Data Mining Assignment 5

Name : Rigved S. Patki
E-mail : patki@kth.se
Group: Assignment group 8

Topic : K-way Graph Partitioning Using JaBeJa

Solution :

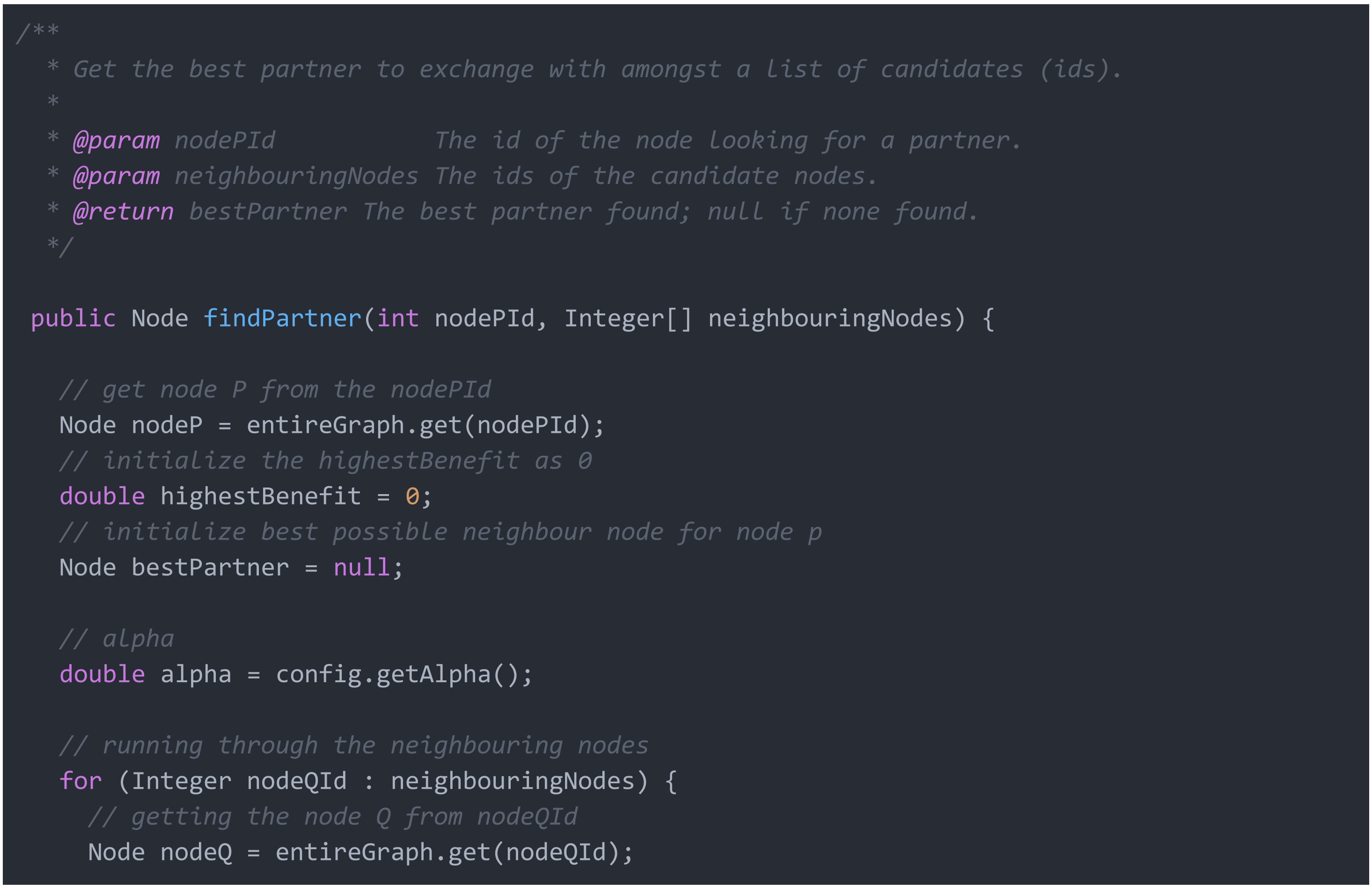
The implementation of the assignment is available on Github . Each branch of the code represents the solution to each task, which is as follows:

- Master Branch : It consists of the code for the solution of task-1.
- Task-2 Branch: It consists of the code for the solution of task-2.
- Optional Branch: It consists of the code for the solution of optional task.

