ASME Journal of Mechanisms and Robotics
Online journal at:
https://asmedigitalcollection.asme.org/mechanismsrobotics

Construction, Reduction, and Decomposition of Planar Mechanisms Via an Extended Pebble Game Framework

Zhijie Lyu
Department of Mechanical Engineering,
Computer-Aided Design and Innovation Lab,
Stony Brook University,
Stony Brook, NY 11794-2300
e-mail: zhijie.lyu@stonybrook.edu

Anurag Purwar[1]
Department of Mechanical Engineering,
Computer-Aided Design and Innovation Lab,
Stony Brook University,
Stony Brook, NY 11794-2300
e-mail: anurag.purwar@stonybrook.edu

This article introduces a computational framework for analyzing geometric constraint systems in planar mechanisms, built upon an extended pebble game framework tailored to the point-based model. The proposed approach synthesizes key ideas from classic geometric constraint-solving methods: degrees-of-freedom are tracked using pebbles, and directed edges capture algebraic dependencies between variables. Each geometric constraint—such as distance, line, angle, or angular relationships—is encoded as one or more edges in the constraint graph, consistent with the point-based modeling formalism. The analysis is two-phased: a reduction phase that eliminates redundant constraints and a decomposition phase that partitions the system into minimal, rigid Assur graphs. To enhance solving efficiency, the method adopts a skeleton-first, body-next strategy, supported by constraint-type-specific rules. As a result, the framework achieves the following: (1) robust handling of arbitrary mixtures of planar geometric constraints, including those involving rolling joints and circular gears; (2) efficient performance with an overall time complexity of $O(|V|^2)$ for reduction and decomposition, enabling real-time simulation suitable for a web-based computer-aided design application. By introducing conceptual edges and rule-based deduction from the point-based model, this method offers a unified and scalable tool for mobility analysis, constraint reduction, and structure-preserving decomposition of complex planar linkages. [DOI: 10.1115/1.4069901]

Keywords: algebraic graph theory, kinematic simulation, mobility, planar mechanisms, directed graphs, pebble game algorithm, kinematics, computational geometry, mechanism synthesis and analysis, theoretical and computational kinematics

1 Introduction

Kinematic simulation plays a vital role in reducing design costs and enabling rapid prototyping by allowing engineers to analyze and refine mechanisms without the need for physical prototypes [1]. Mechanism design has largely been an iterative process, where performance is evaluated through simulation, and configurations are adjusted accordingly. For example, evolutionary algorithms [2–4] are commonly employed to gradually improve designs by iterating through cycles of simulation, evaluation, and refinement [1]. Since each iteration depends on running simulations, reducing simulation time is crucial for making the design process more efficient and practical.

Recent advances in mechanism design have incorporated data-driven approaches, which rely on generating large sets of kinematic samples to train neural network models [5–14]. However, these approaches require efficient generation of massive datasets to be useful. One effective strategy in kinematic simulation is the idea of constraint decomposition, which breaks down the geometric constraint system of mechanisms into smaller, sequentially solvable subsystems. This method reduces numerical complexity and accelerates simulation times [15], thereby enhancing both optimization-based and data-driven design methodologies. Other recent works, such as Ref. [16], have therefore focused on establishing systematic guidelines for generating and validating synthetic datasets in engineering design, emphasizing the importance of representation, sampling strategy, and realism for effective downstream learning.

To achieve this, we adopt a *point-based constraint model* previously developed in our work [17], which provides an algebraic structure in which the constraint equations for distance, line (slot), angle (rotary actuator), and angle–angle (wheel/gear train) are constructed using point locations. Building on this foundation, we propose a two-phase algorithm: a *reduction* phase that reduces redundant components, followed by a *decomposition* phase that applies strongly connected component (SCC) analysis and topological sorting to construct a sequentially solvable subsystem hierarchy.

Our methodology integrates insights from multiple strands of prior work. The reduction process is conceptually related to the graph-constructive method of Fudos and Hoffmann [18], while our skeleton-first, body-next solving philosophy draws from the

[1]Corresponding author.

Contributed by Mechanisms and Robotics Committee of ASME for publication in the Journal of Mechanisms and Robotics. Manuscript received April 5, 2025; final manuscript received September 16, 2025; published online October 9, 2025. Assoc. Editor: Vincenzo Parenti Castelli.

decomposition–recombination strategy proposed by Ait-Aoudia and Foufou [19], which first solves the numerical system from the skeleton and then extends the solution to the full body that contains it. However, solving a constraint system for a linkage demands a solution path that originates from the ground. In this regard, the structural foundation of our decomposition builds upon the Pinned Laman Theorem [20], which ensures generic rigidity with respect to a fixed ground. Furthermore, our adaptation of the pebble game introduces two new constructs—*cluster* and *rigid structure*—that parallel the notion of "component" (i.e., maximal block) in the work of Lee and Streinu [21]. Unlike their setting, however, our framework accommodates multiple edge types, and these constructs have different meanings depending on their specific uses.

In terms of theoretical underpinnings, our independence tests and sparsity conditions align closely with the graded sparse graph theory developed by Lee and Streinu [21]. Their formalization of pebble game algorithms for (k, ℓ)-sparse graphs supports the correctness of our approach in the combinatorial sense. However, unlike their uniform treatment for all edges in the graph, we differentiate between static, dynamic, and conceptual edges, each governed by additional geometric semantics tailored to linkage mobility analysis. Furthermore, their other related work [22] utilizes a self-loop as the graph semantics for the slider, which differs from our setup.

Comparatively, our graph semantics is closer to what is proposed by Jackson and Owen [23], who model 2D point-line rigidity via a unified vertex-edge formulation. While their framework explicitly includes both point and line vertices (*P*- and *L*-vertices) with defined edge types (i.e., *PP*-, *PL*-, *LL*-edges), our point-based model avoids the use of *LL*-edges by encoding angle constraints through three-point relations. This abstraction helps prevent edge cases involving geometric degeneracy of rigid graphs constructed by a set of *LL*-edges.

Complementary to the pebble game algorithm, a related key work [24] by Berg and Jordán recasts the pebble-game-style rigidity tests as an orientation-and-reachability framework, yielding an $O(n^2)$ algorithm that computes rigid, redundantly rigid, M-connected, and maximal globally rigid components. For a broader background in graph theory and combinatorial optimization, see Frank's monograph [25].

Notably, this work makes the following contributions:

(1) We demonstrate the applicability of the pebble game framework to arbitrary combinations of planar revolute, prismatic, and rolling joints, including complex couplings such as gears and slots. In particular, the graph decomposition technique stems from the pinned Laman theorem presented in the works by Sljoka et al. [20], and the graph semantics stems from the point-based model [17].
(2) We propose a two-phase analysis algorithm—reduction and followed by decomposition-with $O(|V|^2)$ time complexity.
(3) By using the edge-cluster dictionary data structure, we treat the decomposition of linkages as finding the rigid structure from the combination of clusters, to further decompose the numerical system underlying the constraint graph.

The remainder of this article is structured as follows: Sec. 2 presents the reduction and decomposition procedure for the pebble game framework for mechanisms with distance constraints only, before we move to mechanisms with linear and/or angle-angle constraints in Sec. 3. Next, a planar geared linkage system is analyzed in detail in Sec. 4. Then, we discuss the potential use of our work in Sec. 5.

Furthermore, for ease of reading, we split the presentation of the point-based model into two parts: Sec. 2 introduces the first two types of edges, while Sec. 3 presents the remaining types of edges and their corresponding rules as they become necessary.

1.1 A Note on Terminology and Symbols.

While this article is intended for the mechanical engineering community, the terminology employed aligns more closely with graph theory conventions. For instance, elements commonly referred to as *joints* in mechanical design are denoted as *vertices* in this work. Following the nomenclature used by Jackson and Owen [23], each *P*-vertex is a *point* corresponding to a specific pair of (x, y)-coordinates in the numerical simulation. Similarly, what is typically called a *link*—rigid body comprising multiple points—is referred to here as a *P*-cluster, representing a rigid group of *P*-vertices. Throughout this article, the term *cluster* always refers to such a rigid group of vertices.

To refer to graph vertices more generally, we use the symbol v, regardless of their type. An edge between vertices i and j is written as v_iv_j, and its directed version is denoted $v_i\vec{v}_j$ when specifying its orientation is necessary.

The symbol P is context-dependent in our framework. In figures, P refers to a specific point in the linkage model (i.e., its geometric position), whereas in the text, P may also denote a set of graph vertices. To distinguish these uses, we explicitly refer to the location of v_i when discussing P_i in figures.

The length of a *PP*-edge v_iv_j is denoted by $|v_iv_j|$. Finally, for any set A, we use the standard notation $|A|$ to indicate the number of elements it contains.

2 The Classic Pebble Game Framework

We use the term *classic* to refer to cases where each edge in the graph corresponds to exactly one distance constraint, as established in prior foundational works [20,26,27]. Without delving into implementation details, the mobility analysis framework proceeds as follows:

(1) Extract the constraint edges and categorize them into two types: non-dynamic and dynamic.
(2) Analyze the non-dynamic edges using the constraint graph and reduce the constraint system as much as possible.
(3) Construct a pinned constraint graph from the non-dynamic and dynamic edges.
(4) Decompose the pinned constraint graph into Assur graphs, and determine the appropriate solver for each Assur graph based on the types of edges it contains.

The constraint edges are derived from a set of input matrices. As this process is peripheral to the pebble game itself, the details are deferred to the Appendix.

This section is organized as follows. First, we introduce the two primary types of edges in the point-based model, which can be directly integrated into the classic pebble game framework, and provide a simple example to illustrate the overall process. We then present the mathematical background and describe the algorithms used within the pebble game framework in the remainder of the section.

2.1 The *PP*-Edges in the Point-Based Model.

Following the convention from Jackson and Owen [23], we term the edges corresponding to the distance constraints *PP*-edges, from the fact that two *P*-vertices can define a *PP*-edge. However, there are two types of *PP*-edges in the point-based model: *static* and *dynamic*. Specifically, a static edge is a *PP*-edge with a constant length, and a dynamic edge is a *PP*-edge whose length is actuator-dependent. A *PP*-edge between v_i and v_j corresponds to the equation as shown in Eq. (1).

$$\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} - d_{i,j} = 0 \tag{1}$$

where $d_{i,j}$ is the distance between P_i and P_j. Figure 1 illustrates the two types of actuators in the point-based model, rotary and linear. For the rotary actuator, the dynamic edge is v_2v_3, whose length can be computed according to Eq. (2).

$$d_{3,2} = \sqrt{d_{1,3}^2 + d_{1,2}^2 - 2d_{1,3} \cdot d_{1,2} \cos(\alpha_{3,1,2})} \tag{2}$$

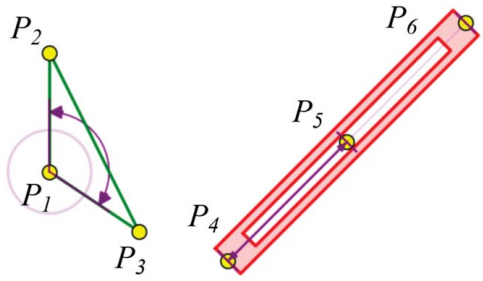

Fig. 1 Two types of dynamic edges: rotary (P_2, P_3) on the left, and linear (P_4, P_5) on the right

where $\alpha_{3,1,2}$ is the angle input from the rotary actuator defined by the three P-vertices v_1, v_2, and v_3. Comparatively, the length of a dynamic edge from a linear actuator is the output of a linear actuator, and we can directly get its value before solving any unknown variables.

In the pebble game framework, each degrees-of-freedom (DOF) is manifested with exactly one pebble. Figure 2 illustrates its application to a simple planar kinematic structure featuring a PP-edge and a rotary actuator. In this figure, v_1 and v_3 are the ground P-vertices. v_1 and v_2 define the PP-edge v_1v_2. The actuator is defined by the three P-vertices v_1, v_2, and v_3, and represented with the dynamic edge v_2v_3. In a pinned constraint graph, each ungrounded vertex is assigned two pebbles, and each grounded vertex has zero pebbles, as illustrated in Fig. 2(*b*). One edge is inserted into the graph at a time, at the cost of one pebble, as demonstrated in Figs. 2(*c*) and 2(*d*). Furthermore, when all the pebbles are used up, it has zero mobility, and we can create the solving path.

