CSCI 3901: Project: Milestone 3

Waleed R. Alhindi (B00919848)

Program Components:

- PowerService.java: This class serves as the core of the program. It aggregates data about postal codes and distribution hubs provided by the database interface class (Database.java). Using that data, it provides the reporting methods such as fixOrder, mostDamagedPostalCodes, and rateOfRestoration. Furthermore, this class accepts data that needs to be added into the database (i.e., new hubs, new postal codes, updating hub repair estimates, etc.). Before passing that data to the database class, it performs the necessary validation and formatting.

- Database.java: This class constitutes a database interface class that acts an intermediary between PowerService.java and the MySQL database. Thus, it is meant to isolate any database connectivity and SQL operations from the rest of the program.

- DamagedPostalCodes.java: A class to encapsulate the information of individual postal codes. In other words, each postal code in the database (or ones that have been added during execution) are stored in an object of this class.

- HubImpact.java: A class that encapsulates the information of individual distribution hubs. In other words, each distribution hub in the database (or hubs that have been added during execution) are stored in an object of this class.

- Point.java: A class that stores the coordinates of a distribution hub as (x, y) coordinates. That is to say, each HubImpact object contains an object of this class that represents its (x, y) coordinate locations. Note that the constructor of this class is: Point(int x, int y).

- credentials.prop: A properties file to store a user's MySQL username, password, and the database they intend to use (i.e., csci3901).

Note: It is highly likely that an addition class will be added to facilitate and isolate operations concerning PowerService's repairPlan method.

Assumptions:

The project instructions granted two important assumptions that affected how this program is being implemented: overlapping postal codes will not be provided; and postal codes shall not be deleted. Thus, this program does not validate whether postal codes being added overlap with an existing one nor does it provide any methods that allow for the deletion of postal codes. However, although this program does not check whether postal codes overlap, it does check whether they are identical (see design decision 3, 4, and 5 for more details).

Additionally, this implementation assumes that the user can access the Dalhousie database server (i.e., they have a valid CSID username, password, and database). Furthermore, this implementation assumes that a varchar(100) MySQL datatype is sufficient for postalCode, hubIdentifier, and emp_id data supplied by the user.

Design Decisions:

During the implementation of this program, numerous design decision needed to be made; especially when encountering cases that were ambiguous. As such, the following are the design decisions made thus far:

1. All error states are handled by throwing the appropriate exception. An appropriate exception in this case refers to a specific type of exception that is supplied with an error message conveying as much information regarding the error as possible. For example, calling hubDamage with a hubIdentifier of null will throw an IllegalArgumentException, rather than just a generic exception, supplied with the message "HubIdentifier is null! \nSource: hubDamage \nDetails: e.getMessage()".

2. There are two exceptions to design decision (1) where error states are handled by return values rather than exceptions: addPostalCode and addDistributionHub. These two methods are the only methods that return a boolean value. As such, it would make sense to return true or false rather than throwing an exception. Personally, I would have designed these method to not return boolean values, and instead throw exceptions since error messages could be provided to convey what went wrong. However, given the PowerService interface specified in the project instructions, these two method will return false when encountering an error state.

3. A postal code String passed to addPostalCode that corresponds to an existing postal code will be considered invalid. Note that this does not mean that they overlap, rather it means they are the same postal code. For example, adding "B3J" when postal code "B3J" already exists is invalid, while adding "B3J1H6" in the same scenario is valid. The latter is valid since the method does not take into account overlapping postal codes since the project instructions grant the assumption that they will not be provided.

4. A postal code String passed to addPostalCode must meet the Canadian postal code formatting; which consist of a letter then alternates between numbers and letters. For example, "B3J" will be accepted as a valid postal code, but not "3BJ" since it neither starts with a letter nor alternates between letters and numbers.

5. All postal code and hub identifier Strings passed throughout the program will be converted to uppercase. For example, passing "b3j" will be converted to "B3J". Furthermore, all postal codes and distribution hubs passed will have their spaces removed. For example, "b 3 j " will be converted to "B3J". As such, all postal codes are normalized to a standard format to ensure data consistency. Consequently, this means that

adding the postal code “b 3 J” when “B3J” already exists will be invalid. Similarly, passing “A e 3” as a hubIdentifier to hubDamage when “AE3” exists will update hub “AE3”.

