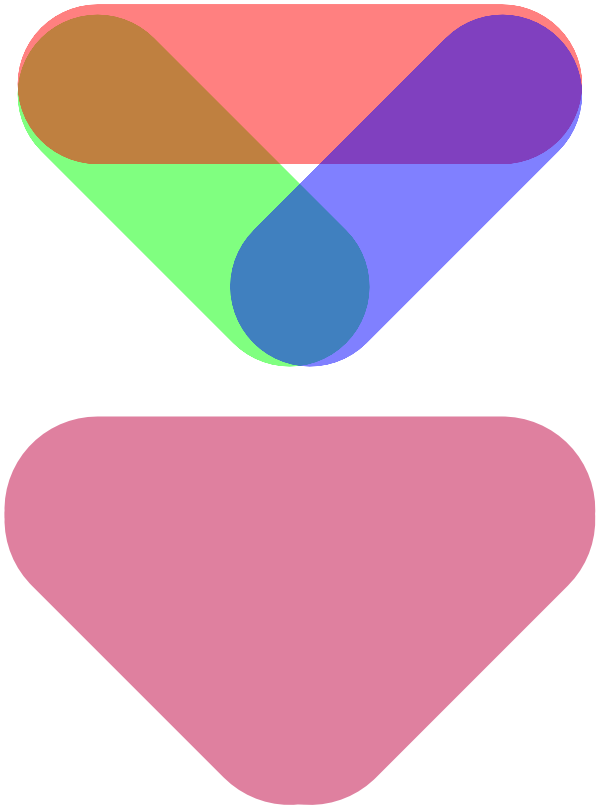
Utrecht University

Equivalence Relations For Hypergraph Partitioning

Master thesis

Rik Riesmeijer



May, 2022

Version: Final

Utrecht University

Department of Mathematics
Mathematical Institute
Scientific Computing

Equivalence Relations For Hypergraph Partitioning

Rik Riesmeijer

Supervisors Prof. Dr. Rob H. Bisseling
MSc. Sarita de Berg
Dr. Erik Jan van Leeuwen

May, 2022

Rik Riesmeijer

Equivalence Relations For Hypergraph Partitioning

May, 2022

Supervisors: Prof. Dr. Rob H. Bisseling, MSc. Sarita de Berg and Dr. Erik Jan van Leeuwen

Utrecht University

Scientific Computing

Mathematical Institute

Department of Mathematics

Utrecht

Abstract

Hypergraph partitioning is an important problem at the core of many parallel computations. The input data in real world applications where hypergraph partitioning is used has become increasingly large. This has created a need for parallel hypergraph partitioning algorithms, which can handle larger data. This is the problem for which PMondriaan was created. However, it also needs to be efficient. So in this thesis, we discuss ways to reduce the complexity of hypergraph partitioning problems by use of equivalence relations. We will define a new equivalence relation for hypergraph partitioning problem instances to find redundant aspects of input data and remove the redundancies. We also look at extended definitions of the conventional hypergraph partitioning problem. And we also find implications of the theory of instance equivalence we define here to other projects. We also find formal explanations of well known experimental results. While this thesis does not completely succeed in making PMondriaan the most efficient method for hypergraph partitioning, we will find ways to to use equivalence relations to solve this problem more efficiently.

Contents

1 Introduction

1.1 Motivation

In many cases in the real world, problems arise where some set of entities needs to be divided into groups according to relations defined on these entities [Kar+96; Len12]. For example, we may have a set of employees of some company. Here, the employees have meetings with each other that are the same every week. Now we have some building with k floors and we want to divide all the employees of the company over the floors such that the least amount of meetings take place between employees on different floors. The same analogy could also work for servers and data that is needed for different computations like sparse matrix vector multiplication[PB14].

The abstraction of this problem is called hypergraph partitioning. In many cases where one is solving this problem one can access a multitude of processing units that can run in parallel. Therefore we are interested in leveraging the qualities of parallel computing for hypergraph partitioning. This is why before this thesis the PMondriaan project was started, which is a hypergraph partitioning algorithm that uses parallel computation. This thesis will discuss modifications made for PMondriaan. Since the current state of PMondriaan is relevant to this thesis we first need to understand this and the problem it was made to solve.

1.2 Notation

We will now mostly use the notation of the thesis where PMondriaan was first introduced[Ber20] (this section was taken from that text) as well as editing some notation with parts of [Bis20] in mind.

A hypergraph $H = (V, E)$ is a generalization of a graph. It consists of a set of vertices (or nodes) V and a set of hyperedges (or nets) E. Hyperedges connect two or more vertices of a hypergraph. We identify a hyperedge e by the subset $e \subseteq V$ of vertices it connects. The size of a hyperedge $e \in E$ is defined by $|e|$. An example of a hypergraph can be seen in Figure 3.1. By $w(v)$ we denote the weight of a vertex $v \in V$ and by $C(e) \geq 0$ the cost of a hyperedge $e \in E$. We denote by

$I(v)$ the set of hyperedges containing $v \in V$. The degree $d(v)$ of v is defined as $|I(v)|$. In applications, often $w(v)$ is set to $d(v)$. We denote the maximum vertex degree in the hypergraph by $\Delta(H)$. We call a vertex v and u adjacent if there exists a net containing both vertices. By $N(v)$ we denote the set of adjacent vertices (or neighbors) of v. By $L(v)$ we denote the label of vertex v, that is the part v is assigned to.

We define the hypergraph partitioning problem as follows: given a hypergraph $H = (V, E)$ and an integer k, partition the vertices V into k nonempty, mutually disjoint parts $V_0, V_1, \ldots, V_{k-1}$ satisfying $\bigcup_{i=0}^{k=1} V_i = V$, such that all parts have relatively equal weight and a cut metric is minimized. We call $P = \{V_0, \ldots, V_{k-1}\}$ a partitioning and $V_0, \ldots, V_{k-1}$ its parts. For $k = 2$, we also call the problem the hypergraph bipartitioning problem. The weight W_i of part V_i is the sum of the weights of the vertices it contains. We say that the partitioning adheres to the balance constraint if

$$W_i \leq \frac{1}{k} \sum_{j=0}^{k-1} W_j (1 + \varepsilon), \quad \text{for } i = 0, 1, \ldots, k-1,$$

where $\varepsilon \geq 0$ is the load imbalance parameter. This ε expresses the relative maximum load imbalance that is allowed.

Under this constraint, we minimize a so-called cut metric. Common choices for a cut metric are the hyperedge-cut or cutnet, the sum of external degrees (SOED) and the $(\lambda - 1)$-cut metric[KK00]. Let λ_i be the number of parts spanned by $e_i \in E$. Then the hyperedge-cut metric minimizes (a), the SOED minimizes (b) and $(\lambda - 1)$-cut minimizes (c): (a) $\sum_{i:\lambda_i \geq 2} C(e_i)$, (b) $\sum_{i:\lambda_i \geq 2} \lambda_i C(e_i)$, (c) $\sum_{i:\lambda_i \geq 1} (\lambda_i - 1) C(e_i)$

Here, we have generalized the definition of the SOED metric from [KK00] to account for hyperedge weights. For hypergraph bipartitioning, the hyperedge-cut and the $(\lambda - 1)$-cut metric result in the same minimization problem. We call any hyperedge e for which $|e| \leq 1$ free, and these hyperedges are usually removed because they do not matter. After removing such hyperedges we call a vertex free if they do not occur in any hyperedge after such a removal.

We can also view a hypergraph as a sparse matrix. We use the row-net model to express the hypergraph as a matrix. In this model, a row of the matrix represents a hyperedge (net) and a column a vertex of the hypergraph. For each $v_j \in V$ and $e_i \in E$ we have that a_{ij} is 1 if $v_j \in e_i$ and 0 otherwise. We denote by $nz(A)$ the number of nonzero elements in this matrix A. In this representation, the weights of the vertices and the cost of the hyperedges are given by two additional vectors. Note that any binary matrix can be transformed into a hypergraph by reversing this process.

We will use the following notations regarding parallel computing. By p we denote the total number of active processors. Following common notation in graph theory, we denote by s the index of the current processor (source) and by t the index of a remote processor (target). We denote a specific processor by $P(s)$ or $P(t)$.

In the recursive bipartitioning method, the set of vertices of the hypergraph is first partitioned into two parts V_0 and V_1. The induced hypergraphs of V_0 and V_1, containing only the vertices in the respective set, are again bipartitioned. We bipartition the resulting hypergraphs recursively until we reach the desired number of parts. Because all parts are disjoint, we now have a partitioning of the entire hypergraph. We use the multilevel approach for each bipartitioning, so in total we apply the multilevel scheme $k-1$ times. Note that, when k is not a power of two, we cannot create direct k-way partitionings when k is not a power of 2, we need to split the graph into uneven parts at some levels. For example, if we want three parts, we first bisect the graph into parts of size roughly equal to $\frac{1}{3}$ and $\frac{2}{3}$ of the total weight. In the next bisection step, we bipartition only the bigger part corresponding to $\frac{2}{3}$, to obtain three parts of roughly equal weight.

1.2.1 NP-Completeness

Hypegraph partitioning is in NP, because it can be put in terms of integer linear programming efficiently, as we will later see. Also, the partitioning problem is a specific case of this problem by making $\varepsilon = 0, k = 2$ and having no hyperedges and having the weights of vertices equal the integers of the partitioning problem[GJ79]. Therefore it follows this problem is NP-complete, because both an NP-complete problem can be reduced in linear time to and from this problem. It is common to use heuristics to solve NP-complete problems approximately in practice. The particular architecture of PMondriaan can now be examined, knowing the core problem it tries to solve. Since here we have not discussed cases of specifically $\varepsilon > 0$ and sparse instances, it is important to note they have also been shown to be NP-hard using the sparse matrix partitioning problem[KB20].

1.3 Current State Of PMondriaan

PMondriaan is a parallel hypergraph partitioner built using C++ and Bulk[BBB18]. At the core of the main architecture of the algorithm is a multi-level paradigm. Here we first try to reduce the complexity of the original bigger input problem by finding vertices that have similar connections to other vertices in the hypergraph. The first phase in the multi-level paradigm is called coarsening. In each coarsening step, the size of the hypergraph is reduced by finding vertices that are similar with respect

to some rating function. Then the vertices that were matched with one another are grouped together as if they are one vertex in a smaller coarsened hypergraph. In PMondriaan, contrary to projects like KaHyPar[Sch20], many vertices may be grouped together in each coarsening step. Then, in the reduced problem, we find a partitioning which we then refine while restoring the complexity of the original problem. Since we have access to a multitude of processing units both coarsening and uncoarsening have a parallel mode as well as a sequential mode. We could look at k-way partitioning but in PMondriaan recursive bisection is used for k-way partitioning so we will focus on $k = 2$. Now we will look at some components that are particularly interesting for the purposes of this thesis.

1.3.1 The BSP Model

BSP stands for Bulk Synchronous Parallel. The BSP model was originally proposed by Valiant[Val90]. In the BSP model, algorithms are divided up into supersteps and the framework gives a way to interpret the computational cost for a given BSP computer to perform a given computation. In the BSP model there is a notion of h-relations in communication steps. This counts the maximum number of incoming or outgoing data from different processors. Then g is the gap time between data words that are sent. At the end the processors need to synchronize after such steps in order to follow the same computation supersteps (since they usually depend on each others data from previous steps), this has a cost of l. Then we also have w which is the cost of the work done in a computation step.

The communication time is given by $T_{comm}(h) = hg + l$. The computation time of a given computation step is given by $T_{comp}(w) = w + l$. Finally, a mixed step has cost $T_{mixed}(w, h) = w + hg + l$. Some important parameters in the BSP model are p the number of processors, r the rate of computation of a single processor, g the cost of a data word communication and l the synchronisation cost.

1.3.2 Sequential Coarsening

After the original input has already been coarsened partially in parallel the sequential coarsening phase is reached. Here, subsets of similar vertices (clusters) are found by randomly visiting vertices and computing their best match. A match is another vertex with at least one common hyperedge, the best match is a match with highest rating function value, for example:

$$r(u, v) := \frac{1}{\min(d(u), d(v))} \sum_{e \in I(u) \cap I(v)} \frac{C(e)}{|e| - 1}.$$

Then a maximum cluster size is imposed which is usually 50, which ensures that we won't end up with imbalances of coarsened vertex weights. We want to limit the imbalance in coarsened vertex weights to improve the input hypergraph for the initial partitioning phase. We can greedily assign vertices to clusters that are not at maximum size, and after all vertices have been visited we contract all clusters into single vertices that encode the connectivity of the cluster they stand for.

1.3.3 Label Propagation

In the initial partitioning phase, the first partitioning is made using a label propagation algorithm. Here the vertices are randomly assigned to partitions and then iteratively put into the same partitions as the majority of their neighbors. This results in the following algorithm.

Algorithm 1: PMondriaan label propagation where $R(i, j)$ generates a random integer between i and j by a uniform distribution[Ber20].

Input: Hypergraph $H = (V, E)$, number of labels k, maximum number of iterations I.

```
1  forall v ∈ V do
2  |  L(v) ← R(0, k − 1)
3  iter := 0
4  change := True
5  while iter < I and change do
6  |  change ← False
7  |  forall v ∈ V do
8  |  |  for i := 0 to k − 1 do
9  |  |  |  c[i] ← 0
10 |  |  forall u ∈ N(v) do
11 |  |  |  c[L(u)] ← c[L(u)] + 1
12 |  |  m ← arg max_{0<i<k} c[i]
13 |  |  if m ≠ L(v) then
14 |  |  |  L(v) ← m
15 |  |  |  change ← True
16 |  |  iter ← iter + 1
17 |  return L
```

Output: The labels L of the vertices.

To understand Algorithm 1, we will now look at it step by step. First in lines 1 and 2 we see that every vertex is assigned to a random partition. Then we see in line 5 that we will iterate until the labels do not change anymore or our maximum allowed

number of iterations is reached. Then from line 7 on we compute labels for the vertices. Here the counts of neighbors labels are computed and then the maximum count is chosen to change the label of the vertex to. If at any point a label is changed then in line 13, the change variable is set to true.

1.3.4 Refinement

When we already have a partitioning of the vertices in a coarsened hypergraph, then we can refine this partitioning. In PMondriaan this is done by using the Fiduccia-Mattheyses heuristic [FM82]. Here, each vertex is moved to the other part alternating from each part according to the highest improvement to the cut this gives. This makes use of a gain bucket data structure which is a double linked list and here the gain buckets are put into an array for each gain value. This results in the following algorithm for updating the gain values of moving vertex v.

Algorithm 2: Update of the gain values for the move of vertex v.[Ber20] The term "free" is meant as explained in the notation section.

Input: Vertex v that is moved from part F to part T.

1 **forall** $e \in I(v)$ **do**
2 **if** $c_T(e) = 0$ **then**
3 $gain_u \leftarrow gain_u + C(e)$ for all free $u \in e$
4 **else if** $c_T(e) = 1$ **then**
5 $gain_u \leftarrow gain_u - C(e)$ for the only $u \in e$ with $L(u) = T$ if u is free
6 $c_F(e) \leftarrow c_F(e) - 1$
7 $c_T(e) \leftarrow c_T(e) + 1$
8 **if** $c_F(e) = 0$ **then**
9 $gain_u \leftarrow gain_u - C(e)$ for all free $u \in e$
10 **else if** $c_F(e) = 1$ **then**
11 $gain_u \leftarrow gain_u + C(e)$ for the only $u \in e$ with $L(u) = F$ if u is free

It is important to note here that the cases are split on the $c_T(e)$ values which count how many of the vertices in the hyperedges are in a certain partition. Only single-vertex moves that either break a hyperedge that was intact in a partition before or single-vertex moves for a hyperedge where the entire hyperedge now is contained in one partition. This is what we see above, either values of 1 or 0 are looked at.

2 Instance Equivalence For Hypergraph Partitioning

2.1 Hypergraph Partitioning Problem Definition

To correctly stage the hypergraph partitioning problem with generality, as well as defining it in a way that is conducive to the proofs that will follow, we will construct the following integer linear programming formulation.

For convenience, in the following text we will want any hypergraph we speak of with the same number of vertices, to have the same number of hyperedges. To mend this difference in the number of hyperedges, we will add any hyperedge of size 2 or larger with a cost of 0. Therefore any contribution of the cost of such an hyperedge will be 0 while still existing. On the other hand removing any hyperedge of cost 0, will also not change the value of any sum of hyperedge costs. This will always allow us to assume that a hypergraph with n vertices will have $2^n - n - 1$ hyperedges without loss of generality in the partitioning problem (since there are $n + 1$ subsets of size 1 or 0). Outside of theory, of course, we would never want to use this exponentially sized representation. Now that we know this we will build up the properties of legitimate weight functions and cost functions, legitimate here meaning that they have the proper domain and range.

