A Polynomial Time Approximation Algorithm for Graph Isomorphism

游棫荃

May 28, 2023

Abstract

In previous studies, *"It is obvious that $GI \in NP$ but unknown whether $GI \in co-NP$. As that implies, no polynomial time algorithm is known (despite many published claims), but neither is GI known to be NP-complete. NP-completeness is considered unlikely since it would imply collapse of the polynomial-time hierarchy (Goldreich et al., 1991). The fastest proven running time for GI has stood for three decades at $e^{O(\sqrt{n \log n})}$ (Babai et al., 1983)."[1]* However, it is possible to apply the cospectrum property of the adjacency matrix to filter out potential solutions for a given pair of matrices A, B. This paper further investigates the cospectrum and utilizes the Singular Value Decomposition (SVD) to estimate a permutation matrix P such that $PAP^T = B$. Through testing, the predicted accuracy exceeds 90%, and as the value of n (the size of the matrices) increases, the accuracy improves even further, with an accuracy rate of over 99% for n > 10.

Keywords: Graph Isomorphism; Approximation Algorithm; Polynomial Time; Cospectrum; Eigen decomposition; SVD

1 Introduction

Let A the adjacency matrix of G_A and B the adjacency matrix of G_B. G_A and G_B are isomorphic if there is a permutation matrix P with $PAP^T = B$. The adjacency matrices of isomorphic graphs have equal eigenvalues.

the algorithm described in [2] only tests if the graphs have the same eigenvalues. But unfortunately, there a non-isomorphic graphs with the same eigenvalue. In the next section, we will show some basic concepts that we use in the algorithm.

2 Basic Concepts

Isomorphism

Let $G_A \cong G_B$ are isomorphism, Iff $\exists\ P$ is a permutation matrix s.t. $PAP^T = B$.

Theorem. If G_A and G_B are isomorphism then A and B have the same eigenvalues, or call A and B are cospectrum. Note that there are non-isomorphic graphs with the same eigenvalues.

Eigen decomposition (or spectral decomposition)

Let A be a square n × n diagonalizable matrix with n linearly independent eigenvectors v_i (where i = 1, ..., n). Then A can be factorized as $A = V\Lambda V^T$,where V is the square n × n matrix whose i_{th} column is the eigenvector v_i^T of A, and Λ is the diagonal matrix whose diagonal elements are the corresponding eigenvalues, $\Lambda_{ii} = \lambda_i$.

Singular Value Decomposition (SVD)

The singular value decomposition of an m × n complex matrix M is a factorization of the form $M = U\Sigma V^*$ where U is an m × m complex unitary matrix, Σ is an m × n rectangular diagonal matrix with non-negative real numbers on the diagonal, V is an n × n complex unitary matrix.

3 Theoretical Analysis

Our goal is to estimate P using the properties of cospectrum and SVD.

3.1 Diagonalizable Cases

Suppose $G_A \cong G_B$ and A, B

$$A = V_a \Lambda V_a^T, \qquad B = V_b \Lambda V_b^T$$

where V_A, V_B are orthonormal matrices.

then

$$V_a^T A V_a = \Lambda = V_b^T B V_b$$

then

$$V_b V_a^T A V_a V_b^T = B$$

Let

$$Q = V_b V_a^T$$

then

$$QAQ^T = B$$

Let

$$\hat{Q} = [\hat{q}_{ij}],$$

$$where\ \hat{q}_{ij} = 1, if\ |q_{ij}| = ||q_i||_\infty, \qquad else\ \hat{q}_{ij} = 0.$$

We have found that although Q is not necessarily equal to P, if we only retain the extreme positions of each row in Q, there is a high probability that $\hat{Q}$ will be equal to P.

i.e.

$$\hat{Q}A\hat{Q}^T = B$$

3.2 General Cases (SVD)

However, since not all matrices have eigen decompositions, we use the matrices U_A , U_B , V_A and V_B from the singular value decomposition (SVD) instead of the previously mentioned V_A and V_B. Note that the two parts have different meanings for the variable V_{eigen} and V_{SVD}.