6. A distribution hub can exist when it services no postal codes and vice versa. That is to say, adding a distribution hub where Set<String> servicedAreas is empty is valid, and that adding a postalCode that is not serviced by any existing hubs is also valid.

7. A distribution hub can service a postal code that has not been added. However, if that postal code is added later on, then the relation between the two is update. For example, postal code “B3J” is added, then hub “A1” is added with its servicedAreas including “B3J” and “A2H”. This is valid and only the relation between hub “A1” and postal code “B3J” is established. However, if the postal code “A2H” is added later on, then a connection between hub “A1” and “A2H” is then established.

8. A distribution hub cannot share locations. In other words, no two distribution hubs can have the same x *and* y coordinate values as part of their Point location. As such, adding a distribution hub with a location that an existing hub already resides in will be considered an error state.

9. A hub’s repairEstimate that is passed to hubDamage cannot be less than or equal to zero. This is because a negative repairEstimate does not make any sense and a repairEstimate of zero would suggest that there is no damage to be reported.

10. Similarly to design decision (8), passing a negative repairTime to hubRepair is considered as an error state. However, passing a repairTime of zero is not invalid. This is because a zero repairTime might indicate that an employee was simply surveying a hub or that a hub has been misidentified as being damaged.

11. Calling hubRepair on a hub that is already in service is invalid. This is because it makes no sense repairing a hub that is not damaged. Rather, if it is damaged and has not been reported yet, then the employee should call the hubDamage method before calling hubRepair.

12. If a repairTime greater than or equal to a hub’s repairEstimate is passed alongside a inService value of false, then the hub’s repairEstimate will not be changed. This is because a hub’s repairEstimate cannot be negative and because there is no way of readjusting its repairEstimate in a way that accurately reflects the misestimation in its repairEstimate. For example, if a hub’s repairEstimate is 10 and hubRepair is called with a repairTime of 15 and an inService of false. Then that hub’s repairEstimate will remain 10 since it cannot be -5 and because there isn’t enough information to readjust the repairEstimate to reflect how much it has been misestimated (i.e., how much the initial repairEstimate should have been).

13. Building upon design decision (11), if an inService of true is passed to hubRepair, then that hub's repairEstimate and impact are reverted to 0 regardless of the values of repairTime and repairEstimate.

14. If the number of people out of service is not a whole number, then it will be rounded up. For example, if the number of people out of service is 10.5, then calling peopleOutOfService will return 11. Such cases arise when a postal code is serviced by multiple hubs. For example, a postal code with a population of 10 that is serviced by 3 hubs, where 2 are damaged, will effectively have 6.667 people without power.

15. MostDamagedPostalCodes, fixOrder, underservedPostalByArea, and underservedPostalByPopulation will handle ties at the limit by adding them to the returned list in a manner that is similar to assignment 3. In other words, any entities passed the limit that tie with the entity at the limit will be added to the returned list.

16. The increment floating point value passed to rateOfRestoration must be greater than 0 and less than or equal to 1. Any values outside this range are considered invalid. This is because an increment of 0 would produce and infinite list and a negative increment is logically impossible as it would be asking for a negative population. On the other hand, an increment of over 1 would be invalid because it asking for more than the population we have information about.

17. In addition to design decision (16), an increment value of less than 0.001 (0.1%) is considered invalid since it would constitute a very fine degree of granularity which would result in a very large list, and would result in the inverse amount of granularity on the number of hours in that returned list.

18. underservedPostalByArea and underservedPostalByPopulation will disregard any postal codes that are not being serviced by any hubs. This is because such postal codes do not provide a point of reference to compare against. For example, if two postal codes are not serviced by any hubs, then both their hubs per capita would be 0 regardless of their actual population, meaning we cannot identify which is more underserviced. Thus, such postal codes are disregarded.
    Note: upon further feedback, this design decision will be changed to include postal codes that are not serviced by any hubs.

