POLITECNICO DI TORINO

Dipartimento di Elettronica e Telecomunicazioni
Communications and computer network engineering

Tesi di Laurea Specialistica

Social networks deanonymization

On the performance of the Percolation Graph Matching algorithm
over synthetic graphs with community structure

Relatore
Emilio LEONARDI

Candidato
Fabio GENNARI

Ottobre 2018

Dedico questo lavoro ai miei genitori per il loro instancabile sostegno e ai miei amici più cari che sono sempre stati vicini e partecipi.

Un ringraziamento vada ai miei colleghi universitari con i quali ho collaborato e con cui mi sono confrontato in questi anni.

In ultimo, la mia gratitudine giunga alle persone che, con la loro gentilezza e premurosa presenza, hanno reso indimenticabile questo mio periodo torinese.

Index of pages

Introduction

Social interactions increasingly take place by means of social networks and other online services. Friendship ties and, more broadly, all kinds of activities performed by users such as communications, chats and information sharing inherently and inevitably leave electronic traces that represent sensitive information.

Such traces are indeed able to help in reconstructing one's personality and interests, therefore representing extremely valuable sources of information for governments, advertisers, data miners and many other entities. Moreover, such kind of reconstruction can become more accurate and complete if the targeted users belong to different networks and the corresponding extracted data are combined together.

Typically, that is the case, with many individuals belonging to more than one social networks, even though their identity in each of them might be different or inaccessible. In fact, very often network owners and operators release anonymized and possibly sanitized network graphs to commercial partners and academic researchers, in order to facilitate their work and, at the same time, protect the privacy of their users as much as possible. However, many studies have by now proved that such a condition is not an insurmountable obstacle.

In fact, we can represent the traces of the activities carried out by the network users by means of a graph in which nodes represent users, whereas edges show that a certain kind of activity has been performed by some user (a node) with another one. Of course, such traces can be more or less complete, providing information with different levels of detail that could be represented through the use of edge weights or other means. In the simplest and worst case, i.e., without the help of any side information and with the nodes labelled in a different way, also random, it's immediate to realize that the whole problem reduces to a graph matching one. Graph matching is the problem of finding a similarity between two graphs, which consists in finding a one-to-one correspondence or mapping between the vertex sets of such graphs, with only information about their topological structure.

More formally, given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, where $V_{1,2}$ are the vertex sets and $E_{1,2}$ are the edge sets of the two graphs respectively, with $|V_1| = |V_2|$, *exact graph matching* is the problem of finding a one-to-one mapping $f: V_1 \to V_2$ such that an edge $(u, v) \in E_1$ between two nodes $u, v \in V_1$ iff $\big(f(u), f(v)\big) \in E_2$. If such a mapping f exists, this is called an isomorphism. In most of the real cases, it is not possible to find it, since the two graphs could be obtained using different and partial information, and the graph matching problem should consist in finding the best matching between the vertex sets of the two graphs, or at least a subset of them, not an exact one. This type of problem is said to be *inexact graph matching*.

Graph matching has important applications also in other fields. For example, it can be used to recognize different scenes of the same image or similar images if the images are represented through graphs: regions in an image can be seen as vertices of a graph, and the adjacency relations between them can be seen as edges. In bioinformatics, gene sequences can be modeled as graphs, therefore graph matching is applied for gene/protein networks alignment [7, 9, 15]. In ontology alignment, it is used to match sets of labels describing data [16].

Previous work does show that it's possible to develop and design graph-matching algorithms that are able to find such a matching by using only the topological structures of graphs. In [12] Narayanan and Shmatikov act as an attacker intending to extract sensitive information from an accessible large-scale anonymized network graph through its de-

anonymization. The attacker is assumed to have also access to an auxiliary network serving as side information. Such an assumption is quite realistic: for example, parts of some online networks can be automatically crawled, or the attacker may have colluded with an operator of another network whose membership overlaps with the original one.

Their algorithm consists of two phases. In the preliminary one, the attacker must identify a set of "seed" nodes which belong to both graphs and map them. In the main phase, a propagation algorithm takes as input the two graphs and the seed mapping between them, returning a final, more extended mapping as output. The algorithm is called a propagation one since it finds new mappings exploiting the feedback from previously formed ones, in addition to the topological structure of the graphs. Specifically, at each iteration, the algorithm picks an arbitrary, still unmatched node i from one of the graphs $G_1 = (V_1, E_1)$ and, for each unmatched node j of the other graph $G_2 = (V_2, E_2)$, computes a matching score equal to the number of neighbors of i that have already been matched to neighbors of j. Various heuristics are then proposed to compare different pairs (i, j) that are candidates for being added to the mapping as a valid match.

The most peculiar one that is used in this algorithm is the eccentricity which is defined in [11] in the context of de-anonymizing databases. It is defined as

$$\frac{\max(X) - \max_2(X)}{\sigma(X)}$$

where max and $\max_2$ denote the highest and second highest values in a set X, respectively, and σ denotes the standard deviation. It intuitively measures how much the maximum value in the set X "stands out" from the rest. The algorithm measures the eccentricity of the set of matching scores (between the single node i in V_1 and each unmatched node j in V_2) and rejects the match with the highest matching score if the eccentricity (representing the strength of the match) is below a certain threshold. If it is above the threshold, the mapping between i and j is added to the list, and the next iteration starts.

Narayanan and Shmatikov were the first to succeed in de-anonymizing two real, large social networks, using only their topological structures. In the more generic field of privacy protection, there have been several reported successes in matching and/or inferring social network data from different domains [18, 10] and all the large-scale attack methods that have been proposed are mostly heuristic in nature. In the more specific area of the graph matching problem, the majority of algorithms proposed so far are characterized by the same basic mechanism: they start from an initial set of seed nodes given as input and then progressively expand it into a larger set of matched or mapped nodes, trying to identify all of the other ones [8, 9, 12, 14, 17].

Moreover, also the fundamental feasibility of network de-anonymization through graph matching has been investigated. In [13], it has been shown that two social networks, if modeled as Erdös-Rényi random graphs, can be matched perfectly by an attacker with unlimited computational power under rather benign conditions, even without seeds.

More specifically, the authors adopted a model which assumes that the two observed networks $G_{1,2}$ are partial manifestations of a true underlying network $G = (V, E)$. This ground-truth network graph G was modeled as an Erdös-Rényi random graph $G(n, p)$ with n nodes and where an edge exists between every pair of nodes with identical probability p, independently of all the other edges. The two observed graphs $G_{1,2} = \left(V, E_{1,2}\right)$ are generated by sampling the edge set E with probability s: in other words, each edge $e \in E$ is also in any of the two edge sets $E_{1,2}$ with probability s, independently of each other and

of everything else. As a result, the sampled graphs $G_{1,2}$ are themselves Erdös-Rényi random graphs $G(n, ps)$.

In this work, a very simple and intuitive error function was also defined, able to measure to what extent the graph structures of $G_{1,2}$ resemble each other, given a certain mapping between the two graphs. The authors managed to prove that if the sampling probability s is beyond some threshold, as n grows large, a perfect mapping between the graphs is the one that minimizes the error function. They did not deal with algorithmic aspects or the computational complexity of evaluating the error of all the possible mappings; instead, they showed that, under the above-mentioned hypotheses and conditions, de-anonymization is feasible.

It's rather clear from this whole body of work, that node anonymization can be overcome and therefore cannot guarantee privacy protection.

Another valuable contribution was given in [19] by Yartseva and Grossglauser who proposed a very simple graph matching algorithm named Percolation Graph Matching algorithm (PGM), based on bootstrap percolation [6] and with a single tuning parameter. Basically, any pair of nodes $(i \in G_1, j \in G_2)$ that have at least r neighbors already matched to each other, are added as a valid match to the mapping, in this way the algorithm can propagate.

Still in [19] the authors investigated the performance of the PGM algorithm in function of the size of the seed set given as input. In [12] Narayanan and Shmatikov already noticed the presence of a sharp phase transition in the seed set size: below a certain threshold their algorithm failed almost completely, whereas mainly succeeded when the seed set size was above the threshold. In [19] Yartseva and Grossglauser, exclusively considering sampled Erdös-Rényi graphs as input ones, formally proved the presence of the phase transition and also determined the corresponding critical value for the seed set size.

Lastly, significant research has been carried out to tackle the graph matching problem when the considered networks cannot be modeled as Erdös-Rényi random graphs. Specifically, in [3] the authors have addressed the problem when accounting for scale-free relationships between users, proposing a novel degree-driven graph matching algorithm (DDM) designed to perform successfully when applied to graphs with power-law node degree. In [4] the same authors have investigated how to mitigate the matching errors occurring when the PGM algorithm from [19] is directly applied to networks characterized by dense clusters. To do so, they have developed an improved matching algorithm still based on bootstrap percolation.

This thesis addresses an experimental investigation into the effectiveness and practical problems of applying the PGM algorithm to stochastic block model graphs. The work is organized as follows. Firstly, a brief description of the PGM algorithm, its deferred matching variant and how they work is given. Then, a definition of the model used in this study is provided. Lastly, the performance of the PGM algorithm over stochastic block model graphs is investigated, together with its deferred matching variant and a personal one based on a degree similarity metric. In the Appendix is given the source code of the program developed to test the performance of the different variants.

1. Percolation Graph Matching algorithm

Here is given a description of how the PGM algorithm works, since the investigation of its performance when applied to Block Model graphs is the basis of this study.

Two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are given with the two sets of vertices V_1 and V_2 that are assumed to be equal, $V_1 = V_2 = V$. However, the equivalence between the nodes of the two graphs is assumed to be hidden since only the structure of the two graphs is available to the PGM algorithm: in other terms, the algorithm sees unlabeled versions of $G_{1,2}$. The edge sets E_1 and E_2 are in general different with some correlation between them: intuitively, this means that, if an edge (x, y) exists in E_1 between nodes $x, y \in V_1$, it is likely that an edge also exists in E_2 between the corresponding nodes in V_2 (see Figure 1).

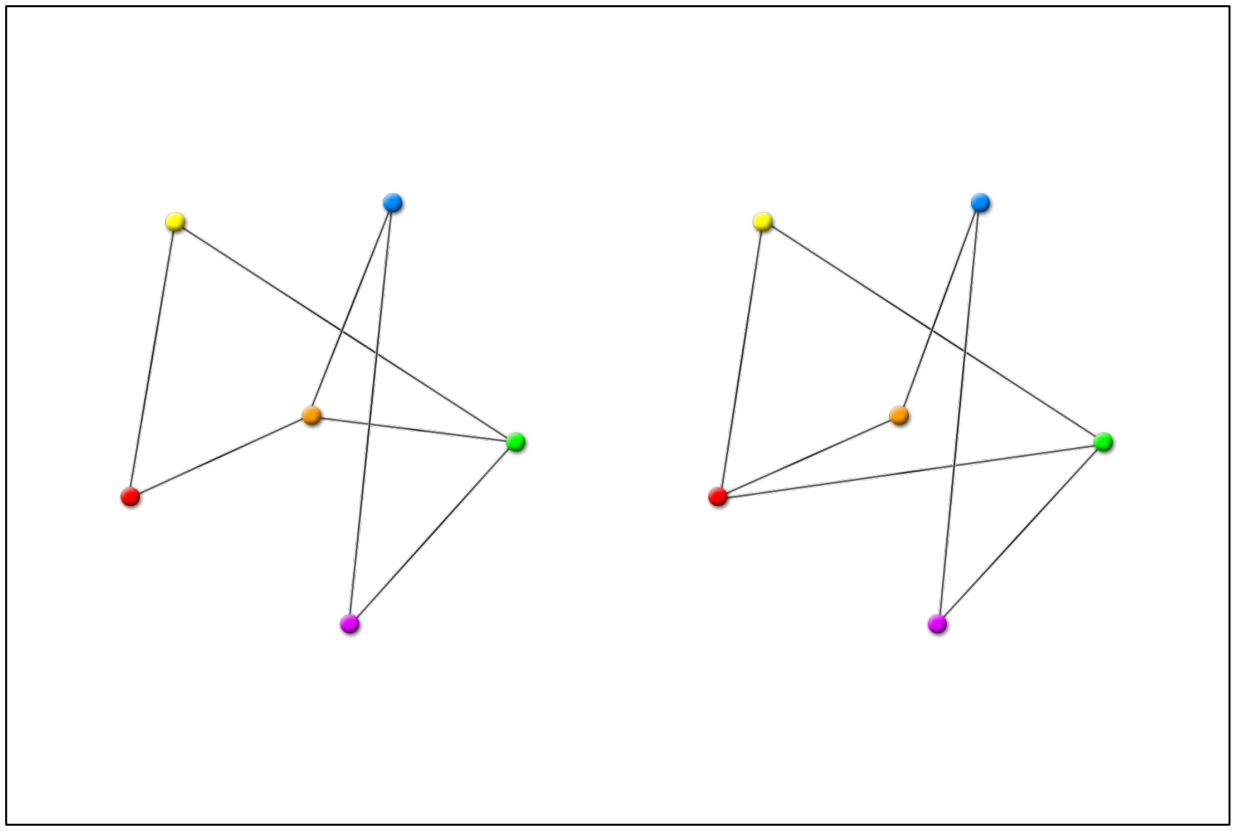


Figure 1: Typical example of two graphs to be given as input to the PGM algorithm, highlighting the correlation between the corresponding edge sets.

The algorithm returns a map as output, i.e., a set of ordered pairs $A \subset V_1 \times V_2$ such that each node belonging to V_1 or V_2 appears at most in one pair. The map is obtained starting from a known seed set A_0 of mapped pairs given as input, and then the algorithm iteratively expands it by finding new matches, i.e., by identifying new suitable pairs (i, j) in the "vicinity" of the already matched ones to be added to the map. A pair (i, j) is added as a match if there are at least r already mapped pairs that are neighbors of (i, j), where two pairs (i', j') and (i'', j'') are defined to be neighbors iff $(i', i'') \in E_1$ and $(j', j'') \in E_2$. The process continues until no more suitable pairs can be found. The size of the final map, that is its cardinality $|A|$, isn't necessarily equal to $|V|$. Elements of A in the form (i, i) are good or correct matches, whereas all the others are of course wrong matches. The error rate is given by $|(i, j) : i \neq j, (i, j) \in A| \,/\, |A|$.

The algorithm can also be described in the following way, which is easier to implement and emphasizes the iterative nature of the process. Any pair (i,j) is added as a match if it has at least r neighboring pairs that are already mapped as already said. Equivalently, a count of marks $M_{i,j}$ can be associated to every pair $(i \in V_1, j \in V_2)$. At each iteration, the algorithm selects a pair among those that have already been mapped but haven't been visited yet. This pair adds one mark to all its neighboring pairs. As soon as any pair gets at least r marks, it is added to the map. If there are several candidate pairs that have reached r marks but are conflicting, the algorithm selects one of them uniformly at random and adds it to the map. Two pairs are defined as conflicting when they are in the form (i',j') and (i',j''), or (i',j') and (i'',j') respectively. The process iterates until all matched pairs have been visited.

In Alg. 1 is given the formal description of the PGM algorithm, where:
- $A(t)$ is the set of pairs that have been mapped until time t;
- $Z(t) \subset A(t)$ is the set of mapped pairs that have been *visited* until t;
- $N_1(i)$ is the set of nodes that are neighbors of $i \in V_1$;
- $N_2(j)$ is the set of nodes that are neighbors of $j \in V_2$;
- $N_1(i) \times N_2(j)$ is therefore the set of all the pairs that are neighbors of (i,j).

Algorithm 1: The PGM algorithm

$A(0) = A_0$, $Z_0 = \emptyset$, $t = 0$
while $A(t) \setminus Z(t) \neq \emptyset$ **do**
  $t = t + 1$
  Randomly select a pair $(i_t, j_t) \in A(t-1) \setminus Z(t-1)$ and add one mark to all pairs $(i',j') \in N_1(i_t) \times N_2(j_t)$.
  Let $B(t)$ be the set of all pairs (i',j') whose mark counter $M_{i',j'}$ has reached threshold r at time t.
  while $B(t) \neq \emptyset$ **do**
    Select a pair (i',j') from $B(t)$ uniformly at random and insert it in the map $A(t)$.
    All other conflicting candidates (i'',j') and (i',j'') are permanently removed both from $B(t)$ and from consideration in future iterations.
    Remove the pair (i',j') from $B(t)$.
  Let $\Delta A(t)$ be the set of pairs which have been added to the map $A(t)$ at time t. Then $A(t) = A(t-1) \cup \Delta A(t)$ and $Z(t) = Z(t-1) \cup \{(i_t, j_t)\}$. Note that $|A(t)| \geq |Z(t)| = t$.
return $T = t$, $Z(T) = A(T)$, where $A(T)$ is the final map and $|A(T)| = T$.

In **Figure 2** are shown the first three iterations of the algorithm applied to a very simple couple of input graphs, as an example.

It is clear that the parameter r plays a critical role, since it represents the amount of evidence, in favor of a pair of nodes, needed to add the pair as a permanent match. If r is chosen too low, the probability of a false match increases. If it is chosen too high, the algorithm may stop propagating early for lack of candidates with enough marks.

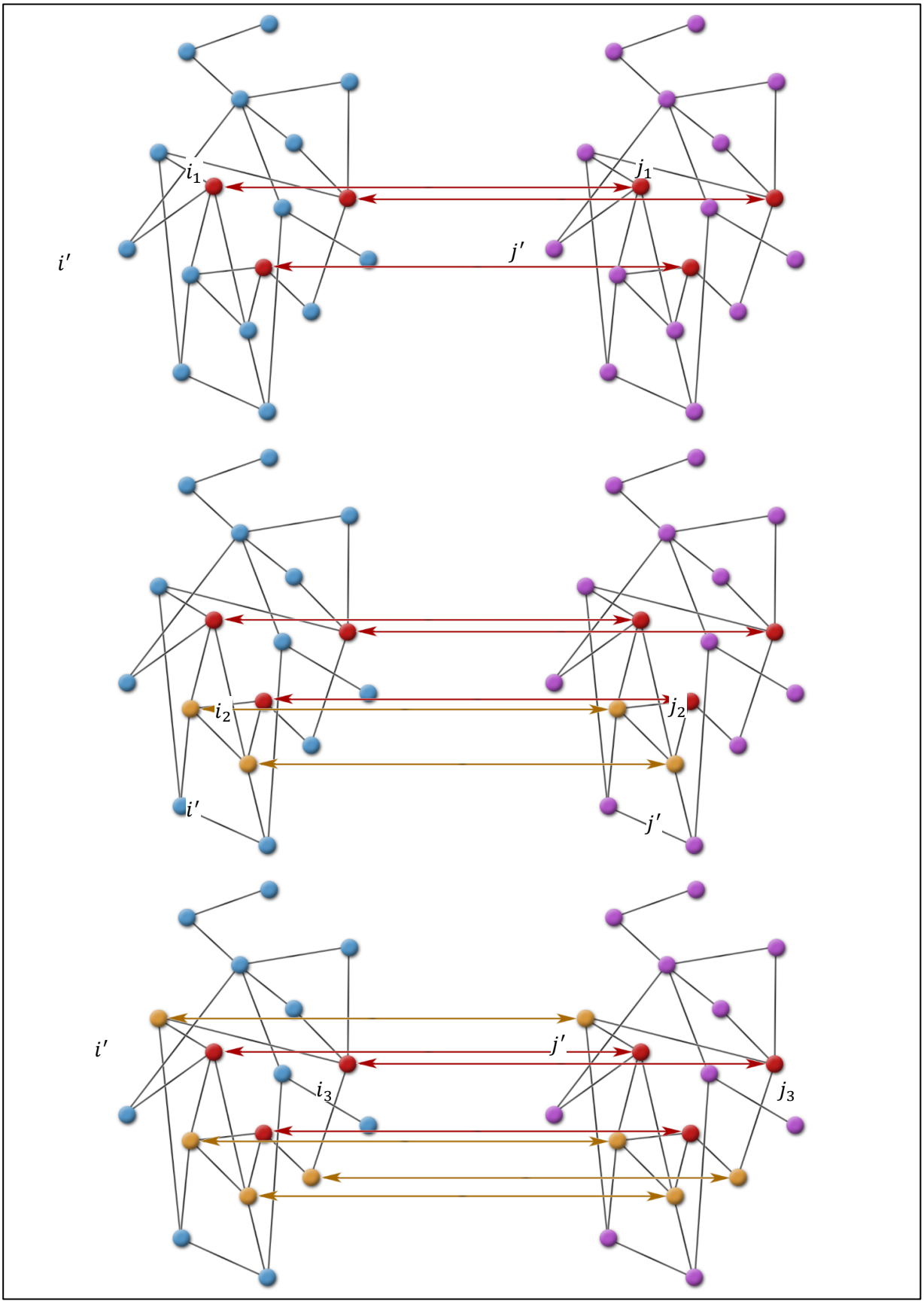


Figure 2: Example of application of the PGM algorithm, with $r = 2$. Red nodes are the seeds, orange nodes are the set of matched pairs after the first three iterations.

2. Deferred matching variant

The basic PGM algorithm as defined above is not optimal in most cases, since it greedily matches any candidate pair as soon as its number of marks gets greater or equal to r. If $A(t) \setminus Z(t) \neq \emptyset$, it means that there are mapped pairs that haven't been visited yet and can add more marks, possibly improving the $M_{i,j}$ counters and avoid false matches.

The variant described here is a simple way to fix this problem, allowing the algorithm to be more conservative. Whenever $A(t) \setminus Z(t) \neq \emptyset$, the algorithm simply keeps visiting new matched pairs and attributing new credits to candidate pairs, without forming any new matches. Once there are no more matched pairs to visit, i.e., $A(t) \setminus Z(t) = \emptyset$, exactly one pair is selected, the one with the maximum $M_{i,j}$ of all candidates. If the mark counter is above the threshold r, the pair is added to the map $A(t)$ as a new match and the algorithm can keep propagating; if it is not, the algorithm stops.

This variant is clearly conservative about forming new matches: it first exploits all the available evidence that can be collected by visiting all the unused matched pairs before irreversibly choose a candidate pair as a valid match. Moreover, the role played by the parameter r is less important: if chosen too low, the variant ensures that only the best candidate pairs, according to all the possible evidence that can be gathered at each iteration, are matched.

In **Alg. 2** is given the formal description of the variant.

Algorithm 2: The Deferred Matching Variant of PGM algorithm

1: $A(0) = A_0, Z_0 = \emptyset, t = 0$
2: **while** $A(t) \setminus Z(t) \neq \emptyset$ **do**
3: **while** $A(t) \setminus Z(t) \neq \emptyset$ **do**
4: $t = t + 1$
5: Randomly select a pair $(i_t, j_t) \in A(t-1) \setminus Z(t-1)$ and add one mark to all pairs $(i', j') \in N_1(i_t) \times N_2(j_t)$.
6: $A(t) = A(t-1)$ and $Z(t) = Z(t-1) \cup \{(i_t, j_t)\}$.
7: Let $B(t)$ be the set of all pairs (i', j') whose mark counter $M_{i',j'}$ is maximal and at least r at time t.
8: **if** $B(t) \neq \emptyset$ **then**
9: Select a pair (i', j') from $B(t)$ uniformly at random and insert it in the map $A(t)$.
10: All other conflicting candidates (i'', j') and (i', j'') are permanently removed from consideration in future iterations.
11: $A(t) = A(t) \cup \{(i', j')\}$.
12: **return** $T = t, Z(T) = A(T)$, where $A(T)$ is the final map and $|A(T)| = T$.

The simulation results obtained in [19] by the authors clearly show that this variant exhibits similar threshold behavior in the seed set size A_0 as the basic version, but decreases the error rate in the considered scenarios.

