Homework 1: Finding Similar Items: Textually Similar Documents

Konstantin Sozinov, sozinov@kth.se
Kim Hammar, kimham@kth.se

November 7, 2017

1 Solution

The implementation is done in pure Scala without any big data processing framework. The functionality is split into different class files: `Shingling.scala`, `MinHashing.scala`, `CompareSets.scala`, `CompareSignatures.scala`, `LSH.scala`, `Dataset.scala` and `Main.scala`. The first four classes have the functionality as described in the problem description. Dataset is a class with functionality for reading the dataset used for evaluation [1]. Main is a class for orchestrating the different steps of the pipeline: $Shinglinng \rightarrow MinHashing \rightarrow LSH \rightarrow Filter(CompareSignatures) \rightarrow Evaluation$.

2 How to run

Clone this repository and navigate to *similar_items* project. Then use:

```
sbt compile //compile
sbt test //test
sbt run //run
sbt assembly //generate fat jar
```

3 Evaluation and results

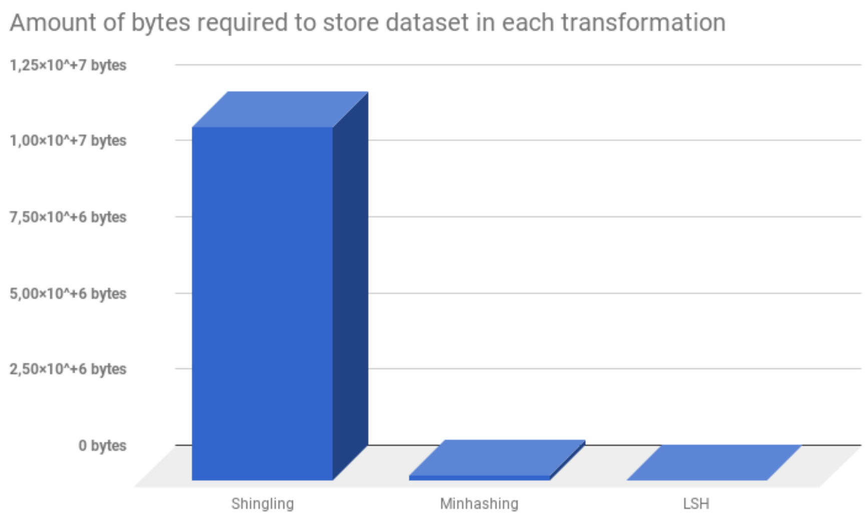


Figure 1: Memory analysis for different stages in comparing the documents

[1] https://archive.ics.uci.edu/ml/datasets/Twenty+Newsgroups

Example output (t=0.8, b=10, r=10, n=100)

```
[info] Running kth.se.id2222.Main
Shingles size: 11624856 bytes
Size after minhashing: 153112 bytes
Number of candidates pre LSH is approx: 10816.0
Number of candidates after LSH: 5
Size after LSH: 15656 bytes
Similar items: 4
Similar pair:
 src/resources/mini_newsgroups/alt.atheism/54485_copy,
 src/resources/mini_newsgroups/alt.atheism/54485
 similarity: 0.99
Similar pair:
 src/resources/mini_newsgroups/alt.atheism/51131,
 src/resources/mini_newsgroups/alt.atheism/51131copy
 similarity: 0.96
Similar pair:
 src/resources/mini_newsgroups/alt.atheism/54244,
 src/resources/mini_newsgroups/alt.atheism/54244_copy
 similarity: 0.99
Similar pair:
 src/resources/mini_newsgroups/alt.atheism/53653_copy,
 src/resources/mini_newsgroups/alt.atheism/53653
 similarity: 0.98
Time to compute similar items: 7.256831409 seconds, number of similar items found: 4
[success] Total time: 8 s, completed 2017-nov-07 10:41:05
```

Discovery of Frequent Itemsets and Association Rules

Konstantin Sozinov, sozinov@kth.se
Kim Hammar, kimham@kth.se

November 16, 2017

1 Solution

We implemented the solution in pure scala without any big data processing framework. We used the T10I4D100K.dat dataset uploaded on canvas. The source code is split into four classes, `Apriori.scala` that implementes the Apriori algorithm; `AssocRules.scala` that mines association rules from counted itemsets; `DataUtils.scala` that reads the data into a item-basket data model and `Main.scala` that orchestrates the pipeline and prints the results.

2 How to run

Clone this repository and navigate to *frequent_itemsets* project. Then use:

```
sbt compile //compile
sbt test //test
sbt run //run
sbt assembly //generate fat jar
```

3 Evaluation and results

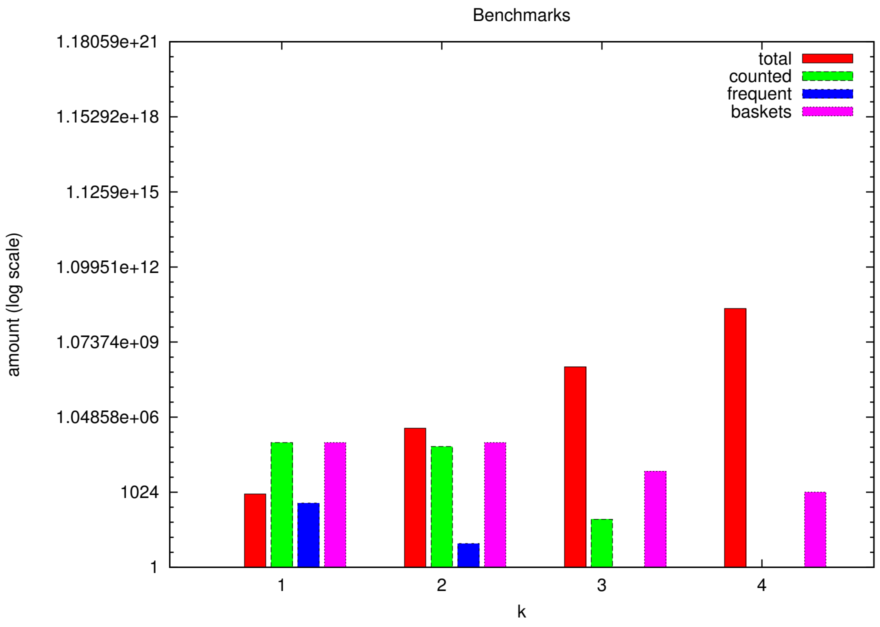


Figure 1: Analysis of number of counts made at each stage (log scale). Frequent itemsets of length 3 was 1, and counted itemsets of length 4 was thus 0. In-between each iteration we also filter the baskets (we hoped to reduce the complexity of the double-loop to count itemsets which has complexity $\mathcal{O}(b \cdot f \cdot k)$ where b is the number of baskets, f is the number of frequent sets and k is the size of each set ($\mathcal{O}(k)$ is the complexity to check if the set is subset of basket).

Example output (s=1000, c=0.5, k=3)

```
Counting all singletons for 100000 baskets

Total unique items to count: 870

Number of frequent singletons 375

Filtering out baskets with no frequent itemsets..
Processing frequent items for 2-sets, approximately 70312.5 sets to check and 99933 baskets
Filtering out baskets with no frequent itemsets..
Finding association rules for 2-sets
Processing frequent items for 3-sets, approximately 40.5 sets to check and 7087 baskets
Filtering out baskets with no frequent itemsets..
Finding association rules for 3-sets
Processing frequent items for 4-sets, approximately 0.5 sets to check and 1035 baskets
Filtering out baskets with no frequent itemsets..
Finding association rules for 4-sets
Done. Evaluating

Number Frequent Items of length 1: 375

Number Frequent Items of length 2: 9
Number of association rules for itemsets length: 2: 3

Association Rule: AssociationRule(Set(Item(227)),Item(390)),
confidence: 0.577007700770077,
interest: 0.550157700770077
Association Rule: AssociationRule(Set(Item(704)),Item(825)),
confidence: 0.6142697881828316,
interest: 0.5834197881828316
Association Rule: AssociationRule(Set(Item(704)),Item(39)),
confidence: 0.617056856187291,
interest: 0.574476856187291

Number Frequent Items of length 3: 1
Number of association rules for itemsets length: 3: 3

Association Rule: AssociationRule(Set(Item(825), Item(704)),Item(39)),
confidence: 0.9392014519056261,
interest: 0.8966214519056261
Association Rule: AssociationRule(Set(Item(39), Item(704)),Item(825)),
confidence: 0.9349593495934959,
interest: 0.9041093495934959
Association Rule: AssociationRule(Set(Item(39), Item(825)),Item(704)),
confidence: 0.8719460825610783,
interest: 0.8540060825610784

Number Frequent Items of length 4: 0
Number of association rules for itemsets length: 4: 0

[success] Total time: 189 s, completed 2017-nov-16 11:09:11
```

Homework 3: Mining Data Streams

Konstantin Sozinov, sozinov@kth.se
Kim Hammar, kimham@kth.se

November 17, 2017

1 Solution

We implemented the TRIÈST-IMPR algorithm[1] for estimating triangle counts on the Euroroad graph dataset [2]. The graph is undirected, nodes represent cities and an edge between two nodes denotes that they are connected by an E-road. The algorithm estimates both global and local triangle counts. The dataset is bounded but we process it in a streaming fashion by reading edge by edge and applying the reservoir sampling.

2 Questions

1. *What were the challenges you have faced when implementing the algorithm?*

   One problem that we encountered was about choosing the right streaming graph processing platform. To the best of our knowledge Apache Flink does not support streaming graph processing. We tried to use plain Apache Flink and stream every event as an edge in our graph but it was not clear to us how our TRIÈST-IMPR counters were updated since Flink uses updated counters in parallel way. Flink streaming typically considers the data as unbounded and if using this approach we would generate estimates per window rather than a global estimate of the triangle count. Since our dataset in this were bounded it would over-complicate things to use Flink so we simply streamed the edges ourself in a non-parallel fashion.

2. *Can the algorithm be easily parallelized? If yes, how? If not, why? Explain.*

   Yes it can be parallelized, different streaming nodes can run the TRIÈST-IMPR algorithm in parallel and maintain local estimates. When querying the stream or materializing the results, the local estimates have to be merged to create the final estimate. This can for example be done in Flink-Streaming, where the stream of edges can be split uniformly among a set of nodes and each node updates its local sample and estimates.

3. *Does the algorithm work for unbounded graph streams? Explain.*

   Yes, since the algorithm uses reservoir sampling it is meant to be used for unbounded graph streams. The difference if the stream of edges is unbounded is that the mindset have to be shifted. With a unbounded stream we cannot wait until all edges have been received to materialize the estimates but rather some form of windowing strategy should be applied to construct rolling estimates for given time periods. What notion of time to use depends on the characteristics of the stream, for instance if the edges are time-stamped we could use event-time, otherwise we could use processing-time. Data-driven windows are also a possibility.

4. *Does the algorithm support edge deletions? If not, what modification would it need? Explain.*

   TRIÈST-IMPR does not support edge deletions. Our stream for this lab did not contain any edge deletions so we did not implement the edge deletion part. To extend the algorithm for edge deletion, two counters, d_i and d_o should be maintained to keep track of how many times edges have been removed versus in the stream. This is necessary since the stream

[1] http://www.kdd.org/kdd2016/papers/files/rfp0465-de-stefaniA.pdf
[2] http://konect.uni-koblenz.de/networks/subelj_euroroad

might arrive unordered. The reservoir should only keep those edges that have been inserted more times than deleted. Effectively the counters works like a sort of tombstone.

3 How to run

Clone this repository and navigate to *mining_ data_ streams* project. Then use:

```
sbt compile //compile
sbt test //test
sbt run //run
sbt assembly //generate fat jar
```

4 Evaluation and results

Number of Edges in a Sample, M	Estimated Number of Triangles	Actual Number of Triangles
350	20	32
750	28	32
1000	30	32
1100	32	32

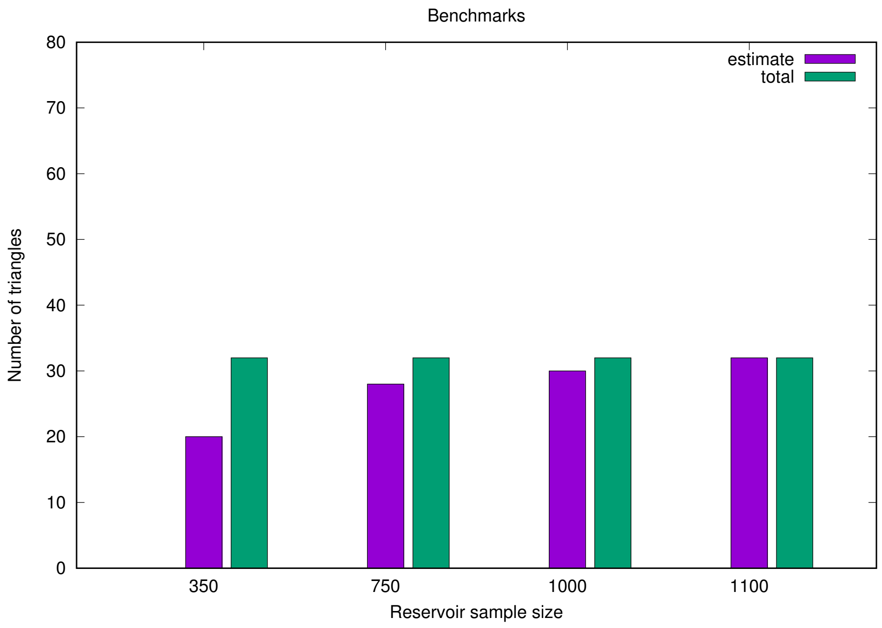


Figure 1: Estimate accuracy as sample size increases, when sample size is 1100 the estimate is correct (32).

The number of total edges in the graph we used is 1417, total number of vertices is 1,174 and average degree is ≈ 2.4. As we increase number of edges in the sample the precision of our implementation gets better. This is based on the second implementation of the TRIÈST-IMPR algorithm. The intuition behind this is that the algorithm was meant to be use at the very large graphs (number of edges order of 10^9) and precision gets better if number of edges in the reservoir increases. As we saw in the actual paper, in order to estimate number of triangles for the Twitter network graph (contains billions of edges), the authors choose very high M, $M = 10^6$.

ID2222 Data Mining

Homework 4: Graph Spectra

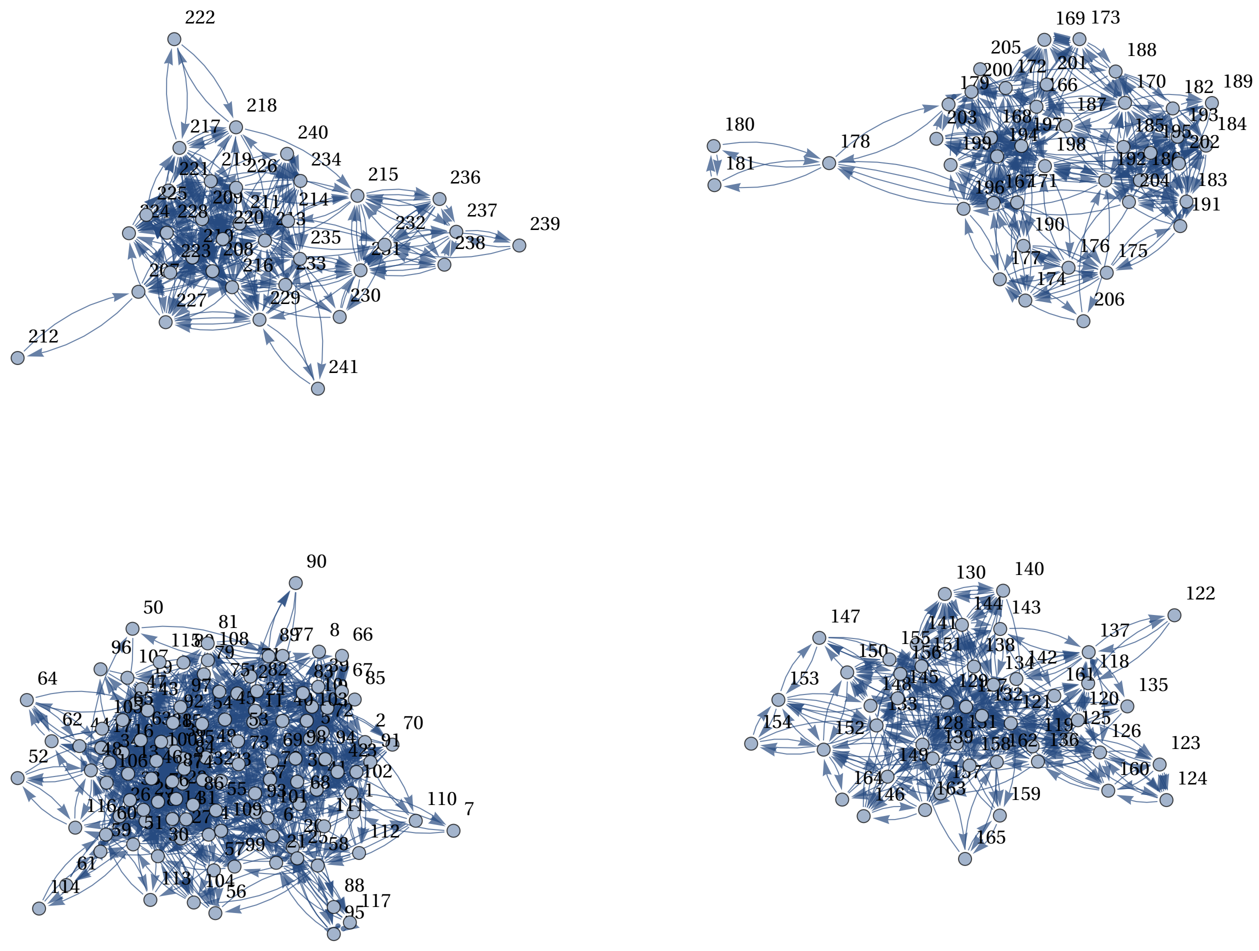
Graph 1

Kim Hammar
KTH Royal Institute of Technology

Konstantin Sozinov
KTH Royal Institute of Technology

Graph Import

In[65]:=
```
SetDirectory[NotebookDirectory[]];
edgeList = Import["example1.csv","Data"];
graph = Graph[DirectedEdge@@@ edgeList,VertexLabels→"Name"];
```



