

EÖTVÖS LORÁND UNIVERSITY
FACULTY OF SCIENCE

PEBBLE GAME ALGORITHMS
AND THEIR IMPLEMENTATIONS

OTDK

LÓRÁNT MATÚZ
APPLIED MATHEMATICS MSc

SUPERVISOR:
PÉTER MADARASI
DEPARTMENT OF OPERATIONS RESEARCH



BUDAPEST
2023

Abstract

A multigraph $G = (V, E)$ is called (k, l) -sparse if every subset $X \subseteq V$ induces at most $\max\{k|X| - l, 0\}$ edges. Finding a largest (k, l) -sparse subgraph is a well-studied, polynomial-time-solvable problem, which is widely used in rigidity applications and serves as the basis of several combinatorial algorithms.

We present a new implementation and compare it with the library called KINARI-web on a wide range of random and real-world datasets. The computational study shows that the new implementation is consistently faster by one order of magnitude. Furthermore, we propose several heuristics to fine-tune the free parameters of the algorithm and investigate their practical efficiency.

We also implement an algorithm to find k arc-disjoint r -arborescences in a digraph and k edge-disjoint spanning trees in an undirected graph, which corresponds to the case $l = k$. Our implementation is available at lemon.cs.elte.hu/repos/sparseGraphs, and it is proposed to be part of the LEMON library.

We also introduce (j, k, l) -sparsity as a natural generalization. We prove that it is NP-hard when j is part of the input, and give an improved algorithm for testing $(3, k, 2k)$ -sparsity, which is a crucial tool for 3D rigidity applications.

1 Introduction

An undirected multigraph $G = (V, E)$ is called (k, l) -sparse if every subset of the vertices induces at most $\max\{0, k|X| - l\}$ edges. Essentially, this means that there are only a limited number of edges induced in each subset of the vertices, in other words, the graph is “uniformly sparse”. Testing sparsity and finding an inclusion-wise maximal sparse subgraph are widely-used tools in rigidity applications. The concept of sparsity also often occurs in combinatorial optimization, for example, a graph is (k, k) -sparse if and only if its edge set can be partitioned into k forests. Therefore, efficient algorithms for testing sparsity and their implementations are crucial.

Historical overview

The definition of (k, l) -sparse graphs was introduced in 1979 by Loréa [1] as an example of matroidal families. They have been studied intensively in the last decade, and it soon became apparent that they have a variety of applications. For example, (k, k) -tight graphs also appeared in the classical results of Nash-Williams [2] and Tutte [3] as the characterization of the graphs that can be decomposed into k edge-disjoint spanning trees. Later, Laman [4] showed that $(2, 3)$ -tight graphs are the generic minimally rigid graphs for bar-and-joint frameworks in the plane, and $(2, 3)$ -spanning graphs are the rigid ones. Recski [5] analyzed rigidity theory in the plane, and Jacobs [6] considered the special case of 3D for the practical application of protein flexibility.

Previous work

The family of the celebrated Pebble Game algorithms [7, 8] can find a largest (k, l) -sparse subgraph in $O(|V||E|)$ time. These algorithms play a crucial role in rigidity applications and serve as the basis of several combinatorial algorithms. For graphs decomposable into disjoint unions of spanning trees and pseudo-forests, corresponding to the range $0 \leq l \leq k$ of (k, l) -sparse graphs, we remark that there are $O(n\sqrt{n \log n})$ algorithms due to Gabow and Westermann [9], later improved to $O(n\sqrt{n})$ by Daescu and Kurdia [10]. We also note that there is an $O(n \log^3 n)$ algorithm due to Rollin, Schlipf and Schulz [11] for planar Laman

graphs, that is, $(2, 3)$ -tight graphs. However, no better algorithms than the Pebble Game are known for the entire range of (k, l) -sparse graphs. To the best of our knowledge, the only (non-open-source) implementation aimed to recognize (k, l) -sparse graphs is KINARI-web [12], due to Fox, Jagodzinski, Yang and Streinu. Another related implementation and algorithm are due to Király and Mihálykó [13, 14], which makes a (k, l) -tight (hyper)graph (k, l) -redundant.

Our results

We introduce (j, k, l) -sparsity as a direct generalization of (k, l) -sparsity, which excludes violating vertex sets of size smaller than j . We prove that it is NP-hard when j is part of the input, and give an improved algorithm for testing $(3, k, 2k)$ -sparsity, which is a crucial tool for 3D rigidity applications [15, 16, 17]. Moreover, we present a new implementation of all versions of the Pebble Game and compare them with the library called KINARI-web [12] on a wide range of random and real-world molecular and protein graphs. The computational study shows that our implementation of the Component Pebble Game is consistently faster by an order of magnitude. Furthermore, we propose several heuristics to fine-tune the free parameters of the Basic Pebble algorithm and investigate their practical efficiency. We also implement an algorithm to find k arc-disjoint r -arborescences in a digraph and k spanning trees in an undirected graph, which corresponds to the case $l = k$. In addition, we implement an algorithm to cover the arc set of a digraph with k arc-disjoint branchings and to cover the edge set of a graph with k edge-disjoint forests [18]. The algorithm uses the Pebble Game algorithm to determine a proper orientation of the graph then we try to find k arc-disjoint arborescences. Our implementation is available at lemon.cs.elte.hu/repos/sparseGraphs [19], and it is proposed to be part of the LEMON library [20, 21].

In the rest of this section, we show the basic definitions of sparsity. Then, Section 2 gives an overview of the Pebble Game algorithms. In Section 3, we propose a new generalization and improve the best-known algorithm for a special case. After that, we give an overview of our efficient implementation in Section 4, and then we compare it with another implementation, called KINARI. Finally, we introduce and benchmark some heuristics to improve the running time of the Basic Pebble Game.

1.1 Definitions

The following principal definitions are needed throughout this paper.

Definition 1.1

A multigraph $G = (V, E)$ is called (k, l) -**sparse** if any subset X of vertices induces at most $\max\{0, k|X| - l\}$ edges. Furthermore, if G is (k, l) -sparse and it has exactly $k|V| - l$ edges, then it is called (k, l) -**tight**. Last but not least, G is (k, l) -**spanning** if it contains a (k, l) -tight subgraph that spans the entire vertex set V .

In most cases, parameters k and l are discarded if they are clear from the context. Now we show another definition, which plays a crucial role in speeding up the Basic Pebble Game.

Definition 1.2

A **component** is an inclusion-wise maximal induced proper subgraph G' of a (k, l) -sparse graph which induces exactly $k|V(G')| - l$ edges.

1.2 Problem statement

In this paper, we focus on the following problems.

Solved problems

1. **Decision:** Decide if G is sparse, tight, spanning, or none.

2. **Extraction:** Extract an inclusion-wise maximal sparse subgraph from G .
3. **Optimization:** Extract a maximum weight sparse subgraph from G with respect to a non-negative weight function on the edges.
4. **Components:** Find all components of G .

1.3 Properties of sparse graphs

In this section, we introduce the main properties of the (k, l) -sparse graphs.

The parameters of (k, l) -sparsity should be diminished to $k > 0$ and $0 \leq l < 2k$. To verify that the problems mentioned above are only interesting in this range, let us present some claims, which easily follow from the definition of sparsity.

Lemma 1.3

1. If $l \geq 2k$, then each sparse graph is empty.
2. If $l < 0$, then the union of two vertex-disjoint sparse graphs may not be sparse.

Let us further divide the basic range of $0 \leq l < 2k$ into a **lower range** for $0 \leq l \leq k$ and an **upper range** for $k < l < 2k$. It soon turns out that the sparse graphs and their components have significantly different properties within these two ranges.

Lemma 1.4

The components of a sparse graph are edge-disjoint. Furthermore, they are vertex-disjoint in the lower range and may overlap in at most one vertex in the upper range.

1.4 Sparsity matroid

Another significant result claims that there exists a matroid over the (k, l) -sparse graphs. Since the matroid property holds, the Basic Pebble Game may work greedily, which provides a method for processing the edges in a sorted order to get a maximum weight sparse subgraph.

Theorem 1.5 (Sparsity matroid)

Assume that k, l and n are such that

- $l \in [0, k]$ and $n \geq 1$, or
- $l \in (k, \frac{3}{2}k)$ and $n \geq 2$, or
- $l \in [\frac{3}{2}k, 2k)$ and $n = 2$ or $n \geq \frac{l}{2k-l}$.

