**An Explanation of Graphs, Spanning Trees and Matrices**

Justin Perkins

Southern New Hampshire University

MAT – 239: Mathematics for Computing

Dr. Melanie Fraser

December 17, 2021

**An Explanation of Graphs, Spanning Trees and Matrices**

Graphs are usually thought of as existing only on a coordinate plane. However, this is not always the case. A graph can be any diagram that shows the relationship between items. In this case, a connected graph shows how different vertices are connected to each other. Trees are a type of connected graph in which there are no cycles. A cycle can be loosely defined as a path from one vertex back to itself, making a loop. While cyclical graphs cannot be considered trees, we can form subgraphs that do meet the requirements for being a tree. These subgraphs are known as spanning trees.

In order to be considered a spanning tree, the subgraph must be formed from a cyclical connected graph. Second, the spanning tree must connect all of the vertices that were present in the original connected graph, without forming cycles. The example below shows all of the possible, spanning trees from the original connected graph. The original graph has a cycle from {2, 3, 4} and thus is not a tree. However, there are 3 unique permutations in which the vertices of the graph are connected without a cycle.

|  |  |  |  |
| --- | --- | --- | --- |
| Original Connected Graph | Spanning Tree 1 | Spanning Tree 2 | Spanning Tree 3 |
|  |  |  |  |

Graphs can also can be encoded into matrices to perform various calculations, like finding the degrees of a vertex or calculating the number of spanning trees that exist. An adjacency matrix is an *n x n* matrix, where n is the number of vertices, that shows all of the edges that emanate from a vertex. When reading an adjacency matrix, each row corresponds to a vertex. The intersection of that row with each column represents the existence of an edge between the other vertex. If an edge exists, the element will be a 1, if no edge exists the element will be a 0. The intersection of a vertex with itself will always be 0 as you cannot have a vertex that connects to itself.

Another way you can encode a is in a degree matrix. A degree matrix is also an *n x n* matrix, where n is the number of vertices, that tells you how many edges are connected to a vertex but not to what vertices it's connected to. In order to read a degree matrix, you look at each row and read the number on the diagonal. That number tells you the number of edges that emanate from that vertex. All elements that are not on the diagonal are always 0 in a degree matrix. Not only can we get a degree matrix from looking at the graph, but we can also calculate the degree matrix from an adjacency matrix. To do this, you must calculate the sum of the given row *i* and enter that sum as element *ai, i.* Once all the rows have been summarized, all the other elements of the matrix should be set to 0.

