A Genetic Algorithm for Large Graph Partitioning Problem

Xuan-Tung Nguyen
Software Engineering
Hanoi University of Science and Technology
Ha Noi, Vietnam
tungxbk@gmail.com

Phuong-Nam Cao
Software Engineering
Hanoi University of Science and Technology
Ha Noi, Vietnam
nam.cp170219@sis.hust.edu.vn

Van-Quyet Nguyen
Information Technology
Hung Yen University Technology and Education
Hung Yen, Vietnam
quyetict@utehy.edu.vn

Kyungbaek Kim
Electronics and Computer Engineering
Chonnam National University
Gwangju, Korea
kyungbaekkim@jnu.ac.kr

Quyet-Thang Huynh†
Software Engineering
Hanoi University of Science and Technology
Ha Noi, Vietnam
thanghq@soict.hust.edu.vn

ABSTRACT

This paper considers the problem of partitioning large graphs. We propose a genetic algorithm to distribute large-scale graphs for parallel computations including individual representation, fitness function, and genetic operators. We first propose individual representation with the ability to change the number of clusters of the graph as well as the ability to flexibly change the cluster of each node. We then increase the diversity of the population by a random hybrid algorithm. Finally, we optimize the solution with the heuristic hybrid and the heuristic mutation algorithms. To validate our approach, we test the proposed algorithm and compare the results with other algorithms such as Greedy algorithm and Bulk Swap algorithm on both synthetic and real-world datasets. Experimental results show that our algorithm outperforms other ones in the aspect of the number of cross edges reduction among graph partitions.

CCS CONCEPTS

• Computing methodologies→ Heuristic • Applied computing→ Graph Partition;

KEYWORDS

Graph Partitioning, Large-scale Graph, Parallel Computation, Genetic Algorithm

ACM Reference format:

Xuan-Tung Nguyen, Phuong-Nam Cao, Van-Quyet Nguyen, Kyungbaek Kim and Quyet-Thang Huynh. 2019. A Genetic Algorithm for Large Graph Partitioning Problem. In *SoICT '19: The Tenth International Symposium on Information and Communication Technology, December 4–6, 2019, Hanoi – Ha Long Bay, Viet Nam.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3368926.3369724

SoICT 2019, December 4–6, 2019, Hanoi - Ha Long Bay, Viet Nam

ACM ISBN 978-1-4503-7245-9/19/12...$15.00
https://doi.org/10.1145/3368926.3369724
† Corresponding Author

1 Introduction

Nowadays, petabytes of data can be modeled as a graph with nodes and edges are generated and processed. For example, data of a social network can be modeled as a large graph with users as nodes, and relationships as edges; likewise, websites as nodes, hyperlink as edges. Such data are large and complex. They have a lot of real calculation problems related to large graphs which takes a number of vertices and edges, such as web graphs and social networks. The sizes of the graphs, in some cases, are billions of vertices, trillions of edges which leads to challenges to processing performance [1]. Therefore, these graphs are needed to process and storage with multi-pieces and each piece has a small number of cross edges between different pieces [2]. It is also the reason for defining the large graph partitioning problem (LGPP).

LGPP problem is defined as the following: given an undirected graph G (V, E) such that V is a set of nodes and E is a set of edges of the graph G. The goal of the problem is to divide the graph's vertices into several sets such that the number of cross edges between different sets is small (called sum_weight_cutedges). By this way, we can put each set into a processor and these processors can run side by side. The more processors, the higher the speed and performance.

Besides, the problem of graph partitioning is a NP-complete problem [3], so all known algorithms for creating partitions merely return approximations for the optimal solution. Although, in this theory, the achievement of good results is limited, over the years, there are many partition algorithms developed to achieve good results in a short time. The fundamental issues which must be addressed in every parallel application are the workload distribution, data distribution, and computation on a processor. The optimal distribution minimizes the overall running time, usually by

ensuring that each processor has the same amount of work, and minimizes parallel costs due to distribution caused by the interaction between processors (defined by the cross edges between two processors).

In this paper, we propose a genetic algorithm (GA) to distribute large-scale graphs for parallel computations including individual representation, fitness function, and genetic operators. We divide the graph G with the number of edges and vertices large enough into k sets so that the total number of cross edges is minimal. To do this, we first propose individual representation with the ability to change the number of clusters of the graph as well as the ability to flexibly change the cluster of each node. We then increase the diversity of the population by a random hybrid algorithm. Finally, we optimize the solution with the heuristic hybrid and the heuristic mutation algorithms. We evaluate the proposed algorithm and compare the results with other algorithms such as Greedy algorithm (DG) and Bulk Swap algorithm (BS) on both synthetic and real-world datasets. Experimental results show that our algorithm outperforms other ones in the aspect of the number of cross edges reduction among graph partitions.

