RigiComp — A Mathematica package for computational rigidity of graphs

Georg Grasegger*

RigiComp is a collection of functions for combinatorial rigidity theory. In particular it contains commands for constructing rigid graphs, testing rigidity and computing the number of realizations for minimally rigid graphs. The package runs in Mathematica. This document serves as a documentation and handbook of RigiComp. All commands and their options are described together with references to literature about the algorithms in use.

*Johann Radon Institute for Computational and Applied Mathematics (RICAM), Austrian Academy of Sciences (ÖAW)
Supported by the Austrian Science Fund (FWF): P31888

Contents

1 Basic Information

This section contains some important information before you start. It describes how this document and the package shall be read and what requirements we have on data types.

1.1 Installation

RigiComp can be downloaded from Zenodo with doi:10.5281/zenodo.7457820.

RigiComp itself does not need to be installed but it needs Mathematica [11] to be used. Copy `RigiComp.wl` to your working directory and load it in Mathematica with

Example

```
In[1]:= Get["RigiComp.wl"]
```

RigiComp was developed and tested with Mathematica 12.

1.2 How to use

RigiComp runs in Mathematica. As such for every command `?Command` a message with a brief description is printed. For further help please read this document. On page 31 there is an index including all the commands described here.

This documents describes aims and output of all commands and contains input descriptions

Input

```
Function[a,b]
      a    description of a (data type of a)
      b    description of b (data type of b)
```

and examples

Example

```
In[2]:= Function[a,b]
Out[2]= output
```

For some commands there are options for adjustment. If there is more than one they are described in a box

Options

`Option1`	description of option 1	`default`
`Option2`	description of option 2	`default`

For most of the commands syntax information is given such that, when working in a notebook, Mathematica would highlight for instance when there are too few or too many input elements.

1.3 Data Types and Requirements

In RigiComp we only consider simple connected graphs without loops. Graphs can be represented in two different ways, by the built in Mathematica `Graph` data type and by an integer representation (see Section 2 for details). Most commands do allow both of them as input arguments. In this document we are not emphasizing on this any more but rather pick a convenient representation for each example. Since the `Graph` data type is not viewed nicely on the command line we usually either take integer representations or edge lists for the output style.

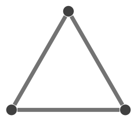
Note that a graph is always assumed to have vertices $\{1, 2, \dots, n\}$. Some commands would crash if this is not the case. In order to avoid this transform your graph first using `StandardGraph`.

In Mathematica edges have a data type `UndirectedEdge[v1,v2]`. In case a function of RigiComp needs an edge we require `{v1,v2}` instead.

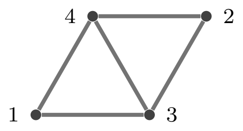
Many rigidity arguments depend on the dimension in which the graph is supposed to be. Since two dimensional rigidity is the most common one, in many functions of RigiComp one can omit the specification of the dimension to stay in dimension two. In this document, however, we always specify the dimension.

2 Graphs and Representations

Mathematica provides a data type `Graph` which is convenient to use but sometimes less convenient to store. For this reason RigiComp also works with an integer representation of graphs (compare also [2, 9, 1, 3, 7]). This representation is obtained from taking the upper triangular part of the adjacency matrix and interpret it as binary digits. The decimal number we get from this is our integer representation.



$$\longleftrightarrow \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix} \longleftrightarrow (111)_2 = 7$$



$$\longleftrightarrow \begin{pmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \longleftrightarrow (011111)_2 = 31$$

Since we do not allow multiedges, loops or isolated vertices the representation is unique. The following example shows how to get the integer representation for the triangle graph.

Example

```
In[3]:= Graph2G[Graph[{{1,2},{1,3},{2,3}}]]
Out[3]= 7
```

The inverse functions is the following

Example

```
In[4]:= G2Graph[7]
```

On the command line this will return `Graph[<3>, <3>]` but when using a notebook the output will be shown as a figure. `G2Graph` allows all Options that the built in `Graph` accepts.

RigiComp has also a shorthand for producing and reading from `Graph6` data format (where the header `>>graph6<<` is omitted):

Example

```
In[5]:= G2Gs[7]
Out[5]= Bw
```

Similarly `Gs2Graph` and `Graph2Gs` transfer between Mathematica `Graph` data type and the `Graph6` data type.

Sometimes it is convenient to work with edge lists. We can transform to and from those by

Example

```
In[6]:=  G2Edges[7]
Out[6]=  {{1, 2}, {1, 3}, {2, 3}}
In[7]:=  Edges2G[{{1, 2}, {1, 3}, {2, 3}}]
Out[7]=  7
```

Mathematica provides the function `EdgeList` which is applied to a `Graph` object. The result is a list of objects of type `UndirectedEdge`. Since RigiComp usually works with edges as lists, we have `GEdges`.

Example

```
In[8]:=  graph=Graph[{1,2},{UndirectedEdge[1,2],UndirectedEdge[2,3]}]
In[9]:=  GEdges[graph]
Out[9]=  {{1,2},{2,3}}
```

Similarly to edges we can get the adjacency matrix from integer representations.

Example

```
In[10]:=  G2Mat[7]
Out[10]=  {{0, 1, 1}, {1, 0, 1}, {1, 1, 0}}
In[11]:=  Mat2G[{{0, 1, 1}, {1, 0, 1}, {1, 1, 0}}]
Out[11]=  7
```

When we do graph constructions we want to make sure to construct each graph isomorphically just once. Therefore, we use a normal form, which currently is based on the Mathematica command `CanonicalGraph`. Working with integer representations we use `GraphNormalForm`. Then the graphs represented by 30 and 45 are both a 4-cycle and isomorphic. The normal representative is 30. Note that this might depend on the version of Mathematica.

Example

```
In[12]:=  GraphNormalForm[45]
Out[12]=  30
In[13]:=  G2Edges[45]
Out[13]=  {{1, 2}, {1, 4}, {2, 3}, {3, 4}}
In[14]:=  G2Edges[30]
Out[14]=  {{1, 3}, {1, 4}, {2, 3}, {2, 4}}
```

Sometimes we do not need a normal form, but we would like to have a graph with consecutive vertex names $\{1, 2, \ldots, n\}$. This is necessary for many of the commands in this package. We get such a graph by applying `StandardGraph`.

Example

```
In[15]:= GEdges[StandardGraph[Graph[{3, 4, 6},
           UndirectedEdge@@@{{3, 4}, {4, 6}, {6, 3}}]]]
Out[15]= {{1, 2}, {2, 3}, {3, 1}}
```

Note for the example that `UndirectedEdge@@@` transforms the following list from a list of pairs to a list of undirected edges.

