A Parallel Projection Method for Metric Constrained Optimization

Cameron Ruggles * Nate Veldt † David F. Gleich ‡

Abstract

Many clustering applications in machine learning and data mining rely on solving *metric-constrained* optimization problems. These problems are characterized by $O(n^3)$ constraints that enforce triangle inequalities on distance variables associated with n objects in a large dataset. Despite its usefulness, metric-constrained optimization is challenging in practice due to the cubic number of constraints and the high-memory requirements of standard optimization software. Recent work has shown that iterative projection methods are able to solve metric-constrained optimization problems on a much larger scale than was previously possible, thanks to their comparatively low memory requirement. However, the major limitation of projection methods is their slow convergence rate. In this paper we present a parallel projection method for metric-constrained optimization which allows us to speed up the convergence rate in practice. The key to our approach is a new parallel execution schedule that allows us to perform projections at multiple metric constraints simultaneously without any conflicts or locking of variables. We illustrate the effectiveness of this execution schedule by implementing and testing a parallel projection method for solving the metric-constrained linear programming relaxation of correlation clustering. We show numerous experimental results on problems involving up to 2.9 trillion constraints.

1 Introduction

Many tasks in machine learning and data mining, in particular problems related to clustering, rely on learning pairwise distance scores between objects in a dataset of n objects. One particular paradigm for learning distances, that arises in a number of different contexts, is to set up a convex optimization problem involving $O(n^2)$ distance variables and $O(n^3)$ *metric constraints* which enforce triangle inequalities on the variables. This approach has been applied to problems in sensor location [19, 20], metric learning [6, 7], metric nearness [8,14,15], and joint clustering of image segmentations [21, 39]. Metric-constrained optimization problems also frequently arise as convex relaxations of NP-hard graph clustering objectives. A common approach to developing approximation algorithms for these clustering objectives is to first solve a convex relaxation and then round the solution to produce a provably good output clustering [11, 28, 38].

The constraint set of metric-constrained optimization problems may differ slightly depending on the application. However, the common factor among all of these problems is that they involve a cubic number of constraints of the form $x_{ij} \leq x_{ik} + x_{jk}$ where (i, j, k) is a triplet of points in some dataset and x_{ij} is a distance score between two objects i and j. This leads to an extremely large, yet very sparse and carefully structured constraint matrix. Given the size of this constraint matrix and the corresponding memory requirement, it is often not possible to solve these problems on anything but very small datasets when using standard optimization software. In recent work [37] we showed how to overcome the memory bottleneck by applying memory-efficient iterative projection methods, which provide a way to solve these problems on a much larger scale than was previously possible. Unfortunately, although projection methods come with a significantly decreased memory footprint, they are also known to exhibit very slow convergence rates. In particular, the best known results are obtained by specifically applying Dykstra's projection method [16], which is known to have a only a linear convergence rate [17].

Given the slow convergence rate of Dykstra's method, a natural question to ask is whether one can improve its performance using parallelism. There does in fact already exist a parallel version of Dykstra's method [26], which performs independent projections at all constraints of a problem simultaneously, and then averages the results to obtain the next iterate. However, this procedure is ineffective for metric-constrained optimization, since averaging over the extremely large constraint set leads to changes that are so small no meaningful progress is made from one iteration to the next. As another challenge, we note that many of the most commonly studied metric-constrained optimization problems are linear programs [1, 11, 19, 21, 38, 39].

*Dept. of Computer Science, Purdue University. Email: *cruggles@purdue.edu*.

†Center for Applied Mathematics, Cornell University. Email: *nveldt@cornell.edu*.

‡Dept. of Computer Science, Purdue University. Email: *dgleich@purdue.edu*. Supported by DARPA SIMPLEX and NSF award CCF-1149756, IIS-1422918, IIS-1546488, CCF093937 and the Sloan Foundation.

Linear programming is P-complete, and therefore it is widely assumed that they are hard to parallelize in general. Although there is some existing work on parallel interior point solvers, these only solve LPs approximately and involve highly ill-conditioned problems nearby the solution. Thus, finding meaningful ways to solve metric-constrained optimization problems in a way that is both fast and memory efficient possess several significant challenges.

In this work we take a first step in parallelizing projection methods for metric-constrained optimization. This leads to a modest but consistent reduction in running time for solving these challenging problems on a large scale. Our approach relies on the observation that when applying projection methods to metric-constrained optimization, two projection steps can be performed simultaneously and without conflict as long as the (i, j, k) triplets associated with different metric constraints share at most one index in common. Based on this, we develop a new parallel execution schedule which identifies large blocks of metric constraints that can be visited in parallel without locking variables or performing conflicting projection steps. Because Dykstra's projection methods also relies on carefully updating dual variables after each projection, we also show how to keep track of dual variables in parallel and update them at each pass through the constraint set. We demonstrate the performance of our new approach by using it to solve the linear programming relaxation of correlation clustering [5]. Solving this LP is an important first step in many theoretical approximation algorithms for correlation clustering [2, 10, 11, 34, 35, 38]. In our experiments we consistently obtain a speedup of roughly a factor 5 over the serial method using even a small number cores, and achieve a speedup of over a factor of 11 for our largest problem. Our new approach allows us to handle problems containing up to nearly 3 trillion constraints in a fraction of the time it takes the serial method.

2 Background

We use the term *metric-constrained optimization* or more simply *metric optimization* to refer to any convex optimization problem involving constraints of the form $x_{ij} \leq x_{ik} + x_{jk}$ where x_{ij} represents a distance variable between two points i and j in a large graph or dataset. Our work builds directly on previous results for solving optimization problems of this form using projection methods [8, 36, 37]. In this section we specifically consider the metric-constrained linear programming relaxation for correlation clustering and its relationship to what is known as the metric nearness problem. We will use this LP relaxation as a special case study in this paper, although the parallel approach we develop can in principle be applied to any metric optimization problem.

2.1 Metric Nearness and Correlation Clustering One key example of metric optimization is the metric nearness problem [8,36], in which one is given matrix $\boldsymbol{D} = (d_{ij})$ of dissimilarity scores between objects in a dataset. The goal is to find the matrix $\boldsymbol{X} = (x_{ij})$ whose entries satisfy the triangle inequality and for some value of p minimizes

$$||\boldsymbol{X} - \boldsymbol{D}||_p = \left(\sum_{ij} w_{ij} |x_{ij} - d_{ij}|^p \right)^{1/p}, \tag{2.1}$$

where w_{ij} is a nonnegative weight indicating the how strongly we wish x_{ij} to be similar to d_{ij}. The problem can be cast as a linear program when $p = 1$, a quadratic program when $p = 2$, and a slightly more complicated convex optimization problem for other finite values of p. One can also consider a $p = \infty$ norm version of the problem which minimizes the the maximum of $|x_{ij} - d_{ij}|$ over all pairs i, j. This can also be cast as an LP.

