NYILATKOZAT

Név: Matúz Lóránt

ELTE Természettudományi Kar, szak: Matematika Bsc

NEPTUN azonosító: A67MNL

Szakdolgozat címe:
(k,l)-sparse graphs and their applications

A **szakdolgozat** szerzőjeként fegyelmi felelősségem tudatában kijelentem, hogy a dolgozatom önálló szellemi alkotásom, abban a hivatkozások és idézések standard szabályait következetesen alkalmaztam, mások által írt részeket a megfelelő idézés nélkül nem használtam fel.

Budapest, 2022.05.30.

a hallgató aláírása

Eötvös Loránd University
Faculty of Science

(k, l)-sparse graphs and their applications

Thesis

Lóránt Matúz
Mathematics BSc
Applied Mathematics Major

Supervisor:
Péter Madarasi
Department of Operations Research

Budapest
2022

Contents

Acknowledgement

I would like to express my deepest gratitude to my supervisor, Péter Madarasi, who first recommended this exciting topic, then inspired and supported me through it. Our effective work together also helped me to learn how to accomplish a hard project by myself.

I am also grateful to Csaba Király and András Mihálykó for pointing us to relevant literature and for some discussions.

Last but not least, I also appreciate all the support I got from my friends and my family during these tough times.

1 Introduction

In this chapter, we briefly introduce the history of (k,l)-sparse graphs in Section *1.1*, then the fundamental problems that are solved by the described Pebble Game algorithms in Section *1.2*, and finally, we describe the notations and definitions that are used throughout this paper from [1].

1.1 Historical overview

The definition of (k,l)-sparse graphs was first introduced in 1979 by Loréa [2] as example of matroidal families. They have been intensively studied in the last decade and it soon became apparent that they have a wide variety of applications. Hence, (k,k)-tight graphs also appeared in the classical results of Nash-Williams [3] and Tutte [4] as the characterization of the graphs that can be decomposed into k edge-disjoint spanning trees.

Later, Laman showed in [5] that $(2,3)$-tight graphs are the generic minimally rigid graphs for bar-and-joint frameworks in the plane, and the $(2,3)$-sparse graphs are the rigid ones. Further analysis on rigidity theory in the plane was done by Recski in [6] and the special case of three-dimension for the practical application of protein flexibility is handled by Jacobs in [7].

The approach of the Pebble Game algorithm was first presented by Jacobs and Hendrickson in [8], and was further analyzed by Lee and Streinu with pebble game algorithms and sparse graphs in our main reference, [1].

1.2 Fundamental problems

Now we briefly introduce the problems that are solved by the algorithms described later. For better understanding of the following problems, we need some definitions, which will be further discussed in Section *1.3*.

An undirected multigraph G is called (k,l)-sparse if every subset X of vertices induces at most $\max\{0, k|X|-l\}$ edges. Furthermore, $G=(V,E)$ is called (k,l)-tight if it is (k,l)-sparse and it has exactly $k|V|-l$ edges.

Throughout this paper, we investigate the following problems:

1. **Decision:** Decide if G is a (k,l)-sparse (or (k,l)-tight) graph.
2. **Spanning:** Detect if G contains a (k,l)-tight subgraph.
3. **Extraction:** Extract a maximal (k,l)-sparse subgraph from G.
4. **Optimization:** Extract a maximum weight (k,l)-sparse subgraph from the given graph G with weighted edges.
5. **Components:** Find all components (maximal (k,l)-tight subgraphs) in the given non-spanning graph G.

We introduce the Basic Pebble Game in Chapter *3* which solves the first four of these, then the Unweighted Pebble Game in Section *4.2* which can solve the first three and the last, and finally the Component Pebble Game in Section *4.1* to solve all of them if $k>0$ and $0 \leq l < 2k$. In Section *6.1*, some generalizations are introduced in which some of these problems are solved for special parameters k and l.

1.3 Basic notations and definitions

In this section, we introduce the basic definitions required later. Note that sparsity is defined only for undirected multigraphs, however, all the algorithms construct an inner directed multigraph, so some notations are also introduced for them.

Notations

A **graph** is a pair $G = (V, E)$, where V is the set of vertices and E is a set of two-element subsets of the vertices, called edges. An edge $e = uv$ is **incident** to a vertex w if one of its endpoints is w, that is, if $w = u$ or $w = v$. A **directed graph** or a **digraph** is a pair $D = (V, A)$, where V is the set of vertices and A is a set of ordered pairs of distinct vertices, called directed edges, or arcs. Each arc $a = uv$ of A has a direction, such that it points from vertex u (called source) to v (called target). A **multigraph** is a pair $G = (V, E)$, where V is the set of vertices and E is a multiset of paired vertices, called edges. Note that a multigraph may have **parallel edges** (at least two edges with the same incident vertices) and **loops** (edges with the same endpoints). A **directed multigraph** is a multigraph that may have parallel arcs (the same arcs multiple times) and loops (arcs with the same source- and target vertices). A **subgraph** $G' = (V', E')$ of a multigraph $G = (V, E)$ is a multigraph formed from a subset of the vertices and edges, i.e. $V' \subseteq V$ and $E' \subseteq E$, such that each endpoint of the edges of E' is included in V'. A **proper subgraph** $G' = (V', E')$ of a multigraph $G = (V, E)$ is a subgraph whose edge set E' is a proper subset of E, that is, E' is not equal to E. An **induced subgraph** $G' = (V', E')$ of a multigraph $G = (V, E)$ is a subgraph whose edge set E' contains exactly the edges of G with both endpoints in V'. Thus, in a multigraph $G = (V, E)$, a vertex subset $V' \subseteq V$ uniquely determines an induced subgraph $G' = (V', E')$, therefore V' is said to **induce** an edge set E' if G' is an induced subgraph. The **degree** of a vertex v of an undirected multigraph is the number of edges incident to v, including loops, which we denote by $\deg(v)$. The **indegree** or the **outdegree** of a vertex v of a directed multigraph is the number of incoming or outgoing arcs into or from v. Denote them by $\deg^{in}(v)$ and $\deg^{out}(v)$, respectively.

Definitions

Now we turn to the main definitions related to sparsity on undirected multigraphs:

Definition 1.3.1
*A multigraph G is (k, l)-**sparse** if any subset X of vertices induces at most $\max\{0, k|X| - l\}$ edges.*

Throughout this paper, $i(X)$ denotes the cardinality of the induced edge set of a vertex subset X.

Definition 1.3.2
*A multigraph $G = (V, E)$ is (k, l)-**tight** if it is (k, l)-sparse and it has exactly $k|V| - l$ edges.*

Definition 1.3.3
*A multigraph $G = (V, E)$ is (k, l)-**spanning** if it contains a (k, l)-tight subgraph that spans the entire vertex set V.*

Throughout this paper, we may refer these concepts without the parameters k and l if it is clear from the context.

Lower and upper range

Throughout the paper, we mostly focus on the case of $k > 0$ and $0 \leq l < 2k$. It will be useful to divide this case according to the following.

- **Lower range:** if $k > 0$ and $0 \leq l \leq k$.
- **Upper range:** if $k > 0$ and $k < l < 2k$.

The (k, l)-sparse graphs in the lower- and upper ranges have different properties, therefore the distinction facilitates their study.

2 Properties of sparse graphs

In this chapter, we introduce the basic properties of (k,l)-sparse graphs, largely based on [1].

Throughout this chapter, $G=(V,E)$ denotes a multigraph with $n := |V|$ vertices and $m := |E|$ edges, unless stated otherwise. Let n' and m' denote the number of vertices and edges of a subgraph $G'=(V',E')$ of G, respectively.

For the detailed description below, let $K_n^{a,b}$ denote the complete multigraph on n vertices with a loops on each vertex and b parallel edges between any two distinct vertices. Let $K_n^b := K_n^{0,b}$.

2.1 Basic characterization

Now we are ready to show the reason behind the interval of $0 \leq l < 2k$.

Lemma 2.1.1
If $l \geq 2k$, then only the empty graph is (k,l)-sparse.

Proof.
For every subset X of size $n'=2$, the number of induced edges $i(X)$ is at most $2k-l \leq 0$, therefore no (k,l)-sparse graph may have any edges, when $l \geq 2k$. □

Lemma 2.1.2
If $l<0$, then the union of two vertex-disjoint (k,l)-sparse graphs may not be (k,l)-sparse.

Proof.
Let the graphs be G_1 with n_1 vertices and G_2 with n_2 vertices. As they are vertex-disjoint, the union has exactly $n := n_1+n_2$ vertices, and induces

$$(kn_1-l)+(kn_2-l) = k(n_1+n_2)-2l > kn-l$$

edges, so the union cannot be sparse. □

Lemma 2.1.3
All (k,l)-sparse graphs may contain at most $k-l$ loops on each vertex and the multiplicity of any parallel edge is at most $2k-l$.

Proof.
By *definition*, any single vertex v may induce at most $k-l$ loops and all pairs of vertices may induce at most $2k-l$ edges. The latter gives the desired upper bound on the multiplicity of parallel edges. □

Corollary 2.1.4
All (k,l)-sparse graphs are loopless when $l \geq k$.

The following statement comes from Szegő [9].

Lemma 2.1.5
In the range of $l \in \left[\frac{3}{2}k, 2k\right)$, there are no (k,l)-tight graphs on n vertices for $2 < n < \frac{l}{2k-l}$.

Proof.
By Corollary *2.1.4*, all vertices are loopless for $l \geq k$, so $n \geq 2$ is required. As a tight graph on n vertices is a subgraph of the complete, loopless multigraph K_n^{2k-l}, the following upper bound can be shown:

$$kn - l \leq (2k - l) \cdot \binom{n}{2}.$$

Extending the inequality for n leads to the following quadratic function:

$$0 \leq (2k - l) \cdot n^2 + n(l - 4k) + 2l := f(n)$$

The solutions for $f(n) = 0$ are $n_1 = 2$ and $n_2 = \frac{l}{2k-l}$, which are exactly the mentioned bounds. The interval (n_1, n_2) is nonempty if $n_2 \geq 3$, which is equivalent with $l \geq \frac{3}{2}k$. Within this interval, all subgraphs of K_n^{2k-l} are sparse, and no tight ones can be shown. □

Corollary 2.1.6
In the range of $l \in \left[\frac{3}{2}k, 2k\right)$, the smallest non-trivial (k,l)-tight graphs have $\left\lceil \frac{l}{2k-l} \right\rceil$ vertices. If $\frac{l}{2k-l}$ is integer, then the only (k,l)-tight graph on $\frac{l}{2k-l}$ vertices is the complete multigraph K_n^{2k-l}; otherwise there can be several.

2.2 Blocks and components

This section introduces a commonly used concept, called components. They play a role in speeding up the Basic Pebble Game, which is discussed in Section *4.1*. They also appear in rigidity applications because they correspond to rigid clusters, which is described in Section *5.3*.

Definition 2.2.1
*An induced proper subgraph $G' = (V', E')$ of a (k,l)-sparse graph G is a **block** if it induces exactly $k|V'| - l$ edges.*

Definition 2.2.2
*A maximal block, with respect to the number of vertices, is a **component**.*

Definition 2.2.3
*A vertex v of a multigraph is called **free vertex** if no component contains v.*

Definition 2.2.4
*An edge e of a multigraph is called **free edge** if e is not induced by any block, hence component.*

Now we focus on the basic properties of components, and make general observations regarding decomposing a multigraph into components.

(a) *A $(1,1)$-sparse graph.* **(b)** *A $(2,3)$-sparse graph.*

Figure 2.1: *Examples for Lemma 2.2.5. Figure (a) shows a $(1,1)$-sparse graph with two blocks, where the intersection is also a block, even though it contains only a single vertex without any edges. Figure (b) shows a $(2,3)$-sparse graph with two blocks, where both the union and the intersection induce a block.*

Lemma 2.2.5
If two blocks of a (k,l)-sparse graph G intersect in at least

a) one vertex for the lower range, (see Figure 2.1 (a)) or

b) two vertices for the upper range (see Figure 2.1 (b)),

then the intersection and the union of their vertex sets induce blocks.