Then the collection of the edge sets of all (k, l) -tight graphs on n vertices is the set of bases of a matroid whose ground set is the edge set of $K_n^{k-l, 2k-l}$, that is, the complete graph on n vertices with at most $k-l$ loops and $2k-l$ parallel edges.

2 Overview of Pebble Game algorithms

Pebble Game algorithms were introduced in [7] to find a largest (k, l) -sparse subgraph in polynomial time. They are widely used in rigidity applications [17] and also serve as the basis of several combinatorial algorithms.

Now we summarize the celebrated Pebble Game algorithms [7, 8]. The main tool used by these algorithms is the orientation lemma:

Lemma 2.1 (Orientation lemma)

Let $G = (V, E)$ be an undirected graph, $g : V \rightarrow \mathbb{Z}_+$ an upper-bound function. Then G has an orientation with an out-degree function satisfying $\text{out}(v) \leq g(v)$ for every vertex v if and only if $\sum_{v \in X} g(v) \geq i_G(X)$ holds for every non-empty subset $X \subseteq V$.

2.1 Basic Pebble Game

The algorithm takes an undirected multigraph G and the parameters k and l . It determines a solution for the first three problems: the decision, the extraction, and the optimization.

More precisely, the algorithm processes the edges of G in an arbitrary order one by one, that is, it either accepts or rejects them keeping the properties of the orientation lemma during the whole execution. To check the conditions of the orientation lemma, the algorithm constructs an inner digraph D of the already accepted edges, such that the underlying undirected graph remains sparse.

Note that the free choice of the edge order allows us to process the edges by weights to get a solution for the optimization problem. The optimality of the greedy algorithm for both the extraction and the optimization problem is guaranteed by the sparsity matroid, see Theorem 1.5.

The algorithm uses the concept of pebbles, which are placed for each vertex v and are correlated with the number of further potentially insertable arcs leaving v . Now we present the rules, the enabled moves, and the output evaluation of the algorithm.

Rules of the algorithm

- **Initialization:** Let the inner digraph D be an empty graph on the vertex set V of the input graph, and place k pebbles on each vertex.
- **Pebbles:** At most k pebbles may be present on each vertex at any time. Denote the number of pebbles on a vertex v by $\text{peb}(v)$.
- **Edge acceptance:** An edge uv is accepted for insertion if more than l pebbles are present on the vertices u and v in total. In the case of a loop, count the pebbles of the endpoint only once.

Enabled moves

- **Pebble collection:** If an edge uv is not yet accepted, then an additional pebble may be collected on one of its endpoints u and v via breadth-first search (BFS) in D . It is performed from u and v by marking them visited, so they are not searched, and their pebbles are protected from being moved. If a pebble is found on a reachable vertex w (say, from v), then every arc in the directed path from v to w is reversed, and a pebble is transferred from w to v .
- **Edge insertion:** If an edge uv is accepted, then there is at least one pebble on at least one of the endpoints u and v (say, u). Insert arc uv into D and remove a pebble from u .

Output evaluation

If all edges are processed, the output may be as follows, depending on the final state of the game. If exactly l pebbles remained in the final state, and every edge has been accepted, then it is tight; otherwise, if some edges have been rejected, then it is spanning. Otherwise, if more than l pebbles remained, and every edge has been accepted, then the output is sparse; otherwise, if some edges have been rejected, then it is neither sparse nor spanning.

Note that the description of the algorithm may be modified by working with the number of outgoing arcs instead of pebbles since $\text{out}(v) + \text{peb}(v) = k$ holds for each vertex v during the whole execution, where out denotes the number of outgoing arcs from v in D .

The description of the algorithm is the following.

Algorithm 1: Basic Pebble Game**Input:** An undirected multigraph $G = (V, E)$, and integers $k > 0$, $0 \leq l < 2k$.**Output:** Sparse, spanning, tight, or none.**Method:** Initialize the game and start processing the edges one by one in arbitrary order by checking the condition of edge acceptance. Let $e = uv$ denote the edge being processed.

- If the condition of edge acceptance holds for e , then it is accepted and inserted to D .
- Otherwise, attempt a pebble collection for the endpoints u and v .
 - If it fails to collect a new pebble, then e is rejected.
 - Otherwise, the total number of pebbles on the endpoints is increased, and the edge is processed again.

Complexity: Processing an edge e takes $O(lkn)$ time, since at most $l + 1$ pebble collections may be required in D to accept or reject e , which has at most $O(kn)$ arcs. Overall, the complexity is $O(klmn)$ for m edges.

2.2 Component Pebble Game

A high-level description of the Component Pebble Game relies on the observation that once a vertex set X becomes tight, the further induced edges would violate the sparseness of X , meaning that they can be rejected immediately. It also implies that these algorithms must be able to find and update the components, that is, the inclusion-wise maximal sparse subgraphs. First, we show some meaningful claims about the cardinality of the components.

Lemma 2.2

1. The number of components during the whole execution of the algorithms is $O(kn)$.
2. The total size of the intersections of the components is $O(n)$.

These claims will imply that the space complexity of the Component Pebble Game algorithms is linear in n . The following lemma claims our result on giving an upper bound on the total size of the vertex sets of the components.

Lemma 2.3

The total size of the vertex sets of the components during the whole execution of the algorithms is $O(kn)$.

Proof.

Denote the number of i -element components by C_i . The number of components consisting of a single vertex is at most n , because two components with the same vertex set may not exist, so $C_1 \leq n$. Observe that $i \leq 2(ki - l)$ is met for each $i \geq 2$, because the range of $l \leq 2k - 1$ implies that $2l \leq 2(2k - 1) \leq (2k - 1)i$, from where the observation follows. Now the multiplicity of the vertices in the components can be determined:

$$\sum_{i=1}^n iC_i = C_1 + \sum_{i=2}^n iC_i \leq n + 2 \sum_{i=2}^n (ki - l)C_i \leq n + 2(kn - l) = O(kn),$$

where the last inequality holds because each C_i induces exactly $ki - l$ edges, and each edge belongs to at most one component. $\square$

Now we present the algorithm to detect such a component after an edge insertion.

Algorithm 2: Component detection

Input: The inner digraph $D = (V, A)$, into which arc $e = uv$ has just been inserted.

Output: The vertex set X of the new component induced by e .

Method: If more than l pebbles are present on u and v , then e does not belong to any components, so return $\emptyset$. Otherwise, compute the reachable region $R \subseteq V$ from u and v in D .

- If any $w \in R \setminus \{u, v\}$ has at least one free pebble, then more than l pebbles may be gathered on u and v via some pebble collections from w , therefore return $\emptyset$.
- Otherwise, from each vertex $w \in V \setminus R$ with at least one free pebble, perform a BFS in the reversed digraph of D and return the set of non-visited vertices.

Complexity: Both the graph reversal and the BFS takes $O(kn)$ steps.

As stated before, some additional data structures are used for storing the components and indicating whether two vertices belong to the same component. Assume that the components are stored in a `list`, specifically a so-called dynamic array. Also, assume that each component is identified with an id that specifies their positions in the list. Now we introduce an algorithm for the unweighted case and show that the weighted case in $O(n^2)$ time is still an open problem.

2.2.1 The unweighted case

In this section, we introduce our version of the Unweighted Component Pebble Game, which stores the vertex sets of the components despite the edge sets in [8].

Since the order of edges may be chosen arbitrarily for the basic algorithm, the principal idea is to define both a specific edge order and a specific data structure to speed up the basic version in the unweighted case. For the data structure, let $\mathbf{x}_u$ be the characteristic vector of the union of the components containing the vertex u which is being processed. Furthermore, maintain an indicator vector on the edges, called `processed`, to mark the already processed edges.

Algorithm 3: Unweighted Component Pebble Game

Input: An undirected multigraph $G = (V, E)$, and integers $k > 0$, $0 \leq l < 2k$.

Output: Sparse, spanning, tight, or none.

Method: Play Algorithm 1 with the following modifications. When considering a new vertex u , initialize $\mathbf{x}_u$ and start processing each incident edge $e = uv$ of u that has not been `processed` yet.

- If $\mathbf{x}_u(v)$ is true or $u = v$ and $l \geq k$, then u and v belong to the same component, so the edge is rejected.
- Otherwise, start a pebble collection while the condition of the edge acceptance holds, then insert it. After that, detect new components by Algorithm 2, and update as follows. Delete all components with a detected vertex and for each detected vertex v set $\mathbf{x}_u(v)$ to true. Finally, add the detected vertices as a new component into `list`.

