Programming Assignment 5 (BFS, DFS, and Algorithms on DAG)

Department of Computer Science, University of Wisconsin – Whitewater
Data Structure (CS 223)

1 Overview

We are going to:

- **Implement Breadth First Search, and use it to count the number of components**.
- **Implement Depth First Search, and use it to compute the transitive closure**.
- **Find Single Source Longest Paths in a DAG**.
- **Use Topological Sorting to help mutants**.

To this end, **your task is to implement the following methods**:

- `helpBFS`, `executeBFS`, `countComponents` in `BFS.h/BFS.java`
- `helpDFS`, `executeDFS`, and `computeTransitiveClosure` in `DFS.h/DFS.java`
- `longestPaths` in `DAG.h/DAG.java`
- `readLanguage`, `makeGraph`, and `getOrder` in `MutantLanguage.h/MutantLanguage.java`

The project also contains additional files (which you do not need to modify).

Use `TestCorrectness.cpp/TestCorrectness.java` to test your code.

For each part, you will get an output that you can match with the output I have given to verify whether your code is correct, or not. Output is provided separately in the `ExpectedOutput` file. Should you want, you can use `www.diffchecker.com` to tally the output.

2 Graph Data

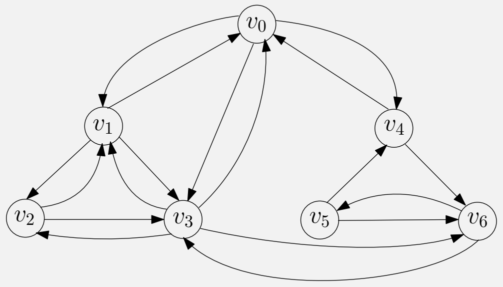
To test BFS and DFS, we use the graphs in files: `unweighted1.txt,` and `unweighted2.txt`. To test ComponentCounter, we use the graphs in files: `undirected1.txt,` and `undirected2.txt`. To test longest paths, we use: `dag1.txt` and `dag2.txt`. Next, a description is given.

2.1 Unweighted Graph File Structure

Consider the first graph on the next page; the file `unweighted1.txt` represents this graph. The first line in the file stores the number of vertices, which is 7. Then, there are 7 lines, representing v_0 through v_6. The first number in each line is the number of outgoing edges of that vertex. Then, we have the vertices to which each outgoing edge leads. For e.g., in line 2 (i.e., 3 1 3 4): the first number is 3, indicating vertex v_0 has 3 outgoing edges to the vertices v_1, v_3, and v_4 respectively.

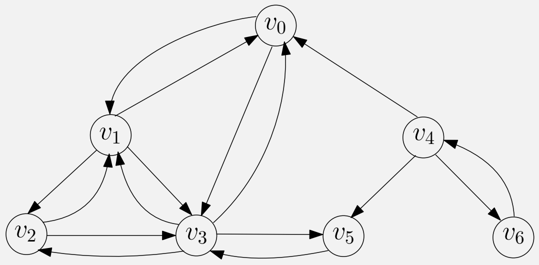
unweighted1.txt and corresponding graph

```
7
3 1 3 4
3 0 2 3
2 1 3
4 0 1 2 6
2 0 5
2 4 6
2 3 5
```



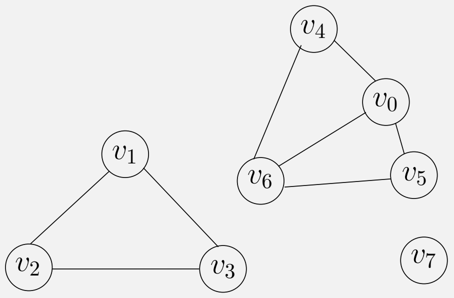
unweighted2.txt and corresponding graph

```
7
2 1 3
3 0 2 3
2 1 3
4 0 1 2 5
3 0 5 6
1 3
1 4
```



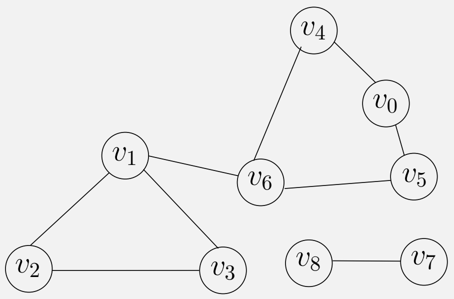
undirected1.txt and corresponding graph

```
8
3 4 5 6
2 2 3
2 1 3
2 1 2
2 0 6
2 0 6
3 0 4 5
0
```



undirected2.txt and corresponding graph

```
9
2 4 5
3 2 3 6
2 1 3
2 1 2
2 0 6
2 0 6
3 1 4 5
1 8
1 7
```

