Computing Shortest Paths Using Sparse Gaussian Elimination

Aydin Buluç, John Gilbert, Sivan Toledo

SIAM Workshop on Network Science 2014
July 6-7 · Chicago

Introduction
We aim to bring attention an alternate method of computing shortest paths using sparse matrix factorization. On a graph \(G(E, V)\), consider the shortest distance from a starting vertex \(v_i\) to an end vertex \(v_j\). We use the adjacency matrix \(A\) to conceptualize the graph. The length of an edge \(e(i, j)\) is denoted by \(a_{ij}\) in \(A\). If an edge \(e(i, j)\) does not exist, \(a_{ij} = \infty\). Suppose we know shortest paths from \(v_i\) to all the vertices except \(v_j\), and they are encoded in a vector \(x\) whose \(j\)th entry is unknown, but can be computed as: \(x_j = \min(x_1 + a_{1j}, x_2 + a_{2j}, \ldots, x_n + a_{nj})\).

A closed semiring is formally denoted by \((S, \oplus, \otimes, 0, 1)\), where \(\oplus\) and \(\otimes\) are binary operations defined on the set \(S\) with identity elements 0 and 1 respectively. Consider the tropical semiring \((\mathbb{R}^+, \min, +, \infty, 0)\). The equations in matrix form, for all \(j = \{1, 2, \ldots, n\}\), results in the system \(x = A^T x + b\) on this semiring, with \(b\) being all infinities except for \(b_i = 0\) as it is the starting vertex.

This is well-defined in the absence of negative cycles. If the matrix closure \(A^* = I \oplus A \oplus A^2 \oplus \ldots\) converges, then the system of linear equations \(x = A^T x + b\) becomes \((A - I)x + b = 0\), which is \(x = (A - I)^{-1}(-b)\) [1].

Factorization

Algorithm 1 Factorize \(A\) in the \((\min,+)\) semiring

1: procedure LU-Factorize(\(A\))
2: \hspace{1cm} \(n \leftarrow \text{rows}[A]\)
3: \hspace{1cm} for \(k = 1\) to \(n\) do \hspace{1cm} \(\triangleright\) Loop computes \(A_{k,n,k:n}^{(k-1)}\)
4: \hspace{2cm} for \(i = k + 1\) to \(n\) do
5: \hspace{3cm} for \(j = k + 1\) to \(n\) do
6: \hspace{4cm} \(a_{ij} \leftarrow \min(a_{ij}, a_{ik} + a_{kj})\)

We perform standard Gaussian elimination on \(A\). The algorithm is shown in-place, and in a right looking manner for ease of explanation but any compliant implementation would work. The algorithm is also described by Tarjan [3] as ELIMINATE and SOLVE. We will sometimes refer to the upper triangular part as \(U\), and lower triangular part as \(L\) whenever we need to explicitly emphasize directed paths.

For example, \(U^*\) encodes the paths from lower numbered vertices to higher numbered vertices.

Figure 1 shown the input and corresponding directed graph of a running example. The certain parts of these matrices are already filled. When the factorization is over, we get the filled graph and matrix in Figure 2 (note that the changed elements are in bold):

\[
A = \begin{pmatrix}
0 & 8 & \infty & \infty \\
\infty & 0 & 9 & -3 \\
\infty & 5 & 0 & \infty \\
\infty & 2 & \infty & 0
\end{pmatrix}
\]

Figure 1: Adjacency matrix (left) of the graph (right)

\[
A = \begin{pmatrix}
0 & 8 & \infty & \infty \\
\infty & 0 & 9 & -3 \\
\infty & 5 & 0 & 14 & 2 \\
\infty & 2 & 16 & 0
\end{pmatrix}
\]

Figure 2: Factorized matrix (left) and filled graph (right)

For \(A_{k,n,k:n}^{(k-1)}\), all the paths from vertices \(k\ldots n\) to vertices \(k\ldots n\) passing through intermediate vertices \(1\ldots(k - 1)\) are discovered. This invariant implies:

1. Any path from \(i\) to \(j\) that passes through any intermediate vertex \(k > j\) is not discovered.
2. All paths from \(i\) to \(j\) that passes through no intermediate vertex \(k > j\) are discovered. Think about a path \(4 \rightarrow 1 \rightarrow 2 \rightarrow 3\). The subpath \(1 \rightarrow 2 \rightarrow 3\) is never actually discovered. However, when vertex 1 is eliminated, the subpath \(4 \rightarrow 1 \rightarrow 2\) is discovered.
and the full path from 4 to 3 will be discovered when vertex 2 is eliminated.

