The Role of Stratification in Decomposing a DAG into Spanning Trees

Supervisor: Dr. Yangjun Chen
By: Daniel Levy

1. Introduction:

Graph reachability studies the connections between vertices in directed acylic graphs (DAG) and the capability of reaching a vertex from another vertex. Graph reachability applications span across many different areas. Social networking sites can apply graph reachability to discover whether there is a relationship between two people. Navigation software can use graph reachability to ascertain if there is a path between two locations.

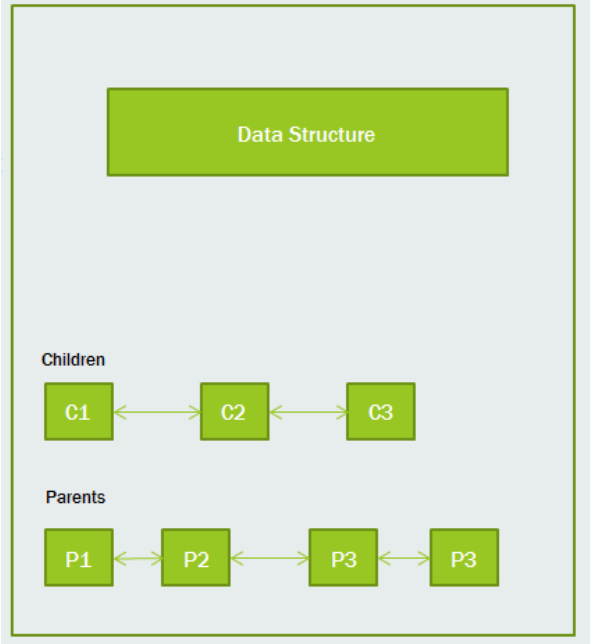
The work presented in this paper is meant to study the preliminary steps for compressing transitive closure by means of decomposing a DAG into spanning trees [1] [2] [3]. Section 2 will discuss the data structures used in this paper. Section 3 will discuss methodologies for creating large randomized graphs which are used as test data for work in this paper. Section 4 will discuss the two algorithms used for discovering strongly connected components – Tarjan's Algorithm [4] and Kosaraju's Algorithm [5]. Section 5 will discuss topological sorting. Section 6 will discuss the stratification algorithm [1] [2], which is the main focus of this paper. Section 7 will discuss the applications of stratification and future work. Section 8 will conclude the paper and summarize results.

All algorithms and test in this paper were run on a 4GB Athlon 64 X2 5200 2.70 GHz computer.

2. Data Structures:

Vertex:

The vertex structure contains a pointer to a data structure which can be used to store the data related to the vertex. For example, if the graph is being used to represent connections in a social network, the data structure could point to a person object. Links to the vertices' children and parents are stored as two separate linked lists.



SCC Vertex:

This is a modified version of the basic vertex structure that is used while searching for SCCs. After the SCC algorithm is run, new components are created which are compressed into single vertices. The SCC vertex structure adds a link from the original vertex to the compressed component vertex.

Stratification Vertex:

This is a modified version of the basic vertex structure that is used when performing the stratification algorithm. This new structure adds a pointer to the stratification result for this vertex.

3. Creating Graphs:

Creating large graphs is required to test the stratification algorithm's performance. The first step in creating a graph is creating a set of vertices *V*. A set of edges must then be created to connect vertices. Two different methods of creating edges have been implemented, randomized linking and top-down linking, which are discussed below. Additionally, we have the capability of reading graphs from text files.

Randomized Linking:

This method randomly selects a vertex *A'* from *V* and then proceeds to randomly select a second vertex *B'*. A directed edge is created between *A'* and *B'* where *A'* is the parent, and *B'* is the child. Before this process is finalized, we must first check that this edge has not already been created. The process is repeated until the desired numbers of edges have been created. This method is fully randomized without constraints and simulates many real-world scenarios very well; however, it creates many large cycles which makes it a poor choice for the SCC algorithms. The run-time to create a graph with randomized linking is O(E+V) where E is the number of edges and V is the number of vertices.

Top-down Linking:

This method randomly selects a vertex *A'* from *V* and then randomly selects a second vertex *B'* where *B'* must be located "after" *A'*. "After" is defined as being found later in an ordered list of the vertices. For example, the vertices *G'* and *Z'* are both located after the vertex *E'*. The vertices must have quantitative data structures to use this method. The run-time to create a graph with top-down linking is O(E+V) where E is the number of edges and V is the number of vertices.