3 Constructions

In rigidity theory there is a collection of constructions which in some cases preserve rigidity of a graph. RigiComp has implemented many of them. Note, however, that not all of them do always preserve rigidity and they do not check rigidity themselves. See Section 4 for details on checking rigidity.

3.1 Extensions

A classical type of construction are k-extensions or sometimes also called Henneberg moves.

Definition 3.1.
Let $G = (V, E)$ be a graph considered in dimension d and $F \subset E$ with $|F| = k$ and let $v \notin V$. Let $H = (W, F)$ be the induced graph of F. Let further S be a set of vertices with $S \cap W = \emptyset$ and $|S| + |W| = d + k$. We define $E_v = \{\{v, u\} \mid u \in W \cup S\}$. Then $G' = (V \cup \{v\}, (E \setminus F) \cup E_v)$ is called a d-dimensional k-extension of G.

In RigiComp we can do such constructions in two different ways. We can either specify the deleted edges and chosen vertices, or we can do all such extensions that are possible on the input graph. The first one, though, only works for 0- and 1-extensions. `K0Extension` takes as input a graph and a list of vertices. From this list the dimension is determined, since the list needs to have as many vertices as the dimension of the space we want the extension to take place in. The output is a graph in integer representation.

Input

```
K0Extension[g, v]
     g    graph (integer or Graph)
     v    list of d vertices, where d defines the dimension (list)
```

Example

```
In[16]:= edges={{1, 2}, {1, 3}, {2, 3}};
In[17]:= G2Edges[K0Extension[Edges2G[edges], {1, 2}]]
Out[17]= {{1, 2}, {1, 3}, {1, 4}, {2, 3}, {2, 4}}
In[18]:= edges={{1, 2}, {1, 3}, {1, 4}, {2, 3}, {3, 4}};
In[19]:= G2Edges[K0Extension[Edges2G[edges], {1, 2, 3}]]
Out[19]= {{1, 2}, {1, 3}, {1, 4}, {1, 5}, {2, 3}, {2, 5}, {3, 4}, {3, 5}}
```

`K1Extension` takes as input a graph, a pair of vertices that are an edge in the graph and a list of additional vertices. Again the dimension is determined from the latter. The output is a graph in integer representation.

Input

```
K1Extension[g, e, v]
     g    graph (integer or Graph)
     e    edge (list of 2 vertices)
     v    list of d-1 vertices, where d defines the dimension (list)
```

Example

```
In[20]:= edges={{1, 3}, {1, 4}, {2, 3}, {2, 4}, {3, 4}};
In[21]:= G2Edges[K1Extension[Edges2G[edges], {1, 3}, {4}]]
Out[21]= {{1, 4}, {1, 5}, {2, 3}, {2, 4}, {3, 4}, {3, 5}, {4, 5}}
In[22]:= edges={{1,4},{1,5},{2,3},{2,4},{2,5},{3,4},{3,5}};
In[23]:= G2Edges[K1Extension[Edges2G[edges],{1,4},{2,3,5}]]
Out[23]= {{1, 5}, {1, 6}, {2, 3}, {2, 4}, {2, 5}, {2, 6}, {3, 4},
          {3, 5}, {3, 6}, {4, 6}, {5, 6}}
```

In order to get all possible extensions of a given graph in some dimension, we use `KExtensions` with the shorthand of `K0Extensions` and `K1Extensions`. `KExtensions` constructs k-extensions of the input graph for $k \leq d-1$, where d is the dimension. The output is a list of integer representations in normal form (i. e. all graphs in the list are non-isomorphic).

Input

```
KExtensions[g,d]
     g    graph (integer or Graph)
     d    dimension (integer)
```

Example

```
In[24]:= K0Extensions[254,2]
Out[24]= {3326, 3934, 4011, 10479, 12511}
In[25]:= K1Extensions[254,2]
Out[25]= {3934, 4011, 6891, 7672, 7916}
In[26]:= KExtensions[254,2]
Out[26]= {3326, 3934, 4011, 6891, 7672, 7916, 10479, 12511}
```

In general the k can be specified by setting the option `SetStart` and `SetLimit`.

Options

`SetStart`	integer for the minimal k to be used for the extension	`0`
`SetLimit`	integer for the maximal k to be used for the extension	`d-1`

Example

```
In[27]:= K1Extensions[511, 3]
Out[27]= {7935, 8187, 16350}
In[28]:= KExtensions[511, 3, SetStart -> 1, SetLimit -> 1]
Out[28]= {7935, 8187, 16350}
```

Sometimes special cases of extensions are of interest. For instance 2-extensions can be distinguished by whether the two chosen edges share a vertex (V-Replacement) or not (X-Replacement). Note that, these extensions do not necessarily preserve rigidity.

Example

```
In[29]:= VReplacement[511, 3]
Out[29]= {4095, 7679, 7935, 8187}
In[30]:= XReplacement[511, 3]
Out[30]= {7935, 8187}
In[31]:= KExtensions[511, 3, SetStart -> 2]
Out[31]= {4095, 7679, 7935, 8187}
```

In other cases we want to distinguish k-extensions on the edges that are induced on the chosen vertices. For instance in a 0-extension in dimension two there can be an edge between the two chosen vertices or not. For some of these we have short hand functions. For instance `K0ExtensionD2Sub1` only constructs 0-extensions where there is an edge between the vertices, where `K0ExtensionD2Sub0` does the opposite. Similarly we have short hands for `K1ExtensionD2Sub7`, `K1ExtensionD2Sub3`, `K1ExtensionD2Sub1` which consider 1-extensions in dimension two where the chosen edge with the additional vertex form a triangle, a path or a single edge, respectively. In general we can set the option `UseSubgraphOnChosenVertices` of `KExtensions` which by default is `False` but can be set to an integer representation of a graph.

Example

```
In[32]:= KExtensions[7916, 2, SetStart -> 1,
          UseSubgraphOnChosenVertices -> 7]
Out[32]= {1256267}
In[33]:= K1ExtensionD2Sub7[7916]
Out[33]= {1256267}
```

3.2 Coning

The coning operation is usually used for construction graphs that are considered in one dimension higher. It is known that coning preserves some rigidity properties.

Definition 3.2.
Let $G = (V, E)$ be a graph and let $u \notin V$. Then the cone of G is the graph $(V \cup \{u\}, E \cup \{\{u, v\} \mid v \in V\})$.

