Linear Algebraic Louvain Method in Python

Tze Meng Low, Daniele G. Spampinato
Electrical and Computer Engineering Department
Carnegie Mellon University
Pittsburgh, PA, USA
lowt@cmu.edu, spampinato@cmu.edu

Scott McMillan
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA, USA
smcmillan@sei.cmu.edu

Michel Pelletier
Graphegon, Inc.
Portland, OR, USA
pelletier.michel@gmail.com

***Abstract*—We show that a linear algebraic formulation of the Louvain method for community detection can be derived systematically from the linear algebraic definition of modularity. Using the pygraphblas interface, a high-level Python wrapper for the GraphBLAS C Application Programming Interface (API), we demonstrate that the linear algebraic formulation of the Louvain method can be rapidly implemented.**

***Index Terms*—Louvain Method, Community Detection, Graph Algorithms, GraphBLAS**

I. Introduction

Community detection is a critical component in the analisys of real-world network systems with important applications in social media analytics, biology, recommendation systems, and telecommunications among others [1]. The Louvain method [2] based on modularity optimization [3] is a commonly adopted technique to identify communities.

Current implementations of the Louvain method adopt a traditional vertex-based approach in their algorithmic formulation despite there being a linear algebraic formulation for the computation of modularity [4]. In this paper, we show that by starting with the linear algebraic formulation of modularity, a linear algebraic algorithm for computing the Louvain method can be determined. This formulation can leverage recent community-driven efforts behind linear algebraic graph interfaces, such as the GraphBLAS [5], to rapidly develop an implementation of the Louvain method. Productivity is enhanced through the use of pygraphblas [6], a python interface for the GraphBLAS effort.

II. Background

We assume the goal is to identify communities within a graph graph $\mathcal{G} = (V, E)$ where V is the set of vertices and E is the set of weighted edges connecting them. The Louvain method is an iterative algorithm where each iteration is composed of two major steps:

- *Step 1)* Vertices are moved between communities such that each move increases the overall modularity of G. Step 1 is initialized by creating $|V|$ communities where each vertex is in its own community. Vertices are sequentially reassigned to new communities as long as modularity can be increased. The choice of communities for a vertex is based on the community that yields the highest modularity increase.
- *Step 2)* A new graph is created from the newly partitioned graph. Identified communities form the set of new vertices, and edges between vertices in different communities are aggregated into a new set of edges. Step 1 is then applied to this newly created graph.

The two steps above are repeated until no further improvement in modularity is obtained.

A. Definition of Modularity

Modularity is a scalar measure used to quantify the strengh of community structures within a graph [3]. It is computed as

$$Q = \frac{1}{2m}\sum_{ij}\left[A_{ij} - \frac{k_i k_j}{2m}\right]\delta(c_i, c_j),$$

where A_{ij} is the edge weight between vertices $v_i, v_j \in V$, m is the sum of all edge weights, k_i is the sum of the weights of all edges incident on vertex v_i, $c_i \in C$ is the community to which v_i is currently assigned, and $\delta(c_i, c_j) = 1$ if vertices v_i and v_j are in the same community (i.e., $c_i = c_j$) and 0 otherwise.

We will build on the matrix formulation of Q from [4]:

$$Q = \frac{1}{2m}\Gamma(S^T B S), \tag{1}$$

where $\Gamma(\cdot)$ computes the trace of a matrix, S is a $|V| \times |C|$ matrix that captures the community membership of each vertex, and B is the $|V| \times |V|$ modularity matrix where element B_{ij} is defined as

$$B_{ij} = A_{ij} - \frac{k_i k_j}{2m}.$$

Each column s_i of S represents a community of vertices, where a one in the j^{th} position of s_i indicates that vertex v_j is in community c_i. We denote vertex v_j as a column basis vector e_j of length $|V|$, and s_i is the sum of all vertices that belong to the same community c_i, i.e.,

$$s_i = \sum_{j:v_j \in c_i} e_j.$$

An empty community is simply the zero vector, i.e. $s_i = \vec{0}$.

