Graph Management Exercises Report

Breadth-First Search, Depth-First Search, and Graph Coloring

Sara Sherif Daoud Saad

October 7, 2025

Contents

1 Breadth-First Search and Bipartite Graphs **3**
1.1 Exercise 1: BFS Distance Computation . . . 3
1.2 Exercise 2: Odd Cycles Analysis . . . 3
1.3 Exercise 3: Odd Cycle Detection Algorithm . . . 4
1.4 Exercise 4: Bipartite Graphs and Odd Cycles . . . 4
1.5 Exercise 5: Bipartite Graph Detection . . . 5

2 Depth-First Search and 2-Colorable Graphs **5**
2.1 Exercise 1: Link with Previous Exercise . . . 5
2.2 Exercise 2: 2-Colorability Algorithm . . . 5

3 Dijkstra's Algorithm Results **7**
3.1 Step-by-Step Execution of Dijkstra's Algorithm . . . 7
3.1.1 Initial Setup . . . 7
3.1.2 Iteration-by-Iteration Execution . . . 8
3.1.3 Final Results . . . 9

4 Matrix Multiplication and Graph Powers **9**
4.1 Exercise: Matrix Multiplication & Power . . . 9
4.1.1 1) Different Representations of These Graphs . . . 9
4.1.2 2) Compute A^2, A^3 and Meaning of A^r_ij . . . 10
4.1.3 3) Complexity of $A^r and Possible Reduction$ . . . 11

5 Linear Algebra Exercises **12**
5.1 Exercise 1: Matrix Analysis . . . 12
5.2 Exercise 2: Matrix Powers and Diagonalization . . . 12
5.3 Exercise 3: Eigenvalues and Eigenvectors of $A^T A and AA^T$ . . . 13
5.4 Exercise 4: Matrix Decomposition Analysis . . . 13
5.5 Exercise 5: Positive Definite Conditions . . . 14
5.6 Exercise 6: Eigenvalues of Matrix Powers . . . 15
5.7 Exercise 7: Determinant of Orthogonal Matrices . . . 15
5.8 Exercise 8: Laplacian Matrix Properties . . . 15

6 Random Walk on Graphs **16**
6.1 Proof 1: Lazy Random Walk is still Stochastic . . . 16

7 Code References **16**

1 Breadth-First Search and Bipartite Graphs

1.1 Exercise 1: BFS Distance Computation

Problem: Using graph traversal algorithms, propose an algorithm that computes the number of edges between a given vertex and all other vertices.

Solution: I adjusted the BFS algorithm to find the shortest paths in unweighted graphs by exploring vertices level by level.

```python
# See exercise1_bfs_distances.py for complete implementation
def compute_distances_from_vertex(graph, start_vertex):
    distances = {}
    for vertex in graph:
        distances[vertex] = float('inf')

    distances[start_vertex] = 0
    queue = deque([start_vertex])

    while queue:
        current_vertex = queue.popleft()
        for neighbor in graph[current_vertex]:
            if distances[neighbor] == float('inf'):
                distances[neighbor] = distances[current_vertex] + 1
                queue.append(neighbor)

    return distances
```

Listing 1: BFS Distance Computation

Time Complexity: O(V + E) where V is the number of vertices and E is the number of edges.

Implementation: See `exercise1_bfs_distances.py` for complete implementation with custom Queue class.

1.2 Exercise 2: Odd Cycles Analysis

Problem: Given the following cycles with even and odd length (with the distances or depths from the grey vertex), what do you think about the case of graphs with an odd cycle (in number of edges)? Is this a characteristic property? State the general case.

Solution:

Observation from the graphs:

- A graph with only even cycles can be bipartite
- A graph with an odd cycle cannot be bipartite

Characteristic property:

- A graph is bipartite $\Leftrightarrow$ it contains no odd-length cycles

General case:

- Odd cycles break bipartiteness; even cycles do not

Mathematical Foundation:

- In bipartite graphs, all cycles must have even length
- If a graph contains an odd cycle, it cannot be bipartite
- This provides a necessary and sufficient condition for bipartiteness
- The presence of an odd cycle is a definitive proof that a graph is not bipartite

1.3 Exercise 3: Odd Cycle Detection Algorithm

Problem: Propose an algorithm that determines if a graph contains an odd cycle.