3. Model

The model used for this study is defined in the following. The ground-truth network graph $G = (V, E)$ is assumed to be a stochastic block model one, characterized by the following parameters:

- the number n of vertices;
- a partition of the vertex set V into disjoint subsets $C_1, \dots, C_m$ called *communities*;
- a symmetric $m \times m$ matrix $P = [p_{ij}]$, where p_{ij} is the probability that any two vertices $u \in C_i$ and $v \in C_j$ are connected by an edge and $p_{ij} \in [0,1]$.

Denoting by D_{ij} the number of edges that connect a vertex belonging to C_i to any vertex belonging to C_j and by D_i the overall degree of a vertex belonging to C_i:

$$\Pr(D_{ij} = k) = \binom{|C_j|}{k} p_{ij}^k (1 - p_{ij})^{|C_j| - k} \text{ for } k = 0,1, \dots, |C_j| \text{ and } i \neq j$$

$$D_{ij} \sim \text{Bi}(|C_j|, p_{ij}), \qquad E[D_{ij}] = |C_j| p_{ij}, \qquad \text{for } i \neq j$$

$$\Pr(D_{ii} = k) = \binom{|C_i| - 1}{k} p_{ii}^k (1 - p)^{|C_i| - 1 - k} \text{ for } k = 0,1, \dots, |C_i| - 1$$

$$D_{ii} \sim \text{Bi}((|C_i| - 1), p_{ii}), \qquad E[D_{ii}] = (|C_i| - 1) p_{ii} \simeq |C_i| p_{ii}$$

$$D_i = \sum_{j=1}^{m} D_{ij}$$

$$E[D_i] = E\left[\sum_{j=1}^{m} D_{ij}\right] = \sum_{j=1}^{m} E[D_{ij}] \simeq \sum_{j=1}^{m} |C_j| p_{ij}$$

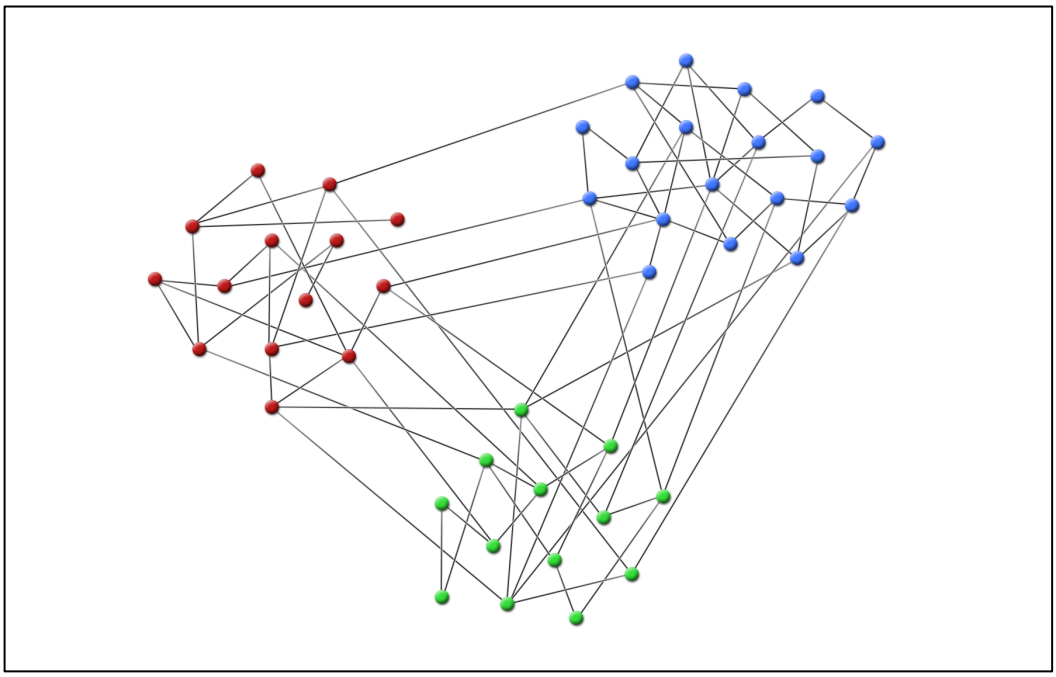


Figure 3: Typical example of a stochastic block model random graph with three communities. Nodes with the same color belong to the same community.

The model is called *strongly assortative* if $p_{i,i} > p_{j,k}$ whenever $j \neq k$, i.e., when all diagonal entries of P are greater than all off-diagonal ones. The model is called *weakly assortative* if $p_{i,i} > p_{i,j}$ whenever $i \neq j$, i.e., each diagonal entry is only required to be greater than the other entries of its own row or column. Similarly, the model is called *strongly dissortative* if $p_{i,i} < p_{j,k}$ whenever $j \neq k$, and *weakly dissortative* if $p_{i,i} > p_{i,j}$ whenever $i \neq j$.

If the probability matrix P is a constant, i.e., $p_{i,j} = p$ for all i, j, then the model becomes equivalent to the Erdös-Rényi one $G(n, p)$. In this degenerate case, the partition into communities becomes irrelevant but shows the close relationship between the two models.

4. Performance of PGM over Erdös-Rényi random graphs

In this chapter, my implemented version of the PGM algorithm is tested over Erdös-Rényi random graphs $G(n,p)$, in order to check if the obtained simulation results comply with those obtained in [19].

The metric used to evaluate the performance of the algorithm is the size of the final map, i.e., the total number of mapped nodes, which says how far the algorithm manages to propagate.

Both the basic version and the deferred matching version of the algorithm are run, with $r = 2$. For the input graphs, the edge sampling model is used: therefore, starting from a ground-truth network graph $G(n,p)$, each edge appears in the observed network with probability s.

Figures 4, 5 show the dependence of the metric on the size of the seed set A_0 and each of them displays three curves for different values of the sampling parameter s. The generator graph $G(n,p)$ has $n = 10000$ and $p = 20/n$ as parameters. All the results are averaged over 15 realizations.

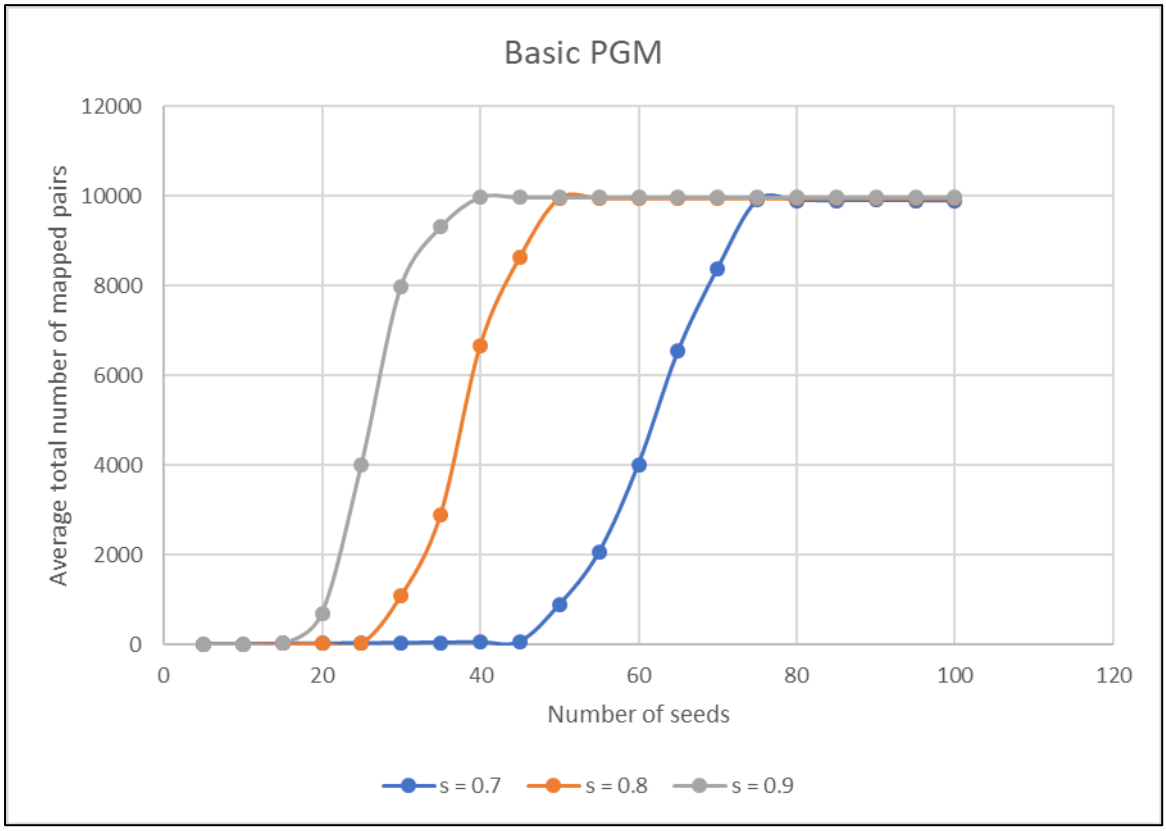


Figure 4: Average total number of mapped nodes vs number of seeds for the basic version of PGM over $G(n,p)$ with $n = 10000$ and $p = 20/n$.

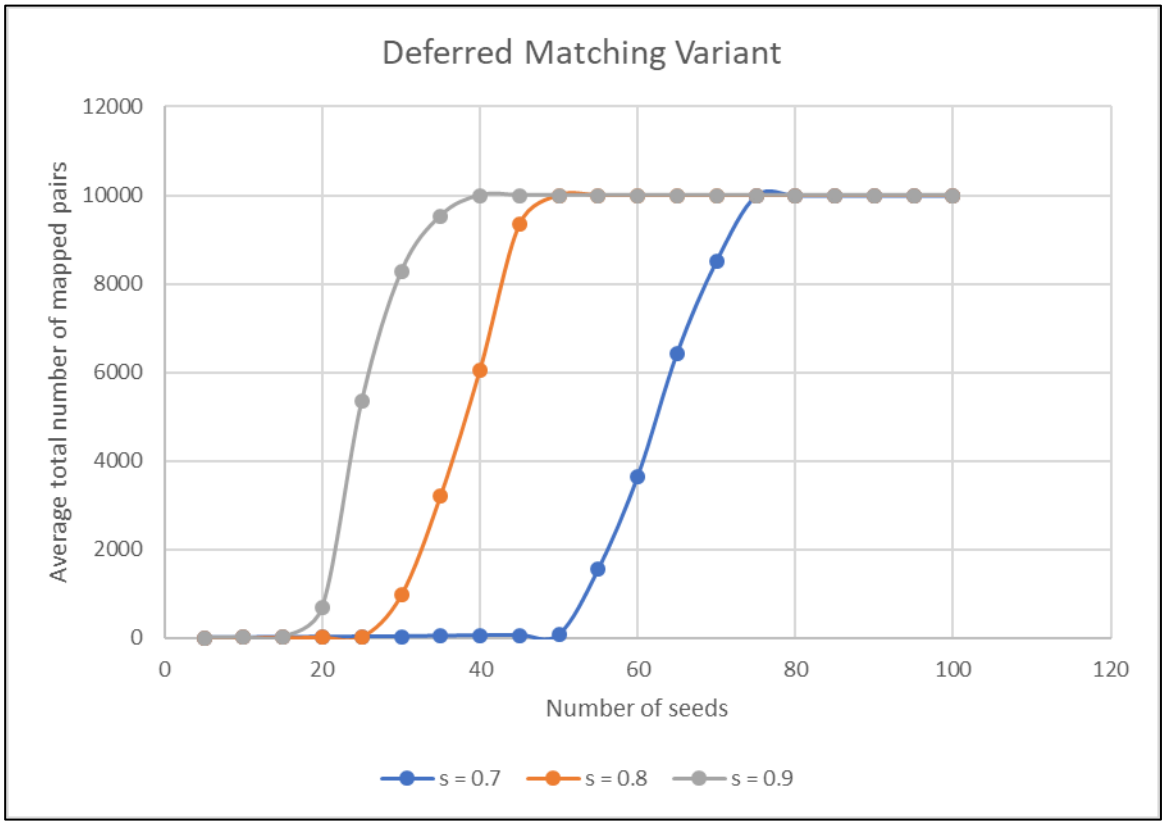


Figure 5: Average total number of mapped nodes vs number of seeds for the basic version of PGM over $G(n,p)$ with $n = 10000$ and $p = 20/n$.

The obtained results show the same behavior as the one described in [19] and are qualitatively similar for both the variants of the algorithm. The graphs in the figures exhibit the presence of the phase transition in correspondence with the critical seed set size. As the sampling parameter s decreases, the critical value for the seed set size gets larger. Intuitively, starting from the seeds, the larger the size of the seed set, the more neighboring pairs there are and can be visited and the farther the algorithm is likely to propagate. Moreover, the higher the sampling parameter, the more likely that a certain edge exists in one of the two input graphs, given that it exists in the other: considering the propagation mechanism of the PGM algorithm, this also means that fewer seeds, thus fewer initial neighboring pairs to be visited are needed.

5. Performance of PGM over stochastic block model graphs

In this chapter, my implemented version of the PGM algorithm is tested over stochastic block model graphs, to investigate the performance of the algorithm on this kind of networks.
This can be considered an interesting study, since this type of random graphs is useful to model real networks with a dominant community structure.

The metrics used to evaluate the performance of the algorithm are the size of the final map and the error rate, i.e., the fraction of wrong pairs in the map.

Both the basic version and the deferred matching version of the algorithm are run, with $r = 2$. For the input graphs, the edge sampling model is used as in the previous chapter: starting from a ground-truth stochastic block model network graph, each edge appears in the observed network with probability $s = 0.7$.

I limited the study to a ground-truth graph that always consists of three communities for simplicity, with a total number of nodes $n = 10000$. Each of the following figures shows the obtained simulation results for different edge probability matrices $P = [p_{ij}]$ and for different sizes of the community sets. Within each graph, different curves are displayed, one for each different way the seed set was built. In particular, the probability that a seed to be chosen belongs to community set i is denoted by σ_i, so that $\boldsymbol{\sigma}$ is the vector of such probabilities. Given that the seed to be chosen belongs to a certain community i, the seed is then picked uniformly at random among its vertices.

Figures 6-10 show the dependence of the size of the final map on the size of the seed set A_0. The curves show the same qualitative behavior as the one described in the previous chapter, which is similar for both the variants of the algorithm: they are characterized by a phase transition in correspondence with the critical seed set size. This critical size clearly depends on the particular $\boldsymbol{\sigma}$ adopted to build the seed set. The results show that, as expected, the more seeds belong to a community i in which the average degree of the nodes $E[D_i]$ is high, the easier it is for the algorithm to propagate, thus fewer seeds are needed. This is always true, independently of whether the ground-truth block model is assortative or dissortative, even though the curves have a tendency to show phase transitions that are closer to each other when the model is dissortative. The relations between the sizes of the different community sets don't seem to have an impact on the curve behavior at all. Finally, in each figure are also included two curves corresponding to Erdös-Rényi random graphs $G(n, p)$ with the parameter p set equal to the maximum and the minimum p_{ij} values respectively. As it could be intuitively expected, the curve relative to the maximum value represents an upper bound for all the other curves, whereas the one relative to the minimum value represents a lower bound.

Figures 11-13 show the behavior of the error rate as a function of the size of the seed set. In this case, it's difficult to make any kind of considerations depending on the changes in the values of the parameters, but a general behavior is evident: after an initial peak, all the curves tend to a lower asymptotic value as the number of seeds exceeds the corresponding critical seed set size and then increases. An important observation is the fact that the deferred matching variant of the PGM algorithm performs considerably better than the basic version with regard to the error rate, whereas the curves relative to the first metric don't show any improvement between the two versions.

All the obtained results and curves are averaged over 15 realizations.

	Number of nodes
Community 0	(1/3)n
Community 1	(1/3)n
Community 2	(1/3)n
Overall	n = 10000

p_{ij}	Community 0	Community 1	Community 2
Community 0	10/(n/3)	2/(n/3)	2/(n/3)
Community 1	2/(n/3)	30/(n/3)	2/(n/3)
Community 2	2/(n/3)	2/(n/3)	20/(n/3)

Sampling parameter s = 0.7

PGM threshold r = 2

C0 Avg. Degree $E[D_0]$ = 14

C1 Avg. Degree $E[D_1]$ = 34

C2 Avg. Degree $E[D_2]$ = 24

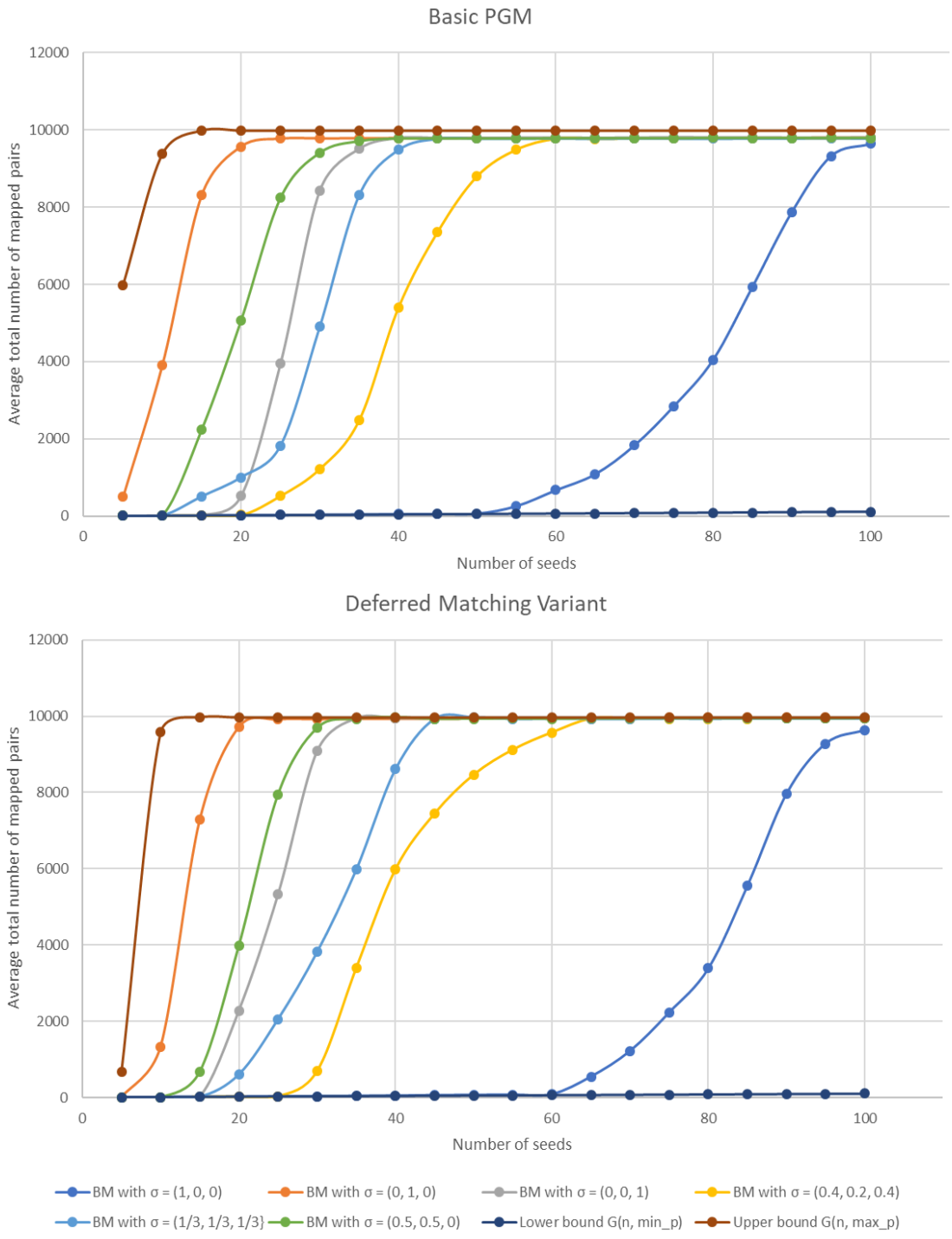


Figure 6

	Number of nodes
Community 0	(1/3)n
Community 1	(1/3)n
Community 2	(1/3)n
Overall	n = 10000

p_{ij}	Community 0	Community 1	Community 2
Community 0	2/(n/3)	10/(n/3)	5/(n/3)
Community 1	10/(n/3)	2/(n/3)	25/(n/3)
Community 2	5/(n/3)	25/(n/3)	2/(n/3)

Sampling parameter s = 0.7

PGM threshold r = 2

C0 Avg. Degree $E[D_0]$ = 17

C1 Avg. Degree $E[D_1]$ = 37

C2 Avg. Degree $E[D_2]$ = 32

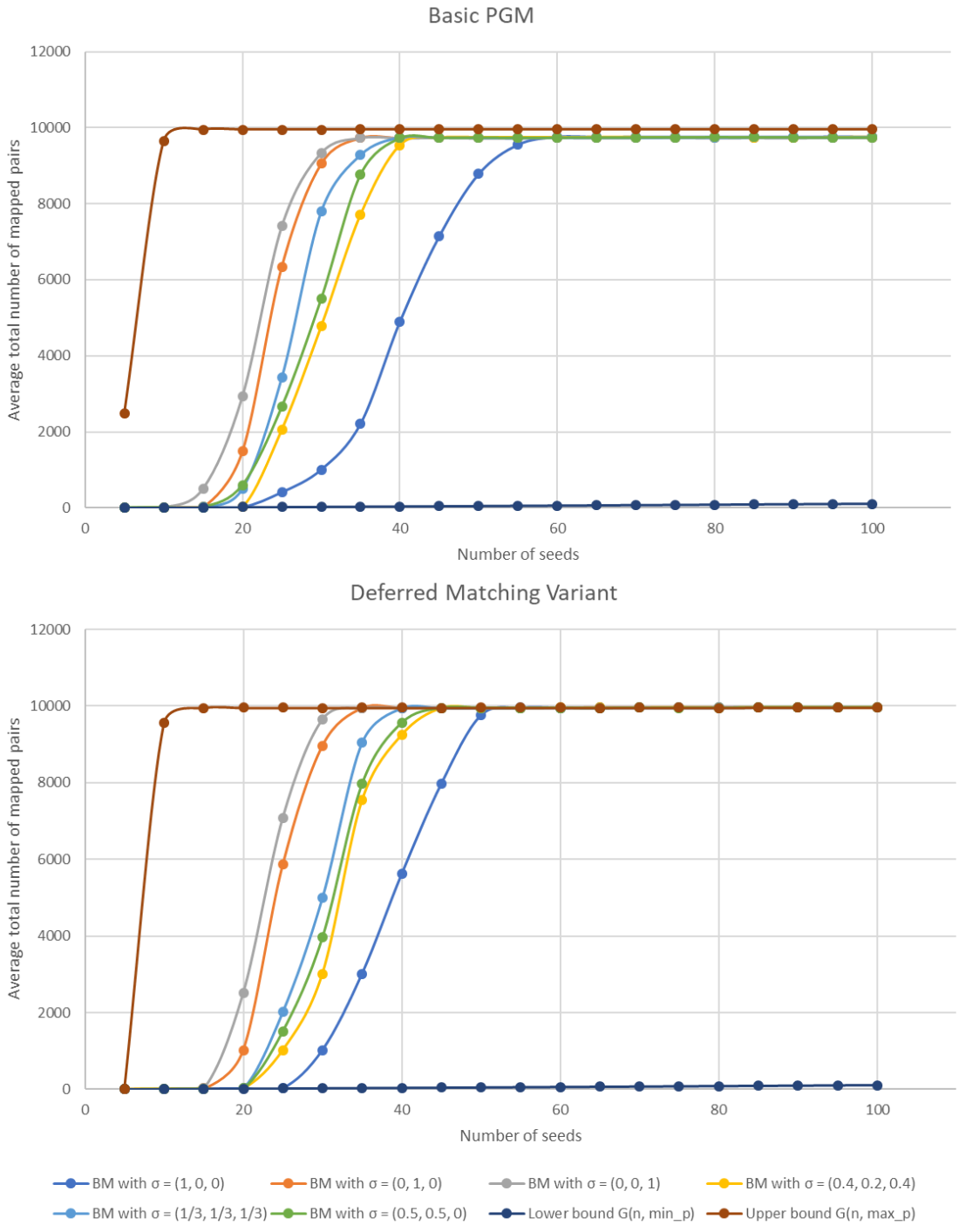


Figure 7

	Number of nodes
Community 0	(0.4)n
Community 1	(0.5)n
Community 2	(0.1)n
Overall	n = 10000

p_{ij}	Community 0	Community 1	Community 2
Community 0	2/(n/3)	10/(n/3)	5/(n/3)
Community 1	10/(n/3)	2/(n/3)	25/(n/3)
Community 2	5/(n/3)	25/(n/3)	2/(n/3)

