Emilyn Green
Department of Mathematics and Statistics,
San José State University,
San José, CA 95192
e-mail: emilyn.green@sjsu.edu

Spenser Estrada
Department of Mathematics and Statistics,
San José State University,
San José, CA 95192
e-mail: spenser.estrada@sjsu.edu

Praveen Kumare Gopalakrishnan
Department of Mechanical and Aerospace Engineering,
University at Buffalo,
The State University of New York,
Buffalo, NY 14260
e-mail: pgopalak@buffalo.edu

Sogol Jahanbekam
Department of Mathematics and Statistics,
San José State University,
San José, CA 95192
e-mail: sogol.jahanbekam@sjsu.edu

Sara Behdad[1]
Department of Environmental Engineering Sciences,
University of Florida,
Gainesville, FL 32606
e-mails: sara.behdad@essie.ufl.edu;
sarabehdad@ufl.edu

A Graph Partitioning Technique to Optimize the Physical Integration of Functional Requirements for Axiomatic Design

According to the concept of physical integration as understood in axiomatic design, design parameters of a product should be integrated into a single physical part or a few parts with the aim of reducing the information content, while still satisfying the independence of functional requirement. 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. As new manufacturing 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. This study proposes an algorithm that 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 co-allocation of incompatible functional requirements in the same part. The operation and viability of the algorithm are demonstrated via two numerical examples and a practical example of designing a pencil. [DOI: 10.1115/1.4052702]

Keywords: design integration, design theory and methodology, graph partitioning, axiomatic design

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 (FRs). 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 [1]. 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 FRs—the actual purposes or functions of different parts of the final product—must remain as independent as possible [2]. The value of the independence axiom is to ensure that if one of the design parameters (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 [3].

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 DPs. According to the independence axiom, DPs must be chosen such that the independence of FRs are maintained. Once DPs are determined based on design embodiment principles, designers consider the process domain and identify the process variables (PVs). PVs often act as constraints in the system since designers are not free to change the existing manufacturing processes [2]. AD uses design matrices to relate FRs with 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 [4]. The relationship between the FRs and the DPs is characterized as follows:

$$\mathrm{FR} = [A]\mathrm{DP}$$

Here each element of matrix A, Aij, connects a component of the FR vector to a component of the DP vector [5]. 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 [6].

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

[1]Corresponding author.

Contributed by the Design Theory and Methodology Committee of ASME for publication in the Journal of Mechanical Design. Manuscript received April 14, 2021; final manuscript received October 1, 2021; published online December 6, 2021. Assoc. Editor: Tahira Reid Smith.

Table 1 Three different types of design matrices

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

systems [7], software design [8], production scheduling [9], manufacturing system design [10], supplier selection [11], interactive art [12], decision science [13], and additive manufacturing [14].

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 [15], consumer preference, market demand [16], manufacturing considerations, and the potential to require a preference structure of designers [17]. 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 [5]. 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 beverage can is designed to satisfy 12 FRs and has 12 DPs, but has only three pieces [2]. 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 nondestructive way as a unit [18,19]. Ishii et al. [20] have referred to modular design as minimizing the number of functions per part. According to Ulrich and Eppinger [21], the most modular design is one in which each function is implemented by exactly one module or subassembly, 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. Decomposition of FR-DP can result in concrete process variables, which is essential for practically applicable solutions. However, most of the complexities have been resorted between FRs and DPs [22–24]. 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 [25,26].

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 [27] 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. [28] 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. [29] 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 [30] 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. [31] 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. [32] 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.

In this article, 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 [33–35]. Buluç et al. [36] 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. While different graph partitioning techniques are used, these approaches tend to share certain basic algorithms [37]. As computing power evolves, multiple graph partitions can be run in parallel, and ever more complex systems can be analyzed [38,39].

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 article. Li et al. [40] used graph partitioning techniques to extract reusable 3D computer-aided design models to improve design reusability. Borisovsky et al. [41] 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.

In this article, we have taken a graph partitioning approach that which assigns a weighted value to potential conflicts between functional requirements, in order to construct a systematic method for achieving physical integration in design. Integrating functional requirements facilitates fewer assembly parts, greater flexibility, and less logical efforts.