Let us presume we have a complete hypergraph $H = (V, E)$, where $V = \{v_1, v_2, \ldots, v_n\}$, $m = 2^n - n - 1$ and $E = \{e_1, e_2, \ldots, e_m\} = \{e | e \in \mathcal{P}(V) \wedge |e| > 1\}$ with $\mathcal{P}(V)$ the powerset of V. We also have a cost function for the hyperedges

$$C : E \to \mathbb{Z}$$

and a weight function for the vertices

$$w : V \to \mathbb{Z}_{\geq 0}.$$

Note here that the cost function will map $C(e) = 0$ for any hyperedge that is not in the set of hyperedges in the conventional notation of the hypergraph partitioning problem. One reason to use a definition like this is that we do not have to worry about duplicate hyperedges, another reason is to circumvent having two hypergraphs

with the same number of vertices but different hyperedges. To clarify, the second reason comes in handy when we want to create or destroy hyperedges or compare hypergraphs with different connectivity without having to explicitly state differences in connectivity. We also have an imbalance parameter

$$\varepsilon \in [0, 1],$$

and a number of parts

$$k \in \mathbb{N}_{\geq 2}.$$

In the case where we want to fix vertices to parts we also have a fix set of vertices $f_\alpha \subseteq V$ for each part α, where in the standard case we have $f_\alpha = \emptyset, \forall \alpha \in \{1, 2, \ldots, k\}$.

To state the integer linear programming problem we need some variables first. We first define a variable to indicate that vertex v_i is in part α, we do this by

$$p_{\alpha i} \in \{0, 1\}, \forall \alpha \in \{1, 2, \ldots, k\}, i \in \{1, 2, \ldots, n\}.$$

Now we need a variable to indicate to what extent a hyperedge is spread over multiple parts, we call this variable the "brokenness" of hyperedge e_l, which gives

$$b_l \in \mathbb{N}, \forall e_l \in E.$$

Note that the "brokenness" of an hyperedge is the multiplicity of the cost of an hyperedge that we need to take into account for the cost function induced by some specific metric (i.e. for the λ-metric the brokenness b_l is the connectivity λ_l). So then an hyperedge is "broken" when $b_l > 0$ and not broken otherwise. We also have the brokenness of hyperedge e_l with respect to each part, meaning that there are both vertices of the hyperedge in the part and outside of the part, which gives us

$$b_{\alpha l} \in \{0, 1\}, \forall \alpha \in \{1, 2, \ldots, k\}, e_l \in E.$$

Now that we know the variables we are going to use we will define a set of constraints for feasible configurations in the hypergraph partitioning problem. We first need to make sure that one vertex cannot be assigned to multiple parts at the same time and is always assigned to some part by

$$\sum_{\alpha=1}^{k} p_{\alpha i} = 1, \forall i \in \{1, 2, \ldots, n\}. \tag{2.1}$$

Then, we have the conventional imbalance constraints,

$$\sum_{i=1}^{n} p_{\alpha i} w(v_i) \leq \frac{1}{k}(1 + \varepsilon) \sum_{i=1}^{n} w(v_i), \forall \alpha \in \{1, 2, \ldots, k\}. \tag{2.2}$$

We also have the constraints for fixed vertices given by

$$p_{\alpha i} = 1, \forall v_i \in f_\alpha, \forall \alpha \in \{1, 2, \ldots, k\}. \tag{2.3}$$

After we have defined the more conventional constraints, we will now go into some forcing constraints to get very well behaving brokenness variables. We first work towards brokenness with respect to a part. We note that we want $b_{\alpha l} = 0$ if either

$$\forall v_i \in e_l : p_{\alpha i} = 0$$

or

$$\forall v_i \in e_l : p_{\alpha i} = 1,$$

which leads to constraints

$$b_{\alpha l} \leq \sum_{v_i \in e_l} p_{\alpha i}, \forall \alpha \in \{1, 2, \ldots, k\} \tag{2.4}$$

and

$$b_{\alpha l} \leq \sum_{v_i \in e_l} (1 - p_{\alpha i}), \forall \alpha \in \{1, 2, \ldots, k\}, \tag{2.5}$$

respectively. This is functional as either the first constraint forces $b_{\alpha l} = 0$ because all $p_{\alpha i}$ in the hyperedge are 0, or the second constraint forces this because all $1 - p_{\alpha i} = 0$ in the hyperedge. Now, because we want

$$b_{\alpha l} = 1 \iff \exists v_i, v_j \in e_l : p_{\alpha i} \neq p_{\alpha j}$$

to match the conventional explanation of the problem. To get $b_{\alpha l} = 1$ if $\exists v_i, v_j \in e_l : p_{\alpha i} \neq p_{\alpha j}$ we define forcing constraints as follows

$$p_{\alpha i} - p_{\alpha j} \leq b_{\alpha l}, \forall \alpha \in \{1, 2, \ldots, k\}, \forall v_i, v_j \in e_l. \tag{2.6}$$

This works because

$$\exists v_i, v_j \in e_l : 0 = p_{\alpha i} \neq p_{\alpha j} = 1 \implies b_{\alpha l} \geq p_{\alpha j} - p_{\alpha i} \geq 1$$

and $b_{\alpha l} \in \{0, 1\}$ by definition. We get

$$\forall v_i, v_j \in e_l : p_{\alpha i} = p_{\alpha j} \implies b_{\alpha l} \geq p_{\alpha j} - p_{\alpha i} \leq 1$$

from equations (2.4) and (2.5). To simplify these constraints we will make a cutnet metric helper variable. We create helper variable

$$\gamma_l \in \{0, 1\}, \forall e_l \in E$$

which models the brokenness in the cutnet metric, so we have

$$b_{\alpha l} \leq \gamma_l, \forall l \in \{1, 2, \ldots, m\}, \alpha \in \{1, 2, \ldots, k\},$$

as well as

$$\gamma_l \leq \sum_{\alpha=1}^{k} b_{\alpha l}, \forall l \in \{1, 2, \ldots, m\}.$$

The first ensures that brokenness with respect to any part will propagate brokenness to the helper variable. The second also ensures non-brokenness with respect to all parts will also propagate to non-brokenness of the helper variable.

Now that we have all the basic components to work towards the utility function to optimize for the problem, we need to make different adjustments to the problem for each cut metric by adding extra constraints.

1. **Cutnet**: For this case we get $b_l = \gamma_l$.

2. λ: Second we have the λ metric, where we want to account for all brokenness with respect to parts in the general brokenness b_l. This gives

   $$b_l = \sum_{\alpha=1}^{k} b_{\alpha l}, \forall e_l \in E.$$

   This metric is sometimes called the "sum of external degrees" metric, or SOED for short.

3. $\lambda - 1$: Last, we have the $\lambda - 1$ cut metric, which is the λ metric subtracted by the cutnet metric. So then we complete the definition with

   $$b_l = \left(\sum_{\alpha=1}^{k} b_{\alpha l} \right) - \gamma_l.$$

This finally brings us to the utility function of the problem that we can completely define in terms of b_l values and $C(e_l)$ values, namely the objective is

$$\min_{b} \sum_{l=1}^{m} b_l C(e_l).$$

This coincides with the idea of brokenness, as clarified before. This concludes the definition of the problem. For convenience, we define function $J(b) = \sum_{l=1}^{m} b_l C(e_l)$ defined over the b's induced by the feasible solutions of the ILP problem. In further sections J is also called the cut size, the objective function or utility function.

2.2 Equivalent Instances Of Hypergraph Partitioning

One widely known fact about hypergraph partitioning is that duplicate nets can be contracted by summing their costs. This is possible because duplicate hyperedges will always have the same brokenness factor in the cut size and the cut size is additive. To find out what other transformations can make hypergraph partitioning easier, we will introduce the notion of instance equivalence. This can give us ways to simplify hypergraph partitioning problem instances that do not affect the ordering of the utility of feasible solutions, for example.

2.2.1 Defining An Equivalence Relation For Hypergraph Partitioning Instances

In this subsection we will see one particular way to define equivalence of hypergraph partitioning problem instances. This way is very specifically suited to keep high similarity, to make it useful in practice. We allow any feasible solution to differ by some constant utility universally, as well as having the same set of feasible solutions. One reason to have this as definition, is that the relative gain from one solution to another will then be the exact same, therefore some deterministic local search algorithm will find the same neighbors in both instances for the same starting solution. We will however mostly restrict to exactly equal objective function outputs for all feasible solutions.

2.2.1.1 The Equivalence Relation

Now that we stated what we want to construct, we first need to formalize the definition of an instance.

Definition 2.2.1. An **instance** of the hypergraph partitioning problem is defined as a tuple of $(w, C, f, \varepsilon, k)$, where w is the weight function of the vertices, c is the cost function for the hyperedges, f is the vector of sets of fixed vertices for each part, ε is the imbalance constraint parameter, and k is the number of parts. Note that n is omitted because it can be deduced from the size of the domain of w. When we have $w(v) = 0, \forall v \in V$, then we write $(0, C, f, \varepsilon, k)$ instead of $(w, C, f, \varepsilon, k)$.

And using this definition we can now define the relation.

Definition 2.2.2. We say $(w, C, f, \varepsilon, k)$ and $(w', C', f', \varepsilon', k)$ (as in Definition 2.2.1) are **equivalent instances** if and only if:

1. The set B of b vectors that are a part of a feasible solution of the ILP problem induced by $(w, C, f, \varepsilon, k)$ is equal to the set B' of b vectors that are a part of a feasible solution of the ILP problem induced by $(w', C', f', \varepsilon', k)$. From this point, whenever B is mentioned, this is the set B of b vectors that are a part of a feasible solution of the ILP problem induced by some instance of the hypergraph partitioning problem.

2. And there exists $a \in \mathbb{Z}$ such that for all $b \in B = B'$, we have

$$J(b) - J'(b) = a,$$

   where J is the utility function induced by $(w, C, f, \varepsilon, k)$, and J' is the utility function induced by $(w', C', f', \varepsilon', k)$.

If $a = 0$ we call this **strict equivalence**. For instance equivalence, we may also use the notation

$$(w, C, f, \varepsilon, k) \equiv (w', C', f', \varepsilon', k),$$

and for strict equivalence we may use

$$(w, C, f, \varepsilon, k) \equiv_0 (w', C', f', \varepsilon', k).$$

Note that instance equivalence is always taken with respect to some metric to make it usable (otherwise there is no induced ILP formulation for an instance).

In the definition, it is important to see that one of the two conditions is usually trivial if only costs are different or only weights, fixed vertices and ε, therefore splitting it up in the two parts is helpful in proofs. And k is never really relevant, but needed because some equivalences only hold for certain k values. We can show this is an equivalence relation in the usual sense.

Lemma 2.2.1. The relation defined in Definition 2.2.2 is an equivalence relation.

Proof. To prove this we need to show the relation is reflexive, symmetric and transitive.

1. **Reflexive**: We have

$$(w, C, f, \varepsilon, k) \equiv (w, C, f, \varepsilon, k),$$

because the induced ILP problem is identical in such a case.

2. **Symmetric**: We have

$$(w, C, f, \varepsilon, k) \equiv (w', C', f', \varepsilon', k) \iff (w', C', f', \varepsilon', k) \equiv (w, C, f, \varepsilon, k),$$

because the equality operator on sets is symmetric for the first part of Definition 2.2.2, and for the second part we have

$$J(b) - J'(b) = a \iff J'(b) - J(b) = -a \in \mathbb{Z}, \forall b \in B$$

therefore fitting the second part of Definition 2.2.2.

3. **Transitive**: If

$$(w, C, f, \varepsilon, k) \equiv (w', C', f', \varepsilon', k)$$

and

$$(w', C', f', \varepsilon', k) \equiv (w'', C'', f'', \varepsilon'', k)$$

then

$$(w, C, f, \varepsilon, k) \equiv (w'', C'', f'', \varepsilon'', k).$$

Again, the first part of Definition 2.2.2 satisfies this because of the transitivity of equality of the B sets. For the second part of Definition 2.2.2 we see that if $J(b) - J'(b) = a, \forall b \in B$ and $J'(b) - J''(b) = a', \forall b \in B$ then

$$J(b) - J''(b) = J(b) - J'(b) + J'(b) - J''(b) = a + a' \in \mathbb{Z}, \forall b \in B$$

where J, J', J'' are the utility functions induced by $(w, C, f, \varepsilon, k)$, $(w', C', f', \varepsilon', k)$, and $(w'', C'', f'', \varepsilon'', k)$ respectively.

This concludes the proof that instance equivalence is an equivalence relation on instances. Similarly, $\equiv_0$ is an equivalence relation. □

Now that we have this relation that works as we intended, we can use it to find some equivalent instances.

2.2.1.2 Small Examples Of Equivalence

All these examples are w.r.t cutnet and $\lambda - 1$ metrics.

Lemma 2.2.2. See Figure 2.1 for a visual representation of this example. We have 3 vertices, v_1, v_2, v_3, all unweighted $w(v) = 0$, we have $f_1 = f_2 = \emptyset$. Then we

construct two cost functions c and C'. For c we have $C(\{v_1, v_2\}) = C(\{v_1, v_3\}) = C(\{v_2, v_3\}) = 1$ and $C(e) = 0$ otherwise. And for C' we have $C'(\{v_1, v_2, v_3\}) = 2$ and $C'(e) = 0$ otherwise. Then we have

$$(w, C, f, 0, 2) \equiv_0 (w, C', f, 0, 2).$$

Proof. First, we note that B is the same set in both cases, because the instances only differ with respect to cost functions. To show the utility functions are the same on both sides we simply note that if $e_4 = \{v_1, v_2, v_3\}$ is broken, then either e_1 and e_2, e_1 and e_3 or e_2 and e_3 are broken. This can be seen by the pigeonhole principle, because either part 1 or part 2 must contain 2 of the three vertices, therefore at least one of e_1, e_2 and e_3 is not broken and the others are, therefore $J(b) = J'(b) = 0$ in both instances or $J(b) = J'(b) = 2$ in both instances, with $(w, C, f, 0, 2)$ having utility function $J(b)$ and $(w, C', f, 0, 2)$ having utility function $J'(b)$. We can compute in similar fashion for the SOED metric, where we get $J(b) = J'(b) = 4$. □

In the next section we will see how we can use smaller equivalent instances to construct an infinitude of other equivalent instances. To intuit each step we will take, we will now visually see small instances that signify these constructions. At this point one can verify all of the following equivalent instances by hand and using the proof of Lemma 2.2.2. The first such equivalent instances are found in Figure 2.2, here nothing substantial, but the vertices are labeled differently. This first expansion of relabeling the same way on both sides of an equivalence should be a very intuitive way to generate other equivalent instances from one equivalence. The next figure, Figure 2.3, shows how to add unconnected and unweighted vertices to generate bigger equivalent instances from smaller equivalent instances. In Figure 2.4, we see how scaling all costs uniformly on both sides allows us to create new equivalent instances, that fit a practical case we come across. In Figure 2.5, we see how we can move from the much less restricted case of unweighted vertices, to more restricted cases with arbitrary weight functions and imbalance constraints. Finally, in Figure 2.6, we see that we can add both sides ("both sides" meant as in equations, but here meant for equivalence) of the equivalence between instances with the same cost function and generate a new set of equivalent instances doing so. In the next section, we will show these properties in a very general setting.

