CIS 121—Data Structures and Algorithms with Java—Spring 2018

Due: April 25th, 2018 on Gradescope

6 Required Problems (165 points), Summary and Analysis(30 points), Milestones (30 points), Style and Tests (25 points)

DO NOT modify methods or method headers that were already provided to you.

DO feel free to create additional classes, files, or public methods as necessary.

1 Outline

Important: Before asking any questions on Piazza, be sure to check and keep updated with the pinned clarifications post, where important clarifications will be posted.

In this project, you will parse Wikipedia link data and implement several graph algorithms, including finding connected components, measures of popularity, and other interesting applications. This will culminate in your implementation of 'The Wikipedia Game'.

We will be releasing all files necessary for the project all at once, however, it will be due in parts. We will have three milestone dates, as well a final submission date.

You have over two months, so definitely spread the work out. This also gives you time to discuss your design decisions with your style grading TA and possibly make any changes.

1.1 Milestone 1: Wednesday, March 14 11:59 PM

For this milestone, you will implement the parsing of page link data from an XML file. This milestone consists of section 4. Please look at **Section 1.7** for submission instructions. Required classes for submission:

- `WikiXmlDumpParserImpl`
  implements `edu.upenn.cis121.project.WikiXmlDumpParser`
- Any other classes that you might have written
- Test classes for all classes that you have written

1.2 Milestone 2: Wednesday, April 4 11:59 PM

For this milestone, you will implement a heap and Dijkstra's algorithm. This milestone consists of section 5.

Please look at **Section 1.7** for submission instructions.

- `BinaryMinHeapImpl`
  implements `edu.upenn.cis121.project.data.BinaryMinHeap`
- `ShortestPathImpl`
  implements `edu.upenn.cis121.project.graph.ShortestPath`
- Any other classes that you might have written
- Test classes for all classes that you have written

1.3 Milestone 3: Wednesday, April 18 11:59 PM

For this milestone, you will implement the canonical graph algorithms for connected components as well as a basic inverted index data structure. This milestone consists of section 6 and section 7. Please look at **Section 1.7** for submission instructions.

Required classes for submission:

- `WikiXmlDumpParserImpl`
  implements `edu.upenn.cis121.project.WikiXmlDumpParser`
- `ConnectedComponentsImpl`
  implements `edu.upenn.cis121.project.ConnectedComponents`
- `RecordInvertedIndexImpl`
  implements `edu.upenn.cis121.project.index.RecordInvertedIndex`
- Any other classes that you might have written
- Test classes for all classes that you have written

1.4 Final Submission: Wednesday, April 25th 11:59PM

For this milestone and final deadline, you will complete your project by implementing the Wikipedia Game, integrating all your previous work on this project. Please look at **Section 1.7** for submission instructions.

You will submit the following files:

- `WikiXmlDumpParserImpl`
  implements `edu.upenn.cis121.project.WikiXmlDumpParser`
- `BinaryMinHeapImpl`
  implements `edu.upenn.cis121.project.data.BinaryMinHeap`
- `ShortestPathImpl`
  implements `edu.upenn.cis121.project.graph.ShortestPath`
- `RecordInvertedIndexImpl`
  implements `edu.upenn.cis121.project.index.RecordInvertedIndex`
- `ConnectedComponentsImpl`
  implements `edu.upenn.cis121.project.ConnectedComponents`
- `WikipediaGameImpl`
  implements `edu.upenn.cis121.project.engine.WikipediaGame`
- `WikipediaGameFactoryImpl`
  implements `edu.upenn.cis121.project.engine.WikipediaGameFactory`
- `BasicTfIdfSearchEngine`
  implements `edu.upenn.cis121.project.engine.SearchEngine`
- `BasicTfIdfSearchEngineFactory`
  implements `edu.upenn.cis121.project.engine.SearchEngineFactory`
- Any other classes that you might have written
- Test classes for all classes that you have written

1.5 Milestone Grading

In order to encourage you to complete the milestones on time, they will be worth 10 points each, for a total of 30 out of the 250 project points. Earning these 10 points means that you completed and submitted the milestone by its due date.

Because the project has multiple parts built on each other, all milestone tests will be available after their respective due date so you will have an opportunity to test your code again and fix any issues. However, you will not be able to earn previous milestone points after the deadline.

There will be no late days for milestones.

1.6 Late Policy

As stated in the late policy of the course syllabus, this final project will have a late policy different from the other homework assignments. As such, *no late days can be used* for this project. In addition, will allow you to submit your project up to two days later than the extended due date according to the following penalty schedule:

- For one late day—submission before **April 26th**—you will be deducted 50 points.
- For two late days—submission before **April 27th**—you will be deducted 100 points.
- ***No submissions will be accepted after midnight on April 27th***

1.7 Submission

For all of your source files **DO NOT** place any in a package, and ensure that no file has a package declaration e.g. `package src;`. When you submit, be sure that all files are located at the root level i.e. do not submit a folder that contains the source files. The easiest way to do this is to drag the files individually onto Gradescope.

Important: Be sure that your own test cases pass and do not stall. Otherwise, your submission will not be processed correctly.

You have unlimited submissions for this homework. Upon submission, our infrastructure will compile your code and run some basic test cases against the interface. These are here to test that your code compiles (i.e. you didn't forget to add a class etc.) when we run real tests. They should not be used to test your own code's correctness since the functionality they test is very very basic.

2 Setup

We have provided you with a jar file (named `project-sources.jar`)with all the interfaces as well as utilities needed for the project. JAR stands for **J**ava **AR**chive, so you can simply think of `project-sources.jar` as a compressed file that contains all the classes and interfaces you might need for the project. Much like how adding the JUnit4 *library* (which is really just a JAR file) to your project allows you to use annotations such as `@Test` and methods such as `assertEquals`, adding `project-sources.jar` to your project allows you to use methods, classes, and interfaces that were packaged into `project-sources.jar`.

Here, we will detail the easiest way to create the project in your Eclipse workspace.

Setting up the Project:

1. In Eclipse, go to File →New →Java Project. Input your desired project folder name into the "Project name" field and hit Finish.
2. Now, we want to add the jar to our project. Download it from the homework page under "extra files", and add it to the root of your project directory (not in the src folder). In Eclipse, right-click the project folder and select Build Path →Configure Build Path →Add JARs... and select the jar from the dropdown (click on the arrow to expand your project folder if necessary).
3. Add JUnit 4 to your project by repeating the previous step, except choosing Add Library instead of Add JARs.