We organize the paper as follows. In Section 2, we first introduce the graph partitioning algorithms that existed earlier and then evaluated these algorithms, eventually introducing (GA). We present the graph partitioning algorithm based on a genetic algorithm in Section 3. The results of GA, DG, BS made to evaluate the performance of our algorithms are presented in Section 4. After reviewing the results, we draw conclusions in Section 5.

2 Related Works

In recent years, more and more graph data are generated from different sources, such as social networks and websites [4]. Such data can be formed as large graphs, which requires optimization techniques for storing and processing these data [5]. There have been many researches focusing on large graph data partitioning problems.

Charbel Farhat [6] et al. studied and proposed the greedy algorithm, which is quite effective in theory. If the data sets tend to be fragmented, then selecting the vertices and then browsing in-depth (width) will produce good results. For example, the DG algorithm selects the next vertex in a cluster with the optimal function is the minimalism of the cutting edge and the connecting edge in the region, but the problem is that the original vertex is selected for approval. has a great impact on the results, especially when it is necessary to divide the set of data into multiple clusters, must select multiple times.

George Karypis and Vipin Kumar propose t*he greedy graph growing algorithm [7] (*GGP), which is a heuristic algorithm for greedy graph based structure fragmentation. The algorithm gives k-loops, each loop selects an unobtrusive free peak and generates a partition around it according to the depth search algorithm (DFS) until n / k vertices are included in that partition.

Chris Walshaw proposed Iterated Multilevel Algorithms [12]. The main idea of Iterated Multilevel Algorithms is to device the graph into 2 parts and balance them. The algorithm will repeat and stop when the graph is divided into k-wav given. But, this algorithm will be locally optimized. Therefore, Peter Sanders and Christian Schulz propose Global Search [11] for graph partition to solve locally optimized in [12]. This algorithm using two recursive calls using different random seeds during contraction and local search on each level, and they are very effective. However, the algorithms are very costly which makes them not scale on large graphs well.

In 2016, Tefeng Chen and Bo Li propose Bulk Swap (BW) algorithm [8], the advantage of this algorithm is that it is possible to quickly reduce the number of cutting edges (or the total weight of cutting edges) between two data clusters, combining simulated annealing technique for escaping local optimizations. But it only cares for two data clusters to be selected for optimization, maybe the next optimal time will cancel the results that have just been done. If a region consists of a small number of vertices, the BS algorithm is almost useless.

The large graph data has a clear data structure with vertices and edges but it still has randomness so this paper proposes a random algorithm that is GA. Although partitioning is a problem of NP-complete, GA algorithm can check a large number of cases, in addition to the crossover, it is possible to find the best generation between two parents, giving results tend to be better while still having randomness with a mutation to avoid local optimization. The shortcoming is that it needs a long time to execute.

3 Proposed Algorithm

In this section, we propose GALGPP algorithm to solve the problem. As mentioned in Section 1, this algorithm is a GA, which is a heuristic algorithm and its improvements applied in k-means graph partitioning problem.

Besides, we use a greedy algorithm [3] (DG algorithm) to mutate individual with purpose find the best result.

3.1 The Greedy Algorithm (DG)

The DG selects random k nodes in graph G and sets the index of k partitions for k nodes. Next, the remaining free vertices are divided into pieces by heuristic algorithms such as 3.2.1, 3.2.2, 3.2.3.

The total weight increases of a vertex when partitioning for that vertex (a common function in a greedy structure) is defined:

$$t_P\ (v, p') = \sum_{p \neq p'} g_p(v, p) \quad (1)$$

This is the number of sheer edges that increase when assigning vertex v to partition p' of partition p. This function is important because the selected vertex v is free and when it is assigned to p', the cut edges will be obtained and counted to the total cutting edge of the partition. The right side is the sum of the edges from v to the other partitions p'.

