Data Mining - Assignment 5

Group 18
Gustavo Teodoro D. Beck - Fernando García Sanz

December 6, 2020

1 Ja-Be-Ja implementation

In order to process large graphs, is desirable that the graph is partitioned into different hosts (processors) to assess computations. Therefore, is required that the graphs are partitioned in way that graph partitions' have a strong relationship, otherwise the unrelated nodes will be sent to a host and few reliable and important analysis would made.

Ja-Be-Ja algorithm [1] 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 nodes. The local energy corresponds to the entropy between the connections of a node and its partners (connected nodes and subset). Therefore, the goal of the algorithm is to reduce this entropy (energy) in order to define a balanced partitioning. To reduce this entropy, the nodes are 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 partner should swap colors.

Our implementations of Ja-Be-Ja can be seen as follows:

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

  Node nodep = entireGraph.get(nodeId);

  Node bestPartner = null;
  double highestBenefit = 0;

  // TODO
  for(Integer q: nodes) {
    Node nodeq = entireGraph.get(q);
    int degree_pp = getDegree(nodep, nodep.getColor());
    int degree_qq = getDegree(nodeq, nodeq.getColor());
    double old_d = Math.pow(degree_pp, config.getAlpha()) + Math.pow(degree_qq, config.getAlpha());
    int degree_pq = getDegree(nodep, nodeq.getColor());
    int degree_qp = getDegree(nodeq, nodep.getColor());
    double new_d = Math.pow(degree_pq, config.getAlpha()) + Math.pow(degree_qp, config.getAlpha());
    if (annealing) {
      Random random = new Random();
      double prob = random.nextDouble();
      double acceptance = computeAcceptance(new_d, old_d);
      // Check this in order to avoid having a 100% acceptance rate (convergence)
      if (new_d != old_d  && acceptance > prob && acceptance > highestBenefit) {
        bestPartner = nodeq;
        highestBenefit = acceptance;
      }
    }
    else {
      if(new_d * T > old_d && new_d > highestBenefit) {
      bestPartner = nodeq;
      highestBenefit = new_d;
      }
    }
  }

  return bestPartner;
}
```

Figure 1: findPartner function.

```
/**
 * Sample and swap algorith at node p
 * @param nodeId
 */
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: sampleAndSwap function.

```java
/**
 * Simulated annealing cooling function
 */
private void saCoolDown(){
  // TODO for second task
  if (annealing){
    exponent_round++;
    T *= config.getDelta();
    if (T < MIN_T) {
      T = MIN_T;
    }
    if (T == MIN_T) {
    reset_rounds++;
      if (reset_rounds == 400) {
        T = 1;
        reset_rounds = 0;
        exponent_round = 0;
      }
    }
  }
  else {
    if (T > 1)
    T -= config.getDelta();
    if (T < 1)
    T = 1;
  }
}
```

Figure 3: Simulated annealing coolDown function.

```java
/**
 * Computing the acceptance probability
 */
private double computeAcceptance(double new_val, double old_val){
  if (FLAG.equals("MyFun")) {
    return Math.exp((new_val - old_val) / Math.pow(T, exponent_round));
  }
  else {
   return Math.exp((new_val - old_val) / T);
  }
}
```

Figure 4: computeAcceptance function.

In order to not get stuck into a local minimum, Ja-Be-Ja employs a linear simulated annealing method based on the temperature of the systems, which cools down during the process to reduce the chances of passing by the global minimum.

The outcomes of the Ja-Be-Ja are going to be tested on the `3elt` and `add20`, and can be seen as follows:

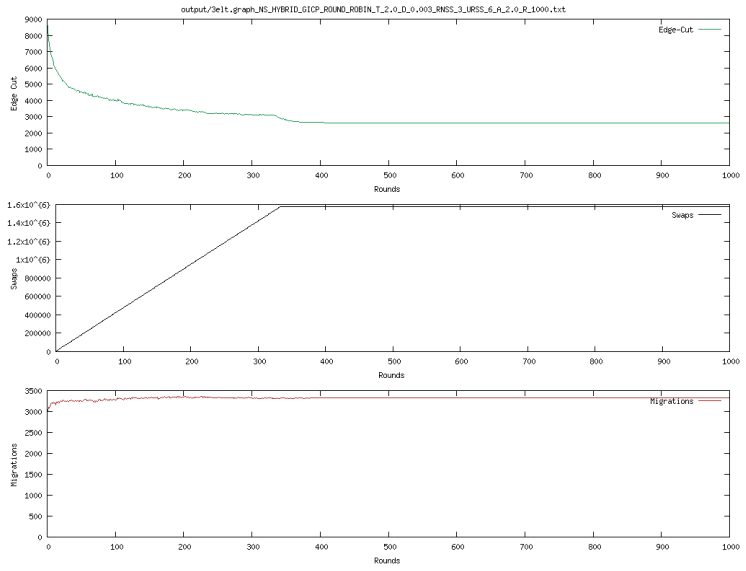

Figure 5: `3elt` graph with original Ja-Be-Ja.

