AN ALGORITHM FOR DETERMINING ISOMORPHISM USING LEXICOGRAPHIC SORTING AND THE MATRIX INVERSE

Christopher J. Augeri[1], Barry E. Mullins[1], Leemon C. Baird III[3], Dursun A. Bulutoglu[2], and Rusty O. Baldwin[1]

[1]Department of Electrical and Computer Engineering
[2]Department of Mathematics and Statistics
Air Force Institute of Technology
Wright Patterson Air Force Base, Dayton, Ohio, 45433

{chris.augeri, barry.mullins, dursun.bulutoglu, rusty.baldwin}@afit.edu

[3]Department of Computer Science
United States Air Force Academy
United States Air Force Academy, Colorado Springs, Colorado, 80840

leemon.baird@usafa.edu

Abstract

The PageRank algorithm perturbs the adjacency matrix defined by a set of web pages and hyperlinks such that the resulting matrix is positive and row-stochastic. Applying the Perron-Frobenius theorem establishes that the eigenvector associated with the dominant eigenvalue exists and is unique. For some graphs, the PageRank algorithm may yield a canonical isomorph. We propose a ranking method based on the matrix inverse. Since the inverse may not exist, we apply two isomorphism-preserving perturbations, based on the signless Laplacian, to ensure that the resulting matrix is diagonally dominant. By applying the Gershgorin Circle theorem, we know this matrix must have an inverse, namely, a set of vectors unique up to isomorphism. We concatenate sorted rows of the inverse with its unsorted rows, lexicographically sort on the concatenated matrix, and apply the ranking as an induced permutation on the input adjacency matrix. This preliminary report shows IsoRank identifies most random graphs and always terminates in polynomial time, illustrated by the execution run times for a small set of graphs. IsoRank has been applied to dense graphs of as many as 4,000 vertices.

Keywords: graph isomorphism, lexicographic sorting, inverse, PageRank

This research is supported in part by the Air Force Communications Agency (AFCA). The views expressed in this paper are those of the authors and do not reflect the official policy or position of the United States Air Force (USAF), Department of Defense (DoD), or the U.S. Government.

1 Introduction

1.1 Motivation

In our work, we study weighted graphs derived by computing the pair-wise distances of n vertices distributed in k-dimensional (k-D) space. For instance, if the 3-D geographic coordinates of an unmanned aerial vehicle (UAV) swarm are given as input, these coordinates would be mapped a 2-D distance matrix, i.e., a weighted graph, by an arbitrary distance metric, such as their Euclidean distance. The algorithm we describe herein accepts weighted graphs (distance matrices); however to simplify our discussion, we assume the resulting 2-D distance matrix is a symmetric $\{0,1\}$ matrix whose main diagonal is everywhere zero, i.e., it is the adjacency matrix, **A,** representing a simple and connected graph, G.

Our first objective is to rank the n vertices (UAVs) relative to their importance within the 2-D distance matrix; ideally, this ordering should also be canonical. Since canonically ranking a graph's vertices is equivalent to the difficult problem of determining graph isomorphism, we initially restricted ourselves to finding an ordering that addressed our fundamental objective, ordering vertices with respect to their relative importance. We began by considering spectral algorithms, i.e., those based on the eigen decomposition, as they have been similarly useful when drawing graphs [14] and ranking web pages for search engines [19].

We observed that the PageRank algorithm [19] yields canonical isomorphs for many random graphs. The PageRank algorithm perturbs an input matrix, **A**, such that the resulting matrix, $\mathbf{A}'$, is strictly positive and row-stochastic. By applying the Perron-Frobenius theorem, we know that the eigenvector associated with the leading eigenvalue of such a matrix exists and is unique [22]. PageRank orders vertices on this eigenvector's entries; since an entry may occur multiple times, PageRank does not typically yield a canonical isomorph for an arbitrary graph.

Further work revealed that iteratively applying the PageRank algorithm yields a canonical isomorph more often, where iteration is logarithmic with respect to the number of vertices. We then investigated if sorting lexicographically on an *information matrix*, **X**, e.g., on all eigenvectors, versus on a single vector, further improved performance. We concluded **X** must be unique up to isomorphism, i.e., **X** must satisfy $\mathbf{P}\cdot\mathbf{A}\cdot\mathbf{P}^{-1} \leftrightarrow \mathbf{P}\cdot\mathbf{X}\cdot\mathbf{P}^{-1}$, where **P** is a permutation matrix and $\mathbf{P}^{-1}$ denotes the matrix inverse. One such matrix satisfying this expression, i.e., a matrix that is unique up to isomorphism, is $\mathbf{X} = \mathbf{A}^{-1}$, the matrix inverse of **A**.

Since $\mathbf{A}^{-1}$ may not exist, we apply two *isomorphism-preserving perturbations* and thus obtain a strictly diagonally dominant matrix, $\mathbf{A}'$. The Gershgorin circle theorem can be used to prove a diagonally dominant matrix is positive definite, i.e., that $(\mathbf{A}')^{-1}$ exists [24]. By suitably constructing **X** from $(\mathbf{A}')^{-1}$, ordering a graph's vertices based on iterative lexicographic sorting of **X** yields a canonical isomorph in polynomial time for many graphs, including certain regular graphs.

2 Background

2.1 Deciding Isomorphism

An oft-cited application for an algorithms that decides graph isomorphism is the comparison of two chemicals, i.e., identifying isomers [8][23]. Other uses are locating electrical circuits within larger circuits [18], merging attack trees [17], data mining [11], and validating deployed sensor networks, e.g., by a UAV [5]. The plethora of research in deciding graph isomorphism has been of such extent a classic survey paper is aptly titled "The Graph Isomorphism Disease" [21].

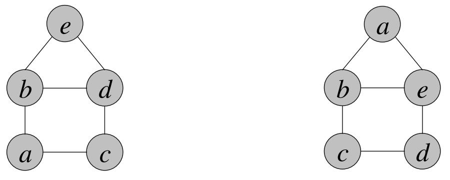
An algorithm for *deciding* graph isomorphism accepts arbitrary graphs; an algorithm for *determining* graph isomorphism fails in one or more instances. Thus, the algorithm described herein, IsoRank, determines isomorphism, e.g., it has difficulty with strongly regular graphs. The key contribution is the simplicity and novelty of the approach, along with the promising, albeit preliminary results. In particular, IsoRank yields canonical isomorphs more often than PageRank, the algorithm we derived it from, and like PageRank, terminates in polynomial time.

2.1.1 Deciding Graph Isomorphism

Two graphs, G_1 and G_2, are mutual isomorphs, denoted $G_1 \cong G_2$, if their edge sets define equivalent relationships on their vertices. Formally, $G_1 \cong G_2$ if and only if a permutation, ϕ, satisfying (1) exists. For each edge, $e_i = \{v_a, v_b\} \in E_1$, an equivalent edge, e_j, exists in E_2, where $v_a, v_b \in V_1$ and $e_j = \{\phi(v_a), \phi(v_b)\}$.

$$\begin{gathered} G_1 \cong G_2 \\ \updownarrow \\ \exists \phi(V_1) = V_2 \text{ s.t.} \\ \forall e_i = \{v_a, v_b\} \in E_1 \ \wedge \ v_a, v_b \in V_1, \\ \exists e_j = \{\phi(v_a), \phi(v_b)\} \in E_2 \end{gathered} \tag{1}$$

For instance, permuting the "house" graph [10] shown in Figure 1(a) by the permutation $\phi = [a \to e, b \to b, c \to a, d \to c, e \to d]$ yields the isomorph shown in Figure 1(b). The difficulty is in finding a suitable ϕ; thus far, the problem of deciding isomorphism remains in NP and is not yet known to be in P.