The pseudo-code below represents our idea:

Algorithm 1: Greedy Algorithm

Input: Graph G = (V, E), k

Output: k partitions

```
0 Program Greedy Algorithm
1 Begin
2      P←{P_0, …, P_{k-1} }
3      V'←V
4      for p∈[0,k-1] do
5         u←select_random_node∈V'
6         p_P←{u}
7          V'←V'\{u}
8       end
8       p←0
9       while |V'|>0 do
10        b←greedyfunction (V', P,p,G)
11        p_P←{u}
12        V'←V'\{b}
13        p←(p+1) mod k
14      return P;
15 end
```

3.2 The Greedy Function

3.2.1 Graph Growing Partitioning Algorithm

The algorithm produces k loops, each loop chooses a random free vertex and generates a partition around it according to the depth search algorithm (DFS) until the top n/k vertices are put into that partition.

3.2.2 Greedy Graph Growing Partitioning Algorithm

This algorithm selects vertices to produce the smallest number of edges, and if more than one vertex satisfies, one of them is randomly chosen. Besides, the vertices are partitioned in the circular order of the pieces, while the GGP grows in size from one piece to the maximum in a loop.

The pseudocode below represents our idea:

Algorithm 2: Greedy Graph Growing Partitioning Algorithm

Input: V'ϵV &V' ∉ $P\{1..k\}$, P (current partition), P'(next partition), Graph G = (V, E)

Output: node ϵ V'

```
0 Program GGGP Algorithm
1 Begin
2   m ← min_{v∈V}(t_p(v,p))
3   C ← {v ∈ V'|t_p(v,p) = m}
4   node = select_random(C)
5   return node
6 end
```

3.2.3 Min-max Greedy (MMG) Algorithm

Battiti and Bertossi [13] found that the GGGP's greedy function produces many cases where the increased cut edge weights are the same for real regions. The MMG algorithm mimics the GGGP but adds an improvement to solve the above problem: among the edges with the smallest rising edge weight, selecting the vertex makes the edge weight in one piece increase the maximum or the maximum chemistry $g_P(v,p)$ with p is the fragment that v is partitioned into.

The pseudocode below represents our idea:

Algorithm 3: MMG Algorithm

Input: V'ϵV &V' ∉ $P\{1..k\}$, P (current partition), P' (next partition), Graph G = (V, E)

Output: node ϵ V'

```
0 Program MMG Algorithm
1 Begin
2   m ← min_{v∈V}(t_p(v,p))
3   C ← {v ∈ V'|t_p(v,p) = m}
5   m ← max_{v∈C}(g_p(v,p))
6   C ← {v ∈ C|g_p(v,p) = m}
4   node = select_random(C)
5   return node
6 end
```

3.3 The Genetic Algorithm (GA)

In GA, a population of candidate solutions (called individuals, we propose the structure of the individual as shown in the presentation 3.4.) to an optimization problem is evolved toward better solutions. The individuals are randomly generated, then they are hybridized and mutated. Finally, we select individuals for the next life cycle. The algorithm ends when the result is focused at one point.

3.4 Individual Representation

In our algorithm, each individual, which is a solution of the problem has a chromosome, this is represented by a string of numbers from 1 to k and the length of it is the number of vertices in the graph. When vertex i is assigned to partition j, unit i-th in the string is set to number j. The chromosome generated in this way can store one form of a distributed graph in which if the unit i-th in the string is j, the vertex i in the graph is assigned to partition j.

We initial assign every vertex to a partition uniformly at random to form k partitions.

The evolution usually starts from a population of randomly generated individuals, and it is an iterative process, with the population in each iteration called a generation. In each generation, the fitness of every individual in the population is evaluated; the fitness is usually the value of the objective function in the optimization problem being solved. The more fit individuals are stochastically selected from the current population, and each individual's genome is modified (recombined and possibly randomly mutated) to form a new generation. The new generation of candidate solutions is then used in the next iteration of the algorithm. Typically, the algorithm terminates when either a maximum number of generations has been produced, or a satisfactory fitness level has been reached for the population.

To initialize an individual, each gene is assigned to any partition in the required k-way partition.

The pseudocode below represents our idea:

Algorithm 4: Init individual

Input: Graph G = (V, E), k

Output: individual

```
0 Program Inialization GALGPP
1 Begin
2  load_data_for_individual
2  for i from 0 to length_of_gens
3    partition_index = random(k)
4    gens[i].partion_index = partition_index
5 end
```

3.5 Fitness Function

The fitness function of one population is measured by:

$$\frac{1}{2}\sum_{i=1}^{n}\sum_{j\in adj_i} m_{ij} \quad (2)$$

where adj_i is the set of adjacent vertices of vertex i and m_{ij} is the weight of the edge connecting vertex i and vertex j.