Now that the project is set up, let us set up the Javadocs.

Setting up the Javadocs:

1. Ensure that you're connected to Wi-Fi. If you're offline, you will not be able to view the Javadocs.
2. Expand Referenced Libraries →Right click "project-sources.jar" →Properties

3. Left sidebar, click "Javadoc Location"

4. Type the following link: "https://webdav.seas.upenn.edu/∼cis121/webdav/maven2/site/project/stub/apidocs/" in the "Javadoc Location" text-box. Please type and **do not** copy it. Formatting can be messed up if you copy it.

5. Click Validate and Apply.

6. You're done! Please come to office hours if any of the above steps don't work.

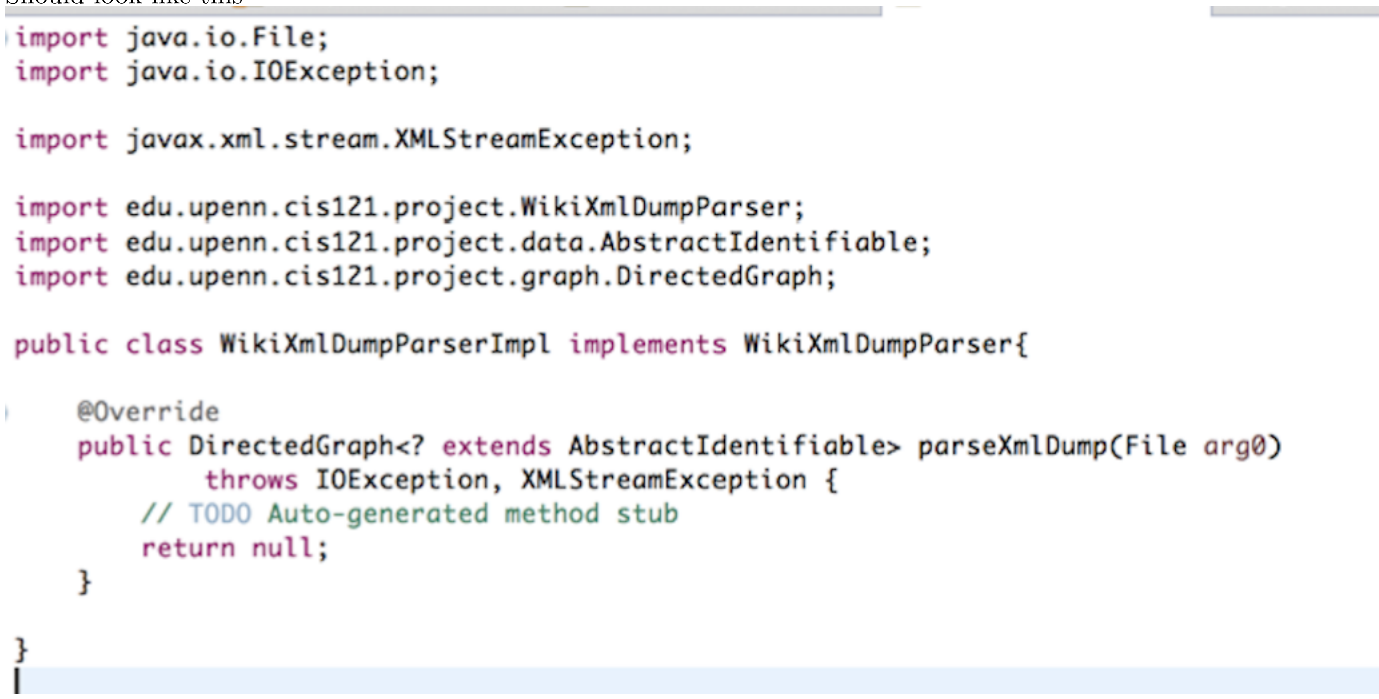
Checking if Everything Works:

1. Create a new file called "WikiXmlDumpParserImpl"

2. Make the file implement "WikiXmlDumpParser"

3. If there is a red squiggly line underneath WikiXmlDumpParserImpl, hover over it and click "Add unimplemented methods"

4. Should look like this



```
import java.io.File;
import java.io.IOException;

import javax.xml.stream.XMLStreamException;

import edu.upenn.cis121.project.WikiXmlDumpParser;
import edu.upenn.cis121.project.data.AbstractIdentifiable;
import edu.upenn.cis121.project.graph.DirectedGraph;

public class WikiXmlDumpParserImpl implements WikiXmlDumpParser{

    @Override
    public DirectedGraph<? extends AbstractIdentifiable> parseXmlDump(File arg0)
            throws IOException, XMLStreamException {
        // TODO Auto-generated method stub
        return null;
    }

}
```

5. Hover over WikiXmlDumpParser, it should say "A WikiXmlDumpParser specifies parsing of a wiki XML dump file."

6. If the above is the case, and there's no errors, then you have set up the project correctly! :) Otherwise, **Please come to OH as soon as you can to get it working.**

3 Obtaining the Simple English Wikipedia XML Dump

1. Navigate to `https://dumps.wikimedia.org/backup-index.html`.

2. Look for an link entry called "simplewiki" and click on that link.

3. Download the XML dump file, which has a name of the form `simplewiki-*-pages-meta-current.xml.bz2`. The * will be replaced some number indicating the date that the data was collected; for example, at the time of this writing, the full filename was `simplewiki-20161001-pages-meta-current.xml.bz2`.

4. Extract the downloaded bzip2 archive. On a Linux or Mac, you can use the `bunzip2` utility. The command-line invocation is

   ```
   bunzip2 -dk simplewiki-*-pages-meta-current.xml.bz2
   ```

   For a Windows computer, we recommend downloading and using 7-zip to extract this file.

When you finish the project, you can run it on the full XML dump, but we recommend testing on smaller XML files first!

4 Parsing the Simple English Wikipedia Dump as a Graph (30 points)

Required classes:

- `WikiXmlDumpParserImpl`
  implements `edu.upenn.cis121.project.WikiXmlDumpParser`
- Any other classes that you might have written
- Test classes for all classes that you have written

In this section, you will parse page link data from a Simple English Wikipedia XML dump file.