(a) G_1, random isomorph (b) G_2, another isomorph

Figure 1. Isomorphs of the "houseg Graph

2.1.2 Deciding Matrix Isomorphism

Given the adjacency matrices, $\mathbf{A}_1$ and $\mathbf{A}_2$, of graphs, G_1 and G_2, we similarly can decide whether $\mathbf{A}_1 \cong \mathbf{A}_2$. Formally, $\mathbf{A}_1 \cong \mathbf{A}_2$ if and only if there exists a permutation matrix, $\mathbf{P}$, satisfying (2), where $\mathbf{P}$ is obtained by permuting the columns (rows) of the identity matrix, $\mathbf{I}^{n,n}$, via a permutation, $\phi^{n,1}$, and $n = |V|$.

$$\mathbf{A}_1 \cong \mathbf{A}_2 \leftrightarrow \exists \mathbf{P} \text{ s.t. } \mathbf{A}_2 = \mathbf{P} \cdot \mathbf{A}_1 \cdot \mathbf{P}^T = \mathbf{P} \cdot \mathbf{A}_1 \cdot \mathbf{P}^{-1} \quad (2)$$

The graphs shown in Figure 1 yield the adjacency matrices of Tables 1(a) and 1(b). We satisfy (2) by mapping $\phi = [5,2,1,3,4]$ to a permutation matrix, $\mathbf{P} = \mathbf{I}^{n,n}_{\phi,:}$, as shown in Table 1(c). By comparing all $n!$ permutations of $\mathbf{A}_1$ with $\mathbf{A}_2$, we can equivalently decide matrix isomorphism, i.e., graph isomorphism.

Table 1. Isomorphic adjacency matrices of the house graph

	a	*b*	*c*	*d*	*e*
a	0	1	1	0	0
b	1	0	0	1	1
c	1	0	0	1	0
d	0	1	1	0	1
e	0	1	0	1	0

(a) $\mathbf{A}_1$

	a	*b*	*c*	*d*	*e*
a	0	1	0	0	1
b	1	0	1	0	1
c	0	1	0	1	0
d	0	0	1	0	1
e	1	1	0	1	0

(b) $\mathbf{A}_2$

	5	2	1	3	4
5	0	0	0	0	1
2	0	1	0	0	0
1	1	0	0	0	0
3	0	0	1	0	0
4	0	0	0	1	0

(c) $\mathbf{P}$

2.1.3 Canonical Isomorphs

An approach used in many algorithms for deciding isomorphism is to compute a *canonical isomorph*, $\mathbf{A}_\omega$, where if $[\mathbf{A}_1]_\omega = [\mathbf{A}_2]_\omega$, then $\mathbf{A}_1 \cong \mathbf{A}_2$. For instance, the *minimum canonical isomorph* (MCI) is the isomorph that yields the smallest number, $\text{num}(\mathbf{A})$, if we concatenate consecutive columns of **A**'s upper triangle, i.e., $\text{MCI}(\mathbf{A}) = \min(\text{num}(\mathbf{A}_i)), i \le n!$. Thus, with respect to Table 1, we have that $\text{num}(\mathbf{A}_1) = 1100011101_2$ and that $\text{num}(\mathbf{A}_2) = 1001101101_2$. The MCI of the house graph is shown in Table 2, where $\text{num}(\mathbf{A}_\omega) = 0011101101_2$, obtained by *lexicographically sorting* all $n!$ isomorphs of **A**. To further reinforce this idea, we note the lexicographic MCI of “logarithm” and “algorithm” is “aghilmort”.

Table 2. Minimum canonical isomorph (MCI) of the house graph

	v_1	v_2	v_3	v_4	v_5
v_1	0	0	0	1	1
v_2	0	0	1	0	1
v_3	0	1	0	1	0
v_4	1	0	1	0	1
v_5	1	1	0	1	0

2.1.4 Invariants

An invariant, Ψ, is a necessary, but insufficient, condition for two graphs to be isomorphic, i.e., $\psi_1 = \psi_2$ if $\mathbf{A}_1 \cong \mathbf{A}_2$. It is generally useful to compare a set of invariants of increasing complexity prior to executing a more complex algorithm for computing their canonical isomorphs. A common set of invariants is given in Figure 2, where we assume the *lower* bound on computing $\mathbf{A}_\omega$ is $\Omega\left(n^3 \cdot \log n\right)$ and that all matrices are stored in a dense, i.e., non-sparse, format.

```
function $is\_match = \text{is\_invars\_match}(\mathbf{A}_1, n_1, \mathbf{A}_2, n_2)$
    // $\psi_1$: compare number of vertices, $\Theta(1)$
    // $\psi_2$: compare number of edges, $\Theta(n^2)$
    // $\psi_3$: compare sorted degree sequence, $\Theta(n^2 \cdot \log n)$
    // $\psi_4$: compare eigenvalues, $\Theta(n^3)$
end
```

Figure 2. Comparing graph invariants

2.1.5 A Template Method for Deciding Isomorphism

Invariants and canonical isomorphs provide the machinery to define a template for deciding isomorphism, shown in Figure 3. The difficulty lies in efficiently finding a canonical isomorph. An oft-cited algorithm of choice is *nauty* [16], which computes the MCI of a reduced set of permutations that is obtained via pruning based on discovered automorphisms. A variant of this template approach is to directly match two graphs versus finding a canonical isomorph [4].

```
function $isIso = \text{is\_iso}(\mathbf{A}_1, n_1, \mathbf{A}_2, n_2)$
    // compare invariants
    $isInvarsMatch = \text{is\_invars\_match}(\mathbf{A}_1, n_1, \mathbf{A}_2, n_2)$
    if $(isInvarsMatch)$
        // compute & compare canonical isomorphs
        $[\mathbf{A}_1]_\omega = \text{find\_iso\_canon}(\mathbf{A}_1, n_1)$
        $[\mathbf{A}_2]_\omega = \text{find\_iso\_canon}(\mathbf{A}_2, n_2)$
        if $([\mathbf{A}_1]_\omega = [\mathbf{A}_2]_\omega)$ return $true$ end
    end
    return $false$
end
```

Figure 3. Deciding isomorphism with canonical isomorphs

2.2 The PageRank Algorithm

Our work is motivated by the PageRank algorithm [19], which orders vertices on the dominant eigenvector of a perturbed adjacency matrix that is forced to be positive and stochastic. By the Perron-Frobenius theorem [22], such a matrix has a unique eigenvector that is associated with its largest eigenvalue [1].

Furthermore, it also corresponds to the stationary distribution if the matrix is also a positive single stochastic matrix [22], where a *single stochastic* matrix is a matrix whose rows (columns) sum to one, i.e., $\sum_i \mathbf{A}_{i,:} = 1, \forall i.$ The PageRank algorithm applies an *isomorphism-preserving perturbation*, shown in Figure 4, to the input matrix such that the resulting matrix is a positive single stochastic matrix, where $\alpha \in [0,1]$ scales the non-zero entries and $\delta = (1-\alpha)/n$ is the assumed probability a surfer randomly selects an arbitrary page. The matrix, **D**, is the vertex degree defined previously, i.e., each diagonal entry is equal to the maximum row or column sum of their respective rows (columns). Since **D** is also a diagonal matrix, $\mathbf{D}^{-1}$ is similarly everywhere zero with diagonal entries that are simply the reciprocals of the diagonal entries of **D**.

```
function $\mathbf{A}' = \text{iso\_perturb}(\mathbf{A}, n, \alpha)$
        $\alpha \in [0,1]$
        $\delta = (1-\alpha)/n$
        $\mathbf{D} = \text{diag}\left(\sum_i \mathbf{A}_{i,:}\right)$
        $\mathbf{A}'_{i,j} = \alpha \cdot \mathbf{D}^{-1} \cdot \mathbf{A} + \delta$
end
```

Figure 4. Isomorphism–preserving stochastic perturbation (PageRank)

