IE 599 – Introduction to Data Mining:

Project 2
Agglomerative Hierarchical Clustering

Benjamin Fields

931-707-381

November 22, 2016

Introduction:

Previously we explored clustering algorithms through the K means and Bisecting K-means algorithms, but there are many more clustering techniques that can be used. Agglomerative Hierarchal Clustering is a second category of clustering techniques and is the focus of this study. In the context of hierarchal clustering there are two general approaches. The first is Agglomerative, which is when the algorithm is started with the points as individual clusters and, at each step, merge the closest pair of clusters. The second approach is Divisive, which starts the algorithm with one, all-inclusive cluster and, at each step, split a cluster until only singleton clusters of individual points remain (Tan, Steinbach, & Kumar, 2005). In order to visualize the results of these algorithms graphical tools called dendrograms and nested cluster diagrams are most often used. In the case of this study, a tree like structure is used that is most related to the dendrogram. Figure 1 below shows an example of a dendrogram and nested cluster diagram.

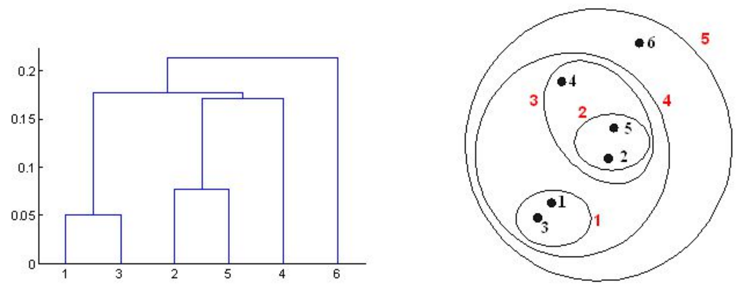

Figure 1: Example of a dendrogram and nested cluster diagram

Most agglomerative hierarchal algorithms that are implemented are based on a core approach that can be represented with the following pseudocode:

1. Compute the proximity matrix, if necessary
2. **Repeat**
3. Merge the closest two clusters
4. Update the proximity matrix to reflect the proximity between the new cluster and the original clusters
5. **Until** Only one cluster remains

(Tan, Steinbach, & Kumar, 2005)

One of the key operations of the algorithm is the computation of proximity between two clusters. This involves first defining the distance between each point in the beginning and then updating these distances as each cluster is merged. In this study this step is completed using the Euclidian distance and the group average calculation. The group average calculation is completed by taking the sum of all pairwise distances between points in each cluster and dividing by the product of the cardinality of each cluster. The details of this calculation are explained in further detail in later sections. In the implementation of this algorithm the C++ programming language was used.

Datasets:

In this study two data sets were to be selected for use in testing the implementation of the BAHC (Basic Agglomerative Hierarchical Clustering) algorithm that contained a dimensionality of at least two. The first data set selected contains data relating the crime rates of cities in the year 1970. The set has a dimensionality of 7 but for the purposes of this study only 3 were used and run through the BAHC algorithm. The three selected were “Burglary”, “Larceny” and “Auto”.

"City Crime Rates Per 100,000, Hartigan page 28"
8 columns
16 rows

"City"	"Murder"	"Rape"	"Robbery"	"Assault"	"Burglary"	"Larceny"	"Auto"
"Atlanta"	16.5	24.8	106	147	1112	905	494
"Boston"	4.2	13.3	122	90	982	669	954
"Chicago"	11.6	24.7	340	242	808	609	645
"Dallas"	18.1	34.2	184	293	1668	901	602
"Denver"	6.9	41.5	173	191	1534	1368	780
"Detroit"	13.0	35.7	477	220	1566	1183	788
"Hartford"	2.5	8.8	68	103	1017	724	468
"Honolulu"	3.6	12.7	42	28	1457	1102	637
"Houston"	16.8	26.6	289	186	1509	787	697
"Kansas City"	10.8	43.2	255	226	1494	955	765
"Los Angeles"	9.7	51.8	286	355	1902	1386	862
"New Orleans"	10.3	39.7	266	283	1056	1036	776
"New York"	9.4	19.4	522	267	1674	1392	848
"Portland"	5.0	23.0	157	144	1530	1281	488
"Tucson"	5.1	22.9	85	148	1206	756	483
"Washington"	12.5	27.6	524	217	1496	1003	739

Figure 2: Original data set for crime rates in cities in the year 1970.

The second data set used in this study contains information pertaining to the moons in the solar system and the relation to each orbiting planet. The information captured contains the Distance in thousands of miles between the moon and planet, the diameter of the moon in miles, and the Period of the orbit measured in days.

"Planets and Moons, Hartigan page 122"
4 columns
31 rows

"Planet #"	"Distance"	"Diameter"	"Period"
"Earth 1"	239	2160	655
"Mars 1"	5.8	10.0	7.7
"Mars 2"	14.6	10.0	30.0
"Jupiter 1"	112	100	12.0
"Jupiter 2"	262	2020	42
"Jupiter 3"	417	1790	85
"Jupiter 4"	665	3120	172
"Jupiter 5"	1171	2770	401
"Jupiter 6"	7133	50	6014

Figure 3: Sub set of the original data set on solar system moons.

In order to get an initial understanding of the data being used, each set was placed into excel to produce scatter plots of the initial data. From this a preliminary analysis is conducted to extract any preexisting relationships. These plots can be seen below.

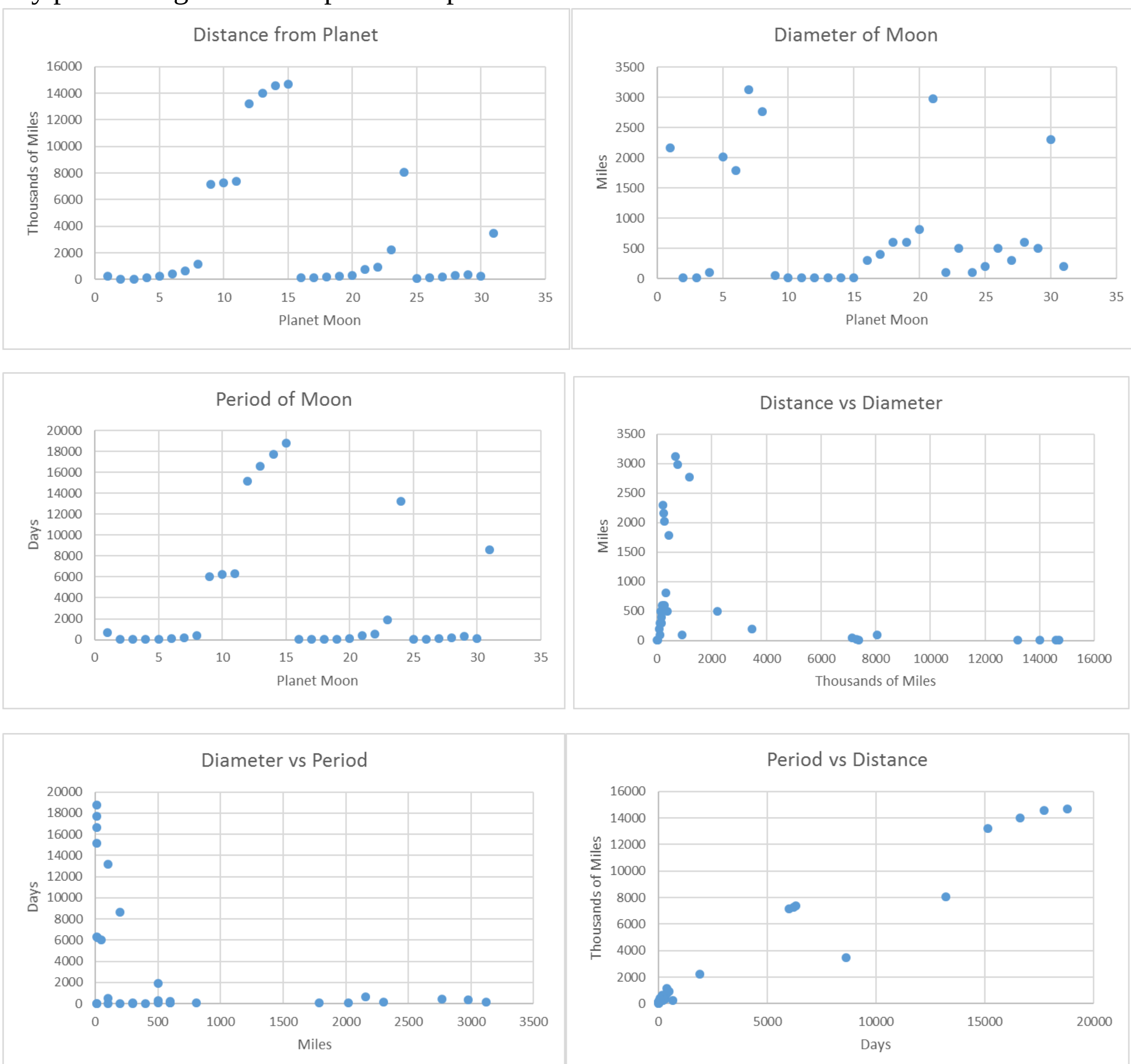

Figure 4: Scatter Plots of Moon data

From these plots, we can see that there may exist linear relationships in the case of Period vs Distance and the other plots show areas that could be identified as clusters. A similar approach was used to plot the relationships in the data for the crime rates data set. These plots can be seen below.

