GRAPH THEORETICAL APPROACHES TO OPTIMIZE FUNCTIONAL REQUIREMENT INTEGRATION

A Writing Project

Presented to

The Faculty of the Department of Mathematics and Statistics

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Spenser M. Estrada

May 2023

The Designated Writing Project Committee Approves the Writing Project Titled

GRAPH THEORETICAL APPROACHES TO OPTIMIZE FUNCTIONAL REQUIREMENT INTEGRATION

by

Spenser M. Estrada

APPROVED FOR THE DEPARTMENT OF MATHEMATICS AND STATISTICS

SAN JOSÉ STATE UNIVERSITY

May 2023

Sogol Jahanbekam, Ph.D.	Department of Mathematics and Statistics
Daniel Brinkman, Ph.D.	Department of Mathematics and Statistics
Yan Zhang, Ph.D.	Department of Mathematics and Statistics

ABSTRACT

GRAPH THEORETICAL APPROACHES TO OPTIMIZE FUNCTIONAL REQUIREMENT INTEGRATION

by Spenser M. Estrada

In accordance with the concept of physical integration as understood in Axiomatic Design, the design parameters of a product should be integrated into a single physical part or few parts with the aim of reducing the information content, while still satisfying the independence of functional requirements. However, no specific method is suggested in the literature for determining the optimal degree of physical integration in a given design. This is particularly important with the current advancement in technologies such as additive manufacturing. Furthermore digitization, connected networks, embedded software, and smart devices have resulted in a major paradigm shift in business models. As new technologies allow physical elements to be integrated in new ways, new methods are needed to help designers optimize physical integration given the specific constraints and conflicts of each design. These paradigm shift calls for new design approaches. This paper proposes two methods to address these challenges. The first uses graph partitioning to allow a designer to optimize the integration of functional requirements into a target number of parts, with the goal of minimizing the integration of incompatible functional requirements into the same part. The application of that algorithm is demonstrated via two numerical examples and a practical example of designing a pencil. In the second, we demonstrate a graph coloring technique that can model changes in the functional requirements of a product and determine the minimum number of physical parts needed to meet future functionalities. This technique relies on vertex labeling by the designer and the construction of a core graph combining key elements of all desired iterations, which is then colored by label. One numerical example and one real-world example are provided. The two parts of this paper represent a consolidation and refinement of two papers previously completed by the author [14][24].

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation–USA under grants CMMI-2017968 and CMMI-1903810. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 INTRODUCTION

Axiomatic Design (AD) was developed by MIT mechanical engineering professor Num P. Suh in 1976 as the first design methodology to focus on the independence of functional requirements. AD systematically maps a design problem into several domains (e.g. customer domain, functional domain, physical domain, and process domain), to enable designers to select the best design solution while prioritizing two main axioms: the Independence Axiom and the Information Axiom [43]. Suh developed these axioms based on the philosophy that good designs share the same characteristics regardless of their physical nature or their domain of application. The Information Axiom requires that the information content of the design be minimized. The Independence Axiom requires that Functional Requirements (FRs), i.e., the actual purposes or functions of different parts of the final product, must remain as independent as possible [44]. The value of the independence axiom is to ensure that if one of the DPs were to fail, not all FRs would be affected. The Independence Axiom transforms a multi-input/multi-output system into a set of one-input/one-output systems to maintain the independence of FRs and build a more robust product [45].

The first step in designing a product is to define the set of FRs. The minimum set of independent functions that the design should satisfy is considered the set of FRs. The next step is to map the set of FRs into the physical domain, or a set of Design Parameters (DPs). According to the Independence axiom, DPs must be chosen such that the independence of FRs are maintained. AD uses design matrices to relate FRs to DPs and represents the design using a set of equations. What makes Axiomatic Design powerful is that it provides a quantitative approach to the formation of normative theories of design [17]. The relationship between the FRs and the DPs is characterized as follows:

$$FR = [A]DP$$

	Uncoupled Design	Decoupled Design	Coupled Design
Design Matrix	$\begin{bmatrix} A_{11} & 0 \\ 0 & A_{22} \end{bmatrix}$	$\begin{bmatrix} A_{11} & 0 \\ A_{21} & A_{22} \end{bmatrix}$	$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$

Table 1: Three Different Types of Design Matrices

Where each element of Matrix A, Aij, connects a component of the FR vector to a component of the DP vector [40]. The characteristics of design matrix A determine the degree to which the proposed design satisfies the Independence Axiom (see Table 1).

For example, a diagonal matrix is an ideal matrix, where each FR is independently satisfied by one corresponding DP. This is also referred to as uncoupled design. In the case of a full matrix the design violates the Independence Axiom, since the change of any single DP has an impact on all FRs. The independence axiom is particularly useful in the case of multi-objective optimization problems, due to the fact that each FR is independently satisfied by a set of design variables [27].

Since its inception in the late 1970s, Axiomatic Design has been subject of ample academic research, has been used widely across many disciplines, and has been taught internationally as part of engineering curricula [39] [48]. So far, over 11 international conferences on Axiomatic Design have been held in countries around the world. In addition to the field of engineering design, AD has impacted a wide range of practices in other disciplines including, but not limited to: healthcare delivery systems [6], software design [22], production scheduling [9], manufacturing system design [16], supplier selection [51], interactive art [20], decision science [15], and additive manufacturing [41].

Despite this broad adaptation, there are several flaws in Axiomatic Design, including the lack of a structured method for generating design matrices based on the axioms. Furthermore, the two axioms do not sufficiently capture all that is needed in a given design, leaving gaps in the application of human aspects of design [36], consumer preference, market demand [25], manufacturing considerations, and the potential to

require a preference structure of designers [37]. Another challenge of AD is that the goal of uncoupled design can be confusing at face value - often designers believe that a simple design is a good design. From this belief, we may conclude that a coupled design in which one DP satisfies multiple FRs is preferred [40]. However, the Independence Axiom does not mean that the DPs must be independent nor that each DP must correspond to a separate physical part. For example, a bottle-can opener is designed to satisfy two FRs and has more than 10 DPs, but has only one piece. In this study, we would like to address this issue and propose a method to help designers minimize the total number of physical parts regardless of the number of DPs. Therefore, a good design could satisfy many DPs and FRs with a minimum number of parts.

It should be noted that the concept of physical integration is completely different from modular design. A module is defined as a part or a group of parts that can be dismantled from the product in a non-destructive way as a unit [1] [21]. Ishii et al. [29] have referred to modular design as minimizing the number of functions per part. According to Ulrich and Eppinger [49], the most modular design is one in which each function is implemented by exactly one module or sub-assembly and there are limited interactions between modules.

With the focus on physical integration of multiple design features into a single part, researchers have come up with various methods to quantify the complexity of a design. There are several existing complexity quantifying methods. Generally, these existing methods first introduce the concept of changeability and the use of Axiomatic Design when designing production equipment, and then design-solution-specific barriers to flexibility and changeability are described [19; 47].

The idea behind physical integration or physical coupling is to integrate more than one FR in a single component, as long as FRs remain independent. Therefore, physical integration reduces the design complexity (at least in the physical domain). While

designers are in favor of physical integration, there is no normative approach on how to achieve physical integration using scientific engineering design techniques.

Kirschman and Fadel [31] have emphasized the usefulness of function-based methodologies in the design field. Researchers have already shown the necessity of considering the linkage between FRs and the number of parts for different reasons ranging from sustainability to reliability, simplicity, and even offering new functionalities to existing products. They have developed qualitative architectural roles and mathematical models to map functions to physical parts. To name a few, Bonjour et al. [4] developed a fuzzy method as an inference system in which membership functions define the structure of the design matrix. Then, a clustering algorithm groups elements of the matrix into modules. Devanathan et al. [11] emphasized the need for function-oriented methods at the early design process and suggested considering FRs and their impact on the number of parts and ultimately on product sustainability. Kurtoglu and Tumer [33] also considered the linkage between FRs and physical parts and combined hierarchical models of functionality and physical configuration at the early design stage to minimize the risk of functional failure of physical parts. Zhang et al. [54] discussed the importance of function-based analyses and proposed a function recommendation process to suggest adding new functions to an existing product. Bhasin et al. [2] also discussed function-sharing, enabling multiple functions to be performed by a single structure, as a success factor in biological systems and how it can be employed in the bio-inspired design field. Along this line, the current study considers the connection between FRs and physical parts and aims to define the minimum number of parts needed to satisfy the list of FRs.

A class of products that poses unique challenges with regards to physical integration is that of tethered appliances. The term “Tethered appliances” was first used in 2008 by Zittrain [56], in his book about the future of the Internet, to refer to an emerging class of

products defined by their ability to be monitored and altered by their sellers or owners due to advancements in wireless and GPS technology. “Internet of Things” devices, smart appliances, embedded-sensor devices, and voice assistants are examples of tethered products which require a consistent connection between users and device makers. Hooftnagle et al. [28] defined “tethered products” as devices whose functionalities and future iterations rely on ongoing connections between user and producer.

Corporations have come to the understanding that a profitable approach for maintaining an ongoing connection with customers is to focus on providing a continuous service, rather than one-time purchased products. In this marketing model, manufacturers have better control over the life cycle of a product, and in fact benefit from managing issues ranging from quality and dependability of service, to end-of-use recovery, and ultimately recycling of materials. In today’s IoT market, designers need specific design tools and techniques that enable them to a) model the dynamic nature of the various iterations of this new type of product and b) bundle the concepts of product/service together to create a successful commercial model that appeals to consumers.

In the first part of this paper, we will introduce a technique which uses algorithmic graph partitioning to integrate the FRs of a product in a way that minimizes undesirable combinations from the designer standpoint. This method is general in its applicability. The second part of this paper will present an integration technique that is specifically well-suited to the design of tethered products. This process works by taking in a finite number of graphs representing different iterations or versions of the product and uses input from the designer to segregate the FRs under consideration via criteria of their choosing. The graphs are then colored, where FRs receiving the same color are then integrated into the same physical part.

2 INTEGRATION VIA GRAPH PARTITIONING

In this section, we introduce a graph theory algorithm to help designers enhance the degree of physical integration and minimize product complexity by reducing the number of parts. Graph theory algorithms are widely used in making design decisions [3; 38; 52]. Buluc et al. discuss effectiveness of graph partitioning in analyzing complex networks. Division of graphs into small partitions is the primary step for making algorithmic operations more efficient. Therefore, one of the important sub-steps for complexity reduction or parallelization is graph partitioning. Large graphs are first partitioned into small ones and then they are analyzed. This is highly helpful in simulations, social networks or road networks [7]. While different graph partitioning techniques are used, these approaches tend to share certain basic algorithms [42]. As computing power evolves, multiple graph partitions can be run in parallel and ever more complex systems can be analyzed [18; 26].

A look at a few specific studies with different applications of graph partitioning methods will serve to illustrate the context for the algorithm proposed in this section. Li et al. used graph partitioning techniques to extract reusable 3D CAD models to improve design reusability [35]. Borisovsky et al. [5] worked on a machining line design problem consisting of sequences of workstations equipped with processing modules, called blocks, each of which performs specific operations. They used a graph partitioning technique to integrate machines to perform different sets of operations.