3.6 Genetic Operators

3.6.1 Crossover with a heuristic selection

From 2 individuals we called them as parents. We select n first gens. Next, we find cutting point *i*, so that the new individual has *i* gene belonging to parent 1, *n-i* gene belongs to parent 2 and the *sum_weight_cutedges* (the number of cross edges between different sets) is smaller than that of the parents. By the way, the number of nodes of the input graph can be large, so we decide to choose max *n_positions* random positions and process them. The proportion of crossover implementing in our population is *rate_crossover* percent.

The pseudocode below represents our idea:

Algorithm 5: Crossover with heuristic select

Input: 2 parents

Output: new individual

```
0 Program Crossover GALGPP
1 Begin
2 Initialization
3 num_node=min(number_of_nodes, n_positions)
4 create new_gen
5 min = Infinity
6 best_state = 0
7 for mask from 0 to num_node-1 do
8   new_gen[:mask] = gen_1[:mask]
9   new_gen[mask+1:] = gen_gen2[mask+1:]
10  cutsize = sum_weight_cutedges in new_gen
11  if cutsize < min then
12    best_state = mask
13    min = cutsize
14  fi
15 end
16 new_gen[:best_state] = gen_1[:best_state]
17 new_gen[best_state+1:] = gen_2[best_state+1:]
```

3.6.2 Crossover with random select

From 2 individuals we called them as parents (parent1 and parent2). We select random positions of gen in parent1 and call is *index_choose* (because the index of gen in individuals is the same). Next to, we select random gen of parent1 and parent2 at *index_choose*. As session 3.4.1, we decided to choose max 22 random positions and process them. The proportion of crossover implementing in our population is 40 percent.

The pseudocode below represents our idea:

Algorithm 6: Crossover with random select

Input: 2 parents

Output: new individual

```
0 Program Crossover GALGPP
1 Begin
2 Initialization
3 Num_node is number of nodes
4 new_gen = clone(parent1)
5 number_select = max (length_of_gen, 22)
6 for i from 0 to number_select
7   index_current = random(length_of_gen)
8   index_of_partition=random
    (parent1.partition_index,
    parent2.partition_index)
9   new_gen[index_current].partition_index =
index_of_partition
16 end
```

3.6.3 Mutation with DG algorithm select

We use a custom mutation method by choosing at least 3 positions and the max number of positions up to $\frac{number\_node}{10}$. Depend on current population state, we choose new partition for that node by calculating the total cost of cut edges and internal edges follow DG as we mentioned at Section 3.1 and 3.2 to get the best result. This mean find new partition such as weight of edges in new partition include $p\left(g_p(v, p')\right)$ subtracts the weight of cut edges $t_p(v,p')$ is minimum. This optimization is formulated by:

$$\min g_p(v,p') - t_p(v,p') \quad (3)$$

The proportion of mutation implementing in our population is 30 percent. The pseudocode below represents our idea:

Algorithm 7: Mutation

Input: 1 parent

Output: new individual

```
0 Program Mutate GALGPP
1 Begin
2 Initialization
3 choosed_gen is a gen chosen for mutation
4 num_node is the number of nodes
5 create new_gen
6 new_gen ← choosed_gen
7 num ← max of 3 and num_node/10
8 V ← include random num nodes
9 for i is choosed nodes in V do
10  Choose cluster for i such as weight of
    edges include i in new the cluster
    subtracts weight of edges in a different
    cluster
11  new_gen[i] = new cluster
12 end
```

4 Experimental Evaluation

4.1 Datasets

In order to test the performance of our algorithm and compare with other algorithms, we used 4 datasets with different sizes.

The first dataset (Synthetic 1) is a small graph that is generated randomly with 9 nodes and 40 edges. The target is to divide graph into two partitions. The main purpose of this dataset is comparing the result of our GA algorithm implement with backtracking algorithm in a limited time.

The second dataset (Synthetic 2) is a larger random graph with 999 nodes and 2430 edges. The target of this data set is to divide the graph into 428 partitions. It means each partition will have from 2 to 3 nodes.

The third dataset is a real-world graph representing a social circle provided by Facebook [9]. It is a social network with 4039 nodes and 88234 edges. We divide into 10 networks. By default, we set the cost of an edge is 1.