There are a variety of other ways to obtain a single stochastic or a double stochastic matrix, where a *double stochastic* matrix is a matrix whose rows *and* columns all sum to one, i.e., $\sum_i \mathbf{A}_{i,:} = \sum_i \mathbf{A}_{:,i} = 1, \forall i.$ Two equivalent methods of obtaining a double stochastic matrix are iterated diagonal scaling or Sinkhorn scaling, where $\mathbf{A}_{i+1} = \mathbf{D}_i^{-1/2} \cdot \mathbf{A}_i \cdot \mathbf{D}_i^{-1/2}$ and $\mathbf{A}_{i+1} = \left[\mathbf{A}_i \cdot \mathbf{D}_i^{-1}\right]^T$, respectively. An alternative method of generating a double stochastic matrix from a given matrix uses the graph complement, $\overline{G}$, where

$$\mathbf{A}' = \frac{\mathbf{A} + (n \cdot \mathbf{I} - \mathbf{D})}{n} = \frac{\mathbf{A} + \overline{\mathbf{D}}}{n}, n = |V|, \tag{3}$$

$\overline{\mathbf{A}} = \mathbf{J} - \mathbf{I} - \mathbf{A}$, and $\overline{\mathbf{D}}$ is the corresponding degree matrix of $\overline{\mathbf{A}}$ [13]. Although we do not use stochastic matrices, our research on them motivated us to use $\mathbf{D}^{-1}$ in ensuring that the matrix inverse exists to great effect.

The PageRank algorithm is given in Figure 5, where we first apply the stochastic perturbation, and compute the eigen decomposition. We assume the eigenvectors are ordered by the magnitude of their eigenvalues and extract the leading eigenvector, $\mathbf{U}_{:,n}$. We then concatenate the vertex positions with the leading eigenvector and sort lexicographically, and extract the vertex ordering, **p**.

```
function $\mathbf{v}_{ord} = \text{compute\_ page\_rank}(\mathbf{A}, n, \alpha)$
        $\mathbf{A}' = \text{iso\_perturb}(\mathbf{A}, n, \alpha)$
    $\mathbf{U} \cdot \Lambda \cdot \mathbf{U}^{-1} = \mathbf{A}'$
        $\mathbf{x} = \mathbf{U}_{:,n}$
        $\mathbf{n} = [1, 2, \ldots, n]^T$
        $\mathbf{S} = \text{lex\_sort}([\mathbf{x} \quad \mathbf{n}], [1])$
        $\mathbf{v}_{ord} = \mathbf{S}_{:,2}$
end
```

Figure 5. PageRank algorithm

A MATLAB implementation is listed in Figure 6. Rounding on line 14 is due to the use of finite precision (the *roundn* function is in the mapping toolbox). The *sortrows* function performs lexicographic sorting on line 17.

```
1.  function [tA] = iso_perturb(A, n, a)
2.    % compute degree matrix
3.    D = diag(sum(xA));
4.
5.    % compute transform
6.    tA = a * D^(-1) * A + (1 - a) / n;
7.  end
8.
9.  function [p] = compute_page_rank(A, n, a)
10.   tA = iso_perturb (A, n, a);
11.
12.   % compute leading eigenvector
13.   [U, V] = eig(tA);
14.   x = roundn(U(:, n), -15);
15.
16.   % sort lexicographically
17.   S = sortrows([x, [1:n]'], [1]);
18.
19.   % extract vertex ordering
20.   p = S(:, 2);
21. end
```

Figure 6. PageRank algorithm (MATLAB source code)

3 Fundamental Constructs

This section describes several key abstractions of PageRank that greatly aided IsoRank's development. We recall we are interested in computing a canonical isomorph, $\mathbf{A}_\omega$. Thus, key idea is to apply the induced permutation yielded by a vertex ordering algorithm, such as PageRank, to the input adjacency matrix, **A**, as shown in Figure 7. The equivalent MATLAB source code is listed in Figure 8, where lines 3–5 are replaced with "`Aomega = A(phi, phi);`" in practice.

```
function $\mathbf{A}_\omega = \text{find\_isomorph}(\mathbf{A}, n, a)$
        $\phi_\omega = \text{compute\_page\_rank}(\mathbf{A}, n, a)$
        $\mathbf{P}_\omega = \mathbf{I}^{n,n}(\phi,:)$
        $\mathbf{A}_\omega = \mathbf{P}_\omega \cdot \mathbf{A} \cdot \mathbf{P}_\omega^T$
end
```

Figure 7. Applying an induced pPermutation

```
1.  function [Aomega] = find_iso(A, n, a)
2.    phi = compute_page_rank(A, n, a);
3.    I = eye(n);
4.    Pomega = I(phi, :);
5.    Aomega = Pomega * A * Pomega';
6.  end
```

Figure 8. Applying an induced permutation (MATLAB source code)

3.1 Information Matrices

The PageRank algorithm computes only one eigenvector; this is primarily driven by the fact that the Perron-Frobenius theorem only guarantees the existence of a single eigenvector. Thus began our search for a more robust set of vectors that we refer to as an information matrix. The first information matrix we considered was the entire set of eigenvectors; however, they did not significantly improve our ability to find a canonical isomorph. Previous work has also considered such information matrices; the eigenvectors are a frequent candidate [6][13].

We conjectured an ideal information matrix would be unique up to isomorphism, i.e., $\mathbf{P} \cdot \mathbf{A} \cdot \mathbf{P}^T \leftrightarrow \mathbf{P} \cdot \mathbf{X} \cdot \mathbf{P}^T$, where **X** is the desired information matrix. One such matrix is the all-pairs shortest path (APSP) distance matrix, which is obtainable in $\mathrm{O}(n^3)$ time. This led us to consider similar matrices computable in $\mathrm{O}(n^3)$ time, most notably, the pseudoinverse, $\mathbf{A}^\dagger$, and the matrix inverse, $\mathbf{A}^{-1}$, but several issues preclude the immediate use of either. First, the pseudoinverse, although it always exists and has been used in other algorithms for determining isomorphism [2], it may yield provide less than our goal of *n* information vectors and can numerically difficult to compute. The inverse may simply not exist; a key result of our work is how we perturb **A** such that $\mathbf{A}^{-1}$ is guaranteed to exist.

3.2 Isomorphism-Preserving Perturbations

A *graph perturbation* (matrix perturbation), induces changes on the underlying graph (matrix), e.g., by adding random edges between arbitrary vertices. If $G_1 \cong G_2$, an *isomorphism-preserving perturbation* yields $G_1' \cong G_2'$, where G' is a perturbed graph (matrix), e.g., adding a loop to all vertices. Such a perturbation should increase computing efficiency, increase our ability to find $\mathbf{A}_\omega$, decrease the condition number, $\kappa(\mathbf{A})$, and be invertible, i.e., G is obtainable from G'. We use isomorphism-preserving perturbations to ensure the information matrix of interest, $(\mathbf{A}')^{-1}$, exists, where $\mathbf{A}'$ is obtained by perturbing $\mathbf{A}$.