General Graph Properties

Edge Count

```
In[68]:= EdgeCount[graph];
```

```
2196
```

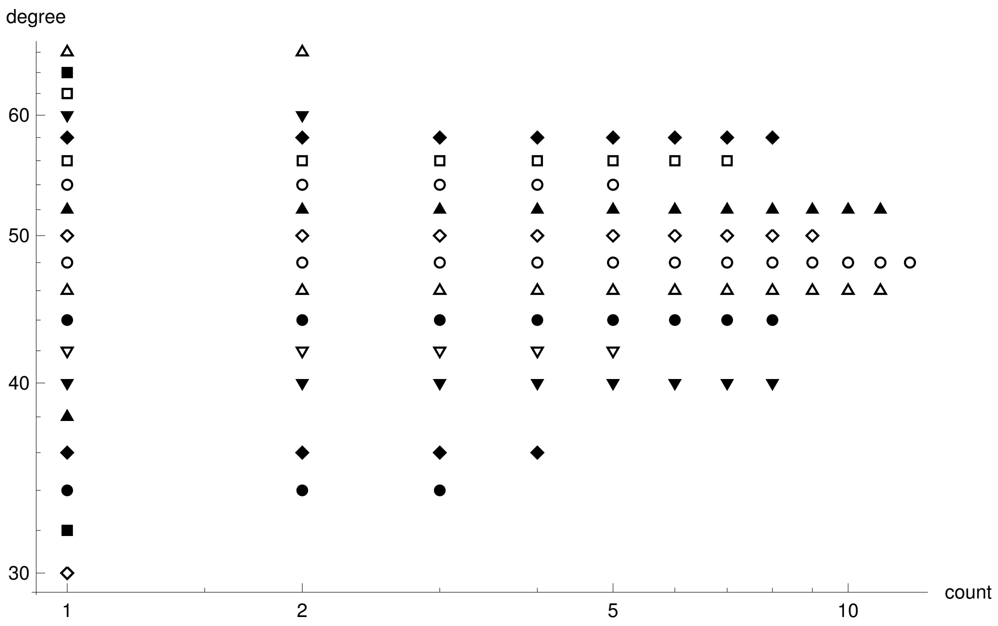
Vertex Count

```
In[69]:= VertexCount[graph];
```

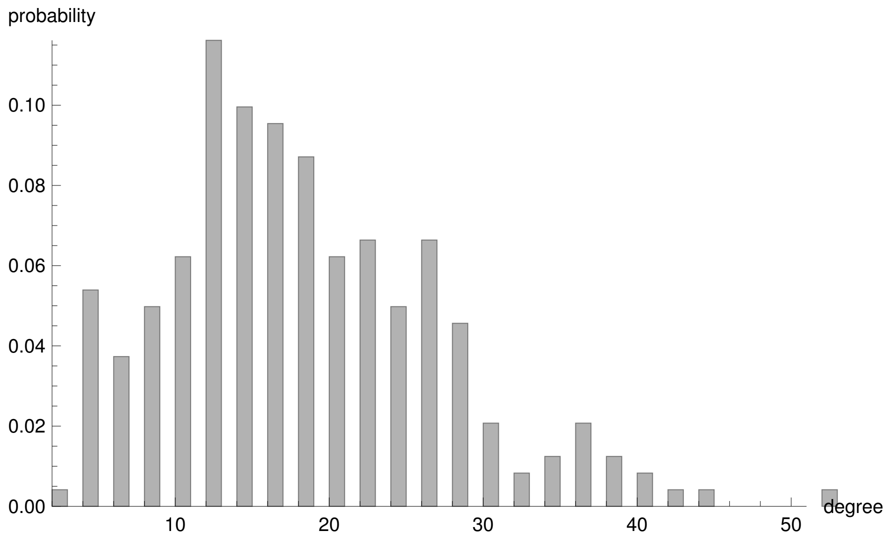
```
241
```

Degree Distribution

In[70]:=
```
Histogram[VertexDegree[graph],{1},"Probability",AxesLabel→{"degree","probability"}];
ListLogLogPlot[GroupBy[VertexDegree[graph], Count], AxesLabel→{"count","degree"}];
```



Log-Log plot over degree distribution to do a rough test for powerlaw. Since the distribution is not linear it is probably not a powerlaw.



Regular Histogram plot over degree distribution.

Global Clustering Coefficient

In[72]:=
```
GlobalClusteringCoefficient[graph];
```

$$\frac{1008}{4013}$$

Graph Communities

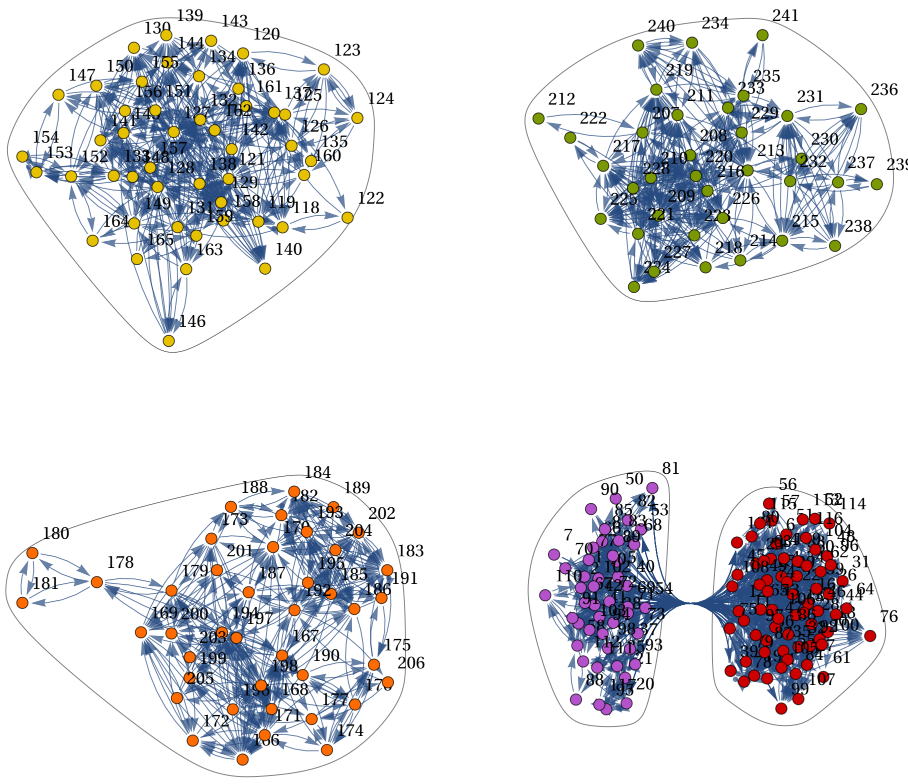
Communities Count

In[73]:=
```
Length[FindGraphCommunities[graph]];
```

```
5
```

Communities Plot

In[74]:=
```
CommunityGraphPlot[graph];
```



Graph Spectra

Graph Spectra

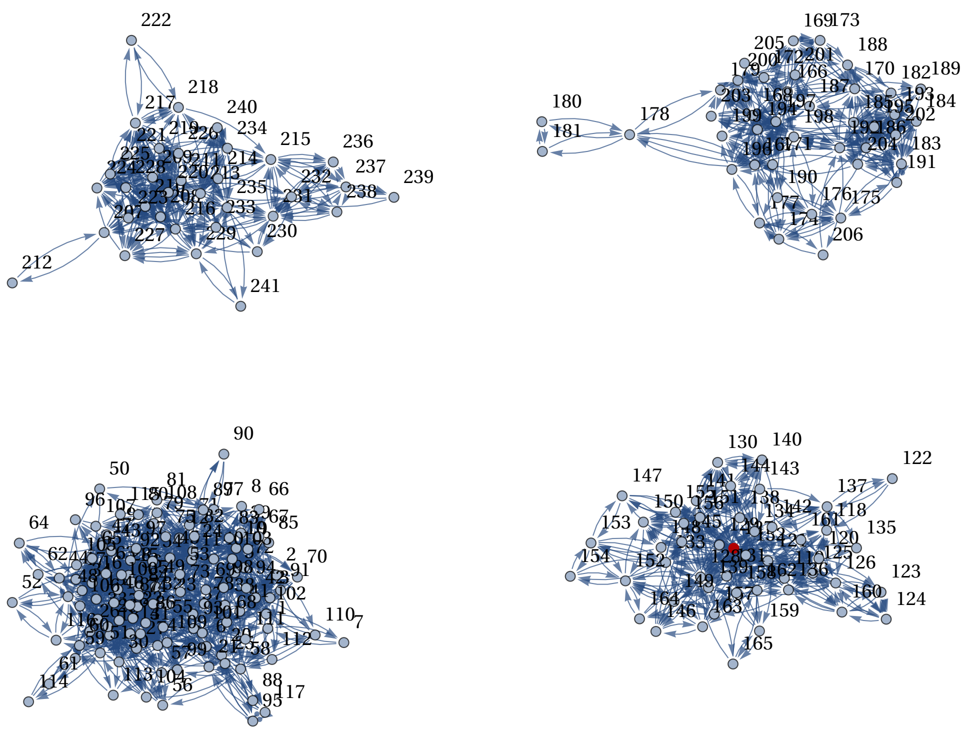
In[75]:=
```
A = AdjacencyMatrix[graph];
{eigenVals,eigenVecs} =Eigensystem[N[A]];
```

Node Centralities

PageRank Centrality

In[77]:=
```
MaxPageRankCentralNode = VertexList[graph][[Position[PageRankCentrality[graph],
Max[PageRankCentrality[graph]]][[1]]]];
HighlightGraph[graph, MaxPageRankCentralNode];
```

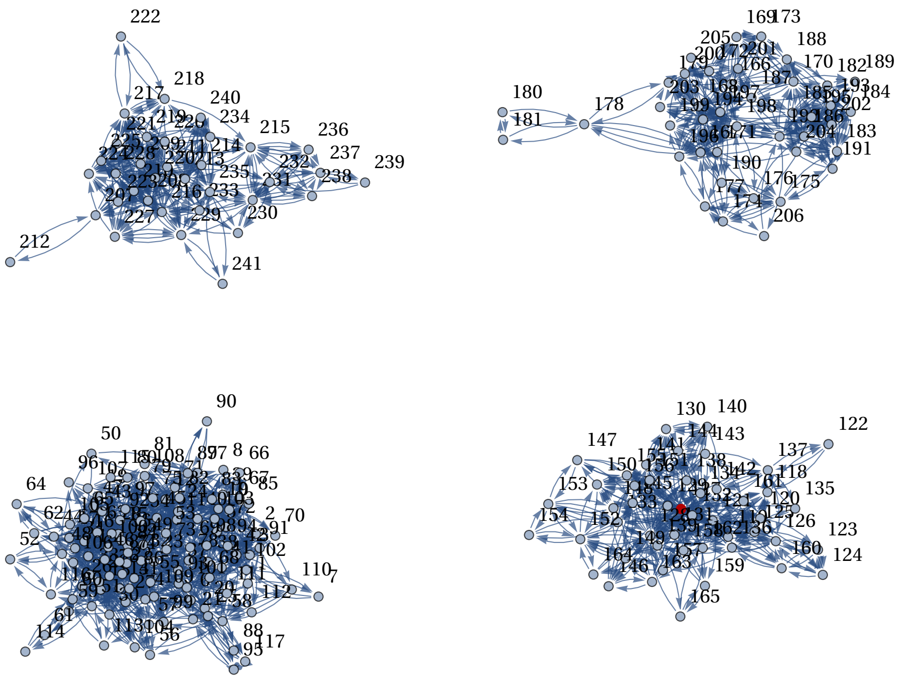
```
{127}
```



Degree Centrality

In[79]:=
```
MaxDegreeCentralNode = VertexList[graph][[Position[DegreeCentrality[graph],
Max[DegreeCentrality[graph]]][[1]]]];
HighlightGraph[graph, MaxDegreeCentralNode];
```

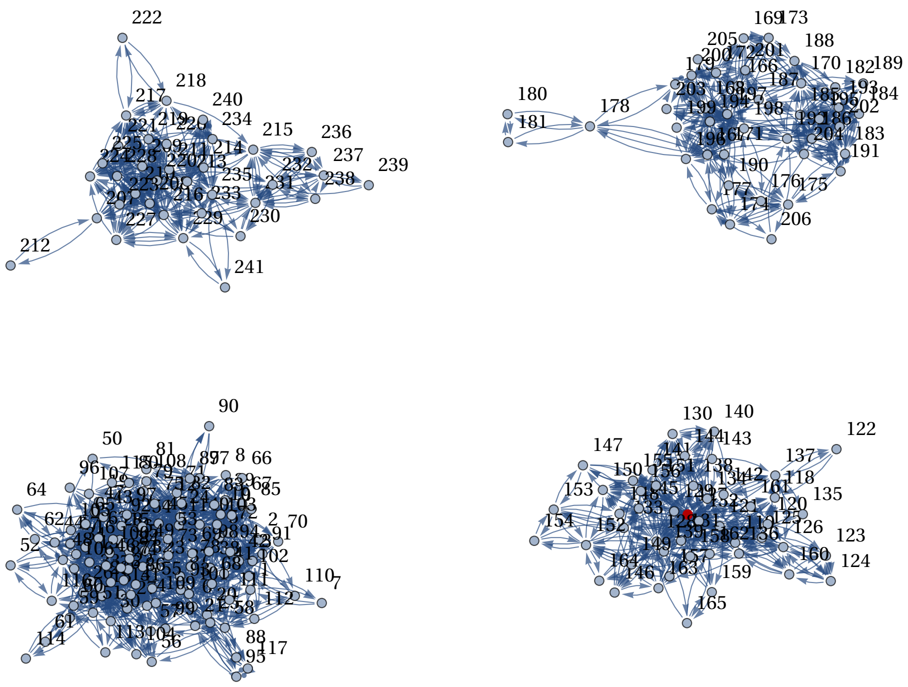
```
{127}
```



Closeness Centrality

In[81]:=
```
MaxClosenessCentralityNode = VertexList[graph][[Position[ClosenessCentrality[graph],
Max[ClosenessCentrality[graph]]][[1]]]];
HighlightGraph[graph, MaxClosenessCentralityNode];
```

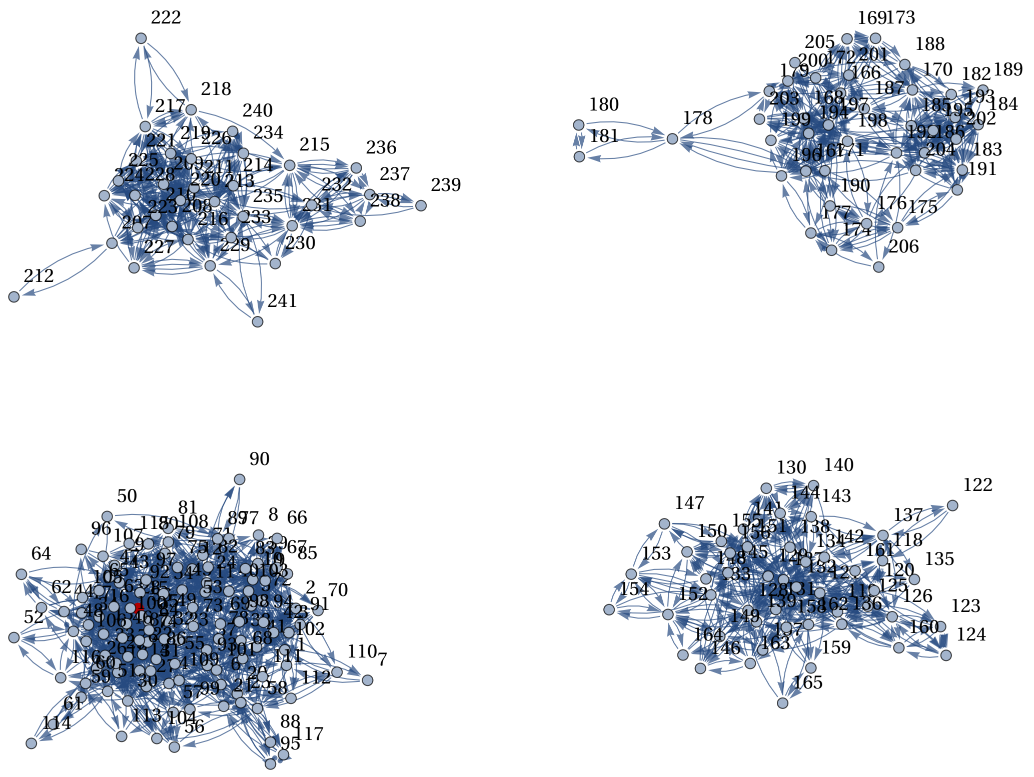
```
{127}
```



Betweenness Centrality

In[83]:=
```
MaxBetweenessCentralityNode = VertexList[graph][[Position[BetweennessCentrality[graph],
Max[BetweennessCentrality[graph]]][[1]]]];
HighlightGraph[graph, MaxBetweenessCentralityNode];
```

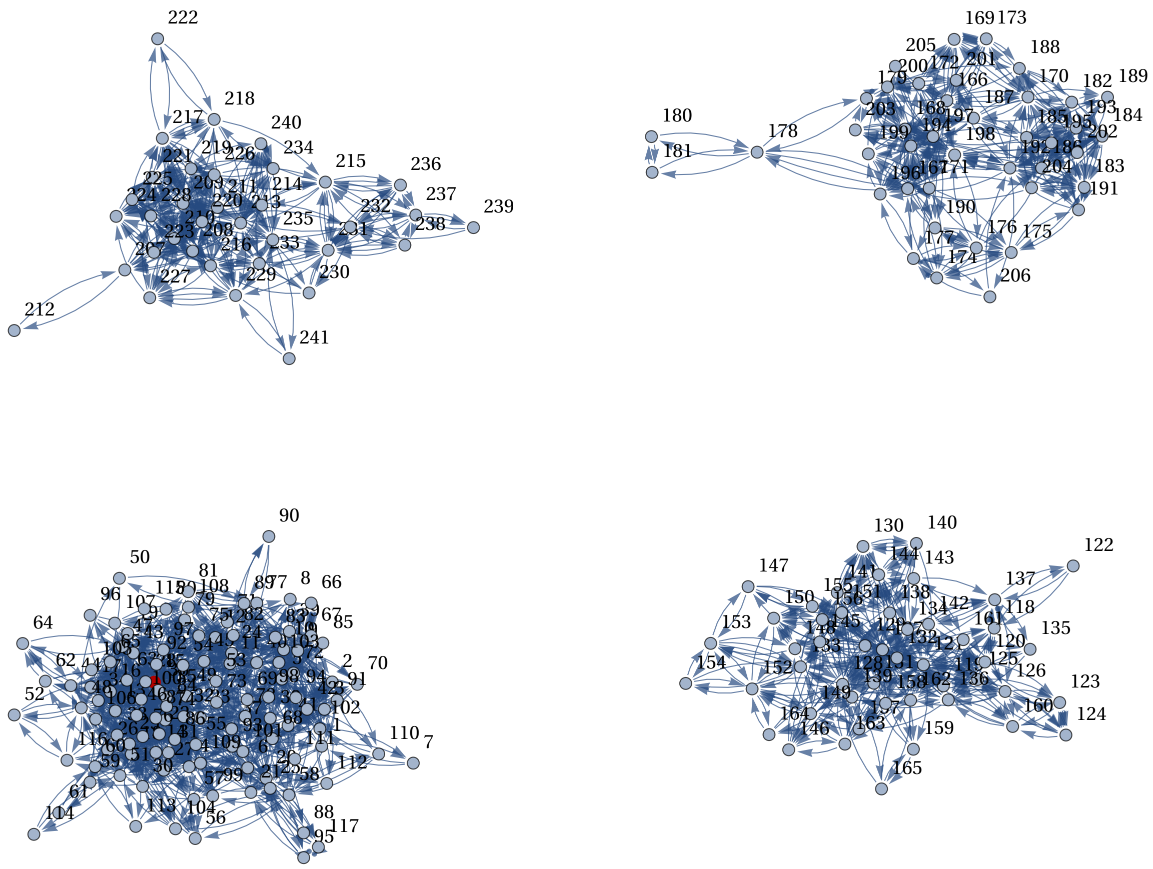
```
{15}
```



EigenVector Centrality

```
In[85]:= MaxEigenVectorCentralityNode = VertexList[graph][[Position[EigenvectorCentrality[graph]
         Max[EigenvectorCentrality[graph]]][[1]]]];
         HighlightGraph[graph, MaxEigenVectorCentralityNode];
```

