MST-Based and Optimal Grammar-Based (0,1)-Matrix-Vector Product Algorithms

Jeffrey D. Witthuhn and Andrew A. Anda
Computer Science Department
St Cloud State University
St Cloud, MN, 56301
jeffdwitt@gmail.com
aanda@stcloudstate.edu

March 20 2017

Abstract

The elements of a (0,1)-matrix are either 0 or 1. This class of matrices arise from computational problems in fields including graph theory and information retrieval. We compare our two algorithms for computing the (0,1)-matrix-vector product. After the minimum spanning tree or grammar based preprocessing phases have generated code, we show substantial reductions in both execution timings and in arithmetic operations counts.

1 Introduction

A matrix, $A \in \mathbb{R}^{m \times n}$, is a (0,1)-matrix if each of its elements, α_{ij}, is in the set $\{0, 1\}$. These types of matrices appear in problems from a variety of application areas including graph theory,[15] information retrieval,[8] and matrix calculus.[18] Matrix-vector products (MVPs) are common computational kernels in many matrix computation algorithms. Matrix eigenvalues, essential to areas including principal component analysis and spectral graph theory,[7,17] can be computed via MVPs.[5,16]

$$A = \begin{bmatrix} \alpha_{11} & \alpha_{12} & \dots & \alpha_{1n} \\ \alpha_{21} & \alpha_{22} & \dots & \alpha_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_{m1} & \alpha_{m2} & \dots & \alpha_{mn} \end{bmatrix} \quad \text{with} \quad \alpha_{ij} \in \{0, 1\}$$

We focus on computing the (0,1)-MVP more efficiently. Our previous publications on the (0,1)-MVP introduce three redundancy exploiting algorithms.[1,2,21,26,27] Our algorithms rely on the observation that many operations in the product are repetitive due to commonalities in the matrix rows. Our algorithms all require expensive precomputations such as: computing all possible result vector elements,[2,27] building a grammar from the matrix,[21] or generating a minimum spanning tree (MST) on a complete weighted graph.[26,27] The (0,1)-MVP is performed while traversing the generated data structures for a specific matrix. After the precomputation is completed, the total number of additive operations required to compute the (0,1)-MVP is often fewer than if using the conventional algorithm. Because of our algorithms respective initialization expense, our algorithms will be practical only where the matrices are reused for many additional products.

We now compare two of our redundancy exploiting algorithms. One algorithm builds a MST to minimize arithmetic operation counts.[26,27] Our other algorithm builds a hierarchical grammar to exploit commonalities within the matrix rows.[21] We compare our algorithms in terms of their complexity, number of additive operations required, and their execution time compared with the conventional MVP algorithm. We demonstrate that our algorithms reduce operation counts in most cases. We also show that our implementations are competitive in terms of execution time with use of *metaprogramming*.

2 Our Differencing MST-Based Algorithm

2.1 Our Differencing Method

The conventional MVP $Ax = y$, with $A \in \mathbb{R}^{m \times n}$, $x \in \mathbb{R}^n$, and $y \in \mathbb{R}^m$ can be computed in general with two nested loops:

$$y_i = \sum_{j=1}^{n} \alpha_{ij} x_j \quad \text{for } i = 1, 2, \ldots, m \tag{1}$$

Equation 1 is simplified in the case of a (0,1)-matrix. Each right hand side element is multiplied by 1 or 0, so we either add x_j to the sum or ignore it (Algorithm 1).

Algorithm 1 The Conventional (0,1)-Matrix-Vector Product Algorithm

Require:
$A \in \{0,1\}^{m \times n}$
$x \in \mathbb{R}^n$
Ensure:
returns the matrix-vector product, $y \in \mathbb{R}^m$

function CMVP(A, x)
1: $y \leftarrow \vec{0}_m$
2: **for** $i = 1 \ldots m$ **do:**
3: **for** $j = 1 \ldots n$ **do:**
4: **if** $A(i,j) == 1$
5: $y(i) \leftarrow y(i) + x(j)$
6: **end if**
7: **end for**
8: **end for**
9: **return** y
end function

Now, consider the difference between two elements in the product y.

$$y_i - y_k = \sum_{j=1}^{n} \alpha_{ij} x_j - \sum_{j=1}^{n} \alpha_{kj} x_j = \sum_{j=1}^{n} x_j (\alpha_{ij} - \alpha_{kj}) \tag{2}$$

And if we already have a value for y_k:

$$y_i = y_k + (y_i - y_k) = y_k + \sum_{j=1}^{n} x_j (\alpha_{ij} - \alpha_{kj}) = y_k + \sum_{j=1}^{n} x_j d_j \tag{3}$$

where $d(i,k)$ is the difference vector between rows i and k of A

Once we compute a single element y_k, each subsequent y_i can be computed as a sum of some previous value y_k with the inner product between x and $d(i,k)$. This essentially transforms y_k into y_i by adding or subtracting the corresponding elements of x from y_k where differences in rows i and k of the (0,1)-matrix exist. The number of operations required to compute y_i from y_k is the number of non-zero entries in $d(i,k)$—this is the *Hamming distance* between rows i and k of A.