Specifically, the directed graph in Fig. 2(*d*) shows that v_2 depends on v_1 and v_3, suggesting that we can use the two distance constraints corresponding to $v_2\vec{v}_1$ and $v_2\vec{v}_3$, and the location of v_1 and v_3, to find the location of v_2. In practice, the location of v_2 is solved using arc intersection, with v_1 and v_3 serving as two circle centers whose radii are the $|v_2v_1|$ and $|v_2v_3|$, respectively. Finally, we add the cross-product constraint equation, as shown in Eq. (3), to determine the solution, because the graph includes the actuator loop defined by v_1, v_2, and v_3.

$$(x_3 - x_1)(y_2 - y_1) - (x_2 - x_1)(y_3 - y_1) - d_{3,1} \cdot d_{1,2} \cdot \sin(\alpha_{3,1,2}) = 0 \tag{3}$$

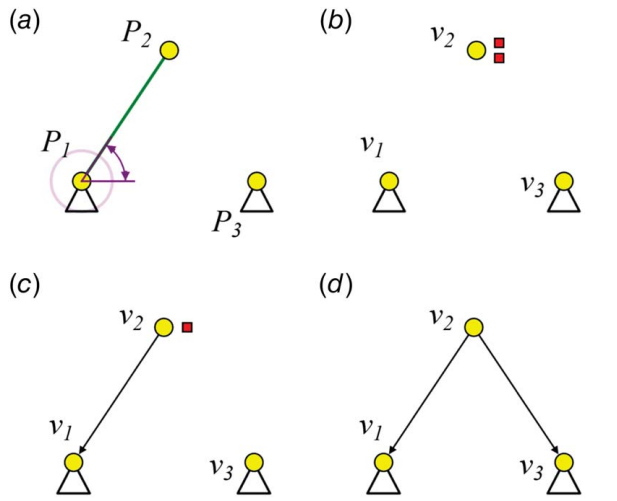

Fig. 2 Pebble assignment in a simple planar mechanism during pinned constraint graph construction

2.2 Math Preliminary for Mobility Analysis. The pebble game framework is an algorithmic tool designed to test the rigidity conditions defined by Laman's theorem [28], which states that:

A graph $G = G(V; E)$ (where V is the set of vertices and E is the set of edges) is generically minimally rigid, or (2, 3)-tight, if it satisfies the following:

(1) Cardinality condition: The graph has exactly $2|V| - 3$ edges, where $|V|$ is the number of vertices.
(2) Subset condition: For every subset of edges $E' \subseteq E$, the set of vertices V' from these edges satisfies: $|E'| \leq 2|V(E')| - 3$. This is also known as the (2, 3)-sparsity condition.

The three terms, "generic," "rigid," and "minimal" correspond to three different properties of a constraint graph. Specifically, the term "generic" is used against the edge cases (e.g., singularity cases) [29], where the exact distances among these vertices are non-trivial. For example, if the length sum of three edges in a four-bar linkage is equal to the length of the last edge, then all its four bars cannot move with respect to each other, while a generic four-bar linkage has one DOF when we fix one of the bars to the ground. Next, the term "rigid" means that every vertex within the graph does not change distance with respect to all other vertices in the graph. Finally, the word "minimal" means every edge within the graph is independent. In terms of edge counts, "minimal" corresponds to Laman's subset condition. If the graph is minimally rigid, then it satisfies both Laman's cardinality and subset conditions, and is (2, 3)-*tight*. Removing an edge from a (2, 3)-tight graph results in a generically flexible structure.

Following the paradigm introduced by Jacobs and Hendrickson [27], and later generalized by Lee and Streinu [21], a pebble is a conceptual token used to test whether an edge can be added to a graph without violating the (k, ℓ)-sparsity condition. For planar geometric constraint systems, the typical parameters are $k = 2$ and $\ell = 3$. The pebble game proceeds as follows:

(1) Each vertex is initialized with k pebbles.
(2) The directed graph begins with no edges.
(3) To insert an edge, the algorithm must collect at least $\ell + 1$ pebbles on its two endpoints.
(4) If successful, one pebble is removed and the edge is added with an orientation indicating the direction of pebble removal.
(5) Pebbles may be moved along existing directed edges, subject to specific rules, to facilitate gathering pebbles at the intended endpoints.

The first core operation is edge insertion. Each insertion comes with an independence test: an edge is considered *independent* if its addition preserves the (k, ℓ)-sparsity condition. Rules 3 and 4 above guarantee that, after inserting the edge and consuming one pebble, the graph and all its subgraphs retain at least ℓ free pebbles—thereby satisfying Laman's condition.

The second key operation is pebble redistribution, which reallocates pebbles by traversing directed paths. For instance, given edges $v_i\vec{v}_j$ and $v_j\vec{v}_k$, if v_i and v_j have no free pebbles but v_k has one, we can reverse the edges to $v_j\vec{v}_i$ and $v_k\vec{v}_j$, allowing v_i to reclaim a pebble. This process is shown in Fig. 4. We employ breadth first search (BFS) [30] to locate such paths efficiently.

Finally, we define the key terms used throughout this work. Some concepts span different levels of abstraction, as illustrated in Fig. 3.

- A *fixed* vertex is either a ground P-vertex or one assigned a specific solving method during decomposition. Otherwise, it is considered *unfixed*. For example, a ground P-vertex is fixed, because it does not change its location.
- *Pinning* a pebble refers to removing a free pebble from a vertex without adding an edge. It can later be restored through *unpinning*. A *pinned vertex* is one with zero free pebbles.
- An *edge* is a set of two vertices. When inserted as a *directed* edge, it consumes one free pebble from its tail. The *length* of a PP-edge is defined as the Euclidean distance between

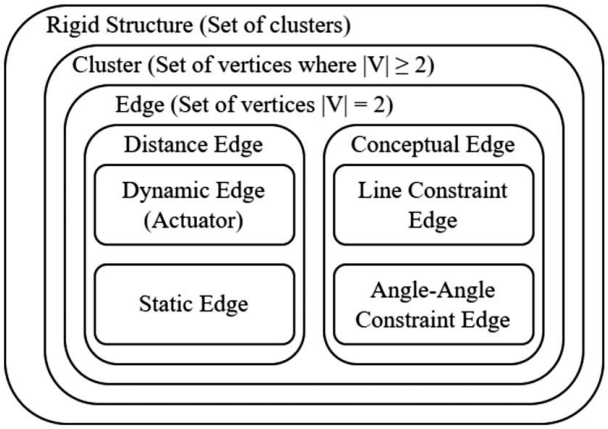

Fig. 3 Concept hierarchy

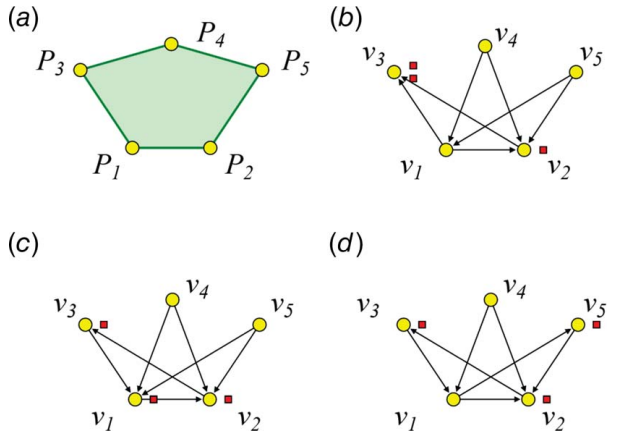

Fig. 4 (*a*) and (*b*) A *P*-cluster and its block representation; (*c*) and (*d*) two consecutive pebble redistributions—first from v_3 to v_1, then from v_1 to v_5

its two *P*-vertex endpoints. *PP*-edges are classified as either *static* (constant length) or *dynamic* (length depends on actuator input). An edge is called *conceptual* if one of its endpoints is not a *P*-vertex.

- A *block* is a (2, 3)-tight graph [21], with a single edge being the smallest possible block. A *P*-block contains only *P*-vertices.
- A *cluster* is a set of vertices that a block represents. A *P*-cluster is a *P*-block composed solely of static edges. Each dynamic or conceptual edge is treated as a cluster of two vertices. A cluster is unfixed if it contains any unfixed vertex.
- A *rigid structure* is a collection of clusters whose corresponding blocks together form a (2, 3)-tight subgraph. This concept is used in the decomposition phase, where rigidity is assessed based on the union of the blocks representing these clusters.

2.3 Representing a *P*-Cluster With a Minimal Set of Edges. In the classic case, a *P*-cluster contains two or more *P*-vertices. If it contains exactly two vertices, it can be represented by a single edge. For *P*-clusters with more than two vertices, a minimal representation can be constructed using Henneberg I moves [31]. Specifically, we begin by selecting two vertices to form an initial *PP*-edge, then iteratively connect each remaining vertex to the initial pair using two new edges. This process preserves minimal rigidity, as the base case satisfies Laman's condition ($|E| = 1 = 2 \times 2 - 3 = 2|V| - 3$), and each new vertex contributes exactly two edges, maintaining the (2, 3)-tight condition. See Fig. 4 for an illustration.

While other minimally rigid representations of a *P*-cluster exist, we adopt this construction for ease of implementation. Any graph satisfying Laman's condition is equivalent in terms of its effect on the remaining structure during decomposition.

This equivalence is further leveraged in Sec. 2.8, where we deliberately rearrange intra-cluster edges to use an alternative representation. Since all such forms are (2, 3)-tight, they are interchangeable from the perspective of structural analysis.

2.4 Source of Edges. We have created a set of matrices to uniformly describe the kinematic components that appear in the planar linkage system. Their definitions are presented in the Appendix, as this increases the reading load, but it is not directly related to the pebble game.

In our framework, the edges are derived from these matrices and are stored in two bins: *EB* for non-dynamic edges and *ED* for dynamic ones. Edges in *EB* participate in both the reduction and decomposition phases, while those in *ED* are used only during decomposition, since their actuator-dependent length changes violate rigidity assumptions and cannot be merged into a rigid body. Figure 2 illustrates the edge adding process. In particular, Fig. 2(*d*) shows $v_2\vec{v}_1$ as a static edge and $v_2\vec{v}_3$ as dynamic.

2.5 Graph Construction in Reduction. Algorithm 1 outlines the graph construction process with independence testing during the reduction phase. Each vertex is initialized with two pebbles, and the directed graph starts without any edges, following Rules 1 and 2 from Sec. 2.2. The algorithm then performs edge insertion according to Rules 3 and 4, and reallocates pebbles as needed following Rule 5. The result is a directed, minimal graph *G* constructed from the edge bin *EB*.

Algorithm 1: Graph construction for reduction phase

Input: Edge bin *EB* (non-dynamic edges)
Output: Directed graph $G = (V, E)$
Initialize vertex set V induced by *EB*;
Initialize edge set $E \leftarrow \varnothing$;
Assign two pebbles to each vertex in V;
foreach *edge* $e = v_1v_2 \in EB$ **do**
  if v_1 *and* v_2 *each have at least 2 free pebbles* **then**
    Insert directed edge $v_1\vec{v}_2$ into E;
    Remove one pebble from v_1;
  else
    Reallocate pebbles in $G = (V, E)$ to maximize free pebbles at v_1 and v_2;
    if v_1 *and* v_2 *each have at least 2 free pebbles* **then**
      Insert directed edge $v_1\vec{v}_2$ into E;
      Remove one pebble from v_1;
    else
      // Edge is considered redundant and skipped
return $G = (V, E)$;

2.6 Reduction With Rigidity Test. The second step of the reduction phase aims to minimize the number of clusters by identifying the maximal *P*-cluster for each *PP*-edge in the classical setting. This is done by progressively expanding each cluster from a single edge—the smallest possible cluster—to include as many vertices following Laman's rigidity conditions. This expansion process is referred to as a "merge" in the work of Fudos and Hoffmann [18].

The reduction process can begin with any edge in the graph. The two endpoints of the edge can always be made to collectively hold three free pebbles through a redistribution process, since any subgraph induced by *G* satisfies Laman's theorem. Once these three pebbles are removed—i.e., pinned and no longer available for redistribution—any adjacent vertex is considered rigidly connected to

the initial pair of vertices if no free pebble can be retrieved through any valid redistribution path. This condition holds because the existence of four or more free pebbles in such a configuration would violate Laman's cardinality condition, implying the subgraph lacks at least one necessary edge. This check is formalized as a *rigidity test* (see Algorithm 2).