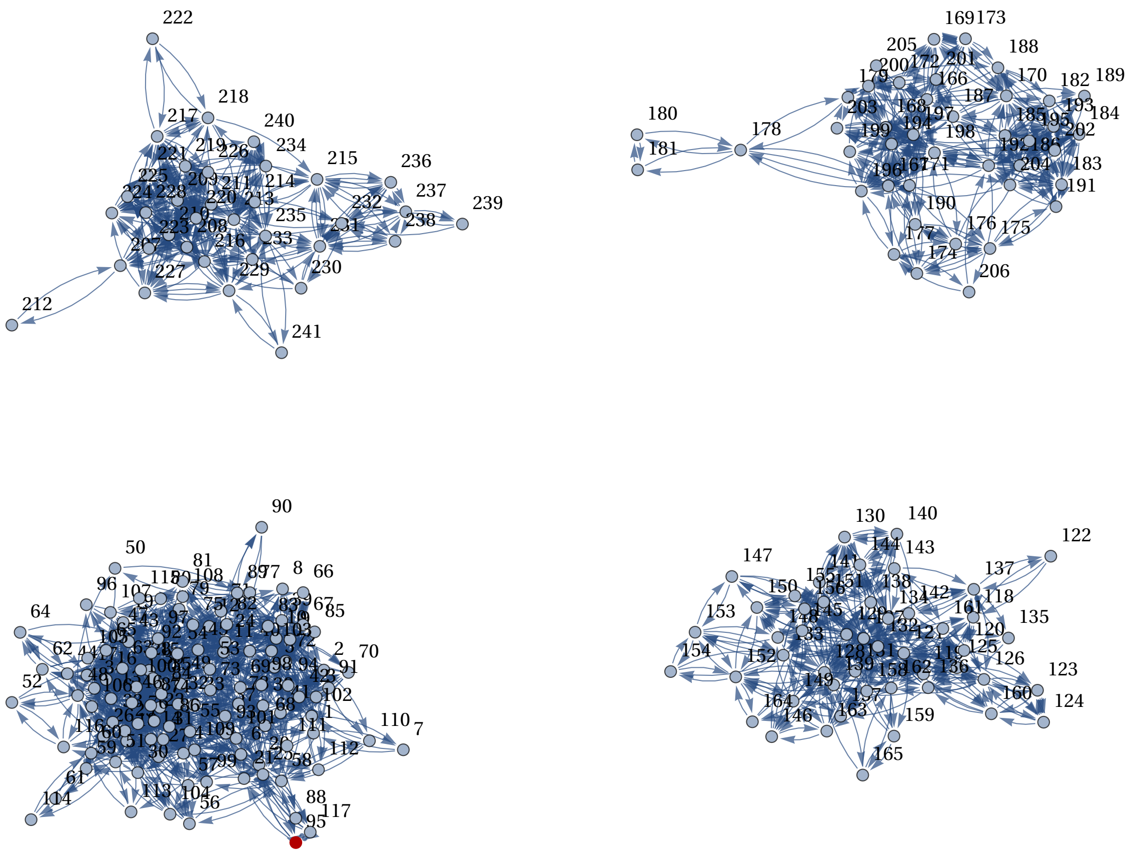
```
{15}
```



Clustering Coefficient

In[87]:=
```
MaxClusterNode = VertexList[graph][[Position[LocalClusteringCoefficient[graph],
Max[LocalClusteringCoefficient[graph]]][[1]]]];
HighlightGraph[graph, MaxClusterNode];
```

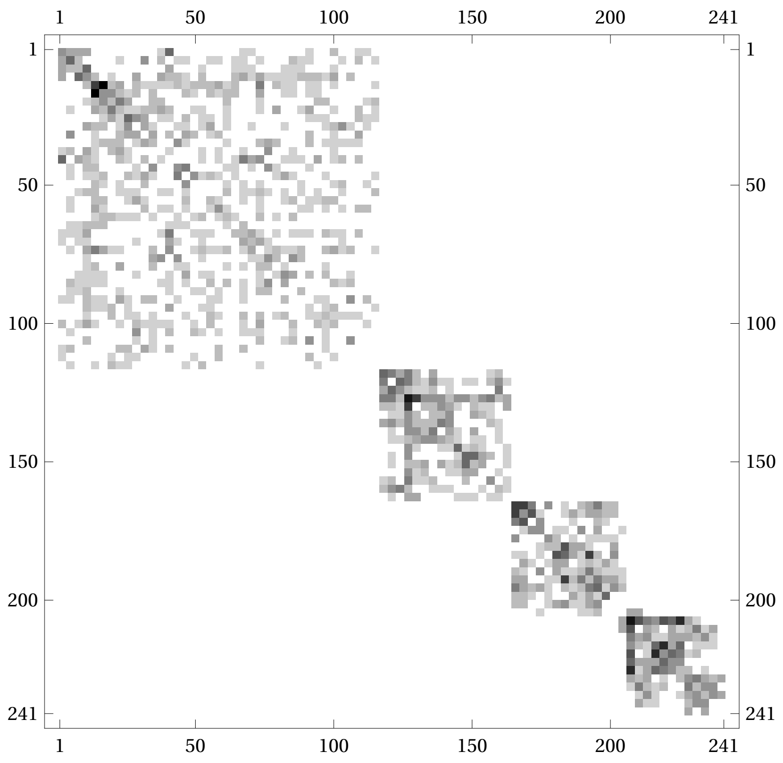
```
{95}
```



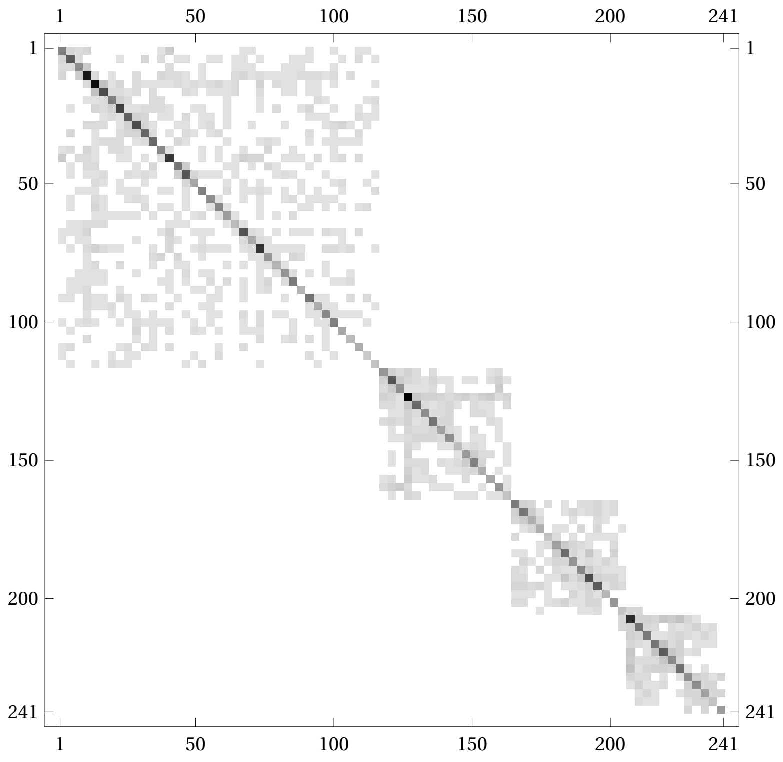
Adjacency Matrix Heatmap, Non-Normalized Laplacian (Kirchoff), Affinity Matrix

Plotting the heatmap of the adjacency matrix is a common way to visualize a network, it is especially effective when the node partitons are ordered based on node-ids. In our case the partitions are perfectly ordered sequentially on the node ids so the heatmap gives a good indication of the number of clusters.
There exists multiple versions of the Laplacian matrix with small modifications, the Kirchoff matrix is one of them. The affinity matrix is computed as the Jordan Decomposition of the Adjaceny matrix. There are many spectrums that are useful for graph analysis, including the spectrum of the adjacency matrix, the transition matrix and the Laplacian matrix. However, the Laplacian matrix is the most useful for reasoning about the connectivity of the graph as well as its clustering.

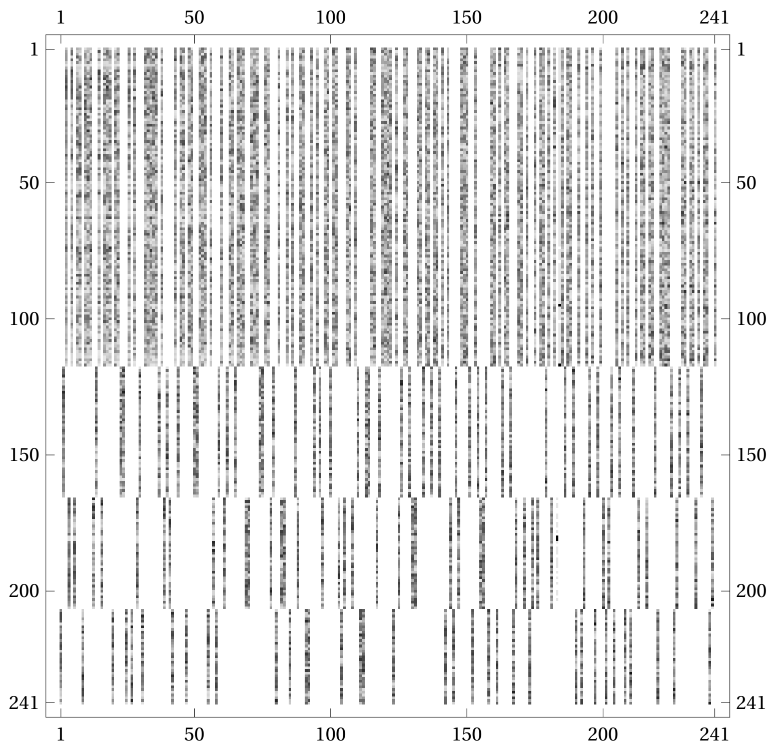
In[89]:=
```
MatrixPlot[A];
kirchoffLaplacianMatrix = KirchhoffMatrix[graph];
MatrixPlot[kirchoffLaplacianMatrix];
{affinityMatrix, s} = JordanDecomposition[N[Transpose[A]]];
MatrixPlot[affinityMatrix];
```



Adjacency Matrix plot



KirchoffLaplacian Matrix (not normalized)

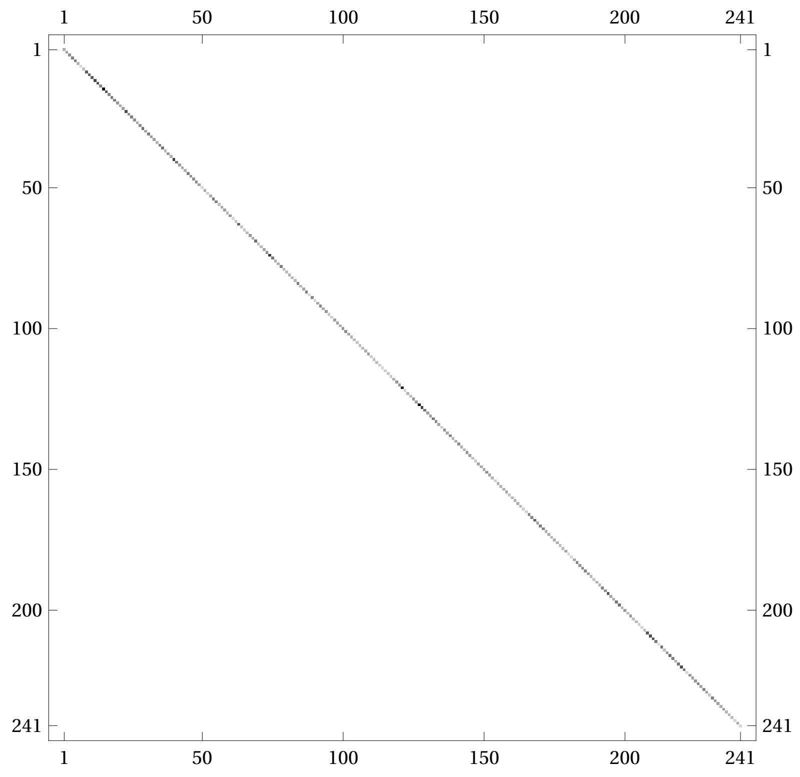


Affinity Matrix Plot

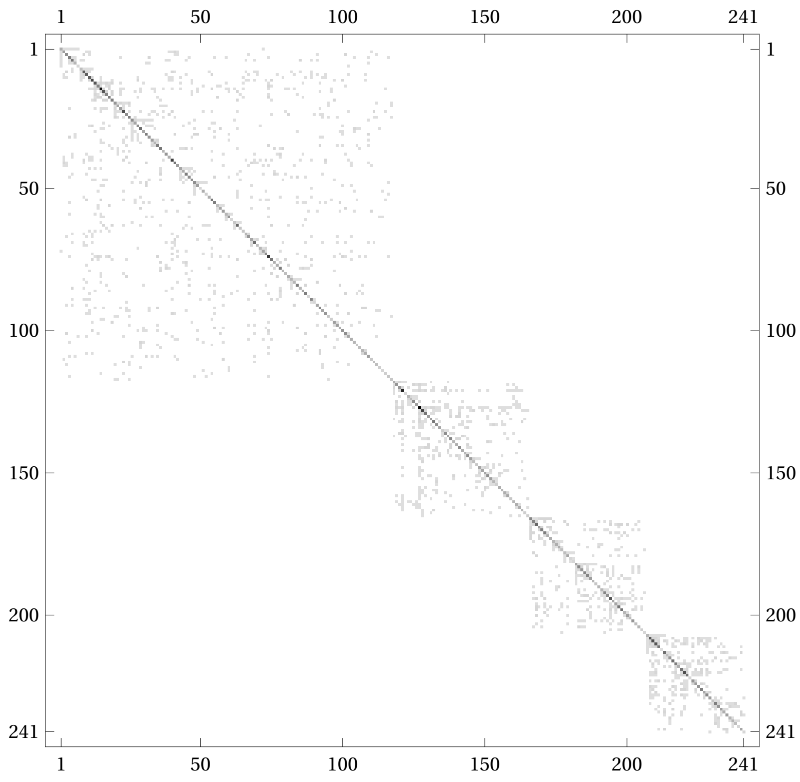
Degree Matrix, Simple Laplacian

The Degree Matrix D is a diagonal matrix with the degree of each node i on the diagonal (i,i). Here I also compute the simple Laplacian matrix which is L = D - A.

```
In[94]:= {n,n} = Dimensions[A];
DegreeMatrix = ConstantArray[0, {n,n}];
For[i = 1, i <= n, i++, DegreeMatrix[[i,i]] = Total[A[[i]]]];
MatrixPlot[DegreeMatrix];
simpleLaplacian = DegreeMatrix - A;
MatrixPlot[simpleLaplacian];
```



Degree Matrix Plot



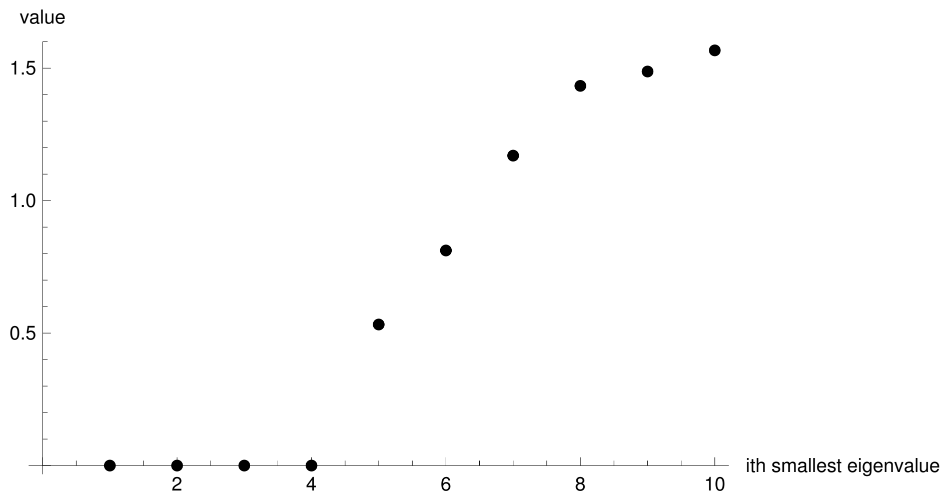
Simple Laplacian Matrix Plot

EigenGap of Simple Laplacian and Fiedler Vector

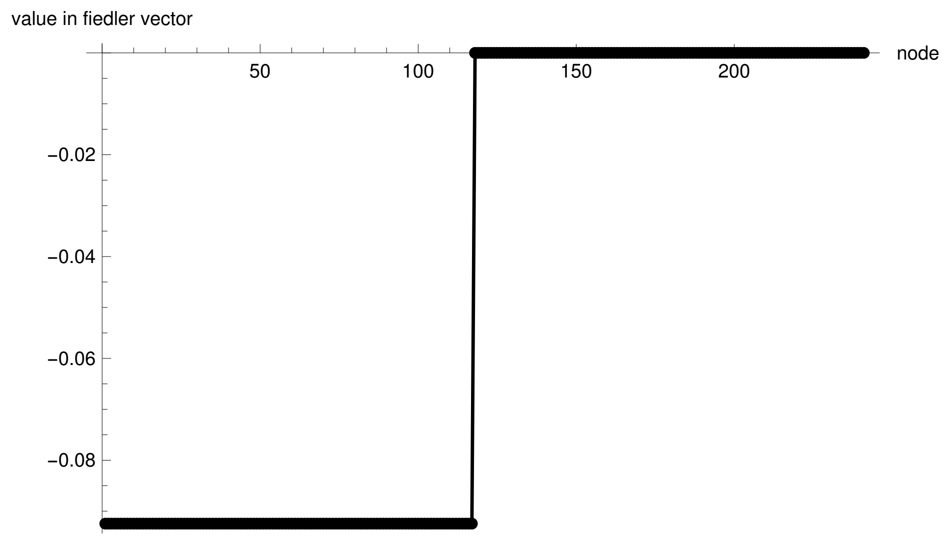
In the Laplacian we know that $\lambda_1 = 0$ with a corresponding eigenvector $v_1 = [1, \ldots, 1]$. This follows from the fact that the rows and the columns of the Laplacian sum up to 0 (each row contains number of -1 as number of neighbors plus one entry on the row which is the degree of the node). Further more we can tell by analyzing the higher-order eigenvalues how many connected components the graph has and whether it is a good expander or not. In our case $\lambda_1 = \lambda_2 = \lambda_3 = \lambda_4 = 0$ which means that the graph has 4 connected components. We also know from spectral graph theory that $\lambda_{n \leq 2}$which is in concordance with our results below. Furthermore, by looking at the eigen-gap between $\lambda_4$and $\lambda_5$we can tell whether it is close or not that there is a fifth disconnected component, and we can see that the fifth components seems to be quite well connected since the eigen-gap is quite large.