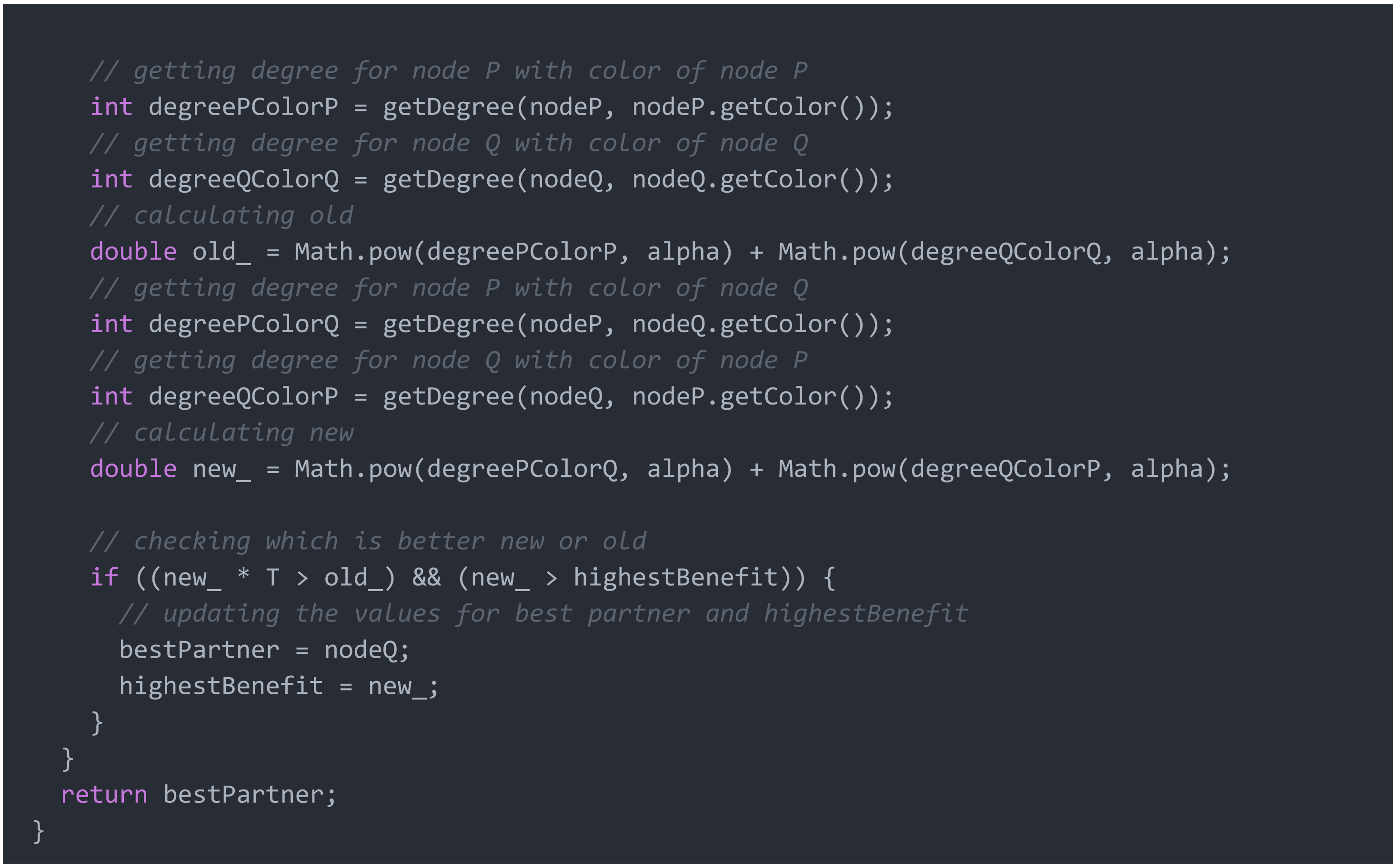
1. Task-1 :

For this task I modified the code in Jabeja.java file at the placed where //TODO comments were present.