3.2.1 Distinguishing Graphs by Vertex Augmentation

A simple isomorphism-preserving perturbation for ensuring connectivity is to add a vertex to the graph, which we call the β-vertex, and an edge between this vertex and all existing vertices. This perturbation appears in many contexts and has been shown to aid distinguishing the eigenvalues of non-isomorphic graphs, however, it does not serve as a complete invariant [20], Section 4.5.5 in [6]. We observed a similar effect—even for connected graphs, adding a single vertex linked to all vertices improves our ability to find a canonical isomorph. One effect of adding v_β is that it forces the diameter to be either one or two. Thus, we have $G_\beta = (V_\beta, E_\beta)$, where $V_\beta = V \cup \{v_\beta\}$ and $E_\beta = E \cup \{\{v_\beta, v_i\}\}, \forall v_i \in V$. Therefore, $n_\beta = |V_\beta(G_\beta)| = |V(G)| + 1 = n + 1$, and $|E_\beta(G_\beta)| = |E(G)| + |V(G)|$. This perturbation is akin to adding a '1's column to ensure a y-intercept, i.e., β_0, in linear regression, hence are dubbing this as the β-vertex perturbation (4).

$$\mathbf{A}' = \mathbf{A}_\beta = \begin{bmatrix} \mathbf{0}^{1,1} & \mathbf{1}^{1,n} \\ \mathbf{1}^{n,1} & \mathbf{A} \end{bmatrix} \tag{4}$$

3.2.2 Ensuring Invertibility by Diagonal Dominance

The pseudoinverse, $\mathbf{A}^\dagger = (\mathbf{A}^T \cdot \mathbf{A})^{-1} \cdot \mathbf{A}^T$, is often used in linear regression, always exists; furthermore, $\mathbf{A}^\dagger = \mathbf{A}^{-1}$ (if $\mathbf{A}^{-1}$ exists). To ensure $\mathbf{A}^{-1}$ exists, where $\mathbf{A} \cdot \mathbf{A}^{-1} = \mathbf{I}$, we apply an isomorphism-preserving perturbation based on spectral graph theory [3], where the *Laplacian*, $\mathbf{L} = \mathbf{D} - \mathbf{A}$, is often studied and $\mathbf{D}$ is the degree matrix of $\mathbf{A}$. We are interested in the *signless Laplacian* [12], $\mathbf{L}^+ = \mathbf{D} + \mathbf{A}$. It is known either Laplacian is positive semi-definite, i.e., they do not always have an inverse. We propose the *modified* signless Laplacian, of the form $\mathbf{L}^{+\varepsilon} = \mathbf{D} + \mathbf{A} + \vec{\varepsilon} \cdot \mathbf{I}$, where $\vec{\varepsilon}$ is a vector of constants. We initially focused on $\mathbf{L}^{+\varepsilon} = \mathbf{D} + \mathbf{A} + \mathbf{I}$, however, $\mathbf{L}^{+\varepsilon} = \mathbf{D} + \mathbf{A} + \mathbf{D}^{-1}$ yields the best results. Since $\mathbf{L}^{+\varepsilon}$ is diagonally dominant, it is positive definite and invertible!

3.3 Potential Equivalent Vertex Grouping

A key reason for using the inverse as the source of our information matrix, **X**, is it can group potentially equivalent vertices, i.e., since $\mathbf{P}\cdot\mathbf{A}\cdot\mathbf{P}^T \leftrightarrow \mathbf{P}\cdot\mathbf{A}^{-1}\cdot\mathbf{P}^T$. This is based on two ideas: first, the inverse of a matrix is unique up to isomorphism and second, within each row (column) the sorted entries of that row (column) are unique. Namely, two vertices that are in the same orbit must share identical entries in their corresponding rows (columns) of the inverse. Since the inverse of a matrix has *n* vectors, the sorting of each vector within the inverse is computable in $\mathrm{O}\left(n^2\cdot\log n\right)$ time, if we assume that the sorting is done by an implementation of *quicksort*, e.g, as in MATLAB's *sort* function.

3.4 Lexicographic Sorting

We previously introduced lexicographic sorting in the context of the MCI, e.g., we recall the MCI of "logarithm" is the sorted string "aghilmort". We assume we have obtained an information matrix, **X**, from the inverse, $\left(\mathbf{A}'\right)^{-1}$, of a perturbed matrix, $\mathbf{A}'$, based on the adjacency matrix, **A**, of a graph, *G*, i.e., **X** is unique up to isomorphism. For instance, if Table 3(a) is **X**, sorting each row from left to right, as shown in Table 3(b), reveals vertices {*a*, *c*} *may* be in the same orbit, since they both share entries $[3,3,7,7]$. Lexicographically sorting on Table 3(b) augmented with the identity vector, $\phi=[1,2,\ldots,n]$, yields Table 3(c) along with an induced permutation on the identity vector, ϕ.

Table 3. Individual row and lexicographic sorting of an information matrix

(a) Raw matrix

	r	*s*	*t*	*u*
r	7	5	3	7
s	5	9	5	3
t	3	5	7	7
u	7	3	7	3

(b) Row sorting (L → R)

	r	*s*	*t*	*u*	ϕ
r	3	5	7	7	1
s	3	5	5	9	2
t	3	5	7	7	3
u	3	3	7	7	4

(c) Lexicographic sorting

	r	*s*	*t*	*u*	ϕ
u	3	3	7	7	4
s	3	5	5	9	2
t	3	5	7	7	3
r	3	5	7	7	1

The MATLAB *sortrows* function lexicographically sorts a matrix by rows and allows us to specify the columns to sort on and uses the same *quicksort* implementation as the *sort* function. The underlying *quicksort* implementation is *stable*, i.e., two equal elements retain their original *relative* positions after being s Assuming pair-wise comparisons are used, the *quicksort* algorithm's complexity is $\Theta\left(n\cdot\log n\right)$ in the worst case to sort arbitrary data. If we assume *n* columns are sorted on and that a complete pair of rows may be swapped during a comparison, i.e., that this is a non-pointer based implementation, then sorting lexicographically via *quicksort*, i.e., *sortrows*, is $\mathrm{O}\left(n^3\cdot\log n\right)$. Some efficiency can be gained via by using the underlying *sortrowsc* function, which only returns the induced permutation versus the lexicographically sorted $n\times n$ matrix.

3.5 Iterative Ranking

For a variety of reasons, e.g., numerical conditioning, we considered iteration to further improve performance. Loosely stated, if computing the inverse, sorting lexicographically on it, and applying the induced permutation are effective once, are more iterations beneficial? One question is how much iteration is useful? A series of permutations creates a *permutation chain*, $\mathbf{P}_\omega = \mathbf{P}_m \cdot \cdots \cdot \mathbf{P}_2 \cdot \mathbf{P}_1$, where

$$\mathbf{A}_\omega = \mathbf{P}_\omega \cdot \mathbf{A} \cdot \mathbf{P}_\omega^T = \mathbf{A}_1 \xrightarrow{\mathbf{P}_1} \mathbf{A}_2 \xrightarrow{\mathbf{P}_2} \cdots \xrightarrow{\mathbf{P}_m} \mathbf{A}_m \tag{5}$$

and $\mathbf{A}_i \to \mathbf{A}_{i+1}$ denotes $\mathbf{A}_{i+1} = \mathbf{P}_i \cdot \mathbf{A}_i \cdot \mathbf{P}_i^T$. If we obtain $\mathbf{P}_i$ randomly, $\mathbf{A}_\omega$ is determined by a random process. However, by computing $\mathbf{P}_i$ deterministically, we can, for some *k* and some *m*, decompose the permutation chain via

$$\mathbf{A}_\omega = \underbrace{\mathbf{A}_1 \xrightarrow{\mathbf{P}_1} \cdots \xrightarrow{\mathbf{P}_{k-1}} \mathbf{A}_k}_{\text{limit sequence}} \xrightarrow{\mathbf{P}_k} \underbrace{\mathbf{A}_{k+1} \xrightarrow{\mathbf{P}_{k+1}} \cdots \xrightarrow{\mathbf{P}_{m-1}} \mathbf{A}_m}_{\text{limit cycle}} \underbrace{\xrightarrow{\mathbf{P}_{m+1}} \mathbf{A}_{k+1} \xrightarrow{\mathbf{P}_{k+1}} \cdots}_{\text{cycling!}}, \tag{6}$$

where a *limit cycle* is the permutation sequence, that after being deterministically reached, e.g., by lexicographic sorting, repeats and a *limit sequence* is the set of permutations traversed to reach a limit cycle. By using a limit cycle's MCI as its terminal isomorph (*attractor)*, $\mathbf{A}_\omega$, we have an *attractor set*, $\mathbf{A}_\Omega = \left\{\mathbf{A}_{\omega_1}, \mathbf{A}_{\omega_2}, \ldots\right\}$. For an iterative approach to be useful, the limit sequences (cycles) must be short and the attractor set must be small, i.e., $\left|\mathbf{A}_\Omega\right| \ll n!$.