The count of 1's in a matrix row or a bit string is termed its *population*. If the Hamming distance between rows i and k is less than the population of row i, then given y_k the required operation count to compute y_i is reduced by using Equation 3 rather than Equation 1. When elements in the result vector are computed by modifying previously computed elements, we term this *differencing*.

2.2 Our Differencing MST-Based Algorithm

The count of additive operations required by Equation 3 depends on the Hamming distance between the rows. We want the minimum number of operations in the (0,1)-MVP using differencing. To obtain this minimum, we find a subset of Hamming distances, which minimizes the sum of Hamming distances required to compute the (0,1)-MVP using differencing. We solve this problem by using graph theory.

A (0,1)-matrix can be represented as a complete weighted graph, where each row of the matrix is a vertex, and edges between its vertices have a weight equal to the Hamming distance between the each pair of corresponding rows. If a minimum spanning tree (MST) is computed for this complete graph, a breadth-first traversal on this MST has a *minimum total Hamming distance*.[26,27] (Algorithm 2)

Algorithm 2 Our Differencing MST-Based Algorithm for the (0,1)-Matrix-Vector Product

Require:
$A \in \{0,1\}^{m \times n}$
$x \in \mathbb{R}^n$
Ensure:
returns the matrix-vector product, $y \in \mathbb{R}^m$

function DMST(A, x)
1: $G \leftarrow buildGraph(A)$ ▷ Build the graph representation for A
2: $T \leftarrow calcMST(G)$ ▷ Calculate the MST on G
3: $R \leftarrow chooseRoot(T)$ ▷ We choose the row with smallest population
4: Compute the element in y corresponding to R.
5: Perform a breadth-first traversal of T from R where visiting each vertex, v, consists of computing the element in y corresponding to the v.
6: **return** y
end function

Calculation of the MST can be performed disjointly from the function and saved for a specific matrix. The MST can then be applied to any set of right-hand-side vectors. So our algorithm is most suited for situations where the precomputation cost is amortized over reuse of the same matrix. Examples of such cases are the Krylov subspace methods.[11]

This graph is restricted to integer weights so a MST algorithm performing linear complexity sorting can be used. In this case the time complexity of the MST calculation is linear with $\mathcal{O}(\mathcal{V} + \mathcal{E})$, where $\mathcal{V}$ is the number of vertices and $\mathcal{E}$ is the number of edges.[10] In terms of the matrix, the number of vertices is the number of rows in the matrix, m, and the number of edges in the complete graph is m^2, which is reduced to $m(m-1)/2$ by eliminating symmetries and self-loops. So the MST time complexity in terms of the number of rows in the matrix can be as low as $\mathcal{O}(m^2)$.

Algorithm 2 can be applied to vertical partitions of a matrix as well.[26,27] Partitioning allows for redundancies to be exploited more aggressively as each partition yields a separate MST.

There are several choices for the root of the MST for the calculation. We choose the vertex whose row has a minimum population. Another choice is to choose the *graph center*[22] as the root of the tree which allows for the shortest operational latency for parallel execution.

3 Our SEQUITUR-Based Product Algorithm

Our MST-based algorithm works by directly exploiting differencing and Equation 3. Our next algorithm identifies and exploits these redundancies in another way.

SEQUITUR is an *online* string compression algorithm that incrementally builds a hierarchical grammar from a stream of characters. Every pair of adjacent characters is recorded, then repeated pairs are turned into rules. A pair can consist of any combination of rules or characters. As the grammar is built, many rules are recursively and incrementally created which are two tokens long. After the input is read, any rule that is not used more than once is incorporated into another rule and eliminated. More detail on SEQUITUR can be found from Craig Nevill-Manning's work.[13] For example SEQUITUR encodes the string "abcabdabcabd" as the following grammar:

$$\begin{aligned} S &\rightarrow AA \\ A &\rightarrow BcBd \\ C &\rightarrow ab \end{aligned}$$

SEQUITUR can be used to find and eliminate redundancies in the (0,1)-MVP by compressing the (0,1)-matrix rows so that rules represent commonalities between rows. We encode a (0,1)-matrix as a stream of tokens with unique separators between rows. The use of unique separator tokens is our method for ensuring that no rule in the grammar will span row boundaries. After the matrix is processed by SEQUITUR, the remaining rules will represent segments of rows that are repeated. The (0,1)-MVP then is computed by traversing the hierarchical grammar, computing each rule once, storing the result for reuse.[21]

In Algorithm 3, the matrix A must be encoded. We do so using a sparse form, compressed-sparse-row (CSR),[4] where each nonzero element is represented as the column number to which it belongs. The unique boundary tokens are the negative row numbers. For space efficiency, sufficiently dense matrices should not be encoded in a sparse form.

