Program 4

**Important information about this assignment:**

* **Due date**: Sunday 12/4 at 10 pm
* **Submission**: Please push your submission to Github
* **Collaboration**: This is a pair programming assignment, all design, development, and debugging must be done synchronously with your partner.

# Learning Objectives

1. Implement the Dijkstra’s shortest path algorithm.
2. Apply the shortest path concept to real-world accessibility.
3. Understand data querying and cleaning.

# Overview

[Project Sidewalk](https://dl.acm.org/doi/10.1145/3290605.3300292) is an open-source web-based [tool](https://sidewalk-sea.cs.washington.edu/) designed to collect street-level accessibility data through crowdsourcing, involving both volunteers and paid workers. Its primary function is to enable users to annotate “street segments” with valuable sidewalk accessibility information, including details about curb ramps, missing or present, obstacles, surface problems, and the absence of sidewalks.

To facilitate easy access to the collected dataset, Project Sidewalk provides an [API](https://sidewalk-sea.cs.washington.edu/api). This allows users to retrieve the dataset, which consists of Accessibility Scores for streets within a specified region. These scores are determined based on a comprehensive scoring model that takes into account the number of accessibility issues present. The scores range from 0 to 1, where 0 indicates inadequate accessibility and 1 denotes full accessibility. In the dataset, street segments are represented as a [Feature Collection](https://datatracker.ietf.org/doc/html/rfc7946#section-3.3) of [LineString features](https://datatracker.ietf.org/doc/html/rfc7946#appendix-A.2), with each LineString having an array of coordinate positions, such as:

{

"type": "LineString",

"coordinates": [

[100.0, 0.0],

[101.0, 1.0]

]

}

For easier data manipulation, the dataset has been extracted into a CSV file, which includes columns such as “origin”, “destination”, and “distance” (refer to SidewalkAccessibilityScore.csv in the provided skeleton code).

In this project, your task is to implement Dijkstra’s Single-Source Shortest Paths algorithm using a Graph class provided to you. Specifically, you will be working on the ShortestPaths class, where you will write methods to execute Dijkstra’s algorithm, reconstruct the shortest paths once the algorithm has run, and develop a command-line program capable of computing the shortest paths on a graph read from a test file.

The main method in the ShortestPaths class has the ability to read directed graphs from two types of text files: “basic” and “db”. When a “db” file is provided, the DBParser class will parse a Graph from a CSV file containing the actual coordinates of sidewalks (origin and destination) along with their corresponding accessibility scores (distance). In this context, a shortest path indicates the path with the highest number of accessibility barriers that require attention and should be addressed.

Overall, Project Sidewalk serves as a powerful platform for collecting and utilizing street-level accessibility data, while your contribution involves implementing essential algorithms and functionalities to enhance its capabilities.

# Setup

To begin on this project, please follow these steps:

1. Click on the Github Classroom invitation link for this assignment: <https://classroom.github.com/a/IxOjJTOB>
2. Accept the invitation. You will be prompted by Github to create or join a team.
3. Work with your partner to decide which one of you will create the team, and which one will join. Only one of you needs to create the team, and the other student should join the team.
4. Choose a team name that follows the format of “student1LastName\_student2LastName” (in no particular order).

Once you have created your team and joined it, clone a local working copy of your repository as you have done in previous assignments. It is important to clone it somewhere outside of your previous cloned repositories (e.g., ~/cs241/program3) to avoid nesting local repositories. Your repository contains skeleton code to help you get started.

Also, please keep in mind that this is a pair programming assignment, and all design, development, and debugging must be done synchronously with your partner. This means that you should work together throughout the project, discussing ideas and making decisions together. By doing so, you will be able to learn from each other and produce a higher quality project.

If you have any questions or issues during this process, please don’t hesitate to ask for assistance.

# Dijkstra’s Shortest Path Algorithm

The abstract version of Dijkstra’s algorithm is as follows:

/\* Compute shortest paths to all nodes from origin nodes \*/

shortest\_paths(v);

S = {};

F = (v);

v.d = 0;

v.bp = null;

while (F != {}) {

f = node in F with min d value;

Remove f from F, add it so S;

for each neighbor w of f {

if (w not in S or F) {

w.d = f.d + weight(f, w);

w.bp = f;

add w to F;

} else if (f.d + weight(f, w) < w.d) {

w.d = f.d + weight(f, w);

w.bp = f;

}

}

}

Your implementation should follow this abstract algorithm as closely as possible, but the graph representation won’t match the pseudocode exactly because we have to deal with practical implementation considerations. The following design decisions have been made for you:

* So that we can efficiently find the node in F with minimum d value, the Frontier set is stored in a min-heap, using d-values as priorities. Because a Node’s d-value may change, you will need to use the Heap’s changePriority method to keep the priorities updated. **heap.java** is given to you in the skeleton code repo.
* Instead of the Node class having a field for *d* and *bp*, we store these things separately. The ShortestPaths class maintains a Map that associates each Node with a PathData object, which stores the distance and backpointer for a node.
* The Settled set does not need to be explicitly stored. If a Node has a PathData object associated with it, it is in either the Settled (not in the heap) or Frontier set (in the heap); otherwise it is in the Unexplored set.

