Report 4: k-way graph partitioning using JaBeJa

Group 150: Lorenzo Deflorian, Riccardo Fragale

December 6, 2025

1 Introduction

This assignment aims to understand distributed graph partitioning using gossip-based peer-to-peer techniques; focusing in particular on the strategy suggested in the following paper[1]. We were given the skeleton of the algorithm and we needed to implement a couple of very important features and then doing some tests and experimenting possible improvements.

2 Task 1

This first task was related to the initial implementation of the JaBeJa class. The algorithm provides a smart and decentralized solution to partition large graphs by randomly assigning classes (colors) to the nodes and computing the local energy between nodes, their connections and an assigned random subset of ndoes. The local energy corresponds to the entropy between the connection of a ndoe and its partners (connected node and subset). Therefore, the goal of the algorithm is to reduce the entropy in order to define the best partitioning into clusters. To reduce it, in each round of the algorithm the energy of each node is compared to their partners by means of computing their energy with its current color and when switching colors with the presented partner. If the energy between the node and partner reduces when they switch colors, it means that a better balance has been found and the node and the partner should swap color. We were asked to complete the implementation of the methods **sampleAndSwap(...)** and **findPartner(...)**.

[1] `https://publicatio.bibl.u-szeged.hu/5295/1/taas15.pdf`

2.1 Implementation Details

```java
public Node findPartner(int nodeId, Integer[] nodes){

  Node nodep = entireGraph.get(nodeId);

  Node bestPartner = null;
  double highestBenefit = 0;

  // TO BE CHECKED AND FINISHED
  for(Integer q: nodes){
    Node nodeq = entireGraph.get(q);
    //Computing actual utility
    int degreeQQ = getDegree(nodep, nodep.getColor());
    int degreePP = getDegree(nodeq, nodeq.getColor());
    double oldU = Math.pow(degreePP, config.getAlpha())+Math.pow(degreeQQ, config.getAlpha());

    //Computing utility after hypothetical swap
    int degreePQ = getDegree(nodep, nodeq.getColor());
    int degreeQP = getDegree(nodeq, nodep.getColor());
    double newU = Math.pow(degreePQ, config.getAlpha())+Math.pow(degreeQP, config.getAlpha());

    if(useOriginalSimulatedAnnealing)
    {
      double U = (newU * T) - oldU;
      if (U > highestBenefit) {
          highestBenefit = U;
          bestPartner = nodeq;
      }
    }
    else
    {
      double prob = Math.random();
      double acceptance = Math.exp((newU-oldU)/T);
      if (acceptance > prob && acceptance > highestBenefit) {
          highestBenefit = acceptance;
          bestPartner = nodeq;
      }
    }


  }

  return bestPartner;
}
```

Figure 1: Partner swapping mechanism in JaBeJa algorithm

The first part computes the degrees and the energy with and without a swap for the node with all its partners; then using annealing (or not) we compute the acceptance probability if needed and we assign the bestPartner to swap with for a node. Annealing is a concept related especially for the second and third task of this homework.

```java
private void sampleAndSwap(int nodeId) {
    Node partner = null;
    Node nodep = entireGraph.get(nodeId);

    if (config.getNodeSelectionPolicy() == NodeSelectionPolicy.HYBRID
            || config.getNodeSelectionPolicy() == NodeSelectionPolicy.LOCAL) {
      // swap with random neighbors
      // TODO
      partner = findPartner(nodeId, getNeighbors(nodep));
    }

    if (config.getNodeSelectionPolicy() == NodeSelectionPolicy.HYBRID
            || config.getNodeSelectionPolicy() == NodeSelectionPolicy.RANDOM) {
      // if local policy fails then randomly sample the entire graph
      // TODO
      if (partner == null) {
        partner = findPartner(nodeId, getSample(nodeId));
      }
    }

    // swap the colors
    // TODO
    if (partner != null) {
        int aux = partner.getColor();
        partner.setColor(nodep.getColor());
        nodep.setColor(aux);
        numberOfSwaps++;
    }
  }
```

Figure 2: Our sample and swap implementation

The function sampleAndSwap selects a candidate partner for a given node using the configured node-selection policy (LOCAL, RANDOM, or HYBRID), by sampling neighbors first and falling back to a global sample. It calls findPartner on the sampled nodes to evaluate swap utility and chooses the best partner if one improves or is probabilistically accepted. When a partner is found, sampleAndSwap swaps the two nodes' colors and increments the swap counter, applying the change immediately to the graph.