4 IsoRank: Ordering Vertices on the Matrix Inverse

The IsoRank algorithm is presented in Figure 9. Broadly stated, the algorithm applies isomorphism-preserving perturbations to the adjacency matrix, computes the inverse of the perturbed matrix, lexicographically sorts on the information matrix yielded by the inverse, and applies the induced permutation to the input adjacency matrix. The most expensive computations are obtaining the inverse and sorting lexicographically, which are $\mathrm{O}(n^3)$ and $\mathrm{O}(n^3 \cdot \log n)$, respectively. This process may iterate for as many as $\left\lceil \log_2(n+1) \right\rceil + 1$ iterations and we track two previous iterations; reasons for this are presented in our results discussion. Thus, the IsoRank algorithm executes in $\mathrm{O}(n^3 \cdot \log^2 n)$ time, if using numerical libraries, e.g., those used in MATLAB. As will be discussed, this complexity can be reduced significantly by a rather large factor with a more efficient design.

Perhaps the most critical step of the algorithm is shown on lines 14–15, where we round entries of the inverse we have obtained. Since we are using finite precision, and since we are sorting on these entries, it is critical that theoretically identical entries are also numerically identical. Although rounding handles many numerical problems we encounter, this step is an active area of our research.

```
1.  function $\mathrm{A}_\omega = \text{compute\_iso\_rank}(\mathbf{A}, n, t)$
2.      $n_\beta = n + 1$
3.      $\mathbf{A}_{\omega\_old\_2} = \mathbf{A}_{\omega\_old\_1} = \mathbf{A}_\omega = \mathbf{A}$
4.      // iterate based on base-2 logarithm relative to size of vertex set
5.      for $i = 1$ to $\left(\left\lceil \log_2(n_\beta) \right\rceil + 1\right)$ do
6.          // add beta vertex
7.              $\mathbf{A}_\beta = \begin{bmatrix} \mathbf{0}^{1,1} & \mathbf{1}^{1,n} \\ \mathbf{1}^{n,1} & \mathbf{A}_\omega \end{bmatrix}$
8.          // form modified signless Laplacian to ensure inverse exists
9.              $\mathbf{A}_\beta^{+\varepsilon} = \mathbf{A}_\beta + \mathbf{D}_\beta + \mathbf{D}_\beta^{-1}$
10.         // compute source of information matrix, i.e., the inverse
11.             $\mathbf{S}_\beta = \left(\mathbf{A}_\beta^{+\varepsilon}\right)^{-1}$
12.         // remove corresponding row, but not column, of beta vertex
13.             $\mathbf{S}'_\beta = \mathbf{S}_\beta(2 : n_\beta, :)$
14.         // round entries before sorting due to finite precision
15.             $\mathbf{T}_\beta = \text{round}(\mathbf{S}'_\beta, t)$
16.         // sort individual rows of source information matrix
17.             $\mathbf{T}'_\beta = \text{sort\_row\_vectors}(\mathbf{T}_\beta)$
18.         // construct information matrix (row-sorted + raw inverse)
19.             $\mathbf{X}_\beta = \begin{bmatrix} \mathbf{T}'_\beta & \mathbf{T}_\beta & \mathbf{I}^{n,n} \end{bmatrix}$
20.         // sort information matrix lexicographically
21.             $\mathbf{X}'_\beta = \text{sort\_cols\_lexically}\left(\mathbf{X}_\beta, \left[1 : (2 \cdot n_\beta)\right]\right)$
22.         // extract induced permutation matrix
23.             $\mathbf{P}_\omega = \mathbf{X}_\omega\left(:, (2 \cdot n_\beta + 1) : (2 \cdot n_\beta + n)\right)$
24.         // permute adjacency matrix
25.             $\mathbf{A}_\omega = \mathbf{P}_\omega \cdot \mathbf{A}_\omega \cdot \mathbf{P}_\omega^T$
26.         // check for limit cycle lengths {1,2}, i.e., terminal isomorph
27.             if $\left(\mathbf{A}_\omega \equiv \mathbf{A}_{\omega\_old\_1} \vee \mathbf{A}_\omega \equiv \mathbf{A}_{\omega\_old\_2}\right)$ break($i$) end
28.             $\mathbf{A}_{\omega\_old\_2} = \mathbf{A}_{\omega\_old\_1}$
29.             $\mathbf{A}_{\omega\_old\_1} = \mathbf{A}_\omega$
30.     end
31. end
```

Figure 9. The IsoRank algorithm

5 Implementation Optimization

The IsoRank algorithm is $\mathrm{O}\left(n^3 \cdot \log^2 n\right)$ if implemented using numerical linear algebra libraries. A variety of improvements reduce complexity by a large factor.

5.1 Faster Permutations and Inversions

Given an orthogonal matrix, e.g., a permutation matrix, $\mathbf{P}$, its inverse is defined by $\mathbf{P}^{-1} = \mathbf{P}^T$; this reduces computing $\mathbf{P}^{-1}$ from $\mathrm{O}\left(n^3\right)$ to $\mathrm{O}(n)$. Thus, we can obtain a permutation, $\mathbf{P} \cdot \mathbf{A} \cdot \mathbf{P}^{-1}$, by $\mathbf{P} \cdot \mathbf{A} \cdot \mathbf{P}^T$ and again, since $\mathbf{P}$ is sparse, reduce this complexity from $\mathrm{O}\left(n^3\right)$ to $\mathrm{O}\left(n^2\right)$. Furthermore, we can augment the information matrix with the identity matrix, $\mathbf{I}$, versus an identity vector, $\mathbf{n} = [1, 2, \ldots, n]^T$, reducing the size of the matrix being sorted on by a factor of *n*.

Perhaps most significantly, since permuting a matrix permutes its inverse, i.e., $\left[\mathbf{P} \cdot \mathbf{A} \cdot \mathbf{P}^{-1}\right]^{-1} = \mathbf{P} \cdot \mathbf{A}^{-1} \cdot \mathbf{P}^{-1}$, by the last result, we only need to compute $\left[\mathbf{A}'_\beta\right]^{-1}$ and permute it after each of the first $\left\lceil \log_2 (n+1) \right\rceil$ iterations. Furthermore, since $\mathbf{A}'_\beta$ is positive definite, we can use Cholesky decomposition to obtain $\left[\mathbf{A}'_\beta\right]^{-1}$ at an approximate cost of $f(n) = 1/6 \cdot n^3$ [7]. Finally, we know that $\mathbf{D}$, the degree matrix, is a diagonal matrix, and thus we only need to reciprocate its *n* diagonal entries to obtain $\mathbf{D}^{-1}$, reducing this computation from $\mathrm{O}\left(n^3\right)$ to $\mathrm{O}(n)$.

5.2 Implementation-Specific Issues

There are several implementation issues to consider in MATLAB, the following ones have yielded the most significant improvement: using vectors in lieu of for loops, calling the *sortrowsc* function versus *sortrows*, using sparse matrices if applicable, and using *linsolve* if operating on dense matrices. We currently use the β-vertex perturbation to process graphs with multiple components—this can be significantly improved by pre-processing the graph to separate components.

5.3 Leveraging Parallel Libraries

We have scaled up to 8 processors using the Intel BLAS libraries provided with MATLAB. These libraries are accessed by setting the "BLAS_VERSION" and "OMP_NUM_THREADS" environment variables to specify the BLAS library and number of CPUs, e.g., "mkl_p4.dll" and "2", respectively.

