Liu Chengyu

A Faster Algorithm for Information Flow Analysis

Abstract:
Information flow analysis is an effective way to identify and characterize important nodes in a network. This method can be apply not only to analyze interactions between genes and proteins, but also interaction networks in general. However, the method used to solve the linear system in the algorithm scales very fast ($O(n^4)$) as the size of the system increases. In this paper, we present optimized algorithms that can solve this kind of system under $O(n^2m)$ time complexity while having a reasonable time constant and using twice the among of memory as the input data.

Introduction:
Information flow analysis, developed by Dr. Missiuro and her colleagues[1] offers a better way to extract information of gene interactions than betweenness and shortest path methods. The most computationally intense problem involved in this analysis can be described as below:

"Given an n x n symmetric matrix A, for i = 1 to n, zero the row i and column i in A, except A[i,i] = 1 (isolate the node from the circuit). Then, for j = i+1 to n, solve for X on the linear system: AX=Y, in which Y has 1 for element j and 0 for all other elements."

In the paper, Dr. Missiuro and her colleagues offer a solution as below (which I paraphrased to make later explanations easier):

1. assemble matrix A based on the conductance matrix
2. for i =1...N:
    a. remove the row and column i of A to get A^{**}
    b. decompose A^{**} into L and U matrices ($O(n^3)$)
    c. for j=(i+1)...N:
        i. assemble vector Y to have all 0s except 1 in the j^{th} entry
        ii. solve X: $X=U^{-1}(L^{-1}Y)$
        iii. compute the absolute sum of currents for each node

LU decomposition is a $O(n^3)$ operation. Running it for n times will result in a $O(n^4)$ runtime that scales up very fast. Even though many of the genetic interactome network matrices can be sparse, LU decomposition has a "fill-in" effect which causes the resulting L and U matrices to be dense. Previous research has developed strategies such as using pivoting matrices or hypergraph[2] to reduce this effect, which is not the focus here. Besides, making the algorithm asymptotically faster is meaningful despite the density of the matrices.

Here, we have developed an algorithm that can solve the problem in $O(n^2m)$, where m is the number of branches in the network. They are also more memory efficient than the original algorithm (In dense

circumstances, A, A^{**}, L and U can take as much as 3 times the memory of the input) and have a reasonable time constant.

Basic Implementation:

We first states two formula and two facts that we will use a lot in constructing this algorithm:

1. If v is an 1xN vertical unit vector with 1 as its i^{th} element, M is a NxN matrix, then $M \cdot v$ is equivalent to extracting the i^{th} column of M and $v^T \cdot M$ is equivalent to extracting the i^{th} row of M.
2. Sherman-Morrison formula[3]: If a matrix $W = uv^T$, where u or v is a unit vector and both of them are vertical vectors. Then $(A+W)^{-1} = A^{-1} - \frac{A^{-1}uv^TA^{-1}}{1+v^TA^{-1}u}$.
3. Since A is a symmetric matrix and A^{**} is equal to A with i^{th} row and column removed, A^{-1}, A^{**} and thus A^{**-1} are symmetric as well.
4. Row operations are generally preferred over column operations due to cache locality (for instance, extracting rows is more efficient than extracting columns).

Given that A is a symmetric matrix, we use LDL^T decomposition instead of LU which is faster and more numerically stable. Then, we uses backward elimination on L and L^T to compute A^{-1}.

For voltage X on each node, we solve $A^{**}X=Y$, which is essentially $X=A^{**-1}Y$. Since Y is a unit vector with 1 in position j and A^{**} is symmetric, $X=A^{**-1}Y=j^{th}$ column of A^{**-1} = j^{th} row of A^{**-1} ($A^{**-1}[j]$).

Thus, we only need to calculate A^{**-1} from A^{-1} to get X. Suppose matrix ε is all 0, except the i^{th} row is the minus of i^{th} row of A and $\varepsilon[i,i] = \frac{1}{2}(1 - A[i,i])$. This can be done by applying the Sherman-Morrison formula to A^{-1} twice (**Fig. 1**), once with ε^T and once with ε.