Algorithm 2: Rigidity test for one edge and a vertex (subprocess of Algorithm 3)

Input: Directed Graph $G = (V; E)$ from Algorithm 3, vertices v_1, v_2 from edge $v_1\vec{v}_2$, and testing vertex v_3
Output: True / False
// Each vertex is assigned two free pebbles initially in Algorithm 1, but their assignment can be different in G from Algorithm 3.
The distribution of free pebbles can be either (2, 1) or (1, 2) for v_1 and v_2.
Find 1 through BFS, and pin this pebble;
Find 1 through BFS, and pin this pebble;
Find 1 free pebble for v_1, if not possible then find 1 free pebble for v_2;
Pin the three free pebbles on the two vertices from the above step;
If any of the above steps is not possible, raise Error; // The graph violates Laman's subset condition
Search 1 free pebble for v_3;
Unpin all pebbles in v_1 and v_2; // Test completed
if *a free pebble is found for* v_3 **then**
  return False; // Flexible
else
  return True; // v_1, v_2, and v_3 are rigidly connected

Algorithm 3: Reduction via rigidity condition

Input: Directed Graph $G = (V; E)$ from Algorithm 1
Output: Cluster Set *CS*
Initialize an empty set *CS*;
Initialize an empty set *ESS*;
Initialize an edge bin *EB* by copying all directed edges in E and making them undirected;
while *EB is not empty* **do**
  Extract vertices v_1 and v_2 from edge v_1v_2 from *EB*;
  Create a new empty cluster *NC* and add v_1, v_2 to it;
  Create a new empty edge set *NES* and add v_1v_2 to it;
  Initialize pivot set $PV \leftarrow \{v_1, v_2\}$;
  Initialize an empty new pivot set $NPV \leftarrow \emptyset$;
  while *PV is not empty* **do**
    foreach *vertex* $v \in PV$ **do**
      foreach *adjacent vertex* v_3 of v **do**
        if v_3v *is not in EB* **then**
          Continue; // Skip the tested edges
        Perform the rigidity test for v_1, v_2, and v_3 with Directed Graph G;
        if v_3 *is not in NC and satisfies rigidity condition against* v_1 *and* v_2 **then**
          Add v_3 to *NC*;
          Add v_3 to *NPV*;
          foreach *adjacent vertex* v_4 of v_3 **do**
            if $v_4 \in NC$ **then**
              Remove edge v_3v_4 from *EB*;
              Add edge v_3v_4 to *NES*;
    if *NPV is empty* **then**
      Add *NC* to the cluster set *CS*;
      Add *NES* to the edge set *ESS*;
      break // Exit inner while loop because expansion is no longer possible
    otherwise, $PV \leftarrow NPV$; // Continue cluster expansion
  Set the cluster with at least two ground *P*-vertices as the ground cluster C_g in *CS*;
  Apply *rules in Sec. 3* to update *CS*;
  return *CS*;

Algorithm 4: Edge-cluster dictionary creation

Input: Dynamic Edge Set *ED* and Cluster Set *CS* from Algorithm 3
Output: Edge Cluster Dictionary *ECD*
Initialize an empty Edge Cluster Dictionary *ECD*;
foreach *Cluster* $C_j \in CS \cup ED$ **do**
  For every two vertices v_1 and v_2 ($v_1 \neq v_2$) in C_j, assign key (v_1, v_2) with value C_j;
return *ECD*;

The rigidity test is applied iteratively to expand each cluster. Specifically, for two vertices v_1 and v_2 connected by the directed edge $v_1\vec{v}_2$, we first evaluate all adjacent vertices (e.g., v_3) that are connected to either v_1 or v_2, regardless of edge direction. If v_3 is determined to be rigid with respect to v_1 and v_2, the search is expanded from v_3 to additional untested vertices. Once a cluster is fully maximized, the edges induced by that cluster are removed from further consideration.

The complete clustering procedure is described in Algorithm 3. Here, *CS* denotes the set of discovered clusters, and *ESS* is the corresponding set of edge sets (hence the double "S").

When the clustering procedure concludes, the algorithm designates one cluster as the ground cluster—a cluster whose *P*-vertices remain fixed throughout the kinematic simulation. In the point-based model, at least two such vertices are fixed by construction and belong to the same cluster, enabling unambiguous identification of the ground cluster. This cluster serves as the reference frame for both the Assur graph decomposition and subsequent numerical computations. During the reduction phase, the ground cluster may expand as additional *P*-vertices are merged into it.

Furthermore, it is important to note that the resulting clusters after the outermost **while** loop are not necessarily *P*-clusters, as this process may involve the introduction of conceptual edges—particularly when the constraint graph includes a line or angle–angle constraints, in the non-classic case. In such cases, additional rule-based processing is required, and this goes after the identification of ground clusters. However, for graphs composed exclusively of *P*-vertices and distance constraints, all identified clusters are *P*-clusters, and the step labeled "Apply rules in Sec. 3" can be omitted entirely.

2.7 Creation and Use of Edge-Cluster Dictionary.

With the cluster set *CS* obtained from Algorithm 3, we construct an edge-cluster dictionary *ECD* that maps all possible intra-cluster edges. Specifically, every pair of *P*-vertices within a *P*-cluster corresponds to a potential distance constraint. Thus, a *P*-cluster containing n vertices implies up to $n(n-1)/2$ such constraints. However, not all of these are represented explicitly as edges in the directed graph. Otherwise, the graph would be over-constrained and no longer minimal, rendering the decomposition algorithm adapted from Ref. [20] infeasible. Nevertheless, these implicit constraints remain informative and can be utilized to reduce computational costs. See Algorithm 4 for the creation of the edge-cluster dictionary *ECD*, which takes in the vertex pair as input and returns the cluster it belongs to as output.

Furthermore, it is important to note that the clusters in *CS* are not necessarily *P*-clusters; They may also contain the dynamic edges from the actuators introduced by *ED* and the conceptual edges introduced by line or angle–angle relationships. In such cases, these edges form their own clusters of two vertices.

2.8 Rigid Structure and Its Skeleton.

Inspired by the *skeleton* concept by Ait-Aoudia and Foufou [19], we formally define the skeleton of a rigid structure and the related concepts in our work as follows:

(1) Each *cluster* is representable by a *block*, regardless of which minimal representation is chosen.

(2) Each *skeletal vertex* appears in at least two different clusters in the *rigid structure*.
(3) Each *skeletal edge* appears in between the *skeletal vertices* of the same cluster.
(4) In a particular minimal representation of a cluster in the rigid structure, the subgraph induced by all the *skeletal edges* within that cluster is (2, 3)-tight. The remaining vertices in the cluster can be rigidly represented using Henneberg I moves by connecting each to two skeletal vertices within the same cluster.
(5) The *skeleton* is the (2, 3)-tight graph formed by the union of all skeletal vertices and skeletal edges across the clusters in the rigid structure.

We should note that the skeleton is sometimes found by dynamically changing the (2, 3)-tight graph within one cluster, so that the least number of edges is used in each solving step. The detailed process of extracting the skeleton is presented in Algorithm 6 in linkage decomposition. Furthermore, an example is demonstrated in Fig. 6.

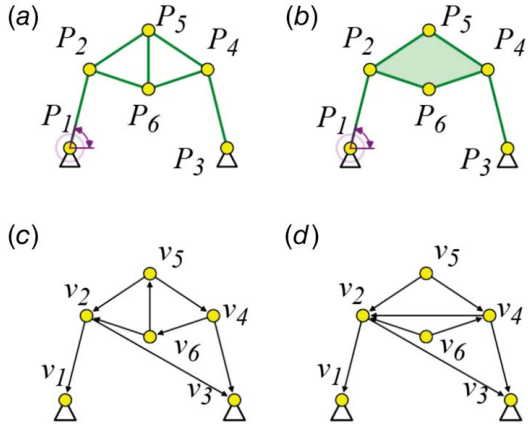

Fig. 6 (*a*) Initial structure, (*b*) reduced structure, (*c*) possible pinned graph 1, and (*d*) possible pinned graph 2

2.9 Assur Graph Decomposition. The decomposition is the pivotal part of the work. During this process, the graph is reconstructed and decomposed, and the ordered solving steps are produced. This part is inspired by Ref. [20], but we add the use of skeleton edges in our work to further decompose the numerical system as much as possible.

In the decomposition phase, the vertices are split into two categories: pinned and unpinned. In terms of the pebble game framework, the pinned vertices have two pebbles, just like the unpinned vertices, but all their pebbles are reclaimed by themselves and are made unavailable after edge addition is complete. This ensures the independence of each edge before we move to the next stage. We define a pinned graph by $G(I, P; E)$, where I is the set of inner vertices, also known as the moving vertices, and P is the set of pinned vertices, also known as fixed vertices. E is the set of edges, where each edge has at least one endpoint in I. The edge that has one inner vertex and one pinned vertex is termed *outer edge* in this work, and others are termed *inner edges*.

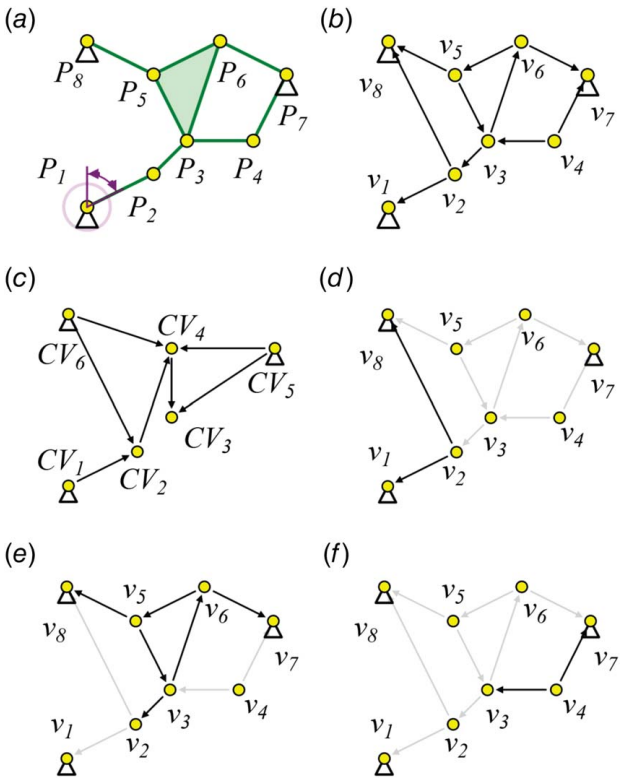

Fig. 5 (*a*) and (*b*) *P*-vertex-only linkage mechanism and its graph representation, (c) condensed graph, and (*d*)–(*f*) consecutive Assur graphs in the mechanism

We can decompose the constraint graph using the pinned Laman Theorem [20]. Specifically, a pinned graph $G = G(I, P; E)$ is statically determinate if and only if $|E| = 2|I|$ and for all subgraphs G(I′, P′; E′) the following conditions hold:

(1) $|E'| \leq 2|I'|$ if $|P'| \geq 2$,
(2) $|E'| \leq 2|I'| - 1$ if $|P'| = 1$, and
(3) $|E'| \leq 2|I'| - 3$ if $|P'| = \varnothing$

These three subset conditions of the Pinned Laman Theorem are automatically satisfied when all edges in the graph are independent. Therefore, the core condition we verify is $|E| = 2|I|$, where the set P contains the pinned vertices, and I contains the unfixed vertices to be fixed. Each vertex in I can be computed from the known positions in P using the constraints defined by the edges in E. Examples of such pinned, statically determinate graphs are shown in Figs. 2(*d*), 5(*b*), 6(*c*), and 6(*d*).

Given the cluster set *CS* obtained from the reduction phase, we reconstruct a constraint graph using these non-reducible clusters and generate edges via Henneberg I moves. In addition to these cluster-based edges, the decomposition phase includes all dynamic edges in *ED*. After all edges are added, we remove the edges from the ground cluster to ensure compatibility with the pinned graph model. See Algorithm 5 for details.

Algorithm 5: Pinned graph construction

Input: Set of Vertices V, Dynamic Edge Set *ED*, and Cluster Set *CS* from Algorithm 3
Output: Pinned Directed Graph $G = G(I, P; E)$
// Step 1: Represent a moving cluster with Henneberg I moves
Initialize an empty edge bin *EB*;
foreach $C_j \in CS$ **do**
  Pick the first two vertices v_1 and v_2 from the cluster;
  Add edge v_1v_2 to *EB*;
  For every remaining vertex v_3 in C_j, Add v_1v_3 and v_2v_3 to *EB*;
//Step 2: Assign edges in the pinned graph
Assign 2 pebbles to each vertex in V;
Copy the ground cluster C_g to set P;
Copy remaining vertices $V \setminus P$ to set I;
Initialize an empty edge set $E \leftarrow \varnothing$;
Create a directed graph $G = G(I, P; E)$;
foreach *edge* $e = v_1v_2 \in EB \cup ED$ **do**
  if v_1 *and* v_2 *each have 2 pebbles* **then**
    Remove one free pebble from v_1;
    Insert edge $v_1\vec{v}_2$ to E;

else
Maximize pebbles for v_1, v_2;
if *v_1 and v_2 each have 2 pebbles* **then**
Remove one free pebble from v_1;
Insert edge $v_1\vec{v}_2$ to E;
else, terminate; // Over-determinate
// Step 3: Pin all ground vertices
Remove all edges generated from C_g. In addition, reclaim the pebbles from these edges for the ground vertices;
foreach *vertex $v \in C_g$* **do**
Reassign pebbles in G to maximize pebbles for v;
if *v has two free pebbles* **do**
Pin two pebbles on v;
else, **terminate**; // over-determinate
return $G(I, P; E)$;

Next, the resulting pinned graph can be decomposed into *Assur graphs*—minimal, rigid subgraphs with zero mobility that do not contain a smaller substructure of the same property [32]. Notably, this decomposition minimizes the number of variables to solve per numerical subsystem, thereby reducing solving time in the subsequent numerical analysis [15].