Sampling parameter s = 0.7

PGM threshold r = 2

C0 Avg. Degree $E[D_0]$ = 18.9

C1 Avg. Degree $E[D_1]$ = 22.5

C2 Avg. Degree $E[D_2]$ = 44.1

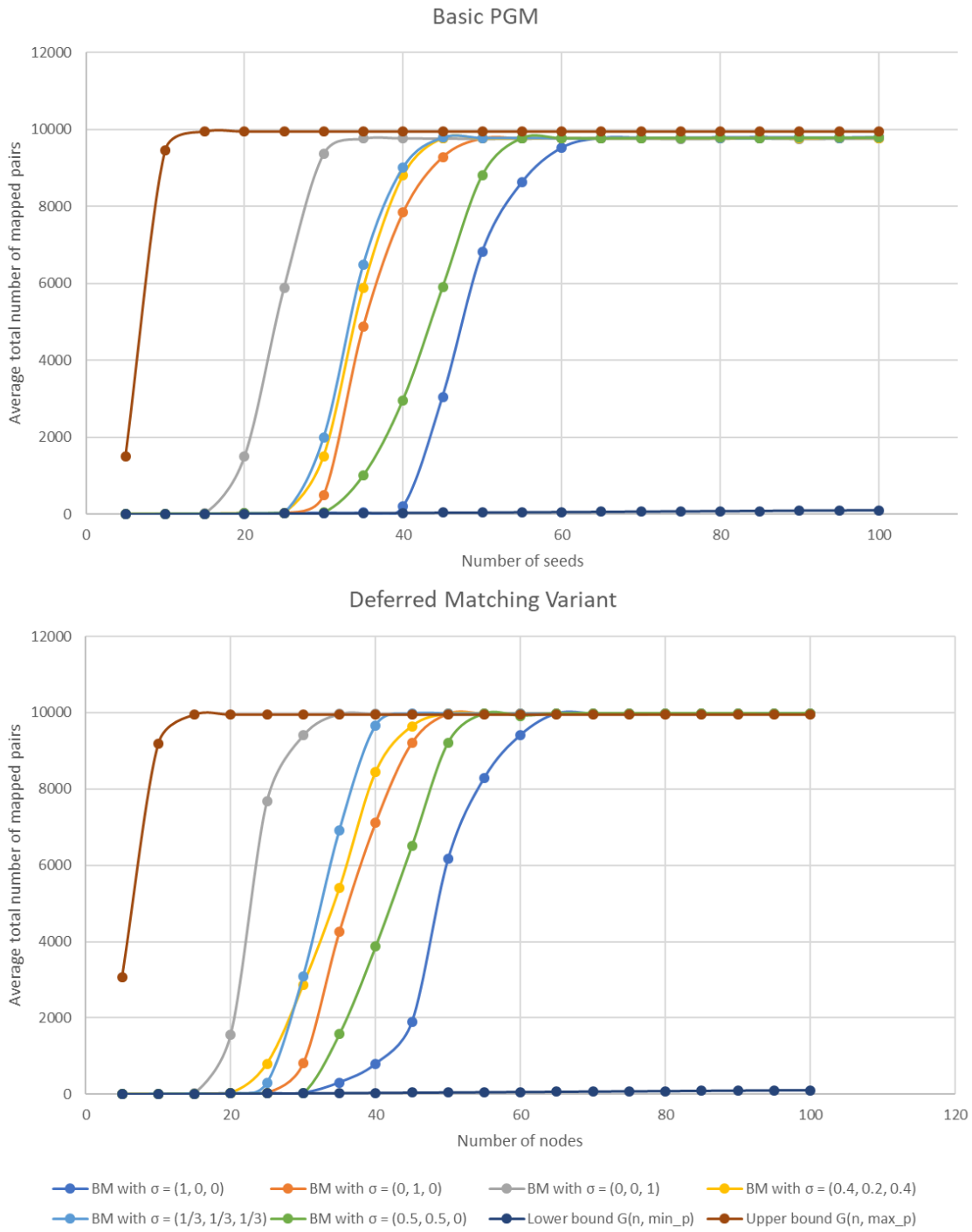


Figure 8

	Number of nodes
Community 0	(0.2)n
Community 1	(0.1)n
Community 2	(0.7)n
Overall	n = 10000

p_{ij}	Community 0	Community 1	Community 2
Community 0	2/(n/3)	10/(n/3)	8.7/(n/3)
Community 1	10/(n/3)	2/(n/3)	5.9/(n/3)
Community 2	8.7/(n/3)	5.9/(n/3)	17.7/(n/3)

Sampling parameter s = 0.7

PGM threshold r = 2

C0 Avg. Degree $E[D_0]$ = 22.47

C1 Avg. Degree $E[D_1]$ = 18.99

C2 Avg. Degree $E[D_2]$ = 44.16

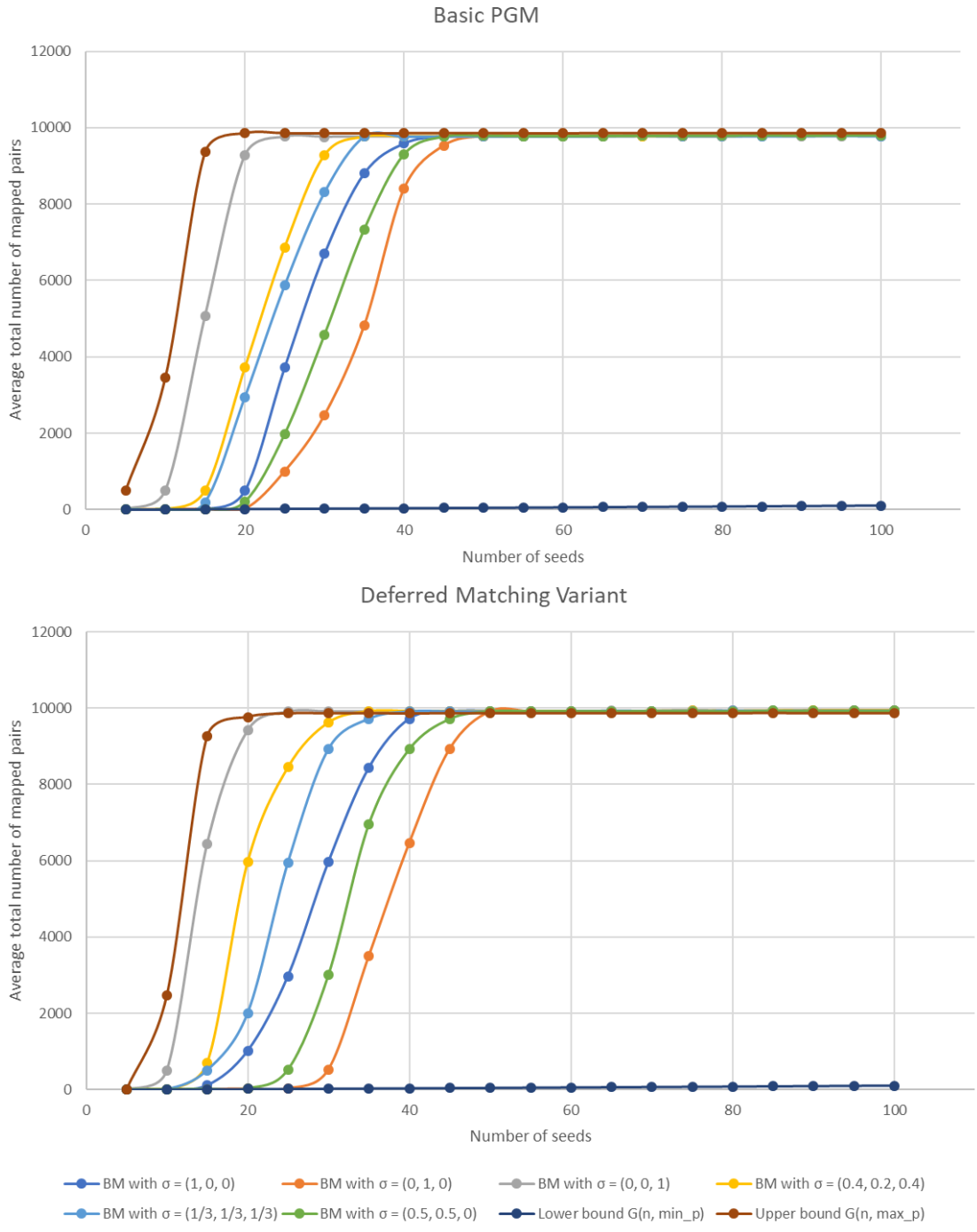


Figure 9

	Number of nodes
Community 0	(0.2)n
Community 1	(0.1)n
Community 2	(0.7)n
Overall	n = 10000

p_{ij}	Community 0	Community 1	Community 2
Community 0	2/(n/3)	10/(n/3)	25/(n/3)
Community 1	10/(n/3)	2/(n/3)	5/(n/3)
Community 2	25/(n/3)	5/(n/3)	2/(n/3)

Sampling parameter s = 0.7

PGM threshold r = 2

C0 Avg. Degree $E[D_0]$ = 56.7

C1 Avg. Degree $E[D_1]$ = 17.1

C2 Avg. Degree $E[D_2]$ = 20.7

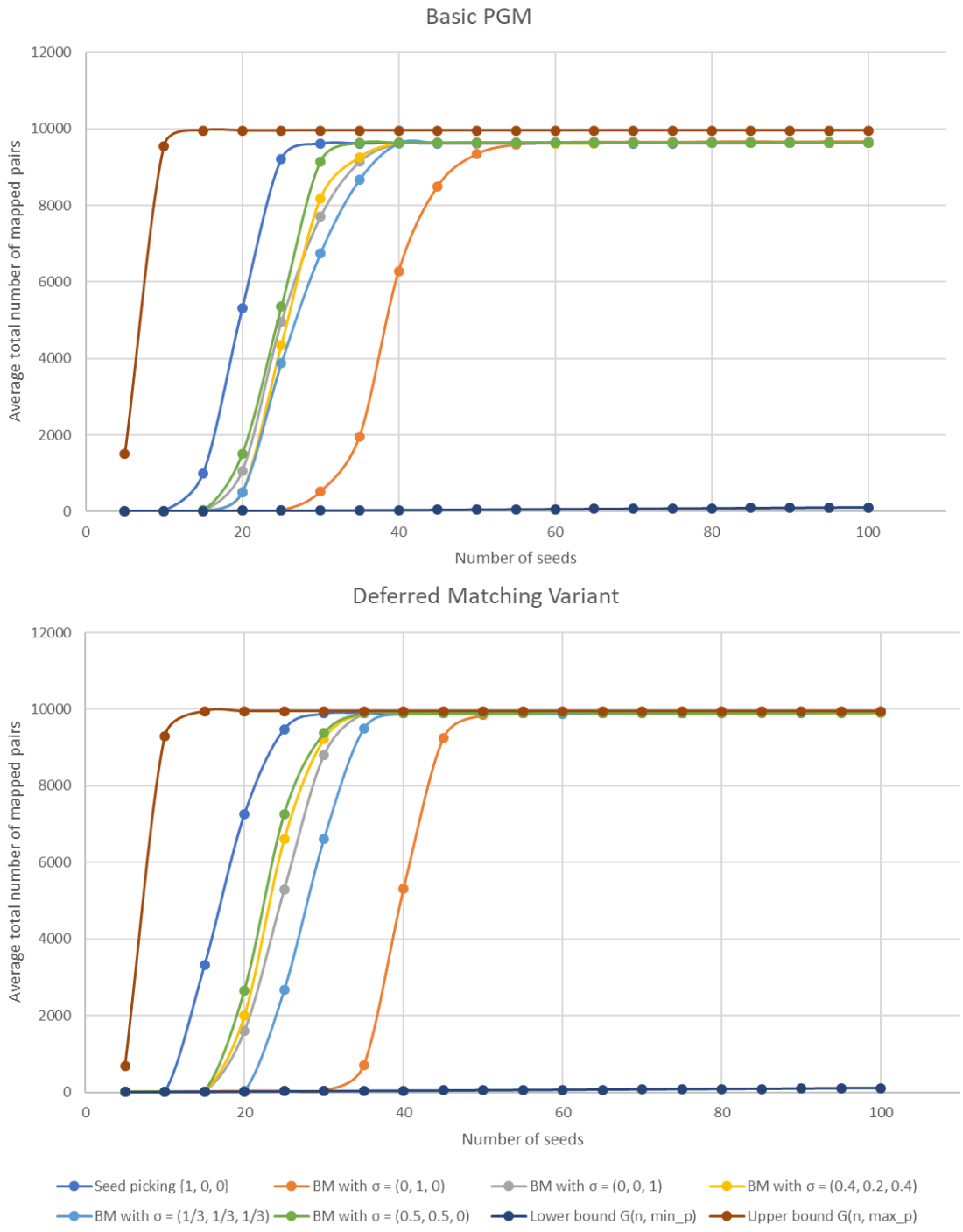


Figure 10

	Number of nodes
Community 0	(1/3)n
Community 1	(1/3)n
Community 2	(1/3)n
Overall	n = 10000

p_{ij}	Community 0	Community 1	Community 2
Community 0	10/(n/3)	2/(n/3)	2/(n/3)
Community 1	2/(n/3)	30/(n/3)	2/(n/3)
Community 2	2/(n/3)	2/(n/3)	20/(n/3)

Sampling parameter s = 0.7

PGM threshold r = 2

C0 Avg. Degree $E[D_0]$ = 14

C1 Avg. Degree $E[D_1]$ = 34

C2 Avg. Degree $E[D_2]$ = 24

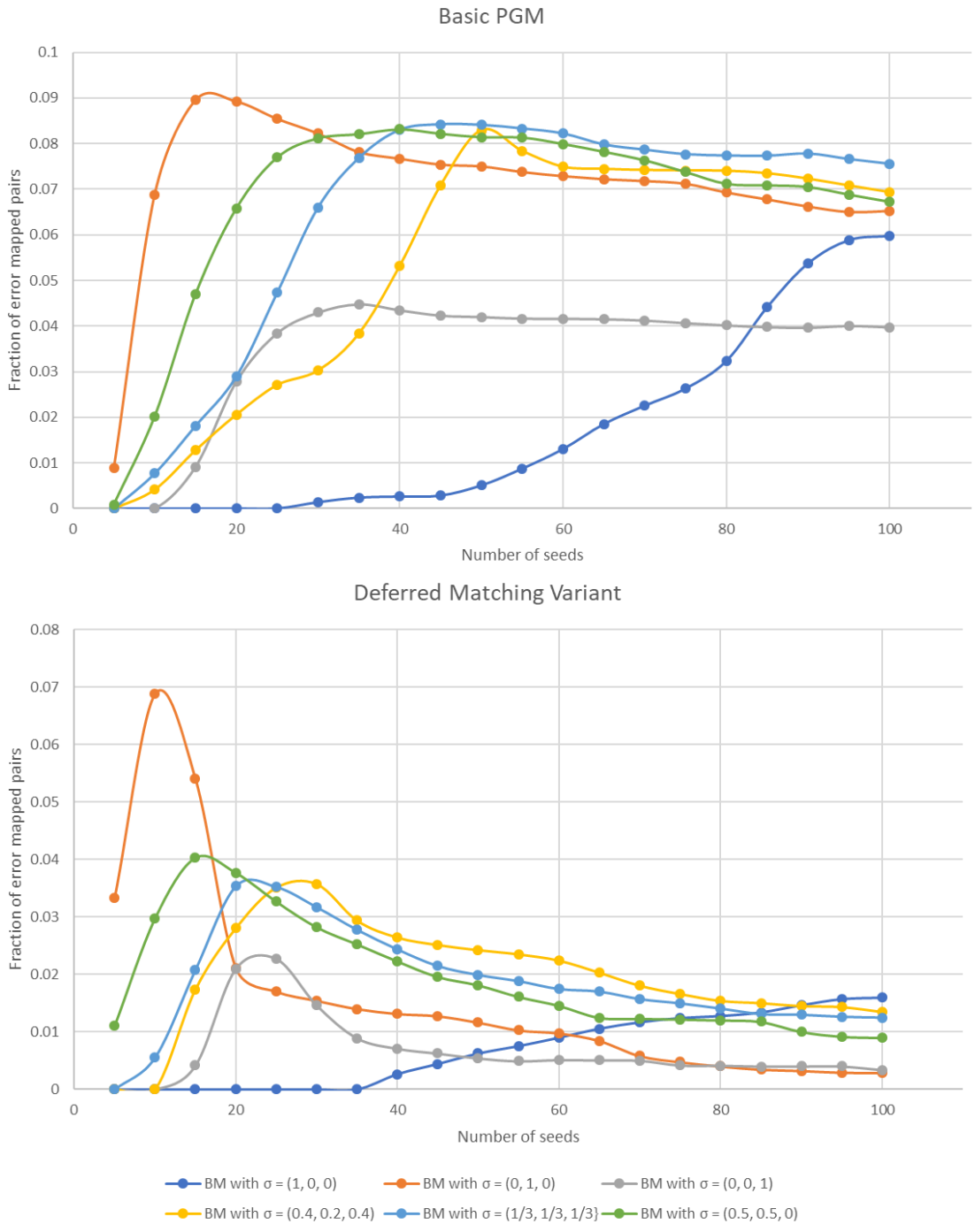


Figure 11

	Number of nodes
Community 0	(1/3)n
Community 1	(1/3)n
Community 2	(1/3)n
Overall	n = 10000

p_{ij}	Community 0	Community 1	Community 2
Community 0	2/(n/3)	10/(n/3)	5/(n/3)
Community 1	10/(n/3)	2/(n/3)	25/(n/3)
Community 2	5/(n/3)	25/(n/3)	2/(n/3)

Sampling parameter s = 0.7

PGM threshold r = 2

C0 Avg. Degree $E[D_0]$ = 17

C1 Avg. Degree $E[D_1]$ = 37

C2 Avg. Degree $E[D_2]$ = 32

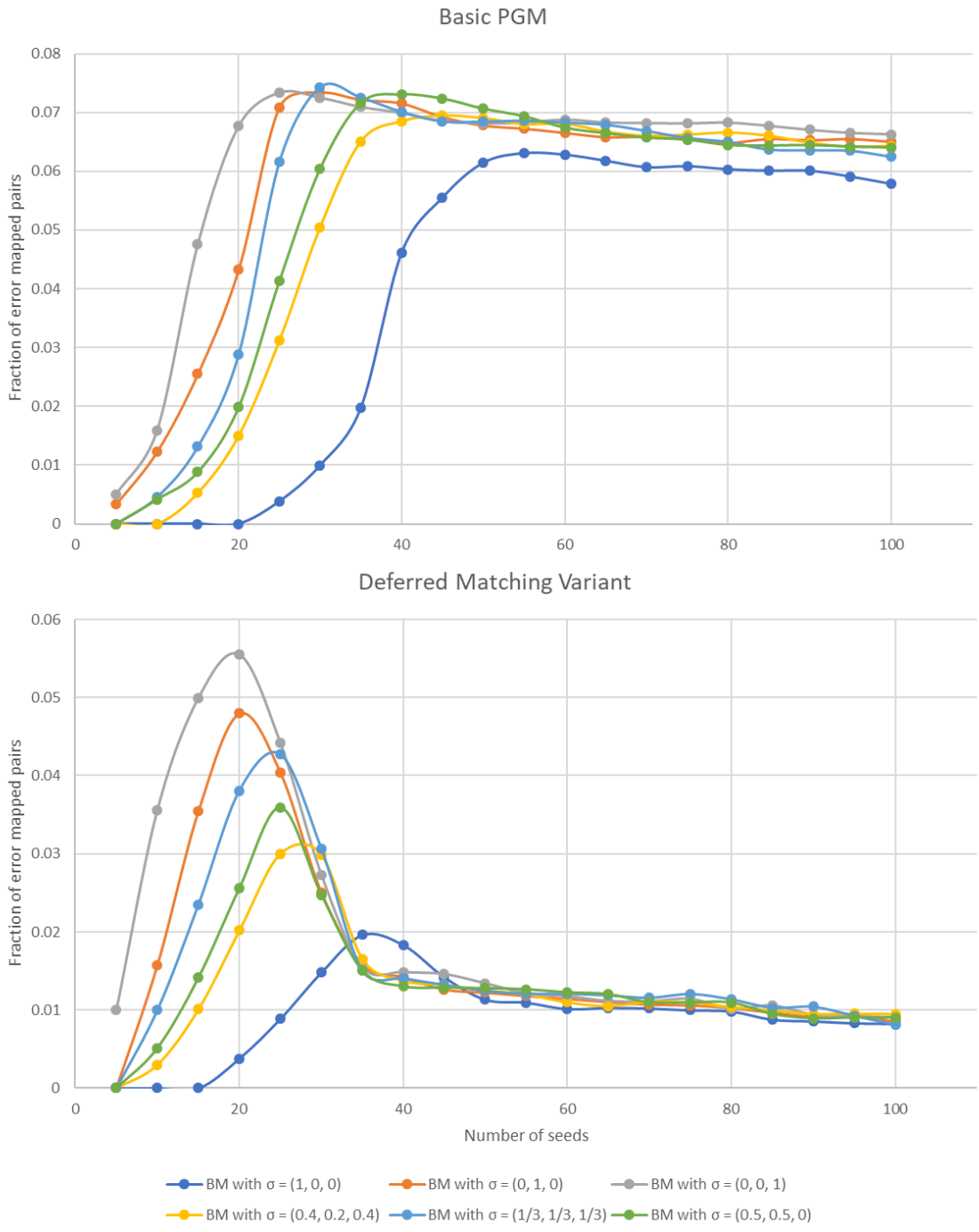


Figure 12

	Number of nodes
Community 0	(0.4)n
Community 1	(0.5)n
Community 2	(0.1)n
Overall	n = 10000

p_{ij}	Community 0	Community 1	Community 2
Community 0	2/(n/3)	10/(n/3)	5/(n/3)
Community 1	10/(n/3)	2/(n/3)	25/(n/3)
Community 2	5/(n/3)	25/(n/3)	2/(n/3)

Sampling parameter s = 0.7

PGM threshold r = 2

C0 Avg. Degree $E[D_0]$ = 18.9

C1 Avg. Degree $E[D_1]$ = 22.5

C2 Avg. Degree $E[D_2]$ = 44.1

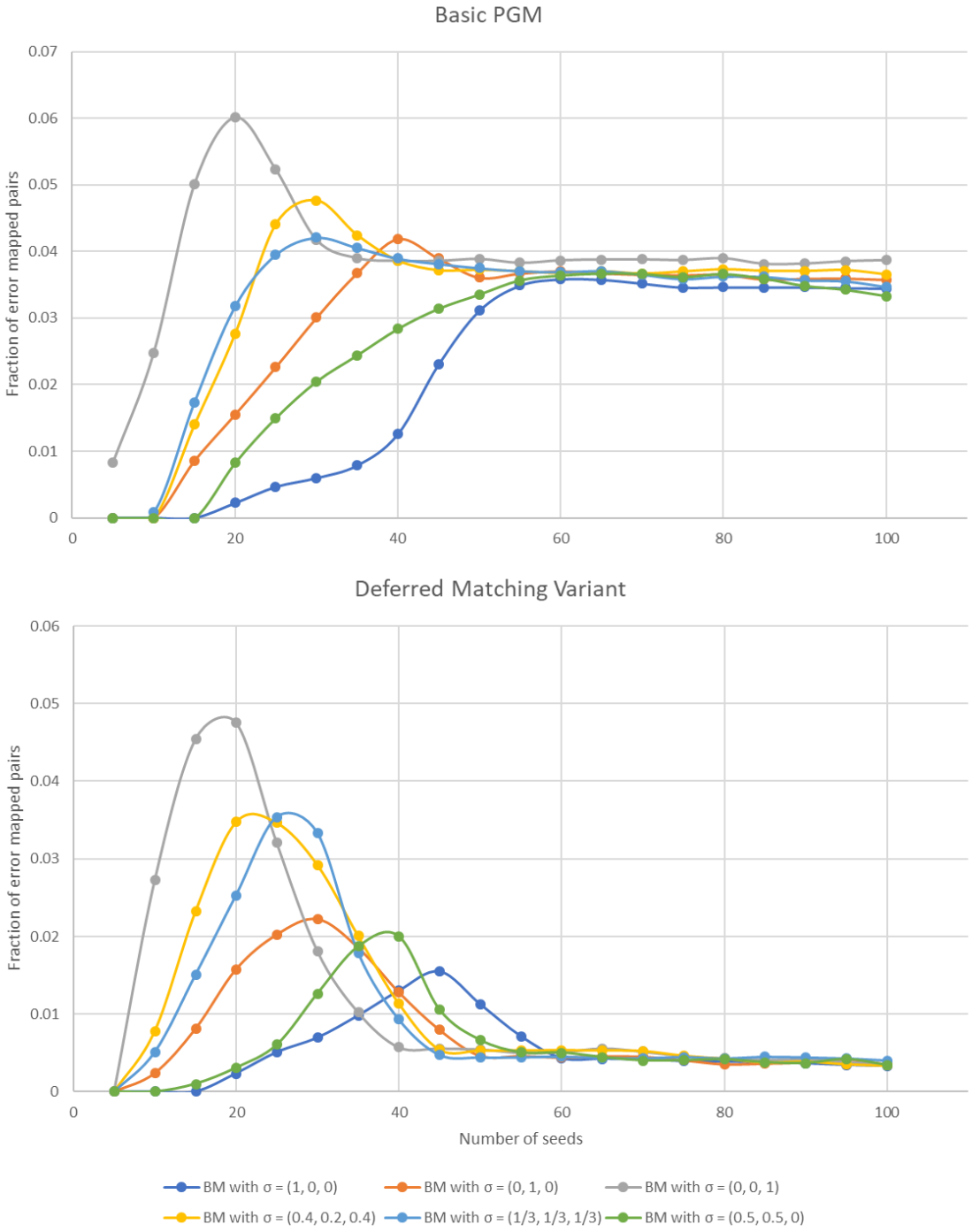


Figure 13

6. Performance of the degree similarity matching variant

As observed in the previous chapter, the capability of the algorithm to propagate doesn't seem to depend on the particular variant used. In fact, it depends only on the topological structure of the graph and the characteristics of the seed set used as input.