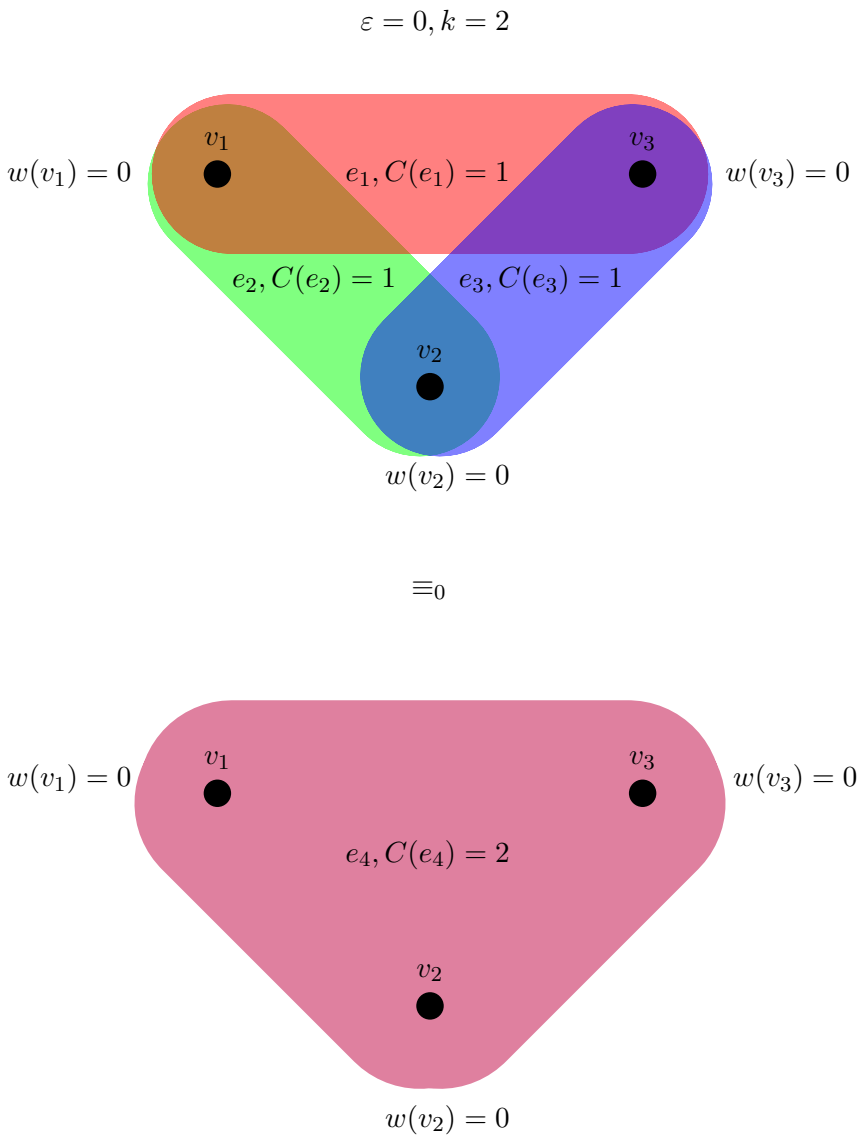


Fig. 2.1: A visual representation of Example 2.2.2, but with relabeled vertices. The top instance and the bottom instance of the partitioning problem are equivalent instances.

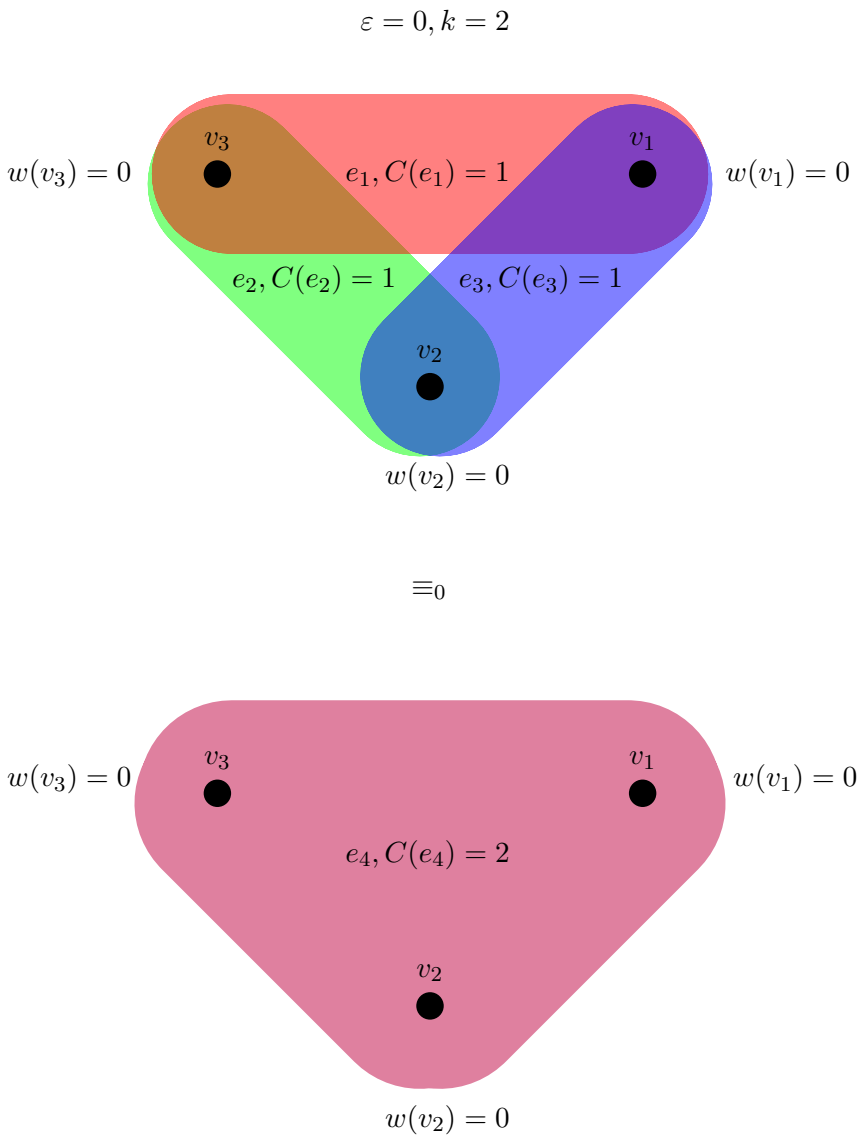


Fig. 2.2: A visual representation of Example 2.2.2, but with different vertex labels. The top instance and the bottom instance of the partitioning problem are equivalent instances.

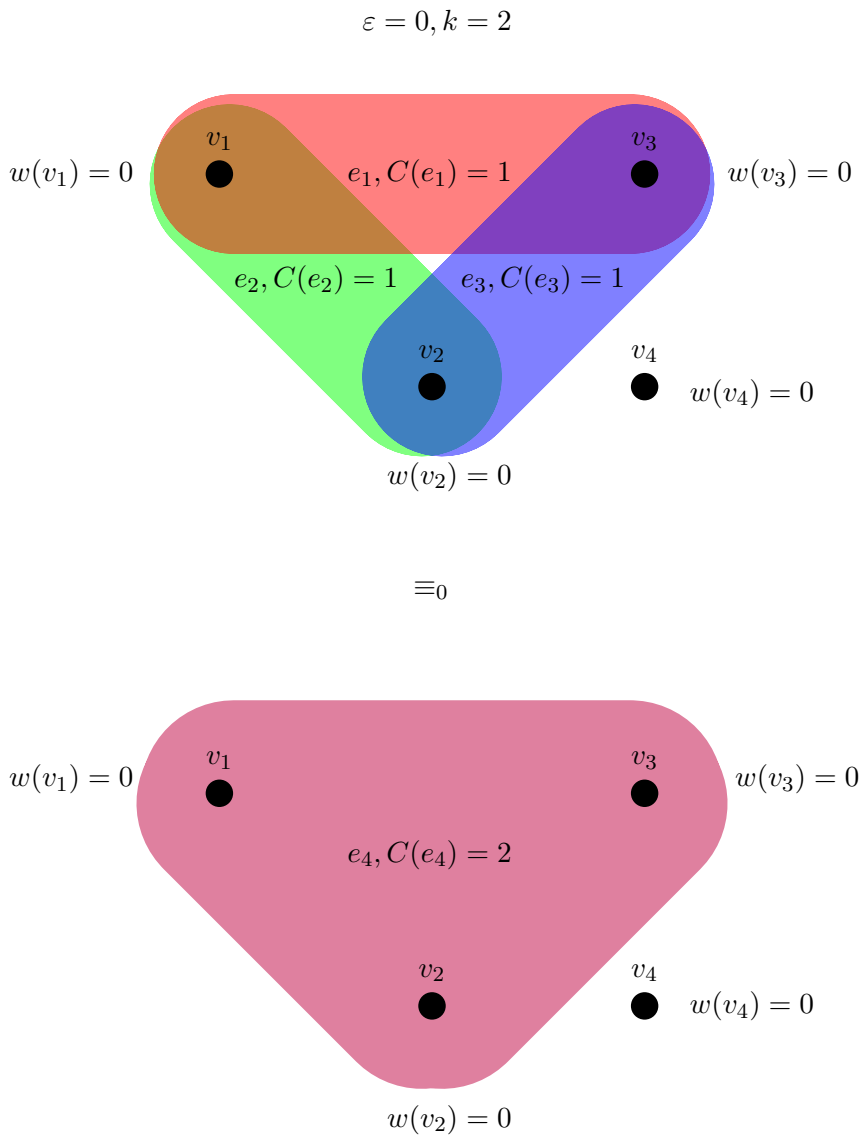


Fig. 2.3: A visual representation of Example 2.2.2, but with an added unconnected and unweighted vertex on both sides of the equivalence. The top instance and the bottom instance of the partitioning problem are equivalent instances.

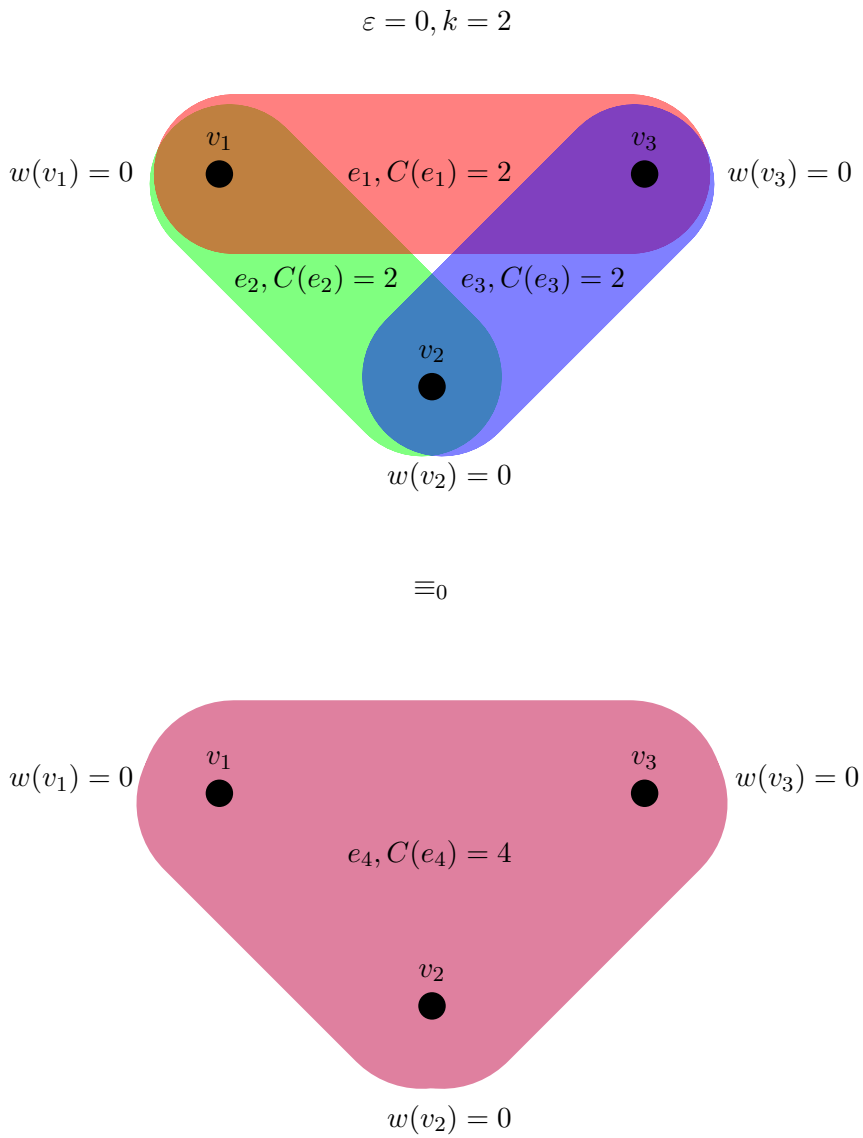


Fig. 2.4: A visual representation of Example 2.2.2, but with hyperedge costs doubled on both sides of the equivalence. The top instance and the bottom instance of the partitioning problem are equivalent instances.

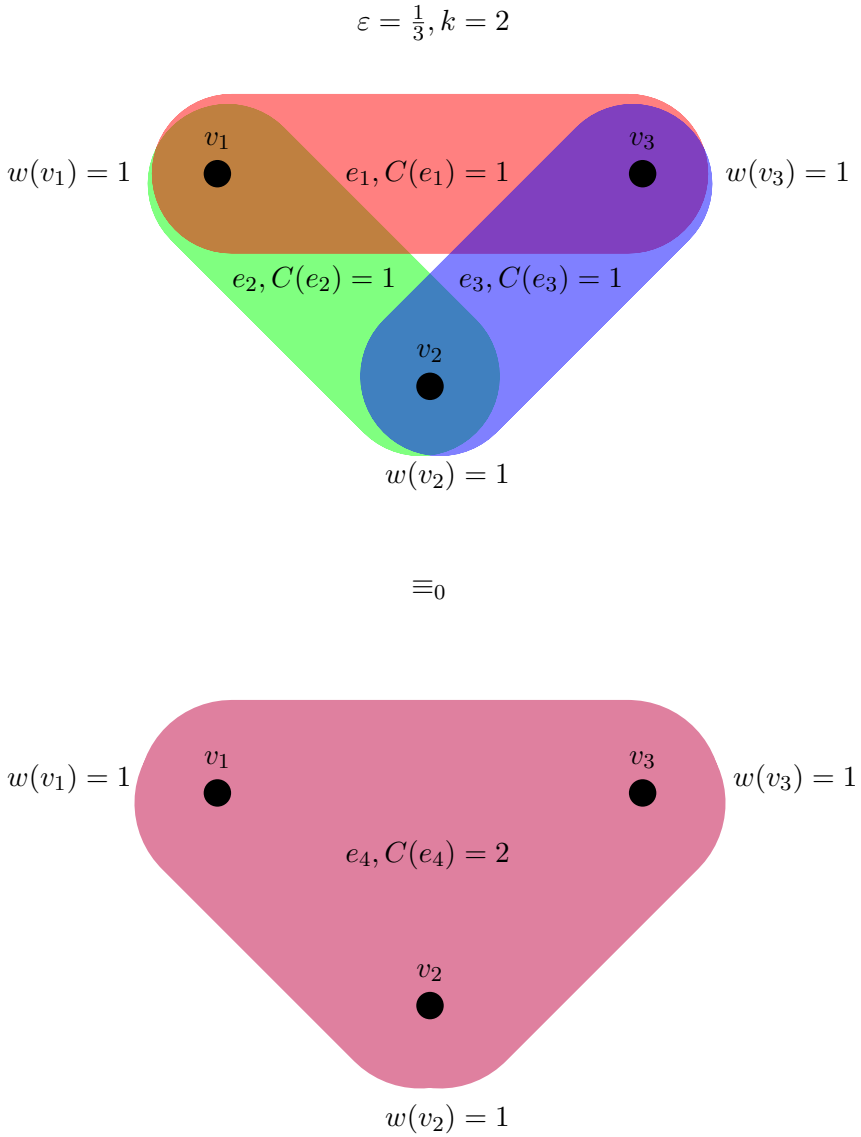


Fig. 2.5: A visual representation of Example 2.2.2, but with augmented ε values and vertex weights. The top instance and the bottom instance of the partitioning problem are equivalent instances.