Complexity: All updates are performed $O(n^2)$ time and the edge insertions in $O(lkn)$ time. Algorithm 2 and the updates are accomplished in $O(kn)$ steps. For every accepted edge, this gives $O(k^2ln^2)$ steps, so the overall running time is $O(k^2ln^2 + m)$.

2.2.2 An observation of the weighted case

Reaching the complexity $O(n^2)$ is an open problem since we figured out that the attempt to achieve this in [8] is based on the following incorrect assumption. They use the so-called bounded union-pair find method, which requires that $|C_i \cap C_j| \leq 1$ holds for each $i \neq j$, which is referred to as the so-called bounded property. However, when we take the union of more than two components, this property may be violated.

The following figure illustrates an example of the problem:

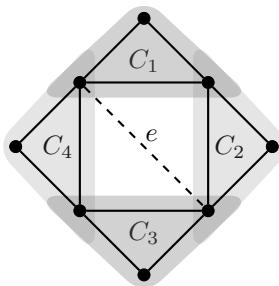


Figure 1: A $(2,3)$ -sparse graph in which the bounded union-pair find method fails.

Assume that the union method described in [8] is called in the order of the components indices. After creating the union of C_1 , C_2 and C_3 , the intersection of them and the remainder component C_4 consist of two vertices, which violates the bounded property. As a result, the given running time of $O(n^2)$ does not follow from the argument given in [8].

We also tried to come up with another approach to this problem that only stores the vertex sets of the components instead of their edges, but it also exceeded the running time $O(n^2)$.

3 | New generalizations and improving the case $l = 2k$

In this section, we propose a generalization of sparsity with a lower bound on the size of the investigated subsets, which we call (j, k, l) -sparsity. Then, we propose a better version of the already known algorithm for this particular case to decide the $(k, 2k)$ -sparsity. This algorithm is widely used in 3D rigidity [15, 16], since it is well-known that if a multigraph is rigid in 3D, then it is $(3,6)$ -tight. Therefore, it is essential to design efficient algorithms to check $(k, 2k)$ -sparsity as fast as possible.

3.1 Excluding small violating vertex subsets

Since the family of $(k, 2k)$ -sparse graphs contains only the empty graph as per the original definition, it is necessary to propose a modification. The major problem is that if the graph is non-empty, then the two-element vertex subsets are violated. Therefore, we restrict the cardinality of the violating vertex subsets by giving a lower bound on them as follows.

Definition 3.1

A multigraph G is (j, k, l) -**sparse** if any subset X of the vertices of size at least j induces at most $\max\{0, k|X| - l\}$ edges. In addition, if G is (j, k, l) -sparse and it has exactly $k|V| - l$ edges, then it is called (j, k, l) -**tight**. Last but not least, G is (j, k, l) -**spanning** if it contains a (j, k, l) -tight subgraph that spans the entire vertex set V .

Now we prove that solving the decision problem in the general case of (j, k, l) -sparsity is NP-hard.

Theorem 3.2

The identification of (j, k, l) -sparse graphs is NP-hard.

Proof.

It is well-known that the clique problem is NP-complete, as it is one of Richard M. Karp's original 21 problems [22], which asks if there exists a K -clique in a graph. Now we reduce the clique problem to the (j, k, l) -sparsity. It is easy to see that this problem remains hard when we assume that K is odd. The proof of the following lemma also verifies the theorem:

Lemma 3.3

Let $K > 1$ be an odd integer and $l > 0$ be an integer. Then a multigraph G is $(K, \frac{K-1}{2}, l)$ -sparse if and only if G does not contain a K -clique.

Proof.

First, we prove that if G contains a K -clique, then it is not sparse. Denote the vertex set of the clique by X . Then $|X| = K$, so X violates the sparsity because

$$\binom{K}{2} = \binom{|X|}{2} = i(X) > k|X| - l = \frac{K-1}{2}|X| - l = \binom{K}{2} - l.$$

For the other direction, let X be a vertex subset of V with at least K elements, that is, $|X| \geq K$. Then from sparsity

$$i(X) \leq k|X| - l = \frac{K-1}{2}|X| - l \leq \frac{|X|-1}{2}|X| - l = \binom{|X|}{2} - l < \binom{|X|}{2}$$

holds, so X cannot be a K -clique. □

We proved that finding a K -clique in a multigraph is equivalent to testing $(K, \frac{K-1}{2}, l)$ -sparsity, which completes the proof. □

To our best knowledge, finding an inclusion-wise maximal (j, k, l) -sparse subgraph in polynomial time is still an open problem.

3.2 Algorithms for the case $l = 2k$

In this section, we present the already known algorithm to decide the $(3, k, 2k)$ -sparsity which runs in time $O((k+n)knm)$, then we show our results on this topic in time $O(k^2nm)$. We note that the improved version is also proposed to be part of the LEMON library.

The following lemma justifies both Algorithm 4 and Algorithm 5, which are discussed later.

Lemma 3.4

Assume that edge $e = uv$ is not yet accepted and no more pebbles can be collected on u and v . Let X be the reachable region from u and v .

- (a) *If there exists an additional vertex $w \in X$ different from u and v , then X induces a tight subgraph.*
- (b) *Otherwise, if X only contains u and v , and in the BFS each vertex $w \in V \setminus \{u, v\}$ is reached in the reversed digraph from all the sources containing at least one pebble, then e is insertable.*

Proof.

- (a) By assumption, e is not accepted, so $\text{peb}(u) + \text{peb}(v) \leq l = 2k$. Each vertex $w \in X \setminus \{u, v\}$ can be reached from u or v , therefore, $\text{peb}(w) = 0$; otherwise the pebbles of w could have been collected on u and v , so $\text{peb}(X) \leq 2k$. As X has at least $j = 3$ vertices,

$$\text{peb}(X) + i(X) + \text{out}(X) = k|X|$$

holds. Since X is a reachable region, no arcs can leave it, meaning $\text{out}(X) = 0$. Therefore,

$$i(X) = k|X| - \text{peb}(X) \geq k|X| - 2k = k|X| - l$$

holds, from where tightness follows by using the sparsity of X .

- (b) Let $X' := X \cup \{w\}$ be a subset with an arbitrary vertex $w \in V \setminus \{u, v\}$. Since each w is reached by assumption, they may have an additional pebble. It follows that there exists a reorientation of the graph in which each subset X' has exactly $l + 1$ pebbles, therefore, e is insertable. □

3.2.1 The best-known version

This algorithm largely hinges on the Basic Pebble Game, however, according to the rules of that algorithm, it is impossible to gather enough pebbles to accept an edge. Hence, some additional moves are taken to replace a further pebble collection. It is achieved by checking the pebbles of the vertices in the current reachable region in this algorithm.

Algorithm 4: Decision Pebble Game for $l = 2k$

Input: An undirected simple graph $G = (V, E)$ on at least three vertices and an integer $k > 0$.

Output: Whether G is $(3, k, 2k)$ -sparse or not.

Method: Play Algorithm 1 with the following modifications. Assume that an edge $e = uv$ is neither accepted nor rejected, and the pebble collections have already gathered exactly the total of $l = 2k$ pebbles on the endpoints u and v . Denote the reachable region of u and v by $R \subseteq V$.

- If there exists a vertex in $R \setminus \{u, v\}$, then e is rejected by Lemma 3.4 (a).
- Otherwise, check if a pebble can be collected for each vertex $w \in V \setminus \{u, v\}$ as follows. Start a pebble collection for each vertex w to check if one can be gathered on them without reversing the current path or moving any pebbles.
 - If every collection succeeds, then e can be accepted by Lemma 3.4 (b).
 - Otherwise, $R \cup \{w\}$ forms a tight subgraph with a vertex $w \in V \setminus \{u, v\}$ with no pebbles, so e is rejected by Lemma 3.4 (a).

Complexity: Processing an edge takes at most $l + (n - 2)$ pebble collections in time $O(lkn + nkn) = O((k + n)kn)$. For all edges, it is $O((k + n)knm)$ steps, which is the same as $O(n^2m)$ if k is considered as a constant.

3.2.2 The improved version

Now we propose our improved version for recognizing the $(3, k, 2k)$ -sparse graph. It is based on the previous algorithm with the following modifications.

Observe that the actual pebble collections for each vertex in Algorithm 4 are not necessary to check whether we can collect a pebble for each vertex. The idea is that we should check

all vertices simultaneously. To achieve this, instead of n pebble collections, we execute merely one BFS from the vertices with at least one pebble in the *reversed* digraph and check if all vertices are reached.