RigiComp does this construction with the `Coning` command.

Input
```
Coning[g]
     g    graph (integer or Graph)
```

Example
```
In[34]:= Coning[31]
Out[34]= 511
```

3.3 Vertex-Splitting

In vertex-splitting a vertex is split into two and the original incident edges are distributed.

Definition 3.3.
Let $G = (V, E)$ be a graph with $v \in V$ and let $V_1 \cup W \cup V_2$ be a partition of the neighbors of v where $|W| = d - 1$ for dimension d. The vertex-splitting operation deletes the edges $\{v, w_2\}$ with $w_2 \in V_2$, adds a new vertex $\bar{v}$ and adds the edges $\{\bar{v}, w\}$ for all $w \in W$, $\{v, \bar{v}\}$ as well as $\{\bar{v}, w_2\}$ for all $w_2 \in V_2$.

Similarly to the k-extensions we can either do one specific vertex-split or construct all possible ones, both with `VertexSplitting` and different input arguments.

Input
```
VertexSplitting[g,d,v,W,V1]
VertexSplitting[g,d]
     g    graph (integer or Graph)
     d    dimension (integer)
     v    vertex of g (integer)
     W    d-1 neighbors of v in g (list)
     V1   neighbors of v not in W (list, possibly empty)
```

Example
```
In[35]:= edges = {{1, 4}, {1, 5}, {1, 6}, {2, 3}, {2, 5}, {2, 6}, {3, 4},
           {3, 6}, {4, 5}};
In[36]:= GEdges[VertexSplitting[Graph[edges], 2, 1, {4}, {6}]]
Out[36]= {{1, 4}, {1, 5}, {2, 3}, {2, 5}, {2, 6}, {3, 4}, {3, 6}, {4, 5},
           {7, 4}, {7, 1}, {7, 6}}
In[37]:= VertexSplitting[7916, 2]
```

```
Out[37]= {127575, 1256267, 112525, 1269995}
```

The specific construction gives error messages if the input is invalid.

3.4 Spider-Splitting

In spider-splitting a vertex is split into two and the original incident edges are distributed. This operation is sometimes also called diamond-splitting, or vertex-to-4-cycle operation. Often it is only defined for dimension two for rigidity reasons but we give a somehow general definition.

Definition 3.4.
Let $G = (V, E)$ be a graph and $v \in V$ with d neighbors $W = \{w_1, \dots, w_d\}$. The remaining neighbors of v are partitioned in two sets N_1 and N_2 (possibly empty). Let $v' \notin V$. Then spider-splitting yields the graph $G' = (V \cup \{v'\}, (E \setminus \{\{u, v\} \mid u \in N_2\}) \cup \{\{u, v'\} \mid u \in N_2 \cup W\})$.

Similarly to the vertex-splitting we can either do one specific vertex-split or construct all possible ones, both with `SpiderSplitting` and different input arguments. There are aliases `DiamondSplitting` and `VertexToC4Splitting`. Note that for the specific operation the dimension is determined from the set W.

Input

```
SpiderSplitting[g,v,W,V1]
SpiderSplitting[g,d]
     g    graph (integer or Graph)
     d    dimension (integer)
     v    vertex of g (integer)
     W    d neighbors of v in g (list), d is determined from W
     V1   neighbors of v not in W (list, possibly empty)
```

Example

```
In[38]:= edges = {{1, 4}, {1, 5}, {1, 6}, {2, 3}, {2, 5}, {2, 6}, {3, 4},
          {3, 6}, {4, 5}};
In[39]:= GEdges[SpiderSplitting[Graph[edges], 1, {4, 5}, {}]]
Out[39]= {{1, 4}, {1, 5}, {1, 6}, {2, 3}, {2, 5}, {2, 6}, {3, 4}, {3, 6},
          {4, 5}, {7, 4}, {7, 5}}
In[40]:= SpiderSplitting[7916, 2]
Out[40]= {127575, 120478}
```

3.5 Reductions

The inverse of k-extensions are k-reductions. RigiComp does not yet allow general k-reductions, but the most common ones for $k \in \{0, 1\}$. Similarly to the extensions we can either do one particular one specifying the vertex to be deleted, or get all reductions on all suitable vertices. For 0-reductions the process is clear, so `K0Reduction` just removes a given degree two vertex.

Input

```
K0Reduction[g,d,v]
     g    graph (integer or Graph)
     d    dimension (integer)
     v    vertex of g with degree d (integer)
```

Example

```
In[41]:= GEdges[K0Reduction[Graph[{{1, 2}, {1, 3}, {2, 3}}], 2, 3]]
Out[41]= {{1, 2}}
```

To do all possible single step 0-reductions we use `K0Reductions`. Here we get a list of (possibly isomorphic) graphs obtained from the input by a 0-reduction. We can use normal forms with the option `UseNormalForm` (which by default is `False`). The option `Unify` deletes duplicates.

Input

```
K0Reductions[g,d]
     g    graph (integer or Graph)
     d    dimension (integer)
```

Options

`UseNormalForm`	specify whether output is normalized	`False`
`Unify`	specify whether isomorphic output is ignored	`False`

Example

```
In[42]:= K0Reductions[223, 2]
Out[42]= {31, 199, 217}
In[43]:= K0Reductions[223, 2, UseNormalForm -> True]
Out[43]= {31, 31, 31}
In[44]:= K0Reductions[223, 2, UseNormalForm -> True, Unify -> True]
Out[44]= {31}
```

For 1-reductions the process is not so immediate, since we need to add an edge and we have up to three possibilities for this. Hence, in `K1Reduction` there is an option for which edge(s) we want to add. The default value for `PickEdges` is `All`. Instead one can put an integer or a list of integers which refer to the index of the chosen edge. Note that there is no check whether the index makes sense.

Input

```
K1Reduction[g,d,v]
     g    graph (integer or Graph)
     d    dimension (integer)
     v    vertex of g with degree d+1 (integer)
```

Example

```
In[45]:= graph = Graph[{{1, 4}, {1, 5}, {1, 6}, {2, 3}, {2, 5}, {2, 6},
           {3, 4}, {3, 6}, {4, 5}}];
In[46]:= GEdges /@ K1Reduction[graph, 2, 6]
Out[46]= {{{1,4},{1,5},{2,3},{2,5},{3,4},{4,5},{1,2}},
           {{1,4},{1,5},{2,3},{2,5},{3,4},{4,5},{1,3}}}
In[47]:= GEdges /@ K1Reduction[graph, 2, 6, PickEdges -> 1]
Out[47]= {{{1, 4}, {1, 5}, {2, 3}, {2, 5}, {3, 4}, {4, 5}, {1, 2}}}
```

To do all possible single step 1-reductions we use `K1Reductions`. Here we get a list of (possibly isomorphic) graphs obtained from the input by a 1-reduction. Again the options `UseNormalForm` and `Unfiy` work.