Proof.
Assume that the two blocks of $G = (V, E)$ are $B_1 = (V_1, E_1)$ and $B_2 = (V_2, E_2)$ with n_1 and n_2 vertices, and $m_1 = kn_1 - l$ and $m_2 = kn_2 - l$ edges.

Let $G_\cap$ and $G_\cup$ be the subgraphs of G induced by the intersection $V_1 \cap V_2$ and the union $V_1 \cup V_2$, respectively. Denote the number of their vertices by $n_\cap$ and $n_\cup$, and the number of their edges by $m_\cap$ and $m_\cup$. Then the following equation holds for their edges:

$$m_\cup = m_1 + m_2 - m_\cap = (kn_1 - l) + (kn_2 - l) - m_\cap = k(n_1 + n_2) - 2l - m_\cap =$$

$$= k(n_\cup + n_\cap) - 2l - m_\cap = kn_\cup - l - (m_\cap - (kn_\cap - l)).$$

As G is (k,l)-sparse, $m_\cap \leq kn_\cap - l$, from where $m_\cap - (kn_\cap - l) \geq 0$. It follows that $m_\cup \geq kn_\cup - l$.

Assume that $n_\cap \geq 1$ in the lower range and $n_\cap \geq 2$ in the upper range. From the sparsity condition, $m_\cup \leq kn_\cup - l$, from where we get $m_\cup = kn_\cup - l$ and $m_\cap = kn_\cap - l$ as well. It means that both the intersection and the union are blocks. □

(a) *Components in a $(2,2)$-sparse graph.* **(b)** *Components in a $(3,4)$-sparse graph.*

Figure 2.2: *Components properties in lower (a) and upper (b) range.*

Corollary 2.2.6
Components of a (k,l)-sparse graph G are edge-disjoint. They are also vertex-disjoint in the lower range (see Figure 2.2 (a)), and they may overlap in at most one vertex in the upper range (see Figure 2.2 (b)).

Lemma 2.2.7
Let G be a (k,l)-sparse graph.

- *If $l = 0$, then there is at most one disconnected component in G (see Figure 2.3).*
- *If $l > 0$, then all blocks (and components) in G are connected (see Figure 2.2).*

Proof.
- If $l = 0$, then suppose that there exist two different, disconnected components C_1, C_2. By *definition*, $i(C_1) = k|C_1|$, $i(C_2) = k|C_2|$ holds and by Corollary *2.2.6*, components are edge-disjoint, so the following equality is met for their union:

  $$i(C_1 \cup C_2) = k|C_1| + k|C_2| = k(|C_1| + |C_2|).$$

  In fact, this means that the two components form another one.

- The case of $l > 0$ is a direct corollary of Lemma *2.4.3*.

□

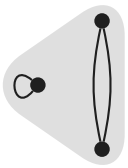


Figure 2.3: *A disconnected component in a* $(1,0)$*-sparse graph.*

Lemma 2.2.8
Let $G = (V, E)$ *be a* (k, l)*-sparse graph. A single vertex induces a block if and only if it has* $k - l$ *loops.*

- *If* $l = 0$*, then a block may be an isolated part of a larger component, otherwise it is a component in itself (see Figure 2.3).*
- ***Lower range:*** *If* $0 \leq l \leq k$*, then a vertex with fewer than* $k - l$ *loops is either free or part of a larger block (component).*
- ***Threshold case:*** *If* $l = k$*, then a single vertex is loopless by Corollary 2.1.4, and is always a block. Thus, there are no free vertices, and every vertex of* V *belongs to exactly one component (see Figure 2.2 (a)).*
- ***Upper range:*** *If* $k < l < 2k$*, then each vertex is either free or part of some larger blocks (components).*
- *If* $l = 2k - 1$*, then there are no loops or parallel edges. A single vertex is free only if it is isolated. A single edge is always a block, and there are no free edges, thus every edge belongs to exactly one component (see Figure 2.2 (b)).*

Proof.
For analyzing the single vertices, let $X := \{v\}$ be the subset. The definition of sparsity for X is $i(X) \leq k|X| - l = k - l$. A vertex v forms a block if and only if $i(X) = k - l$, that is, $k - l$ loops are incident to v.

- If $l = 0$, then by Lemma *2.2.7*, there may be an isolated component, and v can also form one. We also showed in the proof of Lemma *2.2.7* that the union of two components is also a component, meaning that v can be a part of a larger component.
- **Lower range:** If $0 \leq l \leq k$, then fewer than $k - l$ loops means that X does not form a block (component). If there is no component containing v, then it is a free vertex; otherwise it is a part of exactly one component because components cannot overlap in this range by Corollary *2.2.6*.
- **Threshold case:** If $l = k$, then no vertex v contains a loop by Corollary *2.1.4*. For each single-element subset X, the *definition of sparsity* claims $i(X) \leq k - l = 0$, the initial single vertices are already tight subgraphs, which means that the whole V is covered by components (and there are no free vertices). Furthermore, components cannot overlap in the lower range by Corollary *2.2.6*, so each vertex is in exactly one component.
- **Upper range:** If $k < l < 2k$, then all vertices are loopless by Corollary *2.1.4*, so no single vertex can form a block. Furthermore, if a vertex v is not part of any components, then it is free; otherwise it is a part of some larger blocks.
- If $l = 2k - 1$, then as we mentioned in the upper range, no loops are possible. For $X := \{u, v\}$, the *definition of sparsity* is $i(X) \leq 2k - l = 2k - (2k - 1) = 1$, which prevents the presence of parallel edges and also implies that a single edge is always a block, and there are no free edges. Moreover, it follows that every edge belongs to one component. Finally, if a single vertex v is connected with another one, they already form a component, so v cannot be free.

□

2.3 Matroidal aspects

In this section, we show that tight graphs form the set of bases of a matroid. It ensures that the greedy algorithm, which considers the edges in an arbitrary order, also solves the optimization of the *fundamental problems*, because it allows us to produce the edges in decreasing order by the given weights.

Theorem 2.3.1 (Sparsity-matroid)
Assume that k, l and n satisfies:

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

Then the collection 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 $K_n^{k-l,2k-l}$.

In this theorem, we only claimed the ground set $K_n^{k-l,2k-l}$ produces all bases, however, we did not mention other multigraphs. The following shows that the matroid property holds for each (k, l)-tight multigraph:

- **Extension:** To enlarge the ground set by adding extra loops or parallel edges, the basis remains restricted to the number of edges required by the sparsity condition.
- **Reduction:** To shrink the ground set by deleting edges or loops from it, the basis is the maximal sparse subgraph of G.

2.4 Partitioning

Nash-Williams and Tutte studied coverings with k edge-disjoint forests in [3], and in [4]. In particular, they realized that the edge set of a graph contains k edge-disjoint spanning trees if and only if every partition of the vertex set V into p parts induces at least $k(p-1)$ edges. If, moreover, it has exactly $kn - l$ edges, then it is the edge-disjoint union of k spanning trees or equivalently a (k, k)-tight graph.

First, we define the partitions of the vertex set V of a graph $G = (V, E)$, then we give an overview of the connection between partitions and (k, l)-sparse graphs.

Definition 2.4.1
*In a multigraph $G = (V, E)$ a vertex set $P \subseteq 2^V$ is a **partition** of V if and only if the following conditions are met:*

1. *The empty set is not in P.*
2. *The vertex sets of P cover all the vertices of V.*
3. *The vertex sets of P are pairwise disjoint.*

Lemma 2.4.2
Assume that $G = (V, E)$ is a (k, l)-tight graph, $P = (V_1, \ldots, V_p)$ is a partition of the vertices of V. In addition, also assume that each $|V_i| \geq 2$ in the upper range. Then there are at least $l(p-1)$ edges between the partition sets V_i.

Proof.
Assume that the number of vertices and edges of a set V_i of P are n_i and m_i. By the (k, l)-*sparsity*, for the edges inside each partition holds $m_i \leq kn_i - l$, therefore the number of the inner edges is

$$\sum_{i=1}^{p} m_i \leq \sum_{i=1}^{p} (kn_i - l) = kn - pl.$$

It follows that the number of the edges among the partitions has a bound of

$$m - \sum_{i=1}^{p} m_i \geq (kn - l) - (kn - pl) = l(p - 1).$$

□

Lemma 2.4.3
Let $G = (V, E)$ be a (k, l)-tight graph. Then the degree of every vertex is at least k. Moreover, if $l > 0$, then there is at least one edge between a vertex v and the rest of vertices $V \setminus \{v\}$.

Proof.
Suppose that there exists a vertex v such that $\deg(v) =: d < k$. Then for the induced edges of the subgraph $V \setminus \{v\}$ would hold

$$kn - l - d > kn - l - k = k(n - 1) - l,$$

which contradicts the *sparsity* of G. As Corollary *2.1.4* states, if $l \geq k$, then the sparse graphs are loopless, which justifies the second part of the lemma in that range. The other case, when $0 < l < k$, comes from Lemma *2.4.2*. □

Corollary 2.4.4
If $l > 0$, then each (k, l)-tight graph is connected.

3 | Basic Pebble Game

Now we are ready to describe the Basic Pebble Game algorithm to identify the (k, l)-sparse graphs. This algorithm is able to solve the first four points of the *fundamental problems*. Later, we introduce the Component Pebble Game in Chapter *4* to speed up this basic version with components. The detailed description of this algorithm is given in Section *3.1*, and the correctness of this algorithm is proved in the Section *3.2*. Note that this chapter is a summary of [1].

The algorithm depends on the following parameters.

- $G = (V, E)$: the input undirected multigraph to be tested.
- k: the parameter of (k, l)-sparsity, practically, the initial number of pebbles on each vertex.
- l: the parameter of (k, l)-sparsity, practically, the strict lower bound on the total number of pebbles being present at the endpoints of an edge to be accepted.

All Pebble Game algorithms are built on the commonly known game, called Pebble Game. It is played on a board with the set V of n vertices, initialized with k pebbles on each. The player may insert edges and orient them. The rules of the game define whether an edge is accepted (and inserted with an orientation) or rejected, and allow to collect pebbles and reorient the already inserted arcs.

The game takes the given multigraph G as an input and considers its edges in an arbitrary order, which allows us to add the edges decreasing or increasing by their weights to solve the *optimization problem*. The algorithm performs the moves of the game to accept or reject each edge.

The following section contains the rules, moves, output and the description of the algorithm.

3.1 Description of the algorithm

Now we turn to the detailed description of the Basic Pebble Game algorithm. First, we determine the concrete rules and possible moves which are followed during the game, then introduce the algorithm in details.

A note on the inner digraph

The algorithm maintains an inner digraph $D = (V, A)$, which has the same vertex set V of $G = (V, E)$ and its arc set is initialized to be empty. It processes the edges of E one by one, and D assures a special structure in which the acceptance of the edges can be accomplished efficiently.

(a) *The input graph G.* **(b)** *The inner digraph D.*

Figure 3.1: *The inner digraph at a moment of the algorithm for $k = 2$, $l = 1$.*

This digraph also serves several purposes: an edge $e = uv$ of E is decided to be accepted by the total number of pebbles on its endpoints u and v, which may be increased by some breadth first search (BFS) algorithms

in D. If e is accepted, then it is inserted with an orientation into D. Throughout the game, D has a special orientation which assures that the underlying undirected graph of D is (k, l)-sparse. If an edge insertion into D would violate this orientation, then the edge is rejected, and the algorithm continues with the next edge.