III. Linear Algebraic Louvain Method

In this section we develop a fully linear algebraic formulation of the Louvain algorithm starting from the linear algebraic definition of modularity.

A. Modularity of a Community

Starting with Eqn. (1) and expanding on the definition of the trace of a matrix, we express the modularity Q as

$$\begin{aligned}Q =& \frac{1}{2m}\left(s_0^T B s_0 + s_1^T B s_1 + \ldots + s_{|C|-1}^T B s_{|C|-1}\right)\\ =& \frac{1}{2m}\sum_{i=0}^{|C|-1} s_i^T B s_i = \frac{1}{2m}\sum_{i=0}^{|C|-1} Q_i,\end{aligned}$$

where Q_i is the modularity of community c_i.

Consider an arbitrary vertex v_j where v_j is a vertex (possibly, the only one) in community c_i. We can define a community vector s_i in terms of the vertex v_j and the remaining set of vertices $s_{i\setminus\{j\}}$ as follows

$$s_i = s_{i\setminus\{j\}} + e_j.$$

Using the above definition, we can derive the following definition of modularity for community i in terms of the contribution to modularity by introducing vertex v_j into community i:

$$\begin{aligned}Q_i &= s_i^T B s_i\\ &= (e_j + (s_i - e_j))^T B (e_j + (s_i - e_j))\\ &= e_j^T B e_j + s_{i\setminus\{j\}}^T B e_j + e_j^T B s_{i\setminus\{j\}} + s_{i\setminus\{j\}}^T B s_{i\setminus\{j\}}\\ &= \underbrace{B_{jj} + s_{i\setminus\{j\}}^T B e_j + e_j^T B s_{i\setminus\{j\}}}_{\text{Contribution by vertex } v_j} + s_{i\setminus\{j\}}^T B s_{i\setminus\{j\}}\end{aligned}$$

B. Linear Algebraic Louvain Method

We address the two steps of Louvain separately.

- Step 1) Initially, vertices are assigned to their own community and the community matrix S is the identity matrix, I.
  Each vertex, v_j is evaluated for the change in modularity when reassigned from community c_i to community c_ℓ. Mathematically, the change in modularity due to such movement ($\Delta Q_{i \xrightarrow{j} \ell}$, where $i \neq \ell$) can be computed as:

$$\begin{aligned}\Delta Q_{i \xrightarrow{j} \ell} &= (B_{jj} + e_j^T B s_{\ell\setminus\{j\}} + s_{\ell\setminus\{j\}}^T B e_j)\\ &\quad - (B_{jj} + e_j^T B s_{i\setminus\{j\}} + s_{i\setminus\{j\}}^T B e_j)\\ &= e_j^T B s_{\ell\setminus\{j\}} + s_{\ell\setminus\{j\}}^T B e_j\\ &\quad - (e_j^T B s_{i\setminus\{j\}} + s_{i\setminus\{j\}}^T B e_j).\end{aligned}$$

  Essentially, $\Delta Q_{i \xrightarrow{j} \ell}$ is computed by subtracting the contribution of vertex v_j to community c_i from the contribution of vertex v_j to community c_ℓ.
  Let q be a vector of length $|V|$ defined as follows:

$$q^T = \left(\Delta Q_{i \xrightarrow{j} 0} \;\middle|\; \cdots \;\middle|\; \Delta Q_{i \xrightarrow{j} |V|-1} \right).$$

  As the Louvain algorithm only considers moving a vertex to a community that includes at least one neighbor, we can restrict the modularity computation to the communities containing neighbors of v_j via a mask in the following manner:

$$q_1^T = q^T \circ (\underbrace{(e_j^T (A + A^T))S}_{t_q} \neq 0). \qquad (2)$$

  The maximum increase in modularity is computed as $\kappa = \max(q_1)$, and the community c_ℓ associated with κ can be identified by

$$t = (q_1 == \kappa),$$

  where t is a boolean vector of length $|V|$ indicating the new community for vertex v_j. The original community c_i for vertex v_j can be computed by extracting the j^{th} row of S. Using the original community and the vector t, the community matrix S can be updated as