However, several concepts from graph theory [33] are necessary to understand the decomposition process

(1) A directed graph G is said to be *strongly connected* if, for any two vertices v_i and v_j, there exists a directed path from v_i to v_j. An SCC is a maximal set of vertices in which every vertex is reachable from every other vertex in the set. SCCs are disjoint, and the smallest possible SCC consists of a single vertex.
(2) A *cycle* in a graph G is a sequence of edges that starts and ends at the same vertex without repeating any edges. A *directed cycle* is a cycle where all edges follow the same orientation. A directed graph is *acyclic* if it contains no directed cycles.
(3) SCCs can be efficiently computed using Tarjan's algorithm [34], which we adopt in our implementation due to its linear time complexity of $O(|E|)$. For each SCC, we classify edges as either *inner flows*—edges between vertices within the SCC—or *outgoing flows*—edges from a vertex in the SCC to a vertex outside the SCC. We use the term "flow" to distinguish this from the notions of "inner" and "outer" edges in the Pinned Laman Theorem, because a pinned and statically determinate graph is not necessarily an Assur graph.
(4) Contracting each SCC into a single vertex (by removing inner flows and replacing each group of vertices with a representative vertex) results in a directed acyclic graph. This transformation is useful for establishing a solvable order of subsystems.

Because ground vertices never hold free pebbles, they always serve as endpoints in directed edges. Each ground vertex forms its own singleton SCC with no inner flows. Additionally, these SCCs correspond to the starting points of the constraint-solving process, and their locations are known and unchanged throughout the simulation. Furthermore, an Assur graph is defined by the set of edges in the inner flows and outgoing flows of a given SCC, along with the vertices induced by these edges [20]. Subsequently, we can sort these Assur graphs and find the underlying numerical solving path. See Algorithm 6 for the detailed decomposition process, and Fig. 5 for the decomposition example of a P-vertex-only linkage.

Algorithm 6 is lengthy and complex, so we break it down into several steps and explain the variables within. The first step is finding the Assur graphs. This can be achieved by using Tarjan's algorithm [34] to find the SCCs and their corresponding inner flow and outgoing edges. For a set of Assur graphs *AGS*, we denote the ith Assur graph as AG_i, a directed subgraph consisting of (1) the vertex set SCC_i, (2) the inner flow edges $SCCIE_i$, which are contracted during the condensation process, and (3) the outgoing edges $SCCOE_i$, which originate from vertices in SCC_i and point to vertices in other SCCs. The corresponding vertex of SCC_i in the condensed graph is denoted CV_i, and the associated condensed edge set is denoted CE_i. When the pinned directed graph is condensed to a condensed graph CG, we have its condensed edges CE and condensed vertices CV. In particular, a condensed vertex CV_i corresponds to the set of vertices SCC_i, and they are the inner vertices for the Assur graph AG_i. Additionally, when we exclude the vertices in SCC_i from the set of vertices induced by the outgoing edges of $SCCOE_i$, we have the set of pinned vertices for the Assur graph AG_i. Figures 5(*b*) and 5(*c*) illustrate the pinned graph and its condensed form, respectively. Furthermore, the pinned graph includes three Assur graphs shown in Figs. 5(*d*)–5(*f*).

The second step is sorting these Assur graphs. As demonstrated in Fig. 5, some Assur graph has their pinned vertices on the inner vertices of other Assur graphs, so the former is dependent on the latter. Sorting this dependency is termed *topological sort*, and we use Kahn's algorithm [35] to sort the vertices in the condensed graph so that we know the order and dependency of each condensed vertex and correspondingly the order and dependency of each Assur graph. Notably, Kahn's algorithm sorts the vertices based on the exact opposite direction of the edges in the condensed graph CG, so we need to reverse all the edges in CE to get the sort order.

The third step is to analyze the Assur graphs in sorted order and mark computational methods to solve the underlying numerical system. In the algorithm, the Assur graph is reflected by its SCC. Furthermore, the first SCCs are not necessarily Assur graphs because each of these SCCs includes only one ground vertex and has an empty *SCCOE* and *SCCIE*. When these SCCs are processed, the algorithm starts finding the skeleton in each Assur graph AG_i. To achieve this, it first finds the unfixed clusters in the rigid structure using *ECD*, and stores them in a list CL_i. Then, the algorithm performs operations on each of these clusters to find the skeletal vertices. Next, it computes the skeleton edges for the unfixed clusters. In particular, if the cluster has more than two skeleton vertices, these vertices are connected using the Henneberg I moves. Specifically, if the cluster has non-skeletal vertices, this cluster is a P-cluster, and each non-skeletal vertex can be rigidly and minimally connected with two selected skeletal P-vertices within the same cluster. The location of these non-skeletal P-vertices is solved after the numerical system corresponding to the skeletal edges is solved. Finally, after analyzing all Assur graphs, the algorithm determines the complete solving path *SP*.

A P-cluster-only mechanism that shows the extraction of the skeleton with the use of *ECD* is shown in Fig. 6. In this example, the P-cluster of four vertices, v_2, v_4, v_5, and v_6 is represented with two different induced graphs shown in Figs. 6(*c*) and 6(*d*). We can see that starting from the ground PP-edge v_1v_3, we can use Hennberg I moves to create the remaining part of this graph according to Fig. 6(*d*). Specifically, the construction order is v_2, v_4, then v_5 and v_6. In numerical solving, this means we can find the location of these four P-vertices in the construction order using arc intersection. However, this is not possible for the graph shown in Fig. 6(*c*), starting from v_4. Instead, when we finished solving the location of v_2, the location of the remaining unfixed P-vertices v_4, v_5, and v_6 will be solved using six distance constraints corresponding to the six PP-edges, namely v_2v_5, v_2v_6, v_3v_4, v_4v_5, v_4v_6, and v_5v_6, and the location of two depending P-vertices v_2 and v_3, as indicated by the graph in Fig. 6(*c*). In general, numerically solving $2|V|$ unknown variables incurs a solving time of $O((2|V|)^3)$ [19], which is highly inefficient, compared to the computing process indicated by the graph in Fig. 6(*d*).

Algorithm 6 presents three solving methods: "fixed," "arc intersection," and "optimization." The first two are self-explanatory. For the optimization-based approach, we employ the Levenberg–Marquardt algorithm [36]. It is important to note, however, that

while the solving method may be labeled as "optimization," the pebble game framework itself only outputs the system of constraint equations; it does not prescribe a specific numerical method. In practice, various alternative solvers—such as other numerical optimization techniques [37,38] or symbolic approaches [39,40]—may also be used.

Algorithm 6: Decomposition of constraint system

Input: Cluster Sets (*CS*) from Algorithm 5
Output: Solving Path *SP*
if $|E| \neq 2|I|$ **then**
  Raise Error; // Generically flexible graph
Initialize empty solving path *SP*;
Apply **Tarjan's Algorithm** on *G* to find the set of Assur graphs *AGS*;
foreach AG_i *in AGS* **do**
  Extract vertex set SCC_i, innerflow edges $SCCIE_i$, and outgoing edges $SCCOE_i$;
Condense the *SCCs* into condensed vertices *CV* and find condensed directed edges *CE* with *SCCOEs*;
Reverse all edges in *CE* to represent solving order;
Construct a directed **Condensed Graph** $CG = G(CV, CE)$;
Apply **Kahn's Algorithm** to *CG* for topological sorting;
foreach SCC_i *in sorted order* **do**
  if SCC_i *corresponds to a ground vertex* **then**
    Mark the vertex as fixed, set "fixed" as solving method, and add to *SP*;
    continue // Skip to next SCC
  Identify clusters for each edge in $SCCIE_i \cup SCCOE_i$ using *ECD*, and store unfixed cluster C_j in list CL_i;
  Initialize empty skeleton edge set *SES*;
  Initialize empty solving subpath *SSP*;
  Count the number of appearances for each vertex in the cluster of *CL*, and store the vertices as skeletal vertices V_j for each cluster C_j when the number of appearances is equal to or greater than 2;
  foreach C_j *in* CL_i **do**
    // Compute skeletal edges
    Compute SE_j from V_j using Henneberg 1 moves and add to *SES*;
    // Assign computing method for non-skeletal *P*-vertices
    Pick two vertices v_1 and v_2 from V_j;
    foreach *remaining vertex* v_3 *in cluster* $C_j \setminus V_j$ **do**
      Mark v_3 as fixed and set ``arc intersection'' as solving method, using *PP*-edges v_1v_3 and v_2v_3, and location of v_1 and v_2;
      Add solving step to *SSP*;
  if *SES includes exactly two PP-edges* **then**
    Mark the unfixed vertices in *SES* as fixed and set "arc intersection" as solving method, using edges in *SES*, and fixed vertices in *SES*;
  else, mark the unfixed vertices in *SES* as fixed, and set "optimization" as solving method, using edges in *SES*, and fixed vertices in *SES*;
  if *new set of fixed vertices includes actuation loop(s) that was not previously included* **then**
    Add these constraint equations to the last solving step in *SP*;
  Add *SSP* to the end of *SP*;
return *SP*;

2.10 Time Complexity. The time complexity of each step in our framework is analyzed below. Let $|V|$ denote the number of vertices, $|E|$ the number of edges, $|EB|$ the number of non-dynamic edges in the initial edge bin, and $|ED|$ the number of dynamic edges (i.e., actuators).

2.10.1 Henneberg I-Based Edge Generation. For each cluster, the Henneberg I move-based construction produces at most $2|V|$ edges in a minimally rigid graph, yielding a complexity of $O(|V|)$.

2.10.2 Pebble Reallocation. Pebble reallocation via path reversal has a worst-case cost of $O(|V| + |E|)$, which simplifies to $O(|V|)$ in sparse graphs. Both depth-first search [34] and BFS [30] can be used, each running in linear time with respect to $|V|$.

2.10.3 Independence and Rigidity Tests. The independence test checks whether a new edge violates Laman's condition by locating four free pebbles. In the worst case, this takes $O(4|V|) = O(|V|)$. The rigidity test (Algorithm 2) has similar complexity, except that the final pebble must lie on a different vertex, which does not affect the asymptotic bound. As a result, the graph construction during reduction (Algorithm 1) takes $O(|V\|EB|)$, and the graph construction during decomposition (Algorithm 5) takes $O(|V\|ED| + |V|^2)$. Given that $|EB| = O(|V|)$ in common cases (i.e., not many repeated edges on the same two vertices) and $|ED| = O(1)$ in typical actuator-driven systems, both processes reduce to $O(|V|^2)$.

2.10.4 Edge-Cluster Dictionary Creation. The edge-cluster dictionary (Algorithm 4) is built from a minimal graph with $O(|V|)$ vertices. There are at most $O(|V|^2)$ vertex pairs, and edges from *ED* are added directly. Thus, the overall cost is $O(|V|^2 + |ED|) = O(|V|^2)$.

2.10.5 Clustering Procedure. The clustering process (Algorithm 4) also has complexity $O(|V|^2)$, despite involving three nested loops. This bound is justified as follows:

(1) The outer loop is $O(|E|)$, effectively $O(|V|)$, because the algorithm tests each edge in *EB* which satisfies Laman's subset condition in this stage.
(2) The inner loop is $O(|V|)$, because the rigidity test is performed, and other operations are either constant or sublinear in $|V|$.
(3) The input graph is sparse with $|E| < 2|V|$, so the average degree of a vertex is $2|E|/|V| < 4$.
(4) Each edge is tested against its adjacent vertices. For an edge $e = v_iv_j$, the number of rigidity tests is bounded by

$$|E| \times \left(\frac{2|E|}{|V|} \times 2 - 2 \right)$$

which evaluates to fewer than six tests per edge, on average.
(5) Therefore, the total time complexity is $O(|E|) \times O(6) \times O(|V|) = O(|V|^2)$.

2.10.6 Decomposition. In the decomposition phase (Algorithm 6), Assur graph detection takes $O(|E|)$ [34], or $O(|V|)$ in sparse graphs. Sorting the condensed graph takes $O(|CV| + |CE|)$ [35]. These two values are both $O(|V|)$. In the remaining part of this algorithm, each edge can be visited at most once because they turn from unfixed to fixed in their respective flags, resulting in an $O(|E|)$ time cost, which is again effectively $O(|V|)$.

2.10.7 Summary. Both the reduction and decomposition phases have an overall time complexity of $O(|V|^2)$. This bound holds under the assumption that the constraint graph is sparse and the number of dynamic edges $|ED|$ remains constant or sublinear in $|V|$.