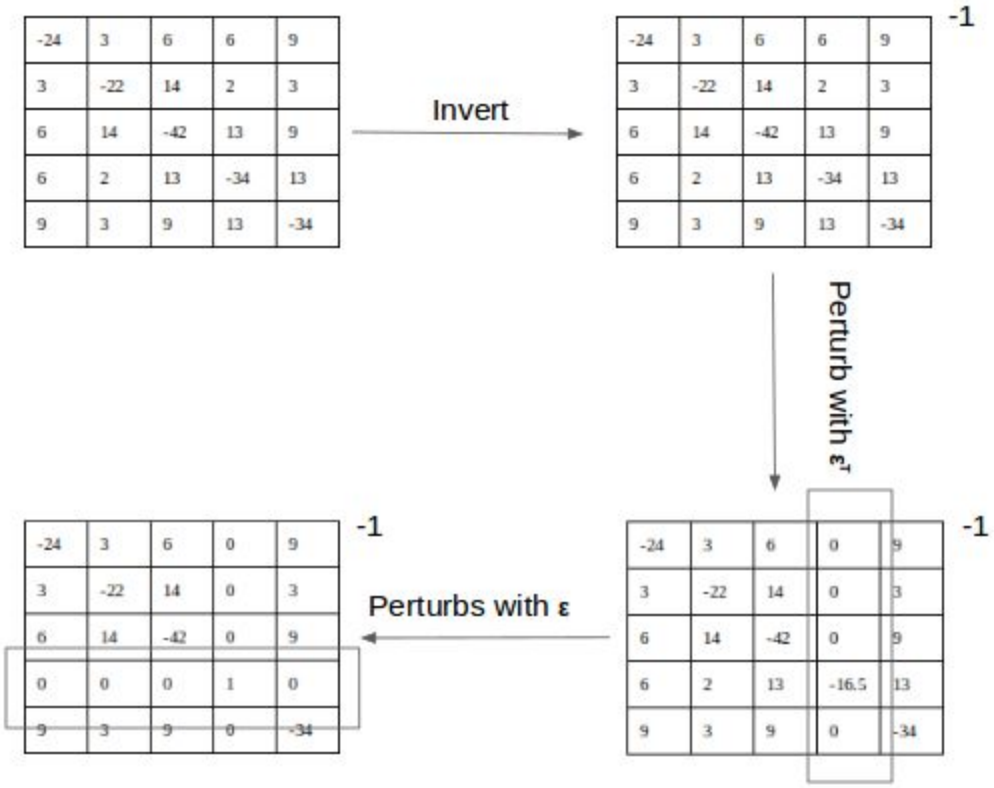

Figure. 1, obtaining the desired inverted matrix by perturbing the original inverted matrix twice using the Sherman-Morrison formula

Here, we will perturb A^{-1} with ε^T first. Suppose A^{*-1} is the matrix after the first perturbation, $\varepsilon^T = uv^T$, v is the unit vector, then we have:

$$A^{*-1} = (A+\varepsilon^T)^{-1} = A^{-1} - \frac{A^{-1}uv^TA^{-1}}{1+v^TA^{-1}u} \quad (1)$$

$$A^{**-1} = (A^*+\varepsilon)^{-1} = A^{*-1} - \frac{A^{*-1}vu^TA^{*-1}}{1+u^TA^{*-1}v} \quad (2)$$

Let $u^TA^{-1}v = \lambda$ and $u^TA^{-1}u = \mu$, since A^{-1} is symmetric and v is a unit vector, $u^TA^{-1}v = v^TA^{-1}u$. Plug formula (1) into formula (2) we have:

$$\begin{aligned} & u^TA^{*-1}v \\ &= u^TA^{-1}v - \frac{u^TA^{-1}uv^TA^{-1}v}{1+v^TA^{-1}u} \\ &= \lambda - \frac{\mu A^{-1}[i,i]}{1+\lambda} \end{aligned} \quad (3)$$

$$\begin{aligned} & A^{*-1}vu^TA^{*-1} \\ &= (A^{-1} - \frac{A^{-1}uv^TA^{-1}}{1+v^TA^{-1}u})\, vu^T (A^{-1} - \frac{A^{-1}uv^TA^{-1}}{1+v^TA^{-1}u}) \\ &= A^{-1}vu^TA^{-1} + \frac{A^{-1}uv^TA^{-1}vu^TA^{-1}uv^TA^{-1}}{(1+\lambda)^2} - \frac{A^{-1}uv^TA^{-1}vu^TA^{-1}}{1+\lambda} - \frac{A^{-1}vu^TA^{-1}uv^TA^{-1}}{1+\lambda} \\ &= A^{-1}v(A^{-1}u)^T + \frac{\mu A^{-1}[i,i]}{(1+\lambda)^2}A^{-1}u(A^{-1}v)^T \end{aligned}$$

$$- \frac{A^{-1}[i,i]}{1+\lambda} A^{-1} u (A^{-1} u)^T - \frac{\mu}{1+\lambda} A^{-1} v (A^{-1} v)^T \quad (4)$$