$$A = U_A \Sigma V_A^T, \qquad B = U_B \Sigma V_B^T$$

where U_A, U_B, V_A, V_B are unitary matrices.

then

$$U_B U_A^T A V_A V_B^T = B$$

Let

$$Q_U = U_B U_A^T, \qquad Q_V = V_B V_A^T$$

then

$$Q_U A Q_V^T = B$$

We apply the same treatment to Q_U and Q_V,

let

$$\hat{Q}_U = \left[\hat{q}_{U_{ij}}\right],$$

$$where\ \hat{q}_{U_{ij}} = 1, if\ |q_{U_{ij}}| = ||q_{U_i}||_\infty, \qquad else\ \hat{q}_{U_{ij}} = 0,$$

and

$$\hat{Q}_V = \left[\hat{q}_{V_{ij}}\right],$$

$$where\ \hat{q}_{V_{ij}} = 1, if\ |q_{V_{ij}}| = ||q_{V_i}||_\infty, \qquad else\ \hat{q}_{V_{ij}} = 0.$$

There is a high probability that both $\hat{Q}_U$ and $\hat{Q}_V$ will be equal to P.

i.e.

$$\hat{Q}_U A \hat{Q}_U^T = B, \qquad \hat{Q}_V A \hat{Q}_V^T = B$$

Corollary 1. If A and B can be eigen decomposed, and all eigenvalues are non-negative,

$$\left(V_{eigen}, \Lambda, {V_{eigen}}^T\right) = (U_{SVD}, \Sigma, V_{SVD}^T),$$

therefore

$$Q = Q_U = Q_V, \qquad \widehat{Q} = \widehat{Q}_U = \widehat{Q}_V.$$

Note that if we use $(\widehat{Q_U \circ Q_V})$ or $(\widehat{Q_U + Q_V})$ to estimate P, it actually reduces the accuracy. (where $\circ$ is Hadamard product)

4 Adjustments I

Furthermore, we can improve the accuracy by applying additional preprocessing adjustments to matrices A and B. We know that positive semidefinite matrices have the property that all their eigenvalues are non-negative. Since

$$sum(\lambda_i) = trace(A)$$

, we have come up with two ways to adjust the matrices. Moreover, synchronously adjusting the diagonal elements does not affect the permutation matrix P.

4.1 add_n_times_identity (ant)

Although changing the eigenvalues to be non-negative does not affect the V_{eigen} matrix in the eigen decomposition.

$$For\ A \in M_{n\times n}, \qquad A' = A + nI_n.$$

4.2 add_absolute_sum_to_diagonal (aasd)

We add the absolute sum of the corresponding row and column to each diagonal element. This operation significantly improves the accuracy and guarantees non-negative eigenvalues.

$$For\ A \in M_{n\times n},$$

$$A' = [A'_{ij}]$$

$$if\ i \neq j\ A'_{ij} = A_{ij}, \qquad else\ A'_{ii} = A_{ii} + ||A_i|| + ||A_i^T||$$

5 Adjustments II

After SVD, we know that both U and V are unitary matrices. To reduce the non-uniqueness of the SVD decomposition, we make adjustment to U and V.

$$A = U\Sigma V^T$$

$$where\ U = (u_1|u_2|\dots|u_n), \qquad V = (v_1|v_2|\dots|v_n).$$

Let

$$U_c = (c_1u_1|c_2u_2|\dots|c_nu_n), \qquad V_c = (c_1v_1|c_2v_2|\dots|c_nv_n),$$

$$where\ c_i = \pm 1,$$

then

$$A = U_c\Sigma V_c^T$$

is also SVD of A.