4.1 Introduction to XML

You will need to read a structured description of a wiki page and outbound link graph, and build a Java object hierarchy to represent that graph. The data provided is in a format called XML (eXtensible Markup Language), which is a widely-used, language-independent standard for encoding data.

Please refer to Figure 1 at the end of this writeup for a complete explanation of XML.

4.2 XML dump format

The dump format (simplified) is as follows, with unneeded XML elements omitted:

```
<mediawiki>
  <siteinfo>
    <sitename>Wikipedia</sitename>
    <dbname>simplewiki</dbname>
    <base>https://simple.wikipedia.org/wiki/Main_Page</base>
    ...
    <case>first-letter</case>
    ...
  </siteinfo>
  <page>
    <title>April</title>
    ...
    <revision>
      ...
      <format>text/x-wiki</format>
      <text>'''April''' is the fourth [[month]] of the [[year]], and comes betwe
en [[March]] and [[May]]. It has 30 [[day]]s. April begins on the same day of we
ek as [[July]] in all years and also [[January]] in leap years.

April's [[flower]]s are the [[Sweet Pea]] and [[Asteraceae|Daisy]]. Its [[births
```

```
tone]] is the [[diamond]]. The meaning of the diamond is innocence.

== The Month ==
[[File:Colorful spring garden.jpg|thumb|180px|right|[[Spring]] flowers in April
in the [[Northern Hemisphere]].]]
...
      </text>
      ...
    </revision>
  </page>
  ...
</mediawiki>
```

The individual page revisions may be in any of the formats `application/json`, `text/css`, `text/javascript`, `text/plain`, `text/x-wiki`, but you only need to handle

- `text/plain`
- `text/x-wiki`

We will use the following simplifying assumption:

- If the format is `text/plain`, then there are no outbound page links in that wiki page. Pages should still link to it, and it should be present as a node.
- Outlinks are only present in MediaWiki markup (`text/x-wiki`).
- Links to pages not contained in the dump (including pages with other formats besides `text/plain` and `text/x-wiki`, such as `text/css`, which are only seen as a link from another page), should be handled as pages with `null` text body and no outbound links.

We provide the class `edu.upenn.cis121.project.util.mediawiki.MediaWikiUtils` to assist you with parsing the MediaWiki markup for links as well as interpolating text.

4.3 `WikiXmlDumpParserImpl.java`

Implement the interface `edu.upenn.cis121.project.WikiXmlDumpParser`. You will need to create the file `WikiXmlDumpParserImpl.java` that, given an XML dump file as described above, produces an instance of a `edu.upenn.cis121.project.graph.DirectedGraph<? extends AbstractIdentifiable>`.

4.3.1 A note on inheritance

Note that the `edu.upenn.cis121.project.data.AbstractIdentifiable` abstract class specifies that an instance has to have a unique identifier. For our Simple English Wikipedia dataset, this is just the page title. Since we assume that subclasses will obey the unique identifier per instance contract, we can optimize the `equals(Object)` and `hashCode()` methods to only check the identifier. You do not have to fill these methods out since they are already present in the abstract class.

Since you need to return instances of a concrete class extending `edu.upenn.cis121.project.data.AbstractIdentifiable`, you will need to call the superclass constructor. The `AbstractIdentifiable` constructor stores a `String` identifier, so you do not need to store this identifier in your concrete class (otherwise you may be docked points).

Note that the interface method we are implementing is

```
DirectedGraph<? extends AbstractIdentifiable> parseXmlDump(File file)
            throws IOException, XMLStreamException;
```

We can actually utilize covariant return typing and actually return a more specific subclass than what is specified in the interface. So this means in your implementation class called `WikiXmlDumpParserImpl`, you can write

```
@Override
DirectedGraph<MyPageClass> parseXmlDump(File file)
      throws IOException, XMLStreamException;
```

or even

```
@Override
MyDirectedGraphImpl<MyPageClass> parseXmlDump(File file)
      throws IOException, XMLStreamException;
```

where `MyDirectedGraphImpl` is a subclass of `DirectedGraph` and `MyPageClass` is a subclass of `AbstractIdentifiable`.

4.3.2 XML parsing

Most of the hard work in parsing XML will come from a basic iterator loop.

```
// Create the factory first
XMLInputFactory xmlif = XMLInputFactory.newInstance();
try (BufferedReader br = Files.newBufferedReader(file.toPath())) {
    // Get a reader for XML from the factory
    XMLEventReader r = xmlif.createXMLEventReader(br);
    while (r.hasNext()) {
        XMLEvent event = r.nextEvent();
        System.out.println(event);
    }
}
```

You'll want to check out the documentation for both `XMLEventReader` and `XMLEvent` for more information.

In order to parse XML, there is a lot of documentation to poke through. You should try to read through and figure out what each class is doing. Learning to find your way through documentation is an important skill! If you get stuck, please refer to Appendix B at the end of this writeup for some hints on how to navigate the documentation.

You are strongly encouraged to reach out to your style grading TA with your planned set of data structures to store the Wikipedia page graph. We are here to help you!

4.4 Getting started

Think about which objects you'll need, and begin to map out the class structure. Recall that a class at its essence is an encapsulation of some entity, maintaining its data as well as methods that can manipulate and return its data. What entities might you need to manipulate in Wikipedia link data? How do these entities relate to each other? What kind of information do those entities possess, and what data structures do a good job of storing it? Finally, how would you extract or optimize the stored data? (We will discuss the exact contents of the graph in the next section.)

A quick hint: you should use `Collections.unmodifiableXXX()` as a fast way to get an unmodifiable view of a collection out of a modifiable one.

While there is no single correct solution, we are looking for graph representations that are clean, efficient, modular, and extensible. Your code should be able to handle changes and new functionality smoothly without breaking, since you will most likely be refactoring your code many times throughout this project. It will greatly help you if the organization of your code is intuitive: You should be able to easily explain why each data structure or method exists in its respective class, and how the methods are implemented. The methods should all be asymptotically optimal with regards to running time and space usage.