Finally, the eigenvector associated with λ_2, also know as the “Fiedler Vector”, can give a bi-partition of the graph. The Fiedler does not indicate the k-clusters but it can indicate the optimal 2-clusters (and if applied recursively it can even find k clusters, but typically to exploit higher order eigenvectors is a better approach).

In[179]:=
```
{smallestEigenVals, smallestEigenVecs} = Eigensystem[N[simpleLaplacian], -10];
ListPlot[Reverse[Chop[smallestEigenVals]], AxesLabel→{"ith smallest eigenvalue","value"
fiedlerVector = smallestEigenVecs[[-2]];
ListLinePlot[Sort[fiedlerVector], AxesLabel→{"node","value in fiedler vector"}];
```



We can see that there is a gap between the 4th smallest eigenvalue and the 5th smallest of the simple Laplacian. This indicates that there are 4 clusters.

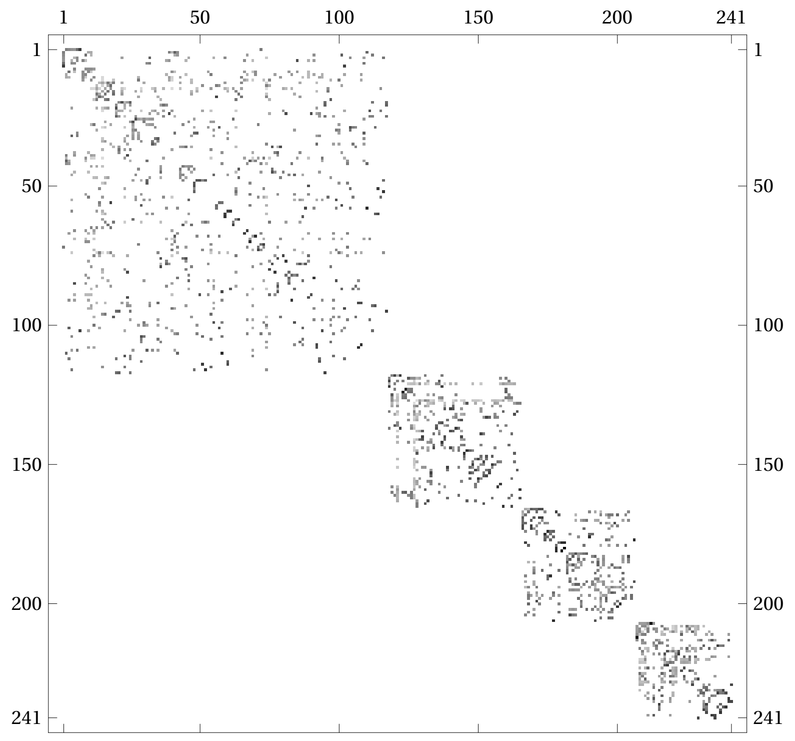


Sorted Fiedler Vector plot, gives a near optimal 2-way partitioning by assigning nodes to partitions based on their index in the Fiedler vector and the sign of the value in the vector.

Ng, Jordan, & Weiss Laplacian

As mentioned, there are many variants of the laplacian matrix used in different contexts, they all have the same basic characteristics and differ only slightly. In the code snippet below the Laplacian of the Ng, Jordan & Weiss paper is computed as L = $D^{-1/2}$ AD$^{-1/2}$

In[104]:=
```
k = 4;
{n,n} = Dimensions[A];
njwLaplacian = MatrixPower[DegreeMatrix,-(1/2)].(A.MatrixPower[DegreeMatrix, -(1/2)]);
MatrixPlot[njwLaplacian];
```



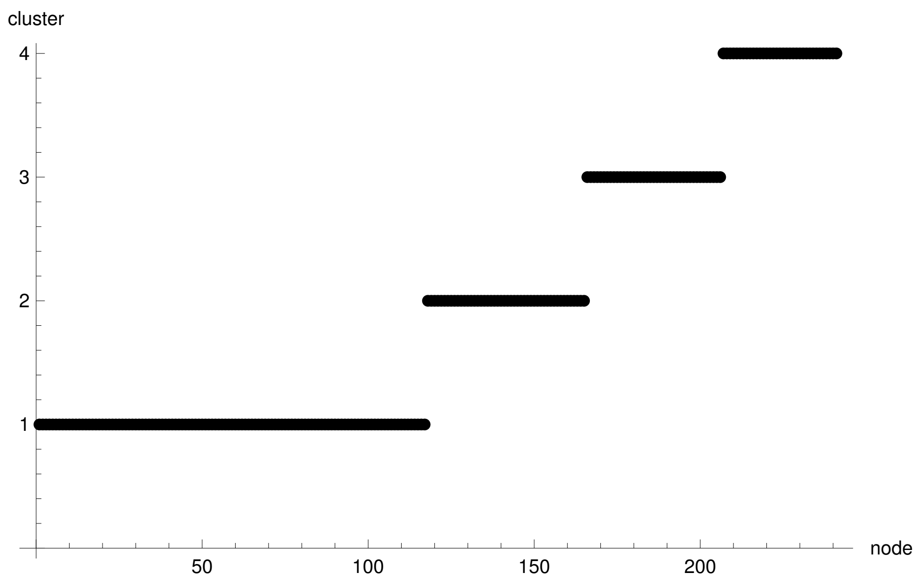
Ng, Jordan, & Weiss Laplacian

Ng, Jordan, & Weiss Spectral Clustering

Here the bulk of the algorithm in the paper is implemented.

1. Decide k, we chose k = 4 since this was obvious from the preprocessing, e.g there are 4 connected components for instance.
2. The eigendecomposition of the Laplacian is computed
3. X is formed as a matrix with columns being the k largest eigenvectors
4. Define Y as X with all columns normalized to unit length
5. Cluster Y with K-means (a row in Y is considered as a datapoint to cluster)
6. Assign the original datapoints (from the adjacency matrix) to their clusters based on what their corresponding row in Y was clustered as.

In[108]:=
```
k = 4;
{largestEigenVals, largestEigenVecs} = Eigensystem[N[njwLaplacian],4];
X = Transpose[largestEigenVecs];
MatrixPlot[X];
{rows,cols} = Dimensions[X];
Y = ConstantArray[0, {rows,cols}];
For[i = 1, i <= cols, i++, Y[[All,i]] = Normalize[X[[All, i]]]];
clusters = ClusteringComponents[Y,k,1, Method→ "KMeans"];
ListPlot[clusters, AxesLabel→{"node","cluster"}];
```



As can be seen from the plot , the 4 clusters found by the spectral clustering are:

Cluster 1: Approximately 0-120

Cluster 2 : Approximately: 120-170

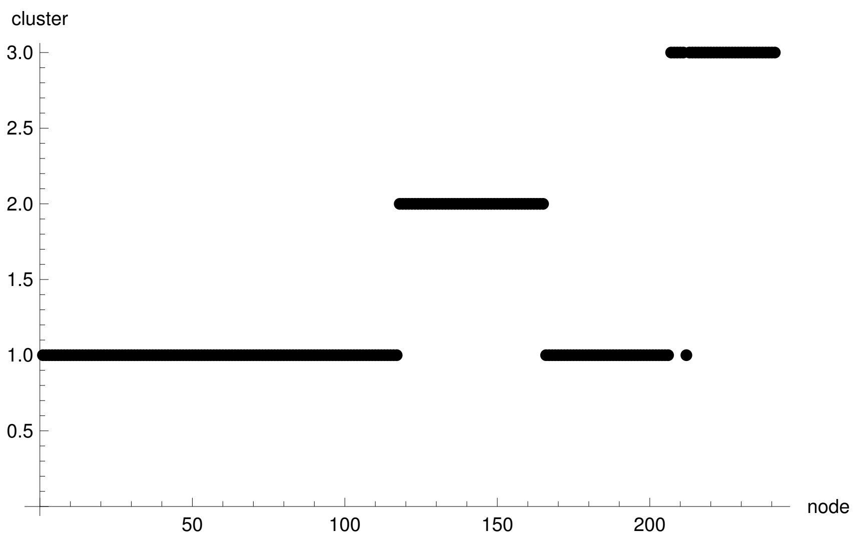
Cluster 3 : Approximately: 170-210

Cluster 4: Approximately: 210 -241

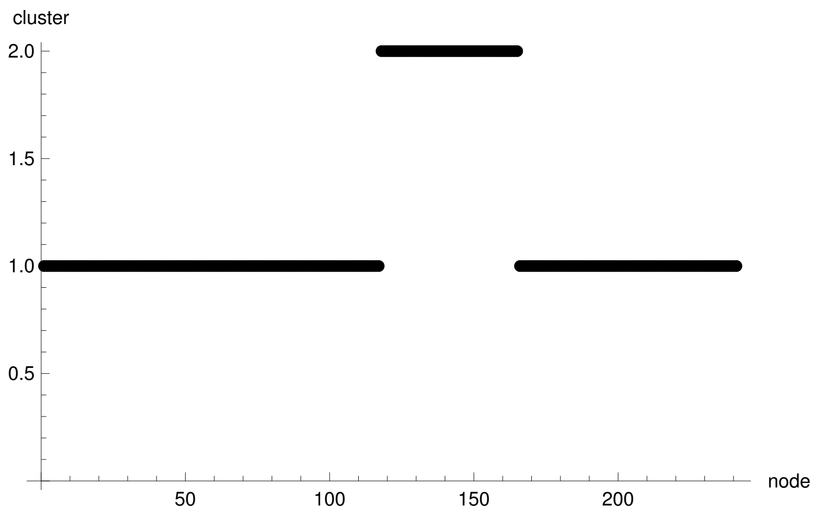
Test clustering with “wrong” k

In[117]:=

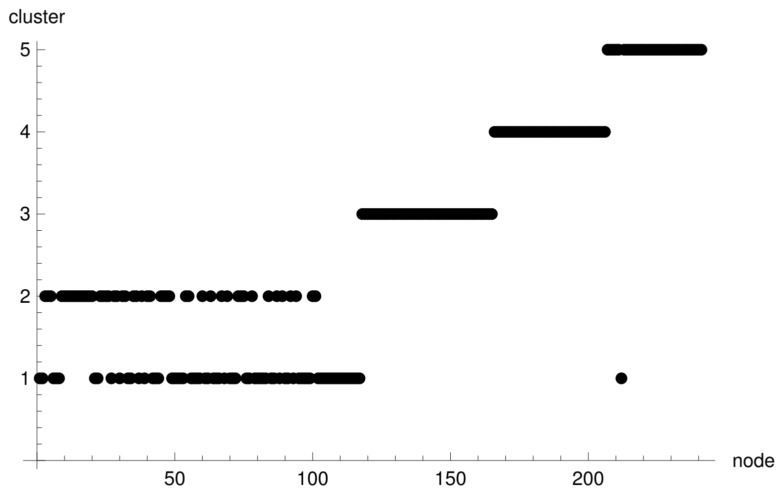
```
k = 3;
clusters = ClusteringComponents[Y,k,1, Method→ "KMeans"];
ListPlot[clusters, AxesLabel→{"node","cluster"}];
k = 2;
clusters = ClusteringComponents[Y,k,1, Method→ "KMeans"];
ListPlot[clusters, AxesLabel→{"node","cluster"}];
k = 5;
clusters = ClusteringComponents[Y,k,1, Method→ "KMeans"];
ListPlot[clusters, AxesLabel→{"node","cluster"}];
k = 6;
clusters = ClusteringComponents[Y,k,1, Method→ "KMeans"];
ListPlot[clusters, AxesLabel→{"node","cluster"}];
```


cluster
3.0
2.5
2.0
1.5
1.0
0.5
50
100
150
200
node

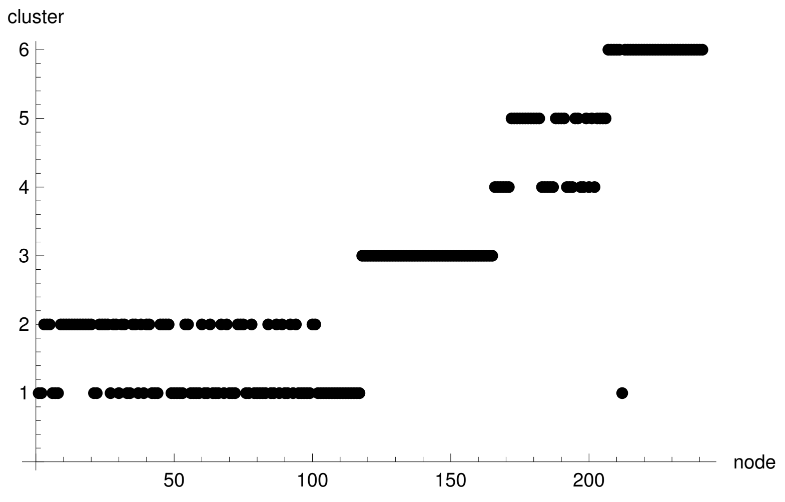
k = 3


cluster
2.0
1.5
1.0
0.5
50
100
150
200
node

k = 2


cluster
5
4
3
2
1
50
100
150
200
node

k = 5



k = 6

ID2222 Data Mining

Homework 4: Graph Spectra

Graph 2

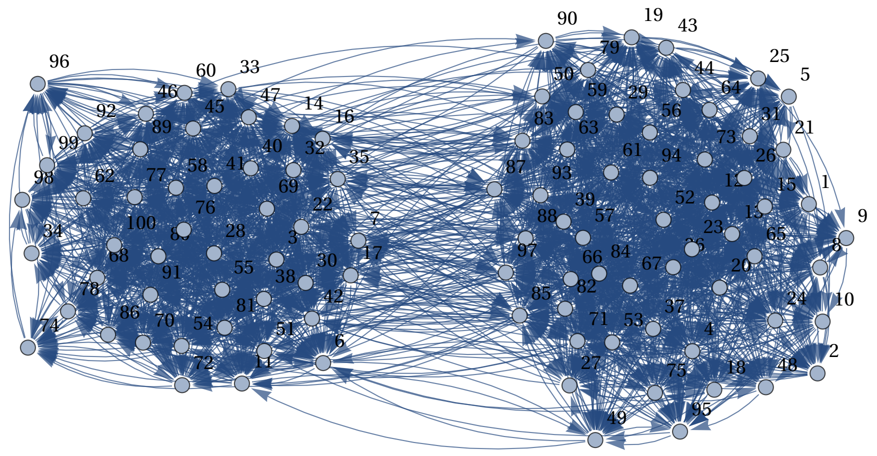
Kim Hammar
KTH Royal Institute of Technology

Konstantin Sozinov
KTH Royal Institute of Technology

Graph Import

In[247]:=

```
SetDirectory[NotebookDirectory[]];
edgeList = Import["example2_clean.csv","Data"];
graph = Graph[DirectedEdge@@@ edgeList,VertexLabels→"Name"];
```



General Graph Properties

Edge Count

In[250]:=
```
EdgeCount[graph];
```

```
2418
```

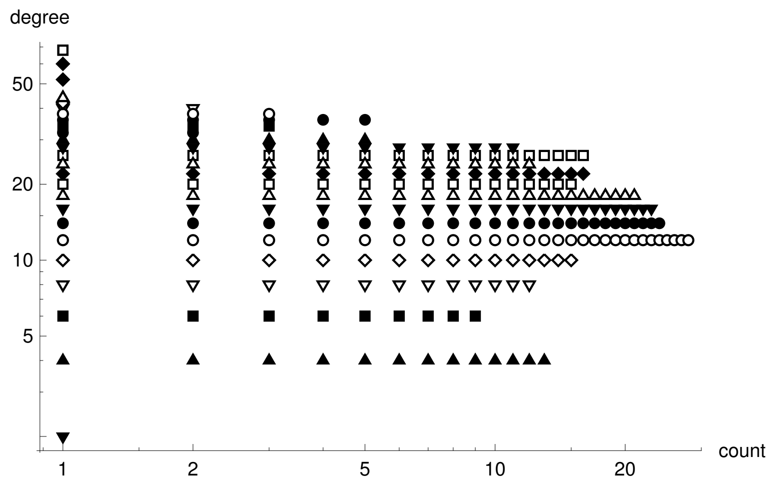
Vertex Count

In[251]:=
```
VertexCount[graph];
```

```
100
```

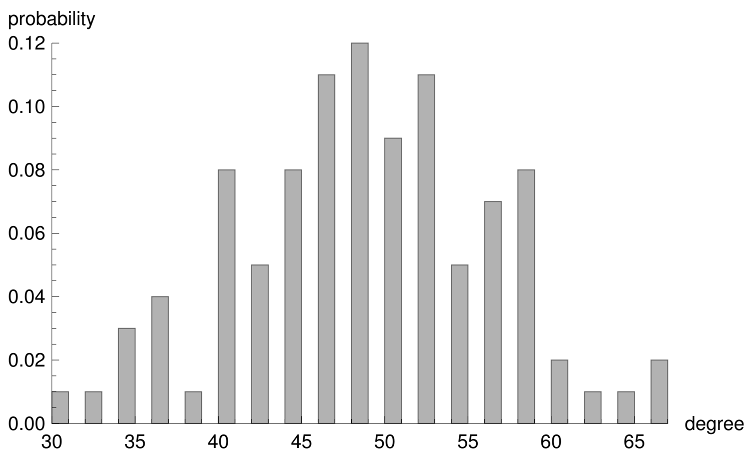
Degree Distribution

In[252]:=
```
Histogram[VertexDegree[graph],{1},"Probability",AxesLabel→{"degree","probability"}];
ListLogLogPlot[GroupBy[VertexDegree[graph], Count], AxesLabel→{"count","degree"}];
```



Log-Log plot over degree distribution to do a rough test for powerlaw. The distribution has some linear tendency but not clear enough to be powerlaw.

