets. One such relationship is the *is-a* relationship, which connects a *hyponym* (more specific synset) to a *hypernym* (more general synset). For example, the synset {*gate, logic gate*} is a hypernym of {*AND circuit, AND gate*} because an AND gate is a kind of logic gate.

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 $v \to w$ denotes 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 shown below.


event
happening
occurrence
occurrent
natural_event
miracle
act
human_action
human_activity
change
alteration
modification
miracle
forfeit
forfeiture
sacrifice
action
group_action
damage
harm
impairment
transition
increase
resistance
opposition
change
transgression
leap
hump
saltation
jump
leap
demotion
motion
movement
move
variation
locomotion
travel
descent
run
running
jump
parachuting
dash
sprint

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 *comma-separated values* (CSV) format: each line contains a sequence of fields, separated by commas.

- *List of synsets.* The file `synsets.txt` contains all noun synsets in WordNet, one per line. Line i of the file (counting from 0) contains the information for synset i. The first field is the *synset id*, which is always the integer i; the second field is the synonym set (or synset); and the third field is its dictionary definition (or *gloss*), which is not relevant to this assignment.

```
% more synsets.txt                     synset
 ⋮

34,AIDS acquired_immune_deficiency_syndrome,a serious (often fatal) disease of the immune system
35,ALGOL,a programming language used to express computer programs as algorithms
36,AND_circuit AND_gate,a circuit in a computer that fires only when all of its inputs fire
37,APC,a drug combination found in some over-the-counter headache remedies
38,ASCII_character,any member of the standard code for representing characters by binary numbers
39,ASCII_character_set,(computer science) 128 characters that make up the ASCII coding scheme
40,ASCII_text_file,a text file that contains only ASCII characters without special formatting
41,ASL American_sign_language,the sign language used in the United States
42,AWOL,one who is away or absent without leave <- gloss
```
id

For example, line 36 implies 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 constitute a synset are separated by spaces. If a noun contains more than one word, the words are connected by the underscore character.

List of hypernyms. The file `hypernyms.txt` contains the hypernym relationships. Line i of the file contains the hypernyms of synset i. The first field is the synset id, which is always the integer i; subsequent fields are the id numbers of the synset's hypernyms.

```
% more hypernyms.txt
  ⋮
34,47569,48084
35,19983
36,42338
37,53717
38,28591
39,28597
40,76057
41,70206
42,18793
  ⋮
```

hypernyms

id

For example, line 36 implies that synset 36 (AND_circuit AND_Gate) has 42338 (gate logic_gate) as it only hypernym. Line 34 implies that synset 34 (AIDS acquired_immune_deficiency_syndrome) has two hypernyms: 47569 (immunodeficiency) and 48084 (infectious_disease).

Problem 1

Problem 1. (`WordNet` *Data Type*) Implement an immutable data type called `WordNet` with the following API:

WordNet	
`WordNet(String synsets, String hypernyms)`	constructs a `WordNet` object given the names of the input (synset and hypernym) files
`Iterable<String> nouns()`	returns all WordNet nouns
`boolean isNoun(String word)`	returns `true` if the given word is a WordNet noun, and `false` otherwise
`String sca(String noun1, String noun2)`	returns a synset that is a shortest common ancestor of `noun1` and `noun2`
`int distance(String noun1, String noun2)`	returns the length of the shortest ancestral path between `noun1` and `noun2`

Two Separate Chaining Hashes
One whose key is a noun and holds a set of IDs who share the same key
Another whose key is an Integer noun ID, and value is a noun

ShortestCommonAncestor sca ----------------------------------> Data Type Created in Problem 2

Instance Variables + Constructor

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

  Split into tokens?