The algorithm proposed in this article provides a new method for optimizing physical integration, which is especially relevant as new manufacturing technologies emerge, for example, additive manufacturing, which enable novel configurations 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 Proposed Graph Partitioning Method

This article introduces a method to determine the optimal distribution of functional requirements among k parts in a product, given that the functional requirements may have varying degrees of compatibility with one another. We will use graphs to model the relationships between functional requirements in a product and graph partitioning in the algorithms 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. In this article, each of the n vertices $v_1, \ldots v_n$, represents one of the n FRs that we are attempting to group. We will use identification of vertices in our attempt to find an optimal grouping. To identify two vertices in a graph G means that we contract or merge two vertices v_i and v_j. This process entails forming a new vertex $v_{i,j}$ in G. Furthermore, 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, if present, is removed from the graph. The graph that results from these operations is denoted as $G.v_{i,j}$, where $v_{i,j}$ is the new vertex created in the contraction process. This contraction of identified vertices is the primary graph operation in our algorithm.

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 $2 \leq k < n$, where k is the number of discrete parts desired in the final product and n is the total number of vertices in the graph, where each vertex represents and FR. We specify that $2 \leq k$ because if our desired k equals one, then the only solution possible is the trivial one wherein all FRs are condensed onto one part. If one desires, it is possible to use other types of weight besides integers, but whichever set of numbers is used, it should be an ordered one, i.e., "less than" and "greater than" must be definable attributes. The selection of weights is admittedly subjective and relies on the designer's expertise and interpretation of the relations between FRs. However, the strength of the algorithm is that it allows the designer to generate solutions to the FR grouping problem based on said expertise beyond what the designer can generate manually. This is especially important as the number of FRs increase: for a graph modeling n FRs, there are $n(n-1)$ possible edges representing the relations between the FRs to consider. Thus, as n increases, it becomes decreasingly feasible for the designer to conceptualize these relations. The designer labels the vertices of G as $v_1, \ldots, v_n$, representing the n FRs. The designer places an edge $e_{i,j}$ between each pair of vertices v_i, v_j. Each edge is assigned an integer 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 functional requirements associated with those vertices. In this notation system, the designer can indicate the degree to which they would prefer to keep two functional requirements in separate parts of the final product, where a higher edge weight indicates a stronger preference to keep functional requirements in separate parts. The designer will label the edge $e_{i,j}$ with the weight ∞ when it is not desirable or possible for two functional requirements v_i and v_j to be in the same part under any circumstances. In the case where there is no conflict between two functional requirements, the designer may assign a weight of 0, or omit that edge.

Algorithm Overview. We propose to enact this process using the recursive algorithm that is 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. Recall that the algorithm works by attempting different identifications of vertices. Recall also that the designer has assigned a certain weight, $\omega(e_{i,j})$, to each edge $e_{i,j}$ in G. Each time when vertices v_i and v_j are identified and contracted, the weight of the edge between them, $\omega(e_{i,j})$, will be added to what is known as the cost of G, which starts at zero before the algorithm has taken any action. The algorithm will recur among all possible vertex identifications in the graph G while comparing the cost incurred by those graphs, which have k vertices and searching for the one with minimum cost. For a graph to have minimum cost implies that the fewest number of vertices with great weight in between them, i.e., the vertices representing the most incompatible FRs, have been identified and contracted together. When this graph with minimum cost is found, it 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, which is graph with k vertices and minimum cost. It is then up to the human designer to take that solution and use it to develop a physical product whose FRs are grouped in accordance with the configuration of G'. We shall now explain in depth the algorithm's process, starting with its inputs and then the functions `main`, `recur`, and `contract` in-turn.

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.

Function `recur`. This is a recursive function that 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 prevent further exploration when a solution of minimum cost 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 now 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 iterate 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 $\{v_1, v_2, \ldots, v_n\}$. Suppose vertices v_1 and v_2 are identified and contracted into one vertex. Then, the new vertex set will be $\{v_{1,2}, v_3, \ldots, v_n\}$.