Algorithm 3 Our SEQUITUR-Based Algorithm for the (0,1)-Matrix-Vector Product

Require:
$A \in \{0,1\}^{m \times n}$
$x \in \mathbb{R}^n$
Ensure:
returns the matrix-vector product, $y \in \mathbb{R}^m$

function SEQP(A, x)
$G \leftarrow generateGrammar(A)$ ▷ Encode and use SEQUITUR on A
$S \leftarrow startSymbol(G)$
$\alpha \leftarrow \{1, \ldots, n\}$
$\beta \leftarrow \{-1, \ldots, -m\}$ ▷ Separators
$i \leftarrow 1$
$y \leftarrow \vec{0}$
for each $s \in S$ **do:**
if $s \in G$ ▷ s is a rule
▷ If s has been evaluated, use its value, else recursively evaluate s as with S
$y(i) \leftarrow y(i) + evaluate(s)$
else if $s \in \alpha$ ▷ s represents a 1 in A
$y(i) \leftarrow y(i) + x(s)$
else if $s \in \beta$ ▷ s is a row boundary
$i \leftarrow i + 1$
end for
return y
end function

As with our MST-based method, calculation of the grammar can be performed disjointly from the function and saved for a specific matrix. This grammar can then be applied to any set of right-hand-side vectors. So this algorithm as well is most suited for situations where the precomputation cost is amortized over reuse of the same matrix.

The time complexity of SEQUITUR is linear in the size of the stream, $\mathcal{O}(S)$, where S is the number of elements in the stream. Using CSR, the encoded stream's size is the count of 1's in the matrix plus one for each separator token. The matrix stream encoding then has size complexity $\mathcal{O}(mn)$. So the time complexity of computing a matrix's SEQUITUR grammar is $\mathcal{O}(mn)$.

4 Modifying the Matrix

Recall that, Algorithms 2 and 3 can be practical only if the (0,1)-matrix is reused multiple times. This is because the precomputation time is large so the total amortized time savings must be greater than computation time of the MST or grammar. We can also consider using these algorithms in cases where a matrix is modified: adding a row, column, or altering elements. These modifications require a representation of the MST, or SEQUITUR data structures to be maintained so they can be modified when reflecting changes.

With respect to Algorithm 2, adding a row, column or modifying an element changes the complete graph and possibly requires updating the MST. Updating a MST can be performed with fewer operations than total recomputation.[6,9,14,19] We have not investigated how to efficiently update the grammar if the matrix is modified.

5 Implementation

Our MST-based and SEQUITUR-based algorithms are implemented primarily with C++ and the Standard Library (STL). For our MST-based algorithm, the BOOST Graph Library[3] classes and functions are used to compute the minimum spanning tree. Recall that the time complexity of the MST computation can be improved.[10] For our SEQUITUR-based algorithm, we used a templated open source implementation found on James Wilson's Github.[23,24] Our implementations make extensive use of the STL containers. Our code and data sets are archived on Github.[25]

We implemented versions of each algorithm which performed vertical partitioning. The matrix is split into w partitions. Then we compute w sub-products which when summed form the complete product y. A disjoint MST or grammar is computed for each partition.

Our implementations proved to be substantially slower than the conventional product implementation because our algorithms require traversing and manipulating large complicated data structures such as linked lists and queues, but they suffice for counting arithmetic operations. We were able to eliminate this repeated overhead for matrix reuse by employing an additional metaprogramming technique which adds one more step to the precomputation by generating new source code that is a set of optimal sums. This step performs the MST or grammar traversal once, and records each operation required to compute the (0,1)-MVP into a new C++ source file to perform the product. Since we are using a compiled language, metaprogramming requires separate compilation. Whereas, if we were using an interpreted language such as MATLAB[12] we would be able to write to a string which we could then execute dynamically. This generated file is then compiled and run to perform the product with multiple right-hand sides.

Operations from Algorithm 2 are a series of assignments, additions, and subtractions that represent a breadth-first traversal of the now implicit MST. Operations from Algorithm 3 compute and reuse values corresponding to rules of the now implicit grammar.

