Comparison of (0,1)-Matrix-Vector Product Difference-Based Algorithms

Jeffrey D. Witthuhn and Andrew A. Anda
Department of Computer Science and Information Technology
Saint Cloud State University
Saint Cloud, MN 56301
aanda@stcloudstate.edu

Abstract

All elements of a (0,1)-matrix are from the set {0,1}. The (0,1)-matrix-vector product, with the vector being over any algebraic ring, arises in many application areas. Although there is no algorithm that can reduce the quadratic complexity of the general matrix-vector product, we can exploit redundancies, if they exist, in the calculation of the (0,1)-matrix-vector product to reduce the count of scalar operations. We describe and compare our two redundancy-reducing algorithms. To reduce scalar operations in computing a (0,1)-matrix-vector product, one of our algorithms uses a Gray code and the other algorithm uses a minimum spanning tree. After implementing our two algorithms, we compare the count of addition operations performed by both our algorithms against the count required by the general (0,1)-matrix-vector product algorithm. We explore how the counts of operations change as we vary matrix sizes, ratios of columns to rows, and the sparsities of the matrices. We show that both our algorithms can reduce operation counts compared to the general method. The operation reduction benefits increase as the matrices become denser or become relatively taller. Based on our analyses, we find that between our two algorithms, our minimum spanning tree algorithm requires fewer operations in most cases. However, the Gray code algorithm often requires fewer operations for relatively large matrices.

1. Introduction

A (0,1)-matrix is an $m \times n$ matrix where each element a_{ij} is from the set $\{0,1\}$. (0,1)-matrices arise from problems in a variety of application areas such as graph theory [4], information retrieval [5], and matrix calculus [6]. The general matrix-vector product operation, $\boldsymbol{Ax} = \boldsymbol{y}$, represents the product of a matrix $\boldsymbol{A} \in \mathbb{R}^{m \times n}$, by the vector, $x \in \mathbb{R}^n$, to yield the vector, $\boldsymbol{y} \in \mathbb{R}^n$. More generally, the vectors may be over any Abelian group. If $\boldsymbol{A}$ is a (0,1)-matrix, the *i*th element in the resultant vector, $\boldsymbol{y}$, can be computed by the sum of the elements in the operand vector, $\boldsymbol{x}$, corresponding to the non-zero elements in the *i*th row of $\boldsymbol{A}$.

Consider the general product of two matrices which has cubic operational complexity. For this operation there exist known algorithms of lower complexity, for example Strassen's algorithm which has the less-than-cubic complexity of $O(n^{lg7})$. For the general matrix-vector product algorithm however, which has quadratic complexity, there is no known similar general complexity-reducing algorithm, however there are specific classes of low-rank structured matrices where the complexity may be reduced below quadratic to log-linear.

In our previous publications, we describe three original algorithms for applying what we term the *differencing method* which relies an observation that elements in the result vector of the (0,1)-matrix-vector product can be computed using values of previously computed elements. [1, 2, 3, 7] Here we restrict our investigations to two of our three algorithms: the Gray code-based algorithm and the MST-based algorithm. We compare, and discuss these two of our algorithms. The *differencing method* relies our observation that elements in the result vector can be computed using values of previously computed elements. Specifically the count of differing bits, the *Hamming distance,* between the two rows of the (0,1)-matrix corresponding to the element that is being computed and the previously computed element is the minimum count of operations that will be required to compute the new element.

For our first algorithm, which is *data oblivious,* we generate an *n*-bit circular *Gray Code* which is a circular ordering of all 2^n binary strings in which every adjacent bit string has a Hamming distance of 1. We then make use of the Gray Code properties and the differencing method to calculate all of the 2^n possible elements of the resultant vector in 2^n steps.

In our second algorithm, which is *data sensitive,* we generate a complete graph where each row of the (0,1)-matrix represents the vertices and the Hamming distance between each row represents the weights of each edge of a complete graph. Generating a minimum spanning tree on this complete graph represents an efficient way to apply the differencing method to any given (0,1)-matrix.

