Speeding Up BigClam Implementation on SNAP

C. H. Bryan Liu[1]
Department of Computing, Imperial College London, United Kingdom
liu.ch.bryan@gmail.com

Benjamin Paul Chamberlain
Department of Computing, Imperial College London, United Kingdom
b.chamberlain14@imperial.ac.uk

Abstract

We perform a detailed analysis of the C++ implementation of the Cluster Affiliation Model for Big Networks (BigClam) on the Stanford Network Analysis Project (SNAP). BigClam is a popular graph mining algorithm that is capable of finding overlapping communities in networks containing millions of nodes. Our analysis shows a key stage of the algorithm – determining if a node belongs to a community – dominates the runtime of the implementation, yet the computation is not parallelized. We show that by parallelizing computations across multiple threads using OpenMP we can speed up the algorithm by 5.3 times when solving large networks for communities, while preserving the integrity of the program and the result.

2012 ACM Subject Classification Computing methodologies → Concurrent algorithms

Keywords and phrases BigClam, Community Detection, Parallelization, Networks

Digital Object Identifier 10.4230/OASIcs.ICCSW.2018.1

Category Main Track

Acknowledgements The authors thank Marc P. Deisenroth for useful discussions and the anonymous reviewers for providing many improvements to the original manuscript.

1 Introduction

Networks can represent many systems including social interactions, transport systems, financial transactions, communications infrastructure and biological functions. In all cases they describe interactions (edges) between dependent entities (nodes). One of the most important and best studied fields of network science is community detection [8, 13, 14]. A community can be thought as a group of nodes having a higher density of internal than external connections [4]. Early community detection algorithms partitioned small networks into disjoint regions, assigning each node to a single community [1, 17]. Later algorithmic advances both relax the disjointness requirement (allowing overlapping communities) and scale to much larger networks. Overlapping community detection algorithms are more general than partitioning methods, which they include as special cases [3, 18, 22]. Methods that focus on scaling community detection have allowed communities to be detected in networks with millions or even billions of nodes [2, 16, 24].

The Cluster Affiliation Model for Big Networks (BigClam), proposed by Yang and Leskovec [26] is both scalable and discovers overlapping communities. Under BigClam, nodes can be in multiple communities, and affiliation weight between a node and a community is modeled as a positive continuous number. The right half of Figure 1 shows the affiliation

[1] Now at ASOS.com, London, UK

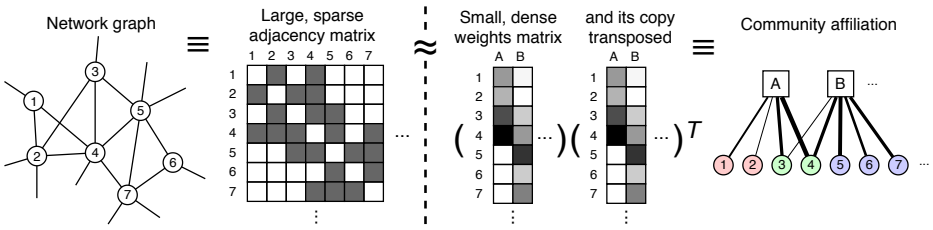


Figure 1 Illustration of community detection in a graph in terms of non-negative matrix factorization. (Left half) It is common to represent a network graph by a large, sparse adjacency matrix. (Right half) Yang and Leskovec proposed modeling community affiliations with a bipartite graph between communities and nodes, with affiliation weights represented by a small, dense, non-negative matrix [26]. (Middle two panels) By finding the most likely non-negative matrix, which when multiplied with its transpose best resembles the given adjacency matrix, we can obtain the most likely community affiliation w.r.t. the given network graph.

weights for seven nodes to two communities. This can be represented as a *bipartite graph*, or an *affiliation weights matrix*. The graph of a network is usually represented by a sparse binary adjacency matrix (left half). BigClam infers the affiliation weights matrix by applying non-negative matrix factorization [6] to the adjacency matrix. The algorithm learns the affiliation weights matrix that is best able to reconstruct the underlying adjacency matrix subject to the constraints of positivity and local optimality.

BigClam is a popular and highly cited method that features in a number of lectures and tutorials [10, 19]. The related software project [11] has attracted hundreds of GitHub stars. Due to the popularity of this model amongst both researchers and practitioners, we perform a rigorous analysis of the C++ implementation provided on the Stanford Network Analysis Project (SNAP) [11]. Our analysis of the BigClam source code reveals that the algorithm has three stages. In particular, the final Community Association (CA) stage, which makes assignments of nodes to communities, generally dominates the runtime, yet its computation is not parallelized across CPU threads. The runtime domination of CA is especially true for networks with large numbers of communities, which is common in real networks (see [9]). Not parallelizing computation where available results in lengthened runtime and wastes available hardware resources as they are put on idle.