19. rateOfServiceRestoration will not take into account any postal codes that are not serviced by any hubs. This is because doing so would lead to an infinite list since the people in service will never reach 100% even if all the damaged hubs were repaired.

20. If rateOfServiceRestoration is called when no hubs exist, then it will throw an exception. This is because such a case would lead to an infinite sequence of zeroes to populate the returned list.

21. The program calculates a hub's impact by dividing the total amount of people effected by a hub's outage by its repairEstimate.

Note: since repairPlan is still being implemented, its design decision will be finalized and documented later on once more progress is made.

Data Structures:

Since the program relies heavily on a MySQL database to store and organize data and their relationships, it only utilizes three major data structures:

- A <String, DamagedPostalCodes> map in the PowerService class that aggregates all existing and newly added postal codes. The String key is a DamagedPostalCode's postal code and its value is the corresponding DamagedPostalCode object. This data structure reduces the amount of SQL operation needed since it is populated with the database's existing postal codes during the PowerService constructor.

- A <String, HubImpact> map in the PowerService class that aggregates all existing and newly added distribution hubs. The String key is a hub's identifier and its value is the correspond HubImpact object. Similarly, to the above data structure, this map is populated with the database's existing hub data during the PowerService constructor to reduce SQL fetch operations.

- A Set of Strings in each HubImpact object that stores the postal codes it services. Unlike the other two data structures, this set is not meant to reduce SQL operations. Rather, it is meant to facilitate the fetching of information concerning each of a hub's serviced areas. For example, when a hub is restored, this set will indicate which specific postal codes need to be updated in the database (and in the DamagedPostalCodes objects) to reflect this restoration.

Note: Data structures that relate to repairPlan have been omitted from this documents since the method has not been finalized and since additional data structures will likely be added while repairPlan's implementation progresses.

Database Design:

The database used to facilitate this program's operations consists of four tables (see **Figure 1)**:

- PostalCodes:
    - id: varchar(100) not null [primary key]
    - population: int
    - area: int

- DistributionHubs:
    - Id: varchar(100) not null [primary key]
    - x (i.e., hub location's x coordinate): int

- y (i.e., hub location's y coordinate): int
- repairTime: float
- inService: Boolean

- PostalHubRelation (*bridge table to facilitate many-to-many relationship between PostalCodes and DistributionHubs tables*):
  - postalId: varchar(100) not null
  - hubId: varchar(100) not null [foreign key referencing DistributionHubs(id)]
  - primary key (postalId, hubId)

- RepairLog:
  - Id: int auto-increment not null [primary surrogate key]
  - Emp_id: varchar(100) not null
  - hubId varchar(100) not null [foreign key referecing DistributionHubs(id)]
  - repairTime: float
  - hubRestored (i.e., whether this repair restored the hub): boolean