In[254]:=



Global Clustering Coefficient

In[255]:=
```
GlobalClusteringCoefficient[graph];
```

$$\frac{3649}{9580}$$

Graph Communities

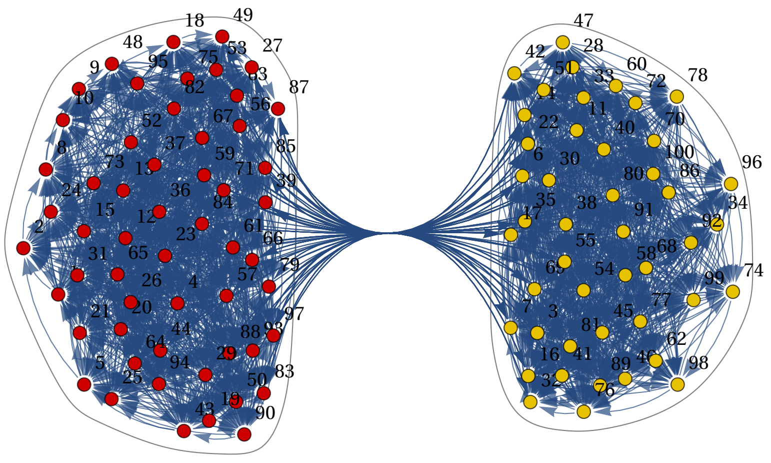
Communities Count

In[256]:=
```
Length[FindGraphCommunities[graph]];
```

```
2
```

Communities Plot

In[257]:=
```
CommunityGraphPlot[graph];
```



Graph Spectra

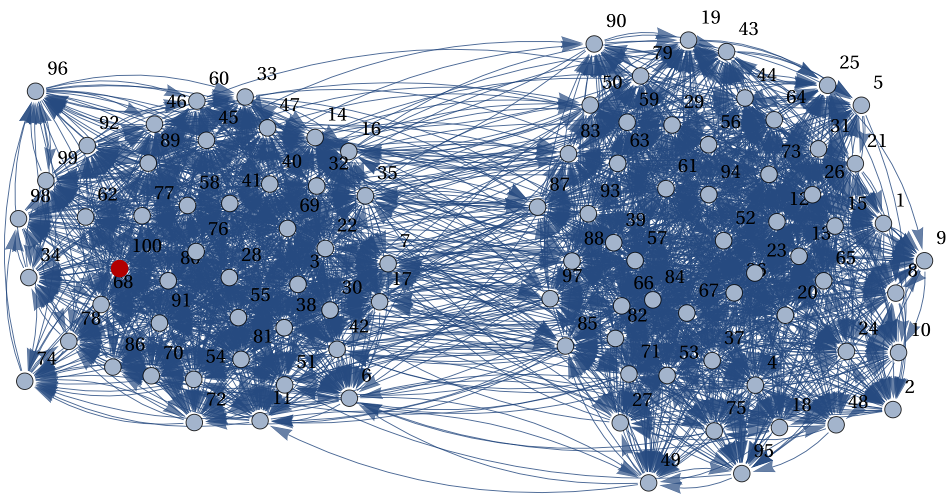
Graph Spectra

In[258]:=
```
A = AdjacencyMatrix[graph];
{eigenVals,eigenVecs} =Eigensystem[N[A]];
```

Node Centralities

PageRank Centrality

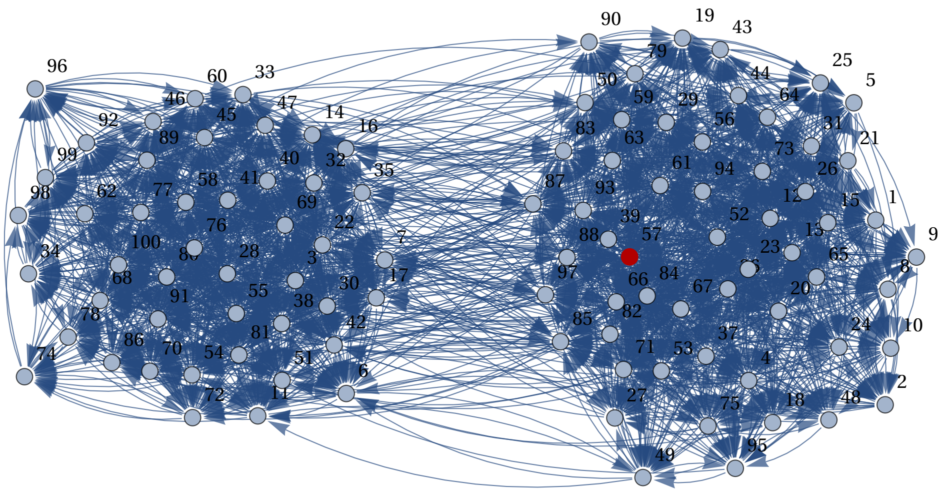
In[260]:=
```
MaxPageRankCentralNode = VertexList[graph][[Position[PageRankCentrality[graph],
Max[PageRankCentrality[graph]]][[1]]]];
HighlightGraph[graph, MaxPageRankCentralNode];
```



Degree Centrality

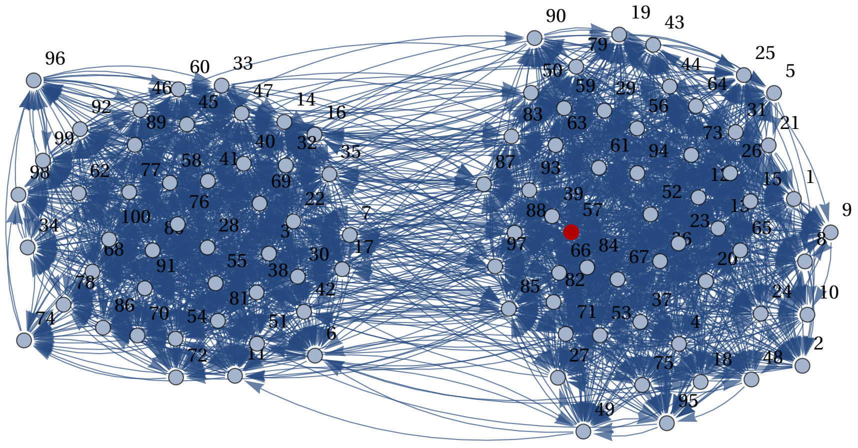
In[262]:=
```
MaxDegreeCentralNode = VertexList[graph][[Position[DegreeCentrality[graph],
Max[DegreeCentrality[graph]]][[1]]]];
HighlightGraph[graph, MaxDegreeCentralNode];
```

In[264]:=



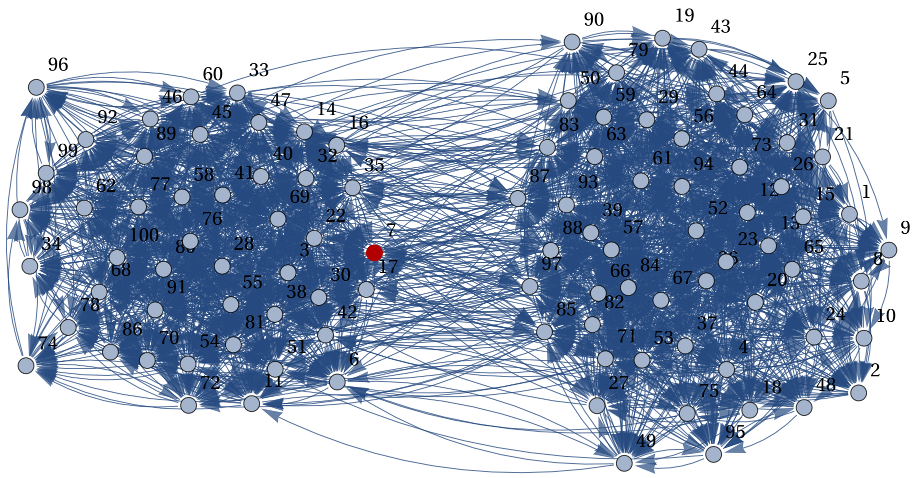
Closeness Centrality

In[265]:=
```
MaxClosenessCentralityNode = VertexList[graph][[Position[ClosenessCentrality[graph],
Max[ClosenessCentrality[graph]]][[1]]]];
HighlightGraph[graph, MaxClosenessCentralityNode];
```



Betweenness Centrality

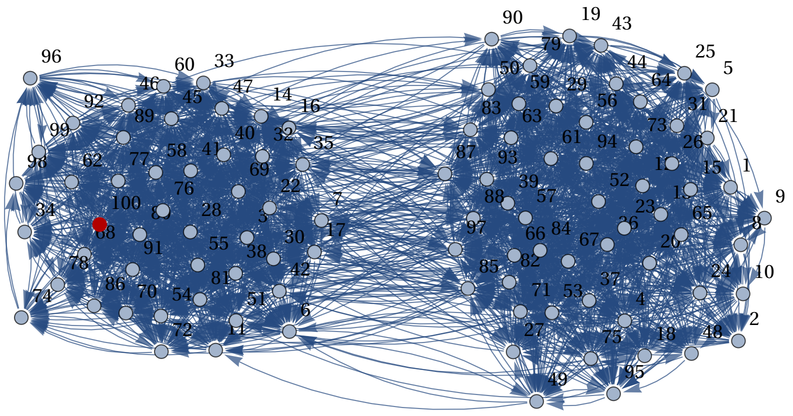
In[267]:=
```
MaxBetweenessCentralityNode = VertexList[graph][[Position[BetweennessCentrality[graph],
Max[BetweennessCentrality[graph]]][[1]]]];
HighlightGraph[graph, MaxBetweenessCentralityNode];
```



```
{15}
```

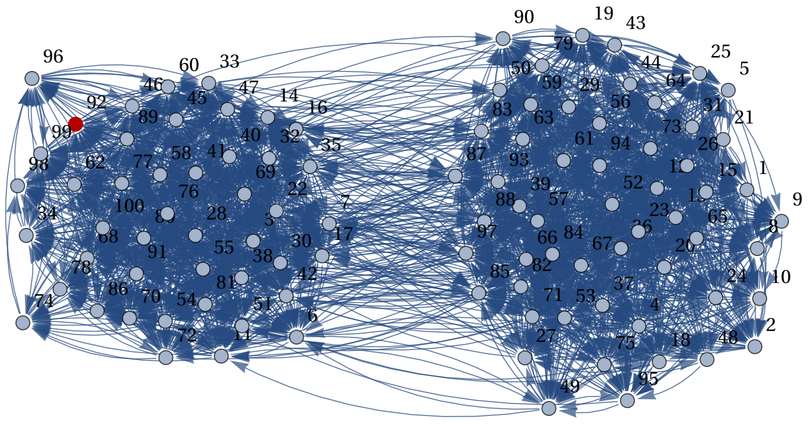
EigenVector Centrality

In[269]:=
```
MaxEigenVectorCentralityNode = VertexList[graph][[Position[EigenvectorCentrality[graph]
Max[EigenvectorCentrality[graph]]][[1]]]];
HighlightGraph[graph, MaxEigenVectorCentralityNode];
```



Clustering Coefficient

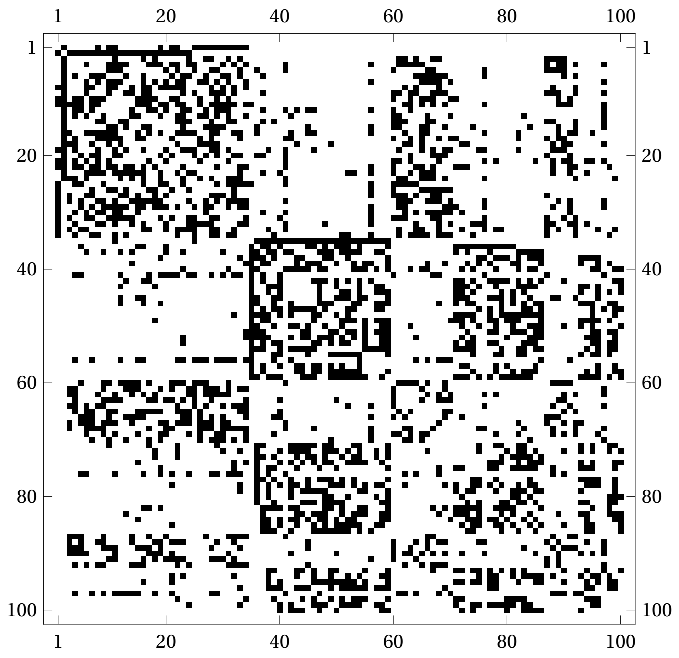
In[271]:=
```
MaxClusterNode = VertexList[graph][[Position[LocalClusteringCoefficient[graph],
Max[LocalClusteringCoefficient[graph]]][[1]]]];
HighlightGraph[graph, MaxClusterNode];
```



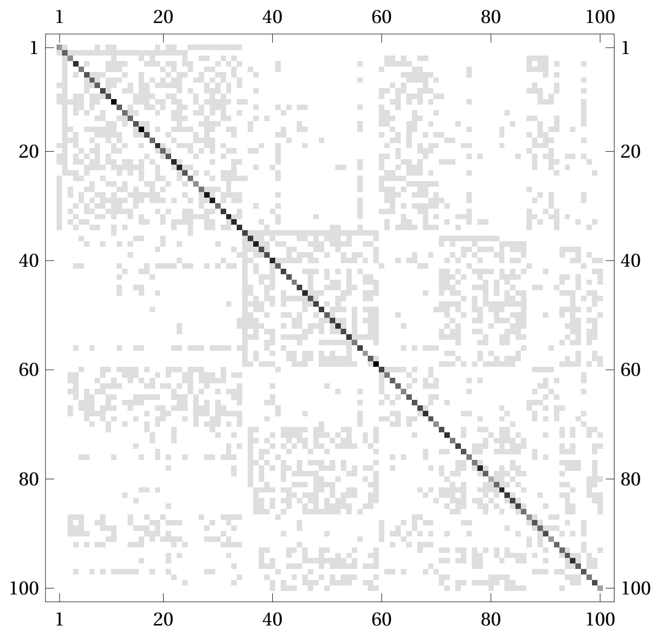
Adjacency Matrix Heatmap, Non-Normalized Laplacian (Kirchoff), Affinity Matrix

Plotting the heatmap of the adjacency matrix is a common way to visualize a network, it is especially effective when the node partitons are ordered based on node-ids. In our case (unlike the first example graph) there is no correlation between node-ids and the partitions, thus the adjacency matrix is not as informative as it was for example graph 1
There exists multiple versions of the Laplacian matrix with small modifications, the Kirchoff matrix is one of them. The affinity matrix is computed as the Jordan Decomposition of the Adjaceny matrix. There are many spectrums that are useful for graph analysis, including the spectrum of the adjacency matrix, the transition matrix and the Laplacian matrix. However, the Laplacian matrix is the most useful for reasoning about the connectivity of the graph as well as its clustering.

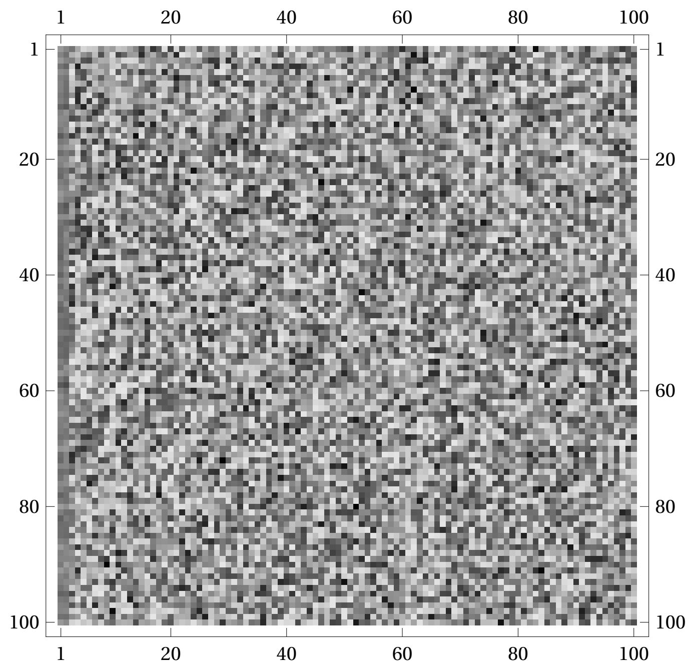
In[273]:=
```
MatrixPlot[A];
kirchoffLaplacianMatrix = KirchhoffMatrix[graph];
MatrixPlot[kirchoffLaplacianMatrix];
{affinityMatrix, s} = JordanDecomposition[N[Transpose[A]]];
MatrixPlot[affinityMatrix];
```



Adjacency Matrix plot



KirchoffLaplacian Matrix (not normalized)

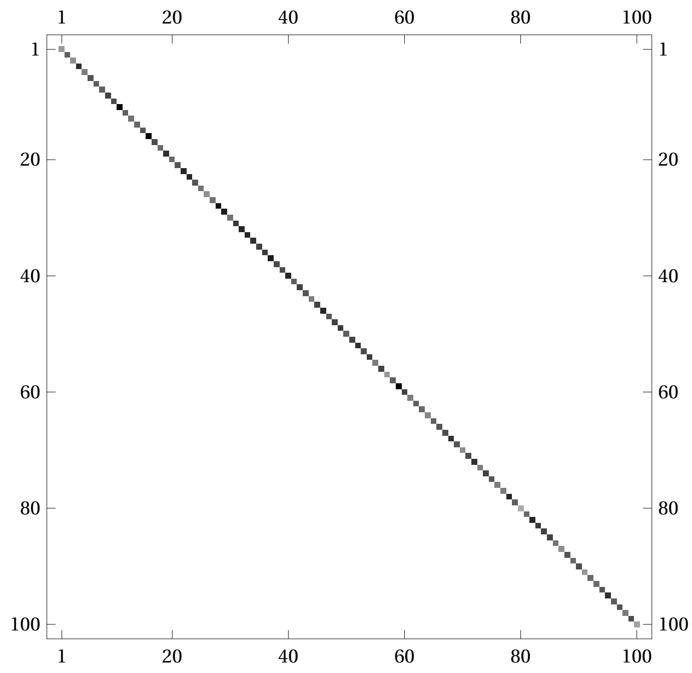

Affinity Matrix Plot

Degree Matrix, Simple Laplacian

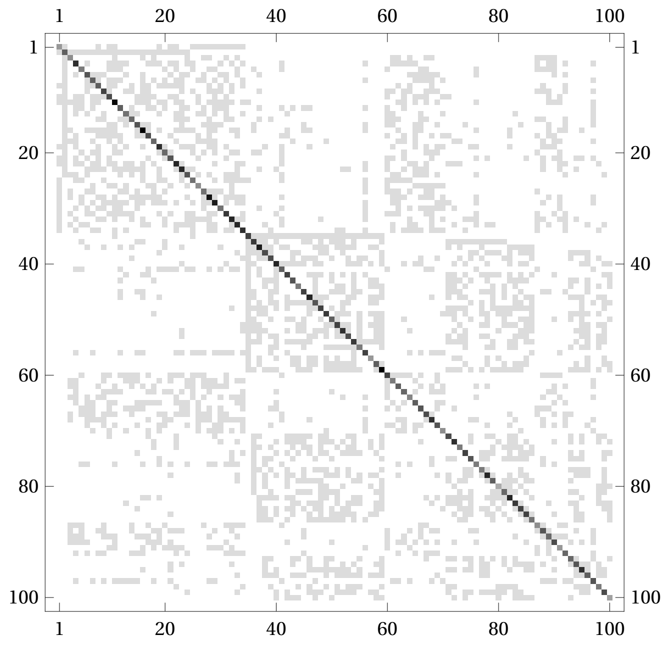
The Degree Matrix D is a diagonal matrix with the degree of each node i on the diagonal (i,i). Here I also compute the simple Laplacian matrix which is L = D - A.