6 Benchmarking

To analyze our algorithms we count the additive operations required for each. We vary the size, density (ratio of 1 to 0 counts in the matrix), and the number of equal vertical partitions in the matrix. During execution, the additive operation counts and precomputation time are logged. We compare our implementations of Algorithms 1, 2, and 3.

We performed each measurement 25 times with random matrices and operand vectors. The average is used as the final value and a standard deviation is computed. We measured the operation count, precomputation time, and execution timings for our metaprogramming generated code.

Our first test uses a 128×128 (0,1)-matrix and increases density from 1% to 96%. In our second test we vary the size of random $n \times n$ square matrices and let n go from 32 to 512; this is done for density levels of 10%, 25%, 50%, 75% and 90%. Our third test again uses a 128×128 (0,1)-matrix and varies the number of equal vertical partitions in the matrix. Our algorithms are used separately on each partition and the result vectors are combined. This is done for density levels of 10%, 25%, 50%, 75% and 90%.

For execution time benchmarks, we compute 1000 iterations of the *power iteration* method,[5] and perform this for varying densities between 1% and 96%. Again each value is the average of 25 tests. To get comparable benchmarks for each algorithm, our code-generation method is used to write out and compile the raw operations.

7 Results

Figure 1 shows our results of our first test, varying the density of the (0,1)-matrix for each of our three algorithms. We plot the operation count vs the density of the matrix. The conventional product algorithm operation counts grow linearly as expected with the number of 1's in the matrix. More interestingly, the operation counts of both Algorithm 2 and 3 do not grow significantly after attaining 20% density and are close to constant. So the operation savings of both algorithms then is dominant for denser matrices. Our MST-based algorithm yields more operation savings than our SEQUITUR-based algorithm in these cases.

For our second test, we varied the size of the matrix at different density levels. Our results are displayed in Figure 2. Here for each density level, a plot of operation count vs the number of rows is displayed on a logarithmic scale. We observe that the operation counts grow similarly for each algorithm, however, as the density increases, the operation savings is greater. For very dense matrices, operation counts differ by multiple orders of magnitude. For example, the large 90% dense matrices in Figure 2 require around 2^{18} operations for the conventional algorithm while our algorithms require about 2^{14} or 2^{15} operations respectively.

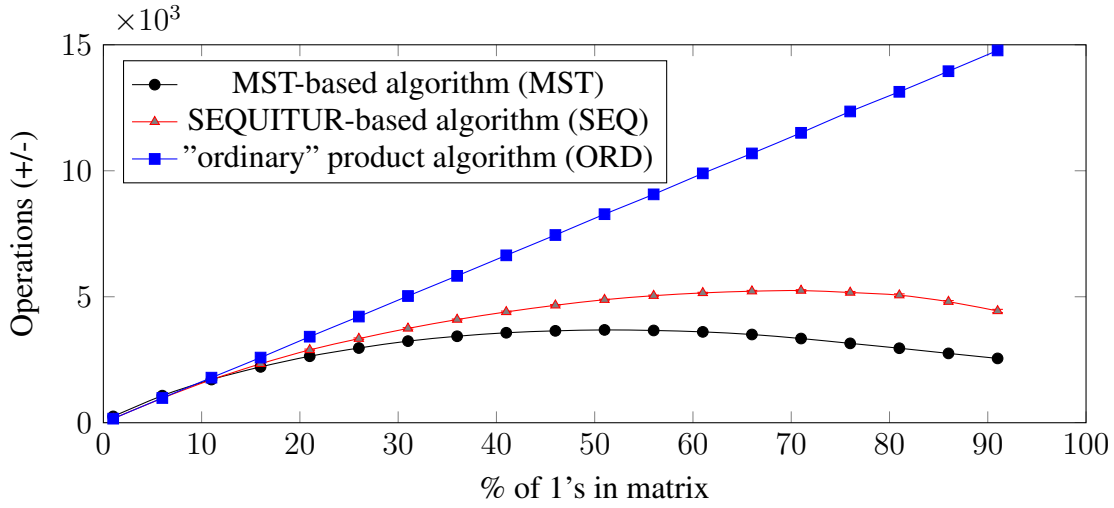

Figure 1: *Additive operation counts vs matrix density for each algorithm with* 128×128 *matrices*