We will discuss the differencing method and our two algorithms that make use of it. We will then discuss our implementations of these algorithms which count the number of

addition and subtraction operations required to compute the product. We then test these algorithms and observe how the count of operations changes when we vary the size, sparsity, and the columns-to-rows ratio of the matrix. We conclude by discussing the implications of our results.

2. The Differencing Method[1]

Consider the general matrix-vector product $\boldsymbol{Ax} = \boldsymbol{y}, \boldsymbol{A} \in \mathbb{R}^{m\times n}, \boldsymbol{x} \in \mathbb{R}^{n}, \boldsymbol{y} \in \mathbb{R}^{m}$. This can be computed by a doubly nested loop represented by the following equation:

$$y_i = \sum_{j=1}^{n} \alpha_{ij} x_j \ , 1 \le i \le m,$$

where α_{ij} is the element in the ith row and jth column of A.

For a {0,1}-matrix this calculation with $\forall i, j,\ \alpha_{ij} \in \{0,1\}$ instead of multiplication we compute a sum where if $\alpha_{ij} = 1$ then the product $\alpha_{ij}\boldsymbol{x}_j$ contributes to the sum as $\boldsymbol{x}_j$ and in the case of $\alpha_{ij} = 0$ the element is skipped. This operation results in at most $m \times n$ additions which occurs when each $\alpha_{ij} = 1$.

Next, consider taking the difference between two elements of $\boldsymbol{y}, \boldsymbol{y}_i$ and $\boldsymbol{y}_k$:

$$y_i - y_k = \sum_{j=1}^{n} \alpha_{ij} x_j - \sum_{j=1}^{n} \alpha_{kj} x_j = \sum_{j=1}^{n} (\alpha_{ij} x_j - \alpha_{kj} x_j) = \sum_{j=1}^{n} x_j (\alpha_{ij} - \alpha_{kj})$$

Now consider computing $\boldsymbol{y}_i$ given that $\boldsymbol{y}_k$ has already been computed:

$$y_i = y_k + \sum_{j=1}^{n} x_j (\alpha_{ij} - \alpha_{kj}) = y_k + \sum_{j=1}^{n} x_j (d_j),\ d_j = \alpha_{ij} - \alpha_{kj}$$

This implies that aside from the first element of *y* to be computed, each subsequent element,$\boldsymbol{y}_i$, can be computed as the sum of a previously computed element, $\boldsymbol{y}_k$, with the inner product of $\boldsymbol{x}$ with the difference vector of the *ith* and *kth* rows, *d.* Let $\boldsymbol{A}_i$be the *ith* row of $\boldsymbol{A}$. And let $\|\boldsymbol{A}_i\|_1$ be the count of 1's in $\boldsymbol{A}_i$ and $\|\boldsymbol{d}\|_1$ be the count of 1's in $\boldsymbol{d}$. Then $\|\boldsymbol{A}_i\|_1 - \|\boldsymbol{d}\|_1 - 1$ equals the count of operations saved in the computation of $\boldsymbol{y}_i$. $\|\boldsymbol{d}\|_1$ is also the *Hamming distance* between the two rows of $\boldsymbol{A}$. So operations can be saved reusing previously computed results and adding or subtracting the elements in $\boldsymbol{x}$ where the two rows differ. In our two algorithms we fully exploit this observation.

For example, consider the following triangular (0,1)-matrix-vector product:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} a \\ a+b \\ a+b+c \\ a+b+c+d \end{pmatrix} = \begin{pmatrix} y_0 = a \\ y_1 = y_0 + b \\ y_2 = y_1 + c \\ y_3 = y_2 + d \end{pmatrix}$$

In this example we halved the count of operations required from $\frac{n(n-1)}{2}$ to $(n-1)$ by the applying our differencing method.

Several notes:

- Duplicate rows can be computed using zero additional operations by loading the previously computed element.
- The minimum possible count of operations to calculate any given element is given by the minimum Hamming distance between two rows in the (0, 1)-matrix.
- The optimal case with no duplicate rows would be where every row has a Hamming distance from its two adjacent rows of exactly one.