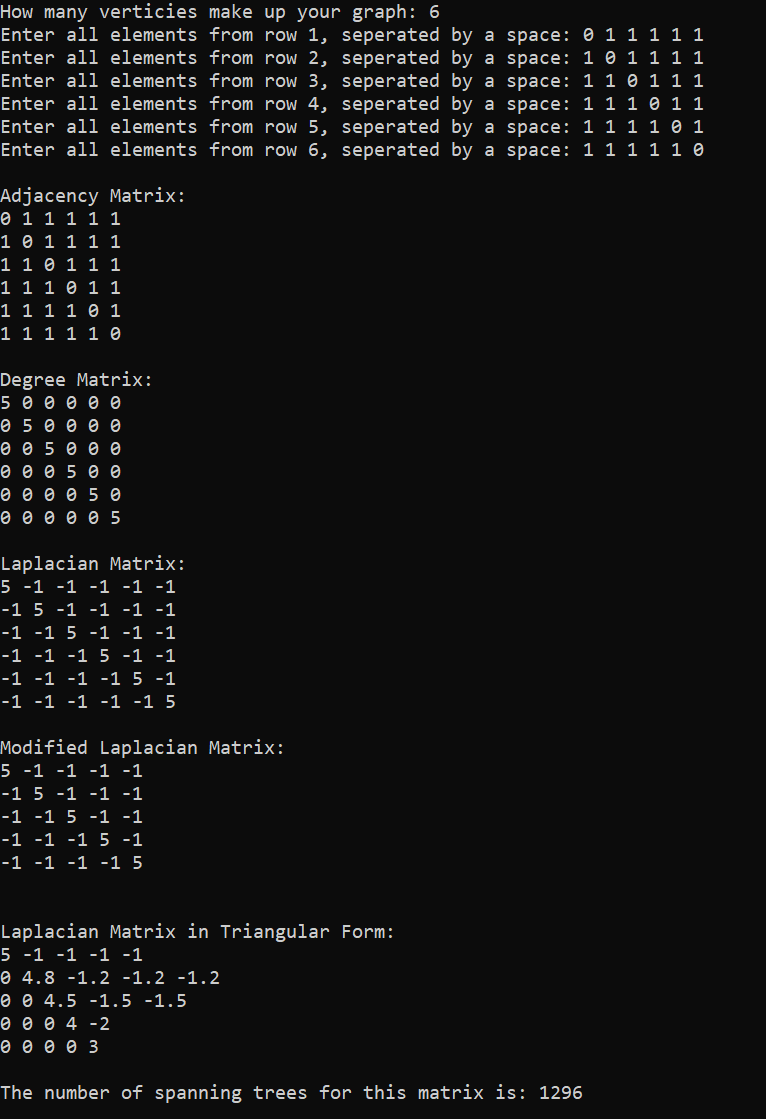
With these two matrices, we can calculate the Laplacian matrix. The Laplacian matrix is found by the formula *L = D – A*, or by subtracting the adjacency matrix from the degree matrix. The Laplacian matrix is quite important as it is used to calculate the number of spanning trees possible from any graph. Each element in the Laplacian matric can be defined by a similar equation *li, j = di, j – ai, j.* In order to find the number of spanning trees though, we have to modify the Laplacian matrix by removing both the last row and column. This modified Laplacian Matrix can be denoted with the symbol *Ln.* The determinant of this modified Laplacian is what gives us the number of spanning trees for our original graph.

There are two methods for finding the determinant that could be used here. Both row reduction to triangular form or cofactor expansion are viable options. In the program I wrote to calculate the determinant of any *n x n* matrix, I used the row reduction method. It is important to remember that in this method, some of your elementary row operations will affect your determinant and need to be accounted for. For example, if you multiply a row by a scalar *k* to produce matrix B, then det(A) = det(B) / *k.* However, we don’t need to keep track of our row operations if we only use the row sum operation, adding one row to the scalar multiple of another. This operation has no effect on the determinant of the matrix.

Triangular form is defined as a matrix that has only 0’s either below or above the diagonal of a matrix. Once you have completed all of your row operations to get the matrix into this form, calculating the determinant is easy. The determinant of a triangular matrix is equal to the product of the sum. In the example below, I have gone through all of the steps for finding the number of spanning trees for the previous example’s connected graph. In the final step, the determinant of the triangular matrix is equal to 1.5 \* 2 \* 1 or 3. Meaning that all possible spanning trees are present in the example.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Step 1: Encode the  Graph into a matrix | Step 2: Calculate the degree matrix | | Step 3: Calculate the Laplacian matrix | Step 4: Modify the Laplacian matrix | Step 5: Calculate the determinant of the modified Laplacian matrix |
|  |  |  | |  |  |

**Appendix A**



**Appendix B**

#include <iostream>

#include <vector>

using namespace std;

void printMatrix(vector<vector<int>> matrix) {

for (int i = 0; i < matrix.size(); i++)

{

for (int j = 0; j < matrix[i].size(); j++)

{

cout << matrix[i][j] << " ";

}

cout << endl;

}

cout << endl;

}

void printLaplacianMatrix(vector<vector<double>> matrix) {

for (int i = 0; i < matrix.size(); i++)

{

for (int j = 0; j < matrix[i].size(); j++)

{

cout << matrix[i][j] << " ";

}

cout << endl;

}

cout << endl;

}

double determinant(vector<vector<int>>& matrix) {

vector<double> modifiedRow;

vector<vector<double>> laplacianMatrix;