This motivates our work in parallelizing computation in the CA stage to speed up the BigClam implementation on SNAP. Our major consideration is the parallelization must not introduce race condition on shared objects that compromise the integrity of the results. We parallelize the CA stage with OpenMP, a specification for high-level parallelism in C++ programs, and we show that the parallelization achieves as much as 5.3 times speed up and saves as much as 12.8 hours when solving networks by Leskovec and Krevl [9] using an eight-thread machine (Intel i7-4790 @ 3.60 GHz CPU).

To summarize, our contributions are as follow: (1) We profile the runtime of the BigClam implementation on SNAP in terms of its three stages. (2) We show that the CA stage dominates the runtime in current BigClam implementation on SNAP when solving networks with large numbers of communities, which is common in real networks. (3) We provide a detailed description, and the code implementation of how we parallelize computation on the CA stage, with a comprehensive discussion on avoiding race conditions. We also provide experimental results showing that the speed up is statistically significant, and preserves the result's integrity. [2]

[2] All code and experiment data are available on `https://github.com/liuchbryan/snap/tree/master/contrib/ICL-bigclam_speedup`.

Table 1 The average-case runtime complexity for the three stages of the BigClam community detection algorithm. $|V|$, $|E|$, $|C|$, r, k, t^* represents the number of nodes, edges, communities, community affiliations per node, epochs, and the speed-up multiple achieved by parallelizing computation across threads respectively. Derivations of the complexity are detailed in Appendix B.

Stage	Conductance Test	Gradient Ascent	Community Association
Complexity	$O\left(\lvert V\rvert \left(\frac{\lvert E\rvert}{\lvert V\rvert}\right)^2\right)$	$O\left(\lvert V\rvert \frac{kr}{t^*} \left(\frac{\lvert E\rvert}{\lvert V\rvert}\right)^2\right)$	$O(\lvert V\rvert\lvert C\rvert)$

Table 2 Number of nodes ($|V|$), communities ($|C|$), edges ($|E|$), and the average number of affiliations (r) recorded in the networks by Leskovec and Krevl [9].

	$\lvert V\rvert$	$\lvert C\rvert$	$\lvert E\rvert$	r
Amazon product co-purchase network	334,863	75,149	925,872	6.78
DBLP collaboration network	317,080	13,477	1,049,866	2.27
LiveJournal online social network	3,997,962	287,512	34,681,189	1.79
Youtube online social network	1,134,890	8,385	2,987,624	0.113

2 SNAP Implementation: The Bottleneck

We first examine the BigClam community detection algorithm and identify the bottleneck(s) in its implementation on SNAP. The core idea of BigClam is to find the affiliation weights matrix F that maximizes the log-likelihood function.[3] The mathematical formulation is detailed in Appendix A.

By examining its implementation on SNAP, we observe that the community detection algorithm has three stages: Conductance Test (CT), which initializes the affiliation strength matrix; Gradient Ascent (GA), which finds the optimal affiliation weights matrix; and Community Association (CA), which determines if an affiliation exists between a community and a node based on the value of affiliation weight recorded under the said matrix in relation to a pre-specified threshold.

We show the average-case runtime complexity of the three stages in Table 1. The full derivation is available in Appendix B. It can be seen that the CA stage will dominate the runtime if the number of communities is large, which we formalize as: $|C| \gg \frac{kr}{t^*}\left(\frac{|E|}{|V|}\right)^2$, where $|V|$, $|E|$, $|C|$, r, k, t^* represents the number of nodes, edges, communities, community affiliations per node, epochs, and the speed-up multiple achieved by parallelizing computation across threads respectively (see Appendix A.1).

Networks satisfying the inequality above are common. For example, all networks with ground-truth communities featured in Leskovec and Krevl [9] (shown in Table 2) satisfy the inequality when $k = 100$ and $t^* = 4$.[4] We confirm this by running the BigClam implementation on the networks shown in Table 2 using an eight-thread machine (Intel i7-4790 @ 3.60 GHz CPU), and measure the proportion of runtime spent in each of the three stages. Figure 2 shows the results of these experiments. While the time spent on the CT

[3] The $(u, c)^{\text{th}}$ entry of F represents the strength of the community affiliation between user u and community c in a network (see Figure 1 for an illustration).

[4] A conservative estimate of the speed up achieved by parallelizing the GA stage across eight threads.