3. Our Gray Code Algorithm[1]

As noted above, the optimum case to exploit the differencing method would be the case in which rows have a Hamming distance of exactly one between them. This is the exactly the case in a *Gray Code*. A Gray Code is an arrangement of bit strings in which every adjacent bit strings has a Hamming distance of at most one. A circular *n*-bit Gray Code is a sequence of 2^n bit strings of length *n* in which every adjacent bit string has Hamming distance of one, with the first and the last bit strings also having a Hamming distance of one.

This algorithm is based on our observation that for some general (0,1)-matrix-product, $\boldsymbol{Ax} = \boldsymbol{y},\ \boldsymbol{A} \in \{0,1\}^{m \times n}, \boldsymbol{x} \in \mathbb{R}^n, \boldsymbol{y} \in \mathbb{R}^m$, we can generate a *n*-bit Gray Code which, since it represents all *n*-bit binary strings, will contain each row of $\boldsymbol{A}$ exactly once. Then we can use our differencing method to compute the inner product between each row of the Gray Code and the $\boldsymbol{x}$ vector to generate a temporary vector in 2^n steps. Subsequently, only *m* lookups and retrievals are required, one for each row in $\boldsymbol{A}$, to compute the resultant vector $\boldsymbol{y}$.

Our Gray code algorithm is:

- *Input:* **A**, n, m, **x**
- *Generate an* n*-bit Gray Code*
- *Array* **y**' [2^n]
- *Array* **y** [m]
- *For* $i \leftarrow 1$ *to* $i \leftarrow 2^n$
    - *Calculate the inner product of row* i *in the Gray Code and* **x** *and store into* **y**'[i]
- *For* i ← 1 *to* i ← m
    - *Search row* $\mathbf{A}_i$ *in the Gray Code, and store the corresponding element in* **y**' *in* **y**[i]
- *Return* **y**

This algorithm, in terms of space requirements, grows exponentially with *n* since there is 2^n rows in the Gray Code. So, we extend the algorithm to partition the (0,1)-matrix into $\frac{n}{k}$, *k*-bit partitions. That is, we choose a partition size, *k*, and then generate a *k*-bit Gray code which will then be used to compute $\frac{n}{k}$ sub-products which will add together to compute $\boldsymbol{y}$. This algorithm's space requirement grows linearly with *n* since here is $\frac{n}{k}$ scratch $\boldsymbol{y}$'s of length 2^k.

4. Our Minimum Spanning Tree Algorithm. [7]

The Hamming distance between any two binary vectors is the minimum count of operations required to compute a corresponding result vector element if the other element has been previously computed. This leads to the problem of finding the order in which to calculate each element and from what elements they should be calculated from that yields the leads count of additions. To solve this problem we use graphs.

Given a (0,1)-matrix, $\boldsymbol{A} \in \{0,1\}^{m \times n}$, we generate a complete graph $\boldsymbol{K}_m$ with $\frac{m(m-1)}{2}$ edges where each row $\boldsymbol{A}_i$ of $\boldsymbol{A}$ is represented by a vertex in $\boldsymbol{K}_m$, and each edge on the graph has weight equal to the Hamming distance between the two rows that it connects. Next we compute the minimum spanning tree of $\boldsymbol{K}_m$ choosing some vertex as the root. A traversal of this graph will represent computing the (0,1)-matrix-vector product where visiting each vertex consists of calculating the corresponding element using the difference between the parent of the current row. This will yield the fewest additions. (See Figure 4.1)

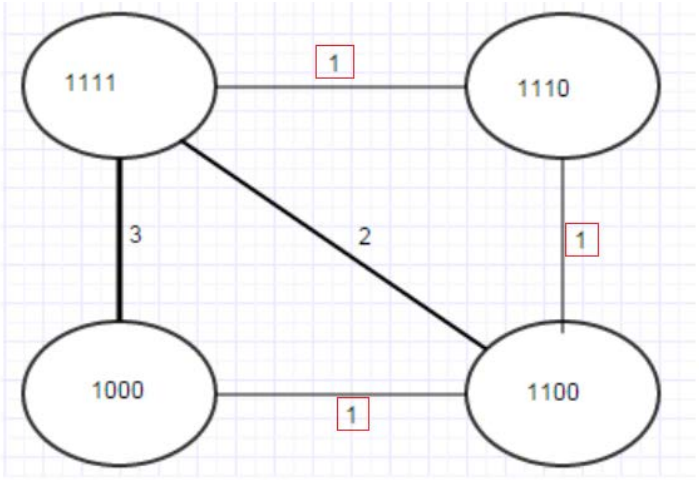

Figure 4.1: Graph of the (0,1)-matrix: $\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$