Metric-constrained optimization is also a key ingredient in approximation algorithms for correlation clustering [5]. In correlation clustering one is given a weighted and signed graph $G = (V, E^+, E^-, W)$. Each pair of nodes (i, j) in G defines either a positive edge $(i, j) \in E^+$ or a negative edge $(i, j) \in E^-$. The goal is to partition V in such a way that negative edges tend to link nodes between different clusters, and positive edges link nodes inside the same cluster. The problem also comes with weights $W = (w_{ij})$ where w_{ij} indicates the strength of the relationship between i and j. One formulation of the problem is to minimize the weight of mistakes, which can be cast as the following binary linear program:

$$\begin{array}{lll} \text{minimize} & \sum_{(i,j)\in E^+} w_{ij} x_{ij} + & \sum_{(i,j)\in E^-} w_{ij}(1 - x_{ij}) \\ \text{subject to} & x_{ij} \leq x_{ik} + x_{jk} & \text{for all } i, j, k \\ & x_{ij} \in \{0, 1\} & \text{for all } i, j. \end{array} \tag{2.2}$$

A positive mistake happens when two nodes with a positive edge are clustered apart ($x_{ij} = 1$), and this comes with a penalty equal to the weight w_{ij}. A negative mistake is when two nodes sharing a negative edge are clustered together, in which case the penalty is again $w_{ij} = w_{ij}(1 - x_{ij})$ since in this case $x_{ij} = 1$. We can relax (2.2) to a linear program by substituting $x_{ij} \in \{0, 1\}$ with the constraint $x_{ij} \in [0, 1]$. Solving this relaxation and then rounding the solution is a general strategy that has lead to a number of approximation algorithms for different variants of correlation clustering. For arbitrary

weights, there exists an $O(\log n)$ approximation rounding scheme [13]. When the graph is unweighted (i.e. $w_{ij} = 1$ for all pairs i, j), the best rounding scheme produces an approximation ratio near 2 [11]. Several other special weighted cases also obtain their best known approximation factor by solving the relaxation of (2.2) and rounding [2, 35, 38].

In recent work [37] we proved that the LP relaxation of (2.2) can be cast equivalently as a special case of the metric nearness problem (2.1) when $p = 1$. Specifically, given an instance of correlation clustering, define a dissimilarity score $d_{ij} = 1$ if $(i, j) \in E^-$, and set $d_{ij} = 0$ otherwise. Then the ℓ_1 metric nearness problem and the LP relaxation of correlation clustering are both equivalent to the following metric-constrained LP:

$$\begin{array}{lll} \text{minimize} & \sum_{i<j} w_{ij} f_{ij} & \\ \text{subject to} & x_{ij} \leq x_{ik} + x_{jk} & \text{for all } i, j, k \\ & x_{ij} - d_{ij} \leq f_{ij} & \text{for all } i, j \\ & d_{ij} - x_{ij} \leq f_{ij} & \text{for all } i, j. \end{array} \tag{2.3}$$

2.2 Projection Methods for Metric Optimization Although solutions to problems such as (2.1), (2.3), and other metric optimization problems are desirable from a theoretical perspective, they are challenging to solve for even modest values of n due to the cubic constraint set. Standard commercial optimization packages are typically unable to handle problems with even a few hundred nodes when the full constraint set is included, due to memory limitations. Sra et al. began to address this problem specifically for the metric nearness problem [36]. Their approach was to apply memory-efficient projection methods, which visit constraints cyclically and iteratively update variables in a manner that is proven to converge to the optimal solution.

Recently, we showed how the techniques of Sra et al. can be adapted and improved to apply more broadly to a wider range of linear and quadratic metric-constrained optimization problems [37]. These results come with new approximation guarantees for specific graph clustering objectives, and are designed to produce output solutions with better constraint satisfaction and convergence guarantees. Here we review the main background for applying Dykstra's method to metric-constrained linear programming. For details on how to apply projection methods to metric-constrained convex optimization problems that are not linear programs, we refer the reader to other work [8, 36].

Metric-Constrained Linear Programming Consider a general linear program of the form

$$\min \ \mathbf{c}^T \mathbf{x} \text{ s.t. } \boldsymbol{A}\mathbf{x} \leq \mathbf{b}. \tag{2.4}$$

Encoding a metric-constrained LP in this format can be accomplished by letting $\mathbf{x}$ encode a linearization of the distance variables x_{ij} and potentially other variables depending on the specific optimization problem. The constraint matrix $\boldsymbol{A}$ will encode metric constraints and other problem specific constraints, (e.g. the non-metric constraints $x_{ij} - d_{ij} \leq f_{ij}$ in (2.3)). Because of the metric constraints, $\boldsymbol{A}$ will be large, sparse, and very structured.

Projection methods do not apply directly to solving linear programs, so we first consider a regularized linear program

$$\min \mathbf{c}^T \mathbf{x} + \frac{\varepsilon}{2} \mathbf{x}^T \boldsymbol{W} \mathbf{x} \text{ s.t. } \boldsymbol{A}\mathbf{x} \leq \mathbf{b} \tag{2.5}$$

where ε is a positive constant and $\boldsymbol{W}$ is a positive definite diagonal matrix of weights. Both ε and $\boldsymbol{W}$ are viewed as parameters that can be chosen to control the relationship between (2.4) and (2.5). When $\boldsymbol{W}$ is the identity matrix, solving (2.5) for a small enough value of ε will output the smallest norm solution to the LP (2.4) [31]. Furthermore, our recent work provides specific details for how to set ε and $\boldsymbol{W}$ to bound the difference between the original linear program and the related quadratic program (2.5) for specific graph clustering relaxations [37].

Applying Projection Methods The quadratic program (2.5) can be solved using memory-efficient projection methods, which iteratively visit constraints and perform correction and projection steps that slowly fix constraint violations, update dual variables, and eventually converge to the unique optimal solution. Following previous work [36,37], we specifically consider Dykstra's method, which for quadratic programs is equivalent to Hildreth's method [23] and Han's method [22]. We provide pseudocode for applying this method to (2.5) in Algorithm 1.