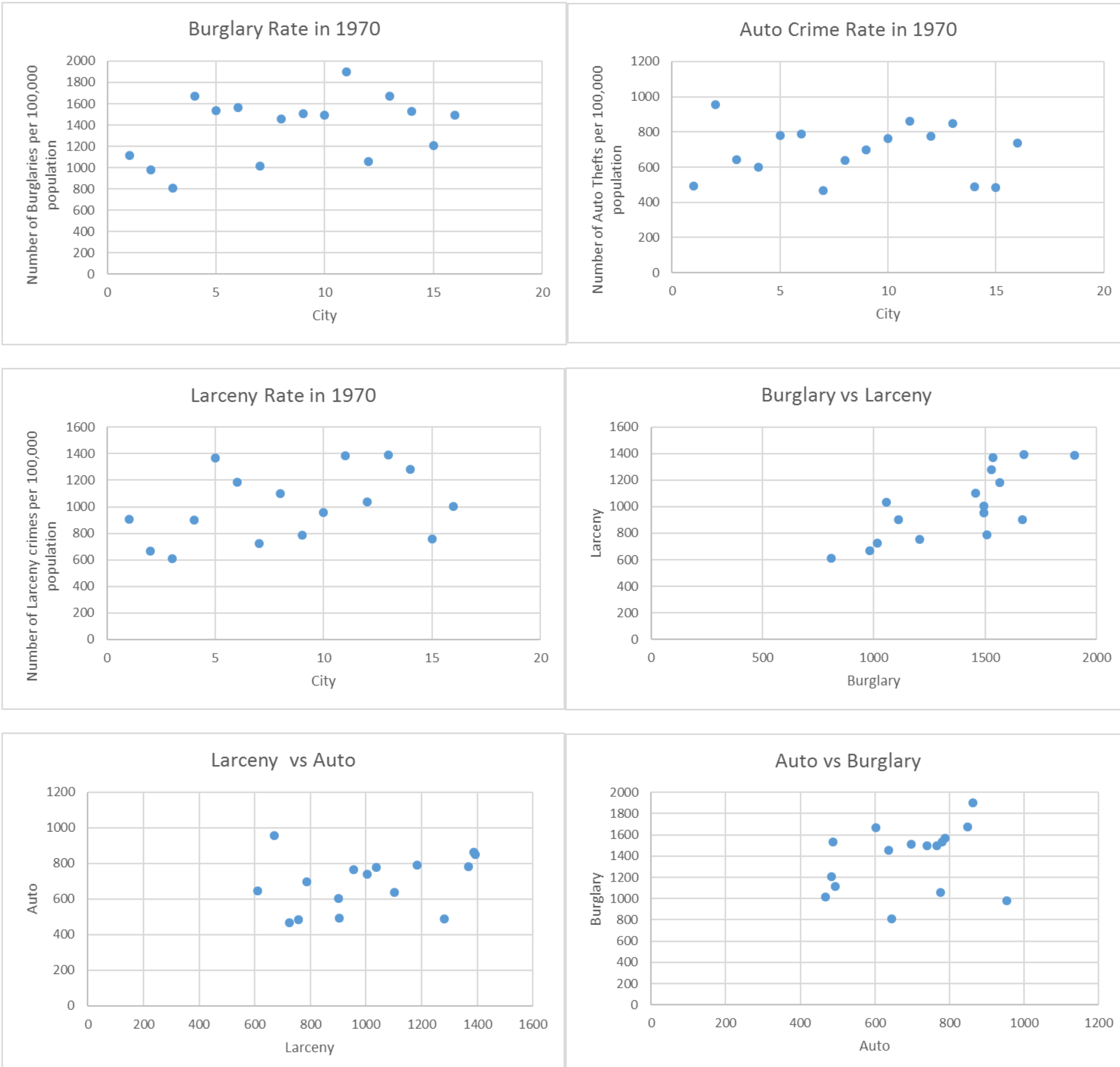

Figure 5: Scatter plots of crime data

Within the crime rates data set we can see that the data does not seem to contain clear separation between each attribute, but the clustering algorithm can still be applied to extract clustering information pertaining to each combined data point.

Algorithm Implementation:

The BAHC algorithm contains some key steps that control how each iteration is completed and ultimately what results are obtained. One of the first steps key to completing an Agglomerative Hierarchical clustering study is creating a proximity matrix for the initial data set. In this study this was achieved by applying the Euclidian distance equation to each data point. This equation can be seen below.

$$\text{Euclidian distance } d(x,y) = \left(\sum_{n=0}^{n} |x_k - y_k|^r\right)^{1/r}$$

Where r = 2

Once the proximity matrix is completed the algorithm can proceed to the next critical step, which is selecting the closest clusters to merge. The proximity matrices obtained with this program can be seen in figure 6 and 7.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	0	533.1	450.37	566.406	688.657	608.141	206.063	422.236	461.24	471.025	995.432	315.945	823.607	562.26	176.516	465.924
2	533.1	0	359.662	805.186	907.514	795.492	490.353	716.661	598.082	616.166	1170.02	414.547	1006.39	944.46	528.759	649.598
3	450.37	359.662	0	909.238	1058.95	961.504	297.044	815.055	725.113	777.632	1359.28	510.876	1185.01	998.758	454.155	798.383
4	566.406	805.186	909.238	0	517.425	352.88	687.812	293.508	217.49	244.461	597.981	650.419	549.211	420.048	498.628	242.398
5	688.657	907.514	1058.95	517.425	0	187.918	882.819	311.663	587.431	415.204	377.455	582	157.48	304.711	755.207	369.256
6	608.141	795.492	961.504	352.88	187.918	0	783.889	203.084	410.3	240.202	399.476	530.898	242.786	317.648	636.36	199.251
7	206.063	490.353	297.044	687.812	882.819	783.889	0	604.19	546.328	607.535	1173.33	440.147	1011.08	757.508	192.276	617.028
8	422.236	716.661	815.055	293.508	311.663	203.084	604.19	0	324.852	198.399	573.852	429.509	419.178	244.072	454.349	147.397
9	461.24	598.082	725.113	217.49	587.431	410.3	546.328	324.852	0	181.86	735.17	522.925	645.02	536.804	372.245	220.429
10	471.025	616.166	777.632	244.461	415.204	240.202	607.535	198.399	181.86	0	601.36	445.563	479.852	429.303	449.521	54.626
11	995.432	1170.02	1359.28	597.981	377.455	399.476	1173.33	573.852	735.17	601.36	0	919.572	228.508	537.852	1012.4	571.537
12	315.945	414.547	510.876	650.419	582	530.898	440.147	429.509	522.925	445.563	919.572	0	716.829	606.337	432.145	442.784
13	823.607	1006.39	1185.01	549.211	157.48	242.786	1011.08	419.178	645.02	479.852	228.508	716.829	0	403.308	869.911	441.459
14	562.26	944.46	998.758	420.048	304.711	317.648	757.508	244.072	536.804	429.303	537.852	606.337	403.308	0	616.949	376.086
15	176.516	528.759	454.155	498.628	755.207	636.36	192.276	454.349	372.245	449.521	1012.4	432.145	869.911	616.949	0	458.961
16	465.924	649.598	798.383	242.398	369.256	199.251	617.028	147.397	220.429	54.626	571.537	442.784	441.459	376.086	458.961	0

Figure 6: Proximity Matrix for crime rates data

The proximity measure between clusters can be completed in a few different ways that consist of a single link, complete link and average approach. The single link uses a shortest edge calculation, while the complete link uses the furthest edge approach. The group average approach combines these two and takes into account every pairwise distance between each point in each cluster and then divides the sum by the product of each cluster's cardinality. This can be represented by the equation below.

$$\frac{1}{|\mathcal{A}| \cdot |\mathcal{B}|} \sum_{x \in \mathcal{A}} \sum_{y \in \mathcal{B}} d(x, y)$$

When implementing this in code. A clustering data structure was created and applied in order to contain each clustering within each iteration. This clustering was then evaluated based on each cluster and the closest clusters were then selected to be merged. Once selected the clusters could be joined and the copied cluster deleted. For this study the dimensionality of 3 was coded into each calculation where needed.