Rules

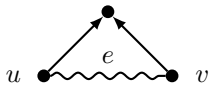
- **Pebbles:** At most k pebbles may be present on each vertex at any time. Denote the number of the pebbles on a vertex v by $\text{peb}(v)$.
- **Edge acceptance:** An edge uv is accepted for insertion if more than l pebbles are present on the vertices u and v. In case of a loop, count the pebbles of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

Allowed moves

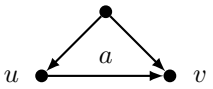
- **Pebble collection:** If an edge uv is not yet accepted, then an additional pebble may be collected on its endpoints u and v via (BFS) in D. It is performed from u and v by marking them as visited, so they will not be searched and their pebbles are protected from being moved. If one pebble is found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

Illustration for pebble collection

Note that as a result of the insertion and pebble collection processes, the following invariant holds throughout the game: $\text{peb}(v) + \deg^{out}(v) = k$. This invariant is precisely described and proved in Lemma *3.2.4*. The following figures give an example of the states of the inner digraph D before and after two pebble collections:



(a) *Before, there are only two pebbles on the endpoints u and v.*



(b) *After, the required four pebbles are present on u and v, therefore, e is accepted.*

Figure 3.2: *Two pebble collections in a $(2, 3)$-sparse graph to insert arc a.*

Edge e is processed in Algorithm *1* by checking the *edge acceptance*: $\text{peb}(u) + \text{peb}(v) > l$ is required, which is exactly equivalent with $2k - l > \deg^{out}(u) + \deg^{out}(v)$ by the above invariant. In this example, the parameters are $k = 2$ and $l = 3$, so the condition is $1 > \deg^{out}(u) + \deg^{out}(v)$. However, as Example (a) shows, $\deg^{out}(u) = \deg^{out}(v) = 1$, therefore, e cannot be accepted yet. Thus, a pebble collection is accomplished from both u and v to gather pebbles, that is, to decrease the values of $\deg^{out}(u)$ and $\deg^{out}(v)$ to zero. In Figure *3.2* (b), the process ends because the *edge acceptance* condition is met, therefore a is inserted.

Output evaluation

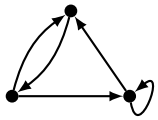
When every edge is processed, the output can be the followings depending on the final state of the game. If exactly l pebbles remained in the final state, and every edge has been inserted, then it is **well-constrained**; otherwise, if some edges have been rejected, then it is **over-constrained**. Otherwise, if more than l pebbles remained, and every edge has been inserted, then the output is **under-constrained**; otherwise, if some edges have been rejected, then it is **other**. These evaluations accurately correspond with tight, spanning, sparse and neither sparse nor spanning graphs, respectively, which is proved in Section *3.2*.

The following table contains the evaluation in a compact form.

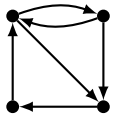
	Exactly l pebbles	More than l pebbles
All inserted	**Well-constrained**	**Under-constrained**
Some rejected	**Over-constrained**	**Other**

Illustration for evaluation

The following figure illustrates the possible final states of the algorithm.



(a) ***Well-constrained.***



(b) ***Under-constrained.***

(c) ***Over-constrained.***

(d) ***Other.***

Figure 3.3: *The possible outputs of the algorithm for parameters $k = 2$ and $l = 1$. The dashed edges have been rejected.*

The exact algorithm is the following.

Algorithm 1: Basic Pebble Game

Input: An undirected multigraph $G = (V, E)$, possibly with parallel edges and loops.
Output: *Well-constrained*, *under-constrained*, *over-constrained* or *other*.
Maintenance: Maintain the inner digraph D, a container of the current number of pebbles for each vertex, called `peb`, and the sum of the pebbles currently on the vertices, called `sum`.
Initialization: Let the inner digraph D be an empty graph on V, and place k pebbles on each vertex, that is, let `peb` be k for each vertex and `sum` be kn.
Method: Initialize the game and start processing the edges one by one in an arbitrary order by checking the *edge acceptance*. If the `sum` is at most l, then the remaining edges can be rejected immediately because the edge acceptance surely would not be met. Assume that `sum` $> l$ and the edge being processed is $e = uv$.

- If the edge acceptance condition holds for e, then it is accepted and *inserted*, furthermore, the `sum` and `peb` for the source of the *inserted* arc are decreased by one.
- Otherwise, attempt a *pebble collection* for the endpoints u and v.
  - If it fails to collect a further pebble, then e is rejected.
  - Otherwise, the total number of pebbles on the endpoints is increased and the edge is processed again.

Complexity analysis

Since each accepted edge requires the removal of a pebble and initially the total number of pebbles is kn, the number of inserted edges, m_a, is $O(kn)$. Note that this also follows from the *definition of sparsity* because the underlying undirected graph of the inner digraph is always (k, l)-sparse, therefore,

$$m_a \leq kn - l = O(kn).$$

- **Space complexity:** The additional structures are the inner digraph D, the container `pebbles` and the value `sum`. The size of the required storage of them is $O(n + m_a) = O(kn)$.

- **Time complexity:** As each edge is considered exactly once and requires at most $l+1$ *pebble collections* in D, which are essentially some BFS algorithms, an edge is processed in time $O(lm_a)$. As the graph has m edges, the total time of the algorithm is $O(mlm_a) = O(klmn)$.

In case of considering the input parameters k and l as constants, the running time of the algorithm is $O(nm)$. This will be slightly improved with Component Pebble Game in Chapter *4*.

3.2 Coincidence with sparse graphs

In this section, we prove that the classification of graphs given by Algorithm *1* coincide with tight, sparse, spanning and neither sparse nor spanning ones. First, we describe the main theorem, then we introduce some notations and prove basic lemmas. Afterwards, the main theorem follows as a corollary of the lemmas.

Theorem 3.2.1
The class of under-constrained Pebble Game graphs coincides with the class of sparse graphs, well-constrained ones coincide with tight graphs, over-constrained coincide with spanning ones and other are neither sparse nor spanning.

Definition 3.2.2
Let $v \in V$ be an arbitrary vertex of graph $G = (V, E)$. Then during the whole execution of Algorithm 1

- $\mathrm{peb}(v)$ *is the* ***number of pebbles*** *currently on v in D.*
- $i(v)$ *is the* ***number of induced edges*** *by v, so the number of loops currently on v in D.*
- $\mathrm{out}(v)$ *is the* ***number of outgoing edges*** *of v in D, excluding loops.*

The following definition is a natural extension of these concepts for subsets:

Definition 3.2.3
Let V' be an arbitrary nonempty subset of V in graph $G = (V, E)$. Then during the whole execution of Algorithm 1

- $\mathrm{peb}(V')$ *is the summary of* $\mathrm{peb}(v)$ *values for each vertex v in V'.*
- $i(V')$ *is the number of edges induced by V' in D, including loops.*
- $\mathrm{out}(V')$ *is the number of edges starting at a vertex in V', and ending at a vertex in $V \setminus V'$.*

Now we focus on proving the invariant properties of Algorithm *1*.

Lemma 3.2.4
During the whole execution of Algorithm 1,

$$\mathrm{peb}(v) + i(v) + \mathrm{out}(v) = k$$

holds for each vertex v.

Proof.
The invariant holds at the beginning of the game because each vertex v has exactly k pebbles, and D is empty. We prove that no process of the game changes the invariant.

- *Edge insertion:* A pebble is removed, and either i or out is increased, depending on whether the edge is a loop or not, so the total of them stays unchanged.
- *Pebble collection:* If v is an inner vertex of the path to be reversed, then $\mathrm{out}(v)$ does not change. If v is the source of the path reversal, then $\mathrm{out}(v)$ is decreased and $\mathrm{peb}(v)$ is increased by one; and vice versa, if v is the target.

Eventually, the sum $\mathrm{peb}(v) + i(v) + \mathrm{out}(v)$ remains constant k throughout the game for each vertex v. □

Lemma 3.2.5
During the whole execution of Algorithm 1,

$$\mathrm{peb}(V') + i(V') + \mathrm{out}(V') = kn'$$

holds for each subset V' of V, where $|V'| = n'$, such that $n' \geq 1$ in the lower range and $n' \geq 2$ in the upper range.

Proof.
Let $m_{V'}$ be the number of non-loop edges induced by V'. With this notation, it follows that $\mathrm{out}(V') = \sum_{v \in V'} \mathrm{out}(v) - m_{V'}$ and $i(V') = m_{V'} + \sum_{v \in V'} i(v)$. Therefore:

$$\mathrm{peb}(V') + i(V') + \mathrm{out}(V') = \mathrm{peb}(V)' + \left(m_{V'} + \sum_{v \in V'} i(v) \right) + \left(\sum_{v \in V'} \mathrm{out}(v) - m_{V'} \right) =$$

$$= \sum_{v \in V'} \mathrm{peb}(v) + \sum_{v \in V'} i(v) + \sum_{v \in V'} \mathrm{out}(v) = \sum_{v \in V'} \big(\mathrm{peb}(v) + i(v) + \mathrm{out}(v) \big) = kn'$$

□

Lemma 3.2.6
During the whole execution of Algorithm 1,

$$\mathrm{peb}(V') + \mathrm{out}(V') \geq l$$

holds for each subset V', where $|V'| \geq 1$ in the lower range and $|V'| \geq 2$ in the upper range. In particular, there are at least l free pebbles in D.

Proof.
At the beginning of the game, D is empty, so $\mathrm{out}(V') = 0$ and $\mathrm{peb}(V') = kn' \geq l$. Consider the last time an edge $e = uv$ incident to V' was *inserted* or *reoriented* by the algorithm. The following cases may occur:

- If both endpoints u and v are in V', then at least l pebbles must be on u and v after the *insertion*, so $\mathrm{peb}(V') \geq l$.
- If e went between V' and $V \setminus V'$ and it is oriented away from V', then a pebble was removed from V' and an outgoing arc was added.
- If e went between V' and $V \setminus V'$ and it is oriented towards V', then both the number of pebbles and the number of outgoing arcs remained unchanged.
- If e were reoriented, then it would either bring in a pebble inside V' and decrease the number of outgoing arcs, or vice versa.

All the above cases preserve the invariants, which had to be proved. □

Now we focus on showing some important corollaries.

Corollary 3.2.7
During the whole execution of Algorithm 1,

$$i(V') \leq kn' - l$$

holds for each subset V' of V, where $|V'| = n'$, such that $n' \geq 1$ in the lower range and $n' \geq 2$ in the upper range.

Proof.
As a result of Lemma *3.2.5* and Lemma *3.2.6*, the following holds:

$$i(V') = kn' - (\mathrm{peb}(V') + \mathrm{out}(V')) \leq kn' - l.$$

□

Corollary 3.2.8
A subset V' of V with n' vertices, such that, $n' \geq 1$ in the lower range and $n' \geq 2$ in the upper range, induces a block if and only if $\text{peb}(V') + \text{out}(V') = l$.

Corollary 3.2.9
Under-constrained pebble game graphs are sparse, well-constrained ones are tight, over-constrained ones are spanning.

This corollary proves that the output classifications of Algorithm *1* are well-defined, so the first direction of Theorem *3.2.1* is proved. We only have to prove that the algorithm identifies the sparse, tight and spanning graphs correctly.

Now we need another notation:
Definition 3.2.10
Let D be the inner digraph at any point of the execution. The ***reachable region*** *of a vertex v in D is the set of vertices that can be reached from v in D. It is referred to as $Reach(v)$.*

Lemma 3.2.11
If $e = uv$ is a free edge (but not yet inserted) and the total number of pebbles on the endpoints u and v is at most l, then a pebble can be brought to one of u or v.

Proof.
Denote the union of *reachable regions* $Reach(u) \cup Reach(v)$ by V'. Observe that there must be a vertex in $V' \setminus \{u, v\}$, otherwise, with at most l pebbles in u and v, V' would be a component, therefore, e would not be *free*. By assumption, e is *free*, so $i(V') < k|V'| - l$. As no arcs leave the *reachable region*, $\text{out}(V') = 0$. By Lemma *3.2.6* and using that V' does not form a block, $\text{peb}(V') > l$ and by assumption, $\text{peb}(u) + \text{peb}(v) \leq l$. It follows that there exists a vertex $w \in V' \setminus \{u, v\}$ with at least one free pebble, so it can be collected on either u or v. □

Corollary 3.2.12
An edge is accepted by Algorithm 1 if and only if it was free in D.

Proof.
Assume that the edge is denoted by $e = uv$. Apply Lemma *3.2.11* at most $l + 1$ times. As a corollary, $l + 1$ pebbles can be *collected* on the endpoints u and v, so e can be *inserted*. □

The following lemma is the main theorem, which justifies that Algorithm *1* classifies each graph correctly. It shows the other direction of Theorem *3.2.4*.