Input

```
K1Reductions[g,d]
     g    graph (integer or Graph)
     d    dimension (integer)
```

Options

`UseNormalForm`	specify whether output is normalized	`False`
`Unify`	specify whether isomorphic output is ignored	`False`

Example

```
In[48]:= K1Reductions[7916, 2]
Out[48]= {750, 749, 7228, 7213, 7620, 7366, 11976, 3800, 22120, 5992,
           749, 493}
In[49]:= K1Reductions[7916, 2, UseNormalForm -> True, Unify -> True]
Out[49]= {254}
```

4 Rigidity Check

RigiComp is capable of checking different rigidity properties of graphs and partially frameworks. Some checks work probabilistically but many checks can be done symbolically as well, though they might be computationally expensive. The method to be used for checking can be specified by an option.

In this section we give a few definitions and theorems in order to clarify how we check the properties. For more details of the definitions we refer the user to literature on rigidity theory like for instance [17, 4, 10].

4.1 Rigidity Matrix

Let $G = (V, E)$ be a graph and ρ a realization of the graph in dimension d. Then the *d-dimensional rigidity matrix* $\mathcal{R}_d(G, \rho) = (r_{i,j})_{\substack{i\in\{1,\dots,m\}\\ j\in\{1,\dots,dn\}}}$, where n is the number of vertices and m the number of edges, is defined by

$$r_{k,\ell} = \begin{cases} \rho(v_i)_\kappa - \rho(v_j)_\kappa & \text{if } e_k = \{v_i, v_j\} \text{ and } \left\lfloor \frac{\ell-1}{d} \right\rfloor = i-1 \text{ and } \kappa = \ell - d(i-1) \\ \rho(v_j)_\kappa - \rho(v_i)_\kappa & \text{if } e_k = \{v_i, v_j\} \text{ and } \left\lfloor \frac{\ell-1}{d} \right\rfloor = j-1 \text{ and } \kappa = \ell - d(j-1) \\ 0 & \text{otherwise} \end{cases}$$

In RigiComp the rigidity matrix can be generated with a framework as an input, i. e. a graph with a realization. Alternatively a random placement can be chosen, or symbolic edge lengths can be used.

Input

```
RigidityMatrix[g,p]
     g    graph (integer or Graph)
     p    coordinates for the vertices of g (list of lists)
```

For frameworks `RigidityMatrix` yields the following for the three cycle with given realization $\rho(1) = (1, 1)$, $\rho(2) = (3, 2)$ and $\rho(3) = (2, 3)$:

Example

```
In[50]:= RigidityMatrix[7, {{1, 1}, {3, 2}, {2, 3}}]
Out[50]= {{-2, -1, 2, 1, 0, 0},
          {-1, -2, 0, 0, 1, 2},
          {0, 0, 1, -1, -1, 1}}
```

For a random choice of realizations `RandomRigidityMatrix` provides some options to control parameters. `RandomRange` defines the maximum for the random number. By default it is `Automatic` which means it takes $10^6 \cdot |V| \cdot d$, where d is the dimension. `RandomSet` allows to have integer or real realizations (default is `"Reals"`).

Input

```
RandomRigidityMatrix[g,d]
     g    graph (integer or Graph)
     d    dimension (integer)
```

Options

`RandomRange`	maximum value for random numbers	`Automatic`
`RandomSet`	set of which random numbers are taken from (integers, reals)	`"Reals"`

Example

```
In[51]:= RandomRigidityMatrix[7, 2, RandomRange -> 10,
          RandomSet -> "Integers"]
Out[51]= {{-8, -4, 8, 4, 0, 0},
          {11, 0, 0, 0, -11, 0},
          {0, 0, 19, 4, -19, -4}}
```

For a symbolic rigidity matrix `SymbolicRigidityMatrix` a variable has to be given in addition to the graph and the dimension.

Input

```
SymbolicRigidityMatrix[g,d,x]
     g    graph (integer or Graph)
     d    dimension (integer)
     x    variable for coordinates (symbol)
```

Example

```
In[52]:= SymbolicRigidityMatrix[7, 2, x]
Out[52]= {{x[1,1]-x[2,1],x[1,2]-x[2,2],-x[1,1]+x[2,1],-x[1,2]+x[2,2],0,0},
          {x[1,1]-x[3,1],x[1,2]-x[3,2],0,0,-x[1,1]+x[3,1],-x[1,2]+x[3,2]},
          {0,0,x[2,1]-x[3,1],x[2,2]-x[3,2],-x[2,1]+x[3,1],-x[2,2]+x[3,2]}}
```

4.2 Rigidity

For frameworks, i. e. graphs with realizations we can check infinitesimal rigidity.

Theorem 4.1.
A framework $(G = (V, E), \rho)$ is infinitesimally rigid if and only if the rank of the rigidity matrix fulfills $\operatorname{rank}(\mathcal{R}(G, \rho)) = 2|V| - 3$.

`InfinitesimallyRigidFrameworkQ` does check exactly this. The triangle graph with the realization from above is indeed infinitesimally rigid as a framework, whereas if we place all vertices on a line we get an infinitesimally flexible framework.

Input

```
InfinitesimallyRigidFrameworkQ[g,p]
     g    graph (integer or Graph)
     p    coordinates for the vertices of g (list of lists)
```

Example

```
In[53]:= InfinitesimallyRigidFrameworkQ[7, {{1, 1}, {3, 2}, {2, 3}}]
Out[53]= True
In[54]:= InfinitesimallyRigidFrameworkQ[7, {{1, 1}, {3, 1}, {2, 1}}]
Out[54]= False
```

For graphs we can either pick a random realization hoping it is generic or a symbolic one.

Definition 4.2.
A graph is rigid if and only if there is a generic realization for which the framework is generically rigid.