5.1 multiply_negative_columns

Consider a case:

$$A = B \; but \; V_A \neq V_B$$

$$V_A = (v_1|v_2| \dots |v_n), \qquad V_B = (\pm v_1|\pm v_2| \dots |\pm v_n)$$

Although

$$V_A^T V_B = \begin{pmatrix} \pm 1 & 0 & \cdots & 0 \\ 0 & \pm 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \pm 1 \end{pmatrix},$$

but

$$Q = V_B V_A^T \neq \begin{pmatrix} \pm 1 & 0 & \cdots & 0 \\ 0 & \pm 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \pm 1 \end{pmatrix},$$

then

$$\hat{Q} \neq I_n$$

To solve the problem, we multiply the negative columns in matrix V by -1.

$$\tilde{V} = (c_1 v_1|c_2 v_2| \dots |c_i v_i| \dots |c_n v_n)$$

$$where \; c_i = \pm 1, \qquad c_i = -1 \; iff \; sum(v_i) < 0.$$

Then

$$\tilde{V}_A^T \tilde{V}_B = I_n$$

and

$$Q = \tilde{V}_B \tilde{V}_A^T = I_n,$$

then

$$\hat{Q} = I_n$$

So as U.

6 Algorithm

6.1 Pseudocode

Testing(A,B)

1. Input $n \times n$ isomorphism matrices A and B.
2. Let $\_\_, S_A, \_\_ = \mathrm{SVD}(A)$ // $A = U_A S V_A^T$
3. Let $\_\_, S_B, \_\_ = \mathrm{SVD}(B)$ // $B = U_B S V_B^T$
4. **if** $S_A \neq S_B$
5. **then return** false
6. $\widehat{Q}_U,\ \widehat{Q}_V = \mathrm{P\_approx}(A, B)$
7. **if** $\widehat{Q}_U A \widehat{Q}_U^T = B$ **|** $\widehat{Q}_V A \widehat{Q}_V^T = B$
8. **then return** true
9. **else**
10. **return** false

P_approx(A,B)

1. P_approx(A, B):
2. $A =$ add_absolute_sum_to_diagonal(A) // aasd
3. $B =$ add_absolute_sum_to_diagonal(B) // aasd
4. Let $U_A, \_\_, V_A = \mathrm{SVD}(A)$ // $A = U_A S V_A^T$
5. Let $U_B, \_\_, V_B = \mathrm{SVD}(B)$ // $B = U_B S V_B^T$
6. $U_A \leftarrow$ multiply_negative_columns(U_A)
7. $U_B \leftarrow$ multiply_negative_columns(U_B)
8. $V_A \leftarrow$ multiply_negative_columns(V_A)
9. $V_B \leftarrow$ multiply_negative_columns(V_B)
10. Let $Q_U = U_B U_A^T$
11. Let $Q_V = V_B V_A^T$
12. $\widehat{Q}_U =$ keep_max_abs_position_per_column(Q_U)
13. $\widehat{Q}_V =$ keep_max_abs_position_per_column(Q_V)
14. **return** $\widehat{Q}_U,\ \widehat{Q}_V$

6.2 Python Code

P_approx's complexity = SVD's complexity = $O(n^3)$ [3]

```python
import numpy as np

def P_approx(A, B):
    # aasd
    A = add_absolute_sum_to_diagonal(A)
    B = add_absolute_sum_to_diagonal(B)


    # SVD Ua@Sa@Va.T = A
    Ua, _, Va = np.linalg.svd(A)
    Va = Va.T
    Ub, _, Vb = np.linalg.svd(B)
    Vb = Vb.T

    Ua = multiply_negative_columns(Ua).real
    Ub = multiply_negative_columns(Ub).real
    Va = multiply_negative_columns(Va).real
    Vb = multiply_negative_columns(Vb).real

    # raw Q
    Qu = (Ub @ Ua.T).real
    Qv = (Vb @ Va.T).real

    #Q = Qu #* Qv
    #print("\n\nraw Q:")
    #print(ignore_small_values((Vb @ Va.T).real, -0.0001))

    Qu = keep_max_abs_position_per_column(Qu)
    Qv = keep_max_abs_position_per_column(Qv)
    return Qu, Qv
```

add_absolute_sum_to_diagonal's complexity = $O(n^2)$

```python
import numpy as np

def add_absolute_sum_to_diagonal(matrix):
    n = matrix.shape[0]  # 矩陣的大小

    # 複製矩陣
    result = np.copy(matrix)

    # 對角線上加上絕對值和
    for i in range(n):
        result[i, i] += np.sum(np.abs(matrix[i, :])) + np.sum(np.abs(matrix[:, i]))

    return result