A personal variant to the PGM algorithm is now introduced and analyzed, designed to minimize, if possible, the error rate of the final maps built.

According to the stochastic block model, the ground-truth network graph $G = (V, E)$ is characterized by a certain number of communities and an edge probability matrix $P = [p_{ij}]$. In each community, nodes show a peculiar average degree which, in the most general case, is different from that of nodes belonging to other communities, since it depends on the entries of the edge probability matrix P and the sizes of the communities. Hence, the idea of selecting pairs to be added to the map $A(t)$ according to a degree similarity metric designed to exploit this characteristic, instead of selecting those with the maximum mark counter $M_{i,j}$ of all candidates.

Two possible definitions of such a metric were considered, under the hypothesis that the input graphs are sampled from the ground-truth one with the same sampling parameter s and thus have similar graph density values. Firstly, the metric could be simply defined as the absolute value of the difference between the degrees d_i and d_j of the nodes of a pair (i, j):

$$X_{i,j} = |d_i - d_j|$$

Lastly, the metric could also be defined in the following way. Considering a pair (i, j), the arithmetic mean of the degrees of all the nodes belonging to $N_1(i)$ is calculated, being $N_1(i)$ the set of nodes that are neighbors of $i \in V_1$. After that, the same is performed with the degrees of all the nodes belonging to $N_2(j)$, being $N_2(j)$ the set of nodes that are neighbors of $j \in V_2$. The final metric is defined as the absolute value of the difference between the two means.

$$Y_{i,j} = \left| \frac{\sum_{i' \in N_1(i)} d_{i'}}{d_i} - \frac{\sum_{j' \in N_2(j)} d_{j'}}{d_j} \right|$$

A linear combination of the two definitions can also be considered:

$$Z_{i,j}(w) = (1 - w)X_{i,j} + wY_{i,j}$$

The final variant of the algorithm is identical to the deferred matching variant, except for the fact that, every time a pair is selected, it selects the one with the minimum $Z_{i,j}(w)$ of all candidates with a mark counter above the threshold r and adds it to the map $A(t)$ as a new match. If there are no candidates with a mark counter above the threshold, the algorithm stops.

In **Alg. 3** is given the formal description of the variant.

Algorithm 3: Degree similarity matching variant of PGM algorithm

$A(0) = A_0$, $Z_0 = \emptyset$, $t = 0$
while $A(t) \setminus Z(t) \neq \emptyset$ **do**
  while $A(t) \setminus Z(t) \neq \emptyset$ **do**
    $t = t + 1$
    Randomly select a pair $(i_t, j_t) \in A(t-1) \setminus Z(t-1)$ and add one mark to all pairs $(i', j') \in N_1(i_t) \times N_2(j_t)$.
    $A(t) = A(t-1)$ and $Z(t) = Z(t-1) \cup \{(i_t, j_t)\}$.
  Let $B(t)$ be the set of all pairs (i', j') whose metric value $Z_{i',j'}(w)$ is minimal and whose mark counter $M_{i',j'}$ is at least r at time t.
  if $B(t) \neq \emptyset$ **then**
    Select a pair (i', j') from $B(t)$ uniformly at random and insert it in the map $A(t)$.
    All other conflicting candidates (i'', j') and (i', j'') are permanently removed from consideration in future iterations.
    $A(t) = A(t) \cup \{(i', j')\}$.
return $T = t$, $Z(T) = A(T)$, where $A(T)$ is the final map and $|A(T)| = T$.

As already done for the previously considered ones, this variant of the PGM algorithm is now tested over stochastic block model graphs, to investigate and compare its performance on this kind of networks to that of the other variants.

The metrics used to evaluate the performance of the variant are again the size of the final map and the error rate.

The variant is run over input graphs sampled from ground-truth stochastic block model network graphs with the same parameters listed in Figures 6, 7. For each case, three different values for the weight parameter w are considered: $w = 0$, $w = 1$, $w = 0.5$.

Figures 14-19 show the simulation results obtained for different $\boldsymbol{\sigma}$ vectors, the same ones adopted to build the seed sets in the testing of the previous chapter. With regard to the dependence of the size of the final map on the size of the seed set A_0, the displayed curves are consistent with the ones described in Figures 6, 7, independently on the particular value of the parameter w that is used. The curves are characterized by a phase transition in correspondence with the critical seed set size which, again, clearly depends on the particular $\boldsymbol{\sigma}$ adopted to build the seed set. The critical seed set sizes have approximately the same values as the corresponding ones in Figures 6, 7. Thus, the conclusion that the more seeds belong to a community i in which the average degree of the nodes $E[D_i]$ is high, the fewer seeds are needed, is confirmed. However, as the number of seeds increases, the total number of mapped pairs, with sampled block model graphs as input, tend to a maximum value which is lower than the one reached in the curve relative to Erdös-Rényi random graphs $G(n, p)$ with the parameter p set equal to the maximum p_{ij} value.

With regard to the error rate, its behavior as a function of the size of the seed set is much more regular and predictable in this case: all the curves show a phase transition in correspondence with the critical seed set size and are very similar to the corresponding ones relative to the size of the final map. Unfortunately, as the number of seeds increases, the error rate tends to an asymptotic value which is much higher than the typical error rate values obtained with basic PGM or its deferred matching

variant. Thus, this degree similarity matching variant of the algorithm seems to perform considerably worse than the other versions. This is probably due to the fact that the probability that the same node has similar degree values in both the input sampled graphs is much lower than expected. This apparently is also valid for the average of the degree values of its neighbor nodes. The edge sampling mechanism works in such a way that good pairs are likely to have degree similarity metric values that are worse than those of bad pairs. Therefore, the degree similarity matching variant performs so poorly because in such a scenario the mere mark distribution mechanism based on bootstrap percolation is better at counteracting the edge sampling effects, at least for the random graph model adopted. The variant keeps deferring the selection of a new match as long as possible like the deferred matching one, but, once there are no more matched pairs to visit, it selects the pair with the minimum $Z_{i,j}(w)$ of all candidates with a mark counter above the threshold r, instead of the one with the maximum $M_{i,j}$. Even the basic PGM performs better although it simply adds a pair to the map as soon as it gets at least r marks. If there are several candidate pairs that have reached r marks but are conflicting, the algorithm selects one of them uniformly at random and adds it to the map.

All the obtained results and curves are again averaged over 15 realizations.

	Number of nodes
Community 0	(1/3)n
Community 1	(1/3)n
Community 2	(1/3)n
Overall	n = 10000

p_{ij}	Community 0	Community 1	Community 2
Community 0	10/(n/3)	2/(n/3)	2/(n/3)
Community 1	2/(n/3)	30/(n/3)	2/(n/3)
Community 2	2/(n/3)	2/(n/3)	20/(n/3)

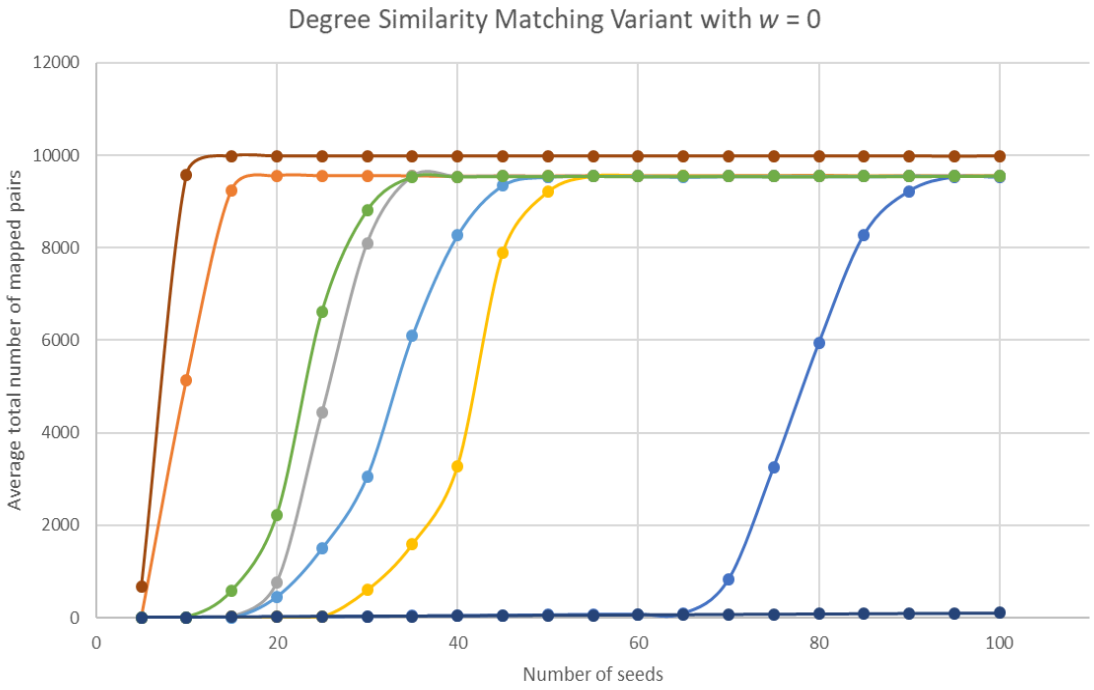
Sampling parameter s = 0.7

PGM threshold r = 2

C0 Avg. Degree $E[D_0]$ = 14

C1 Avg. Degree $E[D_1]$ = 34

C2 Avg. Degree $E[D_2]$ = 24



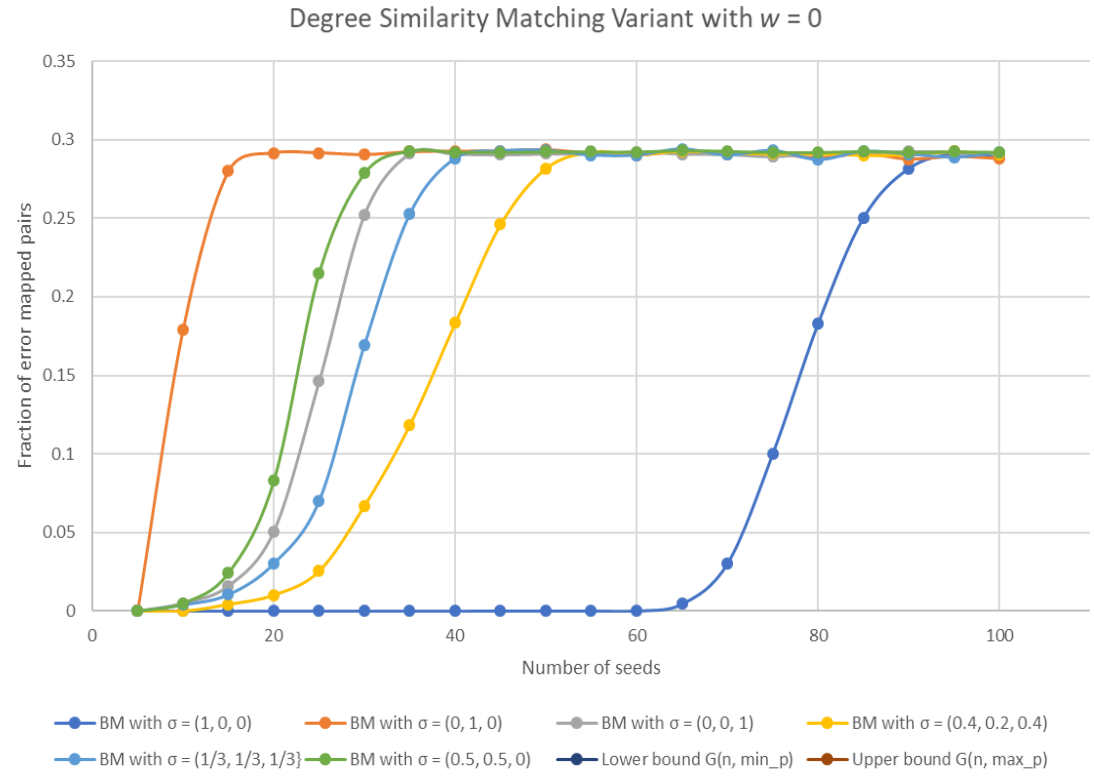


Figure 14

	Number of nodes
Community 0	(1/3)n
Community 1	(1/3)n
Community 2	(1/3)n
Overall	n = 10000

p_{ij}	Community 0	Community 1	Community 2
Community 0	10/(n/3)	2/(n/3)	2/(n/3)
Community 1	2/(n/3)	30/(n/3)	2/(n/3)
Community 2	2/(n/3)	2/(n/3)	20/(n/3)

Sampling parameter s = 0.7

PGM threshold r = 2

C0 Avg. Degree $E[D_0]$ = 14

C1 Avg. Degree $E[D_1]$ = 34

C2 Avg. Degree $E[D_2]$ = 24

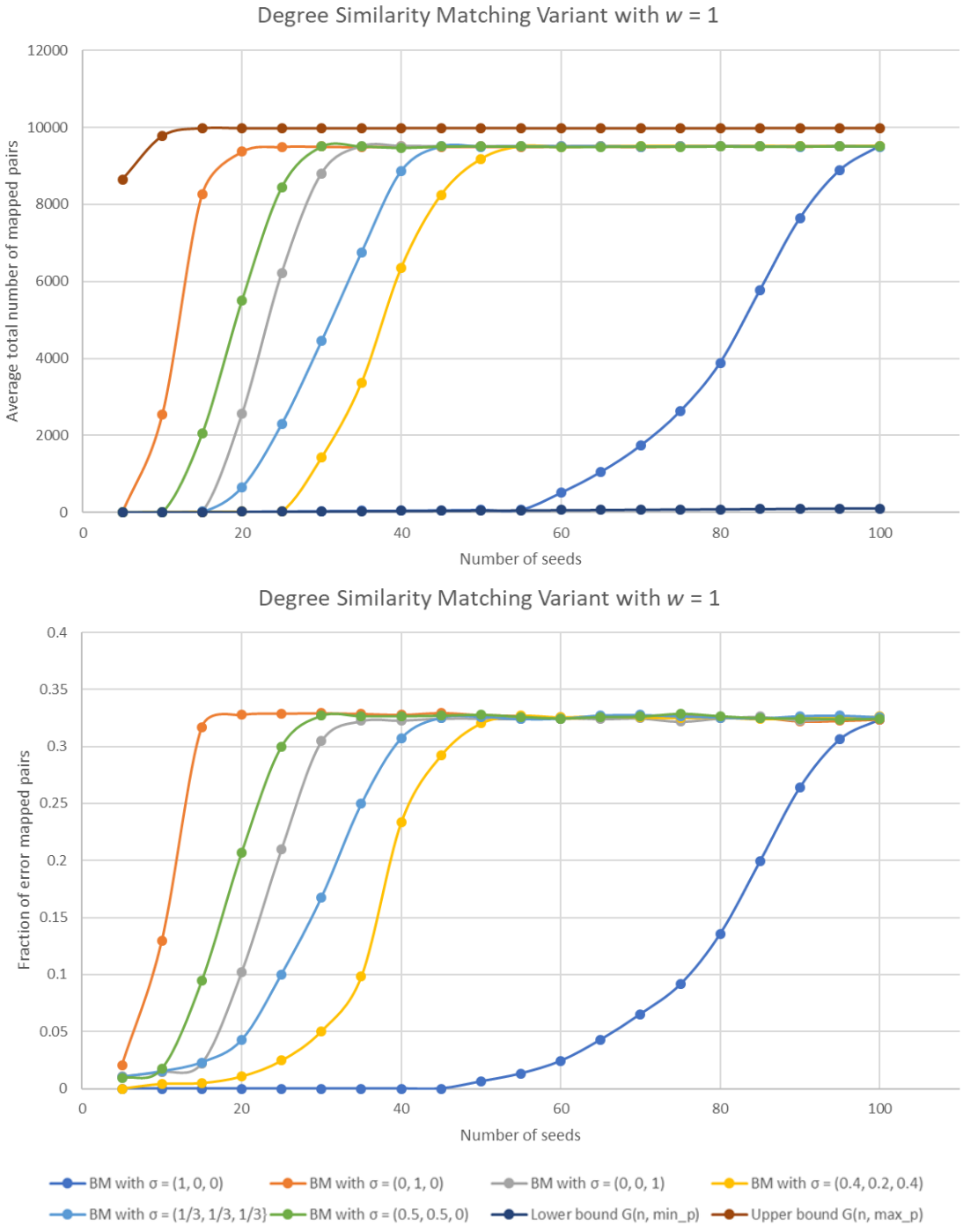


Figure 15

	Number of nodes
Community 0	(1/3)n
Community 1	(1/3)n
Community 2	(1/3)n
Overall	n = 10000

p_{ij}	Community 0	Community 1	Community 2
Community 0	10/(n/3)	2/(n/3)	2/(n/3)
Community 1	2/(n/3)	30/(n/3)	2/(n/3)
Community 2	2/(n/3)	2/(n/3)	20/(n/3)

Sampling parameter s = 0.7

PGM threshold r = 2

C0 Avg. Degree $E[D_0]$ = 14

C1 Avg. Degree $E[D_1]$ = 34

C2 Avg. Degree $E[D_2]$ = 24

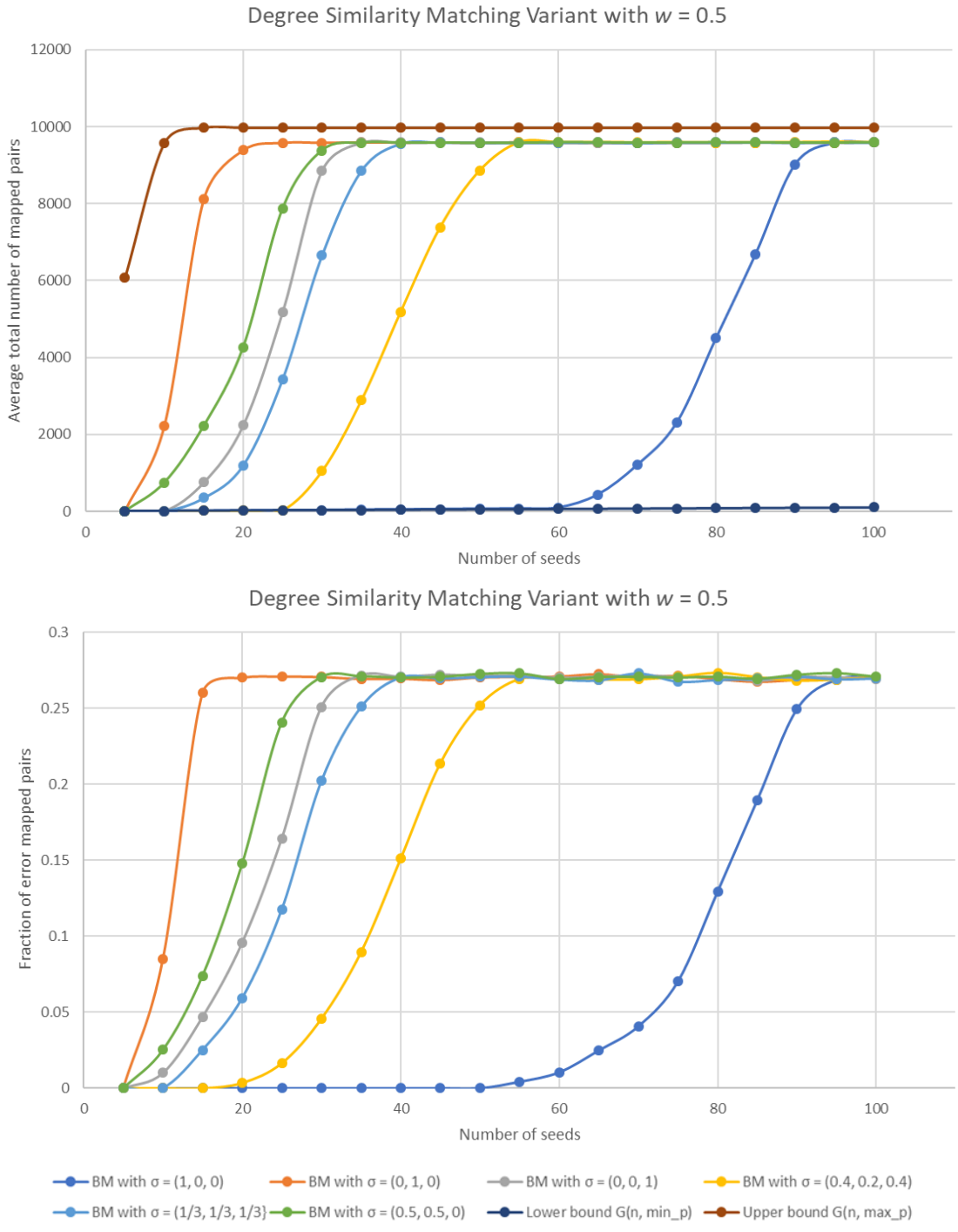


Figure 16

	Number of nodes
Community 0	(1/3)n
Community 1	(1/3)n
Community 2	(1/3)n
Overall	n = 10000

p_{ij}	Community 0	Community 1	Community 2
Community 0	2/(n/3)	10/(n/3)	5/(n/3)
Community 1	10/(n/3)	2/(n/3)	25/(n/3)
Community 2	5/(n/3)	25/(n/3)	2/(n/3)

