Project 2 Report

Carlyn Thomsen, Reevs Cota, Kay Harris, Vi Nguyen

Project Description

This project is designed to read in, store and process data points along with k values and their associated centroids. While processing the data points, we cluster them using k-means, centroid-linkage and k-medoids clustering algorithms. Finally, we calculate the Dunn-Index and output an analysis for each clustering.

Piece 1: Reading and Storing Files (Carlyn Thomsen)

For this portion of the project, I asked the user to input the filename of the configuration file. If a file name doesn't exist, the user is notified of the issue and can input another name for the file. Then, the function begins to read in the file, using ifstream. I chose to read in the first lines of the configuration individually in order to accurately assign values to the correct variables. To do this, I used getline, and then created a substring of that getline in order to only get the necessary characters that represented the specific variable. I used this method to read in the kmin and the kmax.

Using the variable for the points filename that was read from the configuration file, I read in the first instance of each variable for the ID, X Position, and Y Position. Then, I created a while loop to run until it reached the end of the points file. Inside of the while loop, it adds the ID, X Position, and Y Position to a new point object and then adds that point to the vector of point object. It then adds the point to the map of Point objects. Lastly, the while loop continues to read in the next set of variables.

Then the constructor continued to read in the rest of the configuration file using ifstream. I created another while loop to run until it reached the end of the configuration file. Inside the while loop, it accounts for the ":" after the numbers using a substring. After removing the ":", I used the number that was read in to enter a for-loop. Inside the for-loop, the k-point ID's are read in and the point at that ID is made a cluster and added to the list. Then, outside of the for-loop, but inside of the while loop, the list of clusters is added to kclusters. Lastly, the while loop reads in the next number.

I then had get functions for kmin, kmax, the vector of kpoints, and the vector of Point objects. This allowed my group members to have easy access to all variable created and stored in partOne.

The Points class consisted of an integer variable for the ID, float variables for both the X Position and Y Position, and then get() and set() functions for each variable to allow my group members easy access to all variables created.

Piece 2: K-Means Clustering (Vi Nguyen)

For this portion of the project we created a main function **kMeans** to perform k-means clustering on the points Carlyn had stored. This function took a vector points, which stored all points, and a vector of lists of type Cluster, which stored clusters created from k and centroids from config text file, as parameters. For example, if the first k value is 2 with centroids *a* and *b*, the first list will store two clusters with centroids *a, b* and we can get k values easily using the list's size. When we looped through the vector of lists, a local list of type Cluster **clusters** was used to copy the values from each vector element and **cluster** was cleared out before moving on to the next element. We also chose list to store all clusters to match with part 3. Then we created a while loop, and inside this loop, we repeated 2 steps:

1. Assign points to the closest clusters
2. Update the centroid to the average of all points within a cluster

We also created a list **temp** that holds all clusters from the previous change, and if **temp** was similar to all current clusters, we broke the loop.

From the provided k-means algorithm, we decided it was efficient to have three other functions: **closeClus** to find the closest cluster to a point, **assignPoints** to assign points to all cluster and **closePoint** to find the closest point to the centroid of a cluster; we could update the centroid easily by calling **updateCent** function from class Cluster. **closeClus** returned an iterator to the cluster within the list, and this cluster was found by comparing the distance from the point to each centroid to find the smallest distance. **assignPoints** first cleared out vector points of all clusters and then assigned new points to vector points of closest clusters, which was found by calling **closeClus**. Since we were required to assign points to clusters after updating their centroids, we could either check which point in vector of points would stay the same and which point would change cluster; or we could just clear out those vector of points and assign every points again. It was faster to loop through all clusters than loop through points, thus we decided to go with the latter. In addition,

closePoint returned a point within a cluster, and this point was found by comparing the distance of the centroid to each point in this cluster.

For bonus, we were required to use one of the cluster' points as centroid instead. Since k-medoid was very similar to k-means clustering, we decided that we just need to add one more step to the previous 2 steps:

3. Update centroid to the closest point to average of all points

We also added a new parameter **int x** to the previous **kMeans** function. **X** would simply tell the function to include step 3 or not; when x was 0, **kMeans** only performed the first 2 steps (k-means), otherwise, **kMeans** performed step 3 by calling the function **closePoint** (k-medoids). We then printed associated outputs.

Piece 3: Centroid-Linkage Hierarchical Clustering (Kay Harris)

Firstly, I created a Cluster class for use in k-mean, centroid-linkage, and Dunn index. By having everyone use the same class it helped prevent redundant code and avoid conversions between different parts. The class contains the three members each part would need: id, centroid, and vector of Points. The getter **getPoints** was returned as a reference as typical getters and setters would have led to a lot of vector copying. Function **getCentroid** was done similarly because there was rarely any need to make a copy of it, so a reference was fine. Some convenience methods were added into the class as well. Function **updateCent** is used in all three clusterings, and calculates the average position of all the Points and sets it as the centroid. Function **getDistance** calculates the Euclidean distance between two Points, while **getCentDistance** conveniently passes two centroids into it. Lastly, Vi added an **operator==** so that her code could compare Clusters. In my own centroid-linkage part I created a Line struct, which essentially holds the distance between two Clusters plus related data (like who those Clusters are, the sum of their ids, and this Line's multiset iterator location, because trying to do erasures without it can lead to multiple deletions). This struct made it far easier to sort and work with the Clusters. It has two functions: **recalculate**, which recalculates the distance between the centroids after a change, and an **operator<**, which is used for sorting.

