Exact sparse matrix partitioning

Bachelor Thesis

June 16, 2016

T. E. Knigge

supervised by
Prof. Dr. R. H. Bisseling

Utrecht University

Contents

1 Introduction

Matrix-vector multiplication is a common elementary operation performed in numerical algorithms, particularly *sparse* matrix-vector multiplication (SpMV). We call an $n \times m$ matrix with N nonzeros sparse if $N \ll nm$, that is, if a large number of the elements in the matrix are zero. These zeros can of course be ignored during the multiplication, meaning that if we only store the nonzeros of the matrix, we can perform the multiplication in $\mathcal{O}(N)$ time rather than $\mathcal{O}(nm)$ time.

While the operation itself is very simple, in some situations we encounter instances that are so large that we cannot perform the multiplication on a single machine in the desired timeframe. To solve this, we can distribute the problem over multiple processors: each processor is given a subset of the nonzeros of the matrix to multiply. Since the processors can work in parallel, we would want to distribute the nonzeros in a roughly equal manner over the processors, i.e. if we have k processors, each processor should get around $\frac{N}{k}$ nonzeros.

However, an even distribution is not enough to guarantee an optimal running time: the processors also have to communicate about the input vector and the output vector. Consider the situation displayed in Figure 1: we are given a 5×5 matrix A and a vector x, and are asked to find $y = Ax$. The white squares in the matrix represent zeros, the remaining squares are nonzero (for the purpose of distributing the nonzeros we do not care about their exact value, only that they are nonzero, hence they are displayed as colored squares). In particular, we have $k = 2$ processors, with the red nonzeros going to processor 0 and the blue nonzeros to processor 1.

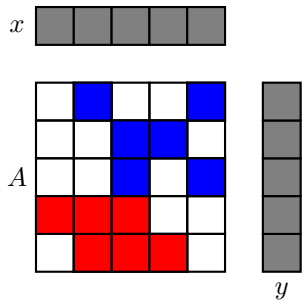

Figure 1: A multiplication spread over two processors.

Each processor is given ownership of some of the elements in the input vector x. Based on the partition, we should probably give processor 0 ownership of x_1 and processor 1 ownership of x_5, since these are the only processors that need those elements (respectively), and then no communication between the processors would be necessary. However, columns 2 through 4 each 'contain' two distinct processors, so whichever processor is given ownership of such an element will have to communicate it to the other processor.

In general, if a column c contains λ_c distinct processors, the communication volume induced by this column is proportional to $\lambda_c - 1$.

When each processor is done multiplying its nonzeros with the relevant parts of x, the results have to be added up into an output vector y. We have a similar problem as before: different processors may each have values that have to be added to the same position in y. We again give each processor ownership of some of the values of y, and with λ_r distinct processors in some row r, a communication volume proportional to $\lambda_r - 1$ is needed.

Bringing this together, the total volume of some partition of the nonzeros of A is simply the sum of the communication volume in each of the rows and columns. More formally, let $B \subseteq \{1, 2, \ldots, n\} \times \{1, 2, \ldots, m\}$ denote the nonzeros of A, then a partition of B is a series of pairwise disjoint sets $B_1, B_2, \ldots B_k \subseteq B$ with $\cup_i B_i = B$. The volume of such a partition equals:

$$VOL(B_1, B_2, \ldots, B_k) = \sum_{\text{column } c} (\lambda_c - 1) + \sum_{\text{row } r} (\lambda_r - 1)$$

Looking at the partition in Figure 1, we find that it has volume 3. However, far worse partitions can be made – the partition in Figure 2 has volume 8[1].

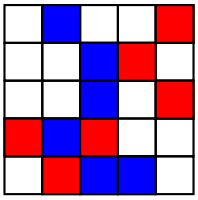

Figure 2: Another partition of the matrix in Figure 1.

To perform our multiplication as fast as possible, we want to minimize the communication volume. Thus, the sparse matrix partitioning problem can be formulated as follows: given the nonzeros $B \subseteq \{1, 2, \ldots, n\} \times \{1, 2, \ldots, m\}$ of a matrix, a number of processors $k \geq 2$ and a load imbalance $\epsilon \geq 0$, find a partition $B_1, B_2, \ldots, B_k$ such that:

$$|B_i| \leq (1 + \epsilon)\frac{|B|}{k}$$

$$VOL(B_1, B_2, \ldots, B_k) \text{ is minimized}$$

Here, the load imbalance ϵ can be set to a positive value to allow a small imbalance in the relative sizes of the partition. This flexibility is useful when, for example, k does not divide $|B|$ – with $\epsilon = 0$, no solution would exist.

This thesis can be split into two large parts: first, in section 2 we will study the sparse matrix partitioning problem using tools from the field of theoretical computer science. Specifically, not a lot of research has been done into the complexity of this problem. We prove for the first time that the sparse matrix partitioning problem is $\mathcal{NP}$-Complete even when the number of processors is fixed to 2.

In section 3 we will build on existing work by Pelt & Bisseling in [9] to solve the sparse matrix partitioning problem exactly using branch and bound. We introduce new methods for deriving lower bounds and discuss some results.

[1] A partition of volume 9 exists, this one is left as an exercise for the reader.

2 Problem Complexity

In this section we will study the complexity of the matrix partitioning problem, and take a closer look at its relation to other problems. We will first review some prerequisites in the field of complexity theory. A more extensive introduction may be found in e.g. [5].

When talking about the complexity of computational problems, we usually formulate them as decision problems. When working with a minimization problem, such as the matrix partitioning problem we are considering in this thesis, the decision problem is usually of the form 'can we achieve a solution of cost V or less?'.

Recall that $\mathcal{P}$ is the class of all decision problems that can be solved in polynomial time, and that $\mathcal{NP}$ is the class of all decision problems for which solutions can be verified in polynomial time. Inside $\mathcal{NP}$ we have an equivalence class of problems called the '$\mathcal{NP}$-Complete problems'. These problems have the property that there exists a polynomial time algorithm solving them if and only if $\mathcal{P} = \mathcal{NP}$.

A problem is typically proven $\mathcal{NP}$-Complete by a reduction: given an $\mathcal{NP}$-Complete problem Problem A and some other problem Problem B, we show that Problem B is $\mathcal{NP}$-Complete by proving that a polynomial time algorithm for Problem B allows us to derive a polynomial time algorithm for Problem A. In that case we say that Problem A can be reduced to Problem B in polynomial time, or Problem A $\leq_{\mathcal{P}}$ Problem B.

Additionally, for problems that contain numerical parameters we often make distinctions between problems that are *strongly* $\mathcal{NP}$-Complete and problems that are *weakly* $\mathcal{NP}$-Complete. Numerical parameters can be specified in binary notation, hence if n is the size of the input, such parameters can be of size $\mathcal{O}(2^n)$. We call a problem strongly $\mathcal{NP}$-Complete if it remains $\mathcal{NP}$-Complete even when the numerical parameters are of size $\mathcal{O}(n)$.

Another interesting question we can ask ourselves is: if we fix some parameter in the problem, does the problem still remain $\mathcal{NP}$-Complete? As an example, in the Maximum Clique problem we are given a graph (V, E) and an integer k, and are asked to find k distinct vertices, such that they are all pairwise connected by an edge (such a subset of vertices is called a k-clique). This was one of Karp's 21 original $\mathcal{NP}$-Complete problems [7]. However, if we fix k to 2, the problem can simply be reformulated as: 'does (V, E) contain an edge?', which is clearly solvable in polynomial time. We might wonder if something similar happens when we fix the k in Matrix k-partition to 2 (the smallest interesting value).

2.1 Matrix k-partition is $\mathcal{NP}$-Complete

We begin by formulating a decision variant of the matrix partitioning problem following the style used by Garey and Johnson [5].

matrix k-partition

Input: The indices of the nonzeros of a matrix $Z \subseteq \{1, \ldots, n\} \times \{1, \ldots, m\}$, an integer k, an integer V.

Question: Does there exist a partition of $Z = Z_1 \cup \cdots \cup Z_k$ such that for all $1 \leq i < j \leq k$ we have $|Z_i| = |Z_j|$ and $Z_i \cap Z_j = \emptyset$, and such that the volume of the partition is less than or equal to V, i.e. $\text{VOL}(Z_1, \ldots, Z_k) \leq V$?

We will now show that the Matrix k-partition problem is $\mathcal{NP}$-Complete by a reduction from 3-partition. This reduction mimics the idea used in [1] to prove the $\mathcal{NP}$-Completeness of a vertex partitioning problem. The 3-partition problem is defined as follows:

3-partition

Input: A multiset S of $n = 3m$ positive integers $\{a_1, \ldots, a_n\}$ such that for some integer B we have $\sum_{i=1}^{n} a_i = mB$, and for any $1 \leq i \leq n$ we have $B/4 < a_i < B/2$.

Question: Does there exist a partition of S into triples $S_1, \ldots, S_m$ such that the numbers in each triple add up to B?

Remarkably, this problem is *strongly* $\mathcal{NP}$-Complete [5, SP15, p224], i.e. it is $\mathcal{NP}$-Complete even if the integers in the input (B and the a_i) are bounded by the size of the input (as opposed to many $\mathcal{NP}$-Complete problems that only require the logarithm of numbers in the input to be bounded by the size of the input).