As each iteration completes the program creates Merge objects that contain all the information related to the recent iteration and the state of the clustering. Each merge is tracked throughput the program and eventually exported to a text file as well as a .csv file for further analysis. While in both cases Detailed information is gathered relating to the merge conditions and values, a graphical tree like representation was generated for the crime rate data and not the solar moon data as the solar moon data set contained significantly more data points. The results can be seen in the following section.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
1	0	2257.41	2250.22	2161.75	629.204	702.484	1156.01	1142.47	8983.22	9244.98	9359.93	19560.1	21186.5	22417.7	23295.8	1968.29	1868.89	1675.96	1667.49	1459.26	1008.36	2174.42	2865.71	14931.7	2062.4	1767.49	1942.41	1622.84	1697.56	533.064	8818.15
2	2257.41	0	23.9735	139.273	2026.55	1828.51	3183.34	3021.59	9320.63	9585.12	9701.8	20081.2	21721.1	22961.1	23848.9	310.609	415.886	617.164	635.635	867.888	3086.92	1047.46	2950.9	15462.6	204.6	505.618	343.923	677.853	684.499	2303.86	9287.07
3	2250.22	23.9735	0	133.831	2025.2	1825.75	3180.45	3015.38	9299.54	9563.95	9680.61	20058.6	21698.4	22938.3	23825.9	307.296	412.195	613.746	630.85	862.368	3082.15	1029.18	2930.01	15439	200.024	501.896	334.547	668.129	669.872	2301.87	9263.1
4	2161.75	139.273	133.831	0	1926.08	1718.85	3074.38	2898.57	9236.93	9502.12	9619.05	20008.5	21649.7	22890.6	23780.4	200.342	302.881	506.093	517.731	748.025	2975	949.666	2855.5	15403.7	108.208	402.931	225.078	560.722	566.33	2206.42	9242.66
5	629.204	2026.55	2025.2	1926.08	0	280.667	1178.69	1231.93	9314.31	9580.15	9696.68	19989.1	21624.1	22861	23749.9	1726.29	1624.03	1422.2	1420.46	1213.54	1133.53	2083.1	3095.78	15421	1829.4	1526.82	1723.65	1429.82	1549.18	299.942	9339.76
6	702.484	1828.51	1825.75	1718.85	280.667	0	1355.72	1276.23	9126.08	9392.84	9509.8	19834.5	21472.6	22711.5	23603.5	1521.36	1416.74	1213.45	1203.99	984.393	1273.53	1814	2863.3	15279.3	1626.75	1324.21	1511.07	1205.2	1312.8	549.586	9205.58
7	1156.01	3183.34	3180.45	3074.38	1178.69	1355.72	0	656.488	9240.61	9502.13	9616.4	19774.2	21401.7	22633.6	23523.6	2876.8	2772.19	2568.82	2558.62	2335.47	270.106	3049.65	3501.5	15287.9	2981.81	2678.63	2864.71	2550.73	2641.44	933.481	9370.92
8	1142.47	3021.59	3015.38	2898.57	1231.93	1276.23	656.488	0	8628.42	8891.93	9007.03	19228.4	20862.8	22099.5	22997.4	2712.34	2607.46	2410.76	2386.89	2154.02	462.783	2684.03	2915.08	14784.7	2817.17	2525.05	2683.57	2356.69	2410.11	1092.2	8916.26
9	8983.22	9320.63	9299.54	9236.93	9314.31	9126.08	9240.61	8628.42	0	273.254	392.45	10963.7	12635.1	13896.6	14850.5	9230	9202.45	9177.91	9124.89	9043.23	8995.61	8299.81	6426.59	7255.74	9250.41	9211.34	9142.04	9004.1	8854.14	9345.81	4508.73
10	9244.98	9585.12	9563.95	9502.12	9580.15	9392.84	9502.13	8891.93	273.254	0	119.269	10692.5	12364	13625.7	14580.4	9495.69	9468.45	9444.36	9391.46	9310.24	9256.98	8566.03	6692.44	7020.5	9515.63	9477.29	9407.57	9270.14	9119.85	9610.11	4523.63
11	9359.93	9701.8	9680.61	9619.05	9696.68	9509.8	9616.4	9007.03	392.45	119.269	0	10574.1	12245.7	13507.5	14462.7	9612.77	9585.62	9561.65	9508.79	9427.66	9371.25	8683.33	6809.53	6920.47	9632.55	9594.42	9524.6	9387.28	9236.89	9725.95	4539.07
12	19560.1	20081.2	20058.6	20008.5	19989.1	19834.5	19774.2	19228.4	10963.7	10692.5	10574.1	0	1677.1	2942.4	3942.5	19999.5	19972.7	19945.7	19895.9	19811.5	19533.2	19104.7	17213.5	5499.45	20015.5	19973.5	19908.6	19763.5	19613.7	19971.8	11721.5
13	21186.5	21721.1	21698.4	21649.7	21624.1	21472.6	21401.7	20862.8	12635.1	12364	12245.7	1677.1	0	1265.3	2282.01	21640.5	21613.9	21586.8	21537.4	21453.1	21160.9	20750.8	18860.9	6855.38	21656	21613.8	21549.3	21403.9	21254.3	21603	13229.2
14	22417.7	22961.1	22938.3	22890.6	22861	22711.5	22633.6	22099.5	13896.6	13625.7	13507.5	2942.4	1265.3	0	1062.72	22881.2	22854.7	22827.6	22778.5	22694.4	22393.1	21995.1	20106.4	7957.94	22896.4	22854.1	22790	22644.3	22494.8	22837.3	14389.9
15	23295.8	23848.9	23825.9	23780.4	23749.9	23603.5	23523.6	22997.4	14850.5	14580.4	14462.7	3942.5	2282.01	1062.72	0	23770.8	23744.7	23718	23669.5	23586.3	23282.3	22893	21008.8	8679.76	23785.1	23743.1	23679.4	23533.9	23384.5	23722.9	15155.8
16	1968.29	310.609	307.296	200.342	1726.29	1521.36	2876.8	2712.34	9230	9495.69	9612.77	19999.5	21640.5	22881.2	23770.8	0	105.47	308.177	325.592	558.432	2779.47	961.54	2824.11	15393.5	107.898	203.416	91.8096	385.917	438.179	2006.18	9230.96
17	1868.89	415.886	412.195	302.881	1624.03	1416.74	2772.19	2607.46	9202.45	9468.45	9585.62	19972.7	21613.9	22854.7	23744.7	105.47	0	203.394	220.586	453.614	2674.36	956.278	2788.34	15370.1	212.231	107.564	121.709	293.857	375.751	1904.43	9211.71
18	1675.96	617.164	613.746	506.093	1422.2	1213.45	2568.82	2410.76	9177.91	9444.36	9561.65	19945.7	21586.8	22827.6	23718	308.177	203.394	0	56.0803	262.307	2471.93	1005.15	2754.41	15347	413.953	119.67	305.473	186.593	346.999	1703.11	9194.5
19	1667.49	635.635	630.85	517.731	1420.46	1203.99	2558.62	2386.89	9124.89	9391.46	9508.79	19895.9	21537.4	22778.5	23669.5	325.592	220.586	56.0803	0	233.084	2457.53	957.732	2701.99	15302.4	431.263	153.271	309.705	147.709	304.875	1701.72	9156.46
20	1459.26	867.888	862.368	748.025	1213.54	984.393	2335.47	2154.02	9043.23	9310.24	9427.66	19811.5	21453.1	22694.4	23586.3	558.432	453.614	262.307	233.084	0	2229.61	1009.04	2622.73	15227.7	663.382	376.388	534.869	239.428	379.169	1494.2	9096.72
21	1008.36	3086.92	3082.15	2975	1133.53	1273.53	270.106	462.783	8995.61	9256.98	9371.25	19533.2	21160.9	22393.1	23282.3	2779.47	2674.36	2471.93	2457.53	2229.61	0	2887.34	3252.38	15035.1	2883.63	2581.54	2759.37	2435.54	2511.82	900.825	9109.13
22	2174.42	1047.46	1029.18	949.666	2083.1	1814	3049.65	2684.03	8299.81	8566.03	8683.33	19104.7	20750.8	21995.1	22893	961.54	956.278	1005.15	957.732	1009.04	2887.34	0	1942.24	14566	973.744	1002.5	881.724	872.415	709.485	2338.14	8504.11
23	2865.71	2950.9	2930.01	2855.5	3095.78	2863.3	3501.5	2915.08	6426.59	6692.44	6809.53	17213.5	18860.9	20106.4	21008.8	2824.11	2788.34	2754.41	2701.99	2622.73	3252.38	1942.24	0	12732.4	2854.71	2790.19	2735.8	2578.86	2432.01	3212.51	6843.45
24	14931.7	15462.6	15439	15403.7	15421	15279.3	15287.9	14784.7	7255.74	7020.5	6920.47	5499.45	6855.38	7957.94	8679.76	15393.5	15370.1	15347	15302.4	15227.7	15035.1	14566	12732.4	0	15403.2	15364.2	15301.7	15160.7	15012.2	15395.5	6489.89
25	2062.4	204.6	200.024	108.208	1829.4	1626.75	2981.81	2817.17	9250.41	9515.63	9632.55	20015.5	21656	22896.4	23785.1	107.898	212.231	413.953	431.263	663.382	2883.63	973.744	2854.71	15403.2	0	304.039	149.255	478.174	506.424	2107.58	9234.39
26	1767.49	505.618	501.896	402.931	1526.82	1324.21	2678.63	2525.05	9211.34	9477.29	9594.42	19973.5	21613.8	22854.1	23743.1	203.416	107.564	119.67	153.271	376.388	2581.54	1002.5	2790.19	15364.2	304.039	0	209.306	235.818	360.118	1804.65	9199.75
27	1942.41	343.923	334.547	225.078	1723.65	1511.07	2864.71	2683.57	9142.04	9407.57	9524.6	19908.6	21549.3	22790	23679.4	91.8096	121.709	305.473	309.705	534.869	2759.37	881.724	2735.8	15301.7	149.255	209.306	0	336.329	359.625	2001.15	9141.1
28	1622.84	677.853	668.129	560.722	1429.82	1205.2	2550.73	2356.69	9004.1	9270.14	9387.28	19763.5	21403.9	22644.3	23533.9	385.917	293.857	186.593	147.709	239.428	2435.54	872.415	2578.86	15160.7	478.174	235.818	336.329	0	177.89	1702.15	9009.75
29	1697.56	684.499	669.872	566.33	1549.18	1312.8	2641.44	2410.11	8854.14	9119.85	9236.89	19613.7	21254.3	22494.8	23384.5	438.179	375.751	346.999	304.875	379.169	2511.82	709.485	2432.01	15012.2	506.424	360.118	359.625	177.89	0	1814.98	8866.51
30	533.064	2303.86	2301.87	2206.42	299.942	549.586	933.481	1092.2	9345.81	9610.11	9725.95	19971.8	21603	22837.3	23722.9	2006.18	1904.43	1703.11	1701.72	1494.2	900.825	2338.14	3212.51	15395.5	2107.58	1804.65	2001.15	1702.15	1814.98	0	9322.52
31	8818.15	9287.07	9263.1	9242.66	9339.76	9205.58	9370.92	8916.26	4508.73	4523.63	4539.07	11721.5	13229.2	14389.9	15155.8	9230.96	9211.71	9194.5	9156.46	9096.72	9109.13	8504.11	6843.45	6489.89	9234.39	9199.75	9141.1	9009.75	8866.51	9322.52	0