First I added the code for the function findPartner. The code is as follows:

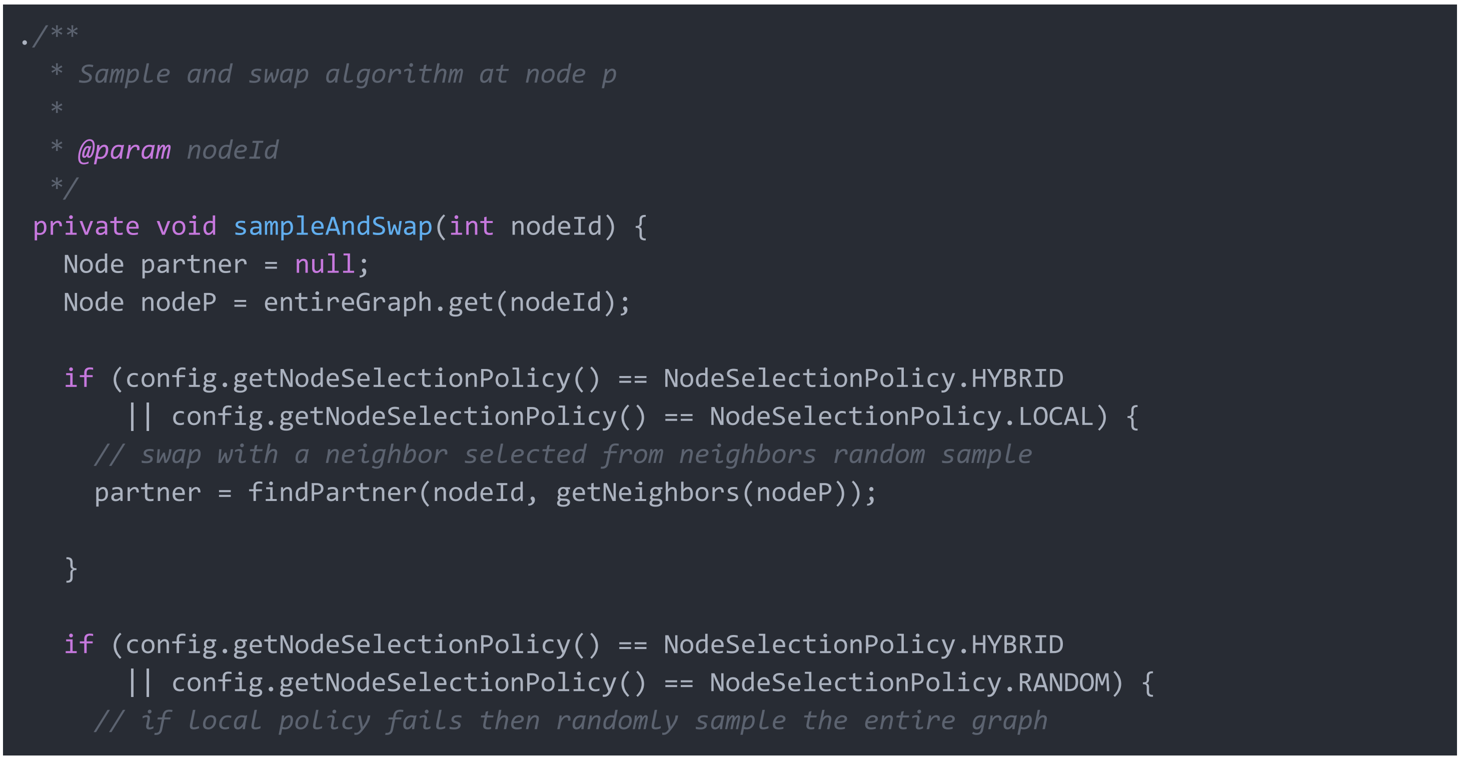


```java
/**
  * Get the best partner to exchange with amongst a list of candidates (ids).
  *
  * @param nodePId           The id of the node looking for a partner.
  * @param neighbouringNodes The ids of the candidate nodes.
  * @return bestPartner The best partner found; null if none found.
  */

 public Node findPartner(int nodePId, Integer[] neighbouringNodes) {

   // get node P from the nodePId
   Node nodeP = entireGraph.get(nodePId);
   // initialize the highestBenefit as 0
   double highestBenefit = 0;
   // initialize best possible neighbour node for node p
   Node bestPartner = null;

   // alpha
   double alpha = config.getAlpha();

   // running through the neighbouring nodes
   for (Integer nodeQId : neighbouringNodes) {
     // getting the node Q from nodeQId
     Node nodeQ = entireGraph.get(nodeQId);
```