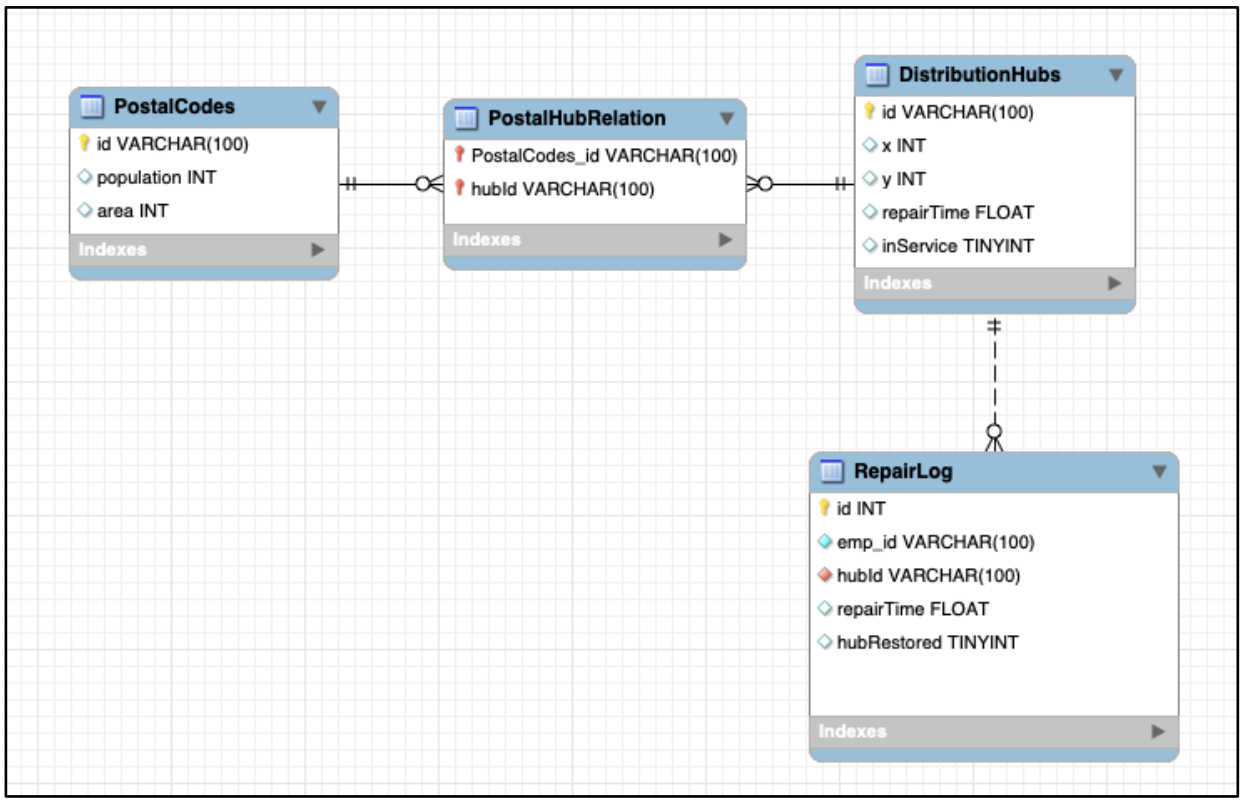

Figure 1: ERD diagram of database's tables

Key Algorithms:

Adding a Postal Code

This algorithm is rather simple. It first checks if the arguments are valid, then checks if the postal code is formatted correctly and that it does not already exist in the PowerService object's postalCodes map. Note that this map remains up to date with the database's data since it is populated with the database's existing data when the object is instantiated and is updated alongside any database operations. If the arguments supplied pass the validation checks, then the method simply created a new DamagedPostalCode object which it passes to the database interface object. The database interface object takes care of inserting it into the MySQL database. Then, we pass the object to another database interface method that updates the bridge table that relates postal codes and hubs. After which, the database interface object calculates the new postal code's repairEstimate, which the method then uses to set the new DamagedPostalCode object's repairEstimate value. Finally, if the insertion and updates were successful, the PowerService object's postalCodes map is updated to reflect the insertion.

Perform all necessary validations (i.e., postalCode is not null, population is not negative, etc.)
    Return false if any of the validations failed
If the postalCode already exists as a key in the PowerService object's postalCodes map
    Return false
Create a new DamagedPostalCodes object using the arguments supplied
Try
    Pass the new DamagedPostalCodes object to the interface database object's "addPostalCodeToDB" method
Catch any SQLExceptions
    Return false
Try
    For each entry in the distributionHubs map
        If that HubImpact object's servicedAreas contains the new postal code
            Pass the new postal code and this hub to the database interface object's "updatePostalHubRelation" method
    Use the database interface object to calculate the postal code's repairEstimate (this algorithm is detailed below)
Catch any SQLExceptions
    Return false
Add the new DamagedPostalCode object to the postalCodes map
Return true

Calculating a Postal Code's Repair Estimate

This algorithm is part of the Database class and is used throughout various PowerService methods. It cross references the PostalHubRelation and DistributionHubs tables to calculate the total repairEstimate of a postal code.