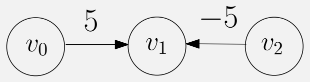


2.2 DAG File Structure

Consider the larger graph below; the file content in `dag2.txt` represents this graph. The first line in the file stores the number of vertices, which is 8. Then, there are 7 lines, representing vertices v_0 through v_7. The first number in each line is the number of outgoing edges of that vertex. Then, we have the vertices to which each outgoing edge leads and the respective edge weight in an alternating way. For e.g., in line 2 (i.e., 2 1 4 2 5): the first number is 2, indicating vertex v_0 has 2 outgoing edges – one to the vertex v_1 with weight 4 and the other to vertex v_2 with weight 5.

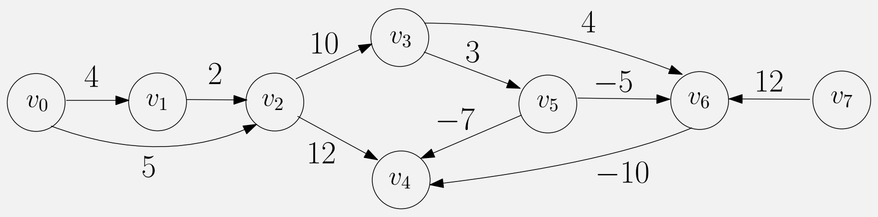
dag1.txt and corresponding DAG

```
3
1 1 5
0
1 1 -5
```



dag2.txt and corresponding DAG

```
8
2 1 4 2 5
1 2 2
2 3 10 4 12
2 5 3 6 4
0
2 4 -7 6 -5
1 4 -10
1 6 12
```



2.3 Adjacency List: Representing Graphs in Memory

The vertices in the graph are numbered 0 through $n-1$, where n is the number of vertices. We use a two-dimensional jagged array **adjList** (called *adjacency list*) to represent the graph. Specifically, row index i in the array corresponds to the vertex v_i, i.e., row 0 corresponds to v_0, row 1 corresponds to v_1, and so on. Each cell in row i stores an outgoing edge of the vertex v_i. Each edge has 3 properties – *src*, *dest*, and *weight*, which are respectively the vertex from which the edge originates, the vertex where the edge leads to, and the edge weight. Additionally, we use an array **outDegree** of length n, where $outDegree[i]$ = the number of outgoing edges of the vertex v_i.

As an example, consider the larger graph in Figure 2. Row 0 of adjList contains the following edges: $\langle 0,1,4\rangle$ and $\langle 0,2,5\rangle$; this is to be interpreted as vertex v_0 has 2 outgoing edges – one edge to the vertex v_1 with weight 4 and the other to vertex v_2 with weight 5. Hence, $outDegree[0] = 2$. Likewise, row 1 contains the edge $\langle 1,2,2\rangle$, row 2 contains the edges $\langle 2,3,10\rangle$ and $\langle 2,4,12\rangle$, and so on. Consequently, $outDegree[1] = 1$, $outDegree[2] = 2$, and so on.

In case of unweighted graphs, the edge weight is always 1; rest remains the same.

In a nutshell, Edge is a class which has three integer variables – *src*, *dest*, and *weight*. The adjacency list, therefore, is a jagged array, whose type is Edge. In C++, we implement *adjList* as a vector of Edge vectors. In Java, we implement *adjList* as an ArrayList of Edge ArrayLists.

Here, each jagged array cell stores an Edge object. Previously, you have used dynamic array (vector or ArrayList) to return numbers from a method. Over there, the type of the dynamic array was integer. Here, the type of the dynamic array is Edge. Each such dynamic array forms a row of *adjList*, and *adjList* itself is a dynamic array of such Edge-type dynamic arrays.

3 Implementing Queue with a Linked List

3.1 C++

- To create an integer queue: `list<int> nameOfList`;
- To get the size of the queue: `nameOfList.size();`
- To enqueue: `nameOfList.push_back(15);`
- To dequeue, use `nameOfList.front()` to get the first number and then use `nameOfList.pop_front();` to remove it.