5.4 Using Symbolic Libraries

A different type of performance issue arises from the use of finite precision. We have used three symbolic libraries: the Maple engine in the symbolic toolbox, Mathematica, and the Gnu Multiple Precision (MP) library. The use of symbolic libraries particularly benefits from the suggestions in Section 5.1.

6 Results

To evaluate these ideas, we constructed 1,024 x {16, 64}-vertex random graphs using $\Pr(e_i) = 0.5$ and $\Pr(e_i) \in [1/n, 1]$ along with two isomorphs of each test graph. Each entry in the tables below reflect the number of pairs successfully identified, i.e., ideally, 1024 pairs. These are small, easy graphs; we observe similar results on various regular graphs, e.g., ladders and Mobiüs ladders, random regular graphs, and Paley graphs [9]. In addition, we have tested variants of IsoRank on dense (sparse) graphs having many as 4,000 (40,000) vertices.

6.1 Eiegenvectors (one iteration)

Table 4 shows sorting on a single eigenvector, as in PageRank, finds a canonical isomorph for ~50% of the graph pairs. Although overall performance decreases if $\Pr(e_i) \in [1/n, 1]$, sorting on all eigenvectors has a slight very advantage. This experiment did not apply the β-vertex or signless Laplacian perturbation.

Table 4. One versus All Eigenvectors (1 iteration)

Pr($e_i \in E$)	**0.5**		**[1/*n*, 1]**	
Number of Vertices	**16**	**64**	**16**	**64**
One eigenvector	509	515	483	498
All eigenvectors	509	515	506	502

6.2 Eiegenvectors ($\lceil \log_2(n+1) \rceil + 1$ iterations)

Iterating by as many as $\log_2(n)+1$ iterations improves the chances of finding a canonical isomorph from ~50% to ~75%, as shown in Table 5. This experiment also did not leverage the β-vertex or signless Laplacian perturbation.

Table 5. One versus All Eigenvectors ($\lceil \log_2(n+1) \rceil + 1$ iterations)

Pr($e_i \in E$)	**0.5**		**[1/*n*, 1]**	
Number of Vertices	**16**	**64**	**16**	**64**
One eigenvector	766	763	728	749
All eigenvectors	766	763	778	766

6.3 Using the Inverse and Pseudoinverse

Table 6 reveals sorting on either the pseudoinverse or the inverse dramatically improve performance. We use the β-vertex perturbation to obtain the inverse and the pseudoinverse; we used the signless Laplacian when obtaining the inverse, but not the pseudoinverse. We see IsoRank correctly determines isomorphism, using the inverse for all 1,024 random test pairs.

Table 6. $A^\dagger$ versus A^{-1}, β-vertex, w/iterations

Pr($e_i \in E$)	**0.5**		**[1/*n*, 1]**	
Number of Vertices	**16**	**64**	**16**	**64**
Pseudoinverse	917	1024	798	968
Inverse	1024	1024	1024	1024

6.4 Timing

We recall we are primarily interested in the resulting ranking. Thus, we have not compared our execution times (thus far) with *nauty* or additional algorithms that determine isomorphism. We provide a sample of our execution times using a reasonably efficient implementation in Table 7, where all times are in seconds. The experiments were conducted on an 8-way Intel Xeon machine operating at 3.00 GHz with 3.00 GB of RAM. The graphs were all dense random graphs of 50% edge density, i.e., $\Pr(e_i) = 0.5$. We generated 100 test pairs for $n = \{1, 2, 4, 8, 16, 32, 64, 128, 256\}$ and 10 pairs for $n > 256$. This is not a rigorous analysis; it is simply intended to provide an idea of IsoRank's execution time. All times are for computing/comparing canonical isomorphs of two input graphs.

Table 7. Average Execution Times on Random Graphs, $\Pr(e_i) = 0.5$
(time in seconds to compute and compare A_ω given two random isomorphs)

	Number of CPUs for BLAS Libraries			
$n = \lvert V \rvert$	**1**	**2**	**4**	**8**
1	0.0016	0.0017	0.0020	0.0016
2	0.0022	0.0023	0.0023	0.0015
4	0.0023	0.0023	0.0023	0.0016
8	0.0022	0.0022	0.0023	0.0015
16	0.0027	0.0030	0.0031	0.0032
32	0.0053	0.0041	0.0042	0.0031
64	0.0099	0.0112	0.0120	0.0109
128	0.0330	0.0314	0.0327	0.0313
256	0.1599	0.1467	0.1517	0.1470
512	0.9062	0.7998	0.8002	0.7828
1024	5.4764	4.6922	4.5957	4.3407
2048	29.0875	24.6955	22.0735	20.0425
4096	167.5541	124.8512	108.5633	99.1695

6.5 Symbolic Testing and Iteration Limits

We have tested IsoRank using symbolic libraries on graphs of up to eight vertices (12,598 unique graphs). By linking a symbolic library to IsoRank, we found the same canonical isomorph for all but seven of these graphs; adding the equivalent vertex grouping feature resolved this issue, i.e., IsoRank can determine isomorphism of graphs with eight or fewer vertices. Although for small graphs, this experiment aided in verifying IsoRank's correctness.

Omitted analysis reveals at most $\lceil \log_2(n+1) \rceil + 1$ iterations are present in a limit sequence and limit cycles are of length 1 or 2, including all tested "pathological" graphs, e.g., challenge graphs [15] and those based on Hadamard matrices [16]. The attractor set, unfortunately, can be quite large for pathological graphs.

7 Conclusions

We present a polynomial-time algorithm for ranking the vertices of a graph in a numerically stable manner. In many instances, this ranking is also canonical, i.e., it can be used to determine graph isomorphism. If we assume the input graph is non-pathological, i.e., that it is not strongly regular, then the most critical issue affecting our results are the condition number, $\kappa(\mathbf{A})$, and the numerical stability of the algorithm computing $\mathbf{A}^{-1}$. We are researching ways to improve $\kappa(\mathbf{A})$ by another isomorphism-preserving perturbation or using iterative solvers. For example, we some measurable differences in our results if we sort in ascending versus descending order. We are also exploring three areas beyond the impacts of numerical conditioning.

The first area lies at the heart of our reason for tackling this problem: finding an efficient method of canonically ranking a large sensor network, containing many tens of thousands of nodes. To this end, we are exploring IsoRank's behavior if we only sort on the vector associated with the *β*-vertex, versus computing the entire inverse. This immediately reduces the complexity by a factor of *n*, renders iterative methods for solving such a system more attractive, and when coupled with sparse matrices, enables us to tackle relatively large systems. i.e., we can test trees having 30,000 vertices (in under one second).

The fundamental graph-theoretic issue we are exploring, assuming we calculate the entire inverse, are useful and correct methods to map IsoRank into more traditional approaches of determining graph isomorphism. The primary route we are considering is using the inverse of the propsed modified signless Laplacian as the selection method using a traditional backtracking approach. This aspect of the research also involves some work involving coloring nodes in a deterministic manner and (re)-computing the inverse based on this coloring. In particular, we are exploring how many colors are needed to ensure the inverse has unique rows.

The last area we are studying is related to our original problem: linearizing a set of *k*-dimensional points by ranking vertices based on their 2-D distance matrix. This has implications related to data storage, logistics, and network security. We are at an early stage of this portion of the research and welcome any suggestions.

We close by emphasizing the simplicity of the IsoRank algorithm: by adding a single vertex, v_β, linking it to all other existing vertices, adding the degree sum and its reciprocal to the diagonal of this new adjacency matrix, and applying up to $\log_2(n+1)+1$ induced permutations, obtained by lexicographically sorting on the inverse of the perturbed matrix, the IsoRank algorithm ranks vertices in polynomial time, and often yields a canonical isomorph.