Localized Metric Projections For a more in-depth explanation of the algorithm, we refer to our previous work [37]. The key thing to realize is that Algorithm 1 is simply Dykstra's method applied specifically to solve (2.5). Most importantly, updates of the form $\mathbf{x} := \mathbf{x} + c\boldsymbol{W}^{-1}\mathbf{a}_i$ for a constant c can be performed very quickly for metric constraints, since in this case $\mathbf{a}_i$ (the ith row of constraint matrix $\boldsymbol{A}$) has only three nonzero entries.

For illustration, we show how to perform the projection step in Algorithm 1 when $\mathbf{x} = (x_{ij})$ is a linearization of the distance variables, and $\boldsymbol{W}$ is the identity matrix (note that the projection step is unaffected by the value of ε, so we do not specify its value here). Row $\mathbf{a}$ of $\boldsymbol{A}$ and entry b of $\mathbf{b}$ encode the constraint $\mathbf{a}^T\mathbf{x} = x_{ij} - x_{ik} - x_{jk} \leq 0 = b$. For this constraint, $\mathbf{a}$ has three nonzero entries: 1, -1, and -1, corre-

Algorithm 1 Dykstra's Method for Quadratic Program (2.5)

(M, N) = number of rows and columns of $\boldsymbol{A}$ respectively
$\mathbf{y} := \mathbf{0} \in \mathbb{R}^M$ (dual variables)
$\mathbf{x} := -\frac{1}{\varepsilon}\boldsymbol{W}^{-1}\mathbf{c}$, $k := 0$
while *not converged* **do**
$k := k + 1$
(Visit next constraint): $i := (k-1) \bmod M + 1$
(Correction step): $\mathbf{x} := \mathbf{x} + y_i(\frac{1}{\varepsilon}\boldsymbol{W}^{-1}\mathbf{a}_i)$
where $\mathbf{a}_i$ is the ith row of $\boldsymbol{A}$
(Projection step): $\mathbf{x} := \mathbf{x} - \theta_i^+(\frac{1}{\varepsilon}\boldsymbol{W}^{-1}\mathbf{a}_i)$
where $\theta_i^+ = \varepsilon\frac{\max\{\mathbf{a}_i^T\mathbf{x}-b_i,0\}}{\mathbf{a}_i^T\boldsymbol{W}^{-1}\mathbf{a}_i}$
(Dual variable update): $y_i := \theta_i^+ \geq 0$

sponding to the locations of x_{ij}, x_{ik}, and x_{jk} in $\mathbf{x}$. If $\delta = \mathbf{a}^T\mathbf{x} = x_{ij} - x_{ik} - x_{jk} \leq 0$, then the constraint is already satisfied, and $\max\{\mathbf{a}^T\mathbf{x} - b, 0\} = 0$, thus there is no update to the vector $\mathbf{x}$. If $\delta > 0$, then $\max\{\mathbf{a}^T\mathbf{x} - b, 0\} = \delta = x_{ij} - x_{ik} - x_{jk}$, and $\mathbf{a}^T\mathbf{a} = 3$. The projection step in Algorithm 1 updates only three entries of $\mathbf{x}$:

$$x_{ij} \leftarrow x_{ij} - \delta/3, \qquad x_{ik} \leftarrow x_{ik} + \delta/3, \qquad x_{jk} \leftarrow x_{jk} + \delta/3.$$

The correction step in Algorithm 1, which is necessary to guarantee convergence, can be performed in a similar localized manner.

Slow Convergence Rate The decreased memory footprint of Dykstra's method makes it possible to solve metric constrained problems on a much larger scale than was previously possible [37]. However, this method converges very slowly, given that the convergence rate for Dykstra's method applied to quadratic programs is only linear [17].

3 Parallel Metric Constrained Optimization

The primary contribution of our work is to show how to parallelize projection methods specifically for metric-constrained optimization problems. We accomplish this by showing how to visit multiple metric constraints at once and perform a large number of projections simultaneously without conflicts or locking variables.

3.1 Performing Two Simultaneous Projections

To develop intuition for our approach, we consider two sets of triplets $t_1 = (a, b, c)$ and $t_2 = (i, j, k)$, where the indices within each triplet are distinct, but some indices may be the same across both triplets. Each of these triplets is associated with three metric constraints, thus three projection steps that must be performed during one pass through the constraint set using Dykstra's method.

Performing projections associate with triplet t_1 involves variables $\{x_{ab}, x_{bc}, x_{ac}\}$. Similarly, triplet t_2 is associated with variables $\{x_{ij}, x_{jk}, x_{ik}\}$. Note that if these triplets share two indices (e.g. $a = i$ and $b = j$), then we cannot perform projections at both constrains in parallel without conflict, since one variable (e.g. $x_{ab} = x_{ij}$) would be updated by both projections. However, if t_1 and t_2 share at most one index in common, then $\{x_{ab}, x_{bc}, x_{ac}, x_{ij}, x_{jk}, x_{ik}\}$ are all distinct and we can perform projection steps in Dykstra's iteration at t_1 and t_2 at the same time. Our goal is to use this observation to develop a parallel execution schedule that will allow us to visit a large number of metric constraints at once and perform simultaneous projection steps without conflicts. Because this amounts simply to a re-ordering of constraints in a way that is more easily parallelizable, this will not affect the convergence guarantees of Dykstra's method.

3.2 New Ordering for Visiting Triplets

We abstract the process of visiting metric constraints to the process of enumerating triplets of the form (i, j, k) where $1 \leq i < j < k \leq n$. Let T denote this set of ordered triplets. Each fixed ordered triplet will be associated with three different metric constraints, and hence three different projection steps, that we assume will be handled by the same processor in a parallelized projection method.

Based on our intuitive observation in the previous section, we wish to group the triplets in T into subsets $S_1, S_2, \ldots, S_\ell$ in such a way that $S_u \cap S_v = \emptyset, \bigcup_{u=1}^{\ell} S_u = T$, and such that any two triplets in different sets will share at most one index in common. If we can accomplish this, then we can assign each set S_u to a different thread or processor. The work done at each processor (i.e. each set of triplets) will be then completely independent of work performed at other sets by different processors.