Theorem 2.1. 3-PARTITION $\leq_{\mathcal{P}}$ MATRIX k-PARTITION

Proof. Suppose we are given an instance of the 3-PARTITION problem, i.e. numbers $a_1, \ldots, a_n$, B for some $n = 3m$. We create a matrix that has a group of a_i nonzeros for each i, such that none of the groups share a row or column, but each group itself is connected, meaning that we cannot place two nonzeros from the same group in two different sets in the partition without inducing positive volume.

To formalize this, for each a_i we define a $\lceil\sqrt{a_i}\rceil \times \lceil\sqrt{a_i}\rceil$ matrix M_i that contains exactly a_i nonzeros, for example:

$$(M_i)_{uv} = \begin{cases} 1 & \text{if } u + (v-1)\lceil\sqrt{a_i}\rceil \leq a_i \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

After this, we define a single matrix $\tilde{M}$ containing the M_i on the diagonal as submatrices, and zeros elsewhere:

$$\tilde{M} = \begin{pmatrix} M_1 & & 0 \\ & \ddots & \\ 0 & & M_n \end{pmatrix} \tag{2}$$

This construction is probably most easily explained using an image, Figure 3 displays the matrix that results from the integers $\{\,4, 3, 5, 1, 1, 2\,\}$.

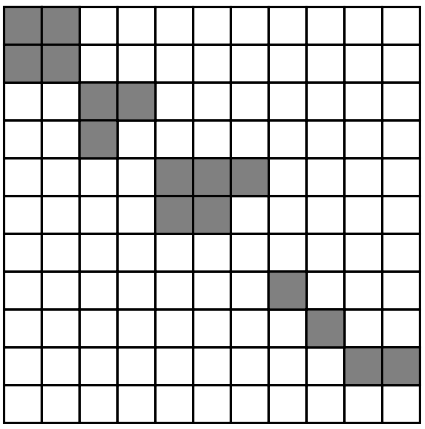

Figure 3: $\tilde{M}$ for the integers $\{\,4, 3, 5, 1, 1, 2\,\}$. Gray squares indicate a 1, white squares indicate a 0.

Note that the width and height of this matrix equal $\sum_{i=1}^{n}\lceil\sqrt{a_i}\rceil$, which is certainly bounded by a polynomial in the input size (as each of the a_i is also bounded by such a polynomial).

Now consider the following instance of the MATRIX k-PARTITION problem: let $k = m$, $V = 0$ and let Z represent the nonzeros of $\tilde{M}$. We claim that this problem has a solution if and only if the original 3-PARTITION problem does.

Indeed, suppose the 3-PARTITION instance had a solution $S_1, \ldots, S_k$. We can then partition the nonzeros of $\tilde{M}$ according to this solution, since by definition the submatrices corresponding to each S_i together contain exactly B nonzeros, and due to the positioning of the submatrices this partition has volume zero.

Conversely, suppose we find a zero volume k-partitioning of the nonzeros Z. As discussed, this means that each submatrix M_i is completely contained in a single set of the partition, and each of the k sets in the partition contains exactly B nonzeros. We can thus map the submatrices back to the original numbers a_i, where the numbers in each set sum to B. The condition $B/4 < a_i < B/2$ guarantees that each set contains exactly three integers. □

From the $\mathcal{NP}$-Completeness of 3-PARTITION we can now conclude that MATRIX k-PARTITION is $\mathcal{NP}$-Complete, even when $V = 0$.

2.2 GRAPH EDGE-BISECTION is $\mathcal{NP}$-Complete

Before proving the $\mathcal{NP}$-Completeness of bipartitioning a matrix, we need to prove the intractability of an intermediate problem that will function as a stepping stone for the $\mathcal{NP}$-Completeness of the bipartitioning problem.

The goal of this problem is to partition the edges of a graph into two equisized subsets, such that the number of vertices with incident edges from both subsets is minimized. This relates to matrix partitioning in the following manner: given a matrix, we create a vertex for each row and column, and we connect two vertices only if their corresponding row and column intersect. Then, partitioning the nonzeros of the matrix is equivalent to partitioning the edges of the new graph (the astute reader will notice that this graph is always bipartite, so there is no immediate 'bijection' between problem instances, but we will look at this in more detail in subsection 2.3). We again start off by formulating a decision variant of the problem:

GRAPH EDGE-BISECTION

Input: An undirected graph (V, E), an integer M.

Question: Does there exist a partition of $E = E_1 \cup E_2$ such that $|E_1| = |E_2|$, $E_1 \cap E_2 = \emptyset$, and $\left|\left(\bigcup E_1\right) \cap \left(\bigcup E_2\right)\right| \leq M$?[2]

For our reduction, we use a very similar problem which is commonly called GRAPH BISECTION, although it was first proven $\mathcal{NP}$-Complete in [6] under the name MINIMUM CUT INTO EQUAL-SIZED SUBSETS. As the name suggests, this problem is very similar to GRAPH EDGE-BISECTION:

GRAPH BISECTION

Input: An undirected graph (V, E), an integer M

Question: Does there exist a partition of $V = V_1 \cup V_2$ such that $|V_1| = |V_2|$, such that $V_1 \cap V_2 = \emptyset$ and such that $|\{\, \{u, v\} \mid u \in V_1, v \in V_2 \,\}| \leq M$?

Informally, we are asked to split the vertices of a graph into two equisized subsets while minimizing the number of edges that touch vertices from both groups. Although these two problems appear more similar than the two we encountered in subsection 2.1, the reduction ends up being quite involved. We will first give a short sketch of the reduction.

Suppose we are given an instance of the GRAPH BISECTION problem, that is, an undirected graph (V, E) and an integer M. We will try to encode this problem instance into an instance of the GRAPH EDGE-BISECTION problem, whose solution we can then convert back into a solution for the GRAPH BISECTION problem.

We construct this new instance by replacing each vertex in V by a sufficiently large clique (we will formalize 'sufficient' later), and for every edge in E, we connect the cliques corresponding to its endpoints by merging two (previously unmerged) vertices (one from each clique) into a single vertex (see Figure 4 for details). We then find a solution of the GRAPH EDGE-BISECTION problem on this new graph (with the same M), and then interpret the coloring of the edges of the new graph into a coloring of the vertices in the old graph.

[2] We typically denote an edge from v_1 to v_2 as the set $\{v_1, v_2\}$, so $\bigcup E_1$ is precisely the collection of vertices touched by edges in E_1.

Once again, this process is more easily made understandable using an image, the process is shown in Figure 4.

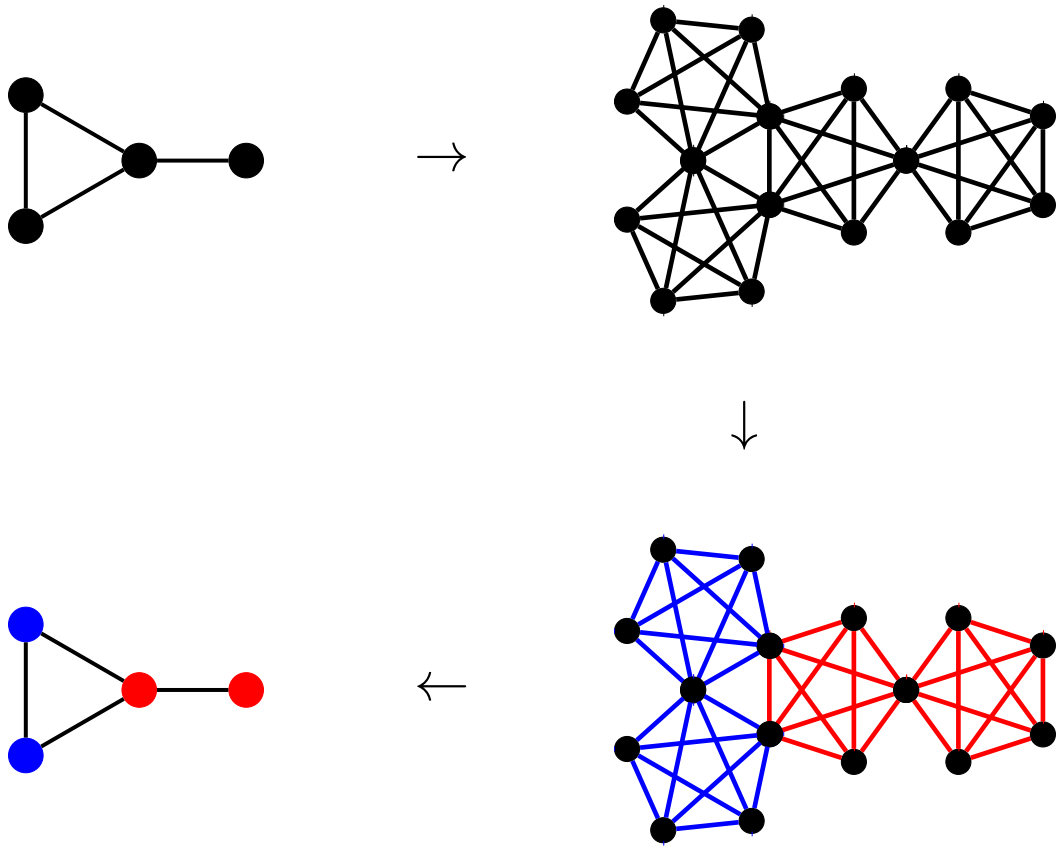

Figure 4: Solving Graph Bisection using Graph edge-bisection .