Solution: Use BFS with vertex coloring to detect odd cycles. If we find a back edge connecting two vertices of the same color, we have found an odd cycle.

```python
# See exercise3and5_odd_cycle_detection_bipartite.py for complete implementation
def has_odd_cycle(graph):
    color = {}
    for node in graph:
        if node not in color:
            queue = deque([node])
            color[node] = 0

            while queue:
                u = queue.popleft()
                for v in graph[u]:
                    if v not in color:
                        color[v] = 1 - color[u]
                        queue.append(v)
                    elif color[v] == color[u]:
                        return True
    return False
```

Listing 2: Odd Cycle Detection

Time Complexity: $O(V + E)$ **Implementation:** See `exercise3and5_odd_cycle_detection_bip` for complete implementation with detailed comments and test cases.

1.4 Exercise 4: Bipartite Graphs and Odd Cycles

Problem: In a bipartite graph, can there be a cycle with an odd number of edges? Is this a characteristic property? Justify your answer.

Answer: No, bipartite graphs cannot contain odd cycles. This is indeed a characteristic property.

Property:

- Bipartite graphs cannot have odd cycles

Justification: Bipartite graphs divide vertices into two sets, all edges go across sets. If there were an odd cycle, you would need an edge connecting vertices in the same set, which is impossible.

Mathematical Foundation:

1. In a bipartite graph, vertices can be divided into two disjoint sets U and V
2. Every edge connects a vertex in U to a vertex in V
3. Any cycle must alternate between sets U and V
4. To return to the starting vertex, we need an even number of steps
5. Therefore, all cycles in bipartite graphs have even length

1.5 Exercise 5: Bipartite Graph Detection

Problem: Propose an algorithm that allows to determine if a graph is bipartite. Test your algorithm on the following graph. Is it bipartite? Justify your answer.

Solution: BFS-based bipartite detection algorithm.

This is the same BFS coloring algorithm. If you can color the graph with 2 colors without conflict, it's bipartite.

Result: If `has_odd_cycle` returns `True`, the graph is not bipartite. If `False`, it is bipartite.

Algorithm: The same BFS coloring algorithm from Exercise 3 can be used to detect bipartiteness.

2 Depth-First Search and 2-Colorable Graphs

2.1 Exercise 1: Link with Previous Exercise

Problem: What is the link with the previous exercise? Justify your answer.

Answer:

1) Link with the previous exercise

In the previous exercise, we studied BFS, odd cycles, and bipartite graphs. A graph is bipartite $\Leftrightarrow$ 2-colorable $\Leftrightarrow$ contains no odd cycle. So the question of 2-colorability is exactly the same as asking whether the graph is bipartite.

Justification:

- If a graph is 2-colorable, then all cycles are even (so no odd cycle)
- If there's an odd cycle, 2-coloring fails
- Hence, the link is that checking bipartiteness (with BFS in the previous exercise) is equivalent to checking 2-colorability (with DFS here)

Implementation: See `dfs_exercise2_2colorability_algorithm.py` for detailed analysis.

2.2 Exercise 2: 2-Colorability Algorithm

Problem: We want to write an algorithm, inspired by DFS search which takes as input a graph G(V, E) and which returns a pair (result, color) where result is true if the graph is colorable, false otherwise and color is a dictionary associating a color 0 or 1 to each vertex. This algorithm should stop as soon as possible when the graph is not 2-colorable. Propose an iterative version or a recursive version.