3.2 Java

- To create an integer queue: `LinkedList<Integer> nameOfList = new LinkedList<Integer>()`;
- To get the size of the queue: `nameOfList.size();`
- To enqueue: `nameOfList.addLast(15);`
- To dequeue: `nameOfList.removeFirst();` This method returns the first number and removes it from the queue.

4 Breadth First Search

First implement the `helpBFS` and `executeBFS` methods in `BFS.h/BFS.java` by using the following pseudo-codes.

In C++, use INT_MAX for ∞. In Java, use Integer.MAX_VALUE for ∞.

BFS Helper

```
• Create an integer queue $vertexQ$.
• Enqueue $s$ to $vertexQ$.
• Set $level[s] = 0$.
• while ($vertexQ's\ size > 0$), do the following:
    – let $v$ be the vertex obtained by dequeueing $vertexQ$
    – for ($i = 0$ to $i < outDegree[v]$), do the following:
        ∗ let $adjEdge$ be the $i^{th}$ outgoing edge of $v$
          C++ syntax: $Edge\ \&adjEdge = adjList.at(v).at(i);$
          Java syntax: $Edge\ adjEdge = adjList.get(v).get(i);$
        ∗ let $w$ be the destination of $adjEdge$
        ∗ if ($level[w]\ equals\ \infty$) then
            · set $level[w]$ to $level[v] + 1$
            · enqueue $w$ to $vertexQ$
```

BFS Main

```
• Allocate $numVertices$ cells for the $level[\,]$ array.
• Use a loop to initialize all cells of $level[\,]$ to $\infty$.
• Call the BFS Helper method with $s$ as argument
```

Counting the number of components. A component in an undirected graph is a subset of vertices (and edges) such that there is a path between every pair of vertices. For example, `undirected1` graph has 3 components – $\{v_0, v_4, v_5, v_6\}$; $\{v_1, v_2, v_3\}$; and $\{v7\}$. Likewise, `undirected12` graph has 2 components – $\{v_0, v_1, v_2, v_3, v_4, v_5, v_6\}$; and $\{v_7, v_8\}$.

Use the following pseudo-code to fill up the `countComponents` method.

Count Components

```
• Allocate $numVertices$ cells for the $level[\,]$ array.
• Use a loop to initialize all cells of $level[\,]$ to $\infty$.
• Initialize a counter to 0.
• for ($i = 0$ to $i < numVertices$), do the following:
    – if ($level[i]$ $equals$ $\infty$) then
        ∗ call BFS helper method with $i$ as argument
        ∗ increment $counter$ by one
• return the counter
```

5 Depth First Search

First implement the `helpDFS` and `executeDFS` methods in `DFS.h`/`DFS.java` by using the following pseudo-codes.

In C++, use INT_MAX for ∞. In Java, use Integer.MAX_VALUE for ∞.

DFS Helper

```
• Set $closed[v]$ to true
• For each adjacent edge $adjEdge$ of $v$, do the following:[a]
    – let $w$ be the destination of $adjEdge$
    – if $w$ is not closed, do the following:
        ∗ set $level[w] = level[v] + 1$
        ∗ recursively call DFS Helper on $w$
```

[a] Use *outDegree*[ ] and a for-loop to get hold of each adjacent edge; you have done the same for BFS

```
DFS Main
  • Allocate $numVertices$ cells for the $level[\,]$ and $closed[\,]$ arrays.
  • Use a loop to initialize each cell of $closed[\,]$ to $false$ and each cell of $level[\,]$ to $\infty$.
  • Set $level[s]$ to 0.
  • Call DFS Helper method with $s$ as argument.
```

Computing the transitive closure. A transitive closure is a matrix M such that $M[i][j]$ is true if there is a path from vertex v_i to vertex v_j, else $M[i][j]$ is false.

Use the following pseudo-code to complete the `computeTransitiveClosure` method.

```
Compute Transitive Closure
  • Create a two-dimensional boolean matrix $M$ that has $numVertices$ many rows.
    C++ syntax: bool **M = new bool*[numVertices];
    Java syntax: boolean[ ][ ] M = new boolean[numVertices][ ];
  • for ($i = 0$ to $i < numVertices$), do the following:
      – call DFS main method with $i$ as argument
      – set $M[i]$ to $closed$
  • return the matrix
```

6 Single Source Longest Paths in a DAG

In the lecture notes, you have seen how to compute the shortest paths in a DAG after its topological order has been found. We remarked that the shortest path algorithm can be easily modified to find longest paths. Here, we will implement the same; moreover, we will find longest paths and compute topological order at the same time. See the pseudo-code in the next page.