4. Strongly Connected Components

A directed graph *G* is strongly connected if and only if there is a path from each vertex in *G* to every other vertex in *G*. A strongly connected component is a subgraph that fulfills the property of strong connectivity.

Compressing each strongly connected component in a graph to a single vertex while maintaining the component's edges results in a directed acylic graph (DAG). DAGs are required for many algorithms, such as Dr. Chen's stratification algorithm and the topological sorting algorithm.

Two SCC algorithms have been implemented for comparison: Tarjun's algorithm and Kosaraju's algorithm. The pseudocode for both of these algorithms is presented below.

Kosaraju's algorithm:

Computes the stratification levels for a graph using Dr. Chen's stratification algorithm
Input: Dag G
Output: An array of the stratification levels for G
Set $childrenFound$ to 0 for all vertices;
$level_1$ = all vertices with no children;
Set $i = 1$;
while $level_i$ *is nonempty* **do**
    v = set of all vertices in $level_i$;
    if $v \rightarrow childrenFound$!= $v \rightarrow children$ **then**
        remove v from $level_i$;
    end
    $level_{i+1}$ = all parents of $level_i$;
    p = set of all vertices in $level_{i+1}$;
    $vp \rightarrow childrenFound++$;
    $i++$;
end

Tarjan's algorithm

```
algorithm tarjan is
  input: graph G = (V, E)
  output: set of strongly connected components (sets of vertices)

  index := 0
  S := empty
  for each v in V do
    if (v.index is undefined) then
      strongconnect(v)
    end if
  repeat

  function strongconnect(v)
    // Set the depth index for v to the smallest unused index
    v.index := index
    v.lowlink := index
    index := index + 1
    S.push(v)

    // Consider successors of v
    for each (v, w) in E do
      if (w.index is undefined) then
        // Successor w has not yet been visited; recurse on it
        strongconnect(w)
        v.lowlink := min(v.lowlink, w.lowlink)
      else if (w is in S) then
        // Successor w is in stack S and hence in the current SCC
        v.lowlink := min(v.lowlink, w.index)
      end if
    repeat

    // If v is a root node, pop the stack and generate an SCC
    if (v.lowlink = v.index) then
      start a new strongly connected component
      repeat
        w := S.pop()
        add w to current strongly connected component
      until (w = v)
      output the current strongly connected component
    end if
  end function
```

Results:

To test these algorithms, we create randomized graphs of different sizes with both randomized and top-down linking methods. We then run both Tarjan's algorithm and Kosarju's algorithm on the same graph. Both algorithms return the same sets of components for a given graph. All results were averaged over five trials. Finally, we compute the time to reconnect all the edges of the components.

Randomized linking:

#Vertices	#Edges	Tarjun (ms)	Kosaraju (ms)	Connecting Components (ms)
10,000	20,000	12	16	6
50,000	130,000	41	66	34
80,000	160,000	73	108	61
300,000	500,000	311	470	155
300,000	1,200,000	438	692	277325
1,000,000	2,200,000	1333	1980	760

Top-down linking:

#Vertices	#Edges	Tarjun (ms)	Kosaraju (ms)	Connecting Components (ms)
10,000	20,000	17.5	20	11
50,000	130,000	70	96	100
80,000	160,000	110	147	130
300,000	500,000	385	502	490
300,000	1,200,000	520	750	1260
1,000,000	2,200,000	1479	2090	2971

From the results of top-down linking and randomized linking we can see that Tarjun's algorithm performs at approximately 73.4% of the run-time of Kosarju's algorithm.

5. Topological Sorting:

Topological sorting is the process of creating a linear ordering for a DAG. If we select any edge in a graph *E*, with parent vertex *P'* and child vertex *C',* then *P'* will always appear prior to *C'* in the graph's linear ordering. To compute the topological sorting of a DAG *G*, we perform a DFS on *G* and store each vertices' finishing time in an array. The order of this array is the graph's linear ordering. Note that there are many different possibilities for the linear ordering of a DAG.