3 Task 2 and Task 3

The danger of the previous implementation is that our algorithm is non-deterministic and we might end up into a local minimum where we find a good solution but not the optimal one. In order to try to solve this issue the second task required us to implement a linear simulated annealing technique that decreases the temperature over time, allowing the algorithm to escape local minima by accepting worse solutions with decreasing probability as iterations progress[2]. We were also asked to experiment a bit with all the parameters and to understand that we are improving our solutions.

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

```
1  private void saCoolDown(){
2      // TODO for second task
3      if(useOriginalSimulatedAnnealing)
4      {
5        T = T - config.getDelta();
6          if (T < 1)
7          {
8            T = 1;
9          }
10     }
11     else
12     {
13       T *= config.getDelta();
14
15         if (T < MIN_T)
16           {
17             T = MIN_T;
18           }
19
20         if (T == MIN_T) {
21             reset_rounds++;
22             if (reset_rounds >= 400) {
23                 T = 1;
24                 // T = 2 REHEATING
25                 reset_rounds = 0;
26             }
27         }
28     }
29   }
```

Figure 3: saCoolDown

The boolean originalSimulatedAnnealing tells us whether we are using the linear annealing or the exponential one (essential for task 3). That's why the code above is basically divided into two different "sections". Task 3 asked us also to define our acceptance probability function, that is computed inside the findPartner function whose code is inside 1. We are calculating the acceptance probability as

```
double acceptance = Math.exp((newU-oldU)/T);
```

This probability gives to the algorithm a sort of selectivity when accepting a swap and defines in a way the speed through which the algorithm converges to a good solution.

I will paste below some pictures of the tests done with the different configurations of task 1, 2 and 3.

4 Test

We worked on the following 3 graphs:

- 3elt
- add20
- facebook

First of all we realized that the HYBRID selection policy is by far the best with respect to RANDOM and LOCAL. In particular, since in local there is a reduced knowledge of the graph for a single node, the time it needs to converge is a bit more so maybe increasing the number of rounds would lead to better results. The following three results are related to a test with 1000 rounds, delta = 0.003 and linear annealing (all three for the Facebook graph).

```
round: 999,edge cut: 180264,swaps: 20621248,migrations: 46805 LOCAL
round: 999,edge cut: 122756,swaps: 17736954,migrations: 47840 RANDOM
round: 999,edge cut: 118087,swaps: 21208716,migrations: 47731 HYBRID
```

A second parameter we tested is the influence of *delta*; in particular it is by default set to 0.003; we tried also different values and we realized that the best results are obtained with a lower delta, such as 0.01. This was proved with annealing and with RANDOM selection policy.