One thing that probably needs clarifying is why we need to merge the cliques in distinct vertices. The idea behind this can be seen in Figure 4: if the edges of each clique are colored in a single color, then an increase in volume of exactly 1 is attained when two cliques of different colors share a vertex - precisely the cost attained when we color their vertices in the original graph with different colors.

We will now give a formal reduction. First of all, the size K of each of the cliques should satisfy a few lower bounds. One clear lower bound is that we need $K \geq 2$, otherwise there are no edges to partition. Another important requirement is that $K \geq |E|$: one vertex in (V, E) may have up to $|E|$ outgoing edges, and since each edge requires us to merge a vertex from the clique to a different clique, we may need $|E|$ distinct vertices in a clique.

There is a third lower bound whose use will become apparent later: $K \geq 4 + 2|V|\binom{|E|}{2}$. Since we can safely assume that $|V| \geq 1$, this bound supersedes both of the preceding bounds. We can now formalize the conversion under the name *clique expansion*:

Definition 2.2. *Given a graph (V, E), let $K = 4 + 2|V|\binom{|E|}{2}$, then we define the clique expansion of (V, E) as a new graph (V', E') which is the disjoint union of $|V|$ copies of the complete graph on K vertices.*

Additionally, labelling the edges in E as $e_1, e_2, \ldots, e_{|E|}$, for each edge $e_i = \{u, v\}$ $(1 \leq i \leq |E|)$ we merge the i-th vertex of the clique representing u with the i-th vertex of the clique representing v.

From now on, we will denote the original graph by (V, E) and its *clique expansion* as (V', E'). Additionally, denote the optimal cut cost of the Graph Bisection problem on

(V, E) as $\text{GB}(V, E)$ and the optimal volume of the GRAPH EDGE-BISECTION problem on (V', E') as $\text{GEB}(V', E')$.

Proposition 2.3. *The size of the clique expansion (V', E') of a graph (V, E) is polynomial in the size of (V, E).*

Proof. For any graph we have $E \leq V^2$, then from the formula in Definition 2.2 we get $K = \mathcal{O}(|V|^5)$ and hence $|V'| = \mathcal{O}(|V|^6)$. It also follows that $|E'| = \mathcal{O}(K^2|V|) = \mathcal{O}(|V|^{11})$. □

Proposition 2.4. $\text{GEB}(V', E') \leq \text{GB}(V, E)$

Proof. This follows directly from the observation we made earlier. Suppose we have some partitioning of the vertices of (V, E), then we can transform this into a partitioning of the edges in the *clique expansion* of (V, E) by coloring the edges of a clique in (V', E') with the color of its corresponding vertex in (V, E) (i.e. we reverse the process shown by the last arrow in Figure 4). As we saw earlier, the volume increases by exactly one for each pair of cliques that are connected but have different colors - these are the vertices in the original graph that have different colors and are connected by an edge. Since each clique has the same number of edges, and exactly half the cliques are assigned to a single color, this conversion gives us a solution to the GRAPH EDGE-BISECTION problem with volume $\text{GB}(V, E)$. □

If we could prove the reverse of the above proposition, then we would be done. This would be easy if every optimal coloring of the edges of (V', E') would color each clique in a single color. Unfortunately, this is not the case.

It turns out however, that we can still deterministically associate a color with each clique. Suppose we have some **optimal** solution (a coloring of the edges) to the GRAPH EDGE-BISECTION problem on (V', E'), then:

Definition 2.5. *The dominating color of a clique in the clique-expansion is the unique color such that there exists a vertex for which all of its incident edges have the aforementioned color.*

Proposition 2.6. *The concept of a dominating color is well-defined.*

Proof. To show that each clique in the *clique expansion* has a well-defined *dominating color* requires us to show that such a color both exists and is unique.

Uniqueness is trivial: if a clique contained both a vertex with only red incident edges and a vertex with only blue incident edges, then by definition (since we are talking about a clique) these two vertices are connected by an edge, but this edge must both be blue and red - a contradiction.

As for the existence, this follows indirectly from Proposition 2.4 when we combine it with the fact that $\text{GB}(V, E)$ will not exceed $|E|$ (i.e. its optimal solution for the GRAPH BISECTION problem). As a result, neither does $\text{GEB}(V', E')$. This means that no clique contains more than $|E|$ cut vertices, and since $K > |E|$ (recall that K is the size of the cliques in a *clique expansion*), there are at least $K - |E| > 0$ vertices in the clique that have only red edges incident or only blue edges incident. □

We can now assign a color to each clique in the *clique expansion* (V', E'), and by extension, to each vertex in the original graph (V, E), but what would be the cost of such a partition?

Proposition 2.7. *Recoloring the edges of each clique with its dominating color will not increase the number of cut vertices.*

Proof. Pick any non-cut vertex v. Note that this vertex may be shared by several cliques. Since the vertex is not cut, all of its incident edges have the same color, and by extension, the *dominating color* of all the cliques containing v is the same. Therefore, after the recoloring v will still not be cut. □

Corollary 2.8. *If we color each vertex in (V, E) with the dominating color of its clique in (V', E'), the cut size of (V, E) will not be larger than the volume in (V', E').*

Proof. To reiterate: if all cliques are colored in a single color, we can map the solution to the Graph edge-bisection problem on (V', E') back into a solution for the Graph Bisection problem on (V, E) with the exact same cost. By the above proposition, if we color each vertex with the *dominating color* of its clique, we get a solution to the Graph Bisection problem on (V, E) with cut cost less than or equal to the volume of our given optimal solution to the Graph edge-bisection problem on (V', E'). □

Finally, the Graph Bisection problem requires both sets in the partition to have the same size. Hence, we would want there to be an equal number of cliques with red and blue as their *dominating color* (since we map cliques in (V', E') to vertices in (V, E)). This is where the constraint on K from Definition 2.2 comes into play:

Proposition 2.9. *In (V', E') there are as many cliques with red as their dominating color as there are cliques with blue as their dominating color.*

Proof. Let $r, b \geq 0, r + b = |V|$ denote the number of cliques of each kind, assuming without loss of generality that $r \geq b$. Now consider a lower bound on number of red edges in any bipartitioning of the edges in (V', E'): in each red clique, we have at most $|E|$ cut vertices, and the edges between these vertices may be blue, but all other edges should be red, so a lower bound on the number of red edges is [3]:

$$r\left(\binom{K}{2} - \binom{|E|}{2}\right)$$

Similarly, we can find an upper bound for the number of blue edges by the following reasoning: we color each blue-dominated clique entirely blue, and as many edges as possible in each red clique (at most $\binom{|E|}{2}$ in a single clique, since we can cut at most $|E|$ vertices). Note that we can cut at most $|E|$ vertices in total, not just per clique, so this is only an upper bound, not an actual maximum. We then get the following upper bound:

$$b\binom{K}{2} + r\binom{|E|}{2}$$

Since the Graph edge-bisection problem required us to partition the edges into two equisized subsets, certainly the lower bound on the amount of red edges should be smaller than or equal to the upper bound on the amount of blue edges:

$$r\left(\binom{K}{2} - \binom{|E|}{2}\right) \leq b\binom{K}{2} + r\binom{|E|}{2}$$

Rewrite:

$$(r - b)\binom{K}{2} \leq 2r\binom{|E|}{2} \tag{3}$$

Note that since $K = 4 + 2|V|\binom{|E|}{2} \geq 4$ we have $K \leq \binom{K}{2}$, and also $r \leq |V|$ holds. Then, if Equation 3 holds, certainly the following holds:

$$(r - b)K \leq 2|V|\binom{|E|}{2}$$

Substituting in K and rewriting we get:

$$4(r - b) + 2(r - b - 1)|V|\binom{|E|}{2} \leq 0$$

Since $r \geq b$, clearly this can only hold if $r = b$. □

[3] Recall that a complete graph on n vertices has $\binom{n}{2}$ edges.

We have now assembled all necessary results for the main theorem of this subsection:

Theorem 2.10. GRAPH BISECTION $\leq_{\mathcal{P}}$ GRAPH EDGE-BISECTION

Proof. Let (V, E) be a graph, then let (V', E') be its *clique expansion* (which, by Proposition 2.3 has size polynomial in the size of (V, E)), and find the optimal solution to the GRAPH EDGE-BISECTION problem on this graph. Color each vertex in (V, E) with the *dominating color* of its clique in (V', E'). By Proposition 2.9 this partitions the vertices into two equisized subsets, and by Proposition 2.4 and Corollary 2.8 this partitioning is optimal. □

2.3 MATRIX BIPARTITION is $\mathcal{NP}$-Complete

We now consider another restricted variant of the MATRIX k-PARTITION problem: the case when k is fixed to 2. We will call this problem MATRIX BIPARTITION. Its definition is the same as the MATRIX k-PARTITION problem we defined in subsection 2.1, except k is now fixed to 2.