#Vertices	#Edges	Topological Sorting (ms)
10,000	20,000	4
50,000	130,000	13
80,000	160,000	29
300,000	500,000	56
300,000	1,200,000	77
1,000,000	2,200,000	181

6. Graph Stratification:

Graph stratification is the process of separating the vertices of a graph into individual levels based on their positioning in the graph. The leaf vertices will always be in the lowest level – level 1. Some of the leaf's parents will be in level 2 (this step is explained in the algorithm pseudo code below) and so on.

The graph stratification algorithm was created by Dr. Chen of the University of Winnipeg. For graph stratification to have real-world applications, it must run efficiently on large graphs. By implementing the algorithm in C++, we show that graph stratification can be completed in linear time – *O(E + V)* where *E* is the number of edges and V is the number of vertices in a graph.

```
Computes the stratification levels for a graph using Dr. Chen's stratification algorithm
Input: Dag G
Output: An array of the stratification levels for G
Set discoveredChildren to 0 for all vertices;
level_1 = all vertices with no children;
Set i = 1;
while level_i is nonempty do
    p = all parents of level_i ;
    for all vertices in p do
        discoveredChildren++;
        if discoveredChildren == actual number of children then
            Add vertex to level_{i+1};
        end
    end
    i++;
end
```

Results:

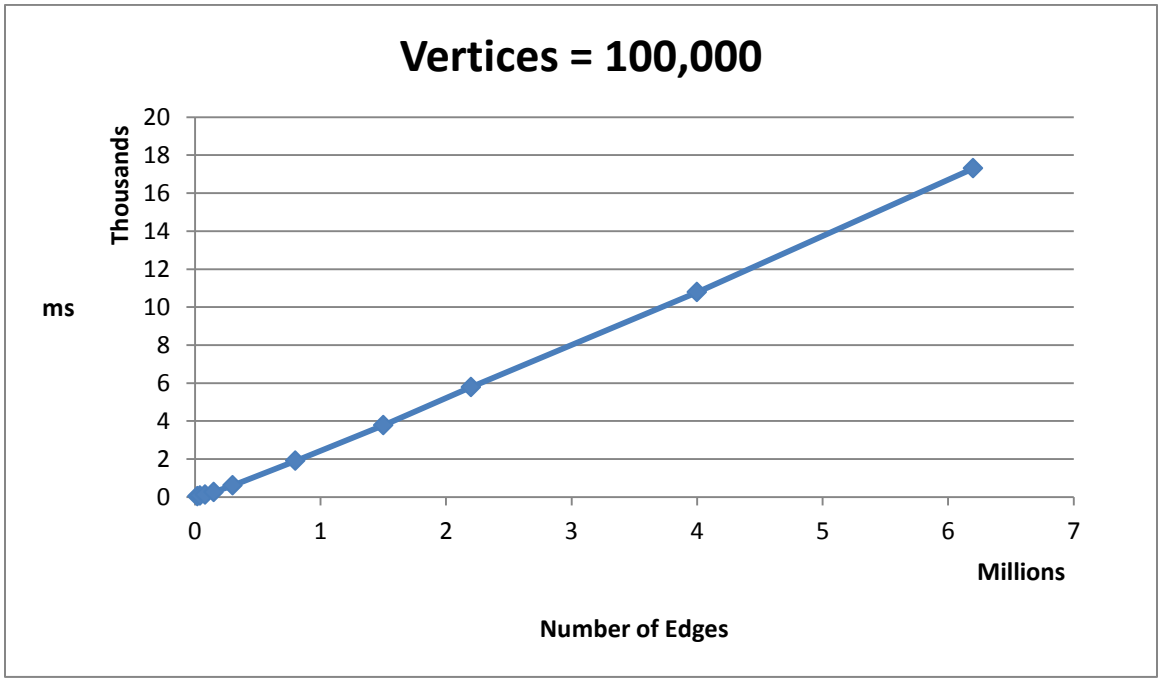
In this section, we will present the run-time results of running Dr. Chen's stratification algorithm under multiple conditions. For each set of results, ten separate trials were performed and averaged, each using a different graph.

This first set of results runs the stratification algorithm on different combinations of vertices and edges to demonstrate the algorithms capabilities.

#Vertices	#Edges	Run-time (ms)
10,000	20,000	33
50,000	130,000	240
80,000	240,000	473
300,000	500,000	960
300,000	1,200,000	2799
1,000,000	2,200,000	5068

In order to study the relationship between run-time and edges, we will hold the number of vertices constant at 100,000 while changing the number of edges.