Figure 7: Proximity Matrix for the solar system moons

Results:

The results of this algorithm can be seen in the form of a text file generated by the program with the relevant merge information and the clusters within each clustering as well as with a graphical representation generated in excel. The graphical representation uses an approach similar to a dendrogram to create a tree like structure that shows how each iteration clusters the data points.

Merge Results

Merge 1
Cluster 1 ID: 10
Cluster 2 ID: 16
Merge Value: 54.626

Merge 2
Cluster 1 ID: 5
Cluster 2 ID: 13
Merge Value: 157.48

Merge 3
Cluster 1 ID: 10, 16
Cluster 2 ID: 8
Merge Value: 172.898

Merge 4
Cluster 1 ID: 1
Cluster 2 ID: 15
Merge Value: 176.516

Merge 5
Cluster 1 ID: 1, 15
Cluster 2 ID: 7
Merge Value: 199.169

Merge 6
Cluster 1 ID: 10, 16, 8
Cluster 2 ID: 6
Merge Value: 214.179

Merge 7
Cluster 1 ID: 4
Cluster 2 ID: 9
Merge Value: 217.49

Merge 8
Cluster 1 ID: 5, 13
Cluster 2 ID: 11
Merge Value: 302.981

Merge 9
Cluster 1 ID: 10, 16, 8, 6
Cluster 2 ID: 14
Merge Value: 341.777

Merge 10
Cluster 1 ID: 2
Cluster 2 ID: 3
Merge Value: 359.662

Merge 11
Cluster 1 ID: 1, 15, 7
Cluster 2 ID: 12
Merge Value: 396.079

Merge 12
Cluster 1 ID: 10, 16, 8, 6, 14
Cluster 2 ID: 4, 9
Merge Value: 1291.02

Merge 13
Cluster 1 ID: 1, 15, 7, 12
Cluster 2 ID: 2, 3
Merge Value: 1839.6

Merge 14
Cluster 1 ID: 10, 16, 8, 6, 14, 4, 9
Cluster 2 ID: 5, 13, 11
Merge Value: 4239.28

Merge 15
Cluster 1 ID: 10, 16, 8, 6, 14, 4, 9, 5, 13, 11
Cluster 2 ID: 1, 15, 7, 12, 2, 3
Merge Value: 26103.8

Figure 8: Merge Results for the Crime Rates

Merge Results

Merge 1
Cluster 1 ID: 2
Cluster 2 ID: 3
Merge Value: 23.9735

Merge 2
Cluster 1 ID: 18
Cluster 2 ID: 19
Merge Value: 56.0803

Merge 3
Cluster 1 ID: 16
Cluster 2 ID: 27
Merge Value: 91.8096

Merge 4
Cluster 1 ID: 17
Cluster 2 ID: 26
Merge Value: 107.564

Merge 5
Cluster 1 ID: 4
Cluster 2 ID: 25
Merge Value: 108.208

Merge 6
Cluster 1 ID: 10
Cluster 2 ID: 11
Merge Value: 119.269

Merge 7
Cluster 1 ID: 18, 19
Cluster 2 ID: 28
Merge Value: 167.151

Merge 8
Cluster 1 ID: 18, 19, 28
Cluster 2 ID: 20
Merge Value: 244.94

Merge 9
Cluster 1 ID: 7
Cluster 2 ID: 21
Merge Value: 270.106

Merge 10
Cluster 1 ID: 5
Cluster 2 ID: 6
Merge Value: 280.667

Merge 11
Cluster 1 ID: 18, 19, 28, 20
Cluster 2 ID: 29
Merge Value: 302.233

Merge 12
Cluster 1 ID: 10, 11
Cluster 2 ID: 9
Merge Value: 332.852

Merge 13
Cluster 1 ID: 5, 6
Cluster 2 ID: 30
Merge Value: 424.764

Merge 14
Cluster 1 ID: 7, 21
Cluster 2 ID: 8
Merge Value: 559.636

Merge 15
Cluster 1 ID: 5, 6, 30
Cluster 2 ID: 1
Merge Value: 621.584

Merge 16
Cluster 1 ID: 16, 27
Cluster 2 ID: 17, 26
Merge Value: 639.901

Merge 17
Cluster 1 ID: 2, 3
Cluster 2 ID: 4, 25
Merge Value: 677.728

Merge 18
Cluster 1 ID: 18, 19, 28, 20, 29
Cluster 2 ID: 22
Merge Value: 910.764

Merge 19
Cluster 1 ID: 14
Cluster 2 ID: 15
Merge Value: 1062.72

Merge 20
Cluster 1 ID: 12
Cluster 2 ID: 13
Merge Value: 1677.1

Merge 21
Cluster 1 ID: 18, 19, 28, 20, 29, 22
Cluster 2 ID: 23
Merge Value: 2505.37