Sampling parameter s = 0.7

PGM threshold r = 2

C0 Avg. Degree $E[D_0]$ = 17

C1 Avg. Degree $E[D_1]$ = 37

C2 Avg. Degree $E[D_2]$ = 32

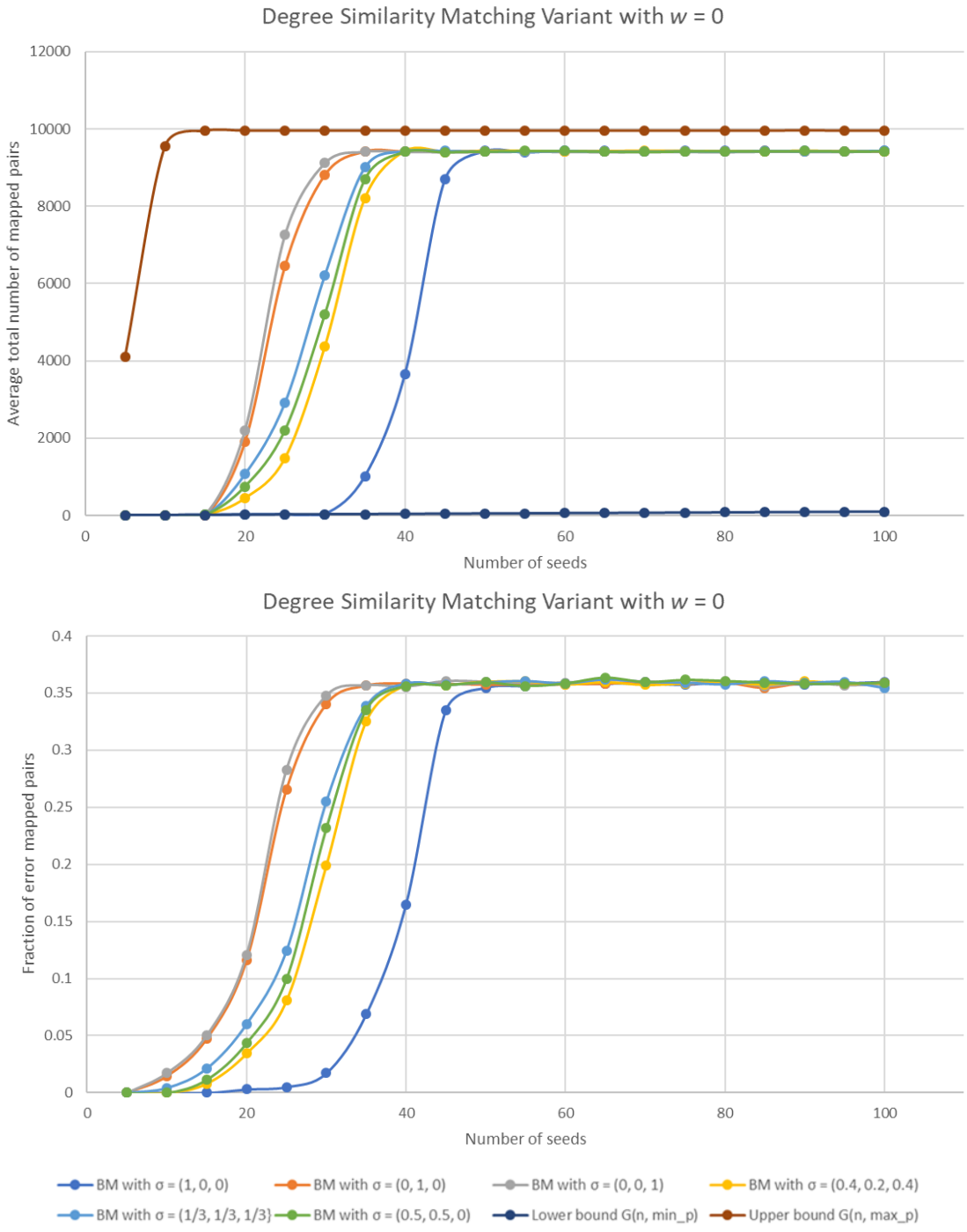


Figure 17

	Number of nodes
Community 0	(1/3)n
Community 1	(1/3)n
Community 2	(1/3)n
Overall	n = 10000

p_{ij}	Community 0	Community 1	Community 2
Community 0	2/(n/3)	10/(n/3)	5/(n/3)
Community 1	10/(n/3)	2/(n/3)	25/(n/3)
Community 2	5/(n/3)	25/(n/3)	2/(n/3)

Sampling parameter s = 0.7

PGM threshold r = 2

C0 Avg. Degree $E[D_0]$ = 17

C1 Avg. Degree $E[D_1]$ = 37

C2 Avg. Degree $E[D_2]$ = 32

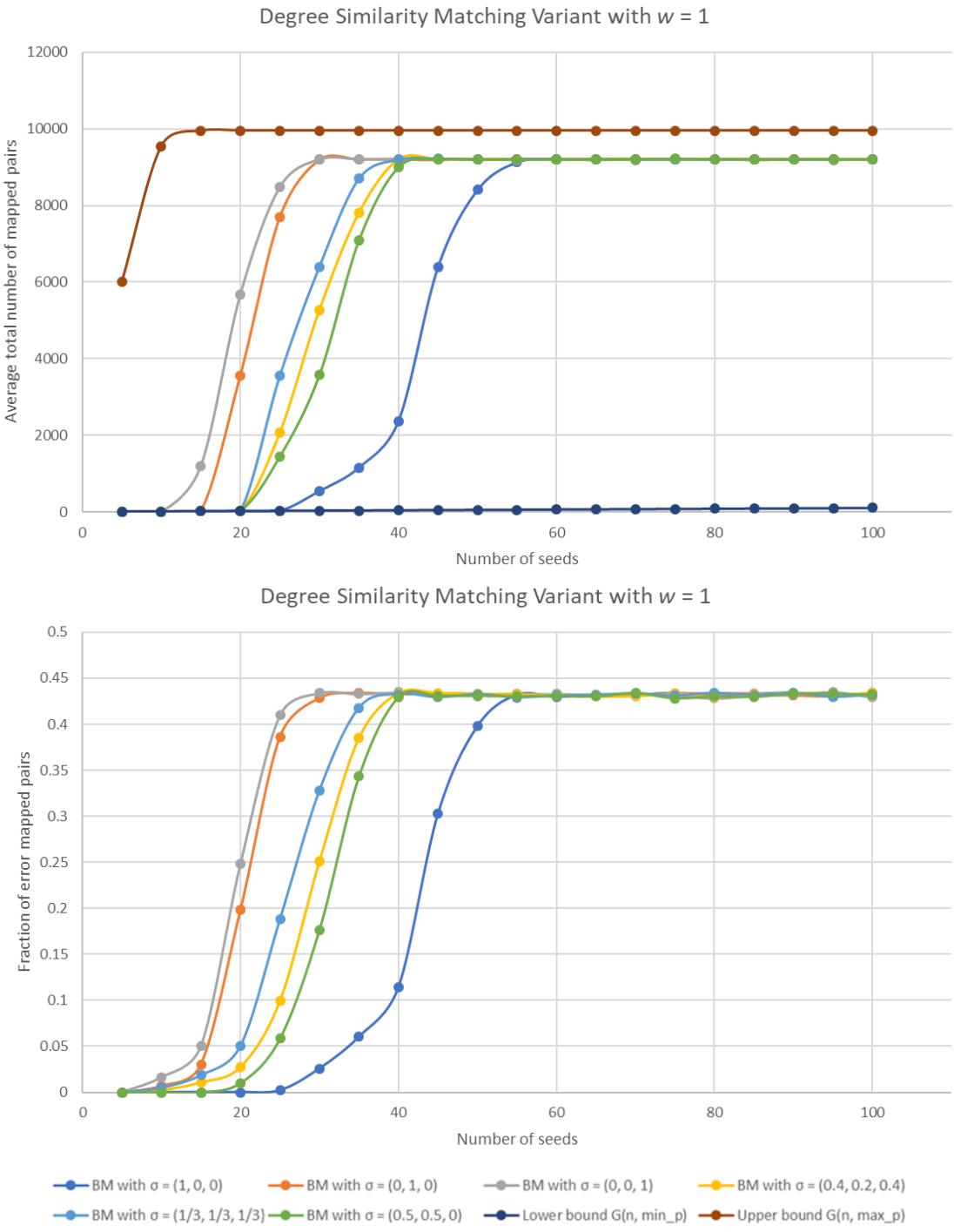


Figure 18

	Number of nodes
Community 0	(1/3)n
Community 1	(1/3)n
Community 2	(1/3)n
Overall	n = 10000

p_{ij}	Community 0	Community 1	Community 2
Community 0	2/(n/3)	10/(n/3)	5/(n/3)
Community 1	10/(n/3)	2/(n/3)	25/(n/3)
Community 2	5/(n/3)	25/(n/3)	2/(n/3)

Sampling parameter s = 0.7

PGM threshold r = 2

C0 Avg. Degree $E[D_0]$ = 17

C1 Avg. Degree $E[D_1]$ = 37

C2 Avg. Degree $E[D_2]$ = 32

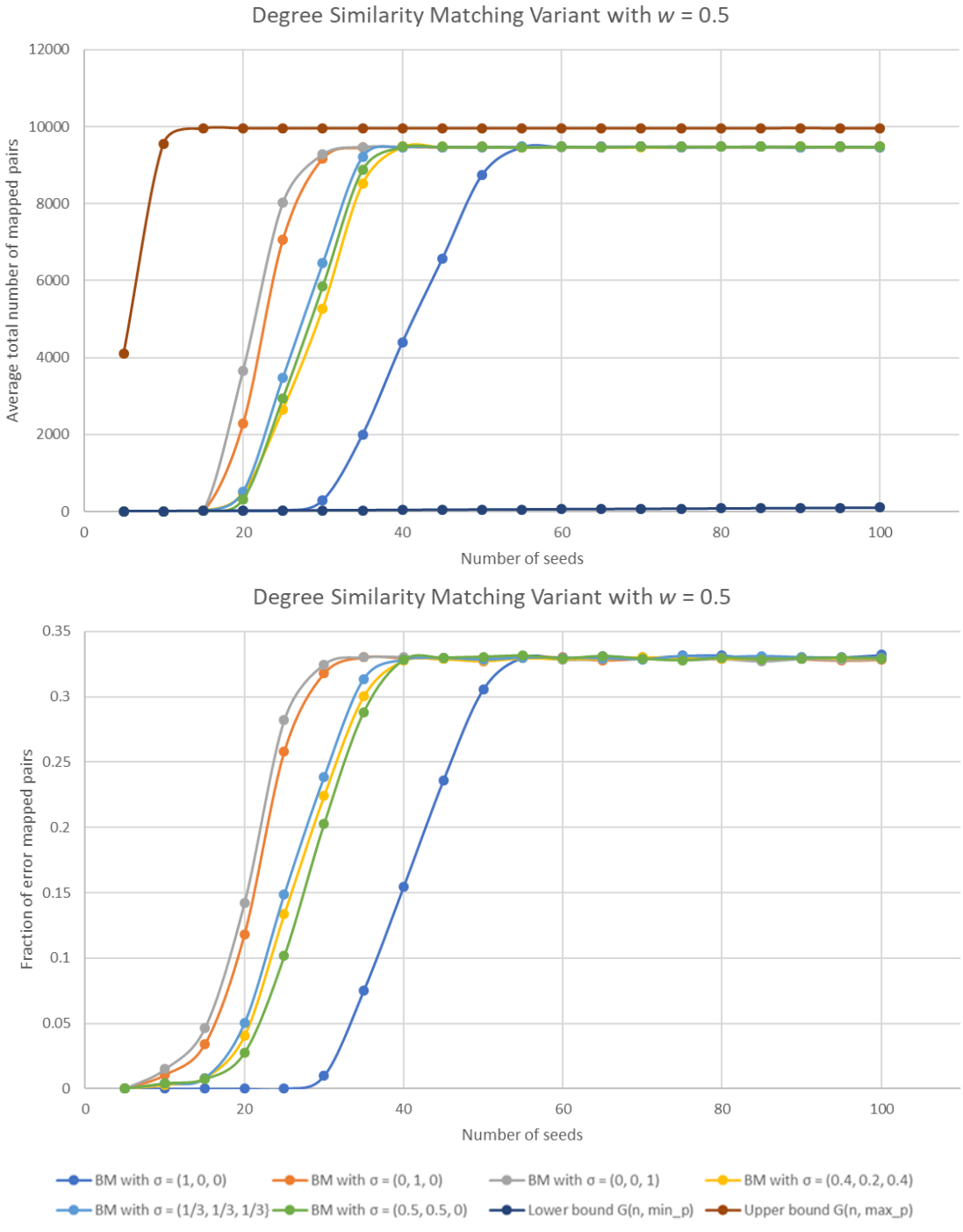


Figure 19

7. Conclusion

This thesis addresses an experimental investigation into the performance of the PGM algorithm proposed by Yartseva and Grossglauser in [19] for large-scale graph matching. The basic PGM algorithm and its deferred matching variant are described. Then, a definition of the stochastic block model used in this study is provided. Lastly a personal variant of the PGM algorithm, based on a degree similarity metric, was proposed. This variant is identical to the deferred matching one, except for the fact that it selects the pairs to be added to the map as new matches on the basis of a metric $Z_{i,j}(w)$. Such a metric was designed to measure the degree similarity between the nodes of a pair and also among the respective neighbors. The use of the PGM mark counter is still kept in order to perform a preliminary discrimination among candidate pairs.

Several experiments have been run over stochastic block model random graphs, in order to investigate the conditions on the graph parameters and the seed set size such that all the variants of the PGM algorithm propagate and perform well over them. In all cases, there is a phase transition of the size of the final map, depending on the size of the initial seed set, confirming the results expressed in [19].

A separate discussion is reserved for the error rate of the final map. When the basic PGM and its deferred matching variant are applied, the same pattern is observed: after an initial peak, the error rate tends to a lower asymptotic value as the number of seeds exceeds the corresponding critical seed set size and then increases. As it could be expected, the deferred matching variant of the PGM algorithm leads to smaller values of the error rate than the basic version, thus performs considerably better. When the degree similarity matching variant is run, the error rate of the final map always shows a clear phase transition in correspondence with the critical seed set size and tends to an asymptotic value which is much higher than that observed with the basic PGM or its deferred matching variant.

In summary, the proposed variant of the algorithm seems to perform considerably worse than the other versions. The mere mark distribution mechanism based on bootstrap percolation is better at solving the graph matching problem than the proposed degree similarity metric, at least for stochastic block model random graphs.

Bibliography

[1] E. Abbe. Community detection and stochastic block models: recent developments. Journal of Machine Learning Research, Special Issue, 2017.

[2] B. Bollobás. Random graphs, volume 73. Cambridge University Press, 2001.

[3] C.-F. Chiasserini, M. Garetto, and E. Leonardi. Social network de-anonymization under scale-free user relations. IEEE/ACM Trans. Netw., vol. 24, no. 6, pages 3756-3769, 2016.

[4] C.-F. Chiasserini, M. Garetto, and E. Leonardi. De-anonymizing clustered social networks by percolation graph matching. ACM Transactions on Knowledge Discovery from Data, TKDD, vol. 12, no. 2, art. 21, 2018.

[5] F. Chung, L. Lu. The average distance in a random graph with given expected degrees. Internet Mathematics, vol. 1, no. 1, pages 91-113, 2004.

[6] S. Janson, T. Łuczak, T. Turova, and T. Vallier. Bootstrap percolation on the random graph $G_{n,p}$. The Annals of Applied Probability, 22(5), 2012.

[7] G. W. Klau. A new graph-based method for pairwise global network alignment. BMC Bioinformatics, vol. 10, suppl. 1, 2009.

[8] N. Korula and S. Lattanzi. An efficient reconciliation algorithm for social networks. PVLDB, 7(5), 2014.

[9] O. Kuchaiev and N. Pržulj. Integrative network alignment reveals large regions of global network similarity in yeast and human. Bioinformatics, 27(10), 2011.

[10] A. Mislove, B. Viswanath, K. P. Gummadi, and P. Druschel. You are who you know: inferring user profiles in online social networks. In Proceedings of the third ACM international conference on Web search and data mining, WSDM, pages 251-260, 2010.

[11] A. Narayanan and V. Shmatikov. Robust de-anonymization of large sparse datasets. In S&P, 2008.

[12] A. Narayanan and V. Shmatikov. De-anonymizing social networks. In Proceedings of the 2009 30th IEEE Symposium on Security and Privacy, 2009.

[13] P. Pedarsani and M. Grossglauser. On the privacy of anonymized networks. In Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD, 2011.

[14] W. Peng, F. Li, X. Zou, J. Wu. A two-stage deanonymization attack against anonymized social networks. *IEEE Trans. Comput.*, vol. 63, no. 2, pages 290-303, 2014.

[15] Y.-K. Shih and S. Parthasarathy. Scalable global alignment for multiple biological networks. BMC Bioinformatics, vol. 13, suppl. 3, 2012.

[16] P. Shvaiko and J. Euzenat. Ontology matching: State of the art and future challenges. IEEE Transactions on Knowledge and Data Engineering, vol. 25, issue 1, 2013.

[17] R. Singh, J. Xu, and B. Berger. Pairwise global alignment of protein interaction networks by matching neighborhood topology. Research in Computational Molecular Biology, vol. 4453, pages 16-31, 2007.

[18] G. Wondracek, T. Holz, E. Kirda, and C. Kruegel. A practical attack to de-anonymize social network users. In IEEE Symposium on Security and Privacy, 2010.

[19] L. Yartseva and M. Grossglauser. On the performance of percolation graph matching. In Proceedings of the First ACM Conference on Online Social Networks, COSN, 2013.

Appendix

Here is given the source code of the program developed and used to test the performance of the different variants of the PGM algorithm. It is written in C++, with a strong object-oriented approach.

In `Thesis_PGM.cpp`, which is the main file, are defined the values of the different parameters, including those of the specific random graph model used to create the input graphs over which the PGM is run, the ones useful to specify the different ways of picking the seeds and the parameter r of the PGM.

Ground-truth network graphs can be built by calling the constructors of either `Gnp` or `BlockModel` classes which are both derived classes of `Network`. Each `Network` object has its own member map `Nodes` which stores the pointers to all the `Node` objects of the `Network` itself, indexed by their `net_ID`. Each `Node` object has its own member `net_ID` and `true_ID`: the first one is a random identifier which is merely given to distinguish a `Node` from the others in the same `Network` object, whereas the second one represents the true and hidden identity of a `Node`. This means that, in the graphs that are given to the PGM as input, built by edge sampling the original ground-truth `Network`, same `Nodes` have identical `true_IDs` but different `net_IDs`. Lastly, each `Node` in a `Network` has its own `neighbors` array which contains the `net_IDs` of all its neighbor `Nodes`. In all the developed classes, only bidirectional links are considered.

The sampled network graphs are built by calling the constructors of either `Gnps` or `BlockModelSampled` classes, depending on whether the ground-truth `Network` is a `Gnp` or a `BlockModel` one respectively.