As mentioned in subsection 2.2, we will rely on the $\mathcal{NP}$-Completeness of GRAPH EDGE-BISECTION for this proof. The idea is that given a matrix, we can create a graph by making a vertex for all rows and columns, and connect a row and column when their intersection contains a non-zero. However, since we know that GRAPH EDGE-BISECTION is $\mathcal{NP}$-Complete, we would like to reverse this process (i.e. create a MATRIX BIPARTITION instance from a GRAPH EDGE-BISECTION instance). The problem is that these 'matrix based graphs' are bipartite (since no two rows (or columns) intersect), while there is no such restriction on the input to the GRAPH EDGE-BISECTION problem.

We can again fix this by building a new graph:

Definition 2.11. *Given a graph (V, E), its edge-split graph is a new graph (V', E') where each edge in (V, E) is replaced by a path of length two. That is:*

$$V' = V \cup \{ v_e \mid e \in E \}$$

$$E' = \bigcup_{e=\{ u,w \} \in E} \{ \{ u, v_e \}, \{ v_e, w \} \}$$

Note that the *edge-split graph* of some other graph is always bipartite. The idea is now that when we have a graph (V, E), we take its *edge-split graph*, create the corresponding matrix, partition this, and then partition the edges of (V, E) based on this. This process is shown in Figure 5.

We run into a similar problem as we did in the previous section: we'd want each path of length two in the *edge-split graph* to be colored in the same color, but this is not always the case for an optimal solution. However, we can once again find a safe recoloring strategy that will not increase the volume.

We will denote the graph given as an instance for the GRAPH EDGE-BISECTION problem as (V, E), and its *edge-split graph* as (V', E'). Denote the optimal coloring of the edges in (V, E) as $\text{GEB}(V, E)$ and the optimal coloring of the edges in (V', E') as $\text{GEB}(V', E')$. We will call two edges in the (V', E') a *split pair* if they were generated by the same edge in (V, E). Assume we have some optimal coloring of the edges in (V', E') (optimal meaning: the number of vertices with both blue and red incident edges is minimal).

Proposition 2.12. *We can recolor the edges of (V', E') such that each split pair is either fully red or fully blue, without increasing the volume of the bipartition.*

Proof. We can assume that the number of edges in (V, E) is even (otherwise, no solution to the GRAPH EDGE-BISECTION problem would exist as it is impossible to bipartition E). It follows that the number of *split pairs* with both a blue and red edges is also even: let n_r denote the number of fully red split pairs, n_b the number of fully blue split pairs, and n_{rb}

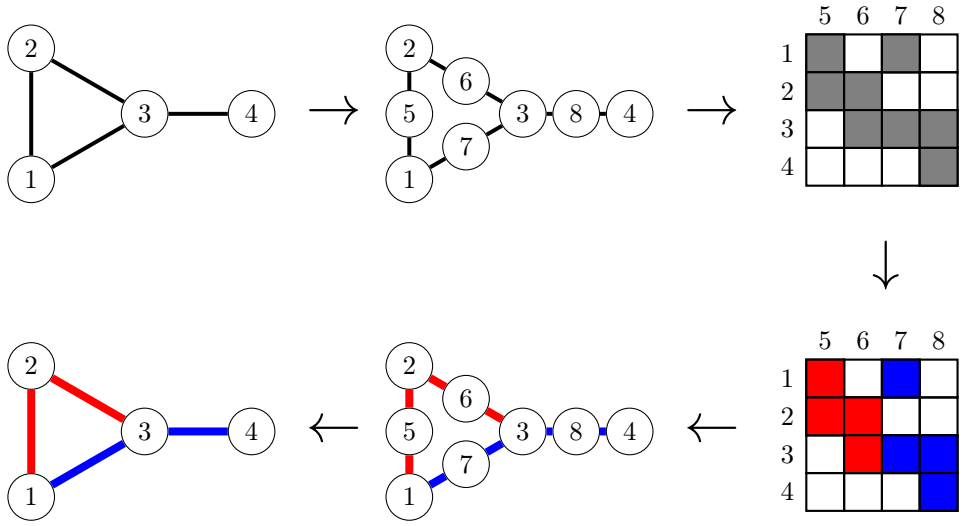

Figure 5: Solving MATRIX BIPARTITION using GRAPH EDGE-BISECTION.

the number of red and blue split pairs, then $|E| = n_r + n_b + n_{rb}$ is even. Also, the partition we found is certainly admissable, we have $2n_r + n_{rb} = 2n_b + n_{rb}$, and hence $n_r = n_b$, so $n_{rb} = |E| - (n_r + n_b)$ is the difference between two even numbers, and hence also even.

Pick any two *split pairs* in (V', E') that are not fully red or fully blue, and color one of the pairs fully red, and one of the pairs fully blue (arbitrarily chosen). Notice that the amount of red edges (blue edges respectively) did not change, so we still have a valid solution to the GRAPH EDGE-BISECTION problem on (V', E').

Let us focus on the pair that becomes fully red, denote the vertices as v_r, v_{mid}, v_b where the red edge connects v_r and v_{mid} and the blue edge connects v_{mid} and v_b.

Before the recoloring, v_{mid} was cut, and v_b may or may not have been cut. After coloring $\{ v_{mid}, v_b \}$ red, v_{mid} is no longer cut and v_b may now have been cut. Thus the volume either stays the same or decreases by one.

The same logic applies to the pair that becomes fully blue, and thus the number of *split pairs* with both types of edges decreases by two. Since this amount is finite and even, we can repeat until there are no such pairs left. □

Corollary 2.13. $\text{GEB}(V, E) \leq \text{GEB}(V', E')$

Proof. By the above proposition, any optimal partitioning of (V', E') can be transformed into a coloring of the same volume where all *split pairs* have a single color. We can now color each edge in E with the same color as its *split pair* in (V, E). It is easy to see that this is a valid bipartition with volume equal to the bipartition of (V', E'). □

Proposition 2.14. $\text{GEB}(V', E') \leq \text{GEB}(V, E)$

Proof. Given an optimal coloring of the edges of (V, E), color each edge in E' with the color of the edge in E that generated it. It is easy to see that this is a valid bipartition of (V', E') where the vertices in V' that are cut are precisely the vertices in $V \subseteq V'$ that are cut. □

Theorem 2.15. GRAPH EDGE-BISECTION $\leq_{\mathcal{P}}$ MATRIX BIPARTITION

Proof. Given an instance of the GRAPH EDGE-BISECTION problem, i.e. a graph (V, E), let (V', E') be its *edge-split graph.* This graph is bipartite, let $L, R \subseteq V$ be such that $V = L \cup R$, $L \cap R = \emptyset$ and there are no internal edges in L or R.

We create a matrix M with a row for each vertex in L, a column for each vertex in R, and for every edge $\{ l, r \} \in E'$ we add a non-zero in the intersection of the row corresponding to l and the column corresponding to r.

We then solve the MATRIX BIPARTITION problem on this matrix to optimality, and as observed earlier, we can convert this back into a valid, optimal bipartition of (V', E') with equal volume.

By Corollary 2.13 and Proposition 2.14 we can then derive an optimal bipartitioning of (V, E). □

We conclude that MATRIX BIPARTITION is $\mathcal{NP}$-Complete.

2.4 MATRIX BIPARTITION with $V = 0$

In the previous sections we showed that MATRIX k-PARTITION is $\mathcal{NP}$-Complete even when we restrict k to 2 or V to 0. This leads us to the question: what if we both restrict k to 2 and V to 0? That is, is the problem of partitioning the nonzeros of a matrix into two equisized subsets, such that no row or column contains a nonzero from both subsets, $\mathcal{NP}$-Complete?

As discussed in subsection 2.1, such a partition would logically require two nonzeros in the same row or column to go to the same partition. In other words, the matrix consists of a series of $c \geq 1$ components of sizes $a_1, a_2, \ldots, a_c$ (the size of a component equals the number of nonzeros it contains), where each component should be fully assigned to a single side of the partition. Note that these components can be found in polynomial time using breadth-first search (as described in e.g. [2]) in $\mathcal{O}(n+m+N)$ time for an $n \times m$ matrix with N nonzeros.

We have now reduced our problem to the following question: is there a subset $S \subseteq \{ 1, 2, \ldots, c \}$ such that $\sum_{i \in S} a_i = \sum_{i \notin S} a_i$? This is actually a well known problem called PARTITION (or more generally, SUBSET SUM). While this problem is $\mathcal{NP}$-Complete, it is only *weakly* $\mathcal{NP}$-Complete, and in our case, $a_i \leq N$ for all i.

In other words, this problem is solvable in polynomial time. Specifically, a dynamic programming algorithm exists that solves this problem in $\mathcal{O}(c \sum_{i=1}^{c} a_i)$ time, described in e.g. [2]. In our case, the number of components is bounded by $n + m$, since there are $n + m$ rows and columns (in fact, it is easy to see that if every row or column contains at least one nonzero, there are at most $\min(n, m)$ components). The sum of the sizes of the components is also exactly equal to N. Thus, the MATRIX k-PARTITION problem with the restriction that $V = 0$ and $k = 2$ may be solved in $\mathcal{O}(N(n + m))$.