Merge 22
Cluster 1 ID: 10, 11, 9
Cluster 2 ID: 31
Merge Value: 4523.81

Merge 23
Cluster 1 ID: 2, 3, 4, 25
Cluster 2 ID: 16, 27, 17, 26
Merge Value: 5036.63

Merge 24
Cluster 1 ID: 12, 13
Cluster 2 ID: 24
Merge Value: 6177.41

Merge 25
Cluster 1 ID: 5, 6, 30, 1
Cluster 2 ID: 7, 21, 8
Merge Value: 10262.2

Merge 26
Cluster 1 ID: 12, 13, 24
Cluster 2 ID: 14, 15
Merge Value: 18046.6

Merge 27
Cluster 1 ID: 2, 3, 4, 25, 16, 27, 17, 26
Cluster 2 ID: 18, 19, 28, 20, 29, 22, 23
Merge Value: 43318.2

Merge 28
Cluster 1 ID: 2, 3, 4, 25, 16, 27, 17, 26, 18, 19, 28, 20, 29, 22, 23
Cluster 2 ID: 5, 6, 30, 1, 7, 21, 8
Merge Value: 108978

Merge 29
Cluster 1 ID: 2, 3, 4, 25, 16, 27, 17, 26, 18, 19, 28, 20, 29, 22, 23, 5, 6, 30, 1, 7, 21, 8
Cluster 2 ID: 10, 11, 9, 31
Merge Value: 146394

Merge 30
Cluster 1 ID: 2, 3, 4, 25, 16, 27, 17, 26, 18, 19, 28, 20, 29, 22, 23, 5, 6, 30, 1, 7, 21, 8, 10, 11, 9, 31
Cluster 2 ID: 12, 13, 24, 14, 15
Merge Value: 477783

Figure 9: Merge Results for Moon data

The data presented above are generated by the program and contain the information for every merge involved in each iteration of the program. The merge contains the ids of each cluster so we are aware of what clusters are being merged at each iteration as well as the merge value obtained by finding the smallest group average amongst the clusters. It is also important to note that with N data points this algorithm must complete N-1 merges as each time the algorithm performs an iteration 2 clusters are joined which means that one cluster is copied and deleted. We can observe this and see that this relationship does in fact hold for this study.

While the data presented in the test file does contain all the relevant information it is often useful to generate graphical representations of the data. After the program completed I placed the data from the crime rates set and created a tree like graphical representation that shows the relationship between each data point as the algorithm progressed. The values at which each merge takes place is also noted in the graph. The graphical results of the Crime rates data set can be seen in Figure 10 on the following page.

By observing this graph, we can see that there are some cities that were grouped similarly based on the crime rates. In the preliminary analysis, we were able to see that there was not much grouping in the original data set but by applying this algorithm we can build relationships within the scale of the data set to create clusters. For example, we can see that Kansas City and Washington are very similar in their crime rates and we can also see similar results for cities such as Denver and New York or Atlanta and Tucson. This information can then be used to further define the issue within the context of the data set and further the study being performed.

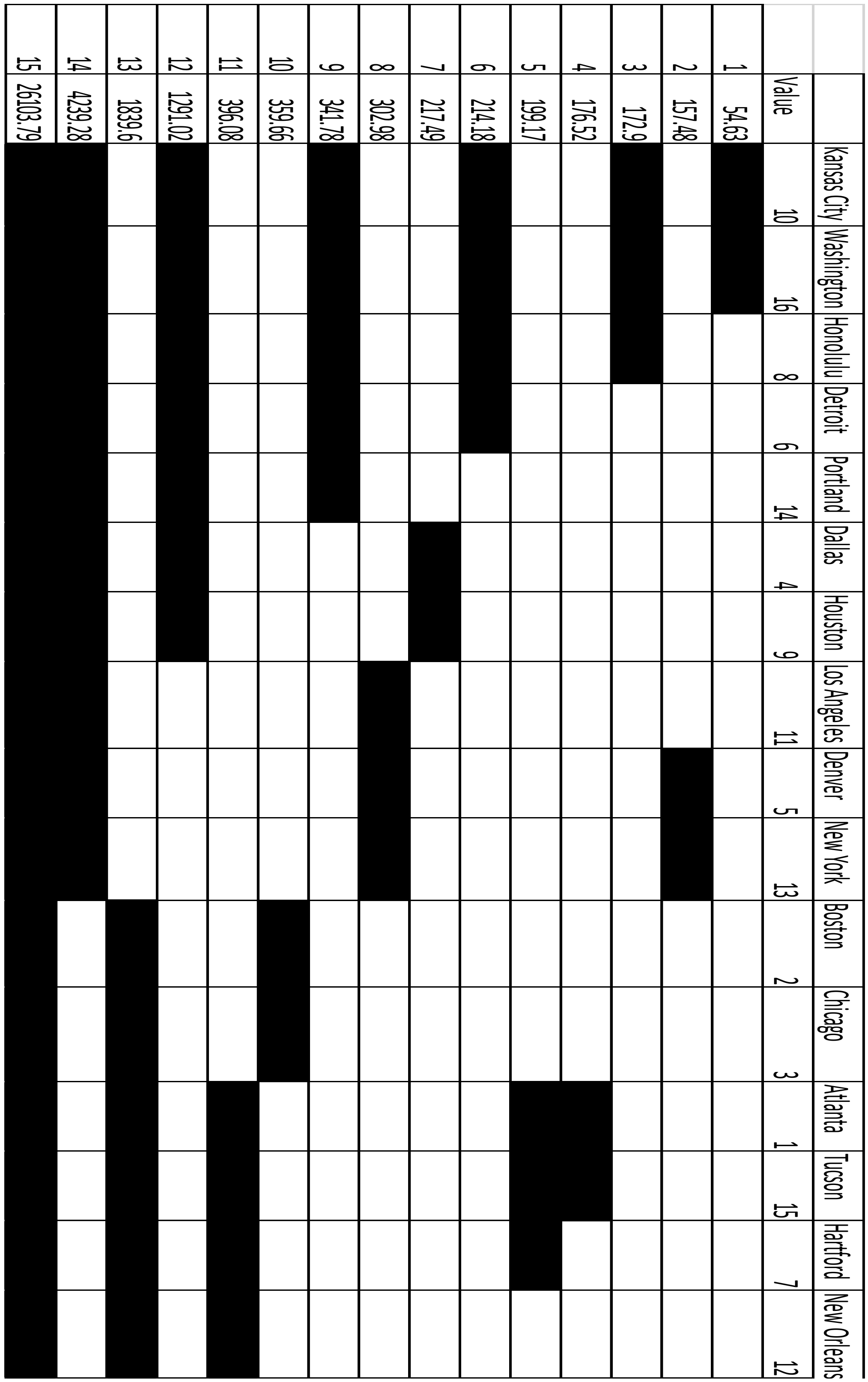

Figure 10: Graphical representation of the Crime rates clustering results.

Conclusions:

With the completion of this study and the results obtained it can be concluded that the BAHC algorithm has performed quite well in clustering the data sets. In the case of the crime rates data, while there seemed like little relationship between the relative groupings the algorithm successfully measured and grouped each data point successfully. Similarly, the results obtained in the solar moon data was promising as well. From the results, we can clearly identify were relationships exist and where data points begin to deviate from each other. In the case of the moons, similar moons can be identified and then compared to see why they are similar.