3 Additional Rules

With the point-based model in our earlier work [17], the analysis of planar geared linkages can be done using the same framework, with some additional rules. Therefore, this section provides additional rules to deal with the non-distance constraints in the pebble game framework. Furthermore, the two types of non-distance point-based models are illustrated in Fig. 8 and in Fig. 11, respectively.

3.1 Line Constraint Semantics. Our work shares the core ideas with the work from Jackson and Owen [23]. While we start from a different root [15], doing a comparison can be necessary. Specifically, their work treats the points and the lines as individual

vertices (*P*-vertices and *L*-vertices), while we treat each line as an element defined by a set of points.

Specifically, the graph semantics from Jackson and Owen's work describe a geometric constraint system with the following rules:

(1) *PP*-edges: a *P*-vertex connects with another *P*-vertex, i.e., the two points have a fixed distance in between.
(2) *PL*-edges: a *P*-vertex connects with an *L*-vertex, i.e., the point has a fixed distance from the line.
(3) *LL* edges: a line vertex connects with a line vertex, i.e., the two lines have a fixed angle.

This classic setup extends the application of the Laman theorem, in which each edge removes one DOF, regardless of its type. Notably, both this work and that of Jackson and Owen adopt the same definition for the *PP*-edge. However, additional semantic rules are necessary for *PL*- and *LL*-edges to ensure that the resulting graph is rigid and solvable. Without such rules, symbolic computation (e.g., algebraic methods) may be required as a fallback to handle edge cases [23]. For example, consider a graph containing three *LL*-edges and three line vertices (*A*, *B*, and *C*) that form a triangle. Suppose we also have three pairs of *PL*-edges, each connecting a *P*-vertex to two of the three *L*-vertices. This graph satisfies Laman's count condition but may not correspond to a solvable system without further constraints. See Fig. 7 for an illustration.

In contrast, our point-based model explicitly forbids the use of *LL*-edges. Instead, a line constraint is represented using a set of three *PL*-edges, each with its corresponding point-line distance set to zero. In this formulation, *a PL-edge explicitly suggests that a point lies on a line*.

Furthermore, we term the *L*-vertex as *conceptual vertex*, or the vertex that does not correspond to a specific point. Correspondingly, we call a *PL*-edge *conceptual edge*-an edge that has at least one conceptual vertex at its two endpoints.

Notably, the combination of three *PL*-edges for a line constraint stems from the use of *far joint*. See Fig. 8, where the behavior of an RRRP mechanism is computed using approximation. Specifically, we use three long *PP*-edges, *CF*, *DF*, and *EF*, to represent the original constraint system. Notably, the arc for point *D* is "flat" when these edges are sufficiently long [15]. However, the three long *PP*-edges can be replaced by three *PL*-edges to faithfully represent the original constraint system, which jointly correspond to the following constraint equation:

$$(x_C - x_D)(y_E - y_D) - (x_E - x_C)(y_C - y_D) = 0 \tag{4}$$

which is the scalar cross-product of vectors $\vec{CD}$ and $\vec{CE}$, and equals to zero if and only if the three points *C*, *D*, and *E* are collinear.

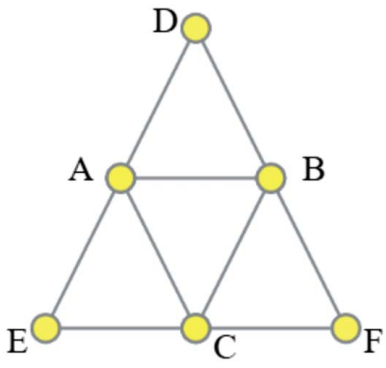

Fig. 7 In this constraint graph, *A*, *B*, and *C* are *L*-vertices; *D*, *E*, and *F* are *P*-vertices; The *LL*-edges in the graph are *AB*, *BC*, *AC*, and the *PL*-edges are *AE*, *CE*, *CF*, *BF*, *BD*, and *AD*. This graph is rigid while the *P*-vertices in its corresponding geometric structure are undetermined.

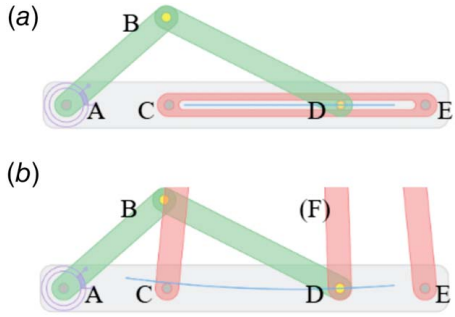

Fig. 8 (*a*) An RRRP mechanism and (*b*) its approximation drawn in MotionGen [41]. We reduce the length of three edges to emphasize the arc property of the point path for *D*.

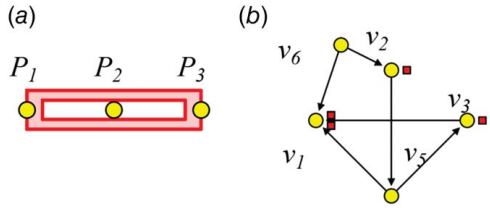

Fig. 9 (*a*) Slot and (*b*) its interpretation

Last but not least, we add two *PP*-edges to the graph to limit the movable range of the *P*-vertex in the slot (see Fig. 9). The two *PP*-edges are v_6v_1 and v_6v_2, whose lengths can be computed according to our old work [15].

3.2 Reduction Rules for Line Constraints. There are three additional reduction rules in total for the slot constraints, stated as follows:

- Rule 1: The two *P*-vertices on two *PL*-edges for a same *L*-vertex defines the line. So, if two *L*-vertices connect to two same *P*-vertices, then these two *L*-vertices are merged into one *L*-vertex, because they correspond to the same line.
- Rule 2: If more than two *P*-vertices connect to the same *L*-vertex and these *P*-vertices are in the same cluster we found during the reduction, then only two *P*-vertices within these *P*-vertices are connected to the *L*-vertex, because the positions of the remaining *P*-vertices can be represented with *PP*-edges for this cluster. As a result, the *PL*-edges and the *L*-vertex are not added into any *P*-cluster, and every *L*-vertex connects at most twice to a specific *P*-cluster.
- Rule 3: An *L*-vertex and all its *PL*-edges are removable if they are rigid to only one *P*-cluster, because all the corresponding line constraints can be represented using distance constraints.

Additionally, the reduction of *PL*-edge is a two-phase process. The first reduction occurs at the slot level, where Rule 1 is applied; the second reduction happens during *P*-cluster finding, where Rules 2 and 3 are applied. See Figs. 10(*a*)–10(*c*) for the first phase, and Figs. 10(*d*) and 10(*e*) for the second phase.

In the first phase, the algorithm checks for each pair of *L*-vertices. If they share the same two *P*-vertices, which are v_2 and v_3 in Fig. 10(*b*), then the two *L*-vertices v_5 and v_7 correspond to the same line defined by these two *P*-vertices. We can merge the two vertices into one *L*-vertex into one.

The second phase of *PL*-edge reduction happens when all *PL*-edges and *PP*-edges are added to the directed graph. Notably, the difference between the kinematic structures in Figs. 10(*a*) and 10(*d*) is that an additional *PP*-edge, P_2P_3 is in Fig. 10(*d*). Correspondingly, after the first phase *PL*-edge reduction, we can see the resultant graph shown in Fig. 10(*e*). This resultant graph has

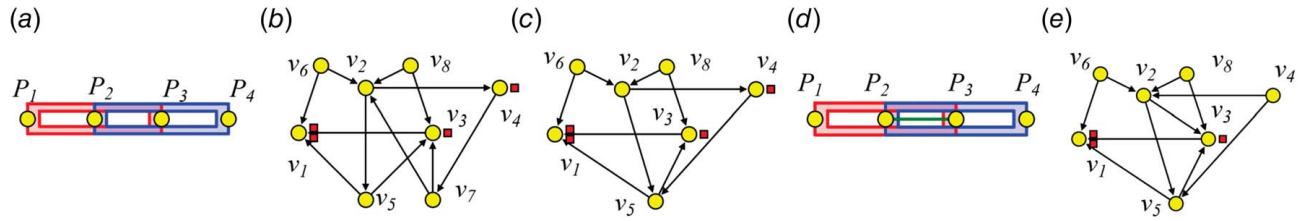

Fig. 10 (*a*) Double slot, (*b*) its original constraint graph, and (*c*) its reduced graph; (*d*) double slot with a *P*-cluster of {P_2, P_3} and (*e*) its reduced graph after reduction rule 1

only three pebbles left. In other words, this is a *P*-cluster of the four *P*-vertices, v_1, v_2, v_3, and v_4. For the efficiency of computation, we can remove the two *P*-vertices v_6 and v_8 in the graph, because they serve as the moving range limiter for the *P*-vertex in the slot. Furthermore, we can remove the *PL*-edges on this graph until only two *PL*-edges are left, because if any additional *P*-vertex exists in the slot, it can find the line defined by the two *PL*-edges.

3.3 Decomposition Rules for Line Constraints. During the decomposition phase, we have the following rules for the *PL*-edges. Furthermore, these two rules are demonstrated with the example shown in Sec. 4.

- Rule 1: Every three *PL*-edges brings exactly one line constraint equation for the *P*-vertices connecting to the same *L*-vertex.
- Rule 2: When two *P*-vertices that are connected to a particular *L*-vertex are fixed in the previous step, this *L*-vertex is marked as fixed, and these two *P*-vertices are treated as the two bases of the line constraint. In other words, if any other unfixed *P*-vertex is connected to the fixed *L*-vertex in a specific step, then a line constraint is added to the numerical system, where we set point *C* and point *D* as the two bases in Eq. (4).

3.4 Angle–Angle Constraint Semantics. Similar to the line constraint, the point-based model uses a set of conceptual edges to represent an angle–angle constraint. Figure 11 demonstrates how we represent rolling with a set of conceptual edges, which are v_1v_7, v_3v_7, v_5v_8, v_6v_8, and v_7v_8. These five edges jointly correspond to exactly one angle–angle constraint equation, or five expressions if we take the intermediate t-equations into account (see Eq. (5)).

$$\begin{cases} t_1 = det(\mathbf{p}_3 - \mathbf{p}_1, \mathbf{p}_5 - \mathbf{p}_1) = r_1(r_1 + r_2)\sin(\alpha_1) \\ t_2 = dot(\mathbf{p}_3 - \mathbf{p}_1, \mathbf{p}_5 - \mathbf{p}_1) = r_1(r_1 + r_2)\cos(\alpha_1) \\ t_3 = det(\mathbf{p}_1 - \mathbf{p}_3, \mathbf{p}_6 - \mathbf{p}_3) = r_2(r_1 + r_2)\sin(\alpha_2) \\ t_4 = dot(\mathbf{p}_1 - \mathbf{p}_3, \mathbf{p}_6 - \mathbf{p}_3) = r_2(r_1 + r_2)\cos(\alpha_2) \\ \arctan\left(\frac{t_1}{t_2}\right) \cdot r_1 - \arctan\left(\frac{t_3}{t_4}\right) \cdot r_2 = 0, \end{cases} \tag{5}$$

Here r_1 and r_2 are the radii of the two wheels on the left and right, respectively. Then, the definition of *det* and *dot* functions are

$$\begin{cases} det(\mathbf{p}_1, \mathbf{p}_2) = x_1y_2 - x_2y_1 \\ dot(\mathbf{p_1}, \mathbf{p_2}) = x_1x_2 + y_1y_2 \end{cases} \tag{6}$$

In this structure, $\mathbf{p}_5$ and $\mathbf{p}_6$ are the *P*-vertices, which are refreshed for each state, and their initial position in each state is at the intersection point of the two wheels; Next, v_7 is called an *assisting vertex* and v_8 is called a *rolling vertex*. Additionally, a complete wheel-pair constraint includes three distance constraints corresponding to the three *PP*-edges v_1v_5, v_3v_6, and v_1v_3, whose lengths are r_1, r_2, and $(r_1 + r_2)$ in this scene, respectively.

3.5 Reduction Rule for Angle–Angle Constraints. There is one additional reduction rule for the angle–angle constraint: If the rolling vertex v_8, as shown in Fig. 11(*c*), is rigid to any of the four *P*-vertices from the five conceptual edges, then it is also rigid to the remaining three other *P*-vertices. In this case, the angle–angle constraint can be directly removed, and the four points that define a wheel pair are in the same *P*-cluster. If not, then these five conceptual edges are five respective clusters in cluster set *CS* and edge-cluster dictionary *ECD*, respectively.