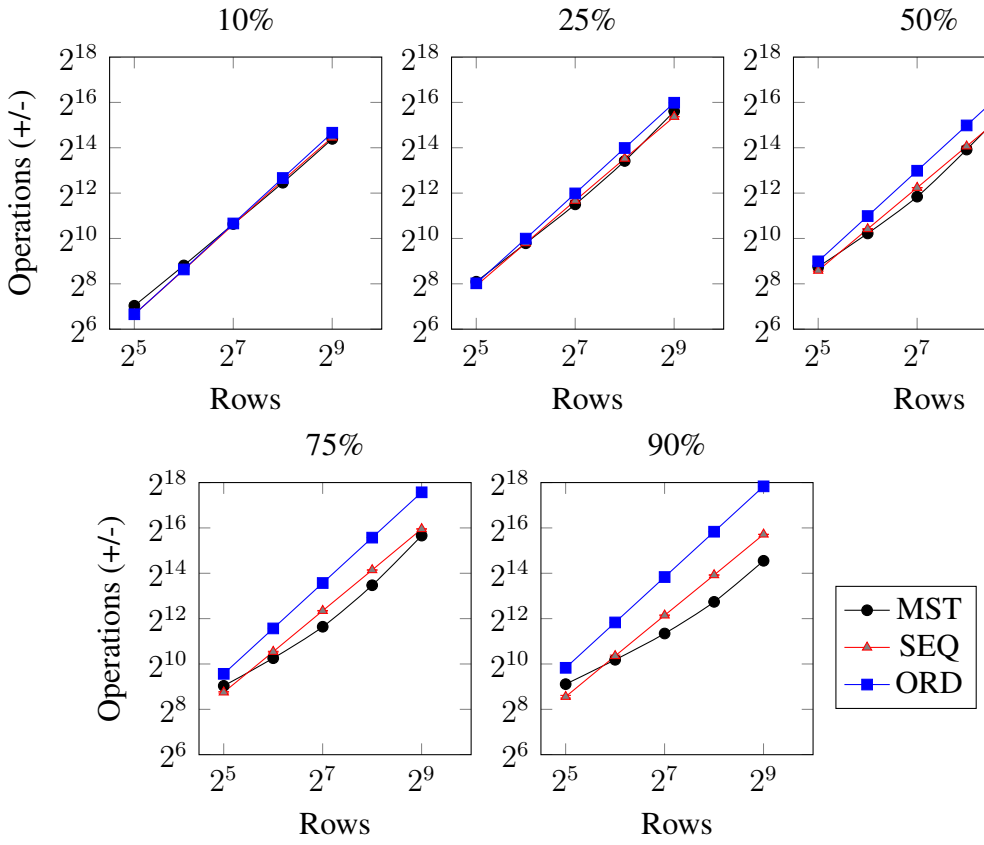

Figure 2: *Additive operation count for each algorithm vs the size of the matrices.*

Results from varying the number of partitions are shown in Figure 3. The conventional algorithm is not partitioned, and it is used as a near constant baseline. The total operations required by our SEQUITUR-based algorithm remains relatively constant. However our

MST-based algorithm benefits greatly from the partitioning in each case. After a certain number of partitions, in this case about 8 for a 128×128 matrix, our MST-based algorithm no longer benefits from additional partitions as the cost of combining the partitions becomes dominant. This happens for both algorithms at 2^6 partitions where each partition is only 2-bits in width. As the density of the matrices increases, the effect on partitioning is similar in each case. Our results show that the break even points where the matrices stop benefiting from further partitioning vary.