Solution:
We need an algorithm that:

- Uses DFS traversal
- Assigns alternating colors (0 and 1)
- Stops immediately when a conflict (same color on adjacent vertices) is found
- Returns (result, color)

Recursive DFS implementation:

```python
def is_2_colorable_dfs(graph):
    color = {}

    def dfs(node, c):
        color[node] = c
        for neighbor in graph[node]:
            if neighbor not in color:
                if not dfs(neighbor, 1 - c):  # alternate color
                    return False
            elif color[neighbor] == color[node]:
                return False  # conflict: same color for adjacent nodes
        return True

    # handle disconnected graphs
    for v in graph:
        if v not in color:
            if not dfs(v, 0):
                return False, color
    return True, color
```

Listing 3: Recursive DFS 2-Colorability Detection

Iterative DFS implementation (stack-based):

```python
def is_2_colorable_dfs_iterative(graph):
    color = {}

    for start in graph:
        if start not in color:
            stack = [(start, 0)]

            while stack:
                node, c = stack.pop()

                if node in color:
                    if color[node] != c:
                        return False, color  # conflict
                    continue

                color[node] = c
```

```
                for neighbor in graph[node]:
                    stack.append((neighbor, 1 - c))

    return True, color
```

Listing 4: Iterative DFS 2-Colorability Detection

Example usage:

```
graph = {
    0: [1, 2],
    1: [0, 3],
    2: [0, 3],
    3: [1, 2]
}

print(is_2_colorable_dfs(graph))
# Output: (True, {0:0, 1:1, 2:1, 3:0})     the graph is 2-colorable
    .
```

Listing 5: Example Usage

If there's an odd cycle, the function stops early and returns (False, color).
Features:

- Early termination when conflict is found
- Returns (result, color) pair as specified
- Both iterative and recursive versions
- Time complexity: O(V + E)
- Handles disconnected graphs

3 Dijkstra's Algorithm Results

3.1 Step-by-Step Execution of Dijkstra's Algorithm

I'll solve this step-by-step using Dijkstra's algorithm with source vertex $s = a$.

3.1.1 Initial Setup

- dist $= \{a : 0\}$
- $P = \emptyset$ (processed vertices)
- dist$[v] = \infty$ for all $v \neq a$

3.1.2 Iteration-by-Iteration Execution

Iteration 1:

- Select $w = a$ (min distance = 0)
- $P = \{a\}$
- Update neighbors of a:
  - f: dist$[f] = \min(\infty, 0 + 3) = 3$, predecessor$[f] = a$
  - g: dist$[g] = \min(\infty, 0 + 1) = 1$, predecessor$[g] = a$

Current distances: $\{a : 0, f : 3, g : 1, b : \infty, e : \infty, c : \infty, d : \infty\}$
Iteration 2:

- Select $w = g$ (min unprocessed distance = 1)
- $P = \{a, g\}$
- Update neighbors of g:
  - f: dist$[f] = \min(3, 1 + 1) = 2$, predecessor$[f] = g$
  - e: dist$[e] = \min(\infty, 1 + 2) = 3$, predecessor$[e] = g$

Current distances: $\{a : 0, g : 1, f : 2, e : 3, b : \infty, c : \infty, d : \infty\}$
Iteration 3:

- Select $w = f$ (min unprocessed distance = 2)
- $P = \{a, g, f\}$
- Update neighbors of f:
  - b: dist$[b] = \min(\infty, 2 + 4) = 6$, predecessor$[b] = f$
  - e: dist$[e] = \min(3, 2 + 2) = 3$ (no change)

Current distances: $\{a : 0, g : 1, f : 2, e : 3, b : 6, c : \infty, d : \infty\}$
Iteration 4:

- Select $w = e$ (min unprocessed distance = 3)
- $P = \{a, g, f, e\}$
- Update neighbors of e:
  - c: dist$[c] = \min(\infty, 3 + 1) = 4$, predecessor$[c] = e$
  - d: dist$[d] = \min(\infty, 3 + 5) = 8$, predecessor$[d] = e$

Current distances: $\{a : 0, g : 1, f : 2, e : 3, c : 4, b : 6, d : 8\}$
Iteration 5:

- Select $w = c$ (min unprocessed distance = 4)
- $P = \{a, g, f, e, c\}$

- Update neighbors of c:
  - b: $\text{dist}[b] = \min(6, 4+3) = 6$ (no change)
  - d: $\text{dist}[d] = \min(8, 4+2) = 6$, $\text{predecessor}[d] = c$