In[278]:=

```
{n,n} = Dimensions[A];
DegreeMatrix = ConstantArray[0, {n,n}];
For[i = 1, i <= n, i++, DegreeMatrix[[i,i]] = Total[A[[i]]]];
MatrixPlot[DegreeMatrix];
simpleLaplacian = DegreeMatrix - A;
MatrixPlot[simpleLaplacian];
```



Degree Matrix Plot



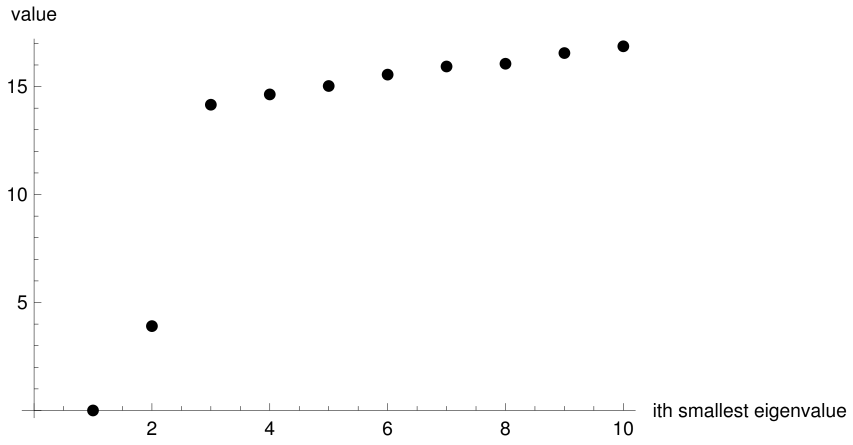
Simple Laplacian Matrix Plot

EigenGap of Simple Laplacian and Fiedler Vector

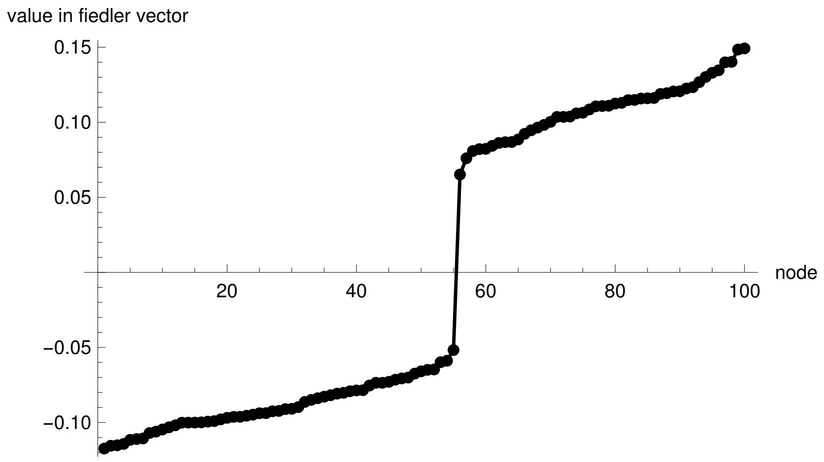
In the Laplacian we know that $\lambda_1 = 0$ with a corresponding eigenvector $v_1 = [1, \ldots, 1]$. This follows from the fact that the rows and the columns of the Laplacian sum up to 0 (each row contains number of -1 as number of neighbors plus one entry on the row which is the degree of the node). Further more we can tell by analyzing the higher-order eigenvalues how many connected components the graph has and whether it is a good expander or not. In our case $\lambda_1 = \lambda_2 = 0$ which means that the graph has 2 connected components. We also know from spectral graph theory that $\lambda_{n \leq 2}$which is in concordance with our results below. Furthermore, by looking at the eigen-gap between $\lambda_2$and $\lambda_3$we can tell whether it is close or not that there is a third disconnected component, and we can see that the third component seems to be quite well connected since the eigen-gap is quite large.

Finally, the eigenvector associated with λ_2, also know as the “Fiedler Vector”, can give a bi-partition of the graph. The Fiedler does not indicate the k-clusters but it can indicate the optimal 2-clusters (and if applied recursively it can even find k clusters, but typically to exploit higher order eigenvectors is a better approach).

```
{smallestEigenVals, smallestEigenVecs} = Eigensystem[N[simpleLaplacian], -10];
ListPlot[Reverse[Chop[smallestEigenVals]], AxesLabel→{"ith smallest eigenvalue","value"
fiedlerVector = smallestEigenVecs[[-2]];
ListLinePlot[Sort[fiedlerVector], AxesLabel→{"node","value in fiedler vector"}];
```



We can see that there is a gap between the 2th smallest eigenvalue and the 3th smallest eigenvalue of the simple Laplacian. This indicates that there are 2 clusters.

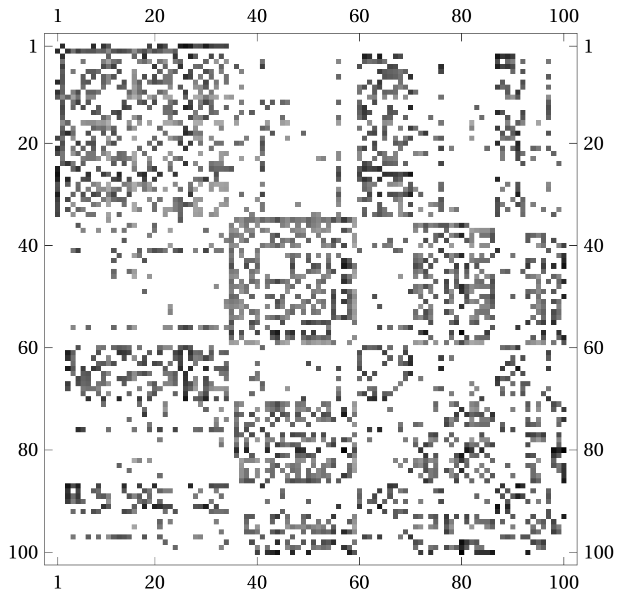


Sorted Fiedler Vector plot, gives a near optimal 2-way partitioning by assigning nodes to partitions based on their index in the Fiedler vector and the sign of the value in the vector.

Ng, Jordan, & Weiss Laplacian

As mentioned, there are many variants of the laplacian matrix used in different contexts, they all have the same basic characteristics and differ only slightly. In the code snippet below the Laplacian of the Ng, Jordan & Weiss paper is computed as L = $D^{-1/2}\,\mathrm{A}D^{-1/2}$

In[288]:=
```
k = 2;
{n,n} = Dimensions[A];
njwLaplacian = MatrixPower[DegreeMatrix,-(1/2)].(A.MatrixPower[DegreeMatrix, -(1/2)]);
MatrixPlot[njwLaplacian];
```



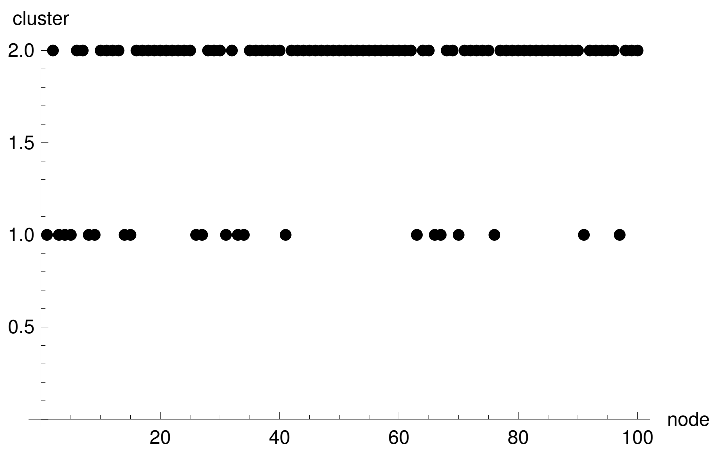
Ng, Jordan, & Weiss Laplacian

Ng, Jordan, & Weiss Spectral Clustering

Here the bulk of the algorithm in the paper is implemented.

1. Decide k, we chose k = 2 since this was obvious from the preprocessing, e.g there are 2 connected components for instance.
2. The eigendecomposition of the Laplacian is computed
3. X is formed as a matrix with columns being the k largest eigenvectors
4. Define Y as X with all columns normalized to unit length
5. Cluster Y with K-means (a row in Y is considered as a datapoint to cluster)
6. Assign the original datapoints (from the adjacency matrix) to their clusters based on what their corresponding row in Y was clustered as.

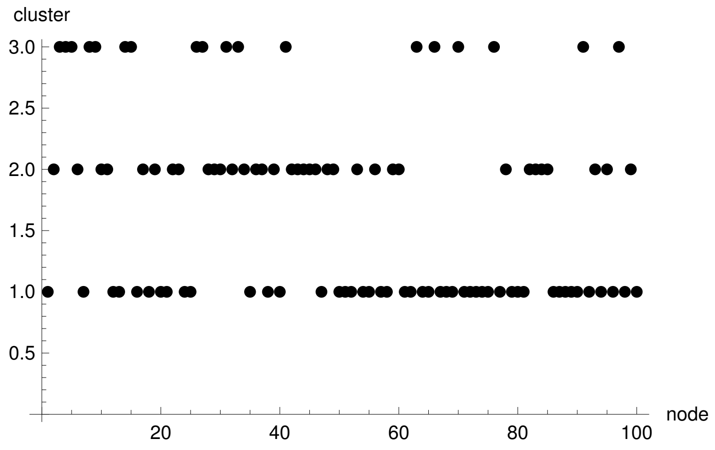
In[292]:=
```
k = 2;
{largestEigenVals, largestEigenVecs} = Eigensystem[N[njwLaplacian],4];
X = Transpose[largestEigenVecs];
MatrixPlot[X];
{rows,cols} = Dimensions[X];
Y = ConstantArray[0, {rows,cols}];
For[i = 1, i <= cols, i++, Y[[All,i]] = Normalize[X[[All, i]]]];
clusters = ClusteringComponents[Y,k,1, Method→ "KMeans"];
ListPlot[clusters, AxesLabel→{"node","cluster"}];
```



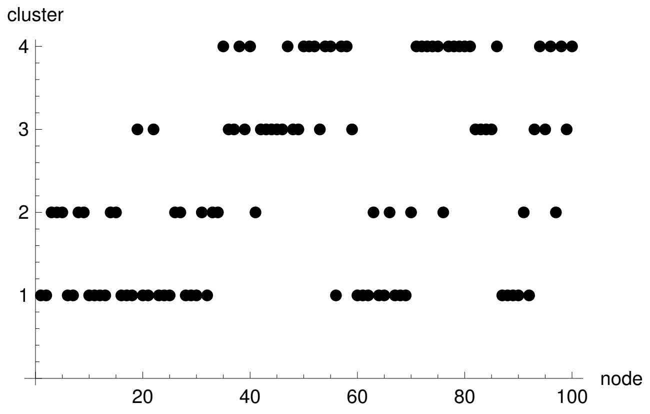
As can be seen from the plot , the 2 clusters have quite mixed node-ids. One cluster is larger than the other.

Test clustering with “wrong” k

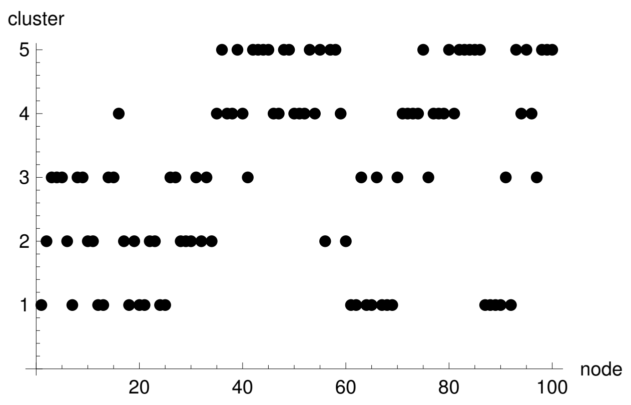
In[301]:=
```
k = 3;
clusters = ClusteringComponents[Y,k,1, Method→ "KMeans"];
ListPlot[clusters, AxesLabel→{"node","cluster"}];
k = 4;
clusters = ClusteringComponents[Y,k,1, Method→ "KMeans"];
ListPlot[clusters, AxesLabel→{"node","cluster"}];
k = 5;
clusters = ClusteringComponents[Y,k,1, Method→ "KMeans"];
ListPlot[clusters, AxesLabel→{"node","cluster"}];
k = 6;
clusters = ClusteringComponents[Y,k,1, Method→ "KMeans"];
ListPlot[clusters, AxesLabel→{"node","cluster"}];
```



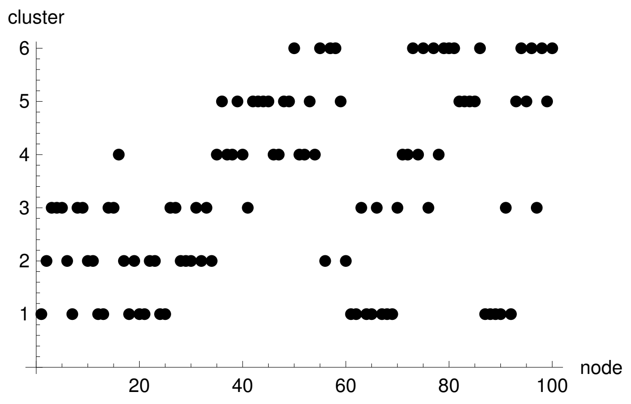
k = 3



k = 4


cluster
5
4
3
2
1
20
40
60
80
100
node

k = 5


cluster
6
5
4
3
2
1
20
40
60
80
100
node

k = 6

Homework 5: K-way Graph Partitioning Using JaBeJa

Konstantin Sozinov, sozinov@kth.se
Kim Hammar, kimham@kth.se

December 12, 2017

1 Introduction

The main algorithm can be found in the *Jabeja.java* file. We implemented the Ja-Be-Ja distributed graph partitoning algorithm in Java, following the given template code as well as the pseudo-code in the paper. This report includes evalautions of the algorithm on three graphs: add20, elt3 and twitter. The evaluations indicate that different parameters of the algorithm are suitable for each of the graphs in a number of metrics: convergence time, edge-cut, swaps, and node-migrations. Finally, we implemented our own extension to the simulated annealing of Ja-Be-Ja and demonstrate some interesting results.

The algorithm is vertex-parallel. Each vertex performs local search to improve the partitioning by minimizing the edge cuts locally. Formally, the local search of node p tries to minimize $arg\min_c \sum_{v \in N_p} d_p - d_p(c)$, where N_p is the neighborhood of p, $d_p = |N_p|$, and $d_p(c)$ is the number of neighbors with color c. Each vertex only has access to its local view of nodes and edges in the graph, and attempts to swap colors with its neighbors to improve its local situation, and (hopefully) also improve the global partitioning. The global edge-cut size is also denoted as the energy of the system. Ja-Be-Ja is an heuristic algorithm based on a portion of randomness when using simulated annealing. Ja-Be-Ja does not provide any upper or lower bound guarantees on the resulting partitions. Nodes only swap colors if it reduces their local energy. There are three policies to select nodes for swapping: random, local, and uniform. A node will go through all its selected nodes and see which one is best to swap with for each iteration.

To escape local minimas, Ja-Be-Ja utilizes simulated annealing. The basic simulated annealing outlined in the paper uses a temperature factor T, when $T > 1$ swap-decisions are biased towards swapping rather than not-swapping, even if it could increase the energy of the system. T is reduced over time until it reaches 1. The second version of simulated annealing uses the technique outlined in a blog post [1]. To summarize, this approach to simulated annealing goes over all neighboring solutions and selects the best solution using the following formula for acceptance probability, $ap = e^{\frac{c_{old} - c_{new}}{T}}$. Furthermore, we also implemented this technique using restarts, meaning that when $T = 0$ it is reset back to 1.

Finally, our own version of simulated annealing got inspiration from the Momentum technique, known to improve Gradient Descent convergence time. Formally we use momentum as follows:

$$momentum = max(0, \mu \cdot (c_{new}^{(t)} - c_{new}^{(t-1)}))$$

$$ap = e^{\frac{c_{old} - (c_{new} - momentum)}{T}}$$

Where μ is the momentum coefficient. When using momentum we did not use any restarts of T.

2 Evaluation and results

For all tasks we used the hybrid selection policy as that was presented as the best policy in the paper. Additionally we tried to stick to the values of α and δ that performed best in the paper.