- Initialize `sca` using `G`

```
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)`

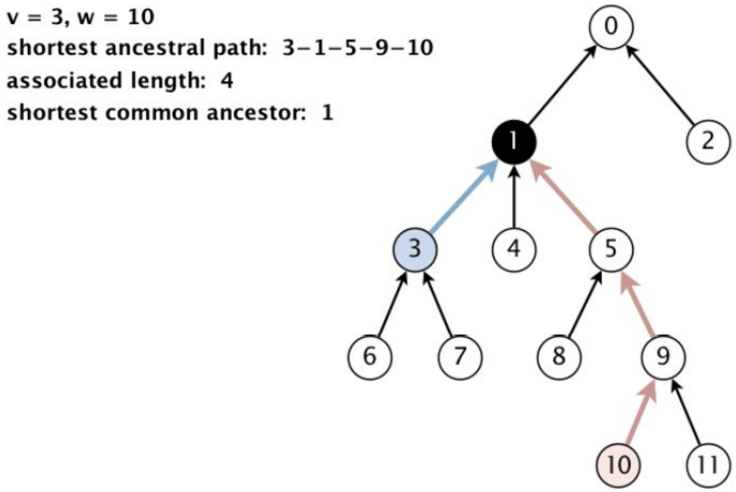
⇝ Use `sca` to compute and return the length of the shortest ancestral path between the given nouns

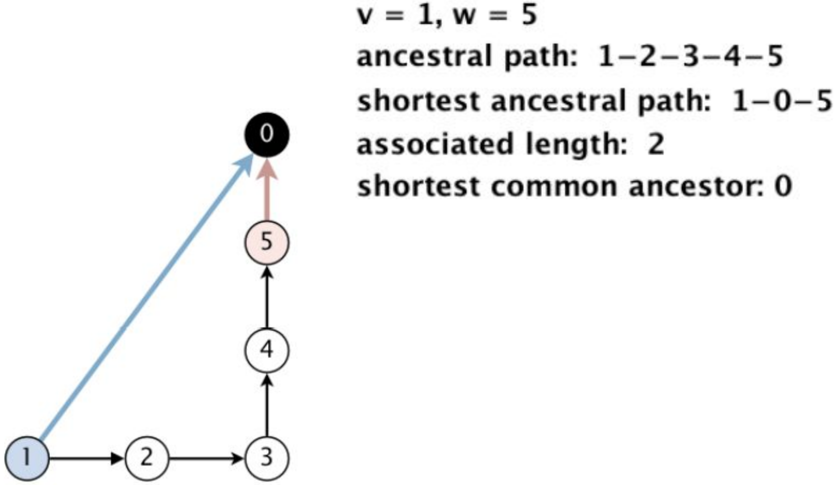
Both of above methods require Problem 2 to be complete to work