In general, when vertex \( k \) is eliminated, the shortest path to it from any vertex that goes through intermediate vertices 1, 2, \ldots, \((k - 1)\) is known. Same is true for the other direction (i.e. the shortest path from it to any vertex that goes through intermediate vertices 1, 2, \ldots, \((k - 1)\) is known). After elimination, all vertices \( k' > k \) will know shortest paths going through intermediate vertices 1...\( k \). However, for vertices labeled \( k'' < k \), the shortest paths that has to go through at least one intermediate vertex in the set \( \{k, k+1, \ldots, n\} \) are not yet known. Floyd-Warshall or Kleene’s algorithms compute those paths as well by recursing on both directions.

### Triangular Solve

Next, we solve the triangular systems with \( v \) as the initial starting costs. For computing shortest paths starting from vertex \( j \), we take \( v = e_j \) where \( e_j \) is the \( j^{th} \) unit vector in the \((\min,+)\) semiring. The forward substitution will make a sequence of updates to \( v \). The invariant is that after the \( k^{th} \) recursive call, \( v_{(k+1):n} \) will contain the length of the shortest paths that goes through vertices 1...\( k \). This is because those paths are encoded in matrix \( U \) from the factorization step. For our particular example, execution is as shown in (1) where still active elements are in bold.

#### Algorithm 2 Triangular Solve

1: \textbf{procedure} `SOLVE`(\( A^*_n, v \))
   2: \hspace{1em} \( n \leftarrow\) rows[\( A^*_n \)]
   3: \hspace{1em} \textbf{for} \( j = 2 \) \textbf{to} \( n \) \textbf{do}
   4: \hspace{2em} \( v_j = \min(v_j, v_1 + U^*_1) \)
   5: \hspace{1em} \textbf{if} \( n > 2 \) \textbf{then}
   6: \hspace{2em} \( u_{2:n} \leftarrow \text{SOLVE}(A^*_2n, v_{2:n}) \)
   7: \hspace{1em} \textbf{else}
   8: \hspace{2em} \( u_2 \leftarrow v_2 \)
   9: \hspace{2em} \( u_1 \leftarrow v_1 \)
   10: \hspace{1em} \textbf{for} \( j = 2 \) \textbf{to} \( n \) \textbf{do}
   11: \hspace{2em} \( u_1 = \min(u_1, u_j + L^*_j) \)

The backward substitution step in the last loop completes the algorithm. For instance, the shortest path to 3 may or may not contain 4,5 as intermediate vertices. The former case is covered by the loop \( \min(u_4 + L^*_{43}, u_5 + L^*_{53}) \), as \( u_4 \) and \( u_5 \) already has the shortest paths from the starting vertex 2 (by means of recursion). The latter case is covered by \( v_3 \) since it contains the shortest path to vertex 3 that only goes through vertices 1, 2, 3. The situation is illustrated in Figure 3. After the solve routine completes, the final distance vector is \( v = (16, 0, -1, 9, -3)^T \)

![Figure 3: Shortest paths by solving triangular systems](image)

#### Implications and Future Work

The algorithm presented works on directed as well as undirected graphs. Since it does not depend on the order of edge and paths (Dijkstra type algorithms do), it works in the presence of negative edge weights. Its data access patterns are identical to sparse Gaussian elimination and the fill is directly regulated by the vertex elimination order. Therefore, the mature apparatus that is used to minimize fill, parallelize the factorization, and increase computational intensity (e.g. supernodal and multifrontal methods) is readily applicable to shortest path problems.

We implemented this algorithm as modifications of existing sparse LU codes such as CSparse [2]: by turning off pivoting in left-looking versions and changing the underlying algebra. We will provide experimental results comparing the performance with state-of-the-art implementations and report on break-even points for this algorithm versus Bellman-Ford and Dijkstra type algorithms.

#### References

