EVALUATION OF MATRIX AND HYBRID TRANSITIVE CLOSURE ALGORITHMS WITH TRANSIT GRAPHS

by

Thomas Patrick Bailey Jr.

B.S., University of Idaho, 1999

A thesis submitted to the

University of Colorado Denver

in partial fulfillment

of the requirements for the degree of

Master of Science

Computer Science

2010
ABSTRACT

Knowing if you can get from point A to point B is almost as important as knowing a path between them. The Transitive Closure gives us this answer in the form of a reachability matrix of a graph. This thesis will discuss and test several transitive closure algorithms, a few foundational, for creating a reachability matrix/hybrid matrix from an adjacency matrix/hybrid matrix. Also, this thesis will evaluate the Floyd-Warshall, Warren, Agrawal-Jagadish, Yang-Yu-Liu-Dao-Wang-Phan, and Reduced Hybrid algorithms over graphs of varying sizes and present performance comparisons. Transitive Closure algorithms are useful in numerous areas and amongst them are database relationships and transit systems.

This abstract accurately represents the content of the candidate’s thesis. I recommend its publication.

Signed __________________________

Tom Altman
DEDICATION

To my wonderful wife, Lisa, a virtuous woman. Thanks for putting up with me as I struggled with my school work and listening to me as I thought out loud. Thanks for taking more than your fair share of our daily duties while I completed this thesis.

To my lovely children, Corbin and Kyndall, I hope you reach as far as you can in life and are blessed by learning the wonders of God’s creation.
ACKNOWLEDGEMENT

Thanks to my advisor, Tom Altman, for your time and commitment as I wavered. Also, I would like to thank Professor Gethner for launching my interest in graph theory. Thanks to all the hard working staff at CU Denver who can appreciate what it means to be a student with both a job and a family.
TABLE OF CONTENTS

Figures .................................................................................................................. vii
Tables ................................................................................................................ viii

Chapter
1. Inspiration ........................................................................................................... 1
2. Introduction ......................................................................................................... 1
3. Definitions .......................................................................................................... 2
   3.1. Directed/Acyclic Graph ................................................................................. 2
   3.2. Adjacency Matrix ......................................................................................... 3
   3.3. Reachability Matrix ..................................................................................... 4
   3.4. Topological Sort ........................................................................................... 5
4. TC Sequential Matrix-Based Algorithms .......................................................... 5
   4.1. Floyd-Warshall’s Algorithm ......................................................................... 5
   4.2. Warren’s Algorithm .................................................................................... 8
5. TC Hybrid Matrix-Based Algorithm ................................................................ 11
   5.1. Agrawal-Jagadish Hybrid Algorithm ............................................................ 11
   5.2. Yang-Yu-Dao-Wang-Phan’s Hybrid Algorithm ............................................. 15
   5.3. The Proposed Reduced Hybrid Algorithm .................................................... 19
6. TC Parallel Algorithms ....................................................................................... 29
   6.1. Yang-Yu-Dao-Wang-Phan’s Hybrid Algorithm ............................................. 29
   6.2. The Proposed Reduced Hybrid Algorithm .................................................... 31
7. Additional TC Algorithms ................................................................................. 34
   7.1. Matrix Multiplication .................................................................................. 34
   7.2. Graph Based ................................................................................................ 34
   7.3. Maintaining TC ........................................................................................... 35
8. Performance Analysis ......................................................................................... 35
   8.1. Results ........................................................................................................ 35
   8.2. Conclusions ................................................................................................ 46
Appendix .................................................................................................................. 47
Bibliography ............................................................................................................ 134
LIST OF FIGURES

Figure 3.1 Directed graph with cycle ................................................................. 3
Figure 3.2 Acyclic directed graph (DAG) ............................................................. 3
Figure 3.3 Transitive closure of figure 3.2 .......................................................... 4
Figure 5.1 Row block ....................................................................................... 13
Figure 5.2 $P(j)$ and upper triangular matrix ...................................................... 16
Figure 5.3 Rows block upper right triangle .......................................................... 18
Figure 5.4 Fixed route ....................................................................................... 19
Figure 5.5 Fixed route w/ transfer edges ............................................................. 20
Figure 5.6 Start/End nodes of fixed route ........................................................... 21
Figure 5.7 Start/End nodes split ....................................................................... 22
Figure 5.8 Connecting start end nodes ............................................................... 22
Figure 5.9 Transit graph ................................................................................... 23
Figure 5.10 Required accessible rows ............................................................... 25
Figure 5.11 Required accessible rows (path $>>$ adjacent edges) ....................... 27
Figure 6.1 $K$ limit, immediate non-connected rows ....................................... 32
Figure 8.1 Time vs. node count comparison of 5 algorithms (memory) ............. 36
Figure 8.2 Small-disk test results (milliseconds) .............................................. 37
Figure 8.3 Small disk test results Floyd-Warshall removed (milliseconds) ........ 38
Figure 8.4 Mid-disk/blocked/queue test results .............................................. 40
Figure 8.5 Mid-disk/blocked/queue test results (Yang and Warren removed) .... 41
Figure 8.6 Large blocked/queued test ............................................................. 43
Figure 8.7 Best algorithm very large node count test ..................................... 45
LIST OF TABLES

Table 8.1 System details ........................................................................................................ 35
Table 8.2 Small memory test results (milliseconds) ................................................................. 37
Table 8.3 Small-disk I/O test results (milliseconds) ................................................................. 39
Table 8.4 Mid-disk/blocked/queue test results (milliseconds) .................................................. 42
Table 8.5 Large-disk/blocked/queue/64bit/threads test results (milliseconds) ...................... 44
Table 8.6 Very large graphs test results (milliseconds) ............................................................ 46
1. Inspiration

In 2008, I started to commute to work via the local transportation authority, RTD. During that time, I utilized RTD’s website to plan my route. On their website, I selected an approximate start time and a destination and it provided several route options. The first iteration, of using this tool, yielded a route of roughly 90 minutes, which I utilized for several days. However, the distance from home to my place of work did not justify a 90 minute commute time. I went back to their website and began to experiment and adjust start times, which yielded an alternate trip with a travel time of roughly 60 minutes. I used this faster route for a few days then tried some additional start times. I eventually found a route of less than 45 minutes. I then asked myself, “Why don’t they use a tool to discover a fastest route in a transit system over a given time range?”

Over the past few years I have considered this problem and tried to find or create a viable solution. One tool I think will work, and will hopefully work fast enough for real world use, would require a reachability matrix as a part of the solution. To that end, I have been researching and writing this paper on Transitive Closure in relation to graph theory.

2. Introduction

The first algorithm proposed to solve the transitive closure (TC) of a directed graph was the Floyd-Warshall, named after Stephen Warshall and Robert Floyd [24, 11]. This algorithm was independently discovered by Bernard Roy in 1959 [5] and Stephen Warshall [24] 1962, and later modified by Robert Floyd [11] to find the length of the shortest path between nodes. Since then, several different matrix
based TC algorithms have been discovered [23, 17, 4], as well as, hybrid algorithms [1, 25, 7] all with varying degrees of efficiencies.

In this paper, I will discuss and evaluate five different algorithms: the Floyd-Warshall [5, 24, 11], Warren [23], Agrawal-Jagadish [1], Yang-Yu-Liu-Dao-Wang-Phan [25], and a Reduced Hybrid algorithm.

The Reduced Hybrid algorithm, which is based on Agrawal-Jagadish [1], is presented in section 5.3. The Reduced Hybrid can only handle a specific type of acyclic directed graph, called a transit graph. A transit graph models a transportation system where the goal is routing a passenger, be it a person or package, through a fixed route transportation system. More specifics characteristics of the transit graph will be covered with the description of the Reduced Hybrid algorithm.

3. Definitions
3.1. Directed/Ayclic Graph

A directed graph, also known as a digraph, is a graph where the edges connecting nodes are one-way (see Figure 3.1). The digraph can be denoted by the following, $G = (n, E)$, where $n$ is the set of nodes, and $E$ is the set of edges for the graph. An edge is defined as $e = (i, j)$, where $i$ is the parent node, and $j$ is the child node (a path exists from $i$ to $j$). For more details, please read Sedgwick [18].
An acyclic directed graph, also known as a DAG, is a digraph where no cyclic path exists. A path is a sequence of nodes and for each node in the series there exists an edge to the next node in that sequence. A cyclic path repeats at least one node in its sequence. An example of a cyclic path is $Path = (1, 2, 3, 1)$, taken from Figure 3.1, where node 1 is repeated once. An example of a graph with no cycles, and therefore a DAG, can be seen in Figure 3.2 where no cyclic path exists.

![Figure 3.1 Directed graph with cycle](image1)

![Figure 3.2 Acyclic directed graph (DAG)](image2)

### 3.2. Adjacency Matrix

The adjacency matrix, $M_G$, of a graph is a $n \times n$ matrix, where $n$ is the number of nodes in the graph and ...

$$M_G[i, j] = \begin{cases} 
1 & \text{if } \{i, j\} \in E \\
0 & \text{otherwise}
\end{cases}$$
For example, the adjacency matrix of Figure 3.2 is …

\[
\begin{pmatrix}
0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 \\
\end{pmatrix}
\]

3.3. Reachability Matrix

The reachability matrix \( R_g \) of a graph is the adjacency matrix of the transitive closure of a graph. For example, the transitive closure of the graph in Figure 3.2 is shown in Figure 3.3.

![Figure 3.3 Transitive closure of figure 3.2](image)

The reachability matrix of Figure 3.3 is …

\[
\begin{pmatrix}
0 & 1 & 1 & 1 & 0 \\
0 & 0 & 1 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 \\
\end{pmatrix}
\]
3.4. Topological Sort

A topological sort is an ordering of the nodes of a DAG. In this ordering, every node precedes other nodes with which it shares an outbound edge. A topological sort of the graph, in Figure 3.3, is \{1, 2, 5, 3, 4\}. However, this is not the only topological sort of this graph. Another valid sort is \{1, 5, 2, 3, 4\}. In a reverse topological sort, every node succeeds any node with which it shares an outbound edge. A reverse topological sort of the graph, in Figure 3.3, is \{4, 3, 5, 2, 1\}. There are several algorithms for sorting a DAG, such as Kahn [14], Knuth-Szwarcfiter [15], and also Tarjan [22]. Tarjan’s algorithm, which has a run time of \(O(e)\), sorts a graph as a byproduct of determining its strongly connected components.

4. TC Sequential Matrix-Based Algorithms

The efficient computation of the TC of a graph was first explored by Bernard Roy in 1959 [5] and Stephen Warshall in 1962 [24] with sequential matrix based algorithms. Since then, many sequential matrix based algorithms have been presented, based in large part, on the work of Roy and Warshall [23, 1, 19, 2]. This section will review the Floyd-Warshall [5, 24, 11] and Warren [1] algorithms which will be compared against several hybrid algorithms in Section 8.

4.1. Floyd-Warshall’s Algorithm

Floyd-Warshall algorithm’s, also referred to as the Roy-Floyd Algorithm, was independently discovered by Bernard Roy [5] and Stephen Warshall [24]. It computes the TC of a graph by comparing all possible routes between two nodes sequentially. In his paper, ”A Theorem on Boolean Matrices” [24], Stephen
Warshall puts forth and proves Theorem 4.1.1 describing how to create the TC from an adjacency matrix.

**Theorem 4.1.1 [24]**

Given a square $(d \times d)$ matrix, $M$ each of whose elements $m_{ij}$ is 0 or 1. Define $M'$ by $m'_{ij} = 1$, if and only if, either $m_{ij} = 1$, or there exists integers $k_1, \ldots, k_n$, such that $m_{i_{k_1}k_{k_2}} = \cdots = m_{k_{n-1}k_n} = m_{k_n} = 1; m'_{ij} = 0$ otherwise. Define $M^*$ by the following construction:

0. Set $M^* = M$.
1. Set $i = 1$.
2. $(\forall j \exists: m^*_{ji} = 1)(\forall k) set \ m^*_{jk} = m^*_{jk} \lor m^*_{ik}$.
3. Increment $i$ by 1.
4. If $i \leq d$, go to step 2; otherwise stop.

We assert $M^* = M'$

The input for the algorithm is an adjacency matrix. The result is a reachability matrix.

**Algorithm**

```
    For i from 1 to n
        For j from 1 to n
            If $M[j, i] = 1$
                For k from 1 to n
                    If $M[i, k] = 1$
                        $M[j, k] = 1$
```

The algorithm starts by considering the first node, setting \( i \) to 1. Then it searches the \( i \)-th column of every row. If a column’s value is found to be one, then that node, node \( j \), has a path to node \( i \). Next, if a path can be found from node \( i \) to node \( k \), then we know a path exists from node \( j \) to node \( k \), through node \( i \). If this is the case, then \( M[j, k] \) is set to 1, indicating a path exist between \( j \) and \( k \). After every row \( j \) is considered for node \( i \), then \( i \) is incremented by one and the process is repeated until \( i \) is greater than the number of nodes. The runtime of this algorithm is \( O(V^3) \).

Floyd [11] presented a modification of the Warshall algorithm that not only finds the TC of a graph, but also finds the length of a shortest path between nodes. Floyd’s algorithm takes an adjacency matrix that contains the lengths between nodes. If no path between nodes exists, list the length as infinity. The Floyd algorithm is as follows.

**Algorithm (Floyd)**

\[
\text{For } i \text{ from 1 to } n \\
\quad \text{For } j \text{ from 1 to } n \\
\quad \quad \text{If } M[j, i] < \text{infinity} \\
\quad \quad \quad \text{For } k \text{ from 1 to } n \\
\quad \quad \quad \quad \text{If } M[i, k] < \text{infinity} \\
\quad \quad \quad \quad \quad s = M[j, i] + M[i, k] \\
\quad \quad \quad \quad \quad \text{If } s < M[j, k] \text{ then } M[j, k] = s
\]

This algorithm works like the Warshall algorithm, with the exception of maintaining the length of the shortest path between nodes. The result of this algorithm is a
reachability matrix that contains the length of the shortest path, such that \( M[i, j] = \) length of the shortest path between \( i \) and \( j \).

### 4.2. Warren’s Algorithm

In 1975, Henry Warren [23] presented a modification to the Floyd-Warshall Algorithm [5, 24, 11] to address the I/O issue of paging. Warren points out that the Floyd-Warshall algorithm, “scans by columns (so that it may ‘or’ by rows)”. This nature of the Floyd-Warshall algorithm requires a great deal of I/O between disk and main memory for matrices too large to fit in memory.

When storing an adjacency matrix memory, size was an issue in 1975, and is still an issue in 2010. Warren mentions using an IBM System/360, which was introduced in 1964 and had a memory capacity between 4KiB to 8MiB. The largest adjacency matrix that can be stored in 4KiB could represent a graph with 181 nodes. For 8MiB, it jumps up to 8,192 nodes. For the system I will be using in evaluating TC algorithms, which has 12GiB, it can, in theory, handle a matrix of at most 321,060 nodes. Of course, this does not account for OS, programs, etc., running in the same memory, which would reduce this number.

Transit graphs, that represent real systems, contain 1,800,000 to 8,000,000 nodes. This would require 377GiB – 7TiB of memory to contain the entire adjacency matrix. These numbers are achievable on a modern supercomputer, and if Moore’s law holds out, on a home PC in a decade or two. However, compared to other problems, these are miniscule. For the foreseeable future, brute force storage may not be the answer.
Warren’s algorithm scans by rows, removing the need to or by columns. It does, however, require a second pass at the matrix “to add the final arc” [23].

Algorithm

For $i$ from 2 to $n$
    For $j$ from 1 to $i - 1$
        If $M[i, j] = 1$
            For $k$ from 1 to $n$
                $M[i, k] = M[i, k] \lor M[j, k]$

For $i$ from 1 to $n - 1$
    For $j$ from $i + 1$ to $n$
        If $M[i, j] = 1$
            For $k$ from 1 to $n$
                $M[i, k] = M[i, k] \lor M[j, k]$

The algorithm first considers row $i$, starting from row 2 to row $n$. Then it checks every $j$-th bit of row $i$ starting from the first bit to the $i$-th bit, giving a stepped calculation. If the $j$-th bit is found to be ‘1’, then an ‘or’ operation is performed on the $k$-th bit of row $i$ against the $k$-th bit of row $j$, and the result stored in the $k$-th bit of row $i$, where $k$ runs from 1 to $n$ thus completing the first pass.

The second pass considers row $i$, starting from row 1 to row $n-1$. Then, it checks every $j$-th bit of row $i$ starting from the $i$-th + 1 bit to the $n$-th bit, giving a reverse stepped calculation. If the $j$-th bit is found to be ‘1’, then an ‘or’ operation is performed on the $k$-th bit of row $i$ against the $k$-th bit of row $j$, and the result stored in the $k$-th bit of row $i$, where $k$ starts from 1 to $n$ thus completing the second pass.
The runtime of this algorithm is $O(V^3)$, which is the same as Floyd-Warshall’s algorithm, but it has two advantages over Floyd-Warshall. One advantage, for the case of a very sparse graph, is that Warren’s algorithm “will execute faster than Warshall’s by a factor of approximately $w$, and in the worst case it is about equal to Warshall’s” [23]. Here $w$ is the number of bits that can be tested at a time. The second advantage is, for a sparse adjacency matrix. Roughly $\frac{n}{2}$ fewer disk read operations are required.

The reduction in disk I/O can be shown in this manner. Assume a row can be read into memory from disk in one $rowRead$ function. The Matrix can then be read in $n$ calls to $rowRead$. The worst case for the Floyd-Warshall algorithm requires $n$ $rowRead$ calls per row, giving $n^2$ $rowRead$ calls. For the Warren algorithm, the fastest case is if the matrix is all 0’s. For the best case, this gives one $rowRead$ per row, for the first part of Warren’s algorithm, and the same for the second. Giving a total of $2n$ $rowRead$ calls for the best case.

$$\frac{n^2\text{rowreads}}{2n\text{rowReads}} = \frac{n}{2} \text{ fewer rowReads (best case)}$$

In the worst case, which is a matrix of all 1’s, Warren’s algorithm for each row reads in an additional $d$ rows where $d \leq n$. Assuming it is $n$, we get $n^2$ $rowRead$ calls, for the first part of the Warren algorithm, and $n^2$ $rowRead$ calls for the second part giving $2n^2$ $rowRead$ calls. The worst case is approximately the same as Floyd-Warshall algorithm in disk I/O.
5. TC Hybrid Matrix-Based Algorithm

Hybrid algorithms try to, “Incorporate the optimization features of graph-based algorithms [7, 13] within a matrix framework” [1]. Agrawal and Jagadish presented this concept in their paper “Hybrid Transitive Closure Algorithms”[1]. This section will review the Agrawal-Jagadish [1] and Yang-Yu-Dao-Wang-Phan [25] algorithms. The Reduced Hybrid Algorithm will also be presented, which is based on the Agrawal-Jagadish algorithm. All of these algorithms will be compared against the Floyd-Warshall and Warren algorithms in Section 8.

5.1. Agrawal-Jagadish Hybrid Algorithm

In 1990, Agrawal and Jagadish presented a hybrid algorithm for computing the transitive closure of a graph [1]. It is so named because they felt their solution was a hybrid of matrix-based solutions [5, 24, 11, 23] while utilizing optimization features of graph-based algorithms [7, 13]. The method that they propose can only be used for reverse topologically sorted DAG graphs.