Assume that the edge $e = uv$ is being processed. Note that u, v are always reached in the BFS, since both have k pebbles. Reaching a vertex w in the reversed graph from the vertices with at least one pebble means that none of the subsets of the vertices containing u, v and w becomes too dense when uv is inserted. If every vertex $w \in V \setminus \{u, v\}$ is reachable, then this holds for any subsets of the vertices containing u, v and w , hence uv is insertable.

Algorithm 5: Decision Pebble Game for $l = 2k$, improved version

Input: An undirected simple graph $G = (V, E)$ on at least three vertices and an integer $k > 0$.

Output: Whether G is $(3, k, 2k)$ -sparse or not.

Method: Play Algorithm 4 with the following modifications. If the reachable region R of u and v contains only u and v , then we check if a pebble can be gathered for each $w \in V \setminus \{u, v\}$ with a simple search: in the reversed digraph start a BFS from all vertices (different from the endpoints u and v) with at least one free pebble.

- If each vertex (different from u and v) is reached, then all vertices may have a pebble, and e can be accepted by Lemma 3.4 (b).
- Otherwise $\{u, v, w\}$ violates with a non-reached vertex w , therefore, e is rejected by Lemma 3.4 (a).

Complexity: Each edge requires at most l pebble collections and a simple BFS, so overall $O(k^2n)$ steps. For every edge, the algorithm runs in time $O(k^2nm)$, which is the same as $O(nm)$ if k is considered a constant.

4 | An efficient implementation and benchmarks

In this section, we give a detailed description of our implementations and an in-depth practical comparison with a previous implementation. Also, we present some of the tricks we used to speed up the Basic Pebble Game, and finally, we discuss different heuristics to fine-tune the order of edges in the Basic Pebble Game, which we can choose freely in the unweighted case.

4.1 The proposed algorithms

We implemented the following algorithms.

1. Basic Pebble Game, which was previously described as Algorithm 1. Moreover, this implementation includes some tricks, which are discussed later in Section 4.2.
2. Recognizing or extracting an inclusion-wise maximal $(k, 2k)$ -sparse graphs, which was discussed as Algorithm 5.
3. Unweighted Component Pebble Game, described as Algorithm 3. It is based on [8], however, our implementation stores the vertex sets of the components. Furthermore, since it derives from the Basic Pebble Game, it also contains our terminating condition, discussed in Section 4.2.
4. Finding k arc-disjoint r -arborescences and covering with k arc-disjoint branchings, which were described in [18, p. 904-928].

5. Finding k edge-disjoint spanning trees and covering with k edge-disjoint forests, which were described in [18, p. 904-928].

To the best of our knowledge, no open-source implementations have been available for any of these problems. Therefore, this implementation is the first efficient, open-source code of these algorithms.

4.2 Implementation details

In this section, we present various details of our implementations.

4.2.1 User-friendly environment

All of our algorithms provide loads of flexible options, which are configurable through a user-friendly interface. For example, different template parameters can be set including the graph type. In addition, all of the Pebble Game implementations provide execution control, that is, the algorithm can be run step-by-step. Lots of query functions make sure that all relevant information produced by the algorithms can be revealed in an easily understandable form.

4.2.2 Pebble Games

Now we show some interesting details of the Basic Pebble Game algorithm. Since the Component Pebble Game inherits most properties of it, the following also holds for the Component Pebble Game.

Terminating condition

A simple but essential observation is the following. The condition of edge acceptance is to have more than l pebbles on the endpoints, which was discussed in Section 2.1. Therefore, if the total number of pebbles drops to l , then each remaining edge is surely rejected. It can be really useful to speed up the algorithm, especially if the inclusion-wise maximal sparse subgraph is found quickly, and there are many unprocessed edges. That is why we attempted to find heuristics that determine the maximal sparse subgraph as soon as possible in Section 4.5.

Breadth-first search

Since the most time-consuming part of the Basic Pebble Game is the graph traversal algorithm, we made it as efficient as possible. We changed the complete implementation of BFS in the LEMON library in the following sense. The initializing of the BFS consists of clearing the queue and marking the vertices as non-visited. Whilst the original BFS of LEMON iterates through all vertices and marks them, our version iterates only on the elements of the queue. In addition, all unnecessary data structures and updates of the complete BFS are omitted to speed it up as much as possible. Furthermore, we implemented a more efficient function for the path searches, which are crucial because of loads of pebble collection moves.

4.2.3 Disjoint arborescences and covering with branchings

Finding k arc-disjoint r -arborescences and covering the arcs with k arc-disjoint branchings are well-studied problems, which often occur in combinatorial optimization, therefore their efficient implementations are also important. Roughly speaking, the arc-disjoint arborescences and branchings [18, p. 904-928] can be found iteratively one by one in such a way that each arborescence is constructed in $O(k^2 m^2)$ steps by adding arcs into it so that the remainder of the arc set also contains $k - 1$ arc-disjoint arborescences. For each arc, a network flow algorithm is performed using path reversals. The growing process of an arborescence stops if the whole vertex set is covered or no more arcs can be added to the arborescence preserving the above property of the remainder. In this presentation of the algorithm, the arcs of the original

digraph are removed as the current arborescence is grown. In our implementation, we used a labeling process that does not remove any arcs but marks them as deleted. Furthermore, the reversal of the arcs needed in the network flow algorithm is also performed by a labeling process. This way, the arc set of the original graph remains unchanged during the whole execution.

4.2.4 Disjoint spanning trees and covering with forests

Finding k edge-disjoint spanning trees and covering the edge set with k edge-disjoint forests are well-studied problems. Both are built on the previously discussed algorithm such that finding k arc-disjoint r -arborescences and covering with k arc-disjoint branchings, respectively. In the given undirected graph, a proper orientation is found by the Component Pebble Game with testing (k, k) -sparsity. After that, the previous algorithms are accomplished in this digraph to find k arc-disjoint arborescences or branchings in $O(k^2 m^2)$ steps, where $m = O(kn)$.

4.3 Benchmark environment

Now we describe the exact environment of the benchmarks that are discussed later.

All compared algorithms solved the extraction problem, that is, they extract an inclusion-wise maximal sparse subgraph.

Parameters of the algorithms

The parameters of the benchmark problems were chosen as follows.

- The parameters k and l are tested for each possible pair until a limit K on k , that is, each pair (k, l) with $1 \leq k \leq K$ and $0 \leq l < 2k$.
- For a fixed number n of vertices, multiple graphs were generated, and the running times represent the average of the graphs on n vertices with each pair of parameters (k, l) . In each figure, the horizontal axis represents the number of vertices of the graphs and the vertical axis shows the running time of the particular algorithms in seconds.

The types of multigraphs we tested were the following.

1. **Rigid:** Rigid graphs in the plane, that is, $(2, 3)$ -spanning graphs generated by the following process due to András Mihálykó. Let T denote the union of three trees on a vertex set. Replace each vertex $v \in T$ with a complete graph on $d_T(v) + 1$ vertices.
2. **Tight:** Tight graphs for a fixed parameter k , generated similarly to the previous replacing process.
3. **Random:** Erdős-Rényi random loopless multigraphs with different probabilities for the edges, meaning that each edge is independently included with the given probability. The probabilities we chose were $\{0.2, 0.4, 0.6, 0.8\}$, therefore, the graphs had different densities.
4. **Molecular:** Molecular graphs from the Protein Data Bank (<https://www.rcsb.org>). Their rigidity in $3D$ can be determined by using the molecular conjecture [23]. Briefly, they prove that a graph G can be realized as an infinitesimally rigid panel-hinge framework in $\mathbb{R}^3$ by mapping each vertex to a rigid panel and each edge to a hinge if and only if $5G$ contains six edge-disjoint spanning trees, where $5G$ is the graph obtained from G by replacing each edge with five parallel edges. Therefore, we tested $5G$ for each molecular graph G .

5. **Proteins:** Proteins from Protein Data Bank (<https://www.rcsb.org>) with an average of a few thousand vertices.

These cases cover all possibilities of the parameters k and l in the sense that each feasible solution of the Pebble Game algorithms can be achieved, that is, sparse, tight, spanning and none of them. Plus, both the lower and upper range is tested. Therefore, the following algorithms have been tested on a wide variety of parameters to surely give a detailed overview.

Technical details

The benchmarks mentioned above were executed on Linux operating system, 32 GB RAM and AMD Ryzen 9 3950X Processor.

4.4 Comparison of the implementations

In this section, we compare our implementation with a previous one, called KINARI.

