Introduction to Big Data

Project: Grid Graph

Prepared By:
Hasan Iqbal

Presented to:
Professor Wenguang Chen
TA Wei Qin

Date: 23/6/2017

Table of Contents

1. Grid Graph Introduction:

GridGraph is a system for processing large-scale graphs on a single machine. GridGraph breaks graphs into 1D-partitioned vertex chunks and 2D-partitioned edge blocks using a first fine-grained level partitioning in preprocessing. A second coarsegrained level partitioning is applied in runtime. Through a novel dual sliding windows method, GridGraph can stream the edges and apply on-the-fly vertex updates, thus reduce the I/O amount required for computation. The partitioning of edges also enable selective scheduling so that some of the blocks can be skipped to reduce unnecessary I/O. This is very effective when the active vertex set shrinks with convergence.

2. Label Propagation Implementation:

Within complex networks, real networks tend to have community structure. Label propagation is an algorithm for finding communities. In comparison with other algorithms, label propagation has advantages in its running time and amount of a priori information needed about the network structure (no parameter is required to be known beforehand). The disadvantage is that it produces no unique solution, but an aggregate of many solutions.

I will explain step by step about the Label Propagation algorithm and how I implemented it.

2.1. Graph Input (Data input):

Let's suppose that we have following directed graph with 4 vertices

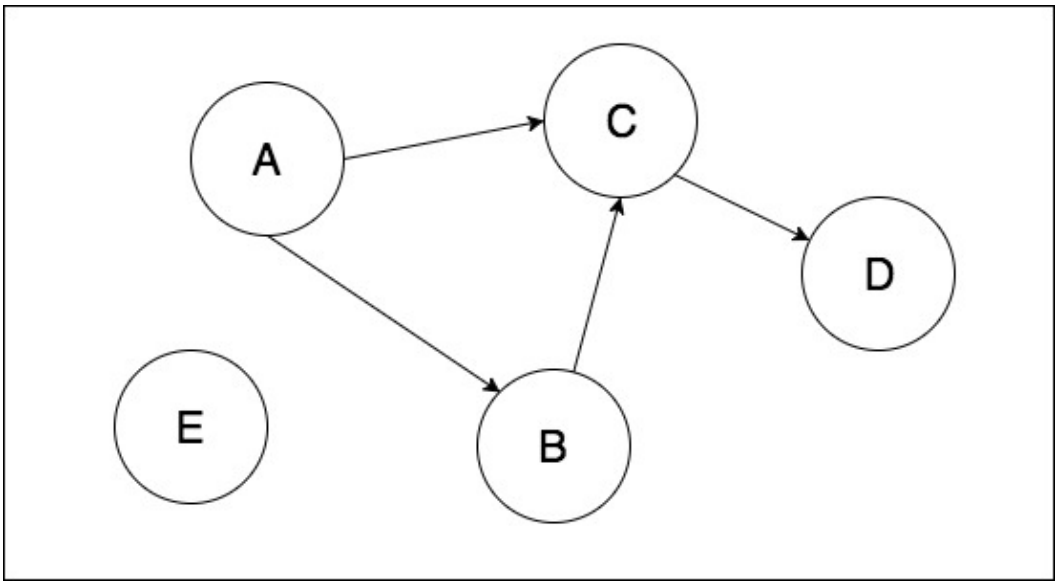

Figure 1: Directed Graph

In above figure, we have a directed graph with 4 nodes A, B, C, D and E. They all are connected except E. The have directed edges. Now this is our sample dataset. User gives the input of number of iterations, number of labels and memory budget in form of parameters while running the program. In code, this dataset is initialized by setting the path of this graph like:

```
Graph graph(path);
graph.set_memory_bytes(memory_bytes);
BigVector<double> ProbabilityOfVertices(graph.path+"/ProbabilityOfVertices", graph.vertices);
BigVector<double [20]> TempLabelGrid(graph.path+"/TempLabelGrid", graph.vertices);
BigVector<VertexId> LabelsOfVertices(graph.path+"/LabelsOfVertices", graph.vertices);
BigVector<VertexId> OutDegreesOfVertices(graph.path+"/OutDegreesOfVerticess", graph.vertices);
long vertex_data_bytes = (long) graph.vertices * (sizeof(double) + sizeof(double) + sizeof(VertexId));
graph.set_vertex_data_bytes(vertex_data_bytes);
```