First, Agrawal-Jagadish’s algorithm can be explained starting with an adjacency matrix. Take an adjacency matrix for an acyclic directed graph, and if unordered, sort the matrix in a reverse topological order. This sorting preparation step takes time, but is negligible as Kahn presented, in his paper, “Topological Sorting of Large Networks” [14], an algorithm that runs in $O(V + E)$. At this point the graph no longer needs to be stored in an $n \times n$ matrix, but rather it can be stored in an $n \times i$ lower triangular matrix, where $n$ is the number of nodes in the graph and $i$ is the row number. This reduces the storage size by half.
Algorithm (Basic)

For $i$ from 1 to $n$
    Copy row $i$ into Temporary row $i'$

For $j$ from $i - 1$ to 1
    If $i'[j] \neq 0$
        Call $add\_succ(i, j, i')$

Procedure $add\_succ(i, j, i')$

For $k$ from 1 to $j - 1$
    If $j[k] = 1$
        If $i'[k] = 1$
            $i'[k] = 0$
        Else
            $j[k] = 1$

The algorithms starts with the first row being copied to a separate variable, $i'$. Then, it iterates from right to left through the copied row from the $i$-th $-1$ bit to the first bit of the copied row. If the $j$-th bit of the copied row is set to 1, iterate through the $j$-th row, from 1, to $j - 1$. If the $k$-th bit of the $j$-th row is set to 1 and the $k$-th bit of the copied row is also set to 1, then set the $k$-th bit of the copied row to 0, otherwise set the $k$-th of the original row to 1.

Setting the $k$-th bit of the copied row to 0, in the case that it was 1, is where the optimization takes place. Since a connection between $i$ and $j$ exists, all of the successor nodes of $j$-th row will be added to the $i$-th row. If one of the successor nodes is already present in the $i$-th row, it can be removed from the copied row so it is not considered, since it and all its successors have already been added. There is no need to include what has already been added.
Agrawal and Jagadish also present a blocked form of their algorithm to help with memory and disk I/O. The basic Agrawal-Jagadish Algorithm is already suited for disk I/O performance similar to Warren’s Algorithm. However, this can be improved upon by keeping more rows accessible in memory.

The matrix is partitioned into blocks of rows from row $i_s$ to $i_e$, as shown in Figure 5.1. For each block, the rows $i_s$ to $i_e$, are read into memory and a copy of these rows are created into $i'_s$ to $i'_e$ for optimization calculations. The off-diagonal block portion is processed first, from right to left, from $i_s$ -1 to 1, and from top to bottom, per column. If the given $j$-th column is set to 1, then fetch the $j$-th row from disk into memory. If it is not already in there, then process it with $add\_succ$. This is where additional optimization takes place. The $j$-th row will only be read into memory once, for all the block, from rows $i_s$ to $i_e$, as it is used by them all before
the next $j$-th row is considered. After the off-diagonal block elements has been processed, the diagonal-block is processed for rows $i_s$ to $i_e$.

**Algorithm (Blocked)**

Assume matrix is partitioned into $m$ blocks
Do the following for each block $b_i = 1, 2, ..., m$

Let the block $b_i$ consists of rows $i_s$ to $i_e$

Fetch rows $i_s$ through $i_e$ into memory
Copy into rows $i_s'$ through $i_e'$

//Process elements in off-diagonal block
For $j$ from $i_s - 1$ to 1
    For $i$ from $i_s$ to $i_e$
        If $i'[j] \neq 0$
            //Fetch the row $j$ if not already in memory
            Call $add\_succ(i, j, i')$

//Process elements in diagonal block
For $i$ from $i_s$ to $i_e$
    For $j$ from $i$ to $i_s$
        If $i'[j] \neq 0$
            Call $add\_succ(i, j, i')$

**Procedure** $add\_succ(i, j, i')$
For $k$ from 1 to $j - 1$
    If $j[k] = 1$
        If $i'[k] = 1$
            $i'[k] = 0$
        Else
            $i[k] = 1$
Unlike the Floyd-Warshall’s algorithm, the Agrawal-Jagadish’s compares rows to rows rather than rows to columns. This is similar to the Warren algorithm, but unlike Warren’s, Agrawal-Jagadish’s has optimization features and does not require a second pass at the matrix.

5.2. Yang-Yu-Dao-Wang-Phan’s Hybrid Algorithm

Yang, Yu, Liu, Dao, Wang, and Pham present a hybrid algorithm for computing the transitive closure, of a DAG, in their paper, “A Hybrid Transitive Closure Algorithm for Sequential and Parallel Processing” [25]. Their hybrid algorithm solves the transitive closure problem, in such a way, that it can support sequential and parallel processing of the solution.

The algorithm requires a topologically sorted adjacency matrix for the DAG, which results in an “upper triangular matrix” [25] versus a lower for [1, 7] (see Figure 5.2). Also, an original parent relationship, $P(j)$, must be created and maintained (see Figure 5.2). “$P(j)$ is the set of all parents of the $j$-th column” [25].
**Algorithm (Basic)**

For i from 1 to n - 1
  
  Initiate $t_{i,j} = 0$ for $i < j \leq V$

For j from $i + 1$ to n
  
  Call Procedure $CT(i, j)$

**Procedure $CT(i, j)$**

For each $x \in P(j)$
  
  If $x = i$ or $T_{i,x} = 1$
    
    $T_{i,j} = 1$ and terminate

Yang-Yu-Liu-Dao-Wang-Phan’s Hybrid algorithm works in this way: iterate through the rows in the reachability matrix, remove all edges in the row, iterate through the Parent relationships $P(j)$ from $i + 1$ to $n$, if $P(j)$ contains $i$ or $T(i, x) = 1$, where $x \in P(j)$, set $T_{i,j} = 1$. 

Figure 5.2 $P(j)$ and upper triangular matrix
A modified version of this algorithm is presented to help with I/O performance similar to Agrawal and Jagadish’s [1] idea of blocking. The basic algorithm is not suited well for I/O performance, as the \( P(j) \) must be called for each row for which it might be a parent.

**Algorithm (Blocked)**

Do the following for each block \( b_l = 1, 2, \ldots, m \)
- Let the block \( b_l \) consists of rows \( i_s \) to \( i_e \)
- Fetch rows \( i_s \) through \( i_e \) into memory
- For each row Initiate \( t_{i,j} = 0 \) for \( i < j \leq n \)

For \( j \) from \( i_s + 1 \) to \( n \)
- For \( i \) from \( i_s \) to \( \min(j - 1, i_e) \)
  - Call Procedure \( CT(i, j) \)

**Procedure \( CT(i, j) \)**

For each \( x \in P(j) \)
- If \( x = i \) or \( T_{i,x} = 1 \)
  - \( T_{i,j} = 1 \) and terminate

The matrix is partitioned into blocks of rows from \( i_s \) to \( i_e \), as shown in Figure 5.3. For each block, the rows \( i_s \) to \( i_e \), are read into memory. Each row is initiated to remove all edges in the row \( t_{i,j} = 0 \) for \( i < j \leq n \). The block is processed by columns, from left to right, so that \( P(j) \) for \( i_s < j \leq n \) will only be read once per column per block and applied to every valid row within the block. This will greatly reduce the number of times \( P(j) \) will be accessed and will greatly improve I/O.
Algorithm (Blocked)

Do the following for each block $b_l = 1, 2, \ldots, m$

- Let the block $b_l$ consists of rows $i_s$ to $i_e$
- Fetch rows $i_s$ through $i_e$ into memory
- For each row $t_{i,j} = 0$ for $i < j \leq n$

For $j$ from $i_s + 1$ to $n$

For $i$ from $i_s$ to $\min(j - 1, i_e)$

Call Procedure $CT(i, j)$

Procedure $CT(i, j)$

For each $x \in P(j)$

- If $x = i$ or $T_{i,x} = 1$
  
  $T_{i,j} = 1$ and terminate
5.3. The Proposed Reduced Hybrid Algorithm

The hybrid algorithm of Agrawal and Jagadish [1] is the basis of the reduced hybrid algorithm introduced in this section. The problem that inspired me to research transitive closure has a very specific type of graph, a transit graph. The best way to explain a transit graph is to show how it is created.

![Figure 5.4 Fixed route](image)

A transpiration system, for example a regional bus/rail system, consists of fixed routes. Figure 5.4 shows a fixed route that has five nodes where it picks up and/or drops off passengers. Each of the five nodes has a specific geographical location and a time when the bus will arrive/depart from that location. As an example, node C’s location is 5th and Elm St. and its arrival/departure time is 1:30 PM.

To transfer within the system, edges are added between fixed routes. These edges are called transfer edges (see Figure 5.5). A transfer edge is added between all pairs of nodes from different fixed routes, if the nodes are within a transferable time and a transferable distance of each other. As an example, given a transferable time between 2 to 15 minutes, transfer edges can only be added between nodes that are at least 2 minutes and no more than 15 minutes from each other. Additionally, given a transferable distance of 500 meters, transfer edges can only be added between nodes that are at most 500 meters from each other.
Figure 5.5 shows the transfer edges, indicated in red, from the fixed route (A-E) in Figure 5.4 and between two other fixed routes (F-I and J-M). This graph is not the transit graph, however, the TC of this graph can be solved with my proposed algorithm, but the graph itself has a few problems with modeling a transportation system with fixed routes. The first problem is it produces invalid paths. A passenger will only get on a fixed route if they intend to travel between nodes (A passenger will not get on and off a bus at the same stop). The path \( \{ J, K, B, G, H \} \) has a passenger getting on and off at node B and this is an example of an invalid path.

Another problem is paths exist in the graph, from Figure 5.5, that contain superfluous nodes. These nodes do not require the passenger to take any action. For example, the path \( \{ J, K, L, M, D, E \} \) has two superfluous nodes K and L. The passenger does not get on, off, or transfer from these nodes, but rather stays on the fixed route. To fix these problems all of the fixed routes must be adjusted.
To fix these problems, identify nodes which are both start nodes and end nodes. These nodes must be split. A, B, C, and D are nodes at which a passenger can get on the fixed route, a start node (Figure 5.6). Nodes B, C, D, and E are nodes at which a passenger can get off the fixed route, an end node (Figure 5.6). A passenger cannot get off the fixed route at node A as it is the initial start of the route and likewise, a passenger cannot get on at node E, as it is the end of the fixed route.

The edges are then removed from the fixed route. Then, nodes which are both start and end nodes are split into two nodes. One of the two new nodes becomes a start node and the other becomes an end node (see Figure 5.7).

Figure 5.6 Start/End nodes of fixed route
Next, add *travel* edges. Travel edges represent a passenger’s entire trip along a fixed route. Each *start* node, of a fixed route, has an outbound edge to every *end* node, of the same fixed route, whose time is after the *start* node’s own time. Figure 5.8 represents the fixed route updated in this manner.
Transfer edges are added just as they were in Figure 5.5 with one exception. Start nodes can only connect to end nodes and vice versa. Figure 5.9 shows an update of Figure 5.5 using the revised fixed routes. This is the transit graph. Connecting the fixed routes in this manner removes the invalid paths and paths with superfluous nodes.

Figure 5.9 Transit graph
These fixes do come at a cost, nearly doubling the number of nodes and significantly increasing the number of edges, which will increase the computational cost of computing a reachability matrix. The good news is every path in a transit graph is a shortest path.

The reachability matrix of a transit graph follows some additional restrictions.

The first restriction is based on the concept that a passenger traveling through a transit system is limited on the length of time it takes to travel between the start and end nodes of their trip. A node can only reach other nodes within the node’s system imposed travel time limit. Any nodes beyond this limit are not considered.

Since each node in a transit graph is associated with a time, the graph can easily be ordered by reverse time. This ordering is a reverse topological sort. With the sorted graph and the knowledge of a start node path’s time limit, it can be said that a node $i$ can only connect to the previous $x$ rows in the reverse topologically sorted matrix. In other words, if node $i$ has a restriction stating that it can only connect to nodes within $m$ minutes, this is equivalent to $x$ rows.

If $x < i$, then the left portion of the row cannot contain reachability data. If it cannot contain reachability data, then it cannot contain adjacency data and never needs to be considered or stored (see Figure 5.10).

Each row now has a non-reachable portion defined that is between 0 and $i$ in length (see Figure 5.10). This can greatly reduce the size of the data that needs to be stored and analyzed. The reduced hybrid matrix is, at most, as large as the hybrid-matrix,
where the non-reachable portion always equals 0. In practice, for simulated transit systems, a 65-70% reduction in graph size has been seen compared to a hybrid-matrix.

It should be noted that because of the nature of the transit graph, no node that shares an adjacency edge with node \( i \), has a path to any of the other nodes that also have an adjacency edge with node \( i \). Therefore, the optimization step Agrawal-Jagadish’s algorithm contains can be dropped. However, my current simulation of a transit graph is incomplete and cannot currently account for this characteristic. For the time being the optimization will be included.
Algorithm (Basic)

For $i$ from 1 to $n$
    Copy row $i$ into Temporary row $i'$
    For $j$ from $i-1$ to $i_n$
        If $i'[j] \neq 0$
            Call $add\_succ(i, j, i')$

Procedure $add\_succ(i, j, i')$

For $k$ from $i_n$ to $j-1$
    If $j[k] = 1$
        If $i'[k] = 1$
            $i'[k] = 0$
        Else
            $i[k] = 1$

Each row only needs the rows contained within its required accessible rows (see Figure 5.11) to create its reachability version. With the non-reachable portion number we have per row, and the size of the row itself, we can calculate the maximum number of rows in the largest required accessible area. If this area does not exceed the available memory, a rolling queue of rows can be stored in memory to create each row’s reachability. Once a row is finalized, the first row in the queue is removed and the current, complete row is added to the queue. The rolling queue size can further be reduced, in some cases, by utilizing a second restriction on the graph.

The second restriction that can be applied, is based on the idea that the adjacency edges of node $i$ are among nodes that are within $e$ nodes as numbered from the
reverse-topologically-sorted matrix, where $e \ll i_d$ (see Figure 5.11). Therefore, the rows that will be accessed in creating the reachability of row $i$ are between $i_e$ and $i$. If there is sufficient memory to hold the maximum $|i_e - i|$ over all the rows, then it is possible to read each row into memory only once (to be processed and used in other row processes then removed and written to disk). This reduced-rolling-queue will greatly reduce the memory footprint required to process the reachability matrix. In practical cases a reduction of 80-85% in memory required has been observed. Of course, this is all dependent on how small $e$ is compared to $i_d$.

Figure 5.11 Required accessible rows (path $>>$ adjacent edges)

In addition to the same inputs needed for the basic algorithm, the rolling queue version requires an additional $n \times 1$ matrix that contains the $i_e$ length for each row.
Algorithm (reduced-rolling-queue)

\[ \text{maxRows} = \max(i_v \in V) \]

Store rows 1 to \( \text{maxRows} \) in rowCache

For \( i \) from 1 to \( n \)
   If \( j > \) largest row in rowCache
       Remove lowest row from rowCache
       And write it to disk
       Add row \( i \) to rowCache
   Copy row \( \text{rowCache}[i] \) into Temporary row \( i' \)
   For \( j \) from \( i - 1 \) to \( i_n \)
      If \( i'[j] \neq 0 \)
         Call \( \text{add_succ}(\text{rowCache}(i), \text{rowCache}(j), i') \)
   Write rows remaining in rowCache to Disk

Procedure \( \text{add_succ}(i, j, i') \)

For \( k \) from \( i_n \) to \( j - 1 \)
   If \( j[k] = 1 \)
      If \( i'[k] = 1 \)
         \( i'[k] = 0 \)
      Else
         \( i[k] = 1 \)
6. TC Parallel Algorithms

The large computational power needed to solve for the TC of a graph has resulted in a great deal of research into creating parallel algorithms to generate faster solution for computing TC [25, 3, 6]. This section will review a parallel version of the Yang-Yu-Dao-Wang-Phan [25] and a parallel version of Reduced Hybrid algorithms. The parallel version of the Reduced Hybrid algorithm will be compared with sequential TC algorithms in Section 8.

6.1. Yang-Yu-Dao-Wang-Phan’s Hybrid Algorithm

The blocked version of Yang-Yu-Liu-Dao-Wang-Phan’s hybrid algorithm allows the blocks to be processed individually with no need to get information from other blocks [25]. Since the blocks do not interact with each other, they can be processed in parallel over \( K \) processors, such that \( K = b_n \ t \), where \( b_n \) is the number of blocks the hybrid adjacency matrix is split across.

The nature of the blocking can be adjusted to suite the situation. If the processors share the same characteristics, such as speed and memory allocation, then the blocks can be modified to be proportional to each other. However, if the available processor’s characteristics are mixed, then the block sizes can be changed to accommodate the differences and to allow real time runs per block to be approximately the same (each takes a load according to its means). Also, the problem does not need to be split into \( b_n \) blocks. It could be split into even smaller blocks such that \( 1 \leq K \leq b_n \). If \( K < b_n \), then each processor can be assigned more than one block to complete. This would allow for each processor to require less memory than if \( K = b_n \).
Other issues to solve arise from this parallel solution. These issues include: dividing the blocks appropriately, transmitting the blocks and the $P(j)$ to each processor, and reassembling the hybrid-reachability matrix once the blocks have been processed.

**Algorithm (Blocked-Parallel)**

Split the hybrid-reachability matrix into $K$ blocks, where $K$ is the number of processors …

**For** $b_k$ where $1 \leq k \leq K$

Assign Block $b_k$ to processor $k$

Let the block $b_k$ consists of rows $i_s$ to $i_e$

Fetch rows $i_s$ through $i_e$ into memory (the entire block)

For each row Initiate $t_{i,j} = 0$ for $i < j \leq n$

**For** $j$ from $i_s + 1$ to $n$

**For** $i$ from $i_s$ to $\min(j - 1, i_e)$

Call Procedure $CT(i, j)$

**Procedure** $CT(i, j)$

**For** each $x \in P(j)$

If $x = i$ or $T_{i,x} = 1$

$T_{i,j} = 1$ and terminate
6.2. The Proposed Reduced Hybrid Algorithm

The Reduced-Hybrid algorithm can be run in parallel with a limited number of processors. Two details about the type of graph can be exploited to allow it to run in parallel.

First, each node of the transit graph, that the reduced-hybrid algorithm solves for, has a time component. The units being used in my graph are one minute intervals (other solutions could use 30 or 15 minutes). No node can connect to other nodes in its own unit of time. As an example, all nodes that have a time component of 78 minutes cannot connect to another node that also has a time component of 78 minutes. This is because no instantaneous transfers are allowed between nodes. The number of nodes in the same time unit is represented with $i_u$ (see Figure 6.1). Each node in the same time unit can be processed in parallel since they cannot share edges and therefore, will not share successor sets.

Second, each node will also contain a transfer buffer value $i_t$. No edge from node $i$ can connect to nodes within $i_t$. This value is adjusted in the system it represents to allow for typical variances. In the case I am modeling, buses must contend with varying traffic patterns and need a buffer of $x$ minutes to reduce the likelihood of missing a valid transfer. There will be no edge between row $i$ and the rows in its transfer buffer $i_t$.