[1] http://katrinaeg.com/simulated-annealing.html

2.1 Task 1 - Linear Simulated Annealing, no restarts, no randomness

graph	delta	T	edge-cut	rounds	swaps	migrations	partitions	converge	alpha	policy
add20	0.003	2	2095	1000	1090263	1751	4	yes	2	hybrid
3elt	0.003	2	2604	1000	1580209	3328	4	yes	2	hybrid
twitter	0.003	2	41156	1000	899515	2049	4	yes	2	hybrid

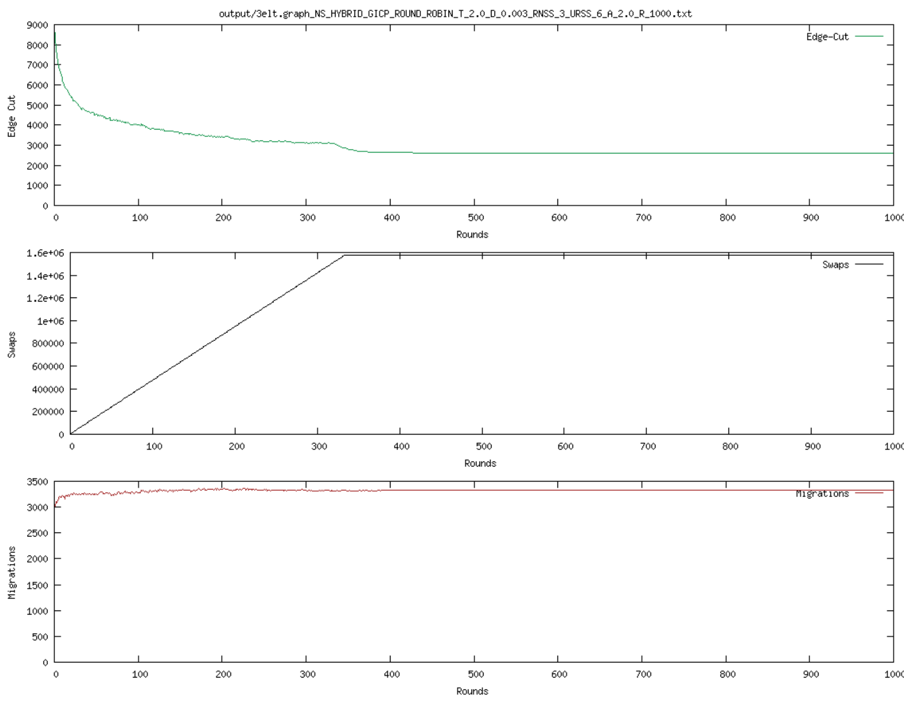


Figure 1: 3elt

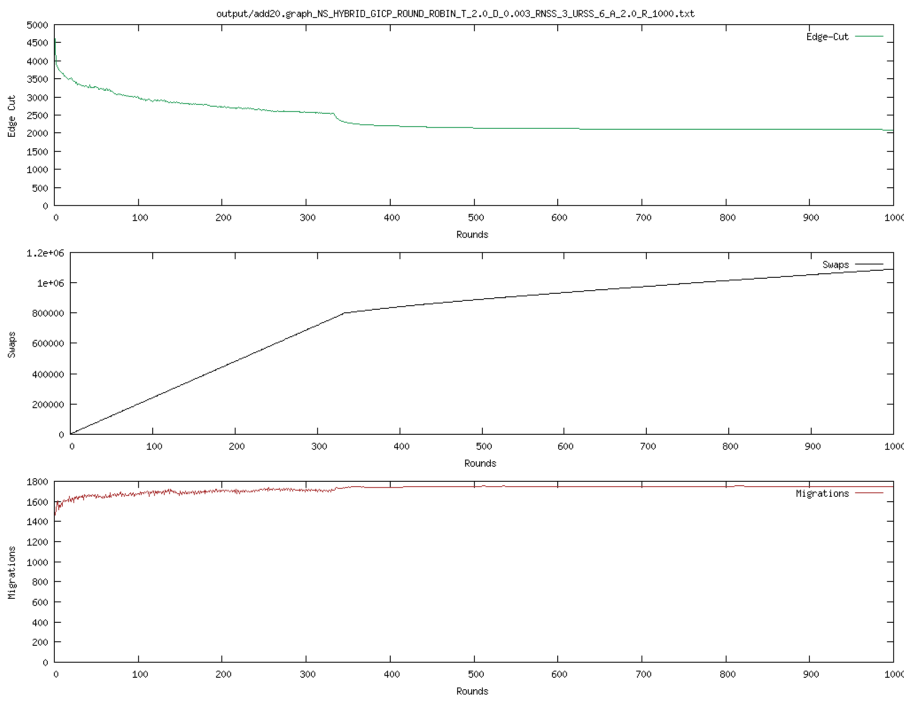


Figure 2: add20

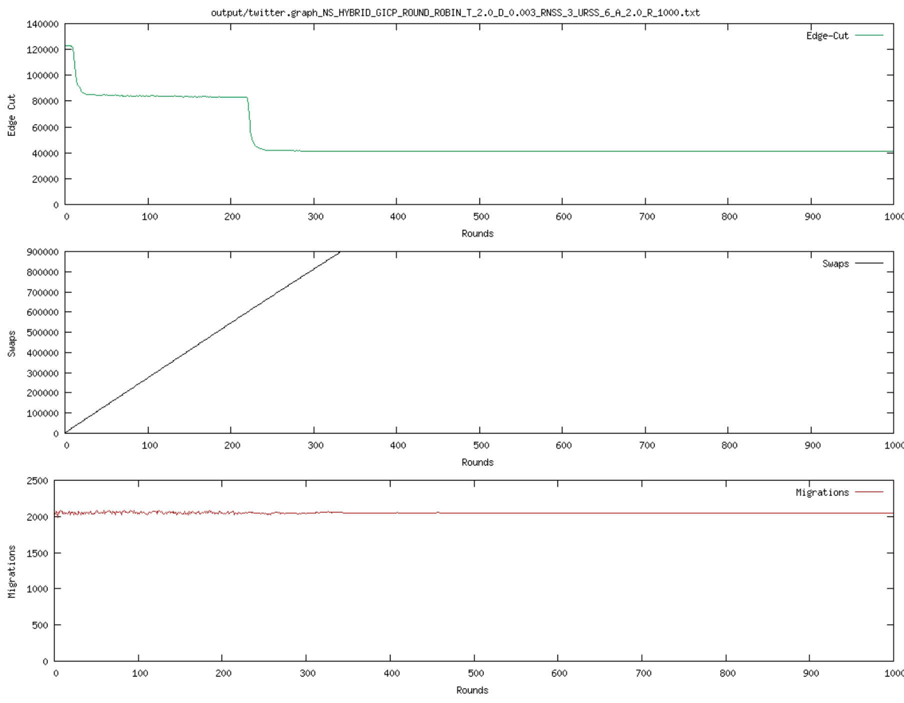


Figure 3: twitter

What can be noted about these results are that the algorithm converged early and to a bad solution (local optima) in all three cases.

2.2 Task 2.1 - Linear Simulated annealing, no restarts, randomness and acceptance probability

graph	delta	T	edge-cut	rounds	swaps	migrations	partitions	converge	alpha	policy
3elt	0.003	1	2190	1000	103586	3274	4	yes	2	hybrid
add20	0.003	1	2060	1000	373826	1745	4	yes	2	hybrid
twitter	0.003	1	41115	1000	48804	2046	4	yes	2	hybrid

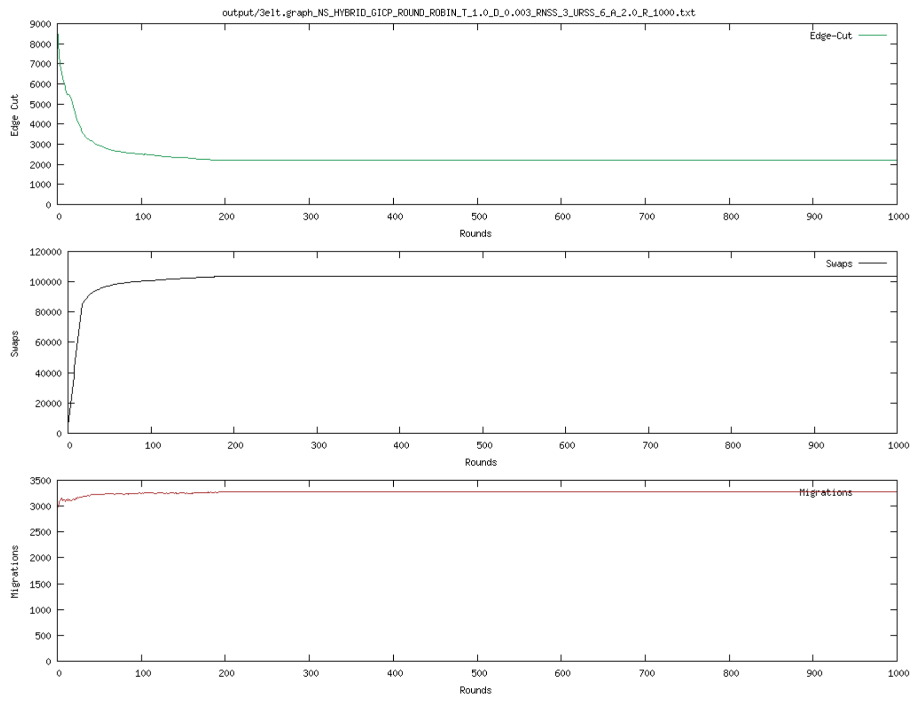


Figure 4: 3elt

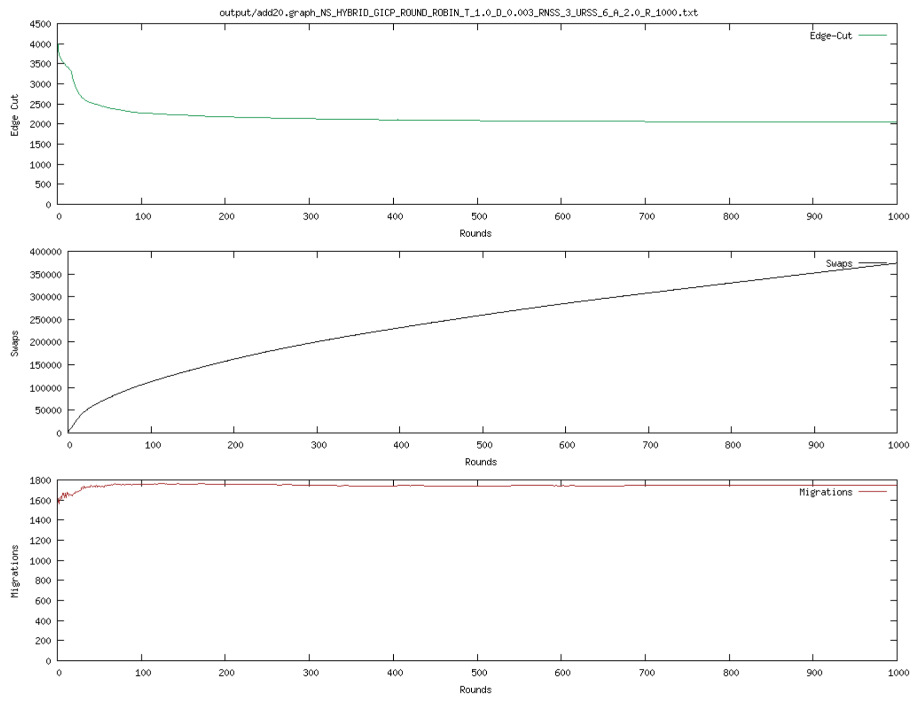


Figure 5: add20

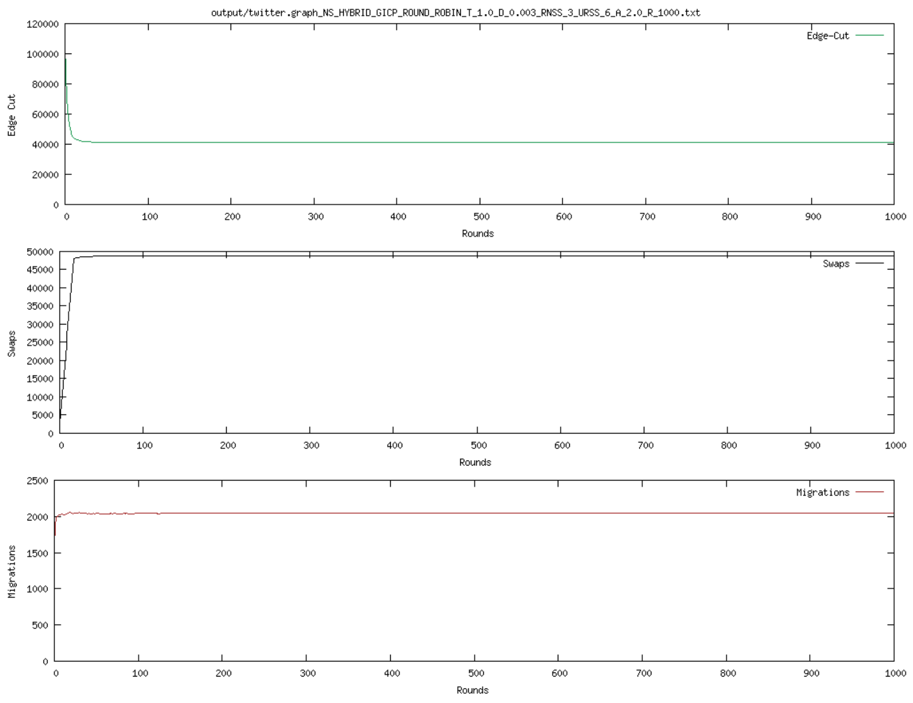


Figure 6: twitter

What can be noted about these results are that the new simulated annealing technique gave a lot better results on 3elt, but almost the same results on add20 and twitter graphs. Furthermore the algorithm converged in all cases (local optima problem again).

2.3 Task 2.2 - Linear Simulated annealing with restarts, randomness and acceptance probability

graph	delta	T	edge-cut	rounds	swaps	migrations	partitions	converge	alpha	policy	restart
3elt	0.003	1	2037	1000	4463446	3296	4	no	2	hybrid	1
add20	0.003	1	2348	1000	2303961	1746	4	no	2	hybrid	1
twitter	0.003	1	41147	1000	2494681	2049	4	yes	2	hybrid	1

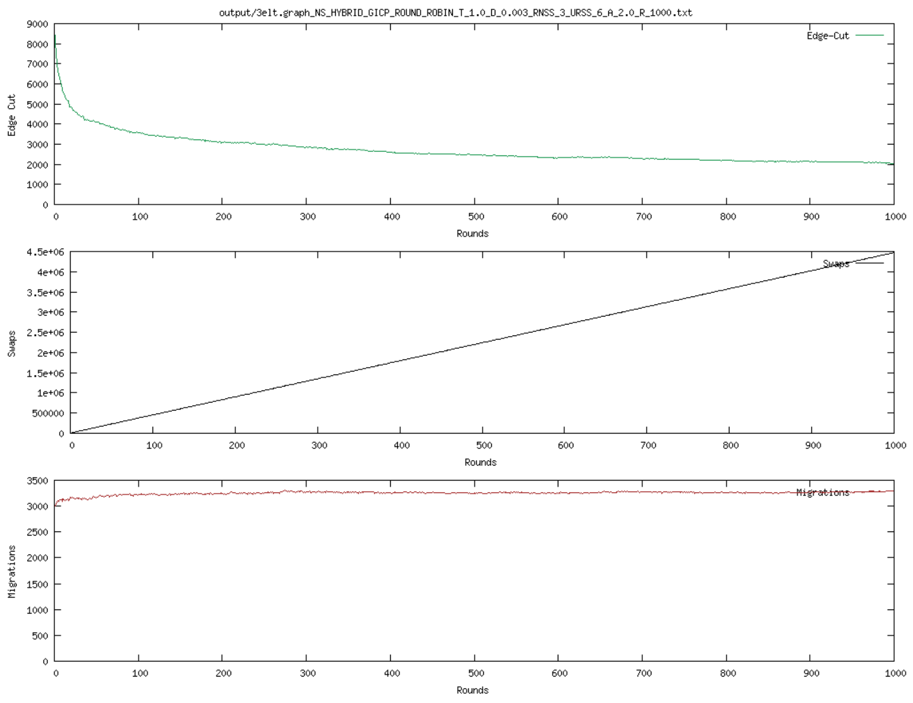


Figure 7: 3elt

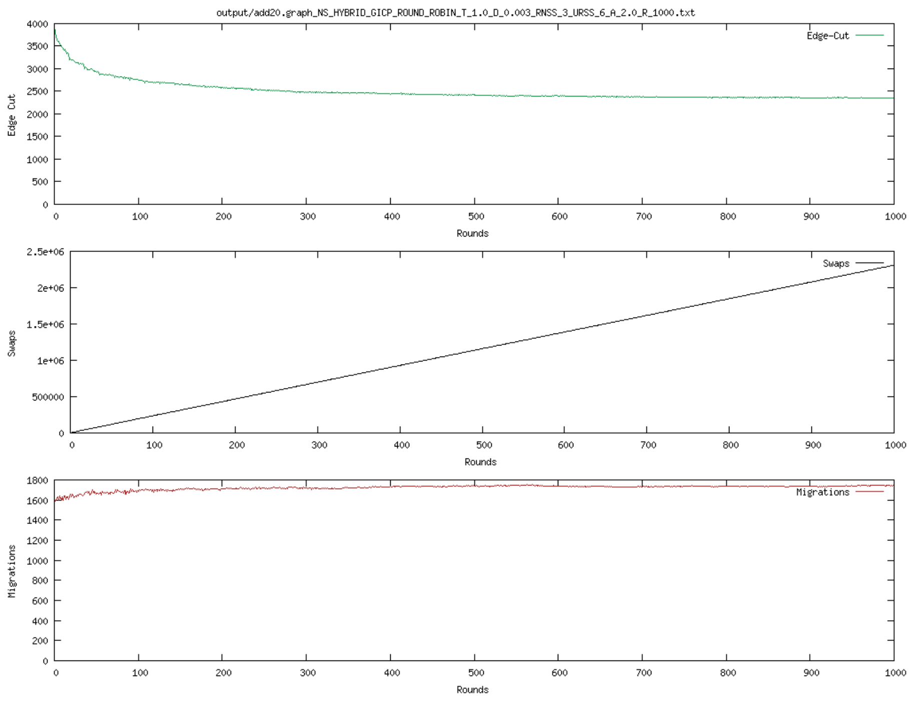


Figure 8: add20

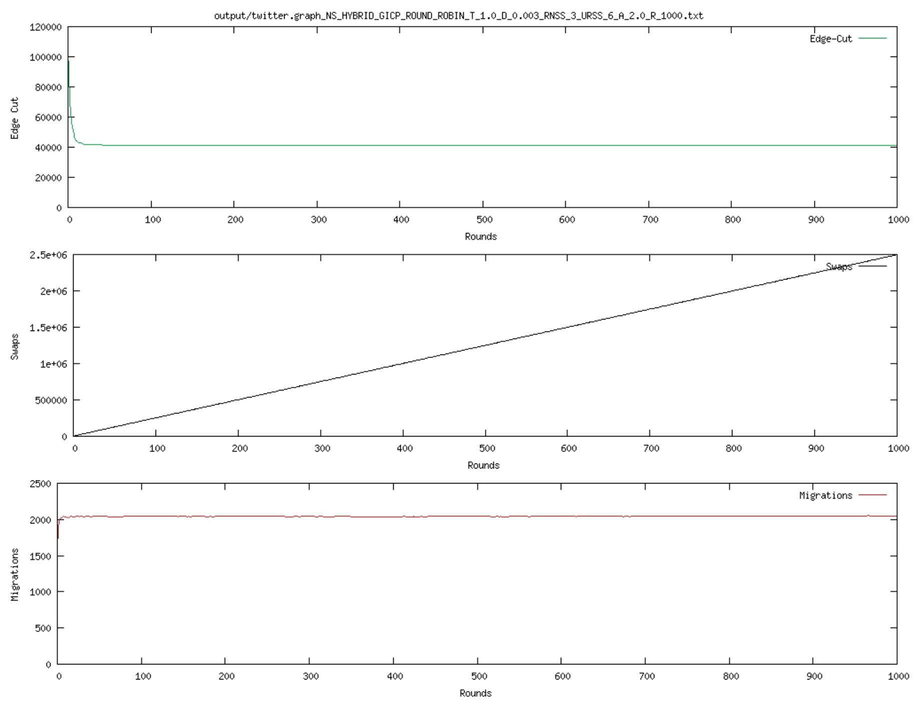


Figure 9: twitter

The results demonstrate that adding restarts did not help much without tuning the rest of the parameters.