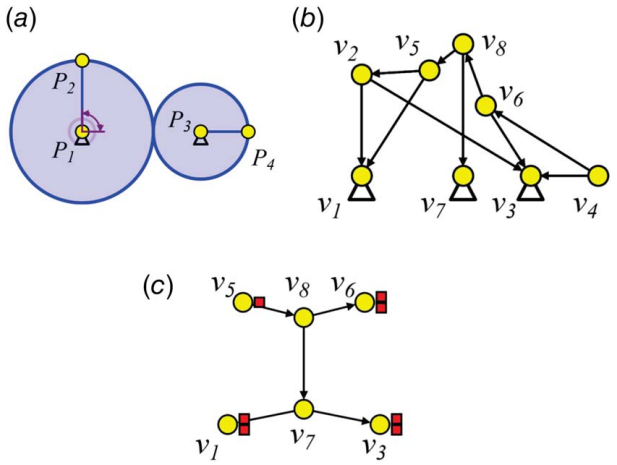

Fig. 11 (*a*) Simple gear train system, (*b*) its constraint graph, and (*c*) its conceptual edges

The reduction rule can be demonstrated by changing the graph in Fig. 11(*b*). If we change the dynamic edge v_2v_3 to static, then this wheel pair is fixed to the ground, because there are no free pebbles left. This also means v_1, v_3, v_5, v_6, and v_8 form a rigid cluster. While all these vertices still satisfy the angle–angle constraint, we can represent them with distance constraints, but computing them becomes unnecessary. So, in this case, v_5, v_6, v_7, and v_8 can be removed from the graph. As a result, this angle–angle constraint is reducible to the distance constraints, and v_1, v_2, v_3, and v_4 together form a *P*-cluster.

3.6 Decomposition Rule for Angle–Angle Constraints. There is only one additional decomposition rule specific to wheel joints. Unlike an *L*-vertex in a slot, which can connect to an arbitrary number of *P*-vertices, a rolling vertex exists exclusively between a particular pair of wheels. This rolling vertex serves as one of the two endpoints for each of the three conceptual edges. When these three edges are added in a specific step—and not all were included previously—an angle–angle constraint equation is introduced into the geometric constraint system.

3.7 Incorporation of Additional Rules. As a result of adding the conceptual vertices and edges, the updated reduction phase will first check and reduce the number of *L*-vertices according to the first reduction rule for line constraints. Next, all the conceptual edges are added, followed by the static *PP*-edges from the initial clusters. After the clustering procedure, i.e., detecting which vertices are rigid with each other, the additional reduction rules are applied to

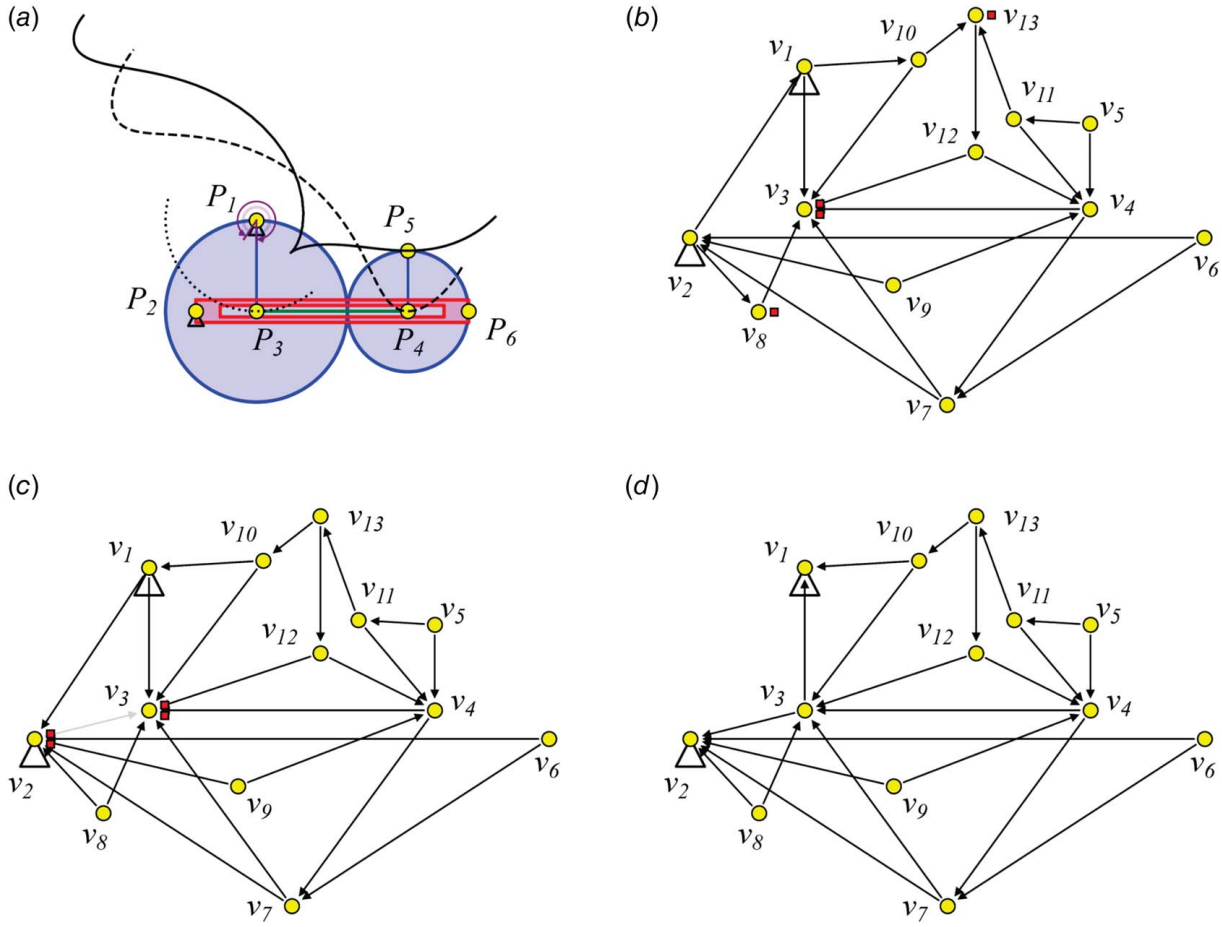

Fig. 12 (*a*) Planar geared linkage, (*b*) constraint graph before the dynamic edge insertion, (c) constraint graph after the dynamic edge insertion, and (*d*) constraint graph after pinning the ground vertices

extract the conceptual edges from larger clusters. Each remaining conceptual edge is exactly one cluster, while each set of mutually rigid P-vertices forms exactly one P-cluster.

Next, the updated decomposition phase is similar to the one in the classic pebble framework, but the additional decomposition rules are applied during solver planning (we set the solver to be an optimization-based method as demonstrated in Algorithm 6), and they are checked for each decomposed Assur graph.

4 Example

We present the analysis process of a geared and slotted planar linkage system shown in Fig. 12(*a*) in two subsections. The first section focuses on the construction of the constraint graph, and the second section focuses on the decomposition of the constraint system.

4.1 Pinned Graph Construction. In this example, the two wheels defined by (P_3, P_1) and (P_4, P_5) make contact externally. Specifically, P_3 and P_4 are the two wheel centers, respectively, while P_1 and P_5 are the circular edge points. The two centers of the two wheels are connected through a PP-edge (P_3, P_4), which is constrained within the slot defined by its endpoints (P_2, P_6). Furthermore, P_1 and P_2 are attached to the ground, and an actuator is placed on P_1, which determines the angle between the lines defined by (P_1, P_2) and (P_1, P_3), respectively.

The list of figures in Fig. 12 shows the key state of the pinned graph construction process. Furthermore, Table 1 provides the property of each vertex shown in this list of figures.

The analysis begins with the reduction phase. In this phase, the static edges and conceptual edges for the kinematic structure in Fig. 12(*a*) are tested following Algorithm 1, and the resultant directed graph is shown in Fig. 12(*b*). To ensure visual clarity, we raise the positions of the vertices for the wheel pair and lower the positions of the vertices for the slots. This resultant graph has four free pebbles left: one on v_8, one on v_{13}, and two on v_3. Notably, the ground edge $v_2\vec{v}_1$ is in the graph, and the dynamic edge between v_2 and v_3 is not added in this phase. Additionally, there are no redundant edges in this kinematic structure.

Next, the graph is reconstructed according to Algorithm 5, and the resultant graph is shown in Fig. 12(*c*). The change can be described as follows:

(1) Before adding the dynamic edge between v_2 and v_3, the static edges are added, and they result in the same graph in the reduction phase as shown in Fig. 12(*b*).

Table 1 Vertex property table

ID	Property	ID	Property
v_1	Ground, Normal	v_8	Normal
v_2	Ground, Normal	v_9	Normal
v_3	Normal	v_{10}	Refresh upon state
v_4	Normal	v_{11}	Refresh upon state
v_5	Normal	v_{12}	Assisting, Conceptual
v_6	Normal	v_{13}	Angle-angle, Conceptual
v_7	Linear, Conceptual		

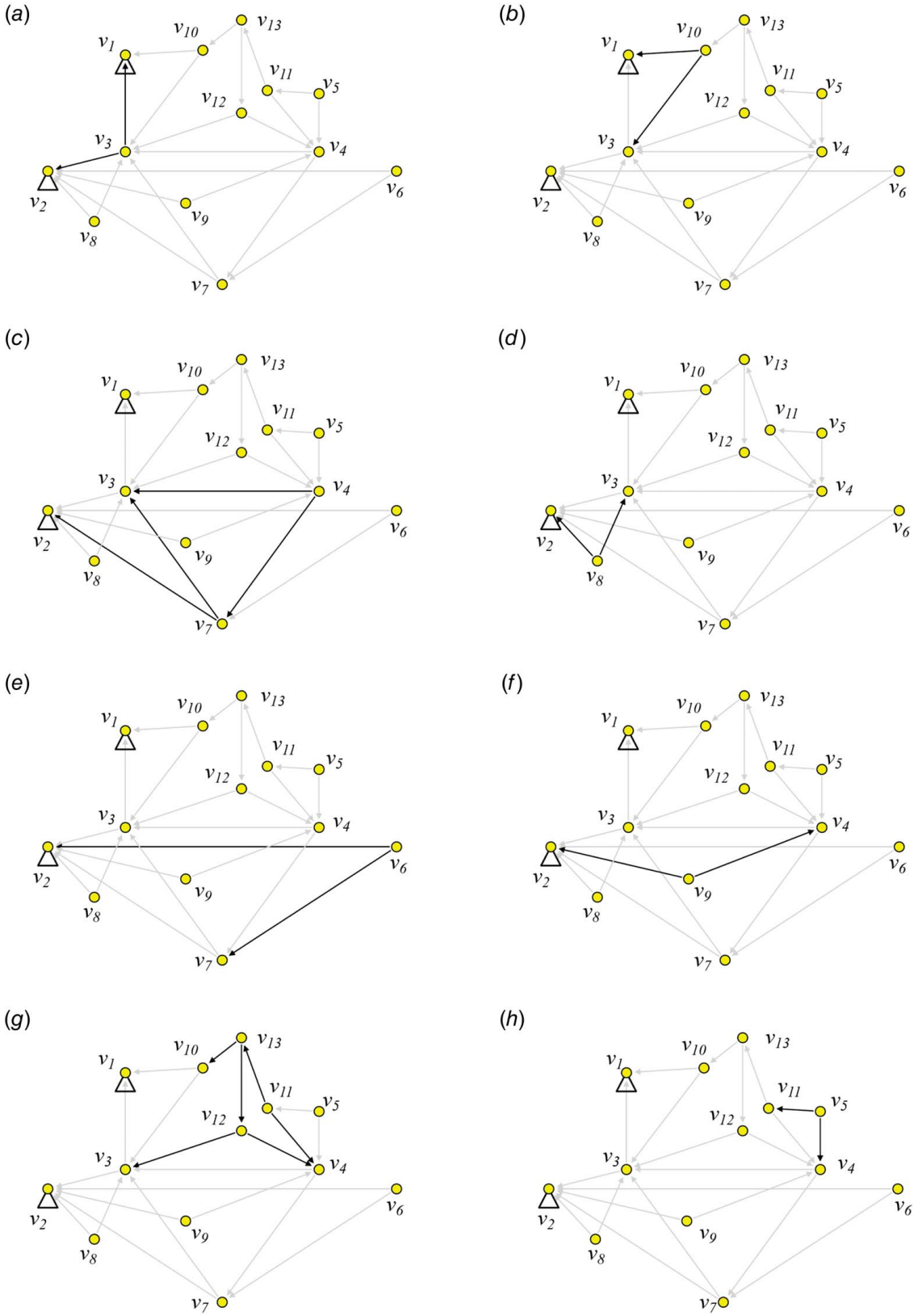

Fig. 13 (*a*)–(*h*): Solving steps for the planar geared linkage system in Fig. 12(*a*). The edges colored in black correspond to the constraint equations for the respective solving steps.

(2) To add the edge between the two vertices, the free pebbles must be rearranged. In this case, we can start from v_2 and find the first free pebble on v_8. As a result, the edge between v_2 and v_8 shifts direction, and v_2 gains its first free pebble.

