Homework 5: Pattern Matching

CS 1332 Section C

Fall 2021

1 Important

There are general homework guidelines you must always follow. If you fail to follow any of the following guidelines, you risk receiving a **0** for the entire assignment

1. All submitted code must compile under **JDK 11**. This includes unused code, so don't submit extra files that don't compile. Any compile errors will result in a 0.
2. Do not include any package declarations in your classes.
3. Do not change any existing class headers, constructors, instance/global variables, or method signatures. For example, do not add **throws** to the method headers since they are not necessary.
4. Do not add additional public methods. You may create helper methods, but any helper method you create should be **private**.
5. Do not use anything that would trivialize the assignment. (e.g. don't import/use java.util.ArrayList for an ArrayList assignment. Ask if you are unsure.)
6. Always consider the efficiency of your code. Even if your method is *O(n)*, traversing the structure multiple times is considered inefficient unless that is absolutely required (and that case is very rare).
7. You must submit your source code - the .java files. Do not submit compiled code - the .class files.
8. Only the last submission will be graded. Make sure your last submission has all required files. Resubmitting voids prior submissions.

2 Style and Formatting

It is important that your code is not only functional, but written clearly and with good programming style. Your code will be checked against a style checker. The style checker is provided to you and it located on Canvas. A point is deducted for every style error that occurs. Please double check before you submit that your code is in the appropriate style so that you don't lose any unnecessary points!

2.1 Javadocs

Javadocs should be written for any private helper methods that you create. They should follow a style similar to the existing javadocs on the assignment. Any javadocs you write must be useful and describe the contract, parameters, and return value of the method. Random or useless javadocs added only to appease checkstyle will lose points.

2.2 Vulgar/Obscene Language

Any submission that contains profanity, vulgar, or obscene language will receive an automatic zero on the assignment. This policy applies to all aspects of your code, such as comments, variable names, and javadocs.

2.3 Exceptions

When throwing exceptions, you must include a message by passing in a String as a parameter. **The message must be useful and tell the user what went wrong.** "Error", "Oof - Bad things are happening", and "FAIL" are *not* good messages. Additionally, the name of an exception itself is not a good message.

In addition, you may not use try catch blocks to catch an excpetion unless you are catching an exception you have explicitly thrown yourself with the **throw new ExceptionName('Exception Message");** syntax

2.4 Generics

If available, use the generic type of the class; do not use the raw type of the class. For example, use new **LinkedList<Integer>()** instead of **new LinkedList()**. Using the raw type of the class will result in a penalty.

3 Forbidden Statements

You may not use any of the following in your code at any time in CS 1332. If you are not sure whether you can use something, and it is not explicitly listed here, just ask. Debug print statements are fine, but should be either removed or commented out prior to submission. If print statements are left in, assignments are messy to grade, and checkstyle points will be deducted.

- package
- System.arraycopy()
- clone()
- assert()
- Arrays class
- Thread class
- Collections class
- Collection.toArray()
- Refleciton APIs
- Inner or nested classes
- Lambda Expressions
- Method References (using the :: operator to obtain a reference to a method)

4 JUnits

We have provided a **very basic** set of tests for your code. These tests do not guarantee the correctness of your code, nor do they guarantee you any grade. You may additionally post your own set of tests for others to use on the Georgia Tech GitHub as a gist. Do **NOT** post your tests on the public GitHub.

5 Deliverables

You must submit all of the following file(s) to the corresponding assignment on Gradescope. Make sure all file(s) listed below are in each submission, as only the last submission is graded. Make sure the filename(s) matches the filename(s) below, and that *only* the following file(s) are present.

1. PatternMatching.java

5.1 Graph Algorithms

For this assignment, you will code 3 different graph algorithms. This homework has many files included, so be sure to read ALL of the documentation given before asking questions.

5.1.1 Graph Data Structure

You are provided a Graph class. The important methods to note from this class are:

- getVertices returns a Set of Vertex objects (another class provided to you) associated with a graph.
- getEdges returns a Set of Edge objects (another class provided to you) associated with a graph.
- getAdjList returns a Map that maps Vertex objects to Lists of VertexDistance objects. This Map is especially important for traversing the graph, as it will efficiently provide you the edges adjacent to any vertex (the outgoing edges of any vertex). For example, consider an adjacency list where vertex A is associated with a list that includes a VertexDistance object with vertex B and distance 2 and another VertexDistance object with vertex C and distance 3. This implies that in this graph, there is an edge from vertex A to vertex B of weight 2 and another edge from vertex A to vertex C of weight 3.

5.1.2 Vertex Distance Data Structure

In the Graph class and Dijkstra's algorithm, you will be using the VertexDistance class implementation that we have provided. In the Graph class, this data structure is used by the adjacency list to represent which vertices a vertex is connected to. In Dijkstra's algorithm, you should use this data structure along with a PriorityQueue. When utilizing VertexDistance in this algorithm, the vertex attribute should represent the destination vertex and the distance attribute should represent the minimum cumulative path cost from the source vertex to the destination vertex.