As you should come to expect by now, each class you write needs to have a corresponding JUnit test that should be included in your submission. All non-trivial methods need to be tested. As usual, methods should be tested on invalid inputs to check error handling - for instance, if `parseXmlDump(null)` is called, you should throw an `IllegalArgumentException`.

4.5 Graph generation algorithm

You should aim to have your graph generation algorithm run linearly with respect to the pages and links between the pages. You also want to have reasonable space usage. Here are some guiding questions to help move you in the right direction.

- Does my graph contain all pages of the correct format which are present in the dump?
- How does my algorithm behave if I come across links to pages I haven't seen yet?
- How does my algorithm behave if I come across links to pages I have already seen?
- What data structures can I use to achieve optimal runtimes?
- How much space will this algorithm use?
- How much space will the graph need when the algorithm is done running?

We strongly recommend you think through your algorithm and write test cases before you start coding! It will help you to organize your thinking which leads to cleaner code. We provide a more thorough walkthrough of how to think about an optimal algorithm in Appendix A.

5 Dijkstra's Shortest Path Algorithm (35 points)

Required classes:

- `BinaryMinHeapImpl`
  implements `edu.upenn.cis121.project.data.BinaryMinHeap`
- `ShortestPathImpl`
  implements `edu.upenn.cis121.project.graph.ShortestPath`
- Test cases for both classes

In this project, you will be working towards building a Wikipedia Game solver, so an integral part is finding the shortest path between two wiki pages. To solve this problem, you will implement Dijkstra's algorithm on a `DoubleWeightedDirectedGraph`. It should return the shortest path from a node `src` to a node `tgt`, which exist as the first and last elements of the output path. If there is no path between the two nodes, it should return an empty iterable.

Since Java's `PriorityQueue` does not support `decreaseKey`, you will write your own `BinaryMinHeapImpl` that supports this, and other standard heap operations like `add` and `extractMin`, efficiently (so $O(\lg n)$ on average).

Please note the following:

- We are asking you to write your own priority queue data structure. Implementations that use Java's `PriorityQueue` class will receive no credit.
- For testing purposes, you should write your own class that implements the `DoubleWeightedDirectedGraph` interface. You may adapt the graph class you wrote for the BFS/MST homework, if appropriate.

6 Connected Components (45 points)

Required classes:

- `ConnectedComponentsImpl`
  implements `edu.upenn.cis121.project.ConnectedComponents`
- Any other classes that you might have written

- Test classes for all classes that you have written

Connected components in a directed graph are a bit different than in an undirected one. For example, a directed component discovered in the way that a connected component on a undirected graph would be discovered might be different depending on where you start a DFS or BFS in a directed graph (do you see why?). Instead, we introduce the notions of weakly and strongly connected components.

Definition (Weakly Connected)**.** A directed graph is said to be *weakly connected* if, for every pair of vertices in the graph, there exists an undirected path between those two vertices.

One way to think about undirected paths in a directed graph is to simply pretend that all directed edges are now undirected. Similarly, a weakly connected component is just a subgraph that is weakly connected.

Definition (Strongly Connected)**.** A directed graph is said to be *strongly connected* if, for every pair of vertices in the graph, there exists a directed path between those two vertices in both directions.

A strongly connected component is a subgraph that is both strongly connected and maximal, in the sense that no additional edges or vertices from the graph can be added to the subgraph without breaking its property of being strongly connected.

Notice that the generic parameters for `edu.upenn.cis121.project.ConnectedComponents` declares the generic parameter `V extends Comparable<V>`. We do this for testing purposes, so that the method `iterativeDepthFirstOrderIterator(Graph<V>)` has a well-defined, deterministic ordering.

6.1 Non-recursive depth-first search

Determining both weakly and strongly connected components will require a depth-first search subroutine. We **require** that you implement a lazy **iterative** implementation of depth-first search. The method to implement is `iterativeDepthFirstOrderIterator(Graph<V>)`. You will be implementing an iterator instead of the standard recursive depth-first search because on large graphs, a recursive implementation may run out of stack space.

Every call to the `next()` method of this iterator explores a new node, starting with the first vertex (given by its natural ordering). A correct implementation should return vertices as they are first discovered (as opposed to when they are finished being visited). Moreover, a correct implementation must be *lazy.* A lazy iterator means that the iterator must only discover and return one node at a time. That is, it should not prepare the whole list of nodes to return in advance. More specifically, your search should not go deeper on any one given step than is necessary to find the next node to be returned. This optimization will be checked in manual grading.

The DFS iterator should yield the vertices of the graph in depth-first order, where neighbors are explored by their natural ordering, as given by the `Comparable` that they implement. You should start with the "least" vertex.

Your DFS iterator should yield the vertices of a DFS forest. When you restart your DFS in a new component, you should continue with the component that has the "least" vertex.

Notice that the simple iterator interface does not provide a way for you to distinguish when a DFS of a single connected component ends. You can actually utilize a covariant return type to return a more specific iterator that is more useful for connected components (so that you can more efficiently use the iterator in the other methods in `ConnectedComponents`.

The DFS iterator we would like you to write should use $O(V + E)$ space. To implement DFS iteratively, we will have to keep track of the current vertex we are exploring. A stack is a good idea for this. When a vertex gets added to our stack, we will want to note that it was discovered and return it in our `next` and `nextInComponent` methods. Then we will discover each of its neighbors in a depth-first manner. Note that you should not rediscover a vertex that was already discovered.

You will also want to keep track of when a node finishes (when all of its neighbors have been discovered). This information is important for retrieving the reverse postorder for use in Kosaraju's algorithm. An additional data structure is recommended to accomplish this. Note that you are allowed to create your own methods in your iterator, as long as you implement the required ones.

6.2 Weakly connected components

Once you have thoroughly tested the previous part, implement `weaklyConnectedComponents(Graph<V>)` with your newly written DFS iterator.

6.3 Strongly connected components

For this next part, you will need to use Kosaraju's algorithm to find the strongly connected components of a graph. As in the previous part, use your DFS iterator to write this method. The method to complete is `stronglyConnectedComponents(Graph<V>)`. Kosaraju's algorithm is predicated on the fact that the SCCs of a graph are the same as the SCCs of the transpose of that graph (the graph with all edge directions reversed).