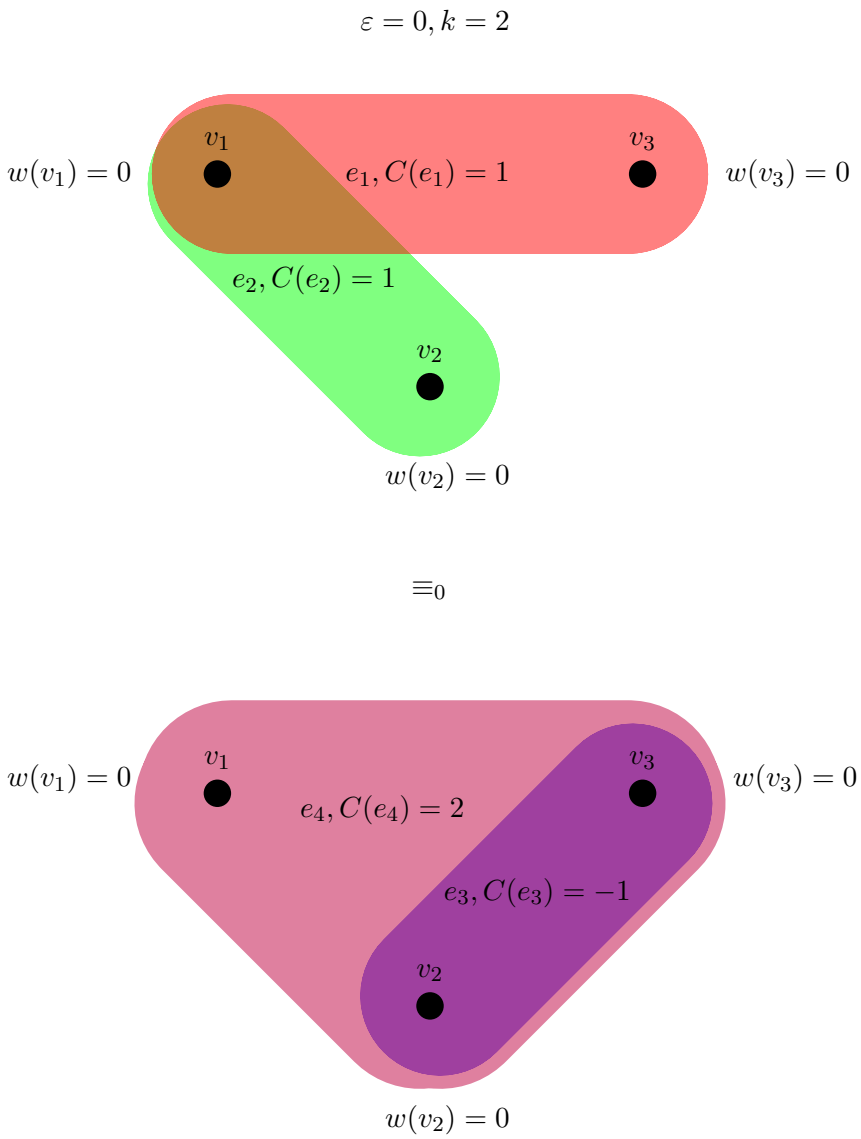


Fig. 2.6: A visual representation of Example 2.2.2, but with the cost of e_3 subtracted at both sides of the equivalence. The top instance and the bottom instance of the partitioning problem are equivalent instances.

2.2.2 How To Use Small Equivalent Instances

To clearly separate each of the concepts we try to substantiate, we will discuss each one in a separate subsection now. The lemmas that are to follow are about preserving equivalence under certain transformations like scaling.

2.2.2.1 Permuting Vertices

Here we will see the theory that we tried to intuit from Figure 2.2.

Lemma 2.2.3. Let $(w, C, f, \varepsilon, k) \equiv_0 (w', C', f', \varepsilon', k)$, and $\Pi(i)$ a permutation function on the n vertices, then the permutation on the hyperedges $\Pi'(l)$ is determined by Π. Using these assumptions we construct $\bar{w}(v_i) = w(v_{\Pi(i)}), \bar{f}_\alpha = \{v_{\Pi(v_i)} | v_i \in f_\alpha\}$ and $\bar{C}(e_l) = C(e_{\Pi'(l)})$ and $\bar{w'}(v_i) = w'(v_{\Pi(i)}), \bar{f'}_\alpha = \{v_{\Pi(v_i)} | v_i \in f'_\alpha\}$ and $\bar{C'}(e_l) = C'(e_{\Pi'(l)})$. Then,

$$(\bar{w}, \bar{C}, \bar{f}, \varepsilon, k) \equiv_0 (\bar{w'}, \bar{C'}, \bar{f'}, \varepsilon', k).$$

Proof. This is actually a direct consequence of the more general fact that variable names do not alter ILP problem solutions. □

2.2.2.2 Adding Vertices

Adding a vertex without connections or weight on both sides of an equivalence, will again give us equivalent instances. Simply put, we want to show the phenomenon we can intuit from looking at Figure 2.3, but formally stating and proving it will be quite technical.

Lemma 2.2.4. Let

$$(w, C, f, \varepsilon, k) \equiv_0 (w', C', f', \varepsilon', k).$$

Define $\bar{w}(v_i) = w(v_i), \forall i \in \{1, 2, \ldots, n\}$ and $\bar{w}(v_{n+1}) = 0$ and define $\bar{C}(e_l) = C(e_l), \forall l \in \{1, 2, \ldots, m\}$ and $\bar{C}(e_l) = 0, \forall l \in \{m+1, \ldots, 2m+n\}$. Now similarly define $\bar{w'}(v_i) = w'(v_i), \forall i \in \{1, 2, \ldots, n\}$ and $\bar{w}(v_{n+1}) = 0$, and $\bar{C'}(e_l) = C'(e_l), \forall l \in \{1, 2, \ldots, m\}$ and $\bar{C'}(e_l) = 0, \forall l \in \{m+1, \ldots, 2m+n\}$. Now, either let $\bar{f}_\alpha = f_\alpha \cup \{v_{n+1}\}$ or $\bar{f}_\alpha = f_\alpha, \forall \alpha \in \{1, 2, \ldots, k\}\}$, and respectively let $\bar{f'}_\alpha = f'_\alpha \cup \{v_{n+1}\}$ or $\bar{f'}_\alpha = f'_\alpha$. Then

$$(\bar{w}, \bar{C}, \bar{f}, \varepsilon, k) \equiv_0 (\bar{w'}, \bar{C'}, \bar{f'}, \varepsilon', k).$$

Proof. Since any effect of v_{n+1}'s weight is 0, and v_{n+1} induces a cost

$$\sum_{l=m+1}^{2m+n} b_l \cdot C(e_l) = \sum_{l=m+1}^{2m+n} b_l \cdot C'(e_l) = 0,$$

we can use for all $b \in B$ that

$$\begin{aligned}
\sum_{l=1}^{2m+n} b_l \cdot C(e_l) &= \sum_{l=1}^{m} b_l \cdot C(e_l) + \sum_{l=m+1}^{2m+n} b_l \cdot C(e_l) \\
&= \sum_{l=1}^{m} b_l \cdot C(e_l) \\
&= \sum_{l=1}^{m} b_l \cdot C'(e_l) \\
&= \sum_{l=1}^{m} b_l \cdot C'(e_l) + \sum_{l=m+1}^{2m+n} b_l \cdot C'(e_l) \\
&= \sum_{l=1}^{2m+n} b_l \cdot C'(e_l)
\end{aligned} \tag{2.7}$$

to conclude the proof of the lemma. □

2.2.2.3 Scaling Costs

Now that some of the intuitively easier but abstract concepts have been formalized, we move onto more applicable concepts. The easiest to show is the one shown in Figure 2.4.

Lemma 2.2.5. Let

$$(w, C, f, \varepsilon, k) \equiv_0 (w', C', f', \varepsilon', k),$$

and assume $a \in \mathbb{Z}$, then

$$(w, a \cdot C, f, \varepsilon, k) \equiv_0 (w', a \cdot C', f', \varepsilon', k).$$

Proof. We find that with the unaltered feasible sets, we only need to prove

$$\sum_{l=1}^{m} b_l \cdot a \cdot C(e_l) = \sum_{l=1}^{m} b_l \cdot a \cdot C'(e_l), \forall b \in B$$

we get this directly by noting that we were given

$$\sum_{l=1}^{m} b_l \cdot C(e_l) = \sum_{l=1}^{m} b_l \cdot C'(e_l), \forall b \in B$$

and multiplying both sides by a. This concludes the proof. □

2.2.2.4 Adding Weights And Imbalance Constraint For Unweighted Equivalences

The following is what we have seen in Figure 2.5. We want to show that the ultimate generality of equivalence in the case of unweighted vertices will be applicable to more restricted cases.

Lemma 2.2.6. If

$$(0, C, f, 0, k) \equiv_0 (0, C', f, 0, k)$$

then for any legitimate weight function w and value for ε, we have

$$(w, C, f, \varepsilon, k) \equiv_0 (w, C', f, \varepsilon, k).$$

Proof. Since only the costs differ in the second case, we know that the sets B are equal on both sides, and since $J(b) = J'(b)$ for the respective utility functions of the first equivalence, we find that this is also true for B in the second equivalence as it is a subset of the B set in the first equivalence. □

2.2.2.5 Adding Legitimate Cost Functions On Both Sides

The final step to unlock the potential of equivalent instances is adding legitimate cost functions to both sides. This can be seen in Figure 2.6. The basic use for this, as we will see, is to replace hyperedge structures with equivalent ones, see the next subsection.

Lemma 2.2.7. Let

$$(w, C, f, 0, k) \equiv_0 (w, C', f, 0, k)$$

and some legitimate cost function $\bar{C}$ for the instances, then

$$(w, C + \bar{C}, f, 0, k) \equiv_0 (w, C' + \bar{C}, f, 0, k).$$

Proof. Since only the cost functions differ we can use

$$\sum_{l=1}^{m} b_l \cdot C(e_l) = \sum_{l=1}^{m} b_l \cdot C'(e_l), \forall b \in B$$

and add

$$\sum_{l=1}^{m} b_l \cdot \bar{C}(e_l)$$

to both sides and find

$$\begin{aligned}\sum_{l=1}^{m} b_l \cdot (C + \bar{C})(e_l) &= \sum_{l=1}^{m} b_l \cdot C(e_l) + \sum_{l=1}^{m} b_l \cdot \bar{C}(e_l) \\ &= \sum_{l=1}^{m} b_l \cdot C'(e_l) + \sum_{l=1}^{m} b_l \cdot \bar{C}(e_l) \\ &= \sum_{l=1}^{m} b_l \cdot (C' + \bar{C})(e_l).\end{aligned} \tag{2.8}$$

This concludes the proof. □

Admittedly, this is stated in a less general way (having 0 instead of ε as imbalance parameter, for example). But the lack of generality it brings makes it easier to prove most lemmas in here, and these generalizations aren't needed for the specific purposes of this text.

2.2.2.6 Replacement

Now we see how we can use these lemmas to transform instances in practice.

Theorem 2.2.8. We can take any sub-hypergraph that contains the smaller unweighted equivalence on one side, and replace it by the other side. So when we have

$$(0, \bar{C}, f, 0, k) \equiv (0, \bar{C}', f, 0, k)$$

then we can use these transformations to get

$$(w, C, f, \varepsilon, k) \equiv (w', C', f, \varepsilon, k)$$

for some larger instance $(w, C, f, \varepsilon, k)$.

Proof. We will now try to prove this in a way that is readable, we do this by showing the transitive transformations we use to build up to our target equivalence conceptually:

1. First we add unweighted unconnected vertices to increase the smaller equivalent instances to match the number of vertices. We have seen this is valid in Lemma 2.2.4;

2. Then we permute the vertices to match the relevant subgraph, this is valid because of Lemma 2.2.3;

3. Note: We could scale the costs of the smaller unweighted equivalence if needed, we can do this because of Lemma 2.2.5. However, we are allowed to assume that costs were already scaled correctly as the theorem does not mention scaling;

4. Now, we add in all the hyperedges that we are missing from the bigger instance on both sides, we can do this because of Lemma 2.2.7;

5. Then finally we can match the weights and imbalance constraints of the bigger instance on both sides, because of Lemma 2.2.6. This yields the exact equivalence we wanted to show where we replace a sub-hypergraph to an equivalent one that follows the relevant assumptions.

□

2.2.2.7 Size Three Hyperedge Splitting

We will see that the theorem is quite insightful and useful, but it is important to note that every tool we have created so far was necessary to build up the theorem. This theorem will now lead us to the following lemma:

Lemma 2.2.9. Any instance of hypergraph bisection with maximum hyperedge cardinality of 3, can be converted to an equivalent instance with maximum hyperedge cardinality of 2 with respect to bipartitioning for all three discussed metrics.

Proof. We use Lemma 2.2.2 with Theorem 2.2.8, for the cutnet and $\lambda - 1$ metrics. Also we find that for the λ-metric we can do the same but with slightly different cost function, namely the size 3 hyperedge has cost 1 and the remainder of the equivalence are the same. This concludes the proof. □

This will likely be most useful as a transformation used somewhere in recursive bipartitioning algorithms for specific cases. This is interesting because, just to name one advantage, there are $O(n^3)$ possible hyperedges of size 3, and $O(n^2)$ of size 2, which yields a factor of $O(n)$ improvement. Sometimes graph partitioning may be easier for some appropriate problem, then we can use a graph bipartitioning algorithm after preprocessing.

3 Instance Equivalence For Hypergraph Partitioning With Fixed Vertices

Now that we have a foundation for instance equivalence, we will develop it into deeper and more insightful transformations, specifically, for instances with fixed vertices.

3.1 Contraction Of Fixed Vertices In The Same Part

We will start off with some more well known equivalences. One of them being that when $|f_\alpha| > 1$ then we can contract these fixed vertices into one. To formalize this in our framework we have the following lemma:

Lemma 3.1.1. An instance of the hypergraph partitioning problem $(w, C, f, \varepsilon, k)$ is equivalent to instance $(w', C, f', \varepsilon', k)$, where all fixed vertices f_α in part α are contracted. I.e., their weights are added up for one fixed vertex and the fixed vertices are replaced by the new vertex in all hyperedges.

Proof. We can split this up by first proving that we can have one fixed vertex with all the aggregated weights in one part for the fixed vertices. Then we have all others at weight 0 and achieve instance equivalence. We first note that changing weights does not change the cost function, so we only need to prove that the set of feasible configurations is the same. We can do this by noting that any fixed vertex only has an effect on the weight constraint for its part. Define weight function

$w'(v_{f_{\alpha 1}}) = \sum_{i \in f_\alpha} w(v_i)$ for $v_{f_{\alpha 1}}$, $w'(v_{f_{\alpha j}}) = 0$ for $j \neq 1$ and $w'(v_i) = w(v_i)$ otherwise, as we wanted. So we find

$$\begin{aligned}
\sum_{i=1}^{n} p_{\alpha i} w(v_i) &= \sum_{i \in f_\alpha} p_{\alpha i} w(v_i) + \sum_{i \notin f_\alpha} p_{\alpha i} w(v_i) \\
&= w'(v_{f_{\alpha 1}}) + \sum_{i \neq 1} w'(v_{f_{\alpha i}}) + \sum_{i \notin f_\alpha} p_{\alpha i} w(v_i) \\
&= \sum_{i \in f_\alpha} p_{\alpha i} w'(v_i) + \sum_{i \notin f_\alpha} p_{\alpha i} w(v_i) \\
&= \sum_{i \in f_\alpha} p_{\alpha i} w'(v_i) + \sum_{i \notin f_\alpha} p_{\alpha i} w'(v_i) \\
&= \sum_{i=1}^{n} p_{\alpha i} w'(v_i).
\end{aligned} \tag{3.1}$$

Therefore we have the required instance equivalence. □

This together with Lemma 2.2.4, means we will assume $|f_\alpha| = 1$ without loss of generality, and a simpler notation we will use is to denote the fixed vertex of part α by f_α.

3.2 Lowering Of Fixed Vertex Weights To Increase Imbalance Constraints

Now if we assume that $w(f_\alpha) > 0$ for each part, meaning $w(f_\alpha) \geq 1$, then this gives us a nice equivalence, as we will see now.

Lemma 3.2.1. Let f_α be the unique fixed vertex of part α and let there be such a fixed vertex for each part, augmenting the weight function by $w'(f_\alpha) = w(f_\alpha) - 1, \forall \alpha \in \{1, 2, \ldots, k\}$ and taking

$$\varepsilon' = \varepsilon \cdot \left(\frac{\sum_{i=1}^{n} w(v_i)}{(\sum_{i=1}^{n} w(v_i)) - k} \right).$$

This will give us

$$(w, C, f, \varepsilon, k) \equiv_0 (w', C, f, \varepsilon', k).$$

Proof. Filling in the new ε in bounds and filling in the new weights for fixed vertices one can check that the weight constraints are the same, as follows. We have

$$\sum_{i=1}^{n} w(v_i) - k = \sum_{i=1}^{n} w'(v_i).$$

We compute

$$\begin{aligned} w'_{\max} = \left(\frac{1+\varepsilon'}{k}\right)\sum_{i=1}^{n} w'(v_i) &= \frac{1}{k}\sum_{i=1}^{n} w'(v_i) + \frac{\varepsilon'}{k}\sum_{i=1}^{n} w'(v_i) \\ &= \frac{1}{k}\sum_{i=1}^{n} w'(v_i) + \frac{\varepsilon \cdot \sum_{i=1}^{n} w(v_i)}{k \sum_{i=1}^{n} w'(v_i)} \cdot \sum_{i=1}^{n} w'(v_i) \\ &= \frac{1}{k}\left(\left(\sum_{i=1}^{n} w(v_i)\right) - k\right) + \frac{\varepsilon}{k} \cdot \sum_{i=1}^{n} w(v_i) \\ &= \left(\frac{1+\varepsilon}{k}\right)\sum_{i=1}^{n} w(v_i) - 1. \end{aligned} \tag{3.2}$$

Which is exactly as we wanted because each part also has 1 decremented from its total weight by construction, therefore we get the exact same set of feasible solutions. Since this is the only place weights are used, we find instance equivalence as we wanted to show. □

Repeated application of Lemma 3.2.1 means that we can always assume that $\exists \alpha : w(f_\alpha) = 0$, without loss of generality with respect to instance equivalence.