(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 contracts the vertices v_i, v_j into the new vertex $v_{i,j}$. The graph produced is called $G.v_{i,j}$. 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 $G.v_{i,j}$ 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 $G.v_{i,j}$ for solutions, supposing they exist. Only then can we proceed to line **9**. Note that we are *not* proceeding to line **9** with $G.v_{i,j}$, 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 $G \cup e_{i,j}$.
(10) The function calls `recur`, which effectively sends us back to line **1**, but with $G \cup e_{i,j}$ 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 $G \cup e_{i,j}$ for solutions, supposing they exist. Then, we return to line **6** and move on to the next pair of vertices.

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.
(6) 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.
(7) We make v_y adjacent to the new vertex $v_{i,j}$ created in line **2** by creating a new edge, which we call $e_{i,j,y}$.
(8) 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.
(9) The new vertex $v_{i,j}$ is now correctly initialized, so we discard v_i and v_j. The graph G is now called $G.v_{i,j}$.

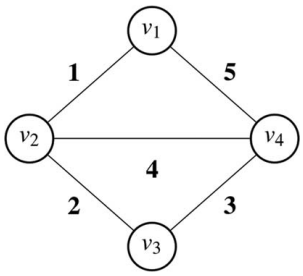

Fig. 1 Example graph G_1

`FUNCTION 1` `main`

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 $minCost = \infty$;
2 Let *graphStack* be a new stack;
3 recur(G);
4 pop G' off of *graphStack*;

`FUNCTION 2` `recur`

Input: A graph G
Output: void
Result: A list of graphs contracted from G having k vertices each
1 **if** $G.cost \leq minCost$ 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** $\omega(e_{i,j}) < \infty$ **then**
8 recur(contract(G,v_i,v_j));
9 $\omega(e_{i,j}) = \infty$;
10 recur(G);
11 **end**
12 **end**
13 **end**

`FUNCTION 3` `contract`

Input: A graph G, and two vertices v_i, v_j in G
Output: G with v_i and v_j contracted
1 $G.cost \mathrel{+}= \omega(e_{i,j})$;
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 $\omega(e_{i,j,x}) \mathrel{+}= \omega(e_{i,x})$);
6 **end**
7 **for** each vertex v_y adjacent to v_j **do**
8 make v_y adjacent to $v_{i,j}$;
9 $\omega(e_{i,j,y}) \mathrel{+}= \omega(e_{j,y})$;
10 **end**
11 delete v_i, v_j from G;

3 Numerical Example

In this section, we provide two numerical examples that illustrates how the algorithm acts on a graph to assign its functional requirements to a fixed number of parts. In the first example, we attempt to optimally assign the four functional requirements represented by the vertices v_1, v_2, v_3, and v_4 of the graph G_1 into three parts. The graph G_1 itself is shown in Fig. 1. The weights between 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 Fig. 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 cost from 0 to 1. The graph now has

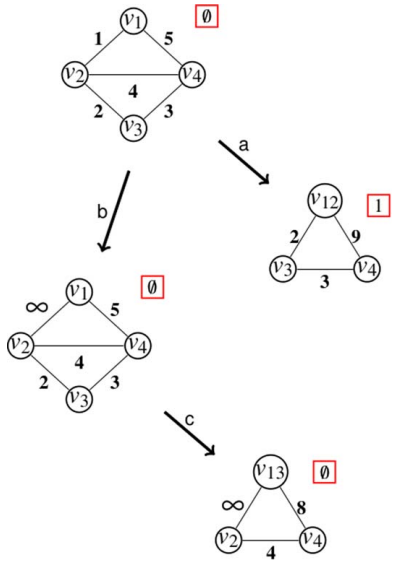

Fig. 2 The algorithm reduces G_1 to three parts

three parts as desired and finite cost, 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 cost is incurred from this action, and the graph now has three parts as required and has lesser cost than our previous graph obtained in a. Furthermore, since no lesser cost 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 cost. But this is not the case by necessity, and we should not in general expect a result with 0 cost. 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 cost part-wise. That is, the cost 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.