```

multiply_negative_columns's complexity = $O(n^2)$

```python
import numpy as np

def multiply_negative_columns(matrix):
    column_sums = np.sum(matrix, axis=0)  # 計算每列的總和
    negative_columns = column_sums < 0  # 判斷哪些列總和為負
    result = np.where(negative_columns, -matrix, matrix)  # 如果列總和為負，則同乘-1
    return result
```

keep_max_abs_position_per_column's complexity = $O(n^2)$

```python
import numpy as np

def keep_max_abs_position_per_column(matrix):
    abs_matrix = np.abs(matrix)  # 取得矩陣的絕對值
    result = np.zeros_like(matrix)  # 創建與原矩陣相同形狀的零矩陣

    for i in range(matrix.shape[0]):
        max_idx = np.argmax(abs_matrix[i, :])  # 找到第i行絕對值最大元素的索引
        result[i, max_idx] = 1  # 保留第i行絕對值最大的元素

        # 將第max_idx列的元素設置為0
        abs_matrix[:, max_idx] = 0

    return result
```

generate_matrices (for testing samples)

```python
import numpy as np

def generate_matrices(n):
    # 隨機生成連結矩陣
    A = np.random.randint(0, 2, size=(n, n))
    A = np.triu(A) + np.triu(A, 1).T  # 將矩陣變成對稱

    # 隨機生成排列矩陣
    P = np.eye(n).astype(int)
    np.random.shuffle(P)

    B = (P @ A @ P.T)

    return A, P, B

# 測試
n = 3  # 矩陣的大小
A, P, B = generate_matrices(n)
```

6.3 Computational Complexity

Since SVD's complexity = $O(n^3)$[3],

add_absolute_sum_to_diagonal's complexity = $O(n^2)$

multiply_negative_columns's complexity = $O(n^2)$

multiply_negative_columns's complexity = $O(n^2)$

keep_max_abs_position_per_column's complexity = $O(n^2)$

Testing's complexity = P_approx's complexity = SVD's complexity = $O(n^3)$ [3]

7 Performance

7.1 Testing

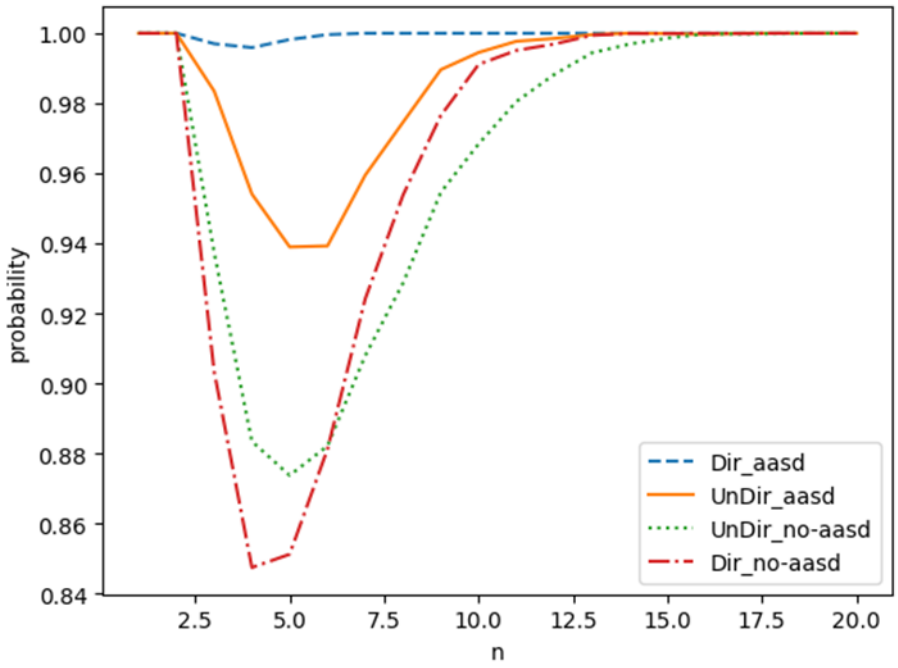
In order to test the accuracy, we generate random adjacency matrix A and permutation matrix P. However, the testing samples exclude non-isomorphism cases because