Together each row has a time unit buffer, $i_u$, and a transfer buffer, $i_t$, giving a row buffer of $i_b = i_t + i_u$ (see Figure 6.1). The value of $i_b$ represents the number of rows, before row $i$, with which row $i$ does not share an edge. Thus, all rows within
\(i_b\) can be processed in parallel as they do not share any edges and therefore, will not share any successor sets. The number of processors \(K\) that can be utilized to process rows is \(1 \leq K \leq i_b\), where \(K \ll n\).

![Figure 6.1 K limit, immediate non-connected rows](image)

**Algorithm (Parallel reduced-rolling-queue)**

Set \(K\) to be the \(\min(i_b \in i)\) rows where \(|i_b| - i < 0\) are ignored

\[\text{maxRows} = \max(i_e \in V) + K\]

Store rows 1 to \(\text{maxRows}\) in \(\text{rowCache}\)

For each processor \(K\) Call \(\text{Process}_k(i)\) where \(i\) is the processor number
**Procedure** $Process_k(i)$

Copy row $rowCache(i)$ into Temporary row $i'$

For $j$ from $i - 1$ to $i_n$

If $i'[j] \neq 0$

Call $add\_succ(rowCache(i), rowCache(j), i')$

next = call $getNextRow$

If next $\leq n$

Call $Process_k(next)$

Else

Terminate Process

**Procedure** $add\_succ(rowCache(i), rowCache(j), i')$

For $l$ from $i_n$ to $j - 1$

If $j[l] = 1$

If $i'[l] = 1$

$i'[l] = 0$

Else

$i[l] = 1$

**Procedure** $getNextRow()$

$i = i + 1$

If $i > \text{largest row in rowCache}$

Remove lowest row from $rowCache$

And write it to disk

Add row $i$ to $rowCache$

Return $i$

Once all processes have terminated

Write remaining rows in $rowCache$ to disk
7. Additional TC Algorithms

7.1. Matrix Multiplication

Matrix multiplication can be used in the following manner to solve the TC of a graph. Let $A$ be an adjacency matrix of a DAG and then, convert $A$ into an identity matrix by setting the values along the diagonal to 1. The TC of $A$, $A^+ = A^n$. This can be obtained by multiplying the matrix in the following manner $A^2 = A^1A^1$, $A^4 = A^2A^2$, $A^8 = A^4A^4$, until $A^n$ has been obtained. As a last step, set the diagonal to 0 [3].

With this in mind, many papers have been dedicated to the idea of utilizing matrix multiplication to solve for TC [10, 3, 16]. Two examples are the Macii [16] and Altman [3] papers. Macii utilizes the Strassen matrix multiplication algorithm [20] to compute TC. The Altman paper shows a method to compute TC, in parallel, in $O(\log n)$, where $n$ is the number of nodes in the graph, and achieves it using only $n^3$ processors.

7.2. Graph Based

Matrix manipulation is not the sole mechanism for explaining and creating TC. There are many graph based solutions to solving TC [7, 10, 12, 13]. Many graph based solutions rely on graph traversal, as shown in the aptly titled paper, “Transitive Closure Algorithms Based on Graph Traversal” [12].

7.3. Maintaining TC

Another interesting area of TC research is in maintaining a TC of a graph in a dynamic system [21, 8, 9]. This area of research explores algorithms that maintain the TC of a graph as nodes or edges are added, removed, changed.

8. Performance Analysis

The performance of five different algorithms and their I/O optimized derivates have been tested for this thesis: the algorithms of Floyd-Warshall [5, 24, 11], Warren [23], Agrawal-Jagadish [1], Yang-Yu-Liu-Dao-Wan-Pham [25], and Reduced Hybrid. The code for the algorithms is available in the appendix. The test environment is described in Table 8.1.

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Ubuntu 9.0.4 (64 bit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Motherboard</td>
<td>Gigabyte EP45-DS3L (LGA 775 socket)</td>
</tr>
<tr>
<td>CPU</td>
<td>Intel E1400 2.0 Dual-Core Processor</td>
</tr>
<tr>
<td>Memory</td>
<td>12GB (2 x 2GB + 2 x 4GB) DDR2 800</td>
</tr>
<tr>
<td>Hard Drive</td>
<td>Raid 0 w/ 2xWestern Digital WD10EADS 1TB SATA2</td>
</tr>
</tbody>
</table>

8.1. Results

The first set of results compare the computational time of the five algorithms in their basic form, maintaining the matrix in memory (No disk I/O). Each test was run five times and the average was used for comparison.
The input matrix is generated dynamically and loaded into memory before beginning each algorithm. Two matrices are created. The first, is a topologically sorted DAG required for the Yang-Yu-Liu-Dao-Wan-Pham [25] algorithm. The second, is a reverse topologically sorted DAG with transit type connectivity used by the other four algorithms. The results can be seen in Figure 8.1 which was computed from data in Table 8.2.

The graph, in Figure 8.1, shows that Floyd-Warshall’s algorithm [5, 24, 11] has the slowest performance followed by Yang-Yu-Liu-Dao-Wan-Pham’s algorithm [25]. The remaining are clustered together and need further testing with a greater number of nodes to better distinguish their differences.

![Time (millisec) vs. Node Count Comparison of 5 Algorithms Run In Memory](image.png)

Figure 8.1 Time vs. node count comparison of 5 algorithms (memory)
Table 8.2 Small memory test results (milliseconds)

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Floyd-Warshall</th>
<th>Warren</th>
<th>Agrawal-Jagadish</th>
<th>Reduced Hybrid</th>
<th>YangYuLiuDaoWanPham</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>17</td>
<td>11</td>
<td>1</td>
<td>0.4</td>
<td>21</td>
</tr>
<tr>
<td>100</td>
<td>56</td>
<td>26</td>
<td>10</td>
<td>6</td>
<td>40</td>
</tr>
<tr>
<td>150</td>
<td>57</td>
<td>39</td>
<td>33</td>
<td>17</td>
<td>48</td>
</tr>
<tr>
<td>200</td>
<td>90</td>
<td>50</td>
<td>38</td>
<td>27</td>
<td>114</td>
</tr>
<tr>
<td>250</td>
<td>145</td>
<td>43</td>
<td>42</td>
<td>22</td>
<td>98</td>
</tr>
<tr>
<td>300</td>
<td>245</td>
<td>52</td>
<td>51</td>
<td>35</td>
<td>155</td>
</tr>
<tr>
<td>350</td>
<td>418</td>
<td>65</td>
<td>57</td>
<td>37</td>
<td>194</td>
</tr>
<tr>
<td>400</td>
<td>605</td>
<td>76</td>
<td>60</td>
<td>47</td>
<td>300</td>
</tr>
<tr>
<td>450</td>
<td>1,012</td>
<td>83</td>
<td>63</td>
<td>50</td>
<td>375</td>
</tr>
<tr>
<td>500</td>
<td>1,436</td>
<td>123</td>
<td>57</td>
<td>52</td>
<td>485</td>
</tr>
</tbody>
</table>

The second set of results compare the computational time of the five algorithms using disk I/O. Each test was run five times and the average was used for comparison. The results can be seen in Figure 8.2 and Figure 8.3 which was computed from data in Table 8.3.
The Floyd-Warshall diverges upwards so quickly that the other algorithms are indistinguishable from each other by comparison (see Figure 8.3). This is a result of the heavy I/O the algorithm requires as pointed out in Warren’s paper [23].

![Time (millisec) vs. Node Count](image)

**Time (millisec) vs. Node Count**

*Comparison of 4 Algorithms Run From Disk*

Figure 8.3 Small disk test results Floyd-Warshall removed (milliseconds)

The computational arcs of the other algorithms were better seen when the Floyd-Warshall results were removed (see Figure 8.3). Similar to the memory results, the disk results were still clustered (once the Floyd-Warshall was removed). Testing graphs with larger node counts was needed to better reveal the differences between algorithms.
Table 8.3 Small-disk I/O test results (milliseconds)

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Floyd-Warshall</th>
<th>Warren</th>
<th>Agrawal-Jagadish</th>
<th>Reduced Hybrid</th>
<th>Yang-Yu-Liu-Dao-Wan-Pham</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>736</td>
<td>69</td>
<td>27</td>
<td>31</td>
<td>41</td>
</tr>
<tr>
<td>100</td>
<td>3,856</td>
<td>111</td>
<td>48</td>
<td>70</td>
<td>56</td>
</tr>
<tr>
<td>150</td>
<td>11,874</td>
<td>196</td>
<td>119</td>
<td>141</td>
<td>120</td>
</tr>
<tr>
<td>200</td>
<td>31,876</td>
<td>219</td>
<td>160</td>
<td>197</td>
<td>175</td>
</tr>
<tr>
<td>250</td>
<td>69,292</td>
<td>271</td>
<td>252</td>
<td>244</td>
<td>190</td>
</tr>
<tr>
<td>300</td>
<td>133,896</td>
<td>310</td>
<td>307</td>
<td>291</td>
<td>269</td>
</tr>
<tr>
<td>350</td>
<td>234,708</td>
<td>358</td>
<td>334</td>
<td>312</td>
<td>449</td>
</tr>
<tr>
<td>400</td>
<td>375,789</td>
<td>389</td>
<td>388</td>
<td>341</td>
<td>509</td>
</tr>
<tr>
<td>450</td>
<td>594,052</td>
<td>483</td>
<td>433</td>
<td>378</td>
<td>674</td>
</tr>
<tr>
<td>500</td>
<td>869,729</td>
<td>573</td>
<td>465</td>
<td>419</td>
<td>868</td>
</tr>
</tbody>
</table>

The third set of results compare the computational time of Warren [23], Agrawal-Jagadish [1], Yang-Yu-Liu-Dao-Wan-Pham [25], Reduced Hybrid, blocked version of Agrawal-Jagadish, blocked version of Yang-Yu-Liu-Dao-Wan-Pham, and rolling queue version of the Reduced Hybrid. The Floyd-Warshall was dropped from this test as it was found to have a vast gap in performance compared to the other algorithms. Each test was run five times and the average was used for comparison. The results can be seen in Figure 8.4 and Figure 8.5 which was computed from data in Table 8.4.

The connectivity of the nodes, from the simulated graph, for the third set differs from those of the first and second set. The nodes have a fixed average connectivity of 50, ranging per node, from 0 to 100 edges. This connectivity is representative of a transit graph created from a regional public transportation system.
Utilizing larger graphs, it is clear to see, that the Yang-Yu-Liu-Dao-Wan-Pham, and to a lesser extent, the Warren, have a poorer performance than the other algorithms (see Figure 8.4). The sluggishness of the Yang-Yu-Liu-Dao-Wan-Pham can be attributed to its need to read in the $P(j)$ for every column of every row with which it intersects. The blocked version of the Yang-Yu-Liu-Dao-Wan-Pham has a much better performance as it limits the number of times $P(j)$ is read from disk per row, as seen in Figure 8.4, and in detail, Figure 8.5.
Warren’s algorithm, although outperforming Yang-Yu-Liu-Dao-Wan-Pham’s, does not perform as well as the remaining algorithms tested in this section. However, Warren’s algorithm is less fragile than the others tested, because it does not need a topologically ordered matrix as input.

To better view the differences between the remaining algorithms, remove the basic Yang-Yu-Liu-Dao-Wan-Pham and Warren (see Figure 8.5). The blocked version of the Yang-Yu-Liu-Dao-Wan-Pham still suffers from poor performance, even though reads from $P(j)$ have been reduced, but not eliminated. The basic forms of the algorithms of Agrawal-Jagadish and Reduced Hybrid outperform the Yang-Yu-Liu-Dao-Wan-Pham blocked version.
It is interesting to note that at this tested node range, the basic forms of Agrawal-Jagadish and Reduced Hybrid closely follow each other’s computational time, with the Reduced Hybrid slightly ahead. The same goes for their respective blocked and queued versions. This makes sense as the Reduced Hybrid is tightly based on the Agrawal-Jagadish Algorithm.

Table 8.4 Mid-disk_blocked_queue test results (milliseconds)

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Warren</th>
<th>Agrawal-Jagadish</th>
<th>Agrawal-Jagadish Blocked (20)</th>
<th>Reduced Hybrid</th>
<th>Reduced Hybrid Rolling-Queue</th>
<th>YangYuLiu-DaoWanPham</th>
<th>YangYuLiu-DaoWanPham Blocked (20)</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>537</td>
<td>483</td>
<td>231</td>
<td>453</td>
<td>214</td>
<td>908</td>
<td>263</td>
</tr>
<tr>
<td>1,000</td>
<td>1,553</td>
<td>975</td>
<td>431</td>
<td>872</td>
<td>341</td>
<td>5,235</td>
<td>832</td>
</tr>
<tr>
<td>1,500</td>
<td>3,342</td>
<td>1,656</td>
<td>790</td>
<td>1,426</td>
<td>579</td>
<td>16,447</td>
<td>1,841</td>
</tr>
<tr>
<td>2,000</td>
<td>6,056</td>
<td>2,650</td>
<td>1,036</td>
<td>2,045</td>
<td>783</td>
<td>37,913</td>
<td>3,293</td>
</tr>
<tr>
<td>2,500</td>
<td>15,493</td>
<td>3,434</td>
<td>1,458</td>
<td>2,959</td>
<td>1,095</td>
<td>72,325</td>
<td>5,248</td>
</tr>
<tr>
<td>3,000</td>
<td>25,430</td>
<td>4,867</td>
<td>1,946</td>
<td>3,904</td>
<td>1,353</td>
<td>126,497</td>
<td>7,895</td>
</tr>
<tr>
<td>3,500</td>
<td>38,876</td>
<td>6,411</td>
<td>2,674</td>
<td>5,403</td>
<td>1,607</td>
<td>194,375</td>
<td>10,837</td>
</tr>
<tr>
<td>4,000</td>
<td>54,461</td>
<td>7,923</td>
<td>3,378</td>
<td>6,808</td>
<td>1,888</td>
<td>302,511</td>
<td>14,337</td>
</tr>
<tr>
<td>4,500</td>
<td>81,447</td>
<td>10,358</td>
<td>4,165</td>
<td>8,588</td>
<td>2,322</td>
<td>408,784</td>
<td>18,752</td>
</tr>
<tr>
<td>5,000</td>
<td>103,833</td>
<td>12,863</td>
<td>5,206</td>
<td>11,252</td>
<td>2,736</td>
<td>558,532</td>
<td>23,864</td>
</tr>
</tbody>
</table>

The fourth set of results compare the computational time of the blocked Agrawal-Jagadish [1], rolling queue version of the Reduced Hybrid, 64-bit rolling queue of the Reduced Hybrid, and a 64-bit rolling queue version of the Reduced Hybrid run in parallel. Each test was run five times and the average is used for comparison. The results can be seen in Figure 8.4 and Figure 8.5 which was created from data in Table 8.4.

The input matrices, aside from the number of nodes, have the same characteristics as those used in the third set of results.
The Reduced Hybrid algorithms far outperform Agrawal-Jagadish’s blocked algorithm as seen in Figure 8.6 created from data in Table 8.5. The Reduced Hybrid matrix file is approximately 1/3 the size of the Agrawal-Jagadish matrix file and therefore the I/O and computational time are reduced.
Table 8.5 Large-disk/blocked/queue/64bit/threads test results (milliseconds)

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Agrawal-Jagadish Blocked (20)</th>
<th>Reduced Hybrid Rolling-Queue</th>
<th>Reduced Hybrid Rolling-Queue 64 bit</th>
<th>Reduced Hybrid Rolling-Queue 64 bit with 2 processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>10,000</td>
<td>8,446</td>
<td>898</td>
<td>942</td>
<td>24,398</td>
</tr>
<tr>
<td>20,000</td>
<td>30,235</td>
<td>2,068</td>
<td>1,760</td>
<td>122,591</td>
</tr>
<tr>
<td>30,000</td>
<td>66,672</td>
<td>3,573</td>
<td>3,035</td>
<td>311,081</td>
</tr>
<tr>
<td>40,000</td>
<td>114,119</td>
<td>6,019</td>
<td>4,673</td>
<td>586,980</td>
</tr>
<tr>
<td>50,000</td>
<td>179,695</td>
<td>9,207</td>
<td>6,548</td>
<td>998,646</td>
</tr>
<tr>
<td>60,000</td>
<td>259,106</td>
<td>12,321</td>
<td>8,753</td>
<td>1,501,882</td>
</tr>
<tr>
<td>70,000</td>
<td>333,989</td>
<td>16,105</td>
<td>10,412</td>
<td>2,001,950</td>
</tr>
<tr>
<td>80,000</td>
<td>458,831</td>
<td>18,453</td>
<td>14,007</td>
<td>2,919,352</td>
</tr>
<tr>
<td>90,000</td>
<td>583,194</td>
<td>25,066</td>
<td>17,194</td>
<td>3,795,719</td>
</tr>
<tr>
<td>100,000</td>
<td>723,305</td>
<td>29,376</td>
<td>21,433</td>
<td>4,774,739</td>
</tr>
</tbody>
</table>

The fifth, and final set of results, focus on very large graphs. These graphs are large enough to approximate real transportation systems. Currently, Denver’s local transit authority, RTD, requires a transit graph of 1,839,526 nodes. This graph size is approximately the size of several other transit authorities, such as Dallas, and San Francisco (BART). One extreme outlier is the New York Bus system, which would require a transit graph of over 8,000,000 nodes.
Figure 8.7 Best algorithm very large node count test

Figure 8.7, generated from data in Table 8.6, shows an exponential cubic trend for the time taken to compute the TC of a transit graph. One abnormal data point is the test with 1,839,526 nodes in its graph. This represents the size of the Denver RTD system. I was taken by surprise, and pleased to see, that the TC of the transit adjacency graph was computed as fast as it was on a consumer level machine. I would have been more than content to see 4 to 14 day runtimes to compute the TC, but instead only a few hours were required.

An interesting trend, shown in Table 8.6, is the percentage of time spent in I/O. Simply retrieving and writing rows increases from 48% at 250,000 nodes to approximately 74% at 2,000,000 nodes. Increasing the disk performance with more modern disks, such as SSD disks, would have a big impact on the overall computation time of this algorithm.
Table 8.6 Very large graphs test results (milliseconds)

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Reduced Hybrid Rolling-Queue 64 bit with 2 processes</th>
<th>Percentage of time spent in I/O with disk</th>
</tr>
</thead>
<tbody>
<tr>
<td>250,000</td>
<td>134,782</td>
<td>48%</td>
</tr>
<tr>
<td>500,000</td>
<td>511,309</td>
<td>51%</td>
</tr>
<tr>
<td>750,000</td>
<td>1,156,303</td>
<td>57%</td>
</tr>
<tr>
<td>1,000,000</td>
<td>2,368,806</td>
<td>64%</td>
</tr>
<tr>
<td>1,250,000</td>
<td>4,583,407</td>
<td>69%</td>
</tr>
<tr>
<td>1,500,000</td>
<td>7,142,919</td>
<td>69%</td>
</tr>
<tr>
<td>1,750,000</td>
<td>11,473,473</td>
<td>74%</td>
</tr>
<tr>
<td>1,839,526</td>
<td>13,135,813</td>
<td>72%</td>
</tr>
<tr>
<td>2,000,000</td>
<td>16,398,128</td>
<td>74%</td>
</tr>
</tbody>
</table>

### 8.2. Conclusions

For the specific case of a transit graph, the Reduced Hybrid 64-bit rolling queue run in parallel, outperforms all the other considered algorithms. It is fast enough to compute the TC of real transit systems in a reasonable time (3.65 hours for an RTD transit graph, on a consumer level machine). The other algorithms considered in this paper are inadequate for this task given its parameters, with the exception of the sequential version of the 64-bit Rolling Queue Reduced Hybrid algorithm.

