Condensed Representations Of Incidence Matrix and Adjacency Matrix of a Graph

Graph Theory: Assignment 1

Akshat Aggarwal
IIM2014005
7th Semester Dual Degree (IT)

Yogesh Gupta
IRM2014004
7th Semester Dual Degree (IT)

Vishesh Middha
ISM2014007
7th Semester Dual Degree (IT)

***Abstract*—In this assignment we strive to achieve a compressed and condensed form of incidence and adjacency matrix representation of a graph. We have worked on three different algorithms, namely run length encoding, binary to decimal conversion and matrix to list conversion to achieve successful results.**

***Index Terms*—graph, adjacency, incidence, run length, list, compression, decimal.**

I. Introduction

Graphs are powerful abstractions, that serve an important role in modeling data. They are useful in solving many real-life applications such as social networks, transportation networks, document-link graphs, robot planning and many more. We first introduce important notations and definitions and then state our objective.

A. Basic Definitions

1) Graph: Formally, graph is a pair G =(V, E) where,

- V is a set of vertices (or nodes), and
- E $\subseteq$ (V x V) is a set of edges.

A graph [4] may be directed or undirected. If the graph is undirected, then the adjacency relation defined by edges is symmetric.

2) Simple Graph: A simple graph (also called as a strict graph) is an unweighted, undirected graph containing no self-loops or multiple edges.

3) Incidence Matrix: Incidence matrix of an undirected graph is a matrix, B of dimension N x M, where N is the number of vertices and M is the number of edges, such that $B_{i,j}$= 1, if the vertex v_i and edge e_j are incident and 0 otherwise.

4) Adjacency Matrix: Adjacency matrix of an undirected graph is a matrix, B of dimension N x N, where N is the number of vertices, such that $B_{i,j}$= 1, if there is an edge between vertices *i* and *j*, and 0 otherwise.

B. Objective

Here, we propose 3 different condensed representations of incidence matrix of a simple graph based on the following methods:

1) Run length encoding [1]
2) Binary to Decimal conversion [2]
3) Incidence matrix to Incidence List [3]

We further propose similar representations for adjacency matrix of a simple graph. In addition to above three, particularly for adjacent matrix we can store the upper triangular or lower triangular half along with the diagonal to save space. Corresponding to both, incidence and adjacency matrices - we perform time and space complexity analysis of each method, and suggest the best condensed representation.

II. Motivation

Graphs are a very important data structure. They are used in network related algorithms, routing, finding relations among objects, shortest paths and many more real life related applications. On e-commerce websites, relationship graphs are used to show recommendations. Connecting with friends on social media, where each user is a vertex and the connection between two users is represented by an edge, is also an important application of graphs.
The adjacency matrix and incidence matrix representation of a graph have a very high space complexity. So for above mentioned real life problems, where the size of graph is large and the cost of allocating the space is more, these representations become costly. This motivates us to represent our graphs in some condensed and compressed form, so that space complexity should not be a problem and we can focus on other important aspects.

III. Methods and Algorithms

A. Use of Run Length Encoding

Run length encoding is a simple data compression algorithm that, stores sequences of same data value as single data and it's count, instead of the original sequence.
Incidence or adjacency matrix of a simple graph consists of 0s and 1s, which makes it suitable to apply run length encoding on either dimension of the matrix.

Following is the algorithm used -

Algorithm I: Algorithm to perform run length encoding on rows of matrix
Input: Incidence (or Adjacency) Matrix - M
Output: Condensed Matrix - Result
The algorithm for run length encoding -

```
result ← []
for row in M do
    encoded_row ← [], count ← 1
    for i ← 1 to row.length-1 do
        if row[i] == row[i+1] then
            count ←  count + 1
        else
            encoded_row.append( {row[i], count} )
            count ← 1
        endif
    endfor
    encoded_row.append( {row[row.length], count} )
    result.append(encoded_row)
endfor
return result
```

B. Use of Incidence List

A list is a collection of unordered lists used to represent a finite graph. Each list describes the set of edges connected to a vertex in a graph in case of incidence list. In adjacency lists, each list describes the set of neighboring vertices.
Incidence or adjacency matrix of a simple graph contains 0's and 1's, where 1 represents that the edge is connected to the vertex and 0 represents not connected. So this makes it suitable to apply incidence list representation on each row.
Following is the algorithm used-

Algorithm II: Algorithm to perform incidence (or adjacency) list representation on rows of matrix.
Input: Incidence(or Adjacency) Matrix - M
Output: Incidence (or Adjacency) List - result

```
result ← []
for row in M do
    encoded_list ← []
    for i ← 1 to row.length do
        if row[i] == 1 then
            encoded_list.append(i)
        endif
    endfor
    result.append(encoded_list)
endfor
return result
```

C. Using Binary to Decimal Conversion

Since the matrix basically consists of rows of binary data, which can also be seen as binary strings (numbers), they can be converted to decimal numbers. The issue of integer overflow is handled by dividing the row into 64 length segments and then store decimal value of each 64 length segment.
For example, a row for a vertex could be 1001. This can be stored as a single decimal number 9 instead of saving an array of 0, 1 values.