To accomplish this we define sets of triplets in which the smallest and largest indices are fixed values i, k such that $k \geq i + 2$. We specifically define

$$S_{i,k} = \{(i, j, k) \in T : k \geq i + 2, 1 \leq i < j < k \leq n\}$$

which includes all triplets with i as the smallest index and k as the largest index. In Figure 2 we show a grid of (i, k) pairs associated with $S_{i,k}$ sets. Observe that drawing lines along downward-sloping diagonals of this grid highlights a large number of sets that can be processed simultaneously, i.e any two triplets taken from different sets along the diagonal will share at most one common index. Note that for a fixed x, z satisfying $z \geq x + 2$, the diagonals in Figure 2 are made up of

```
Input: integer n
Output: triplets (i,j,k) s.t. 1 ≤ i < j < k ≤ n
x = 1 // First double loop fixes x
for z = n : −1 : 3
  g = floor((z − x − 2)/2)
  for c = 0 : 1 : g
    i = x + c
    k = z − c
    List triplets in S_{i,k}
    // Different sets of triplets
    // share at most one index
  end
end
z = n // Second double loop fixes z
for x = 2 : 1 : (n − 2)
  g = floor((z − x − 2)/2)
  for c = 0 : 1 : g
    i = x + c
    k = z − c
   List triplets in S_{i,k}
   // Different sets of triplets
   // share at most one index
  end
end
```

Figure 1: Loops for listing all triplets in T. The inner loops can be perfectly parallelized when we are performing projections at metric constraints of the form $x_{ij} \leq x_{ik} + x_{jk}$, since any two triplets from different $S_{i,k}$ sets will share at most one triplet index in common.

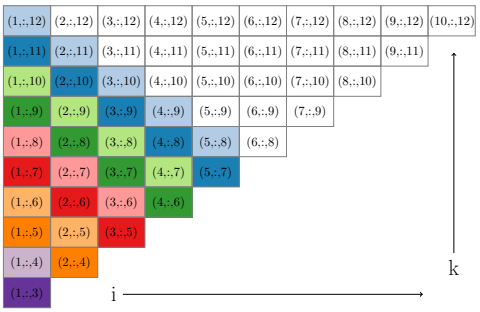

Figure 2: We illustrate how to parallelize visiting metric constraints when $n = 12$. Rectangle $(i, :, k)$ in the grid represents the set of all ordered triplets $S_{i,k}$ for which i is the first index and k is the last index. When performing projections at metric constraints, sets of the same color can be processed simultaneously without conflict. Each color corresponds to all the $S_{i,k}$ sets that are listed by the inner loop of the first double loop in Figure 1, for a fixed z. The second double loop in Figure 1 corresponds to the upper triangular portion of the grid (in white).

sets of the form $S_{x+c,z-c}$ for $c = 0, 1, 2, \ldots, \lfloor \frac{z-x-2}{2} \rfloor$. The upper bound $c \leq \frac{z-x-2}{2}$ is chosen to guarantee that $2 + (x + c) \leq (z - c)$, implying that $S_{x+c,z-c}$ contains at least one ordered triplet from T. Based on this observation, in Figure 1 we show how to loop through all triplets in T in such a way that the inner loop iterates through sets $S_{x+c,z-c}$ that can be processed simultaneously. The code in Figure 1 contains two double loops for visiting $S_{i,k}$ sets. The first double loop handles the main diagonal of sets in Figure 2 and everything above below it, and the second double loop iterates through the sets above the main diagonal. Equivalently, for the first double loop we set $x = 1$ and $z \leq n$, and then in the outer loop decrement z by one at each step. The second double loop fixes $z = n$ and iterates through all possibilities $x \in [2, n - 2]$ in the outer loop.

3.3 Load Balancing and Tiled Triplet Assignment One issue we must address with our listing of triplets in Figure 1 is the load balance. There is variability in both the number of triplet sets in the parallelized inner loop (i.e. the number of entries along a given diagonal in Figure 2), as well as the size of each triplet set within the same inner loop (i.e. different entries in the same diagonal of the grid in Figure 2). For example, when $x = 1$ and for a fixed z, there are $(z-3)/2+1$ sets, and these sets have variable size $z-2(c+1)$ for different values of c ranging from 0 to $(z - 3)/2$). We begin by noting that the vast majority of the triplets are visited for values of $z = O(n)$. Secondly, we assume that the number of threads or processors p we use when iterating over sets is significantly smaller than the problems size n, and we can assign sets of triplets to processors in a way that will not be too imbalanced. For a diagonal defined by fixed x, z values, there are $\lfloor \frac{z-x-2}{2} \rfloor$ sets of triplets. If we assigned the first group of n/p triplet sets to the first processor, and in general assigned the rth group of n/p triplet sets to the rth processor, this would indeed lead to a significant imbalance. However, in practice, we balance the load much more effectively by assigning the rth set $S_{i,k}$ to processor $r \mod p$. In this way each processor is responsible for different triplet sets with a range of different sizes, for an overall load that is roughly balanced. We illustrate this load balanced assignment in Figure 3.

We also improve our parallel execution schedule by implementing a tiled approach to triplet set assignment for better cache efficiency when accessing distance variable in the matrix $\boldsymbol{X} = (x_{ij})$. This is inspired by previous work on tiled matrix multiplication, though it differs slightly in order to apply to enumerating triplets specifically for metric constrained projection methods. In short, this tiled strategy corresponds to substituting the diagonal pattern in Figure 2 with the block diagonal pattern shown in Figure 4. In more detail, for a tile

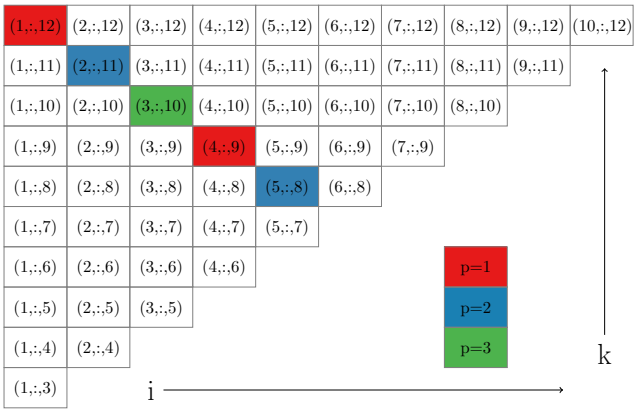

Figure 3: In theory, all sets of triplets along the same diagonal could be handled simultaneously if they were assigned to different processor. In practice, the number processors p is much smaller than n. We balance the load among processors by assigning the rth set on a diagonal to processor $r \mod p$ (where processor 0 and processor p are the same). Here we illustrate the assignment of triplet sets to processors along the main diagonal when $n = 12$ and $p = 3$.