2.2. Out-Degrees Calculation:

Then Next step is initialization of out-degrees of each vertex with 0 and then the calculation of out degrees of each vertex as shown below:

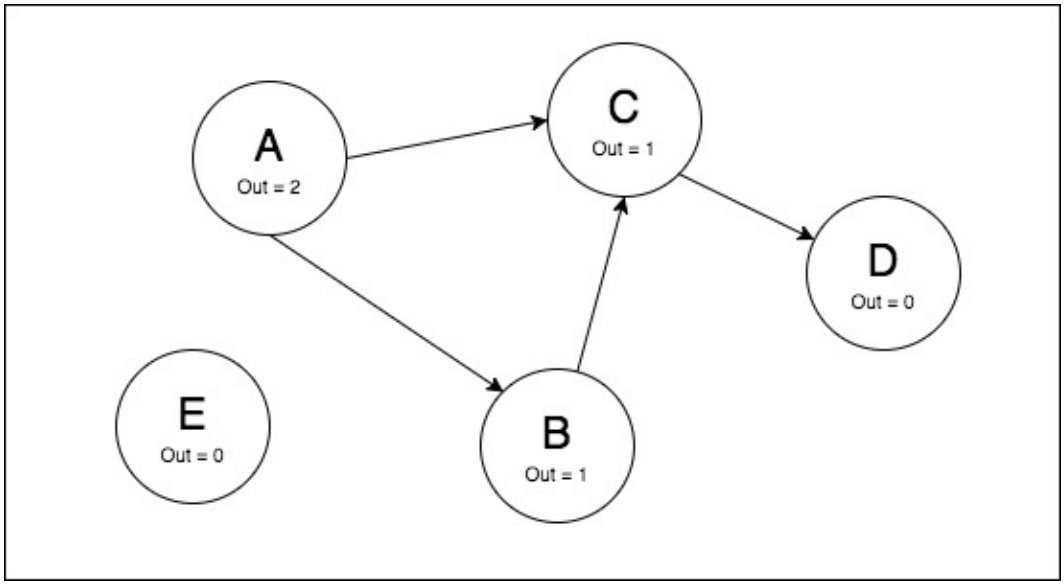

Figure 2: Calculation of Out-Degrees

Following code is used for for calculation of out degrees:

```
OutDegreesOfVertices.fill(0);
graph.stream_edges<VertexId>(
        [&](Edge & e){
                write_add(&OutDegreesOfVertices[e.source], 1);
                return 0;
        }, nullptr, 0, 0
);
```

2.3. Probability Calculation:

Then Next step is the calculation of probability of each vertex based on its out-degree. The basic concept here is that the probability that any given label of a vertex can be transferred to its connected vertices. Let's say there is a node which has out-degree of 3, then it will have 0.333 probability. Following the formula used for calculating the probability:

If out-degree of node is not 0 then
    Probability = 1/out-degree of node
Else
    Probability = 0

After calculation of probabilities of each vertex, we will get graph as shown below:

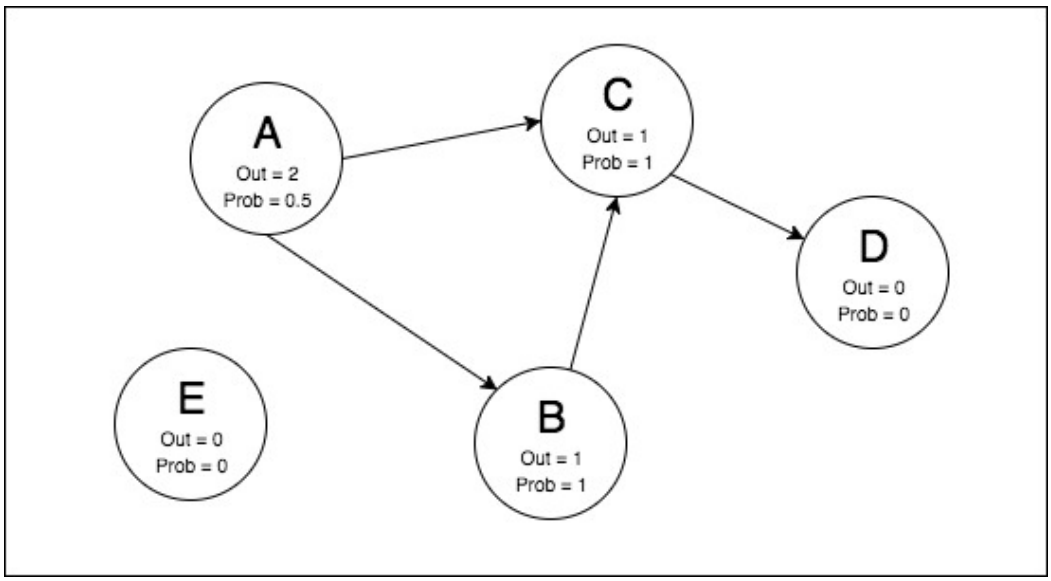

Figure 3: Graph after probabilities calculation

If we analyze this figure, then we can see that A has probability of 0.5 because it has chance 50-50% chance of transferring its label to B and C. Whereas C and B has Probability of 1 as they can can just transfer their label to one node so the chance of its transferring the label is 100%. Whereas D and E has 0 probability as D and E have no out going edge. So which ever label they get, they cannot transfer it to other node. Following code is used for calculating the probabilities:

```
graph.hint(ProbabilityOfVertices);
graph.stream_vertices<VertexId>(
    [&](VertexId i){
        if(OutDegreesOfVertices[i] != 0)
            ProbabilityOfVertices[i] = 1.f / (OutDegreesOfVertices[i]);
        else
            ProbabilityOfVertices[i] = 0.f;
        return 0;
    }, nullptr, 0
);
```

2.4. Label Assignment:

Then Next step is the assignment of the labels to all the vertices first. For this purpose, initially all the vertices are given -999 value. Which means that this vertex does not have any label yet.

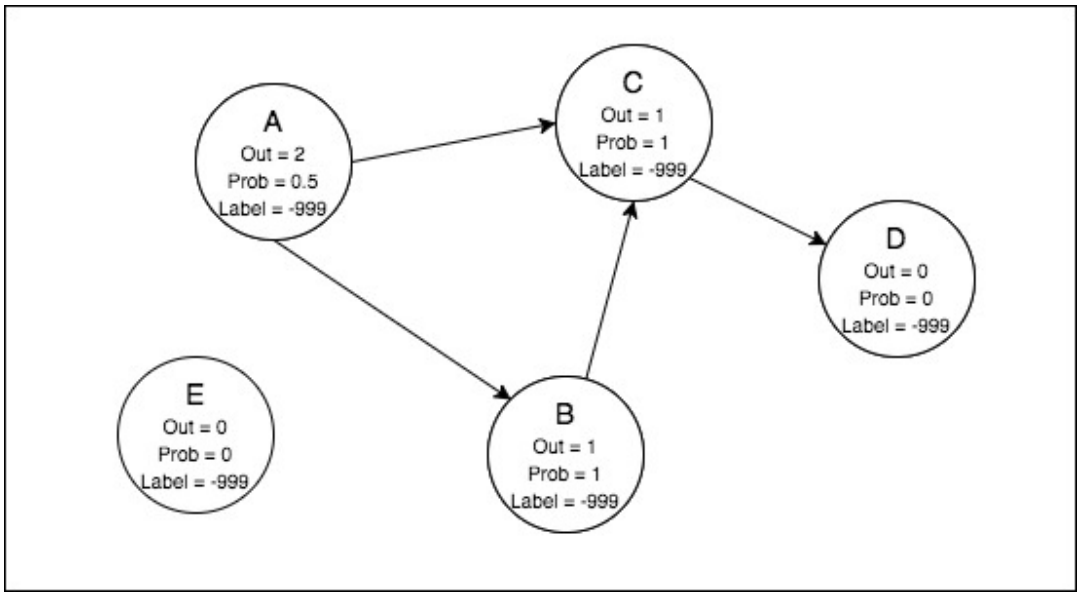

Figure 4: Initializing all vertices with -999

After initializing every label with -999, any random number of vertices will be assigned label according to the user input. For example, if user inputs 2 number of labels, then it means that labels [0 and 1] will be assigned to any random vertices. If use inputs 3, then labels [0,1 and 2] will be assigned to any random vertices. Let's suppose in our scenario, user Inputs 2 number of labels and by random assignment, label = 0 is assigned to B whereas label =1 is assigned to A as shown below:

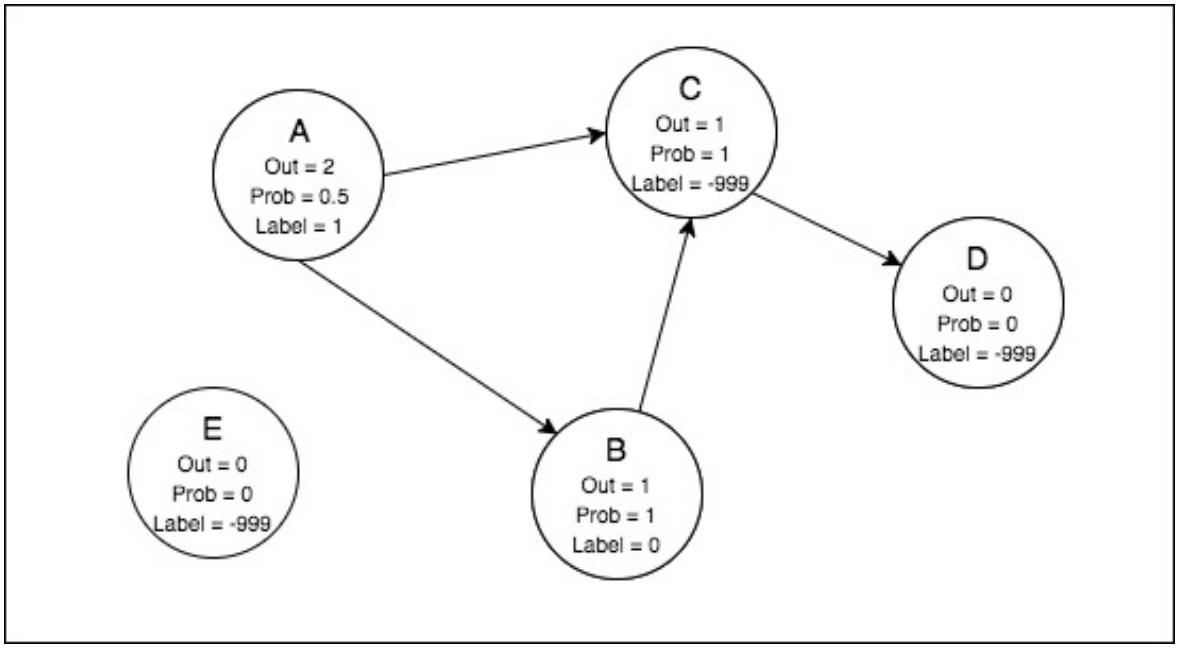

Figure 5: Labels after random assignment

Following code ensures this whole initial label assignment of -999 and then random assignment of labels according to the user specification:

```
LabelsOfVertices.fill(-999);
srand((unsigned)time(0));
for (int i=0;i<LabelsNum;i++){
        int RandomIndex = rand();
        RandomIndex = RandomIndex % graph.vertices;
        LabelsOfVertices[RandomIndex] = i;
}
```

2.5. Loop Iterations:

After the random assignment of labels, the main loop starts running according to the user specified number of iterations. If the user inputs 5, then the algorithm will run for 5 times. Let's suppose user inputs 3 for number of iterations in our scenario. Then the algorithm will run for three times in a loop. Firstly it will stream all the edges. Let's say it starts with A, it will see that it has label and its targets do not have labels, then it will start processing every edge of it. Let's suppose it stream A->B edge first. It will check that B already has a label, then it will do nothing. Then it stream A->C edge, it will see that C do not have label i.e. it has label -999. Then it will give Temporary Grid of C its probability value + the value of temporary grid, which is initialized with 0 in start, and the value which it gives is 0+0.5 = 0.5. Then let's say the algorithm checks B->C edge and it does the same thing again, probability + value of temporary grid = 0 + 1 = 1. Then C->D edge is streamed but it does not do anything as both C and D do not have any label. Following figure shows the state of of graph after completion of this step:

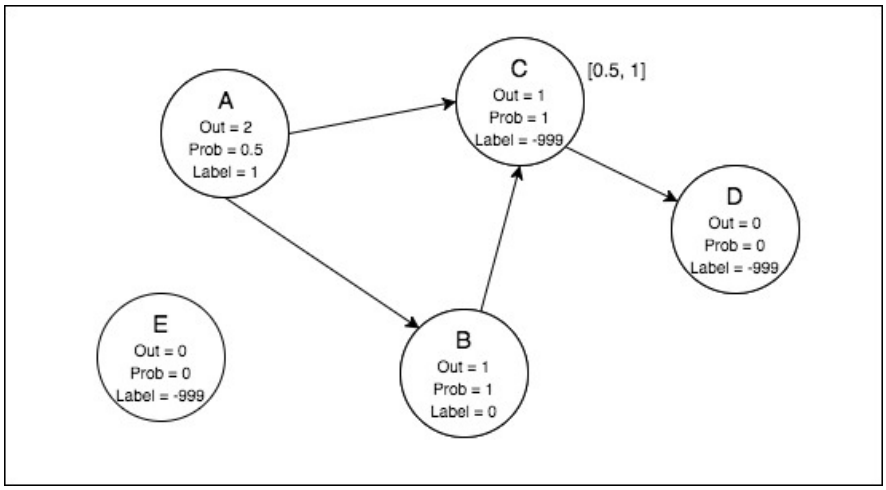

Figure 6: Graph after streaming of edges in 1st iteration

Following code implements all the things discussed above:

```
graph.stream_edges<VertexId>(
        [&](Edge & e){
                if ((LabelsOfVertices[e.source]>=0))
                        if (LabelsOfVertices[e.target]<0)
                        write_add(&TempLabelGrid[e.target][LabelsOfVertices[e.source]],
TempLabelGrid[e.target][LabelsOfVertices[e.source]]+ProbabilityOfVertices[e.source]);
                return 0;
        }, nullptr, 0, 1
);
```

Then the program will stream all the vertices. It will go to all the vertices. It will go to E and D and will find [0,0] inside temporary grid of each so it will do nothing. When it will go A and B, it will still do nothing as they already have labels. When it will go to C, it will see that it has [0.5,1] in temporary grid so it will see who gave it 1, it was B so it will retain the label of B which is 0. So label of C will be 0 too as shown below:

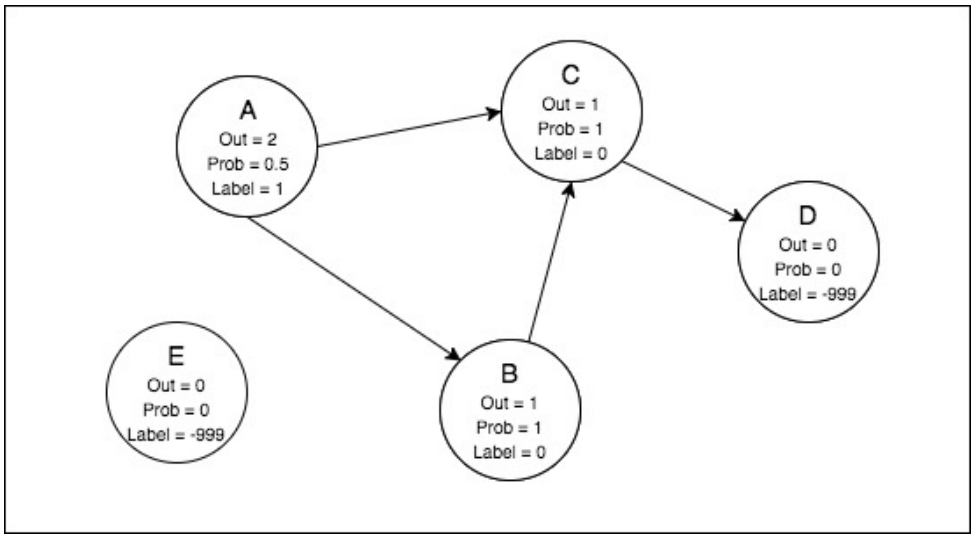

Figure 7: After streaming of all the vertices

Following code ensures streaming of all the vertices:

```
graph.stream_vertices<VertexId>(
        [&](VertexId i){
                if (LabelsOfVertices[i]<0){
                        float maxCount = 0;
                        int label = 0;
                        for (int j=0;j<LabelsNum;j++){
                                if (TempLabelGrid[i][j]>maxCount){
                                        maxCount = TempLabelGrid[i][j];
                                        label = j;
                                }
                        }
                                if (maxCount>0){
                                write_add(&LabelsOfVertices[i], label+999);
                                }
                }
                return 0;
        }
);
```

In the same manner, the program will go on and will stop eventually when all the vertices have some labels and are not changing anymore that is they are converged. Finally we will get following figure at the end of all the iterations or when the graph is converged.

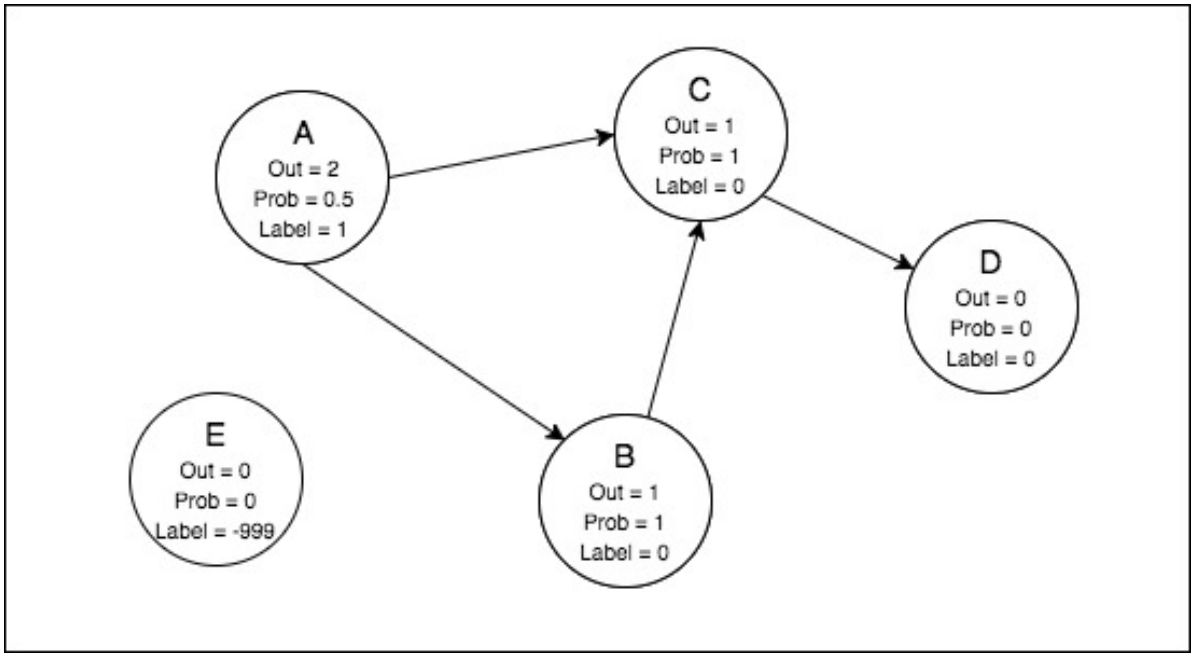

Figure 8: After all the iterations

2.6. Calculation of total Labels:

Final step in this implementation is to find the total number of vertices each label has. So in our case, label 0 has vertices B, C and D. Whereas Label 1 has vertex A. So Label 0 have 3 number of vertices whereas Label 1 have 1 number of vertex.