```
>_ ~/workspace/project6

$ java WordNet data/synsets.txt data/hypernyms.txt worm bird
# of nouns = 119188
isNoun(worm)? true
isNoun(bird)? true
isNoun(worm bird)? false
sca(worm, bird) = animal animate_being beast brute creature fauna
distance(worm, bird) = 5
```

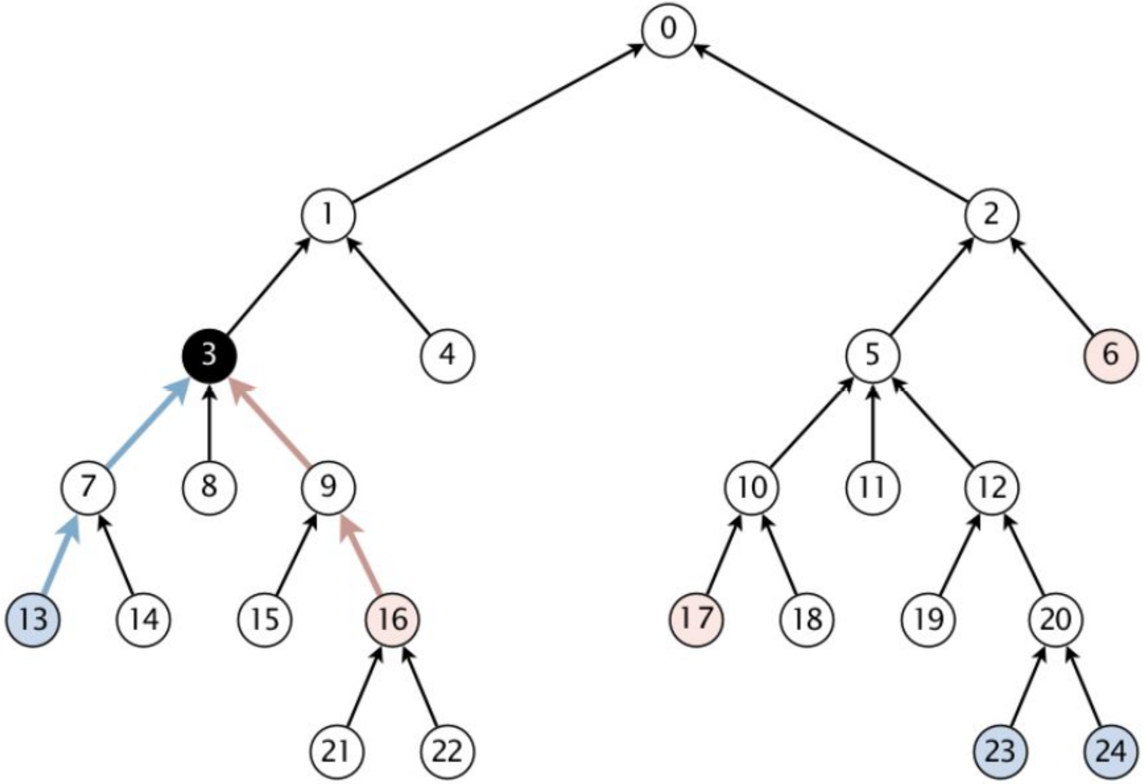
Problem 2

Shortest Common Ancestor An *ancestral path* between two vertices v and w in a rooted DAG 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. Note also that an ancestral path is a path, but not a directed path.





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

Problem 2 API

Problem 2. (*ShortestCommonAncestor Data Type*) Implement an immutable data type called `ShortestCommonAncestor` with the following API:

ShortestCommonAncestor	
`ShortestCommonAncestor(Digraph G)`	constructs a `ShortestCommonAncestor` object given a rooted DAG
`int length(int v, int w)`	returns length of the shortest ancestral path between vertices `v` and `w`
`int ancestor(int v, int w)`	returns a shortest common ancestor of vertices `v` and `w`
`int length(Iterable<Integer> A, Iterable<Integer> B)`	returns length of the shortest ancestral path of vertex subsets `A` and `B`
`int ancestor(Iterable<Integer> A, Iterable<Integer> B)`	returns a shortest common ancestor of vertex subsets `A` and `B`

Instance variable

$\rightsquigarrow$ A rooted DAG, `DiGraph G`

`ShortestCommonAncestor(DiGraph G)`

$\rightsquigarrow$ 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]`