We can summarize the results of this chapter concerning the MATRIX k-PARTITION problem in the following table:

	$k \geq 2$	$k = 2$
$V \geq 0$	$\mathcal{NP}$-Complete	$\mathcal{NP}$-Complete
$V = 0$	$\mathcal{NP}$-Complete	Solvable in $\mathcal{O}(N(n + m))$

3 Exact partitioning using branch and bound

Since optimally partitioning a matrix is $\mathcal{NP}$-Complete, we may not hope to find an exact polynomial time algorithm (unless $\mathcal{P} = \mathcal{NP}$). This typically leaves one with two options: finding a fast algorithm that only approximately solves the problem, or finding an exact algorithm that runs in superpolynomial time. In this section we will focus on the latter case. Specifically, we will use a common algorithmic technique, 'branch and bound', to solve the sparse matrix partitioning problem to optimality.

3.1 An introduction to branch and bound

Branch and bound is a general technique for solving optimization problems. The idea is that we start off with a complete set of all possible solutions, say S, and we seek to find some solution with minimal cost. We pick some property P_1, and split S into two sets: those solutions for which P_1 holds, and those for which it does not. We then break these new sets into smaller sets as well, until the sets have size 1 – when they represent a single solution.

This process generates a rooted tree called the 'branch and bound tree', which is shown in Figure 6. Of course, we don't actually store a list of solutions in each vertex. Rather, the vertices represent collections of solutions, i.e. the root represents all solutions, and its direct children represent solutions satisfying P_1 and not satisfying P_1, respectively. If the solution space is finite, then the tree has finitely many leaves, each representing a single solution.

The branch and bround algorithm works by traversing this tree: if we are at a leaf node then we examine the solution it represents (saving it if it is better than any other solution we saw before), and if we are at a non-leaf node we examine each of its subtrees one by one.

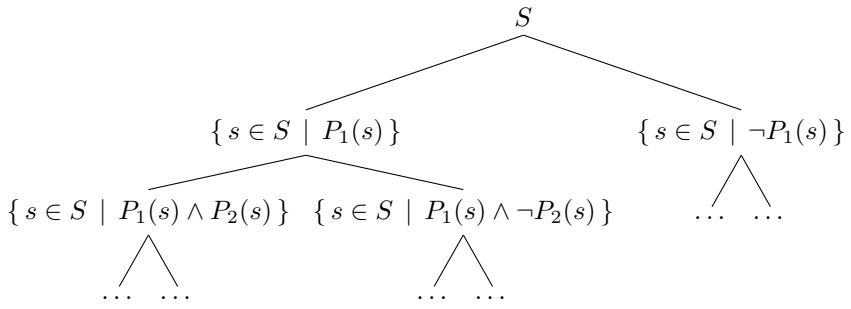

Figure 6: The branch and bound tree

At this stage the method does not appear to be an improvement over naive enumeration – it is at best *a technique* for naive enumeration. The key however, is to use the properties we branch on to our advantage. During the execution of the algorithm, we maintain the best solution found so far. Clearly, the cost of this solution, say U, is an upper bound on the cost of the optimal solution.

Then, suppose that we are in some vertex v in the branch and bound tree for which some properties $P_1, P_2, \ldots, P_i$ hold, and that we can prove that each solution satisfying these properties has cost $L \geq U$, then none of the leaves below v represent a better solution than the one we already found, and we can skip exploring this part of the branch and bound tree.

Ideally, we would want to skip large parts of the branch and bound tree, solving the problem much faster than naive enumeration. It should be clear however, that this largely depends on how often and how early we can cut off branches from the tree. In other words, while the algorithm itself is fairly simple, a lot of work and thought should be put into picking the properties we branch on, and how to calculate a lower bound based on these properties.

One useful observation we can make is that it isn't really necessary that we branch on exactly two properties (e.g. P and $\neg P$) – in fact, the branches we consider don't even have

to represent disjoint sets of solutions (as long as their union equals the original set). We will exploit these facts in later sections.

3.2 Prior work

We will build upon prior work on applying the branch and bound algorithm to the sparse matrix partitioning problem in [9]. In the following sections we will give a short description of the bounds and branching rules given in this paper.

3.2.1 Branching rules

First of all, we have to decide on a set of branching rules. An obvious first choice would be to branch on which side of the partition we place each nonzero, in other words, we number the nonzeros from 1 to N, and at depth i in the branch and bound tree, we branch on 'assign the ith nonzero to processor 0' and 'assign the ith nonzero to processor 1'. This would give our branch and bound tree a total of 2^N leaves.

Note however, that while the goal is to partition the nonzeros, we also try to minimize the number of cut rows and columns. Hence, a second choice would be to branch on *which* rows and columns we cut, and if we don't cut a row or column, to which processor we assign the nonzero it contains. Thus, if we number the rows and columns from 1 to $n+m$, at depth i we branch on 'assign all nonzeros contained in the ith row/column to processor 0', 'assign all nonzeros contained in the ith row/column to processor 1' and 'cut row/column i, then the nonzeros it contains may be assigned arbitrarily'.

Note that we now have three choices at each vertex, for a total of 3^{n+m} leaves. Simple rewriting shows that $3^{n+m} < 2^N$ if and only if $n + m < \log_3(2)N \approx 0.63N$. This bound is already fulfilled if each row or column contains an average of about four nonzeros, but we note that there are actually far fewer than 3^{n+m} admissable leaves: if there is a nonzero at place (r, c), we cannot e.g. assign row r to processor 0 and column c to processor 1. Of course, branches that represent such an assigment may be skipped.

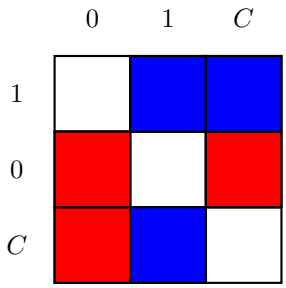

Figure 7: An optimal partition ($V = 2$) induced by an assignment of all rows and columns to one of three categories: 0 for processor 0, 1 for processor 1 and C for a cut row or column.

In subsection 3.1 we mentioned that each leaf represents a single solution, so how can different branching rules result in a different number of leaves for the same problem? The reason is that with the second branching scheme, each leaf does not necessarily represent a single solution: if for some nonzero (r, c) both row r and column c are cut, then we can assign (r, c) to either processor without increasing the volume of the partition. Thus, each leaf represents a collection of solutions with equal volume.

We will be working with the second set of branching rules, not only because it typically results in smaller branch and bound trees, but also because it makes reasoning about lower bounds easier.

Now that we have defined a set of branching rules, we need to start thinking about lower bounds: given a partial assignment of the rows and columns to one of the three categories,

can we find a lower bound on the volume of any solution extending this partial assignment? That is, what would be a minimum number of rows and columns any extension of this assignment should cut?

If we denote such a partial partition as (B_0, B_1, B_c), then one lower bound is obvious: the number of rows and columns that are already cut by this assignment. These rows and columns come in two forms:

1. Rows and columns explicitly cut by placing them in B_c.

2. Rows and columns implicitly cut by the placement of intersecting columns and rows in B_0 and B_1. If we have a row $r \notin B_c$ that intersects a column in B_0 and a column in B_1, then this row is already cut, even if we did not place it in B_c. We will refer to such rows and columns as 'implicitly cut'. For example, in Figure 8, the last row is implicitly cut.

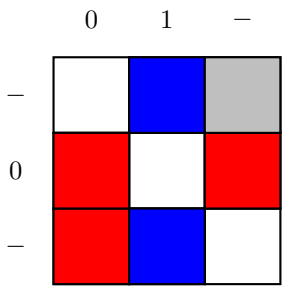

Figure 8: A partial partition that can be extended to the one shown in Figure 7. A dash (−) denotes a row or column that was not yet assigned to any category.

Following the notation used in [9], we will refer to these two lower bounds as L_1 and L_2 respectively. Then $L_1 + L_2$ is a lower bound on the volume of any extension of the partial assignment.

3.2.2 Packing bound

The first non-trivial bound that is used is the *packing bound*. This bound exploits the fact that both sides of the partition should have a similar size, and hence if a large side of the partition has a lot of adjacent unassigned nonzeros, some of these will have to go to the other side of the partition, and thus result in rows or columns being cut.

Consider the partial partition in Figure 8 again. We will fix one side of the partition, say, the side of processor 1 (blue), and a dimension of the matrix, say, the columns. Let n_0 be the number of nonzeros assigned to processor 0 (red) and n_1 the number of nonzeros assigned to processor 1 (blue). Then the remaining $N - n_0 - n_1$ nonzeros can be placed into the following categories, based on the column they are contained in:

1. Nonzeros that can be assigned to the blue processor without increasing the volume *induced by the columns alone*, either because they are contained in a column that is implicitly or explicitly cut, or because they are contained in an unassiged column that does not intersect with any rows assigned to the red processor.

2. Nonzeros whose assignment to the blue processor would increase the volume *induced by the columns alone* by one since the column containing it intersects with a row assigned to the red processor.