Our graph partitioning approach assigns a weighted value to potential conflicts between FRs, in order to construct a systematic method for achieving physical integration in design. Integrating FRs facilitates fewer assembly parts, greater flexibility, and less logical effort. The proposed algorithm provides a new method for optimizing physical integration, particularly for additively manufactured parts that have fewer manufacturing constraints in terms of geometry and shape. Our graph partitioning method allows the

designer to quantitatively determine which FRs to combine in single part, even for very complex designs, while reducing potential conflict between those FRs.

2.1 Proposed Graph Partitioning Algorithm

The method here described can be used to determine the optimal distribution of FRs among k parts in a product, given that the FRs may have varying degrees of compatibility with one another. We will use graphs to model the relationships between FRs in a product, and employ algorithmic graph partitioning to optimize the design.

A graph G is made up of two sets:

$$V(G) = \text{set of all vertices } v_i \text{ in } G$$

$$E(G) = \text{set of all edges } e_{i,j} \text{ in } G$$

where each member $e_{i,j} \in E(G)$ corresponds to a pair of vertices $v_i, v_j \in V(G)$. The order of the vertex set, $|V| = n$, is the number of vertices in G.

We will use identification of vertices in our optimization process. To identify two vertices in a graph G means that we merge two vertices v_i and v_j. The merged vertices form a new vertex $v_{i,j}$ in G. All edges formerly connected to v_i are connected to the new vertex $v_{i,j}$ and the same for all edges formerly connected to v_j. The edge $e_{i,j}$ between v_i and v_j is removed from the graph. The resulting graph is denoted as $G.v_{i,j}$. The weight of the removed edge, $\omega(e_{i,j})$, is defined as the cost of the contraction for our purposes. This contraction of identified vertices is the primary graph operation in our algorithm.

The algorithm proper is a recursive one composed of the function 1: `main`, with its associated helper functions 2: `recur` and 3: `contract`. The designer builds a graph G, where each vertex of G corresponds to a functional requirement of the product. The algorithm will recur among all possible vertex identifications in the graph G while comparing the penalties incurred by those graphs which have k vertices and searching for the one with minimum penalty. When it is found, that graph is referred to as G' and is the

final product of the algorithm. Essentially, `main` directs the input graph to `recur` which is where most of the computation occurs. As needed, `recur` calls `contract` to perform vertex identifications. When the process is over, `main` returns the result. We shall now explain in depth the algorithm's process, starting with its inputs and then the functions `main`, `recur`, and `contract` in-turn.

2.1.1 Inputs

The input to our proposed algorithm is a graph G with weighted edges chosen by the designer to represent the product under consideration. The designer chooses an integer $2 \leq k < n$, which is the number of discrete parts desired in the final product. The designer labels the vertices of G as $v_1, \ldots, v_n$. The designer places an edge $e_{i,j}$ between each pair of vertices v_i, v_j. Each edge is assigned an integral weight $\omega(e_{i,j})$ such that

$$0 \leq \omega(e_{i,j}) \leq \infty$$

by the designer to indicate the level of conflict between the FRs associated with those vertices. In this notation system, the designer can indicate the degree to which they would prefer to keep two FRs in separate parts of the final product, where a higher edge weight indicates a stronger preference to keep FRs in separate parts. The designer will label the edge $e_{i,j}$ with the weight ∞ when it is not desirable or possible for two FRs v_i and v_j to be in the same part under any circumstances. In the case where there is no conflict between two FRs, the designer may assign a weight of 0, or omit that edge.

2.1.2 Function **`main`**

This function sets the initial conditions of the algorithm and is its start and end point. The function shall be explained line-by-line.

1 The function sets the minimum possible final cost of G, *minCost*, to ∞, since there is no finite expected ceiling on the total final cost of the graph. The initial value of

G.cost, the cost of the input graph, is zero, since no vertices have been contracted yet.

2 The function initializes a stack of graphs, empty at first, onto which candidate solution graphs will be pushed as they are discovered.

3 The function invokes `recur` on G. The end result of `recur` is the solution graph G' which will be placed onto the *graphStack*. The specifics of what occurs in the function `recur` will be described in the next subsection.

4 The solution graph G' is popped off *graphStack* and returned as the algorithm output.

2.1.3 Function **`recur`**

This is a recursive function which traverses all possible configurations of G until it finds a graph with k parts and minimum cost among all the graphs with k parts. The function shall be explained line-by-line.

1 A conditional check occurs to ensure that the current running cost *G.cost* of the graph G does not exceed the established minimum cost, *minCost*. Also, it checks to see whether *minCost* is greater than zero. The purpose of this step is to halt the exploration of any branch of graphs whose identifications have already exceeded the minimum cost, and also to prFevent further exploration when a solution of minimum penalty has already been derived. Since the initial values are set to $G.cost = 0$ and $minCost = \infty$ for the input graph, the first run through the algorithm will not trigger this cutoff, and we can proceed.

2 The function checks to see if the graph has successfully been contracted to k or fewer vertices. If so, we proceed to line **3**. If G still has more than k vertices, the algorithm skips to line **6**. During the initial call to `recur`, G still has n vertices, so the algorithm skips to line **6**.

3 The function now sets *minCost* to be $G.cost$, that is, because of the check done on line **1**, we know that the graph we are new looking at must have the new minimum cost.

4 Since the graph under consideration has k parts and minimum cost, the function pushes it onto the *graphStack* as our current best solution. Exploration of the current branch of graphs ends at this point, and the algorithm backtracks to explore a new one. Note that we need not explicitly tell it to do so: in a recursive algorithm this occurs spontaneously.

5 The **if** block spanning lines **2-4** ends.

6 Here a **for** loop begins which iterates over the vertex pairs in G. Note that as G is modified, the set of vertices will change. For example, at the outset, the vertex set of G is $\{v_1, v_2, ..., v_n\}$. Suppose the vertices v_1, v_2 are identified and contracted into one vertex. Then the new vertex set will be $\{v_{1,2}, v_3, ..., v_n\}$.

7 A conditional check occurs to see if the edge $e_{i,j}$ between v_i, v_j, the vertices under investigation, has infinite weight. If so, there is nothing more to do with these vertices and we go back to line **6** and move on to the next pair. If $e_{i,j}$ has finite weight, we proceed to line **8**.

8 Here the function first identifies and contract the vertices v_i, v_j into the new vertex $v_{i,j}$. The graph produced is called $G.v_iv_j$. The specifics of how `contract` does this will be explained in the next subsection. Once `contract` is finished, the function calls `recur`, which effectively sends us back to line **1**, but with $G.v_iv_j$ instead of G, which then goes through the same processes we described in lines **2-7**. This process will recur for as long as it takes to exhaust all branches descendant of $G.v_iv_j$ for solutions, supposing they exist. Only then can we proceed to line **9**. Note that we are *not* proceeding to line **9** with $G.v_iv_j$, since that graph is created and used specifically on the current line.

9 Instead of identifying and contracting v_i, v_j, we instead assign infinite weight to the edge $e_{i,j}$ between them. The graph produced is called $G \cup v_i v_j$.

10 The function calls `recur`, which effectively sends us back to line **1**, but with $G \cup v_i v_j$ instead of G, which then goes through the same processes we described in lines **2-7**. This process will recur for as long as it takes to exhaust all branches descendant of $G \cup v_i v_j$ for solutions, supposing they exist. Then, we return to line **6** and move on to the next pair of vertices.

2.1.4 Function **`contract`**

This function takes as input a graph G and a pair of vertices v_i, v_j and contracts them into a single new vertex $v_{i,j}$. This entails taking every edge adjacent to v_i and attaching them to $v_{i,j}$ and likewise for v_j. Whatever weight was on the edge $e_{i,j}$ between them gets added to the cost of G.

1 The function increases the cost of G by the weight of $e_{i,j}$, since this edge will soon be contracted.

2 We create a new vertex in G, called $v_{i,j}$.

3 The function enters a **for** loop that iterates over every vertex v_x adjacent to v_i. That is, we shall iterate over every vertex v_x such that $e_{i,x}$ is an existing edge.

4 We make v_x adjacent to the new vertex $v_{i,j}$ created in line **2** by creating a new edge which we call $e_{i,j,x}$.

5 We assign the weight $\omega(e_{i,x})$ to the new edge $e_{i,j,x}$. We then return to line **3** while there are still unprocessed vertices adjacent to v_i.

7 The function enters a **for** loop that iterates over every vertex v_y adjacent to v_j. That is, we shall iterate over every vertex v_y such that $e_{j,y}$ is an existing edge.

8 We make v_y adjacent to the new vertex $v_{i,j}$ created in line **2** by creating a new edge which we call $e_{i,j,y}$.

Input: A graph G, with an initial cost, $G.cost = 0$
Output: A set of graphs contracted from G having k vertices each

```
1  Let minCost = ∞;
2  Let graphStack be a new stack;
3  recur(G);
4  pop G′ off of graphStack;
```

Function 1: `main`

Input: A graph G
Output: void
Result: A list of graphs contracted from G having k vertices each

```
1  if G.cost ≤ minCost and minCost > 0 then
2      if G has k or fewer vertices then
3          minCost = G.cost;
4          push G onto graphStack;
5      end
6      for each vertex pair v_i, v_j in G do
7          if ω(e_{i,j}) < ∞ then
8              recur(contract(G,v_i,v_j));
9              ω(e_{i,j}) = ∞;
10             recur(G);
11         end
12     end
13 end
```

Function 2: `recur`

9 We assign the weight $\omega(e_{j,y})$ to the new edge $e_{i,j,y}$. We then return to line **7** while there are still unprocessed vertices adjacent to v_j.

11 The new vertex $v_{i,j}$ is now correctly initialized, so we discard v_i and v_j. The graph G is now called $G.v_iv_j$.

2.2 Numerical Examples

In this section, we provide two numerical examples that illustrates how the algorithm acts on a graph to assign its FRs to a fixed number of parts. In the first example, we attempt to optimally assign the four FRs represented by the vertices v_1, v_2, v_3, v_4 of the graph G_1 into 3 parts. The graph G_1 itself can be seen in Figure 1. The weights between

Input: A graph G, and two vertices v_i, v_j in G
Output: G with v_i and v_j contracted

```
1  G.cost += ω(e_{i,j});
2  Let v_{i,j} be a new vertex in G;
3  for each vertex v_x adjacent to v_i do
4  |   make v_x adjacent to v_{i,j};
5  |   ω(e_{i,j,x}) += ω(e_{i,x});
6  end
7  for each vertex v_y adjacent to v_j do
8  |   make v_y adjacent to v_{i,j};
9  |   ω(e_{i,j,y}) += ω(e_{j,y});
10 end
11 delete v_i, v_j from G;
```

Function 3: `contract`

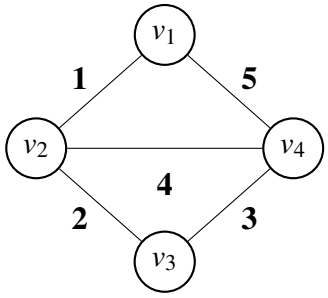


Fig. 1: Example graph G_1.

the vertices, i.e., $\omega(e_{i,j})$ quantify the extent to which we desire to keep the FRs represented by those vertices in separate parts in the final product.