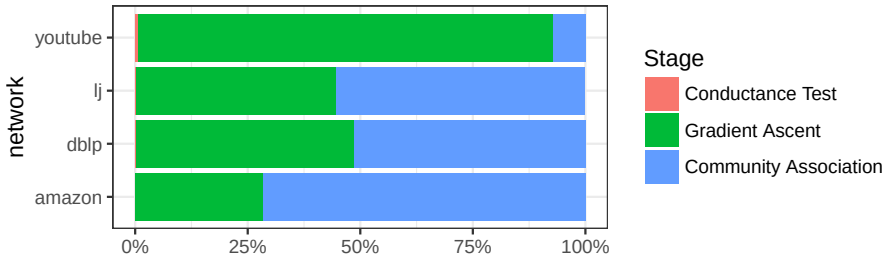


Figure 2 Average proportion of time spent on the three stages of the BigClam community detection algorithm (From left to right: Conductance Test (red), Gradient Ascent (green), and Community Association (blue)) in Leskovec and Sosič's implementation [11] for the four networks shown in Table 2. The implementation is tested on an eight-thread machine (Intel i7-4790 @ 3.60 GHz CPU), with the number of communities to detect for each network set to that recorded in Table 2.

stage is negligible, the time spent on the CA stage generally accounts for more than half of the entire runtime.[5]

Thus, the CA stage is usually the bottleneck in the algorithm. We notice that unlike the GA stage, in which computation is parallelized, the CA stage is not parallelized. Therefore, the majority of the CPU resources and man-time is wasted by idling. Parallelizing computation in the CA stage will better utilize available resources and hence improve scalability.

3 Speeding Up BigClam Via Parallel Computing

In this section we describe how to speed up BigClam with the use of OpenMP, a specification for parallel programming [15] that is supported in C++ and currently used in SNAP. The goal is to speed up the CA stage while ensuring that the input to output mapping is identical to the non-parallelized version.

3.1 Requirements in Result Correctness

As with any parallel computing application, it is important to prevent race conditions between threads from undermining the correctness of the result. In the context of speeding up the CA stage of BigClam, this means that the parallelized version must produce the same set of community affiliations as the unparallelized version given the same input F.

The theoretical representation of the communities returned by the algorithm is a set of sets: $M = \{M_1, M_2, ..., M_c\}$ where each community is itself a set $M_c = \{u_1, u_2, ...u_k\}$. Two runs of BigClam produce the same output if $M^{(1)} = M^{(2)}$. However, the BigClam SNAP implementation uses vectors of vectors (implemented as C++ STL-like objects) instead of sets of sets and enforces additional ordering that is not present theoretically. We denote the vector of vectors representation as $\mathcal{M}$. Using this representation it is possible to have $M^{(1)} = M^{(2)}$ and $\mathcal{M}^{(1)} \neq \mathcal{M}^{(2)}$ (see Figure 3).

[5] The Youtube network is an exception: we believe the average number of affiliations per node estimated by the BigClam algorithm is far greater than that recorded in the ground-truth (over 90% of the nodes do not have any community affiliations). This results in a far greater value of r than that reported in Table 2, which violates the inequality.

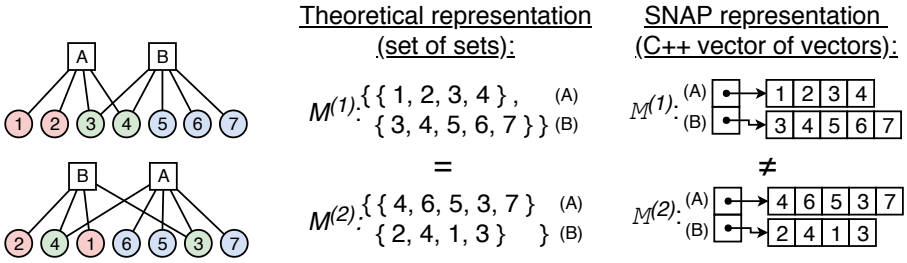


Figure 3 The two representations of the community affiliations. (Left) the underlying bipartite graph. (Middle) the theoretical representation as a set of sets. (Right) the SNAP C++ STL-like vector of vectors representation. In the theoretical representation the ordering is exchangeable, in the SNAP representation it is not.

As the ordering of values in the inner vectors and the single outer vector does not matter, we can allow race conditions between threads when performing append operations into these vectors, and thus increase the degree of parallelism in our implementation.

3.2 Methodology

Algorithm 1 outlines the existing CA stage implementation. The algorithm is simplified to include only operations related to scanning the matrix F and extracting community affiliations.[6] The algorithm is rewritten in pseudocode to enhance readability.

Algorithm 1 The existing implementation of the CA stage, simplified and rewritten in pseudocode. F is the affiliation weights matrix, and δ is the minimum affiliation strength threshold for a node to be considered a member of a community.

```
1: Initialize 𝓜 as an empty vector
2: for all c ∈ C do
3:     Initialize 𝓜_c as an empty vector
4:     for all u ∈ V do
5:         if F_uc ≥ δ then
6:             Append u to 𝓜_c
7:         end if
8:     end for
9:     Append 𝓜_c to 𝓜
10: end for
11: return 𝓜
```

As discussed in Section 3.1, we can spread the task of scanning a particular column of F over multiple threads while maintaining correctness – all node IDs will be added to the correct community vector, and with the proper synchronization mechanism (see discussion below) all community vectors will be present in the final result. In the terminology of OpenMP, we can parallelize the outer **for** loop over all communities, covering operations in lines 3–8 of Algorithm 1.

