Transitive Closure of a Graph using Graph Powering & Further Optimization by Euler's Fast Powering Algorithm

Abhijit Tripathy
Undergraduate Student, Department of Computer Science & Engineering, Guru Ghasidas Vishwavidyalaya, Bilaspur, India
Email: abhijittripathy99[at]gmail.com

Abstract: A graph is a collection of nodes and edges. Transitive closure matrix is a matrix formed by the reach-ability factor, which means if one node A of the graph is reachable from another node B, then there exists a positive reach-ability between A and B, negative reach-ability otherwise. This can be easily denoted by using binary denotation of 0 and 1. Graph powering is a technique in discrete mathematics and graph theory where our concern is to get the path between the nodes of a graph by using the powering principle. In simple words, if we take the rth power of any given graph G then that will give us another graph G(r) which has exactly the same vertices, but the number of edges will change. In the powered graph G(r) there will be a connection between any two nodes if there exits a path which has a length less than r between them. This small intuition can help us in finding the transitive closure of a graph in \(O(V^4)\) time complexity and \(O(V^2)\) space complexity. We can improve the time complexity of the above mentioned algorithm by using Euler's Fast Powering Algorithm to \(O(V^3\log V)\).

Keywords: graph algorithms, transitive closure, graph powering, discrete mathematics, Euler's fast powering

In this article, we will begin our discussion by briefly explaining about transitive closure and graph powering. We will also see the application of graph powering in determining the transitive closure of a given graph. Further we will improve the time complexity of the algorithm by using Euler's Fast Powering Algorithm.

What is Transitive Closure of a graph ?
In any Directed Graph, let’s consider a node i as a starting point and another node j as ending point. For all (i, j) pairs in a graph, transitive closure matrix is formed by the reach ability factor, i.e. if j is reachable from i (means there is a path from i to j) then we can put the matrix element as 1 or else if there is no path, then we can put it as 0.

Suppose we are given the following Directed Graph,

![Graph Image]

This matrix is known as the transitive closure matrix, where ‘1’ depicts the availability of a path from i to j, for each (i, j) in the matrix.

What is Graph Powering?
Graph powering is a technique in discrete mathematics and graph theory where our concern is to get the path between the nodes of a graph by using the powering principle.

In simple words, if we take the rth power of any given graph G then that will give us another graph G(r) which has exactly the same vertices, but the number of edges will change. In the powered graph G(r) there will be a connection between any two nodes if there exits a path which has a length less than r between them.

Suppose we have a directed graph as following.

![Transitive-Closure Matrix]

\[\begin{array}{cccccc}
0 & 1 & 2 & 3 & 4 \\
0 & 1 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 0 \\
2 & 1 & 0 & 1 & 1 & 0 \\
3 & 0 & 0 & 0 & 0 & 0 \\
4 & 1 & 0 & 1 & 1 & 1 \\
\end{array}\]

Then, the reachability matrix of the graph can be given by,
Now let's generate a new graph from the above graph by powering it to \( r = 2 \), i.e. \( G(2) \), Graph powered 2.

As you can see, the existing graph \( G \) has been updated with new edges between those nodes, who has a path difference of less than 2 (as \( r=2 \) here).

But the question arises on how to implement this in programming? We cannot use direct images for the calculations, but there is a solution to every problem for a programmer, and the solution here is the **Adjacent Matrix**.

Adjacent matrix is a matrix that denotes 1 for the position of \((i, j)\) if there is a direct edge between \(i\)th node and the \(j\)th node and denotes 0 otherwise.

Let's perform an experiment for an important conclusion.

For simplicity we have taken \( r = 2 \), adjacent matrix raised to the power 2, gives us another matrix as shown above.

What does the matrix(i.e. generated by the square of Adjacent matrix) signify?