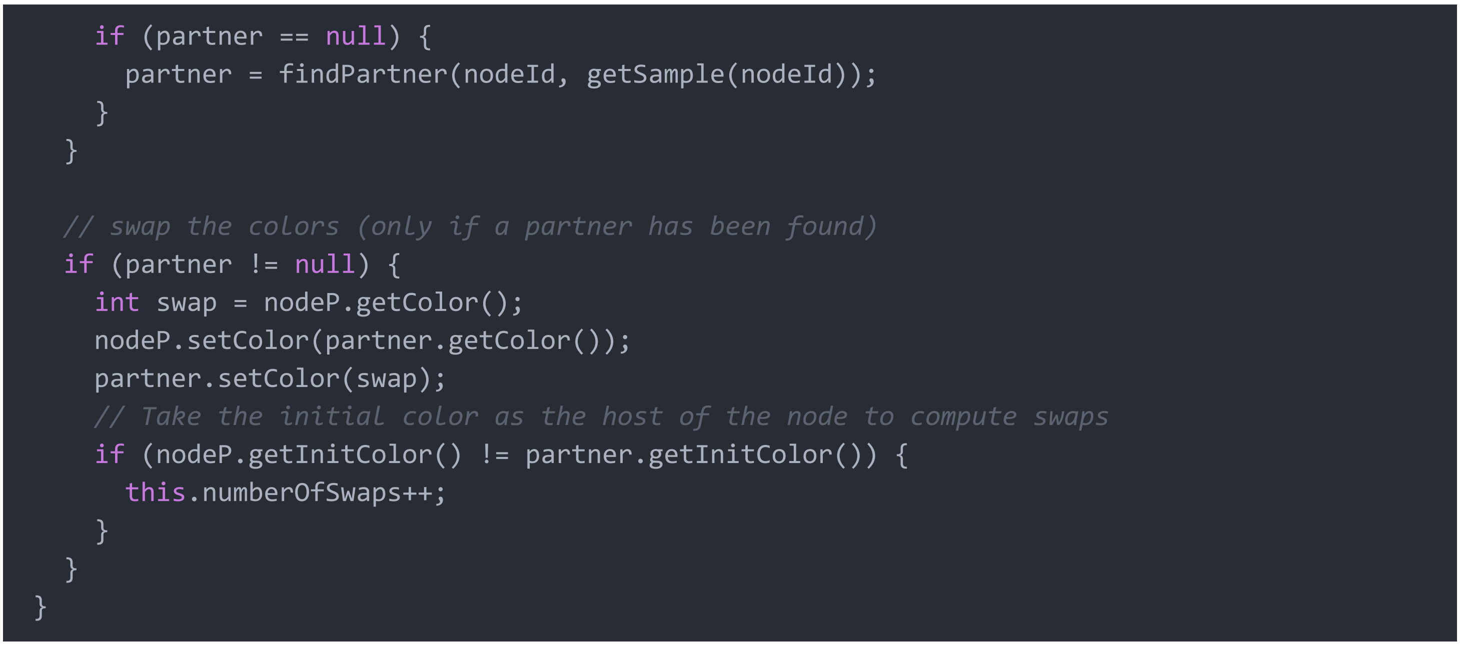


```
    // getting degree for node P with color of node P
    int degreePColorP = getDegree(nodeP, nodeP.getColor());
    // getting degree for node Q with color of node Q
    int degreeQColorQ = getDegree(nodeQ, nodeQ.getColor());
    // calculating old
    double old_ = Math.pow(degreePColorP, alpha) + Math.pow(degreeQColorQ, alpha);
    // getting degree for node P with color of node Q
    int degreePColorQ = getDegree(nodeP, nodeQ.getColor());
    // getting degree for node Q with color of node P
    int degreeQColorP = getDegree(nodeQ, nodeP.getColor());
    // calculating new
    double new_ = Math.pow(degreePColorQ, alpha) + Math.pow(degreeQColorP, alpha);

    // checking which is better new or old
    if ((new_ * T > old_) && (new_ > highestBenefit)) {
      // updating the values for best partner and highestBenefit
      bestPartner = nodeQ;
      highestBenefit = new_;
    }
  }
  return bestPartner;
}
```

