The Sparse + Low-Rank Trick for Matrix Factorization-Based
Graph Algorithms

Nathan de Lara
ndelara@enst.fr
Télécom Paris
Paris, France

ABSTRACT
Matrix factorization is a central block in many graph algorithms
for embedding, clustering and a number of other tasks. This block
is usually the computational bottleneck of these algorithms and
a naive implementation can prevent them from scaling to large
datasets. However, the matrices to factorize often have a closed-
form “sparse + low-rank” structure. In this paper, we show how to
adapt state-of-the-art matrix factorization techniques to this class
of matrices. We demonstrate that the method is highly competitive
with respect to a naive implementation and that it comes at a very
small extra cost as compared to the decomposition of the sparse
component alone.

CSC CONCEPTS
• Computing methodologies → Machine Learning;

KEYWORDS
algorithms, matrix factorization, graph mining

ACM Reference format:
Nathan de Lara. 2019. The Sparse + Low-Rank Trick for Matrix Factorization-
Based Graph Algorithms. In Proceedings of KDD ’19: 25th ACM SIGKDD
conference on knowledge discovery and data mining, Anchorage, Alaska, Au-
gust 04–08, 2019 (KDD ’19), 4 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION
Also known as "spectral methods", matrix factorization-based algo-
rithms are widely used in graph mining: spectral clustering [21],
spectral modularity optimization [17], manifold learning [1], graph
classification [4], node ranking [19], natural language processing
[14], connected-components search [5], anomaly detection [9]... As
the decomposition of the matrix is usually the computational bot-
tleneck, it has to be able to scale to large graphs in order for these
algorithms to compete with iterative methods for analogous tasks
[2, 6, 8, 11, 15]. Making such graph algorithms scale is mandatory
to obtain more comprehensive benchmarks.

In the course of this paper, we use the following notations:
• $A \in \mathbb{R}^{n \times n}$ denotes the adjacency matrix of directed or undi-
rected graph,
• $B \in \mathbb{R}^{m \times n}$ denotes the biadjacency matrix of a bipartite
graph,
• $m$ is the number of non-zero coefficients in a matrix (i.e. the
number of edges in the graph),
• $d = A_1$ or $B_1$ denotes the out-degree vector, and $D = \text{diag}(d)$,
• $f = A^T 1$ or $B^T 1$ denotes the in-degree vector,
• $w = 1^T A_1$ or $1^T B_1$ is the total weight of the graph.

There are two distinct issues to tackle in order for spectral meth-
ods to scale: the matrix factorization itself, but also the storage of
the input. Indeed, even though the adjacency matrix of a graph
and some derived matrices are usually very sparse (random walk
transition matrix, Laplacian matrix,...), other matrices of interest
are dense, and in the case of large graphs, they do not fit in
the RAM. Such dense matrices include
• random surfer’s transition matrix [18]: $R_{ij} = \alpha \frac{A_{ij}}{d_i} + \frac{1 - \alpha}{n}$,
• modularity [16]: $Q_{ij} = \frac{1}{w} A_{ij} - \frac{Y^2}{w^2} d_i d_j$,
• shifted pointwise mutual information$^1$ [3]:
  \[ SPM_{ij} = \log \left(\frac{\alpha A_{ij}}{d_i d_j}\right) A_{ij} > 0 - \log(k), \]
• regularized graph Laplacian [22]: $L^T_{ij} = 1 - \frac{A^T_{ij}}{\sqrt{d_i d_j}}$,
• principal components analysis matrix [20]: $\tilde{A}_{ij} = A_{ij} - \frac{f_j}{w}$.

However, note that these matrices all have a closed-form “sparse + low-
rank” decomposition (even sparse + rank-1 in this case) i.e. one can
write them as $S + xy^T$ where $S$ is a sparse matrix and $x$ and $y$ are
vectors of appropriate dimensions. For the sake of simplicity, we
only consider “sparse + rank-1” matrices. However, the extension
to higher ranks is trivial.