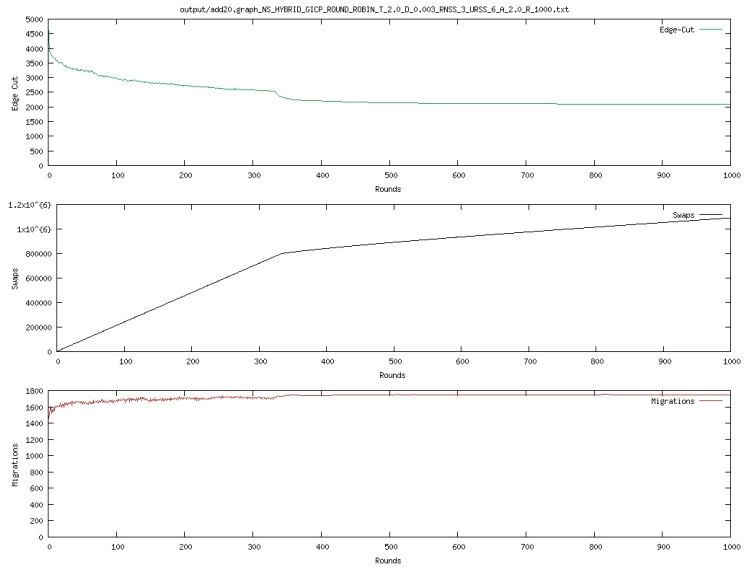

Figure 6: `add20` graph with original Ja-Be-Ja.

As can be seen, the convergence of the method is dictated by the the simulated annealing, which in this case updates linearly the temperature. This can be clearly seen in the number of swaps. Therefore, in the next section we will explore exponential updates of the temperature as well as other strategies for faster convergence in terms of number of edge-cuts.

2 Effect of the simulated annealing parameters and the acceptance probability function

As an improvement of the Ja-Be-Ja algorithm, the simulated annealing technique can be also performed as described in [2]. This provides a faster (exponential) cool down of the temperature and defines a different acceptance criteria in order to define the best partner of the current node. However, in order to guarantee that we do not cool down to fast, we reset to the initial temperature our temperature after 400 rounds "convergence". This will show us if our model had found the global minimum or if it was stuck into a local minimum due to a fast cool down.

Our implementation of the proposed simulated annealing can be seen in figure 3 and 4. And the outcomes as follows:

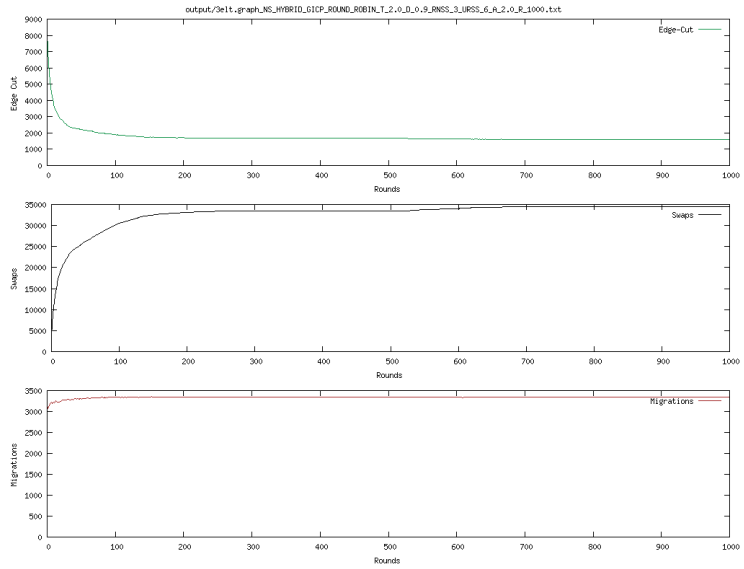

Figure 7: `3elt` graph with proposed simulated annealing.