A Remark about Implementation: Note that we are setting the *dist* array entries to very small values (INT_MIN for C++ and $Integer.MIN\_VALUE$ for Java) to simulate $-\infty$. Let us assume that we have a graph

$$v_0 \rightarrow v_1 \leftarrow v_2$$

where the edge weight from v_0 to v_1 is 5 and the edge weight from v_2 to v_1 is -5. A topological order is v_0, v_2, v_1. If we compute the longest path from v_0, initially $distance[1]$ would be set to

$$distance[1] = distance[0] + 5 = 0 + 5 = 5$$

Now on processing v_2, we will compute $(distance[2] - 5)$, which will be a large positive number due to rounding of numbers in Java/C++. This will lead to $distance[1]$ being erroneously set to a positive number larger than 5 (which is incorrect as there is no way to reach v_2 from v_0).

Hence, if *distance* of the dequeued vertex equals $-\infty$, we simply skip the remaining code inside the outer for-loop.

Longest Path in a DAG

```
• Allocate $numVertices$ cells for the $topoOrder[\,]$, $distance[\,]$, and $inDegree[\,]$ arrays.
• Use a loop to initialize all cells of $inDegree[\,]$ to 0 and all cells of $distance[\,]$ to $-\infty$
  In C++, use INT_MIN for $-\infty$. In Java, use Integer.MIN_VALUE for $-\infty$.
• for ($i = 0$ to $i < numVertices$), do the following:
    – for($j = 0$ to $j < outDegree[i]$), do the following:
        ∗ Let $adjEdge$ be the $j^{th}$ outgoing edge of $i$. See BFS pseudo-code for syntax.
        ∗ increment $inDegree[adjEdge's\ destination]$ by 1
• Create an integer queue $vertexQ$
• for ($i = 0$ to $i < numVertices$), do the following:
    – if ($inDegree[i]$ equals 0), enqueue $i$ into the $vertexQ$
• Set $distance[s] = 0$ and initialize an integer variable $topoLevel = 0$
• while ($vertexQ's\ size > 0$), do the following:
    – let $v$ be the vertex obtained by dequeueing $vertexQ$
    – assign $topoOrder[topoLevel] = v$
    – increment $topoLevel$ by 1
    – for ($j = 0$ to $j < outDegree[v]$), do the following:
        ∗ let $adjEdge$ be the $j^{th}$ outgoing edge of the vertex $v$
        ∗ let $adjVertex$ be the destination of $adjEdge$
        ∗ decrement $inDegree[adjVertex]$ by 1
        ∗ if ($inDegree[adjVertex]$ equals 0), then enqueue $adjVertex$ to $vertexQ$
        ∗ if ($distance[v] \neq -\infty$), do the following:
            · let $len = distance[v]+$ weight of $adjEdge$
            · if ($len > distance[adjVertex]$), set $distance[adjVertex] = len$
```

7 Mutant Language

Professor Xavier is trying to develop a new language for fellow mutants. As with any language, the main constituent are words via which one can effectively communicate with one another. However, the language is new, and mutants are not yet conversant about the meaning of the words; therefore, professor Xavier has to design a standard dictionary which will map mutant words to their English meaning. Since fellow mutants need to be able to search the dictionary, the Professor needs to make sure that words can be ordered (like a normal English dictionary). Therefore, after creating the words, Professor Xavier needs to ensure that there are no cyclic dependencies. Being sympathetic to the mutant cause, we want to help Professor to make sure that the language is meaningful.

To test, we use the language in files: `mutant1.txt, mutant2.txt` and `mutant3.txt`. Next, a description of the files are given.

7.1 File Description

The first line contains two numbers. The first is the number of distinct characters in the language. For example, in `mutant1.txt`, it is 5 (the distinct characters being $\{a, b, c, d, e\}$). The second is the number of words in the language; here, it is 9. Then, we have the 9 words of the language.

7.2 Comparing Strings

Before proceeding further, recall how you compare strings lexicographically (i.e., dictionary order wise). You match the strings one character at a time, until you find a mismatch (or exhaust one of the strings). The lexicographically smaller string is the one with the smaller mismatched character (or the shorter one if you exhaust one string).