4.4.1 KINARI-web

KINARI-web [12, 24, 25] is a software project implementing data structures and algorithms for rigidity analysis, with a special focus on applications in mechanical structures, abstract sparse graphs, and molecules. It is an interactive web server for making rigidity analysis and presenting properties of proteins. It also offers tools such as selecting chains from PDB files, adding further atoms and identifying stabilizing interactions.

Unfortunately, the KINARI website has not been available since the beginning of September 2022. Luckily, we had downloaded a precompiled version of their source code, which we use in our benchmarks.

After a lookup in the header files of their implementation, we regretfully figured out that separate implementations of the Basic and the Component Pebble Game cannot be found in their project. Therefore, our benchmarks include only a single version of KINARI, that is, the Component Pebble Game.

4.4.2 The compared algorithms

Now we show the algorithms whose running times have been compared.

1. **KINARI:** the Unweighted Component Pebble Game implemented in KINARI. It is based on the Basic Pebble Game described in Section 2.1. An essential difference between the implementations of the Component Pebble Game in KINARI and LEMON is that KINARI probably stores the edge sets of the components, but LEMON stores their vertex sets.
2. **Old:** an old version of Basic Pebble Game implemented in LEMON. The major shortcoming of this version is our terminating condition regarding the total number of pebbles in the game. Moreover, unlike the current version, it does not contain the fast initialization of the BFS algorithm.
3. **Basic:** the final version of Basic Pebble Game implemented in LEMON. This algorithm was described previously in Section 2.1. Furthermore, it contains every other trick, described in Section 4.2.
4. **Comp:** the Unweighted Component Pebble Game implemented in LEMON. It uses the concept of components to reject edges in constant time, described precisely in Section 2.2.1. Since it derives from the Basic Pebble Game in LEMON, it also contains the tricks described in Section 4.2.

4.4.3 Benchmarks

Now we present the running times of the algorithms for the graph types. For each set of instances, there are two figures next to each other, except for the random graphs. The left compares all algorithms, while the right focuses on the implementations in LEMON.

Rigid graphs

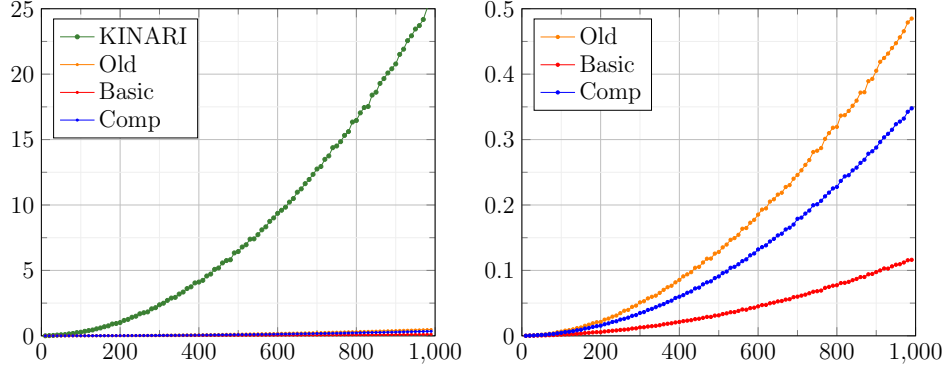


Figure 2: *The comparison on rigid graphs.*

Based on the figure on the left, KINARI is an order of magnitude slower than the LEMON algorithms on rigid graphs. The Component Pebble Game implemented in LEMON seems to have larger running times than the Basic Pebble Game on rigid graphs. This is a special family of graphs because no edge is rejected during the execution since rigid graphs are sparse, meaning it makes no sense to update the components.

Tight graphs

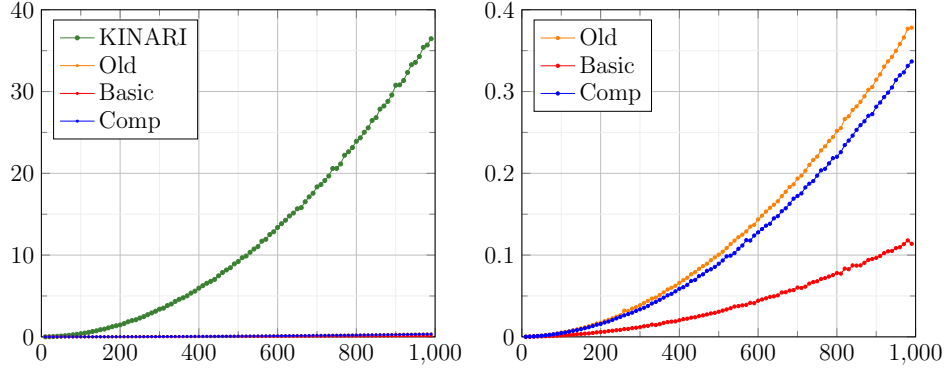


Figure 3: *The comparison on tight graphs.*

Similarly to the rigid graphs, the running time of KINARI seems an order of magnitude slower than the other implementations in LEMON on tight graphs. Furthermore, the gap between the Component Pebble Game and the Basic Pebble Game is greater on tight graphs than on rigid ones. It is because they contain more accepted edges than the rigid ones, and no edge is rejected, which means even more unnecessary updates in the data structures.

Moreover, the average times of the Basic Pebble Game and the Component Pebble Game in LEMON remain unchanged compared to the rigid graphs. Note that the running times for rigid and tight graphs differ the most for KINARI, as it runs more than one and a half times slower on tight ones. It seems that KINARI is slower by an order of magnitude, and the other algorithms run in $O(n^2)$ time on tight graphs.

Random graphs

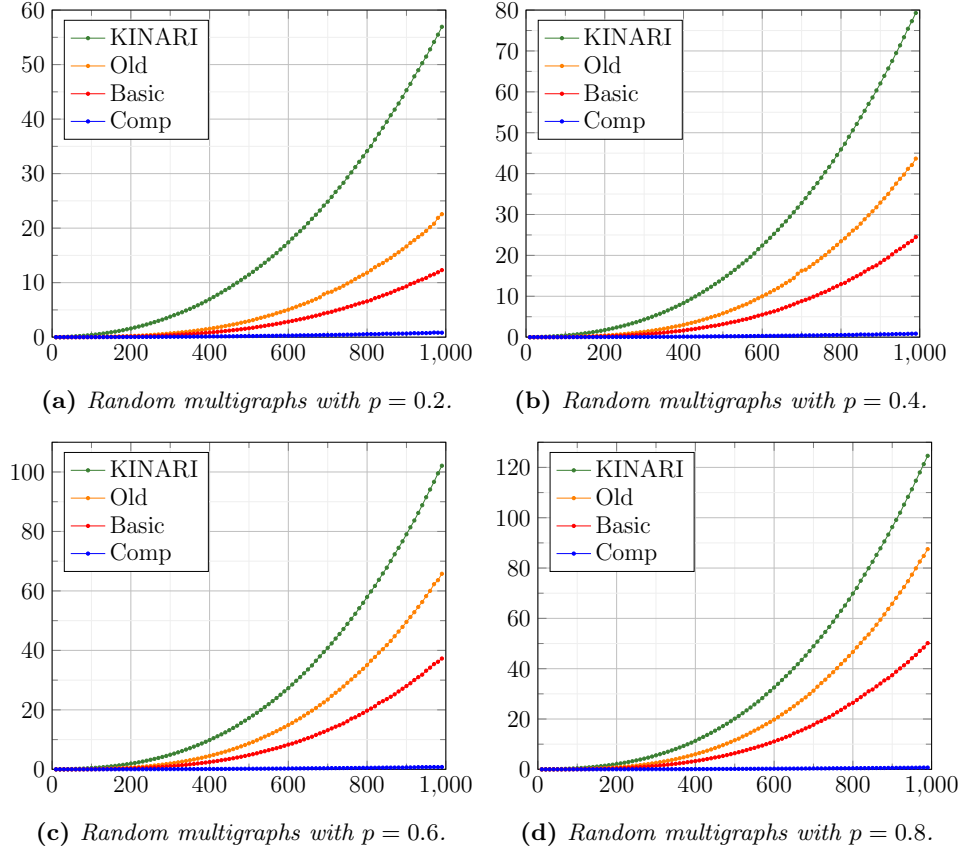


Figure 4: The comparison on random graphs with the shown probabilities on the edges.