Figure 2 demonstrates this process graphically as a binary tree which is traversed in a depth-first fashion. The changes made to G_1 are explained with regards to the letter-labeled arrows. That is, each letter in the list below describes the effect on the input graph of the arrow labeled with that letter. We start at the initial input graph G_1 seen at the top-left of Figure 2.

a: The vertices v_1 and v_2 are identified and become the vertex $v_{1,2}$. The edge $e_{1,2}$ has weigh 1. Therefore this action increases the total penalty from 0 to 1. The graph now

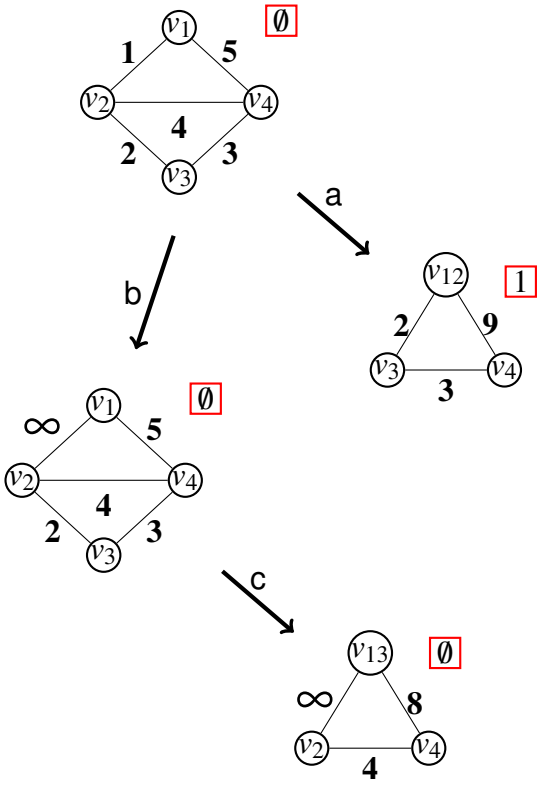


Fig. 2: The algorithm reduces G_1 to 3 parts.

has 3 parts as desired and finite penalty, so this graph becomes the first solution candidate.

b: The vertices v_1 and v_2 are insulated from each other by setting the weight of the edge $e_{1,2}$ to ∞.

c: The vertices v_1 and v_2 are identified and become the vertex $v_{1,2}$. The edge $e_{1,2}$ has weigh 0. No penalty is incurred from this action and the graph now has 3 parts as required, and has lesser penalty than our previous graph obtained in a. Further, since no lesser penalty is possible, the algorithm returns this graph as an optimal solution.

In the previous example, the best solution happens to be an answer that combines vertices with no weight on the edges between them and therefore 0 penalty. But this is not the case by necessity, and we should not in general expect a result with 0 penalty. In fact,

the contraction operation makes this unlikely with repeated use since contraction combines edges as well as vertices. The aim is to partition the vertex set into a given number of parts in a way that minimizes the penalty part-wise. That is, the cost penalty is the sum of weights of edges within the parts. When it is 0 it means that we have a perfect answer that necessitates no compromise from the designer's perspective. Our next example, however, has no such solution.

Suppose instead that we desire to reduce the graph G_1 to 2 parts. Our algorithm is able to accomplish this just as well, though the generated process is more complex. The process is illustrated in Figure 3. As before, the changes made to G_1 are explained with regards to the letter-labeled arrows. We start at the initial input graph G_1 seen at the top-left of Figure 3.

a: The vertices v_1 and v_2 are identified and become the vertex $v_{1,2}$. The edge $e_{1,2}$ has weigh 1. Therefore this action increases the total penalty from 0 to 1. The graph now has 3 parts.

b: The vertices $v_{1,2}$ and v_3 are identified and become the vertex $v_{1,2,3}$. The edge $e_{1,2,3}$ has weigh 9. Therefore this action increases the total penalty from 1 to 3. The graph now has 2 parts as desired and finite penalty, so this graph becomes the first solution candidate.

c: The vertices $v_{1,2}$ and v_3 are insulated from each other by setting the weight of the edge $e_{1,2,3}$ to ∞.

d: The vertices $v_{1,2}$ and v_3 are identified and become the vertex $v_{1,2,3}$. The edge $e_{1,2,3}$ has weigh 9. Therefore this action increases the total penalty from 1 to 10. The graph now has 2 parts as desired, but its penalty is higher than that obtained in step b and is discarded.

e: The vertices $v_{1,2}$ and v_4 are insulated from each other by setting the weight of the edge $e_{1,2,4}$ to ∞.

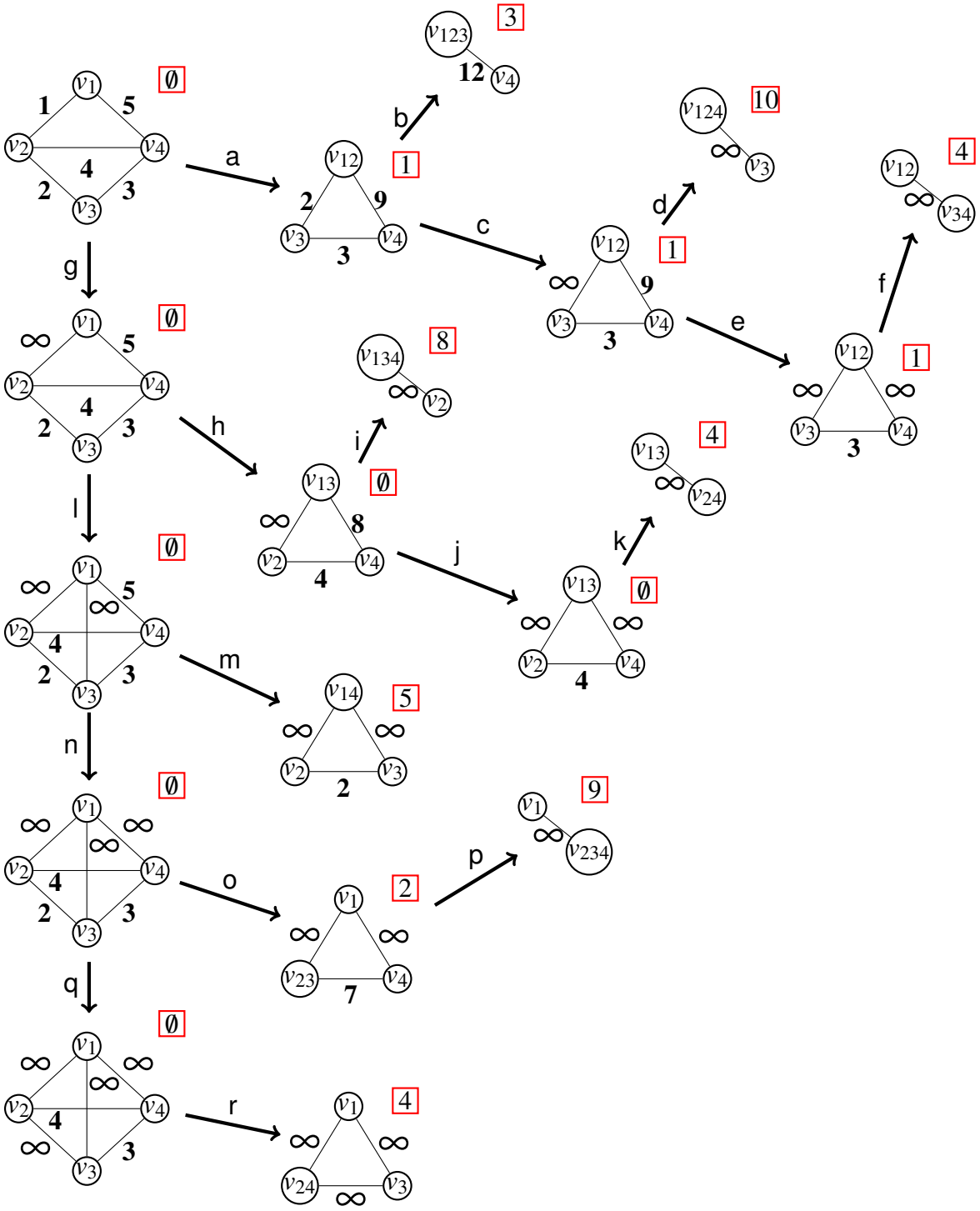


Fig. 3: The algorithm reduces G_1 to 2 parts.

f: The vertices v_3 and v_4 are identified and become the vertex $v_{3,4}$. The edge $e_{3,4}$ has weigh 3. Therefore this action increases the total penalty from 1 to 4. The graph now has 2 parts as desired, but its penalty is higher than that obtained in step b and is discarded.

g: The vertices v_1 and v_2 are insulated from each other by setting the weight of the edge $e_{1,2}$ to ∞.

h: The vertices v_1 and v_3 are identified and become the vertex $v_{1,3}$. The edge $e_{1,3}$ has weigh 0. No penalty is incurred.

i: The vertices $v_{1,3}$ and v_4 are identified and become the vertex $v_{1,3,4}$. The edge $e_{1,3,4}$ has weigh 8. Therefore this action increases the total penalty from 0 to 8. The graph now has 2 parts as desired, but its penalty is higher than that obtained in step b and is discarded.

j: The vertices $v_{1,3}$ and v_4 are insulated from each other by setting the weight of the edge $e_{1,3,4}$ to ∞.

k: The vertices v_2 and v_4 are identified and become the vertex $v_{2,4}$. The edge $e_{2,4}$ has weigh 4. Therefore this action increases the total penalty from 0 to 4. The graph now has 2 parts as desired, but its penalty is higher than that obtained in step b and is discarded.

l: The vertices v_1 and v_3 are insulated from each other by setting the weight of the edge $e_{1,3}$ to ∞.

m: The vertices v_1 and v_4 are identified and become the vertex $v_{1,4}$. The edge $e_{1,4}$ has weigh 5. We discard this graph without further exploration since its penalty is higher than that obtained in step b.

n: The vertices v_1 and v_4 are insulated from each other by setting the weight of the edge $e_{1,4}$ to ∞.

o: The vertices v_2 and v_3 are identified and become the vertex $v_{2,3}$. The edge $e_{2,3}$ has weigh 2. Therefore this action increases the total penalty from 0 to 2.

p: The vertices $v_{2,3}$ and v_4 are identified and become the vertex $v_{2,3,4}$. The edge $e_{2,3,4}$ has weigh 7. Therefore this action increases the total penalty from 2 to 9. The graph now has 2 parts as desired, but its penalty is higher than that obtained in step b and is discarded.

q: The vertices v_2 and v_3 are insulated from each other by setting the weight of the edge $e_{2,3}$ to ∞. Note that the algorithm will not any attempt more insulation, since doing so would make it impossible to end up with 2 parts.

r: The vertices v_2 and v_4 are identified and become the vertex $v_{2,4}$. The edge $e_{2,4}$ has weigh 4. The algorithm terminates for two reasons: all vertices are now insulated, meaning no further action is possible, and the penalty is higher than that obtained in step b. The graph obtained in step b with k parts and penalty 3, which is now assured to be the minimum of all generated, is returned as the solution.