Note this algorithm is supplied with a postalCode String as an argument

Create a totalRepairTime variable and set it to 0

Try

    Connecting to the database

    Selecting all the columns from PostalHubRelation joined with DistributionHubs on hubId where PostalHubRelation's postalId field equals this postalCode

    For each of the rows returned

        If the row's "inService" field is false

            Increment the totalRepairTime by the row's "repairTime" field value

Catch any SQLExceptions

    Throw the exception back to the calling method (since this method is called as part of PowerService methods, which handle the exceptions)

Return the totalRepairTime

Adding a Distribution Hub

Similarly to the algorithm for adding postal codes, this algorithm is rather simple since it basically just validates the arguments passed then passes them to the database interface object to insert and update the relevant tables.

Perform all necessary validations (i.e., hubIdentifier is not null, servicedAreas is not null, etc.)

    Return false if any of them failed

If the hubIdentifier already exists as a key in the distributionHubs map

    Return false

For each entry in the distributionHubs map

    If this HubImpact's Point object's coordinates equal the coordinates of the hub being added

    Return false

Create a new HubImpact object using the arguments supplied

Try

    Pass the new HubImpact object to the database interface object's "addDistributionHubToDB" method

Catch any SQLExceptions

    Return false

Try

    For each postalCode in the servicedAreas set argument

        Pass this postalCode and the newly added hub to the database interface object's "updatePostalHubRelation" method

Catch any SQLExceptions

    Return false

Add the new HubImpact object to the distributionHubs map

Return true

Updating a Hub's Damage

This algorithm updates a hub's repairTime value by passing the relavent arguments to the database interface object, which then updates the DistributionHubs table to reflect the damage being reported. After which, each of the postal codes being serviced by the hub have their repairEstimate values modified to reflect the damage being reported. Finally, the hub's impact and populationEffected values are recalculated to reflect the damage being reported.

Perform validation checks (i.e., hubIdentifier is null, repairEstimate is negative, etc.)

    Throw an IllegalArguementException if any of them failed

If the hubIdentifer does *not* exist as a key in the distributionHubs map

    Throw an IllegalArgumentException

Get the HubImpact object from the distributionHubs map that corresponds to the hubIdentifier

Increment that HubImpact object's repairTime value by the repairEstimate argument's value

Set that HubImpact object's inService value to false

Try

    Pass the hubIdentifier and repairEstimate to the database interface object's "updateHubDamage" method

Catch any SQLExceptions

    Throw that SQLException

Try

    For each of Strings in this HubImpact's servicedAreas set

        Use the database interface object to calculate that postalCode's updated repairEstimate

        Set this new repairEstimate as the corresponding DamagedPostalCode object's repairEstimate value

    Calculate the number of people affected by this hub outage by passing the hubIdentifier and its corresponding servicedAreas set to the database interface object's "calculatePopulationEffect" method

    Set the calculated value as this HubImpact's populationEffected value

Set this HubImpact's impact value as (populationEffected / repairTime)

Catch any SQLExceptions

Throw that SQLException

Repairing a Hub

This algorithm creates a new row in the RepairLog table to log this hub repair. It then updates the DistributionHubs table to reflect the repair and updates the corresponding HubImpact objects as well. It then updates each serviced postal code's DamagedPostalCode object and updates the PostalCodes table to reflect changes in repairTime due to this repair.

Perform validation checks (i.e., hubIdentifier is null, repairTime is negative, etc.)

Throw an IllegalArguementException if any of them failed

If the distributionHubs map does not contain hubIdentifier as a key

Throw an IllegalArguementException

If this hub's inService value is true

Throw an IllegalArgumentException

Try

Pass the relevant argument to the database interface object's "updateRepairLog" method that will insert a row into the RepairLog table to reflect this repair

Catch any SQLExceptions

Throw that SQLException

If the inService argument is true

Set this hub's repairEstimate to 0

Set this hub's impact to 0

Set this hub's populationEffected to 0

Set this hub's inService value to 0

Try

Update the DistributionHubs table to reflect this restoration by passing the relevant argument to the database interface object's "applyHubRepairToDB" method