Similarly to rigid and tight graphs, KINARI is an order of magnitude slower than the Component Pebble Game in LEMON on random graphs. Surprisingly, the running time of the KINARI and the Basic Pebble Game algorithms in LEMON seem somewhat similar on these graphs. The Component Game in LEMON runs in $O(n^2 + m) = O(n^2)$ time and the other algorithms in $O(nm) = O(n^3)$ time on dense graphs. Plus, among the algorithms running in $O(n^3)$ time, the Basic Pebble Game has the smallest constant due to the used accomplishments, precisely described in Section 4.2.

In addition, as Figure 4 shows, the running times of the Basic Pebble Game and the old version change proportionally for each density of the random graphs. This means that the speed of the acceleration does not depend on the density of the graphs, but on how fast we find the largest sparse subgraph. Furthermore, the denser the graphs are, the larger the gap grows between the Component Pebble Game in LEMON and the other algorithms. It is because there

are more and more rejected edges in the graphs, therefore, the constant-time rejections using components are efficient. Moreover, the old version is also getting slower on dense random graphs, since the influence of our terminating condition increases heavily.

Observe that the running times of the algorithms do not vary proportionally for the different probabilities. For example, while the time of KINARI doubles from 0.2 to 0.8, the time of the Basic version quadruples. The following figure shows that the Component Pebble Game in LEMON is similar for all ranges of random graphs.

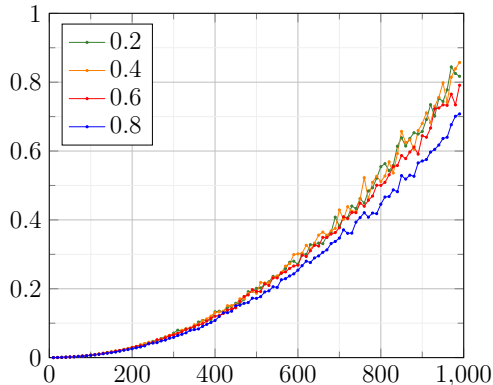


Figure 5: Comparison of the Component Pebble Game on random graphs with the given probabilities. Unlike the other algorithms for random graphs, its running time does not increase but rather decreases as the probability increases.

As the figure shows above, the running time of the Component Pebble Game in LEMON does not depend on the density of the graphs. Nevertheless, the types of graphs significantly affect the running times as the rest of the algorithms run twice as fast on rigid and tight graphs than on random graphs.

Molecular graphs

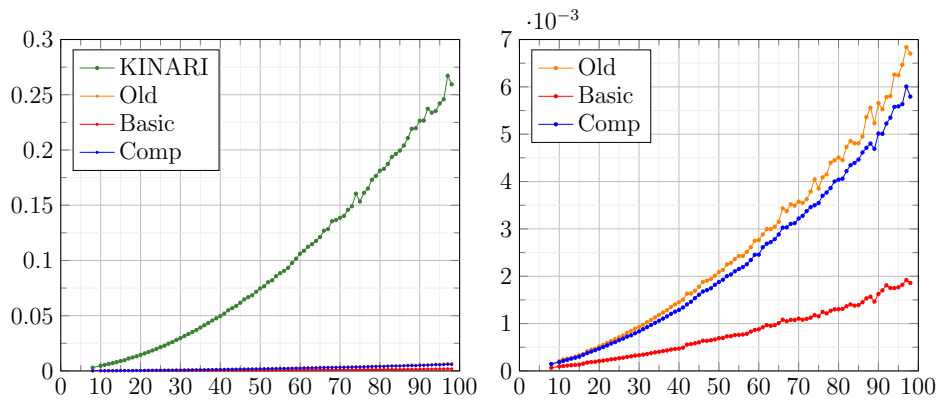


Figure 6: The comparison on molecular graphs.

KINARI was principally developed for the rigidity analysis of molecular and protein graphs in 3D. Nevertheless, it seems an order of magnitude slower than the LEMON versions on molecular graphs. The Component Pebble Game is proven to be slower than the Basic Pebble

Game on molecular graphs because there are a few rejected edges and the unnecessary update processes of the components take a long time. Moreover, the gap between the old version of the Basic Pebble Game and the Component Pebble Game is almost insignificant. This suggests that the needless updates of the components take roughly as much time as the further redundant processing of the rejected edges at the end of the execution.

Protein graphs

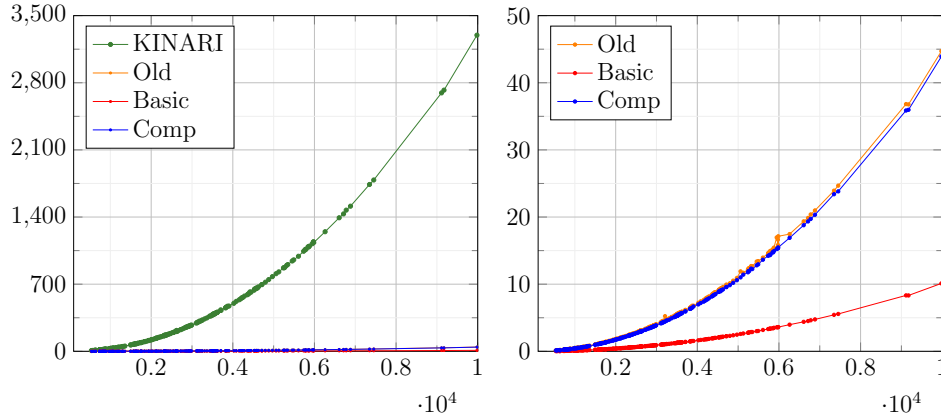


Figure 7: *The comparison on proteins.*

The running times of the LEMON algorithms are also an order of magnitude faster on proteins than the KINARI. Moreover, the old version of the Basic Pebble Game and the Component Pebble Game seem more similar on proteins than on molecular graphs.

4.4.4 Conclusion of the benchmarks

Altogether, we may conclude that the Component Pebble Game in LEMON is an order of magnitude faster than the Component Pebble Game of KINARI for all kinds of graphs. We also saw that the running time of the KINARI implementation heavily depends on the density of the graphs, unlike the Component Pebble Game implemented in LEMON. In addition, the running time of the KINARI also depends more strongly on the types of graphs than the running time of the LEMON version.

Surprisingly, the figures suggest that our implementation of the Component Game in LEMON seems to have the best running time, which is $O(n^2 + m)$ steps, that is, $O(n^2)$, while the KINARI seems to run in $O(n^3)$ time.

Finally, we note that the Component Pebble Game seems outstanding for dense graphs and the Basic Pebble Game for sparse graphs, as we expected based on the worst-case analysis.

Possible reasons of the shortcomings of KINARI

Now we summarize the possible reasons for the shortcomings of the KINARI version.

- The KINARI project and the precompiled library were introduced in 2011, so it may be unfair to compare it using a more recent *C++* compiler, say, *gcc 10.4*. Thus, we carefully checked our comparisons with the same system requirements as they could have, like *gcc 4.7*. Finally, we figured out that there is no substantial difference between the running times of the KINARI algorithm compiled with *gcc 4.7* and *10.4*.

- As we mentioned earlier, the KINARI version is likely to store the edge sets of the components. However, since their project is not open-source, we could not examine the entire code precisely. In addition, they use *std::set* containers for them, which is usually implemented as a balanced binary search tree with a logarithmic lookup time. In our implementation, we used *std::vector* containers, which guarantees constant-time queries.

Note that we checked the cardinality of the maximal sparse subgraph after each execution, and it turned out that KINARI does not handle loops correctly. For example, it does not consider a loop on a single vertex $(2, 1)$ -sparse. Fortunately, all algorithms found feasible solutions of the same cardinality for the extraction problem on each test instance.

LEMON algorithms

The benchmarks on the LEMON versions suggest that the running time in practice coincides with the expected time based on the theoretical analyses.

- The old version of the Basic Pebble Game runs in $O(klmn)$. However, the constant multiplier of it seems to be relatively large compared to the newest version because it does not contain our improvements, discussed in Section 4.2.
- The Basic Pebble Game runs in $O(klmn)$ time, meaning it also depends on the number of edges. This suggests that the denser the graphs are, the greater the times become. However, the tricks profoundly sped up this version, discussed in Section 4.2. Therefore, the average running times improved, as we saw in the previous figures. Moreover, the Basic Pebble Game proved to be the best algorithm on sparse graphs. This is because if relatively few edges are rejected during the algorithm, then no unnecessary component updates are accomplished.
- The Component Pebble Game runs in $O(k^2ln^2 + m)$ meaning a worst case of $O(n^2)$ on these graphs. Therefore, the running time of this algorithm does not depend on the number of edges that heavily, which is what we saw in the previous figures. Moreover, the Component Pebble Game proved to be the best of all algorithms on dense graphs, which was unexpected because of the constant-time rejections of the edges.