```
private int[] triad(Iterable<Integer> A, Iterable<Integer> B)
```

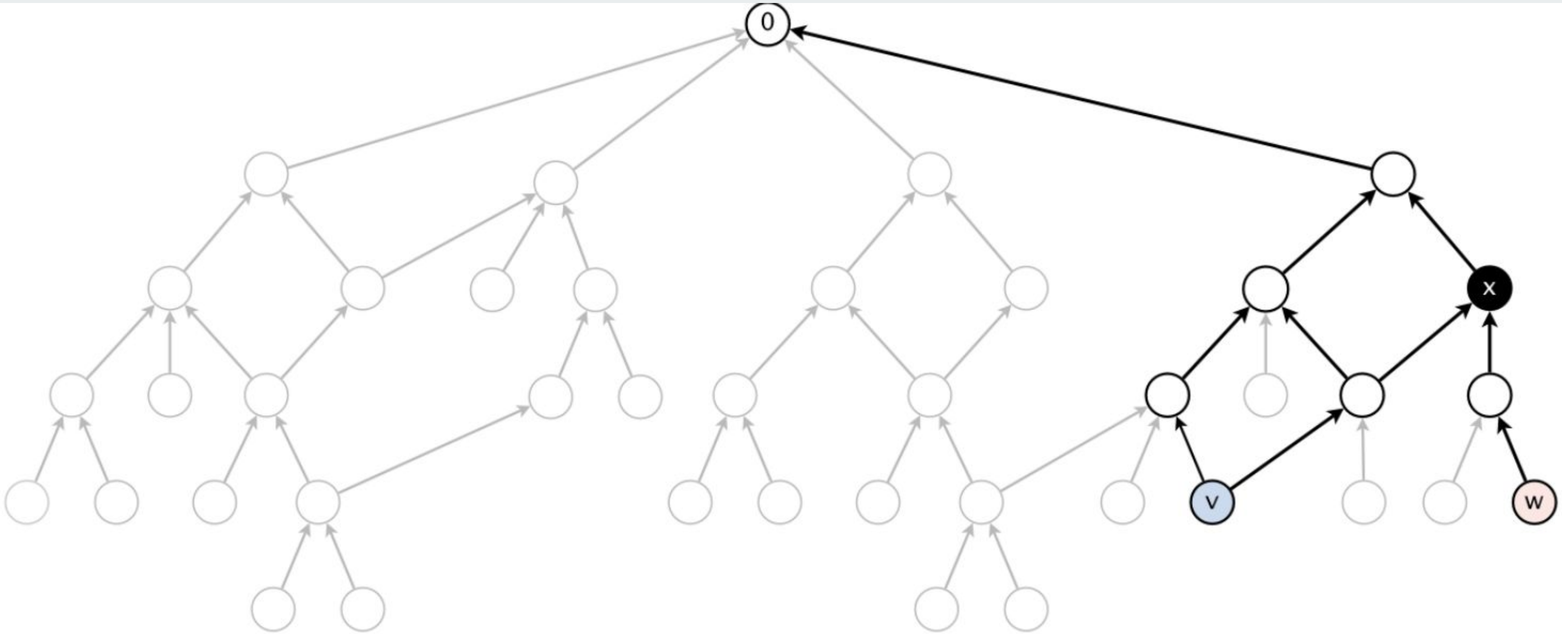
$\leadsto$ Return a 3-element array consisting of a shortest common ancestor a of vertex subsets A and B, a vertex v from A, and a vertex w from B such that the path v-a-w is the shortest ancestral path of A and B; use length(int v, int w) and ancestor(int v, int w) methods to implement this method

```
int length(Iterable<Integer> A, Iterable<Integer> B)
```

$\leadsto$ 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)
```

$\leadsto$ Return a shortest common ancestor of vertex subsets A and B; use triad((Iterable<Integer> A, Iterable<Integer> B) to implement this method