Hence, picking a random realization we can check rigidity probabilistically. This will never return false positive answers but it might give false negative ones. These situations can be reduced by having a large enough range for the random numbers (compare [8]). For this reason `RigidGRaphQ` has the option `RandomRigidityMatrixRange` to change that range. The default is `Automatic` which turns into $10^6 \cdot |V| \cdot d$, where d is the dimension. By [8] the probability of a false negative answer is therefore less than 10^{-6}. The option can be set manually to any integer. It is easy to see that a rigid graph needs at least $d|V| - \binom{d+1}{2}$ edges. This is checked before doing any other check, but it can be turned off by setting `UseCount` to `False`. If the dimension is omitted than it is assumed to be two.

Input

```
RigidGRaphQ[g,d]
RigidGRaphQ[g]
     g graph (integer or Graph)
     d dimension (integer), if omitted d=2
```

Options

`UseCount`	checks number of edges first	`True`
`Method`	fixes whether a random or symbolic matrix is used	`"RandomRigidityMatrix"`
`RandomRigidityMatrixRange`	maximal value for random numbers	`Automatic`
`RandomSet`	set of which random numbers are taken from (integers, reals)	`"Reals"`

Example

```
In[55]:= RigidGraphQ[7,2]
Out[55]= True
In[56]:= RigidGRaphQ[5,2]
Out[56]= False
```

By default random numbers are taken to be real but can be set by the option `RandomSet` to `"Integers"`. Using this and setting the range very low, the following command does regularly give a wrong answer. Note, however, that this needs a lot of manual setup.

Example

```
In[57]:= RigidGraphQ[7, 2, RandomRigidityMatrixRange -> 1,
          UseCount -> False, RandomSet -> "Integers"]
```

Instead of checking the rank of a random rigidity matrix we can also use a symbolic one by setting `Method` to `"SymbolicRigidityMatrix"`. We then have a deterministic output. Note, however, that this needs much more computation time.

4.3 Minimal Rigidity

Checking minimal rigidity can be essentially done the same way as checking rigidity just that the edge count needs to be exactly $d|V| - \binom{d+1}{2}$. This count is checked automatically at the beginning unless the option `UseCount` is set to `False`. If the dimension in `MinRigidGraphQ` is omitted than it is assumed to be two.

Input

```
MinRigidGraphQ[g,d]
MinRigidGraphQ[g]
     g graph (integer or Graph)
     d dimension (integer), if omitted d=2
```

Options

`UseCount`	checks number of edges first	`True`
`Method`	see Table 1	`"RandomRigidityMatrix"`
`RandomRigidityMatrixRange`	maximal value for random numbers	`Automatic`
`RandomSet`	set of which random numbers are taken from (integers, reals)	`"Reals"`
`VertexLimit`	limit for size of input graph, used for some methods (true or integer)	`True`

Example

```
In[58]:= MinRigidGraphQ[7,2]
Out[58]= True
In[59]:= MinRigidGRaphQ[5,2]
Out[59]= False
```

The `Method` option is by default set to `"RandomRigidityMatrix"`. It can as well take `"SymbolicRigidityMatrix"`. These two methods work for arbitrary dimension, though the symbolic one might need quite some computational resources. For dimension two there are several other possibilities than the rigidity matrix to check minimal rigidity (see Table 1 for an overview and the following subsections for details). Some of them are clearly computationally too expensive, but are implemented for comparison.

method	dim	reference	quality
`"RandomRigidityMatrix"`	all	Section 4.2	probabilistic
`"SymbolicRigidityMatrix"`	all	Section 4.2	deterministic
`"Sequence"`	2	Section 4.3.1	deterministic
`"Subgraph"`	2	Section 4.3.2	deterministic
`"PebbleGame"`	2	Section 4.3.2	deterministic
`"TwoSpanningTrees"`	2	Section 4.3.3	deterministic
`"ThreeTrees"`	2	Section 4.3.3	deterministic
`"RealizationCount"`	all	Section 4.3.4	probabilistic

Table 1: Methods available for `MinRigidGraphQ`.

4.3.1 Extension Sequence

Theorem 4.3. ([13, 16])
A graph is minimally rigid in dimension two if and only if there is a sequence of 0- and 1-extensions that construct the given graph starting from a single edge.

This check can be used setting the `Method` option to `"Sequence"`.

Example

```
In[60]:= MinRigidGraphQ[7916, 2, Method -> "Sequence"]
Out[60]= True
```

4.3.2 Subgraphs and Pebble Game

Theorem 4.4. ([13, 16])
A graph $G = (V, E)$ is minimally rigid in dimension two if and only if $|E| = 2|V| - 3$ and for every induced subgraph $G' = (V', E')$ with at least two vertices we have $|E'| \leq 2|V'| - 3$.

This edge count can be checked directly by the option value `"Subgraph"` but it is computationally expensive. Much more efficient for this check is the pebble game algorithm for checking $(2, 3)$-tightness, which is exactly the above property. See Section 4.4 for more details on the general pebble game algorithm. Setting the option value of `Method` to `"PebbleGame"` applies this algorithm.

Example

```
In[61]:= MinRigidGraphQ[7916, 2, Method -> "Subgraph"]
Out[61]= True
In[62]:= MinRigidGraphQ[7916, 2, Method -> "PebbleGame"]
Out[62]= True
```

4.3.3 Subtrees

Minimally rigid graphs can be classified also by subtree properties. For these cases we have just a more or less brute force implementation, which exists mainly for visualization purposes. Therefore, they are by far the slowest two options.

Theorem 4.5. ([15])
A connected graph $G = (V, E)$ is minimally rigid in dimension two if and only if for every edge $e = \{u, v\} \in E$ the multigraph obtained by adding a copy of e is the edge-disjoint union of two spanning trees.

We can search for such a spanning tree with `TwoSpanningTrees`, which takes as input a graph in `Graph` data type and an edge (as a list) an return all partitions into spanning trees with the chosen edge doubled. The output can be chosen by the option `OutputStyle` which by default is `"Data"` but can be set to `"Graph"`. Equivalent trees are ignored unless the option `RemoveEquivalent` is set to `False`.

Input

```
TwoSpanningTrees[g]
     g graph (integer or Graph)
```

Options

`OutputStyle`	defines format of output (data or graph)	`"Data"`
`RemoveEquivalent`	by construction the output might include equivalent items, this is turned off by default	`True`

Example

```
In[63]:= TwoSpanningTrees[7]
Out[63]= {{{{1,2},{1,3}},{{1,2},{2,3}}},
          {{{1,3},{1,2}},{{1,3},{2,3}}},
          {{{2,3},{1,2}},{{2,3},{1,3}}}}
```

The output is a set of pairs where each part of a pair is a set of edges forming a spanning tree. In the check for minimal rigidity the option `Method` is set to `"TwoSpanningTrees"`.