To prevent unintended race conditions while maintaining the highest level of parallelism, we declare a *critical operation* as any operation that involves objects that are shared between threads. Each critical operations is controlled by a mutex that prevents multiple threads from simultaneously writing to an object. It is safe to parallelize operations that involve only

[6] We exclude operations such as 1) sorting the vector $\left(\sum_u F_{uc}\right)_{c \in C}$ and scan columns which have a higher total affiliation strength first, and 2) excluding communities if it does not have enough members from being included, as they are less computationally expensive than scanning the matrix.

Table 3 Average time taken, in seconds, to run **a)** the community association (CA) stage **b)** the entire BigClam community detection algorithm, without and with parallelization of the community association stage. The implementations are tested on eight-thread machines with the same CPU specifications (Intel i7-4790 @ 3.60 GHz CPU).

Networks	CA stage		Overall	
	Unparallelized	Parallelized	Unparallelized	Parallelized
Amazon	1077.58	203.74	1505.38	610.23
DBLP	160.39	30.00	312.47	180.69
LiveJournal	55363.22	9233.02	100146.81	55259.48
Youtube	213.62	43.07	2965.12	2717.85

objects used by a single thread (a.k.a. private objects/variables) and read-only objects that are shared between threads. In our case, the only object that is shared between threads and involves write operations is the set of community affiliations $\mathcal{M}$. All other objects are either shared and read-only, or private to a thread.

- The affiliation weights matrix F is read-only by all threads
- The lower affiliation weight threshold δ is defined as a C++ constant (which is unmodifiable once defined), and hence is read-only
- The vector / list used to keep track of current community's members ($\mathcal{M}_c$) is local in the scope of the outer **for** loop, and hence is private to a thread according to the OpenMP specification [15].

Therefore, the only operation that needs to be declared as critical is the append to $\mathcal{M}$ in line 9 of Algorithm 1. Only one thread can append $\mathcal{M}_c$ to $\mathcal{M}$ at a time while all other operations can be parallelized.

4 Experiments

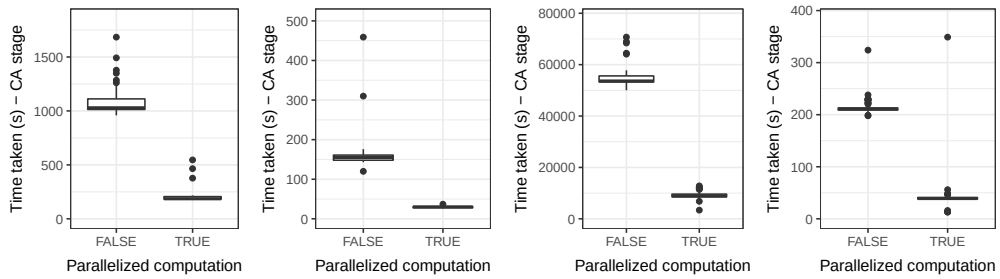
We run a number of experiments to validate the methodology described in Section 3. We show that parallelizing computation of the CA stage over multiple threads 1) reduces the runtime in the CA stage (and hence the overall BigClam implementation), and 2) retains the result correctness.

We use the datasets featured in Leskovec and Krevl [9] (see Table 2 for details of the datasets), which are widely used to benchmark the runtime of overlapping community detection algorithms [20, 22, 25, 27], including by BigClam itself [26].

4.1 Runtime Reduction

To demonstrate that parallelization reduces the algorithmic runtime, we run the unparallelized and parallelized variants of BigClam on multiple machines with Intel i7-4790 @ 3.60 GHz CPU (eight threads) for 100 epochs. Each machine runs only one of the variants at any time to ensure all CPU threads are dedicated to one variant. For each run, the program detects communities in the networks specified in Table 2, with the number of communities to detect set to that specified by the table. We measure the runtime of each stage of the BigClam for both the parallelized and unparallelized implementations across multiple runs.

The average runtime is reported in Table 3 and we perform a Welch's t-test to determine if the parallelized implementation achieves a significantly lower runtime for a) the CA stage, and b) the entire BigClam program. We visualize the results of this experiment in Figure 4.



(a) Amazon **(b)** DBLP **(c)** LiveJournal **(d)** YouTube

Figure 4 Box plot of the time taken, in seconds, to run the community association stage on different networks with ground-truth communities, without and with parallelization on the stage. The time taken with parallelization on the stage is significantly lower.