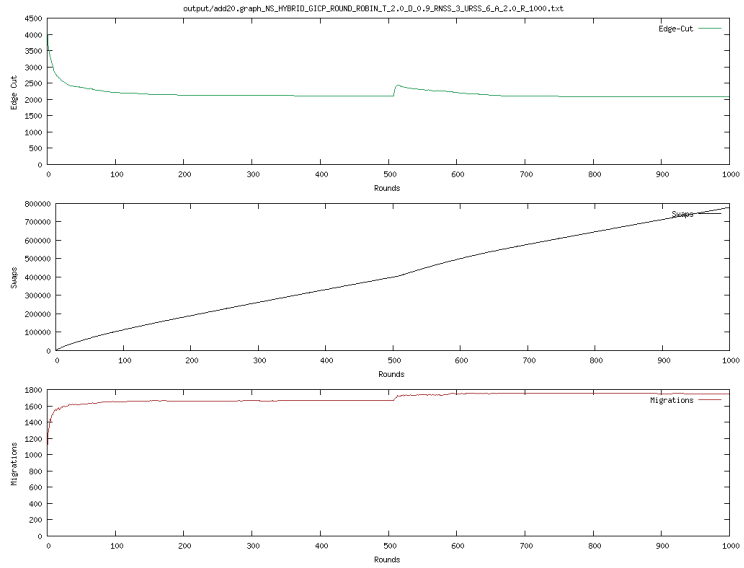

Figure 8: `add20` graph with proposed simulated annealing.

It can be seen that the usage of exponential annealing results into smoother shapes in the number of swaps, and a faster convergence of the other two metrics. It is relevant to say that when the temperature is reseted, a subtle increment in the recorded metrics is experienced. This is way more notorious in the `add20` graph than in the `3elt` one. This is what "unblocks" the algorithm and allows it to look for lower minimums, although in these cases the values converge around the same intervals than before.

3 Own acceptance probability function

As observed in the previous sections, the temperature and defining the best partner have a huge impact on the algorithm. The idea behind our proposal relies on exploring as much as possible in the beginning, i.e. migrating many nodes. We do this by rising the temperature value to the

round number (resetting it when the temperature is reseted). This will rapidly increase the highest acceptance value, which at some point will make the algorithm very selective on the partners that should swap colors or not. Therefore, our simulated annealing cost functions tends to converge faster than the previous implementations.

The outcomes of the Ja-Be-Ja with our own acceptance function are going to be tested on the `3elt` and `add20`, and can be seen as follows:

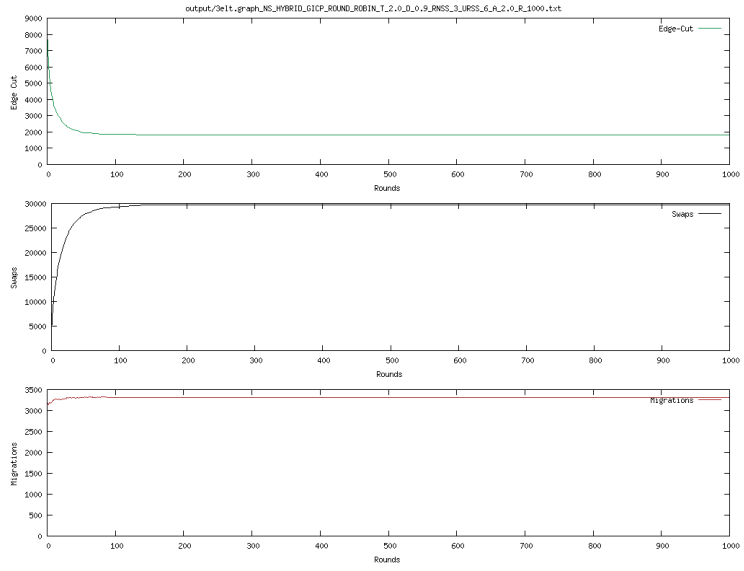

Figure 9: `3elt` graph with own acceptance function.