Catch any SQLExceptions

Throw that SQLException

Try

Update each postal code being serviced by this hub's repairTime in the PostalCodes table by passing the relevant argument to the database interface object's "calculatePostalRepairTime" method

Catch any SQLExceptions

Throw that SQLException

Else

If repairTime is less than this hub's repairEstimate

Set this hub's repairEstimate to (repairEstimate – repairTime)

Try

Update the DistributionHubs table to reflect this restoration by passing the relevant argument to the database interface object's "applyHubRepairToDB" method

Catch any SQLExceptions

Throw that SQLException

Try

Update each postal code being serviced by this hub's repairTime in the PostalCodes table by passing the relevant argument to the database interface object's "calculatePostalRepairTime" method

Catch any SQLExceptions

Throw that SQLException

Calculating the Total Number of People Out of Service

This algorithm calculates the total number of people out of service by passing each postal code in the postalCodes map to the database interface object, which cross references the PostalHubRelations and DistributionHubs tables to find out how many people are out of service in a postal code.

Create a totalPeopleOutOfService variable and set it to 0

Try

For each postal code in the postalCodes map

Calculate the number of people out of service in this postal code by passing the postal code to the database interface object's "postalPopulationOutOfService" method (this algorithm is detailed below)

Increment totalPeopleOutOfService by the calculated value

Catch any SQLExceptions

Throw that SQLException

Round up totalPeopleOutOfService (i.e., if the value is 10.5, then it is rounded to 11)

Return totalPeopleOutOfService

Calculating the Number of People Out of Service in a Postal Code

This algorithm is part of the Database class and is used in PowerService's peopleOutOfService method. It calculates the number of people out of service in a particular postal code by joining the PostalHubRelation and DistributionHubs tables, then multiplying the postal code's population by the fraction of (damaged hubs/total hubs) that service it.

Note this algorithm is supplied with a postalCode String as an argument

Create a totalHubs variable and set it to 0

Create a damagedHubs variable and set it to 0

Try

    Connect to the database

    Select all fields from PostalHubRelation join DistributionHubs on hubId where postalId equals the postalCode argument

    For each of the rows returned

        Increment totalHubs by 1

        If the row's "inService" field is false

            Increment damagedHubs by 1

Catch any SQLExceptions

    Throw the exception back to the calling method (since this method is called as part of PowerService methods, which handle the exceptions)

If totalHubs is 0

    Return 0

Calculate the fraction of downed hubs as (damagedHubs / totalHubs)

Return that calculated fraction

Identifying the Most Damaged Postal Codes

Since the postalCodes map stores each postal code's corresponding DamagedPostalCodes objects, this algorithm simply adds any postal codes with a repair time that is greater than 1 to a temporary map, where the key is the postal code String and its value is that postal code's repair time. Then, that map is sorted by its values in descending order. Now, the algorithm simply adds entries of the map to the list of most damaged postal codes until it reaches the limit or until it has added all the damaged postal codes (in the case where limit is greater than the number of damaged postal codes). Note, however, that this algorithm will add any postal codes passed the limit that tie with the postal code at the limit (similarly to assignment 3).

If limit is less than 1

    Throw an IllegalArgumentException

Create a postalRepairTimes <String, Float> map to store each postal code's repair time

For each DamagedPostalCode in the postalCodes map

If that postal code's repairTime > 0

Add that postal code and its repair time as a key-value pair into the postalRepairTimes map

Sort the postalRepairTimes by its values in descending order by passing it to a sortByValue helper method

Create a list of DamagedPostalCodes called mostDamagedPostalCodes

Create a counter variable and set it to 0

Create a valueAtLimit variable and set it to -1

For each entry in the postalRepairTimes map

If counter is less than limit – 1

Add this postal code to the mostDamagedPostalCodes list

Increment counter by 1

Else if counter equals limit – 1

Add this postal code to the mostDamagedPostalCodes list

Increment counter by 1

Set valueAtLimit to this postal code's repair time

Else

If this postal code's repair time equals valueAtLimit

Add this postal code to the mostDamagedPostalCodes list

Else

Break the loop (stop adding postal codes to mostDamagedPostalCodes)

Return mostDamagedPostalCodes list

Identifying the Most Impactful Hubs (fixOrder)

Similarly to the previous algorithm, since each hub's HubImpact object is stored in the distributionHubs map, this algorithm adds each damaged hub to a temporary map that is then sorted by its value in descending order. Then, the algorithm adds entries of that map to the fixOrder list until the limit is reached or all damaged hubs are added. Note, however, that this algorithm will add any postal codes passed the limit that tie with the postal code at the limit (similarly to assignment 3).