(3) Next, since v_3 has two free pebbles, we can immediately pin one pebble on v_3.

(4) Then, the second free pebble for v_2 can be found from the path (v_2, v_1, v_3). As a result, two edges, $v_2\vec{v}_1$ and $v_1\vec{v}_3$, change their directions.

(5) Finally, we can find the last free pebble for v_3 through the path (v_3, v_1, v_{10}, v_{13}). Three edges, $v_3\vec{v}_1$, $v_1\vec{v}_{10}$, and $v_{10}\vec{v}_{13}$ change their directions.

With all these four pebbles arranged on the two sides, we can spend one pebble from v_2 on edge $v_2\vec{v}_3$, which is colored in gray in Fig. 12(*c*).

The following steps for constructing the pinned graph include: (1) remove the ground edge, and (2) find two pebbles for all ground vertices. In this case, $v_1\vec{v}_2$ is the only ground edge. When this edge is removed, one free pebble goes to v_1. The second free pebble for v_1 can be found from v_3, so edge $v_3\vec{v}_1$ shifts its direction. Then, we can shift the direction of edge $v_2\vec{v}_3$ to take the last free pebble in v_3 to v_2. As a result, all four free pebbles are on the ground vertices. Pinning them results in the pinned graph shown in Fig. 12(*d*). Since there are no free pebbles left, we can conclude the structure is statically determinate. In other words, we can decompose it into Assur graphs.

4.2 Decomposition. The decomposition follows Algorithm 6 and the additional rules are shown in Sec. 3. Figure 13 shows the ordered solving procedure.

Specifically, the solving procedure begins with marking the ground vertices v_1 and v_2 as fixed. Then, as shown in Fig. 13(*a*), v_3 can be fixed using the two distance constraints corresponding to the two *PP*-edges v_3,$\vec{v}_2$ and $v_3\vec{v}_1$. Furthermore, $v_3\vec{v}_2$ is a dynamic edge, and the newly fixed set of vertices includes the loop (v_1, v_2, v_3), so the constraint system for this step has an angle constraint equation for the actuator. In addition, v_3 is marked as a fixed vertex.

Next, as depicted in Fig. 13(*b*), the location of v_{10} can be solved with the location of v_1 and v_3, and the distance constraint equations corresponding to the two *PP*-edges $v_{10}\vec{v}_1$ and $v_{10}\vec{v}_3$. No rotary actuation loop exists, so the constraint system has only two distance equations.

Then, the next solving step shown in Fig. 13(*c*) includes three conceptual edges, $v_4\vec{v}_7$, $v_7\vec{v}_3$, and $v_7\vec{v}_2$, and a *PP*-edge $v_4\vec{v}_3$. From the previous steps, v_2 and v_3 are fixed, so they become the two bases of the line constraint equation in Eq. (4). As a result, this subsystem comprises a line constraint equation and a distance equation, which are used to determine the location of v_4.

Afterward, as shown in Fig. 13(*d*), the location of the *P*-vertex v_8 is solved with two distance constraints corresponding to the two distance edges $v_8\vec{v}_2$ and $v_8\vec{v}_3$.

Then, the location of v_6 is solved using the two constraints for the two edges $v_6\vec{v}_2$ and $v_6\vec{v}_7$ as shown in Fig. 13(*e*). Upon further checking, $v_6\vec{v}_7$ is a *PL*-edge, and v_7 is a *L*-vertex whose bases are v_2 and v_3. So, the location of v_6 is solved with a distance constraint from edge $v_6\vec{v}_2$ and a line constraint.

Next, the location of v_9 is solved using the distance constraints corresponding to $v_2\vec{v}_9$ and $v_4\vec{v}_9$ according to Fig. 13(*f*).

In the analysis of the algorithm, we find that all three conceptual edges related to the rolling conceptual vertex v_{13} are used in this solving step demonstrated in Fig. 13(*g*), meaning a rolling constraint equation set shown in Eq. (5) should be included in this subsystem. Then, we find edge $v_{11}\vec{v}_4$ is a distance constraint, which should also be added to the subsystem.

Finally, v_5 is the last unfixed vertex, and its location can be solved using the locations of v_{11} and v_4 and two distance constraints corresponding to the two *PP*-edges $v_5\vec{v}_4$ and $v_5\vec{v}_{11}$ (see Fig. 13(*h*)).

5 Discussion

5.1 Application. The constraint data structure is stored in a set of matrices, as described in Ref. [17], and in the Appendix. These matrices compactly save all the necessary information for mobility analysis, position analysis, and canvas display. Furthermore, this data structure heavily referenced the matrices in Ref. [42], so the matrices can be easily converted into other formats for other research interests. For example, the incidence matrix we use can be converted to an adjacency matrix for graph isomorphism tests. Additionally, we consider the valuable package in the following scenarios:

(1) Efficient planar linkage simulation. The mobility of the planar geared linkage is analyzed and decomposed as much as possible, ensuring an efficient solving path for the positional analysis. The estimated time and structural complexity of a geared linkage are provided in Ref. [15].
(2) Furthermore, the efficient planar linkage simulation is extremely helpful for massive dataset generation [43] for the data-driven techniques [5–14], and other optimization-based synthesis techniques such as the evolutionary algorithms [2,3] for complex planar mechanisms.
(3) The system of equations is built from the constraint graph, which can be used for the study of Assur graphs like [44], and the study of velocity and higher-order derivatives.

5.2 Limitation and Fallback Plans. The pebble game framework only determines whether or not a structure is generically flexible or rigid. So, identifying the edge cases may need additional math tools like the matroid theory [45]. However, determining the singularity due to the symmetric properties of the kinematic structure is generally costly in time because some singularities are by design, and some singularities only occur when the mechanism reaches a particular state [46].

Another potential limitation arises naturally from the point-based model, similar to the *LL*-edge case shown in Fig. 7, where a set of conceptual edges forms a (2, 3)-tight graph. Yet, the corresponding numerical system has an infinite number of solutions. In essence, our edge case reflects a variant of the Zarankiewicz Problem [47]: a (2, 3)-tight subgraph may exist within a *PL*-edge-only graph. However, such configurations are rare and do not typically arise in the constraint graphs for the planar geared linkages. We provide further discussion of this phenomenon in the Appendix.

To address these limitations, we can employ numerical methods to solve the entire constraint system, thereby eliminating the need for the mobility determination method presented in this work. For the cases related to singularity, we can always make an extra numerical computation before the mobility analysis. As for the cases related to graph semantics, we can check whether at least one *PP*-edge exists in a geometric constraint subsystem. If not, we can always compute the geometric system without any decomposition. However, simulating these edge cases can be time-consuming.

6 Conclusions

This article presents an advanced computational framework for analyzing and decomposing planar mechanisms using an extended pebble game algorithm. By leveraging algebraic graph theory and introducing a structured reduction and decomposition process, we efficiently determined the rigidity and solvability of kinematic systems. The proposed methodology extends the classical pebble game by incorporating *conceptual vertices and edges*, enabling the handling of additional constraint types, including rolling joints, slots, and angle–angle constraints. Another key contribution of this work is the introduction of the *edge-cluster dictionary*, which enhances the efficiency of numerical system decomposition by identifying the skeletal structure of the mechanism. Additionally, we demonstrated how Assur graph decomposition provides an optimal sequence for solving constraint systems, leading to reduced computational overhead. The framework achieves an overall complexity of $O(|V|^2)$, making it suitable for real-time kinematic simulation and CAD applications. Case studies validate the effectiveness of this approach in modeling and analyzing various planar mechanisms, including geared linkages and multi-actuated systems.

Acknowledgment

We sincerely thank Professor David Conlon from the Department of Mathematics at California Institute of Technology for his suggestion on the possible direction to prove/disprove the property of the point-based model. We are also grateful to the anonymous reviewers whose insights and questions helped improve this article significantly.

Conflict of Interest

There are no conflicts of interest.

Data Availability Statement

The datasets generated and supporting the findings of this article are obtainable from the corresponding author upon reasonable request.

Appendix

Matrices for Constraint Representation. First, the definition of the set of matrices **M** goes as follows:

$$\mathbf{M} = \begin{cases} B:\ B_{l\times n} \\ S:\ S_{m\times 4} \\ W:\ W_{w\times 8} \\ I:\ I_{a\times 4} \\ P:\ P_{n\times 2} \end{cases} \tag{A1}$$

where the B-matrix saves the P-clusters row by row such that

$$B_{i,j} = \begin{cases} 1 & \text{if P} - \text{cluster i contains P} - \text{vertex j} \\ 0 & \text{otherwise} \end{cases} \tag{A2}$$

The edges of the P-clusters defined by a row of the B-matrix can be extracted using Henneberg I moves, as demonstrated in Sec. 2.3.

Then, the S-matrix saves each line constraint with each of its rows, such that

$$S = \begin{bmatrix} p_0 & q_0 & r_0 & l_0 \\ \vdots & \vdots & \vdots & \vdots \\ p_{m-1} & q_{m-1} & r_{m-1} & l_{m-1} \end{bmatrix}_{m\times 4} \tag{A3}$$

where the p, q, and r are the P-vertices that are on the same line. For displaying purposes, we also include the link ID in the last column.

The S-matrix guides the initial creation of PL-edges. We can create one L-vertex from each row, and connect all three P-vertices v_p, v_q, and v_r to make the three PL-edges.

Next, we have the actuator matrix I defined as

$$I = \begin{bmatrix} i & j & k & R \\ \vdots & \vdots & \vdots & \vdots \\ p & q & r & P \end{bmatrix}_{a\times 4} \tag{A4}$$

The set of P-vertices $i, j,$ and k corresponds to a particular rotary ("R") actuator, while the set of P-vertices $p, q,$ and r corresponds to a particular linear actuator ("P" for pump).

For the rotary actuator, a dynamic edge can be created between v_i and v_k. For the linear actuator, the dynamic edge is $v_p v_q$.

Then, we have the wheel matrix to define the angle–angle constraint

$$W = \begin{bmatrix} c_{00} & c_{01} & r_{00} & r_{01} & l_{01} & l_{02} & l_{03} & W_o \\ c_{10} & c_{11} & r_{10} & r_{11} & l_{11} & l_{12} & l_{13} & W_i \\ c_{20} & c_{21} & r_{20} & r_{21} & l_{21} & l_{22} & l_{23} & B \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \end{bmatrix}_{w\times 8} \tag{A5}$$

In this matrix, the first and the second columns correspond to the P-vertices of two rotational centers of the wheel pairs. Then, the following two columns correspond to the radius of the two wheels. The next three l-columns correspond to the cluster index of the wheel pair, and the last column denotes the type of wheel-pair type: outer-contact (W_o), inner contact (W_i), and belt (B).

As demonstrated in Fig. 11, four vertices and five edges are created. Furthermore, we should note that processing the W-matrix will expand the original B-matrix so that the rigidity of each wheel is maintained.

Finally, the P-matrix is the positional matrix for the initial n number of P-vertex locations.

Potential Direction of Proof. We seek a definitive answer to the following question: *Can a constraint subgraph composed solely of conceptual edges in the point-based model ever be (2, 3)-tight?* This is nontrivial—if such subgraphs do not exist, it implies that any (2, 3)-tight graph in the point-based model must include at least one PP-edge, and thus, the model avoids the similarity degeneracy issue illustrated in Fig. 7. While a full resolution is beyond the scope of this work, we outline a possible direction for proving the existence of such graphs, inspired by the suggestion from Professor David Conlon.

Rolling Joint. We can easily make a proof for the conceptual edges from the rolling joint because it is impossible that these edges form a rigid subgraph, and that they do not exist in a conceptual edge-only rigid subgraph. After all, they do not contribute any extra pebble consumption from the P-vertices. For example, the five conceptual edges in Fig. 11(*c*) consumes at most one pebble from all the P-vertices (v_5, v_6, v_1, and v_3) in this graph. However, at the stage of inserting the conceptual edges into the graph, each of v_5 and v_6 has at most one edge connecting to them, so their graph is always flexible. Similarly, v_8 is flexible because it connects to two flexible edges $v_5\vec{v}_8$ and $v_6\vec{v}_8$, and there is one free DOF for v_8 when one pebble is spent on $v_8\vec{v}_7$. Then, we can address the two pebbles on v_7 with the edges $v_7\vec{v}_1$ and $v_7\vec{v}_3$, resulting in no cost of pebbles on the rotational centers v_1 and v_3, while v_1 and v_3 are the only two P-vertices on this graph that are the endpoints of other conceptual edges.

Prismatic Joint. It is possible that a combination of PL-edges alone may form a rigid graph during the edge insertion phase of the reduction process. By construction, such a graph is bipartite: PP-edges are not conceptual edges, and LL-edges are absent. Moreover, according to Reduction Rule 1 for PL-edges, we can deduce that this bipartite graph is $K_{2,2}$-free.