More precisely, the algorithm is as follows:

- *Input:* **A**, n, m, **x**
- *Generate a complete graph,* K, *from* **A**
- *Compute the minimum spanning tree,* **T**, *of* **K**
- *Array* **y**[m]
- *Decide on value to be the root of the tree,* r
- *Calculate value in* **y** *corresponding to* r
- *Perform a breadth first traversal of* **T** *starting with* r *where visiting each vertex consists of computing the value of the corresponding element in* **y** *using the value corresponding to its parent vertex*
- *Return* **y**

As with the Gray Code algorithm, we can extend this algorithm by vertically partitioning the (0,1)-matrix into partitions of size *k* and then calculating $\frac{n}{k}$ sub-products using this method. One advantage of this method is that we create $\frac{n}{k}$ separate minimum spanning trees in an attempt to further reduce the total count of operations required. Another advantage is that we reduce the number of degrees of freedom between each vertex from *n*-bits to *k*-bits and hence the maximum Hamming distance between two vertices is reduced.

5. Implementations[8]

Our algorithms were implemented in C++ and the BOOST C++ libraries. The purpose of our implementations is to determine the count of addition and subtraction operations required by our algorithms. If these algorithms were to be considered in applications, our implementations would need much more refactoring to improve their run-time and space efficiency.

5.1. Representation

(0,1)-matrices are represented by the `Bitmatrix` class which stores the bits of a (0,1)-matrix, $\boldsymbol{A} \in \{0,1\}^{m \times n}$ in a `std::vector` containing m `boost::dynamic_bitset<>`s of length n. Vectors $\boldsymbol{x} \in \mathbb{R}^n$ are represented by a $n \times 1$ matrix in the class `Doublematrix`, which stores each element as type `double` in a `vector` of n `vector`s of length 1. These classes contain all of the operations we use to explore these algorithms.

5.2. Gray Code Algorithm

The Gray Code algorithm with the partition extension is implemented as a method in *Bitmatrix*. There are many binary Gray codes but we choose to use the *reflected binary Gray code.* This Gray code is very convenient to implement because the *i*th row of the *n*-bit reflected binary Gray code can be calculated by the following equation:

$$\boldsymbol{G}_i \leftarrow i \oplus (i \gg 1) \text{ [9]}$$

Likewise, given $\boldsymbol{G}_i$ we can find *i:*

$$i \leftarrow \bigoplus_{k=0}^{n-1} \boldsymbol{G}_i \gg k \text{ [9]}$$

So we have a very nice mapping to and from the integers and the Gray code. So, let the partition size be *p*, we first generate a *p*-bit Gray Code in this way (represented in the same way as (0,1)-matrices). Then we compute the $\frac{n}{p}$ scratch **y**'s. Finally, we compute and assemble the output vector.

5.3. MST Algorithm

To represent the graph of the (0,1)-matrix, we use an *adjacency list* which is constructed using another vector of *m* vectors of integers. This representation allows us to quickly access the list of adjacent vertices indexed from zero to *m.* So in the implementation we split the (0,1)-matrix into partitions of size *k* and for each partition utilize the BOOST 1.59 implementation of Prim's MST algorithm to calculate the MST of each individual partition. Then for each partition we convert the output of that algorithm into the adjacency list representation described earlier and utilize it for a breadth first traversal. For the breadth first traversal, we make extensive use of C++ vectors to calculate each sub-product. Finally all sub-products are merged together and the final result vector is returned.

5.4. Counting operations

We count the number of additions and subtractions required by our algorithm in calculating each element in the sub-products as well as combining all of the sub-products into the final result vector.