$$S = S + e_j t^T - e_j e_j^T S.$$

  Step 1 is repeated for all vertices until no increase in modularity can be achieved.
- Step 2) A new graph $\mathcal{G}' = (V', E')$ is constructed, where the set of vertices V' are the communities identified in Step 1. The new adjacency matrix A' is computed as

$$A' = S^T A S,$$

  where the weight at location A'_{ij} is the sum of edge weights in A between vertices in community c_i and vertices in community c_j. The diagonal elements A'_{ii} are the sum of edge weights in A belonging to the same communities c_i.

IV. Implementation Considerations

In this section, we focus on Step 1 of the previously presented algorithm and discuss several optimizations.

1) Removing redundant computation. Recall the objective in Step 1 is to find the maximum increase in modularity. Since the expression $e_j^T B s_{i\setminus\{j\}} + s_{i\setminus\{j\}}^T B e_j$ is removed from all modularity computation, it does not contribute to the decision of which community should vertex v_j be reassigned to. Thus, the expression $e_j^T B s_{i\setminus\{j\}} + s_{i\setminus\{j\}}^T B e_j$ does not need to be computed. Hence, the following computation can be performed instead:

$$\begin{aligned}\Delta \tilde{Q}_{i \xrightarrow{j} \ell} &= e_j^T B s_{\ell\setminus\{j\}} + s_{\ell\setminus\{j\}}^T B e_j.\\ &= e_j^T (B + B^T) s_{\ell\setminus\{j\}} \qquad (3)\end{aligned}$$

2) Bulk computation of q. Notice that Eqn. (3) can be computed at once for all values of ℓ as:

$$q^T = e_j^T (B + B^T)\bar{S}.$$

where

$$\begin{aligned}\bar{S} &= S - e_j e_j^T S\\ &= (I - e_j e_j^T)S, \qquad (4)\end{aligned}$$

which is the community matrix after removing vertex v_j from its original community.

3) Eliminating modularity matrix B. Modularity matrix B is a dense $|V| \times |V|$ matrix. In order to reduce the memory requirements, recall that

$$B = A - \frac{1}{2m} k k^T,$$

where k is a vector where the i^{th} value is the node degree of vertex v_i. Therefore,

$$B + B^T = (A - \frac{1}{2m}kk^T) + (A - \frac{1}{2m}kk^T)^T$$
$$= (A + A^T) - \frac{1}{m}kk^T. \quad (5)$$

Instead of storing B, the above formulation allows us to use a sparse matrix format to store $G = (A + A^T)$, and a dense vector k, thus reducing the memory footprint.

4) Handling same change in modularity. Often, moving vertex v_j into different communities yields the same change in modularity. In such cases, a tie-breaking heuristics has to be applied to favor one community over the rest. In this paper, a community is picked at random.

Applying these optimizations, the final linear algebraic Louvain method algorithm is shown in Fig. 1 (left).

V. Experiments

In this section, we discuss preliminary experimental results in implementing our formulation of the Louvain method discussing both productivity and performance aspects provided by the use of pygraphblas [6], a Python wrapper to the GraphBLAS [5] API for graph algorithms.

TABLE I
Comparison between two Python implementations of the Louvain method: NetworkX (vertex-based, sequential) and pygraphblas (present work, linear algebraic, both sequential, 1T, and parallel, 4T). The number of communities (ground truth) are reported in Column 3. Both implementations reported 100% accuracy.

Graph			NetworkX	pygraphblas			
V	E	C	Time (s)	Time (s) w/o Mask		Time (s) w/ Mask	
			1T	1T	4T	1T	4T
50	319	3	0.02	0.06	0.07	0.07	0.09
100	778	5	0.03	0.07	0.08	0.08	0.10
500	9.4k	8	0.31	0.41	0.47	0.48	0.58
1,000	20.1k	11	0.78	0.79	0.89	0.93	1.11
5,000	101.9k	19	5.59	7.55	8.19	8.63	9.06
20k	408.8k	32	26.29	103.99	70.03	113.43	77.17
50k	1.0M	44	88.12	696.27	387.86	711.83	380.24

A. Experimental Setup

We implemented our linear algebraic Louvain method using pygraphblas [6], a Python wrapper of the SuiteSparse [7] library v3.2.0 that implements the C GraphBLAS API, and compared with the implementation of the original vertex-based Louvain method for the NetworkX package [8] v.2.4. Our implementation is available at [6].