Lemma 3.2.13
The pebble game returns under-constrained for sparse, but not tight graphs, well-constrained for tight ones, over-constrained for spanning graphs and other for graphs that are neither spanning nor sparse.

Proof.
As we mentioned in Theorem *2.3.1*, the sparse graphs form a matroid, so the greedy algorithm works: their edges can be processed in an arbitrary order. Assume that the input graph G has n vertices and m edges and D denotes the inner digraph when the algorithm terminates. We use the result of the previously-proved invariant, described in Lemma *3.2.5*. The following types of graphs should be classified:

- Assume that G is (k, l)-sparse, but not tight. Then by the above-mentioned lemma
$$kn - l \overset{\text{def.}}{>} m = i(V) \overset{\text{Lemma}}{=} kn - \text{peb}(V) - \text{out}(V) = kn - \text{peb}(V),$$
from where $\text{peb}(V) > l$ and the output is *under-constrained*.
- If G is (k, l)-tight, then the number of pebbles is exactly l and the output is *well-constrained*.
- If G is (k, l)-spanning, then it contains a tight subgraph, which is *inserted* into D with a remainder of l pebbles. After that state, there is not enough pebbles for the further *insertions*, and some edges will be rejected. It follows that the result of the game is *over-constrained*.
- If G is neither sparse nor spanning, then there must be more than l pebbles in the final game, and some edges are rejected as well. The output of the algorithm is *other*.

□

4 Component Pebble Game

In this chapter, we investigate some of the applications of the components. In particular, an improved pebble game algorithm will be given in the Unweighted Pebble Game in Section *4.2* whose running time is $O(n^2+m)$, which improves upon the performance of the Algorithm *1* for dense graphs. The main ingredient of this enhanced algorithm is that the rejection of an edge can be performed by checking if it is induced by a component, which will involve no pebble collections. This chapter partially builds on [1].

As an introduction to the topic, we prove two frequently-used lemmas:

Lemma 4.0.1
The number of the components during the whole execution of Algorithm 1 is $O(kn)$.

Proof.
As a straightforward corollary of the *definition of sparsity*, the total number of edges during the whole execution has an upper bound of $kn-l$, because the inner graph is always (k,l)-sparse. Since the formation of a component is always preceded by an *edge insertion* and each edge insertion leads to the formation of at most one component, the number of them is also linear. □

We will also need the following lemma, which we prove based on personal communication with András Mihálykó.

Lemma 4.0.2
The total size of the vertex sets of the components during the whole execution of Algorithm 1 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 every edge is included in at most one component. □

Corollary 4.0.3
The number of intersections of the components is $O(n)$.

Proof.
Suppose that it is not linear. Then the total size of the vertex sets of the components would not be linear, which would contradict Lemma *4.0.2*. □

Now we introduce the Basic Component Pebble algorithm in Section *4.1*, then we describe a special case called Unweighted Pebble Game in Section *4.2*.

4.1 Basic Component Pebble Game

In this section, we summarize the results of [1] and make same observations on their results regarding component maintenance in the upper range from [10].

Data structures for constant queries

This paragraph describes a data structure to determine in constant time whether two vertices belong into the same component. By Corollary *2.2.6* and Lemma *2.2.7*, components may intersect in different ways for different ranges, therefore we must take l into account:

- If $l = 0$, then there is at most one component, which can be maintained by a **marking scheme**: a vertex is marked if and only if it is in the component. Maintain an indicator container, called `marked`, which assigns true for a vertex v if and only if it is in the component.
- If $0 < l \leq k$, then the components are vertex-disjoint, so they may b maintained by a **labeling scheme**: each vertex is labeled with the *id* of the component to which it belongs. Denote the *id* of a vertex v be id_v. Maintain an integer container, called `labeled`, which assigns a value k for a vertex v if and only if v is in the component with id k.
- If $k < l < 2k$, then the components may overlap in at most one vertex. Maintain an indicator matrix `M` indexed by the vertices: let `M[u][v]` be true if and only if u and v belong into the same component.

In addition, we also maintain a `list of components` for the upper range in which we store the vertex sets of the components.

Description

The algorithm is the following:

Algorithm 2: Basic Component Pebble Game

Input: An undirected multigraph $G = (V, E)$, possibly with parallel edges and loops.
Output: *Well-constrained*, *under-constrained*, *over-constrained* or *other*.
Initialization: Initialize the inner digraph and the pebbles similar to Algorithm *1*. In addition, maintain the `list of components` and the constant time data structures depending on the ranges. Initialize them according to the followings:

- If $l = 0$, then let `marked(v)` be false for each vertex v.
- If $0 < l \leq k$, then let `labeled(v)` be -1 for each vertex v.
- If $k < l < 2k$, then let `M[u][v]` be false for each pair u and v of vertices.

Same component: Let u and v be two (not necessarily different) vertices in D. Then the condition of u and v being in the same component is the following:

- If $l = 0$, and both `marked(u)` and `marked(v)` are true or
- If $0 < l \leq k$, and `labeled(u)`=`labeled(v)` and they are non-negative or
- If $k < l < 2k$, and either `M[u][v]` or `M[v][u]` is true.

Method: Play Algorithm *1* with the following modifications. Let $e = uv$ be the current edge.

- If u and v are in a same component, or $u = v$ and $l \geq k$, then e is rejected.
- Otherwise, e is *free*, therefore it is *insertable*. Run *pebble collection* (to find the proper orientation of D) until the *condition* of the edge insertion is met, then *insert* it. Detect new components by Algorithm *3*, if some are formed, and perform necessary component maintenance by Algorithm *4*.

The *complexity analysis* of this algorithm is presented below.

We note that if e is an incident edge of u and it is a loop, then the used data structures would not indicate correctly that e is non-free in the case of $l \geq k$, because of their initializations. At the beginning of the algorithm, no components contain u, so e seems like an insertable edge, however, it is not by Corollary *2.1.4*. Hence, this case is specifically prohibited for loops by the further condition of $u = v$ and $l \geq k$.

Component detection

Algorithm description

The following algorithm is called right after an *insertion* of edge $e = uv$ and uses a BFS algorithm on the reversed digraph of D to find the component of u and v.

Algorithm 3: Component detection

Input: The inner digraph $D = (V, A)$, into which edge $e = uv$ has just been inserted.
Output: The vertex set V' of the new component induced by e, if one was formed; $\emptyset$, otherwise.
Method:

- If more than l pebbles are present on u and v, then return $\emptyset$, because the new edge is *free*.
- Otherwise, compute $Reach(u, v)$, the vertex set of *reachable vertices* from u and v in D.
  - If any $w \in Reach(u, v)$ has at least one free pebble, then return $\emptyset$, because as a result of a *pebble collection* from w, the edge may be *free*.
  - Otherwise, create D' by reversing the direction of each arc of A. From each vertex $w \in V \setminus Reach(u, v)$ with at least one free pebble, perform a BFS in D'. Return the set of non-visited vertices.

Complexity analysis

Checking the condition of the pebbles can be done in constant time. Determining the *reachable region* of u and v requires $O(n+m_a) = O(kn)$ time, and their pebbles are checked in linear time. Producing the reversed digraph takes time $O(m_a) = O(kn)$, and the other BFS is in time $O(kn)$, therefore, the overall time is also $O(kn)$.

Illustration

The following illustration shows an example of Algorithm *3* for a $(1, 0)$-sparse graph. Assume that edge uv has just been inserted into D.

(a) *The inner digraph D.*

(b) *The reversed digraph D'.*

Figure 4.1: *The component detection algorithm first determines that uv is non-free, then creates D' by reversing each arc of D (see Figure (b)). It starts a BFS algorithm in D' from each vertex $w \in V \setminus \{u, v\}$ with at least one free pebble to detect the maximal block (therefore component) containing u and v. In this example, there is no vertex w with the above properties, so every vertex is non-reached and the algorithm returns with the whole set of V. Note that the figure shows a $(1, 0)$-tight graph, so the detection succeeded.*

Updating the components

The following lemma is used in Algorithm *4* below.

Lemma 4.1.1
Assume that Algorithm 3 has just run to detect the components. Then a component $C := \{v_1, \ldots, v_t\}$ is induced by the detected vertices if and only if $t = 1$ and v_1 is detected, or $t > 1$ and there exist indices p, q such that $1 \leq p \neq q \leq t$ and both v_p and v_q are detected.

Proof.
The necessity is trivial, we show the sufficiency. If $t = 1$ and v_1 is detected, then C is induced. If $t > 1$, then by assumption, both v_p and v_q are detected. By Corollary *2.2.6*, components may overlap in at most one vertex, so any two vertices determine obviously a component. □

Data structures

Assume that Algorithm *3* has just been run and the *id* of the found component is i. As we use different data structures, the updates are also accomplished separately:

- If $l = 0$, then the process can easily be done by setting `marked` true for the newly detected vertices in time $O(n)$.
- If $0 < l \leq k$, then the process contains about rewriting `labeled` for the detected vertices by i, such that, $id_u = i$ for each detected vertex u in time $O(n)$.
- If $k < l < 2k$, then the updating process is described by Algorithm *4*.

Algorithm description

For the following algorithm, assume that there exists an ordering between the vertices of V by the *id* of the vertices, such that $u < v$ if and only if $id_u < id_v$. This algorithm is our own method that describes the updating process of the components in the upper range in time $O(n^3)$. It also contains some tricks to speed it up in practice, however, the best bound we could prove on the worst case running time is $O(n^3)$. The results of this algorithm in [1] and in [10] are further analyzed in the part of *observation of a problem* below.

The main problem with the following algorithm is the total number of overwrites, that is, the true-to-true markings in some entries of matrix `M`. No matter how hard we tried, we could not figure out either a better algorithm for updating or a more precise upper bound for the total number of overwrites.

Algorithm 4: Updating of the components in the upper range

Input: The detected vertices from Algorithm *3*, the indicator matrix `M` and the `list of components`.
Output: The updated matrix `M` and `list of components`.
Method: If no detected vertex has been found, then the algorithm is terminated.

1. Create two containers to store the detected vertices, called `detected` and `frees`, depending on whether a vertex v was free before the *insertion*, that is, whether v was not induced by any components.
2. Mark the entries of matrix `M` for each pair u and v of vertices, such that, $u < v$ as true, that is, set `M[u][v]` true.
3. Also create a container, called `marked components` which is a collection of the induced components. As an application of Lemma *4.1.1*, it can be accomplished by checking only the first two vertices of each component (or in one-sized components the first one).
4. Initialize another container for each vertex v, called `first component`, by iterating backwards on each vertex v of the `marked components` C, and setting the `first component` of v to the *id* of C. Note that the backwards iteration assures that if a vertex belongs into more than one component, then the `first component` contains exactly the first one.
5. Start merging the components: for each vertex v_i of the `marked component` C_i mark all vertices u of `frees` in `M[u][`v_i`]` as true, then check if C_i is the `first component` of v_i:
   - If not, then v_i has a more recent component than C_i, where the update of `M` is accomplished. Continue with the next vertex of C_i.
   - Otherwise, pick each `marked component` C_j such that $i < j$ is met. For each vertex $v_j \in C_j$, set the entry of the matrix `M[`v_i`][`v_j`]` to true.

Complexity analysis

Note that these updates are executed if and only if a new component is *detected* after an *edge insertion*, so it is called at most $O(kn)$ times.

1. The containers are created in time $O(n)$, because every vertex is scanned exactly once, so overall $O(kn^2)$ times.

2. During the whole execution, the total number of free vertices is $O(n)$, so the marking process of the pairs is $O(n^2)$.

3. Creating `marked components` is accomplished in time $O(n)$, so in total $O(kn^2)$ time.

4. The `first components` can also created in time $O(n)$ because the number of stored vertices in each component is linear by Lemma *4.0.2*.

5. The process of Descartes production has a worst case of $O(n^3)$: it is trivial that the number of one-to-one-overwrites in the matrix `M` is at most n for each pair of entries, so the total time is $O(n^3)$.