2.3 Practical Example: Pencil Design

Suppose we wished to design a mechanical pencil. Such a device could have the following FRs depending on the exact kind of pencil:

FR1 - Erasing	**FR6** - Lead Positioning
FR2 - Lead Storage	**FR7** - Grip
FR3 - Eraser Storage	**FR8** - Clip
FR4 - Lead Extrusion	**FR9** - Body
FR5 - Lead Support	

Suppose that we desire to segregate FR1, FR6, and FR8 in our design. Then the graph that models these FRs and their relations shall have FR1 and FR6 adjacent to all other vertices of the graph and with infinite weight assigned to those edges. The other edges receive the lesser values enumerated below.

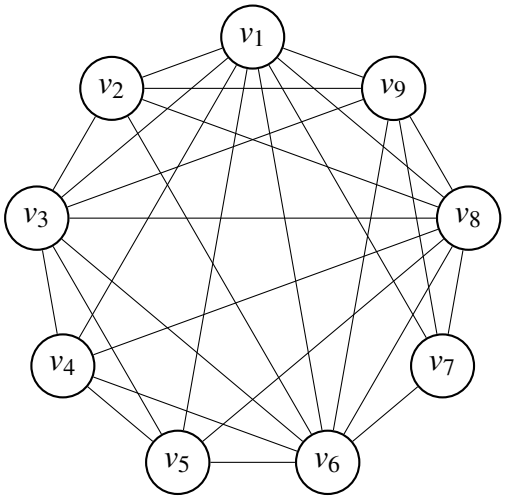


Fig. 4: Graph G_P.

$$\begin{array}{ll}
\omega(e_{1,2}) = \infty & \omega(e_{3,6}) = \infty \\
\omega(e_{1,3}) = \infty & \omega(e_{3,8}) = 1 \\
\omega(e_{1,4}) = \infty & \omega(e_{3,9}) = \infty \\
\omega(e_{1,5}) = \infty & \omega(e_{4,5}) = 2 \\
\omega(e_{1,6}) = \infty & \omega(e_{4,6}) = \infty \\
\omega(e_{1,7}) = \infty & \omega(e_{4,8}) = \infty \\
\omega(e_{1,8}) = \infty & \omega(e_{5,6}) = \infty \\
\omega(e_{1,9}) = \infty & \omega(e_{5,8}) = \infty \\
\omega(e_{2,3}) = 2 & \omega(e_{6,7}) = \infty \\
\omega(e_{2,9}) = 5 & \omega(e_{6,8}) = \infty \\
\omega(e_{2,6}) = \infty & \omega(e_{6,9}) = \infty \\
\omega(e_{2,8}) = \infty & \omega(e_{7,8}) = \infty \\
\omega(e_{3,4}) = 1 & \omega(e_{7,9}) = 1 \\
\omega(e_{3,5}) = 2 & \omega(e_{8,9}) = \infty
\end{array}$$

The graph representing these FRs and edges can be seen in Figure 4. It has 9 vertices represents the 9 FRs of the pencil. The edge weights are absent from the actual figure for the sake of clarity, but may be referenced in the list above.

The result of the algorithm acting on G_p can be seen in Figure 5 where G'_p is displayed along with the penalty accrued in creating it. As desired, FR1 and FR6 are their own parts. In Figure 6, we see an actual pencil that embodies this solution, where the FRs are allocated to the pencil's parts in the following manner:

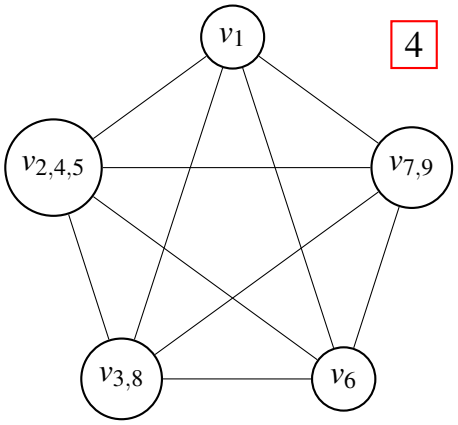


Fig. 5: Graph G'_P.

Fig. 6: A 5-part pencil embodying G'_P.

- Eraser - FR1
- Chuck Assembly - FR2, FR4, FR5,
- Clip - FR3, FR8
- Tip and Sleeve - FR6
- Body - FR7, FR9

3 INTEGRATION VIA GRAPH COLORING

There are several powerful disruptive trends in industry that facilitate the broad adoption of "Tethered Economy" products. First, extensive progress in the field of information technology helps corporations to trace materials anywhere in their supply chain and monitor the status of products during usage as well as the end-of-use phase [30]. Advancements in data collection and sensor-based technologies, big data analytics, distributed ledger technologies, and autonomous systems are just a few examples of novel technologies that revolutionize the way that corporations design and sell their products. Second, there is a pervasive shift in consumer behavior as younger generations of users have shown that they prefer access over ownership [32]. Third, the emergence of new business models paves the way for shifting toward service-based strategies [34], where businesses switch to subscription and membership models as opposed to selling the ownership of their physical products to consumers.

The above-mentioned trends work together to move the market towards the tethered economy. This complicates purchase decisions for users, as they need to think about the future service costs of a product, its compatibility with other devices, data security, and privacy concerns. Similarly, new complications arise in the design process, as manufacturers now need to consider future iterations of products, the merging of services, and blending hardware and software features together.

According to Hooftnagle et al [28], corporations rely on two primary mechanisms to tether a product: tethering through design and tethering through law. Tethered products often have three distinctive features: (1) they depend on software codes for their operations, (2) they are equipped with data collection technologies such as sensors to facilitate communication of product life cycle data as well as consumer behavior, and (3) they require persistent network connections to enable long distance usage and control of the device [28].

While design is the main mechanism by which tethering is implemented, force of law is another strategy used for tethering. Device makers employ carefully drafted service contracts and mechanisms like copyright and patent law to influence the market. For example, companies could regulate the repair rights of consumers with the help of current data privacy laws, restrict the unlocking of smart devices for reuse and recycling, or debar consumers from using third-party repair services by adding legal terms into their warranty contracts [46]. Currently the law is limited in its power to regulate manufacturers' tethering of products. However, as legislatures and government agencies fight to protect consumers from the potential harm of such efforts (e.g. implanted medical devices responding to remote control, automatic alteration or deactivation of appliances), it is expected that new laws and regulations will be enacted that will require manufacturers to pay special attention to the way that they design and manipulate products with multiple iterations [10].

New challenges in design arise from this paradigm shift toward ever-connected products. The ranking of FRs can change based on which iteration of a given product is under consideration, and additional consideration must be given to evolving customer needs over time. Bearing this in mind, the number of physical parts that are included in the product usually remains fixed over the physical life cycle of a device, while the device maker retains the ability to alter product features and functionalities through software codes. Therefore, number of physical parts is often a decision made at the early stage of the design process, despite uncertainties regarding future iterations of a product.

Zhang et al. [53] have highlighted the need for appropriate mathematical models that simulate the entire ecosystem of a product. Such models should consider product iterations over time, as well as the evolution of its components, and even the interaction of various performance attributes of a component throughout the product's life cycle. Mathematical exploration of product life cycle decisions can be conducted with the use of

normative methods such as Monte Carlo simulation [50], statistical analysis, Bayesian approach [55], and probability theories [12], to name a few. In this study, we will discuss the use of graph theory and network modeling techniques for considering product requirements over time. The idea behind this approach is to consider N graphs, each representing the requirements of a specific iteration of a product. This approach has unique applications in different settings, including but not limited to the following scenarios:

1) **Technology Shifts:** technological evolution of a product including both functional and technological changes where new features and functionalities will be added or removed from the product over time.
2) **Product Generations:** design for a product family where platform-based product development is considered.
3) **New Users:** design for multiple iterations where needs of several users should be satisfied
4) **Updated Software:** design in the tethered economy world where the access of the original user is controlled by new upcoming business models
5) **Product Iterations:** design that covers the needs of several phases within the product life cycle, including manufacturing, initial and subsequent users, and end-of-life product recovery, such as recycling.

Designers do not always have a complete picture of the future iterations that will be needed during the entire life cycle of a product, and yet modelling tools that allow them to plan for many types of probable future iterations can improve the design process for tethered products. Modeling tools that provide better information about how to integrate the (sometimes competing) needs of future iterations will enable designers to make informed decisions when identifying the optimal number of parts for a product. In this section, we will use coloring and unification of graphs to represent changing requirements

during different iterations of the product. The final coloring of the unified graph can help designers implement and manage design changes with greater accuracy, and furthermore, can help define the proper number of physical parts that to cover FRs over several product iterations. We further develop a graph coloring and unification process to model a consensus set of FRs, and to help designers determine the minimal optimal number of parts or modules for the product. This approach expands on previous work done by some authors of this paper with Gopalakrishnan, et al [23] in 2019, which also developed a graph coloring technique for determining the minimum number of parts of a product, extending the technique to modeling products with multiple iterations, specifically in the case that FRs change over time.

3.1 Proposed Graph Coloring Algorithm

The proposed method addresses three needs of the designers: (1) designers understand the effects of adding or removing FRs during subsequent iterations, (2) they can define which FRs remain fixed during the product's lifespan based on the service agreements, and (3) they can define which FRs have the most potential for ongoing alteration.

To model these relations we can use graphs. A graph G is composed of a set of nodes called the vertex set $V(G)$ and an edge set defined as
$E(G) = \{vw \mid vw \text{ is an edge connecting nodes } v \text{ and } w\}$. A subgraph $S \subseteq G$ is defined as any subset of vertices of G with edges between them in $E(G)$. Independent sets within G are defined as a set of vertices $I \subseteq G$ such that
$I = \{v_i \mid v_i v_j \text{ is not an edge in } E(G)\ \forall\ i, j\}$. An independent set I is *maximal* if it cannot be extended by the inclusion of any vertex $v \in G \setminus I$. Our preferred coloring algorithm takes advantage of maximal independent sets within G to shorten the coloring process.

3.1.1 Inputs

The method begins with N graphs $\{G_1, G_2, G_3, ..., G_N\}$ created by the designer to represent N iterations of a product. Each graph G_i is made up of:

- $V(G_i)$: a set of vertices, where each vertex represents a Functional Requirement (FR) of the product during the ith iteration.
- $E(G_i)$: a set of edges where each edge represents a fundamental conflict between FRs during the ith iteration.

All vertices and edges are set by the designer. Note that the designer may choose to add FRs (as vertices) to successive G_i's, and/or new edges representing conflicts. For example, in order for a product to be easily repaired, it may need an additional feature that conflicts with one of the FRs of the original design. Alternatively, the designer may choose to remove FRs and/or edges in subsequent iterations of a product - i.e., certain features and requirements of the design may become obsolete in future iterations. See Example 2, below, for a demonstration of how FRs may shift between iterations.

The designer places edges between vertices (FRs) that should not be combined into a single part of the finished product. For example, in the design of a mechanical pencil, the FR of 'marking' and the FR of 'erasing' have the fundamental conflict that they cannot be the same material. A designer would place an edge between the vertices representing these FRs in the graph representing the pencil.

Furthermore, the designer defines a set of labels $k \in \{1, 2, 3, ...\}$, where each label k corresponds to a subset of vertices in G_i. These labels are used to group vertices according to any design specification of interest. Examples of possible uses include grouping FRs with similar expiry dates, or those with equivalent costs to replace. These subsets of vertices grouped by label partition $V(G_i)$, so that each vertex has one and only one label. The information in these labels will allow us to increase the efficiency of our coloring algorithm, in a method inspired by the work of Eppstein [13] and Byskov [8] which improved coloring algorithms by looking at maximal independent subsets of graphs.