Following is the algorithm used-

Algorithm III: Algorithm to convert each row of binary data to a decimal number.
Input: Incidence(or Adjacency) Matrix - M
Output: array of decimal value - result

The algorithm for decimal representation -

```
result ←  []
for row in M do
    value ← 0, power ← 0, numbers ← []
    for i ← 1 to row.length do
        value ← value + row[i] * 2^(power)
        power ← power + 1
        if power == 64 then
            value ← 0, power ← 0
            numbers.append( value )
        endif
    endfor
    result.append( numbers )
endfor
return result
```

D. Saving a Triangular Half of the matrix

The adjacency matrix of a simple graph exhibits the symmetric property (of matrices) i.e. given an adjacency matrix, A of a simple graph , the following property holds, for all i, j -

$$A[i][j] = A[j][i]$$

Hence, we need only to store one half of the matrix ,either the lower triangular half or upper triangular half. Here, we are storing the lower triangular half of the matrix.
Following is the algorithm used-

Algorithm IV: Algorithm to store lower triangular matrix.
Input: number of vertices - n , adjacency matrix - M
Output: lower triangular matrix, Result

The algorithm for decimal representation -

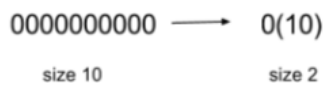
```
1. result ← []
2. for i = 1 to n do
3.     row ← []
4.     for j = 1 to i do
5.         row.append( M[i][j] )
6.     endfor
7.     result.append( row )
8. endfor
9. return result
```

IV. Results

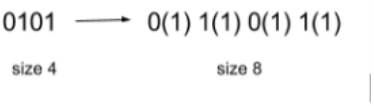
We successfully got a condensed form of adjacency and incidence matrix representation of graph which takes less space than adjacency and incidence matrix representation. For each of the above algorithm proposed, we list below their time and space complexities. We also perform comparison of the time taken in condensing the original matrix and space required in each of the condensed form. We used different methods to get the condensed form and each of them have some advantages and disadvantages.

A. Run Length Encoding

Performing run length encoding on the rows of the incidence matrix produces, an array of value-count pairs. So, in general, whether the graph is sparse (large number of consecutive 0's) or dense (large number of consecutive 1's) run length condensing would save space. It will take constant time when all elements are same i.e in the case where we have only 0's or 1's would result in



However, in the case where 0's or 1's dont occur contiguously may result in an increase in the space required. For example,



This turns out to be a disadvantage of using run length encoding.
Hence, the space required in best case is O(1), whereas in worst case is -

- O(n*m) for incidence matrix.
- O(n*n) for adjacency matrix.

The time complexity of using run length encoding is -

- O(n*m) for incidence matrix.
- O(n*n) for adjacency matrix.

where n - number of vertices, m - number of edges

B. Incidence List

The main disadvantage of incidence list is that implementation is hard and access time is high.
The advantage of the incidence list implementation is that it allows us to compactly represent a graph. The incidence list also allows us to easily find all the links that are directly connected to a particular vertex.It uses lower memory use as you are only storing the edges that actually exists in the graph.
Space required for lists is going to be -

- O(n*m) for incidence matrix.
- O(n*n) for adjacency matrix.

The time complexity is -

- O(n*m) for incidence matrix.
- O(n*n) for adjacency matrix.

where n - number of vertices, m - number of edges

C. Decimal Representation

The main disadvantage is that encoding into/decoding from decimal is a tough task to implement. Note that, for decoding back to the matrix, number of vertices/edges is known, hence it does not pose an issue.
The advantage is that it will always take constant space to represent the graph. Decimal (Base 10) is roughly the right size to memorize and manipulate.
Space required for storing the array of decimal values is going to be -

- O(n*m) for incidence matrix.
- O(n*n) for adjacency matrix.

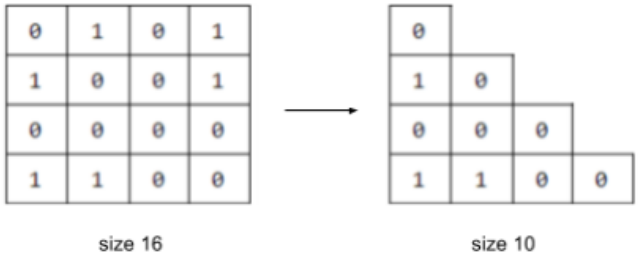
The time complexity is -

- O(n*m) for incidence matrix.
- O(n*n) for adjacency matrix.

where n - number of vertices, m - number of edges

D. Saving Lower Triangular Matrix

In this method, we use the fact that matrix is symmetric, hence it contains redundant entries.