Example

```
In[64]:= MinRigidGraphQ[7916, 2, Method -> "TwoSpanningTrees"]
Out[64]= True
```

There is a further classification using three subtrees.

Theorem 4.6. ([5])
A connected graph G with more than one vertex is minimally rigid if and only if it is the disjoint union of three non-empty trees such that each vertex is part of exactly two of the trees and there are no vertex-induced subtrees on at least two vertices with the same vertex set.

We can search for such trees using `ThreeTrees`. The options are the same as for `TwoSpanningTrees`.

Input

```
ThreeTrees[g]
     g graph (integer or Graph)
```

Options

`OutputStyle`	defines format of output (data or graph)	`"Data"`
`RemoveEquivalent`	by construction the output might include equivalent items, this is turned off by default	`True`

Example

```
In[65]:= ThreeTrees[7]
Out[65]= {{{},{{1,2}},{{1,3},{2,3}}},{{},{{1,3}},
          {{1,2},{2,3}}},{{},{{2,3}},{{1,2},{1,3}}},
          {{{1,2}},{{1,3}},{{2,3}}}}
```

The output is a list of triples where each part of a triple is a set of edges forming a tree. Note that there is an empty set on some of the output but this means that there is no edge in that subtree; it consists of a single vertex. In the check for minimal rigidity the option `Method` is set to `"ThreeTrees"`.

Example

```
In[66]:= MinRigidGraphQ[7916, 2, Method -> "ThreeTrees"]
Out[66]= True
```

4.3.4 Realization Count

Minimally rigid graphs are those that for generic edge lengths have positive but finitely many realizations in the plane. We can therefore theoretically use realization counting for testing minimal rigidity. Due to the computation time this is not recommended but exists for comparison purposes. Here the option value of `Method` is set to `"RealizationCount"`.

Example

```
In[67]:= MinRigidGraphQ[7916, 2, Method -> "RealizationCount"]
Out[67]= True
```

Note that by default this would only start computing with less than 10 vertices. If you want to test larger graphs increase the limit by setting `VertexLimit` to the required integer.

4.4 Tightness and Sparsity

Tightness and sparsity play a role in rigidity but are a more general concept.

Definition 4.7.
A graph $G = (V, E)$ is (k, ℓ)-sparse for every subgraph $G' = (V', E')$ with more than k vertices fulfills $|E'| \leq k|V'| - \ell$. The graph is (k, ℓ)-tight if it is sparse and $|E| = k|V| - \ell$.

This can be checked by `SparseGraphQ` and `TightGraphQ` respectively. For integers k, ℓ with $\ell \leq 2k$ this can be done via the pebble game algorithm (compare [14]). By default the option `Method` is set to `"PebbleGame"`. If $\ell > k$ an error message is returned. Setting the option to `"Subgraph"` the property is checked directly.

Input

```
SparseGraphQ[g,k,l]
SparseGraphQ[g,k]
TightGraphQ[g,k,l]
TightGraphQ[g,k]
      g    graph (integer or Graph)
      k    parameter (integer)
      l    parameter (integer), optional
```

Example

```
In[68]:= SparseGraphQ[5, 2, 3]
Out[68]= True
In[69]:= TightGraphQ[5, 2, 3]
Out[69]= False
```

As a short hand notation `SparseGraphQ` and `TightGraphQ` also work with just k as input. In that case ℓ is set automatically to $\binom{k+1}{2}$.

Example

```
In[70]:= SparseGraphQ[5, 2]
Out[70]= True
In[71]:= TightGraphQ[5, 2]
Out[71]= False
```

We can also see the well known fact that there are $(3, 6)$-tight graphs that are not minimally rigid in dimension three, for instance the double banana graph represented by the integer 134210055.

Example

```
In[72]:= TightGraphQ[134210055, 3, Method -> "Subgraph"]
Out[72]= True
In[73]:= MinRigidGraphQ[134210055, 3]
Out[73]= False
```

4.5 k-Redundancy

Graphs that are not minimally rigid but still rigid might have redundant edges.

Definition 4.8.
A graph is k-redundantly rigid in dimension d if after removing any k edges it is still rigid in dimension d.

A graph is minimally k-redundantly rigid in dimension d if it is k-redundantly rigid in dimension d and no proper spanning subgraph is k-redundantly rigid in dimension d.

Sometimes we omit k, if it is equal to one. We can check redundancy using the function `KRedundantlyRigidGraphQ`.

Input

```
KRedundantlyRigidGraphQ[g,k,d]
RedundantlyRigidGraphQ[g,d]
RedundantlyRigidGraphQ[g]
MinimallyKRedundantlyRigidGraphQ[g,k,d]
```

```
      g    graph (integer or Graph)
      k    parameter for redundancy (integer)
      d    dimension (integer)
```

Options

`CheckInputRigidity`	checks whether input graph is rigid, if the graph is known to be rigid this can be omitted	`True`
`UseCount`	passed on to rigidity check	`True`
`Method`	passed on to rigidity check	`"RandomRigidityMatrix"`
`RandomRigidityMatrixRange`	passed on to rigidity check	`Automatic`
`RandomSet`	passed on to rigidity check	`"Reals"`

By default rigidity is tested with random rigidity matrices. Again a symbolic version is available via setting the method to `"SymbolicRigidityMatrix"`.

Example

```
In[74]:= KRedundantlyRigidGraphQ[7675, 1, 2]
Out[74]= True
In[75]:= KRedundantlyRigidGraphQ[7675, 2, 2]
Out[75]= False
```

4.6 k-Vertex-Redundancy

Similarly to edge redundancy we might also check for redundant vertices.

Definition 4.9.
A graph is k-vertex-redundantly rigid *in dimension d if after removing any k vertices it is still rigid in dimension d.*

A graph is minimally k-vertex-redundantly rigid *in dimension d if it is k-vertex-redundantly rigid in dimension d and no proper spanning subgraph is k-vertex-redundantly rigid in dimension d.*

When $k = 1$ we might omit the k. We can check vertex-redundancy using the function `KVertexRedundantlyRigidGraphQ`.