Note that we are explicitly ignoring what would happen with the communication volume in the rows, we focus on the columns alone. We will denote the number of unassigned

nonzeros in the first category by n_f (free nonzeros). If $n_1 + n_f < N - (1+\epsilon)\lceil \frac{N}{2} \rceil$ then we will have to gather some of the nonzeros for the blue processor from the second category.

Since taking some nonzero from any of the columns in this category cuts them, we may as well take all the nonzeros from this column. Since our goal is to cut as few columns (and rows) as possible, it is optimal to cut the columns in category two in descending order of number of free nonzeros.

In Figure 8, $n_0 = 3$, $n_1 = 2$ and $n_f = 0$. For most reasonable values of ϵ we will need to increase n_1 to 3, and we will thus have to cut column 3 by the above reasoning.

We can calculate this lower bound for the other dimension (rows) and/or color (red), adding all results up together to a new lower bound, which we will denote by L_3.

3.2.3 Matching bound

A second non-trivial lower bound can be derived by exploiting the connectivity of the matrix, or rather, its graph representation which we saw in section 2.

We will call a row or column partially red if it is unassigned but contains a red nonzero (assigned by an intersecting column or row), and define partially blue analogously.

Suppose a partially blue row and a partially red column intersect, then the nonzero at their intersection must, by definition, be unassigned. If we color this nonzero red, then the partially blue row will be cut, and if we color it blue then the partially red column will be cut.

In other words, an intersecting partially blue row and partially red column induce an increase in volume of at least 1 (with either the row or the column being cut). A list of c pairwise disjoint pairs of such rows and columns will thus induce an increase of volume of at least c.

A maximal set of such pairs may be computed using *bipartite matching*, as described in e.g. [2]. This lower bound is described in more detail in [9].

This lower bound is denoted by L_4. It is not hard to see that this bound may conflict with L_3, so our final lower bound is $L_1 + L_2 + \max(L_3, L_4)$.

3.3 A lower bound using vertex disjoint paths

We now describe an improved bound that generalizes subsubsection 3.2.3. The idea behind this bound is to take full advantage of the connectivity of the rows and columns of the matrix.

Consider the partial partitioning shown in Figure 9. The bounds we discussed earlier give us a lower bound of 0 on the volume of an extension of this partition. However, note that in order to accomplish this, we would have to color column 1 red, since row 3 is already red. But then row 1 would have to be colored red as well, and column 3, and finally row 2. However, this conflicts with coloring the second column blue!

We have found a series of intersecting rows and columns such that the first row/column is red, and the last row/column is blue. No coloring could possibly leave all rows and columns uncut, therefore any extension of this partitioning has volume at least 1.

This reasoning is easier to understand when thinking in terms of graphs, as we did in section 2. Let $V \subseteq R \cup C$ denote the collection of rows and columns that are not cut, either implicitly or explicitly. We define an accompanying edge set by connecting intersecting rows and columns:

$$E = \{\, \{r, c\} \mid r \in R \cap V,\, c \in C \cap V,\, (r,c) \text{ is a nonzero} \,\}$$

The graph we get for the matrix in Figure 9 is displayed in Figure 10.

Since all nonzeros correspond to edges in this graph, by definition, any edge adjacent to a vertex in B_0 will have to be colored red, and any edge adjacent to a vertex in B_1 will have

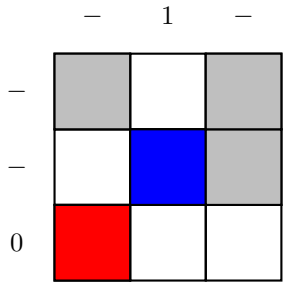

Figure 9: At least one row or column will have to be cut in an extension of this partitioning, yet none of the bounds discussed so far can detect this.

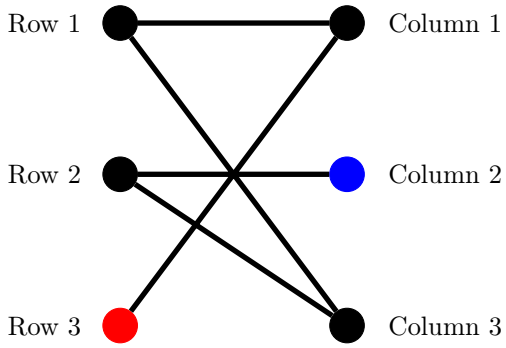

Figure 10: The matrix in Figure 9, represented as a graph.

to be colored blue [4]. Thus, after coloring all the edges in the graph, any path starting at a vertex in B_0 and ending in a vertex in B_1 will contain a vertex with both a blue and red adjacent edge, and hence guarantee an increase in volume of at least one (this statement is simple enough that we will omit a proof, but it can be proven using e.g. induction on the size of the path). We will call such paths B_0-B_1 paths.

Note however, that two distinct B_0-B_1 paths do not necessarily guarantee an increase in volume of at least two: if these two paths share a vertex that is not in B_0 or B_1, then we can simply cut this vertex, taking care of both paths (vertices in B_0 or B_1 cannot be cut anymore, because they are already fully assigned to a single processor).

Therefore, we will call two B_0-B_1 paths *disjoint* if they share no vertices outside of B_0 and B_1, that is, vertices that represent rows and columns that may still be cut. It follows that in a complete coloring of the edges of the graph, two disjoint B_0-B_1 paths must each contain a different cut vertex.

From this definition we can immediately derive a lower bound on the number of cut vertices: every collection P of pairwise disjoint B_0-B_1 paths gives rise to $|P|$ distinct cut vertices. Hence, the size of the largest collection of pairwise disjoint B_0-B_1 paths provides a lower bound on the number of cut vertices.

In fact, by applying the following theorem by Menger we can show that this is also an upper bound, i.e. the smallest number of required cut vertices is exactly the largest number of disjoint B_0-B_1 paths. This statement can be proven by merging each of B_0, B_1 into a single vertex, applying the theorem to these two vertices.

Theorem 3.1. *(Menger, [8]) Let G be a finite undirected graph, and let u and v be two non-adjacent vertices in G, then the size of the smallest vertex cut separating u and v equals the maximum number of vertex disjoint paths between u and v.*

[4]Note that no edges connect vertices from B_0 and B_1, as this would mean we assigned an intersecting row and column to distinct processors

Unfortunately, this observation does not directly help us since we have the additional constraint that the partition of the edges should split them roughly in half. Hence, we can only use the maximum number of paths as a lower bound, not an exact value. We will denote the value of this lower bound as L_5.

However, this theorem may help us optimize our branch and bound algorithm: these cut vertices are apparently a 'chokepoint' in the graph. If we assign these vertices fully to one of the two processors, this may increase the number of B_0-B_1 paths dramatically, therefore allowing us to cut off a large part of the tree at an early stage. On the other hand, if we assign a cut vertex to B_c, this may make the L_5 bound weaker (since this removes it from the graph).

In the beginning we mentioned that this bound generalizes L_4. To see why this is the case, note that the graph (V, E) we defined earlier contains the bipartite graph from L_4 as a subgraph. Specifically, only allowing paths of length 3 will give us L_4 (where the length of a path is the number of edges it contains) (paths of length 1 cannot exist as was explained earlier, nor can paths of length 2: in this case the middle vertex is either implicitly or explicitly cut). L_5 generalizes this bound in the sense that it allows paths of any length. In fact, for each $k \geq 3$ we can define a lower bound l_k: the maximum number of pairwise disjoint B_0-B_1 paths of length at most k. Then $L_4 = l_3$, and $L_5 = \lim_{i \to \infty} l_i$.

Since the l_i clearly form an increasing sequence, we get that $L_4 \leq L_5$, and therefore we can completely replace L_4 with L_5. As we will see in subsection 3.4, computing L_5 is computationally more intensive than computing L_4, therefore we might also consider some of the intermediate l_i as bounds.

We additionally note that L_5 conflicts with L_3, the packing bound, for the same reason that L_4 did. Hence, if we wanted to use both bounds at the same time, we would have to take the maximum of these two bounds.

3.4 Incremental computation of the lower bound

In the previous section we described a new lower bound, but did not yet specify how to efficiently *compute* this bound. Computing this new bound essentially boils down to the following question: given a graph (V, E) and two non-adjacent subsets $S, T \subseteq V$, what is the largest collection of paths starting in S and ending in T, such that no vertex in $V - (S \cup T)$ is used more than once?

This question is very similar to a standard algorithms problem: given a directed graph (V', A) and subsets $S', T' \subseteq V'$, what is the largest collection of paths starting in S' and ending in T' such that no edge is used more than once? This problem can be efficiently solved using maximum flow, as described in e.g. [2].

We will show how to compute our bound using this standard technique. Let the graph (V, E) and subsets $S, T \subseteq V$ be given. We define a new, directed graph based on (V, E). We will replace every vertex $v \in V$ by two vertices: an upper vertex $u(v)$ and a lower vertex $l(v)$. These vertices together represent v in the new graph, and we will connect the new graph in such a way that at most a single path can pass through this pair of vertices representing v.