If limit is less than 1

Throw an IllegalArguementException

Create a <String, Float> map called hubImpacts to store each hub's impact

For each hub in the distributionHubs map

If this hub's inService value is false

Add this hub's identifier and impact as key-pair values into the hubImpacts map

Sort the hubImpacts map by its values in descending order by passing it to a helper method

Create a new list called fixOrder
Create a counter variable and set it to 0
Create a valueAtLimit variable and set it to -1
For each hub in the hubImpacts map
    If counter is less than limit – 1
        Add the HubImpact object that corresponds to this hub to the fixOrder list
        Increment counter by 1
    Else if counter equals limit – 1
        Add the HubImpact object that corresponds to this hub to the fixOrder list
        Set valueAtLimit to this hub's impact value
        Increment counter by 1
    Else
        If valueAtLimit equals this hub's impact value
            Add the HubImpact object that corresponds to this hub to the fixOrder list
        Else
            Break the loop (stop adding hubs to the fixOrder list)
Return fixOrder

Identifying the Postal Codes that Are Underserved Based on Population

This algorithm uses the database interface class to find how many hubs each postal code is serviced by, then divides the postal code's population by that number. It then adds that value (capitaPerHub) to a temporary <String postalCode, Float capitaPerHub> map which is sorted by its values in descending order. Then, the postal codes in that map are added to the underservedByPopulation list until it reaches the limit or all postal codes are added. Note, however, that this algorithm will add any postal codes passed the limit that tie with the postal code at the limit (similarly to assignment 3). Also note that this algorithm disregards any postal codes that are not serviced by any hubs (see design decision () for more details).

If limit is less than 1
    Throw an IllegalArguementException
Create a postalPopHubs <String, Float> map to store each postal code's capitaPerHub
For each DamagedPostalCode in the postalCodes map
    Try
        Get the number of hubs that service that postal code by passing the postal code String to the database interface object's "getNumberOfPostalHubs" method
        If the number of hubs that service this postal code is 0
            Continue the loop (skip this postal code)
        Divide this postal code's population by the number of hubs servicing it

Add the postal code and the value calculated (capitaPerHub) to the postalPopHubs map

Catch any SQLExceptions

Throw that SQLException

Sort the postalPopHubs map by its values in descending order by passing it to a helper method

Create a counter variable and set it to 0

Create a valueAtLimit variable and set it to -1

Create a underservedPostals list

For each entry in the postalPopHubs map

If counter is less than limit – 1

Add this postal code to the underservedPostals list

Increment counter by 1

Else if counter equals limit – 1

Add this postal code to the underservedPostals list

Increment counter by 1

Set valueAtLimit to this postal code's capita per hub value

Else

If this postal code's capita per hub value equals valueAtLimit

Add this postal code to the underservedPostals list

Else

Break the loop (stop adding postal codes to the list)

Return underservedPostals

Identifying the Postal Codes that Are Underserved Based on Area

This algorithm uses the database interface class to find how many hubs each postal code is serviced by, then divides the postal code's area by that number. It then adds that value (meterPerHub) to a temporary <String postalCode, Float meterPerHub> map which is sorted by its values in descending order. Then, the postal codes in that map are added to the underservedByPopulation list until it reaches the limit or all postal codes are added. Note, however, that this algorithm will add any postal codes passed the limit that tie with the postal code at the limit (similarly to assignment 3). Also note that this algorithm disregards any postal codes that are not serviced by any hubs (see design decision () for more details).