6. Testing the Algorithms

In testing these algorithms, we vary the count of elements in the matrix, the ratio of columns to rows, as well as the sparsity of the matrix and we observe how the count of operations change. To do this, we define a set of tests that will work under the constraints of the implementation. Each test will compare the count of operations required by the general (0,1)-matrix-vector product algorithm with the operations required by our MST algorithm and our Gray Code algorithm.

6.1. Count of Elements

We want to vary the count of elements in the matrix while keeping constant the ratio of columns to rows and the sparsity. To do this, we use square matrices with a sparsity of 50% with partitions of size 8. The number of columns are set to be powers of two from the set $\{8, 16, 32, 64, 128, 256, 512\}$.

6.2. Columns-to-Rows Ratio

When varying the columns-to-rows ratio, we need to maintain the total count of elements and the sparsity. Here, we use two sets of data with two different sizes of matrices. For the first test set, we use matrices of size $512 \times 512 = 262{,}144$ elements in the (0,1)-matrix and 50% sparsity. The ratios we test, $\frac{columns}{rows}$, are from the set:

$$\{0.25, 1, 4, 16, 64, 256, 1024, 4096\}.$$

In the other data set, we use matrices of size $128 \times 128 = 16{,}384$ elements in the (0,1)-matrix and 50% sparsity. The ratios we test, $\frac{columns}{rows}$, are from the set:

$$\{0.0039063, 0.015625, 0.0625, 0.25, 1, 4, 16, 64, 256\}.$$

These two data sets are used to further illustrate how the required count of operations changes as the matrices become relatively wider.

6.3. Sparsity

We test how square (0,1)-matrices of dimension 512×512 results vary as the count of 1's changes from 0% to 100%.

7. Results

Using the set of tests described above, we now discuss the generated data (displayed using tables and graphs).

7.1. Count of Elements

Figure 7.1.1 and Table 7.1.1 show the data generated by our tests. From these, we observe that as the count of elements increases, the operation counts of all three algorithms appear to grow near linearly. For very large matrices, our Gray code algorithm appears to require fewer operations than both the general algorithm and our MST algorithm. For all sizes, our MST algorithm case requires fewer operations than the general algorithm case.

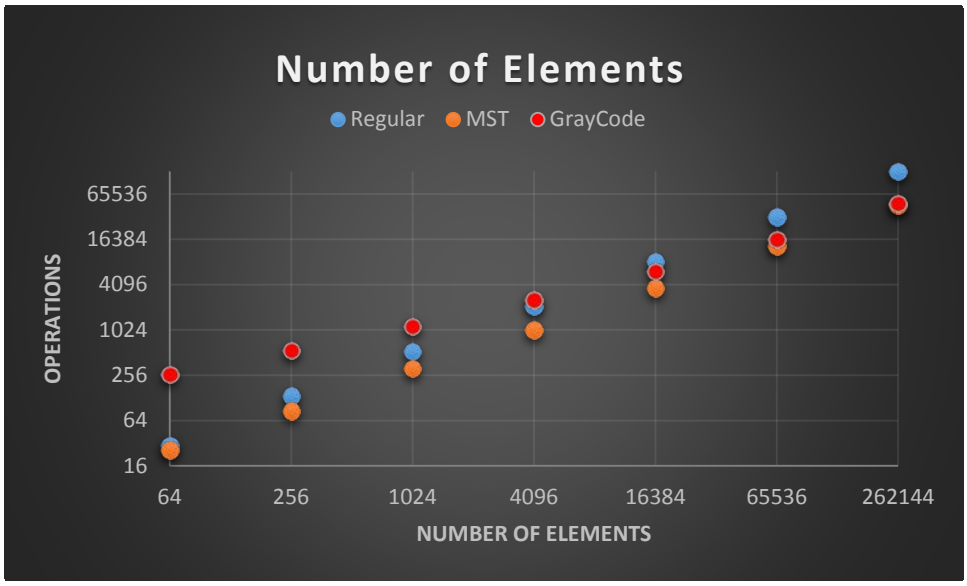

Figure 7.1.1: Plot of the number of operations required for each algorithm vs the number of elements in the square (0,1) matrix.

Size	Regular	MST	GrayCode
64	30	26	263
256	136	86	542
1024	527	314	1147
4096	2097	1031	2549
16384	8264	3668	6118
65536	32682	13383	16317
262144	131032	46737	48957

Table 7.1.1: Table format for the data in Figure 7.1.1.