Formally, these matrices are represented by tuples of the form
$(S, (x, y))$. Storing such tuples only requires $O(m + 2n)$ memory
space ($O(m + 2n)$ if there are $r$ rank-1 tuples) instead of $O(n^2)$
or $O(n_1 n_2)$ for $S + xy^T$. Besides, this specific structure allows to
leverage powerful factorization methods as discussed in Section 2.

The identification of the low-rank factors is almost direct for all
the aforementioned matrices as shown in Table 1.

2 FACTORIZATION METHOD
Computing partial spectral decomposition, i.e. the $k$ first singular
vectors or eigenvectors, is well researched problem. Standard
solvers usually rely on power-iteration methods such as [13]. Faster

\footnote{The pointwise mutual information is set to 0 if $A_{ij} = 0$.}
methods use randomized projections to reduce dimensionality before factorizing \cite{7}. The key point of these methods is that their computational bottlenecks are two simple primitives, namely matrix transposition and matrix-vector product.

Indeed, recall algorithm 4.4 in \cite{7}:

\textbf{Algorithm 1 Randomized Subspace Iteration}

\textbf{Require:} input matrix $M$

1. Generate an $n \times 2k$ Gaussian test matrix $\Omega$.
2. Form $Y_0 = MQ$ and compute its QR factorization $Y_0 = Q_0 R_0$.
3. for $i = 1, \ldots, q$ do

4. Form $\tilde{Y}_i = M^T \tilde{Q}_{i−1}$ and compute its QR factorization $\tilde{Y}_i = \tilde{Q}_i R_i$.
5. Form $Y_i = M \tilde{Q}_i$ and compute its QR factorization $Y_i = Q_i R_i$.
6. $Q = Q_q$.

The parameter $q$ is usually a small integer (between 1 and 3). Once $Q$ is computed, form the small matrices $B_{eig} = Q^T M Q$ and $B_{svd} = Q^T M M^T Q \in \mathbb{R}^{2k \times 2k}$. Compute their respective $k$ first eigenvectors using any method: $B_{eig} \approx U_{eig} \lambda_{eig}^T$, and $B_{svd} \approx U_{svd} \Sigma^{2} U_{svd}^T$, $Q U_{eig}$ and $Q U_{svd} \in \mathbb{R}^{n \times k}$ are respective approximations of the $k$ first eigenvectors and left singular vectors of $M$.

Now, consider the two primitives, transposition and matrix-vector product for *sparse + low-rank* matrices. The dense transposed matrix is simply $S^T \times y^T$. Hence, the transposed tuple is $(S^T, (y, x))$. In the standard compressed sparse row format\textsuperscript{2}, transposing a matrix comes at the cost of sorting its non-zero entries, hence $O(m \log m)$, whereas it comes at a constant cost for dense matrices. However, this drawback is well compensated by the efficiency of the second primitive. Indeed, consider the following matrix vector products:

\begin{align*}
a &= (S + x y^T) z, \\
b &= S z + (y^T) y x.
\end{align*}

Even though these two vectors are formally equal i.e. $a = b$, computing $b$ does not require to store a dense matrix while computing $a$ does. Besides, computing $b$ requires $O(m + 2n)$ floating point operations whereas $O(n^2)$ or $O(n_1 n_2)$ operations are necessary to compute $a$. As a result, decomposing a sparse + low-rank matrix comes at the cost of the overhead of computing the matrix-vector product of its low-rank part with respect to the usual sparse case. Algorithms 2 and 3 summarize the two primitives.

\textbf{Algorithm 2 Transposition}

\textbf{Require:} $(S, (x, y))$

1. return $(S^T, (y, x))$

\textbf{Algorithm 3 Matrix-vector product}

\textbf{Require:} $(S, (x, y))$

1. return $S z + (y^T) y x$.

In the end, the trick to decompose large *sparse + low-rank* matrices works as follows:

1. Identify the sparse and low-rank components (See Table 1 for examples).
2. Overwrite transposition and matrix-vector with Algorithms 2 and 3 respectively.