If limit is less than 1

Throw an IllegalArguementException

Create a postalAreaHubs <String, Float> map to store each postal code's meterPerHub

For each DamagedPostalCode in the postalCodes map

Try

Get the number of hubs that service that postal code by passing the postal code String to the database interface object's "getNumberOfPostalHubs" method

If the number of hubs that service this postal code is 0

Continue the loop (skip this postal code)

Divide this postal code's area by the number of hubs servicing it

Add the postal code and the value calculated (meterPerHub) to the postalPopHubs map

Catch any SQLExceptions

Throw that SQLException

Sort the postalAreaHubs map by its values in descending order by passing it to a helper method

Create a counter variable and set it to 0

Create a valueAtLimit variable and set it to -1

Create a underservedPostals list

For each entry in the postalPopHubs map

If counter is less than limit – 1

Add this postal code to the underservedPostals list

Increment counter by 1

Else if counter equals limit – 1

Add this postal code to the underservedPostals list

Increment counter by 1

Set valueAtLimit to this postal code's capita per hub value

Else

If this postal code's capita per hub value equals valueAtLimit

Add this postal code to the underservedPostals list

Else

Break the loop (stop adding postal codes to the list)

Return underservedPostals

Calculating the Rate of Service Restoration

This algorithm makes use of the fixOrder method and peopleOutOfService method to calculate the percentage of people in service after each hub in the fixOrder list is restored. Using this information, it populates a the rateOfServiceRestoration list with times that reflect the incremental percentage of people that will be in service as hubs are restored.

If increment is less than or equal to 0, or if increment is greater than 1

    Throw an IllegalArgumentException

If increment is less than 0.001

    Throw an IllegalArguementException (granularity issue)

If the distributionHubs map is empty (i.e., no hubs exist)

    Throw an IllegalArguementException

Create a list called rateOfRestoration

Get a list of the most impactful hubs called fixOrder by calling the fixOrder method

Get a variable called peopleOutOfService that stores the number of people out of service by calling the peopleOutOfService method

Get a variable called totalPopulation that stores the total number of people by calling a helper method

Calculate the current percentage of the population in service by subtracting poepleOutOfService from totalPopulation and store that value as percentageOfInServicePop

Divide percentageOfInServicePop by increment and store that value as alreadyHavePowerIncrement

Add a alreadyHavePowerIncrement amount of 0s to the rateOfRestoration list

Create a currIncrement variable and set it to alreadyHavePowerIncrement

Create a repairTimeElapsed variable and set it to 0

For each HubImpact in the fixOrder list

    Incremebt repairTimeElapsed by this HubImpact's repairEstimate value

    Increment percentageOfInServicePop by this HubImpact's populationEffected value

    While percentageOfInServicePop is less than currIncrement

        If currIncrement equals 1

            Add repairTimeElapsed to the rateOfRestoration list

            Break the loop (stop adding to rateOfRestoration)

        Add repairTimeElapsed to the rateOfRestoration list

        Increment currIncrement by increment

Return rateOfRestoration

Identifying an Optimal Repair Plan

This algorithm has not been finalized as the repairPlan method is still being implemented.

Milestone 2 Reflection (Progress in Relation to Earlier Plan):

Regarding the timeline submitted in milestone 2, my current progress is ahead of schedule in the case of all methods excluding repairPlan. I believe this is due to my approach of offloading as much operations as possible to the MySQL database. Thus, PowerSerivce's methods, excluding repairPlan, are relatively simple and took less time to implement that I previously estimated. Initially, I did not think I could offload as many operations as I did onto the database, which is why I think my estimates were longer than what I needed to implement these methods. However, on the other hand, I am slightly behind repairPlan's schedule. Initially, I planned to have repairPlan fully implemented by December 7~8. Although I have made good progress in its implementation, my new estimate of when it will be fully implemented is December 10~11. Upon reflection, I believe this misestimation is due to two factors: I underestimated repairPlan's complexity; and I did not accurately estimate the time I needed to fulfill other obligations (for example, I estimated that I could finish an assignement for CSCI 5100 in 2 days. But, it took me about 3.5 days to complete, which ate away at the time I had initially allocated to work on repairPlan).