Following code ensure this:

```
int FrequencyofLabels [LabelsNum];
for (int j=0;j<LabelsNum;j++){
        FrequencyofLabels[j] = 0;
}
for (int j=0;j<graph.vertices;j++){
        if (LabelsOfVertices[j]>=0){
                FrequencyofLabels[LabelsOfVertices[j]]+=1;
        }
}
graph.stream_vertices<VertexId>(
        [&](VertexId i){
                return 0;
        }, nullptr, 0,
        [&](std::pair<VertexId,VertexId> vid_range){
                LabelsOfVertices.load(vid_range.first, vid_range.second);
        },
        [&](std::pair<VertexId,VertexId> vid_range){
                LabelsOfVertices.save();
        }
```

```
);
```

3. Optimizations:

There can be many more optimizations for this code to be more effective. One of them is checking if the code is converged. So that when the code is converged then the code does not need to run any more and the loop of iterations breaks. Also another optimization can be in change of BigVector to simple vector and use mutex later on as BigVector does not allow to define temporary grid to be dynamic according to the user input. So I put a constraint of 20 in this code below:

```
BigVector<double [20]> TempLabelGrid(graph.path+"/TempLabelGrid", graph.vertices);
```

Now the limitation of program is that it cannot process more than 20 labels unless it is changed in code. It can calculate more than 20 but it will be highly inefficient and a lot more IO will be used. So instead of using this, a vector of vector can be used to solve this problem.

4. How to run:

After downloading all the data, go to LabelPropagation folder and run terminal. There do following command:

```
make
```

Then after than use pre process command like this

```
./bin/preprocess -i [input path] -o [output path] -v [vertices] -p [partitions] -t
[edge type: 0=unweighted, 1=weighted]
```

The example command of pre process is following:

```
./bin/preprocess -i data_input -o data_grid -v 4847571 -p 4 -t 0
```

The data_input is the path of the graph.

Then after successfully doing pre processing of data, follow these commands:

```
./bin/label_prop [path] [total no. of lables] [total no. of iterations] [memory
budget in GB]
```

Following command is for 10 iterations with 10 number of iterations and 1GB memory budget

```
./bin/label_prop data_grid 10 10 1
```

5. Performance Results on Datasets:

I ran all the programs on Linux machine (UBUNTU) and following were the specs of Laptop:

Microprocessor	2.4 GHz Intel Core i7-4700MQ

Microprocessor Cache	6 MB L3 cache
Memory	8 GB 1600 MHz DDR3L
Video Graphics	NVIDIA GeForce GT 740M (2 GB DDR3 dedicated)
Hard Drive	1 TB 5400 rpm SATA

The Main difference was that hard drive was HDD not SSD, so the outputs were slow.

Following is the output of Dataset LiveJournal with 10 iteration, 10 labels and 1 GB memory:

```
hasan@hasanpc:~/Desktop/hasan/GridGraph-master$ sudo ./bin/label_prop live_grid 10 10 1
Number of Labels = 10
Number of Iterations = 10
Memory Budget = 1 GB
Degrees and Probability computation of each Vertex was completed in 0.23 seconds
10 number of Labels were randomly allocated to vertices/nodes in 0.00 seconds.
Iteration no. 1 in process...   Completed in 0.29 seconds
Iteration no. 2 in process...   Completed in 0.28 seconds
Iteration no. 3 in process...   Completed in 0.29 seconds
Iteration no. 4 in process...   Completed in 0.35 seconds
Iteration no. 5 in process...   Completed in 0.56 seconds
Iteration no. 6 in process...   Completed in 0.42 seconds
Iteration no. 7 in process...   Completed in 0.39 seconds
Iteration no. 8 in process...   Completed in 0.37 seconds
Iteration no. 9 in process...   Completed in 0.38 seconds
Iteration no. 10 in process...  Completed in 0.37 seconds
Time taken by 10 number of iterations inside label propagation algorithm =  4.06 seconds
Number of vertices of Label 0 = 1
Number of vertices of Label 1 = 959401
Number of vertices of Label 2 = 3402
Number of vertices of Label 3 = 660389
Number of vertices of Label 4 = 311659
Number of vertices of Label 5 = 1410443
Number of vertices of Label 6 = 144394
Number of vertices of Label 7 = 1
Number of vertices of Label 8 = 909847
Number of vertices of Label 9 = 637
Time taken for calculating number of nodes/vertices of each label =  0.01 seconds
Total time taken is whole execution =  4.07 seconds
```