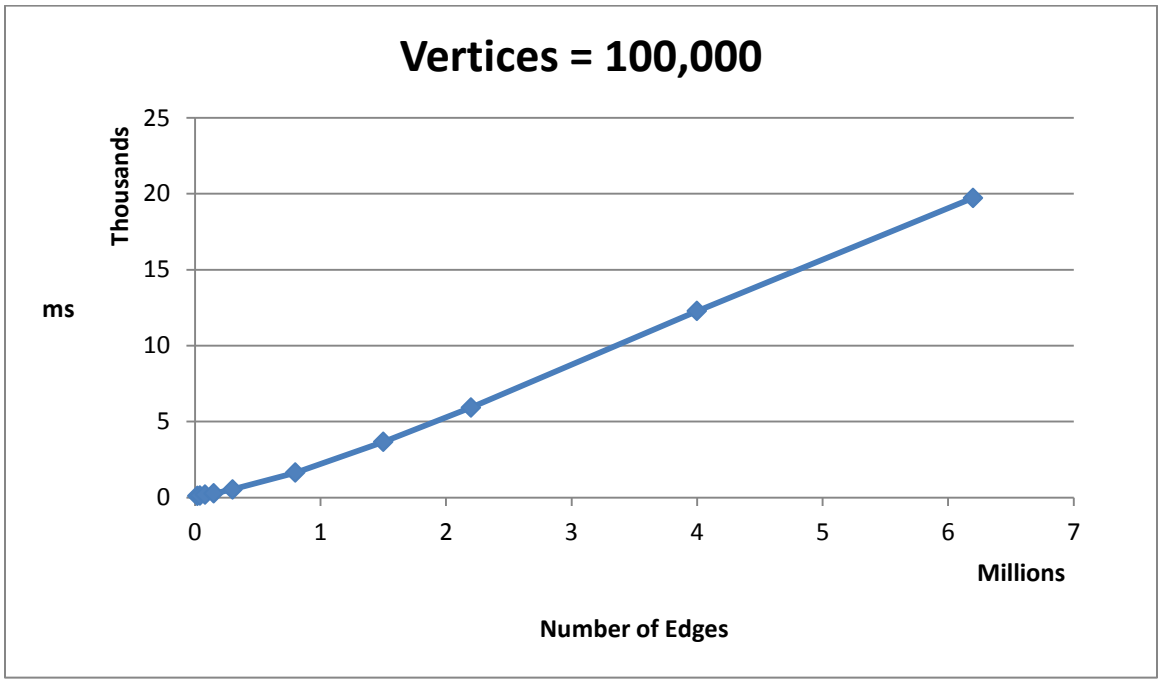
#Vertices	#Edges	Run-time (ms)
100,000	20,000	43
100,000	40,000	66
100,000	80,000	124
100,000	150,000	259
100,000	300,000	606
100,000	800,000	1905
100,000	1,500,000	3779
100,000	2,200,000	5784
100,000	4,000,000	10780
100,000	6,200,000	17301



We can clearly see from the graph that the run time is linear with respect to the growing number of edges.