Two of the more promising algorithms not tested were Altman’s matrix multiplication-based approach and the parallel version of Yang-Yu-Liu-Dao-Wan-Pham’s algorithm. The system requirements to implement these are beyond the scope of this thesis. However, given the resources, it would be worthwhile to explore the speed of these parallel algorithm with very large graphs.
package com._10x13.adjacencymatrixwrtool.readwrite;

import com._10x13.adjacencymatrixwrtool.utilties.BitTool;
import com._10x13.adjacencymatrixwrtool.utilties.GraphProperties;
import com._10x13.adjacencymatrixwrtool.utilties.Row;
import java.io.BufferedInputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.RandomAccessFile;
import java.util.Arrays;

/**
 * @author T. Patrick Bailey
 * Date 10/2010
 */
public class AdjacencyGraphWR {

    /** Graph Properties file. **/
    private GraphProperties graphProp;

    /** Buffered input stream. **/
    private RandomAccessFile randAccessFile = null;
    /** Row Byte Lengths, used for stepped and doublestepped. **/
    private int[] rowByteLengths;
    private int maxRowByteLength = 0;
    /** Row Zero Byte Lengths, used for doublestepped. **/
    private int[] zeroByteLengths;
    /** Row Range lengths, used for doublestepped. **/
    private int[] rowRangeLengths;
    private int maxRowRange = 0;
    /** GraphFile number. **/
    private int graphFileNum = 1;
    /** which row is ready to be read. **/
    private int readyToReadRow = 0;
    /** Bytes read in so far. **/
private long bytesReadInSoFar = 0;
/** Determines if I can write to graph File. **/ private boolean canWrite;
/** how many times a bit is read. **/
private long bitReadCount = 0;
/** how many times a bit was written. **/
private long bitWriteCount = 0;
/** How many times a byte is read. **/
private long byteReadCount = 0;
/** How many times a byte is written to. **/
private long byteWriteCount = 0;

public AdjacencyGraphWR(GraphProperties graphProp,
boolean canWrite) throws IOException {
this.graphProp = graphProp;
this.canWrite = canWrite;
if(graphProp.getRowLengthStoredFile().exists()) {
readInRowLengths();
}
if(graphProp.getZeroLengthStoredFile().exists()) {
readInZeroLengths();
}
if(graphProp.getRowRangeLengthStoredFile().exists()) {
readInRowRangeLengths();
}
}

/**
* Reads in rowLengthStored.graph and stores the
* data.
* @throws IOException exception.
*/
private void readInRowLengths() throws IOException {
BufferedInputStream bisRowLength;
int numVertices = graphProp.getNumVertices();
rowByteLengths = new int[numVertices];
byte[] intByte = new byte[4];

bisRowLength = new BufferedInputStream(
    new FileInputStream(graphProp.
        getRowLengthStoredFile()));

for (int x = 0; x < numVertices; x++) {
    bisRowLength.read(intByte);
    rowByteLengths[x] =
        BitTool.byteArrayToInt(intByte);
    if (maxRowByteLength < rowByteLengths[x]) {
        maxRowByteLength = rowByteLengths[x];
    }
}

bisRowLength.close();

/**
 * Reads in rowZeroFillLength.graph.
 * @throws IOException
 */
private void readInZeroLengths()
    throws IOException {
    BufferedInputStream bisZeroLength;
    int numVertices = graphProp.getNumVertices();
    zeroByteLengths = new int[numVertices];
    byte[] intByte = new byte[4];

    bisZeroLength = new BufferedInputStream(new
        FileInputStream(graphProp.
            getZeroLengthStoredFile()));

    for (int x = 0; x < numVertices; x++) {
        bisZeroLength.read(intByte);
        zeroByteLengths[x] =
            BitTool.byteArrayToInt(intByte);
    }

    bisZeroLength.close();
}

/**
 * read in RowRange Lengths.
 * @throws IOException
 */
private void readInRowRangeLengths()
throws IOException {
BufferedInputStream bisRowRangeLength;
int numVertices = graphProp.getNumVertices();
rowRangeLengths = new int[numVertices];
byte[] intByte = new byte[4];

bisRowRangeLength = new BufferedInputStream(new FileInputStream(graphProp.
getRowRangeLengthStoredFile()));

for (int x = 0; x < numVertices; x++) {
bisRowRangeLength.read(intByte);
rowRangeLengths[x] =
    BitTool.byteArrayToInt(intByte);
if (rowRangeLengths[x] > maxRowRange) {
    maxRowRange = rowRangeLengths[x];
}
}
bisRowRangeLength.close();
}

/**
 * Returns the nth Row.
 * @param nthRow the nth row.
 * @return nth row.
 * @throws Exception
 */
public final Row readRow(final int nthRow)
throws Exception {
if (nthRow >= graphProp.getNumVertices()) {
    throw new Exception("There is not a row:'" + nthRow + "' in this graph");
}
return new Row(graphProp.getNumVertices(),
    getZeroFillLength(nthRow),
    getByteData(nthRow));
}

/**
 * Read in rows based on rows.length.
 * @param rows Array of Row to fill.
 * @param from Which row to begin reading from.
 * @throws Exception
 */
public final void readRows(Row[] rows, int from) throws Exception {
    if (rows.length + from > graphProp.getNumVertices()) {
        throw new Exception("There is not enough rows:'" + (rows.length + from) + "' in this graph");
    }

    for(int x = from; x < rows.length + from; x++) {
        rows[x - from] = readRow(x);
    }
}

/**
 * Write nth Row to file with provided row.
 * Row given must have same parameters as row on
disk
 * rowdata size, zero length size.
 * @param nthRow which row to write to.
 * @param row row to read from.
 * @return true if row is written to else false (row
 * parameters do not match).
 * @throws Exception
 */
public final boolean writeRow(final int nthRow, final Row row) throws Exception {
    if (nthRow >= graphProp.getNumVertices()) {
        throw new Exception("There is not a row:'" + nthRow + "' in this graph");
    }

    boolean isSameRow = false;
    int totalLength = (graphProp.getNumVertices() % 8 != 0) ? (1 + (graphProp.getNumVertices() / 8)) : (graphProp.getNumVertices() / 8);

    if(row.getRowDataLength() ==
        getRowDataLength(nthRow)
        && row.getZeroFillLength() ==
        getZeroFillLength(nthRow)
        && row.getTotalLength() == totalLength){
        isSameRow = true;
    }
if(readyToReadRow > nthRow) {
    reset();
}

if(readyToReadRow < nthRow) {
    // Move to be ready to write to row.
    skipTo(nthRow);
    readyToReadRow = nthRow;
}

writeNext(row.getRowData());
readyToReadRow++;
}

return isSameRow;
}

/**
 * Write the rows[] provided starting from.
 * @param rows rows to write to.
 * @param from nth row to start writing from.
 * @throws Exception
 */
public final void writeRows(Row[] rows, int from)
 throws Exception {
    if (rows.length + from >
            graphProp.getNumVertices()) {
        throw new Exception("There is not enough rows:'"
                + (rows.length + from)
                + "' in this graph");
    }

    for(int x = from; x < rows.length + from; x++) {
        writeRow(x, rows[x - from]);
    }
}

/**
 * Read the xthPlace of the nthRow
 * @param xthBit xth place.
 * @param nthRow nth Row.
 * @return true if the xth place of the nth Row is
 * 1.
 * @throws Exception
 */
public final boolean readBitOfRow(final int xthBit, final int nthRow) throws Exception {
    if (xthBit >= graphProp.getNumVertices()) {
        throw new Exception("There is not a Place:'" + xthBit + "' in this graph");
    }
    bitReadCount++;
    return readRow(nthRow).readBit(xthBit);
}

public final byte readByteOfRow(final int xthBit, final int nthRow) throws Exception {
    if (xthBit / 8 >= graphProp.getNumVertices()) {
        throw new Exception("There is not a (byte) Place:'" + xthBit + "' in this graph");
    }
    return readRow(nthRow).readByte(xthBit);
}

public final boolean writeBitOfRow(final int xthBit, final int nthRow, boolean setTo) throws Exception{
    if (xthBit >= graphProp.getNumVertices()) {
        throw new Exception("There is not a Place:'"
            + xthBit + "' in this graph");
    }
    return readRow(nthRow).writeBit(xthBit, setTo);
}
+ xthBit + "' in this graph";
}  
byte returnedByte;
boolean isUpdate = false;

bitWriteCount++;
returnedByte = readRow(nthRow).writeBit(xthBit, setTo);
return writeByteOfRow(returnedByte, xthBit / 8, nthRow);

/**
 * Overwrite the Xth byte of the nth row.
 * @param b Byte to write.
 * @param xthByte Which byte of the row to write to (counted as matrix).
 * @param nthRow Which row to write to.
 * @return true if byte is written to else false.
 * @throws IOException
 */
public final boolean writeByteOfRow(final byte b,
        final int xthByte, final int nthRow)
        throws IOException {
    int zeroFillLength = getZeroFillLength(nthRow);
    int rowDataLength = getRowDataLength(nthRow);
    int rowDataBytesToSkip = xthByte - zeroFillLength;
    byte[] bytes = new byte[1];
    boolean wasWritten = false;

    bytes[0] = b;
    if (readyToReadRow > nthRow) {
        //Resetting to read from beginning
        reset();
    }

    //Move to be ready to write to row.
    skipTo(nthRow);
    readyToReadRow = nthRow;
    if (zeroFillLength <= xthByte && rowDataLength >= rowDataBytesToSkip 
            && (zeroFillLength + rowDataLength) > xthByte) {
        wasWritten = true;
        readyToWriteRow = nthRow;
if ((xthByte - zeroFillLength) > 0) {
    skipTheNextXBytes(xthByte - zeroFillLength);
}
//Now write to the next byte
writeNext(bytes);
wasWritten = true;
//Reset to beginning
reset();
return wasWritten;
}

/**
 * Returns the zero per fill length (used for double stepped).
 * @param nthRow nthRow to read.
 * @return zero fill length.
 */
private int getZeroFillLength(final int nthRow) {
    int zeroFillLength = 0;
    if (zeroByteLengths != null && zeroByteLengths.length > 0) {
        zeroFillLength = zeroByteLengths[nthRow];
    }
    return zeroFillLength;
}

/**
 * Row range length (used for double stepped)
 * @param nthRow
 * @return Row range length.
 */
public int getRowRangeLength(final int nthRow) {
    int rowRangeLength = 0;
    if (graphProp.getStorageType().equals("doublestepped")) {
        rowRangeLength = rowRangeLengths[nthRow];
    }
    return rowRangeLength;
}
/**
 * Maximum row range.
 * @return max row range length.
 */

public int getMaxRowRangeLength() {
    return maxRowRange;
}

/**
 * Return the rowData from the Nth Row.
 * @param nthRow nthRow.
 * @return RowData.
 */
private byte[] getByteData(final int nthRow) throws IOException {
    byte[] rowData = null;
    if (readyToReadRow == nthRow) {
        rowData =
            readNext(getRowDataLength(nthRow));
        readyToReadRow++;
    } else if (readyToReadRow < nthRow) {
        skipTo(nthRow);
        readyToReadRow = nthRow;
        rowData = getByteData(nthRow);
    } else {
        //Resetting to read from beginning
        reset();
        rowData = getByteData(nthRow);
    }
    return rowData;
}

/**
 * RowLength of nth Row.
 * @param nthRow nthRow.
 * @return length of nthRow in bytes.
 */
private int getRowDataLength(final int nthRow) {
    int rowLength;
    if (graphProp.getStorageType().
        equals("matrix")) {
        rowLength = graphProp.getNumVertices() / 8;
        if (graphProp.getNumVertices() % 8 > 0) {
rowLength++;
}
} else {
    rowLength = rowByteLengths[nthRow];
}

return rowLength;

/**
 * @return max Row byte length.
 */
public int getMaxRowByteLength() {
    return maxRowByteLength;
}

/**
 * This assumes a constant number of rows stored
 * @param buffer buffer added to maxRowByteLength
 * @return number of bytes needed to be stored
 */
public long getMinBytesNeededWithBuffer(int buffer) {
    long memLengthRequired = 0;
    long sum = 0;
    long[] sumOfRowLengths =
        new long[rowByteLengths.length];
    int minRowsNeeded =
        getMinRowsNeededWithBuffer(buffer);
    for(int x = 0; x < rowByteLengths.length; x++){
        sum += rowByteLengths[x];
        sumOfRowLengths[x] = sum;
    }
    if(minRowsNeeded > graphProp.getNumVertices()){
        return sumOfRowLengths[
            sumOfRowLengths.length-1];
    } else {
        memLengthRequired =
            sumOfRowLengths[minRowsNeeded-1];
        for(int x = minRowsNeeded; x <
            graphProp.getNumVertices(); x++) {
            if(memLengthRequired <
                sumOfRowLengths[x] -
sumOfRowLengths[x-minRowsNeeded]) {
    memLengthRequired =
    sumOfRowLengths[x] -
    sumOfRowLengths[x-minRowsNeeded];
}
}
return memLengthRequired;

public int getMinRowsNeededWithBuffer(int buffer) {
    return 8*(getMaxRowRangeLength()) + buffer;
}
/**
* Read next x bytes.
* @param byteLength byte length to read in.
* @return bytes read from file(s)
* @throws IOException
* /
private byte[] readNext(final int byteLength)
    throws IOException {
    byte[] rowData = new byte[byteLength];
    long availableSpaceInCurrentFile;
    if (randAccessFile == null) {
        setInputStream();
    }

    //Number of Bytes left in current file
    if (graphProp.getNumberOfGraphFiles() == 1) {
        //if MaxFileSize is 0 that indicates there
        //is only one file
        availableSpaceInCurrentFile =
            Long.MAX_VALUE;
    } else {
        availableSpaceInCurrentFile =
            getCurrentGraphFileName().length() -
            (bytesReadInSoFar
                % graphProp.getMaxFileSize());
    }

    if (availableSpaceInCurrentFile >
        rowData.length) {

More then enough space simply read from current file
randAccessFile.read(rowData, 0, byteLength);
byteReadCount += (long) rowData.length;
bytesReadInSoFar += (long) rowData.length;
} else if (availableSpaceInCurrentFile == 0) {
    if (graphFileNum <
        graphProp.getNumberofGraphFiles()) {
        graphFileNum++;
        setInputStream();
    }
} else if (availableSpaceInCurrentFile <
    rowData.length) {
    //rowData must be read from more than one
    //file
    //Get the rest of the data from the current
    //file
    rowData = Arrays.copyOf(readNext((int)
        availableSpaceInCurrentFile),
        byteLength);

    //second portion of the data
    byte[] rowData2 = readNext(byteLength
        - (int) availableSpaceInCurrentFile);

    for (int x = (int)
        availableSpaceInCurrentFile; x <
        byteLength; x++) {
        rowData[x] = rowData2[
            x - (int) availableSpaceInCurrentFile];
    }
} else {
    //They are equal, an exact fit write then
    //increment to next file.
    randAccessFile.read(rowData, 0, byteLength);
    byteReadCount += (long) rowData.length;
    bytesReadInSoFar += (long) rowData.length;
    if (graphFileNum <
        graphProp.getNumberofGraphFiles()) {
        graphFileNum++;
        setInputStream();
    }
}
return rowData;
}
private final void writeNext(byte[] bytes) throws IOException {
    //byte[] rowData = new byte[byteLength];
    long availableSpaceInCurrentFile;

    if (randAccessFile == null) {
        setInputStream();
    }

    //Number of Bytes left in current file
    if (graphProp.getNumberOfGraphFiles() == 1) {
        //if MaxFileSize is 0 that indicates there
        //is only one file
        availableSpaceInCurrentFile =
            Long.MAX_VALUE;
    } else {
        availableSpaceInCurrentFile =
            getCurrentGraphFileName().length() -
            (bytesReadInSoFar % graphProp.getMaxFileSize());
    }

    if (availableSpaceInCurrentFile >
        bytes.length) {
        //More then enough space simply write the
        //bytes to the current file
        randAccessFile.write(bytes);
        byteWriteCount += bytes.length;
        bytesReadInSoFar += (long) bytes.length;
    } else if (availableSpaceInCurrentFile <
        bytes.length) {
        //rowData must be read from more than one
        //file
        //Get the rest of the date from the current
        //file
        writeNext(Arrays.copyOf(bytes, (int) availableSpaceInCurrentFile));

        //second portion of the data
        writeNext(Arrays.copyOfRange(bytes, (int) availableSpaceInCurrentFile,
                                        bytes.length));
    } else {
They are equal, an exact fit write then
//increment to next file.
randAccessFile.write(bytes);
byteWriteCount += bytes.length;
bytesReadInSoFar += (long) bytes.length;
if (graphFileNum <
    graphProp.getNumberOfGraphFiles()) {
    graphFileNum++;
    setInputStream();
}

/**
 * Creates InputStream based on currentfilename.
 * @throws IOException
 */
private void setInputStream() throws IOException {
    close();
    if (canWrite) {
        randAccessFile = new
            RandomAccessFile(
                getCurrentGraphFileName(), "rw");
    } else {
        randAccessFile = new
            RandomAccessFile(
                getCurrentGraphFileName(), "r");
    }
}

/**
 * Close buffered output stream.
 * @throws IOException
 */
public final void close() throws IOException {
    if (randAccessFile != null) {
        randAccessFile.close();
        randAccessFile = null;
    }
}

/**
 * Returns the current Graph file name to write to.
 * @return current graph file name.
 */
private File getCurrentGraphFileName() {
    String graphFileName = graphFileNum + "";
    while (graphFileName.length() < 8) {
        graphFileName = "0" + graphFileName;
    }
    return new File(graphProp.getDirectory(),
            graphFileName + ".graph”);
}

/**
 * Skip to nth row.
 * @param nthRow nth Row.
 * @throws IOException
 */
private void skipTo(final int nthRow) throws IOException {
    long skipLength = 0;
    if (randAccessFile == null) {
        setInputStream();
    }
    for (int x = readyToReadRow; x < nthRow; x++) {
        skipLength += getRowDataLength(x);
    }
    if (graphProp.getNumberOfGraphFiles() > 1) {
        //Handle multiple files here
        skipBytesMultiFile(skipLength);
    } else {
        randAccessFile.seek(randAccessFile.
            getFilePointer() + skipLength);
    }
}

/**
 * Skip the next X Bytes.
 * @param byteSkipLength number of bytes to skip.
 * @throws IOException
 */
private void skipTheNextXBytes(
    final int byteSkipLength) throws IOException {
    long skipLength = byteSkipLength;

if (randAccessFile == null) {
    setInputStream();
}

if (graphProp.getNumberOfGraphFiles() > 1) {
    // Handle multiple files here
    skipBytesMultiFile(skipLength);
} else {
    randAccessFile.seek(
        randAccessFile.getFilePointer() + skipLength);
}