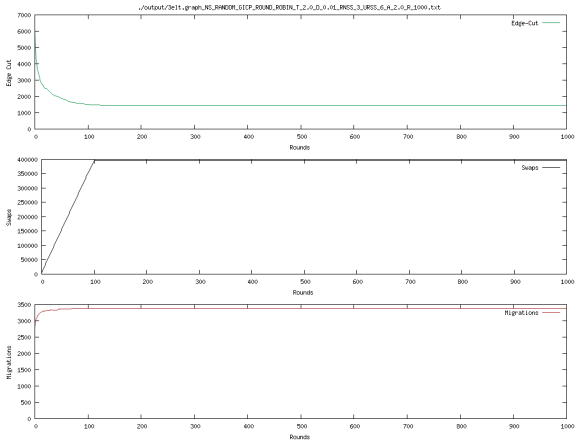

Figure 4: Delta 0.01 on 3elt graph

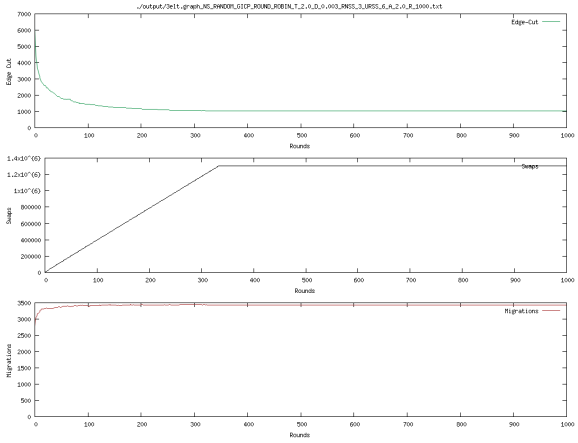

Figure 5: Delta 0.003 on 3elt graph

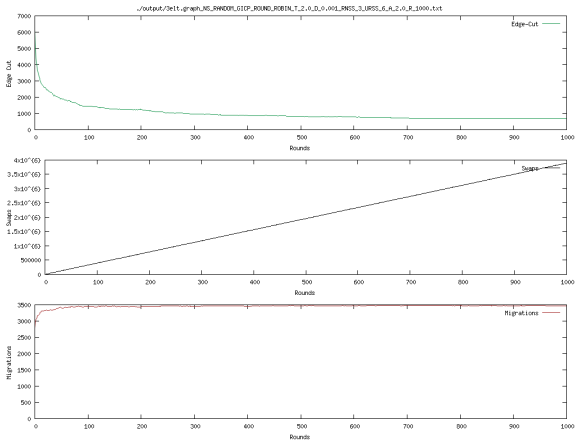

Figure 6: Delta 0.001 on 3elt graph

As 6 shows we are obtaining a better partitioning in terms of edge-cuts but the number of swaps has not reached a plateau; this implies that we could have gone on with a higher number of rounds.

A third aspect is related to the influence of exponential annealing; we can show that once the parameter T reaches its final value (that is, no more bad swaps are allowed), Ja-Be-Ja converges to an edge cut rapidly, and the edge cut does not change over time. For example, if T is 2 and delta is 0.01, then after 200 rounds, the temperature will cool down to 1, and no more bad-swaps will be accepted.

From the figure 7 it is not very clear but from the log it is quite significant that exacty after 200 rounds there are no swaps happening even though from the graph it seems that swaps stops after 100 rounds. It is clearly a limit of this algorithm as we are not improving from a certain point onwards.

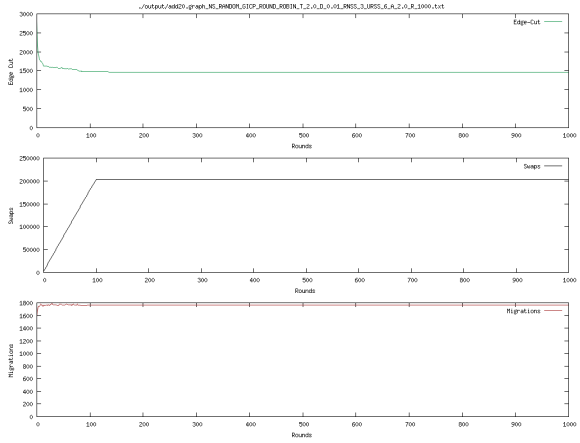

Figure 7: Delta 0.01 T=2.0 on add20 graph

Even by adding a reheat after 400 rounds we weren't able to find any major improvement.

Last, but not the last we tried to analyze the influence of changing tha acceptance probably using a different one with respect to the one provided in the skeleton of jabeja.

5 Conclusion

This laboratory let us try and test a very useful algorithm for graph partitioning. It was very fun and useful at the same to see how performances were changing when working on the variables and on the techniques to better apply Ja-Be-Ja. In a way it was a sort of conclusion of the course and we would have liked to completely implement the algorithm on our own instead of having to use the original skeleton.