Illustration

The following figure shows an example for Algorithm *4*. After the insertion of edge e, Algorithm *3* identifies the newly formed component C. Afterwards, the update is called

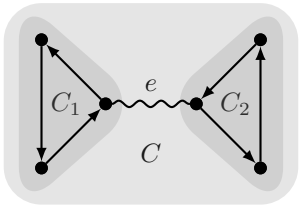


Figure 4.2: *An update process in a $(2,3)$-sparse graph, in which components C_1 and C_2 are treated as* `marked components`*, and the entries of* `M` *are set to true for each pair of vertices $v_1 \in C_1$ and $v_2 \in C_2$, such that, $v_1 < v_2$.*

Correctness

Now we are going to prove the correctness of Algorithm *2* via Algorithm *1*.

Theorem 4.1.2
The graphs recognized by Algorithm 2 are the same as graphs recognized by Algorithm 1, and components are correctly computed.

Proof.
The main difference between the Basic Pebble Game and the Component Pebble Game is the maintenance of the components, and the rejection of non-free edges in constant time. First, observe that Algorithm *2* detects a maximal connected subgraph $G' = (V', E)$ with no outgoing edges in which exactly l pebbles are present. By Lemma *3.2.5*, the invariant of $\text{peb}(V') + i(V') + \text{out}(V') = kn'$ is met, from where G' is a *block*, because it induces exactly $kn' - l$ edges. By Lemma *2.2.7*, the followings hold:

- If $l = 0$, then there may be at most one component, because the union of two blocks also forms a block, and the algorithm detects it.

- If $l > 0$, then components are connected, and it is also detected.

The visited vertices in Algorithm *3* are the ones that can reach a pebble on a vertex different from u and v, and form the complement of the unique component containing e. □

Complexity analysis of Algorithm *2*

The non-free edges are rejected in constant time. The *free* ones are *inserted* right after the successful *pebble collections*. Denote the number of inserted edges by m_a. After each insertion, Algorithm *3* and *updating of the components* are called. Since each detection requires a BFS call, the running times of them are $O(n + m_a) = O(kn)$. As the *updating process* has a worst case of $O(n^3 + m)$, the running time of Algorithm *2*

in the worst case is $O(n^3 + m)$.

The space complexity of the lower range is $O(n)$ and in the upper range is $O(n^2)$, due to its indicator matrix.

Observation of a problem

In [1] and [10], the authors claimed that their algorithm runs in time $O(n^2)$, however, their proof regarding the data structure they used is based on the following incorrect assumption. Their solution in [10] uses the so-called bounded union pair-find method which requires the bounded property: $|C_i \cap C_j| \leq 1$ for each $i \neq j$. However, when we take the union of more than two components, this property may be violated. The following figure illustrates an example for the problem:

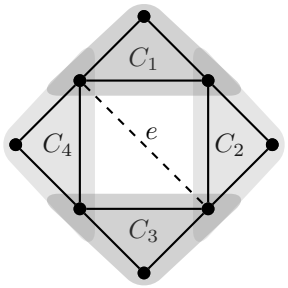


Figure 4.3: *A $(2,3)$-sparse graph in which the inserted edge e provokes the union of the components.*

In Figure *4.3*, inserting edge e into the $(2,3)$-sparse graph involves the union of four components. Assume that the union method described in [10] is called in the order of the components indices. Denote the union of C_1, C_2 and C_3 by C. After creating C, the intersection of C 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 [10].

4.2 Unweighted Pebble Game

In this section, we design the Unweighted Pebble Game algorithm for the whole range of $0 \leq l < 2k$ with a space complexity of $O(n)$ and a running time of $O(n^2+m)$. It uses the basic subalgorithms of the Component Pebble Game, which are referenced from Section *4.1*. We only introduce the algorithms that differ in any ways from the ones used within the Component Pebble Game. Note that this algorithm can solve each of the *fundamental problems*, except the optimization problem, because this algorithm inserts the edges in a special order.

We note that there exists a similar algorithm in [10] which also solves the same *fundamental problems* in a space complexity of $O(n + m)$. However, their algorithm stores the sets of the induced edges of each component, unlike ours, which only stores the vertex sets of them.

The data structure

The principal idea consists in iterating on each vertex and their incident edges. To implement, use the following special data structure.

Definition 4.2.1
Let $\mathtt{x}_u$ be the ***characteristic vector of the union of the components*** *containing the vertex u being processed.*

By definition, $\mathtt{x}_u$ determines whether vertices u and v are in the same component, so an incident edge uv of u may be rejected or inserted in constant time. The updating process of this data structure is different from the so-called union pair-find in [10], because we store the vertex sets of the components instead of their induced edge sets. Note that $\mathtt{x}_u$ is essentially the row of the vertex u of the matrix `M` in the upper range of the Component Pebble Game in the previous section.

Algorithm description

The following description gives an overview of the algorithm.

Algorithm 5: Unweighted Pebble Game

Input: An undirected multigraph $G = (V, E)$, possibly with parallel edges and loops.
Output: *Well-constrained*, *under-constrained*, *over-constrained* or *other*.
Initialization: Initialize the inner digraph and the pebbles similar to Algorithm *1*. In addition, maintain the `list of components`, the special vector `x` and a further variable called `processed` for each edge to indicate whether it is already processed.
Method: Play Algorithm *1* with the following modifications. When considering a new vertex u, define $\mathtt{x}_u$ by Algorithm *6* and start processing each incident edge uv of u. First check whether uv is already `processed`. If so, then continue with the next edge (and possibly with the next vertex); otherwise, mark uv as `processed`, and check if u and v are in the same component using $\mathtt{x}_u(v)$:

- If $\mathtt{x}_u(v)$ is true or $u = v$ and $l \geq k$, then uv is rejected.
- Otherwise, start *pebble collection* while the *condition* of the edge insertion is met, then *insert* it. The success of the *collections* is proved *below*. After the insertion, detect new components by Algorithm *3*, if some are formed, and perform necessary maintenance by Algorithm *7*.

The *correctness* of the algorithm and the *complexity analysis* are shown later.

Updating processes

The following algorithms take care of updating vector `x` before a new vertex u is considered and updating the components and $\mathtt{x}_u$ after each insertion.

Algorithm 6: Defining `x`

Input: The new vertex u, the `list of components` and the indicator vector $\mathtt{x}_u$.
Output: The well-defined vector $\mathtt{x}_u$.
Method: For defining $\mathtt{x}_u$, first let $\mathtt{x}_u(v)$ be false for each vertex v. Then iterate through all `list of components` to find the ones that contain u and mark all vertices of them as true.

Complexity analysis

The false updates are done in time $O(n)$. By Lemma *4.0.2*, the iteration on each vertex of all components takes $O(n)$ time, so the time complexity of the whole algorithm is also $O(n)$.

Algorithm 7: Updating of the `list of components` and `x`

Input: The indicator vector $\mathtt{x}_u$ and the detected vertices from Algorithm *3*. Assume that edge uv has just been inserted into D.
Output: The updated `list of components` and indicator $\mathtt{x}_u$.
Method: First iterate through the `list of components` and check whether they contain a detected vertex. If so, then delete the whole `list of component`. Afterwards, for each detected vertex v set $\mathtt{x}_u(v)$ to true. Finally, add the detected vertices as a new set into `list of components`.

Complexity analysis

By Lemma *4.0.2*, the iteration on each vertex of each component is in time $O(n)$, the deletions of some components are accomplished in time $O(n)$. Also, setting `x` to true and creating the new component is in time $O(n)$, so the algorithm runs in linear time.

The success of the *pebble collections*

The following lemma proves that *pebble collections* succeed in Algorithm *5*.

Lemma 4.2.2
Denote the vertex being processed by u and an incident edge of u by $e = uv$. Assume that $\mathtt{x}_u(v)$ is false and if e is a loop, then $l < k$. Then the pebble collections for the insertion of e are guaranteed to succeed in Algorithm 5.

Proof.
It is sufficient to show that the edge is *insertable*. By Corollary *2.2.6*, e is *insertable* if and only if it is *free*.

- If e is the first edge of u, then:
  - If e is not a loop, then as corollary of the limited interval of $l \in [0, 2k)$, first non-loop edges are always *insertable*.
  - If e is a loop, then by assumption, $l < k$. This requirement implies that the *condition of insertion* holds, because it is the first incident edge of u and it initially has k pebbles, therefore e is insertable immediately.
- If e is not the first one, then there are already inserted incident edges of u. It means that Algorithm *3* have been accomplished at least once, and `x` is up to date. As $\mathtt{x}_u(v)$ is false, u and v are not parts of a same component, so e is *free*, and therefore insertable.

□

The correctness

Now we prove that Algorithm *5* also classifies the graphs well.

Theorem 4.2.3
Algorithm 5 and Algorithm 2 recognize the same graphs and computes the same components.

Proof.
The only differences between the algorithms are the order of the processed edges, and the used data structures. The special order of edges allows us to maintain only the actual row of `M` and consider it as a vector `x`. It is trivial that the *matrix structure* can be used in all ranges, so the correctness for lower range is also provided. Now it is sufficient to show that the *updating processes* are accomplished similarly to the *matrix structure*.

Algorithm *6* updates the newly considered vertices by scanning along all components, so `x` is really the characteristic vector of the union of the components containing the vertex begin processed as its *definition* claims. After the *insertion of an edge*, Algorithm *7* is called to update the `list of components` and `x`. If a new component C is formed, then both the `list of components` and `x` are updated by deleting the old components which are merged into C, inserting C to the list of the components, and marking the vertices v of C as true in $\mathtt{x}_u(v)$ to indicate that u and v is now in the same component. This process correctly updates each data structure. As the algorithms are similar in every other way, we reference to the proof of Theorem *4.1.2*. □

Complexity analysis

Algorithm *6* for a vertex u is accomplished in time $O(n)$, so the overall time is $O(n^2)$. Each *edge insertion* requires at most l *pebble collections* in time $O(kln)$. After that, a *component detection* is done in time $O(kn)$, and the *updating processes* are accomplished in time $O(n)$. As a summary, for each inserted edge, the time is $O(k^2ln^2)$, therefore the whole algorithm has a time complexity of $O(k^2ln^2 + m)$, which is $O(n^2 + m)$ with constant k, l parameters.

Since we only used the additional *data structure* $\mathtt{x}$, the algorithm requires $O(n)$ space.

5 Applications

In this chapter, we give a short summary about some applications of (k, l)-sparsity. First, we give an efficient algorithm for finding edge-disjoint forests covering the edge set of a given graph G via Basic Pebble Game, then we describe the concept of arboricity and finally we show some rigidity applications.

Throughout this chapter, all graphs are assumed to be loopless.

5.1 Covering with edge-disjoint forests

In this section, we sum up an algorithm for finding k edge-disjoint forests whose union covers the whole edge set of a given graph. The approach consists of two main parts: the Pebble Game algorithm and an algorithm for finding disjoint arborescences. We proceed with the latter in the following section.

5.1.1 Disjoint arborescences

Definitions and notations

Now we describe the basic definitions for directed graphs. The followings are from [11].

A directed graph is a **rooted tree** if the underlying undirected graph is a tree, and has a special root from which every other vertex is reachable in directed sense. A directed graph is a **rooted forest** if the underlying undirected graph is a forest whose components are rooted trees.

An $s-t$ **cut** $C = (S, T)$ in a digraph $D = (V, A)$ is a partition of V into two disjoint subsets S and T such that $s \in S$ and $t \in T$. The **cut-set** of an $s-t$ cut $C = (S, T)$ in a digraph is the set of arcs uv such that $u \in S$ and $u \in T$.

The followings are the mainly used definitions in this section.

Definition 5.1.1
*An **arborescence** of a digraph $D = (V, A)$ is a set B of arcs such that (V, B) is a rooted tree. An **r-arborescence** is an arborescence with a root r. A **partial r-arborescence** B of a digraph $D = (V, A)$ is a set B of arcs if B is an r-arborescence in the subgraph induced by $V(B)$.*

Definition 5.1.2
*A **branching** in a digraph $D = (V, A)$ is a set B of arcs such that (V, B) is a rooted forest.*

Solvability theorem

The following theorem claims that k edge-disjoint r-arborescences may be found in polynomial time:

Theorem 5.1.3
In a given digraph $D = (V, A)$, k edge-disjoint r-arborescences can be found in time $O(k^2m^2)$.