Let's bring out the \( G(r = 2) \) graph into picture and observe closely on what the matrix signify,
By a little deep observation, we can say that (i, j) position of the $r$th powered Adjacent Matrix speaks about the number of paths from $i$ to $j$ in $G(r)$ that has a path length less than equal to $r$.

For example the value of the (0,1) position is 3. In the $G(r = 2)$ graph, we can see there are two paths whose path length is equal to 2 from 0 to 1, they are - [0 -- -- 1, 0 -- -- 2 -- -- 1 and 0 -- -- 3 -- -- 1]. Similarly we can determine for other positions of (i,j).

How to Find Transitive Closure by Graph Powering?

What will happen if we find $G(r = n)$ for any given graph $G$, where $n$ is the total number of nodes in $G$?

We will get a graph which has edges between all the $ith$ node and the $jth$ node whose path length is equal to $n$ at maximum. For any graph without loops, the length of the longest path will be the number of nodes in it. So by raising the Adjacent matrix of a given graph $G$ to the power of $n$, we can get a matrix having some entries $(i,j)$ as 0, which means there are not at all any path between $ith$ node and the $jth$ node which has a maximum path difference of $n$, where $n$ is the total number of nodes in the graph.

This gives us the main idea of finding transitive closure of a graph, which can be summarized in the three steps below,

1) Get the Adjacent Matrix for the graph
2) Raise the adjacent matrix to the power $n$, where $n$ is the total number of nodes.
3) Replace all the non-zero values of the matrix by 1 and printing out the Transitive Closure of matrix.

### Step 1 - Get the Adjacent Matrix

We will need a two dimensional array for getting the Adjacent Matrix of the given graph. Here are the steps;

- Get the total number of nodes and total number of edges in two variables namely $num\_nodes$ and $num\_edges$.
- Create a multidimensional array $edges\_list$ having the dimension equal to $num\_nodes \times num\_nodes$.
- Run a loop $num\_nodes$ time and take two inputs namely $first\_node$ and $second\_node$ every time as two nodes having an edge between them and place the $edges\_list[first\_node][second\_node]$ position equal to 1.
- Finally after the loop executes we have an adjacent matrix available i.e $edges\_list$.

```c
int num\_nodes, num\_edges;
cin >> num\_nodes >> num\_edges;
int** edges\_list = newint[num\_nodes][num\_nodes];
for(int i=0; i<num\_nodes; i++)
    for(int j=0; j<num\_nodes; j++)
        edges\_list[i][j] = 0;
int first\_node, second\_node;
cin >> first\_node >> second\_node;
edges\_list[first\_node][second\_node] = 1;
```

### Step 2 - Raising the Adjacent Matrix To The Power Of Total Number Of Nodes

This algorithm uses the simplest approach of matrix powering, just like in algebra we multiply two matrices in row-column method.

We will be following some steps to achieve the end result,

- First of all lets create a function named $matrix\_powering$ that returns void and takes two parameters namely $edges\_list$ i.e. the adjacent matrix and $num\_nodes$ i.e. the number of nodes.
- Create two multidimensional array which has the same dimension as that of edges list. One of them will be a blank matrix namely $result$ which will act as an auxiliary matrix for holding values during main calculation. Another one is named $matrix$, in which the entries of $edges\_list$ should be copied.
- Main algorithm will consist of four loops. The outer most loop is to multiply the matrix up to $num\_nodes$ times. The second and third loop will act as transition vertices for the multiplication and the inner most loop is for the intermediate vertices. We will take the row by column multiplication and place the sum in a variable name sum. After the innermost loop terminated the iteration we will place the sum value in out $result$ array.
- Finally we will copy the entries of $result$ to the entries of $matrix$.

```c
void matrix\_powering(int** edges\_list, int num\_nodes)
{
    int result[num\_nodes][num\_nodes];
    int** matrix = newint[num\_nodes][num\_nodes];
    for(int i=0; i<num\_nodes; i++)
        matrix[i] = newint[num\_nodes];
    for(int j=0; j<num\_nodes; j++)
        matrix[i][j] = edges\_list[i][j];
    int sum = 0;
    for (int i = 0; i < num\_nodes; i++)
    {
        for (int c = 0; c < num\_nodes; c++)
        {
            for (int d = 0; d < num\_nodes; d++)
            {
                if (i < num\_nodes)
                    edges\_list[i][i] = 1;
                sum += matrix[c][k]*matrix[k][d];
                result[c][d] = sum;
                sum = 0;
            }
        }
    }
}
```