7.2. Columns-to-Rows Ratio

7.2.1. $size = 512 \times 512$

Figure 7.2.1.1 and Table 7.2.1.1 show the data generated by this test. For both of our algorithms, the benefits decrease as the matrices become relatively wider. In our MST algorithm case, the count of operations approaches the count of operations required for the general algorithm case. In our Gray code algorithm case, the count of operations becomes greater than that of the general algorithm case sometime after the ratio exceeds 16:1.

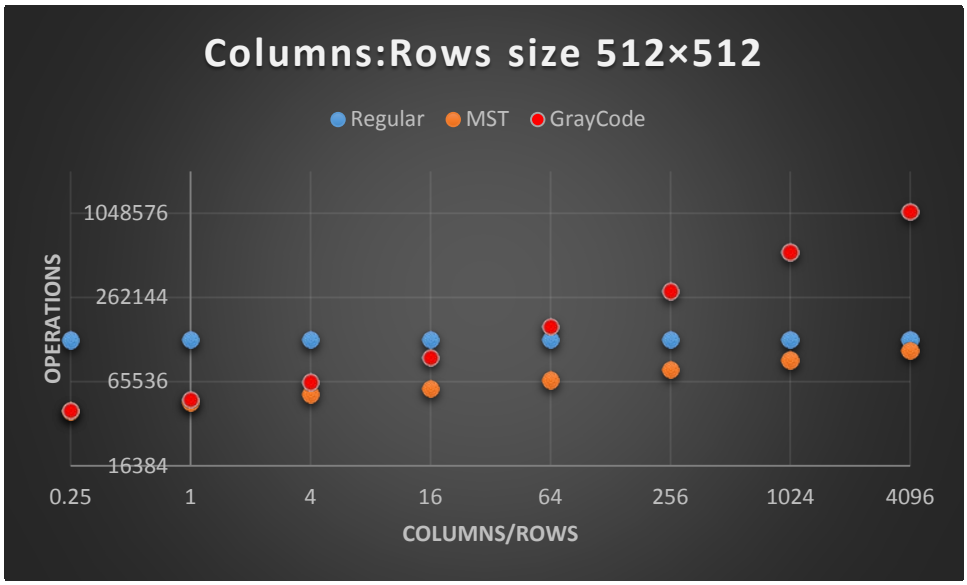

Figure 7.2.1.1: Plot of the number of operations required for each algorithm vs the rows/columns ratio of the (0,1) matrix with 512×512 elements.

columns/rows	Regular	MST	GrayCode
0.25	130469	40629	40792
1	131242	46719	48956
4	131030	53295	65262
16	131268	58796	97907
64	131143	67339	163193
256	131132	80171	293749
1024	130925	93734	554878
4096	130981	110132	1077112

Table 7.2.1.1: Table format for the data in Figure 7.2.1.1

7.2.2. $size = 128 \times 128$

Figure 7.2.2.1 and Table 7.2.2.1 show the data generated by this test. For both of our algorithms, the benefits decrease as the matrices become wider. In our MST algorithm case, the count of operations approaches the count of operations required for the general algorithm case. In our Gray code case, the count of operations becomes greater than the general algorithm case sometime after the ratio exceeds 1:1.