Current distances: $\{a:0, g:1, f:2, e:3, c:4, b:6, d:6\}$
Iterations 6 & 7:

- Process b (distance 6) and d (distance 6)
- No more updates needed

3.1.3 Final Results

Shortest distances from vertex a:

Destination	Distance	Path
$a \to a$	0	a
$a \to g$	1	$a \to g$
$a \to f$	2	$a \to g \to f$
$a \to e$	3	$a \to g \to e$
$a \to c$	4	$a \to g \to e \to c$
$a \to b$	6	$a \to g \to f \to b$
$a \to d$	6	$a \to g \to e \to c \to d$

Table 1: Shortest paths from node a to all other nodes

4 Matrix Multiplication and Graph Powers

4.1 Exercise: Matrix Multiplication & Power

4.1.1 1) Different Representations of These Graphs

Left Graph (Undirected):
Edge List:

- (0,1), (0,4), (1,2), (1,5), (2,3), (3,5), (4,5)

Adjacency Matrix A:

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 \end{bmatrix} \tag{1}$$

Adjacency List:

- 0: {1, 4}

- 1: {0, 2, 5}
- 2: {1, 3}
- 3: {2, 5}
- 4: {0, 5}
- 5: {1, 3, 4}

Right Graph (Directed):
Edge List:

- (0,1), (0,4), (1,5), (2,3), (3,3), (4,5), (5,4)

Adjacency Matrix A:

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \tag{2}$$

Adjacency List:

- 0: {1, 4}
- 1: {5}
- 2: {3}
- 3: {3}
- 4: {5}
- 5: {4}

4.1.2 2) Compute $\mathbf{A^2}$, $\mathbf{A^3}$ and Meaning of $\mathbf{A}_i^r j$

For the Left Graph (Undirected):
$\mathbf{A^2}$:

$$A^2 = \begin{bmatrix} 2 & 0 & 1 & 0 & 0 & 2 \\ 0 & 3 & 0 & 2 & 1 & 0 \\ 1 & 0 & 2 & 0 & 0 & 2 \\ 0 & 2 & 0 & 2 & 1 & 0 \\ 0 & 1 & 0 & 1 & 2 & 0 \\ 2 & 0 & 2 & 0 & 0 & 3 \end{bmatrix} \tag{3}$$

$\mathbf{A^3}$:

$$A^3 = \begin{bmatrix} 0 & 5 & 0 & 4 & 4 & 0 \\ 5 & 0 & 6 & 0 & 0 & 7 \\ 0 & 6 & 0 & 5 & 3 & 0 \\ 4 & 0 & 5 & 0 & 0 & 6 \\ 4 & 0 & 3 & 0 & 0 & 5 \\ 0 & 7 & 0 & 6 & 5 & 0 \end{bmatrix} \tag{4}$$

For the Right Graph (Directed):

$\mathbf{A^2}$:

$$A^2 = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{5}$$

$\mathbf{A^3}$:

$$A^3 = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \tag{6}$$

Meaning of $\mathbf{A}_i^r j$:

A_{ij}^r represents the number of walks (paths that may repeat vertices) of length exactly r from vertex i to vertex j.

Examples:

- **$A^2[0,5] = 2$ (left graph) means there are 2 walks of length 2 from vertex 0 to vertex 5: $(0 \to 1 \to 5)$ and $(0 \to 4 \to 5)$**
- **$A^3[1,5] = 7$ (left graph) means there are 7 walks of length 3 from vertex 1 to vertex 5**

4.1.3 3) Complexity of $\mathbf{A}^r andPossibleReduction$

Naive Approach:

Computing A^r by repeated matrix multiplication:

- **$A^2 = A \times A$**
- **$A^3 = A^2 \times A$**
- **$A^r = A^{r-1} \times A$**

For an $n \times n$ matrix, one multiplication takes $O(n^3)$ time (or $O(n^{2.37})$ with Strassen's algorithm). Computing A^r requires $(r-1)$ multiplications: $O(r \cdot n^3)$