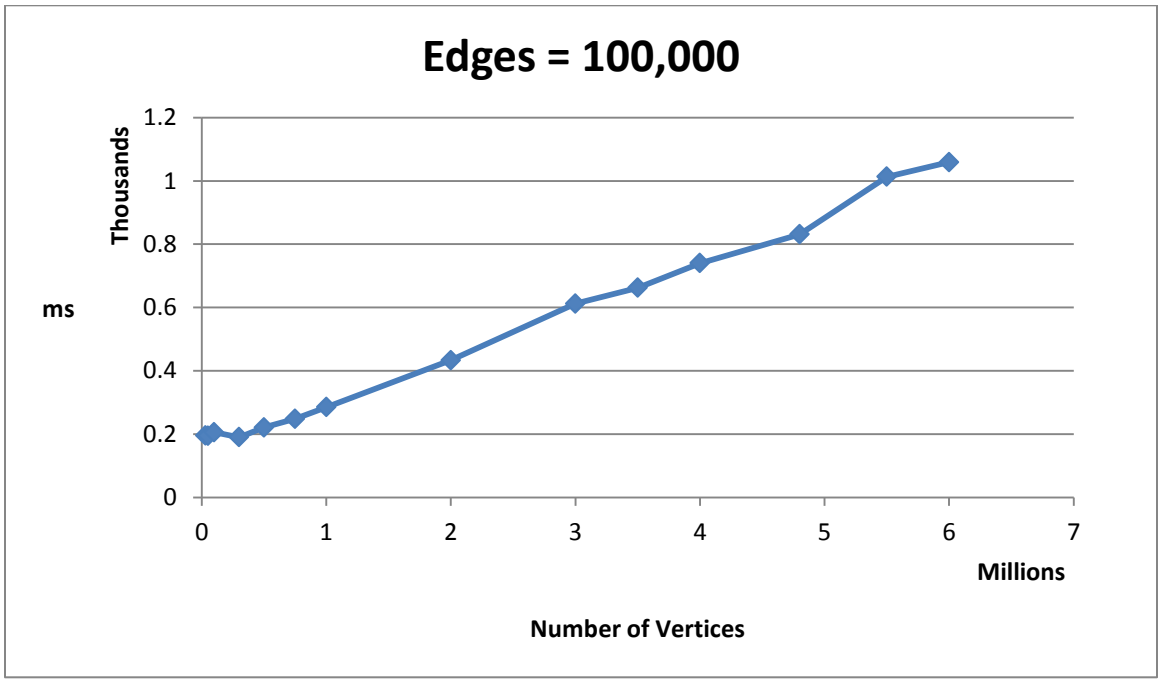
Here, the same experiment is performed on a larger number of vertices to confirm our result and demonstrate the linearity of the algorithm on a larger graph.

#Vertices	#Edges	Run-time (ms)
500,000	20,000	106
500,000	40,000	123
500,000	80,000	190
500,000	150,000	278
500,000	300,000	543
500,000	800,000	1645
500,000	1,500,000	3666
500,000	2,200,000	5925
500,000	4,000,000	12283
500,000	6,200,000	19719



In order to study the relationship between run-time and vertices, we will hold the number of edges constant at 100,000 while changing the number of edges.

#Vertices	#Edges	Run-time (ms)
30,000	100,000	195
50,000	100,000	194
100,000	100,000	206
300,000	100,000	190
500,000	100,000	221
750,000	100,000	248
1,000,000	100,000	285
2,000,000	100,000	433
3,000,000	100,000	612
3,500,000	100,000	662
4,000,000	100,000	740
4,800,000	100,000	831
5,500,500	100,000	1013
6,000,000	100,000	1059



Again, we see that the run-time is linear with respect to a growing number of vertices.

7. Future Work

Transitive closure is the process of computing reachability between two vertices. An inefficient way of discovering transitive closure is to compute the reachability between each pair of vertices in a graph, requiring O(n^2) memory. By decomposing DAGs into spanning trees we can compute the compressed transitive closure, requiring only O(e) memory [1].

The first step in compressing transitive closure is stratifying a DAG G (section 6). Each two adjacent levels from the stratification will form a bipartite graph. We must then calculate the maximum matching [2] in each bipartite graph. Matching occurs when no two edges have a common vertex. Maximum matching is beyond the scope of this paper. This process will form a set of chains which will be used to decompose G into a series of spanning trees, resulting in a compressed transitive closure.

Future work will consist of implementing the completed process for decomposing G into a series of spanning trees and proving that this process requires O(e).

8. Conclusion

The work presented in this paper demonstrates that graph stratification is indeed capable of linear run-times and can run on very large DAGs. This result is important in establishing the method of decomposing DAGs into spanning trees as a viable method for compressing transitive closure.

9. Bibliography

[1] C. Yangjun and C. Yiben, "Core Labeling: A New Way to Compress Transitive Closure," in *Signal Image Technology and Internet Based Systems*, 2008.

[2] Y. Chen, "General spanning trees and reachability query evaluation," in *2nd Canadian Conference on Computer Science and Software Engineering*, New York, 2009.

[3] C. Yangjun and C. Yibin, "An Efficient Algorithm for Answering Graph Reachability Queries," in *24th International Conference on Data Engineering*, 2008.

[4] R. Tarjan, "Depth-first search and linear graph algorithms," *Switching and Automata Theory,* pp. 114-121, 1971.

[5] S. R. Kosaraju, "Fast parallel processing array algorithms for some graph problems," *Theory of computing,* 1979.