Acknowledgments

We thank Dave Doak for support of the symbolic and parallel computation tests. We also thank Terry McKee for providing us a reference to the house graph.

8 References

[1] L. Beineke, and R. Wilson, Eds. *Topics in Algebraic Graph Theory*. Cambridge University Press, 2004.

[2] J. Bennett and J. Edwards. A graph isomorphism algorithm using pseudoinverses. *BIT Numerical Mathematics*. Springer, 36(1):41–53, 1996.

[3] F. Chung. *Spectral Graph Theory*. Regional Conference Series in Mathematics, American Mathematical Society, 1994, 92.

[4] L. Cordella, P. Foggia, C. Sansone, and M. Vento. An improved algorithm for matching large graphs. In *Proceedings of the 3rd IAPR-TC15 Workshop on Graph-Based Representations in Pattern Recognition*, May 2001.

[5] P. Corke, S. Hrabar, R. Peterson, D. Rus, S. Saripalli, and G. Sukhatme. Autonomous deployment and repair of a sensor network using an unmanned aerial vehicle. In *Proc. of the 4th Int'l Conf. on Robotics and Automation*, IEEE, 2004.

[6] D. Cvetković, P. Rowlinson, and S. Simić. *Eigenspaces of Graphs*. Cambridge University Press, 1997.

[7] B. Datta. *Numerical Linear Algebra and Applications*. Brooks/Cole, 1994.

[8] J. Faulon. Isomorphism, automorphism partitioning, and canonical labeling can be solved in polynomial-time for molecular graphs. *Journal of Chemical Information and Computer Sciences*, 38(3):432–444, 1998.

[9] C. Godsil and G. Royle. *Algebraic Graph Theory*. Springer-Verlag, 2001.

[10] M. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Elsevier, 1980.

[11] S. Greenblatt, T. Coffman, and S. Marcus. Behavioral Network Analysis for Terrorist Detection. In *Emergent Information Technologies and Enabling Policies for Counter-Terrorism*. R. Popp and J. Yen, Eds., IEEE, 2006.

[12] W. Haemers and E. Spence. Enumeration of cospectral graphs. *European Journal of Combinatorics*, 25(2):199–211, 2004.

[13] P. He, W. Zhang, and Q. Li. Some further development on the eigensystem approach for graph isomorphism detection. *Journal of the Franklin Institute*, Elsevier, 342(6):657–673, 2005.

[14] Y. Koren. Drawing Graphs by Eigenvectors. *Computers and Mathematics with Applications*. Elsevier, 49(11):1867–1888, 2005.

[15] R. Mathon. Sample graphs for isomorphism testing. In *Proceedings of the 9th Southeastern Conf. on Combinatorics, Graph Theory, and Computing*. *Congressus Numerantium*, Utilitas Publishing, 1978, 21, 499–517.

[16] B. McKay. Practical Graph isomorphism. In *Proc. of the 10th Manitoba Conference on Numerical Mathematics and Computing. Congressus Numerantium*, Utilitas Mathematical Publishing, 30:45-87, 1981.

[17] P. Ning and D. Xu. Learning attack strategies from intrusion alerts. In *Proceedings of the 10th Conference on Computer and Communications Security* (CCS) (Washington D.C.). ACM Press, 2003, 200-209.

[18] M. Ohlrich, C. Ebeling, E. Ginting, and L. Sather. SubGemini: identifying subcircuits using a fast subgraph isomorphism algorithm. In *Proceedings of the 30th Int'l Conference on Design Automation*, Dallas, TX, ACM, 1993.

[19] L. Page, S. Brin, R. Motwani, and T. Winograd. The *PageRank Citation Ranking: Bringing Order to the Web*. TR 1999-66, Stanford University, Stanford, MA, 1998.

[20] G. Prabhu and N. Deo. On the power of a perturbation for testing non-isomorphism of graphs. *BIT*, Springer, 24(3):302-307, 1984.

[21] R. Read and D. Corneil. The graph isomorphism disease. *Journal of Graph Theory*, 1(1):339–363, 1977.

[22] G. Strang. *Linear Algebra and Its Applications*. Thomson Learning, 1988.

[23] G. Tinhofer and M. Klin. Algebraic Combinatorics in Mathematical Chemistry. *Graph Invariants and Stabilization Methods*. TR TUM-M9902, Technische Universität München, München, Germany, 1999.

[24] R. Varga. *Geršgorin and His Circles*. Springer, 2004.

Appendix A

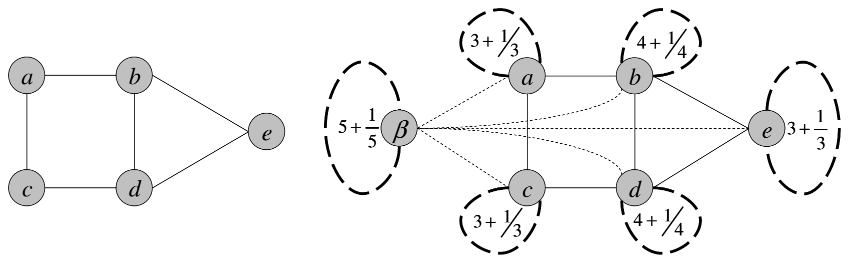
We provide a basic MATLAB implementation of IsoRank in Figure 10. On line 24, we use "`X = tA \ I`", which is more numerically stable and efficient than "`X = tA^(-1)`" or "`X = inv(tA)`". On line 25, we round 14 places to the right of the decimal; the *roundn* function is in the mapping toolbox.

```
1. function [oA] = compute_iso_rank(A, n)
2.    oA = A;
3.    % perform iterations
4.    iters = ceil(log2(n + 1)) + 1;
5.    for i = 1:iters
6.       % compute & apply permutation
7.       pA = compute_iso_perm(oA, n);
8.       oA = oA(pA, pA);
9.    end
10.end
11.
12.function [pA] = compute_iso_perm(A, n)
13.   % add beta vertex
14.   j = ones(n, 1);
15.   bA = [0, j'; j, A];
16.
17.   % force A^(-1) to exist (SDDD)
18.   d = sum(bA);
19.   tA = bA + diag(d + 1./d);
20.
21.   % compute info matrix, A^(-1)
22.   X = tA \ eye(n + 1);
23.
24.   % round information matrix
25.   X = roundn(X, -14);
26.
27.   % sort rows of info matrix
28.   T = sort(X, 2);
29.
30.   % prepare lexicographic sort
31.   augX = [T, X];
32.   nC = 2 * (n + 1);
33.   [S, pA] = sortrows(augX, [1:nC]);
34.
35.   % remove beta vertex permutation entry
36.   iBeta = find(pA(:, 1) == 1);
37.   pA(iBeta, :) = [];
38.   pA = pA – 1;
39.end
```

Figure 10. An IsoRank Implementation (MATLAB)

Appendix B

This appendix applies the IsoRank algorithm to the "house" graph in Figure 1(a). We first observe that $|V| = 5, |E| = 6,$ and $\kappa(G) = 1$. By inspection, we also see that $orb(a) = orb(c) = \{a, c\}$, $orb(b) = orb(d) = \{b, d\}$, and $orb(e) = \{e\}$; vertex labels are used only for convenience. Figure 1(b) shows the result of applying the β-vertex and modified signless Laplacian isomorphism-preserving perturbations. The lighter dotted lines are the edges linking all vertices with v_β. The darker loops linked to a vertex are the result of the modified signless Laplacian; the weight is listed inside each loop, $w(v_i) = \deg(v_i) + 1/\deg(v_i)$.



(a) Original (b) Perturbed (β-vertex / $\mathbf{L}^{+\varepsilon}$)

Figure 1. The "house" graph (original and perturbed)

Table 8 lists the adjacency matrices corresponding with the graphs shown in Figure 1. The β-vertex perturbation adds a row and column, and the row (column) sums are placed on diagonal entries, by way of $\mathbf{A}_\beta^{+\varepsilon} = \mathbf{A}_\beta + \mathbf{D}_\beta + \mathbf{D}_\beta^{-1}$.