3.3 Removal Of hyperedges Spanning All Parts

The next part is the first point at which we will use the full capabilities of Definition 2.2.2, and we leave the exclusive use of strict equivalences. It will soon be clear why.

Lemma 3.3.1. Let $C(e_l) = 1$, when $\{f_1, f_2, \ldots, f_k\} \subseteq e_l$ and $C(e_l) = 0$ otherwise, now let $C'(e) = 0$ for all e defined on the same hypergraph. Then

$$(w, C, f, \varepsilon, k) \equiv (w, C', f, \varepsilon, k).$$

In other words, a hyperedge that spans all parts contributes a constant cost for all possible cuts, therefore making it possible to delete under instance equivalence. This comes from the fact that no matter what feasible solution we look at, the brokenness does not change. Thus, we cannot change this contribution to the utility function by making the right choices. (Note: this is not a strict equivalence.)

Proof. We have to split by metric, in each case converting the expression $b_l C(e_l)$ to an expression defined only by constants:

1. **Cutnet**: Taking $a = b_l C(e_l) = C(e_l)$ in the second part of Definition 2.2.2, gives the proof for the cutnet metric as we do not make changes to the set of feasible solutions.

2. $\lambda - 1$: We have $b_l = k - 1$ by the definition constructed in the first section. Now, taking $a = (k-1) \cdot C(e_l)$ in the second part of Definition 2.2.2, gives the proof for the $\lambda - 1$ metric as we do not make changes to the set of feasible solutions. Here, this will

3. λ: We have $b_l = k$ by the definition constructed in the first section. Now, taking $a = k \cdot C(e_l)$ in the second part of Definition 2.2.2, gives the proof for the λ metric as we do not make changes to the set of feasible solutions.

Therefore we have proven this fact for all the metrics we use in this text. □

The simple idea here is that, if we have no say in the brokenness of an hyperedge by optimizing our solution, then we might as well ignore it.

3.4 Examples Of Possible Equivalent Instances

This subsection was written because it seems that diving right into the next subsections would be too abstract and hard to follow. So, in this section we will try to do something similar to Section 2.2.1.2, however we will not only draw particular cases, but we will see one of the most essential equivalences of this entire text. However, we will refrain from proving every equivalence drawn in detail, we will, instead, intuit with words why the drawn equivalences are valid. As promised, we will take an intuitive and not too rigorous approach that still captures the entire essence of the illustrated equivalences.

A simple analogy to drive home the first small equivalence will now be discussed; One thing we have probably all encountered is the predicament of two kids fighting over a toy, and in our case, we can assure that only one kid gets the toy, we are tasked to decide who gets the toy in a way that leads to the best outcome. Now, when we look at this scenario, most of us intuit that without any further knowledge, it does not matter which kid gets the toy, since there is always one kid who is relieved to get the toy, and one kid will be sad to not have gotten the toy. This scenario is actually applicable to instance equivalences with fixed vertices. We can represent kid 1 by f_1, kid 2 by f_2 and the toy by x_1, this gives us the following partitioning problem as seen in Figure 3.1. From this point, x_i will be used to label non-fixed vertices.

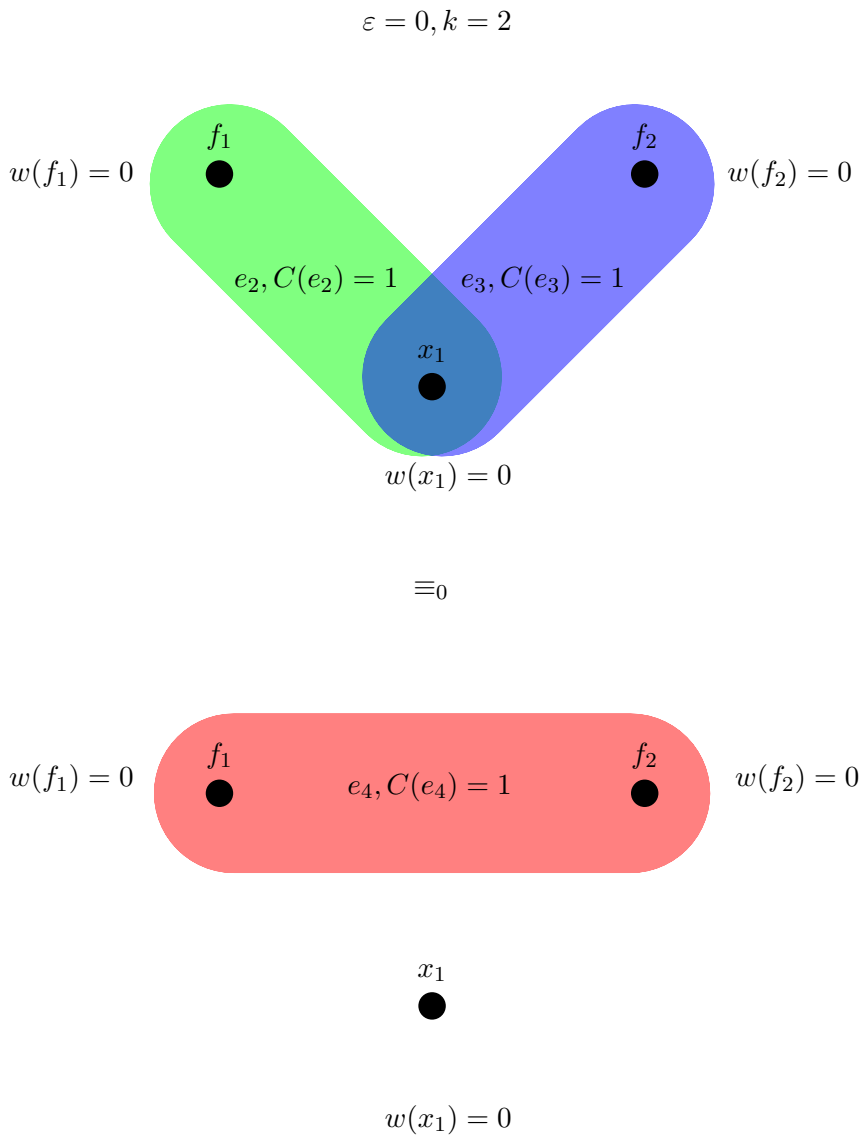


Fig. 3.1: A visual representation of the strict equivalence of both parts preferring to contain x_1 to no connectivity at all, as described.

Now, continuing this explanation, we could also imagine that the toy in question could be broken up, and in this case the toy is a ball filled with air, for example. In this case both the kids would be disappointed if the toy were to be broken, which can be represented f_1 as vertex for kid 1 and f_2 kid 2 as vertex for the left half of the toy we have x_1 and for the right half we have vertex x_2. This leads to the strict instance equivalence in Figure 3.2 for the $\lambda - 1$ and cutnet metrics. And this can be further simplified by using the instance equivalence shown in Figure 3.3 that uses Lemma 3.3.1. And there is no reason to only have 2 parts, so this generalizes further to give us the equivalence in Figure 3.4. And using Lemma 2.2.7, this gives the ultimate goal equivalence of this subsection, as is shown in Figure 3.5.

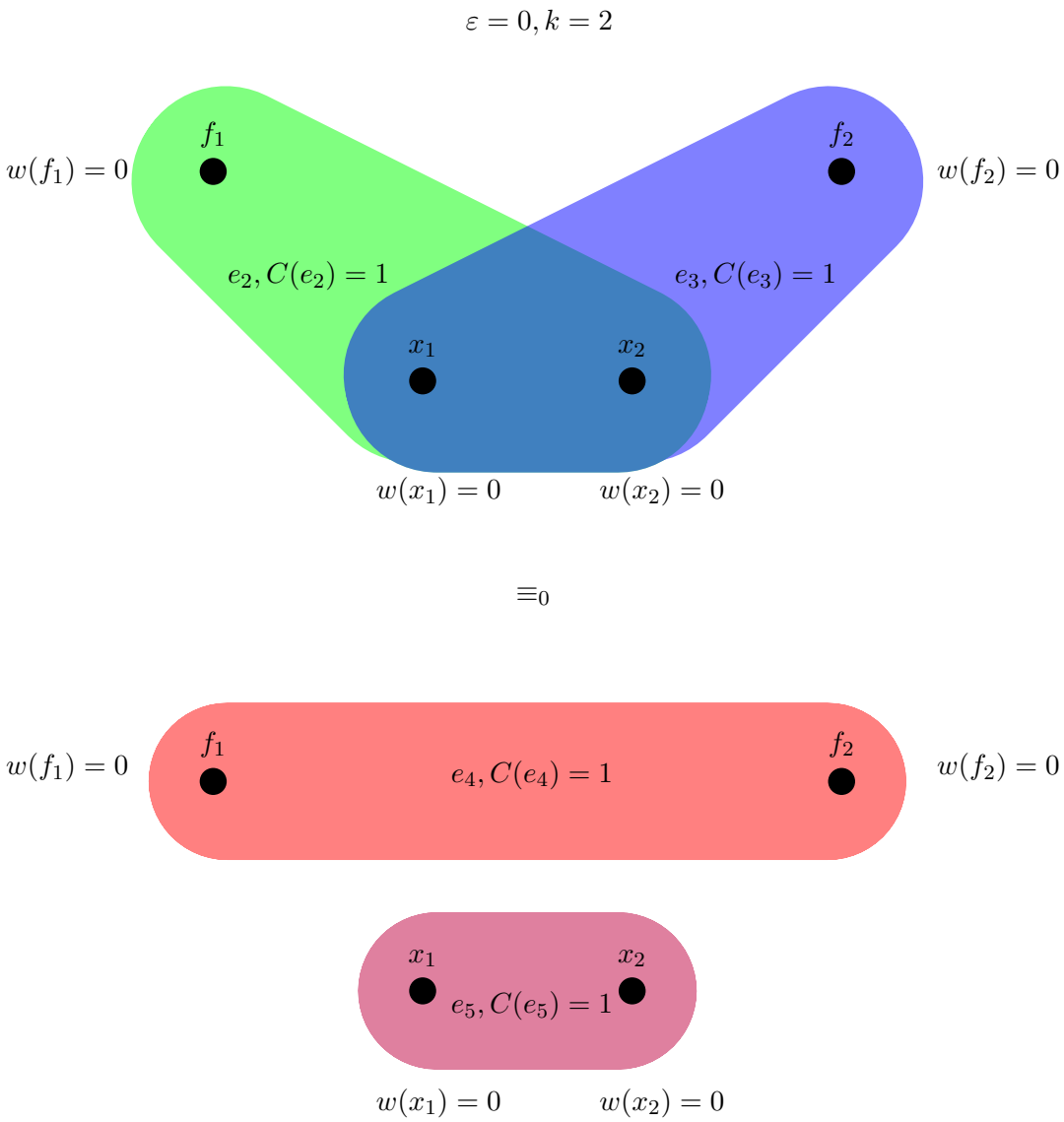


Fig. 3.2: Two fixed vertices f_1 and f_2 both connected to the same two vertices x_1 and x_2.

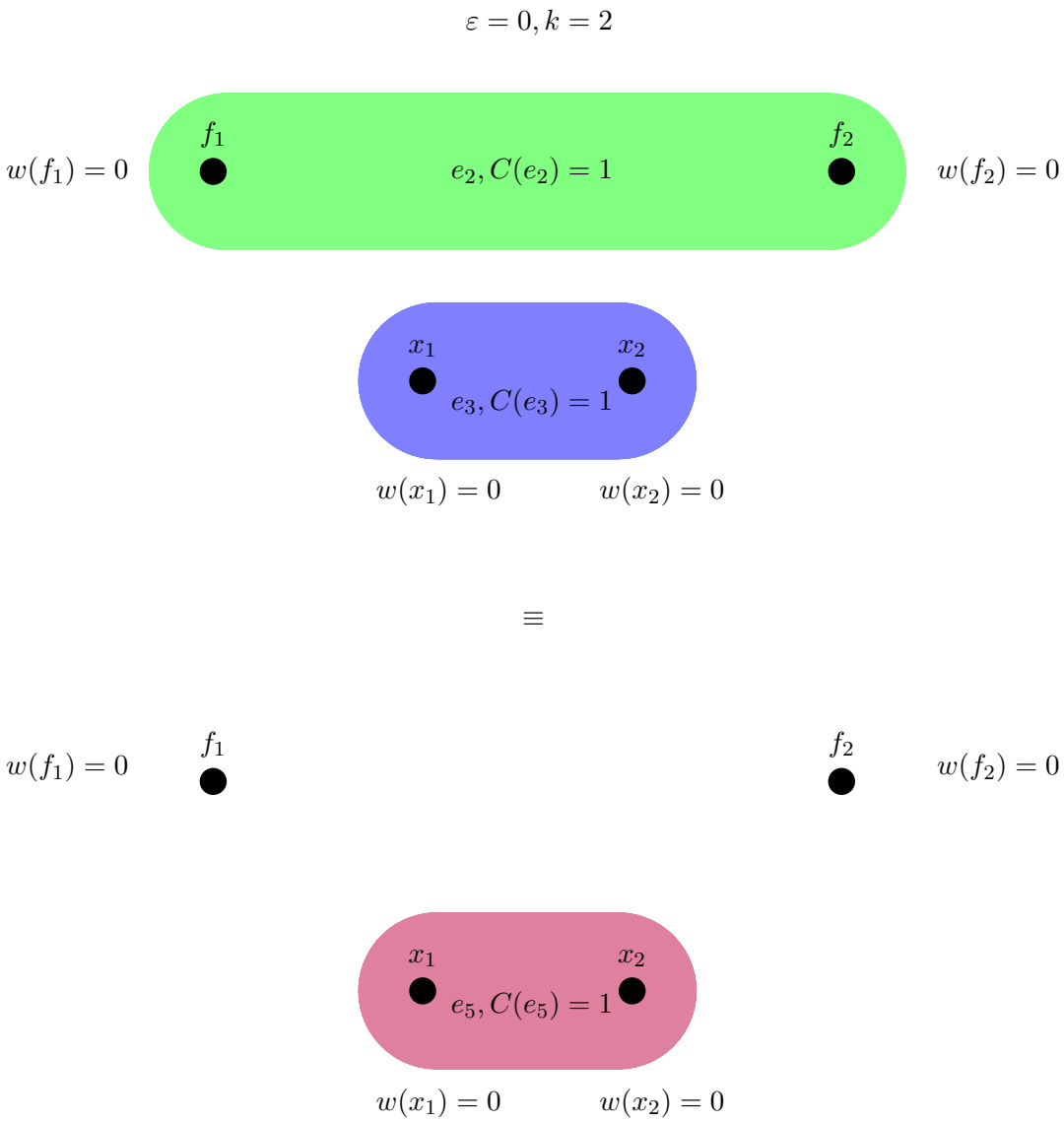


Fig. 3.3: As we can see there is no connection needed between f_1 and f_2 as the constraints force this connection to span 2 parts as shown to be irrelevant in Lemma 3.3.1.