Next I modified the code for the function sampleAndSwap, which would use the findPartner function to sample and swap between neighbouring nodes based on the node selection policy.



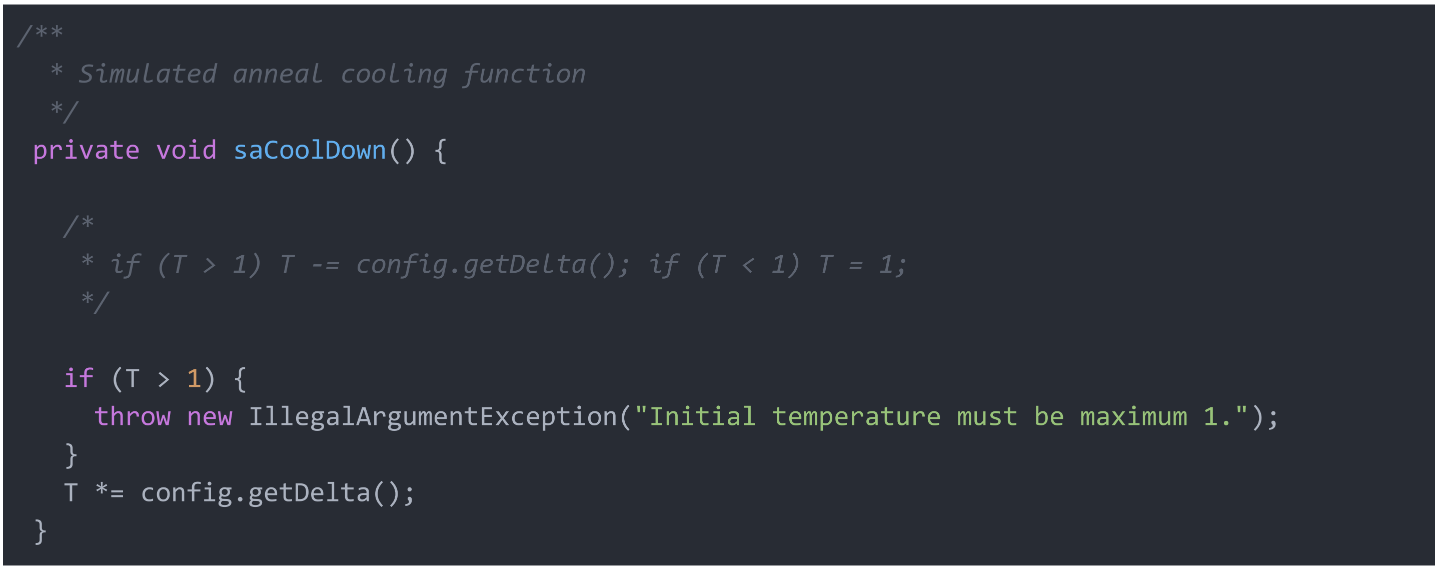
```
./**
  * Sample and swap algorithm at node p
  *
  * @param nodeId
  */
 private void sampleAndSwap(int nodeId) {
   Node partner = null;
   Node nodeP = entireGraph.get(nodeId);

   if (config.getNodeSelectionPolicy() == NodeSelectionPolicy.HYBRID
       || config.getNodeSelectionPolicy() == NodeSelectionPolicy.LOCAL) {
     // swap with a neighbor selected from neighbors random sample
     partner = findPartner(nodeId, getNeighbors(nodeP));

   }

   if (config.getNodeSelectionPolicy() == NodeSelectionPolicy.HYBRID
       || config.getNodeSelectionPolicy() == NodeSelectionPolicy.RANDOM) {
     // if local policy fails then randomly sample the entire graph
```



```
      if (partner == null) {
        partner = findPartner(nodeId, getSample(nodeId));
      }
    }

    // swap the colors (only if a partner has been found)
    if (partner != null) {
      int swap = nodeP.getColor();
      nodeP.setColor(partner.getColor());
      partner.setColor(swap);
      // Take the initial color as the host of the node to compute swaps
      if (nodeP.getInitColor() != partner.getInitColor()) {
        this.numberOfSwaps++;
      }
    }
  }
```