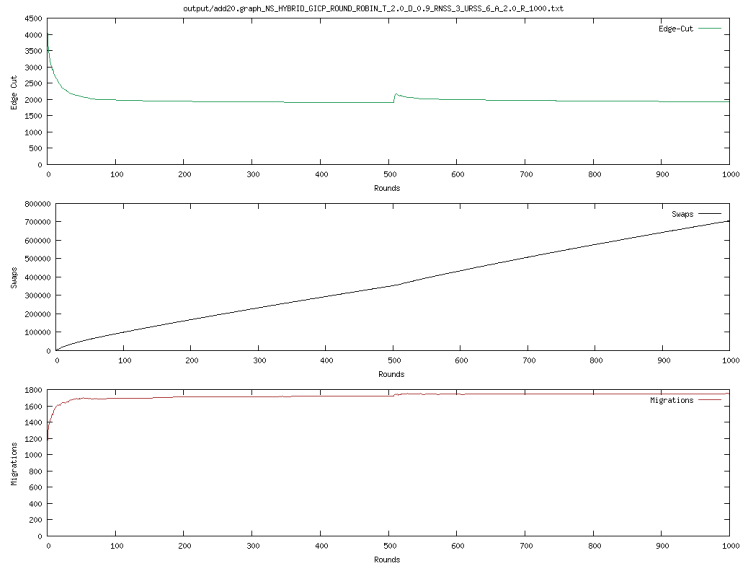

Figure 10: `add20` graph with own acceptance function.

By means of the outcomes, it can be seen that, as we expected, the edge cuts and migrations converged before than in the previous section, and that the results are practically the same as those obtained with the original annealing function.

As an extra experiment, we wanted to analyze the results of increasing the number of clusters in which the algorithm has to classify the different nodes. We did this with the `3elt` graph, increasing

also the number of maximum rounds executed. The obtained results are as follows:

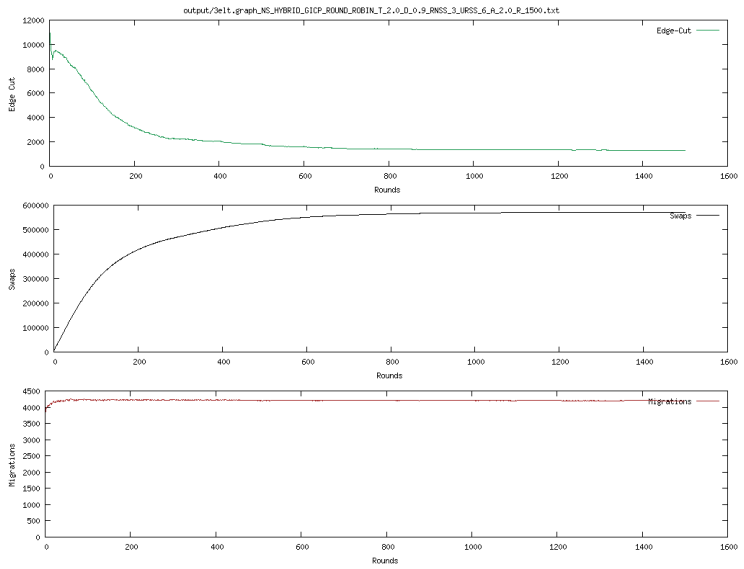

Figure 11: `3elt` graph with own acceptance function and $k = 10$.

As can be seen, the algorithm also converges, although it requires more rounds to do so. The numbers of swaps and migrations have increased, as a result of the increment in the number of clusters, but it is interesting to see that the number of edge cuts is quite similar to the one retrieved when we only used a total of 4 clusters.

4 How to run

In order to properly run this algorithm, it is necessary to install `Maven` in order to build the project, and `gnuplot` in order to display the obtained results in a plot format. It turns out that the code of the bash script `plot.sh` is fully functional only in Linux, as it uses the `xdg-open` command. Therefore, it is necessary to modify the script commenting the last code line, so the program does not break, and manually opening the generated plot later on. `gnuplot` can also be tricky to install in environments different from pure Linux, in our case, macOS, but by using the latest command line beta and `brew` package manager, it is possible to sort this issue out.

The steps to follow are:

- Modify the algorithm parameters if necessary in `CLI.java`.
- Execute `compile.sh`.
- Execute `run.sh -graph <graph-path>`.
- Execute `plot.sh <output-path>`.
- Open the file `plot.png`.

References

[1] F. Rahimian, A. H. Payberah, S. Girdzijauskas, M. Jelasity, and S. Haridi, "Ja-be-ja: A distributed algorithm for balanced graph partitioning," in *2013 IEEE 7th International Conference on Self-Adaptive and Self-Organizing Systems*, pp. 51–60, 2013.

[2] K. E. Geltman, "The simulated annealing algorithm." `http://katrinaeg.com/simulated-annealing.html`, Feb 2014.