Plug (1), (3) and (4) into (2):

$$A^{**-1} = A^{-1} + aA^{-1} v (A^{-1} u)^T + \mathrm{b} A^{-1} u (A^{-1} v)^T$$
$$+ \mathrm{c} A^{-1} u (A^{-1} u)^T + \mathrm{d} A^{-1} v (A^{-1} v)^T \quad (5)$$

The constants are replaced by letters a, b, c and d for simpler expression. This is an ugly expression, but deceptively simple to be calculated by a computer. We will mention in the later section that, if we remove more rows and columns from the original matrix, we can easily construct a polynomial to represent its inverse matrix by plugging formula (5) into itself recursively.
Thus far, we have come up with our basic algorithm implementation:

1. assemble matrix A based on the conductance matrix $O(n^2)$
2. decompose A into LDL^T $O(n^3)$
3. calculate A^{-1} using backward elimination on L and L^T (D is diagonal, so it's simple) $O(n^3)$
4. for i =1...N: $O(n^2m)$
    a. ε^T = -A[i] except $\varepsilon^T[i] = \frac{1}{2}(1 - A[i,i])$ (ε^T and ε are matrices, we represent it as array for simplicity)
    b. u = ε^T and v = [0,0,...1,...,0] all zero except 1 in the i^{th} position
    c. $A^{-1}v = A^{-1}[i]$
    d. calculate $A^{-1}u$ $O(n^2)$
    e. calculate $\lambda = u^T \cdot A^{-1}v$ $O(n)$
    f. calculate $\mu = u^T \cdot A^{-1}u$ $O(n)$
    g. for j = i+1...N: $O(nm)$
        i. $X = A^{**-1}[j] = A^{-1}[j] + aA^{-1}v[j] \cdot A^{-1}u + bA^{-1}u[j] \cdot A^{-1}v + cA^{-1}u[j] \cdot A^{-1}u + dA^{-1}v[j] \cdot A^{-1}v$ (only calculate the row we need on the fly, both save memory and better for cache locality) $O(n)$
        ii. compute the absolute sum of currents for each node $O(m)$

Computing the absolute sum of current for each node requires O(m), in which, m is the number of branches in the circuit. Since in most cases, m > n, this leads to a $O(n^2m)$ total time complexity. The arrays only use memory of level O(n) which is negligible compared to the $O(n^2)$ usage of the input. Also, the matrix L can be discarded after A^{-1} is acquired. So the maximum memory usage of the algorithm is about twice as the input.
However, the problem is not this simple. Since the elements in the A diagonal A[i,i] = - sum(A[i]), A is a singular matrix and thus does not have an inverse matrix. So we have to get A^{**-1} in a roundabout way: first we calculate $A^{\#-1}$. $A^{\#}$ is the same as A except $A^{\#}[1,1] = 1$, which is still symmetric. Then, we perturb $A^{\#-1}$ first with matrix $\varepsilon_1^T = u_1 v_1^T$, u_1 = [p,0,0,...,1,...,0], in which p = $\frac{1}{2}(A[1,1] - 1)$ and the second "1" lies at position i, v_1 = [1,0,...,0] and then with ε_1. After this step, we still have a symmetric matrix and

can use row operation instead of column operation. Then we can do the two step perturbation as we did before (**Fig. 2**).