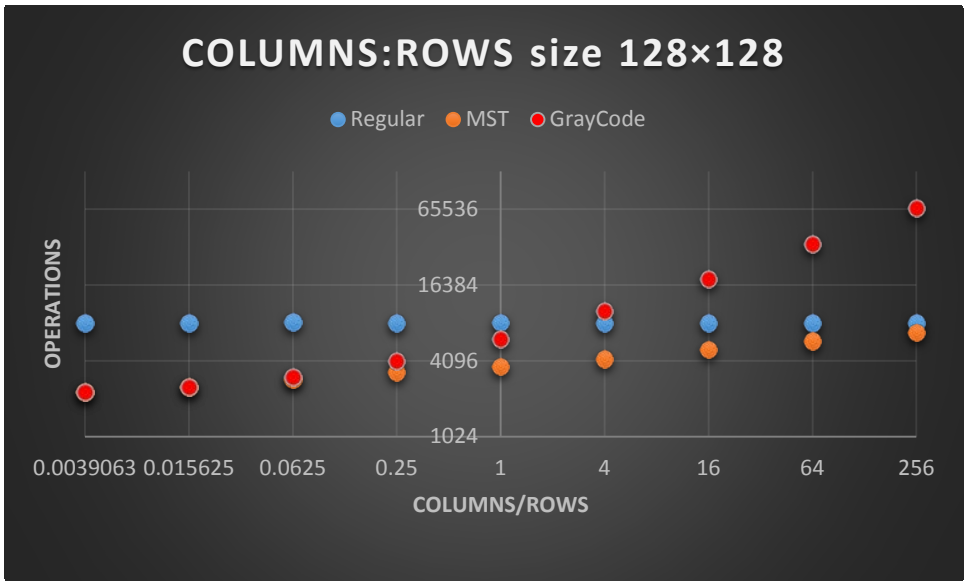

Figure 7.2.2.1: Plot of the number of operations required for each algorithm vs the rows/columns ratio of the (0,1) matrix with 512×512 elements.

columns/rows	Regular	MST	GrayCode
0.003906	8154	2299	2299
0.015625	8233	2542	2552
0.0625	8296	2917	3059
0.25	8208	3338	4079
1	8269	3700	6123
4	8145	4240	10196
16	8150	5036	18359
64	8226	5893	34686
256	8167	6883	67315

Table 7.2.2.1: Table format for the data in Figure 7.2.2.1

7.3. Sparsity

Figure 7.3.1 and Table 7.3.1 illustrate the data generated by these tests. For very sparse matrices, there are no benefits to the using our algorithms. But after there are at least 20% 1's in the matrices, both algorithms require increasingly fewer operations than the general method. Our MST algorithm's count of operations appears to be bounded above by the count of operations required by the Gray code algorithm. And the Gray code algorithm's performance appears to approach a constant.

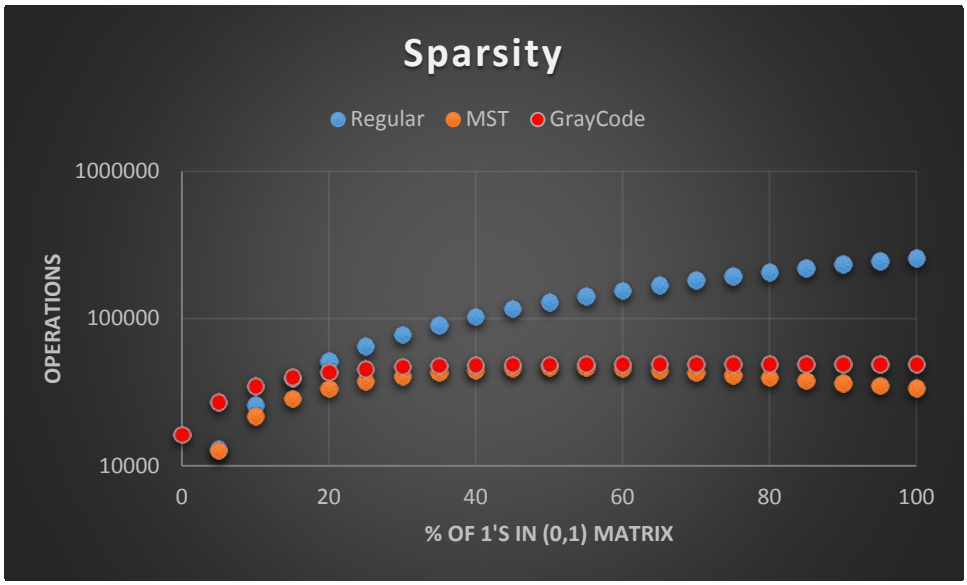

Figure 7.3.1: Plot of the number of operations required for each algorithm vs the number of elements in the square (0,1) matrix.