Before proving this theorem by showing an algorithm, we introduce some notations and lemmas.

Definition 5.1.4
For a digraph $D = (V, A)$, a vertex set $U \subseteq V$ and an arc set $H \subseteq A$, let $\delta^{in}_H(U)$ consists of those arcs $uv \in H$ for which $u \in V \setminus U$ and $v \in U$. Let $d^{in}_H(U)$ denote the cardinality of $\delta^{in}_H(U)$. If the subscript H is omitted, then H is considered as the whole arc set A.

Theorems of branchings

The following theorem of Jack Edmonds characterizes the existence of edge-disjoint branchings with prescribed root sets in a digraph:

Theorem 5.1.5 (Edmonds' disjoint branchings theorem)
Let $D = (V, A)$ be a digraph and let $R_1, \ldots, R_k$ be subsets of V. Then there exist disjoint branchings $B_1, \ldots, B_k$ such that B_i has root set R_i for each $1 \leq i \leq k$ if and only if for each nonempty $U \subseteq V$

$$d^{in}(U) \geq \big|\{i : R_i \cap U = \emptyset\}\big|.$$

We will need the weak form of this theorem in which each $R_i = r$, when we want to find k edge-disjoint r-arborescences.

Theorem 5.1.6
Let $D = (V, A)$ be a directed graph with a specified root $r \in V$, and k be a positive integer. Then D has k edge-disjoint r-arborescences if and only if D is rooted k-edge-connected, that is, for each nonempty $U \subseteq V \setminus \{r\}$

$$d^{in}(U) \geq k.$$

Now we focus on describing another theorem about covering with r-arborescences due to Vidyasankar, which will also be used in Algorithm *10* for covering with k edge-disjoint forests via Pebble Game.

Theorem 5.1.7 (Vidyasankar)
Let $D = (V, A)$ be a digraph, the root be $r \in V$, and k be a positive integer. Then A can be covered by k r-arborescences if and only if the following two conditions hold:

$$\deg^{in}(v) \leq k \text{ for each } v \in V \text{ and } \deg^{in}(r) = 0 \tag{5.1}$$

$$\sum_{v \in H(U)} (k - \deg^{in}(v)) \geq k - d^{in}(U) \text{ for each nonempty subset } U \subseteq V \setminus \{r\}, \tag{5.2}$$

where $H(U)$ denotes the set of outneighbours of $V \setminus U$.

Note that if $\deg^{in}(v) = k$ for each $v \in V \setminus \{r\}$, then condition (5.2) coincides with the condition in Theorem *5.1.6*, because it simplifies to $d^{in}(U) \geq k$ for each nonempty subset U of $V \setminus \{r\}$. This observation will also be useful for Algorithm *10*.

Algorithm for finding disjoint arborescences

The algorithm for finding k edge-disjoint r-arborescences in a loopless digraph is the following:

Algorithm 8: Finding k edge-disjoint r-arborescences

Input: A directed loopless multigraph $D = (V, A)$.
Output: A classification of the arcs into at most one of the k r-arborescences if it exists.
Method: First, we try to find a $\mathtt{B}$ r-arborescence such that for each nonempty $U \subseteq V \setminus \{r\}$

$$d^{in}_{A \setminus \mathtt{B}}(U) \geq k - 1. \tag{$*$}$$

Initialize $\mathtt{B}$ for $\mathtt{B} := \emptyset$ and start searching an arc a leaving the vertex set $V_{\mathtt{B}}$ of $\mathtt{B}$ for which there does not exist $U \subseteq V \setminus \{r\}$ such that $a \in \delta^{in}(U)$ and $d^{in}_{A \setminus \mathtt{B}}(U) = k - 1$. The condition for an arc a can be tested with Algorithm *9* defined below. The growing process of the current partial r-arborescence $\mathtt{B}$ is terminated in the following cases:

- If Algorithm *9* returns with fewer than $k - 1$ paths for all arc a leaving $V_{\mathtt{B}}$. In this case, no arcs can be inserted into $\mathtt{B}$, meaning that there do not exist k edge-disjoint r-arborescences (see [11, Page 918]).
- If $V_{\mathtt{B}}$ covers the whole vertex set V. In this case, mark the edges of $\mathtt{B}$ by k and iterate for the decreased $k - 1$ value on $A \setminus \mathtt{B}$ (until $k > 0$). Note that this recursive iteration is assured by $(*)$.

Complexity analysis

As each arc is considered at most once, processing an arc requires at most one call of Algorithm *9*, which runs in $O(km)$ time. Adding at most m arcs into one of the arborescences takes time $O(km^2)$, therefore the overall time is $O(k^2m^2)$.

The minimum cut algorithm

Note that an arc $a = uv$ can be inserted into the partial r-arborescence `B` if and only if $d^{in}_{A\setminus \mathtt{B}}(U) \geq k$ for all $U \subseteq V \setminus \{r\}$ which is entered by a. In other words, it means that there exist at least $k-1$ edge-disjoint paths from the root r to each vertex v, excluding a. The following algorithm determines the number of these paths. The correctness of this algorithm is discussed by Lemma *5.1.8*.

Algorithm 9: Minimum cut

Input: Digraph D and an arc $a = uv$ to possibly insert into `B`.
Output: The total number of edge-disjoint rv paths in D.
Method: Maintain an indicator vector for each arc, called `reversed`, and initialize it to false. To determine the total number of edge-disjoint paths without a, first erase a, then search $r \to v$ paths in D by BFS, reverse them until one exists and their total number is fewer than k, and negate the indicator `reversed` for each arc of the paths. Finally, if no more paths can be found, reverse all arcs with true `reversed` value and if the number of found paths is fewer then $k-1$, insert back a. Return with the total number of found paths.

Complexity analysis

Each path is found with BFS in time $O(m+n) = O(m)$. As the algorithm finds at most k paths in D, the complexity is $O(km)$.

Lemma 5.1.8
Assume that Algorithm 9 algorithm has just run in digraph D with root r to determine the minimum cut. Then arc $a = uv$ can be inserted into `B` keeping property $(*)$ if and only if the algorithm found at least $k-1$ edge-disjoint paths from $r \to v$, excluding a.

5.1.2 Covering with k edge-disjoint forests

In this section, theorems of Nash-Williams and Edmonds are presented. They gave equivalent conditions for covering the edge set with k edge-disjoint forests and finding k edge-disjoint arborescences with (k,k)-sparsity.

Nash-Williams theorem

First, we define the basic notations on undirected graphs for the following theorem.

A **tree** is an undirected graph that is connected and has no circuits. A **forest** is an undirected graph that has no circuits, in other words, whose components are trees.

The following theorem of Nash-Williams was published in [3].

Theorem 5.1.9 (Nash-Williams)
An undirected loopless multigraph $G = (V,E)$ is the union of k edge-disjoint forests if and only if for all nonempty $X \subseteq V$

$$i(X) \leq k|X| - k.$$

Now we introduce an algorithm with Pebble Game as an application of Algorithm *8*.

Algorithm 10: Covering with k edge-disjoint forests via Pebble Game

Input: An undirected loopless multigraph $G = (V, E)$.
Output: A classification of the edges into exactly one of the k edge-disjoint forests, if it exists.
Method:

1. Run Algorithm *5* on G for (k, k)-sparsity to determine the inner directed graph $D = (V, A)$. If G is (k, k)-sparse, then by Nash-Williams theorem (see Theorem *5.1.9*), E can be partitioned into k edge-disjoint forests; otherwise the algorithm is terminated by false.
2. Add a further root r, and for each vertex $w \in V \setminus \{r\}$ add arc wr with multiplicity $k - \deg^{out}(w)$.
3. Run Algorithm *8* on the reversed digraph $D' = (V, A)$ of D. Note that as a corollary of Vidyasankar's theorem, arc set A can be covered by k edge-disjoint r-arborescences. Delete root r and the further added arcs from D', and forget the classifications of these arcs. Then the edges of the underlying undirected graph of D' has the correct classifications.

Complexity analysis

The execution time of Algorithm *5* is $O(k^3n^2 + m)$ if $l = k$. As Algorithm *8* runs on the inner digraph $D = (V, A)$, the number of arcs is linear: $|A| = m_a = O(kn)$. The running time of Algorithm *8* on D is $O(k^4n^2 + m)$, therefore the overall time of thisn algorithm is $O(k^4n^2 + m)$.

This approach can be easily modified to find k-disjoint (minimum cost) spanning trees in a graph: in the first step of Algorithm *10*, we find a (minimum cost) (k, k)-tight subgraph with the Pebble Game algorithm. In the weighted case, we need to find a minimum cost (k, k)-tight subgraph, which takes $O(k^2mn)$ time, and hence the total running time of the algorithm is $O(k^4n^2 + k^2mn)$.

Note that the root set of the forests can also be produced during the third step of Algorithm *10* by checking the outgoing arcs from the further root r.

5.2 Arboricity

In this section, we make a summary of arboricity in undirected, loopless multigraphs. Note that it differs from r-arborescences and branchings in directed graphs in Section *5.1*.

Basic notations and definitions

Now we describe the most important concepts for arboricity.

Definition 5.2.1
*A **spanning tree** of a multigraph $G = (V, E)$ is a subtree of G (subgraph of G which is a tree) and spans the whole set V.*

Ruth Haas characterized the arborescences in undirected graphs in [12]. He mainly analyzed the following two types of graphs with the parameters k and l in Theorem *5.2.4* and *5.2.5*.

1. For which adding **some** l edges produces a graph which is decomposible into k spanning trees.
2. For which adding **any** l edges produces a graph which is decomposible into k spanning trees.

The following definition describes the concept of qTk-graphs which is useful in classifying the graphs. Their significance is further discussed by Theorem *5.2.4* and *5.2.5*.

Definition 5.2.2
Let qTk be a decomposition of the edges of a loopless multigraph G into q edge-disjoint trees such that each vertex is in exactly k trees. Let a multigraph G be a qTk-graph if there exist a qTk-decomposition in G.

By [5], the special case of $3T2$-graphs correspond to rigid graphs in the plane, because they are the $(2,3)$-tight ones. The equivalence of rigid frameworks and $(2,3)$-tight graphs is claimed by Theorem *5.3.5* in the following section.

Definition 5.2.3
*The **arboricity** of a loopless multigraph $G = (V, E)$ is the least number of edge-disjoint forests whose union covers the whole edge set E.*

Nash-Williams showed in [13] that the arboricity of a graph G can be determined by

$$k := \max\left\{\frac{|E'|}{|V'|-1}\right\},$$

where the maximum is taken over all subgraphs $G' = (V', E')$ on at least two vertices.

Main theorems

The following theorems give equivalent conditions for the decomposition from [12]. The first one characterizes the case of **some**, and the second one the case of **any**.

Theorem 5.2.4
The followings are equivalent for a loopless multigraph $G = (V, E)$ and integers $k > 0$ and $l > 0$.

1. *$|E| = k(|V| - 1) - l$, and for subgraphs $H = (V', E') \subset G$ with at least 2 vertices $|E'| \leq k(|V'| - 1)$.*
2. *There exist **some** l edges which when added to G result in a graph that can be decomposed into k spanning trees.*
3. *G is a $(k + l)Tk$-graph.*

Also, there are equivalent conditions for the decomposition of the case of **any**:

Theorem 5.2.5
The followings are equivalent for a loopless multigraph $G = (V, E)$ and integers $k > 0$ and $l > 0$.

1. *$|E| = k(|V| - 1) - l$, and for subgraphs $H = (V', E') \subset G$ with at least 2 vertices $|E'| \leq k(|V'| - 1)$.*
2. *Adding **any** l edges to G (including parallel edges) results in a graph that can be decomposed into k spanning trees.*
3. *G is a $(k + l)Tk$-graph.*

5.3 Rigidity

Laman described the wide theory of generically rigid graphs in the plane in [5]. He gave equivalent conditions for generic minimally rigid frameworks using sparsity, which is shown below. Afterwards, Jacobs and Hendrickson transformed the Pebble Game algorithms regarding the theory of rigidity in [8]. Now we make a short summary of the theory of rigidity from [5].