/**
 * Used to skip around when reading from a graph
 * that is contained
 * in multiple files.
 *
 * @param bytesToSkip number of bytes to skip.
 * @throws IOException
 */
private final void skipBytesMultiFile(
    long bytesToSkip) throws IOException {
    long availableSpaceInCurrentFile =
        getCurrentGraphFileName().length() -
        (bytesReadInSoFar % graphProp.getMaxFileSize());

    if (bytesToSkip < availableSpaceInCurrentFile) {
        randAccessFile.seek(
            randAccessFile.getFilePointer() + bytesToSkip);
        bytesReadInSoFar += bytesToSkip;
    } else if(availableSpaceInCurrentFile == 0) {
        graphFileNum++;
        setInputStream();
    } else if (bytesToSkip >
        availableSpaceInCurrentFile) {
        // Skip current file
        skipBytesMultiFile(
            availableSpaceInCurrentFile);

        // Skip the rest
        skipBytesMultiFile(bytesToSkip
            + skipLength);
    }
}

63
- availableSpaceInCurrentFile);
} else {
    // They are equal, an exact fit write then
    // increment to next file.
    graphFileNum++;
    setInputStream();
    bytesReadInSoFar += bytesToSkip;
}

/**
 * How many times a bit is read.
 * @return how many times a bit is read.
 */
public long getBitReadCount() {
    return bitReadCount;
}

/**
 * How many times a bit is written to.
 * @return how many times a bit is written to.
 */
public long getBitWriteCount() {
    return bitWriteCount;
}

/**
 * Bytes read.
 * @return bytes read.
 */
public long getByteReadCount() {
    return byteReadCount;
}

/**
 * Bytes written to.
 * @return number of bytes written to.
 */
public long getByteWriteCount() {
    return byteWriteCount;
}

/**
 * resets to read from first file first byte.
 * @throws IOException
 */
/**
 * private void reset() throws IOException {
 *  close();
 *  readyToReadRow = 0;
 *  graphFileNum = 1;
 *  bytesReadInSoFar = 0;
 * }
 */

/**
 * Returns true if file is set to be written to.
 * @return true if set to be written to.
 */
public boolean isWritable() {
    return canWrite;
}
}
package com._10x13.adjacencymatrixwrtool.readwrite;

import com._10x13.adjacencymatrixwrtool.utilities.GraphProperties;
import com._10x13.adjacencymatrixwrtool.utilities.Row;
import java.io.IOException;

/**
 * @author T. Patrick Bailey
 * Date 10/2010
 */
public class FullyBufferedAdjacencyGraphWR {

/**
 * GraphWR tool. **/
 private AdjacencyGraphWR graphWR;
 /** Graph Properties file. **/
 private GraphProperties graphProp;
 /** Rows. **/
 private Row[] rows;

 /** How many times a row is read. **/
 private long rowReadCount = 0;
 /** How many times a row has been written to. **/
 private long rowWriteCount = 0;
 /** Byte read count. **/
 private long byteReadCount = 0;
 /** Byte write count. **/
 private long byteWriteCount = 0;
 /** Bit Read Count. **/
 private long bitReadCount = 0;
 /** Bit write count. **/
 private long bitWriteCount = 0;

 public FullyBufferedAdjacencyGraphWR(
 GraphProperties graphProp, boolean canWrite)
 throws IOException, Exception {
 graphWR = new AdjacencyGraphWR(
 graphProp, canWrite);
 this.graphProp = graphProp;

 rows = new Row[graphProp.getNumVertices()];
 graphWR.readRows(rows, 0);
 }
/**
* Read nth Row.
* @param nthRow row to read.
* @return nth row.
* @throws Exception
*/
public Row readRow(final int nthRow)
        throws Exception {
    checkRow(nthRow);
    rowReadCount++;
    return rows[nthRow];
}

/**
* Write the row into the rows matrix.
* @param row row to overwrite old row.
* @param nthRow nth row.
* @throws Exception
*/
public boolean writeRow(final Row row,
                        final int nthRow) throws Exception {
    checkRow(nthRow);
    boolean isSameRow = false;
    int totalLength = (graphProp.getNumVertices() % 8 != 0) ?
        (1 + (graphProp.getNumVertices() / 8)) :
        (graphProp.getNumVertices() / 8);

    //New row must fit within old row.
    if(row.getRowDataLength() ==
        rows[nthRow].getRowDataLength() 
            && row.getZeroFillLength() ==
        rows[nthRow].getZeroFillLength() 
            && row.getTotalLength() ==
        totalLength) {
        isSameRow = true;
        rows[nthRow] = row;
        rowWriteCount++;
    }

    return isSameRow;
}
public byte readByteOfRow(final int xthByte, final int nthRow) throws Exception {
    checkRow(nthRow);
    checkByte(xthByte);
    byteReadCount++;
    return rows[nthRow].readByte(xthByte);
}

/**
 * Set the xthByte of the nthRow.
 * @param b byte to set to.
 * @param xthByte xth byte.
 * @param nthRow nth row.
 * @return true if set else false (out of range).
 * @throws Exception
 */
public boolean writeByteOfRow(final byte b, final int xthByte, final int nthRow)
    throws Exception {
    checkRow(nthRow);
    checkByte(xthByte);
    boolean wasWritten = false;
    if (rows[nthRow].getZeroFillLength() <= xthByte &&
        (rows[nthRow].getZeroFillLength() +
            rows[nthRow].getRowDataLength()) > xthByte) {
        //Now write to the next byte
        byteWriteCount++;
        wasWritten = true;
        rows[nthRow].writeByte(xthByte, b);
    }
    return wasWritten;
}
public boolean readBitOfRow(final int xthPlace, final int nthRow) throws Exception {
    checkRow(nthRow);
    checkRow(xthPlace);
    bitReadCount++;
    return rows[nthRow].readBit(xthPlace);
}

/**
 * Writes bit to xthPlace of Nth row, returns true if bit was set else false;
 * @param xthPlace xthPlace.
 * @param nthRow nthRow.
 * @param setTo Boolean to set it to.
 * @return true if bit was set else false.
 * @throws Exception
 */
public boolean writeBitOfRow(final int xthPlace, final int nthRow, boolean setTo) throws Exception {
    checkRow(nthRow);
    checkRow(xthPlace);
    boolean wasWritten = rows[nthRow].writeBitWasUpdated(xthPlace, setTo);
    if (wasWritten) {
        bitWriteCount++;
    }
    return wasWritten;
}

/**
 * Chevk if there is an nthRow.
 * @param nthRow nth row to check.
 * @throws Exception
 */
private void checkRow(final int nthRow) throws Exception {
    if (nthRow > rows.length) {
        
7949
44
52
throw new Exception("There is not a row:'" + nthRow + "' in this graph");} 

/**
 * Check to make sure there are enough bytes in the given row.
 * @param xthByte byte to test validity.
 * @throws Exception
 */
private void checkByte(final int xthByte)
        throws Exception {
    if(xthByte > rows.length) {
        throw new Exception("There is not a byte:'" + xthByte + "' in any row");
    }
}

/**
 * Returns number of rows read count.
 * @return number of times all rows has been read.
 */
public final long getRowReadCount() {
    return rowReadCount;
}

/**
 * Returns how many times an entire row has been written to.
 * @return how many times an entire row has been written to.
 */
public final long getRowWriteCount() {
    return rowWriteCount;
}

/**
 * How many bytes read from disk.
 * @return how many bytes read from disk.
 */
public final long getDiskByteReadCount() {
    return graphWR.getByteReadCount();
}
public final long getDiskByteWriteCount() {
    return graphWR.getByteWriteCount();
}

public final long getByteReadCount() {
    return byteReadCount;
}

public final long getByteWriteCount() {
    return byteWriteCount;
}

public final long getBitReadCount() {
    return bitReadCount;
}

public final long getBitWriteCount() {
    return bitWriteCount;
}

/**
 * Returns true if file is set to be written to.
 * @return true if set to be written to.
 */
public boolean isWritable() {
    return graphWR.isWritable();
}

/**
 * Closes RandomFileAccess.
 * @throws IOException
 */
public final void close() throws IOException {
    graphWR.close();
}

/**
 * Writes all rows to disk.
 * @throws Exception
 */
public void writeAll() throws Exception {
    graphWR.writeRows(rows, 0);
}

/**
 * Returns the rowRangeLength.
 * @param nthRow nthRow.
 * @return row range.
 */
public int getRowRangeLength(final int nthRow) {
    return this.graphWR.getRowRangeLength(nthRow);
}
package com._10x13.adjacencymatrixwrtool.utilities;

/**
 * @author T. Patrick Bailey
 * Date 10/2010
 */
public class BitTool {

/** Hidden Constructor. **/
 private BitTool() { }

/**
 * Sets the trailing bits to 0.
 * @param b Byte to adjust
 * @param bit num bits.
 * @return update byte.
 */
 public static byte setLeadingBitsToZero(
 final byte b, final int bit) {
 byte newB = b;
 if (bit == 1) {
 newB = (byte) (b & 0x7F);
 } else if (bit == 2) {
 newB = (byte) (b & 0x3F);
 } else if (bit == 3) {
 newB = (byte) (b & 0x1F);
 } else if (bit == 4) {
 newB = (byte) (b & 0x0F);
 } else if (bit == 5) {
 newB = (byte) (b & 0x07);
 } else if (bit == 6) {
 newB = (byte) (b & 0x03);
 } else if (bit == 7) {
 newB = (byte) (b & 0x01);
 }
 return newB;
 }

/**
 * Sets the leading bits to 0.
 * @param b Byte to adjust
 * @param bit num bits.
 */
public static byte setTrailingBitsToZero(final byte b, final int bit) {
    byte newB = b;
    if (bit == 1) {
        newB = (byte) (b & 0x80);
    } else if (bit == 2) {
        newB = (byte) (b & 0xC0);
    } else if (bit == 3) {
        newB = (byte) (b & 0xE0);
    } else if (bit == 4) {
        newB = (byte) (b & 0xF0);
    } else if (bit == 5) {
        newB = (byte) (b & 0xF8);
    } else if (bit == 6) {
        newB = (byte) (b & 0xFC);
    } else if (bit == 7) {
        newB = (byte) (b & 0xFE);
    }
    return newB;
}

/**
 * Converts an int to a byte array.
 * @param value int to convert.
 * @return converted byte array.
 */
public static byte[] intToByteArray(final int value) {
    return new byte[] {
        (byte) (value >>> 24),
        (byte) (value >>> 16),
        (byte) (value >>> 8),
        (byte) value};
}

/**
 * Converts a byte array to an int.
 * @param b byte array to convert.
 * @return converted int value.
 */
public static int byteArrayToInt(final byte[] b) {
    return (b[0] << 24) + ((b[1] & 0xFF) << 16) + ((b[2] & 0xFF) << 8) + (b[3] & 0xFF);
}

public static byte[] longToByteArray(final long value) {
    return new byte[] {
        (byte) (value >>> 56),
        (byte) (value >>> 48),
        (byte) (value >>> 40),
        (byte) (value >>> 32),
        (byte) (value >>> 24),
        (byte) (value >>> 16),
        (byte) (value >>> 8),
        (byte) value};
}

/**
 * Converts a byte array to an long.
 * @param b byte array to convert.
 * @return converted int value.
 */
public static long byteArrayToLong(final byte[] b) {
}

/**
 * If the bit is one at the xthPlace
 * @param b
 * @return true if the bit is one at the xthPlace.
 */
public static boolean isBitOne(final byte b, final int bit) {
return (((b << bit) & 0x80) == 0x80);
}

/**
 * If the bit is one at the xthPlace
 * @param b
 * @return true if the bit is one at the xthPlace.
 */
public static boolean isBitOne(final long l, final int bit) {
    return (((l << bit) & 0x8000000000000000l) == 0x8000000000000000l);
}

/**
 * Set the bit to 0 if setBitTo is false otherwise set to 1.
 * And returns updated byte
 * @param b byte to set.
 * @param bit bit number to update.
 * @param setBitTo Byte.
 * @return updated byte.
 */
public static byte setBit(final byte b, final int bit, boolean setBitTo) {
    byte newByte = b;
    byte mask = (byte) (0x01 << (7 - bit));
    byte maskOpposite = (byte) (mask ^ 0xFF);
    if(setBitTo) {
        newByte = (byte) (b | mask);
    } else {
        newByte = (byte) (b & maskOpposite);
    }
    return newByte;
}
package com._10x13.adjacencymatrixwrtool.utiltiies;
import java.util.Arrays;
/**
 * @author T. Patrick Bailey
 * Date 10/2010
 */
public class Row {

    /** Row byte Data.
     * This is the data that may not be zero filled
     * It may not be the entire length of the row.
     **/  
    private byte[] rowData;
    /** Number of edges in this row. **/
    private int numEdges=-1;
    /** The number of bytes to fill with 0s preceeding
     * the rowData.**/
    private int zeroFillLength;
    /** The total length of the row in bytes.
     * It will at most be the same as
     * rowData.length+zeroFillLength
     * However there can be trailing 00s after that
     * it will be exactly roundup(NumNodes/8).
     **/
    private int totalLength;
    /** Number of nodes. **/
    private int numNodes;

    /**
     * Row Constructor.
     * @param numNodes Number of Nodes in graph.
     * @param zeroFillLength Zero fill length.
     * @param rowData Actual rowData.
     */
    public Row(final int numNodes, final int zeroFillLength, final byte[] rowData) throws Exception {
        this.totalLength = (numNodes % 8 != 0) ? (1 + (numNodes / 8)) : (numNodes / 8);
        this.numNodes = numNodes;
        this.zeroFillLength = zeroFillLength;
        this.rowData = rowData.clone();
    }
}
if (this.rowData.length + this.zeroFillLength >
    totalLength) {
    throw new Exception("RowData cannot be" + " longer than the totalLength");
}

public void resetValues(final int numNodes,
    final int zeroFillLength,
    final byte[] rowData) throws Exception {
    this.totalLength = (numNodes % 8 != 0) ? (1 + (numNodes / 8)) : (numNodes / 8);
    this.numNodes = numNodes;
    this.zeroFillLength = zeroFillLength;
    this.rowData = rowData;

    if (this.rowData.length + this.zeroFillLength >
         totalLength) {
        throw new Exception("RowData cannot be" + " longer than the totalLength");
    }
}

/**
 * Returns true if the Xth place is 1.
 * <pre>
 * If it is outside the rowData it will return 0
 * </pre>
 * @return true if the Xth place is 1.
 */
public boolean readBit(int xthBit) {
    boolean isOne = false;
    int byteNumber = xthBit / 8;
    int bitOfByte = xthBit % 8;

    if (byteNumber < zeroFillLength + rowData.length
        && byteNumber >= zeroFillLength) {
        //byte is within the rowData
        isOne = BitTool.isBitOne(rowData[byteNumber - zeroFillLength], bitOfByte);
    }
    return isOne;
}
/**
 * Updates the xthPlace of this row to the setBitTo.
 * Will only update it if its within the rowData
 * area
 * @param xthBit xth bit in row.
 * @param setBitTo if true set it to 1 else 0.
 * @return byte that was updated, if not updated
 * will return 0x00.
 */
public final byte writeBit(final int xthBit, final boolean setBitTo) {
    int byteNumber = xthBit / 8;
    int bitOfByte = xthBit % 8;
    byte byteData = 0;

    if (byteNumber < zeroFillLength + rowData.length
        && byteNumber >= zeroFillLength) {
        //byte is within the rowData
        rowData[byteNumber - zeroFillLength] =
            BitTool.setBit(rowData[byteNumber - zeroFillLength], bitOfByte, setBitTo);
        byteData = rowData[byteNumber - zeroFillLength];
    }
    return byteData;
}

/**
 * Updates the xthPlace of this row to the setBitTo.
 * Will only update it if its within the rowData
 * area
 * @param xthBit xth bit in row.
 * @param setBitTo if true set it to 1 else 0.
 * @return true if bit was updated else false;
 */
public final boolean writeBitWasUpdated(final int xthBit, final boolean setBitTo) {
    int byteNumber = xthBit / 8;
    int bitOfByte = xthBit % 8;
    boolean wasUpdated = false;
if (byteNumber < zeroFillLength + rowData.length && byteNumber >= zeroFillLength) {
    //byte is within the rowData
    rowData[byteNumber - zeroFillLength] = 
    BitTool.setBit(rowData[byteNumber - zeroFillLength], bitOfByte, setBitTo);
    wasUpdated = true;
}
return wasUpdated;
}

/**
 * Returns the xth byte of this row.
 * If it is not within row data it will be 0.
 * @param xthByte xth byte to get.
 * @return xth byte returned.
 */
public byte readByte(int xthByte) {
    if (xthByte < zeroFillLength + rowData.length && xthByte >= zeroFillLength) {
        //byte is within the rowData
        return rowData[xthByte - zeroFillLength];
    }
    return 0;
}

/**
 * Returns the xth byte of this row.
 * If it is not within row data it will be 0.
 * @param xthByte xth byte to get.
 * @return xth byte returned.
 */
public byte readByteNoCheck(int xthByte) {
    return rowData[xthByte - zeroFillLength];
}

public boolean writeByte(int xthByte, byte b) {
    boolean isWithin = false;
    if (xthByte < zeroFillLength + rowData.length && xthByte >= zeroFillLength) {
        //byte is within the rowData
        rowData[xthByte - zeroFillLength] = b;
    }
isWithin = true;
}
return isWithin;
}

/**
 * Total Length.
 * @return total row length.
 */
public final int getTotalLength() {
    return this.totalLength;
}

/**
 * byte size of zero fill length.
 * @return zero fill length.
 */
public final int getZeroFillLength() {
    return this.zeroFillLength;
}

/**
 * Row data length.
 * @return row data length.
 */
public final int getRowDataLength() {
    return this.rowData.length;
}

/**
 * Returns the row as a byte array.
 * @return the row as a byte array.
 */
public final byte[] getRow() {
    byte[] rowDataWithZeroFill = new byte[zeroFillLength + rowData.length];
    if (rowData.length == totalLength) {
        rowDataWithZeroFill = Arrays.copyOf(rowData, totalLength);
    } else {
        for (int x = zeroFillLength; x <
                zeroFillLength + rowData.length; x++) {
            rowDataWithZeroFill[x] = rowData[
                    x - zeroFillLength];
        }
    }
    return rowDataWithZeroFill;
}
rowDataWithZeroFill = Arrays.copyOf(
    rowDataWithZeroFill, totalLength);
}

return rowDataWithZeroFill;
}

/**
 * Returns the validRowData.
 * @return the validRowData.
 */
public final byte[] getRowData() {
    return rowData.clone();
}