Saving lower triangular matrix instead of adjacency matrix turns out to be better in terms of space as well as access time. Access time can be kept O(1) as follows -
for cell (i, j) , if i ¡ = j , then condensed[i][j], else

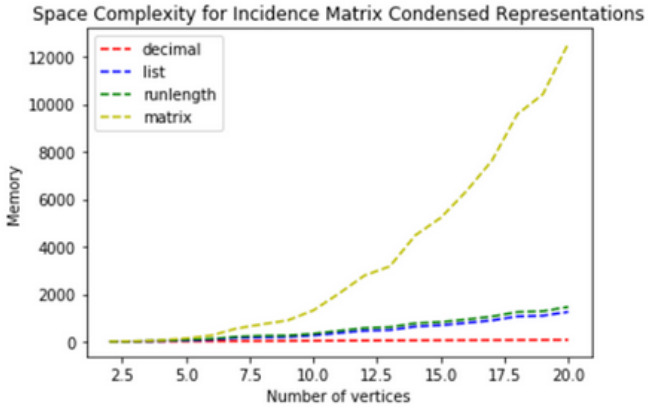
condensed[j][i].
Space required for condensed matrix is n(n-1)/2.
The time complexity of using this method is -

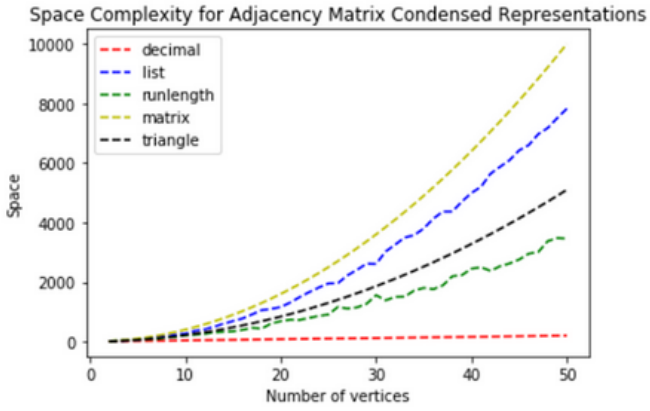
- O(n*n) for adjacency matrix.

where n - number of vertices.

E. A comparison of Space Complexities

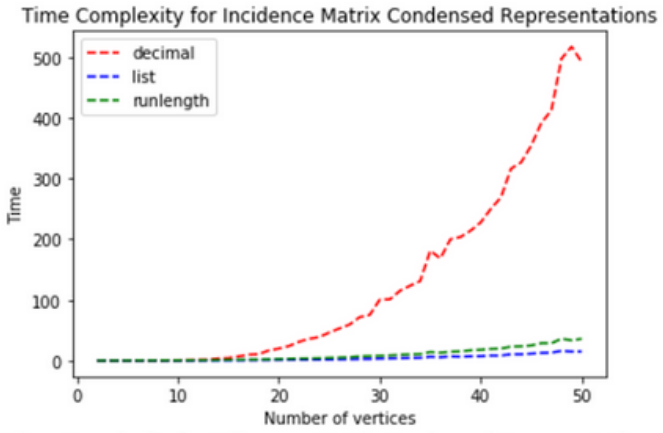
Below are a comparison of space complexities of the different algorithms used here -

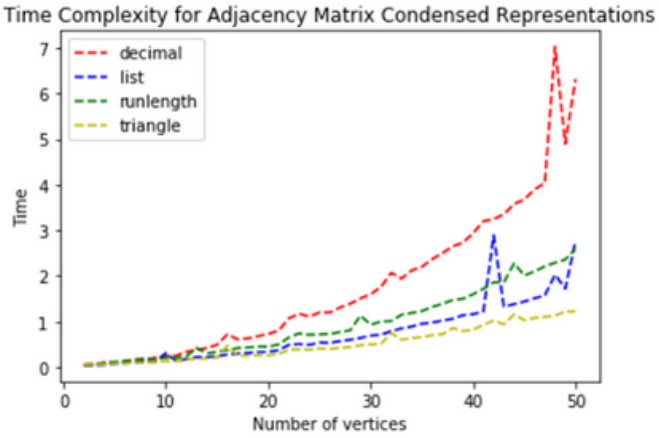




F. A comparison of Time Complexities

Below are a comparison of space complexities of the different algorithms used here -





V. Conclusions

From the above results and charts obtained, we conclude that -

The algorithm which gave the **best space complexity** was the Binary to Decimal Conversion. This is applicable to both incidence matrix and adjacency matrix.

The algorithm which gave the **best time complexity** is:

1) For Incidence Matrix: The Incidence List Conversion gave the best results as observed by the graph above.
2) For Adjacency Matrix: The Upper or Lower Triangle matrix conversion gave the best results as observed by the graph above.

References

[1] $https://en.wikipedia.org/wiki/Run-length_encoding$
[2] $http://www.electronics-tutorials.ws/binary/bin_2.html$
[3] $https://en.wikipedia.org/wiki/Adjacency_list$
[4] $https://en.wikipedia.org/wiki/Graph_(discrete_mathematics)$