% of 1's	Regular	MST	GrayCode
0	0	0	16320
5	13016	12739	27278
10	25951	21691	34742
15	39194	28586	39974
20	51487	33405	43301
25	65341	37659	45687
30	78220	40696	47123
35	90703	43001	47918
40	103665	44836	48449
45	116959	46146	48760
50	129615	46737	48930
55	142639	46549	49031
60	155658	45742	49058
65	168850	44528	49079
70	182027	42999	49085
75	194479	41537	49085
80	207776	39765	49088
85	220703	38136	49088
90	233737	36569	49088
95	246514	35223	49088
100	259481	33766	49088

8. Future Research

- Implement and generate data from the remaining one of our algorithms, excluded from this study, that also uses the differencing method to compute the (0,1)-matrix-vector product via compression by induction on hierarchical grammars. [2]
- Implement in our algorithms optimal lengths of partitions for our given (0,1)-matrix to further reduce operation counts. [1]
- Perform further code optimization to improve run time in comparison to the general algorithm.
- Refactor the minimum-spanning-tree algorithm implementation to use the linear-complexity counting sort, as our problem generates graph edge weights as integers in the range from 1 to $|V|$. This will allow the sorting component of Kruskal's algorithm to be performed with linear complexity making the dominant time factor the time to process the edges. [10]
- Investigate and implement a hybrid algorithm that uses the most efficient algorithm based on the measured characteristics and properties of the (0,1)-matrix.
- Investigate how much of the computation for these algorithms can be performed at compile time using C++ templates.
- Generalize this method to non-(0,1)-matrices by converting any arbitrary matrix to a linear combination of (0,1)-matrices. [1]

9. Conclusion

We discussed, implemented, and tested two algorithms that compute the (0,1)-matrix-vector product which exploit our differencing method. Our tests measure the count of addition operations required for each implementation and also the count of addition operations required by the general algorithm. Both of our methods require fewer operations than the general method in these certain cases: when the (0,1)-matrices that have more rows than columns, when the matrices are taller, for relatively large matrices, and when the matrices are more dense. In general, our data show that our MST algorithm appears to require fewer operations than our Gray code algorithm. However for very large matrices (larger than we have tested) we can estimate from our data that our Gray algorithm will require fewer operations than our MST algorithm.

10. References

[1] Andrew A. Anda . *A Gray Code Mediated Data-Oblivious (0, 1)-Matrix-Vector Product Algorithm*. Conference: Proceedings of the 2005 International Conference on Scientific Computing, CSC 2005, Las Vegas, Nevada, USA, June 20-23, 2005

[2] Aaron Webb and Andrew A. Anda. *(0, 1)-Matrix-Vector Products via Compression by Induction of Hierarchical Grammars*. In Proceedings of the 38th Midwest Instruction and Computing Symposium (MICS), University of Wisconsin, Eau Claire, 2005.

[3] Andrew A. Anda. *A Bound on Matrix-* count *Vector Products for (0, 1)-Matrices via Gray Codes*. In Proceedings of the 37th Midwest Instruction and Computing Symposium (MICS), University of Minnesota, Morris, 2004.

[4] Kenneth H. Rosen, John G. Michaels, Jonathan L. Gross, Jerrold W. Grossman, and Douglas R. Shier, editors. *Handbook of Discrete and Combinatorial Mathematics*. CRC, Boca Raton FL, 2000.

[5] Sándor Dominich. *Mathematical Foundations of Information Retrieval*. Kluwer, Dordrecht, The Netherlands, 2001.

[6] Darrell A. TURKINGTON. *Matrix Calculus and Zero-One Matrices*. Cambridge University Press, New York, 2002. Statistical and econometric applications.

[7] Jeffrey D. Witthuhn and Andrew A. Anda. *(0, 1)-Matrix-Vector Product Computations via Minimum Spanning Trees*, In Proceedings of the 48th Midwest Instruction and Computing Symposium (MICS), University of North Dakota, Grand Forks, ND, 2015.

[8] Jeffrey D. Witthuhn, Implementation and code and testing details: https://github.com/jeffwitthuhn/MICS-2016

[9] Henry S. Warren. *Hacker's Delight* (2nd ed.). Addison-Wesley Professional, Boston, MA, 2012.

[10] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms* (3rd ed.). The MIT Press, 2009.