`PGM` is the class to which belong static methods that are used to run the different variants of the PGM algorithm. All these methods need a seed set as input and return the final map as output. Both the seed set and the final map are given as an instance of class `Matching`. Each `Matching` object has its own member map `matchingNodePairs`. This map stores the pointers to all the `NodePair` objects that represent the pairs of nodes that have been mapped by the algorithm. Intuitively, the implemented algorithm iteratively expands the seed set given as input, returning the final map when it stops. Consequently, the same `Matching` instance represents both the seed set and the final map, at different moments.

```
//============================================================================
// Name        : Thesis_PGM.cpp
// Author      : Fabio GENNARI
// Version     :
// Copyright   :
// Description : Main file
//============================================================================

#include <iostream>
#include <fstream>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include "Gnp.h"
#include "Gnps.h"
#include "Node.h"
#include "PGM.h"
#include "BlockModel.h"
#include "BlockModelSampled.h"

#define M 3                                 // number of communities
#define S 3                                 // number of different sampling probability values
#define P 6                                 // number of different ways of picking the seeds
                                            // from communities
#define N 20                                // number of different sizes for the seed set
#define REALIZATIONS 15

using namespace std;

double arithmeticMean(int *array, int size);
double arithmeticMean(double *array, int size);

double min_p();
double max_p();

/* Due to Gnp::extendedRand() implementation, this code works
 * only for n not greater than a billion (10^9). */

/* Parameters for Block Model testing. */
int n = 10000;
double probNodeAssignment[M] = {1.0/3.0, 1.0/3.0, 1.0/3.0};      // probability that a node
                                                                 // of the BM network to be
                                                                 // created belongs to Comm. i

double p[M][M] = {{2.0/(n/3.0), 10.0/(n/3.0), 5.0/(n/3.0)},      // edge probability matrix P
                  {10.0/(n/3.0), 2.0/(n/3.0), 25.0/(n/3.0)},
                  {5.0/(n/3.0), 25.0/(n/3.0), 2.0/(n/3.0)}};

double s = 0.7;

double probSeedAssignment[P][M] = {{1.0, 0.0, 0.0},              // each row of the
                                   {0.0, 1.0, 0.0},              // two-dimensional array
                                   {0.0, 0.0, 1.0},              // is a different σ_i
                                   {0.4, 0.2, 0.4},              // to be tested
                                   {1.0/3, 1.0/3, 1.0/3},
                                   {0.5, 0.5, 0.0}};

int n_seeds[N];
int n_seeds_resolution = 5;

int r = 2;
```

```
int main() {

	/* Initializes random seed. */
	srand(time(NULL));

	/* Removes all previous output files from the directory. */
	system("del Output_Prints\\*.txt");

	/* Opens the output file for writing. */
	std::ofstream ofs;
	ofs.open("Output_Prints\\Perf_Metrics.txt", std::ofstream::out | std::ofstream::trunc);

	/* Initializes n_seeds. */
	for (int i=0; i<N; i++)
		n_seeds[i] = (i+1)*n_seeds_resolution;

	/* Allocates and initializes the matrix for p values. */
	double **z = (double **)malloc(M*sizeof(double *));
	for (int i=0; i<M; i++) {
		z[i] = (double *)malloc(M*sizeof(double));
		for (int j=0; j<M; j++)
			z[i][j] = p[i][j];
	}

	BlockModel *bm = new BlockModel(n, M, probNodeAssignment, z);
	Gnp *lb = new Gnp(n, min_p());
	Gnp *ub = new Gnp(n, max_p());

	/* Test code. */
	int *communitiesSize = bm->getCommunitiesSize();
	for (int i=0; i<M; i++)
		ofs << "Community " << i << ":\t" << communitiesSize[i] << std::endl;
	free(communitiesSize);

	for (int i=0; i<P; i++) {
		ofs << std::endl << "Seed picking {";
		for (int j=0; j<M-1; j++)
			ofs << probSeedAssignment[i][j] << ", ";
		ofs << probSeedAssignment[i][M-1] << "}" << std::endl << std::endl;

		for (int j=0; j<N; j++) {
			int *performedMatches = (int *)malloc(REALIZATIONS*sizeof(int));
			double *fractionOfWrongMatches = (double *)malloc(REALIZATIONS*sizeof(double));

			for (int k=0; k<REALIZATIONS; k++) {
				BlockModelSampled *bms1 = new BlockModelSampled(n, s, bm);
				BlockModelSampled *bms2 = new BlockModelSampled(n, s, bm);

				Matching *seeds = bm->createSeedSet(bms1, bms2,
								probSeedAssignment[i], n_seeds[j]);

				Matching *matches = PGM::deferredMatchingPGM(seeds, r);

				performedMatches[k] = matches->getNumberOfPerformedMatches();
				fractionOfWrongMatches[k] = matches->getFractionOfWrongMatches();

				delete bms1;
				delete bms2;
				delete matches;
			}

			ofs << "N. of seeds: \t" << n_seeds[j] << "\t";
```

```
                ofs << "N. of performed matches: \t" <<
                        arithmeticMean(performedMatches, REALIZATIONS) << "\t";
                ofs << "Fraction of wrong matches: \t" <<
                        arithmeticMean(fractionOfWrongMatches, REALIZATIONS);
                ofs << std::endl;

                free(performedMatches);
                free(fractionOfWrongMatches);
        }
}

/* Lower bound Gnp. */
ofs << std::endl << "Lower bound G(n,min_p)" << std::endl << std::endl;
for (int j=0; j<N; j++) {
        int *performedMatches = (int *)malloc(REALIZATIONS*sizeof(int));
        double *fractionOfWrongMatches = (double *)malloc(REALIZATIONS*sizeof(double));

        for (int k=0; k<REALIZATIONS; k++) {
                Gnps *lbs1 = new Gnps(n, min_p(), s, lb);
                Gnps *lbs2 = new Gnps(n, min_p(), s, lb);

                Matching *seeds = PGM::getRandomSeedSet(lbs1, lbs2, lb, n_seeds[j]);
                Matching *matches = PGM::arithmeticPGM(seeds, r);

                performedMatches[k] = matches->getNumberOfPerformedMatches();
                fractionOfWrongMatches[k] = matches->getFractionOfWrongMatches();

                delete lbs1;
                delete lbs2;
                delete matches;
        }

        ofs << "N. of seeds: \t" << n_seeds[j] << "\t";
        ofs << "N. of performed matches: \t" <<
                        arithmeticMean(performedMatches, REALIZATIONS) << "\t";
        ofs << "Fraction of wrong matches: \t" <<
                        arithmeticMean(fractionOfWrongMatches, REALIZATIONS);
        ofs << std::endl;

        free(performedMatches);
        free(fractionOfWrongMatches);
}

/* Upper bound Gnp. */
ofs << std::endl << "Upper bound G(n,max_p)" << std::endl << std::endl;
for (int j=0; j<N; j++) {
        int *performedMatches = (int *)malloc(REALIZATIONS*sizeof(int));
        double *fractionOfWrongMatches = (double *)malloc(REALIZATIONS*sizeof(double));

        for (int k=0; k<REALIZATIONS; k++) {
                Gnps *ubs1 = new Gnps(n, max_p(), s, ub);
                Gnps *ubs2 = new Gnps(n, max_p(), s, ub);

                Matching *seeds = PGM::getRandomSeedSet(ubs1, ubs2, ub, n_seeds[j]);
                Matching *matches = PGM::arithmeticPGM(seeds, r);

                performedMatches[k] = matches->getNumberOfPerformedMatches();
                fractionOfWrongMatches[k] = matches->getFractionOfWrongMatches();

                delete ubs1;
                delete ubs2;
                delete matches;
        }
```

```
			ofs << "N. of seeds: \t" << n_seeds[j] << "\t";
			ofs << "N. of performed matches: \t" <<
					arithmeticMean(performedMatches, REALIZATIONS) << "\t";
			ofs << "Fraction of wrong matches: \t" <<
					arithmeticMean(fractionOfWrongMatches, REALIZATIONS);
			ofs << std::endl;

			free(performedMatches);
			free(fractionOfWrongMatches);
		}

		/* Deletes the original BlockModel Network. */
		delete bm;

		/* Deletes the original lower and upper bound Gnp Network. */
		delete lb;
		delete ub;

		/* Frees dynamically allocated memory. */
		for (int i=0; i<M; i++)
			free(z[i]);
		free(z);

		/* Frees dynamically allocated memory of Gnp class. */
		Gnp::freeHeap();

		/* Closes the output file. */
		ofs.close();

		return 0;
}

double arithmeticMean(int *array, int size) {
		int sum = 0;

		for (int i=0; i<size; i++)
			sum += array[i];

		double mean = (double)sum;
		mean /= size;

		return mean;
}

double arithmeticMean(double *array, int size) {
		double mean = 0.0;

		for (int i=0; i<size; i++)
			mean += array[i];

		mean /= size;

		return mean;
}
```

```
double min_p() {
	double min_p = p[0][0];

	for (int i=0; i<M; i++)
		for (int j=0; j<M; j++)
			if (p[i][j]<min_p)
				min_p = p[i][j];

	return min_p;
}

double max_p() {
	double max_p = p[0][0];

	for (int i=0; i<M; i++)
		for (int j=0; j<M; j++)
			if (p[i][j]>max_p)
				max_p = p[i][j];

	return max_p;
}
```

```
/*
 * BlockModel.cpp
 *
 *  Created on: 06 mag 2018
 *      Author: Fabio GENNARI
 */

#include <stdlib.h>
#include <iostream>
#include <string>
#include "BlockModel.h"

/*                                      */
/*     Instance fields and methods.     */
/*                                      */

/* Creates the BlockModel Network, with m communities. The probability that a Node is assigned to
 * a certain community is specified by probAssignment, whereas the probability that a link exists
 * between a Node belonging to community i and a Node belonging to community j is specified by
 * pp[i][j]. pp must be a symmetric matrix. */
BlockModel::BlockModel(int nn, int m, double *probAssignment, double **pp): Network(nn) {

	/* Initialization of instance field. */
	numberOfCommunities = m;

	/* Initialization of instance field. */
	probOfMembership = (double *)malloc(numberOfCommunities*sizeof(double));
	for (int i=0; i<numberOfCommunities; i++)
		probOfMembership[i] = probAssignment[i];

	/* Initialization of instance field. */
	membershipCDF = (double *)malloc((numberOfCommunities+1)*sizeof(double));
	membershipCDF[0] = 0.0;
	for (int i=0; i<numberOfCommunities; i++)
		membershipCDF[i+1] = membershipCDF[i] + probOfMembership[i];

	/* Initialization of instance field. */
	comm_p = (double **)malloc(numberOfCommunities*sizeof(double *));
	for (int i=0; i<numberOfCommunities; i++) {
		comm_p[i] = (double *)malloc(numberOfCommunities*sizeof(double));
		for (int j=0; j<numberOfCommunities; j++)
			comm_p[i][j] = pp[i][j];
	}

	/* Initialization of instance field. */
	communities = (Gnp **)malloc(numberOfCommunities*sizeof(Gnp *));
	for (int i=0; i<numberOfCommunities; i++)
		communities[i] = new Gnp();

	/* Scans all the Nodes of the Network and assigns them to a community according
	 * to the probOfMembership probability values. */
	std::map<int, Node *>::iterator it;
	for (it=Nodes.begin(); it!=Nodes.end(); it++) {
		double rand = Gnp::uniRandZeroToOne();
		bool extractedInterval = false;
		int i = 0;
		for (; i<numberOfCommunities && !extractedInterval; i++)
			if (rand>=membershipCDF[i] && rand<membershipCDF[i+1])
				extractedInterval = true;
		i--;
		communities[i]->addNode(it->second);
	}
```

```
	/* Creates the links of the Network among Nodes belonging to the same community as well as
	 * those between Nodes belonging to different communities. */
	int totIterations = numberOfCommunities + (numberOfCommunities*
				numberOfCommunities - numberOfCommunities)/2;
	int v = 0;
	for (int i=0; i<numberOfCommunities; i++)
		for (int j=i; j<numberOfCommunities; j++) {
			if (i==j)
				communities[i]->createLinks(comm_p[i][i]);
			else
				createLinksBetweenCommunities(i,j);

			v++;
			std::cout << "Link generation: " <<
					(v + 0.0)*100/totIterations << "%" << std::endl;
		}

}

BlockModel::~BlockModel() {
	free(probOfMembership);
	free(membershipCDF);

	for (int i=0; i<numberOfCommunities; i++)
		free(comm_p[i]);
	free(comm_p);

	for (int i=0; i<numberOfCommunities; i++)
		delete communities[i];
	free(communities);

	net_n = 0;
}

/* Returns an array containing the size value of each community. */
int *BlockModel::getCommunitiesSize() {
	int *sizes = (int *)malloc(numberOfCommunities*sizeof(int));

	for (int i=0; i<numberOfCommunities; i++)
		sizes[i] = communities[i]->getNet_n();

	return sizes;
}
```

```
/* For every Node belonging to community i, extracts a number of links towards Nodes
 * belonging to community j, according to a binomial distribution Bi(n_j,comm_p_ij).
 * Then creates as many links as the extracted number between the considered Node and
 * the others which are extracted randomly from community j. */
void BlockModel::createLinksBetweenCommunities(int i, int j) {
	std::map<int, Node *>::iterator it;

	/* Gnp::binomialDistr() was designed to extract the temporary degree of a Node inside
	 * a G(n,p) Network. The maximum possible degree for a Node in this scenario is (net_n -1)
	 * since the considered Node can't be counted. Since the communities are different here,
	 * the maximum number of links is equal to net_n of community j. Hence the first parameter
	 * is increased by 1. */
	Gnp::binomialDistr((communities[j]->getNet_n() + 1), comm_p[i][j]);
	for (it=communities[i]->Nodes.begin(); it!=communities[i]->Nodes.end(); it++) {
		int binDeg = Gnp::randomBinDegree();
		for (int i=0; i<binDeg; i++) {
			int nodeID = extractNeighborID(it->second, j);
			it->second->addNeighbor(nodeID);
			Nodes[nodeID]->addNeighbor(it->first);
		}
	}
}

/* Extracts the net_ID of a Node belonging to community j that is going to become a neighbor
 * of Node node. If the extracted Node is already linked to Node node, pick the following one
 * in communities[j]->Nodes until this condition changes. */
int BlockModel::extractNeighborID(Node *node, int j) {
	std::map<int, Node *>::iterator it;

	int rand = (Gnp::extendedRand()%communities[j]->getNet_n());
	it = communities[j]->Nodes.begin();
	for (int i=0; i<rand; i++)
		it++;

	while (node->isNeighbor(it->first)) {
		it++;
		if (it==communities[j]->Nodes.end())
			it = communities[j]->Nodes.begin();
	}

	return it->first;
}

void BlockModel::printDetailedDegreeDistribution() {
	std::ofstream degDistr;
	std::map<int, Node *>::iterator it;

	/* Global Network degree distribution. */
	degDistr.open("Output_Prints\\GlobalNetwork.txt", std::ofstream::out | std::ofstream::trunc);
	for (it=Nodes.begin(); it!=Nodes.end(); it++)
		degDistr << it->second->getDegree() << std::endl;
	degDistr.close();

	/* Single communities degree distributions. */
	for (int i=0; i<numberOfCommunities; i++) {
		degDistr.open("Output_Prints\\Community_" + std::to_string(i) + ".txt",

	std::ofstream::out | std::ofstream::trunc);
		for (it=communities[i]->Nodes.begin(); it!=communities[i]->Nodes.end(); it++)
			degDistr << it->second->getDegree() << std::endl;
		degDistr.close();
	}
```

```
	/* Single communities towards single communities partial degree distributions. */
	for (int i=0; i<numberOfCommunities; i++)
		for (int j=0; j<numberOfCommunities; j++) {
			degDistr.open("Output_Prints\\Community_" +
				std::to_string(i) + "_towards_Community_" +
				std::to_string(j) + ".txt", std::ofstream::out | std::ofstream::trunc);
			for (it=communities[i]->Nodes.begin(); it!=communities[i]->Nodes.end(); it++)
				degDistr << getPartialDegreeTowardsCommunity(it->second, j) <<
				                                                        std::endl;
			degDistr.close();
		}

}

/* Returns the number of links between Node node and Nodes belonging to community j. */
int BlockModel::getPartialDegreeTowardsCommunity(Node *node, int j) {
	int *neighbors = node->getNeighbors();
	int deg = node->getDegree();

	int partialDeg = 0;
	for (int i=0; i<deg; i++)
		if (communities[j]->Nodes.find(neighbors[i])!=communities[j]->Nodes.end())
			partialDeg++;

	free(neighbors);

	return partialDeg;
}

/* Prepares a seed set of size size from which the PGM algorithm can start. Seeds are not picked
 * uniformly at random among all the Nodes. Every seed has instead a probability of being picked
 * from community i equal to communityProb[i] value. */
Matching *BlockModel::createSeedSet(Network *net1, Network *net2, double *communityProb, int size) {
	/* Initialization of the CDF useful to extract the community
	 * from which the seed is picked. */
	double *communityCDF = (double *)malloc((numberOfCommunities+1)*sizeof(double));
	communityCDF[0] = 0.0;
	for (int i=0; i<numberOfCommunities; i++)
		communityCDF[i+1] = communityCDF[i] + communityProb[i];

	Matching *seedSet = new Matching(net1, net2);

	/* For size times */
	for (int i=0; i<size; i++) {
		/* Extract a community according to the provided probability values. */
		double rand1 = Gnp::uniRandZeroToOne();
		bool extractedInterval = false;
		int j = 0;
		for (; j<numberOfCommunities && !extractedInterval; j++)
			if (rand1>=communityCDF[j] && rand1<communityCDF[j+1])
				extractedInterval = true;
		j--;

		/* Pick a Node from the extracted community, uniformly at random. */
		int n = communities[j]->getNet_n();
		int rand2 = Gnp::extendedRand();
		rand2 %= n;

		std::map<int, Node *>::iterator it;
		it = communities[j]->Nodes.begin();
		for (int k=0; k<rand2; k++)
			it++;
```

```
                /* If the extracted Node has already been added to the seedSet, pick the next
                 * available one. Since the seed set is always made of good pairs only, the check
                 * can be performed on only one of the two sets of net_IDs of Matching seedSet. */
                while (seedSet->checkFirstNodeID(net1->True_IDToNet_ID[it->second->getTrue_ID()])) {
                        it++;
                        if (it==communities[j]->Nodes.end())
                                it = communities[j]->Nodes.begin();
                }

                /* Add the seed to the seedSet. */
                int extractedNodeTrueID = it->second->getTrue_ID();
                Node *firstNetNode = net1->Nodes[net1->True_IDToNet_ID[extractedNodeTrueID]];
                Node *secondNetNode = net2->Nodes[net2->True_IDToNet_ID[extractedNodeTrueID]];

                seedSet->addMatching(new NodePair(firstNetNode, secondNetNode), NOCHECKVAL);

                std::cout << "Seed set generation: " << (i + 1.0)*100/size << "%" << std::endl;
        }

        return seedSet;
}

/* Prints the percentage of good pairs of Matching m that belong to each community. */
void BlockModel::matchingCommunityDistr(Matching *m, std::ofstream *ofs) {
        int *communityCounter = (int *)malloc(numberOfCommunities*sizeof(int));
        for (int i=0; i<numberOfCommunities; i++)
                communityCounter[i] = 0;

        /* Scan all the performed matches. */
        std::map<int, NodePair *>::iterator it;
        for (it=m->matchingNodePairs.begin(); it!=m->matchingNodePairs.end(); it++)
                /* If it is a good pair, find the community to which the corresponding Nodes belong to
                 * and increment the corresponding counter. */
                if (it->second->goodPair()) {
                        int trueID = it->second->getFirst()->getTrue_ID();
                        bool communityFound = false;
                        for (int j=0; j<numberOfCommunities && !communityFound; j++)
                                if (communities[j]->True_IDToNet_ID.find(trueID)!=
                                                communities[j]->True_IDToNet_ID.end()) {
                                        communityFound = true;
                                        communityCounter[j]++;
                                }
                }

        /* Prints the percentage values for each Community. */
        int totalGoodPairs = m->getNumberOfGoodMatches();
        for (int i=0; i<numberOfCommunities; i++)
                *ofs << "% of GPs for Community " << i << "\t" <<
                        (communityCounter[i] + 0.0)*100/totalGoodPairs << "%\t";

        free(communityCounter);
}
```

```
/*
 * BlockModel.h
 *
 *  Created on: 06 mag 2018
 *      Author: Fabio GENNARI
 */

#include "Network.h"
#include "Gnp.h"
#include "Matching.h"

#ifndef BLOCKMODEL_H_
#define BLOCKMODEL_H_

class BlockModel: public Network {
public:
	BlockModel(int nn, int m, double *probAssignment, double **pp);
	virtual ~BlockModel();

	int *getCommunitiesSize();

	void printDetailedDegreeDistribution();

	Matching *createSeedSet(Network *net1, Network *net2, double *communityProb, int size);

	void matchingCommunityDistr(Matching *m, std::ofstream *ofs);

private:
	int numberOfCommunities;
	Gnp **communities;                  // array containing the pointers to the
	                                    // various Gnp communities
	double *probOfMembership;           // probability that a Node belongs to
	                                    // the corresponding community
	double *membershipCDF;              // corresponding CDF useful to extract the community
	                                    // to which a Node will belong
	double **comm_p;                    // comm_p[i][j] stores the value of the probability
	                                    // that a link exists between any Node belonging to
	                                    // community i and any Node belonging to community j

	void createLinksBetweenCommunities(int i, int j);
	int extractNeighborID(Node *node, int j);

	int getPartialDegreeTowardsCommunity(Node *node, int j);
};

#endif /* BLOCKMODEL_H_ */
```

```
/*
 * BlockModelSampled.cpp
 *
 *  Created on: 22 mag 2018
 *      Author: Fabio GENNARI
 */

#include <math.h>
#include <stdlib.h>
#include <iostream>
#include <map>
#include "Network.h"
#include "BlockModelSampled.h"

/*                                      */
/*     Instance fields and methods.     */
/*                                      */

BlockModelSampled::BlockModelSampled(int nn, double ss, BlockModel *net): Network(nn) {
	std::map<int, Node *>::iterator it;
	net_s = ss;
	motherBlockModel = net;

	/* Sampling probability of a link net_s comes from two sampling iterations of the same link
	 * performed by inspecting the neighborhood of the considered adjacent nodes. The resulting
	 * "unidirectional" sampling probability x is defined as the following. */
	double x = 1 - sqrt(1-net_s);

	/* For each node, sample the starting neighborhood with probability x. For each successful
	 * sample create the corresponding bidirectional link.
	 * Note that the net_IDs of the Nodes in the new Network are different w.r.t. the old one.
	 * The sampling operation must be performed considering the true_IDs of the Nodes, which
	 * truly identify them in every Network. */
	for (it=Nodes.begin(); it!=Nodes.end(); it++) {
		int nodeTrueID = it->second->getTrue_ID();

		int *startingNeighborhood =
			motherBlockModel->Nodes[motherBlockModel->
			True_IDToNet_ID[nodeTrueID]]->getNeighbors();
		int startingDegree =
			motherBlockModel->Nodes[motherBlockModel->
			True_IDToNet_ID[nodeTrueID]]->getDegree();

		for (int i=0; i<startingDegree; i++) {
			double rand = Gnp::uniRandZeroToOne();
			if (rand<x) {
				int neighborNewNetID =
					True_IDToNet_ID[motherBlockModel->
					Nodes[startingNeighborhood[i]]->getTrue_ID()];
				it->second->addNeighbor(neighborNewNetID);
				Nodes[neighborNewNetID]->addNeighbor(it->first);
			}
		}

		free(startingNeighborhood);

		std::cout << "Link generation: " << (it->first + 1.0)*100/net_n << "%" << std::endl;
	}
}

BlockModelSampled::~BlockModelSampled() {

}
```

```
/*
 * BlockModelSampled.h
 *
 *  Created on: 22 mag 2018
 *      Author: Fabio GENNARI
 */

#include "Network.h"
#include "Gnp.h"
#include "BlockModel.h"

#ifndef BLOCKMODELSAMPLED_H_
#define BLOCKMODELSAMPLED_H_

class BlockModelSampled: public Network {
public:
	BlockModelSampled(int nn, double ss, BlockModel *net);