/**
 * Returns the validRowData with leading zero bytes.
 * @return the validRowData with leading zero bytes.
 */
public final byte[] getRowDataWithLeadingZeros() {
    byte[] rowDataWithZeroFill = new byte[
        zeroFillLength + rowData.length];

    if (rowData.length == totalLength) {
        rowDataWithZeroFill = Arrays.copyOf(
            rowData, totalLength);
    } else {
        for (int x = zeroFillLength; x <
            zeroFillLength + rowData.length; x++) {
            rowDataWithZeroFill[x] = rowData[
                x - zeroFillLength];
        }
    }

    return rowDataWithZeroFill;
}

/**
 * Returns the entire row as a string in binary
 * format.
 * @return the entire row as a string in binary
 * format.
 */
@Override
public final String toString() {
    final StringBuffer strB = new StringBuffer(100);
    final StringBuffer strB = new StringBuffer(100);
final byte[] allRow = getRow();

for (int x = 0; x < allRow.length; x++) {
    strB.append(zeroPadLeft(Integer.toString(allRow[x] & 0xff, 2)) + ":");
}

if (strB.length() > 0) {
    strB.deleteCharAt(strB.length() - 1);
}
return strB.toString();

/**
 * Returns the entire row as a string in binary format.
 * @return the entire row as a string in binary format.
 */
public final String getRowBinaryAsString() {
    return this.toString();
}

/**
 * Returns the entire row as a string in hex format.
 * @return the entire row as a string in hex format.
 */
public final String getRowHexAsString() {
    final StringBuffer strB = new StringBuffer(500);
    final byte[] allRow = getRow();

    for (int x = 0; x < allRow.length; x++) {
        strB.append(zeroPadHexLeft(Integer.toString(allRow[x] & 0xff, 16)) + ":");
    }

    if (strB.length() > 0) {
        strB.deleteCharAt(strB.length() - 1);
    }
    return strB.toString();
}

/**
 * Returns the rowdata as a string in binary format.
public final String getRowDataBinaryAsString() {
    final StringBuffer strB = new StringBuffer(500);

    for (int x = 0; x < this.rowData.length; x++) {
        strB.append(zeroPadLeft(Integer.toString(this.rowData[x] & 0xff, 2)) + ":");
    }

    if (strB.length() > 0) {
        strB.deleteCharAt(strB.length() - 1);
    }
    return strB.toString();
}

/**
 * Returns the rowdata as a string in binary format.
 * @return the rowdata as a string in binary format.
 */
public final String getRowDataHexAsString() {
    final StringBuffer strB = new StringBuffer(500);

    for (int x = 0; x < this.rowData.length; x++) {
        strB.append(zeroPadHexLeft(Integer.toString(this.rowData[x] & 0xff, 16)) + ":");
    }

    if (strB.length() > 0) {
        strB.deleteCharAt(strB.length() - 1);
    }
    return strB.toString();
}

/**
 * Pad binary data with '0's.
 * @param str padded binary string.
 * @return padded binary string.
 */
public static String zeroPadLeft(String str) {
    while (str.length() < 8) {
        str = "0" + str;
    }
}
public static String zeroPadHexLeft(String str) {
    while (str.length() < 2) {
        str = "0" + str;
    }
    return str;
}

public final Row clone() {
    Row row = null;
    try{
        row = new Row(this.numNodes,
            this.zeroFillLength, this.rowData);
    } catch (Exception e) {
        e.printStackTrace();
    }
    return row;
}

public final int getNumNodes() {
    return this.numNodes;
}
public class FloydWarshallAlgorithm {

    private StringBuffer fullDataStrB =
            new StringBuffer(1000);
    private RandomAccessFile randAccessFile =
            new RandomAccessFile(new File("graphOutput.txt"), "rw");
    private GraphProperties gProp;
    private NumberFormat formatter = new DecimalFormat("###,###,###,###,###,###,###,###");

    /**
     * Runs time test on the Floyd-Warshall Algorithm
     * @param graphFile
     * @throws IOException
     * @throws Exception
     */
    public FloydWarshallAlgorithm(File graphFile) throws IOException, Exception {
        gProp = new GraphProperties(new File(graphFile.getParentFile(),
                "graph.properties");
        long memoryTime = getTimeToRunInMemory();
    }
}

long diskTime = getTimeToRunInDisk();
String firstLine;

firstLine = "Floyd-Warshall: 
    + formatter.format(gProp.getNumVertices()) 
    + ": "+ formatter.format(memoryTime) 
    + ": "+ formatter.format(diskTime);
System.out.println(firstLine);
fullDataStrB.insert(0, firstLine);
fullDataStrB.insert(0, "
--FloydWarshallAlgorithm--");

randAccessFile.seek(randAccessFile.length());
rndAccessFile.write( 
    fullDataStrB.toString().getBytes());
rndAccessFile.close();

/**
 * Returns the runtime in nanoseconds of the Floyd
 * Warshall Algorithm in memory.
 * @return the runtime in nanoseconds of the Floyd
 * Warshall Algorithm in memory.
 * @throws IOException
 * @throws Exception
 */
public long getTimeToRunInMemory() 
    throws IOException, Exception {
    FullyBufferedAdjacencyGraphWR buffGraphWR =
        new FullyBufferedAdjacencyGraphWR( 
            gProp, true);
    int numVert = gProp.getNumVertices();
    long totalTime;;

totalTime = System.nanoTime();
for(int i = 0; i < numVert; i++) {
    for(int j = 0; j < numVert; j++) {
        if (buffGraphWR.readBitOfRow(j, i)) {
            for (int k = 0; k < numVert; k++) {
                if (buffGraphWR.readBitOfRow( 
                    i, k)) {
                    buffGraphWR.writeBitOfRow( 
                        j, k, true);
                }
            }
        }
    }
}
/**
 * Returns the runtime in nanoseconds of the Floyd-Warshall Algorithm from Disk.
 * @return the runtime in nanoseconds of the Floyd-Warshall Algorithm from Disk
 * @throws Exception
 */
public long getTimeToRunInDisk () throws Exception {
    AdjacencyGraphWR graphWR =
        new AdjacencyGraphWR(gProp, true);
    int numVert = gProp.getNumVertices();
    long totalTime = System.nanoTime();

    totalTime = System.nanoTime() - totalTime;
    fullDataStrB.append("\nMemory\nBits Read: "
        + formatter.format(
            buffGraphWR.getBitReadCount())
        + ",\nBits write: "
        + formatter.format(buffGraphWR.
            getBitWriteCount()));

    buffGraphWR.close();
    return totalTime;
}
fullDataStrB.append("\nDisk\nBits Read: "+formatter.format(graphWR.getBitReadCount())
+ "\nBits write: "
+ formatter.format(graphWR.getBitWriteCount()));

graphWR.close();
return totalTime;
}

public static void main(final String...args) {
    File graphFile = new File(args[0]);
    try {
        new FloydWarshallAlgorithm(graphFile);
    } catch(Exception e) {
        e.printStackTrace();
    }
}
package com._10x13.warrenalgorithm;

import com._10x13.adjacencymatrixwrtool.readwrite.AdjacencyGraphWR;
import com._10x13.adjacencymatrixwrtool.readwrite.FullyBufferedAdjacencyGraphWR;
import com._10x13.adjacencymatrixwrtool.utiltiies.GraphProperties;
import com._10x13.adjacencymatrixwrtool.utiltiies.Row;
import java.io.File;
import java.io.IOException;
import java.io.RandomAccessFile;
import java.text.DecimalFormat;
import java.text.NumberFormat;

/**
 * @author T. Patrick Bailey
 * Date: 10/2010
 */
public class WarrenAlgorithm {
    private StringBuffer fullDataStrB =
        new StringBuffer(1000);
    private RandomAccessFile randAccessFile = new
        RandomAccessFile(
            new File("graphOutput.txt"), "rw");
    private GraphProperties gProp;
    private NumberFormat formatter = new
        DecimalFormat("###,###,###,###,###,###,###,###");

    /**
     * Runs time test on the Warren Algorithm
     * @param graphFile
     * @throws IOException
     * @throws Exception
     */
    public WarrenAlgorithm(File graphFile)
        throws IOException, Exception {
        gProp = new GraphProperties(new
            File(graphFile.getParentFile(),
                "graph.properties"));
        long memoryTime = getTimeToRunInMemory();
    }
long diskTime = getTimeToRunInDisk();
String firstLine;

firstLine = "Warren: "
  + formatter.format(gProp.getNumVertices())
  + ": " + formatter.format(memoryTime)
  + ": " + formatter.format(diskTime);
System.out.println(firstLine);
fullDataStrB.insert(0, firstLine);
fullDataStrB.insert(0, "n--WarrenAlgorithm--");

randAccessFile.seek(randAccessFile.length());
randAccessFile.write(fullDataStrB.
  toString().getBytes());
randAccessFile.close();

/**
 * Returns the runtime in nanoseconds of the
 * Warren Algorithm in memory.
 * @return the runtime in nanoseconds of the
 * Warren Algorithm in memory.
 * @throws IOException
 * @throws Exception
 */
public long getTimeToRunInMemory()
  throws IOException, Exception {
  FullyBufferedAdjacencyGraphWR buffGraphWR = new
    FullyBufferedAdjacencyGraphWR(gProp, true);
  int numVert = gProp.getNumVertices();
  long totalTime;
  Row rowI;
  Row rowJ;

  totalTime = System.nanoTime();
  for (int i = 1; i < numVert; i++) {
    rowI = buffGraphWR.readRow(i);
    for (int j = 0; j < numVert; j++) {
      if (rowI.readBit(j)) {
        rowJ = buffGraphWR.readRow(j);
        for (int k = 0; k < numVert; k++) {
          rowI.writeBit(k,
            (rowI.readBit(k) |
              rowJ.readBit(k)));}
          } } } }
for (int i = 0; i < numVert - 1; i++) {
    rowI = buffGraphWR.readRow(i);
    for (int j = i + 1; j < numVert; j++) {
        if (rowI.readBit(j)) {
            rowJ = buffGraphWR.readRow(j);
            for (int k = 0; k < numVert; k++) {
                rowI.writeBit(k, (rowI.readBit(k) | rowJ.readBit(k)));
            }
        }
    }
}

totalTime = System.nanoTime() - totalTime;

fullDataStrB.append("\nMemory\nBits Read: " + formatter.format(buffGraphWR.getBitReadCount()) + "\nBits write: " + formatter.format(buffGraphWR.getBitWriteCount()));
buffGraphWR.close();

return totalTime;

/**
 * Returns the runtime in nanoseconds of the Warren Algorithm from Disk.
 * @return the runtime in nanoseconds of the Warren Algorithm from Disk
 * @throws Exception
 */
public long getTimeToRunInDisk() throws Exception {
    AdjacencyGraphWR graphWR =
        new AdjacencyGraphWR(gProp, true);
    int numVert = gProp.getNumVertices();
    long totalTime;
totalTime = System.nanoTime();
for (int i = 1; i < numVert; i++) {
    rowI = graphWR.readRow(i);
    for (int j = 0; j < numVert; j++) {
        if (rowI.readBit(j)) {
            rowJ = graphWR.readRow(j);
            for (int k = 0; k < numVert; k++) {
                rowI.writeBit(k,
                    (rowI.readBit(k) | rowJ.readBit(k)));
            }
        }
    }
    graphWR.writeRow(i, rowI);
    //write row back to disk
}

for (int i = 0; i < numVert-1; i++) {
    rowI = graphWR.readRow(i);
    for (int j = i+1; j < numVert; j++) {
        if (rowI.readBit(j)) {
            rowJ = graphWR.readRow(j);
            for (int k = 0; k < numVert; k++) {
                rowI.writeBit(k,
                    (rowI.readBit(k) | rowJ.readBit(k)));
            }
        }
    }
    graphWR.writeRow(i, rowI);
    //write row back to disk
}
totalTime = System.nanoTime() - totalTime;

fullDataStrB.append("\nDisk\nBits Read: "+formatter.format(graphWR.getBitReadCount())+
"\nBits write: "+formatter.format(graphWR.getBitWriteCount()));
graphWR.close();
return totalTime;
public static void main(final String...args) {
    File graphFile = new File(args[0]);

    try {
        new WarrenAlgorithm(graphFile);
    } catch(Exception e) {
        e.printStackTrace();
    }
}
package com._10x13.agrawaljagadishhybridalgorithm;

import com._10x13.adjacencymatrixwrtool.readwrite.AdjacencyGraphWR;
import com._10x13.adjacencymatrixwrtool.readwrite.FullyBufferedAdjacencyGraphWR;
import com._10x13.adjacencymatrixwrtool.utilities.GraphProperties;
import com._10x13.adjacencymatrixwrtool.utilities.Row;
import java.io.File;
import java.io.IOException;
import java.io.RandomAccessFile;
import java.text.DecimalFormat;
import java.text.NumberFormat;

/**
 * @author T. Patrick Bailey
 * Date: 10/2010
 */
public class AgrawalJagadishHybridAlgorithm {
    private StringBuffer fullDataStrB = new StringBuffer(1000);
    private RandomAccessFile randAccessFile = new RandomAccessFile(new File("graphOutput.txt"), "rw");
    private GraphProperties gProp;
    private NumberFormat formatter = new DecimalFormat("###,###,###,###,###,###,###,###");

    /**
     * Runs time test on the Warren Algorithm
     * @param graphFile
     * @throws IOException
     * @throws Exception
     */
    public AgrawalJagadishHybridAlgorithm(File graphFile) throws IOException, Exception {
        gProp = new GraphProperties(new File(graphFile.getParentFile(), "graph.properties"));
        long memoryTime = getTimeToRunInMemory();
long diskTime = getTimeToRunInDisk();
String firstLine;

firstLine = "AgrawalJagadish: " + formatter.format(gProp.getNumVertices()) + ": " + formatter.format(memoryTime) + ": " + formatter.format(diskTime);
System.out.println(firstLine);
fullDataStrB.insert(0, firstLine);
fullDataStrB.insert(0, "\n--AgrawalJagadishAlgorithm--");

randAccessFile.seek(randAccessFile.length());
rndAccessFile.write(fullDataStrB.toString().getBytes());
rndAccessFile.close();

/**
 * Returns the runtime in nanoseconds of the Warren Algorithm in memory.
 * @return the runtime in nanoseconds of the Warren Algorithm in memory.
 * @throws IOException
 * @throws Exception
 */
public long getTimeToRunInMemory() throws IOException, Exception {
    FullyBufferedAdjacencyGraphWR buffGraphWR = new FullyBufferedAdjacencyGraphWR(gProp, true);
    int numVert = gProp.getNumVertices();
    long totalTime;
    Row rowITemp;

    totalTime = System.nanoTime();
    for (int i = 0; i < numVert; i++) {
        rowITemp = buffGraphWR.readRow(i).clone();
        for (int j = i - 1; j >= 0; j--) {
            if (rowITemp.readBit(j)) {
                //add_succ
                for (int k = 0; k <= j - 1; k++) {
                    if (buffGraphWR.readBitOfRow(k, j)) {
                        if (rowITemp.readBit(k)) {
                        }
                    }
                }
            }
        }
    }

}
rowITemp.writeBit(k, false);
}
else {
    buffGraphWR.writeBitOfRow(k, i, true);
}
}
}
}

totalTime = System.nanoTime() - totalTime;

fullDataStrB.append("\nMemory\nBits Read: "+formatter.format(buffGraphWR.getBitReadCount())
+ "\nBits write: "+formatter.format(buffGraphWR.getBitWriteCount()));

buffGraphWR.close();
return totalTime;

/**
* Returns the runtime in nanoseconds of the Warren Algorithm from Disk.
* @return the runtime in nanoseconds of the Warren Algorithm from Disk
* @throws Exception
*/
public long getTimeToRunInDisk () throws Exception {
    AdjacencyGraphWR graphWR =
        new AdjacencyGraphWR(gProp, true);
    int numVert = gProp.getNumVertices();
    long totalTime = System.nanoTime();

    Row rowI;
    Row rowITemp;
    Row rowJ;

    totalTime = System.nanoTime();
    for (int i = 0; i < numVert; i++) {
        rowI = graphWR.readRow(i);
        rowITemp = rowI.clone();
for (int j = i - 1; j >= 0; j--) {
    if (rowITemp.readBit(j)) {
        //add_succ
        rowJ = graphWR.readRow(j);
        for (int k = 0; k <= j - 1; k++) {
            if (rowJ.readBit(k)) {
                if (rowITemp.readBit(k)) {
                    rowITemp.writeBit(k, false);
                } else {
                    rowI.writeBit(k, true);
                }
            }
        }
    }
    graphWR.writeRow(i, rowI);
    //write row back to disk
}
totalTime = System.nanoTime() - totalTime;

fullDataStrB.append("\nDisk\nBits Read: "
    + formatter.format(graphWR.getBitReadCount())
    + "\nBits write: "
    + formatter.format(graphWR.
        getBitWriteCount()));
graphWR.close();
return totalTime;
}

public static void main(final String...args) {
    File graphFile = new File(args[0]);
    try {
        new AgrawalJagadishHybridAlgorithm(graphFile);
    } catch(Exception e) {
        e.printStackTrace();
    }
}
package com._10x13.yangyuliudaowanphamhybridalgorithm;

import com._10x13.adjacencymatrixwrtool.readwrite.AdjacencyGraphWR;
import com._10x13.adjacencymatrixwrtool.readwrite.FullyBufferedAdjacencyGraphWR;
import com._10x13.adjacencymatrixwrtool.utilities.GraphProperties;
import com._10x13.adjacencymatrixwrtool.utilities.Row;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.RandomAccessFile;
import java.text.DecimalFormat;
import java.text.NumberFormat;

/**
 * @author T. Patrick Bailey
 */
public class YangYuLiuDaoWanPhamHybridAlgorithm {
    private StringBuffer fullDataStrB =
        new StringBuffer(1000);
    private RandomAccessFile randAccessFile = new RandomAccessFile(new File("graphOutput.txt"), "rw");
    private GraphProperties gProp;
    private NumberFormat formatter = new DecimalFormat("###,###,###,###,###,###,###,###,###,###,###,###,###,###,###,###,###,###,###,###,###,###,###");

    /**
     * Runs time test on the Warren Algorithm
     * @param graphFile
     * @throws IOException
     * @throws Exception
     */
    public YangYuLiuDaoWanPhamHybridAlgorithm(File graphFile) throws IOException, Exception {
        gProp = new GraphProperties(new File(graphFile.getParentFile(), "graph.properties"));
    }
}
long memoryTime = getTimeToRunInMemory();
long diskTime = getTimeToRunInDisk();
String firstLine;

firstLine =
"YangYuLiuDaoWanPhamHybridAlgorithm: "
+ formatter.format(gProp.getNumVertices())
+ ": " + formatter.format(memoryTime)
+ ": " + formatter.format(diskTime); System.out.println(firstLine);
fullDataStrB.insert(0, firstLine);
fullDataStrB.insert(0, "\n--YangYuLiuDaoWanPhamHybridAlgorithm--");

randAccessFile.seek(randAccessFile.length());
randAccessFile.write(
  fullDataStrB.toString().getBytes());
randAccessFile.close();

/**
 * Returns the runtime in nanoseconds of the Warren Algorithm in memory.
 * @return the runtime in nanoseconds of the Warren Algorithm in memory.
 * @throws IOException
 * @throws Exception
 */
public long getTimeToRunInMemory()
  throws IOException, Exception {
  FullyBufferedAdjacencyGraphWR buffGraphWR =
    new FullyBufferedAdjacencyGraphWR(gProp, true);
  int numVert = gProp.getNumVertices();
  long totalTime;
  Row rowI;
  Row rowJ;
  Row[] P = new Row[numVert];
  byte[] pByte;

  totalTime = System.nanoTime(); //Populate P
  for(int j = 0; j < numVert; j++) {
    pByte = new byte[1 + j/8];
    P[j] = new Row(j+1, 0, pByte);
for (int i = 0; i < j; i++) {
    if(buffGraphWR.readBitOfRow(j, i)) {
        P[j].writeBit(i, true);
    }
}