Taks-2:

For the second task, I made changes to Jabeja.java to use a different approach for the simulated annealing that would decrease T exponentially (T) instead of decreasing it linearly.



```
/**
  * Simulated anneal cooling function
  */
 private void saCoolDown() {

   /*
    * if (T > 1) T -= config.getDelta(); if (T < 1) T = 1;
    */

   if (T > 1) {
     throw new IllegalArgumentException("Initial temperature must be maximum 1.");
   }
   T *= config.getDelta();
 }
```

The graphs as a result of these changes are present in the output folder of the master branch.

Optional Task :

As optional work, I decided to try out a different acceptance probability that I found to be used very widely by the community:

$$\frac{1}{1 + e^{\frac{Eold - Enew}{T}}}$$

This acceptance function has the peculiarity that when Eold > Enew (the considered solution yields to a worse state).

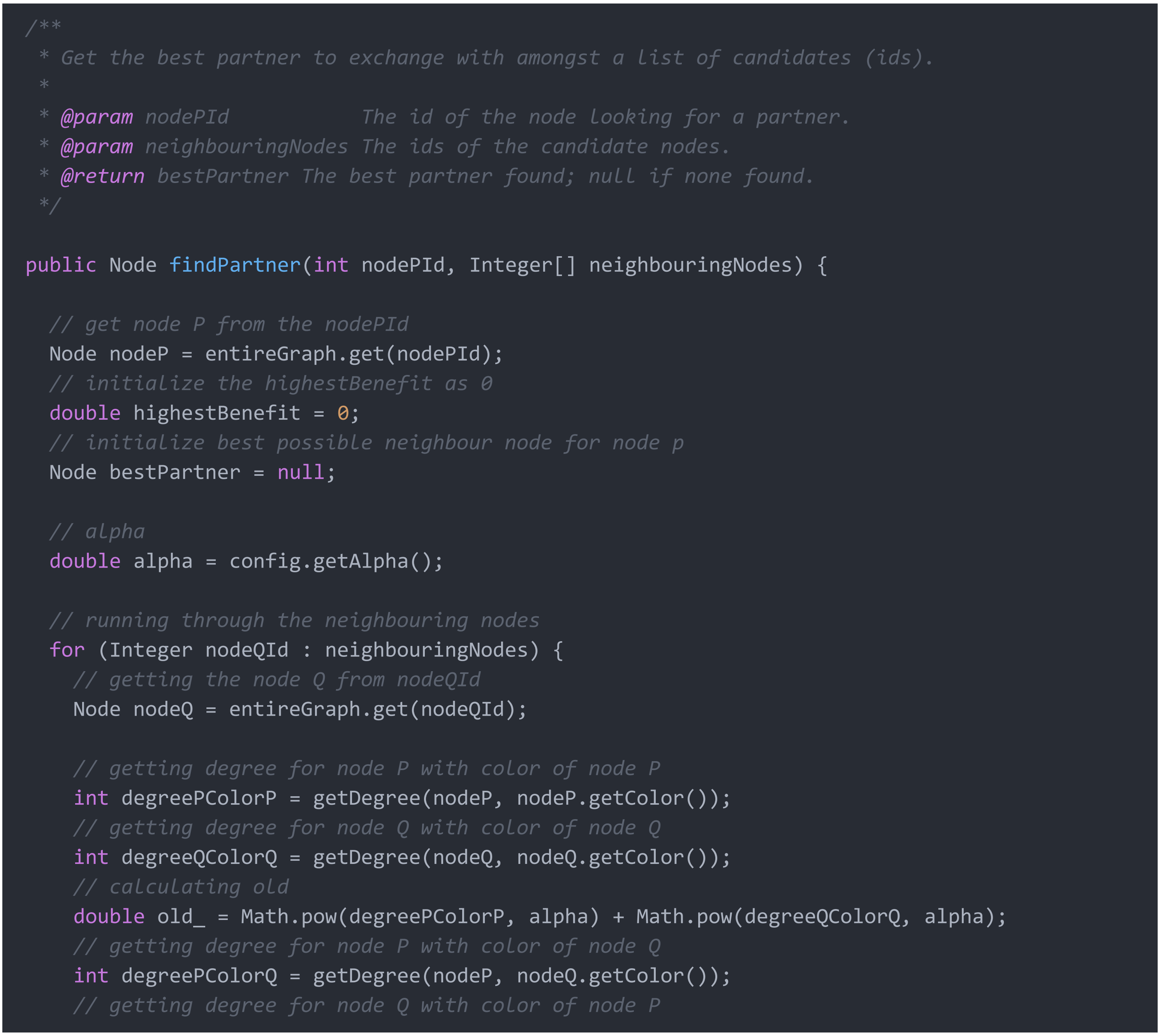
I also tried new cooling functions, motivated by those used in neural networks to update the learning rate, the exponential decay:

$$T_{K+1} = T_k * \delta^{k/100}$$

and the inverse time decay:

$$T_{k+1} = \frac{T_k}{1+\delta^k}$$

Updated Code :



```java
/**
 * Get the best partner to exchange with amongst a list of candidates (ids).
 *
 * @param nodePId           The id of the node looking for a partner.
 * @param neighbouringNodes The ids of the candidate nodes.
 * @return bestPartner The best partner found; null if none found.
 */

public Node findPartner(int nodePId, Integer[] neighbouringNodes) {

  // get node P from the nodePId
  Node nodeP = entireGraph.get(nodePId);
  // initialize the highestBenefit as 0
  double highestBenefit = 0;
  // initialize best possible neighbour node for node p
  Node bestPartner = null;

  // alpha
  double alpha = config.getAlpha();

  // running through the neighbouring nodes
  for (Integer nodeQId : neighbouringNodes) {
    // getting the node Q from nodeQId
    Node nodeQ = entireGraph.get(nodeQId);

    // getting degree for node P with color of node P
    int degreePColorP = getDegree(nodeP, nodeP.getColor());
    // getting degree for node Q with color of node Q
    int degreeQColorQ = getDegree(nodeQ, nodeQ.getColor());
    // calculating old
    double old_ = Math.pow(degreePColorP, alpha) + Math.pow(degreeQColorQ, alpha);
    // getting degree for node P with color of node Q
    int degreePColorQ = getDegree(nodeP, nodeQ.getColor());
    // getting degree for node Q with color of node P
    int degreeQColorP = getDegree(nodeQ, nodeP.getColor());
    // calculating new
    double new_ = Math.pow(degreePColorQ, alpha) + Math.pow(degreeQColorP, alpha);

    // checking which is better new or old
    // updating the values for best partner and highestBenefit
    /*
     * if ((new_ * T > old_) && (new_ > highestBenefit)) { bestPartner = nodeQ;
     * highestBenefit = new_; }
     */

    // Instead of cost use benefit as the difference between
    // new and old state
    double newBenefit = new_ - old_;

    // Apply acceptance probability to simulated annealing
    // based on benefit instead of cost (change sign: new - old)
    double ap = 0;
    if (newBenefit > highestBenefit) {
      ap = 1;
    } else {
      switch (config.getAcceptanceProbabilityMode()) {
      case 1:
        ap = Math.pow(Math.E, (newBenefit - highestBenefit) / T);
        break;

      case 2:
        ap = 1 / (1 + Math.pow(Math.E, (highestBenefit - newBenefit) / T));
        break;

      default:
        throw new IllegalArgumentException("The selected mode for ap is not valid.");
      }
    }
    if ((ap > RandNoGenerator.random()) && (T > Tmin || newBenefit > highestBenefit)) {
      bestPartner = nodeQ;
      highestBenefit = newBenefit;
    }

  }
  return bestPartner;
}
```