7.3 Assumptions

For the sake of simplicity, we assume that the words contains letter from the lower-case English alphabet in a contiguous way. Specifically, it contains the letters starting from a and we will not skip a letter. We will also assume that all words are distinct. Also, Professor Xavier makes sure that if there are two words X and Y, where X is a prefix of Y (such as $X = abc$ and $Y = abca$), she always places X before Y. If that were not the case, more complications arise.

7.4 What we need to ensure?

We have to make sure that the words in the language can be ordered in a dictionary. In other words, the first word must be lexicographically smaller than the second, the second smaller than the third, and so on. However, since this is a new language, it is no longer necessary that a is smaller than b, and b is smaller than c, and so on. We are perfectly happy as long as we can somehow order them (such as $b < a < c < e < d$), and we do not create a cycle (such as $b < a$, $a < c$, and $c < b$).

7.5 How do we do it?

To verify the correctness of the language, we create a graph as follows. For each distinct letter, you create a vertex. Specifically, a corresponds to the vertex v_0, b corresponds to the vertex v_1, and so on. We scan each word one-by-one and compare it to the next one character at a time. Suppose, the first character where the current word mismatches the next one are α and β respectively. Then, we draw an edge from the vertex corresponding to α to the vertex corresponding to β. In `mutant1.txt`, we compare *baa* and *abc* first. Since $b \neq a$, we draw an edge from v_1 to v_0. Now, we compare *abc* and *abca*; since no mismatch occurs (before we exhaust a word), we do not create an edge. Then, we compare *abca* and *cabe*; since $a \neq c$, we draw an edge from v_0 to v_2. Next we compare *cabe* and *cad*; the first two characters match, but then $b \neq d$, and we draw an edge from v_1 to v_3. We proceed like this until we exhaust all the words. See next page for an illustration.

Now, the the language is valid (i.e., we can create a dictionary with the words) if the graph formed is acyclic; in that case, the precedence order of the characters is given by a topological order of the graph. To detect whether or not the graph is acyclic, we apply topological sorting.

7.6 Why does it work?

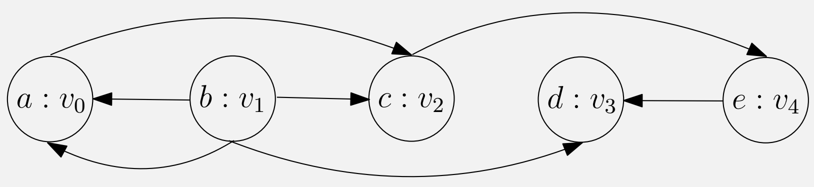
Observe that whenever we find a mismatch, we draw an edge from the mismatched character of current word to that of next word. Hence, if the graph contains a cycle, then it must mean that there are words $X_1, X_2, \ldots, X_k$ in the language where X_1 is smaller than X_2, X_2 is smaller than

X_3, and so on until X_k is smaller than X_1 (because the first mismatched character of X_1 is smaller than X_2, that of X_2 is smaller than X_3, and so on until that of X_k is smaller than X_1). If the graph does not contain a cycle, it must mean that we can order the words lexicographically.

mutant1.txt and corresponding graph

```
5 9
baa
abc
abca
cabe
cad
eba
ecada
dba
daae
```

topoOrder: [b, a, c, e, d]



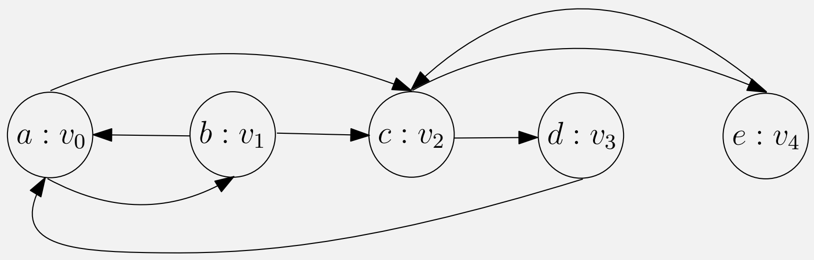
7.7 Example of an invalid language

In `mutant1.txt`, the graph is acyclic and we have a valid topological order. On the other hand, in `mutant2.txt`, if you choose the words *abcd*, *cab*, *daa*, and *abb*, you realize that there is no way to order these words in that order because that implies $a < c < d < a$. If you look at the corresponding graph, it has the cycle $a \to c \to d \to a$; there are other cycles as well such as $a \to b \to a$. As long as there is one cycle, the language is not valid.