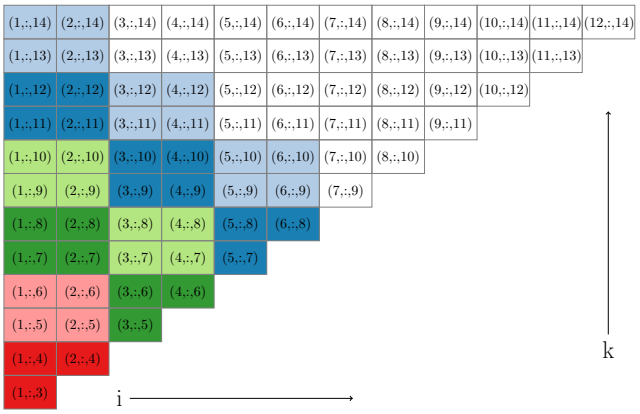

Figure 4: We illustrate the tiling approach for visiting triplet sets when $n = 14$ and tile size $b = 2$. Triplets in different tiles along the same diagonal can be visited simultaneously without conflict when assigned to different processors. Organizing sets $S_{i,k}$ into $b \times b$ tiles and carefully iterating through middles indices j allows for better cache efficiency when accessing variables in the matrix $\boldsymbol{X}$.

size b, each tile is defined by a fixed (x, z) pair, and is made up of all $S_{i,k}$ sets where $i \in \{x, x+1 \ldots, x+b-1\}$ and $k \in \{z, z-1, \ldots, z-b+1\}$. Much like in the untiled case, we note that different tiles of the same color in Figure 4 can be visited by different processors at the same time without conflict. That is, different processors will access completely independent parts of the matrix $\boldsymbol{X}$. When assigning p processors to tiles along a block diagonal, we assign the rth tile to processor $r \mod p$, generalizing the strategy outlined in Figure 3 for the untiled case.

Each tile, which is defined by a fixed pair (x, z) and a tile size b, is associated with b choices for the smallest index i and b choices for the largest index k. The processor assigned to this tile must then iterate through all valid middle indices j and perform projections corresponding to triplets of the form (i, j, k). One approach for doing this would be to consider each (i, k) pair in turn and iterate through all values of j from $j = i + 1$ to $j = k - 1$. However, for better cache efficiency, we instead split the full range of possible j values from $x + 1$ to $z - 1$ into subintervals that are also of length b. This gives us a sequence of $b \times b \times b$ cubes of (i, j, k) values, each associated with entries x_{ij}, x_{ik}, and x_{jk} from $\boldsymbol{X}$. Within each of these cubes, we iterate through triplets in a way that maximizes column locality (assuming $\boldsymbol{X}$ is stored in column major format), before moving on to the next cube. We give a simple illustration of this in Figure 5. Depending on the values of x,z, and b, we will of course not be able to organize all triplets into perfect $b \times b \times b$ cubes of (i, j, k) triplets satisfying $i < j < k$. In practice however for large values of n and $b \ll n$, this approach will still provide a balanced and cache efficient way to access variables in $\boldsymbol{X}$.

3.4 Storing Dual Variables for Parallel Computations In addition to updating primal variables (x_{ij}), Dykstra's method requires we keep track of dual variables as a part of correction step (line 7 in Algorithm 1) that is necessary to guarantee convergence to the optimal solution. Specifically, for each metric constraint associated with a triplet (i, j, k), there is a corresponding dual variable y_{ijk} that is updated during each visit to a constraint. This variable is only nonzero if in the previous pass through the constraints, there was a non-trivial projection step (i.e. the entries x_{ij}, x_{ik}, x_{jk} changed). For serial projection methods for metric optimization, the metric constraints are visited in the same order in every pass through the constraint set [37]. This makes it possible to effectively query dual variables from an array that stores tuples of the form (t_{ijk}, y_{ijk}) where t_{ijk} is a unique index associated with a metric constraint and y_{ijk} is the dual variable. For memory-efficiency, these tuples are only stored for nonzero dual variables: $y_{ijk} > 0$. Because the serial version visits constraints in the same order each round, the array is always traversed in the same order. At each step, the method

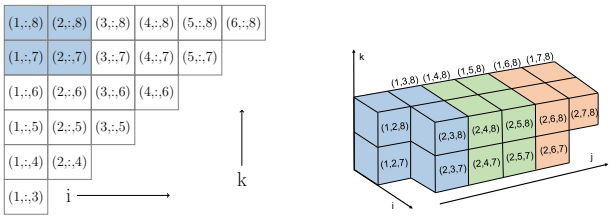

Figure 5: Each tile in the (i, k) grid is assigned to different processor. The blue section in the left image corresponds to a single tile when $b = 2$ and $n = 8$. When a processor is assigned such a tile, it also separates options for the middle index j into groups of b choices. The processor then iterates through triplets (i, j, k) in cubes of size $b \times b \times b$ in a way that maximizes column locality and cache efficiency when accessing entries of the form x_{ij}, x_{ik}, x_{jk} from the distance matrix $\boldsymbol{X}$. The right figure highlights this process specifically for the tile in the figure on the left. The processor handles each colored section in turn. Some of the cubes are incomplete, since triplets such as $(2, 2, 8)$, $(2, 2, 7)$, and $(2, 7, 7)$ do not satisfy $i < j < k$.

can access each necessary dual variable in $O(1)$ time by maintaining a pointer in the array to the next known triplet (i, j, k) associated with a nonzero dual variable. In this way the method can access the necessary dual variables in $O(1)$ time.

Using our new parallel execution schedule, the triplets are no longer visited in a deterministic fashion, so a new approach is necessary. Fortunately, our approach is designed in such a way that each triplet (i, j, k) is always visited by the same processor during each different pass through the constraints. Furthermore, even though globally the triplets are not visited in a deterministic fashion, each individual processor visits its assigned triplets in the same deterministic order at every iteration. Therefore, we can maintain dual variables efficiently by assigning an array to each processor, allowing the processor to keep track of the next triplet it will visit that will require a non-trivial correction step. Thus the main difference between the serial and parallel versions is simply that the latter requires we maintain an array for each processor rather than a single array for storing all dual variables. Accessing dual variables is therefore still performed in $O(1)$ time at each projection step and the theoretical memory complexity is the same.