We ran our experiments on an Intel Core i7-4770K (Haswell) CPU with four cores, 8 MB LLC cache, and 32 GB of memory. We provide both sequential (1T) and parallel (4T, `GxB_CHUNK`=1024) results in Table I.

We report time to solutions (mean of seven runs) for implementations using Static Graphs with known truth for the 2017 Partitioning Challenge provided on the Graph Challenge website [9]. All implementations correctly identified communities when compared to the ground truth.

B. Discussion

In terms of productivity, our pygraphblas implementation can be mapped nearly one-to-one with the linear algebraic formulation of the algorithm, and requires less than 40 lines of code (LOC). In contrast, the vertex-based NetworkX implementation requires 4× as many (158) lines of code (LOC). For a graph algorithm developer, the use of the pygraphblas interface greatly simplifies the implementation; resulting in a boost in algorithmic development productivity.

The ability to port between CPU and GPU platforms to take advantage of different hardware while maintaining the same high-level interface (similar to Blanco et. al [10]) is also another advantage of using GraphBLAS.

Performance-wise, due to the use of SuiteSparse as a black-box library that implement the GraphBLAS API, it is unclear how much of the performance is related to the linear algebraic approach or specific implementation decisions. For example, parallelism was performed by the underlying SuiteSparse library, which may choose to use less than the allocated number of threads. Selecting the chuck size used allowed us some limited form of control over this decision. Nonetheless, we highlight key performance-related features of our implementation.

Lines 23-25 in Fig. 1 (right) are essentially the equivalent of the AXPY operation in the Level 1 BLAS [11]. This can be implemented with a single GraphBLAS `apply` operation in the most recent GraphBLAS 1.3 as follows:

```
k.apply(FP64.TIMES, first=(-k_j/m),
        accum=FP64.PLUS, out=v)
```

However, as this capability is not supported by the version of SuiteSparse used, three separate function calls, each traversing through a vector of length $|V|$, have to be used.

It was interesting to note that while the masking operation as described by Eqn. (2) was meant to reduce the computation cost by considering only neighbor communities when determining changes in modularity, the computation of the mask was a significant cost in our implementation. As such, it was generally faster for our implementation to compute the change in modularity for all existing communities. This cost could potentially be amortized with larger graphs.

A considerable fraction of the runtime was spent extracting from or assigning to single matrix columns or rows. While we have implemented the computation of the change in modularity as a bulk operation, the sequential vertex approach remains a large cost in this implementation.

VI. Conclusion

In this paper, we derived a linear algebraic formulation of the Louvain method for community detection starting from the linear algebraic definition of modularity. We also implemented the derived algorithm using the pygraphblas interface to enhance algorithm design productivity.