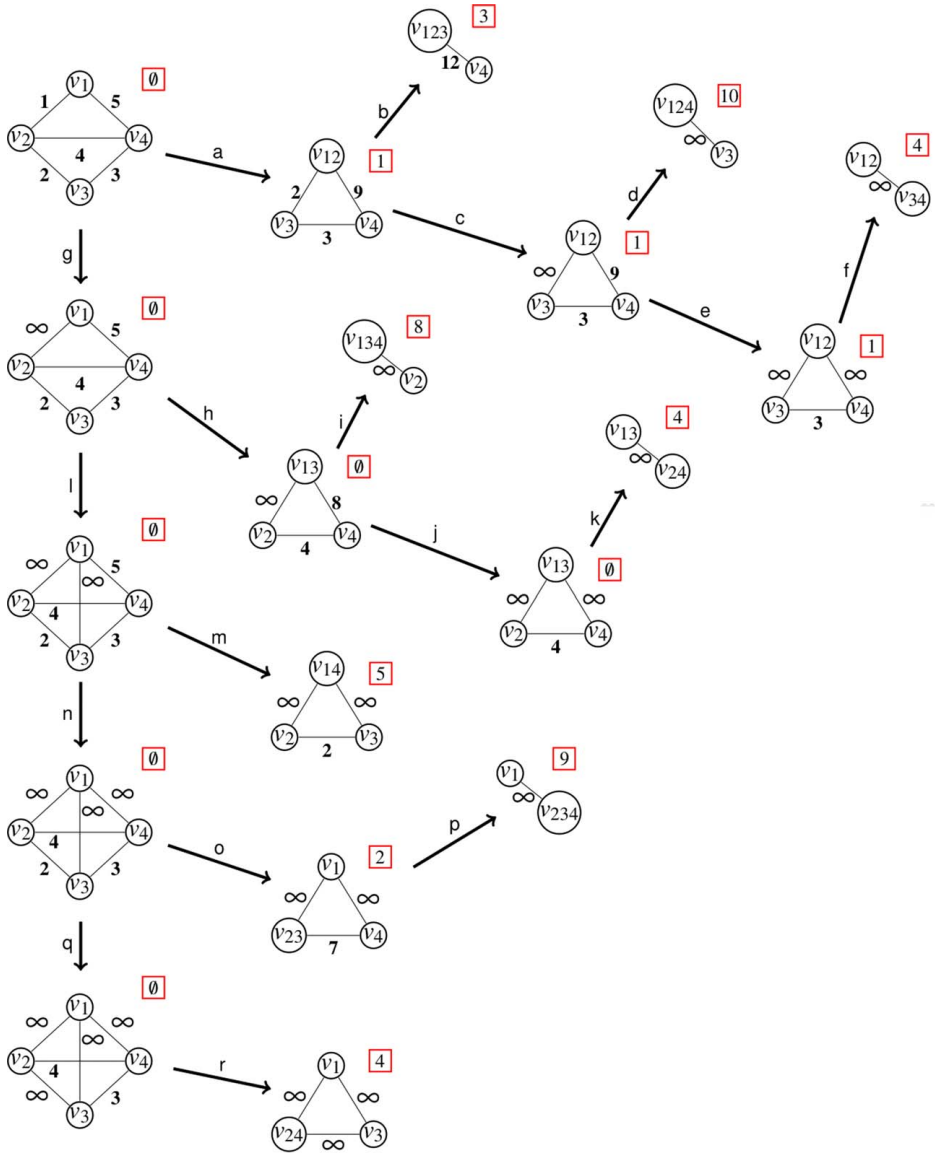

Fig. 3 The algorithm reduces G_1 TO two parts

Suppose instead that we desire to reduce the graph G_1 to two parts. Our algorithm is able to accomplish this just as well, though the generated process is more complex. The process is illustrated in Fig. 3. As mentioned earlier, 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 Fig. 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 cost from 0 to 1. The graph now has three 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 cost from 1 to 3. The graph now has two parts as desired and finite cost, 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 cost from 1 to 10. The graph now has two parts as desired, but its cost 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 ∞.
(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 cost from 1 to 4. The graph now has two parts as desired, but its cost 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 cost 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 cost from 0 to 8. The graph now has two parts as desired, but its cost 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 cost from 0 to 4. The graph now has two parts as desired, but its cost 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 cost 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 cost 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 cost from 2 to 9. The graph now has two parts as desired, but its cost 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 two 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 cost is higher than that obtained in step b. The graph obtained in step b with k parts and cost 3, which is now assured to be the minimum of all generated, is returned as the solution.