Specifically, given (V, E) we define (V', A) in the following manner:

$$V' = \{\, l(v) \mid v \in V \,\} \cup \{\, u(v) \mid v \in V \,\}$$

$$A = \bigcup_{\{\, v,w \,\} \in E} \{\, (u(v), l(w)), (u(w), l(v)) \,\} \cup \{\, (l(v), u(v)) \mid v \in V \,\}$$

Of course, an image is once again useful to explain what is going on:

Now if we consider a path $v_1, v_2, \ldots, v_k$ in (V, E) to be equivalent to a path $l(v_1)$, $u(v_1)$, $l(v_2)$, $u(v_2)$, $\ldots$, $l(v_k)$, $u(v_k)$ in (V', A), then it is clear that two paths passing through the same vertex in (V, E) would pass through the same edge in (V', A).

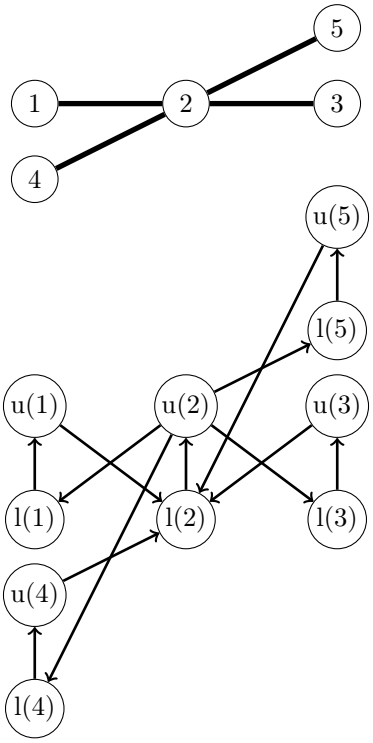

Figure 11: The transformation of (V, E) into (V', A).

Consider for example the paths $1, 2, 3$ and $4, 2, 5$. In (V', A) these paths would become $l(1), u(1), l(2), u(2), l(3), u(3)$ and $l(4), u(4), l(2), u(2), l(5), u(5)$, which share the edge $(l(2), u(2))$.

Thus, the size of the maximal set of vertex disjoint paths in (V, E) starting in S and ending in T is equal to the size of the maximal set of edge disjoint paths in (V', A) starting in $S' = \{ l(s) \mid s \in S \}$ and ending in $T' = \{ u(t) \mid t \in T \}$[5].

Thus, calculating L_5 reduces to finding the maximum flow in a graph. Although this graph can be reconstructed each time when the bound is calculated, note that when moving up and down the branch and bound tree, the graph changes very little. In particular, the following changes (and their reverse) may occur:

1. If a row or column is colored red, its vertex is added to S (becoming a possible source of a path).

2. If a row or column is colored blue, its vertex is added to T (becoming a possible sink of a path).

3. If a row or column is cut (implicitly or explicitly), its vertex is removed from the graph.

Update number three clearly allows a change in the flow of at most one (since at most one path can pass through this vertex), and can thus be solved by a single breadth-first search in $O(|V| + |E|)$ time. As for the other two updates, while these could increase or decrease the number of paths by more than one, all of these paths either start or end at the mentioned vertex. This considerably reduces the search space for new paths, allowing much faster update times than naive recalculation of the flow would give.

[5]Note that this definition results in paths that are also vertex disjoint in $S \cup T$, but it is not hard to modify the graph to drop this constraint (by e.g. not duplicating the vertices in $S \cup T$)

3.5 Combining the lower bound with a new packing bound

The packing bound L_3 is a lower bound based around relative sizes of the partial partition, whereas the flow bound L_5 is a lower bound based around the connectivity of the matrix (or at least, its graph representation). As we saw in subsection 3.3, we cannot combine both L_3 and L_5, instead we are forced to take the maximum.

Clearly this is not desirable: we would like to take into account both connectivity and relative sizes of the partition, at the same time. To achieve this, we apply a lower bound from [4] to our problem. Here, a similar partitioning problem is solved using branch and bound, and a flow bound similar to L_5 is used (we note that the flow bound we developed in subsection 3.3 was discovered independently).

In order to make sure we can 'combine' both a flow bound and a packing bound, we need to make sure the two bounds don't conflict. We do this by removing the flow paths from the graph. In other words, when we find some maximal set of paths P we remove them from the graph we described in subsection 3.3. Specifically, only the vertices in the path that are *unassigned* are removed, since these are the only possible vertices along the path that could have been cut. Of course, the edges in the path are removed as well. The vertices in the path that were assigned to either processor can be left in the graph. In this new graph, there may still be vertices left that are adjacent to vertices fully assigned to either processor, in other words, we can still calculate L_3 on the remaining graph. However, we cannot say anything about the edges contained in the paths we removed, as they may be assigned to any processor.

In matrix terms (which we used when discussing L_3), there are now fewer nonzeros in rows and columns partially assigned to some processor, and a lot less unassigned nonzeros. This will make the L_3 bound a lot weaker. Consider e.g. the graph on the left in Figure 12: after removing the paths, there are two vertices that fall in the category 'partially assigned to processor 0', with 1 and 2 adjacent unassigned edges, respectively. Besides this, there are 3 more completely unassigned edges (only adjacent to unassigned vertices), as well as two edges that were internal to the flow paths (we do not know which vertices along the flow paths were cut, so these edges could go to either processor).

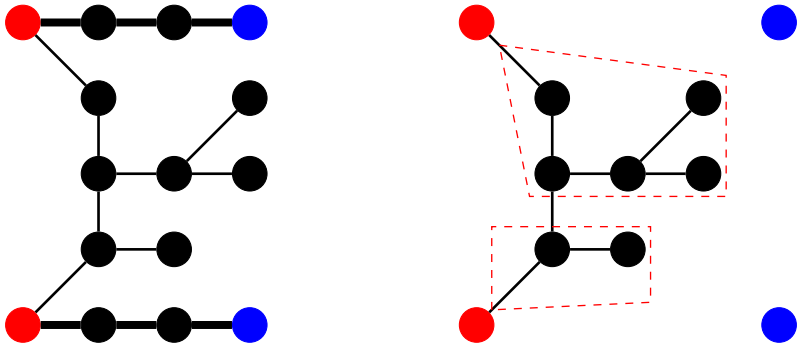

Figure 12: A graph representation of some matrix. Two flow paths have been thickened.

To strengthen the packing bound, we extend it in a manner similar to how we extended the matching bound: we don't just take into account those vertices (or rows and columns) that were partially assigned to some processor, but also take into account a larger, adjacent part of the graph.

Specifically, starting from each remaining vertex that is adjacent to one of the vertices in B_0 or B_1, we grow a connected subgraph, consuming the remaining vertices one by one. This can e.g. be done by a simultaneous breadth-first search starting from each vertex. This gives us a list of cells, each weighted by the number of edges they contain.

To this list of cells we apply the same procedure as we did for the previous packing bound: we try to assign as many of the remaining nonzeros to e.g. processor 1, but if there aren't

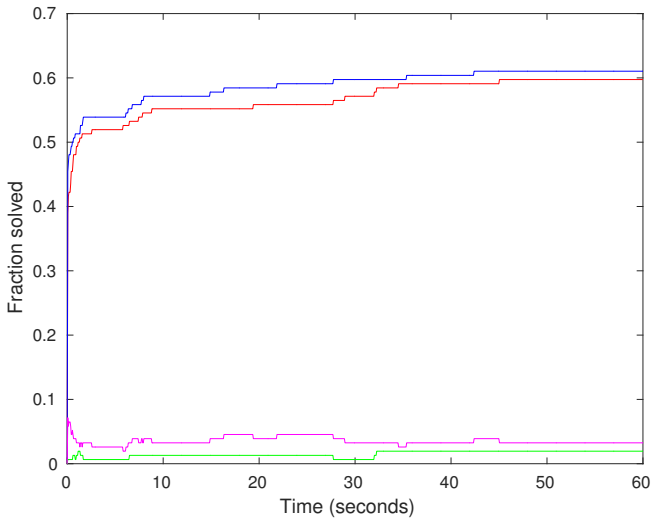

Figure 13: The fraction of matrices with at least 10, and upto 250 nonzeros solved optimally in the time given on the x-axis, using only the packing bound (blue) or using both the flow bound and the packing bound (red). Furthermore we show the fraction of such matrices that could *only* be solved by using only the packing bound (magenta), and the fraction of such matrices that could *only* be solved by using both the flow bound and the packing bound (green).

enough, we will have to start taking nonzeros from the cells adjacent to processor 0. Since each cell is connected and for each cell there is an edge connecting it to some vertex in B_0, assigning an edge in a cell to processor 1 guarantees that the cell will contain at least one cut vertex.

We thus apply the same greedy algorithm we used when calculating the original packing bound: cut the cells in descending order by size.

The graph on the right in Figure 12 contains an example of this: the upper subgraph spans four edges, and the lower subgraph contains just one. We ignore the edge connecting the subgraph to the vertex in B_0, as this edge (and its corresponding nonzero) is by definition already assigned a processor.

Now, at most 4 edges from the flow paths can be assigned to processor 1, as well as the last remaining edge between the two cells we found. So we have 5 edges (or nonzeros) for processor 1, but we need at least 7, so we are missing 2 edges.

The largest cell by size is the one with four edges, so we cut it, increasing our lower bound on the volume by one (on top of the lower bound of two given by the flow bound).