The completed iteration graphs G_i are the inputs to Function 4.

Input: A set of graphs $G_1, G_2, ..., G_N$
Output: Void
Result: A set of N properly-colored graphs

```
1  Let G_C be an empty graph;
2  for each vertex u ∈ V(G_1 ∩ G_2 ∩ ... ∩ G_N) do
3  |   G_C = G_C + u;
4  end
5  for each edge e ∈ E(G_1 ∪ G_2 ∪ ... ∪ G_N) do
6  |   G_C = G_C + e;
7  end
8  Let c = 1;
9  c = colorByLabel(c, G_c);
10 for each G_i of G_1, G_2, ..., G_N do
11 |   G_i=G_i \ G_C;
12 |   colorByLabel(c, G_i);
13 |   G_i = G_i ∪ G_C;
14 end
```

Function 4: main

Input: A graph G, each vertex having a numeric label
A number c, the next available color
Output: A number c representing the last used color
Result: G is now properly colored

```
1  for each label k in G do
2  |   Let G_k be the subgraph of vertices of G having label k;
3  |   c = c + color(G_k);
4  end
5  return c;
```

Function 5: colorByLabel

3.1.2 Function **`main`**

In this function, the main result is to identify the *core graph*, G_C which will include those FRs common to all N graphs. We build the vertex set of G_C by taking the intersection of all $v(G_i)$ for each $G_i \in \{G_1, G_2, ..., G_N\}$, as in Equation 1:

$$\bigcap_{i \leq n} V(G_i) = V(G_C) \tag{1}$$

We build the edge set of G_C by first finding a subgraph $H_{iC} \in G_i$ for each i such that H_{iC} includes all edges induced by $V(G_C)$. We then take the union of edges $E(H_{iC})$ in those subgraphs as in Equation 2:

$$\bigcup_{i \leq n} E(H_{iC}) = E(G_C) \tag{2}$$

This process occurs as described in the following lines of Function 4:

1 The core graph G_C is initialized, currently empty, i.e., with neither vertices nor edges.

2 We enter a for loop that iterates over every vertex that is in the intersection of the vertex sets V_i of the N graphs G_i.

3 Every vertex u in the aforementioned intersection is added to the edge set of G_C.

5 We enter a for loop that iterates over every edge that is in the union of the edge sets E_i of the N graphs G_i.

6 Every edge e in the aforementioned union is added to the edge set of G_C. G_C is now populated, but not colored.

Now that G_C has been identified, it can be colored.

8 We initialize the variable c, which is the next color to be assigned. Note that c is a natural number that starts at 1, but when we display the colored graph, we'll use actual colors to do so.

9 The function `colorByLabel` is invoked on G_C. The actual workings of that function will be explained in the next subsection. G_C is now colored, and c contains the next color after all those used to color G_C.

The next step is to color the parts of the N graphs G_i that are outside of the already colored core graph G_C, which occurs over the following lines:

10 We enter a for loop that iterates over each of the N graphs G_i.

11 Each G_i has the vertices and edges of G_C subtracted from it.

12 Each G_i is colored via the function `colorByLabel`, whose workings will be explained in the next subsection. It is very important to note here that unlike in line **9**, we are not incrementing the value of c by the number of colors used to color $G_i \setminus G_C$. This is because there is no need to avoid use of duplicate colors between graphs, since in the end they are considered separately.

13 The now colored graph G_i is re-united with G_C. It is now referred to as G_i', its final form.

3.1.3 Function **`colorByLabel`**

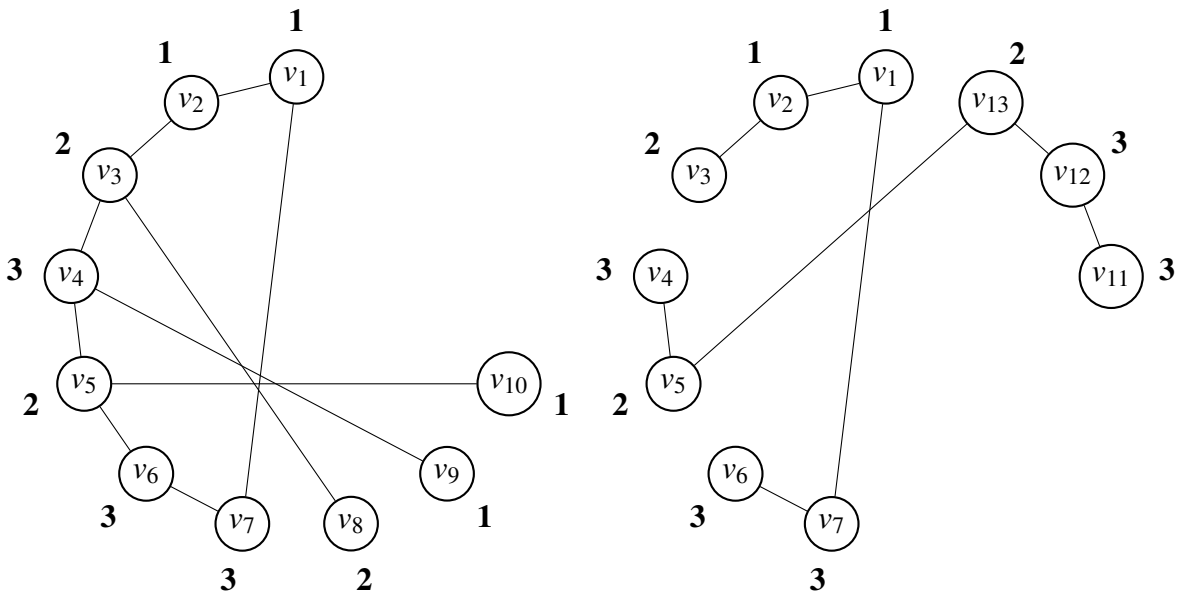
Beginning with color c and the smallest natural number label k present in whichever graph G in input to the function, we properly color all those vertices with label k using any convenient coloring algorithm. We prefer the method enumerated in [8]. This coloring algorithm finds the chromatic number of G by populating an array X of length 2^n with the chromatic numbers of all the subgraphs in the power set of G, including G itself. It then colors G by finding a subgraph $S \subset G$ such that $X[S] = X[G] - 1$, and then assigns the first color to the vertices of $G - S$. It then finds a subgraph $T \subset S$ such that $X[T] = X[S] - 1$ and assigns color 2 to the vertices of $S - T$ and so on until all the vertices of G are colored. The last color used is returned as an output of the function. This process occurs as described in the following lines of Function 5:

1 We iterate over all of the k labels present in the input graph G.

2 We consider the subgraph G_k, which consists of all vertices in G with label k, and the edges between them.

3 G_k is properly colored. In our case, we use the Byskov algorithm described earlier. The variable c, which keeps track of the number of colors used so far, is increased by the number of colors used to color G_k.

5 Once every G_k is colored, the value of the last color used is returned as c.



(a) G_1 (b) G_2

Fig. 7: Input graphs G_1 and G_2 with labels.

3.2 Numerical Example

We here include two examples of the coloring and unification process. The first example represents an abstract product with two input graphs. Each input graph represents some iteration of the product. In each example, labels are shown as boldface numbers beside each node. Each node represents a specific FR, and the labels represent some shared quality such that we would like to unite those FRs into a single part if possible. This desire is reflected in the algorithm itself, which breaks the input graphs down label-wise, and tries to color nodes of the same label with the same color, only failing to do so if the presence of an edge forbids it.

Consider the example input graphs in Fig. 7. From these input graphs G_1 and G_2, we take the intersection of vertices appearing in both graphs to find the core graph G_C. Once G_C is found, we begin to color the graph by first examining and coloring the vertices labelled **1**. Thus, any vertex labelled **1** will receive the same color, unless an edge

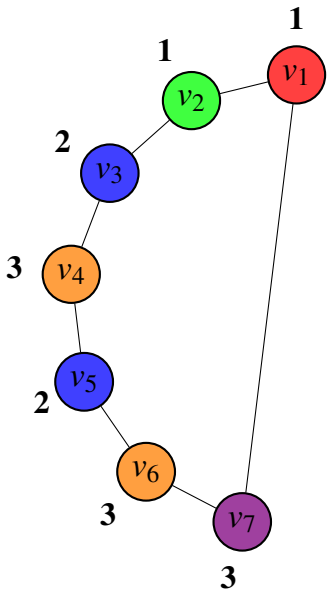
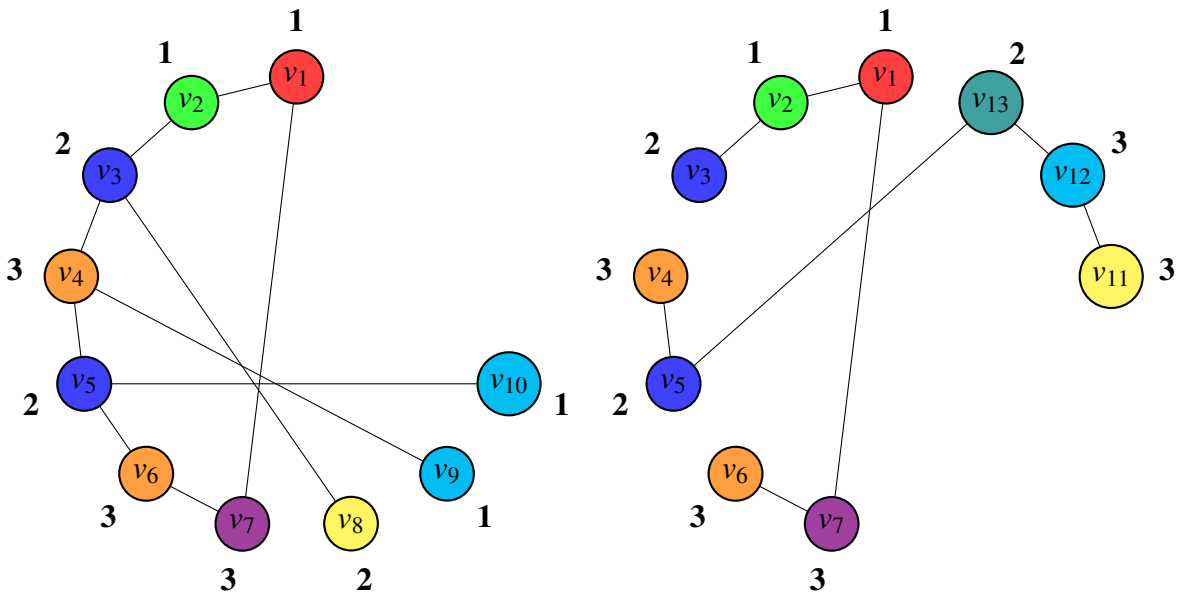


Fig. 8: The core graph of G_1 and G_2 with coloring.

between vertices (representing a conflict between FRs) forbids this. In this case, since the two vertices labelled **1** are adjacent to one another, we color the first, v_1, red, and the second, v_2, green. Once all vertices labelled **1** are colored, we proceed to vertices labelled **2**. At this stage, we choose the next available color not used on vertices of label **1**, thus guaranteeing that vertices of different labels will have different colors. In this case, the two vertices labelled **2** are not adjacent, so they are both colored blue. This process continues through all the labels until G_C is properly colored (see Fig. 8).