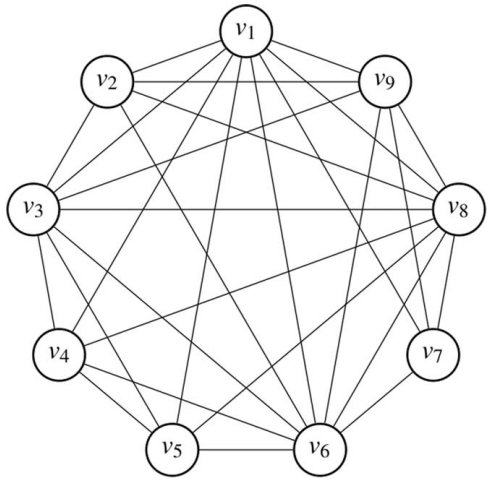

Fig. 4 The graph G_P, showing the relations between the nine functional requirements. Weights are tabulated in the text.

4 Practical Example: Pencil Design

In the previous section, we demonstrated the result of the algorithm on two numerical examples. In this section, we illustrate a more practical case. Suppose we wished to design a mechanical pencil. Such a device could have the following functional requirements depending on the type of pencil:

FR1:	Allow erasing	**FR6**:	Position lead
FR2:	Store lead	**FR7**:	Be hand-holdable
FR3:	Store erasure material	**FR8**:	Allow user to attach to clothes or paper sheets
FR4:	Extrude lead		
FR5:	Grasp lead	**FR9**:	Protect internal parts

The designer in this case has decided to segregate FR1 and FR6 from all the other FRs. In the case of FR1, it is because the designer believes that a part that can allow erasing would be made of an self-ablative substance that cannot satisfactorily embody any of the other FRs enumerated. Similarly, for FR6, the function of positioning lead requires a part of precise dimension as to mate properly with the lead, but the designer does not believe such precision is needed to meet the other FRs. Thus, 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 as follows.

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

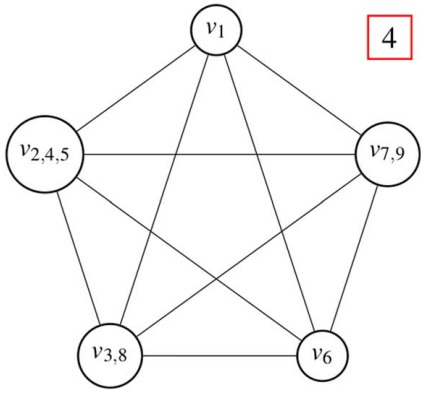

Fig. 5 Graph G_P', after the algorithm has reduced it to five parts. It has cost 4 as indicated in the box.

Fig. 6 A five-part pencil embodying G_P'

The graph representing these FRs and edges is shown in Fig. 4. It has nine vertices v_1, v_2, …, v_9, representing the nine 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 is shown in Fig. 5, where G_p' is displayed along with the cost accrued in creating it. As desired, FR1 and FR6 are their own parts, while FR2, FR4, and FR5 are grouped; FR3 and FR8 are grouped; and FR7 and FR9 are grouped. This matches the set goal of five parts. This tells the designer that according to the criteria that they input, this grouping of FRs minimizes the incidence of incompatible FRs being co-allocated together. It is then the designer's job to create a design that takes this recommendation into account. For example, in Fig. 6, we see an actual pencil that embodies this solution, where the FRs are allocated to the pencil's parts in the following manner:

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

Of course, this is not the only possible design that satisfies this grouping of FRs. But the goal of the algorithm is not to find the specific design, only a grouping of FRs that is favorable according to the designer's criteria. Design is a complex decision-making process moving from the voice of customers to FRs and then converting FRs in the conceptual design phase to DPs and process parameters in the physical domain phase. Once designers identify the list of FRs needed in a design, they may look for solution principles currently available in the market and physical domain incorporate the FRs. For example, if the required function is to facilitate "rotation to translation" in a mechanical device, designers based on previous experience and available solutions principles in the market realize that a "crank-slider" or a "rack-and-pinion" mechanism is needed in the design. Our proposed algorithm can be applied right before designers select the proper solution principles available in the physical domain at which point it can guide them in the process of combining FRs and the process of finding a solution principle to address multiple functions at the same time. It should be noted that the proposed algorithm is by no means a replacement for designer expertise or the feasibility analysis of specific designs, but it rather a tool to help them quantify their extant knowledge on the feasibility of combining FRs and consider it as an input while selecting the proper solution principle in the physical domain.

5 Conclusion

This research deals with analyzing the concept of physical integration originated in axiomatic design field. A graph partitioning method is proposed for determining the best pairs of FRs that can be physically integrated into a single part. The proposed method is employed for two numerical examples and one example of designing a pencil, which initially is made of nine FRs. It has been shown that the number of parts can be reduced to five parts where all the FRs are independent and serves its purpose.

This research can be extended in several ways. The algorithm can be extended to determine the optimum assignment of FRs while satisfying the independence of the FRs as one of the main principles of axiomatic design. In addition, the information content of each design alternative can be calculated as another factor to be added to the algorithm. Furthermore, the proposed method can be extended to determine the optimal number of parts needed to satisfy the predefined set of FRs. The proposed method is one step toward developing methods that can help designers define the optimal degree of physical integration for design alternatives. An important area of concern is the degree to which the algorithm is sensitive to variances in input, especially since its input parameters have a degree of subjectivity. Sensitivity analyses should be run to investigate the impacts of subjective parameters selected by different designers and different levels of expertise, which would enable comparison of the outputs. Also, it should be investigated what effect, if any, using different number sets besides the integers might have on the algorithm output, with the goal of making sure designers are enabled to assign weights in as near an objective way as possible.

This study can be extended to more complex designs with more 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 manufacturable. 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. 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.

Acknowledgment

This material is based upon work supported by the National Science Foundation, USA under Grant Nos. 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.

Conflict of Interest

There are no conflicts of interest.

Nomenclature

G = a graph composed of a vertex set $V(G)$ and an edge set $E(G)$
$|E|$ = the number of edges in a graph G
$|V|$ = the number of vertices in a graph G
$e_{i,j}$ = an edge in a graph between vertices v_i and v_j, where i and j are natural numbers
$v_1, v_2, \ldots, v_n$ = the n vertices of a graph G
$G + e_{i,j}$ = the graph formed from G by adding the edge $e_{i,j}$
$G \cdot v_{i,j}$ = the graph formed from G by identifying the vertices v_i and v_j into the single vertex $v_{i,j}$
$G \cup e_{i,j}$ = the graph formed from G by placing an edge of weight ∞ between the vertices v_i and v_j