It is clear from Table 3 and Figure 4 that our implementation produces a significant runtime reduction in the CA stage for all networks shown in Table 2. With an eight-thread machine, we achieve a 5.3 times speed up in the CA stage, and subsequently a 2.5 times speed up in the overall BigClam algorithm for the Amazon product co-purchase network. For the LiveJournal network the runtime of the CA stage is reduced by 46,130 seconds (or 12.8 hours) on average.[7]

On the other hand, parallelizing the CA stage does not bring massive improvements in runtime on networks with low numbers of communities (those that do not satisfy the inequality in Section 2). We only achieve a 1.1 times speed up on the overall runtime solving the Youtube network, despite achieving a 4.96 times speed up on the CA stage. The speedup is not apparent in networks where the algorithm is dominated by the GA stage, where parallelizing the computation in the CA stage brings only marginal improvements.

4.2 Verification of Result Correctness

To confirm that parallelization of the CA stage produces the same set of community affiliation predictions as the unparallelized version we create a utility program. The program sorts the node IDs in a community and the communities in the program output in lexicographical order before comparing (strict) equality. This is necessary as common approaches to compare program outputs (e.g. `diff` or MD5 check sum) will fail even if two sets of communities are equal, as discussed in Section 3.1.

Our utility does not report any discrepancies between the program outputs produced by the parallelized and unparallelized variants, hence we conclude that our parallelization in the CA stage produces the same output, backed by a theoretical discussion in Section 3.2 and experimental verification.

5 Conclusion

In this work we profile the runtime of the BigClam implementation on SNAP, a popular overlapping community detection algorithm on an extensively used network analysis platform. We are able to split the runtime of the algorithm into three stages – the conductance test

[7] Yang and Leskovec state that, "with 20 threads, it takes about one day to fit BigClam to the LiveJournal network" [26] – we are able to fit this network with only eight threads in less than 16 hours.

(initialization) stage, the gradient ascent (optimization) stage and the community association (extraction) stage – and provide an average-case runtime complexity for each stage.

We show the community association stage is dominating the runtime in the current implementation when solving real networks, and parallelize its implementation to speed up BigClam. We show the speed up is both statistically significant and of practical utility, including a 5.3 times speed up on the community association stage (and 2.5 times overall) when solving the Amazon product co-purchase network, and saving 12.8 hours on the community association stage with an eight-thread machine. We release all relevant code and experimental data on our GitHub repository so that the research community can immediately benefit from our work and replicate our results.[8]

References

1 Reid Andersen, Fan Chung, and Kevin Lang. Local Graph Partitioning Using PageRank Vectors. In *Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science*, FOCS '06, pages 475–486, Washington, DC, USA, 2006. IEEE Computer Society. doi:10.1109/FOCS.2006.44.

2 Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment*, 2008(10):P10008, 2008.

3 T. S. Evans and R. Lambiotte. Line graphs of weighted networks for overlapping communities. *The European Physical Journal B*, 77(2):265–272, September 2010. doi:10.1140/epjb/e2010-00261-8.

4 Santo Fortunato. Community Detection in Graphs. *Physics Reports*, 486(3):75–174, 2010. doi:10.1016/j.physrep.2009.11.002.

5 David F. Gleich and C. Seshadhri. Vertex Neighborhoods, Low Conductance Cuts, and Good Seeds for Local Community Methods. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '12, pages 597–605, New York, NY, USA, 2012. ACM. doi:10.1145/2339530.2339628.

6 Patrik O Hoyer. Non-negative matrix factorization with sparseness constraints. *Journal of Machine Learning Research*, 5(Nov):1457–1469, 2004.

7 Ravi Kannan, Santosh Vempala, and Adrian Vetta. On Clusterings: Good, Bad and Spectral. *J. ACM*, 51(3):497–515, May 2004. doi:10.1145/990308.990313.

8 Zhana Kuncheva and Giovanni Montana. Community Detection in Multiplex Networks Using Locally Adaptive Random Walks. In *Proceedings of the 2015 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining 2015*, ASONAM '15, pages 1308–1315, New York, NY, USA, 2015. ACM. doi:10.1145/2808797.2808852.

9 Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

10 Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. *Mining of Massive Datasets*. Cambridge University Press, New York, NY, USA, 2nd edition, 2014.

11 Jure Leskovec and Rok Sosič. SNAP: A general-purpose network analysis and graph-mining library. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 8(1):1, 2016.

12 C. H. Bryan Liu. On overlapping community-based networks: generation, detection, and their applications. Master's thesis, Imperial College London, London, United Kingdom, June 2016.

[8] https://github.com/liuchbryan/snap/tree/master/contrib/ICL-bigclam_speedup

13 M. E. J. Newman. Detecting community structure in networks. *The European Physical Journal B*, 38(2):321–330, March 2004. doi:10.1140/epjb/e2004-00124-y.

14 M. E. J. Newman and M. Girvan. Finding and evaluating community structure in networks. *Phys. Rev. E*, 69:026113, February 2004. doi:10.1103/PhysRevE.69.026113.