3.6 Experimental results

Due to time constraints, only a rudimentary version of the bound described in subsection 3.3 was implemented (the bound in subsection 3.5 could not be implemented), supplemented with the packing bound from subsubsection 3.2.2, i.e. the final bound was $L_1 + L_2 + \max(L_3, L_5)$.

The algorithm was implemented in `C++` and is available at the following location:

`https://github.com/TimonKnigge/matrix-partitioner`

This implementation was consequently compiled using `GCC 4.8.4` and tested on an Intel Core i7-4500U 1.8 GHz processor with 8 GB of RAM, running Linux Mint 17.1.

Various matrices from the University of Florida Sparse Matrix Collection [3] were used as test instances. An imbalance of $\epsilon = 0.03$ was allowed.

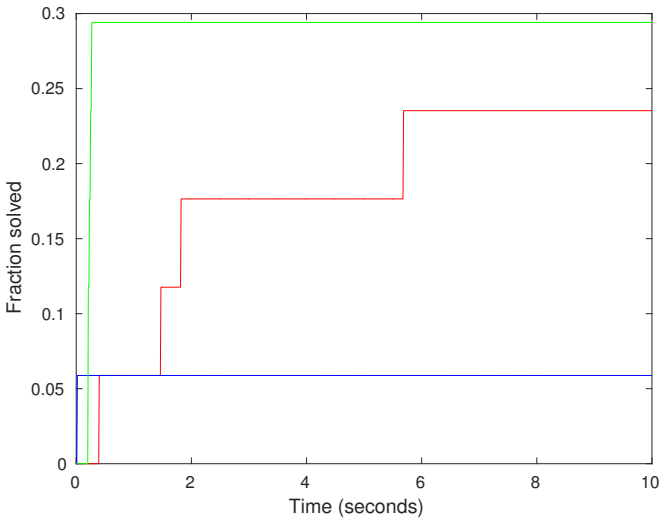

Figure 14: The fraction of matrices with at least 3000, and upto 3200 nonzeros solved optimally in the time given on the x-axis, using only the packing bound (blue), using both the flow bound and the packing bound (red), or using both the matching and packing bound (implementation from [9], green). The algorithm was run for five minutes, but none of the remaining matrices were solved in that time.

First of all, Figure 13 shows the results for all matrices in the aforementioned collection with nonzeros in the range $[10, 250]$. There are 154 such matrices. The program was run with and without the flow bound. From the results it becomes clear that for 'small' matrices, the flow bound is not an improvement. In fact, the results are slightly worse, although the difference is not large. Additionally, as evidenced by the green line, there are a few matrices that are not solvable in under a minute without the flow bound.

Next, Figure 14 shows the results for all matrices in the aforementioned collection with nonzeros in the range $[3000, 3200]$, 18 in total. Here we see that for larger matrices, the flow bound is a very useful addition to the packing bound, with the packing bound alone being unable to solve any of the matrices, except one. We also see that the flow bound can not yet compete with the matching bound on these larger matrices.

This might seem strange: in subsection 3.3 we mentioned that the flow bound *generalizes* the matching bound, i.e. $L_5 \geq L_4$. The key here is that computing the flow bound is computationally quite expensive, so while it allows us to cut off larger parts of the branch and bound tree, we pay for this by an increase in the average time spent in each node in the tree. Thus, there is a tradeoff to be made between stronger bounds and bounds that are computationally less intensive.

There are a few exceptions to the above rule though. One such example is the matrix `tols2000`, which is a 2000×2000 matrix with 5184 nonzeros, which was partitioned using the flow bound in 1.2 seconds, whereas even with the matching bound it could not be solved in a day. We note that this matrix had volume 0. The fact that this matrix could be solved so rapidly using the flow bound can probably be explained in the following way: both the matching bound (L_4) and the packing bound (L_3) are in a sense very 'local' bounds, they only concern unassigned nonzeros that are 'very close' to rows or columns assigned to a single processor (where 'very close' means within one or two steps when we interpret the matrix as a graph).

On the other hand, the flow bound (as well as the new packing bound described in subsection 3.5, although again, this bound was not implemented) is much more global.

If we interpret the `tols2000` matrix as a graph, then assigning two vertices in the same

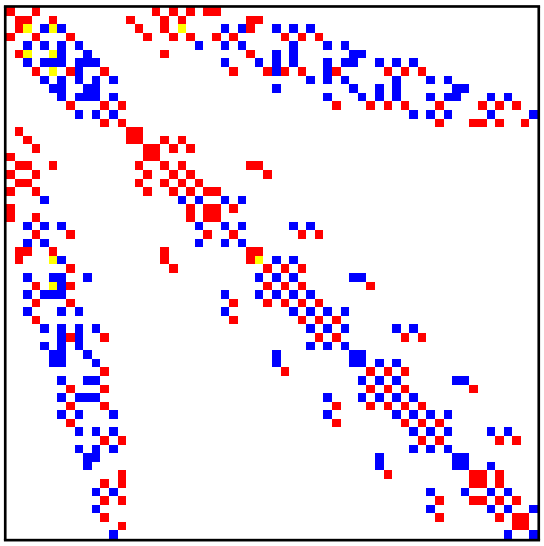

Figure 15: `bfwa62`, a 62×62 matrix with 450 nonzeros. The displayed partition of volume 11 is optimal, found in 5.3 seconds. The yellow nonzeros may be assigned to either processor as long as the load balance constraint is not violated.

connected component to distinct processors will never lead to an optimal solution (recall that the matrix had volume 0). The flow bound will immediately detect this (as there is a path between these vertices in the connected component), but with both the matching bound and the packing bound, we may have to traverse deep into the tree before finding this out.

In general, it seems that the flow bound introduces a slowdown that is not always compensated for by the increased strength of the lower bound. However, we can make the following improvements:

- The time complexity of finding the cells described in subsection 3.5 is close to the time complexity of finding the maximal set of vertex disjoint paths, so this bound can be added without significantly slowing down the program.
- Building on the distinction between 'local' and 'global' bounds we discussed earlier – global bounds are very useful early on in the branching stage, but do not provide large advantages at lower levels in the tree, while still being very computationally expensive. This suggests a hybrid method: in the higher parts of the branch and bound tree we use the global bounds, but in the lower parts of the tree we use the local bounds.
- A large part of this section has focused on the 'bound' part of branch and bound, and very little on the 'branch' part. Currently, rows and columns are branched on in non-increasing order of number of nonzeros. However, lots of other branching strategies could be used. For example, if we applied the suggestions from the previous point, it is important that we quickly cover large paths of the matrix, rather than only assigning to rows and columns that are clustered together. Therefore we could consider branching on rows and columns that are 'far' (by looking at the matrix as a graph) from other rows and columns that were branched on.

4 Conclusions and future work

We have examined the sparse matrix partitioning problem from various perspectives, obtaining various interesting results along the way. We have shown that optimally partitioning a sparse matrix is $\mathcal{NP}$-Complete even if the number of processors is fixed to 2. Recognizing matrices of volume 0 is also $\mathcal{NP}$-Complete – unless the number of processors is fixed to 2, in that case the problem may be solved in $\mathcal{O}(N(n+m))$ time for an $n \times m$ matrix with N nonzeros. Since a lot of research into the sparse matrix partitioning problem revolves around approximation algorithms, future research could consider the status of the problem within the class $\mathcal{APX}$ – for example, is the problem $\mathcal{APX}$-Complete?

Furthermore, we have introduced various new lower bounds to be used in the branch and bound algorithm designed by Pelt & Bisseling [9]. While the initial results concerning the lower bound in subsection 3.3 did not suggest significant improvements, we expect that adding the lower bound in subsection 3.5 will further strengthen the lower bound without noticably increasing computation time. Additionally, a lot more suggestions for future improvements were given at the end of subsection 3.6.

References

[1] Konstantin Andreev and Harald Racke. Balanced graph partitioning. *Theory of Computing Systems*, 39(6):929–939, 2006.

[2] Thomas H.. Cormen, Charles Eric Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*, volume 6. MIT press Cambridge, 2001.

[3] Timothy A Davis and Yifan Hu. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1, 2011.

[4] Daniel Delling, Andrew V Goldberg, Ilya Razenshteyn, and Renato F Werneck. Exact combinatorial branch-and-bound for graph bisection. In *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, pages 30–44. Society for Industrial and Applied Mathematics, 2012.

[5] Michael R Garey and David S Johnson. Computers and intractability: a guide to the theory of $\mathcal{NP}$-completeness. 1979. *San Francisco, LA: Freeman*, 1979.

[6] Michael R Garey, David S. Johnson, and Larry Stockmeyer. Some simplified $\mathcal{NP}$-complete graph problems. *Theoretical computer science*, 1(3):237–267, 1976.

[7] Richard M Karp. *Reducibility among combinatorial problems*. Springer, 1972.

[8] Karl Menger. Zur allgemeinen Kurventheorie. *Fundamenta Mathematicae*, 10(1):96–115, 1927.

[9] Daniël M Pelt and Rob H Bisseling. An exact algorithm for sparse matrix bipartitioning. *Journal of Parallel and Distributed Computing*, 85:79–90, 2015.