Once the core graph is colored, we return to each of the input graphs G_1 and G_2 and color remaining vertices, again proceeding by label, and beginning each G_i with the next available color after coloring the Core Graph. A final coloring of the input graphs is shown in Fig. 9. The graph G_1 has been colored with 7 colors and the graph G_2 with 8 colors. The two graphs are now referred to as G_1' and G_2' respectively. Thus the item represented by these graphs can be manufactured with 7 or 8 colors depending on which input graph is of more importance to the designer. Should the designer choos to retain the



(a) G'_1 (b) G'_2

Fig. 9: The coloring result of the two input graphs.

complete set of FRs of both iterations, they will need 10 distinct parts, as 10 colors are used to color both of the two input graphs completely.

3.3 Practical Example: Speaker Design

The second example is a real-world product: an internet-enabled speaker. In our hypothetical scenario, the designer is considering three iterations of the product:

1) **Basic Model:** The product has modest performance specifications and is controlled via Bluetooth only.
2) **Voice Assistant Model:** The product has modest performance specifications and is Bluetooth and Wi-Fi enabled, and has a microphone for voice control.
3) **Hi-Fi Model:** The product is Bluetooth and Wi-Fi enabled, has a microphone for voice control, uses a high performance speaker and amplifier, and has manual volume control.

The designer will apply our coloring algorithm to fix which FRs will appear in each iteration, and to determine the effects of these differences on the number of parts required. Additionally, application of the algorithm will give the designer an idea of which FRs have the capacity for ongoing development throughout the product's life cycle.

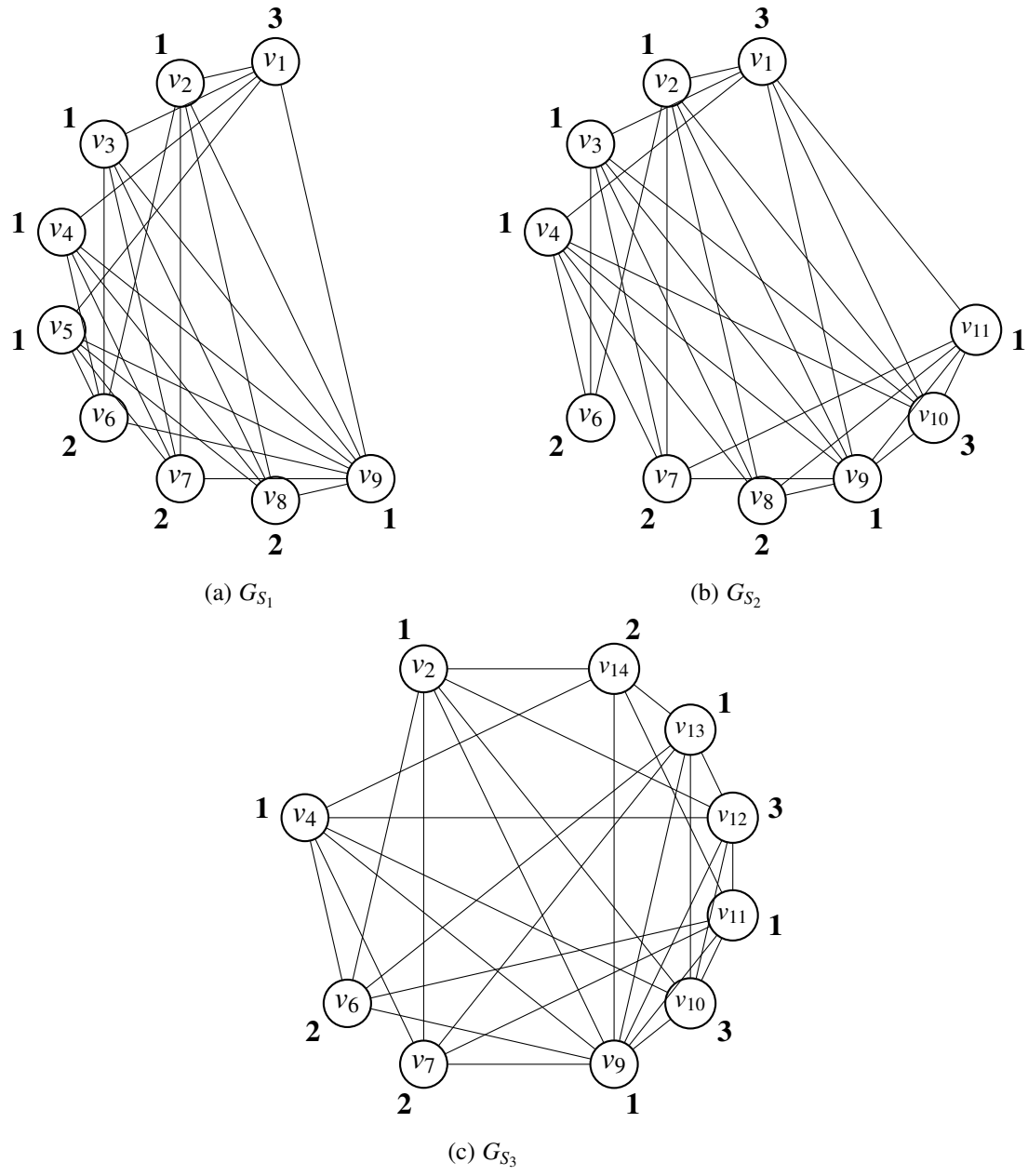


Fig. 10: Input graphs G_{S_1}, G_{S_2}, and G_{S_3} with labels.

Consider the example input graphs in Fig. 10. Each vertex, in addition to its index, is assigned a label (appearing here as a bold numeral next to the vertex). Any number of labels can be used, and these labels could represent any classifications the designer wishes to apply to the various FRs. Possible uses for the labels include distinguishing between FRs with different development or upgrade cycles; FRs designed by different departments in a large corporation; or FRs with different software requirements. In this case, let there be three labels representing supply chain sources for each type of part, roughly classified as **1**: electronic, **2**: physical, and **3**: acoustic. Table 2 specifies the assignment of FRs to labels and vertices, and indicates which FRs are included in each iteration.

Table 2: Tethered speaker functional requirement vertex assignment.

			Iterations		
Functional Requirement	**Vertex**	**Label**	**Basic Model**	**Voice Assistant**	**Hi-Fi Model**
Sound Output	v_1	3	✓	✓	
Improved Sound Output	v_{12}	3			✓
Sound Input	v_{10}	3		✓	✓
Power Input	v_2	1	✓	✓	✓
Amplification	v_3	1	✓	✓	
Improved Amplification	v_{13}	1			✓
Computation	v_4	1	✓	✓	✓
Bluetooth	v_5	1	✓		
Bluetooth & Wi-Fi	v_{11}	1		✓	✓
Enclosure	v_6	2	✓	✓	✓
Manual Level Control	v_{14}	2			✓
Actuation	v_7	2	✓	✓	✓
Portability	v_8	2	✓	✓	
Power Storage	v_9	1	✓	✓	✓

Following the unification process carried out by Algorithm 4, the Core Graph for these inputs is shown in Fig. 11. Note that the Core Graph contains only the following

FRs: v_2 Power Input, v_4 Computation, v_9 Power Storage, v_6 Enclosure, and v_7 Actuation. These are the FRs which are common to each iteration of the speaker.

We color G_C by label with a proper coloring according to Algorithm 5. We first color all vertices labelled **1**. In this case, there is no conflict between v_2 (Power Input) and v_4 (Computation), so these two vertices can both receive the color red. Since the remaining vertex of label **1**, v_9 (Power Storage), has a conflict with the other two vertices labelled **1**, it gets the next available color, green. Note that this conflict stems from the impracticability of affixing a large, removable battery to the same circuit board as the parts associated with the other FRs labelled **1**. Next, we move on to label **2**, and since there is no conflict between v_6 (Enclosure) and v_7 (Actuation), we assign the next available color, blue, to both vertices labelled **2**. Thus we see that three colors are sufficient to achieve a proper coloring of the core graph G_C. This indicates that three basic components, corresponding to the three colors, can be built and used in the assembly of all three iterations:

G_C Core Graph Coloring Interpreted as Parts

Red Single circuit board with power supply (Power Input) and system on a chip (Computation)

Blue Case (Enclosure) with built-in on/off button (Actuation)

Green Replaceable battery (Power Storage)

Once the Core Graph is colored, we return to the remaining uncolored vertices in each iteration. We begin coloring $G_{S_1} \setminus G_C$ with the next available color after core coloring is complete: $c+1$ (in this case, orange) in reference to the output c of Algorithm 4. We then proceed to color by label as in Algorithm 5, beginning with vertices labelled **1**. Since v_3 (Basic Amplification) and v_5 (Bluetooth) have no conflict, they can both be colored orange. We proceed to v_8 (Portability), the only uncolored vertex labelled **2** in this iteration, which receives the next available color: purple. Finally, we color the lone vertex

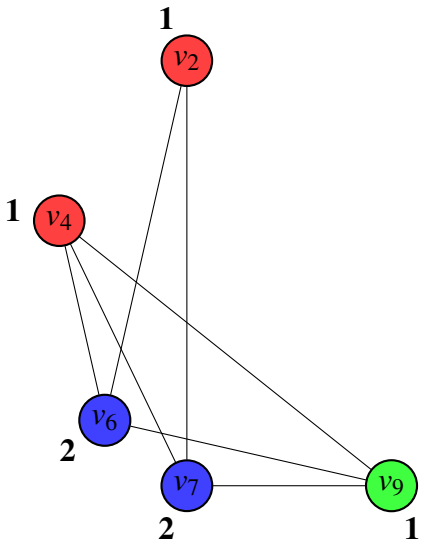


Fig. 11: The core graph of G_{S_1}, G_{S_2}, G_{S_3} with coloring.

labelled **3**, v_1 (Basic Sound Output) yellow. Six colors suffice to completely color G_{S_1}, indicating that the FRs could be combined into six independent parts when the Basic Model of the speaker is built (see Figure 13a for a rendering):

G_{S_1} **Basic Model Coloring Interpreted as Parts**

Red Single circuit board with power supply (Power Input) and system on a chip (Computation)

Blue Case (Enclosure) with built-in on/off button (Actuation)