4 Experiments

We demonstrate the power of our new parallel approach to metric-constrained optimization by using it to solve the linear programming relaxation of correlation clustering on several large instances. We find that using even a modest number of cores consistently leads to a speed up of roughly a factor 5, and up to a speedup over a factor 10 on the largest problem, which involves nearly 3 trillion constraints.

4.1 Implementation Details We implement a solver for the metric-constrained LP relaxation of correlation clustering by incorporating our new parallel execution schedule into our previous serial framework [37]. We use the Julia programming language, using its support for threaded computations to parallelize the inner loops of our new approach to iterating through index triplets. Our code is available publicly online at `https://github.com/camruggles/ParallelDykstras`. In our experiments we compare against our previous serial projection methods, available at at `https://github.com/nveldt/MetricOptimization`.

4.2 Problem Construction and Datasets To test our parallel solver we construct several large instances of correlation clustering from undirected graphs following the approach of Wang et al. [40], and including a slight modification applied in previous work [37]. In short, given a graph $G = (V, E)$, we compute a signed and weighted edge between each pair of nodes (i, j) by computing the Jaccard index between the nodes (which is always nonnegative) and applying a non-linear function to obtain a signed value that either represents similarity or dissimilarity between the nodes. We then offset these scores by $\pm\varepsilon$ for a small $\varepsilon > 0$. This last step ensures the result will be an instance of correlation clustering in which each pair of nodes possesses a nonzero weight and a sign. Partitioning the original graph G using the correlation clustering objective can be used as a way to perform community detection on G. For our purposes, this construction leads to a dense instance of correlation clustering that serves as a good benchmark for solving the LP relaxation of correlation clustering on a large scale.

We apply this procedure to five undirected and unsigned graphs: the graph *power* from the Newman group of matrices in the SuiteSparse Matrix Collection [12, 41], and four collaboration networks available from the SNAP repository [29,30]: *ca-GrQc*, *ca-HepTh*, *ca-HepPh*, and *ca-AstroPh*. We take the largest connected component of each graph before converting it into an instance of correlation clustering. The LP relaxation of the correlation clustering instance corresponding to the largest graph (*ca-AstroPh*) has over 160 million variables and 2.9 trillion constraints.

4.3 Machine Specifications and Computing Environment Our experiments were almost exclusively performed on a computer with 4 16-core Intel Xeon E7-8867 v3 processors. For one experiment on the largest graph, in which we wanted to run a large number of cores, we used a machine with 8 24-core 2.7 GHz Intel Xeon Platinum 8168 processors. For our experiments we did not utilize exclusive access to the computers. Thus, the reported runtimes vary depending on whether there were other users simultaneously using the machine at the same time as our experiments. This emulates the natural and realistic performance that may be expected in settings such as Amazon EC2, with multiple shared VMs on a single machine.

4.4 The Effect of Reordering Constraints Dykstra's method is guaranteed to converge regardless of the order in which the constraints are visited. However, we found that in practice the number of iterations required to solve a problem to within a fixed tolerance for constraint satisfaction and duality gap did vary depending on the constraint ordering. In some cases, the standard serial ordering led to a smaller overall iteration count, though in many other cases the iteration count was lower for our new approach for visiting triplets. Given the variability between problem instances, in our experiments we focus simply on the time it takes to complete a fixed number of iterations of Dykstra's method. In this way, we are always comparing the time it takes to visit and perform a step of Dykstra's method at each individual constraint exactly C times for some fixed integer C.

4.5 Results In Table 1, we report results for running our parallel code on all five graphs using 8, 16, and 32 cores. The runtime for 1 core comes from applying the previous serial version of the algorithm [37]. For the largest graph we additionally run our new algorithm using 64 cores, which is the only experiment for which we used the machine with 8 24-core processors. For each graph we report the time it took in seconds to run Dykstra's method for 20 iterations, using a tile size of $b = 40$. Running our method with 8 cores is consistently 4-5 times faster than the serial implementation. All of the parallelism arises from processing elements along a diagonal at the same time. This means that as the algorithm departs from the main diagonal, there is less work in processing a diagonal, but also less possibility for parallel execution to be helpful. This causes a load imbalance that prevents idealized parallel scaling. We do continue to see performance gains as we increase the number of cores used, which leads to a speedup of over a factor ten on our largest graph.

Table 1: Results for parallel Dykstra's method in solving the metric-constrained LP relaxation of correlation clustering.

Graph	Constr.	Cores	Time (s)	Speedup
ca-GrQc	3.6×10^{10}	1	2632	1
$n = 4158$		8	562	4.68
		16	429	6.14
		32	358	7.35
Power	6.0×10^{10}	1	4521	1
$n = 4941$		8	890	5.08
		16	696	6.50
		32	576	7.85
ca-HepTh	3.2×10^{11}	1	19826	1
$n = 8638$		8	4682	4.23
		16	3252	6.10
		32	2603	7.62
ca-HepPh	7.0×10^{11}	1	47309	1
$n = 11204$		8	10313	4.59
		16	7066	6.70
		32	5889	8.03
ca-AstroPh	2.9×10^{12}	1	187045	1
$n = 17903$		8	40146	4.66
		16	35397	5.28
		32	24374	7.67
		64	16325	11.46

In Figure 6 we display results specifically on *ca-HepPh* using a wider range of core counts. We see the performance of our method increase sharply at first and slowly level off as we increase the number of cores. Finally, we observe what happens as we vary the tile size and keep the number of cores fixed. Figure 7 illustrates the algorithm's performance on *ca-GrQc* as we vary tile size from 5 to 50 and keep the number of cores fixed at 16. The curve in the figure shows the speedup over the serial implementation, which is above a factor 5 for all tile sizes except 5. The performance peaks just above a factor 6 speedup for a tile size of 25, and slowly begins to decrease after this point.

5 Related Work

Our work is related to a number of different areas in machine learning, optimization, graph theory, and matrix computations.