4.5 Heuristic edge ordering

In this section, we introduce some heuristics about the order of the edges for the Basic Pebble Game. As it soon turns out, the running time heavily depends on this order, because as soon as the maximal sparse subgraph is found, our terminating condition halts the algorithm.

4.5.1 Type of tested heuristics

The tested heuristics were the following.

1. **Basic:** the latest version of the Basic Pebble Game implemented in LEMON. It uses a natural order of the edges by the graph representation, that is, processing all edges incident with a vertex. Note that the natural order of the edges is determined in constant time. This heuristic is to be considered as the baseline in our experiments.
2. **Degmin:** select an edge incident with a vertex that has the minimal degree. The degree of a vertex correlates with the number of pebbles in the inner digraph. Therefore, an edge incident with a vertex that has a minimal degree is likely to be accepted, because the number of pebbles is maximal on the vertex.
3. **Disjoint:** select an edge that has the fewest incident edges, that is, the total number of edges incident with the endpoints of the edge is minimal. If an edge is the most disjoint

in the previous sense, then the total number of pebbles seems maximal on the endpoints of the edge, which means that the edge is the most likely to be accepted.

4. **Maxpeb**: select an edge such that the total number of pebbles on the endpoints of the edge is maximal. If an edge is selected in this way, then it is the most likely to be accepted because of the rule of the edge acceptance. Moreover, this heuristic checks the number of pebbles on the endpoints of each edge, we may expect that it selects the desired edge more precisely than the second heuristic. However, selecting the edge to be processed takes longer.

All heuristics are designed to find the maximal sparse subgraph as soon as possible, that is, the edges that are most likely to be accepted should be processed first. This significantly affects the running time of the Basic Pebble Game because as soon as the maximal sparse subgraph is found, the algorithm is terminated by our condition discussed in Section 4.2.

4.5.2 Benchmarks

Now we present a practical study of the heuristics for all types of graphs mentioned above. Note that the represented running times do not take into consideration the process of selecting the following edge.

Rigid and tight graphs

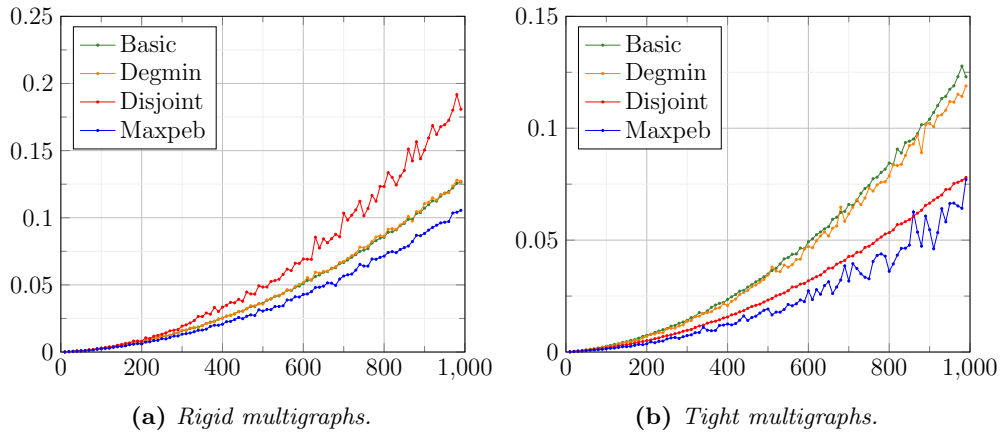


Figure 8: The comparison of the heuristics in rigid and tight graphs.

The running time of each heuristic seems to be similar on both rigid and tight graphs because the maximal sparse subgraph is found almost at the same time as only a few edges are rejected. On both graph types, the Basic and selecting a vertex with minimum degree are similar. However, the running time of the heuristics seems to differ on both rigid and tight graphs.

On rigid graphs, the slowest heuristic is surprisingly not the Basic one but selecting a disjoint vertex, and the fastest heuristic is selecting the maximum number of pebbles. In addition, the disjoint heuristic seems significantly inconsistent on rigid graphs.

On tight graphs, the slowest heuristics are the Basic and selecting a vertex with minimum degree. Surprisingly, the disjoint heuristic seems more consistent on tight graphs than on rigid ones. Selecting an edge with the maximum number of pebbles on the endpoints is proven to be the best due to the rule of edge acceptance.

Random graphs

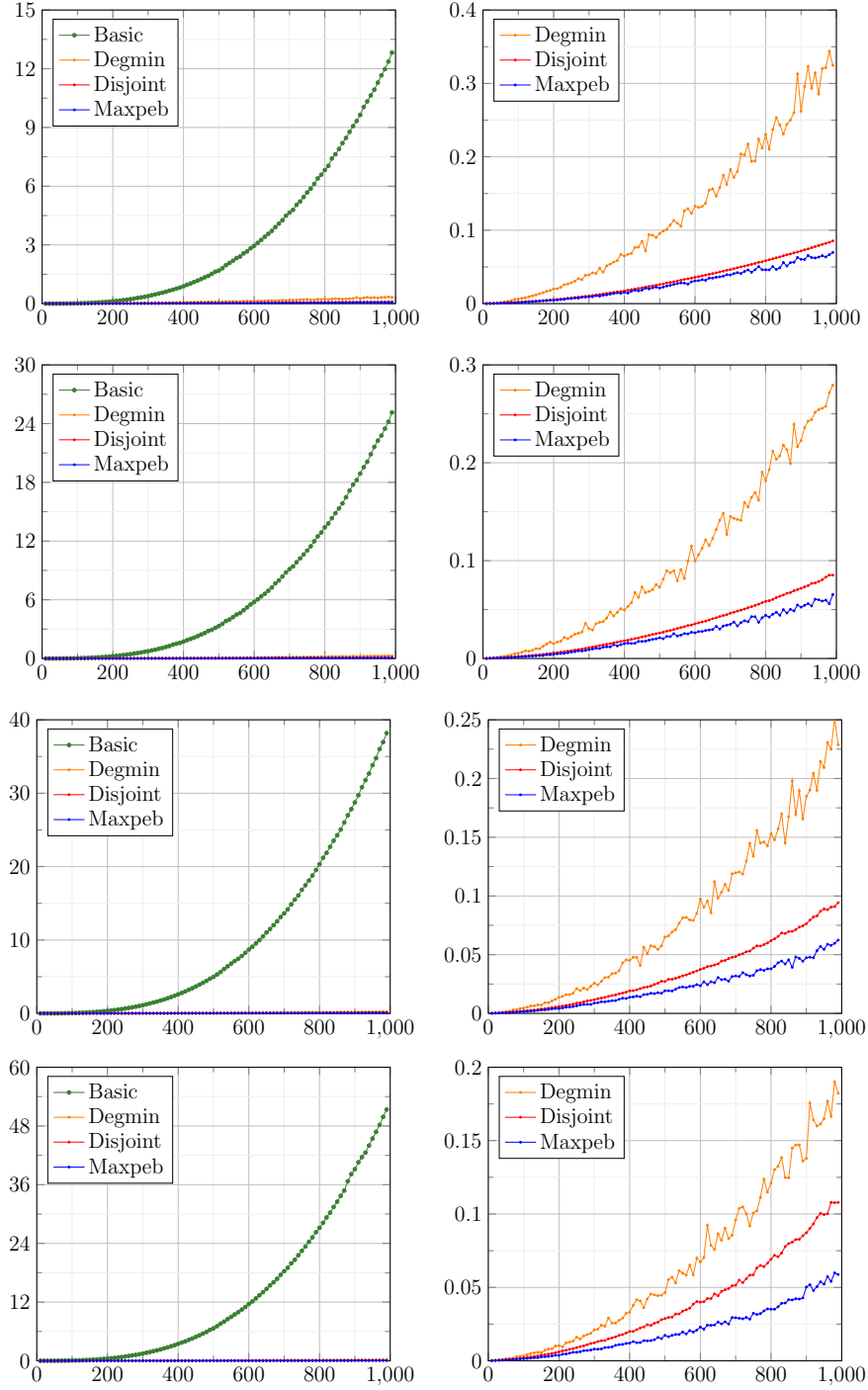


Figure 9: Comparing the heuristics on random graphs with probabilities 0.2, 0.4, 0.6, 0.8 on the edges, respectively. The pairs of figures next to each other show the same case.