By looking deeper into the results of the moon data clustering we can identify which data points were clustered. The first two clusters for example which were points 2, 3 and 18, 19 with proximity measures of 23.97 and 56.08 respectively, each turn out to be moons from the same planet. We of course know this relationship exists prior to running the algorithm, but can now confirm that the BAHC algorithm implemented in this study can correctly identify these relationships and cluster them accordingly.

Within this study there some areas for improvement. One such areas is in the graphical representation of the results. While the information may be there it is somewhat unclear to the unexperienced user and could be improved upon. One such approach would be to develop a custom dendrogram graphing tool that allows you to select which data points are to be merged with the correlating proximity measure at each stage of the algorithm so that a clear dendrogram can be produced. Another area or improvement is in the preliminary analysis of the data. While the original data was graphed using scatter plots, the data is more fitting for a three-dimensional graph that can visualize each of the three dimensions in one chart. The original data could have also been transformed to ensure that the differing scales in between each attribute did not significantly affect the proximity measures.

One of the limiting factors of this program’s implementation is the dependency of three dimensions. For this study the algorithm was specifically built for a three-dimensional data set and would have to be restructured if other dimensionalities were to be used. Due to the Framework that has been built in this implementation the changes would be minimal but it still poses a challenge.

The algorithm implemented is performing well and can be scaled to cluster larger data sets. This study was extremely interesting and informative. I look forward to learning more about various other data mining techniques. A detailed code listing for the implementation used in this study can be found in the following section.