2.4 Task 2.3 Exponential simulated annealing with restarts, randomness and acceptance probability

graph	delta	T	edge-cut	rounds	swaps	migrations	partitions	converge	alpha	policy	restart
3elt	0.003	1	2504	1000	4713193	3328	4	no	2	hybrid	1
add20	0.003	1	2471	1000	2392641	1679	4	no	2	hybrid	1
twitter	0.003	1	41327	1000	2636468	2045	4	yes	2	hybrid	1

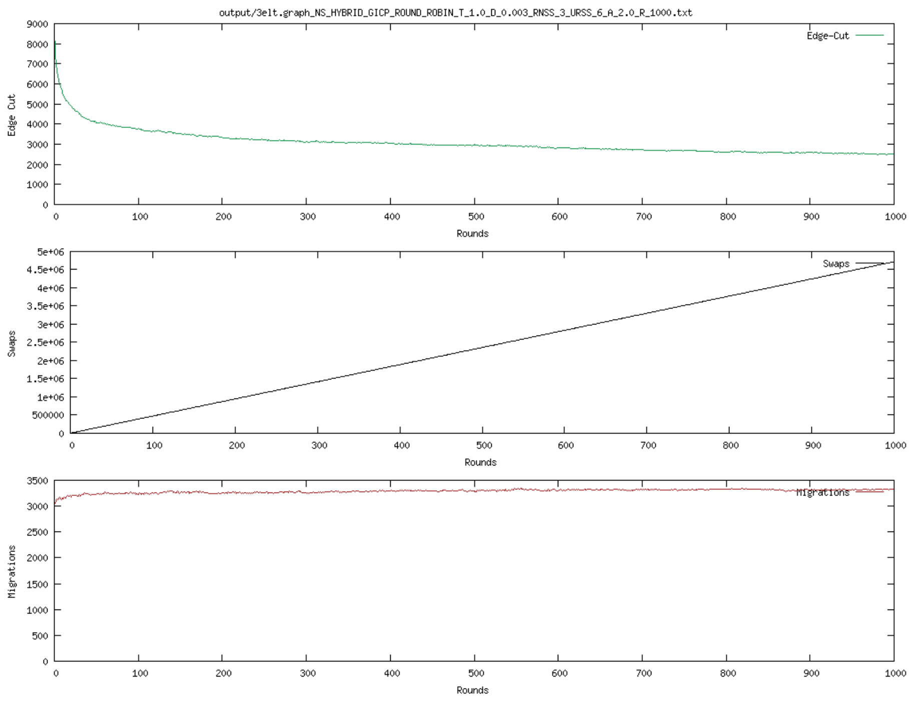


Figure 10: 3elt

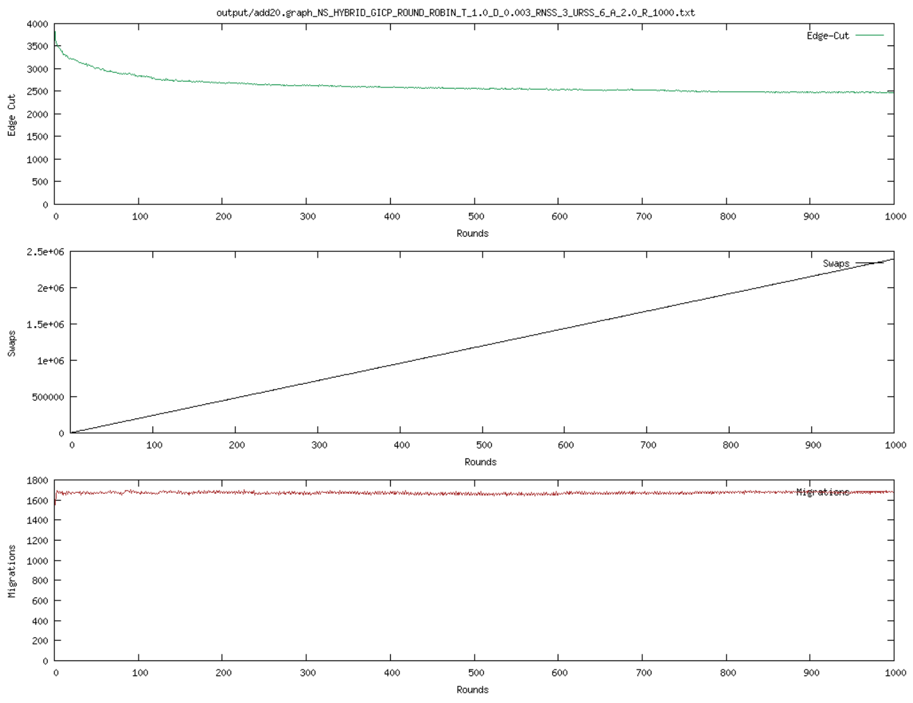


Figure 11: add20

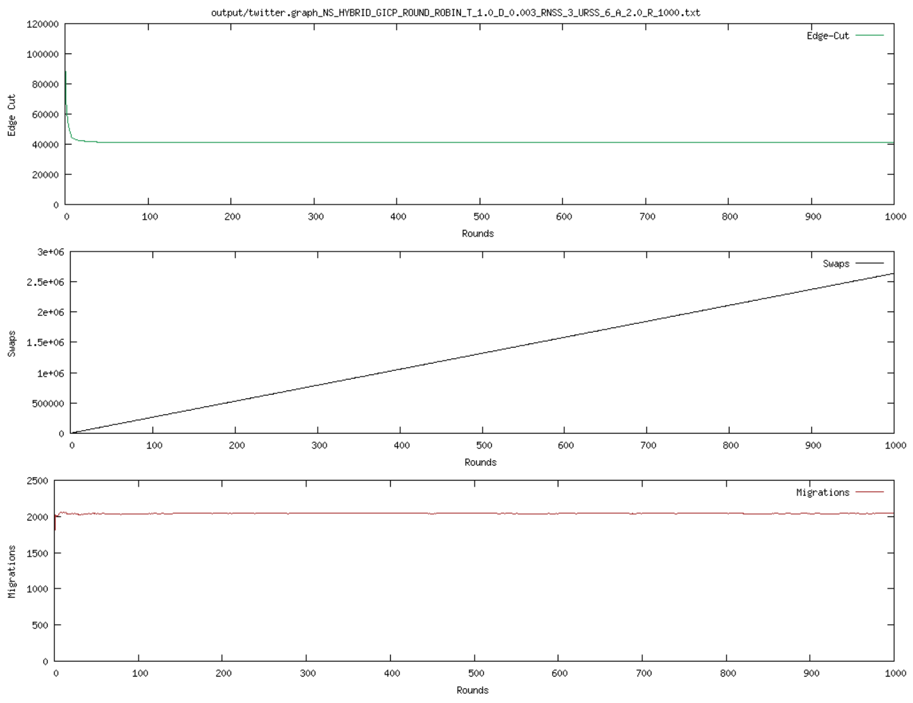


Figure 12: twitter

The results with exponential simulated annealing without parameter tuning gave consistently worse results.

2.5 Task 2.4 Linea simulated annealing with restarts, randomness and acceptance probability - Parameter tuning

graph	delta	T	edge-cut	rounds	swaps	migrations	partitions	converge	alpha	policy	restart
3elt	0.00001	1	1021	10000	42010302	3441	4	no	2	hybrid	1
3elt	0.00001	1	1208	10000	42541398	3425	4	no	1	hybrid	1
3elt	0.003	1	1011	10000	44596460	3422	4	no	2	hybrid	1
3elt	0.003	1	**731**	50000	222928227	3435	4	yes	2	hybrid	1
add20	0.00001	1	2196	10000	22126510	1753	4	no	2	hybrid	1
add20	0.00001	1	1792	10000	21773341	1757	4	yes	1	hybrid	1
add20	0.003	1	**1780**	10000	22704110	1734	4	yes	1	hybrid	1
twitter	0.00001	1	41258	2000	4739316	2046	4	yes	2	hybrid	1
twitter	0.003	1	**40841**	2000	5000891	2043	4	yes	1	hybrid	1

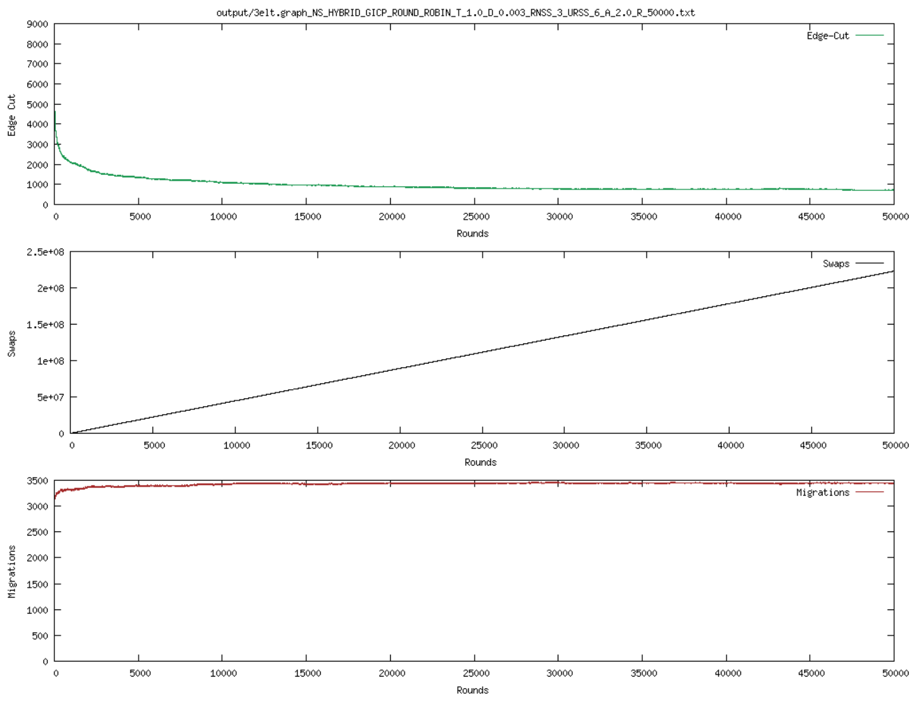


Figure 13: 3elt

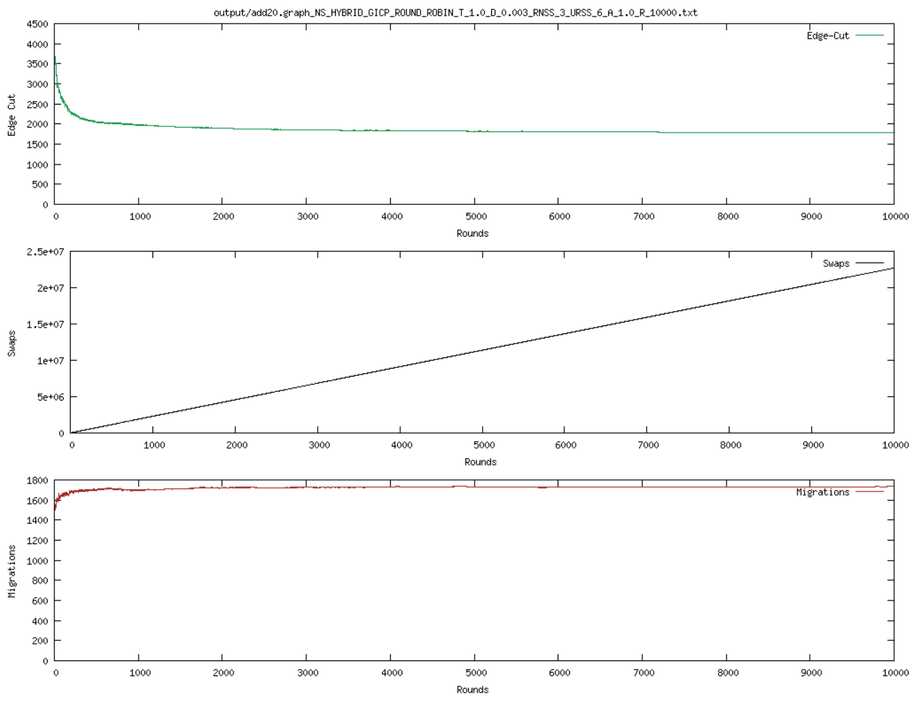


Figure 14: add20

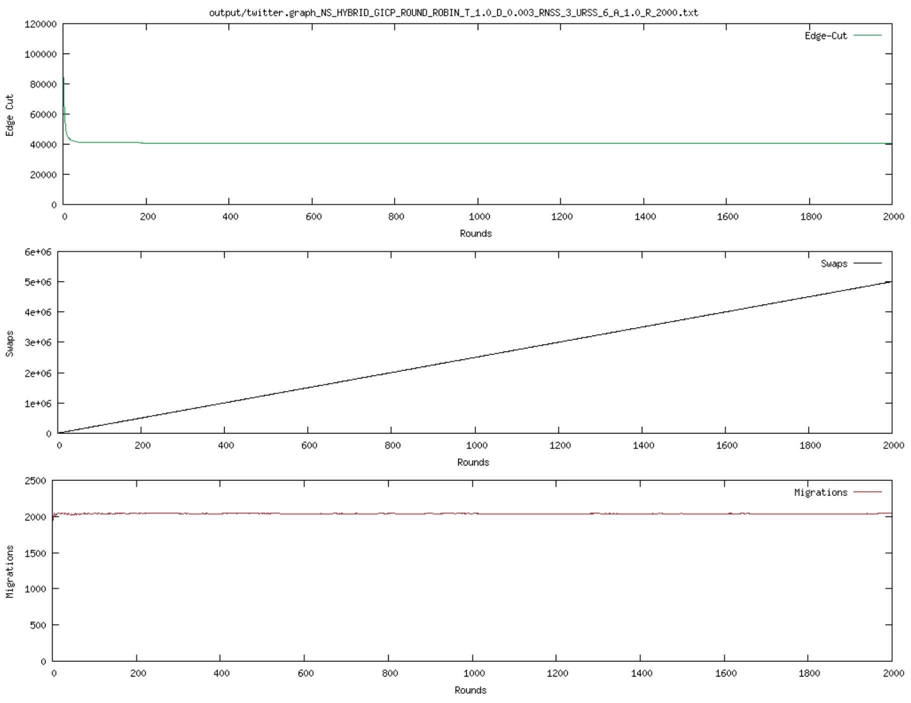


Figure 15: twitter

From these results we can see that by tuning some parameters, primarily alpha, and the number of rounds, we can greatly improve the partitions on all graphs. Both twitter and add20 gave better results with $\alpha = 1$.

2.6 Bonus Task: Momentum + Simulated Annealing

graph	delta	T	edge-cut	rounds	swaps	migrations	partitions	converge	alpha	policy	momentum
3elt	0.003	1	1256	1000	4280889	3420	4	no	2	hybrid	0.001
3elt	0.003	1	5139	1000	4685823	3535	4	no	2	hybrid	10
3elt	0.003	1	1344	1000	4281498	3397	4	no	2	hybrid	0.0001
3elt	0.003	1	697	10000	42315849	3457	4	no	2	hybrid	0.001
3elt	0.003	1	**518**	50000	210569840	3463	4	yes	2	hybrid	0.001
add20	0.003	1	2095	1000	2294945	1815	4	no	2	hybrid	0.001
add20	0.003	1	**1997**	10000	22776283	1785	4	yes	1	hybrid	0.00001
twitter	0.003	1	41137	1000	2485027	2034	4	yes	2	hybrid	0.001
twitter	0.003	1	40878	1000	2498748	2035	4	yes	1	hybrid	0.001
twitter	0.003	1	**40833**	1000	2488748	2041	4	yes	1	hybrid	0.0001
twitter	0.003	1	41436	1000	2490911	2068	4	yes	1	hybrid	0.00001

With the momentum technique we achieved the best results on 3elt! And about the same results on add20 and twitter. Primarily the momentum improved the convergence time as we can see that only after 1000 iterations we got pretty good results on 3elt.

3 Conclusion

We got pretty close to the results presented in the paper but still a bit off, this is likely because we did not tune all parameters, for example we did very little tuning of $\delta, T, \alpha, restart, momentum$. For proper evaluation we could have applied grid search or random search to find the optimal parameters.

Finally, momentum looks like an promising techniques to improve convergence rate on some graphs.

4 How to run

Clone this repository and navigate to *jabeja* project. Then use:

```
/run.sh -graph ./graphs/3elt.graph -rounds 5000 -numPartitions 4 -temp 1
-delta 0.00001 -restart 0.000001 -alpha 1 -nodeSelectionPolicy HYBRID -momentum 0.001
```