Metric-Optimization and Projection Methods The parallel algorithms we have develop build directly on previous serial techniques for metric constrained optimization. These optimization problems arise in algorithm design for graph clustering prob-

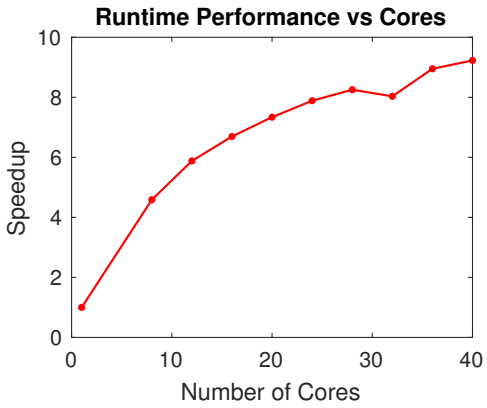

Figure 6: We display results for varying the number of cores on *ca-HepPh* for a fixed tile size of 40. Results are displayed for 1 core, and then for 8 to 40 cores in increments of 4.

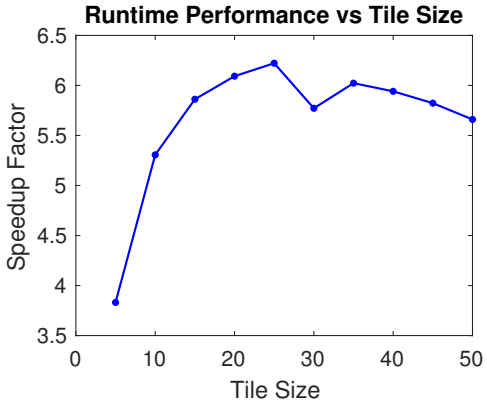

Figure 7: When we vary the tile size, performance climbs to a peak and then slowly decreases if we increase the tile size too much. Results are shown here for graph *ca-GrQc* when tile size ranges from 5 to 50 in increments of 5. The number of threads is fixed at 16.

lems [1, 10, 28, 38], image segmentation [21, 39], sensor location [19, 20], metric nearness [8], and metric learning [6,7]. Sra et al. [36] were the first to apply projection methods for metric-constrained optimization, by using Dykstra's method [16] to solve different variants of metric nearness. In recent work [37], we developed improved techniques for applying this method more broadly to linear programming relaxations of graph clustering objectives.

Graph Coloring The parallel execution schedule we have develop in this paper is related to a number of different graph coloring problems. Consider a graph in which every node corresponds to a triplet (i, j, k), and edges connect nodes (i.e. triplets) if they share two indices in common. Coloring the nodes in this graph in such a way that no adjacent nodes share the same color is equivalent to partitioning all triplets into disjoint sets such that triplets within the same set can be processed simultaneously by our projection methods. Another approach would be to instead assign each pair ij (i.e. each entry in the distance matrix $\boldsymbol{X} = (x_{ij})$) to a node in a hypergraph, and for every triplet of indices (i, j, k) define a hyperedge of the form (ij, jk, ik). Then the problem of finding sets of triplets to process simultaneous is equivalent to edge coloring in 3-uniform hypergraphs [32]. In general, graph coloring arises frequently as a way to determine potential areas for concurrency when completing a given task in parallel. We refer to several helpful resources on coloring algorithms for parallel and multithreaded computations [9, 18, 33].

Block Matrix Multiplication Our tiled approach to triplet enumeration is inspired by techniques for block matrix multiplication, which also involves doubly indexed blocks of data and computational steps corresponding to a triplet of indices. Specifically, multiplying the ij block of a matrix $\boldsymbol{A}$ with the jk block of another matrix $\boldsymbol{B}$ is a step in block matrix-matrix multiplication ($\boldsymbol{AB} = \boldsymbol{C}$), that can be indexed by a triplet (i, j, k). Our tiled triplet enumeration procedure is related to research on communication bounds for dense matrix multiplication. The pioneering work of Hong and Kung [24] proved a lower bound on the communication necessary to move data between slow and fast memory in matrix multiplication. Irony, Toledo, and Tiskin [25] later extended this result to distributed parallel computations. The state of the art numerical linear algebra software package LAPACK [3] determines block sizes automatically for efficient matrix-matrix computations. For an in-depth overview of communication-avoiding and cache efficient algorithms for numerical linear algebra, we refer to the work of Ballard et al. [4] and Knight [27] (see in particular Section 5.5).

6 Discussion and Future Work

We have developed a new approach for parallel projection methods for metric-constrained optimization problems. These problems arise in a number of applications but are challenging to solve due to their massive constraint set. There are also notable challenges in implementing parallel solvers for these methods. Parallel solvers for projection methods do exist, but they rely on averaging out a large number of independent projections equal to the number of constraints, and for metric-constrained optimization problems this is so many constraints that averaging the results in this way makes no meaningful progress in each iteration towards a solution. In our work we begin to overcome these chal-

lenges by demonstrating how to identify large sets of projections at metric constraints that can be performed simultaneously as a part of the standard Dykstra iterate without affecting one another. Our approach allows us to obtain consistent runtime improvements of roughly a factor 5 using even a small number of cores, and even better speedups on large problems. Our result demonstrate the challenges involved in efficiently solving metric-constrained optimization problems, and provide a first step in parallel techniques to solve them. In the future we will continue to explore other ways to visit metric constraints in parallel, as well as other techniques that can leverage the highly structured constraint matrix of metric-constrained optimization problems in order to obtain speedups that are even more significant.

References

[1] G. Agarwal and D. Kempe. Modularity-maximizing graph communities via mathematical programming. *The European Physical Journal B*, 66(3):409–418, Dec 2008.

[2] Nir Ailon, Moses Charikar, and Alantha Newman. Aggregating inconsistent information: ranking and clustering. *Journal of the ACM (JACM)*, 55(5):23, 2008.

[3] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[4] Grey Ballard, Erin Carson, James Demmel, Mark Hoemmen, Nicholas Knight, and Oded Schwartz. Communication lower bounds and optimal algorithms for numerical linear algebra. *Acta Numerica*, 23:1–155, 2014.

[5] Nikhil Bansal, Avrim Blum, and Shuchi Chawla. Correlation clustering. *Machine Learning*, 56:89–113, 2004.

[6] D. Batra, R. Sukthankar, and T. Chen. Semi-supervised clustering via learnt codeword distances. In *Proceedings of the British Machine Vision Conference*, BMVA 2008, pages 90.1–90.10. BMVA Press, 2008. doi:10.5244/C.22.90.

[7] Arijit Biswas. *Semi-supervised and Active Image Clustering with Pairwise Constraints from Humans*. PhD thesis, University of Maryland, College Park, 2014.