// Output time to create P[j]
System.out.println("YangYuLiuDaoWanPhamHybridAlgorithm-Memory-P"
    + "-CalcTime: ",
    + formatter.format(gProp.getNumVertices())
    + ": " + formatter.format(System.nanoTime()
        - totalTime));

totalTime = System.nanoTime();
for (int i = 0; i < numVert; i++) {
    // Initiate T
    for (int j = i + 1; j < numVert; j++) {
        // Procedure CT
        for (int x = i + 1; x < j; x++) {
            if (P[j].readBit(x)
                && buffGraphWR.readRow(i).readBit(x)) {
                buffGraphWR.
            }
        }
    }
}

totalTime = System.nanoTime() - totalTime;
fullDataStrB.append("\nMemory\nBits Read: "
    + formatter.format(
        buffGraphWR.getBitReadCount())
    + "\nBits write: "
    + formatter.format(
        buffGraphWR.getBitWriteCount()));
buffGraphWR.close();
return totalTime;
}

/**
 * Returns the runtime in nanoseconds of the Warren Algorithm from Disk.
 * @return the runtime in nanoseconds of the Warren Algorithm from Disk
 * @throws Exception
 */

public long getTimeToRunInDisk() throws Exception {
    AdjacencyGraphWR graphWR =
        new AdjacencyGraphWR(gProp, true);
    int numVert = gProp.getNumVertices();
    long totalTime;
    Row pJRow;
    byte[] pByte;
    int[] pjRowSize = new int[numVert];
    long[] pjSkipSize = new long[numVert];
    long totalpjSize = 0;
    BufferedOutputStream bos;
    RandomAccessFile randAccessFile = null;
    File PjFile = new File(
        gProp.getRowLengthStoredFile().getParentFile(), "PjFile");
    Row pjRow;
    Row rowI;

    totalTime = System.nanoTime();
    //Populate P
    PjFile.delete();
    bos = new BufferedOutputStream(
        new FileOutputStream(PjFile));
    for(int j = 0; j < numVert; j++) {
        pJRow = new Row(j + 1, 0, new byte[1 + j/8]);
        pjRowSize[j] = 1 + j/8;
        totalpjSize += 1 + j/8;
        pjSkipSize[j] = totalpjSize;
        for (int i = 0; i < j; i++) {
            if(graphWR.readBitOfRow(j, i)) {
pJRow.writeBit(i, true);
}
} // bos.write(pJRow.getRowData())
bos.flush();
} // bos.flush();
bos.close();

// Output time to create P[j]
System.out.println("YangYuliuDaoWanPhamHybridAlgorithm-HD-P"
+ "-CalcTime: 
+ formatter.format(gProp.getNumVertices())
+ ": " + formatter.format(System.nanoTime()
- totalTime));

totalTime = System.nanoTime();
randoAccessFile = new RandomAccessFile(PjFile, "r");
for (int i = 0; i < numVert; i++) {
  // Initiate Ti,j = 0 for i<j<=n already is
  // present in the graph
  rowI = graphWR.readRow(i);
  if(!rowI.readBit(i)) {
    // Find path if not already set.
    for (int j = i + 1; j < numVert; j++) {
      randAccessFile.seek(pjSkipSize[j]
- (long) pjRowSize[j]);
      pByte = new byte[pjRowSize[j]]; 
      randAccessFile.read(pByte);
      pjRow = new Row(j+1, 0, pByte);
      // Procedure CT(i,j)
      for(int x = i + 1; x < j; x++) {
        if(pjRow.readBit(x)
          && rowI.readBit(x)) {
          rowI.writeBit(j, true);
          break;
        }
      }
    }
  }
  graphWR.writeRow(i, rowI);
  // write row back to disk
```java
public static void main(final String... args) {
    File graphFile = new File(args[0]);
    try {
        new YangYuLiuDaoWanPhamHybridAlgorithm(graphFile);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```
package com._10x13.reducedhybridalgorithm;

import com._10x13.adjacencymatrixwrtool.readwrite.AdjacencyGraphWR;
import com._10x13.adjacencymatrixwrtool.readwrite.FullyBufferedAdjacencyGraphWR;
import com._10x13.adjacencymatrixwrtool.utilities.GraphProperties;
import com._10x13.adjacencymatrixwrtool.utilities.Row;
import java.io.File;
import java.io.IOException;
import java.io.RandomAccessFile;
import java.text.DecimalFormat;
import java.text.NumberFormat;

/**<p>@author T. Patrick Bailey  
@Date: 10/2010 </p>*/
public class ReducedHybridAlgorithm {
    private StringBuffer fullDataStrB = new StringBuffer(1000);
    private RandomAccessFile randAccessFile = new
          RandomAccessFile(new File("graphOutput.txt"), "rw");
    private GraphProperties gProp;
    private NumberFormat formatter = new
          DecimalFormat("###,###,###,###,###,###,###,###,###,###,###,###");

    /**<p>Runs time test on the Reduced Hybrid Algorithm
    @param graphFile
    @throws IOException  
    @throws Exception</p>*
    public ReducedHybridAlgorithm(File graphFile)
      throws IOException, Exception {
      gProp = new GraphProperties(new File(
          graphFile.getParentFile(), "graph.properties");
      long memoryTime = getTimeToRunInMemory();


long diskTime = getTimeToRunInDisk();
String firstLine;

firstLine = "Reduced-Hybrid: "  
    + formatter.format(gProp.getNumVertices())  
    + ": " + formatter.format(memoryTime)  
    + ": " + formatter.format(diskTime);
System.out.println(firstLine);
fullDataStrB.insert(0, firstLine);
fullDataStrB.insert(0,  
    "\n--ReducedAlgorithm--");

randAccessFile.seek(randAccessFile.length());
randAccessFile.write(fullDataStrB.  
    toString().getBytes());
randAccessFile.close();
}
/**
 * Returns the runtime in nanoseconds of the
 * Reduced Hybrid Algorithm in memory.
 * @return the runtime in nanoseconds of the
 * Reduced Hybrid Algorithm in memory.
 * @throws IOException
 * @throws Exception
 */
public long getTimeToRunInMemory()  
    throws IOException, Exception {
    FullyBufferedAdjacencyGraphWR buffGraphWR =  
        new FullyBufferedAdjacencyGraphWR(  
            gProp, true);
    int numVert = gProp.getNumVertices();
    long totalTime;
    Row rowITemp;
    int iN;

    totalTime = System.nanoTime();
    for (int i = 0; i < numVert; i++) {
        rowITemp = buffGraphWR.readRow(i).clone();
        iN = rowITemp.getZeroFillLength() * 8;
        for (int j = i - 1; j > iN; j--) {
            if (rowITemp.readBit(j)) {
                //add_succ
                for (int k = iN; k <= j - 1; k++) {
                    if (buffGraphWR.readBitOfRow(
```java
if (rowITemp.readBit(k)) {
    rowITemp.writeBit(k, false);
} else {
    buffGraphWR.writeBitOfRow(k, i, true);
}
```

```java
/**
 * Returns the runtime in nanoseconds of the Reduced Hybrid Algorithm from Disk.
 * @return the runtime in nanoseconds of the Reduced Hybrid Algorithm from Disk
 * @throws Exception
 */
public long getTimeToRunInDisk() throws Exception {
    AdjacencyGraphWR graphWR =
        new AdjacencyGraphWR(gProp, true);
    int numVert = gProp.getNumVertices();
    long totalTime;
    int iN;

    Row rowI;
    Row rowITemp;
    Row rowJ;

    totalTime = System.nanoTime();
    for (int i = 0; i < numVert; i++) {
```
rowI = graphWR.readRow(i);
rowITemp = rowI.clone();
iN = rowITemp.getZeroFillLength() * 8;
for (int j = i - 1; j > iN; j--) {
    if (rowITemp.readBit(j)) {
        //addsucc
        rowJ = graphWR.readRow(j);
        for (int k = iN; k <= j - 1; k++) {
            if (rowJ.readBit(k)) {
                if (rowITemp.readBit(k)) {
                    rowITemp.writeBit(k, false);
                } else {
                    rowI.writeBit(k, true);
                }
            }
        }
    }
}

graphWR.writeRow(i, rowI);
//write row back to disk
}
totalTime = System.nanoTime() - totalTime;

fullDataStrB.append("\nDisk\nBits Read: "
+ formatter.format(graphWR.getBitReadCount())
+ "\nBits write: "
+ formatter.format(
    graphWR.getBitWriteCount()));
graphWR.close();
return totalTime;
}

public static void main(final String...args) {
    File graphFile = new File(args[0]);
    try {
        new ReducedHybridAlgorithm(graphFile);
    } catch(Exception e) {
        e.printStackTrace();
    }
}
package com._10x13.agrawaljagadishblockedhybridalgorithm;

import com._10x13.adjacencymatrixwrtool.readwrite.AdjacencyGraphWR;
import com._10x13.adjacencymatrixwrtool.readwrite.FullyBufferedAdjacencyGraphWR;
import com._10x13.adjacencymatrixwrtool.utilities.GraphProperties;
import com._10x13.adjacencymatrixwrtool.utilities.Row;
import java.io.File;
import java.io.IOException;
import java.io.RandomAccessFile;
import java.text.DecimalFormat;
import java.text.NumberFormat;

/**
 * @author T. Patrick Bailey
 * Date: 10/2010
 */
public class AgrawalJagadishBlockedHybridAlgorithm {
    private StringBuffer fullDataStrB = new StringBuffer(1000);
    private RandomAccessFile randAccessFile = new RandomAccessFile(new File("graphOutput.txt"), "rw");
    private GraphProperties gProp;
    private NumberFormat formatter = new DecimalFormat("###,###,###,###,###,###,###,###");

    /**
     * Runs time test on the Warren Algorithm
     * @param graphFile
     * @throws IOException
     * @throws Exception
     */
    public AgrawalJagadishBlockedHybridAlgorithm(File graphFile) throws IOException, Exception {
        gProp = new GraphProperties(
            new File(graphFile.getParentFile(), "graph.properties");
    }
long blocked20Time = getTimeToRunBlocked(20);
String firstLine;

firstLine = "AgrawalJagadishBlocked_20: "
    + formatter.format(gProp.getNumVertices())
    + ": " + formatter.format(blocked20Time);
System.out.println(firstLine);
fullDataStrB.insert(0, firstLine);
fullDataStrB.insert(0,
    "\n--RakeshJagadishBlockedAlgorithm--");

randAccessFile.seek(randAccessFile.length());
randAccessFile.write(    fullDataStrB.toString().getBytes());
randAccessFile.close();

/**
 * Returns the runtime in nanoseconds of the
 * Blocked Agrawal–Jagadish Algorithm in memory.
 * @return the runtime in nanoseconds of the
 * Warren Algorithm in memory.
 * @throws IOException
 * @throws Exception
 */
public long getTimeToRunBlocked(final int num)
    throws IOException, Exception {
    AdjacencyGraphWR graphWR =
        new AdjacencyGraphWR(gProp, true);
    int numVert = gProp.getNumVertices();
    long totalTime = System.nanoTime();

    Row[] rows = new Row[num];
    Row[] rowsI = new Row[num];
    Row jRow = null;
    int jRowNum = -1;
    int iS;
    int iE;

    totalTime = System.nanoTime();
    // process the first num rows
    // initialize rows
    for (int x = 0; x < rows.length &&
        x < numVert; x++) {
        rows[x] = graphWR.readRow(x);
rowsI[x] = rows[x].clone();
}
for (int i = 0; i < num && i < numVert; i++) {
  for (int j = i - 1; j >= 0; j--) {
    if (rowsI[i].readBit(j)) {
      //add_succ
      for (int k = 0; k <= j - 1; k++) {
        if (rows[j].readBit(k)) {
          if (rowsI[i].readBit(k)) {
            rowsI[i].writeBit(k, false);
          } else {
            rows[i].writeBit(k, true);
          }
        }
      }
    }
  }
  graphWR.writeRow(i, rows[i]);
  //write row back to disk
}

for (int b = 1; b <= numVert/num; b++) {
  //initialize rows
  iS = b*num;
  iE = iS + rows.length;
  for (int x = 0; x < 20 && x+iS < numVert; x++) {
    rows[x] = graphWR.readRow(x+iS);
    rowsI[x] = rows[x].clone();
  }
  //off-diagonal block
  for (int j=iS-1; j >= 0; j--) {
    for (int i = 0; i < num; i++) {
      if(rowsI[i].readBit(j)) {
        if(jRowNum != j) {
          //only read in once per block
          jRowNum = j;
          jRow = graphWR.readRow(j);
        }
        //add_succe
        for (int k = 0; k <= j - 1; k++) {
          //add_succe
          for (int k = 0; k <= j - 1; k++) {
if (jRow.readBit(k)) {
    if (rowsI[i].readBit(k)) {
        rowsI[i].writeBit(k, false);
    } else {
        rows[i].writeBit(k, true);
    }
}

//diagonal block
for (int i = 0; i < num; i++) {
    //for (int i = iS; i <= iE; i++) {
        for (int j = i+iS; j >= iS; j--) {
            if (rowsI[i].readBit(j)) {
                //add_suc
                for (int k = 0; k <= j - 1; k++) {
                    if (rows[j-iS].readBit(k)) {
                        if (rowsI[i].readBit(k)) {
                            rowsI[i].writeBit(k, false);
                        } else {
                            rows[i].writeBit(k, true);
                        }
                    }
                }
            }
        }
    }
}

//Write rows out from memory to disk
for (int x = 0; x < num && x+iS < numVert; x++) {
    graphWR.writeRow(x+iS, rows[x]);
    //write row back to disk
}
totalTime = System.nanoTime() - totalTime;

fullDataStrB.append("\nDisk\nBits Read: "
+ formatter.format(graphWR.getBitReadCount())
+ "\nBits write: "
+ formatter.format(
    graphWR.getBitWriteCount())
);

graphWR.close();
return totalTime;
}

public static void main(final String...args) {
    File graphFile = new File(args[0]);

    try {
        new AgrawalJagadishBlockedHybridAlgorithm(
            graphFile);
    } catch(Exception e) {
        e.printStackTrace();
    }
}
package com._10x13.yangyuliudaowanphamblockedhybridalgorithm;

import com._10x13.adjacencymatrixwrtool.readwrite.AdjacencyGraphWR;
import com._10x13.adjacencymatrixwrtool.readwrite.FullyBufferedAdjacencyGraphWR;
import com._10x13.adjacencymatrixwrtool.utilties.GraphProperties;
import com._10x13.adjacencymatrixwrtool.utilties.Row;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.RandomAccessFile;
import java.text.DecimalFormat;
import java.text.NumberFormat;

/**
 * @author T. Patrick Bailey
 */
public class YangYuLiuDaoWanPhamBlockedHybridAlgorithm {
    private StringBuffer fullDataStrB = new StringBuffer(1000);
    private RandomAccessFile randAccessFile =
        new RandomAccessFile(new File("graphOutput.txt"), "rw");
    private GraphProperties gProp;
    private NumberFormat formatter = new DecimalFormat("###,###,###,###,###,###,###,###");

    /**
     * Runs time test on the Warren Algorithm
     * @param graphFile
     * @throws IOException
     * @throws Exception
     */
    public YangYuLiuDaoWanPhamBlockedHybridAlgorithm(File graphFile, final int blockSize)
                throws IOException, Exception {
        gProp = new GraphProperties(

new File(graphFile.getParentFile(),
"graph.properties");

long memoryTime =
    getTimeToRunInBlocked(blockSize);
String firstLine;
firstLine =
    "YangYuLiuDaoWanPhamBlockedHybridAlgorithm: "
    + formatter.format(gProp.getNumVertices())
    + ": " + formatter.format(memoryTime);
System.out.println(firstLine);
fullDataStrB.insert(0, firstLine);
fullDataStrB.insert(0,
    "\n--YangYuLiuDaoWanPhamBlockedHybridAlgorithm--");

randAccessFile.seek(randAccessFile.length());
r randAccessFile.write(
    fullDataStrB.toString().getBytes());
r randAccessFile.close();

/**
 * Returns the runtime in nanoseconds of the
 * Warren Algorithm from Disk.
 * @return the runtime in nanoseconds of the
 * Warren Algorithm from Disk
 * @throws Exception
 */
public long getTimeToRunInBlocked (final int blockSize) throws Exception {
    AdjacencyGraphWR graphWR =
        new AdjacencyGraphWR(gProp, true);
    int numVert = gProp.getNumVertices();
    long totalTime;
    Row pJRow;
    byte[] pByte;
    int[] pjRowSize = new int[numVert];
    long[] pjSkipSize = new long[numVert];
    long totalpjSize = 0;
    BufferedOutputStream bos;
    RandomAccessFile randAccessFile = null;
    File PjFile = new File(
        gProp.getRowLengthStoredFile().
        getParentFile(), "PjFile");

Row pjRow;
Row [] blockRows = new Row[blockSize];
int numBlocks = (numVert % blockSize == 0)
    ? numVert/blockSize: (numVert/blockSize) + 1;
int fnB;
int lnB;
totalTime = System.nanoTime();

//Populate P
PjFile.delete();
bos = new BufferedOutputStream(
    new FileOutputStream(PjFile));
for(int j = 0; j < numVert; j++) {
pJRow = new Row(j+1, 0, new byte[1 + j/8]);
pjRowSize[j] = 1 + j/8;
totalpjSize += 1 + j/8;
pjSkipSize[j] = totalpjSize;
for (int i = 0; i < j; i++) {
    if(graphWR.readBitOfRow(j, i)) {
        pJRow.writeBit(i, true);
    }
}
bos.write(pJRow.getRowData());
bos.flush();
}
bos.flush();
bos.close();

//Output time to create P[j]
System.out.println(
    "YangYuLiuDaoWanPhamHybridAlgorithm-HD-P"
    "+"-CalcTime: "
    + formatter.format(gProp.getNumVertices())
    + ": " + formatter.format(System.nanoTime()
        - totalTime));

totalTime = System.nanoTime();
randAccessFile = new RandomAccessFile( 
    PjFile, "r"); 

for (int b = 0; b < numBlocks; b++) {
    //initialize rows
    fnB = b*blockSize;
    lnB = fnB + blockRows.length;
for (int x = 0; x < blockSize && x+fnB < numVert; x++) {
    blockRows[x] = graphWR.readRow(x+fnB);
}

for(int j = fnB + 1; j < numVert; j++) {
    for(int i = fnB; i < j-1 && i < lnB; i++) {
        randAccessFile.seek(pjSkipSize[j] -(long) pjRowSize[j]);
        pByte = new byte[pjRowSize[j]];
        randAccessFile.read(pByte);
        pjRow = new Row(j+1, 0, pByte);
        //Procedure CT(i,j)
        for(int x = i + 1; x < j; x++) {
            if(pjRow.readBit(x) &&
               blockRows[i-fnB].readBit(x)) {
                blockRows[i-fnB].writeBit(
                    j, true);
                break;
            }
        }
    }
}

//Write blocks back to disk
for (int x = 0; x < blockSize && x+fnB < numVert; x++) {
    graphWR.writeRow(x+fnB, blockRows[x]);
    //write row back to disk
}

totalTime = System.nanoTime() - totalTime;

fullDataStrB.append("\nDisk\nBits Read: " + formatter.format(graphWR.getBitReadCount()) + "\nBits write: " + formatter.format(graphWR.getBitWriteCount()));
graphWR.close();
return totalTime;

public static void main(final String...args) {

File graphFile = new File(args[0]);
int blockSize = 20;
if(args.length > 1) {
    blockSize = Integer.parseInt(args[1]);
}
try {
    new YangYuLiuDaoWanPhamBlockedHybridAlgorithm(
        graphFile, blockSize);
} catch(Exception e) {
    e.printStackTrace();
}
package com._10x13.reducedhybridrollingqueuealgorithm;

import com._10x13.adjacencymatrixwrtool.readwrite.AdjacencyGraphWR;
import com._10x13.adjacencymatrixwrtool.readwrite.FullyBufferedAdjacencyGraphWR;
import com._10x13.adjacencymatrixwrtool.utilties.GraphProperties;
import com._10x13.adjacencymatrixwrtool.utilties.Row;
import java.io.File;
import java.io.IOException;
import java.io.RandomAccessFile;
import java.text.DecimalFormat;
import java.text.NumberFormat;

/**
 * @author T. Patrick Bailey
 * Date: 10/2010
 */
public class ReducedHybridRollingQueueAlgorithm {
    private StringBuffer fullDataStrB = new StringBuffer(1000);
    private RandomAccessFile randAccessFile = new RandomAccessFile(new File("graphOutput.txt"), "rw");
    private GraphProperties gProp;
    private NumberFormat formatter = new DecimalFormat("###,###,###,###,###,###,###,###");

    /**
     * Runs time test on the Reduced Hybrid Algorithm
     * @param graphFile
     * @throws IOException
     * @throws Exception
     */
    public ReducedHybridRollingQueueAlgorithm(File graphFile) throws IOException, Exception {
        gProp = new GraphProperties(new File(graphFile.getParentFile(), "graph.properties"));
        long rollingQueueTime =
getTimeToRunInMemoryRollingQueue();
String firstLine;
firstLine = "Reduced-Hybrid-Rolling-Queue: " +
") + formatter.format(rollingQueueTime);
System.out.println(firstLine);
fullDataStrB.insert(0, firstLine);
fullDataStrB.insert(0,
"\n--ReducedHybridRollingQueueAlgorithm--");
randAccessFile.seek(randAccessFile.length());
r
randAccessFile.write(
fullDataStrB.toString().getBytes());
r
randAccessFile.close();
}

/**
 * Returns the runtime in nanoseconds of the Reduced Hybrid Algorithm in memory.
 * @return the runtime in nanoseconds of the Reduced Hybrid Algorithm in memory.
 * @throws IOException
 * @throws Exception
 */
public long getTimeToRunInMemoryRollingQueue() throws IOException, Exception {
    AdjacencyGraphWR graphWR =
new AdjacencyGraphWR(gProp, true);
    int numVert = gProp.getNumVertices();
    long totalTime;
    int iN;
    int numRowsRolling;
    int minRowNumCached = 0;
    int maxRowNumCached = -1;
    Row[] rowCache = new Row[numVert];
    Row rowI;
    Row rowITemp;
    Row rowJ;
    totalTime = System.nanoTime();
    //Find the max number of rows to store in queue
    numRowsRolling =
    graphWR.getMaxRowByteLength()*8;
//cache the first rolling queue of rows
for(int x = 0; x < numRowsRolling; x++) {
    maxRowNumCached++;
    rowCache[x] = graphWR.readRow(x);
}

long num;
for (int i = 0; i < numVert; i++) {
    if(i %1000 == 0 && i != 0) {
        num = System.nanoTime() - totalTime;
        num = num/(long) 1000000000;
        num = numVert*num/i;
        System.out.println(i + " "
            + maxRowNumCached + " "
            + minRowNumCached
            + " "
            + formatter.format((
                Runtime.getRuntime().freeMemory()
                + Runtime.getRuntime().maxMemory()
                - Runtime.getRuntime().
                totalMemory())/(1024*1024)) + " MB"
            + " " + formatter.format(num));
    }
    //Roll queue forward
    if (i > maxRowNumCached) {
        //write row out before it is removed
        graphWR.writeRow(maxRowNumCached,
            rowCache[maxRowNumCached]);
        rowCache[minRowNumCached] = null;
        rowCache[i] = graphWR.readRow(i);
        maxRowNumCached++;
        minRowNumCached++;
        if(i % 1000 == 0) {
            System.gc();
        }
    }
}

rowITemp = rowCache[i].clone();
iN = rowITemp.getZeroFillLength() * 8;
for (int j = i - 1; j > iN; j--) {
    if (rowITemp.readBit(j)) {
        //add_succ
        rowJ = rowCache[j];
        for (int k = iN; k <= j -1; k++) {
            if (rowJ.readBit(k)) {
if (rowITemp.readBit(k)) {
    rowITemp.writeBit(k, false);
} else {
    rowCache[i].writeBit(k, true);
}

//write remaining rows to disc
for (int x = minRowNumCached; x < numVert; x++) {
    graphWR.writeRow(x, rowCache[x]);
}  
totalTime = System.nanoTime() - totalTime;

fullDataStrB.append("\nDisk\nBits Read: " + formatter.format(graphWR.getBitReadCount()) + "\nBits write: " + formatter.format(graphWR.getBitWriteCount()));

graphWR.close();
return totalTime;

public static void main(final String...args) {
    File graphFile = new File(args[0]);
    try {
        new ReducedHybridRollingQueueAlgorithm(graphFile);
    } catch(Exception e) {
        e.printStackTrace();
    }
}
```java
package com._10x13.reducedhybridrollingqueuealgorithm;