References

[1] Shirwaiker, R. A., and Okudan, G. E., 2008, "Triz and Axiomatic Design: A Review of Case-studies and a Proposed Synergistic Use," J. Intell. Manuf., **19**(1), pp. 33–47.
[2] Suh, N. P., 1998, "Axiomatic Design Theory for Systems," Res. Eng. Des., **10**(4), pp. 189–209.
[3] Suh, N. P., 2016, *Axiomatic Design in Large Systems*, Spring International Publishing, Cham.
[4] Farid, A. M., and Suh, N. P., 2016, *Axiomatic Design in Large Systems*, Spring International Publishing, Cham.
[5] Park, G. J., 2014, "Teaching Conceptual Design Using Axiomatic Design to Engineering Students and Practitioners," J. Mech. Sci. Technol., **28**(3), pp. 989–998.
[6] Hirani, H., and Suh, N., 2005, "Journal Bearing Design Using Multi-objective Genetic Algorithm and Axiomatic Design Approaches," Tribol. Int., **38**(5), pp. 481–491.
[7] Boshire, J., Wang, S., Khasawneh, M., Gandhi, T., and Srihari, K., 2016, *Ann. Inf. Syst.*, Vol. 19, Springer International Publishing, New York, pp. 73–101.
[8] Girgenti, A., Giorgetti, A., Citti, P., and Romanelli, M., 2015, "Development of a Custom Software for Processing the Stress Corrosion Experimental Data Through Axiomatic Design," Procedia CIRP, **34**, pp. 250–255.
[9] Cochran, D. S., Eversheim, W., Kubin, G., and Sesterhenn, M. L., 2000, "The Application of Axiomatic Design and Lean Management Principles in the Scope of Productions System Segmentation," Int. J. Prod. Res., **38**(6), pp. 1377–1396.
[10] Fan, S. H., Li, J. H., Jiang, Z., and Zhang, Z. G., 2014, "Axiomatic Design of Facility Layout for Reconfigurable Manufacturing System," Appl. Mech. Mater., **703**, pp. 273–276.
[11] Zarali, F., and Yazgan, H. R., 2016, "Solution of Logistics Center Selection Problem Using the Axiomatic Design Method," World Acad. Sci. Eng. Technol. Int. J. Comput. Electr. Autom. Control Inf. Eng., **10**(3), pp. 489–495.
[12] Foley, J. T., and Hardardóttir, S., 2016, "Creative Axiomatic Design," Procedia CIRP, **50**(4), pp. 240–245.
[13] Fan, L., Cai, M., Lin, Y., and Zhang, W. J., 2015, "Axiomatic Design Theory: Further Notes and Its Guideline to Applications," Int. J. Mater. Prod. Technol., **51**(4), pp. 359–374.
[14] Salonitis, K., 2016, "Design for Additive Manufacturing Based on the Axiomatic Design Method," Int. J. Adv. Manuf. Technol., **87**(1–4), pp. 989–996.
[15] Maier, J., and Fadel, G., 2009, "Affordance Based Design: A Relational Theory for Design," Res. Eng. Des., **20**(1), pp. 13–27.
[16] Hazelrigg, G., 2003, "Validation of Engineering Design Alternative Selection Methods," Eng. Optim., **35**(2), pp. 103–120.
[17] Olewnik, A., and Lewis, K., 2005, "On Validating Engineering Design Decision Support Tools," Concurrent Eng., **13**(2), pp. 111–122.
[18] Allen, K., and Carlson-Skalak, S., 1998, "Defining Product Architecture During Conceptual Design," Proceedings of the ASME 1998 Design Engineering Technical Conferences. Volume 3: 10th International Conference on Design Theory and Methodology, Atlanta, GA, Sept. 13–16.
[19] Gershenson, J., Prasad, G., and Zhang, Y., 2003, "Product Modularity: Definitions and Benefits," J. Eng. Des., **14**(3), pp. 295–313.
[20] Ishii, K., Juengel, C., and Eubanks, C., 1995, "Design for Product Variety: Key to Product Line Structuring," Proceedings of the ASME 1995 Design Engineering Technical Conferences collocated with the ASME 1995 15th International Computers in Engineering Conference and the ASME 1995 9th Annual Engineering Database Symposium. Volume 2: 11th Biennial Conference on Reliability, Stress Analysis, and Failure Prevention; 7th International Conference on Design Theory and Methodology; JSME Symposium on Design and Production; Mechanical Design Education and History; Computer-Integrated Concurrent Design Conference, Boston, MA, Sept. 17–20.