[8] Justin Brickell, Inderjit S. Dhillon, Suvrit Sra, and Joel A. Tropp. The metric nearness problem. *SIAM Journal on Matrix Analysis and Applications*, 30(1):375–396, 2008.

[9] Ümit V Çatalyürek, John Feo, Assefaw H Gebremedhin, Mahantesh Halappanavar, and Alex Pothen. Graph coloring algorithms for multi-core and massively multithreaded architectures. *Parallel Computing*, 38(10-11):576–594, 2012.

[10] Moses Charikar, Venkatesan Guruswami, and Anthony Wirth. Clustering with qualitative information. *Journal of Computer and System Sciences*, 71(3):360 – 383, 2005. Learning Theory 2003.

[11] Shuchi Chawla, Konstantin Makarychev, Tselil Schramm, and Grigory Yaroslavtsev. Near optimal LP rounding algorithm for correlation clustering on complete and complete k-partite graphs. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing*, STOC 2015, pages 219–228. ACM, 2015.

[12] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, December 2011.

[13] Erik D. Demaine, Dotan Emanuel, Amos Fiat, and Nicole Immorlica. Correlation clustering in general weighted graphs. *Theoretical Computer Science*, 361(2):172 – 187, 2006. Approximation and Online Algorithms.

[14] Inderjit S Dhillon, Suvrit Sra, and Joel A Tropp. The metric nearness problems with applications. Technical report, 2003.

[15] Inderjit S. Dhillon, Suvrit Sra, and Joel A. Tropp. Triangle fixing algorithms for the metric nearness problem. In *Advances in Neural Information Processing Systems 17*, NIPS 2004, pages 361–368, Cambridge, MA, USA, 2004. MIT Press.

[16] Richard L Dykstra. An algorithm for restricted least squares regression. *Journal of the American Statistical Association*, 78(384):837–842, 1983.

[17] R. Escalante and M. Raydan. *Alternating Projection Methods*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 2011.

[18] Assefaw H. Gebremedhin, Duc Nguyen, Md. Mostofa Ali Patwary, and Alex Pothen. Colpack: Software for graph coloring and related problems in scientific computing. *ACM Trans. Math. Softw.*, 40(1):1:1–1:31, October 2013.

[19] Camillo Gentile. Sensor location through linear programming with triangle inequality constraints. In *IEEE International Conference on Communications*, volume 5, pages 3192–3196. IEEE, 2005.

[20] Camillo Gentile. Distributed sensor location through linear programming with triangle inequality constraints. *IEEE transactions on wireless communications*, 6(7), 2007.

[21] D. Glasner, S. N. Vitaladevuni, and R. Basri. Contour-based joint clustering of multiple segmentations. In *Proceedings of the 2011 IEEE Conference on Computer Vision and Pattern Recognition*, CVPR 2011, pages 2385–2392, Washington, DC, USA, 2011. IEEE Computer Society.

[22] Shih-Ping Han. A successive projection method. *Mathematical Programming*, 40(1-3):1–14, 1988.

[23] Clifford Hildreth. A quadratic programming procedure. *Naval Research Logistics (NRL)*, 4(1):79–85,

1957.

[24] Jia-Wei Hong and Hsiang-Tsung Kung. I/o complexity: The red-blue pebble game. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing*, STOC '81, pages 326–333. ACM, 1981.

[25] Dror Irony, Sivan Toledo, and Alexander Tiskin. Communication lower bounds for distributed-memory matrix multiplication. *Journal of Parallel and Distributed Computing*, 64(9):1017–1026, 2004.

[26] Alfredo N Iusem and Alvaro R De Pierro. On the convergence of Han's method for convex programming with quadratic objective. *Mathematical Programming*, 52(1-3):265–284, 1991.

[27] Nicholas Sullender Knight. *Communication-Optimal Loop Nests*. PhD thesis, UC Berkeley, 2015.

[28] Tom Leighton and Satish Rao. Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms. *Journal of the ACM (JACM)*, 46(6):787–832, November 1999.

[29] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graph evolution: Densification and shrinking diameters. *ACM Trans. Knowl. Discov. Data*, 1(1), March 2007.

[30] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[31] O. L. Mangasarian. Normal solutions of linear programs. *Mathematical Programming at Oberwolfach II*, pages 206–216, 1984.

[32] Pawełll Obszarski and Andrzej Jastrzębski. Edge-coloring of 3-uniform hypergraphs. *Discrete Applied Mathematics*, 217:48 – 52, 2017. Combinatorial Optimization: Theory, Computation, and Applications.

[33] Md. Mostofa Ali Patwary, Assefaw H. Gebremedhin, and Alex Pothen. New multithreaded ordering and coloring algorithms for multicore architectures. In Emmanuel Jeannot, Raymond Namyst, and Jean Roman, editors, *Euro-Par 2011 Parallel Processing*, pages 250–262, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[34] Gregory J. Puleo and Olgica Milenkovic. Correlation clustering with constrained cluster sizes and extended weights bounds. *SIAM Journal on Optimization*, 25(3):1857–1872, 2015.

[35] Gregory J. Puleo and Olgica Milenkovic. Correlation clustering and biclustering with locally bounded errors. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning*, ICML 2016, pages 869–877. JMLR.org, 2016.

[36] Suvrit Sra, Joel Tropp, and Inderjit S. Dhillon. Triangle fixing algorithms for the metric nearness problem. In L. K. Saul, Y. Weiss, and L. Bottou, editors, *Advances in Neural Information Processing Systems*, pages 361–368. MIT Press, 2005.

[37] Nate Veldt, David Gleich, Anthony Wirth, and James Saunderson. A projection method for metric-constrained optimization. *arXiv preprint arXiv:1806.01678*, 2018.

[38] Nate Veldt, David F. Gleich, and Anthony Wirth. A correlation clustering framework for community detection. In *Proceedings of the 2018 World Wide Web Conference*, WWW 2018, pages 439–448, 2018.

[39] S. N. Vitaladevuni and R. Basri. Co-clustering of image segments using convex optimization applied to em neuronal reconstruction. In *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, CVPR 2010, pages 2203–2210, June 2010.

[40] Yubo Wang, Linli Xu, Yucheng Chen, and Hao Wang. A scalable approach for general correlation clustering. In *International Conference on Advanced Data Mining and Applications*, ADMA 2013, pages 13–24. Springer, 2013.

[41] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of 'small-world'networks. *Nature*, 393:440 EP –, 06 1998.