We believe that alternative modularity-based clustering algorithms (e.g. the synchronized Louvain algorithm [12]), should enable better performance when expressed in terms of bulk linear algebra operations.

```
$G = A + A^T$

$k = A\ vec(1)$
$m = \frac{1}{2}\ k^T\ vec(1)$
$S = I$

$v$ changed = True
while $v$ changed:
  $v$ changed = False
  for $j$ in range($|V|$):

      $t_q = (e_j^T G) S$

      $t_j^T = e_j^T S$
      $\bar{S} = (I - e_j e_j^T) S$



      $q^T = e_j^T G + (-k_j/m) k^T$
      $q_1^T = (q^T \bar{S}) \circ (t_q \neq 0)$

      $\kappa_q = \max(q_1)$
      $t = (q_1 == \kappa_q)$
      while $t$.nvals() != 1:
        $p =$ random() $\circ$ $t$
        $\kappa_p = \max(p)$
        $t = (p == \kappa_p)$
      $S = \bar{S} + e_j t^T$


      if $t$ != $t_j$:
        $v$ changed = True
```

```python
def louvain_cluster(graph):
  rows = graph.nrows
  S_row = Vector.from_type(BOOL, rows)
  empty = Vector.from_type(BOOL, rows)

  G = graph.dup()
  graph.transpose(out=G, accum=FP64.PLUS)
  G_rows = [ G.extract_row(i) for i in range(rows) ]
  k = graph.reduce_vector()
  inv_m = 2.0 / k.reduce_int()
  S = Matrix.identity(BOOL, rows, format=lib.GxB_BY_COL)

  vertices_changed = len(k)
  while vertices_changed > 0:

    for j,k_j in k:
      v = G_rows[j]
      t_q = v.vxm(S, semiring=BOOL.ANY_PAIR)

      S.extract_row(j, out=S_row)
      S[j,:] = empty

      q = k.dup()
      q.assign_scalar(-k_j * inv_m, accum=FP64.TIMES, mask=k)
      q += v
      q1 = q.vxm(S, mask=t_q)

      max_q1 = q1.reduce_float(MAX_MONOID)
      t = q1.select("==", max_q)
      while len(t) != 1:
        p = t.apply(random_scaler)
        max_p = p.reduce_float(MAX_MONOID)
        t = p.select("==", max_p)
      S[j,:] = t

      vertices_changed -= 1
      if len(S_row * t) == 0:
        vertices_changed = len(k)-1
      if vertices_changed <= 0:
        break

  return S
```

Fig. 1. (Left) Formulation of of the linear algebraic Louvain method described in Sec. III and (right) its pygraphblas implementation.

Acknowledgement

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center [DM20-0213]. References herein to any specific commercial product, process, or service by trade name, trade mark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by Carnegie Mellon University or its Software Engineering Institute.

References

[1] S. Fortunato, "Community detection in graphs," Physics Reports, vol. 486, no. 3, pp. 75–174, 2010.

[2] V. D. Blondel, J. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of community hierarchies in large networks," Journal of Statistical Mechanics: Theory and Experiment, vol. 10, 2008.

[3] M. E. J. Newman and M. Girvan, "Finding and evaluating community structure in networks," Physical review E, vol. 69, p. 026113, 2004.

[4] M. E. J. Newman, "Modularity and community structure in networks," Proceedings of the National Academy of Sciences (PNAS), vol. 103, no. 23, p. 85778582, 2006.

[5] J. Kepner, P. Aaltonen, D. A. Bader, A. Buluç, F. Franchetti, J. R. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke, S. McMillan, J. E. Moreira, J. D. Owens, C. Yang, M. Zalewski, and T. G. Mattson, "Mathematical foundations of the GraphBLAS," CoRR, vol. abs/1606.05790, 2016. [Online]. Available: http://arxiv.org/abs/1606.05790

[6] M. Pelletier, "pygraphblas: GraphBLAS for Python." [Online]. Available: https://github.com/michelp/pygraphblas

[7] T. Davis, "Algorithm 9xx: SuiteSparse:GraphBLAS: graph algorithms in the language of sparse linear algebra," Submitted to ACM Transactions on Mathematical Software (TOMS), 2018.

[8] T. Aynaud, "Community detection for NetworkX." [Online]. Available: https://python-louvain.readthedocs.io

[9] "Graph Challenge," 2017. [Online]. Available: https://graphchallenge.mit.edu

[10] M. Blanco, T.-M. Low, and K. Kim, "Exploration of fine-grained parallelism for load balancing eager K-truss on GPU and CPU," in High Performance Extreme Computing (HPEC), 2019.

[11] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic linear algebra subprograms for Fortran usage," ACM Transactions on Mathematical Software (TOMS), vol. 5, no. 3, pp. 308–323, 1979.

[12] A. Browet, P.-A. Absil, and P. Van Dooren, "Fast community detection using local neighbourhood search," preprint arXiv:1308.6276, 2013.