//Copy the int matrix into the double matrix

for (int i = 0; i < matrix.size(); i++) {

vector<double> row;

for (int j = 0; j < matrix.size(); j++) {

row.push\_back(matrix[i][j]);

}

laplacianMatrix.push\_back(row);

row.clear();

}

for (int i = 0; i < laplacianMatrix.size() - 1; i++) {

//create modified row

double factor = 1 / laplacianMatrix[i][i];

for (int j = 0; j < laplacianMatrix.size(); j++) {

modifiedRow.push\_back(laplacianMatrix[i][j] \* factor);

}

//computes row operation

for (int k = i + 1; k < laplacianMatrix.size(); k++) { //Iterate through the starting positions

double modifier = laplacianMatrix[k][i] \* -1; // needs to diferent for every row

for (int l = 0; l < laplacianMatrix.size(); l++) { //iterates through all the elements of the vectors

laplacianMatrix[k][l] += modifiedRow[l] \* modifier; //Actual Row operation

}

}

modifiedRow.clear();

}

cout << "\nLaplacian Matrix in Triangular Form: \n";

printLaplacianMatrix(laplacianMatrix);

double product = 1;

for (int i = 0; i < laplacianMatrix.size(); i++) { //Calculates the product of the diagonal

product \*= laplacianMatrix[i][i];

}

return product;

}

int getData() {

int verticies;

cout << "How many verticies make up your graph: ";

cin >> verticies;

return verticies;

}

void createAdjacencyMatrix(vector<vector<int>>& adjacencyMatrix, int verticies) {

int temp;

vector<int> row;

for (int k = 1; k < verticies + 1; k++) {

cout << "Enter all elements from row " << k << ", seperated by a space: ";

for (int i = 0; i < verticies; i++) {

cin >> temp;

row.push\_back(temp);

}

adjacencyMatrix.push\_back(row);

row.clear();

}

}

void createDegreeMatrix(vector<vector<int>>& degreeMatrix, vector<vector<int>> adjacencyMatrix) {

for (int i = 0; i < adjacencyMatrix.size(); i++)

{

int sum = 0;

vector<int> row(adjacencyMatrix.size(), 0);

for (int j = 0; j < adjacencyMatrix[i].size(); j++)

{

sum += adjacencyMatrix[i][j];

}

row[i] = sum;

degreeMatrix.push\_back(row);

}

}

void createLaplacianMatrix(vector<vector<int>>& laplacianMatrix, vector<vector<int>> degreeMatrix, vector<vector<int>> adjacencyMatrix) {

int difference;

vector<int> row;

for (int i = 0; i < degreeMatrix.size(); i++)

{

for (int j = 0; j < degreeMatrix[i].size(); j++)

{

difference = degreeMatrix[i][j] - adjacencyMatrix[i][j];

row.push\_back(difference);

}

laplacianMatrix.push\_back(row);

row.clear();

}

}

void modifyLaplacianMatix(vector<vector<int>>& laplacianMatrix) {

for (int i = 0; i < laplacianMatrix.size() - 1; i++) {

laplacianMatrix[i].pop\_back();

}

laplacianMatrix.pop\_back();

}

int main()

{

int verticies = getData();

vector<vector<int>> adjacencyMatrix;

vector<vector<int>> degreeMatrix;

vector<vector<int>> laplacianMatrix;

createAdjacencyMatrix(adjacencyMatrix, verticies);

createDegreeMatrix(degreeMatrix, adjacencyMatrix);

createLaplacianMatrix(laplacianMatrix, degreeMatrix, adjacencyMatrix);

cout << "\nAdjacency Matrix:\n";

printMatrix(adjacencyMatrix);

cout << "Degree Matrix:\n";

printMatrix(degreeMatrix);

cout << "Laplacian Matrix:\n";

printMatrix(laplacianMatrix);

modifyLaplacianMatix(laplacianMatrix);

cout << "Modified Laplacian Matrix:\n";

printMatrix(laplacianMatrix);

cout << "The number of spanning trees for this matrix is: " << determinant(laplacianMatrix) << endl;

return 0;

}