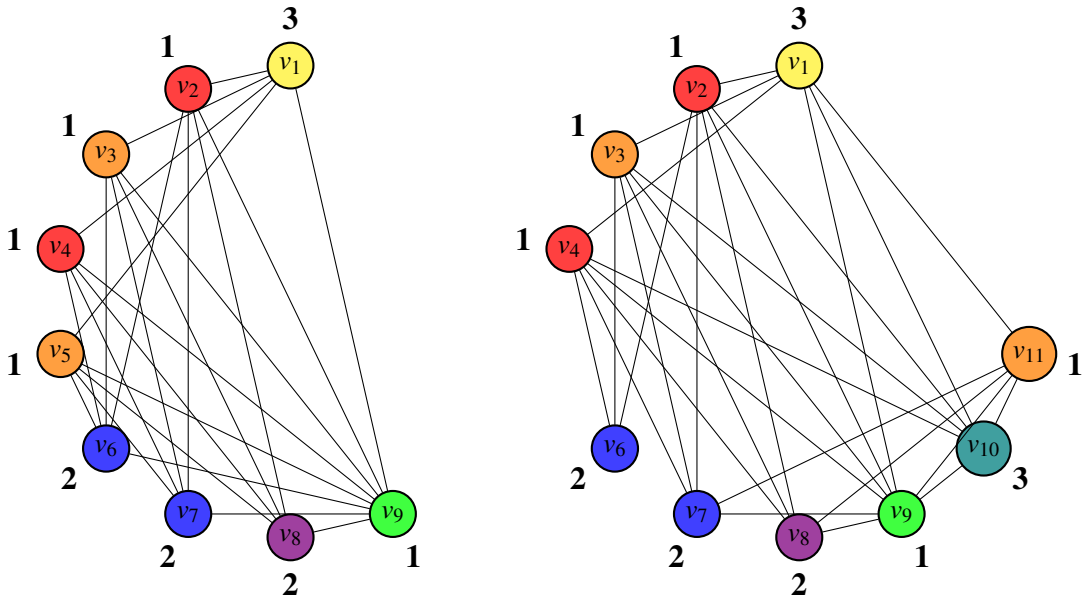
Green Replaceable battery (Power Storage)

Orange Circuit board with a 10 Watt 10db S/N Amplifier (Basic Amplification) and a bluetooth antennae (Bluetooth Connectivity)

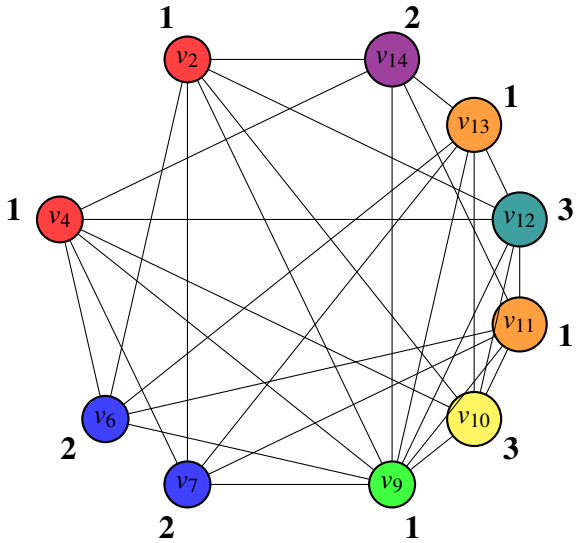
Purple Handle (Portability)

Yellow 5 Watt Speaker (Basic Sound Output)

We proceed next to $G_{S_2} \setminus G_C$, which we will color by label as in Algorithm 5. Note that we will begin coloring $G_{S_2} \setminus G_C$ with the same initial color that we used to begin coloring $G_{S_1} \setminus G_C$: orange. Since the two vertices labelled **1**, v_3 (Basic Amplification) and



(a) G'_{S_1} (b) G'_{S_2}



(c) G'_{S_3}

Fig. 12: Coloring result of the three input graphs.

v_{11} (Bluetooth and Wi-Fi), have no conflict, they are both labelled orange. The lone uncolored vertex labelled **2**, v_8 (Portability), is colored purple. Finally, the two vertices labelled **3** have a conflict in the form of a shared edge, so they are colored yellow for v_1 (Basic Sound Output) and light blue for v_{10} (Sound Input). Thus seven colors complete the

coloring of G_{S_2}, indicating that the FRs could be combined into seven independent parts when the Voice Assistant Model of the speaker is built (see Figure 13b for a rendering):

G_{S_2} Voice Assistant Model Coloring Interpreted as Parts

Red	Single circuit board with power supply (Power Input) and system on a chip (Computation)
Blue	Case (Enclosure) with built-in on/off button (Actuation)
Green	Replaceable battery (Power Storage)
Orange	Circuit board with a 10 Watt 10db S/N Amplifier (Basic Amplification) and a combined Bluetooth and Wi-Fi antennae (Bluetooth and Wi-Fi)
Purple	Handle (Portability)
Yellow	5 Watt Speaker (Basic Sound Output)
Light Blue	Microphone (Sound Input)

Finally, we begin to color $G_{S_3} \setminus G_C$, again starting with orange as the first available color after core coloring is complete. This iteration of the product includes the FRs of Improved Sound Output and Improved Amplification, so will include new vertices. Since the two vertices labelled **1**, v_{13} (Improved Amplification) and v_{11} (Bluetooth and Wi-Fi), have no conflict, they are both labelled orange. The lone uncolored vertex labelled **2**, v_{14} (Manual Level Control), is colored purple. Finally, the two vertices labelled **3** have a conflict in the form of a shared edge, so they are colored yellow for v_{10} (Sound Input) and light blue for v_{12} (Improved Sound Output). Thus seven colors complete the coloring of G_{S_3}, indicating that the FRs could be combined into seven independent parts when the Hi-Fi Model of the speaker is built (see Figure 13c for a rendering):

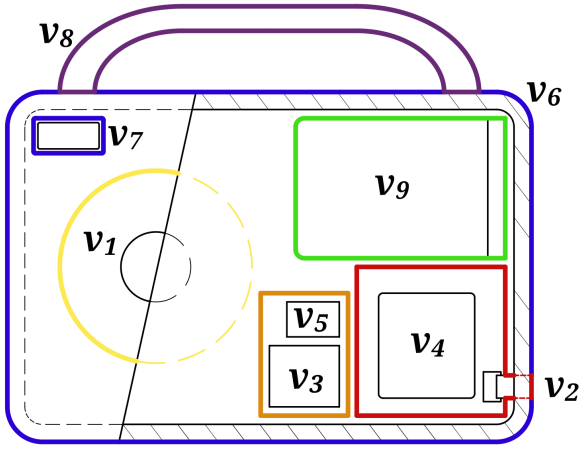
G_{S_3} Hi-Fi Model Coloring Interpreted as Parts

Red	Single circuit board with power supply (Power Input) and system on a chip (Computation)
Blue	Case (Enclosure) with built-in on/off button (Actuation)

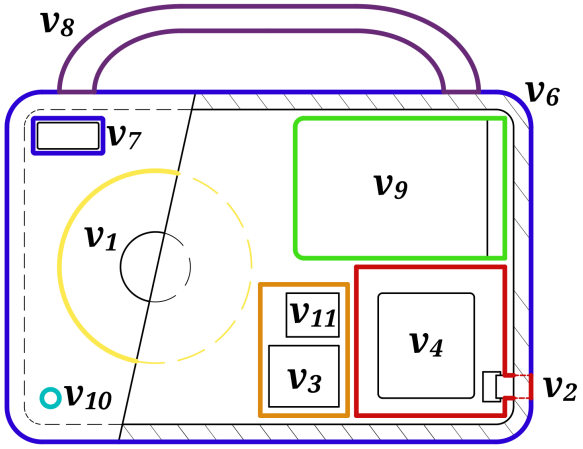
Green	Replaceable battery (Power Storage)
Orange	Circuit board with a 100 Watt 50 db S/N Amplifier (Improved Amplification) and a combined Bluetooth and Wi-Fi antennae (Bluetooth and Wi-Fi)
Purple	Volume Buttons (Manual Level Control)
Yellow	Microphone (Sound Input)
Light Blue	50 Watt Speaker (Improved Sound Output)

A final coloring of the input graphs is shown in Figure 12. A rendering of possible build structures for the three iterations is shown in Figure 13. For each iteration, every color in the final coloring has been interpreted as a part with one or more components. These parts are outlined in the color used in the corresponding graph for each iteration. Note that this technique allows the designer to clearly see which FRs remain fixed over time, and which can be modified over time. Additionally, the changes to the product over time, as FR's are added and removed, is clearly rendered in both the coloring graphs and the final renderings.

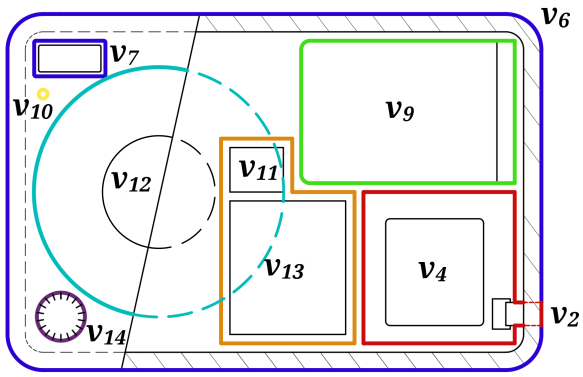
This is a small-scale example of how our proposed coloring method can be used to design for multiple product iterations, and as such is quite simple. We include this example to show how the method works, and to indicate the utility of our proposed algorithm in cases with many more iterations and FRs.



(a) Basic Model Rendering



(b) Voice Assistant Model Rendering



(c) Hi-Fi Model Rendering

Fig. 13: Finished product renderings with colors interpreted as parts.

4 CONCLUSION

This research deals with analyzing the concept of physical integration originated in axiomatic design field. Two approaches are explored and tested. In the first and more general approach, a graph partitioning method is proposed for determining the optimal co-allocation of FRs into distinct physical parts. The proposed method exemplified by designing a pencil, which initially is made of 9 Functional Requirements. It has been shown that the number of parts can be reduced to 5 parts where all the FRs are independent. In the second approach, a graph unification method is suggested to combine different product iterations together under one core graph and further, a graph coloring algorithm is developed to determine the minimum number of parts needed to accommodate dynamic changes in product functionalities. This approach is demonstrated in our example of a IoT speaker with several design iterations. Our algorithm, when applied to the graph representations of the product, quickly identifies those FRs that are common and have consistent mathematical structure between iterations in the form of the Core Graph, and breaks those FRs into a minimal number of parts. Once the core of the product is fixed, the characteristics of the different iterations give rise to sets of parts particular to those iterations. Future iterations beyond those used to compile the core graph may nonetheless use it as a starting point.

The graph partitioning approach can be extended to more complex designs with a greater number of parts and FRs. The implementation of the outputs of the proposed algorithm is feasible through the recent advancement in additive manufacturing where parts with different geometries and shapes are manufactureable. Another area for future research is to study the economic viability and efficiency of physically integrated parts considering that the number of parts, manufacturing processes, and assembly times may be reduced as technologies advance. The edge weights chosen for the input graph can be amended to reflect these changing realities. In addition, the proposed algorithm can be run

for more complicated designs to better reveal the performance of the algorithm under different conditions. In the case of complex systems, not every part needs to be printed at the same time. Instead integrated parts can be printed separately and be assembled together to reduce the operation time and cost.

The graph coloring approach for IoT devices can be extended in several ways. First, the model can be applied to a real case of a smart device and its corresponding service contract, which has an abstract but quantifiable existence. Second, graph coloring techniques can be augmented with uncertainty quantification models to further study the effects of uncertain consumer behavior, market changes, and technological advancement. Third, graph coloring algorithms can be integrated with economic models to quantify the business case of design decisions. Finally, the concept of design for flexibility can be integrated with the circular economy concept to study the sustainability impacts of flexible design.

The combination of novel manufacturing methods, embedded software, and smart devices has fundamentally changed the way products and services are designed and offered to customers. These changes require a new set of design methods such as those explored in this paper, which empower designers to see the impacts of adding or revising product Functional Requirements with the aim of improving their design decisions.