[21] Ulrich, K., and Eppinger, S., 1995, *Product Design and Development*, McGraw Hill Education, New York.
[22] Guenov, M., and Barker, S., 2005, "Application of Axiomatic Design and Design Structure Matrix to the Decomposition of Engineering Systems," Sys. Eng., **8**(1), pp. 29–40.
[23] Eppinger, S., and Browning, T., 2012, *Design Structure Matrix Methods and Applications*, MIT Press, Cambridge, MA.
[24] Pektas, S., and Pultar, M., 2006, "Modelling Detailed Information Flows in Building Design With the Parameter-Based Design Structure Matrix," Des. Stud., **27**(1), pp. 99–122.
[25] Tang, D., Zhang, G., and Dai, S., 2009, "Design as Integration of Axiomatic Design and Design Structure Matrix," Rob. Comput. Integr. Manuf., **25**(3), pp. 610–619.
[26] Foith-Forster, P., Wiedenmann, M., Seichter, D., and Bauernhansl, T., 2016, "Axiomatic Approach to Flexible and Changeable Production System Design," Procedia CIRP, **53**(2), pp. 8–14.
[27] Kirschman, C. F., and Fadel, G. M., 1998, "Classifying Functions for Mechanical Design," ASME J. Mech. Des., **120**(3), pp. 475–482.
[28] Bonjour, E., Denlaud, S., Dulmet, M., and Harmel, G., 2009, "A Fuzzy Method for Propagating Functional Architecture Constraints to Physical Architecture," ASME J. Mech. Des., **131**(6), p. 061002
[29] Devanathan, S., Rananujan, S., Bernstein, D., Zhao, W. Z., and Ramani, K., 2010, "Integration of Sustainability Into Early Design Through the Function Impact Matrix," ASME J. Mech. Des., **132**(8), p. 81004.
[30] Kurtoglu, T., and Tumer, I. Y., 2008, "A Graph-Based Fault Identification and Propagation Framework for Functional Design of Complex Systems," ASME J. Mech. Des., **130**(5), p. 051401
[31] Zhang, Z., Liu, L., Wei, W., Tao, F., Li, T., and Liu, A., 2017, "A Systematic Function Recommendation Process for Data-Driven Product and Service Design," ASME J. Mech. Des., **139**(11), p. 111404
[32] Bhasin, D., McAdams, D. A., and Layton, A., 2021, "A Product Architecture-Based Tool for Bioinspired Function-Sharing," ASME J. Mech. Des., **143**(8), p. 81401.
[33] Zetterburg, A., Mortberg, U., and Balfors, B., 2010, "Making Graph Theory Operational for Landscape Ecological Assessments, Planning, and Design," Landscape Urban Plann., **95**(4), pp. 181–191.
[34] Bondy, J., and Murty, U., 1976, *Graph Theory with Applications*, Springer International Publishing, New York.
[35] Papalambros, P., 1995, "Optimal Design of Mechanical Engineering Systems," J. Vib. Acoust., **117**(B), pp. 55–62.
[36] Buluç, A., Meyerhenke, H., Safro, I., Sanders, P., and Schulz, C., 2016, *Recent Advances in Graph Partitioning*, Springer, pp. 117–158.
[37] Bader, D., Meyerhenke, H., Sanders, P., Wagner, D., Teichmann, M., Jacob, J., Bernardes-lima, F., Hangu, R., and Hayrapetyan, S., 2012, "Graph partitioning and graph clustering," Proceedings of the 10th DIMACS implementation challenge workshop, Atlanta, GA, Feb. 13–14.
[38] Fern, X., and Brodley, C., 2004, "Solving Cluster Ensemble Problems by Bipartite Graph Partitioning," Proceedings of the Twenty-First International Conference on Machine Learning, Banff, Alberta, Canada, July 4–8, p. 36.
[39] Hendrickson, B., and Kolda, T. G., 2000, "Graph Partitioning Models for Parallel Computing," Parallel Comput., **26**(12), pp. 1519–1534.
[40] Li, M., Zhang, Y. F., and Fuh, J. Y. H., 2010, "Retrieving Reusable 3d Cad Models Using Knowledge-Driven Dependency Graph Partitioning," Comput.-Aided Des. Appl., **7**(3), pp. 417–430.
[41] Borisovsky, P., Dolgui, A., and Kovalev, S., 2012, "Algorithms and Implementation of a Set Partitioning Approach for Modular Machining Line Design," Comput. Oper. Res., **39**(12), pp. 3147–3155.