Table 8. Adjacency matrices of the "house" graph (original and perturbed)

(a) Original

	a	*b*	*c*	*d*	*e*
a	0	1	1	0	0
b	1	0	0	1	1
c	1	0	0	1	0
d	0	1	1	0	1
e	0	1	0	1	0

(b) Perturbed (β-vertex / $\mathbf{L}^{+\varepsilon}$)

	β	*a*	*b*	*c*	*d*	*e*
β	$5\frac{1}{5}$	1	1	1	1	1
a	1	$3\frac{1}{3}$	1	1	0	0
b	1	1	$4\frac{1}{4}$	0	1	1
c	1	1	0	$3\frac{1}{3}$	1	0
d	1	0	1	1	$4\frac{1}{4}$	1
e	1	0	1	0	1	$3\frac{1}{3}$

The symbolic inverse of the perturbed matrix from Table 8(b) is shown in Table 9. We only use the symbolic inverse here for illustration; it is computed numerically unless conducting experiments to assess the impact of conditioning i.e., of $\kappa(\mathbf{A})$.

Table 9. The inverse of the perturbed "house" graph

	β	a	b	c	d	e
β	$\frac{11490}{49853}$	$-\frac{2370}{49853}$	$-\frac{1220}{49853}$	$-\frac{2370}{49853}$	$-\frac{1220}{49853}$	$-\frac{2715}{49853}$
a	$-\frac{2370}{49853}$	$\frac{1488912}{3938387}$	$-\frac{382068}{3938387}$	$-\frac{455355}{3938387}$	$\frac{216168}{3938387}$	$\frac{1341}{49853}$
b	$-\frac{1220}{49853}$	$-\frac{382068}{3938387}$	$\frac{1153772}{3938387}$	$\frac{216168}{3938387}$	$-\frac{242112}{3938387}$	$-\frac{3096}{49853}$
c	$-\frac{2370}{49853}$	$-\frac{455355}{3938387}$	$\frac{216168}{3938387}$	$\frac{1488912}{3938387}$	$-\frac{382068}{3938387}$	$\frac{1341}{49853}$
d	$-\frac{1220}{49853}$	$\frac{216168}{3938387}$	$-\frac{242112}{3938387}$	$-\frac{382068}{3938387}$	$\frac{1153772}{3938387}$	$-\frac{3096}{49853}$
e	$-\frac{2715}{49853}$	$\frac{1341}{49853}$	$-\frac{3096}{49853}$	$\frac{1341}{49853}$	$-\frac{3096}{49853}$	$\frac{17628}{49853}$

We remove the row of the inverse belonging to the β-vertex; we could remove the permuted position of the β-vertex later; it is convenient to do so here. We typically round to 12–15 digits of precision, but use 3 digits in this example.

Table 10. Removing the β-vertex row and rounding

	β	a	b	c	d	e
a	-0.048	0.378	-0.097	-0.116	0.055	0.027
b	-0.024	-0.097	0.293	0.055	-0.061	-0.062
c	-0.048	-0.116	0.055	0.378	-0.097	0.027
d	-0.024	0.055	-0.061	-0.097	0.293	-0.062
e	-0.054	0.027	-0.062	0.027	-0.062	0.354

Before sorting and to facilitate presentation, we map unique inverse values, $[-0.116, -0.097, \ldots, 0.378]$ to $[1, 2, \ldots, 12]$, respectively, as shown in Table 11.

Table 11. Mapping entries to integers (presentation only)

	β	*a*	*b*	*c*	*d*	*e*
a	6	12	2	1	9	8
b	7	2	10	9	4	3
c	6	1	9	12	2	8
d	7	9	4	2	10	3
e	5	8	3	8	3	11

We now sort the individual rows of the mapped inverse to facilitate orbit grouping, as shown in Table 12. We see that entries are sorted from left to right *within* each row. To save space, we use the "Equal Rows" column in the remainder of this appendix.

Table 12. Sorting entries within each row of the inverse

	β	*a*	*b*	*c*	*d*	*e*	Equal Rows?
a	1	2	6	8	9	12	a, c
b	2	3	4	7	9	10	b, d
c	1	2	6	8	9	12	a, c
d	2	3	4	7	9	10	b, d
e	3	3	5	8	8	11	e

We construct an information matrix by concatenating the "Equal Rows" column of Table 12, Table 11, and the identity matrix, **I**, as shown in Table 13

Table 13. Constructing the information matrix

	Equal	Mapped Inverse						**I**				
	Rows?	*β*	*a*	*b*	*c*	*d*	*e*	*a*	*b*	*c*	*d*	*e*
a	*a, c*	6	12	2	1	9	8	1	0	0	0	0
b	*b, d*	7	2	10	9	4	3	0	1	0	0	0
c	*a, c*	6	1	9	12	2	8	0	0	1	0	0
d	*b, d*	7	9	4	2	10	3	0	0	0	1	0
e	*e*	5	8	3	8	3	11	0	0	0	0	1

We now lexicographically sort the information matrix, the algorithm's key step. The two equivalent vertex pairs appear equivalent with respect to their entries in the "Equal Rows" and β-vertex columns. The ties between $\{a,c\}$ and $\{b,d\}$ are resolved by sorting on column 'a', where entries involved in sorting are shaded. In this example, we thus do not need to sort on columns $[b,c,d,e]$. We observe sorting induces an ordering, i.e., a permutation, on the identity matrix, our primary objective. If we are using the *quicksort* algorithm, then in the worst case, if all columns had to be sorted, this step would terminate in $\mathrm{O}(n^3 \cdot \log n)$ time.

Table 14. Sorting the information matrix lexicographically

	Equal	Mapped Inverse						**P**				
	Rows?	β	a	b	c	d	e	a	b	c	d	e
a	a, c	6	1	9	12	2	8	0	0	1	0	0
b	a, c	6	12	2	1	9	8	1	0	0	0	0
c	b, d	7	2	10	9	4	3	0	1	0	0	0
d	b, d	7	9	4	2	10	3	0	0	0	1	0
e	e	5	8	3	8	3	11	0	0	0	0	1

We then extract the induced permutation and apply it to **A**, i.e., $\mathbf{A}_\omega = \mathbf{P} \cdot \mathbf{A} \cdot \mathbf{P}^T$, as shown in Table 15.

Table 15. Permuting the adjacency matrix to obtain $\mathbf{A}_\omega$

P						**A**						$\mathbf{P}^T$				
0	0	1	0	0		0	1	1	0	0		0	1	0	0	0
1	0	0	0	0	×	1	0	0	1	1	×	0	0	1	0	0
0	1	0	0	0		1	0	0	1	0		1	0	0	0	0
0	0	0	1	0		0	1		0	1		0	0	0	1	0
0	0	0	0	1		0	1	0	1	0		0	0	0	0	1

The result of this permutation, $\mathbf{A}_\omega$, is shown in Table 16. Further iteration yields the same isomorph, thus, we entered a limit cycle of length one on the first iteration, and the limit sequence required to find the limit cycle is of length zero. Since all isomorphs of the "house" graph yield the same terminal isomorph, there is only one terminal isomorph in the limit cycle set, i.e., $\mathbf{A}_\Omega = \{\mathbf{A}_\omega\}$. Thus, we conclude IsoRank finds a canonical isomorph of the house graph, since $|\mathbf{A}_\Omega| = 1$.

Table 16. The resulting canonical isomorph, $\mathbf{A}_{\omega}$

	c	*a*	*b*	*d*	*e*
c	0	1	0	1	0
a	1	0	1	0	0
b	0	1	0	1	1
d	1	0	1	0	1
e	0	0	1	1	0