Optimized Approach - Exponentiation by Squaring:

We can reduce the number of multiplications using binary exponentiation:

- **Compute $A^1, A^2, A^4, A^8, \ldots$, by repeated squaring**
- **Combine results based on binary representation of r**

Complexity: $O(\log r \cdot n^3)$ or $O(\log r \cdot n^{2.37})$ with fast matrix multiplication

Example: To compute A^{10}:

- **$10 = 1010_2 = 8 + 2$**
- **Compute: $A^1 \to A^2 \to A^4 \to A^8$**

- **Result:** $A^{10} = A^8 \times A^2$
- **Only 4 multiplications instead of 9!**

Reduction: Yes, from $O(r \cdot n^3)$ to $O(\log r \cdot n^3)$ - exponential improvement in r!

5 Linear Algebra Exercises

5.1 Exercise 1: Matrix Analysis

Problem: What could you say about these matrices?

Let me call them $A = \begin{pmatrix} -1 & \frac{3}{2} \\ 1 & -1 \end{pmatrix}$, $B = \begin{pmatrix} -1 & \frac{3}{2} \\ \frac{2}{3} & -1 \end{pmatrix}$, $C = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$

Observations:

- **C is the identity matrix** I_3
- **A is not symmetric** ($A \neq A^T$)
- **B is symmetric** ($B = B^T$)
- **Both A and B have trace = -2 and similar structure**
- **C is orthogonal, symmetric, and idempotent**

5.2 Exercise 2: Matrix Powers and Diagonalization

Problem: Show that $A^n = X\Lambda^n X^{-1}$

Proof: If A is diagonalizable, then $A = X\Lambda X^{-1}$ **where** Λ **is diagonal matrix of eigenvalues and X is matrix of eigenvectors.**

Then:

- $A^2 = (X\Lambda X^{-1})(X\Lambda X^{-1}) = X\Lambda(X^{-1}X)\Lambda X^{-1} = X\Lambda^2 X^{-1}$
- $A^3 = A^2 \cdot A = (X\Lambda^2 X^{-1})(X\Lambda X^{-1}) = X\Lambda^3 X^{-1}$

By induction:

- **Base case:** $A^1 = X\Lambda^1 X^{-1}$
- **Inductive step: If** $A^k = X\Lambda^k X^{-1}$**, then** $A^{k+1} = A^k \cdot A = (X\Lambda^k X^{-1})(X\Lambda X^{-1}) = X\Lambda^{k+1} X^{-1}$

Therefore $A^n = X\Lambda^n X^{-1}$ **for all** $n \geq 1$**.**

5.3 Exercise 3: Eigenvalues and Eigenvectors of $\mathbf{A}^T A and A A^T$

Problem: Find eigenvalues and unit eigenvectors of $A^T A$ and AA^T

Given: $A = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$ **(Fibonacci matrix)**

Computing $A^T A$:

$$A^T A = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 2 & 1 \\ 1 & 1 \end{pmatrix}$$

Eigenvalues of $A^T A$:

$$\det(A^T A - \lambda I) = \det \begin{pmatrix} 2-\lambda & 1 \\ 1 & 1-\lambda \end{pmatrix} = (2-\lambda)(1-\lambda) - 1 = \lambda^2 - 3\lambda + 1 = 0$$

$$\lambda = \frac{3 \pm \sqrt{9-4}}{2} = \frac{3 \pm \sqrt{5}}{2}$$

$\lambda_1 = \frac{3+\sqrt{5}}{2} \approx 2.618$ **(golden ratio squared: ϕ^2)**
$\lambda_2 = \frac{3-\sqrt{5}}{2} \approx 0.382$
Unit eigenvectors of $A^T A$:
For $\lambda_1 = \frac{3+\sqrt{5}}{2}$:

$$v_1 = \frac{1}{\sqrt{2+\sqrt{5}}} \begin{pmatrix} \frac{1+\sqrt{5}}{2} \\ 1 \end{pmatrix}$$

For $\lambda_2 = \frac{3-\sqrt{5}}{2}$:

$$v_2 = \frac{1}{\sqrt{2-\sqrt{5}}} \begin{pmatrix} \frac{1-\sqrt{5}}{2} \\ 1 \end{pmatrix}$$

Computing AA^T:

$$AA^T = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 2 & 1 \\ 1 & 1 \end{pmatrix}$$

Note: $AA^T = A^T A$ in this case! So eigenvalues and eigenvectors are the same.