mutant2.txt and corresponding graph

```
5 9
baa
abcd
cab
eba
eca
cba
daa
abb
bcd
```

Graph has a cycle



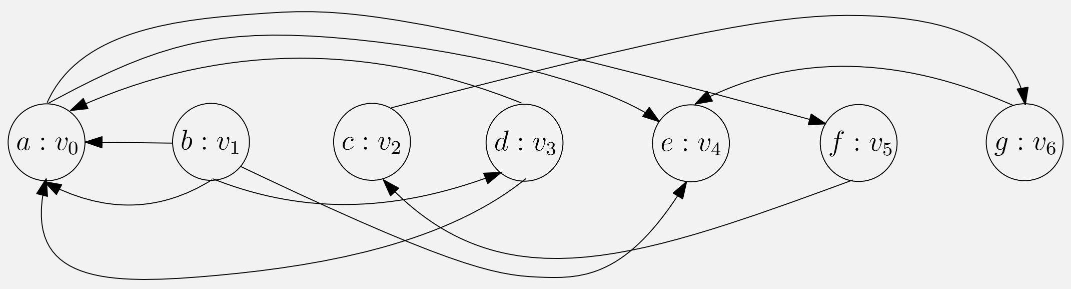
7.8 One final thing before we move to code

One thing you may have observed is that we can add multiple edges from one vertex to another vertex. For example in `mutant3.txt`, we add two edges from b to a. This is because the words *dbbb* and *dbac* add one edge, and the words *abcd* and *aade* add another edge. There is no reason to add both the edges, but trying to avoid that would mean that we scan the outgoing edges of b to make sure an edge to a is not already added (or use specialized structures such as a hashtable to maintain the outgoing edges). Having multiple edges, however, does not cause any harm (in the sense that it does not induce any new cycle) and both edges will be relaxed when we process the vertex b. So, once again, in the interest of keeping things simple, we add multiple edges.

mutant3.txt and corresponding graph

```
7 14
baa
dbbb
dbac
dbacd
abcd
aade
faab
feabc
cab
caef
gdfg
gagfr
ea
eae
```

topoOrder: [b, d, a, f, c, g, e]



7.9 Pseudo-code

Using the pseudo-code below, implement the `readLanguage` method in the file `MutantLanguage.h`/ `MutantLanguage.java`. You'll find similar syntax in `readUnweightedGraph` and `readWeightedGraph` methods in `Graph.h`/`Graph.java` file.

Reading mutant file

- **C++:** Create an input file stream *fileReader* on *filePath*. To create an input file stream on a file `x.txt`: **ifstream fileReader("x.txt", ios::in);**

  Java: Create a Scanner object *fileReader* on *filePath*. To create a scanner on a file `x.txt`: **Scanner fileReader = new Scanner(new FileInputStream("x.txt"));**

- Read the number of distinct characters into the class-variable *numVertices*.

  C++: Syntax to read a value into an integer variable x is: **fileReader ≫ x;**

  Java: Syntax to read a value into an integer variable x is: **x = fileReader.nextInt();**

- Now, read the number of words in the language into the class-variable *numWords*

- Allocate *numWords* cells for the class-array *words*. Note that *words* is a string array.

- Run a for loop from $i = 0$ to $i < numWords$. Within the for-loop, use *fileReader* to read the line into the array cell $words[i]$.

  C++: Syntax to read into a string variable y is: **fileReader ≫ y;**

  Java: Syntax to read into a string variable y is: **y = fileReader.next();**

- After the loop, close *fileReader*.

Notice that the first number on the file is read as the number of vertices in the graph, as desired. Now, we will create the graph based on the idea discussed previously. A detailed pseudo-code is given below to achieve the same. Use it to implement the `createGraph()` method in `MutantLanguage.h/MutantLanguage.java`. You'll find similar syntax in `readUnweightedGraph` and `readWeightedGraph` methods in `Graph.h/Graph.java` file.