Definitions and characterization

In rigidity theory, the ordinary notations of graph theory are extended. Now we introduce these notations from [5].

The vertices of an undirected multigraph G correspond to **sites** and the edges of G to **bonds**. In this section, these concepts are used.

Definition 5.3.1
*A **framework** $p(G)$ is a multigraph $G = (V, E)$ along with a mapping $P : V \to \mathbb{R}^2$ which assigns each site to a point in the plane.*

In a framework, the bonds are always straight between the sites. The main question is whether a framework can be flexed such that the lengths of all bonds remain unchanged and the bonds also remain straight.

The followings define the rigid frameworks in the plane:

Definition 5.3.2
*Two frameworks $p(G)$ and $q(G)$ are **congruent** if $d(p(u), p(v)) = d(q(u), q(v))$ holds for each pair $u, v \in V$, where d denotes the Euclidean distance between the sites in the plane.*

Definition 5.3.3
*A **motion** of a framework $p(G)$ to $q(G)$ in the plane is a continuous function $f_v : [0, 1] \to \mathbb{R}^2$ for each site v such that*

- *$f_v(0) = p(v)$ and $f_v(1) = q(v)$ for each site $v \in V$.*
- *$d(f_v(t), f_u(t)) = d(p(v), q(u))$ for each bond $uv \in E$ and for each $t \in [0, 1]$.*

In the following definition, a framework is said to be rigid in the plane if there is no motion that can change the distance d between any pairs of sites.

Definition 5.3.4
*A framework $p(G)$ is **rigid** if every motion takes it into a congruent framework $q(G)$.*

For the algorithmic decision of whether a framework is rigid, Laman gave an equivalent condition using *sparsity*:

Theorem 5.3.5 (Laman)
A multigraph G is a rigid framework if and only if it is $(2, 3)$-tight.

Higher dimension

Unfortunately, there is no known generalization of Laman's theorem in higher dimensions. Although, a necessary condition of $3D$ rigidity can be described.

Theorem 5.3.6
If a multigraph G is a rigid framework in $3D$, then it is $(3, 6)$-tight.

Note that the range of $0 \leq l < 2k$ does not contain the parameters $k = 3$ and $l = 6$. The special case $l = 2k$ is later discussed in Section *6.1.3*.

To show that the other direction does not hold, see the following example graph from [14], called double banana:

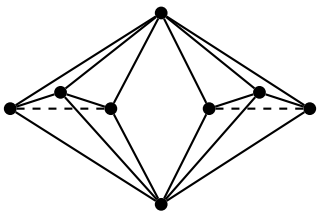


Figure 5.1: *The double banana, which is also $(3, 6)$-tight, however, it is not rigid in $3D$ because the two halves may rotate independently.*

The characterization of rigid graphs in three dimensions is still an open problem.

6 Generalizations

In this chapter, we introduce some practical generalizations for sparsity and its applications. We show new definitions and prove computational hardness for the classification of the graphs with a respect to the new definition. Afterwards, we extend the range of k, l with $l = 2k$ on simple graphs, and $l < 0$ on multigraphs.

6.1 Excluding small violating vertex subsets

In this section, we introduce a practical generalization for (k, l)-sparsity. First, we show the definitions and that the *fundamental problems* for general sparsity is NP-hard in Section *6.1.2*. Afterwards, we describe an algorithm for extending the basic range of $0 \leq l < 2k$ with $l = 2k$ on simple graphs. Note that this section is based on our own idea, as we did not find any references with these concepts and theories.

6.1.1 Basic definitions

Now we are ready to come up with a practical generalization of (k, l)-sparsity for this section. The original definitions regarding (k, l)-sparsity in Section *1.3* are simply expanded by the lower bound of the size of the subset X.

Definition 6.1.1
*A multigraph G is (j, k, l)-**sparse** if any subset X of vertices with at least j vertices induces at most $\max\{0, k|X| - l\}$ edges.*

Definition 6.1.2
*A multigraph $G = (V, E)$ is (j, k, l)-**tight** if it is (j, k, l)-sparse and has exactly $k|V| - l$ edges.*

Definition 6.1.3
*A multigraph $G = (V, E)$ is (j, k, l)-**spanning** if it contains a (j, k, l)-tight subgraph that spans the entire vertex set V.*

Note that it is trivial to see that the cases of $j = 0$ and $j = 1$ are the same. The original definitions are exactly the case of $j = 0$, and therefore $j = 1$, but the further cases are not easy to characterize.

6.1.2 NP-hardness

The decision of *fundamental problems* for general (j, k, l)-sparsity is quite difficult, unlike the basic case of $j = 1$. Now we briefly introduce its NP-hardness by a reduction from the clique problem:

Definition 6.1.4
A clique in a multigraph G is a complete subgraph $G' = (V', E')$ of G such that every two vertices in V' are adjacent. In addition, G' is an α-clique if it is a clique and $|V'| = \alpha$.

Theorem 6.1.5
The identification of (j, k, l)-sparse graphs for general j 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 in [15], which asks if there exists an α-clique in a graph. Now we show a reduction from the (j, k, l)-sparsity to the clique problem. It is trivial that this problem remains hard when we assume that α is odd. The proof of the following equivalence also proves the theorem:

Lemma 6.1.6
Let $\alpha > 1$ be an odd integer and $l > 0$ be an integer. Then a multigraph G is $\left(\alpha, \frac{\alpha-1}{2}, l\right)$-sparse if and only if G does not contain an α-clique.

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

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

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

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

holds, so X cannot be an α-clique. □

We proved that finding an α-clique in a multigraph is equivalent with testing $\left(\alpha, \frac{\alpha-1}{2}, l\right)$-sparsity, which completes the proof. □

6.1.3 A note on $l = 2k$

In this section, we describe an algorithm in the special case of $l = 2k$ for $k > 0$. In this range, the (k, l)-sparse edge sets do not form the independent sets of a matroid, unlike in the basic range of $0 \leq l < 2k$, therefore the *extraction problem* is not assured to be solved optimally with a greedy edge order. The proposed algorithm finds only an inclusion-wise sparse subgraph for simple input graphs. As an application, it forms the base of $3D$ rigidity with the parameters of $k = 3, l = 6$, which was described in Section *5.3*.

For $j = 1$, this range is basically characterized by Lemma *2.1.1*, because only the empty graph is $(1, k, 2k)$-sparse for all $k \geq 0$. In the rest of this section, we assume that $j = 3$, which ensures that the trivial violating two-element vertex subsets are ignored in the case of $l = 2k$.

Note that the *rules* of Basic Pebble Game cannot be extended for general j values on multigraph with fewer than j vertices. By the *definition of (j, k, l)-sparsity*, they must be classified separately at the beginning of the algorithm as sparse, regardless of the number of their edges.

Now we describe the best known algorithm so far, due to Csaba Király (personal communication), which runs in time $O((k + n)knm)$, then we present our improved algorithm which takes $O(k^2nm)$ steps.

The best known algorithm

Algorithm 11: Pebble Game for $l = 2k$

Input: An undirected simple graph $G = (V, E)$ on at least three vertices.
Output: *Well-constrained*, *under-constrained*, *over-constrained* or *other*.
Method: Play Algorithm *1* with the following modifications. Assume that an edge $e = uv$ is not *accepted* yet, and the *pebble collections* has already gathered exactly the total of $l = 2k$ pebbles on the endpoints u and v. Denote the *reachable region* of u and v by $X := Reach(u, v)$.

- If there exists a vertex in $X \setminus \{u, v\}$, then X forms a block by Lemma *6.1.9* (part a), so e is rejected.
- Otherwise, check if a pebble can be gathered for each vertex $w \in V \setminus \{u, v\}$. 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 above succeeds, then e can be *accepted*.
  - Otherwise, $X \cup \{w\}$ forms a block with a vertex $w \in V \setminus \{u, v\}$ such that $\text{peb}(w) = 0$ by Lemma *6.1.9* (part a), so e is rejected.

Complexity analysis

Processing each edge requires at most l invocations of *pebble collections* in time $O(lkn) = O(k^2n)$, and the further collections for at most $n-2$ vertices in time $O(nkn)$, therefore each edge is processed in time $O((k+n)kn)$. As the graph has m edges, the running time of this algorithm is $O((k+n)knm)$, which is $O(n^2m)$ if we consider k as a constant.

Correctness

To prove the correctness of the algorithm, we introduce the orientation lemma from [16], which is used with $\gamma = 0$ in the algorithm:

Lemma 6.1.7 (Orientation lemma)
Let $G = (V, E)$ be an undirected graph, $g : V \to Z_+$ an upper-bound function, and $\gamma \geq 0$ an integer. It is possible to remove at most γ edges from G in such a way that the remaining graph G' has an orientation with out-degree function out *satisfying* $\text{out}(v) \leq g(v)$ *for every vertex v if and only if* $\sum_{v \in X} g(v) + \gamma \geq i_G(X)$ *holds for every $X \subseteq V$.*

The proof of the following lemma justifies the correctness:

Lemma 6.1.8
The classification of Algorithm 11 coincides with tight, sparse, spanning and nor sparse nor spanning graphs, respectively.

Proof.
Note that the *pebble collections* are assured to gather exactly l pebbles by the *orientation lemma*. An edge $e = uv$ is *inserted* if and only if every vertex subset X containing vertices u, v and w has more than l pebbles, because the endpoints u and v have exactly l, and an additional pebble may be collected on each w. Before the insertion, all subsets containing u, v and another vertex are checked, so the condition of the *definition of (j, k, l)-sparsity* is fulfilled, and the edge is *inserted* if and only if every subset X is not tight. □

The improved algorithm

We realised that in Algorithm *11* the actual *pebble collections* for each vertex w are not required to check the condition of the two-element *reachable region* X. Instead of n *pebble collection*, execute only one BFS for all the vertices with at least one pebble in the reverse graph, and check if all vertices of V are reached. Reaching a vertex w indicates if there exists a reorientation of D in which $X \cup \{w\}$ has $l+1$ pebbles, but the *collections* are not actually executed. This condition is checked for all vertices in $V \setminus \{u, v\}$ at once, in time $O(n)$. The concrete algorithm is the following:

Algorithm 12: Pebble Game for $l = 2k$, improved version

Input: An undirected simple graph $G = (V, E)$.
Output: *Well-constrained*, *under-constrained*, *over-constrained* or *other*.
Method: Play Algorithm *11* with the following modifications. If $X := \mathit{Reach}(u, v)$ only contains 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 the 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 *inserted*, by Lemma *6.1.9* (part b).
- Otherwise $\{u, v, w\}$ violates with a non-reached vertex w and e is rejected, by Lemma *6.1.9* (part a).

Complexity analysis

Processing each edge requires at most the basic l pebble collections in time $O(lkn) = O(k^2n)$, and the further BFS in time $O(kn)$, so overall $O(k^2n)$. As the graph has m edges, the running time of this algorithm is $O(k^2nm)$, which is $O(nm)$ if we consider k as a constant.

The correctness

The following lemma shows the correctness of Algorithm *12*.

Lemma 6.1.9
Assume that edge $e = uv$ is not yet accepted and no more pebbles can be collected on u and v. Let $X := Reach(u, v)$ be the reachable region of u and v.

(a) *If X has an additional vertex w different from u and v, then X forms a block.*

(b) *Otherwise, if X only contains u and v, and in the BFS (from all the sources containing at least one pebble in the reversed digraph) each vertex $w \in V \setminus \{u, v\}$ is reached, then e is insertable.*

Proof.

(a) By assumption, e is not *accepted*, so $\text{peb}(u) + \text{peb}(v) \leq l = 2k$. As each vertex $w \in X \setminus \{u, v\}$ can be reached from u or v, $\text{peb}(w) = 0$; 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,

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

and X is a *reachable region*, so no arcs leave it: $\text{out}(X) = 0$. It follows that

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

from where, by the sparsity of X, it is tight and therefore a block.

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

□

6.2 A note on $l < 0$

In this section, we make a summary of the case of $k > 0$, $l < 0$ in multigraphs. Note that this algorithm cannot solve all *fundamental problems*, only the decision problem. The idea of the following algorithm was developed by Csaba Király, in [17]. However, we change the original description to remain consistent: we use the concept of pebbles, similarly to Basic Pebble Game.