5.4 Exercise 4: Matrix Decomposition Analysis

Problem: Without multiplying S, find determinant, eigenvalues, eigenvectors, and why S is positive definite

$$S = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} 2 & 0 \\ 0 & 5 \end{bmatrix} \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}$$

Let $R(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$ **(rotation matrix)**
Then $S = R(\theta) \cdot D \cdot R(\theta)^T$ **where** $D = \mathbf{diag}(2, 5)$
Determinant:

$$\det(S) = \det(R) \cdot \det(D) \cdot \det(R^T) = 1 \cdot 10 \cdot 1 = \mathbf{10}$$

Eigenvalues: S is similar to D (conjugate by rotation), so eigenvalues are $\lambda_1 = 2, \lambda_2 = 5$

Eigenvectors: The eigenvectors of D are $e_1 = (1, 0)^T$ **and** $e_2 = (0, 1)^T$

The eigenvectors of S are rotated versions:

- $v_1 = (\cos\theta, \sin\theta)^T$ **for** $\lambda_1 = 2$
- $v_2 = (-\sin\theta, \cos\theta)^T$ **for** $\lambda_2 = 5$

Why S is positive definite:

- **All eigenvalues are positive** $(2 > 0, 5 > 0)$
- **S is symmetric** $(S = S^T)$
- **Therefore S is positive definite**

5.5 Exercise 5: Positive Definite Conditions

Problem: For what c and d are S and T positive definite?

$$S = \begin{pmatrix} c & 1 & 1 \\ 1 & c & 1 \\ 1 & 1 & c \end{pmatrix}, \quad T = \begin{pmatrix} 1 & 2 & 3 \\ 2 & d & 4 \\ 3 & 4 & 5 \end{pmatrix}$$

For S to be positive definite: Using Sylvester's criterion (all leading principal minors > 0**):**

1. $c > 0$
2. $\det \begin{pmatrix} c & 1 \\ 1 & c \end{pmatrix} = c^2 - 1 > 0 \implies c > 1$
3. $\det(S) = c^3 + 2 - 3c = (c-1)^2(c+2) > 0 \implies c > 1$ **(since** $c + 2 > 0$ **when** $c > 0$**)**

Answer for S: $c > 1$

For T to be positive definite:

1. $1 > 0$
2. $\det \begin{pmatrix} 1 & 2 \\ 2 & d \end{pmatrix} = d - 4 > 0 \implies d > 4$
3. $\det(T) = 1(5d - 16) - 2(10 - 12) + 3(8 - 3d) = 5d - 16 + 4 + 24 - 9d = 12 - 4d > 0 \implies d < 3$

But we need $d > 4$ **AND** $d < 3$**, which is impossible!**

Answer for T: No values of d make T positive definite (T has rank 2 since rows are nearly linearly dependent)

5.6 Exercise 6: Eigenvalues of Matrix Powers

Problem: Show if $\lambda_1, \lambda_2, \ldots, \lambda_n$ are eigenvalues of A, then A^m has eigenvalues $\lambda_1^m, \lambda_2^m, \ldots, \lambda_n^m$

Proof: If $Av = \lambda v$ (v is eigenvector with eigenvalue λ), then:

- $A^2v = A(Av) = A(\lambda v) = \lambda(Av) = \lambda^2 v$
- $A^3v = A(A^2v) = A(\lambda^2 v) = \lambda^2(Av) = \lambda^3 v$

By induction: $A^m v = \lambda^m v$

Therefore, if λ is an eigenvalue of A with eigenvector v, then λ^m is an eigenvalue of A^m with the same eigenvector v.