We recall that bipartite graphs contain only even-length cycles. When further constrained to be $K_{2,2}$-free, i.e., four-cycle-free, the smallest possible cycle has length six. This implies that any minimal loop formed entirely by PL-edges must involve at least six consecutive PL-edges, connecting three P-vertices and three L-vertices. Such configurations are rarely encountered in practical linkage mechanisms. In particular, the presence of three distinct line constraints (e.g., prismatic joints) in a six-edge loop introduces flexibility: even after removing three pebbles to account for global degrees-of-freedom, the loop typically retains at least three free pebbles. Relevant discussions on the enumeration of kinematic chains with prismatic joints can be found in the works of Rojas and Thomas [48] and Eleashy [49].

This construction can also be viewed as a variant of the Zarankiewicz Problem [47], an open problem in extremal graph theory that asks: what is the maximum number of edges in a bipartite graph with a fixed number of vertices and no $K_{s,t}$ subgraph? In our context, a bipartite, $K_{2,2}$-free graph may arise as the Levi graph (i.e., point-line incidence graph), where geometric points and lines are treated as distinct vertices, and an edge is drawn between a point and a line if the point lies on that line. On an

affine plane over $\mathbb{F}_q^2$ of order $q = p^k$, where p is a prime and k is a positive integer, we have:

(1) Any two distinct points lie on exactly one common line;
(2) Any two lines intersect in at most one point;
(3) Each line contains q points;
(4) Each point lies on $(q + 1)$ lines;

Such a plane contains q^2 points and $q(q + 1)$ lines. Its Levi graph contains $q^2(q + 1)$ edges. The number of edges grows asymptotically as $O(q^3)$, while the number of vertices grows as $O(q^2)$. Therefore, for sufficiently large q, the Levi graph becomes too dense to satisfy the (2, 3)-tight condition globally. Nonetheless, suitable subgraphs of the Levi graph may still satisfy (2, 3)-tightness. Thus, a (2, 3)-tight and $K_{2,2}$-free bipartite graph composed entirely of *PL*-edges *may* indeed exist.

For instance, when $q = 4$, the Levi graph has 80 edges and 36 vertices, yielding $80 - (2 \times 36 - 3) = 11$ surplus edges beyond the (2, 3)-tight threshold. However, such an incidence structure—with $q(q + 1)$ line constraints and q^2 distinct point-line incidences—rarely arises in realistic planar linkage designs for $q > 3$.

References

[1] Jalon, J., and Bayo, E., 2001, *Kinematic and Dynamic Simulation of Multibody Systems—The Real-Time Challenge*, Springer-Verlag, Heidelberg, Germany.
[2] Tsai, C.-Y., and Chen, Y.-T., 2003, "An Evolutionary Algorithm for Path Synthesis of Mechanisms," Mech. Mach. Theory, **38**(7), pp. 889–906.
[3] Tsai, C.-Y., and Chen, Y.-T., 2009, "A GA–DE Hybrid Evolutionary Algorithm for Path Synthesis of Four-Bar Linkage," Mech. Mach. Theory, **44**(6), pp. 1079–1094.
[4] Gogate, G. R., and Matekar, S. B., 2012, "Optimum Synthesis of Motion Generating Four-Bar Mechanisms Using Alternate Error Functions," Mech. Mach. Theory, **54**, pp. 41–61.
[5] Khan, N., Ullah, I., and Al-Grafi, M., 2015, "Dimensional Synthesis of Mechanical Linkages Using Artificial Neural Networks and Fourier Descriptors," Mech. Sci., **6**(1), pp. 29–34.
[6] Vermeer, K., Kuppens, R., and Herder, J., 2018, "Kinematic Synthesis Using Reinforcement Learning," Volume 2A: 44th Design Automation Conference, Quebec City, Quebec, Canada, Aug. 26–29, ASME.
[7] Deshpande, S., and Purwar, A., 2020, "An Image-Based Approach to Variational Path Synthesis of Linkages," ASME J. Comput. Inf. Sci. Eng., **21**(2), p. 021005.
[8] Sharma, S., and Purwar, A., 2022, "A Machine Learning Approach to Solve the Alt-Burmester Problem for Synthesis of Defect-Free Spatial Mechanisms," ASME J. Comput. Inf. Sci. Eng., **22**(2), p. 021003.
[9] Nurizada, A., and Purwar, A., 2023, "An Invariant Representation of Coupler Curves Using a Variational AutoEncoder: Application to Path Synthesis of Four-Bar Mechanisms," ASME J. Comput. Inf. Sci. Eng., **24**(1), p. 011008.
[10] Lyu, Z., and Purwar, A., 2025, "Deep Learning Conceptual Design of Sit-to-Stand Parallel Motion Six-Bar Mechanisms," ASME J. Mech. Des., **147**(1), p. 013306.
[11] Yu, S.-C., Chang, Y., and Lee, J.-J., 2022, "A Generative Model for Path Synthesis of Four-Bar Linkages Via Uniform Sampling Dataset," Proc. Inst. Mech. Eng., Part C: J. Mech. Eng. Sci., **237**(4), pp. 811–829.
[12] Chen, C.-H., 2023, "Application of Multiple Deep Neural Networks to Multi-solution Synthesis of Linkage Mechanisms," Machines, **11**(11), p. 1018.
[13] Fogelson, M. B., Tucker, C., and Cagan, J., 2023, "GCP-Holo: Generating High-Order Linkage Graphs for Path Synthesis," ASME J. Mech. Des., **145**(7), p. 073303.
[14] Nobari, A. H., Srivastava, A., Gutfreund, D., Xu, K., and Ahmed, F., 2024, "Link: Learning Joint Representations of Design and Performance Spaces Through Contrastive Learning for Mechanism Synthesis," Trans. Mach. Lear. Res., pp. 2835–2885
[15] Lyu, Z., Purwar, A., and Liao, W., 2024, "A Unified Real-Time Motion Generation Algorithm for Approximate Position Analysis of Planar N-Bar Mechanisms," ASME J. Mech. Des., **146**(6), p. 063302.
[16] Picard, C., Schiffmann, J., and Ahmed, F., 2023, "Dated: Guidelines for Creating Synthetic Datasets for Engineering Design Applications," Volume 3A: 49th Design Automation Conference, Boston, MA, Aug. 20–23,
[17] Lyu, Z., and Purwar, A., 2024, "Point-Based Models: A Unified Approach for Geometric Constraint Analysis of Planar n-Bar Mechanisms With Rotary, Sliding, and Rolling Joints," ASME J. Mech. Rob., **17**(4), p. 044511.
[18] Fudos, I., and Hoffmann, C., 2004, "A Graph-Constructive Approach to Solving Systems of Geometric Constraints," ACM Trans. Graph., **16**(2), pp. 179–216.
[19] Ait-Aoudia, S., and Foufou, S., 2010, "A 2D Geometric Constraint Solver Using a Graph Reduction Method," Adv. Eng. Softw., **41**(10–11), pp. 1187–1194.
[20] Sljoka, A., Shai, O., and Whiteley, W., 2011, "Checking Mobility and Decomposition of Linkages Via Pebble Game Algorithm," Volume 6: 35th Mechanisms and Robotics Conference, Parts A and B, Washington, DC, Aug. 28–31.
[21] Lee, A., and Streinu, I., 2008, "Pebble Game Algorithms and Sparse Graphs," Discrete Math., **308**(8), pp. 1425–1437.
[22] Lee, A., Streinu, I., and Theran, L., 2007, "Graded Sparse Graphs and Matroids," J. Univer. Comput. Sci., **13**(10).
[23] Jackson, B., and Owen, J., 2016, "A Characterisation of the Generic Rigidity of 2-Dimensional Point–Line Frameworks," J. Combin. Theory, Ser. B, **119**, pp. 96–121.
[24] Berg, A. R., and Jordán, T., 2003, "Algorithms for Graph Rigidity and Scene Analysis," *Algorithms - ESA 2003*, G. Di Battista, U. Zwick, eds., Springer, Berlin, Heidelberg, pp. 78–89.
[25] Frank, A., 2011, *Connections in Combinatorial Optimization*, Oxford University Press, New York.
[26] Jacobs, D. J., and Thorpe, M. F., 1995, "Generic Rigidity Percolation: The Pebble Game," Phys. Rev. Lett., **75**(22), pp. 4051–4054.
[27] Jacobs, D. J., and Hendrickson, B., 1997, "An Algorithm for Two-Dimensional Rigidity Percolation: The Pebble Game," J. Comput. Phys., **137**(2), pp. 346–365.
[28] Laman, G., 1970, "On Graphs and Rigidity of Plane Skeletal Structures," J. Eng. Math., **4**, pp. 331–340.
[29] Servatius, B., Shai, O., and Whiteley, W., 2010, "Combinatorial Characterization of the Assur Graphs From Engineering," Eur. J. Combin., **31**(4), pp. 1091–1104.
[30] Moore, E. F., 1959, "The Shortest Path Through a Maze," Proceedings of the International Symposium on the Theory of Switching, Part II, Apr. 2–5, Harvard University Press, Cambridge, MA, pp. 285–292.
[31] Hidalgo, M. R., and Joan-Arinyo, R., 2017, "A Henneberg-Based Algorithm for Generating Tree-Decomposable Minimally Rigid Graphs," J. Symb. Comput., **79**, pp. 232–248.
[32] Pennock, G. R., and Kamthe, G. M., 2006, "Study of Dead-Centre Positions of Single-Degree-of-Freedom Planar Linkages Using Assur Kinematic Chains," Proc. Inst. Mech. Eng., Part C: J. Mech. Eng. Sci., **220**(7), pp. 1057–1074.
[33] Diestel, R., 2017, *Graph Theory*, 5th ed., Graduate Texts in Mathematics, Vol. 173, Springer, New York.
[34] Tarjan, R., 1972, "Depth-First Search and Linear Graph Algorithms," SIAM J. Comput., **1**(2), pp. 146–160.
[35] Kahn, A. B., 1962, "Topological Sorting of Large Networks," Commun. ACM, **5**(11), pp. 558–562.
[36] Moré, J. J., 1978, "The Levenberg-Marquardt Algorithm: Implementation and Theory," *Numerical Analysis*, G. A. Watson, ed., Springer, Berlin, Heidelberg, pp. 105–116.
[37] Petuya, V., Macho, E., Altuzarra, O., and Pinto, C., 2011, "Educational Software Tools for the Kinematic Analysis of Mechanisms," Comput. Appl. Eng. Educ., **6**(4), pp. 261–266.
[38] Wampler, C. W., 1999, "Solving the Kinematics of Planar Mechanisms," ASME J. Mech. Des., **121**(3), pp. 387–391.
[39] W. R. Mathematica Inc., Online, Version 14.1, Champaign, IL, 2024, https://www.wolfram.com/mathematica
[40] Uchida, T., and McPhee, J., 2011, "Triangularizing Kinematic Constraint Equations Using Gröbner Bases for Real-Time Dynamic Simulation," Multibody Syst. Dyn., **25**(3), pp. 335–356.
[41] Mechanismic Inc., 2022, "MotionGen," http://www.motiongen.io
[42] Tsai, L., 2001, *Mechanism Design: Enumeration of Kinematic Structures According to Function*, CRC Press LLC, Boca Raton, FL.
[43] Nobari, A. H., Srivastava, A., Gutfreund, D., and Ahmed, F., 2022, "LINKS: A Dataset of a Hundred Million Planar Linkage Mechanisms for Data-Driven Kinematic Design," Volume 3A: 48th Design Automation Conference, St. Louis, MO, Aug. 14–17 .
[44] Glue, C. M., 2022, "Generating All Rigidity Circuits on at Most 10 Vertices and All Assur Graphs on at Most 11 Vertices," J. Integer Sequences, **25**.
[45] Gogu, G., 2005, "Chebychev–Grübler–Kutzbach's Criterion for Mobility Calculation of Multi-Loop Mechanisms Revisited Via Theory of Linear Transformations," Eur. J. Mech. - A/Solids, **24**(3), pp. 427–441.
[46] Park, F. C., and Kim, J. W., 1999, "Singularity Analysis of Closed Kinematic Chains," ASME J. Mech. Des., **121**(1), pp. 32–38.
[47] Conlon, D., 2022, "Some Remarks on the Zarankiewicz Problem," Math. Proc. Cambr. Philos. Soc., **173**(1), pp. 155–161.
[48] Rojas, N., and Thomas, F., 2012, "Distance-Based Formulations for the Position Analysis of Kinematic Chains," Ph.D. thesis, Universitat Politècnica de Catalunya, Catalonia, Spain.
[49] Eleashy, H., 2018, "A New Atlas for 8-Bar Kinematic Chains With Up to 3 Prismatic Pairs Using Joint Sorting Code," Mech. Mach. Theory, **124**, pp. 118–132.