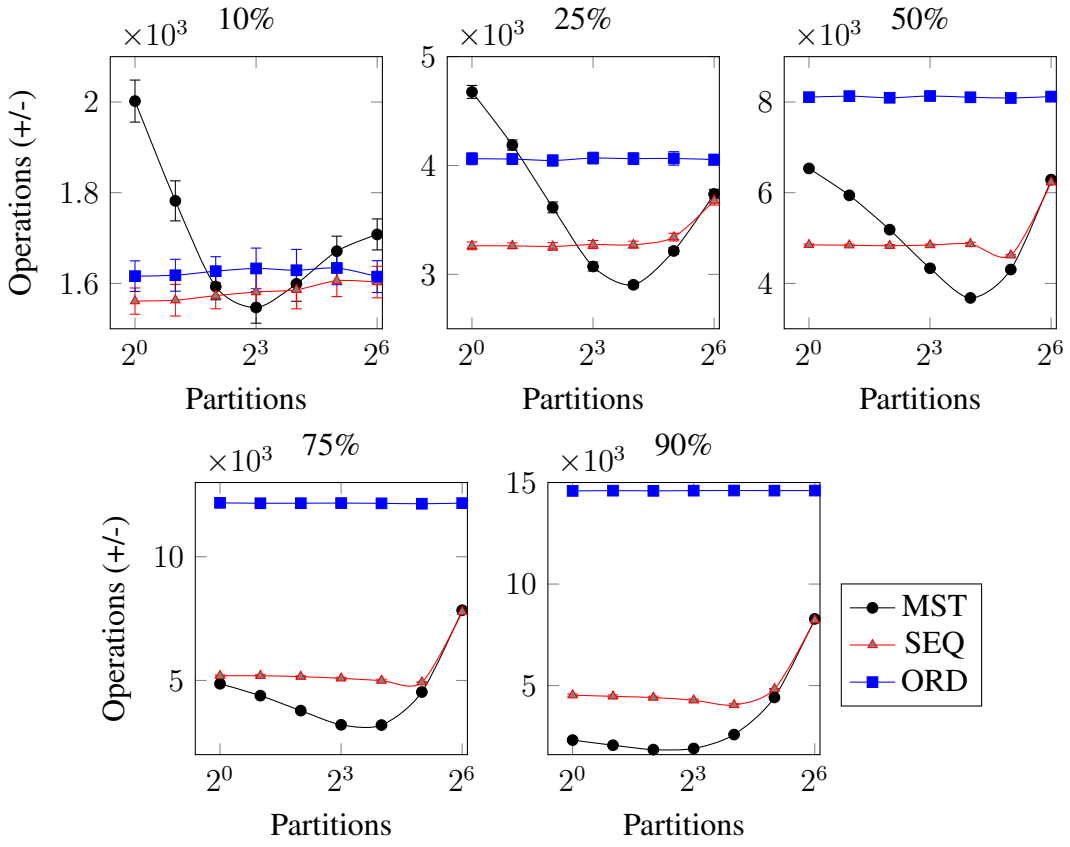

Figure 3: *Additive operation count for each algorithm vs the number of partitions of a* 128×128 *matrix*

As our algorithms are not optimized, the precomputation times for our algorithms are substantial. However, it is useful to look at how the precomputation time increases as we increase the size of the matrix. Figure 4 shows precomputation timing data for the same runs as in Figure 2. The time complexity for the build times for both algorithms appear to be near quadratic in the number of rows. The build time for our SEQUITUR-based algorithm is significantly lower than our MST-based algorithm in all cases. Our results show that as matrix densities increase the differences between the build times decrease.