5.7 Exercise 7: Determinant of Orthogonal Matrices

Problem: What is the determinant of any orthogonal matrix?

An orthogonal matrix Q satisfies $Q^TQ = I$

Taking determinants: $\det(Q^TQ) = \det(I)$

$\det(Q^T) \cdot \det(Q) = 1$

$\det(Q)^2 = 1$

Answer: $\det(Q) = \pm 1$

5.8 Exercise 8: Laplacian Matrix Properties

Problem: Laplacian matrix properties

For an undirected graph, the Laplacian matrix $L = D - A$ where:

- **D = diagonal degree matrix**
- **A = adjacency matrix**

Properties to show:

(a) L is symmetric: Since A is symmetric (undirected graph) and D is diagonal (symmetric), $L = D - A$ is symmetric.

(b) L is positive semi-definite: For any vector x:

$$x^TLx = x^T(D-A)x = \sum_i d_i x_i^2 - \sum_{i,j} A_{ij}x_i x_j = \frac{1}{2}\sum_{(i,j)\in E}(x_i - x_j)^2 \geq 0$$

Therefore L is positive semi-definite.

(c) L has 0 as an eigenvalue:

$$L\mathbf{1} = (D-A)\mathbf{1} = D\mathbf{1} - A\mathbf{1} = \mathbf{0}$$

(since each row sum of A equals the degree)

The vector $\mathbf{1} = (1, 1, \ldots, 1)^T$ is an eigenvector with eigenvalue 0.

This is the smallest eigenvalue because L is positive semi-definite (all eigenvalues $\geq$ 0).

6 Random Walk on Graphs

6.1 Proof 1: Lazy Random Walk is still Stochastic

A *lazy random walk* on a graph $G = (V, E)$ is defined such that at each step:

- **with probability $\frac{1}{2}$, the walker stays at the current node (self cycles),**
- **with probability $\frac{1}{2}$, the walker moves to a neighbor chosen at random from a unform distribution.**

from the definition of the simple random walk, A be the adjacency matrix and $D = \mathrm{diag}(d_1, \ldots, d_n)$ the degree matrix. The transition matrix of a simple random walk is given by

$$P = D^{-1}A.$$

For the lazy random walk:

1. **With probability $\frac{1}{2}$, the walker remains on the same node. This is represented by the *identity matrix* I, since**

$$I_{ij} = \begin{cases} 1, & \textbf{if } i = j, \\ 0, & \textbf{otherwise.} \end{cases}$$

   It encodes a self-loop transition (staying at node i).

2. **With probability $\frac{1}{2}$, the walker follows the standard random walk step given by $P = D^{-1}A$.**

$$P_{\textbf{lazy}} = \frac{1}{2}I + \frac{1}{2}P = \frac{1}{2}\left(I + D^{-1}A\right).$$

To proof that $P_{\textbf{lazy}}$ is stochastic. We must show that each row of $P_{\textbf{lazy}}$ sums to 1. For the $i^{\textbf{th}}$ row:

$$\sum_j (P_{\textbf{lazy}})_{ij} = \frac{1}{2}\sum_j I_{ij} + \frac{1}{2}\sum_j P_{ij} = \frac{1}{2}(1+1) = 1.$$

Hence $P_{\textbf{lazy}}$ is row-stochastic.

Consequence. The addition of self-loops ($\frac{1}{2}I$) makes the chain *aperiodic* and thus *primitive*. By the Perron–Frobenius theorem, $P_{\textbf{lazy}}$ admits a unique stationary distribution π satisfying

$$\pi = \pi P_{\textbf{lazy}}, \quad \textbf{with} \quad \sum_i \pi_i = 1.$$

7 Code References

- `exercise1_bfs_distances.py` **- BFS distance computation**
- `exercise3and5_odd_cycle_detection_bipartite.py` **- Odd cycle detection and bipartite graphs**
- `dfs_exercise2_2colorability_algorithm.py` **- 2-colorability algorithm**