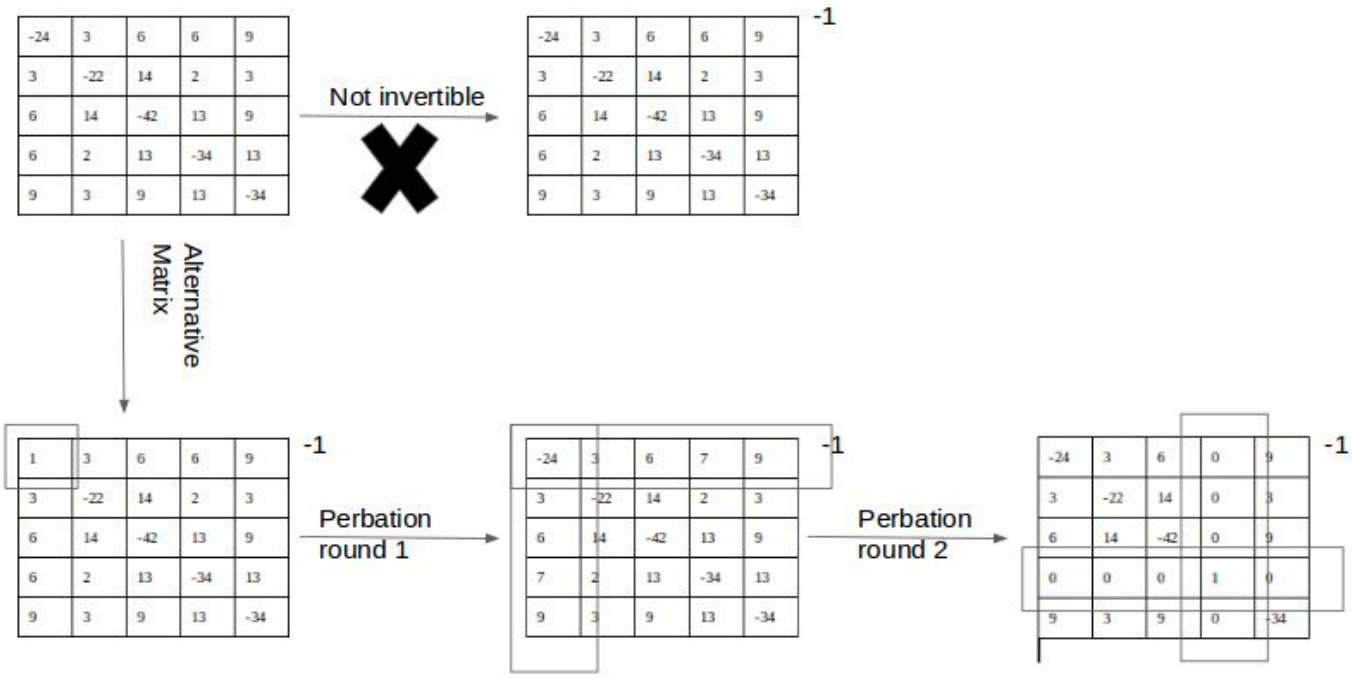

Figure. 2, since A is not invertible, we use a roundabout approach to acquire the matrix we need

Let's call the matrix after first round of perturbation $A^{@@-1}$, we have:

$$
\begin{aligned}
&A^{@@-1} \\
&= A^{\#-1} + a' A^{\#-1} v_1 (A^{\#-1} u_1)^T + b' A^{\#-1} u_1 (A^{\#-1} v_1)^T \\
&+ c' A^{\#-1} u_1 (A^{\#-1} u_1)^T + d' A^{\#-1} v_1 (A^{\#-1} v_1)^T \\
&= A^{\#-1} + [(a'p+c'p^2+b'p+d')A^{\#-1}[1] + (c'p+b')A^{\#-1}[i]]A^{\#-1}[1]^T \\
&+ [(a'+c'p)A^{\#-1}[1] + c'A^{\#-1}[i]]A^{\#-1}[i]^T \quad (6)
\end{aligned}
$$

In this expression:

$$
\begin{aligned}
&\lambda_1 = pA^{\#-1}[1,1] + A^{\#-1}[1,i] \quad (7) \\
&\mu_1 = p^2 A^{\#-1}[1,1] + 2pA^{\#-1}[1,i] + A^{\#-1}[i,i] \quad (8) \\
&A^{\#-1}v_1 = A^{\#-1}[1] \quad (9) \\
&A^{\#-1}u_1 = pA^{\#-1}[1] + A^{\#-1}[i] \quad (10)
\end{aligned}
$$

Using formula (6), we can get the second round of perturbation for example:

$$
\begin{aligned}
&\lambda_2 \\
&= u_2^T A^{@@-1} v_2
\end{aligned}
$$

$$= u_2^T A^{\#-1}[i] + u_2^T[(a'p+c'p^2+b'p+d')A^{\#-1}[1] + (c'p+b')A^{\#-1}[i]]A^{\#-1}[1,i]$$
$$+ u_2^T[(a'+c'p)A^{\#-1}[1] + c'A^{\#-1}[i]]A^{\#-1}[i,i] \quad (11)$$

The other formula can be acquired in the same way. Since the extra calculations related to (6) are in the O(n) level (inside the i loop). All these operations do not increase the constant of the highest order term in time complexity.

Discussion:

We derived algorithms that can solve the information flow analysis problem under $O(n^2m)$ time. In all algorithms, computing the currents between nodes dominates the time complexity. m can be in the order $O(n^2)$ for dense matrices. If we can find another property that describes the information flow without calculating absolute current flow between nodes, this analysis can be even more efficient.

Citations:

[1] Information Flow Analysis of Interactome Networks
[2] Fill-in reduction in sparse matrix factorizations using hypergraphs
[3] Adjustment of an Inverse Matrix Corresponding to a Change in One Element of a Given Matrix