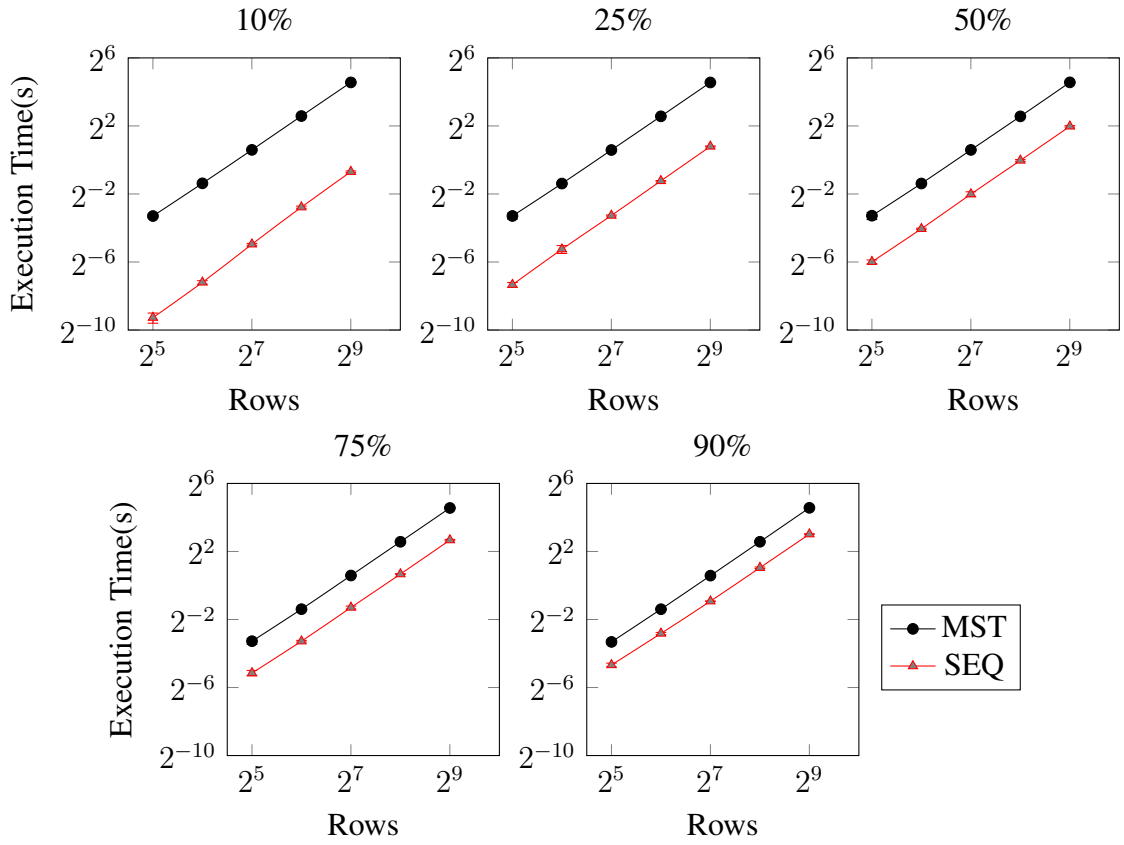

Figure 4: *Execution times for the precomputation stage of each algorithm vs the size of the matrices.*

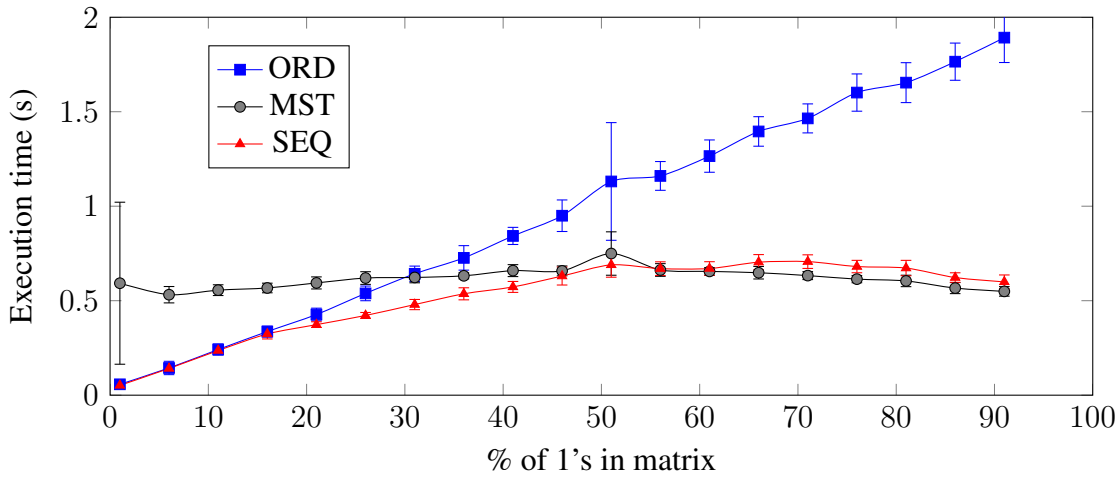

Figure 5: *Execution times for the C++ code-generating implementations for square matrices of varying densities. Note: the vertical error bars represent a standard deviation.*

Our execution time benchmarks for the metaprogramming generated code are shown in Figure 5. This measures the average CPU time required to compute 1000 power iterations

with the metaprogramming generated code as the density is varied (cf. Figure 1). For sparse matrices, there is no speed benefits, however for dense matrices, (greater than 40%), the execution time is competitive.

8 Conclusions

We described, analyzed, and tested our two families of redundancy-exploiting algorithms to compute the (0,1)-matrix-vector products. Our algorithms include a relatively expensive precomputation stage for each (0,1)-matrix. Our experimental results show that, after the precomputation stage, our algorithms reduce significantly the total additive operation count required for computing a (0,1)-matrix-vector product.

Our results show that our algorithm are most viable for dense matrices, and the operation savings increase as matrix sizes increase. For our MST-based algorithm, our results show that vertical partitioning of the matrices yields further reductions in additive operation counts at the expense of additional precomputation work. Our SEQUITUR-based algorithm results are seemingly indifferent to the partitionings. However, partitioning is useful for both algorithms to facilitate coarse-grained parallel execution. The precomputation times increases near quadratically with an increase the number of rows in the matrix for both algorithms using square matrices. We have demonstrated an implementation that outperforms the conventional method in terms of execution time after the precomputations are performed for dense matrices. This shows that our two algorithms are potentially competitive.

8.1 Future Work

- Optimizing the build time for our MST-based algorithm. Including implementing a linear-time complexity MST algorithm[10] and finding more efficient code for computing the population counts and Hamming distances.[20]
- Optimizing the build time for our SEQUITUR-based algorithm.
- Implementations which compute each partition in parallel.
- Determine and implement optimal partitioning break-even points with respect to for both parallel and additive operation count efficiency.
- Investigate which components of our algorithms can be performed at compile time via C++ templates.
- Generalize our algorithms to non-(0,1)-matrices as arbitrary matrices can be represented as a linear combination of (0,1)-matrices plus a remainder
- Investigate techniques for updating our MST and SEQUITUR data structures and metaprogramming generated C++ code for adding rows, columns, and modifying individual elements.[1]

References

[1] ANDA, A. A. A gray code mediated data-oblivious (0, 1)-matrix-vector product algorithm. *Proceedings of The 2005 International Conference on Scientific Computing, CSC 2005* (Las Vegas, Nevada, USA 2005).