First, the Basic edge order seems utterly slower than the other heuristics. This is what we expected because this order does not consider our preference, which is to select an edge that is most likely to be accepted first. Moreover, observe that the running time of the Basic algorithm decreases, but the time of the other heuristics increases on denser graphs. This means that the denser the graphs are, the larger the gaps become.

The figures on the right represent the differences between the rest of the heuristics. Selecting a vertex with a minimal degree seems to be the slowest among them. Plus, it seems inconsistent, which means that the exact running time of this heuristics is uncertain as a function of the number of vertices. Moreover, it suggests that the denser the graphs are, the more inconsistent it becomes.

The second fastest heuristic is selecting a disjoint edge. Observe that the running time of this heuristic is also not correlated with the density of the graphs.

The quickest heuristic on random graphs is selecting an edge with a maximum number of pebbles on the endpoints. The variation in running time through density changes is minimal, which confirms that the heuristic is well-established. It fulfills our expectations because the edge acceptance rule requires the total number of pebbles on the endpoints of an edge to be more than l . Therefore, selecting such an edge seems the best idea to find a maximal sparse subgraph as soon as possible.

Molecular graphs and proteins

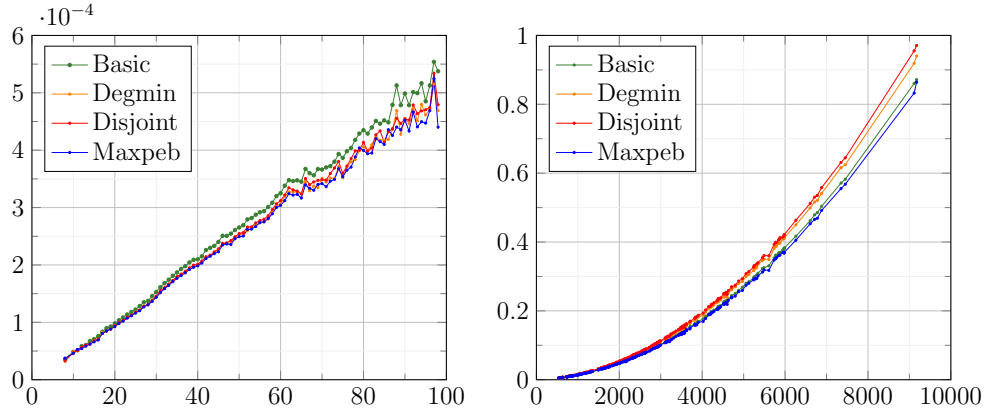


Figure 10: *The comparison on molecular graphs (left) and proteins (right).*

Similarly to rigid and tight graphs, the heuristics seem to have slight differences in molecular graphs and proteins. This is because there are a few rejected edges in these graphs, meaning that it makes no sense to try to pick such an edge which is most likely to be accepted. Despite this, selecting an edge with a maximum number of pebbles on the endpoints remains the best heuristic in these types of graphs.

4.5.3 Conclusion of the heuristics

Altogether, we may conclude that the order of the edges is highly decisive in speeding up the Pebble Game algorithms.

To sum up, the tested heuristics performed as follows.

- **Basic:** it is proven to be the slowest of all heuristics because it does not take into consideration the further study, discussed in Section 4.2.
- **Degmin:** in most cases, it performed better than the Basic version because the minimal degree means that the number of pebbles is probably large. Therefore, the edges incident with vertices that have minimal degrees are likely to be accepted.
- **Disjoint:** as the figures show, this heuristic seems even better than the minimal degree because it considers first the edges whose endpoints have the minimal degrees, which measures better whether an edge is likely to be accepted.
- **Maxpeb:** as we saw, it is proven to be the best of all heuristics, because of the rule of edge acceptance, which is to have more than l pebbles on the endpoints of an edge. Moreover, the performance of this heuristic fluctuates the least.

5 | Future plans and open questions

In this section, we briefly summarize our plans and the open questions.

We are planning to examine more types of heuristics regarding the order of the edges in the Basic Pebble Game. All the designed heuristics aimed to minimize the number of processed edges before finding the largest sparse subgraph. It is also essential to minimize the total running time of the graph traversals needed to accept these edges. Finding such edge orders heuristically is subject to further research. In addition, some heuristics may be analyzed for the Component Pebble Game even though only the order of the vertices is arbitrary in this algorithm. Moreover, the already implemented algorithms may be improved with the best heuristics, apparently the maximal number of pebbles.

One of the main open questions is to design and implement an $O(n^2)$ algorithm for the weighted version of the problem in the upper range. Furthermore, finding the maximum-size sparse subgraph when $l = 2k$ is also an open question, which has many applications in rigidity theory.

Finally, we are also planning to implement faster algorithms for specified ranges, for example, one running in $O(n\sqrt{n})$ time for the lower range.

Acknowledgement

I would like to express my deepest gratitude to my supervisor, Péter Madarasi, who inspired and supported me through this topic. I am also grateful to Tibor Jordán and Csaba Király for pointing us to relevant literature and for discussions.

References

- [1] M. Loréa. On matroidal families. *Discrete Mathematics* 28, 1979.
- [2] C. S. A. Nash-Williams. Edge-disjoint spanning trees of finite graphs. *Journal of the London Mathematical Society*, 1961.
- [3] W. T. Tutte. On the problem of decomposing a graph into n connected factors. *Journal of the London Mathematical Society*, 1961.
- [4] G. Laman. On graphs and rigidity of plane skeletal structures. *Journal of Engineering Mathematics*, 1970.

- [5] A. Recski. A network theory approach to the rigidity of skeletal structures i. *Discrete Applied Mathematics* 7, 1984.
- [6] D. J. Jacobs, A. Rader, M. Thorpe, and L.A. Kuhn. Protein flexibility predictions using graph theory. *Proteins* 44, 2001.
- [7] A. Lee and I. Streinu. Pebble game algorithms and sparse graphs. *Discrete Mathematics* 308, 2008.
- [8] A. Lee, I. Streinu, and L. Theran. Finding and maintaining rigid components. *Canadian Conference on Computational Geometry*, 2005.
- [9] H. Gabow and H. Westermann. Forests, frames, and games: Algorithms for matroid sums and applications. *Algorithmica* 7, 1992.
- [10] O. Daescu and A. Kurdia. Towards an optimal algorithm for recognizing Laman graphs. In *2009 42nd Hawaii International Conference on System Sciences*. IEEE, 2009.
- [11] J. Rollin, L. Schlipf, and A. Schulz. Recognizing planar Laman graphs. In *27th Annual European Symposium on Algorithms (ESA 2019)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2019.
- [12] KINARI: Kinematics and Rigidity. <http://kinari.cs.umass.edu>.
- [13] Cs. Király and A. Mihálykó. Globally rigid augmentation of rigid graphs. *SIAM Journal on Discrete Mathematics*, 2022.
- [14] Cs. Király and A. Mihálykó. Sparse graphs and an augmentation problem. *Mathematical Programming*, 2022.
- [15] B. Servatius J. Graver and H. Servatius. Combinatorial rigidity. *American Mathematical Society*, 1993.
- [16] S. Meera, J. Audrey St, and J. Sidman. *Handbook of geometric constraint systems principles*. Chapman and Hall/CRC, 2018.
- [17] T. Jordán. Rigid block and hole graphs with a single block. Technical Report TR-2021-09, Egerváry Research Group, Budapest, 2021. <https://egres.elte.hu>.
- [18] A. Schrijver. Combinatorial optimization. *Springer*, 2004.
- [19] LEMON: Library for Efficient Modeling and Optimization in Networks. Repository of sparse graphs: <https://lemon.cs.elte.hu/repos/sparseGraphs>.
- [20] LEMON: Library for Efficient Modeling and Optimization in Networks. <https://lemon.cs.elte.hu/trac/lemon>.
- [21] D. Balázs, A. Jüttner, and P. Kovács. LEMON - an Open Source C++ Graph Template Library. *Electron. Notes Theor. Comput. Sci.*, 2011.
- [22] R. M. Karp. Reducibility among combinatorial problems. *University of California at Berkeley*, 1972.
- [23] N. Katoh. A proof of the molecular conjecture. *Discrete Comput Geom* 45, 2009.
- [24] N. Fox, F. Jagodzinski, Y. Li, and I. Streinu. KINARI-web: a server for protein rigidity analysis. *Nucleic Acids Research*, 2011.
- [25] N. Fox, F. Jagodzinski, and I. Streinu. KINARI-lib: A C++ library for mechanical modeling and pebble game rigidity analysis. *Minisymposium on Publicly Available Geometric/Topological Software*, 2012.