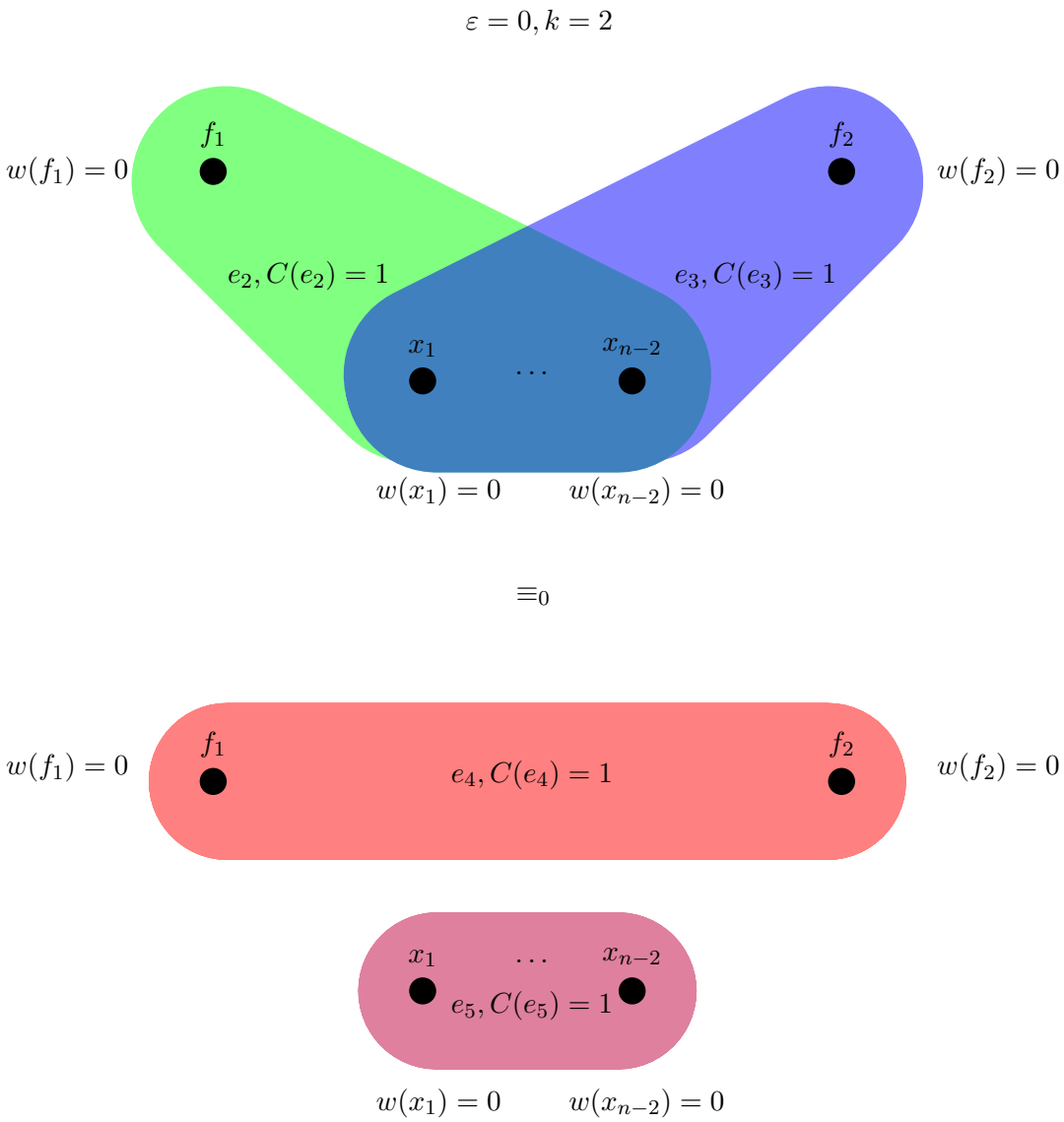


Fig. 3.4: The more general concept we try to get across in the subsection visualized.

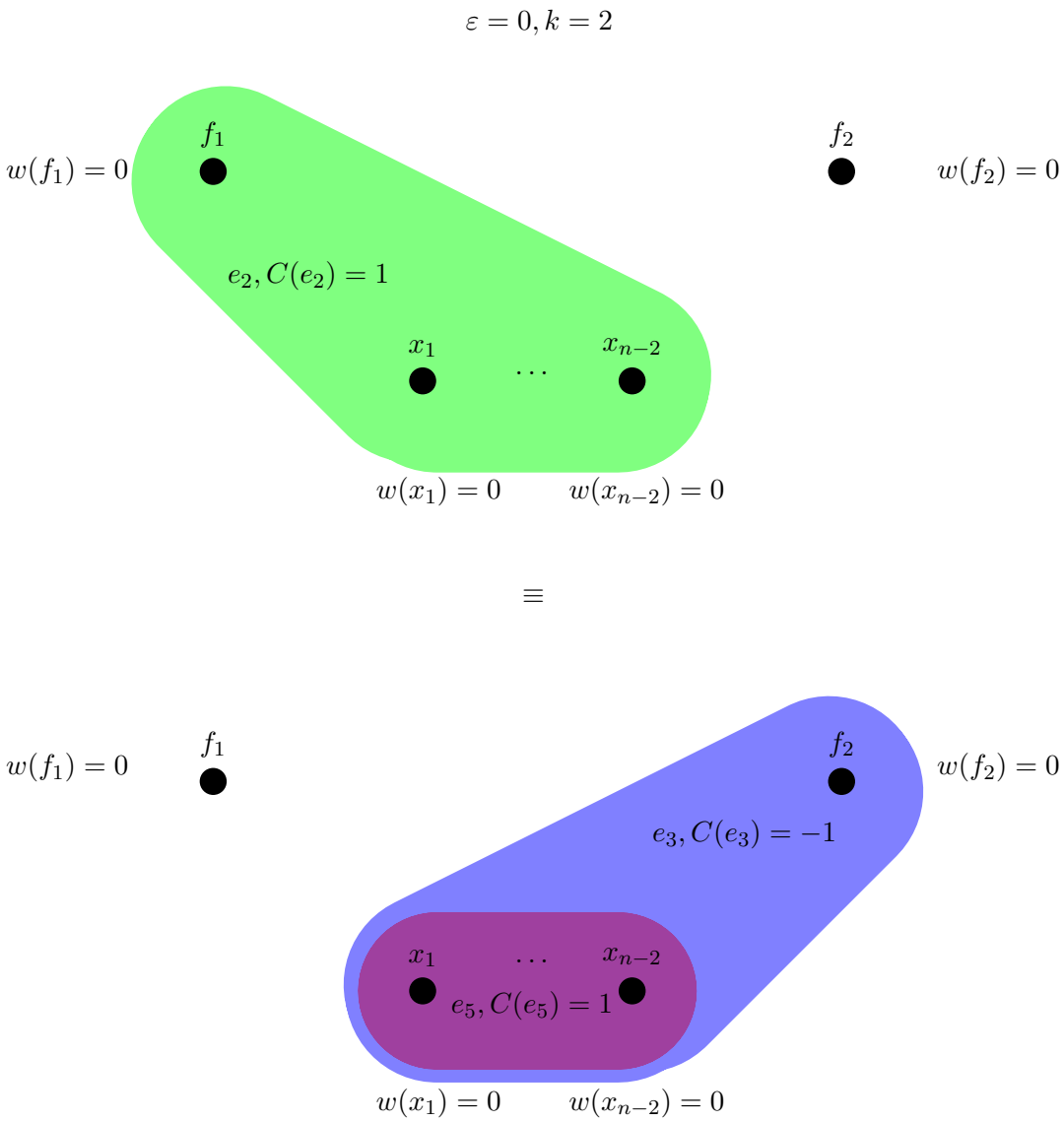


Fig. 3.5: Using Lemma 2.2.7, we can get this equivalence from the previous one.

This means that at least in bipartitioning we can replace a hyperedge containing one fixed vertex into several hyperedges that do not contain this fixed vertex with respect to instance equivalence. This together with, Lemma 3.2.1 can give us the conversion of hypergraph bipartitioning with two fixed vertices to regular hypergraph bipartitioning, which seems to be a quite strong and novel result in this area of research. The next section will generalize the non-rigorous figures and explanations in this subsection and we will prove the concepts we have now intuited.

3.5 General Case

In this section we formalize the equivalences we have seen intuited in the previous subsection, as well as the more useful augmented versions of them. This will yield very general and useful equivalences. The intuition for the more general case is that if we were not to have 2 kids, but k kids, then almost all kids will be disappointed. However, this analogy does fall apart to an extent in the case of other metrics. So now we find that we can best use more abstract descriptions and prove their validity in a more rigorous manner.

Lemma 3.5.1. Here are descriptions of equivalent instances:

1. **Cutnet**: Let $V = \{f_1, f_2, \ldots, f_k, x_1, x_2, \ldots, x_n\}$. We take $C(\{f_\alpha, x_1, x_2, \ldots, x_n\}) = 1, \forall \alpha \in \{1, 2, \ldots, k\}$ and $C(e) = 0$ otherwise. We take $C'(\{x_1, x_2, \ldots, x_n\}) = 1$, $C'(V) = k - 1$ and $C'(e) = 0$ otherwise. We also take $w(v) = 0, \forall v \in V$ and $f = \{f_1, f_2, \ldots, f_k\}$. Then

$$(w, C, f, 0, k) \equiv_0 (w, C', f, 0, k).$$

2. $\lambda - 1$ **and** λ: We have a set of vertices $V = \{f_1, f_2, \ldots, f_k, x_1, x_2, \ldots, x_n\}$. We take $C(\{f_\alpha, x_1, x_2, \ldots, x_n\}) = 1, \forall \alpha \in \{1, 2, \ldots, k\}$ and $C(e) = 0$ otherwise. We take $C'(\{x_1, x_2, \ldots, x_n\}) = k - 1$, $C'(V) = 1$ and $C'(e) = 0$ otherwise. We also take $w(v) = 0$ and $f = \{f_1, f_2, \ldots, f_k\}$. Then

$$(w, C, f, 0, k) \equiv_0 (w, C', f, 0, k).$$

Proof. We can now prove each statement:

1. **Cutnet**: Clearly put, we are guaranteed $k - 1$ broken hyperedges in the first instance which is the same as having one hyperedge of cost $k - 1$ that spans all parts in the second instance. And in both cases it costs 1 more to split the x_q vertices over multiple parts. To prove the statement we need to show that for all $b \in B$ as in Definition 2.2.2, we have the same objective function value in both cases. We see that there are only two objective function values for $(w, C, f, 0, k)$, either we have

$$\exists \alpha \forall v_i = x_j : p_{\alpha i} = 1$$

with an induced objective function value of $k - 1$ or not giving an induced objective function value of k. Since for C' we know that $b_{2^{n+k}-n-k-1} = 1$ since

it has all fixed vertices we know that C' gives an induced objective function value that is at least $b_{2^{n+k}-n-k-1} \cdot (k-1) = k-1$, and only when not all x_j's are assigned to the same part this gives a cost in C' of 1, as constructed, and therefore we have instance equivalence as we wanted.

2. $\lambda - 1$: This case is similar but here we make use of the fact that for any number

$$i = |\{\alpha | \exists v_r = x_q : p_{\alpha r} = 1\}|$$

of parts spanned by vertices $\{x_1, x_2, \ldots, x_n\}$, we have a unique objective function value. The one induced by C' is easy to evaluate as $i \cdot (k-1)$, since the $\{f_1, \ldots, f_k\}$ hyperedge in this metric will always give cost $k-1$ and the hyperedge containing only x_q vertices in this metric will add

$$b_{2^n-n-1} \cdot C'(e_{2^n-n-1}) = (i-1) \cdot (k-1)$$

therefore adding up to $i \cdot (k-1)$. Now, for c we can compute that there will be i hyperedges that span i parts, and $k-i$ hyperedges spanning $i+1$ parts, because the fixed vertices either have a label that matches any x_q label or not. This put together gives an objective function value in the $\lambda - 1$ metric of

$$\begin{aligned} i \cdot (i-1) + (k-i) \cdot i &= i^2 - i + i \cdot k - i^2 \\ &= -i + i \cdot k \\ &= i \cdot k - i \\ &= i \cdot (k-1) \end{aligned} \tag{3.3}$$

and therefore we conclude instance equivalence.

3. λ: We again use the number

$$i = |\{\alpha | \exists v_r = x_q : p_{\alpha r} = 1\}|$$

of parts spanned by vertices $\{x_1, x_2, \ldots, x_n\}$, to show that we have a unique objective function value. The one induced by C' is easy to evaluate as $i \cdot (k-1) + k$ since the $\{f_1, \ldots, f_k\}$ hyperedge in this metric will always give cost k and the hyperedge containing only x_q vertices in this metric will add

$$b_{2^n-n-1} \cdot C(e_{2^n} n - 1) = i \cdot (k-1)$$

therefore adding up to $i \cdot (k-1) + k$. Now, for c we can compute that there will be i hyperedges that span i parts, and $k-i$ hyperedges spanning $i+1$

parts, because the fixed vertices either have a label that matches any x_q label or not. This put together gives an objective function value in the λ metric of

$$\begin{aligned} i \cdot i + (k-i) \cdot (i+1) &= i^2 + i \cdot k - i^2 + k - i \\ &= i \cdot k + k - i \\ &= i \cdot k - i + k \\ &= i \cdot (k-1) + k \end{aligned} \tag{3.4}$$

and therefore we conclude instance equivalence.

□

Now we remark that using these equivalent instances and Lemma 2.2.7 we can find the following equivalent instances:

Theorem 3.5.2. The following are valid equivalences:

1. **Cutnet**: Let $V = \{f_1, f_2, \ldots, f_k, x_1, x_2, \ldots, x_n\}$. We take $C(\{f_1, x_1, x_2, \ldots, x_n\}) = 1$ and $C(e) = 0$ otherwise. We take $C'(\{x_1, x_2, \ldots, x_n\}) = 1$, $C'(V) = k - 1$, $C'(\{f_\alpha, x_1, x_2, \ldots, x_n\}) = -1, \forall \alpha \in \{2, \ldots, k\}$ and $C'(e) = 0$ otherwise. We also take $w(v) = 0$ and $f = \{f_1, f_2, \ldots, f_k\}$. Then

$$(w, C, f, 0, k) \equiv_0 (w, C', f, 0, k).$$

2. $\lambda - 1$ **and** λ: Let $V = \{f_1, f_2, \ldots, f_k, x_1, x_2, \ldots, x_n\}$. We take $C(\{f_1, x_1, x_2, \ldots, x_n\}) = 1$ and $C(e) = 0$ otherwise. We take $C'(\{x_1, x_2, \ldots, x_n\}) = k - 1$, $C'(V) = 1$, $C(\{f_\alpha, x_1, x_2, \ldots, x_n\}) = -1, \forall \alpha \in \{2, \ldots, k\}$ and $C'(e) = 0$ otherwise. We also take $w'(v) = 0$ and $f = \{f_1, f_2, \ldots, f_k\}$. Then

$$(w, C, f, 0, k) \equiv_0 (w, C', f, 0, k).$$

Proof. Combining Lemma 3.5.1 and Lemma 2.2.7, this result is obtained. □

Now using Theorem 2.2.8 with Theorem 3.5.2, Lemma 3.2.1 and Lemma 3.3.1, we find that we can always transform an instance to an equivalent instance where one of the fixed vertices has no weight and no hyperedges. This means that we could do this transformation and then partition it in k parts. Since at least one part will not contain a fixed vertex if we wouldn't include the one without connections and weight, we can take such a part and add the fixed vertex to it again and go on with

the recursive bisection. This is needed because the theory for bisection does not take unequal weight constraints into account. However, especially for bisection this will even be an exact method. It has to be noted that this does hinge on the support of negative weights by the partitioner.

4 Instance Equivalence For Coarsening

Here follow two more equivalences:

Example 4.0.1. Assume $w(v) = 0, \forall v \in V, |V| = 5$, and $C(e) = 4$ if $|e| = 5$ and $C(e) = 1$ if $|e| = 2$ and $C(e) = 0$ otherwise. Now, assume $C'(e) = 2$ if $|e| = 4$ and $C'(e) = 0$ otherwise. Then

$$(w, C, f, 0, 2) \equiv_0 (w, C', f, 0, 2),$$

for the cutnet metric.

Example 4.0.2. Assume $w(v) = 0, |V| = 7$, and $C(e) = 4$ if $|e| = 7$ and $C(e) = 1$ if $|e| = 4$ and $C(e) = 0$ otherwise. Now, assume $C'(e) = 2$ if $|e| = 6$ and $C(e) = 2$ if $|e| = 2$ and $C'(e) = 0$ otherwise. Then

$$(w, C, f, 0, 2) \equiv_0 (w, C', f, 0, 2),$$

for the cutnet metric.

Proving these is beyond the point of this section, since these instances are completely symmetric one can work them out without too much trouble because one only needs to compute for part sizes, not topology. The validity of these is nice, but not needed as we can also use the more established examples, because we will now discuss another aspect. Now if we have rating function

$$r(v_i, v_j) = \sum_{e \in I(v_i) \cap I(v_j)} \frac{C(e)}{|e| - 1},$$

then this rating function gives the same values for Examples 2.2.2, 4.0.1 and 4.0.2. On the other hand this does not hold for other popular rating functions like the inner product function, given by

$$r(v_i, v_j) = \sum_{e \in I(v_i) \cap I(v_j)} 1.$$

The interesting thing is that the rating functions that seem to experimentally yield the best results in practice seem to give exact same rating values on at least these equivalent instances, while ones found to be less effective do not. The interesting thing is that this gives a theoretical argument as to why these rating functions have nicer properties.

And even moreover, the rating function used in KaHyPar[Sch20]

$$r(v_i, v_j) = \frac{1}{w(v_i)w(v_j)} \sum_{e \in I(v_i) \cap I(v_j)} \frac{C(e)}{|e| - 1}$$

even keeps the ranking of matches for nodes in the following example. But there seems to be something to be gained as we can see if we would have multiple coarsening levels.