import com._10x13.adjacencymatrixwrtool.readwrite.AdjacencyGraphWR;
import com._10x13.adjacencymatrixwrtool.utilities.BitTool;
import com._10x13.adjacencymatrixwrtool.utilities.GraphProperties;
import com._10x13.adjacencymatrixwrtool.utilities.Row;
import java.io.File;
import java.io.IOException;
import java.text.DecimalFormat;
import java.text.NumberFormat;

/**
 * @author T. Patrick Bailey
 */
public class ReducedHybridRollingQueueAlgorithm64BitVersion {
    private GraphProperties gProp;
    private NumberFormat formatter = new DecimalFormat("###,###,###,###,###,###,###,###");

    public ReducedHybridRollingQueueAlgorithm64BitVersion(File graphFile) throws IOException, Exception {
        gProp = new GraphProperties(new File(graphFile.getParentFile(), "graph.properties"));
        long rollingQueueTime = getTimeToRunInMemoryRollingQueue();
        String firstLine = "Reduced-Hybrid-Rolling-Queue-64Bit: " + formatter.format(gProp.getNumVertices()) + ": " + formatter.format(rollingQueueTime);
        System.out.println(firstLine);
    }

    public long getTimeToRunInMemoryRollingQueue() {
        // Implementation...
    }

    public long getTimeToRunInMemoryRollingQueue() {
        // Implementation...
    }
}
```
throws IOException, Exception {
    AdjacencyGraphWR graphWR =
        new AdjacencyGraphWR(gProp, true);
    int numVert = gProp.getNumVertices();
    long totalTime;
    int iN;
    int numRowsRolling;
    int rowBuffer = 10000;
    int minRowNumCached = 0;
    int maxRowNumCached = -1;
    Row tempRow = null;
    Row64Bit[] row64Cache = new Row64Bit[numVert];
    int[] rowZeroFillLengths = new int[numVert];
    int[] byteRowLengths = new int[numVert];
    long l;
    long num;
    Row64Bit rowITemp;
    
    totalTime = System.nanoTime();
    //Find the max number of rows to store in queue
    numRowsRolling =
        graphWR.getMinRowsNeededWithBuffer(rowBuffer);
    //cache the first rolling queue of rows
    for(int x = 0; x < numRowsRolling &&
        x < numVert; x++) {
        maxRowNumCached++;
        tempRow = graphWR.readRow(x);
        row64Cache[x] =
            Row64Bit.convertByteRow(tempRow);
        rowZeroFillLengths[x] =
            tempRow.getZeroFillLength();
        byteRowLengths[x] =
            tempRow.getRowDataLength();
    }
    
    for (int i = 0; i < numVert; i++) {
        if(i %1000 == 0 && i != 0) {
            num = System.nanoTime() - totalTime;
            num = num/(long) 1000000000;
            num = numVert*num/i;
            System.out.println(i + "  "
                + maxRowNumCached + "  "
                + minRowNumCached
                + "  ");
        }
    }
}
+ formatter.format(
    Runtime.getRuntime().freeMemory()
+ Runtime.getRuntime().maxMemory() - Runtime.getRuntime().
    totalMemory()/(1024*1024))
+ " MB"
+ " " + formatter.format(num));

// Roll queue forward
if (i > maxRowNumCached) {
    // get the next rowBuffer nodes
    for(int x = 0; x < rowBuffer &&
        minRowNumCached <= maxRowNumCached;
        x ++){
        tempRow.resetValues(numVert,
        rowZeroFillLengths[minRowNumCached],
        new byte[byteRowLengths[
            minRowNumCached]]);
        graphWR.writeRow(minRowNumCached,
        row64Cache[minRowNumCached].
        insertIntoRow(tempRow));
        row64Cache[minRowNumCached] = null;
        minRowNumCached++;
    }
    System.gc();
    for(int x = 0; x < rowBuffer &&
        maxRowNumCached < numVert - 1; x++ ) {
        maxRowNumCached++;
        tempRow =
        graphWR.readRow(maxRowNumCached);
        rowZeroFillLengths[
            maxRowNumCached] =
        tempRow.getZeroFillLength();
        byteRowLengths[maxRowNumCached] =
        tempRow.getRowDataLength();
        row64Cache[maxRowNumCached] =
        Row64Bit.convertByteRow(tempRow);
    }
}

rowITemp = row64Cache[i].clone();
iN = rowITemp.getZeroFillLength();
for (int j = i - 1; j > iN*64; j--) {
    if (rowITemp.readBit(j)) {
        for (int k = iN; k <= (j - 1)/64;
k++) {
    l = (long) (row64Cache[i].readLong(k) | row64Cache[j].readLong(k));
    row64Cache[i].writeLong(k, l);
}
}

//write remaining rows to disc
for (int x = minRowNumCached; x < numVert; x++) {
    tempRow.resetValues(numVert,
                        rowZeroFillLengths[x],
                        new byte[byteRowLengths[x]]);
    graphWR.writeRow(
                        x, row64Cache[x].insertIntoRow(tempRow));
} totalTime = System.nanoTime() - totalTime;
graphWR.close();
return totalTime;

public static void main(String...args) {
    File graphFile = new File(args[0]);
    try {
        new ReducedHybridRollingQueueAlgorithm64BitVersion(
                        graphFile);
    } catch(Exception e) {
        e.printStackTrace();
    }
}
package com._10x13.reducedhybridrollingqueuealgorithm;

import com._10x13.adjacencymatrixwrtool.readwrite.AdjacencyGraphWR;
import com._10x13.adjacencymatrixwrtool.utilties.GraphProperties;
import com._10x13.adjacencymatrixwrtool.utilties.Row;
import java.io.File;
import java.io.IOException;
import java.text.DecimalFormat;
import java.text.NumberFormat;

/**
 * @author T. Patrick Bailey
 */

public class ReducedHybridRollingQueueAlgorithm64BitThreadedVersion {
    private GraphProperties gProp;
    AdjacencyGraphWR graphWR;
    private NumberFormat formatter = new DecimalFormat("###,###,###,###,###,###,###,###");
    int minRowNumCached = 0;
    int maxRowNumCached = -1;
    int numVert;
    Row64Bit[] row64Cache;
    int[] rowZeroFillLengths;
    int[] byteRowLengths;
    int iNum = -1;
    long totalTime;
    int rowBuffer;
    private long diskTime = 0;

    public ReducedHybridRollingQueueAlgorithm64BitThreadedVersion(File graphFile, int numThreads) throws IOException, Exception {
        gProp = new GraphProperties(new File(graphFile.getParentFile(), "graph.properties"));
        graphWR = new AdjacencyGraphWR(gProp, true);
        numVert = gProp.getNumVertices();
        row64Cache = new Row64Bit[numVert];
    }
}
rowZeroFillLengths = new int[numVert];
byteRowLengths = new int[numVert];

long rollingQueueTime =
    getTimeToRunInMemoryRollingQueue(numThreads);
String firstLine;

firstLine = "Reduced-Hybrid-Rolling-Queue-"
    + "64Bit-Threaded-" + numThreads + "": "
    + formatter.format(gProp.getNumVertices())
    + "": " + formatter.format(rollingQueueTime)
    + "": " + formatter.format(diskTime);
System.out.println(firstLine);
}

public long getTimeToRunInMemoryRollingQueue(final int numThreads)
    throws IOException, Exception {
    totalTime = System.nanoTime();
    Row tempRow = new Row(numVert, 0, new byte[1]);

    populateCache();
    startThreads(numThreads);
    writeRemaingInCache();

    //write remaining rows to disc
    for (int x = minRowNumCached; x < numVert;
        x++) {
        tempRow.resetValues(numVert,
            rowZeroFillLengths[x],
            new byte[byteRowLengths[x]]);
        graphWR.writeRow(
            x, row64Cache[x].insertIntoRow(tempRow));
    }
    return System.nanoTime() - totalTime;
}

private void populateCache()
    throws IOException, Exception {
    int numRowsRolling;
    rowBuffer = 1000;
    Row tempRow = null;
    long localDiskTime;

    //Find the max number of rows to store in queue
numRowsRolling =
graphWR.getMinRowsNeededWithBuffer(rowBuffer);

//cache the first rolling queue of rows
localDiskTime = System.nanoTime();
for(int x = 0; x < numRowsRolling &&
  x < numVert; x++) {
  maxRowNumCached++;
  tempRow = graphWR.readRow(x);
  row64Cache[x] =
    Row64Bit.convertByteRow(tempRow);
  rowZeroFillLengths[x] =
    tempRow.getZeroFillLength();
  byteRowLengths[x] =
    tempRow.getRowDataLength();
}
diskTime += System.nanoTime() - localDiskTime;

private void writeRemaingInCache()
  throws IOException, Exception {
  Row tempRow = new Row(numVert, 0, new byte[1]);
  long localDiskTime = System.nanoTime();

  //write remaining rows to disc
  for (int x = minRowNumCached; x < numVert;
       x++) {
    tempRow.resetValues(numVert,
                        rowZeroFillLengths[x],
                        new byte[byteRowLengths[x]]);
    graphWR.writeRow(
      x, row64Cache[x].insertIntoRow(tempRow));
  }
diskTime += System.nanoTime() - localDiskTime;
}

private void startThreads(final int numThreads){
  ProcessThread[] processes =
    new ProcessThread[numThreads];

  //Create threads
  for (int x = 0; x < processes.length; x++) {
    processes[x] = new ProcessThread(this, x+1);
    //create thread
    processes[x].start(); //start thread
try {
    for (int x = 0; x < processes.length; x++) {
        processes[x].join();
        //this waits until the thread ends to continue
    }
} catch (InterruptedException e) {
    e.printStackTrace();
}

public synchronized int getNextRow()
    throws IOException, Exception {
    iNum++;
    if(iNum < numVert) {
        if(iNum %1000 == 0 && iNum != 0) {
            long num = System.nanoTime() - totalTime;
            num = num/(long) 1000000000;
            num = numVert*num/iNum;
            System.out.println(iNum + " " + maxRowNumCached + " " + minRowNumCached + " " + formatter.format((Runtime.getRuntime().freeMemory() + Runtime.getRuntime().maxMemory() - Runtime.getRuntime().totalMemory())/(1024*1024)) + " MB" + " " + formatter.format(num));
        }
        if (iNum > maxRowNumCached) {
            long localDiskTime = System.nanoTime();
            Row tempRow = new Row(numVert, 0,
                new byte[1]);
            //get the next rowBuffer nodes
            for(int x = 0; x < rowBuffer && minRowNumCached <= maxRowNumCached;
                x ++){
                tempRow.resetValues(numVert,
rowZeroFillLengths[minRowNumCached],
new byte[
  byteRowLengths[minRowNumCached]]);
graphWR.writeRow(minRowNumCached,
  row64Cache[minRowNumCached].
    insertIntoRow(tempRow));
  row64Cache[minRowNumCached] = null;
  minRowNumCached++;}
}
System.gc();
for(int x = 0; x < rowBuffer && maxRowNumCached < numVert - 1;
    x++) {
  maxRowNumCached++;
  tempRow = graphWR.
    readRow(maxRowNumCached);
  rowZeroFillLengths[maxRowNumCached] = tempRow.getZeroFillLength();
  byteRowLengths[maxRowNumCached] = tempRow.getRowDataLength();
  row64Cache[maxRowNumCached] = Row64Bit.convertByteRow(tempRow);
}
diskTime += System.nanoTime() - localDiskTime;
}
return iNum;
}

public static void main(String...args) {
  File graphFile = new File(args[0]);
  int numThreads = 2;

  if(args.length > 1) {
    numThreads = Integer.parseInt(args[1]);
  }

  try {
    new ReducedHybridRollingQueueAlgorithm64BitThreadedVersion(
      graphFile, numThreads);
  } catch(Exception e) {
    e.printStackTrace();
  }
/**
 * Threads for processing graph.
 * @author T. Patrick Bailey
 */
class ProcessThread extends Thread {
  private final ReducedHybridRollingQueueAlgorithm64BitThreadedVersion algo;
  private final int iRow = -1;
  private final int num;

  public ProcessThread(ReducedHybridRollingQueueAlgorithm64BitThreadedVersion algo, int num) {
    this.algo = algo;
    this.num = num;
  }

  public void run() {
    int iNum;
    int iN;
    Row64Bit rowITemp;
    long l;

    try{
      iNum = algo.getNextRow();
      while(iNum < algo.numVert) {
        rowITemp =
          algo.row64Cache[iNum].clone();
        iN = rowITemp.getZeroFillLength();
        for (int j = iNum - 1; j > iN*64; j--) {
          if (rowITemp.readBit(j)) {
            for (int k = iN; k <= (j - 1)/64; k++) {
              l = (long)
                (algo.row64Cache[iNum]
                  .readLong(k) |
                algo.row64Cache[j].readLong(k));

              algo.row64Cache[iNum]
writeLong(k, l);
}
}
iNum = algo.getNextRow();
}
} catch(Exception e){
  e.printStackTrace();
}
}
BIBLIOGRAPHY


[25] Yang, Qi; Yu, Clement; Liu, Chengwen; Dao, Son; Wang, Gaoming; Pham, Tracy. (1994). A Hybrid Transitive Closure Algorithm for Sequential and Parallel Processing. Proceedings of the Tenth International Conference on Data Engineering, (pp. 498-505). Houston, Texas, USA.