Following is the output of Dataset Twitter with 10 iteration, 10 labels and 1 GB memory:

```
hasan@hasanpc:~/Desktop/hasan/GridGraph-master$ sudo ./bin/label_prop twitter_grid 10 10 1
Number of Labels = 10
Number of Iterations = 10
Memory Budget = 1 GB
Degrees and Probability computation of each Vertex was completed in 148.73 seconds
10 number of Labels were randomly allocated to vertices/nodes in 0.45 seconds.
Iteration no. 1 in process...   Completed in 149.97 seconds
Iteration no. 2 in process...   Completed in 150.07 seconds
Iteration no. 3 in process...   Completed in 145.24 seconds
Iteration no. 4 in process...   Completed in 145.88 seconds
Iteration no. 5 in process...   Completed in 142.44 seconds
Iteration no. 6 in process...   Completed in 140.27 seconds
Iteration no. 7 in process...   Completed in 138.22 seconds
Iteration no. 8 in process...   Completed in 137.61 seconds
Iteration no. 9 in process...   Completed in 137.53 seconds
Iteration no. 10 in process...  Completed in 137.80 seconds
Time taken by 10 number of iterations inside label propagation algorithm =  1574.21 seconds
Number of vertices of Label 0 = 10979890
Number of vertices of Label 1 = 9433
Number of vertices of Label 2 = 438
Number of vertices of Label 3 = 300
Number of vertices of Label 4 = 81
Number of vertices of Label 5 = 9661
Number of vertices of Label 6 = 49
Number of vertices of Label 7 = 8
Number of vertices of Label 8 = 275
Number of vertices of Label 9 = 91
Time taken for calculating number of nodes/vertices of each label =  0.18 seconds
Total time taken is whole execution =  1574.39 seconds
```

Following is the output of Dataset UK with 10 iteration, 10 labels and 1 GB memory:

```
hasan@hasanpc:~/Desktop/hasan/GridGraph-master$ sudo ./bin/label_prop uk_grid 10 10 1
Number of Labels = 10
Number of Iterations = 10
Memory Budget = 1 GB
Degrees and Probability computation of each Vertex was completed in 418.96 seconds
10 number of Labels were randomly allocated to vertices/nodes in 0.10 seconds.
Iteration no. 1 in process...   Completed in 410.13 seconds
Iteration no. 2 in process...   Completed in 408.80 seconds
Iteration no. 3 in process...   Completed in 421.31 seconds
Iteration no. 4 in process...   Completed in 424.47 seconds
Iteration no. 5 in process...   Completed in 420.86 seconds
Iteration no. 6 in process...   Completed in 408.86 seconds
Iteration no. 7 in process...   Completed in 410.43 seconds
Iteration no. 8 in process...   Completed in 404.41 seconds
Iteration no. 9 in process...   Completed in 405.00 seconds
Iteration no. 10 in process...  Completed in 400.49 seconds
Time taken by 10 number of iterations inside label propagation algorithm =  4114.77 seconds
Number of vertices of Label 0 = 3418
Number of vertices of Label 1 = 653
Number of vertices of Label 2 = 89661
Number of vertices of Label 3 = 28849
Number of vertices of Label 4 = 46115
Number of vertices of Label 5 = 28511
Number of vertices of Label 6 = 26158818
Number of vertices of Label 7 = 2
Number of vertices of Label 8 = 4
Number of vertices of Label 9 = 32215
Time taken for calculating number of nodes/vertices of each label =  0.39 seconds
Total time taken is whole execution =  4534.22 seconds
```

DataSet	Iterations	Labels	Memory	Time
LiveJournal	10	10	1GB	4.07 Seconds
Twitter	10	10	1GB	1574 Seconds
UK	10	10	1GB	4534 Seconds