Note that, for very large graphs, the random projection matrix, which has between $nk$ and $2nk$ floating-number coefficients, can be a memory bottleneck if $k \geq \frac{n}{2}$ (especially if the graph has simple binary edges). Therefore, the average number of neighbors of the nodes in the graph is a natural limit to the number of factors in the case of simple implementations. One way to overcome this issue is to drop the QR factorizations as suggested in \cite{7} and form $B_{eig}$ and $B_{svd}$, accessing the data chunk by chunk. However, this engineering lies beyond the scope of this paper.

\section{3 EXPERIMENTS}

We compute the partial eigenvalue decomposition and the partial singular value decomposition of the modularity matrix of 7 undirected and 7 bipartite graphs respectively. In the bipartite case, the modularity matrix is defined as $\frac{1}{n} B = \frac{1}{n} d f^T$. The graphs are collected from the Koneck \cite{12} database such that they have a varying number of nodes and edges. See Table 2 for some metadata. We extract 16 components in each case and record the computation times for our modified Python implementation of Halko’s method. We compare our results to the computation times of the direct Lanczos method using Scipy \cite{10}.

We also record the running times for the Reactome (undirected) and MovieLens user–tag (bipartite) graphs for $k \in \{8, 16, 32, 64, 128, 256, 512\}$.

All experiments are performed on a laptop equipped with an Intel i7-7820HQ CPU (2.90 GHz) and 16 GB of RAM. The code to reproduce the experiments is available online\textsuperscript{3}. Results are displayed in Figure 1 and Figure 2 respectively.

In this setup, memory errors arise around $n^2$ or $n_1 n_2 = 10^9$ for the dense implementation. At this point, the Lanczos method is already between one and two orders of magnitude slower than Halko’s. On the other hand, the *sparse + low-rank trick* enables to handle graphs with more than 3 million nodes and 234 million edges in approximately five minutes. Furthermore, note that the running time for the implicit dense matrix is quite close to the running time for its sparse component. This holds for a fixed $k$ and varying graphs as well as for fixed graphs and varying $k$.

In the end, we have demonstrated that, without any specific hardware, spectral methods can easily scale to handle graphs at

\begin{table}[h]
\centering
\begin{tabular}{|l|l|l|}
\hline
matrix & $S$ (sparse) & $(x, y)$ \\
\hline
random surfer & $\alpha D^2 A$ & $(\frac{1}{n^2}, 1, 1)$ \\
modularity & $\frac{1}{n^2} A$ & $(-\frac{1}{\sqrt{\tau}}, \frac{1}{\sqrt{\tau}}, 1)$ \\
SPMI & $\log(\frac{d}{\lambda_{\min} A})_{\lambda_{\max} > 0}$ & $(-\tau, \frac{1}{\sqrt{\tau}}, 1)$ \\
Laplacian & $I - D^{-1/2} A D^{-1/2}$ & $(-\tau, \sqrt{\frac{1}{\tau} + \tau}, \sqrt{\frac{1}{\tau} + \tau})$ \\
PCA & $A$ & $(-\frac{1}{\tau}, 1, 1)$ \\
\hline
\end{tabular}
\caption{Identification of sparse and low-rank factors.}
\end{table}

\begin{footnotesize}
\begin{enumerate}
\item \texttt{https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csr_matrix.html}
\item \texttt{https://github.com/nathandelara/Matrix-Factorization}
\end{enumerate}
\end{footnotesize}
A Python module to perform such factorizations is implemented in Scikit-network\(^4\). We hope that this will help produce richer benchmarks for graph algorithms.

REFERENCES


\(^4\)https://github.com/sknetwork-team/scikit-network

---

**Figure 1:** Computation times in seconds for partial decomposition of matrices. Top: undirected graphs. Bottom: bipartite graphs.

**Figure 2:** Computation times in seconds for different values of \(k\). Top: Reactome. Bottom MovieLens user–tag.