The fourth dataset (Arxiv GR-QC) is from Standford called General Relativity and Quantum Cosmology collaboration network [10]. Arxiv GR-QC (General Relativity and Quantum Cosmology) collaboration network is from the e-print arXiv and covers scientific collaborations between authors' papers submitted to General Relativity and Quantum Cosmology category. If an author i co-authored a paper with author j, the graph contains an undirected edge from i to j. If the paper is co-authored by k authors this generates a completely connected (sub)graph on k nodes.

4.2 Evaluation Settings

In our algorithm, the population size is 50. The individuals are initialized randomly. The crossover rate in each generation is 80% and mutation rate is 30%. Our system is run 20 times for each problem instance. The programs were run on a machine with Intel CoreI5 2.80Ghz, RAM 8G DDR3, SSD 240G and were installed by C++ language.

4.2 Experimental Results

Table 1 shows the best costs found by GALGPP and other algorithms on four datasets. These problem instances could be solved by GALGPP with the result is the best in 4 algorithms.

Table 1: Comparison the best result of GALGPP algorithm with DG algorithm, GGP algorithm, and BS algorithm

Data	DG	GGP	BS	GALGPP
Synthetic 1	1401	1078	307	51
Synthetic 2	19344	228679	116115	65671
Facebook	20404	86534	79230	7635
Arxiv GR-QC	1395426	1707292	722613	475404

From the results in Table 1, we found that the results of the GALGPP are almost always superior to the others.

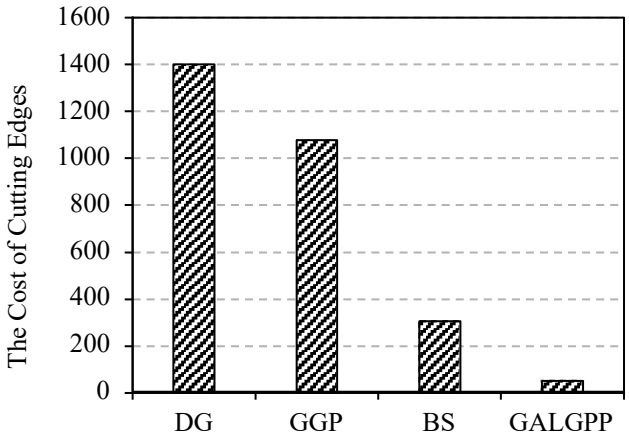

Figure 1: Comparison the best result of GALGPP algorithm with DG algorithm, GGP algorithm and BS algorithm on Synthetic 1.

From the results showing on four figures we found that with dataset random (Synthetic 2), the DG algorithm got the best result in 4 algorithms, but this algorithm has the worst result with dataset random Synthetic 1. But for real-world datasets (Facebook, Arxiv GR-QC), the GALGPP algorithm has the best result.

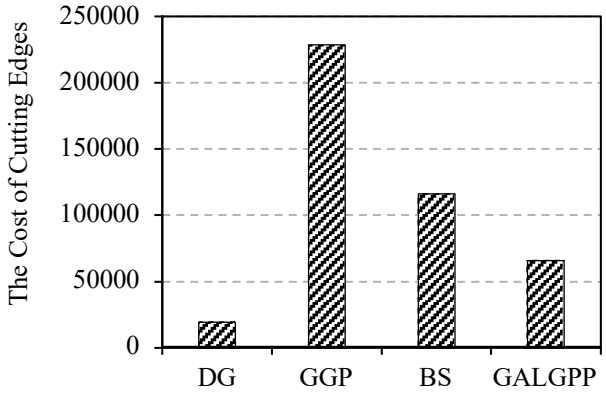

Figure 2: Comparison the best result of GALGPP algorithm with DG algorithm, GGP algorithm and BS algorithm on Synthetic 2.

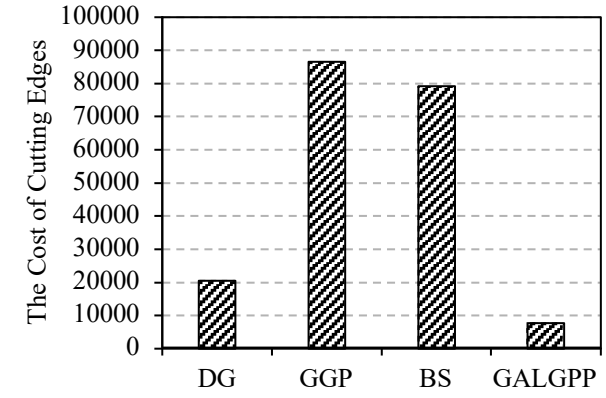

Figure 3: Comparison the best result of GALGPP algorithm with DG algorithm, GGP algorithm and BS algorithm on Facebook.