The general idea of Kosaraju's algorithm is as follows: Start a depth-first search from a vertex and get the reverse post-order traversal (this is the reverse of the post-order traversal). Then perform a series of depth-first searches on the reverse graph in the order computed from the previous step. The connected components of this DFS forest are the strongly connected components of the graph.[1]

Note that you can start with either G or $G^\top$ and still arrive at the same correct answer. That is because the strongly connected components of a graph are the same as the strongly connected components of the transpose graph.

We will require you to reverse the graph in constant time and space. *Hint*: You can "wrap" the input graph with another class. This "wrapper class" takes the input graph as an argument in the constructor, stores the input graph, and delegates all method calls to the underlying class, except when neighbors are retrieved and queried. In the case where neighbors might be involved, the wrapper class can modify the behavior and change the method calls to provide valid semantics for a reversed graph.

You can compute the post-order by adding additional functionality to your DFS iterator. As a hint, you should consider when a vertex is added to the post-order output to find the analogous place in your non-recursive DFS implementation.

7 Building a Inverted Document Index (15 points)

Required classes:

- `RecordInvertedIndexImpl`
  implements `edu.upenn.cis121.project.index.RecordInvertedIndex`
- Any other classes that you might have written
- Test classes for all classes that you have written

In this section, you will construct an inverted document index. You will find a use for this data structure in the search engine you will later write.

7.1 `RecordInvertedIndex.java`

An *inverted index* is a mapping from a particular key to descriptors for locations of that key. The class that we ask you to implement, `RecordInvertedIndex`, is similar to the concept of a multi-map, where you may associate multiple values with a particular key. Adapt a concrete subclass of `java.util.Map` to satisfy the interface of a `edu.upenn.cis121.project.index.RecordInvertedIndex`.

7.2 Possible usage of an inverted index

When we actually utilize the inverted index in subsection 9.2, we will index the text of Simple English Wikipedia. Here is a taste of what is to come:

[1] For more details, start on slide 49 on the Princeton lecture slides

Document	Text
A	Seventeen is the best number
B	David Xu likes the number 17
C	CIS 121 is cool

An inverted index would look like

Term	Document(s)
Seventeen	A
is	A, C
the	A, B
best	A
number	A, B
David	B
Xu	B
likes	B
17	B
CIS	C
121	C
cool	C

8 Wikipedia Game (20 points)

Classes to submit:

- `WikipediaGameImpl`
  implements `edu.upenn.cis121.project.engine.WikipediaGame`
- `WikipediaGameFactoryImpl`
  implements `edu.upenn.cis121.project.engine.WikipediaGameFactory`
- Test cases

You will now get to put together all these algorithms to build your very own Wikipedia Game! This game is typically played by counting the number of links between two pages on Wikipedia. However, this way doesn't have any notion of weighted edges. You can create weights on your graph edges by using measures of centrality as follows:

- Equal weighted edges (all the edges have weight 1)
- Weighting by indegree: for an edge from u to v, weight the edge as the indegree of v, which biases paths through less frequently linked-to pages

Your Wikipedia Game solver will live in a class called `WikipediaGameImpl`. The class will take in an unweighted, directed graph as an input for its constructor and it will calculate the edge weightings for the two schemes listed above and then store them. How will you store your edge weighting schemes? (Hint: create an inner class that you can use to represent edges). In order to utilize your stored edges, you will also need to create a graph class that allows you to supply the edges to the graph and specify edgeweights. (Hint: you can do this by providing an implementation of `DoubleWeightedDirectedGraph`). Finally, you will implement a method that solves the Wikipedia Game according to whichever weighting scheme is specified by combining your new graph implementation with Dijkstra's algorithm to find shortest paths between source and target vertices.

9 Search Ranking (20 points)

Classes to submit:

- `BasicTfIdfSearchEngine`
  implements `edu.upenn.cis121.project.engine.SearchEngine`
- `BasicTfIdfSearchEngineFactory`
  implements `edu.upenn.cis121.project.engine.SearchEngineFactory`
- Any other classes that you might have written
- Test classes for all classes that you have written

For the final section, you will now implement a search engine based on the data structures that you have created.

9.1 GUI for search

A graphical frontend (`edu.upenn.cis121.project.app.SearchEngineApplication`) has already been written for you. If run with no arguments, then a fake backend will be used. The fake backend produces random search results. To use a particular dataset, you should load the XML dump through "File" → "Load dataset...". To actually use your implementation class, specify the fully-qualified name of the factory class implementing `edu.upenn.cis121.project.engine.SearchEngineFactory` as an argument to the program. To do this, find the application class file **in the project jar** (`SearchEngineApplication.class`) and go to Run As →Run Configurations. Under the "arguments" tab, add your factory implementation to the program arguments field, then Apply and Run.

To write the backend, finish `edu.upenn.cis121.project.impl.BasicTfIdfSearchEngineFactory` and `edu.upenn.cis121.project.impl.BasicTfIdfSearchEngine`. You have to implement `loadXmlDump(File)` and `search(String)` from `edu.upenn.cis121.project.engine.SearchEngine`. In the application lifecycle, the method `loadXmlDump` is first called by the application to request that your backend engine implementation prepare and load a dump file. Subsequently, the `search` method may be called multiple times to retrieve results.

In a more sophisticated system, we would have some way of reporting the load progress to the frontend, but we have omitted this functionality for simplicity. These are just some considerations that come to mind when we reexamine our basic search engine.

9.2 Term frequency–inverse document frequency ranking

For this part, you must implement the class `edu.upenn.cis121.project.impl.BasicTfIdfSearchEngine`, which will be backend engine with a ranking based on term frequency–inverse document frequency. You should prepare the tf–idf rankings in `loadXmlDump` for later queries by the `search` method.

In our model, "terms" are just words in a wiki page, "documents" are the text of the wiki pages, and the "corpus" is entire set of wiki pages (i.e., all the text pages of Simple English Wikipedia). Term frequency is a simple measure of how often a term appears in a document. The inverse document frequency measures the importance of a term. Notice that when you compute the term frequency, all terms are considered equally important. However, words like "a", "the", and "that" appear almost everywhere, so don't tell us much about the contents of a document. Thus, the inverse document frequency is lower for the frequently-occurring terms, but scales higher for the rare terms.