### Time Complexity - $O(V^3)$, space complexity - $O(V^2)$, where $V$ is the number of nodes
This algorithm will be operating on \( O(V^3 \log V) \) time complexity, where \( V \) is the number of vertices.

**Matrix Powering** is the function which has a while loop, where the value of \( n \) becomes half with each iteration, which is of \( O(\log V) \) time complexity, later each conditional statement is calling **matrix_multiplication** function, which has three loops embedded and of \( O(V^2) \). This total algorithm thus gives a rise to the complexity of \( O(V^3 \log V) \).

The code snippet provided demonstrates the implementation of the matrix powering algorithm. The algorithm iterates for `num_nodes` time, hence the time complexity of the algorithm is \( O(V^3) \).

Similarly the space complexity of the algorithm is \( O(V^2) \) as we are using two multidimensional arrays having dimension `num_nodes * num_nodes` at maximum.

### Improving the Time Complexity

We can improve the time complexity of the above mentioned algorithm by using **Euler's Fast Powering Algorithm**, that is based on Binary Exponentiation technique for getting a matrix to the nth power.

This algorithm will be operating on \( O(V^3 \log V) \) time complexity, where \( V \) is the number of vertices.
for (int i = 0; i < SIZE; i++)
for (int j = 0; j < SIZE; j++)
a[i][j] = res[i][j];
}

// matrix powering to nth power
void matrix_powering(int** a, int n, int** res, int num_nodes)
{
    identity_matrix(res, num_nodes);
    while (n > 0) {
        if (n % 2 == 0) {
            matrix_multiplication(a, a, num_nodes);
            n /= 2;
        } else {
            matrix_multiplication(res, a, num_nodes);
            n--;
        }
    }
    transitive_closure(res, num_nodes);
}

int main()
{
    int num_nodes, num_edges;
    cin >> num_nodes >> num_edges;
    int** edges_list = new int*[num_nodes];
    for(int i=0;i<num_nodes;i++)
    {
        edges_list[i] = new int[num_nodes];
        for(int j=0;j<num_nodes;j++)
        {
            edges_list[i][j] = 0;
        }
    }
    for(int i=0;i<num_edges;i++)
    {
        int first_node, second_node;
        cin >> first_node >> second_node;
        edges_list[first_node][second_node] = 1;
        if(i<num_nodes)
        edges_list[i][i]=1;
    }
    cout << "Input Adjacent Matrix Graph:'" << endl;
    for(int i=0;i<num_nodes;i++)
    {
        for(int j=0;j<num_nodes;j++)
        {
            cout << edges_list[i][j] << "";
        }
        cout << endl;
    }
    int** result = new int*[num_nodes];
    for(int i=0;i<num_nodes;i++)
    {
        result[i] = new int[num_nodes];
    }
    for(int j=0;j<num_nodes;j++)
    {
        result[i][j] = 0;
    }
    matrix_powering(edges_list, num_nodes, result, num_nodes);
}

References
[1] Tripathy, Abhijit (2021): Transitive Closure Of A Graph using Floyd Warshall Algorithm. figshare. Online resource. https://doi.org/10.6084/m9.figshare.14721555.v1

Author Profile
Abhijit is the founder of Edualgo Academy and currently pursuing Bachelor of Technology in Computer Science and Engineering in Guru Ghasidas Vishwavidyalaya, Bilaspur. He is interested in Mathematical Computing, algorithms and machine learning.