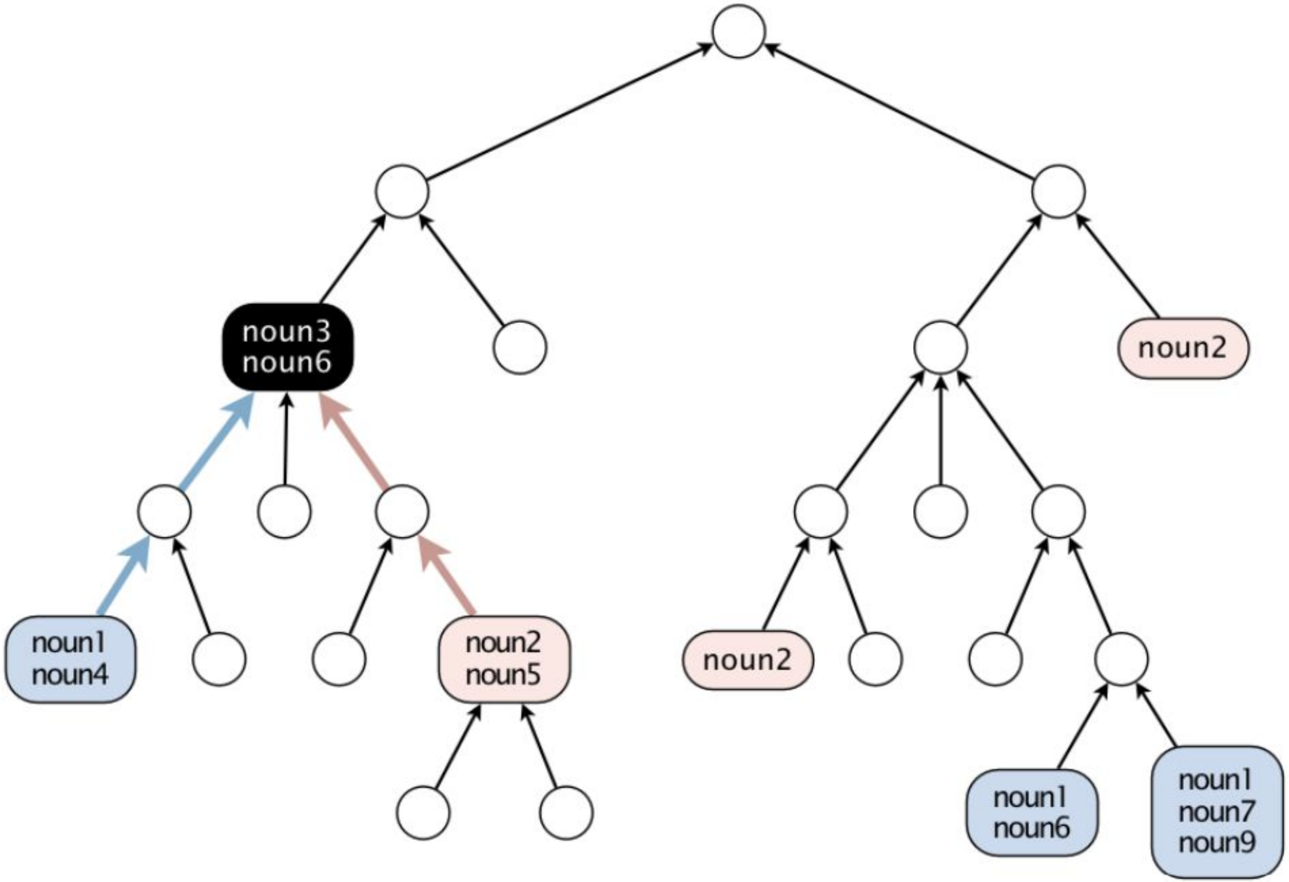

```
>_ ˜/workspace/project6

$ java ShortestCommonAncestor data/digraph1.txt
3 10 8 11 6 2
<ctrl-d>
length = 4, ancestor = 1
length = 3, ancestor = 5
length = 4, ancestor = 0
```

Problem 3

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, you consider *George W. Bush* and *John F. Kennedy* (two U.S. presidents) to be more closely related than *George W. 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. However, both *George W. Bush* and *Eric Arthur Blair* (aka George Orwell) are famous communicators and, therefore, closely related. We define the semantic relatedness of two WordNet nouns x and y as follows:

- A is set of synsets in which x appears;
- B is set of synsets in which y appears;
- $sca(x, y)$ a shortest common ancestor of A and B; and
- $distance(x, y)$ is length of shortest ancestral path of A and B.

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



distance(noun1, noun2) = 4

sca(noun1, noun2) = {noun3, noun6}

Outcast Detection Given a list of WordNet nouns $x_1, x_2, \ldots, x_n$, 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:

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

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

Problem 3. (`Outcast` *Data Type*) Implement an immutable data type called `Outcast` with the following API:

Outcast	
`Outcast(WordNet wordnet)`	constructs an `Outcast` object given the WordNet semantic lexicon
`String outcast(String[] nouns)`	returns the outcast noun from `nouns`

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

You may assume that argument to `outcast()` contains only valid WordNet nouns (and that it contains at least two such nouns).

```
>_ ~/workspace/project6

$ java Outcast data/synsets.txt data/hypernyms.txt < data/outcast10.txt
cat cheetah dog wolf *albatross* horse zebra lemur orangutan chimpanzee
$ java Outcast data/synsets.txt data/hypernyms.txt < data/outcast11.txt
apple pear peach banana lime lemon blueberry strawberry mango watermelon *potato*
$ java Outcast data/synsets.txt data/hypernyms.txt < data/outcast12.txt
competition cup event fielding football level practice prestige team tournament world *mongoose*
```

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
```

Files to submit:

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