The *term frequency* (tf) is defined as

$$\mathrm{tf}(t,d) = f(t,d)$$

where t is a term and d is a document, and the function f returns the number of times that t appears in d. The term frequency is just a raw frequency count of the term in the particular document.

The *inverse document frequency* (idf) is defined as

$$\mathrm{idf}(t,D) = \ln \frac{|D|}{1 + \big|\{d \in D \mid t \in d\}\big|}$$

where t is a term and D is the set of documents comprising our corpus. The inverse document frequency is the log-scaled quotient of the total number of documents and the number of documents that contain the term. Note that Java's `Math.log` is base e.

The *term frequency–inverse document frequency* (tf–idf) is the product of *term frequency* (tf) and *inverse document frequency* (idf):

$$\text{tf–idf}(t, d, D) = \text{tf}(t, d) \cdot \text{idf}(t, D)$$

To actually implement the search functionality for the engine, we need to a way to determine which document is relevant to a particular query. To do this, we will compute a tf–idf score for the query on each particular document.[2] We will handle multi-word queries in a simple manner: Our score for a query on a particular document will be the sum of the tf–idf values for each *unique* word in the query.

$$\text{score}(\text{query}, \text{document}) = \sum_{\text{word} \in \text{unique words of query}} \text{tf–idf}(\text{word}, \text{document}, \text{Simple English Wikipedia pages})$$

Note: When we test your implementation of tf–idf ranking, we will be going through the `edu.upenn.cis121.project.engine.SearchEngine` interface.

Important: When you implement a `edu.upenn.cis121.project.engine.SearchResult`, you must ensure that your tf–idf value is returned in `getSubValueMap()` with the key "tf-idf". The SubValueMap is simply a map of different scores or metrics associated with a search result, with the keys being the metric names as `String`s. This interface could be extended to included other ways to measure search relevance, like PageRank. However, for this project you are only using "tf-idf". Therefore this should be the only key in your map, with the tf-idf score as its value.

Implementation note: You will need to split text into constituent words. For our tests, you need only to split the text on whitespace, like by `str.split("\\s+")`. For simplicity, you can lowercase all characters and throw out all characters that are not in the 26 letters of the English alphabet. So some text like, "Isn't 17 an awesome-great number?" will produce the words "isnt", "an", "awesomegreat", "number", but with no mapping for "17" because "17" does not have any lowercase English letters. Note that you might want to improve upon this naïve word tokenization for extra credit.

Implementation note: You only need to search for *exactly matching* terms in the corpus. That is, you do *not* need to worry about search terms appearing inside of longer word.

Notice that the `search(String, int)` has a parameter for the maximum number of results to return. What will be the most efficient way to find the first k top-scoring pages from n total pages?[3] Keep in mind that $k \ll n$ and that there are >300k pages in the Simple English Wikipedia dump.

10 Summary and Analysis (30 points)

File to submit:

- `questions.txt`

It's time to reflect on what you've learned! Please submit a `questions.txt` file with the answers, in order, to the following questions:

1. For the following algorithms that you implemented, describe your code's big-Theta runtime, provide a justification for that runtime, and explain if that runtime is optimal:
   - Depth-first search
   - Dijkstra's algorithm
   - Kosaraju's algorithm
2. Write several paragraphs on the design choices you made in section 4 (graph parsing). Describe your internal API, and how all the classes interact with each other. Be sure to justify the choices you made by referencing running time efficiency, space efficiency, and object-oriented design principles. Were you able to find a good balance between the indirection and two-pass algorithms? If so, how?

[2]Helpful tip (to check your understanding): There is only one idf score for a particular term for every document, but tf score changes per document.

[3]This is similar to question 5 on the heaps lab.

3. Discuss your own search ranking implementation and some limitations of your own approach. Then provide one or two extensions to how you might improve your search engine (a rough outline of how you would implement it and test it).

11 Style and Tests (25 points)

The above parts together are worth a total of 195 points. The remaining 55 are such that 25 points are awarded for code style, documentation, and sensible tests, and 30 are for milestones (10pts each). Style is worth 10 of the 25 points, and you will be graded according to the 121 style guide.

For testing, refer to the 121 testing guide. Make sure you consider edge cases, i.e., the more "interesting" inputs and situations. Use multiple methods instead of cramming a bunch of asserts into a single test method. Be sure to demonstrate that you considered "bad" inputs such as null inputs. Be sure to also demonstrate that you've tested for inputs to methods that should throw exceptions! Your test cases will be automatically graded by our code coverage tool and are worth 15 of the 25 points. You will have to thoroughly test your code to get full points! This includes testing any helper methods you have written.

A Approach to Graph Parsing

Hopefully now you have spent some time thinking about the algorithm for creating the graph. We aren't going to give you the full optimal algorithm, but we will walk you through two other approaches. In the main writeup, we highlighted some issues you may come across: how do you prevent creating duplicate nodes during your parsing?

A.1 Algorithm the first: "indirection" with a map

We know that all pages are identified by their page title. It would be a natural fit to have a map inside of our graph that helps us map from page titles to vertex objects. Vertex objects would store a collection of page titles to identify the neighbors. If you do it this way, you will be able to add edges without worrying about duplicate vertex objects, since you would be referencing them by their String title. This isn't so much an algorithm as a method of designing your graph class. Since you have freedom to add methods to your class, you can make it such that you add edges between Strings and the graph class will handle finding the correct object to modify.

What are the downsides of this method? In order to look up a vertex and its neighbors, now we have an additional layer of *indirection*—to retrieve a neighboring vertex, we must obtain the page title of the neighbor, and then ask the global map for the object associated with that page title. We want to figure out a way to avoid this—that is, we should be able to access neighboring page objects without going back to the graph class.

A.2 Algorithm the second: two-pass

In order to make sure that the page also updates its inlinks, we need to somehow have access to the object we already created. One intuitive way we can do this is create a map from page titles to page objects. In our first pass, we will only create this map in order to catch all the pages contained in the dump. In our second pass, we will handle the page links. Since we have already created all page objects, to add an edge, we can just do a map lookup to find the already-existing page object. We can then add this object to some collection that we store for the page object.

This method improves on the last one because our map now only exists during the lifetime of the algorithm. Notice that we avoided the additional indirection from the previous algorithm, since each page object directly stores the neighboring page object.