[2] ANDA, A. A. A bound on matrix-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).

[3] BOOST. Boost graph library, March 2017. `http://www.boost.org/libs/graph/2016`.

[4] BULU, A., FINEMAN, J. T., FRIGO, M., GILBERT, J. R., AND LEISERSON, C. E. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *IN SPAA* (2009), pp. 233–244.

[5] BURDEN, R. L., AND FAIRES, J. D. *Numerical Analysis 9th Edition*. Cengage Learning, 2011.

[6] CATTANEO, G., FARUOLO, P., PETRILLO, U. F., AND ITALIANO, G. Maintaining dynamic minimum spanning trees: An experimental study. *Discrete Applied Mathematics* (2010).

[7] CHUNG, F. R. K. *Lectures on Spectral Graph Theory*. University of Pennsylvania, Philadelphia, 2017. Accessed: 2017-03-18. (Archived by WebCite at `http://www.webcitation.org/6p3r5odk5`).

[8] DOMINICH, S. Mathematical foundations of information retrievalatics. *Kluwer* (Dordrecht, The Netherlands, 2001).

[9] EPPSTEIN, D. Offline algorithms for dynamic minimum spanning tree problems. *Lecture Notes in Computer Science, vol 519. Springer, Berlin, Heidelberge* (Algorithms and Data Structures. WADS 1991).

[10] FREDMAN, M. L., AND WILLARD, D. E. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *Journal of Computer and System Sciences* (1994).

[11] GUTKNECHT, M. H. *A Brief Introduction to Krylov Space Methods for Linear Systems*. Springer, 2007, pp. 54–62.

[12] MATHWORKS. Eval, 2017. `https://www.mathworks.com/help/matlab/ref/eval.html` Accessed: 2017-03-18. (Archived by WebCite at `http://www.webcitation.org/6p5YVmyn1`).

[13] NEVILL-MANNING, C. G., AND WITTEN, I. H. Identifying hierarchical structure in sequences: A linear-time algorithm. *Journal of Artificial Intelligence Research* (1997).

[14] RAJA, S. Minimum spanning tree - changing edge weights, June 1, 2016. URL Accessed: 2017-03-18. (Archived by WebCite at http://www.webcitation.org/6p3wT9uPx).

[15] ROSEN, K. H., MICHAELS, J. G., GROSS, J. L., GROSSMAN, J. W., AND SHIER, D. R. Handbook of discrete and combinatorial mathematics. *CRC* (Boca Raton FL, 2000).

[16] SAUER, T. *Numerical Analysis 2nd Edition*. Pearson, 2012.

[17] TIPPING, M. E., AND BISHOP, C. M. Probabilistic principal component analysis. *Cambridge University Press. Statistical and Econometric Applications* (1999).

[18] TURKINGTON, D. A. Matrix calculus and zero-one matrices. *Cambridge University Press. Statistical and Econometric Applications* (New York, 2001).

[19] VARMAN, P., AND DOSHI, K. An efficient parallel algorithm for updating minimum spanning trees. *Theoretical Computer Science Volume 58* (1988).

[20] WARREN, H. S. *Hacker's Delight*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[21] WEBB, A., AND ANDA, A. A. (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).

[22] WEISSTEIN, E. W. Graph center. *From MathWorld–A Wolfram Web Resource.* (2017). http://mathworld.wolfram.com/GraphCenter.html.

[23] WILSON, J. Open source templated sequitur implementation. github.com/jsdw/cpp-sequitur.

[24] WILSON, J. Sequitur: a templated c++ implementation, 2015. http://unbui.lt/#!/post/sequitur-cpp/ Accessed: 2017-03-18. (Archived by WebCite at http://www.webcitation.org/6p4FpfAUz).

[25] WITTHUHN, J., AND ANDA, A. Redundancy expointing (0,1)-matrix-vector product algorithm implementations and testing, March 2017. http://github.com/jeffwitthuhn/MICS2017.

[26] WITTHUHN, J. D., AND ANDA, A. A. (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, 2015).

[27] WITTHUHN, J. D., AND ANDA, A. A. Comparison of (0,1)-matrix-vector product difference-based algorithms. *In Proceedings of the 49th Midwest Instruction and Computing Symposium (MICS)* (University of Northern Iowa, Cedar Falls, 2016).