	virtual ~BlockModelSampled();

private:
	double net_s;
	BlockModel *motherBlockModel;
};

#endif /* BLOCKMODELSAMPLED_H_ */
```

```
/*
 * Gnp.cpp
 *
 *  Created on: 23 dic 2017
 *      Author: Fabio GENNARI
 */

#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
#include <math.h>
#include <limits.h>
#include <iostream>
#include "Gnp.h"
#include "Node.h"

/*                                  */
/*     Static fields and methods.   */
/*                                  */

double *Gnp::binomialCumDistr = NULL;
double *Gnp::binomialPdf = NULL;
double *Gnp::logFact = NULL;
double Gnp::p = 0.0;
int Gnp::n = 0;
int Gnp::imaxCDF = 0;

/* Generates a sample of a random variable uniformly distributed over [0,1). */
double Gnp::uniRandZeroToOne() {
	return (((double)rand())/(RAND_MAX+1.));
}

/* Generates a sample of a random variable uniformly distributed over
 * [0, (RAND_MAX+1)*(RAND_MAX+1)). */
int Gnp::extendedRand() {
	int a = rand();
	int b = rand();
	int c = a*(RAND_MAX+1)+b;

	return c;
}

/* Generates a binomial distribution represented by means of a vector.
 * nn is the number of nodes of the network, pp is the edge probability. */
void Gnp::binomialDistr(int nn, double pp) {
	n = nn;
	p = pp;

	/* Deallocates arrays. */
	free(binomialCumDistr);
	free(binomialPdf);
	free(logFact);

	/* Dynamically allocate arrays. */
	binomialCumDistr = (double *)malloc((n+1)*sizeof(double));
	binomialPdf = (double *)malloc(n*sizeof(double));
	logFact = (double *)malloc((n+1)*sizeof(double));

	/* Evaluate array of logarithmic factorials. */
	logFact[0] = 0.0;
	for(int i=1; i<n+1; i++) {
		logFact[i] = logFact[i-1] + log(i);
	}
```

```
	/* Evaluate array of binomial pdf. */
	if (p!=0 && p!=1)
		for (int i=0; i<n; i++) {
			binomialPdf[i] = exp(logBinCoeff((n-1), i) + i*log(p) + (n-1-i)*log(1-p));
		}
	else if (p==0) {
		binomialPdf[0] = 1.0;
		for (int i=1; i<n; i++) {
			binomialPdf[i] = 0.0;
		}
	} else {
		for (int i=0; i<n-1; i++) {
			binomialPdf[i] = 0.0;
		}
		binomialPdf[n-1] = 1.0;
	}

	/* Evaluate array for binomial CDF. */
	binomialCumDistr[0] = 0.0;
	for (int i=1; i<n+1 && binomialCumDistr[i-1]<1; i++) {
		if (binomialCumDistr[i-1]==0 && binomialPdf[i-1]==0)
			binomialCumDistr[i] = 0.0;
		else {
			int startWin = floor(fmax(0, i-n*0.01));
			binomialCumDistr[i] = binomialCumDistr[startWin];
			for(int j=startWin; j<i; j++)
				binomialCumDistr[i] += binomialPdf[j];
		}

		if (binomialCumDistr[i]>=1 || i==n)
			imaxCDF = i;
	}

}

/* Generates the binomial coefficient nn over m. */
double Gnp::logBinCoeff(int nn, int m) {
	return logFact[nn]-(logFact[m] + logFact[nn-m]);
}

/* Extracts a uniform random variable and returns the corresponding extracted degree for the node
 * according to the binomial distribution. n is the number of nodes of the network. */
int Gnp::randomBinDegree() {
	int i;
	double randNumber = uniRandZeroToOne();
	i = binSearch(randNumber);
	i--;
	return i;
}

/* Clears all dynamically allocated memory. */
void Gnp::freeHeap() {
	free(logFact);
	free(binomialPdf);
	free(binomialCumDistr);
}
```

```
/* Binary search for binomial CDF. */
int Gnp::binSearch(double rand) {
	int p = 0;			// inferior endpoint of search window
	int r = imaxCDF;		// superior endpoint of search window
	int q;				// mean value between current endpoints
	int index;			// the index we are searching for
	int end = 0;			// flag for completed search

	while (p<=r && !end) {
		q=(p+r)/2;
		if (binomialCumDistr[q]>rand && binomialCumDistr[q-1]<=rand) {
			index = q;
			end = 1;
		} else if (binomialCumDistr[q]>rand)
			r = q-1;
		else
			p = q+1;
	}

	return index;
}

void Gnp::printLogFact() {
	std::ofstream logFactorial;

	logFactorial.open("Output_Prints\\logFactorial.txt",
		std::ofstream::out | std::ofstream::trunc);

	for (int i=0; i<n+1; i++)
		logFactorial << logFact[i] << std::endl;

	logFactorial.close();
}

void Gnp::printBinPdf() {
	std::ofstream Pdf;

	Pdf.open("Output_Prints\\Pdf.txt", std::ofstream::out | std::ofstream::trunc);

	for (int i=0; i<n; i++)
		Pdf << binomialPdf[i] << std::endl;

	Pdf.close();
}

void Gnp::printBinCDF() {
	std::ofstream CDF;

	CDF.open("Output_Prints\\CDF.txt", std::ofstream::out | std::ofstream::trunc);

	for (int i=0; i<n+1; i++)
		CDF << binomialCumDistr[i] << std::endl;

	CDF.close();
}
```

```
/*                                          */
/*     Instance fields and methods.         */
/*                                          */

/* Creates an empty G(n,p) network. Nodes have to be added one by one and, after this,
 * the creation of links has to be performed by invoking the corresponding method. */
Gnp::Gnp(): Network() {
	net_p = 0.0;
}

Gnp::Gnp(int nn, double pp): Network(nn) {
	std::map<int, Node *>::iterator it;

	net_p = pp;

	/* Creates the G(n,p) network. For every node, extracts its temporary degree according to
	 * a binomial distribution Bi(n,p/2). Then creates as many links as the extracted degree
	 * between the considered node and the others which are extracted randomly. Every following
	 * node can create new links with already considered nodes (and then incrementing its degree)
	 * as long as there isn't an existing link between the two already. After this procedure
	 * is reiterated for all nodes, the network is ready. */
	binomialDistr(net_n, net_p/2);
	for (it=Nodes.begin(); it!=Nodes.end(); it++) {
		int binDeg = randomBinDegree();
		for (int i=0; i<binDeg; i++) {
			int nodeID = extractNeighborID(it->second);
			it->second->addNeighbor(nodeID);
			Nodes[nodeID]->addNeighbor(it->first);
		}

		std::cout << "Link generation: " << (it->first + 1.0)*100/net_n << "%" << std::endl;
	}

}

Gnp::~Gnp() {

}

/* Creates the links of the G(n,p) network in the same way as above. To be invoked after all the
 * Nodes have been added to the Network. */
void Gnp::createLinks(double pp) {
	std::map<int, Node *>::iterator it;

	net_p = pp;

	binomialDistr(net_n, net_p/2);
	for (it=Nodes.begin(); it!=Nodes.end(); it++) {
		int binDeg = randomBinDegree();
		for (int i=0; i<binDeg; i++) {
			int nodeID = extractNeighborIDGeneralized(it->second);
			it->second->addNeighbor(nodeID);
			Nodes[nodeID]->addNeighbor(it->first);
		}
	}
}
```

```
/* Extracts a Node's net_ID. If this net_ID is equal to the net_ID of the considered Node
 * or a Node which is already linked to it, increment the net_ID until this condition
 * changes. */
int Gnp::extractNeighborID(Node* node) {
	int nodeID = (extendedRand()%net_n);
	while ((nodeID = node->nextAvailableNeighborID(nodeID))>=net_n)
		nodeID = nodeID%net_n;

	return nodeID;
}

/* Generalized version of the above method for which the Nodes' net_IDs can be also non-contiguous.
 * Extracts the net_ID of a Node belonging to this Network that is going to become a neighbor
 * of Node node. If the extracted Node is already linked to Node node or is the same Node,
 * pick the following one in the map Nodes until this condition changes. */
int Gnp::extractNeighborIDGeneralized(Node* node) {
	std::map<int, Node *>::iterator it;

	int rand = extendedRand()%net_n;
	it = Nodes.begin();
	for (int i=0; i<rand; i++)
		it++;

	while (node->isNeighbor(it->first) || node->getNet_ID()==it->first) {
		it++;
		if (it==Nodes.end())
			it = Nodes.begin();
	}

	return it->first;
}
```

```
/*
 * Gnp.h
 *
 *  Created on: 23 dic 2017
 *      Author: Fabio GENNARI
 */

#include <fstream>
#include <map>
#include "Network.h"
#include "Node.h"

#ifndef GNP_H_
#define GNP_H_

class Gnp: public Network {
public:
	static double uniRandZeroToOne();
	static int extendedRand();
	static void binomialDistr(int nn, double pp);
	static int randomBinDegree();		// extracts a random degree
						// according to current distribution
	static void freeHeap();			// always invoke this method
						// at the end to free memory
	static void printLogFact();
	static void printBinPdf();
	static void printBinCDF();

	Gnp();
	Gnp(int nn, double pp);
	virtual ~Gnp();

	void createLinks(double pp);

protected:
	double net_p;				// CARE: net_p could be different
						// from static p (net_p = 2p)

private:
	static int n;				// the current static parameters
						// for the current bin. distribution
	static double p;			// CARE: static p could be different
						// from net_p (p = net_p/2)
	static int imaxCDF;			// index for last element (=1) of bin CDF

	static double *logFact;			// vector for factorials up to n
	static double *binomialPdf;		// vector for binomial pdf
	static double *binomialCumDistr;	// the vector of the binomial distribution Bi(n,p)

	static double logBinCoeff(int n, int m);
	static int binSearch(double rand);

	int extractNeighborID(Node* node);	// Extracts the net_ID of a valid candidate
						// neighbor for Node node
	int extractNeighborIDGeneralized(Node* node);
};

#endif /* GNP_H_ */
```

```
/*
 * Gnps.cpp
 *
 *  Created on: 16 mar 2018
 *      Author: Fabio GENNARI
 */

#include <math.h>
#include <stdlib.h>
#include <iostream>
#include <map>
#include "Network.h"
#include "Gnps.h"

/*                                      */
/*     Instance fields and methods.     */
/*                                      */

Gnps::Gnps(int nn, double pp, double ss, Gnp *net): Network(nn) {
	std::map<int, Node *>::iterator it;

	net_p = pp;
	net_s = ss;
	motherGnp = net;

	/* Sampling probability of a link net_s comes from two sampling iterations of the same link
	 * performed by inspecting the neighborhood of the considered adjacent nodes. The resulting
	 * "unidirectional" sampling probability x is defined as the following. */
	double x = 1 - sqrt(1-net_s);

	/* For each node, sample the starting neighborhood with probability x. For each successful
	 * sample create the corresponding bidirectional link.
	 * Note that the net_IDs of the Nodes in the new Network are different w.r.t. the old one.
	 * The sampling operation must be performed considering the true_IDs of the Nodes, which
	 * truly identify them in every Network. */
	for (it=Nodes.begin(); it!=Nodes.end(); it++) {
		int nodeTrueID = it->second->getTrue_ID();

		int *startingNeighborhood =
			motherGnp->Nodes[motherGnp->True_IDToNet_ID[nodeTrueID]]->getNeighbors();
		int startingDegree =
			motherGnp->Nodes[motherGnp->True_IDToNet_ID[nodeTrueID]]->getDegree();

		for (int i=0; i<startingDegree; i++) {
			double rand = Gnp::uniRandZeroToOne();
			if (rand<x) {
				int neighborNewNetID = True_IDToNet_ID[motherGnp->
					Nodes[startingNeighborhood[i]]->getTrue_ID()];
				it->second->addNeighbor(neighborNewNetID);
				Nodes[neighborNewNetID]->addNeighbor(it->first);
			}
		}

		free(startingNeighborhood);

		std::cout << "Link generation: " << (it->first + 1.0)*100/net_n << "%" << std::endl;
	}
}

Gnps::~Gnps() {

}
```

```
/*
 * Gnps.h
 *
 *  Created on: 16 mar 2018
 *      Author: Fabio GENNARI
 */

#include "Network.h"
#include "Gnp.h"

#ifndef GNPS_H_
#define GNPS_H_

class Gnps: public Network {
public:
	Gnps(int nn, double pp, double ss, Gnp *net);
	virtual ~Gnps();

protected:
	double net_p;
	double net_s;
	Gnp *motherGnp;

private:

};

#endif /* GNPS_H_ */
```

```
/*
 * Macros_PGM.h
 *
 *  Created on: 18 mar 2018
 *      Author: Fabio GENNARI
 */

#define CHECKVAL      0
#define NOCHECKVAL    1
```

```
/*
 * Matching.cpp
 *
 *  Created on: 18 mar 2018
 *      Author: Fabio GENNARI
 */

#include <map>
#include "Matching.h"
#include "Network.h"

Matching::Matching(Network *net1, Network *net2) {
	firstNetwork = net1;
	secondNetwork = net2;
	size = 0;
}

Matching::~Matching() {
	std::map<int, NodePair *>::iterator it;

	for (it=matchingNodePairs.begin(); it!=matchingNodePairs.end(); it++)
		delete it->second;
	matchingNodePairs.clear();

	firstNodesIDs.clear();
	secondNodesIDs.clear();
}

Network *Matching::getFirstNetwork() {
	return firstNetwork;
}

Network *Matching::getSecondNetwork() {
	return secondNetwork;
}

int Matching::getSize() {
	return size;
}

/* If called with CHECKVAL macro, this method adds matchedPair to the Matching only if it is not
 * a conflicting pair. By checking for conflict, it also prevents the adding of duplicates.
 * The check can be avoided with NOCHECKVAL macro. */
void Matching::addMatching(NodePair *matchedPair, int performCheck) {
	if (performCheck==CHECKVAL) {
		if (checkNonConflict(matchedPair)) {
			matchingNodePairs[size] = matchedPair;
			size++;
			/* Updates the two lists of net_IDs for nonConflict check. */
			firstNodesIDs.insert(matchedPair->getFirst()->getNet_ID());
			secondNodesIDs.insert(matchedPair->getSecond()->getNet_ID());
		}
	} else if (performCheck==NOCHECKVAL) {
		matchingNodePairs[size] = matchedPair;
		size++;
		/* Updates the two lists of net_IDs for nonConflict check. */
		firstNodesIDs.insert(matchedPair->getFirst()->getNet_ID());
		secondNodesIDs.insert(matchedPair->getSecond()->getNet_ID());
	}
}
```

```
/* Checks if NodePair pair is conflicting w.r.t. node pairs already added to the
 * Matching. By doing this, it also prevents the adding of duplicates. */
bool Matching::checkNonConflict(NodePair *pair) {
	bool nonConflict;

	if (!checkFirstNodeID(pair) && !checkSecondNodeID(pair))
		nonConflict = true;
	else
		nonConflict = false;

	return nonConflict;
}

/* Checks if the first node's net_ID of NodePair pair is already present in the set. */
bool Matching::checkFirstNodeID(NodePair *pair) {
	bool check;

	int firstID = pair->getFirst()->getNet_ID();

	if(firstNodesIDs.find(firstID)!=firstNodesIDs.end())
		check = true;
	else
		check = false;

	return check;
}

/* Checks if the second node's net_ID of NodePair pair is already present in the set. */
bool Matching::checkSecondNodeID(NodePair *pair) {
	bool check;

	int secondID = pair->getSecond()->getNet_ID();

	if(secondNodesIDs.find(secondID)!=secondNodesIDs.end())
		check = true;
	else
		check = false;

	return check;
}

/* Equivalent methods, but simply accepting the Nodes' net_IDs as parameters. */
bool Matching::checkNonConflict(int firstNodeID, int secondNodeID) {
	bool nonConflict;

	if (!checkFirstNodeID(firstNodeID) && !checkSecondNodeID(secondNodeID))
		nonConflict = true;
	else
		nonConflict = false;

	return nonConflict;
}

bool Matching::checkFirstNodeID(int ID) {
	bool check;

	if(firstNodesIDs.find(ID)!=firstNodesIDs.end())
		check = true;
	else
		check = false;

	return check;
}
```

```
bool Matching::checkSecondNodeID(int ID) {
	bool check;

	if(secondNodesIDs.find(ID)!=secondNodesIDs.end())
		check = true;
	else
		check = false;

	return check;
}

void Matching::printMatches() {
	std::map<int, NodePair *>::iterator it;
	std::ofstream PGM_matches;

	PGM_matches.open("Output_Prints\\PGM_matches.txt",
		std::ofstream::out | std::ofstream::trunc);

	for (it=matchingNodePairs.begin(); it!=matchingNodePairs.end(); it++) {
		int firstNodeNetID = it->second->getFirst()->getNet_ID();
		int firstNodeTrueID = it->second->getFirst()->getTrue_ID();
		int secondNodeNetID = it->second->getSecond()->getNet_ID();
		int secondNodeTrueID = it->second->getSecond()->getTrue_ID();
		PGM_matches << firstNodeNetID << " (" << firstNodeTrueID << "), " <<
			secondNodeNetID << " (" << secondNodeTrueID << ")" << std::endl;
	}

	PGM_matches.close();
}

void Matching::printPerformanceMetrics(std::ofstream *ofs) {
	*ofs << "Total number of performed matches: \t" << getSize() << "\t";
	*ofs << "Fraction of wrong matches: \t" << getFractionOfWrongMatches() << "\t";
}

int Matching::getNumberOfPerformedMatches() {
	return getSize();
}

int Matching::getNumberOfGoodMatches() {
	std::map<int, NodePair *>::iterator it;
	int goodMatches = 0;

	for (it=matchingNodePairs.begin(); it!=matchingNodePairs.end(); it++)
		if (it->second->goodPair())
			goodMatches++;

	return goodMatches;
}

double Matching::getFractionOfWrongMatches() {
	std::map<int, NodePair *>::iterator it;
	int wrongMatches = 0;
	double fraction;

	for (it=matchingNodePairs.begin(); it!=matchingNodePairs.end(); it++)
		if (!it->second->goodPair())
			wrongMatches++;

	fraction = (double)wrongMatches/getSize();

	return fraction;
}
```

```
/*
 * Matching.h
 *
 *  Created on: 18 mar 2018
 *      Author: Fabio GENNARI
 */

#ifndef MATCHING_H_
#define MATCHING_H_

#include <map>
#include <set>
#include "Network.h"
#include "NodePair.h"
#include "Macros_PGM.h"

class Matching {
public:
	std::map<int, NodePair *> matchingNodePairs;	// the set of obtained matchings

	Matching(Network *net1, Network *net2);
	virtual ~Matching();

	Network *getFirstNetwork();
	Network *getSecondNetwork();
	int getSize();

	void addMatching(NodePair *matchedPair, int performCheck);
	bool checkNonConflict(NodePair *pair);
	bool checkFirstNodeID(NodePair *pair);
	bool checkSecondNodeID(NodePair *pair);
	bool checkNonConflict(int firstNodeID, int secondNodeID);
	bool checkFirstNodeID(int ID);
	bool checkSecondNodeID(int ID);

	void printMatches();
	void printPerformanceMetrics(std::ofstream *ofs);

	int getNumberOfPerformedMatches();
	int getNumberOfGoodMatches();
	double getFractionOfWrongMatches();

private:
	Network *firstNetwork;			// the two networks to which the matched
	Network *secondNetwork;			// pairs of nodes belong
	int size;				// the number of obtained matchings

	std::set<int> firstNodesIDs;		// list of net_IDs of net1 nodes that
						// have been already matched
	std::set<int> secondNodesIDs;		// list of net_IDs of net2 nodes that
						// have been already matched
};

#endif /* NODE_H_ */
```

```
/*
 * Network.cpp
 *
 *  Created on: 15 mar 2018
 *      Author: Fabio GENNARI
 */

#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
#include <math.h>
#include <limits.h>
#include <iostream>
#include "Network.h"
#include "Gnp.h"
#include "Node.h"

/*                                      */
/*     Instance fields and methods.     */
/*                                      */

/* Creates an empty Network. */
Network::Network() {
	net_n = 0;
}

/* This Network constructor is in charge of creating the Node objects, whereas the constructor of
 * the specific derived class is in charge of creating the links among the various Nodes.
 * NOTE that in every Network object every element key must always be equal to the
 * corresponding Node's net_ID. This is granted by the Network constructor.
 * The only way to get one Node's true_ID should be to directly point to it and use getTrue_ID().
 * True_IDToNet_ID should only be used if really necessary, e.g. for check purposes
 * or Gnps construction. */
Network::Network(int nn) {
	std::map<int, Node *>::iterator it;

	net_n = nn;

	/* Creates the Node objects of the generic Network and adds them to the map. */
	for (int i=0; i<net_n; i++) {
		int rand = Gnp::extendedRand()%net_n;
		while (Nodes.find(rand)!=Nodes.end())
			rand = (rand+1)%net_n;
		Nodes[rand] = new Node(rand, i);
		True_IDToNet_ID[i] = rand;
		std::cout << "Node generation: " << (i + 1.0)*100/net_n << "%" << std::endl;
	}
}

Network::~Network() {
	for (int i=0; i<net_n; i++)
		delete Nodes[i];
	Nodes.clear();

	True_IDToNet_ID.clear();
}

int Network::getNet_n() {
	return net_n;
}
```

```
/* Adds node to the Network. If any of the Node's two IDs has already been used, does nothing. */
void Network::addNode(Node *node) {
	if (Nodes.find(node->getNet_ID())==Nodes.end() &&
		True_IDToNet_ID.find(node->getTrue_ID())==True_IDToNet_ID.end()) {

		Nodes[node->getNet_ID()] = node;
		True_IDToNet_ID[node->getTrue_ID()] = node->getNet_ID();
		net_n++;
	}
}

void Network::printDegreeDistr() {
	std::ofstream degDistr;

	degDistr.open("Output_Prints\\degDistr.txt", std::ofstream::out | std::ofstream::trunc);

	for (int i=0; i<net_n; i++)
		degDistr << Nodes[i]->getDegree() << std::endl;

	degDistr.close();
}
```

```
/*
 * Network.h
 *
 *  Created on: 15 mar 2018
 *      Author: Fabio GENNARI
 */

#include <fstream>
#include <map>
#include "Node.h"

#ifndef NETWORK_H_
#define NETWORK_H_

class Network {
public:
	std::map<int, Node *> Nodes;
	std::map<int, int> True_IDToNet_ID;

	Network();
	Network(int nn);
	virtual ~Network();

	int getNet_n();
	void addNode(Node *node);

	void printDegreeDistr();

protected:
	int net_n;				// the number of nodes of this network instance
};

#endif /* NETWORK_H_ */
```

```
/*
 * Node.cpp
 *
 *  Created on: 01 gen 2018
 *      Author: Fabio GENNARI
 */

#include <set>
#include <iostream>
#include <stdlib.h>
#include "Node.h"
#include "Gnp.h"

/*                                          */
/*      Instance fields and methods.        */
/*                                          */

Node::Node(int networkID, int trueID) {
        net_ID = networkID;
        true_ID = trueID;
        degree = 0;
}

Node::~Node() {
        neighbors.clear();
}

int Node::getNet_ID() {
        return net_ID;
}

int Node::getTrue_ID() {
        return true_ID;
}

int Node::getDegree() {
        return degree;
}

/* Returns an array of the net_IDs of this Node's neighbors. */
int *Node::getNeighbors() {
        int *list;

        list = (int *)malloc(degree*sizeof(int));
        int i = 0;
        for (it=neighbors.begin(); it!=neighbors.end(); it++) {
                list[i] = *it;
                i++;
        }

        return list;
}

void Node::printNeighbors() {
        for (it=neighbors.begin(); it!=neighbors.end(); it++)
                std::cout << *it << std::endl;
}
```

```
/* Adds a neighbor with nodeID as net_ID to the Node. The net_ID must be different from
 * the Node's one, otherwise the neighbor is not added. I.e. a Node can't be a neighbor
 * of itself. */
void Node::addNeighbor(int nodeID) {
	std::pair<std::set<int>::iterator,bool> ret;

	if (nodeID!=net_ID) {
		ret = neighbors.insert(nodeID);
		if (ret.second)
			degree++;
	}
}

/* CARE. This method removes only the outgoing direction of the link. If the original link is
 * bidirectional, it doesn't remove the other direction. */
void Node::removeNeighbor(int nodeID) {
	if (neighbors.erase(nodeID))
		degree--;
}

/* CARE. This method removes only the outgoing direction of the link. If the original link is
 * bidirectional, it doesn't remove the other direction. */
void Node::removeNeighborsWithProb(double x) {
	int *neighborhood = getNeighbors();
	int n = getDegree();

	for (int i=0; i<n; i++) {
		double rand = Gnp::uniRandZeroToOne();
		if (rand<x)
			removeNeighbor(neighborhood[i]);
	}

	free(neighborhood);
}

/* Checks if Node with nodeID as net_ID is a neighbor of this Node. */
bool Node::isNeighbor(int nodeID) {
	bool thereIs = false;

	it = neighbors.find(nodeID);
	if (it!=neighbors.end())
		thereIs = true;

	return thereIs;
}
```

```
/* Starting from a given candidate neighbor net_ID, returns the same net_ID if available,
 * or the next available one if not. The returned net_ID is granted to be different from
 * this node's net_ID. */
int Node::nextAvailableNeighborID(int nodeID) {
	int availableID = nodeID;

	if (availableID==net_ID)
		availableID++;
	it = neighbors.find(availableID);
	if (it!=neighbors.end()) {
		availableID++;
		it++;
		if (availableID==net_ID)
			availableID++;
		while (availableID==*it) {
			availableID++;
			it++;
			if (availableID==net_ID)
				availableID++;
		}
	}

	return availableID;
}
```

```
/*
 * Node.h
 *
 *  Created on: 01 gen 2018
 *      Author: Fabio GENNARI
 */

#ifndef NODE_H_
#define NODE_H_

#include <set>

class Node {
public:
	Node(int networkID, int trueID);
	virtual ~Node();

	int getNet_ID();
	int getTrue_ID();
	int getDegree();
	int *getNeighbors();
	void printNeighbors();
	void addNeighbor(int nodeID);
	void removeNeighbor(int nodeID);                // CARE! link removal in the outgoing
	                                                // direction ONLY
	void removeNeighborsWithProb(double x);         // CARE! link removal in the outgoing
	                                                // direction ONLY