Begin by carefully reading and understanding the /\* Javadoc comments \*/ in Graph.java, Node.java, and ShortestPaths.java. Read over the code in these files as well. When you’re done you should be able to answer questions such as:

1. What is the purpose of each of the following HashMaps?
   * Graph’s nodes field
   * Node’s neighbors field
   * ShortestPath’s paths field
2. Where is the Graph’s adjacency list stored, how would you iterate over all edges leaving a given Node, and how would you get the weight of each edge?
3. What are types of the Values (V) and Priorities (P) in the min-heap storing F?
4. For a given Node object, where are n.d and n.bp stored, and how would you access them?

If you can’t answer these questions, you’re unlikely to make much progress on the code.

# Software Requirements

The main method behavior is specified in the descriptions below and in comments associated with each TODO item. In brief, the program takes 3 or 4 command-line arguments. The first two specify the file type (basic or db) and the filename where the directed graph data is stored. The third is an origin node id, from which shortest paths should be computed. If no fourth argument is supplied, the program should print all reachable nodes and their shortest path distances. If the fourth argument is supplied, it gives a destination node, and the program should print in order the nodes along the path from the origin to the destination, followed by the length of the path.

**TODO 1.0.** Implement the **compute** method in the *ShortestPaths* class according to its specification.

**TODO 2.0.** Implement the **shortestPathLength** method. Notice that this method’s precondition states that **compute** has been called with the desired origin node, so the **paths** field should already be filled in with the final shortest paths data.

**TODO 3.0.** Implement the **shortestPath** method. Once again, **compute** has already been called with the desired origin so you only need to use the data stored in **paths** to reconstruct the path.

**TODO 4.0.** In the **main** method, create and use an instance of **shortestPaths** to compute shortest-paths data from the origin node specified in the command line arguments.

**TODO 5.0.** If a destination node was not specified on the command line, print a table of reachable nodes and their shortest path lengths.

**TODO 6.0.** If a destination node was specified on the command line, print the nodes in the shortest path from the origin to the destination, followed by the length of the path.

# Sample Output

You are given Simple0.txt in the skeleton code repo. Two sample invocations of the program are given below:

$ gradle run --args "basic Simple0.txt A"

Graph has:

3 nodes.

3 edges.

Average degree 1.0

Shortest paths from A:

B: 1.0

C: 2.0

A: 0.0

$ gradle run --args "basic Simple0.txt A C"

Graph has:

3 nodes.

3 edges.

Average degree 1.0

A C 2.0

# Notes

Your grade will be partly based on unit tests, but this time around you have not been provided with these tests. **It is your responsibility to test your implementation and be certain that the algorithm is implemented correctly**. ShortestPathsTest.java contains a placehold test case to get you started writing your own tests. You will lose a few points if you do not write at least a few unit tests in ShortestPathsTest.java, although we will not be grading your test cases for correctness or comprehensiveness.

Some testing advice:

* You should test your code both using JUnit test cases and the command line program implemented in the main method.
* Three simple graphs (Simple0.txt, Simple1.txt, Simple2.txt) are provided. Run the algorithm by hand to determine the correct answers for these graphs and verify that your implementation arrives at the correct paths and path lengths.
* **The sample graphs given are not sufficient to test your algorithm’s correctness**. It’s your responsibility to write tests that cover all possible cases that the algorithm could encounter.
* Make sure your algorithm handles edge cases correctly, including behaving as specified when the destination node is unreachable. Test this using the simplest possible test cases.
  + For example, this edge case could be tested using a two-node graph with only an edge from destination to origin.
* The *BasicParser* class parses a simple edge list from a text file, such as Simple1.txt and Simple2.txt. The DBParser class parses a csv file, such as DBCrop.csv. Feel free to write and test using additional graph files in these formats. You may get more csv files from the Sidewalk API. You provide lat/lng bounds for the area that you want to query.
* Some cleaning will be needed to such csv files.

1. Split the coordinates in the “coordinates” column to extract “origin” and “destination”. Consider the first coordinate as the “origin”, the second coordinate as the “destination”, and ignore any extra coordinates.
2. Rename the column headers accordingly. The below steps will be useful:
   1. Select the cell or column that contains the text you want to split
   2. Select Data > Text to Columns
   3. In the Convert Text to Columns Wizard, select Delimited > Next
   4. Select the Delimiters for your data. Use ‘)’ as a delimiter
   5. Select Next
   6. Select the Destination in your worksheet which is where you want the split data to appear
3. Select the accessibility attribute you want to consider as the “distance” for your shortest path algorithm.
4. Rename your selected column as “distance”. In the csv sample file given to you, “distance” is considered to be the access score.

# Project Plan and Reflection

Each person must write both a project plan named plan.txt and a reflection named reflection.txt. plan.txt must be completed before the start of the project and reflection.txt must be completed before submitting your project.

plan.txt should include the following information:

1. A one paragraph summary of the program in your own words. What is being asked of you? What will you implement in this assignment?
2. In 2-3 sentences, explain your thoughts on what you anticipate being the most challenging aspect(s) of the assignment.
3. A proposed schedule for when you will work on this assignment with your partner and where you will meet. (e.g., Mondays, Tuesdays, and Wednesdays from 5-7pm in the lab and Thursdays from 9am-2pm over Zoom).
4. A list of at least three different resources you plan to use if you get stuck on something.

reflection.txt should include the following information:

1. Declare/discuss any aspects of your code that are not working. What are your intuitions about why they are not working? Acknowledge and discuss any parts of the program that appear to be inefficient.
2. What are some of the most important lessons you learned while working on this assignment? Why do you think so?
3. What was the most challenging aspect of this assignment? Why?