A.3 The optimal algorithm

However, we can still improve! You can see the strengths of both of these algorithms: one of them is single pass, but takes up a lot of space. The other uses less space, but requires two passes. Can you combine these two approaches?

Note: Since we have given you the outlines for these two algorithms, we are guiding you towards the optimal reference implementation. We ask that you do your best to implement the most efficient algorithm.

B Navigating XML Parsing Documentation

The key to navigating documentation is having an idea of what you want to do, and looking at the documentation to figure out how to accomplish it. Based on what we went through above, you should have a decent idea of what information you need to get out of the XML.

As suggested in the writeup, you should try to write this out on paper. How can you go about writing on paper? Look at the various classes and the methods each provides. Pay close attention to the types of inputs and outputs because this will guide you through the classes.

From the starter code, we have two classes to look at: `XMLEventReader` and `XMLEvent`.

From the sample code, we can see the `XMLEventReader` looks familiar: it acts just like an iterator! The method of iterating through `XMLEvents` looks like any typical iterator pattern. If you aren't familiar with XML, you should read the next section of the appendix. When we skim through the other methods in this class, we see that we can get the text inside a text-only element. Looking ahead, we know that we will want to store the text inside our Page objects too.

Now we can go a level deeper and look at the `XMLEvent` type. This is where we pause and ask what we need to do with the tags. You first need to know what the tag is in order to operate on it: we will be doing different things depending on whether it's a `<text>` or a `<title>`. Figuring out how to isolate the name of the tag will be our first task. We can follow a chain of methods by looking at their outputs and inputs to get to what we want.

While inside `XMLEvent`, you should take a look at the methods until you see something that will get you closer to the name. Look at the `asStartElement` method: this means we want to read this element as an opening tag. How can we tell that this method might be useful? Inside the `StartElement` interface, you will see a few methods that look like they have to do with the name. This is where your XML knowledge will come in handy. The tag isn't formatted exactly as you might expect: it includes a lot of contextual details that we don't really care about. What you want is called the 'local part'. `QName` looks like it will get us there. By following the path, we found our way to getting the tag titles.

Now that you found a particular tag, you need to find the text between the start and end tags. The preferred way to read all of the text between two tags at once is to use the `getElementText()` on your `XMLEventReader` instance.[4]

Now what other tasks do we need to do with XML parsing? We know that we will need to get the links between pages. As said in the main writeup, `edu.upenn.cis121.project.util.mediawiki.MediaWikiUtils` will help you with this. This one is much easier to navigate than the last: you can parse the text to get back a `MediaWikiText` object. That class provides a method to get the outlinks for you as a `List<String>`.

[4]You can also check if an `XMLEvent` is a `Characters` event. But this event can potentially be emitted multiple times depending on the settings given to the `XMLInputFactory`, so you would have to collect all the text fragments and parse at the end.

Chapter 8

XML

Most research on data integration has, as described in the previous chapters of this book, centered on the relational model. In many ways, the relational model (and the Datalog query language) are the simplest and cleanest formalisms for data and query representation, so many of the fundamental issues were considered first in that setting.

However, as such techniques are adapted to meet real-world needs, they typically get adapted to incorporate XML (or its close cousin JSON[1]). For instance, IBM's Rational Data Architect or Microsoft's BizTalk Mapper both use XML-centric mappings.

The reason for this is straightforward. XML has become the default format for data export from both database and document sources; and many additional tools have been developed to export to XML from legacy sources (e.g., COBOL files, IMS hierarchical databases). Prior to XML's adoption, data integration systems needed custom wrappers that did "screen scraping" (custom HTML parsing and content extraction) to extract content from XML, and that translated to the proprietary wire formats of different legacy tools. Today, we can expect most sources to have an XML interface (URIs as the request mechanism and XML as the returned data format), and thus the data integrator can focus on the semantic mappings rather than the low-level format issues.

We note that XML brings standardization not only in the actual data format, but also in terms of an entire ecosystem of interfaces, standards, and tools: DTD and XML Schema for specifying schemas, DOM and SAX for language-neutral parser interfaces, WSDL/SOAP or REST for invoking Web services, text editors and browsers with

[1]JSON, the JavaScript Object Notation, can be thought of as a "simplified XML" although it also closely resembles previous *semistructured data* formats like the Object Exchange Model.

built-in support for XML creation, display, and validation. Any tool that produces and consumes XML automatically benefits from having these other components.

This book does not attempt to cover all of the details of XML, but rather to provide the core essentials. In this chapter, we focus first on the XML data model in Section 8.1 and the schema formalisms in Section 8.2. Section 8.3 presents several models for querying XML, culminating in the XQuery standard that is implemented in many XML database systems. We then discuss the subsets of XQuery typically used for XML schema mappings (Section 8.4) and then discuss XML query processing for data integration (Section 8.5).

8.1 Data Model

Like HTML (HyperText Markup Language), XML (eXtensible Markup Language) is essentially a specialized derivative of an old standard called SGML (Structured Generalized Markup Language). As with these other markup standards, XML encodes document meta-information using *tags* (in angle brackets) and *attributes* (attribute-value pairs associated with specific tags).

XML distinguishes itself from its predecessors in that (if correctly structured, or *well-formed*) it is *always parsable* by an XML parser — regardless of whether the XML parser has any information that enables it to interpret the XML tags. To ensure this, XML has strict rules about how the document is structured. We briefly describe the essential components of an XML document.

Processing instructions to aid the parser. The first line of an XML file tells the XML parser information about the character set used for encoding the remainder of the document; this is critical since it determines how many bytes encode each character in the file. Character sets are specified using a *processing-instruction*, such as `<?xml version="1.0" encoding="ISO-8859-1"?>`, which we see at the top of the example XML fragment in Figure 8.1 (an excerpt from the research paper bibliography Web site DBLP, at `dblp.uni-trier.de`). Other processing instructions may specify constraints on the content of the XML document, and we will discuss them later.