Input

```
KVertexRedundantlyRigidGraphQ[g,k,d]
VertexRedundantlyRigidGraphQ[g,d]
VertexRedundantlyRigidGraphQ[g]
MinimallyKVertexRedundantlyRigidGraphQ[g,k,d]
      g    graph (integer or Graph)
      k    parameter for redundancy (integer)
      d    dimension (integer)
```

Options

`CheckInputRigidity`	checks whether input graph is rigid, if the graph is known to be rigid this can be omitted	`True`
`UseCount`	passed on to rigidity check	`True`
`Method`	passed on to rigidity check	`"RandomRigidityMatrix"`
`RandomRigidityMatrixRange`	passed on to rigidity check	`Automatic`
`RandomSet`	passed on to rigidity check	`"Reals"`

By default rigidity is tested with random rigidity matrices. Again a symbolic version is available via setting the method to `"SymbolicRigidityMatrix"`.

Example

```
In[76]:= KVertexRedundantlyRigidGraphQ[7679, 1, 2]
Out[76]= True
In[77]:= KVertexRedundantlyRigidGraphQ[7679, 2, 2]
Out[77]= False
```

4.7 Global Rigidity

Definition 4.10.
A rigid graph is globally rigid *if all generic frameworks are are globally rigid. A framework is* globally rigid *if every equivalent framework is congruent.*

A graph is minimally globally rigid *if it is globally rigid and after removal of any edge it is not globally rigid any more.*

While this definition is rather complicated to check directly, we use instead the algorithm described in [8] for checking global rigidity with `GloballyRigidGraphQ`.

Input

```
GloballyRigidGraphQ[g,d]
GloballyRigidGraphQ[g]
      g    graph (integer or Graph)
      d    dimension (integer)
```

For dimension two a global rigidity can also be checked via redundancy and vertex connectivity.

Theorem 4.11. ([12])
A graph is globally rigid in dimension two if and only if it is a complete graph on at most three vertices or it is redundantly rigid in dimension two and 3-connected.

This check can be done setting the option `Method` to `"ConnectivityAndRedundancy"`.

Options

`UseCount`	passed on to rigidity check	`True`
`Method`	method to be used (random or symbolic stress)	`"RandomStress"`
`RMethod`	method to be used for checking redundancy	`"RandomRigidityMatrix"`
`RandomRigidityMatrixRange`	passed on to rigidity matrix	`Automatic`

Example

```
In[78]:= GloballyRigidGraphQ[16351, 3]
Out[78]= True
In[79]:= GloballyRigidGraphQ[16351, 4]
Out[79]= False
```

Similarly to rigid graphs we can also here ask for minimality (in terms of edges) and redundancy (in terms of edges or vertices). We therefore have the following commands.

Input

```
MinimallyGloballyRigidGraphQ[g,d]
KRedundantlyGloballyRigidGraphQ[g,k,d]
MinimallyKRedundantlyGloballyRigidGraphQ[g,k,d]
RedundantlyGloballyRigidGraphQ[g,d]
RedundantlyGloballyRigidGraphQ[g]
KVertexRedundantlyGloballyRigidGraphQ[g,k,d]
MinimallyKVertexRedundantlyGloballyRigidGraphQ[g,k,d]
VertexRedundantlyGloballyRigidGraphQ[g,d]
VertexRedundantlyGloballyRigidGraphQ[g]
     g    graph (integer or Graph)
     k    parameter for redundancy (integer)
     d    dimension (integer)
```

The options are the same as for `GloballyRigidGraphQ`.

4.8 Dependence and Circuits

The rank of the rigidity matrix is not only relevant for rigidity but also for the following property.

Definition 4.12.
A Graph $G = (V, E)$ is d-independent if the rank of the rigidity matrix $\mathcal{R}_d(G, \rho)$ for generic ρ is equal to $|E|$. Otherwise it is called d-dependent. The graph is a d-circuit if it is not d-independent but after removing any edge it is.

These properties can be checked via `IndependentGraphQ`, and `CircuitQ`, respectively.

Input

```
IndependentGraphQ[g,d]
DependentGraphQ[g,d]
CircuitQ[g,d]
     g    graph (integer or Graph)
     d    dimension (integer)
```

The options refer to whether a symbolic or a random matrix is chosen

Options

`Method`	choses a random or a symbol matrix for the check	`"RandomRigidityMatrix"`
`RandomRigidityMatrixRange`	defines the maximum value in a random matrix	`Automatic`

The three prism graph is minimally rigid and hence, it is 2-independent but not a 2-circuit.

Example

```
In[80]:= IndependentGraphQ[7916, 2]
Out[80]= True
In[81]:= CircuitQ[7916, 2]
Out[81]= False
```

4.9 Rigid Subgraph Free

Sometimes it is interesting to see whether a minimally rigid graph has a non-trivial minimally rigid subgraph, i. e. a subgraph with at least d vertices except for K_d. It is clear that such a graph cannot have a vertex of degree d. This property can be checked with the function `RigidSubgraphFreeMinRigidGraphQ`.

Input

```
RigidSubgraphFreeMinRigidGraphQ[g,d]
     g    graph (integer or Graph)
     d    dimension (integer)
```

It passes on all options that are used for checking minimal rigidity and it has some options on its own.

Options

`IgnoreSmallSubgraphs`	by default we ignore subgraphs with less than $d+1$ vertices, i. e. we ignore small subgraphs; instead we can ask for being stricter and search for graphs that do not have any minimally rigid subgraphs with at least three vertices	`True`
`CheckMinRigidity`	checks whether the input is minimally rigid	`True`

The complete bipartite graph $K_{3,3}$ is for instance free of rigid subgraphs.

Example

```
In[82]:= RigidSubgraphFreeMinRigidGraphQ[7672, 2]
Out[82]= True
```

In dimension three there are graphs that do not contain a minimally rigid subgraph, not even a triangle subgraph.

Example

```
In[83]:= RigidSubgraphFreeMinRigidGraphQ[1034850648000, 3,
          IgnoreSmallSubgraphs -> False]
Out[83]= True
```

5 Realization Counting

Counting complex realizations in the plane and on the sphere, can be done by combinatorial algorithms. Furthermore, a Gröbner basis approach can be used with random edge lengths which therefore gives a probabilistic answer. This approach can be theoretically used for higher dimensions but is rather restricted by computation power.

Realizations are counted up to isometries, but there is one reflection that remains. This means the triangle has two realizations, where one is the reflection of the other.

5.1 Space

The main command for counting realizations in a complex space, in particular the complex plane, is `ComplexRealizationCount`. The implementation is based on [1] but generalized and unified.

Input

```
ComplexRealizationCount[g,d]
ComplexRealizationCount[g]
     g    graph (integer or Graph)
     d    dimension (integer)
```

The main option decides whether a combinatorial [2] or a Gröbner (compare [1]) basis algorithm is used. In both cases the graph can be simplified (default). Note that there is also a `C++` implementation of the combinatorial algorithm [1].