The whole centroid-linkage algorithm is in one large **cluster** function as there weren't any parts I felt needed their own functions. To start, each Point is turned into a Cluster by using it as the Cluster's initial Point/centroid. A list (linked-list) was preferred because Clusters have to be removed with each merge. A vector would have had to move Clusters around each time, and so didn't make sense in this part. When writing my code, I decided it would be faster to store and reuse distances between Clusters, especially since points4.txt would have over 100,000 distances to calculate at the start. To manage and sort the Lines in each level, **ordered**, a multiset of Line*, is created. The multiset is setup to order the pointers by distance (or id in the case of ties) so that sorting steps aren't needed and distances don't need to be recalculated. Additionally, **connect**, an array of set of Line*, holds all the Lines connected to each Cluster (where the position in the array correlates to Cluster id). This extra data makes it easier to clean and update **ordered** after a merge is done. Because of **connect**, pointers had to be used. Trying to find and update every copy would have required a lot of extra work, and iterators would occasionally be de-referenced due to merges, thus leaving pointers as the best option. Once **ordered** and **connect** have been filled, the actual clustering can start. A for loop performs centroid-clustering until only one Cluster is left. To start, the output required every 10 levels is printed. Then, the shortest Line can simply be grabbed from the beginning of **ordered**. From there we can extract the two Clusters and copy the Points from **merge** into **keep**. Now that **keep** has its new Points, it uses **updateCent** to recalculate its new centroid. Now that the merge is complete, all references to **merge** need to be removed and all Lines connected to **keep** need to be recalculated. The next loop does the first part of that by going through every Line connected to **merge** and deleting it from **ordered** and the other Cluster's **connect**. This removes all references to **merge** in **ordered** and **connect**, including the current shortest Line. As such, all of **merge**'s connected Lines can be erased, **merge** can be removed from the list of Clusters, and the shortest Line can be deleted. The recalculation step is similar, with a loop going through every Line connected to **keep**, removing it from **ordered**, updating its value, and putting it back in **ordered** for sorting. From there, the algorithm can move on to subsequent merges. Once done, a final message is printed.

Piece 4: Dunn's Index (Kay Harris)

After talking with Vi about how to handle communication between parts 2 and 3 with the Dunn index code, it was decided that a public function **dunn** would take a list of Cluster and output the Dunn index as a double. This list would be all the Clusters to be considered when doing the calculation. However, an int **type** is also required so that the Dunn index code can easily tell different which algorithms are which and keep their data separate. Two private arrays in the Dunn class, **kvals** and **dunns**, store the highest Dunn index and respective k value for each **type**. They default to being filled with zeros, which is perfect since any possible Dunn index would be greater or equal to zero. The primary **dunn** function begins by creating two doubles: **max**, and **min**. The value **max** holds the greatest inter-cluster distance found while **min** holds the smallest intra-cluster distance found. As such, to avoid needing a special first case, **max** and **min** needed to be set to values that would be immediately replaced by the ensuing loops. Because I couldn't guarantee that all test data would be on the same scale as the files we were provided with, I played it safe by setting **min** to the largest possible double value. While **max** could have just been 0 (since all distances are going to be positive) I decided I wanted it to match **min**, and thus made it the smallest possible double value. Now that we don't need to worry about their initial values, we can move on to the Dunn index calculation. For inter-cluster distance we'll need to compare each Point in the same Cluster, and for intra-cluster distance we'll need to compare every Point that isn't in the same Cluster. From this, I concluded that a single loop over all Clusters, with different inner loops for each distance calculation, would be the best way to avoid redundancy. At the start of the loop I first verify that each Cluster has more than one Point in it because the instructions say to. After that, I can then compare all the Points in this Cluster to find inter-cluster distance. The first for loop goes through all Points in the Cluster while the second for loop goes through all Points after this one. This is to ensure two Points aren't compared twice or to themselves. With these two Points, distance can be calculated using the same Euclidean distance code everything else uses before being compared to the current **max** value. With inter-cluster distance done, now we can do intra-cluster distance. Like before, we iterate over every Cluster after the current one to ensure two Clusters aren't compared twice or to themselves. With these two Clusters, two more loops are needed to go over all the Points in each one. This gives us one Point from each Cluster, and, like before, distance is calculated and compared to **min**. After all of this occurs, we have our final inter-cluster and intra-cluster distances, and the Dunn index for this list of Clusters can be calculated. As a final step, this Dunn index is compared to previous runs of the current **type**, and, if this Dunn index is greater, it is saved to be output during the analysis. A second public function, **printAnalysis**, prints out the expected output before the program ends by using the greatest Dunn index found for each.