Tags, elements, and attributes. The main content of the XML document consists of tags, attributes, and data. XML tags are indicated using angle-brackets, and must come in pairs: for each open-tag `<tag>`, there must be a matching close-tag `</tag>`. An open-tag / close-tag pair and its contents are said to be an *XML element*. An element may have one or more *attributes*, each with a unique name and a value specified within the open-tag: `<tag attrib1="value1" attrib2="value2">`.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<dblp>
  <mastersthesis mdate="2002-01-03" key="ms/Brown92">
   <author>Kurt P. Brown</author>
   <title>PRPL: A Database Workload Specification Language</title>
   <year>1992</year>
   <school>Univ. of Wisconsin-Madison</school>
  </mastersthesis>
  <article mdate="2002-01-03" key="tr/dec/SRC1997-018">
   <editor>Paul R. McJones</editor>
   <title>The 1995 SQL Reunion</title>
   <journal>Digital System Research Center Report</journal>
   <volume>SRC1997-018</volume>
   <year>1997</year>
   <ee>db/labs/dec/SRC1997-018.html</ee>
   <ee>http://www.mcjones.org/System_R/SQL_Reunion_95/</ee>
 </article>
  ...
</dblp>
```

Figure 8.1: Sample XML data from the DBLP Web site

Of course, an element may contain nested elements, nested text, and a variety of other types of content we will describe shortly. It is important to note that every XML document must contain a single *root element*, meaning that the element content can be thought of as a tree and not a forest.

Example 8.1: Figure 8.1 shows a detailed fragment of XML from DBLP, as mentioned on the previous page. The first line is a processing instruction specifying the character set encoding. Next comes the single *root element*, `dblp`. Within the DBLP element we see two sub-elements, one describing a `mastersthesis` and the other an `article`. Additional elements are elided.

Both the MS thesis and article elements contain two attributes, `mdate` and `key`. Additionally, they contain sub-elements such as `author` or `editor`. Note that `ee` appears twice within the `article`. Within each of the sub-elements at this level is *text content*, each contiguous fragment of which is represented in the XML data model by a text node.

Namespaces and qualified names. Sometimes an XML document consists of content merged from multiple sources. In such situations, we may have the same tag names in several sources, and may wish to differentiate among them. In such a situation, we can give each of the source documents a *namespace*: this is a globally unique name, specified in the form of a Uniform Resource Indicator (URI). (The URI is simply a unique name specified in the form of a qualified path, and does not necessarily represent the address of any particular content. The more familiar Uniform Resource Locator, or URL, is a special case of a URI where there is a data item whose content can be retrieved according to the path in the URI.) Within an XML document, we can assign a much shorter name, the *namespace prefix*, to each of the namespace URIs. Then, within the XML document, we can "qualify" individual tag names with this prefix, followed by a colon, e.g., `<ns:tag>`. The *default namespace* is the one for all tags without qualified names.

Document order. XML was designed to serve several different roles simultaneously: extensible document format generalizing and replacing HTML; general-purpose markup language; structured data export format. It distinguishes itself from most database-related standards in that it is *order-preserving* and generally *order-sensitive.* More specifically, the order between XML elements is considered to be meaningful, and is preserved and queriable through XML query languages: this enables, e.g., ordering among paragraphs in a document to be maintained or tested. Perhaps surprisingly, XML *attributes* (which are treated as properties of elements) are *not* order-sensitive, although XML tools will typically preserve the original order.

We typically represent the logical or data model structure of an XML document as a tree, where each XML node is represented by a node in the tree, and parent-child relationships are encoded as edges. There are seven node types, briefly alluded to above:

- **Document root**: this node represents the entire XML document, and generally has as its children at least one processing instruction (representing the XML encoding information) and a single root element.
- **Processing instruction**: these nodes instruct the parser on character encodings, parse structure, etc.
- **Comment**: as with HTML comments, these are human-readable notes.
- **Element**: most data structures are encoded as XML elements, which include open and close tags plus content. A content-free element may be represented as a single *empty tag* of the form `<tag/>`, which is considered equivalent to an open-tag / close-tag sequence.

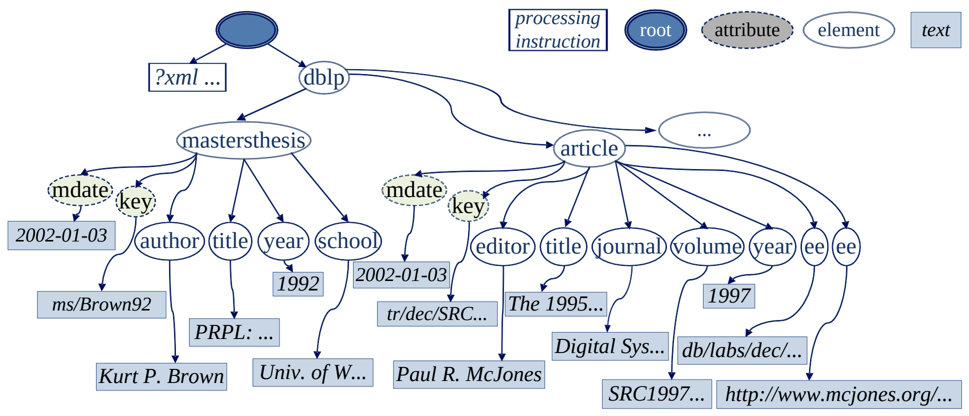


Figure 8.2: Example data model representation for the XML fragment of Figure 8.1. Note that the different node types are represented here using different shapes.

- **Attribute**: an attribute is a name-value pair associated with an element (and embedded in its open tag). Attributes are not order-sensitive, and no single tag may have more than one attribute with the same name.
- **Text**: a text node represents contiguous data content within an element.
- **Namespace**: a namespace node qualifies the name of an element within a particular URI. This creates a *qualified name*.

Each node in the document has a unique identity, as well as a relative ordering and (if a schema is associated with the document) a datatype. A depth-first, left-to-right traversal of the tree representation corresponds to the node ordering within the associated XML document.

Example 8.2: Figure 8.2 shows an XML data model representation for the document of Figure 8.1. Here we see five of the seven node types (comment and namespace are not present in this document).

8.2 XML Structural and Schema Definitions

In general, XML can be thought of as a semi-structured hierarchical data format, whose leaves are primarily string (text) nodes and attribute node values. To be most useful, we must add a *schema* describing the semantics and types of the attributes and elements — this enables us to encode non-string datatypes as well as inter- and intra-document links (e.g., foreign keys, URLs).