Literature Cited

[1] K. Allen and S. Carlson-Skalak, "Defining product architecture during conceptual design," in *Proceedings of the ASME Design Engineering Technical Conference*, 1998.

[2] D. Bhasin, D. A. McAdams, and A. Layton, "A product architecture-based tool for bioinspired function-sharing," *J. Mech. Des.*, vol. 143, no. 8, p. 81401, 2021.

[3] J. Bondy and U. Murty, *Graph Theory with Applications*. Citeseer, 1976.

[4] E. Bonjour, S. Denlaud, M. Dulmet, and G. Harmel, "A fuzzy method for propagating functional architecture constraints to physical architecture," *J. Mech. Des.*, vol. 131, no. 6, 2009.

[5] P. Borisovsky, A. Dolgui, and S. Kovalev, "Algorithms and implementation of a set partitioning approach for modular machining line design," *Computers Operations Research*, vol. 39, no. 12, pp. 3147–3155, 2012.

[6] J. Boshire, S. Wang, M. Khasawneh, T. Gandhi, and K. Srihari, "Designing an integrated surgical care delivery system using axiomatic design and petri net modeling," *Annals of Information Systems*, vol. 19, pp. 73–101, 2016.

[7] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz, *Recent Advances in Graph Partitioning*. Springer, 2016, pp. 117–158.

[8] J. M. Byskov, "Enumerating maximal independent sets with applications to graph colouring," *Operations Research Letters*, vol. 32, pp. 547–556, November 2004.

[9] D. S. Cochran, W. Eversheim, G. Kubin, and M. L. Sesterhenn, "The application of axiomatic design and lean management principles in the scope of productions system segmentation," *International Journal of Production Research*, vol. 38, no. 6, pp. 1377–1396, 2000.

[10] R. Crootof, "The internet of torts: Expanding civil liability standards to address corporate remote intereference," *Duke Law Journal*, vol. 69, no. 3, pp. 583–667, December 2019.

[11] S. Devanathan, S. Rananujan, D. Bernstein, W. Z. Zhao, and K. Ramani, "Integration of sustainability into early design through the function impact matrix," *J. Mech. Des.*, vol. 132, no. 8, p. 81004, 2010.

[12] X. Du, "Uncertainty analysis with probability and evidence theories," in *Proceedings of the ASME Design Engineering Technical Conference*, ser. American Society of Mechanical Engineers Digital Collection, vol. **1**, September 2006, pp. 1025–1038, paper number DETC2006-99078.

[13] D. Eppstein, "Small maximal independent sets and faster exact graph coloring," *Journal of Graph Algorithms and Applications*, vol. 7, no. 2, pp. 131–140, 2003.

[14] S. Estrada, E. Green, K. Gopalakrishnan, S. Jahanbekam, and S. Behdad, "Design for flexibility: A graph coloring technique to study design changes in the tethered economy world," in *ASME 2020 IDETC/CIE*, 2020.

[15] L. Fan, M. Cai, Y. Lin, and W. J. Zhang, "Axiomatic design theory: Further notes and its guideline to applications," *International Journal of Materials and Product Technology*, vol. 51, no. 4, pp. 359–374, 2015.

[16] S. H. Fan, J. H. Li, Z. Jiang, and Z. G. Zhang, "Axiomatic design of facility layout for reconfigurable manufacturing system," *Applied Mechanics and Materials*, vol. 703, pp. 273–276, 2014.

[17] A. M. Farid and N. P. Suh, *Axiomatic Design in Large Systems*. Spring International Publishing, Cham, 2016.

[18] X. Fern and C. Brodley, "Solving cluster ensemble problems by bipartite graph partitioning," in *Proceedings of the Twenty-First International Conference on*

Machine Learning. New York, NY, USA: Association for Computing Machinery, 2004, p. 36.

[19] P. Foith-Forster, M. Wiedenmann, D. Seichter, and T. Bauernhansl, "Axiomatic approach to flexible and changeable production system design," *Procedia CIRP*, vol. 53, pp. 8–14, 2016.

[20] J. T. Foley and S. Hardardóttir, "Creative axiomatic design," *Procedia CIRP*, vol. 50, pp. 240–245, 2016.

[21] J. Gershenson, G. Prasad, and Y. Zhang, "Product modularity: Definitions and benefits," *Journal of Engineering Design*, vol. 14, no. 3, pp. 295–313, 2003.

[22] A. Girgenti, A. Giorgetti, P. Citti, and M. Romanelli, "Development of a custom software for processing the stress corrosion experimental data through axiomatic design," *Procedia CIRP*, vol. 34, pp. 250–255, 2015.

[23] P. Gopalakrishnan, J. Cavallaro, S. Jahanbekam, and S. Behdad, "A graph coloring technique for identifying the minimum number of parts for physical integration in product design," in *International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, ser. American Society of Mechanical Engineers Digital Collection, vol. **4**, August 2019, paper number DETC2019-98251.

[24] E. Green, S. Estrada, K. Gopalakrishnan, S. Jahanbekam, and S. Behdad, "A graph partitioning technique to optimize the physical integration of functional requirements," *submitted*, 2021.

[25] G. Hazelrigg, "Validation of engineering design alternative selection methods," *Engineering Optimization*, vol. 35, no. 2, pp. 103–120, 2003.

[26] B. Hendrickson and T. G. Kolda, "Graph partitioning models for parallel computing," *Parallel Computing*, vol. 26, no. 12, pp. 1519–1534, 2000.

[27] H. Hirani and N. Suh, "Journal bearing design using multi-objective genetic algorithm and axiomatic design approaches," *Tribology International*, vol. 38, no. 5, pp. 481–491, 2005.

[28] C. Hoofnagle, A. Kesari, and A. Perzanowski, "The tethered economy," *George Washington Law Review*, vol. 87, no. 4, p. 783, July 2019.

[29] K. Ishii, C. Juengel, and C. Eubanks, "Design for product variety: Key to product line structuring," in *Proceedings of the ASME Design Engineering Technical Conference*, September 1995.

[30] D. Kiritsis, A. Bufardi, and P. Xirouchakis, "Research issues on product lifecycle management and information tracking using smart embedded systems," *Advanced Engineering Informatics*, vol. 17, no. 3, pp. 189–202, July 2003.

[31] C. F. Kirschman and G. M. Fadel, "Classifying functions for mechanical design," *J. Mech. Des.*, vol. 120, no. 3, pp. 475–482, 1998.

[32] L. Kjaer, D. Pigosso, M. Niero, N. Bech, and T. McAloone, "Product/service systems for a circular economy: The route to decoupling economic growth from resource consumption?" *Journal of Industrial Ecology*, vol. 23, no. 1, pp. 22–35, February 2019.

[33] T. Kurtoglu and I. Y. Tumer, "A graph-based fault identification and propagation framework for functional design of complex systems," *J. Mech. Des.*, vol. 130, no. 5, 2008.

[34] M. Lewandowski, "Designing the buisness models for circular economy - towards the conceptual framework," *Sustainability*, vol. 8, no. 1, p. 43, January 2016.

[35] M. Li, Y. F. Zhang, and J. Y. H. Fuh, "Retrieving reusable 3d cad models using knowledge-driven dependency graph partitioning," *Computer-Aided Design and Applications*, vol. 7, no. 3, pp. 417–430, 2010.

[36] J. Maier and G. Fadel, "Affordance based design: A relational theory for design," *Research in Engineering Design*, vol. 20, no. 1, pp. 13–27, 2009.

[37] A. Olewnik and K. Lewis, "On validating engineering design decision support tools," *Concurrent Engineering*, vol. 13, no. 2, pp. 111–122, 2005.

[38] P. Papalambros, "Optimal design of mechanical engineering systems," *Journal of Vibration and Acoustics*, vol. 117, pp. 55–62, 1995.

[39] G. J. Park, *Analytic Methods for Design Practice*. Springer Science Business Media, 2007.

[40] ——, "Teaching conceptual design using axiomatic design to engineering students and practitioners," *Journal of Mechanical Science and Technology*, vol. 28, no. 3, pp. 989–998, 2014.

[41] K. Salonitis, "Design for additive manufacturing based on the axiomatic design method," *The International Journal of Advanced Manufacturing Technology*, vol. 87, pp. 989–996, 2016.

[42] C. Schulz, S. K. Bayer, C. Hess, C. Steiger, M. Teichmann, J. Jacob, F. Bernardes-lima, R. Hangu, and S. Hayrapetyan, "Course notes: Graph partitioning and graph clustering in theory and practice," 2015.

[43] R. A. Shirwaiker and G. E. Okudan, "Triz and axiomatic design: A review of case-studies and a proposed synergistic use," *Journal of Intelligent Manufacturing*, vol. 19, no. 1, pp. 33–47, 2008.

[44] N. P. Suh, "Axiomatic design theory for systems," *Research in Engineering Design*, vol. 10, no. 4, pp. 189–209, 1998.

[45] ——, *Axiomatic Design in Large Systems*. Spring International Publishing, Cham, 2016.

[46] S. Svensson, J. Richter, E. Maitre-Ekern, R. Pihlajarinne, A. Maigret, and C. Dalhammar, "The emerging 'right to repair' legislation in the eu and the u.s."

Paper Presented at Going Green CARE INNOVATION 2018 Conference, November 2018.

[47] D. Tang, G. Zhang, and S. Dai, "Design as integration of axiomatic design and design structure matrix," *Robotics and Computer-Integrated Manufacturing*, vol. 25, no. 3, pp. 610–619, 2009.

[48] D. Tate, *Volume 15: Advances in Multidisciplinary Engineering*. ASME, 2015.

[49] K. Ulrich and S. Eppinger, *Product Design and Development*. McGraw Hill, 1995.

[50] S. Vinodh and G. Rathod, "Application of life cycle assessment and monte carlo simulation for enabling sustainable product design," *Journal of Engineering, Design and Technology*, vol. 12, no. 3, 2014.

[51] F. Zarali and H. R. Yazgan, "Solution of logistics center selection problem using the axiomatic design method," *World Academy of Science, Engineering and Technology International Journal of Computer, Electrical, Automation, Control and Information Engineering*, vol. 10, no. 3, pp. 489–495, 2016.

[52] A. Zetterburg, U. Mortberg, and B. Balfors, "Making graph theory operational for landscape ecological assessments, planning, and design,," *Landscape and Urban Planning*, vol. 95, no. 4, pp. 181–191, 2010.

[53] G. Zhang, E. Morris, D. Allaire, and D. McAdams, "Research opportunities and challenges in engineering system evolution," *Journal of Mechanical Design*, vol. 142, no. 8, August 2020.

[54] Z. Zhang, L. Liu, W. Wei, F. Tao, T. Li, and A. Liu, "A systematic function recommendation process for data-driven product and service design," *J. Mech. Des.*, vol. 139, no. 11, 2017.

[55] Y. Zhao, J. Xu, and D. Thurston, "A hierarchical bayesian method for market positioning in environmentally conscious design," in *Proceedings of the ASME Design Engineering Technical Conference*, ser. American Society of Mechanical

Engineers Digital Collection, vol. **9**, January 2011, pp. 3–16, paper number DETC2011-47898.

[56] J. Zittrain, *The Future of the Internet - and How to Stop It*. New Haven, CT: Yale University Press, 2008.