Code Listing:

```
/*******************************************************
Author:Benjamin Fields
Date:11/22/2016
File: Source.cpp
Description:Program that implements Agglomerative Clustering algorithm
on a three dimensional data set related to crime rates and moons in
the solar system. The program creates a proximity matrix using
euclidian distances and evaluates the distance between clusters using
the group average approach. Each merge is tracked and written to
a text file for analyis after running the program. Similarly the
proximity matrix can be seen in an external .csv file. The program
terminates once all points have been merged into one cluster.
************************************************************/
#include <iostream>
#include <vector>
#include <fstream>
#include "Point.h"
#include <iomanip>
#include "Cluster.h"
#include "Merge.h"
#include "Clustering.h"

/*************************************************************
Description: read in the data file to be used for program data.
Either reads in the crime data or moon data based on assigned
boolean
************************************************************/
void readInData(std::vector<double> &myData, bool moons) {
	std::ifstream inputFile;
	if(moons == 0)
		inputFile.open("CrimesData.txt");
	else
		inputFile.open("MoonsData.txt");
	double value;
	if (inputFile) {
		while (inputFile >> value) {
			myData.push_back(value);
		}
		inputFile.close();
	}
	else {
		std::cout << "Could not open File!" << std::endl;
	}
}
/*************************************************************
Description:writes the proximity matrix to a .csv file for further
analysis and tracking after running program.
************************************************************/
void writeToFile(double *Matrix,int num) {
	std::ofstream myFile;
	myFile.open("Results.csv");
	int index = 0;
	for (int i = 0; i < num; i++) {
		for (int j = 0; j < num; j++) {
			myFile << Matrix[index]<<",";
			index++;
```

```
                }
                myFile << "\n";
        }

        myFile.close();

}
/**************************************************************
Description:writes the final merge results that were captured in
the program to an external text file for further analysis and tracking
**************************************************************/
void writeMergeResults(Merge *myMerges, int num) {
        std::ofstream myFile;
        myFile.open("MergeResults.txt");

        myFile << "Merge Results\n\n";
        for (int i = 0; i < num; i++) {
                myFile << "Merge " << i + 1 << std::endl;
                myFile << "Cluster 1 ID: " << myMerges[i].getCluster1ID() << "\n";
                myFile << "Cluster 2 ID: " << myMerges[i].getCluster2ID() << "\n";
                myFile << "Merge Value: " << myMerges[i].getMergeValue() << "\n\n";
        }

        myFile.close();
}
/**************************************************************
Description:simple function to print a vector for checking
**************************************************************/
void printVector(std::vector<double> &myData) {
        for (int i = 0; i < (int)myData.size(); i++) {
                std::cout << "Data point " << i + 1 << " : " << myData[i] << std::endl;
        }
}
/**************************************************************
Description:function that takes the data that was read into the
program and assigns in to points based on a given number of dimensions
**************************************************************/
std::vector<Point> createPoints(std::vector<double> &myData, int NumDim) {
        int numData = myData.size();
        int numPoints = numData / NumDim;
        std::vector<Point> myPoints;
        for (int i = 0; i < numPoints; i++) {
                Point next(myData[i], myData[i + numPoints], myData[i + 2 *
numPoints],i+1);
                myPoints.push_back(next);
        }
        return myPoints;
}
/**************************************************************
Description:simple function to print the values of the points
for checking.
**************************************************************/
void printPoints(std::vector<Point> &myPoints) {
        for (int i = 0; i < (int)myPoints.size(); i++) {
                std::cout << myPoints[i].printPoint() << std::endl;
        }
}
/**************************************************************
```

```
Description:Function that calcualtes the proximity matrix for
the given initial data points. The values use the euclidean distance
and outputs the results to the console as well as an array to store the
matrix for further use later in the program
************************************************************/
double* ProximityMatrix(std::vector<Point> myPoints) {
	int numPoints = myPoints.size();
	double *myMatrix = new double[numPoints * numPoints];
	int index = 0;
	int spaces = 10;
	std::cout << std::left << std::setw(spaces+1)<<"   ";
	std::cout << std::fixed << std::showpoint << std::setprecision(2);
	for (int i = 0; i < (int)myPoints.size(); i++) {
		std::cout <<std::right<<std::setw(spaces)<< "Point " << i + 1 << " |";
	}
	std::cout << "\n";
	for (int i = 0; i < (int)myPoints.size(); i++) {
		std::cout <<std::left<<std::setw(spaces)<< "Point " << i + 1 << " | ";
		for (int j = 0; j < (int)myPoints.size(); j++) {
			std::cout <<std::right<<std::setw(spaces)<<
myPoints[i].EuclidianDist(myPoints[j])<<" | ";
			myMatrix[index] = myPoints[i].EuclidianDist(myPoints[j]);
			index++;
		}
		std::cout << "\n";
	}
	return myMatrix;
}

/************************************************************
Description:the main entry point of the program. This function controls
the execution of each step:
	read in data
	organize data
	create proximity matrix
	track merging of clusters
	report results
************************************************************/
int main(int argc, char **argv) {
	bool moons = 0;//boolean for determining which data set to use

	std::vector<double> myData;//initial vector to hold data

	readInData(myData, moons);
	printVector(myData);//checking line

	//create the initial point data structures for use in program
	std::vector<Point> myPoints = createPoints(myData, 3);

	printPoints(myPoints);//checking line
	double *myMatrix = ProximityMatrix(myPoints);//create proximity matrix
	writeToFile(myMatrix, myPoints.size());
	std::cout<<"\n";
	int numPoints = myPoints.size();

	//create initial cluster data structures with points
	Cluster *myClusters = new Cluster[numPoints];
	for (int i = 0; i < numPoints; i++) {
```

```
			myClusters[i].AddPoint(myPoints[i]);
			myClusters[i].setId(std::to_string(i + 1));
		}

		//check clusters
		for (int i = 0; i < numPoints; i++) {
			myClusters[i].printCluster();
		}

		//organize the data into a clustering data structure for easy data management
		Clustering myClustering;
		for (int i = 0; i < numPoints; i++) {
			myClustering.addCluster(myClusters[i]);
		}

		//create merge data structures for easy tracking of results in each iteration
		Merge *myMerges = new Merge[numPoints - 1];
		for (int i = 0; i < numPoints - 1; i++) {
			Merge nextMerge = myClustering.calulateSmallestandMerge();
			myMerges[i] = nextMerge;
		}

		//check results in console
		std::cout << "\n\nMerge Results\n\n";
		for (int i = 0; i < numPoints - 1; i++) {
			std::cout << "Merge " << i + 1 << std::endl;
			myMerges[i].printMerge();
		}

		//output results to test file
		writeMergeResults(myMerges, numPoints - 1);

		system("PAUSE");
		return 0;
	}

	Author:Benjamin Fields
	Date:11/22/2016
	File:Point.h
	Description:Data structure to represent each data point. Each
	point in this program has three attributes related to three
	dimensions.This class also contains the function for euclidian
	distance
	*************************************************************/
	#ifndef POINT_H
	#define POINT_H
	#include <string>

	class Point {
		private:
			int id;
			double xCoord;
			double yCoord;
			double zCoord;
		public:
			Point(double x, double y, double z, int i);
			int getId();
			double getXCoord();
```

```
		double getYCoord();
		double getZCoord();
		void setXCoord(double num);
		void setYCoord(double num);
		void setZCoord(double num);
		void setId(int num);
		double EuclidianDist(Point other);
		std::string printPoint();
};



#endif

/*******************************************************
Author:Benjamin Fields
Date:11/22/2016
File: Point.cpp
Description:Implementation of the point class.
************************************************************/
#include "Point.h"
#include <cmath>
#include <string>

/************************************************************
Description:constructor that builds each point with the proper
attributes and assigns an id.
************************************************************/
Point::Point(double x, double y,double z, int i) {
	setXCoord(x);
	setYCoord(y);
	setZCoord(z);
	setId(i);
}
/************************************************************
Description:standard getter for id
************************************************************/
int Point::getId() {
	return id;
}
/************************************************************
Description:standard getter for xcoord
************************************************************/
double Point::getXCoord() {
	return xCoord;
}
/************************************************************
Description:standard getter for ycoord
************************************************************/
double Point::getYCoord() {
	return yCoord;
}
/************************************************************
Description:standard getter for zcoord
************************************************************/
double Point::getZCoord() {
	return zCoord;
}
```

```cpp
/*************************************************************
Description:standard setter for xcoord
*************************************************************/
void Point::setXCoord(double num) {
	xCoord = num;
}
/*************************************************************
Description:standard setter for ycoord
*************************************************************/
void Point::setYCoord(double num) {
	yCoord = num;
}
/*************************************************************
Description:standard setter for zcoord
*************************************************************/
void Point::setZCoord(double num) {
	zCoord = num;
}
/*************************************************************
Description:standard setter for the id
*************************************************************/
void Point::setId(int num) {
	id = num;
}
/*************************************************************
Description:function to calculate the euclidian distance between
two points with three dimensions.
*************************************************************/
double Point::EuclidianDist(Point other) {
	return sqrt(pow(this->getXCoord() - other.getXCoord(), 2) + pow(this->getYCoord()
- other.getYCoord(), 2) + pow(this->getZCoord() - other.getZCoord(), 2));
}
/*************************************************************
Description:function to print the point data to console for
checking
*************************************************************/
std::string Point::printPoint() {
	return "Point "+std::to_string(id)+" [ " + std::to_string(xCoord) + " , " +
std::to_string(yCoord) + " , " + std::to_string(zCoord) + " ]";
}

/*******************************************************
Author:Benjamin Fields
Date:11/22/2016
File:Cluster.h
Description:Data structure to contain the functionality for
each cluster to be used in the program. This class contains
the functionality for calculating the group average between
clusters.
*************************************************************/
#ifndef CLUSTER_H
#define CLUSTER_H
#include <vector>
#include "Point.h"
class Cluster {
	private:
		std::string id;
		std::vector<Point> ClusterPoints;
```

```cpp
	public:
		Cluster();
		Cluster(Point p);
		void AddPoint(Point p);
		std::string getId();
		void setId(std::string s);
		void addCluster(Cluster &c);
		std::vector<Point> getCluster();
		double getAverage(Cluster c);
		void printCluster();

};


#endif

/*********************************************************
Author:Benjamin Fields
Date:11/22/2016
File:Cluster.cpp
Description:implementation of the cluster class
*************************************************************/
#include "Cluster.h"
#include <iostream>

/**************************************************************
Description:default constructor needed for compiliation. Default
not needed for the execution of program.
*************************************************************/
Cluster::Cluster() {
	id = "none";
}

/**************************************************************
Description:constructor that creates the initial cluster from each
point in the data set.
*************************************************************/
Cluster::Cluster(Point p) {
	id = std::to_string(p.getId());
	ClusterPoints.push_back(p);

}

/**************************************************************
Description:wraper function to add a point to the vector contained
in the class.
*************************************************************/
void Cluster::AddPoint(Point p) {
	ClusterPoints.push_back(p);
}

/**************************************************************
Description:standard getter for the id of the cluster
*************************************************************/
std::string Cluster::getId() {
	return id;
}
```

```
/*************************************************************
Description:standard setter for the id
************************************************************/
void Cluster::setId(std::string s) {
	id = s;
}

/*************************************************************
Description:function that allows another cluster to be added to
another. The contents of the argument cluster are added to the calling
cluster and the id is added to represent the combined cluster.
************************************************************/
void Cluster::addCluster(Cluster &c) {
	std::vector<Point> cluster = c.getCluster();
	for (int i = 0; i < (int)cluster.size(); i++) {
		ClusterPoints.push_back(cluster[i]);
	}
	id = id + ", " + c.getId();

}

/*************************************************************
Description:standard getter for the cluster
************************************************************/
std::vector<Point> Cluster::getCluster() {
	return ClusterPoints;
}

/*************************************************************
Description:function that calculates the group average distance between
two clusters.
************************************************************/
double Cluster::getAverage(Cluster c) {
	double averageP = 0.0;
	for (int i = 0; i < (int)ClusterPoints.size(); i++) {
		for (int j = 0; j < (int)c.getCluster().size(); j++) {
			averageP += ClusterPoints[i].EuclidianDist(c.getCluster()[j]);
		}
	}
	averageP = averageP / (double)ClusterPoints.size()*(double)c.getCluster().size();
	return averageP;
}

/*************************************************************
Description:function that prints the cluster information to the
console for use in checking.
************************************************************/
void Cluster::printCluster() {
	std::cout << "Cluster id: " << id << std::endl;
	std::cout << "Number of points: " << ClusterPoints.size() << std::endl;
	for (int i = 0; i < (int)ClusterPoints.size(); i++) {
		std::cout << ClusterPoints[i].printPoint() << std::endl;
	}
}

/*******************************************************
Author:Benjamin Fields
Date:11/22/2016
```

```
File:Clustering.h
Description:Data structure to allow for easy data management
of the clusters as the program iterates through each merge.
this class contains the functionality for identifying the closest
custers and merging them. It then deletes the copyed cluster.
************************************************************/
#ifndef CLUSTERING_H
#define CLUSTERING_H
#include <vector>
#include "Cluster.h"
#include "Merge.h"

class Clustering {
	private:
		std::vector<Cluster> myClustering;
	public:
		std::vector<Cluster> getMyClustering();
		void addCluster(Cluster c);
		void deleteCluster(int i);
		Merge calulateSmallestandMerge();
		void printClustering();
};
#endif

/*********************************************************
Author:Benjamin Fields
Date:11/22/2016
File:Clustering.cpp
Description:Implementation of the clustering class
************************************************************/
#include "Clustering.h"
#include <iostream>

/*************************************************************
Description:getter for the clustering vector
************************************************************/
std::vector<Cluster> Clustering::getMyClustering() {
	return myClustering;
}

/*************************************************************
Description:function that deletes the copied cluster using an
identified index
************************************************************/
void Clustering::deleteCluster(int i){
	myClustering.erase(myClustering.begin() + i);
}

/*************************************************************
Description: wraper function to allow a new cluster to be added
to the clustering. Used in the initial creation of the clustering
************************************************************/
void Clustering::addCluster(Cluster c) {
	myClustering.push_back(c);
}

/*************************************************************
Description:function that checks the group average distance between
```

```
each cluster in the clustering and identifies the closest clusters.
The index of the pair is recorded along with the group average.
The merge data is passed to a merge object for final results.The
extra cluster is deleted in the end.
***********************************************************/
Merge Clustering::calulateSmallestandMerge() {
	Merge Chosen;
	double smallest = 1000000.00;//high value to initialize minimum check
	for (int i = 0; i < (int)myClustering.size(); i++) {
		for (int j = 0; j < (int)myClustering.size(); j++) {
			if (myClustering[i].getAverage(myClustering[j]) < smallest && i !=
j) {
				smallest =
myClustering[i].getAverage(myClustering[j]);//assign new min
				Chosen.setCluster1(i);//track values in merge object
				Chosen.setCluster2(j);
				Chosen.setcluster1id(myClustering[i].getId());
				Chosen.setcluster2id(myClustering[j].getId());

	Chosen.setMergeValue(myClustering[i].getAverage(myClustering[j]));
			}

		}
	}
	myClustering[Chosen.getCluster1()].addCluster(myClustering[Chosen.getCluster2()]);
	deleteCluster(Chosen.getCluster2());

	return Chosen;
}

/************************************************************
Description:function that prints the contents of the clustering
to the console for checking.
***********************************************************/
void Clustering::printClustering() {
	for (int i = 0; i < (int)myClustering.size(); i++) {
		myClustering[i].printCluster();
	}
}

/******************************************************
Author:Benjamin Fields
Date:11/22/2016
File:Merge.h
Description:Data structure used to contain the information from
each merge.
***********************************************************/
#ifndef MERGE_H
#define MERGE_H
#include<string>
class Merge {
	private:
		int id;
		double mergeValue;
		int cluster1;
		int cluster2;
		std::string cluster1id;
		std::string cluster2id;
```

```cpp
	public:
		Merge();
		int getId();
		double getMergeValue();
		int getCluster1();
		int getCluster2();
		void setId(int num);
		void setMergeValue(double num);
		void setCluster1(int i);
		void setCluster2(int i);
		void printMerge();
		void setcluster1id(std::string s);
		void setcluster2id(std::string s);
		std::string getCluster1ID();
		std::string getCluster2ID();
};

#endif

/*********************************************************
Author:Benjamin Fields
Date:11/22/2016
File:Merge.cpp
Description:implementation of the merge class
*************************************************************/
#include "Merge.h"
#include <iostream>

/**************************************************************
Description:default constructor
*************************************************************/
Merge::Merge() {
	id = 0;
}


/**************************************************************
Description:standard getter for id
*************************************************************/
int Merge::getId() {
	return id;
}
/**************************************************************
Description:standard getter for merge value
*************************************************************/
double Merge::getMergeValue() {
	return mergeValue;
}
/**************************************************************
Description:standard getter for cluster index
*************************************************************/
int Merge::getCluster1() {
	return cluster1;
}
/**************************************************************
Description:standard getter for cluster index
*************************************************************/
int Merge::getCluster2() {
```

```cpp
	return cluster2;
}
/**********************************************************
Description:standard setter for id
*********************************************************/
void Merge::setId(int num) {
	id = num;
}
/**********************************************************
Description:standard setter for merge value
*********************************************************/
void Merge::setMergeValue(double num) {
	mergeValue = num;
}
/**********************************************************
Description:standard setter for setting cluster 1 index
*********************************************************/
void Merge::setCluster1(int i) {
	cluster1 = i;
}
/**********************************************************
Description:standard setter for setting cluster 2 index
*********************************************************/
void Merge::setCluster2(int i) {
	cluster2 = i;
}
/**********************************************************
Description:print function for checking in console
*********************************************************/
void Merge::printMerge() {
	std::cout << "Cluster 1 index: " << cluster1 <<" Cluster 1 id: "<<cluster1id<<
std::endl;
	std::cout << "Cluster 2 index: " << cluster2 <<" Cluster 2 id: "<<cluster2id<<
std::endl;
	std::cout << "Merge Value: " << mergeValue << std::endl;

}
/**********************************************************
Description:setter for cluster 1 strings id. this tells what points
are in each cluster.
*********************************************************/
void Merge::setcluster1id(std::string s) {
	cluster1id = s;
}
/**********************************************************
Description:setter for cluster 2 strings id. this tells what points
are in each cluster.
*********************************************************/
void Merge::setcluster2id(std::string s) {
	cluster2id = s;
}

/**********************************************************
Description:getter for cluster 1 id
*********************************************************/
std::string Merge::getCluster1ID() {
	return cluster1id;
}
```

```
/**********************************************************
Description:getter for cluster 2 id
**********************************************************/
std::string Merge::getCluster2ID() {
	return cluster2id;
}
```