Creating the graph

```
• Allocate $numVertices$ cells for $outDegree$ and $inDegree$ arrays.
• Allocate $numVertices$ rows for $adjList$.
  C++ syntax: adjList.reserve(numVertices);
  Java syntax: adjList = new ArrayList<ArrayList<Edge>>(numVertices);
• for ($i = 0$ to $i < numVertices$), do the following:
    – set $outDegree[i]$ to 0
    – add a blank row to $adjList$
      C++ syntax: adjList.push_back(vector<Edge>());
      Java syntax: adjList.add(new ArrayList<Edge>());
• for ($i = 0$ to $i < numWords - 1$), do the following:
    – let $currentWord$ be the word at index $i$ of $words[\,]$ array
    – let $nextWord$ be the word at index $i + 1$ of $words[\,]$ array
    – let $minLength$ be the minimum of the lengths of $currentWord$ and $nextWord$
      C++/Java syntax for length of a string named $str$: str.length()
    – for ($j = 0$ to $j < minLength$), do the following:
        ∗ let $x$ be the $j^{th}$ character of $currentWord$
          C++ syntax: char x = currentWord.at(j);
          Java syntax: char x = currentWord.charAt(j);
        ∗ let $y$ be the $j^{th}$ character of $nextWord$
        ∗ if ($x \neq y$), then:
            · let int $src = x - 97$
            · let int $dest = y - 97$
            · create an edge $e$ by calling the Edge constructor with arguments $src$ and
              $dest$ respectively
            · add the edge $e$ to the end of $adjList[src]$
              C++ syntax: adjList.at(src).push_back(e);
              Java syntax: adjList.get(src).add(e);
            · increment $outDegree[src]$ by 1
            · break;
```

Finally, we use topological sorting to find the order of the characters or detect that the graph has a cycle. A detailed pseudo-code is given in the next page to achieve the same. Use it to implement the `getOrder()` method in `MutantLanguage.h/MutantLanguage.java`.

7.10 Using ASCII to map character to vertex

One thing to note is the following statement: let **int** $src = x - 97$. What we are essentially doing here is using the ASCII value of the character and subtracting 97 to map it to its rank among the lower-case letters of the English alphabet. The ASCII value of a is 97, b is 98, and so on; see http://www.asciitable.com/ for the full ASCII table. Therefore, if we have the characters a, b, c, d, and e, then they are mapped respectively to $0, 1, 2, 3$, and 4 (corresponding to the vertices v_0, v_1, v_2, v_3, and v_4).

Likewise, once we get the vertex order (integer values), we transform them to the actual characters by adding 97 to get the ASCII value and typecasting to char.

Couple of things I would like to point out. Most of the code here is pretty similar to the longest paths method, and for the most part you should be able to just borrow the code from there. Also, note that at the end we return *null* when all vertices have not been added to the topological order, which implies that the graph has a cycle and an ordering of the characters (and hence words) cannot be obtained.

Finding character order

```
• Use a loop to initialize all cells of $inDegree[\,]$ to 0
• Create a char array $topoOrder[\,]$ having length $numVertices$.
  C++ programmers must use dynamic allocation. So, if you want to return a char array
  $x$ of length 10, it must be declared as char *x = new char[10];
• for ($i = 0$ to $i < numVertices$), do the following:
    – for ($j = 0$ to $j < outDegree[i]$), do the following:
        ∗ Let $adjEdge$ be the $j^{th}$ outgoing edge of vertex $i$
        ∗ increment $inDegree[adjEdge's\ destination]$ by 1
• Create an integer queue $vertexQ$.
• Initialize an integer variable $topoLevel = 0$
• for ($i = 0$ to $i < numVertices$)
    – if ($inDegree[i]$ equals 0), enqueue $i$
• while ($vertexQ's\ size > 0$), do the following:
    – let $v$ be the vertex obtained by dequeue-ing $vertexQ$
    – assign $topoOrder[topoLevel] =$ (char) $(v + 97)$
    – increment $topoLevel$ by 1
    – for ($j = 0$ to $j < outDegree[v]$), do the following:
        ∗ let $adjVertex$ be the destination of the $j^{th}$ outgoing edge of the vertex $v$
        ∗ decrement $inDegree[adjVertex]$ by 1
        ∗ if ($inDegree[adjVertex]$ equals 0), then enqueue $adjVertex$ to $vertexQ$
• if ($topoLevel \neq numVertices$), return $null$, else return $topoOrder$
```