Example 4.0.3. Assume $w(v_5) = 3, w(v_1) = w(v_2) = w(v_3) = w(v_4) = 1, |V| = 5$, and $C(\{v_1, v_2, v_3, v_4\}) = 2$ and $C(e) = 0$ otherwise. Now, assume $C'(e) = 1$ if $|e| = 2$ and $C'(e) = 0$ otherwise. Then

$$(w, C, f, \frac{1}{7}, 2) \equiv_0 (w, C', f, \frac{1}{7}, 2),$$

for the cutnet metric.

Note that there are no coarsening methods that will truly accommodate most of the transformations we have seen for fixed vertices. This means that even if nothing here is applicable in other projects, at least this should be food for thought for engineering rating functions and especially in cases where we have fixed vertices.

Now let us evaluate this rating function for both sides of the examples. For instance, example 2.2.2, here we could see that $r(v_1, v_2) = \frac{1}{1}\frac{1}{2-1} = 1$ on the left-hand side, on the right-hand side we get $r(v_1, v_2) = \frac{1}{1}\frac{2}{3-1} = 1$. Now this is also the case for the inner product rating function, but we can see where the inner product breaks down while evaluating the rating function for Example 4.0.1. In this example we see that the rating function $r(v_1, v_2)$ gives $\frac{1}{1}\frac{4}{5-1} + \frac{1}{1}\frac{1}{2-1} = 2$ since they have 1 connection in common of size 5 and 1 of size 2, on the other hand in the second case we get $r(v_1, v_2) = \frac{1}{1}3\frac{2}{4-1} = 2$, because these vertices have 3 hyperedges in common of size 4. But that also shows us that in the first instance, the vertices have 2 connections in common, but in the second case they have 3 connections in common, therefore meaning that there are equivalences for which the rating function r is equal for some equivalences while the inner product is not. Note here, that I have not classified all equivalences so I cannot say that r is always more resistant to instance equivalence in evaluation, but it certainly seems so, and this seems to

explain why this rating function is preferable to others. Again for Example 4.0.2, in the first case we get $r(v_1, v_2) = \frac{1}{1}\frac{4}{7-1} + \frac{1}{1}10\frac{1}{4-1} = 4$, and in the second case we get $r(v_1, v_2) = \frac{1}{1}\frac{2}{2-1} + \frac{1}{1}5\frac{2}{6-1} = 4$.

Up to this point, the factor of w values have not been needed to keep the invariant ratings. But in coarsening, we need to remember that only the top match counts, so if we just care about the top rated match for each vertex, r is even top match preserving for example 4.0.3, which at this point can be worked out by the reader. But in general it seems we want to choose a rating function that preserves some properties under equivalences under some definition of equivalence, and a good start seems to enforce equal evaluations under this very strict definition of equivalence. One could conjecture that r seems to maximize resistance to strict equivalences, but for now this question is out of reach.

5 Algorithms

We will now see the pseudocode for the mathematical descriptions of algorithms. We start off by looking at algorithms that were tested the most (therefore stable) and were most relevant to PMondriaan; Then we will see how we can modify PMondriaan to handle negative edge costs; Finally, we will see procedures for the partitioning algorithms that can partition with fixed vertex constraints.

5.1 Algorithms For PMondriaan

The most relevant algorithms for the goal of improving running time for PMondriaan are simplification algorithms, but there are some other implementations that are helpful that we will discuss here that were also implemented in the modification of PMondriaan.

5.1.1 Sequential Hyperedge Simplification

As described before, the hyperedge simplification algorithm takes a hypergraph and finds all duplicate sets of vertices in hyperedges and makes a representative

hyperedge with the sum of those costs. It was implemented as follows.

Algorithm 3: Simplification of duplicate nets, we only keep the net with the highest id.

Input: Hypergraph $H = (V, E)$.

```
1  initialize Hashmap costs = 0
2  initialize Hashmap ids
3  forall e ∈ E do
4  |  costs[e.vertices] ← costs[e.id] + C(e)
5  |  ids[e.vertices] ← e.id
6  forall e ∈ E do
7  |  if ids[e.vertices] = e.id then
8  |  |  C(e) ← costs[e.vertices]
9  |  if ids[e.vertices] ≠ e.id then
10 |  |  remove e
11 return H
```

Output: Simplified hypergraph.

This algorithm starts off by initializing hashmaps in line 1 and 2 because we want to efficiently map every vector of vertices to the same keys to sum costs and find duplicate edges to remove. Then we use these maps to sum the costs and choose the edge we keep from deletion to serve as representative for all its duplicates in lines 3,4 and 5. Having found the sums and the edges to keep, we delete all duplicate edges and update the costs of all edges in lines 6 to 10. After this the input hypergraph has been modified by removing duplicate edges while keeping an equivalent partitioning problem, therefore we return H in line 11.

5.1.2 Parallel Hyperedge Simplification

Before we can see how this algorithm works it is helpful to define what pins are. Here, pins are information about which vertices are contained in some hyperedge. Here, external pins are information about which vertices that are not part of the local hypergraph are contained in hyperedges of a local hypergraph. Parallel simplification is supposed to do the same as sequential simplification, but here an obstacle is that vertices are assigned to different computers. This can help with lower maximum memory usage for each processing unit because we only need to communicate the id for each hyperedge instead of the entire global hypergraph. After this, we can add external pins by only requesting the pins of local hyperedges, and not all hyperedges of the combination of all local hypergraphs. This means that we can simplify with more limited communication in parallel, which will cause each local hypergraph to be smaller and therefore more efficient to communicate and combine for sequential

partitioning.

Algorithm 4: Simplification of duplicate nets in parallel.

Input: Hypergraph $H = (V, E)$ with distributed vertices.

```
1  communicate pins
2  add external pins locally
3  initialize Hashmap costs
4  initialize Hashmap ids
5  forall e ∈ E do
6  |  costs[e.vertices] ← costs[e.id] + C(e)
7  |  ids[e.vertices] ← e.id
8  find removal consensus
9  forall e ∈ E do
10 |  if ids[e.vertices] = e.id then
11 |  |  C(e) ← costs[e.vertices]
12 |  if ids[e.vertices] ≠ e.id then
13 |  |  remove e
14 return H
```

Output: Simplified hypergraph.

This algorithm is the parallel version of the one seen before it. In this case we do have to assume we are missing indices locally that in the global hypergraph are contained in the local edges. Therefore we communicate all the global edge ids that are contained on each processing unit and respond to each such message with the local pins of that processing unit with respect to the edges. Then we add those pins to local edges which constitutes line 1 and 2 of the procedure. Then we do the same as in the sequential case for lines 3 to 7. Each part then knows duplicates locally, but we need to have each processing unit remove the same edges, therefore we communicate which edges are removed to each other processing unit, and we find consensus for removal by letting the lowest processor id that has any given edge decide the removal, this is seen in line 8. After the consensus was found, we follow the same steps of removal as we would sequentially from lines 9 to 14.

5.1.3 Breaking Triples

The algorithm induced by Lemma 2.2.2 and Theorem 2.2.8. It is implemented as follows.

Algorithm 5: Breaking triples.

Input: Hypergraph $H = (V, E)$.

```
originalsize ← |E|
forall j ∈ {1, 2, ..., originalsize} do
    if e_j.vertices.size = 3 then
        add three edges of size 2 at C(e_j) cost at back of edge list
    if e_j.vertices.size ≠ 3 then
        C(e_j) ← 2C(e_j)
remove all edges of size 3
return H
```

Output: Hypergraph without triples with twice the cut sizes in bisection.

Note that we could also add edges with costs of value one half, getting equal cut sizes, but the way shown here is equivalent and keeps integer costs.

5.1.4 Rating Function

While making the above modifications for PMondriaan a difference was found in performance because of the dependence of the rating function on the size of the list of edges incident to a vertex. Therefore we take a different approach for rating functions in the sequential coarsening part of PMondriaan, namely

$$\frac{1}{\min(w(v_i), w(v_j))} \sum_{e \in I(v_i) \cap I(v_j)} \frac{C(e)}{|e| - 1}.$$

In label propagation an edit was made to use a rating function as found in the literature [Hen15] as opposed to ignoring edge costs which is only good when simplification does not occur, the rating function used is

$$\sum_{e \in I(v_i) \cap I(v_j)} \frac{C(e)}{|e| - 1}.$$

5.2 Algorithms For Negative Edge Costs

We have seen that we would like to have access to negative edge costs, here we discuss what augmentations are needed to accomplish this goal. Because of limited

time, we only look at the sequential version of PMondriaan. One big modification to be made in PMondriaan to make this possible is to enable reading custom costs and to always copy costs when transferring edge information. However, this is not the only augmentation needed.

5.2.1 Coarsening And Label Propagation

The choice made here is to prefer not matching a vertex when the highest matching rating value is negative, which is likely optimal, but this is only conjecture. Therefore we mostly keep everything the same in coarsening, requiring a non-negative rating between the vertices by using -1 as a placeholder for maximum rating. Label propagation is augmented by having another placeholder than -1, this is needed because we want to choose the highest rated label even when it is negative.

5.2.2 Refinement

In refinement with KLFM the only modified procedure is the computation of gain structure bounds. This is done as follows, but could also be prevented using data structures like linked lists instead of arrays; We take absolute costs instead of costs in the total sum of costs. This is important as we could have gains outside of the bounds we normally would compute because of the implications of having arbitrary negative costs.

5.3 Algorithms For Fixed Vertices

Now that we have a version of PMondriaan that is able to handle negative costs, we can implement algorithms for bisection with fixed vertices using it. We discuss how one could implement conventional methods as well as the one described in the previous chapter. We also discuss a possible algorithm for general partitioning with fixed vertices.

5.3.1 Conventional Bisection Algorithms

The literature shows that graph growing as initial partitioning as well as ignoring fixed vertex moves in coarsening and refinement currently gives the best results[PE16]. One way to model this in our modified version of PMondriaan is to make an edge of cost -1000000 between the canonical fixed vertices of each part, which works for smaller hypergraphs and small costs. However, this still makes

it possible for fixed vertices to be assigned to the same part in label propagation, which is then always resolved in refinement. This label propagation is analogous to graph growing and is used to save time in implementation. We then use this transformation in two ways: the strict version where we do not allow fixed vertex labels to change in initial partitioning, and the weak version where we do allow label changes for fixed vertices. The strict version is analogous to conventional fixed vertex hypergraph partitioning algorithms.

5.3.2 Equivalence Based Bisection Algorithm

The transformation described in the end of the previous chapter is implemented as follows.

Algorithm 6: Fixed vertex hypergraph bipartitioning problem preprocessing algorithm.

Input: Hypergraph $H = (V, E)$.

1 merge fixed vertices by part
2 find lowest weight fixed vertex
3 decrease weights and edit ε to obtain a zero-weighted fixed vertex
4 transfer hyperedges from zero weighted fixed vertex to the other
5 remove the zero weighted fixed vertex
6 **return** H

Output: A hypergraph without fixed vertices, when the one remaining is ignored as having only one fixed vertex in bipartitioning allows us to ignore it. We can ignore the only fixed vertex in bipartitioning because we can partition without restrictions by inverting labels if the only fixed vertex is mislabeled.

As we have seen all the procedures alluded to in the algorithm in chapter 3 by use of chapter 2, we won't discuss it further.

5.3.3 Equivalence Based General Partitioning Algorithm

One way to extend the idea of this transformation to general partitioning is to do a version of recursive bisection where one fixed vertex is eliminated as seen in chapter 3. Then a k-way partition can be made as there are $k - 1$ fixed vertices in the augmented hypergraph. With $k - 1$ fixed vertices one of the parts must have no fixed vertices, and therefore we use this part to assign the eliminated fixed vertex to. We do this recursively until we finally finish the procedure by bisection. This idea was not implemented or benchmarked, however. One big problem is that the biggest edges take up disproportionate processing time. For higher k the usage of

weight constraints gets more favorable; This gives one small advantage for k-way partitioning with $k > 2$. Therefore PMondriaan also includes the option to ignore parts of large edges in coarsening computation.

6 Results

All results were obtained from partitioning sparse matrices in the row-net model. We use geometric means to give averages over results that vary in size, for some set of numbers $\{x_1, x_2, \ldots, x_n\}$ it is computed by taking

$$(x_1 \cdot x_2 \cdot \cdots \cdot x_n)^{1/n}.$$

6.1 Core Results

The data used in this section comes from Suite Sparse[Kol+19], which is a database of benchmark matrices. To run the main comparison for the project before optimizations of this thesis we use known configurations and files ($k = 2, p = 32$), the choice of bipartitioning and number of processors is made because it is sufficient to show the target observations:

Matrix	**Parameters**	
	Sample Size	Max Cluster Size
tbdlinux	1500	20
Si34H36	10000	20
marine1	5000	50
atmosmodd	25000	50
delaunay_n21	50000	50
hugetrace-00000	180000	150
StocF-1465	10000	50
asia_osm	700000	150
CurlCurl_4	40000	50
Emilia_923	8000	50
HV15R	20000	50

Tab. 6.1: In this table we see the sparse matrices we use for the benchmark. The filename is accompanied by the sample sizes used in parallel coarsening as described in the previous thesis about PMondriaan[Ber20]. The maximum cluster sizes used are also given for limiting the number of vertices that are contracted into one. These were chosen because they were well studied in the original PMondriaan thesis[Ber20].

These matrices have the following properties.

Matrix	Properties		
	Rows	Columns	Nonzeros
tbdlinux	112 757	20 167	2 157 675
Si34H36	97 569	97 569	5 156 379
marine1	400 320	400 320	6 226 538
atmosmodd	1 270 432	1 270 432	8 814 880
delaunay_n21	2 097 152	2 097 152	12 582 816
hugetrace-00000	4 588 484	4 588 484	13 758 266
StocF-1465	1 465 137	1 465 137	21 005 389
asia_osm	11 950 757	11 950 757	25 423 206
CurlCurl_4	2 380 515	2 380 515	26 515 867
Emilia_923	923 136	923 136	40 373 538
HV15R	2 017 169	2 017 169	283 073 458

Tab. 6.2: Some properties of the benchmark matrices, ordered by number of nonzeros.

And we also start the initial partitioning phase if there are fewer than 200 vertices left after a coarsening step. Using these configurations we get results on quality differences induced by the new scoring function used in sequential coarsening, as well as the running times of the total partitioning and the initial partitioning phase performance. Initial partitioning is highlighted because it was the main focus of the original research question of the thesis.

Matrix	Initial Part. Time		Total Time		Cut size	
	Old(ms)	New(ms)	Old(ms)	New(ms)	Old	New
tbdlinux	10077	7769	19455	22672	25640	25059
Si34H36	10799	7126	29824	36445	18176	18300
marine1	8781	41	25280	7365	2099	2111
atmosmodd	18978	62	78015	2586	17186	17190
delaunay_n21	11628	11	132592	20507	2878	2932
hugetrace-00000	7522	26	289011	88295	1532	1528
StocF-1465	32252	216	115321	17972	4808	4823
asia_osm	44	14	491754	121550	40	44
CurlCurl_4	57626	187	178316	51273	25254	25515
Emilia_923	27668	235	69730	8159	27234	27219
HV15R	95121	701	299293	94687	62243	62270
Geom. Mean	11181	171	99298	25076	6290	6356

Tab. 6.3: Benchmarks were run using PMondriaan before and after augmentations. We see that the main goal of improving initial partitioning performance was reached, while keeping the other metrics more or less the same or easy to improve. Each result is the highest quality result of 10 runs.

It is important to see that major inefficiencies that are still left in PMondriaan, are inefficiencies of data structures. This should give us some confidence as the theoretical improvements that were made already give major performance boosts in most cases. The minor difference in quality could be fixed by tracking edge multiplicities and adding them together to recover the degree of each vertex in coarsening. We should conclude from this that the major overhaul of data structures used in PMondriaan, though it is a big task, would not be very theoretically demanding. And we see that most of this is due to the specific systems used to compile and run the code on the supercomputer Snellius used in our experiments. All together, the main conclusion is that the main task of initial partitioning phase performance improvement was successful. While running the code a glaring problem for overall quality is that the design of the parallel local search algorithm seems to have an issue with finding suitable improvements when the input partitioning is close to the imbalance constraints. Also when we choose to send the entire hypergraph to each process once we start the sequential phase, we limit ourselves. The limitation comes from the fact that we want to use many processors for bigger problems, but this increases the size of merged coarsened hypergraphs because we have more local hypergraphs.