Original Data Sets:

```
#  file10.txt
#
#  Reference:
#
#    John Hartigan,
#    Clustering Algorithms,
#    Wiley, 1975.
#    ISBN 0-471-35645-X
#    LC: QA278.H36
#    Dewey: 519.5'3
#
#  From astonomical knowledge of 1970, a table of planetary moons
#  was compiled.
#
#  "Planet #" is the planet and the number of the moon.
#
#  "Distance" is the distance in thousands of miles between the moon
#  and the planet;
#
#  "Diameter" is the diameter in miles of the moon;
#
#  "Period" is the period, in days, of the orbit of the moon
#  about the planet.
#
"Planets and Moons, Hartigan page 122"
4 columns
31 rows
"Planet #"    "Distance"  "Diameter"  "Period"
"Earth 1"      239       2160        655
"Mars 1"         5.8       10.0        7.7
"Mars 2"        14.6       10.0       30.0
"Jupiter  1"    112        100        12.0
"Jupiter  2"    262       2020         42
"Jupiter  3"    417       1790         85
"Jupiter  4"    665       3120        172
"Jupiter  5"   1171       2770        401
"Jupiter  6"   7133         50       6014
"Jupiter  7"   7295         20       6232
"Jupiter  8"   7369         10       6325
"Jupiter  9"  13200         10      15146
"Jupiter 10"  14000         10      16620
"Jupiter 11"  14600         10      17734
"Jupiter 12"  14700         10      18792
"Saturn 1"      116        300         23
"Saturn 2"      148        400         33
"Saturn 3"      183        600         45
```

"Saturn 4"	235	600	66
"Saturn 5"	327	810	108
"Saturn 6"	759	2980	383
"Saturn 7"	920	100	511
"Saturn 8"	2213	500	1904
"Saturn 9"	8053	100	13211
"Uranus 1"	77	200	34
"Uranus 2"	119	500	60
"Uranus 3"	166	300	100
"Uranus 4"	272	600	209
"Uranus 5"	365	500	323
"Neptune 1"	220	2300	141
"Neptune 2"	3461	200	8626

```
#  file03.txt
#
#  Reference:
#
#    John Hartigan,
#    Clustering Algorithms,
#    Wiley, 1975.
#    ISBN 0-471-35645-X
#    LC: QA278.H36
#    Dewey: 519.5'3
#
#  A list of cities and the number of crimes per 100,000 population,
#  as of 1970.
#
#  Name is the name of the city.
#
#  Murder is the murder rate.
#
#  Rape is the rape rate.
#
#  Robbery is the robbery rate.
#
#  Assault is the assault rate.
#
#  Burglary is the burglary rate.
#
#  Larceny is the larceny rate.
#
#  Auto is the auto theft rate.
#
"City Crime Rates Per 100,000, Hartigan page 28"
8 columns
16 rows
```

"City"	"Murder"	"Rape"	"Robbery"	"Assault"	"Burglary"	"Larceny"	"Auto"
"Atlanta"	16.5	24.8	106	147	1112	905	494
"Boston"	4.2	13.3	122	90	982	669	954
"Chicago"	11.6	24.7	340	242	808	609	645
"Dallas"	18.1	34.2	184	293	1668	901	602
"Denver"	6.9	41.5	173	191	1534	1368	780
"Detroit"	13.0	35.7	477	220	1566	1183	788
"Hartford"	2.5	8.8	68	103	1017	724	468
"Honolulu"	3.6	12.7	42	28	1457	1102	637

"Houston"	16.8	26.6	289	186	1509	787	697
"Kansas City"	10.8	43.2	255	226	1494	955	765
"Los Angeles"	9.7	51.8	286	355	1902	1386	862
"New Orleans"	10.3	39.7	266	283	1056	1036	776
"New York"	9.4	19.4	522	267	1674	1392	848
"Portland"	5.0	23.0	157	144	1530	1281	488
"Tucson"	5.1	22.9	85	148	1206	756	483
"Washington"	12.5	27.6	524	217	1496	1003	739

References:

Hartigan, John A. *Clustering Algorithms*. New York: Wiley, 1975. Print

https://people.sc.fsu.edu/~jburkardt/datasets/hartigan/hartigan.html

Tan, Pang-Ning, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining*. Boston: Pearson Addison Wesley, 2005. Print.