15 OpenMP Architecture Review Board. OpenMP application program interface version 4.5, 2015. URL: http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf.

16 Pascal Pons and Matthieu Latapy. Computing Communities in Large Networks Using Random Walks. In *Proceedings of the 20th International Conference on Computer and Information Sciences*, ISCIS'05, pages 284–293, Berlin, Heidelberg, 2005. Springer-Verlag. doi:10.1007/11569596_31.

17 Alex Pothen. Graph Partitioning Algorithms with Applications to Scientific Computing. In David E. Keyes, Ahmed Sameh, and V. Venkatakrishnan, editors, *Parallel Numerical Algorithms*, pages 323–368, Dordrecht, 1997. Springer Netherlands. doi:10.1007/978-94-011-5412-3_12.

18 Ioannis Psorakis, Stephen Roberts, Mark Ebden, and Ben Sheldon. Overlapping community detection using Bayesian non-negative matrix factorization. *Phys. Rev. E*, 83:066114, June 2011. doi:10.1103/PhysRevE.83.066114.

19 Rik Sarkar. Community detection and cascades [Lecture]. http://www.inf.ed.ac.uk/teaching/courses/stn/files1516/slides/community-continued.pdf, 2015. Lecture slides from the course Social and Technological Networks (Autumn 2015), University of Edinburgh.

20 Meng Wang, Chaokun Wang, Jeffrey Xu Yu, and Jun Zhang. Community Detection in Social Networks: An In-depth Benchmarking Study with a Procedure-oriented Framework. *Proc. VLDB Endow.*, 8(10):998–1009, June 2015. doi:10.14778/2794367.2794370.

21 Duncan J Watts and Steven H Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440–442, 1998.

22 Joyce Jiyoung Whang, David F Gleich, and Inderjit S Dhillon. Overlapping community detection using seed set expansion. In *Proceedings of the 22nd ACM international conference on Conference on information & knowledge management*, pages 2099–2108. ACM, 2013.

23 Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. An Empirical Evaluation of In-memory Multi-version Concurrency Control. *Proc. VLDB Endow.*, 10(7):781–792, March 2017. doi:10.14778/3067421.3067427.

24 Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, and Ion Stoica. GraphX: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*, GRADES '13, pages 2:1–2:6, New York, NY, USA, 2013. ACM. doi:10.1145/2484425.2484427.

25 J. Yang and J. Leskovec. Community-Affiliation Graph Model for Overlapping Network Community Detection. In *2012 IEEE 12th International Conference on Data Mining*, pages 1170–1175, December 2012. doi:10.1109/ICDM.2012.139.

26 Jaewon Yang and Jure Leskovec. Overlapping Community Detection at Scale: A Non-negative Matrix Factorization Approach. In *Proceedings of the Sixth ACM International Conference on Web Search and Data Mining*, WSDM '13, pages 587–596, New York, NY, USA, 2013. ACM. doi:10.1145/2433396.2433471.

27 Hongyi Zhang, Irwin King, and Michael R. Lyu. Incorporating Implicit Link Preference into Overlapping Community Detection. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, AAAI'15, pages 396–402. AAAI Press, 2015. URL: http://dl.acm.org/citation.cfm?id=2887007.2887063.

A Key Formulation of BigClam Community Detection Algorithm

In this section we first introduce the nomenclature of networks, before moving on to the specifics of the BigClam community detection algorithm.

A.1 Network Preliminaries

A network is a data structure that contains a graph and a set of attributes. A graph $G(V, E)$ is composed of a set of nodes V and a set of edges $E = (v_i, v_j)$ where $v_i, v_j \in V$ that connect two nodes. The graph can be represented by an *adjacency matrix* $A \in \{0, 1\}^{|V|\times|V|}$ where A_{ij} is one if $(v_i, v_j) \in E$ and zero otherwise. Attributes can apply to either edges or nodes.

We denote the set of communities $C = \{c_1, c_2, ..., c_n\}$, where c_i indexes the i^{th} community. The set of community affiliations is defined as a set of sets $M = \{M_c : c \in C\}$, where $M_c = \{u_1, u_2, ..., u_k\}$ is a set containing the nodes affiliated to community $c \in C$.

We also denote $\mathcal{N}(u)$ as the set of neighbours of a node u in G, and the neighborhood containing u and its neighbors $N(u)$.

Part of our runtime analysis involves the average number of communities a node is affiliated with, which we formally define as:

▶ **Definition 1.** Let $D_u = \{c : u \in M_c\}$ be the set of communities that node $u \in V$ is affiliated with. Then the average number of community affiliations for all nodes r is

$$r = \frac{1}{|V|} \sum_{u\in V} |D_u| \,, \tag{1}$$

where $|D_u|$ is the number of communities that node u is affiliated with.