$$if\ G_A \cong G_B\ then \nexists\ P\ s.t. PAP^T = B.$$

Testing (num_trials = 10000)

```python
import matplotlib.pyplot as plt

# 測試
n_values = []  # 儲存不同的 n 值
probability_values = []  # 儲存對應的機率值

num_trials = 10000  # 進行的試驗次數
ran = 1
ge = 20
for n in range(ran, ge+1):
    n_values.append(n)  # 儲存當前的 n 值

    num_correct = 0  # 重置正確的次數

    for _ in range(num_trials):
        A, P, B = generate_matrices(n)
        Qu, Qv = P_approx(A, B)
        result = np.array_equal(Qu@A@Qu.T, B)|np.array_equal(Qv@A@Qv.T, B)


        if result:
            num_correct += 1

    probability = num_correct / num_trials
    probability_values.append(probability)  # 儲存當前的機率值
    print(f"n={n}, probability={probability}")

# 繪製折線圖
plt.plot(n_values, probability_values)

# 設定標題和軸標籤
plt.title('')
plt.xlabel('n')
plt.ylabel('probability')

# 顯示圖表
plt.show()

```

7.2 Results



	Directed Graph		Undirected Graph	
n	Dir_aasd	Dir_no-aasd	UnDir_aasd	UnDir_no-aasd
1	1	1	1	1
2	1	1	1	1
3	0.997	0.9035	0.9835	0.9374
4	0.9959	0.8475	0.9541	0.8835
5	0.9982	0.8513	0.939	0.8738
6	0.9996	0.8813	0.9393	0.8822
7	1	0.9242	0.9595	0.908
8	1	0.9538	0.9746	0.9287
9	1	0.9765	0.9896	0.9546
10	1	0.9911	0.9945	0.9684
11	1	0.9951	0.9977	0.9805
12	1	0.9969	0.9985	0.9882
13	1	0.9994	0.9996	0.9944
14	1	0.9999	1	0.9969
15	1	0.9999	0.9999	0.9986
16	1	1	0.9997	0.9998
17	1	0.9999	1	0.9998
18	1	1	1	0.9999
19	1	1	1	1
20	1	1	1	1

8 Conclusions

The algorithm for proximity described in this paper can find the permutation matrix P between two graphs with high accuracy under the condition of polynomial time. Regarding the test results, we found that the accuracy for undirected graphs is lower than that for directed graphs, possibly because directed graphs contain more information. However, without aasd, the accuracy for undirected graphs surpasses that for directed graphs. Perhaps employing different adjustment methods to handle different types of matrices could yield better results, or machine learning could be utilized to pre-classify matrices of different types.

We believe that there is still much room for improvement and research in this algorithm, both theoretically and algorithmically. There are many areas that can be supplemented and strengthened.

The relevant code will be available on Colab:

https://colab.research.google.com/drive/1uOiAFsHmxJ_LcuHR__MU_v9sVUUqWfaO?usp=sharing

References

[1] MCKAY, Brendan D.; PIPERNO, Adolfo. Practical graph isomorphism, II. Journal of symbolic computation, 2014, 60: 94-112.

[2] OHARA, Kouzou; WASHIO, Takashi. On Feasibility of Graph Spectrum-based Frequent Sub-graph Mining. 人工知能学会第二種研究会資料, 2008, 2008.DMSM-A802: 03.

[3] LI, Xiaocan; WANG, Shuo; CAI, Yinghao. Tutorial: Complexity analysis of singular value decomposition and its variants. arXiv preprint arXiv:1906.12085, 2019.