5.1.3 DFS

Depth-First Search is a search algorithm that visits vertices in a depth based order. Similar to pre/post/inorder traversal in BSTs, it depends on a Stack-like behavior to work. In your implementation, the Stack will be the recursive stack, meaning you should not create a Stack data structure. It searches along one path of vertices from the start vertex and backtracks once it hits a dead end or a visited vertex until it finds another path to continue along. **Your implementation of DFS must be recursive to receive credit.**

5.1.4 Single-Source Shortest Path (Dijkstra's Algorithm)

The next algorithm is Dijkstra's Algorithm. This algorithm finds the shortest path from one vertex to all of the other vertices in the graph. This algorithm only works for non-negative edge weights, so you may assume all edge weights for this algorithm will be non-negative. In order to keep track of the cumulative distance from the source vertex to the vertices you visit in this algorithm, you will need to use the VertexDistance data structure we are providing you. At any stage throughout the algorithm, the PriorityQueue of VertexDistance objects will tell you which vertex currently has the minimum cumulative distance from the source vertex.

There are two commonly implemented terminating condition variants for Dijkstra's Algorithm. The first variant is where you depend purely on the PriorityQueue to determine when to terminate. You only terminate once the PriorityQueue is empty. The other variant, the classic variant, is the version where you maintain

both a PriorityQueue and a visited set. To terminate, still check if the PriorityQueue is empty, but you can also terminate early once all the vertices are in the visited set. You should implement the classic variant for this assignment. The classic variant, while using more memory, is usually more time efficient since there is an extra condition that could allow it to terminate early.

5.1.5 Self-Loops and Parallel Edges

In this framework, self-loops and parallel edges work as you would expect. If you recall, self-loops are edges from a vertex to itself. Parallel edges are multiple edges with the same orientation between two vertices. In other words, parallel edges are edges that are incident on precisely the same vertices. These cases are valid test cases, and you should expect them to be tested. However, most implementations of these algorithms handle these cases automatically, so you shouldn't have to worry too much about them when implementing the algorithms.

5.1.6 Prim's Algorithm

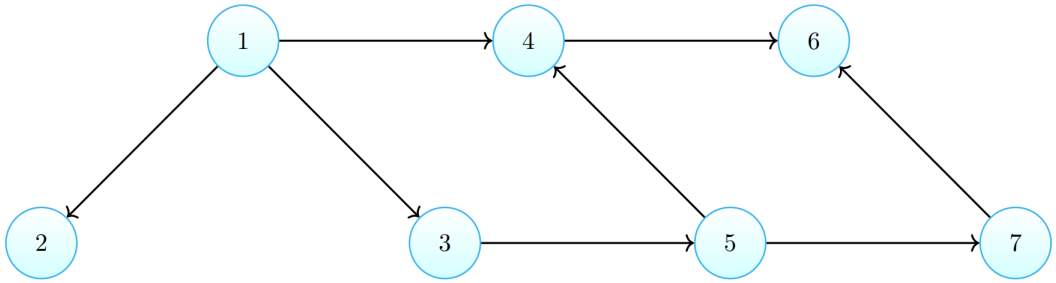
A tree is a graph that is acyclic and connected. A spanning tree is a subgraph that contains all the vertices of the original graph and is a tree. An MST has two main qualities: being minimum and a spanning tree. Being minimum dictates that the spanning tree's sum of edge weights must be minimized.

By the properties of a spanning tree, any valid MST must have $|V| - 1$ edges in it. However, since all undirected edges are specified as two directional edges, a valid MST for your implementation will have $2(|V| - 1)$ edges in it.

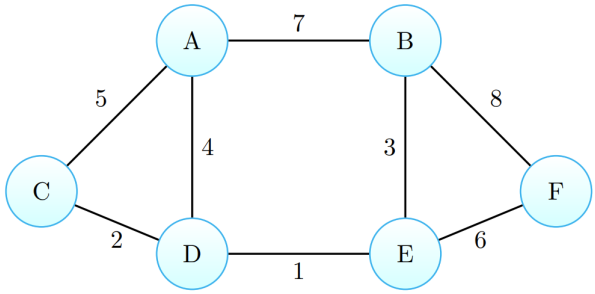
Prim's algorithm builds the MST outward from a single component, starting with a starting vertex. At each step, the algorithm adds the cheapest edge connected to the incomplete MST that does not cause a cycle. Cycle detection can be handled with a visited set like in Dijkstra's.

5.1.7 Visualizations of Graphs

The directed graph used in the student tests is:



The undirected graph used in the student tests is:



6 Grading

Here is the grading breakdown for the assignment. There are various deductions not listed that are incurred when breaking the rules listed in the PDF and in other various circumstances. Note that for this assignment efficiency is explicitly mentioned as contributing to your point total. In contrast to past assignments, unimplemented methods will result in losing efficiency points.

- Graph Algorithms – 80 pts
- Efficiency – 10 pts
- Checkstyle – 10 pts
- Total: 100 pts