A.2 BigClam Community Detection Algorithm

The core idea of the BigClam community detection algorithm is to find the community affiliation weights matrix $F = (F_{uc})_{u\in V, c\in C}$, where the $(u, c)^{\text{th}}$ entry represents the strength of the community affiliation between user u and community c in a network (see Figure 1 for an illustration), that maximizes the log-likelihood function. Yang and Leskovec [26] use an iterative approach, where at each iteration they fix the affiliation weights for all but one node (say u), and perform a gradient ascent on the affiliation weights for node u. The log-likelihood for the corresponding row $\vec{F_u} = (F_{uc})_{c\in C}$ of F is specified as:

$$l(\vec{F_u}) = \sum_{v\in\mathcal{N}(u)} \log\left(1 - \exp(-\vec{F_u}\vec{F_v}^T)\right) - \sum_{v\notin\mathcal{N}(u)} \vec{F_u}\vec{F_v}^T \,. \tag{2}$$

We follow the original BigClam notation and so $\vec{F_u}\vec{F_v}^T$ is an *inner product*.

Differentiating Equation (2) w.r.t. $\vec{F_u}$ gives the gradient:

$$\nabla l(\vec{F_u}) = \sum_{v\in\mathcal{N}(u)} F_v \frac{\exp(-\vec{F_u}\vec{F_v}^T)}{1 - \exp(-\vec{F_u}\vec{F_v}^T)} - \sum_{v\notin\mathcal{N}(u)} \vec{F_v} \tag{3}$$

$$= \sum_{v\in\mathcal{N}(u)} F_v \frac{\exp(-\vec{F_u}\vec{F_v}^T)}{1 - \exp(-\vec{F_u}\vec{F_v}^T)} - \left(\sum_{v\in V} \vec{F_v} - \vec{F_u} - \sum_{v\in\mathcal{N}(u)} \vec{F_v}\right) \,. \tag{4}$$

In Equation (4) $\sum_{v\in V} \vec{F_v}$ can be precomputed, and $\sum_{v\in\mathcal{N}(u)} \vec{F_v}$ is computed on each gradient evaluation. This results in a more computationally efficient formulation as network graphs are usually sparse (i.e. $|\mathcal{N}(u)| \ll |V|$).

The BigClam community detection algorithm initializes F as:

$$F_{(u')(N(u))} = \begin{cases} 1 & \text{if } u' \in N(u) \text{ and } N(u) \text{ is a locally} \\ & \text{minimal neighborhood [5] of } u \\ 0 & \text{otherwise} \end{cases}, \tag{5}$$

where $N(u)$ represents u and its neighbours in G, and regards $u \in V$ as a member of $c \in C$ from the most likely affiliation weights matrix F if:

$$F_{uc} \geq \delta = \sqrt{-\log(1-\epsilon)}\,, \tag{6}$$

where $\epsilon = \frac{2|E|}{|V|(|V|-1)}$ is the background probability for a random edge to form in the graph.

B SNAP Implementation: A Runtime Complexity Analysis

We observe the BigClam community detection algorithm has three stages: Conductance Test, Gradient Ascent, and Community Association. Here we derive the runtime complexity for each of the three stages.

B.1 The Conductance Test Stage

The algorithm begins by testing each node to see if it belongs to a locally minimal neighborhood as defined by Gleich et al. [5]. The initial / seed communities are chosen to be the locally minimal neighborhoods.

For each node $u \in V$ we calculate the conductance of its neighborhood. The conductance of an neighbourhood $N(u)$ is the fraction of edges from nodes within $N(u)$ to nodes in the same neighborhood over that to nodes outside the neighborhood [7]. This involves traversing each neighbor $v \in \mathcal{N}(u)$ and finding out how many members of $\mathcal{N}(v)$ are not in $N(u)$. Hence there are $\sum_{u \in V} \sum_{v \in \mathcal{N}(u)} |\mathcal{N}(v)|$ operations involved.

We simplify the expression above by replacing $|\mathcal{N}(u)|\ \forall u \in V$ with the average number of neighbors, and using the fact that it is by definition the average degree of the network graph ($|E|/|V|$). This leads to an average-case complexity of $O\left(|V|\frac{|E|}{|V|}\frac{|E|}{|V|}\right)$.

B.2 Gradient Ascent Stage

After initialization, the algorithm optimizes the affiliation weights matrix F to maximize the log-likelihood function (see Equation (2)) using gradient ascent. To understand the runtime complexity of the GA stage, we first look at the two building blocks – calculating the dot product and summing $\vec{F_v}$ – and their runtime complexity in the implementation.

B.2.1 Dot Product Runtime Complexity

Calculating the dot product between two vectors $\vec{F_u}$ and $\vec{F_v}$ is required to calculate the gradient given in Equation (3). This is performed on each pairs of connected nodes in G for each epoch.