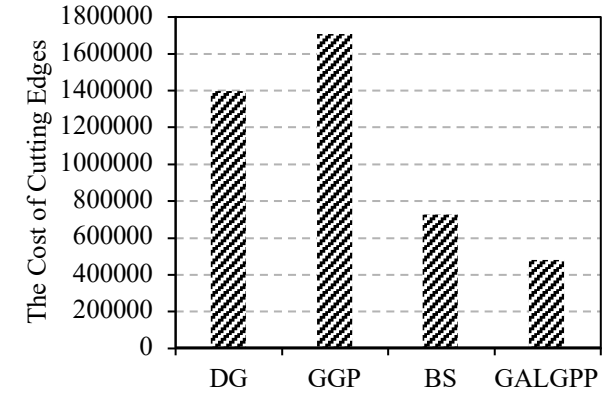

Figure 4: Comparison the best result of GALGPP algorithm with DG algorithm, GGP algorithm and BS algorithm on Arxiv GR-QC.

5 Conclusion

In this paper, we proposed a genetic algorithm for solving the Large Graph Partition Problem called GALGPP. We experimented on four datasets. Experimental results indicated that our algorithm is effective about costs. In the future, we are defining this problem with multi-objective and solving it. Besides, we will study solving the problem for big graphs which has billion of nodes and edges by using the proposed approaches as well as different ones.

ACKNOWLEDGMENTS

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning (NRF-2017R1A2B4012559).

REFERENCES

[1] Nguyen-Van, Q., Tung, L.D. and Hu, Z., 2013, December. Minimizing data transfers for regular reachability queries on distributed graphs. In Proceedings of the Fourth Symposium on Information and Communication Technology (pp. 325-334). ACM.

[2] Rahimian, Fatemeh, et al. "Ja-be-ja: A distributed algorithm for balanced graph partitioning." 2013 IEEE 7th International Conference on Self-Adaptive and Self-Organizing Systems. IEEE, 2013.

[3] Garey, Michael R., David S. Johnson, and Larry Stockmeyer. "Some simplified NP-complete problems." Proceedings of the sixth annual ACM symposium on Theory of computing. ACM, 1974

[4] Nguyen, V.Q. and Kim, K., 2017, December. Estimating the evaluation cost of regular path queries on large graphs. In Proceedings of the Eighth International Symposium on Information and Communication Technology (pp. 92-99). ACM.

[5] V.-Q. Nguyen, Q.-T. Huynh, and K. Kim, "Estimating searching cost of regular path queries on large graphs by exploiting unit-subqueries," Journal of Heuristics, pp. 1–21, Nov 2018. [Online]. Available: https://doi.org/10.1007/s10732-018-9402-0

[6] Charbel Farhat. A simple and efficient automatic FEM domain decomposer. Computers and Structures, 28(5):579–602, 1988

[7] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. to appear in SIAM Journal on Scientific Computing.

[8] Chen, Tefeng, and Bo Li. "A distributed graph partitioning algorithm for processing large graphs." 2016 IEEE Symposium on Service-Oriented System Engineering (SOSE). IEEE, 2016.

[9] J. McAuley and J. Leskovec, 'Social circles: Facebook', NIPS 2012, [Online] Available: http://snap.stanford.edu/data/ego-Facebook.html

[10] J. Leskovec, J. Kleinberg and C. Faloutsosm, 'General Relativity and Quantum Cosmology collaboration network', ACM TKDD 2007, [Online] Available: https://snap.stanford.edu/data/ca-GrQc.html

[11] Sanders, Peter, and Christian Schulz. "Engineering multilevel graph partitioning algorithms." European Symposium on Algorithms. Springer, Berlin, Heidelberg, 2011.

[12] Walshaw, Chris. "Multilevel refinement for combinatorial optimisation problems." Annals of Operations Research 131.1-4 (2004): 325-372.

[13] Battiti, Roberto, and Alan A. Bertossi. "Greedy, prohibition, and reactive heuristics for graph partitioning." IEEE Transactions on Computers 48.4 (1999): 361-385.