Options

`Method`	the method of counting can be combinatorial (only dimension two) or by a Gröbner basis (general), automatically the combinatorial is chosen, if available, but the other can be enforced	`Automatic`
`RemoveDegD`	removes vertices of degree d recursively because they account for a factor of two in the number of realizations, only afterwards applies the chosen method	`True`

Then the triangle graph yields the two realizations we have mentioned earlier. In fact we only count them.

Example

```
In[84]:= ComplexRealizationCount[7, 2]
Out[84]= 2
```

The three-prism graph is the smallest graph where the number of complex realizations in the plane is not a power of two.

Example

```
In[85]:= ComplexRealizationCount[7916, 2]
Out[85]= 24
In[86]:= ComplexRealizationCount[7916, 2, Method -> "Groebner"]
Out[86]= 24
```

For higher dimensions the Gröbner basis approach is used automatically, so it gets computationally expensive rather soon.

Example

```
In[87]:= ComplexRealizationCount[16350, 3]
Out[87]= 16
```

5.2 Sphere

The main command for counting realizations on a complex sphere, in particular the 2-sphere, is `ComplexRealizationCountSphere`.

Input

```
ComplexRealizationCountSphere[g,d]
ComplexRealizationCountSphere[g]
     g    graph (integer or Graph)
```

```
    d    dimension (integer)
```

The main option decides whether a combinatorial [6] or a Gröbner basis algorithm is used. In both cases the graph can be simplified (default). Note that there is also a `Python` and `Cython` implementation of the combinatorial algorithm [7].

Options

`Method`	the method of counting can be combinatorial (only dimension two) or by a Gröbner basis (general), automatically the combinatorial is chosen, if available, but the other can be enforced	`Automatic`
`RemoveDegD`	removes vertices of degree d recursively because they account for a factor of two in the number of realizations, only afterwards applies the chosen method	`True`

Then the triangle graph yields the two realizations we have mentioned earlier. In fact we only count them.

Example

```
In[88]:= ComplexRealizationCountSphere[7, 2]
Out[88]= 2
```

The graph with integer representation 481867 is the smallest graph where the number of complex realizations on the sphere is not a power of two. There are other such graphs with the same number of vertices but they would have a degree two vertex.

Example

```
In[89]:= ComplexRealizationCountSphere[7916, 2]
Out[89]= 32
In[90]:= ComplexRealizationCountSphere[481867, 2]
Out[90]= 48
```

For higher dimensions the Gröbner basis approach is used automatically, so it gets computationally expensive rather soon. In the example we get the same result as on the plane but this is in general not the case.

Example

```
In[91]:= ComplexRealizationCountSphere[16350, 3]
Out[91]= 16
```

Index

References

[1] Jose Capco, Matteo Gallet, Georg Grasegger, Christoph Koutschan, Niels Lubbes, and Josef Schicho. An algorithm for computing the number of realizations of a Laman graph. Zenodo, 2018. doi:10.5281/zenodo.1245506.

[2] Jose Capco, Matteo Gallet, Georg Grasegger, Christoph Koutschan, Niels Lubbes, and Josef Schicho. The number of realizations of a laman graph. *SIAM Journal on Applied Algebra and Geometry*, 2(1):94–125, 2018. doi:10.1137/17M1118312.

[3] Jose Capco, Matteo Gallet, Georg Grasegger, Christoph Koutschan, Niels Lubbes, and Josef Schicho. The number of realizations of all Laman graphs with at most 12 vertices. Zenodo, 2018. doi:10.5281/zenodo.1245517.

[4] Robert Connelly and Simon D. Guest. *Frameworks, Tensegrities and Symmetry*. Cambridge University Press, 2022. doi:10.1017/9780511843297.

[5] Henry Crapo. On the generic rigidity of plane frameworks. INRIA, RR-1278, HAL:inria-00075281, 1990. URL: https://hal.inria.fr/inria-00075281.

[6] Matteo Gallet, Georg Grasegger, and Josef Schicho. Counting realizations of laman graphs on the sphere. *Electronic Journal of Combinatorics*, 27(2):P2.5 (1–18), 2020. doi:10.37236/8548.

[7] Matteo Gallet, Georg Grasegger, and Josef Schicho. Software for counting realizations of minimally rigid graphs on the sphere. Zenodo, 2022. doi:10.5281/zenodo.6810642.

[8] Steven J. Gortler, Alexander D. Healy, and Dylan P. Thurston. Characterizing generic global rigidity. *American Journal of Mathematics*, 132(4):897–939, 2010. doi:10.1353/ajm.0.0132.

[9] Georg Grasegger, Christoph Koutschan, and Elias Tsigaridas. Lower Bounds on the Number of Realizations of Rigid Graphs. *Experimental Mathematics*, 29(2):125–136, 2020. doi:10.1080/10586458.2018.1437851.

[10] Jack Graver, Brigitte Servatius, and Herman Servatius. *Combinatorial rigidity*. American Mathematical Society, Providence, RI, 1993. doi:10.1090/gsm/002.

[11] Wolfram Research, Inc. Mathematica, Version 12. Champaign, IL, 2022. URL: https://www.wolfram.com/mathematica.

[12] Bill Jackson and Tibor Jordán. Connected rigidity matroids and unique realizations of graphs. *Journal of Combinatorial Theory, Series B*, 94(1):1–29, 2005. doi:10.1016/j.jctb.2004.11.002.

[13] Gerard Laman. On graphs and rigidity of plane skeletal structures. *Journal of Engineering Mathematics*, 4:331–340, 1970. doi:10.1007/BF01534980.

[14] Audrey Lee and Ileana Streinu. Pebble game algorithms and sparse graphs. *Discrete Mathematics*, 308(8):1425–1437, 2008. doi:10.1016/j.disc.2007.07.104.

[15] László Lovász and Yechiam Yemini. On Generic Rigidity in the Plane. *SIAM Journal on Algebraic and Discrete Methods*, 3:91–98, 1982. doi:10.1137/0603009.

[16] Hilda Pollaczek-Geiringer. Über die Gliederung ebener Fachwerke. *Zeitschrift für Angewandte Mathematik und Mechanik*, 7(1):58–72, 1927. doi:10.1002/zamm.19270070107.

[17] Meera Sitharam, Audrey St. John, and Jessica Sidman, editors. *Handbook of Geometric Constraint Systems Principles*. CRC Press, Boca Raton, 2018. doi:10.1201/9781315121116.