In a naïve implementation that sums the product of the corresponding (dense) vector elements, the number of operations required scales with the length of $\vec{F_u}$. Many such operations in this context are unnecessary – a node u in real networks is likely to be affiliated

with only a small number of communities,[9] leading to a large number of entries in $\vec{F_u}$ being set to zero (as u is unaffiliated to those communities).

The SNAP implementation stores $\vec{F_u}$ as a sparse vector, where only non-zero elements are recorded along with its position. Using sparse vectors, the number of operations for a dot product between F_u and F_v scales as:

$$d_{\mathrm{DP}}(u,v) \triangleq \min\left(|D_u|, |D_v|\right), \tag{7}$$

which is the minimum number of community affiliations possessed by the two nodes u and v.

B.2.2 Vector Sum Runtime Complexity

We then consider the number of elements to be traversed in each $\vec{F_v}$ when calculating the sum of affiliation weights for all neighbors of a node u, $\sum_{v\in\mathcal{N}(u)} \vec{F_v}$. The sum is featured in Equation (4) as part of the gradient calculation. Similar to the dot product calculation, implementing $\vec{F_v}$ as sparse vectors means the algorithm need not consider all $|C|$ affiliation weights in $\vec{F_v}$ but only the weights associated with communities that the neighbor nodes are a member of: $\bigcup_{v\in\mathcal{N}(u)} D_v \subseteq C$.

The cardinality of the set in the expression above is bounded above by

$$d_{\mathrm{VS}}(u) \triangleq |\mathcal{N}(u)| \max_{v\in\mathcal{N}(u)} (|D_v|), \tag{8}$$

assuming all neighbors of node u belong to disjoint sets of communities.[10]

B.2.3 Overall Runtime Complexity of the GA Stage

We can now estimate the runtime complexity of the GA stage. In this stage the algorithm iterates over the nodes multiple times, calculates the row gradient in Equation (3) and updates the affiliation weights. This is done until the convergence criteria is met, or for a pre-specified number of epochs.

Equation (3) shows that the gradient of the log-likelihood is the difference of two summations. The first summation involves calculating the vector sum and dot product over each neighbor $v \in \mathcal{N}(u)$, and the second summation involves calculating the vector sum over $v \notin \mathcal{N}(u)$. This is made more efficient by Equation (4) as real graphs are sparse and so the number of neighbors of a node is far less than the number of non-neighbors. The number of operations required in calculating the row gradient is then bounded above by:

$$\gamma \left[\sum_{v\in\mathcal{N}(u)} \left(d_{\mathrm{VS}}(u) + d_{\mathrm{DP}}(u,v)\right) + \sum_{v\in\mathcal{N}(u)} d_{\mathrm{VS}}(u) \right], \tag{9}$$

where γ is a constant multiplier.

We notice that $d_{\mathrm{VS}}(u)$ dominates $d_{\mathrm{DP}}(u,v)\ \forall v \in \mathcal{N}(u)$, and hence Expression (9) can be further simplified to $\gamma' \left[|\mathcal{N}(u)|\, d_{\mathrm{VS}}(u)\right]$, where γ' is another constant multiplier.

---

[9] Liu [12] has shown the the maximum number of affiliations for any node is 116 out of 75,149 possible communities in the Amazon product co-purchase network, and 682 out of 957,154 possible communities in the LiveJournal social network.

[10] In practice the cardinality will be much smaller due to the “small world" phenomenon: a node's neighbors are likely to be connected themselves [21], which according to BigClam is due to them being mutual members of one or more communities.

We replace $|\mathcal{N}(u)|$ by $|E|/|V|$, just as we did in Section B.1. Furthermore we approximate $d_{\text{VS}}(u)$ (see Equation (8)) in the average case by $|E|/|V| \times r$.

We have to calculate the row gradient for all $|V|$ nodes over k epochs (which we specify). Moreover, the computation of the GA stage is parallelized onto t threads using OpenMP [15], which will reduce the runtime by $t^* \leq t$ folds due to synchronization overhead [23]. We arrive at our average-case complexity of

$$O\left(\frac{k}{t^*}|V|\frac{|E|}{|V|}\frac{|E|}{|V|}r\right). \tag{10}$$

Note that r in Expression (10) is the BigClam estimate of the average number of affiliations per node, not the value realized in the ground-truth, and the two values can differ significantly.

B.3 Community Association Stage

The final stage of the algorithm takes the most likely community affiliation weights matrix F, and for each community $c \in C$ and each node $u \in V$ determines if u is affiliated to c by examining the entry F_{uc} (see Equation (6)).

The implementation treats rows of F as dense vectors, and requires scanning through all entries of F to determine all community affiliations. Hence, at least $|C||V|$ comparisons must be performed, leading to an average-case complexity of $O(|C||V|)$.