	bool isNeighbor(int nodeID);
	int nextAvailableNeighborID(int nodeID);

private:
	std::set<int> neighbors;                        // list of neighbors' net_IDs TOWARDS
	                                                // which the node is pointing
	std::set<int>::iterator it;                     // iterator for scanning
	                                                // the list of neighbors


	int net_ID;                                     // the ID that the node has in the network
	int true_ID;                                    // the true ID of the node
	int degree;
};

#endif /* NODE_H_ */
```

```
/*
 * NodePair.cpp
 *
 *  Created on: 18 mar 2018
 *      Author: Fabio GENNARI
 */

#include "NodePair.h"

NodePair::NodePair(Node *node1, Node *node2) {
	first = node1;
	second = node2;
	marks = 0;
	metric = -1;
	metricWeight = -1;
}

NodePair::~NodePair() {

}

Node *NodePair::getFirst() {
	return first;
}

Node *NodePair::getSecond() {
	return second;
}

int NodePair::getMarks() {
	return marks;
}

void NodePair::incrementMarks() {
	marks++;
}

/* Returns metric value if weight w is equal to the current metricWeight value,
 * -1 otherwise. Weight w must be a real number in [0,1]. */
double NodePair::getMetric(double w) {
	double returnValue = -1;

	if (w==metricWeight)
		returnValue = metric;

	return returnValue;
}

/* Sets metric and metricWeight values. */
void NodePair::setMetric(double m, double w) {
	metric = m;
	metricWeight = w;
}

bool NodePair::goodPair() {
	bool a;
	if (first->getTrue_ID()==second->getTrue_ID())
		a = true;
	else
		a = false;

	return a;
}
```

```
/*
 * NodePair.h
 *
 *  Created on: 18 mar 2018
 *      Author: Fabio GENNARI
 */

#ifndef NODEPAIR_H_
#define NODEPAIR_H_

#include "Node.h"

class NodePair {
public:
	NodePair(Node *node1, Node *node2);
	virtual ~NodePair();

	Node *getFirst();
	Node *getSecond();

	int getMarks();
	void incrementMarks();

	double getMetric(double w);
	void setMetric(double m, double w);

	bool goodPair();

private:
	Node *first;
	Node *second;

	int marks;			// PGM mark counter
	double metric;			// metric value
	double metricWeight;		// weight with which the metric value is computed
};

#endif /* NODEPAIR_H_ */
```

```
/*
 * PGM.cpp
 *
 *  Created on: 18 mar 2018
 *      Author: Fabio GENNARI
 */

#include <iostream>
#include <set>
#include <math.h>
#include "Gnp.h"
#include "PGM.h"

/*                                  */
/*     Static fields and methods.   */
/*                                  */

Matching *PGM::PGMMatching = NULL;
int PGM::markThreshold = 0;
double PGM::metricWeight = -1;
std::map<int, std::map<int, NodePair *>> PGM::candidates;
std::map<int, std::map<int, NodePair *>> PGM::thresholdCandidates;

/* Prepares a seed set of size size from which the PGM algorithm can start. This seed set is
 * built by picking Nodes from the intersection Network uniformly at random. The intersection
 * Network contains all the Nodes that belong to both net1 and net2 (same true_ID) and must be
 * provided. In this way, net1 and net2 Networks, on which we are trying to perform the matching,
 * can also have different sets of Nodes. Intersection Nodes' net_IDs can also be non contiguous. */
Matching *PGM::getRandomSeedSet(Network *net1, Network *net2, Network *intersection, int size) {
	std::map<int, Node *>::iterator it;

	Matching *seedSet = new Matching(net1, net2);

	/* The number of nodes that belong to both networks. */
	int n = intersection->getNet_n();

	for (int i=0; i<size; i++) {
		int rand = Gnp::extendedRand();
		rand %= n;

		it = intersection->Nodes.begin();
		for (int i=0; i<rand; i++)
			it++;

		/* If the extracted Node has already been added to the seedSet, pick the next
		 * available one. Since the seed set is always made of good pairs only, the check
		 * can be performed on only one of the two sets of net_IDs of Matching seedSet. */
		while (seedSet->checkFirstNodeID(net1->True_IDToNet_ID[it->second->getTrue_ID()])) {
			it++;
			if (it==intersection->Nodes.end())
				it = intersection->Nodes.begin();
		}

		int extractedNodeTrueID = it->second->getTrue_ID();
		Node *firstNetNode = net1->Nodes[net1->True_IDToNet_ID[extractedNodeTrueID]];
		Node *secondNetNode = net2->Nodes[net2->True_IDToNet_ID[extractedNodeTrueID]];

		seedSet->addMatching(new NodePair(firstNetNode, secondNetNode), NOCHECKVAL);

		std::cout << "Seed set generation: " << (i + 1.0)*100/size << "%" << std::endl;
	}
```

```
	return seedSet;
}

/* The classical arithmetic PGM. */
Matching *PGM::arithmeticPGM(Matching *seedSet, int threshold) {
	PGMMatching = seedSet;
	markThreshold = threshold;

	std::set<int> z;		// the set containing the index values of the NodePairs in
					// PGMMatching that still need to add their marks

	/* Initialization of set z. In the beginning, all NodePairs of the seedSet
	 * still need to be considered. */
	for (int i=0; i<PGMMatching->getSize(); i++)
		z.insert(i);

	/* The PGM algorithm stops when there is no new NodePair that still needs to add its mark to
	 * its neighbor pairs. This happens when the next NodePair key i is equal to the current size
	 * of the Matching, i. e. no new NodePair was added to the Matching during last iteration. */
	for (int i=0; i<PGMMatching->getSize(); i++) {
		/* Randomly selects an index value among those of the NodePairs that still need to be
		 * considered. */
		int rand = Gnp::extendedRand()%z.size();
		std::set<int>::iterator it = z.begin();
		for (int j=0; j<rand; j++)
			it++;
		int index = *it;

		/* The selected NodePair adds its mark to all its neighbor pairs. */
		addNodePairMarks(PGMMatching->matchingNodePairs[index]);

		/* After the selected NodePair has added its marks, it must be removed from set z. */
		z.erase(index);

		/* Keep scanning candidates searching for the next valid match as long as possible.
		 * When no other matches can be selected, a new iteration can begin.
		 * Every time a new match is added to PGMMatching, the corresponding index must be
		 * inserted in z. */
		while (addNextMatch()) {
			z.insert(PGMMatching->getSize()-1);
			std::cout << "PGM algorithm: " <<
				PGMMatching->getSize()*100.0/fmin(PGMMatching->
				getFirstNetwork()->getNet_n(), PGMMatching->
				getSecondNetwork()->getNet_n()) << "%" << std::endl;
		}
	}

	/* Clear candidates map allowing for future method calls. */
	clearCandidates();

	return PGMMatching;
}
```

```
/* The Deferred matching variant of the PGM algorithm. */
Matching *PGM::deferredMatchingPGM(Matching *seedSet, int threshold) {
	PGMMatching = seedSet;
	markThreshold = threshold;

	int i = 0;
	/* The PGM algorithm stops when there is no new NodePair that still needs to add its mark to
	 * its neighbor pairs. This happens when the next NodePair key i is equal to the current size
	 * of the Matching, i. e. no new NodePair was added to the Matching during last iteration. */
	while (i<PGMMatching->getSize()) {
		int end = PGMMatching->getSize();
		/* Scan all NodePairs of Matching that still need to be considered and add their
		 * marks. It scans up to the current last NodePair in the Matching.
		 * Any NodePair that will be added to the Matching will be considered and scanned
		 * in the next iteration. */
		for (; i<end; i++)
			addNodePairMarks(PGMMatching->matchingNodePairs[i]);
		/* Search for a candidate whose mark counter is maximal and at least threshold.
		 * If found, it is added to PGMMatching, then a new iteration can begin. */
		addMaximalMatch();
		std::cout << "PGM algorithm: " <<
			PGMMatching->getSize()*100.0/fmin(PGMMatching->getFirstNetwork()->getNet_n(),
			PGMMatching->getSecondNetwork()->getNet_n()) << "%" << std::endl;
	}

	/* Clear candidates map allowing for future method calls. */
	clearCandidates();

	return PGMMatching;
}

/* Personal variant of the PGM algorithm. It follows the same strategy as the Deferred
 * Matching Variant, but searches for a candidate pair whose mark counter is at least
 * threshold and whose metric value is minimal. See computeMetric(NodePair *pair) method
 * for the meaning of the parameter weight. */
Matching *PGM::personalVariantPGM(Matching *seedSet, int threshold, double weight) {
	PGMMatching = seedSet;
	markThreshold = threshold;
	metricWeight = weight;

	int i = 0;
	/* The PGM algorithm stops when there is no new NodePair that still needs to add its
	 * mark to its neighbor pairs. This happens when the next NodePair key i is equal to the
	 * current size of the Matching, i. e. no new NodePair was added to the Matching during
	 * last iteration. */
	while (i<PGMMatching->getSize()) {
		int end = PGMMatching->getSize();
		/* Scan all NodePairs of Matching that still need to be considered and add their
		 * marks. It scans up to the current last NodePair in the Matching.
		 * Any NodePair that will be added to the Matching will be considered and scanned
		 * in the next iteration. */
		for (; i<end; i++)
			addNodePairMarks(PGMMatching->matchingNodePairs[i]);
		/* Search for a candidate whose mark counter is at least threshold and
		 * whose metric value is minimal. If found, it is added to PGMMatching,
		 * then a new iteration can begin. */
		addMinimalMetricMatch();
		std::cout << "PGM algorithm: " <<
			PGMMatching->getSize()*100.0/fmin(PGMMatching->getFirstNetwork()->getNet_n(),
			PGMMatching->getSecondNetwork()->getNet_n()) << "%" << std::endl;
	}
```

```
	/* Clear candidates map allowing for future method calls. */
	clearCandidates();

	return PGMMatching;
}

/* Scans all the neighbor pairs of NodePair pair and adds a mark to each of them if they
 * are still valid candidates (i.e. if they are non conflicting w.r.t. the PGMMatching
 * NodePairs). If the neighbor pair is already present in candidates, it means it's still valid
 * (every time a new match is added to PGMMatching, all the conflicting pairs are immediately
 * got rid of). If it is not present but is a valid candidate, it is added to candidates.
 * If it isn't a valid candidate, the method does nothing since that neighbor pair will never
 * be added to PGMMatching no matter what. Every time that the mark of a NodePair is incremented,
 * it checks if the mark has reached or exceeded markThreshold. If so, the NodePair is added
 * to thresholdCandidates which must always be a subset of candidates. */
void PGM::addNodePairMarks(NodePair *pair) {
	int *firstNodeNeighbors = pair->getFirst()->getNeighbors();
	int *secondNodeNeighbors = pair->getSecond()->getNeighbors();

	int firstNodeDegree = pair->getFirst()->getDegree();
	int secondNodeDegree = pair->getSecond()->getDegree();

	/* For all the neighbor pairs of NodePair pair. To scan them all it's sufficient to
	 * consider all the possible pairs of neighbors of NodePair pair's single nodes. */
	for (int i=0; i<firstNodeDegree; i++) {
		for (int j=0; j<secondNodeDegree; j++) {
			int firstNodeNetID = firstNodeNeighbors[i];
			int secondNodeNetID = secondNodeNeighbors[j];

			/* If the neighbor pair is already present in candidates,
			 * it means it's still valid. */
			if (findCandidate(firstNodeNetID, secondNodeNetID)) {
				candidates[firstNodeNetID][secondNodeNetID]->incrementMarks();
				if (candidates[firstNodeNetID][secondNodeNetID]->
					getMarks()>=markThreshold)

					thresholdCandidates[firstNodeNetID][secondNodeNetID] =
						candidates[firstNodeNetID][secondNodeNetID];
			}
			/* If it is not present but is a valid candidate, it is added to candidates.
			 * Otherwise, nothing is done. */
			else if (PGMMatching->checkNonConflict(firstNodeNetID, secondNodeNetID)) {
				NodePair *newCandidate = new NodePair(PGMMatching->getFirstNetwork()->
					Nodes[firstNodeNetID], PGMMatching->getSecondNetwork()->
					Nodes[secondNodeNetID]);
				newCandidate->incrementMarks();
				candidates[firstNodeNetID][secondNodeNetID] = newCandidate;
				if (candidates[firstNodeNetID][secondNodeNetID]->
					getMarks()>=markThreshold)

					thresholdCandidates[firstNodeNetID][secondNodeNetID] =
						candidates[firstNodeNetID][secondNodeNetID];
			}
		}
	}

	/* Clear the dynamically allocated arrays. */
	free(firstNodeNeighbors);
	free(secondNodeNeighbors);
}
```

```
/* Returns true if NodePair (firstNodeNetID, secondNodeNetID) is already present in candidates,
 * false otherwise. */
bool PGM::findCandidate(int firstNodeNetID, int secondNodeNetID) {
	bool present = false;

	std::map<int, std::map<int, NodePair *>>::iterator it;
	it = candidates.find(firstNodeNetID);
	if (it!=candidates.end())
		if (it->second.find(secondNodeNetID)!=it->second.end())
			present = true;

	return present;
}

/* This method scans thresholdCandidates in order to select the next match. ThresholdCandidates
 * is always a subset of candidates, therefore it always contains only valid pairs. This means
 * that, if it contains pairs, these pairs don't need to be tested in order to be added as
 * matches (one at a time of course, since, after adding a new match, the new conflicting pairs
 * have to be removed first). If the method finds a pair, it adds the match to PGMMatching
 * (validity check is not performed for what we said earlier). After adding the match,
 * candidates and thresholdCandidates are updated (the candidate that has been chosen is removed
 * and so are all the conflicting pairs w.r.t. it) and the method returns true.
 * If a match could not be found (i.e. thresholdCandidates is empty), the method returns false. */
bool PGM::addNextMatch() {
	std::map<int, std::map<int, NodePair *>>::iterator firstIt;
	std::map<int, NodePair *>::iterator secondIt;

	bool foundNextMatch = false;

	/* thresholdCandidates is scanned sequentially, i.e. the first one that is found
	 * is selected and becomes the next match. */
	if (thresholdCandidates.size()>0) {
		firstIt=thresholdCandidates.begin();
		secondIt=firstIt->second.begin();

		PGMMatching->addMatching(secondIt->second, NOCHECKVAL);
		updateCandidates(firstIt->first, secondIt->first);
		foundNextMatch = true;
	}

	return foundNextMatch;
}

/* This method performs the same operations as addNextMatch(). The only difference is that
 * it scans thresholdCandidates in order to select the candidate pair with the
 * maximal mark counter as next match to be added to PGMMatching. */
bool PGM::addMaximalMatch() {
	std::map<int, std::map<int, NodePair *>>::iterator firstIt;
	std::map<int, NodePair *>::iterator secondIt;

	std::map<int, std::map<int, NodePair *>>::iterator maximalFirstIt;
	std::map<int, NodePair *>::iterator maximalSecondIt;
	int maximalMarkCounter;

	bool foundNextMatch = false;

	/* If there is at least a candidate pair whose mark counter is at least threshold,
	 * performs an approximate search for the pair with the maximal mark counter. */
	if (thresholdCandidates.size()>0) {
		firstIt = thresholdCandidates.begin();
		secondIt = firstIt->second.begin();

		maximalFirstIt = firstIt;
```

```
                maximalSecondIt = secondIt;
                maximalMarkCounter = secondIt->second->getMarks();

                for (; firstIt!=thresholdCandidates.end(); firstIt++) {
                        secondIt = firstIt->second.begin();
                        if (secondIt->second->getMarks()>maximalMarkCounter) {
                                maximalFirstIt = firstIt;
                                maximalSecondIt = secondIt;
                                maximalMarkCounter = secondIt->second->getMarks();
                        }
                }

                PGMMatching->addMatching(maximalSecondIt->second, NOCHECKVAL);
                updateCandidates(maximalFirstIt->first, maximalSecondIt->first);
                foundNextMatch = true;
        }

        return foundNextMatch;
}

/* This method performs the same operations as addMaximalMatch(). The only difference is
 * that it scans thresholdCandidates in order to select the candidate pair with the
 * minimal metric value, computed with weight w, as next match to be added to PGMMatching. */
bool PGM::addMinimalMetricMatch() {
        std::map<int, std::map<int, NodePair *>>::iterator firstIt;
        std::map<int, NodePair *>::iterator secondIt;

        std::map<int, std::map<int, NodePair *>>::iterator minimalMetricFirstIt;
        std::map<int, NodePair *>::iterator minimalMetricSecondIt;
        double minimalMetricValue;

        bool foundNextMatch = false;

        /* If there is at least a candidate pair whose mark counter is at least threshold,
         * performs an approximate search for the pair with the minimal metric value,
         * computed with weight w. */
        if (thresholdCandidates.size()>0) {
                firstIt = thresholdCandidates.begin();
                secondIt = firstIt->second.begin();

                minimalMetricFirstIt = firstIt;
                minimalMetricSecondIt = secondIt;
                minimalMetricValue = computeMetric(secondIt->second);

                for (; firstIt!=thresholdCandidates.end(); firstIt++) {
                        secondIt = firstIt->second.begin();
                        if (computeMetric(secondIt->second)<minimalMetricValue) {
                                minimalMetricFirstIt = firstIt;
                                minimalMetricSecondIt = secondIt;
                                minimalMetricValue = computeMetric(secondIt->second);
                        }
                }

                PGMMatching->addMatching(minimalMetricSecondIt->second, NOCHECKVAL);
                updateCandidates(minimalMetricFirstIt->first, minimalMetricSecondIt->first);
                foundNextMatch = true;
        }

        return foundNextMatch;
}
```

```
/* Retrieves pair's metric value. If it is not available yet, computes it and saves it
 * for future uses. */
double PGM::computeMetric(NodePair *pair) {
	double metricValue = pair->getMetric(metricWeight);

	/* If pair's metric value is not available yet, computes it and saves it for future uses. */
	if (metricValue==-1) {
		Network *firstNet = PGMMatching->getFirstNetwork();
		Network *secondNet = PGMMatching->getSecondNetwork();

		/* Computes the absolute value of the difference btwn the degrees of pair's nodes. */
		int firstNodeDegree = pair->getFirst()->getDegree();
		int secondNodeDegree = pair->getSecond()->getDegree();
		double degreeDiff = fabs(firstNodeDegree-secondNodeDegree);

		/* Computes the average degree of the first node's neighbors. */
		double firstNeighborsAvgDegree = 0;
		int *firstNeighborhood = pair->getFirst()->getNeighbors();
		for (int i=0; i<firstNodeDegree; i++)
			firstNeighborsAvgDegree += firstNet->Nodes[firstNeighborhood[i]]->getDegree();
		firstNeighborsAvgDegree /= firstNodeDegree;

		/* Computes the average degree of the second node's neighbors. */
		double secondNeighborsAvgDegree = 0;
		int *secondNeighborhood = pair->getSecond()->getNeighbors();
		for (int i=0; i<secondNodeDegree; i++)
			secondNeighborsAvgDegree += secondNet->
				Nodes[secondNeighborhood[i]]->getDegree();
		secondNeighborsAvgDegree /= secondNodeDegree;

		/* Computes the absolute value of the difference between
		 * the two neighbors' average degrees. */
		double neighborsAvgDegreeDiff =
			fabs(firstNeighborsAvgDegree-secondNeighborsAvgDegree);
		free(firstNeighborhood);
		free(secondNeighborhood);

		/* Computes the final metric value of the pair, which is a linear combination
		 * of the two absolute values. Saves it for future uses. */
		metricValue = (1-metricWeight)*degreeDiff + metricWeight*neighborsAvgDegreeDiff;
		pair->setMetric(metricValue, metricWeight);
	}

	return metricValue;
}

/* To be called after adding the corresponding match to PGMMatching. This method removes
 * the NodePair (firstNodeNetID, secondNodeNetID), that has just been added to PGMMatching, from
 * candidates and thresholdCandidates, without deleting it as an Object. It then removes all pairs
 * that are conflicting w.r.t. it from the two maps, deleting them too (they will never be able to
 * be added to PGMMatching, no matter what, because they are conflicting). */
void PGM::updateCandidates(int firstNodeNetID, int secondNodeNetID) {
	std::map<int, std::map<int, NodePair *>>::iterator firstIt;
	std::map<int, NodePair *>::iterator secondIt;

	/* Removes the pair from candidates and thresholdCandidates without deleting it. */
	candidates[firstNodeNetID].erase(secondNodeNetID);
	thresholdCandidates[firstNodeNetID].erase(secondNodeNetID);
```

```
	/* Removes and deletes pairs that are conflicting in terms of their first node's net_ID.
	 * Deleting the pairs contained in candidates is sufficient, since thresholdCandidates is
	 * always a subset of it. */
	for (secondIt=candidates[firstNodeNetID].begin();
		secondIt!=candidates[firstNodeNetID].end(); secondIt++)

		delete secondIt->second;

	candidates[firstNodeNetID].clear();
	candidates.erase(firstNodeNetID);

	thresholdCandidates[firstNodeNetID].clear();
	thresholdCandidates.erase(firstNodeNetID);

	/* Removes and deletes pairs that are conflicting in terms of their second node's net_ID.
	 * Deleting the pairs contained in candidates is sufficient, since thresholdCandidates is
	 * always a subset of it. */
	for (firstIt=candidates.begin(); firstIt!=candidates.end(); firstIt++) {
		secondIt = firstIt->second.find(secondNodeNetID);
		if (secondIt!=firstIt->second.end()) {
			delete secondIt->second;
			firstIt->second.erase(secondIt);
		}
	}

	for (firstIt=thresholdCandidates.begin(); firstIt!=thresholdCandidates.end(); firstIt++) {
		secondIt = firstIt->second.find(secondNodeNetID);
		if (secondIt!=firstIt->second.end())
			firstIt->second.erase(secondIt);
	}

	/* For how thresholdCandidates works, it must never contain empty rows. Therefore, after
	 * removing the conflicting pairs from it, the possible empty rows that are left must be
	 * erased. */
	bool compacted = false;
	while (!compacted) {
		compacted = true;

		bool foundEmptyRow = false;
		for (firstIt=thresholdCandidates.begin();
			firstIt!=thresholdCandidates.end() && !foundEmptyRow; firstIt++)

			if (firstIt->second.size()==0) {
				thresholdCandidates.erase(firstIt);
				foundEmptyRow = true;
				compacted = false;
			}
	}
}
```

```
/* Clears candidates and thresholdCandidates, allowing for new PGM calls. */
void PGM::clearCandidates() {
	std::map<int, std::map<int, NodePair *>>::iterator firstIt;
	std::map<int, NodePair *>::iterator secondIt;

	for (firstIt=candidates.begin(); firstIt!=candidates.end(); firstIt++) {
		for (secondIt=firstIt->second.begin(); secondIt!=firstIt->second.end(); secondIt++)
			delete secondIt->second;
		firstIt->second.clear();
	}

	candidates.clear();
	thresholdCandidates.clear();
}
```

```
/*
 * PGM.h
 *
 *  Created on: 18 mar 2018
 *      Author: Fabio GENNARI
 */

#include "Matching.h"
#include "Macros_PGM.h"

#ifndef PGM_H_
#define PGM_H_

class PGM {
public:
	static Matching *getRandomSeedSet(Network *net1,
		Network *net2, Network *intersection, int size);
	static Matching *arithmeticPGM(Matching *seedSet, int threshold);
	static Matching *deferredMatchingPGM(Matching *seedSet, int threshold);
	static Matching *personalVariantPGM(Matching *seedSet, int threshold, double weight);

private:
	static Matching *PGMMatching;
	static int markThreshold;
	static double metricWeight;
	static std::map<int, std::map<int, NodePair *>> candidates;
	static std::map<int, std::map<int, NodePair *>> thresholdCandidates;

	static void addNodePairMarks(NodePair *pair);
	static bool findCandidate(int firstNodeNetID, int secondNodeNetID);
	static bool addNextMatch();
	static bool addMaximalMatch();
	static bool addMinimalMetricMatch();
	static double computeMetric(NodePair *pair);
	static void updateCandidates(int firstNodeNetID, int secondNodeNetID);
	static void clearCandidates();
};

#endif /* PGM_H_ */
```