6.2 PMondriaan Benchmark Sample

Here, we will use a different set of matrices from the benchmark set that was used in the benchmarking of KaHyPar[Sch19]. The matrices we use have the following properties:

Matrix	Properties		
	Rows	Columns	Nonzeros
Emilia_923	923 136	923 136	40 373 538
kkt_power	2 063 494	2 063 494	12 771 361
...bug8.cnf.dual	521 179	13 378 617	39 204 907
...bug7.cnf.dual	521 147	13 378 010	39 203 144
...bug8.cnf.hgr	13 378 617	1 042 358	39 204 907
nlpkkt120	3 542 400	3 542 400	95 117 792
wb-edu	9 845 725	9 845 725	57 156 537
RM07R	381 689	381 689	37 464 962
dielFilterV2clx	607 232	607 232	25 309 272
msdoor	415 863	415 863	19 173 163

Tab. 6.4: Some properties of the benchmark matrices, in no particular order. We see these are some more varied properties.

While most of the optimizations were made for the Cartesius supercomputer, the experiments were run on the Snellius system on a node with AMD Rome 7H12 64 Cores/Socket, 2.6GHz, 8 GiB 3200MHz, DDR4 per task. With two units per node we have access to 128 such tasks. The following comparison in bisection with Mondriaan[VB05] and Patoh[ÇA11] was obtained. We use some of the larger benchmark hypergraphs from the KaHyPar benchmark set a for comparison. In the following results T, denotes timeouts and M denotes out of memory errors. In the case of the nlpkkt120 and sat matrices the full 128 tasks were used whereas for all others 64 were used. The edge coarsening limit on sat files was 10 and no restriction was enforced otherwise. We highlight the best running times and cut sizes, all for bisections. Additionally, Patoh was used through the Mondriaan software; In both cases of Mondriaan usage, we set it to partition one dimensionally using the onedimcol setting.

Matrix	Patoh		Mondriaan		PMondriaan	
	Cut	Time(s)	Cut	Time(s)	Cut	Time(s)
Emilia_923	**20955**	**12.41**	25395	23.75	28119	13.53
kkt_power	11579	**9.854**	**10013**	31.50	17842	40.72
...bug8.cnf.dual	**2876**	**143.1**	T	T	39898	600.0
...bug7.cnf.dual	M	M	M	M	**87872**	**671.8**
...bug8.cnf.hgr	M	M	M	M	**507413**	**714.4**
nlpkkt120	**2876**	42.82	M	M	138954	**32.80**
wb-edu	M	M	M	M	M	M
RM07R	**25232**	19.78	26093	21.14	26283	**5.888**
dielFilterV2clx	**2612**	9.979	2638	14.41	3312	**7.941**
msdoor	**1806**	5.960	**1806**	7.077	2793	**2.638**

Tab. 6.5: In the table above we denote software failure by memory constraints by M, and we denote a running time longer than an hour (also failed execution) by T. Here we see results and running time of Mondriaan with onedimcol option with the normal algorithm and using Patoh. This is compared to PMondriaan, and gives some of the current issues with PMondriaan. However, we also can see a clear strength as PMondriaan using an edge size limit for coarsening has two cases where it is the only software to yield a partitioning. Best results in time and quality are highlighted. Each result is the highest quality result of 10 runs.

An attempt was also made to run PMondriaan on twitter7 and AGATHA_2015, which are some of the biggest matrices on Suite Sparse[Kol+19]. However, distributed reading of these files brought forth an out of memory exception, even with 4096 tasks using 8GiB memory.

It is important to note that the results from the PMondriaan benchmark sample of Table [**tab:early**] were chosen by the difficulty they seemed to generate for the PMondriaan software as found before making the described changes or to show its strength. That being said, the results show that there is certainly a use-case for PMondriaan here. The results show that there are cases where other software fails and PMondriaan doesn't. But, we do also see that quality is dependent on the sampling parameters in the parallel coarsening phase. But, in many cases PMondriaan is now the fastest algorithm, even for these problematic cases. Another important notion here is that PMondriaan parameters used in most cases were not optimized, but this should be expected for most end users. Not to mention that it is quite likely that the rating function used in sequential coarsening and label propagation is not yet optimized. The majority of time in PMondriaan is currently used in parallel uncoarsening, which could likely be optimized and is most likely an artefact of switching from Cartesius to Snellius. Another major issue is the efficiency of the sequential version of PMondriaan, this is most likely due to the efficiency

of the data-structures used (vectors instead of arrays) and their allocations and iterators. Last, as the results show there is still an issue with reading a hypergraph from a file with multiple tasks for example, twitter7, it should be possible to read with 4096 tasks without memory overflow. This could also be prevented by having multiple phases in parallel coarsening where we merge local hypergraphs pairwise until the entire coarsened hypergraph can be held in memory of one processing unit.

6.3 Breaking Triples Result

In the following sections we revert to using matrices from Suite Sparse in benchmarking[Kol+19].

Runs	Average Pins Without Breaking	Average Pins With Breaking
10	9843	**6205.6**

Tab. 6.6: Here we look at 10 runs of the effects of breaking triples and simplification in the coarsened version of the hypergraph induced by the Emilia_923 matrix. The best result is highlighted.

We also find that the sum of all the costs of edges of cardinality larger than three makes up less than 5% of the sum of all costs in all cases.

Apart from this result we find that there are simply not too many edges of size 3 after coarsening. However, in the case shown, we clearly find that we can reduce the problem of initial partitioning to graph partitioning with approximation that deviates by at most 5% of the sum of all costs. Other than that, we can obviously reduce 3-uniform hypergraph bisection to graph bisection, even when used as a subroutine for recursive bisection. But it is still not clear when to effectively use the breaking algorithm, but there does seem to be a use-case for it.

6.4 Hypergraph Partitioning With Fixed Vertices Method Comparison

For these results we need smaller matrices to use for the benchmark, here are some of their properties.

Matrix	Properties		
	Rows	Columns	Nonzeros
power9.mtx	155376	155376	1887730
nv1.mtx	75468	75468	1635003
imagesensor.mtx	118758	118758	1446396
radiation.mtx	223104	223104	5526637
Emilia_923.mtx	923136	923136	40373538
tbdlinux.mtx	112757	20167	2157675
CoupCons3d.mtx	416800	416800	17277420
neos1.mtx	131581	133473	599590
usroads.mtx	129164	129164	330870
mosfet2.mtx	46994	46994	1013062

Tab. 6.7: Some properties of the benchmark matrices, in no particular order. We see smaller sizes than before.

As there should not be a difference between weak and strict conventional methods for hypergraph partitioning with fixed vertices, we first compare the equivalence method in outcome quality. Half of the matrices used are VLSI matrices and half are varied in both size and origin to be more representative. The benchmark set was created by assigning each vertex to a random part and then making three fix files, one with 0.3% of vertices randomly selected to be fixed, one with 3% and one with 8% from the predefined set of labels. So first we have the 0.3% case, in all cases here we take the minimum of 3 runs we highlight results that are best:

Matrix	Weak Conventional Quality	Equivalence Based Quality
power9.mtx	7123	**7025**
nv1.mtx	**5972**	6555
imagesensor.mtx	7591	**7555**
radiation.mtx	**14421**	16053
Emilia_923.mtx	79857	**78741**
tbdlinux.mtx	**15446**	18464
CoupCons3d.mtx	37076	**36871**
neos1.mtx	66785	**66656**
usroads.mtx	**644**	698
mosfet2.mtx	**4405**	4586

Tab. 6.8: The above are cut sizes for fixed vertex partitioning where 0.3% of the vertices are fixed vertices. The weak conventional method is implemented as described in the chapter on algorithms for fixed vertex hypergraph partitioning. The equivalence based method is the new transformation based algorithm. The best results are highlighted. Each result is the highest quality result of 3 runs.

And then we have the results for 3% fixed:

Matrix	Weak Conventional Quality	Equivalence Based Quality
power9.mtx	35903	**35615**
nv1.mtx	30133	**29935**
imagesensor.mtx	28871	**28597**
radiation.mtx	**90581**	90743
Emilia_923.mtx	**452763**	453657
tbdlinux.mtx	**20506**	21848
CoupCons3d.mtx	223588	**223556**
neos1.mtx	**67671**	69790
usroads.mtx	4855	**4754**
mosfet2.mtx	19425	**19352**

Tab. 6.9: The above are cut sizes for fixed vertex partitioning where 3% of the vertices are fixed vertices. The weak conventional method is implemented as described in the chapter on algorithms for fixed vertex hypergraph partitioning. The equivalence based method is the new transformation based algorithm. The best results are highlighted. Each result is the highest quality result of 3 runs.

And then we have the results for 8% fixed:

Matrix	Weak Conventional Quality	Equivalence Based Quality
power9.mtx	73566	**73404**
nv1.mtx	53216	**53198**
imagesensor.mtx	**55334**	55337
radiation.mtx	161846	**161833**
Emilia_923.mtx	**754701**	756039
tbdlinux.mtx	**25061**	25394
CoupCons3d.mtx	**354368**	354655
neos1.mtx	73888	**71223**
usroads.mtx	**11450**	11455
mosfet2.mtx	**32643**	32700

Tab. 6.10: The above are cut sizes for fixed vertex partitioning where 8% of the vertices are fixed vertices. The weak conventional method is implemented as described in the chapter on algorithms for fixed vertex hypergraph partitioning. The equivalence based method is the new transformation based algorithm. The best results are highlighted. Each result is the highest quality result of 3 runs.

Here we compare strict and weak conventional partitioning with fixed vertices with equivalence based methods in initial partitioning quality. To not be too restrictive the 0.3% fix files were used and since this is the fix file used, we took the minimum of 10 runs:

Matrix	Weak Quality	Strict Quality	Equivalence Based Quality
power9.mtx	10061	9823	**8961**
nv1.mtx	7964	8257	**7680**
imagesensor.mtx	12329	11740	**10633**
radiation.mtx	20030	19446	**17636**
Emilia_923.mtx	107319	110211	**105171**
tbdlinux.mtx	24199	**21146**	23204
CoupCons3d.mtx	49264	44436	**43532**
neos1.mtx	75813	75907	**75568**
usroads.mtx	1017	1070	**824**
mosfet2.mtx	6595	6225	**5892**
Geom. mean	16254	19980	14796

Tab. 6.11: The above are cut sizes in the initial partitioning phase for fixed vertex partitioning where 0.3% of the vertices are fixed vertices. The weak conventional method is implemented as described in the chapter on algorithms for fixed vertex hypergraph partitioning, the same goes for the strict method. The equivalence based method is the new transformation based algorithm. The best results are highlighted. Each result is the highest quality result of 10 runs. The main difference between the weak and the strict version is that fixed vertices are never allowed to be moved to another part in any phase in the strict version. The weak version allows fixed vertices to move during the initial partitioning phase.

Here, there are many questions that are left unanswered. For example, what do these results look like for actual real-world benchmarks. But, for what we can now see, the results are very promising. We see that the end results are very similar between all methods, but we see very decisive results in table 6.11 for the algorithm run up to the initial partitioning stage. This may hint that coarsening rating functions should take the equivalence found into account, giving better results for initial partitioning. But then we may find that the refinement favors the conventional methods, so that the optimal use of methods may lead to results that are better than both pure methods by switching paradigm between phases. This is all speculative, but it could be interesting to study further. This concludes the obtained results.

7 Conclusion

7.1 Summary

We have found that hypergraph partitioning is a problem particularly suited to parallel computation. We found that PMondriaan solves some issues effectively but lacks on many others, generally having mixed results. We also found that PMondriaan is not yet ready for the purpose it was created for, but we also see clear paths towards these goals. We also have introduced the notion of instance equivalence in a specific manner and its application to hypergraph partitioning with fixed vertices. We found that instance equivalence can be useful, but it is not yet clear how to best take advantage of the concepts. We also saw how to practically implement all of the theoretical algorithms used and have seen some results that follow from the implemented software. Last we also saw that there may be implications of the theory that apply to other similar projects, like what coarsening rating function is used.

7.1.1 Future Work

The following steps give a way to make PMondriaan very competitive with state of the art software. To make PMondriaan very efficient, it is likely enough to do step 1, mostly.

1. Using the c++ compiler available on Snellius, we find that a benchmark about filling a matrix structure that uses a 2d array takes 4.088 seconds and the same benchmark using vectors that are instantiated using push back instead of claiming memory up front takes 16.858 seconds for the same tasks. This is relevant because this is the main backbone of how hypergraph objects function in PMondriaan. This indicates a larger inefficiency in PMondriaan opposed to Mondriaan, which could be solved by building the same use of arrays and memory management in Mondriaan into PMondriaan. This should likely make PMondriaan a noticeable factor faster.

2. Another big improvement would be to make a way to run PMondriaan more effectively without tuning parameters of the configuration file.

3. Then, it would also be good to be able to customize weights and costs in the matrix files instead of always using unit costs or one of two weight options.

4. There also seems to be an inefficiency in reading files in PMondriaan, that could be a problem in Snellius or Bulk.

5. To extend the range of problems that PMondriaan can solve that no other method can, there also seems to be a need to read files more effectively, but also stopping merging all local hypergraphs at once after parallel coarsening, this in combination with parallel simplification that was made here. Also it is best in such a case to deal with large edges effectively by limiting their use in coarsening by the first few indices in the edge.

6. It is best to compute degrees as used in the original PMondriaan scoring function after simplification.

7. It could also be nice to be able to use different coarsening scoring functions.

8. It would be good to find the best ways to use cardinality 3 edge breaking.

9. After adding custom costs, negative costs could be added and fixed vertices could be easily added.

10. Overall, it seems good to test the quality difference between Patoh and PMondriaan on a large benchmark set.

Bibliography

[BBB18] Jan-Willem Buurlage, Tom Bannink, and Rob H Bisseling. “Bulk: a modern C++ interface for bulk-synchronous parallel programs”. In: *European Conference on Parallel Processing*. Springer. 2018, pages 519–532 (cited on page 3).

[Ber20] S de Berg. “PMondriaan: A Parallel Hypergraph Partitioner”. Master’s thesis. 2020 (cited on pages 1, 5, 6, 49).

[Bis20] Rob H Bisseling. *Parallel Scientific Computation: A Structured Approach Using BSP*. Second Edition. Oxford University Press, USA, 2020 (cited on page 1).

[ÇA11] Ümit V Çatalyürek and Cevdet Aykanat. “Patoh (partitioning tool for hypergraphs)”. In: *Encyclopedia of parallel computing*. Springer, 2011, pages 1479–1487 (cited on page 52).

[FM82] Charles M Fiduccia and Robert M Mattheyses. “A linear-time heuristic for improving network partitions”. In: *19th Design Automation Conference*. IEEE. 1982, pages 175–181 (cited on page 6).

[GJ79] Michael R Gary and David S Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. 1979 (cited on page 3).

[Hen15] Vitali Henne. “Label propagation for hypergraph partitioning”. Master’s thesis. 2015 (cited on page 45).

[Kar+96] George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. “Hypergraph partitioning: Applications in VLSI domain”. In: *Technical Report TR-96–060* (1996), pages 412–429 (cited on page 1).

[KB20] Timon E Knigge and Rob H Bisseling. “An improved exact algorithm and an NP-completeness proof for sparse matrix bipartitioning”. In: *Parallel Computing* 96 (2020), page 102640 (cited on page 3).

[KK00] George Karypis and Vipin Kumar. “Multilevel k-way hypergraph partitioning”. In: *VLSI design* 11.3 (2000), pages 285–300 (cited on page 2).

[Kol+19] Scott P Kolodziej, Mohsen Aznaveh, Matthew Bullock, et al. “The suitesparse matrix collection website interface”. In: *Journal of Open Source Software* 4.35 (2019), page 1244 (cited on pages 49, 53, 54).

[Len12] Thomas Lengauer. *Combinatorial algorithms for integrated circuit layout*. Springer Science & Business Media, 2012 (cited on page 1).

[PB14] Daniël M Pelt and Rob H Bisseling. “A medium-grain method for fast 2D bipartitioning of sparse matrices”. In: *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE. 2014, pages 529–539 (cited on page 1).

[PE16] Maria Predari and Aurélien Esnard. “A k-way greedy graph partitioning with initial fixed vertices for parallel applications”. In: *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*. IEEE. 2016, pages 280–287 (cited on page 46).

[Sch19] Sebastian Schlag. *Benchmark Sets used in the Dissertation of Sebastian Schlag*. 46.12.02; LK 01. 2019 (cited on page 52).

[Sch20] Sebastian Schlag. “High-Quality Hypergraph Partitioning”. PhD thesis. Karlsruhe Institute of Technology, Germany, 2020 (cited on pages 4, 40).

[Val90] Leslie G Valiant. “A bridging model for parallel computation”. In: *Communications of the ACM* 33.8 (1990), pages 103–111 (cited on page 4).

[VB05] Brendan Vastenhouw and Rob H Bisseling. “A two-dimensional data distribution method for parallel sparse matrix-vector multiplication”. In: *SIAM review* 47.1 (2005), pages 67–95 (cited on page 52).

List of Figures