Note that the correctness of the following algorithm is assured by the *orientation lemma* with $\gamma = -l$. However, as we mentioned, we keep consistency and use the idea of pebbles to describe the algorithm in details.

Maintenance

The algorithm uses the following data structures:

- An inner digraph $D = (V, A)$ from the vertices of the input graph. It is initialized to be empty.
- A subset of vertices X which has an outdegree of zero and contains vertices with outdegree of k in D throughout the algorithm. It is also empty at the beginning of the algorithm.
- The number of rejected edges which has an upper bound of $\gamma = -l$, called n_γ. Initialize it to zero.

The description

Now we introduce the algorithm:

Algorithm 13: Pebble Game test for negative l values

Input: A multigraph $G = (V, E)$, $k > 0$ and $l < 0$ integers.
Output: Either ***sparse*** or **not**.
Method: Play Algorithm *1* with the following modifications. Denote the actual edge by $e = uv$.

- If both endpoints u and v are in X, then e is rejected. Furthermore, if the number n_γ of rejected edges is at least $-l$, then a violating subset is found and the algorithm is terminated by **not sparse**; otherwise n_γ is increased by one.
- Otherwise, start a *pebble collection* in D to gather a pebble on u or v. If it can be accomplished, then *insert* the edge, otherwise reject and add the reached vertices into X.

Complexity analysis

The algorithm processes each edge at most once and requires at most one BFS search in time $O(n + m_a) = O(kn)$. Even if G is not sparse, the algorithm terminates after inserting $O(kn)$ edges, so the overall running time of the algorithm is $O(k^2n^2)$.

Lemma 6.2.1
The Algorithm 13 correctly recognizes the (k, l)-sparse graphs for all parameters $k > 0$ and $l < 0$.

7 Implementations

In this chapter, we present some implementation details and some charts showing the running time of our implementations of most of the discussed algorithms.

Now we recall the concrete algorithms which have been implemented.

1. *Basic Pebble Game*.
2. *Basic Component Pebble Game*.
3. *Unweighted Component Pebble Game*.
4. *Finding k edge-disjoint r-arborescences*.
5. *Covering with k edge-disjoint forests*.

We hereby note that all these algorithms will be proposed to be part of the LEMON [18] optimization library until the end of the summer of 2022. The repository of our work is regularly updated and can be found in [19].

7.1 Implementation details

In this section, we discuss some tricks for the implementations of the algorithms enumerated above and make some observations on further heuristic ideas.

BFS algorithm

It is easy to see that in the Pebble Game algorithms, most of the time is spent *collecting the pebbles*, which use BFS algorithms on the inner digraph. Therefore, the BFS algorithms should be sped up in order to improve the total running time of the Pebble Game algorithms.

In our implementation, we used a minimalist BFS which only maintains the most important variables. The comparison between the BFS implementation of LEMON and the minimalist BFS is shown by Figure *7.1*.

Note that for *accepting an edge*, more than one collection may be required. Observe that in this case, the additional BFS calls may take some unnecessary steps, because there can be some *reachable regions* of the inner digraph D from the endpoints u and v that despite they do not contain a vertex with a free pebble, they are also reached from the previous BFS algorithms. It is trivial that the pebbles can only be moved onto the endpoints, therefore it is sufficient to detect these regions only once. Nonetheless, the cardinality of these regions proved to be small in practice, therefore this method may not speed the algorithms up significantly. Thus, it is not included in our improvements.

7.2 Tests

In this section, we analyze the running time of the previously enumerated algorithms for different input parameters (k, l) and input graphs G. The following tests were executed on random multigraphs with fixed maximal possible edge multiplicity.

We considered Erdős-Rényi random multigraphs [20]. The model depends on the following parameters: the number n of vertices, the maximal possible multiplicity q of edges or loops, and the probability p of an edge or a loop being present.

Our method using Erdős-Rényi model was the following.

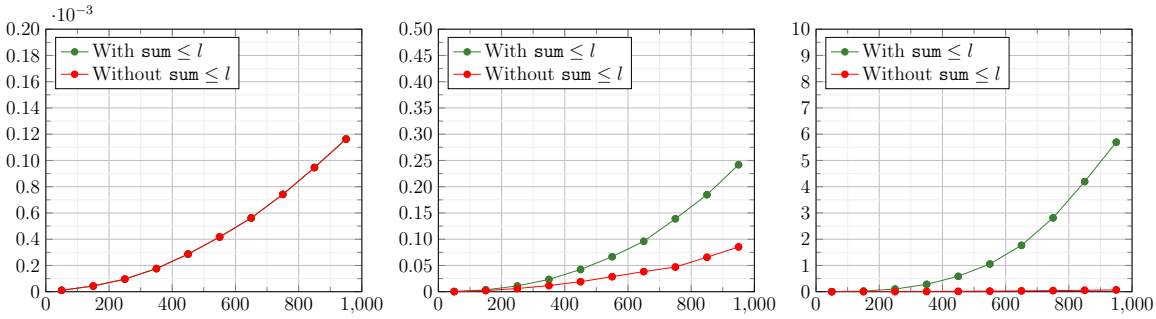
Algorithm 14: Erdős-Rényi random graph generation

Input: An interval $[a, b]$, a multiplicity q of edges and loops, and the expected number of required edges r.
Output: A random generated multigraph G with r edges in expectation, that is, $\mathbb{E}(m) = r$.
Method: Select the number n of vertices of G with discrete uniform distribution $U \sim [a, b]$ at random. Note that $n > 0$ is required. Then define multiplicity $q := 10$, meaning at most 10 parallel copies of each edge and loop may be present in G. As the graph may have at most $\left(\binom{n}{2} + n\right) q$ edges or loops, define $p := \frac{r}{\left(\binom{n}{2}+n\right)q}$ for generating r edges in expectation. Add exactly n vertices to G, then repeat the following process q times. For any two not-necessarily-distinct vertices u and v, determine a random number c with continuous uniform distribution $U \sim (0, 1)$ and if $c < p$, then insert edge uv.

In the following tests, each algorithm solves the extraction problem, which implicitly includes the spanning- and the decision problems. On each figure, the horizontal axis is the number n of vertices of the multigraphs and the vertical axis is the corresponding average running times of the algorithms. In our tests, we investigated three classes of graphs depending on the expected value of the number of their edges: n, $n^{\frac{3}{2}}$ and n^2, which are shown in Figure (a), (b) and (c), respectively. We tested the following intervals regarding the number of vertices: $[0, 100], [100, 200], \ldots, [900, 1000]$. Plus, we repeated Algorithm *14* exactly 100 times in each interval, and the illustrated values represent the average running times in each interval. All these tests were executed with AMD Ryzen 9 3950X Processor, and Linux operation system.

Terminating condition

Now we note that in the description of Basic Pebble Game in [1], the evident terminating condition when the total number of pebbles is at most l is not mentioned. Our description in Algorithm *1* contains it, which makes the running time of the game significantly shorter when the graph has many edges. The following figure shows a comparison between the average running times of the algorithm using the condition of $\texttt{sum} \leq l$ or not. Note that every case was tested for all values $1 \leq k \leq 5$ and $0 \leq l < 2k$.

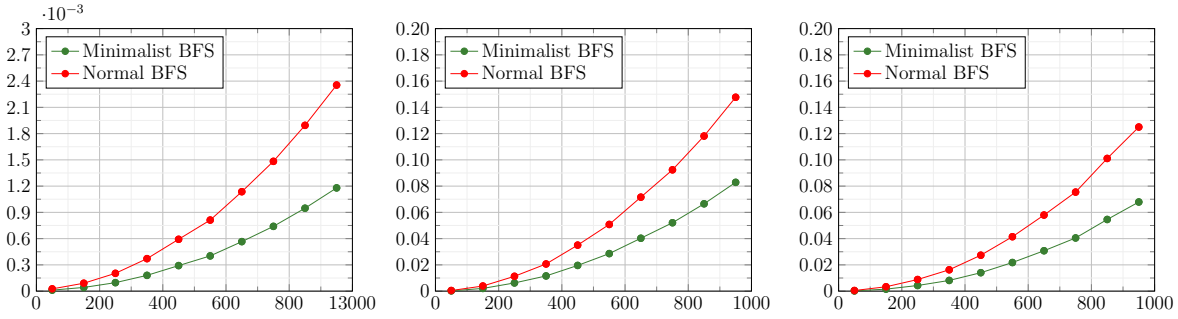


(a) $m \approx n$ *edges.* **(b)** $m \approx n^{\frac{3}{2}}$ *edges.* **(c)** $m \approx n^2$ *edges.*

Figure 7.1: *Comparison between using and skipping the terminating condition* $\texttt{sum} \leq l$. *Figure (a) almost always results in* (k, l)*-sparse graphs, therefore, the condition is not effective (the points overlap). Figure (b) has quite a lot spanning graphs, therefore, the running times differ significantly. Finally, Figure (c) means loads of further edges, which are unnecessary to process, therefore, there are huge differences in the running times.*

BFS

The following figure shows the difference between the running time of Algorithm *1* with normal and minimalist BFS. In practice, we also realized that it is more efficient to use a global BFS object and initialize it over and over, than creating a local BFS object every time it should be called. The Minimalist BFS version is implemented in the more efficient way, not like the Normal BFS, which results the following essential difference between their running times.

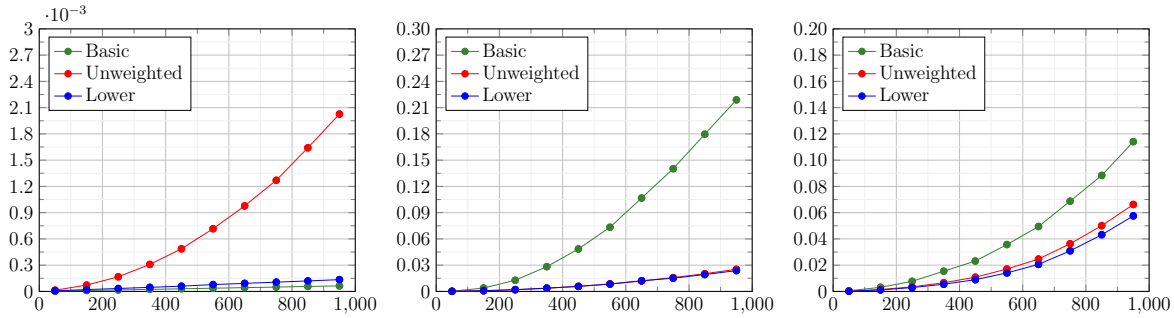


(a) $m \approx n$ *edges.* **(b)** $m \approx n^{\frac{3}{2}}$ *edges.* **(c)** $m \approx n^2$ *edges.*

Figure 7.2: *Comparison between BFS and Minimalist BFS.*

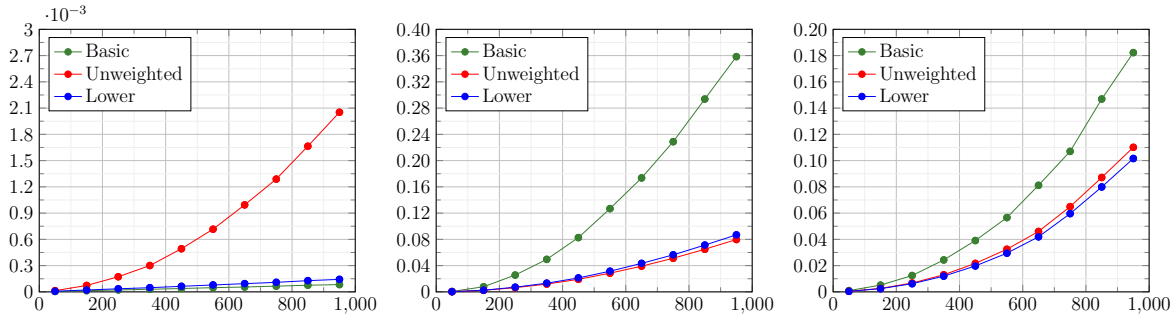
7.2.1 Comparison of the algorithms in different ranges

In this section, each figure investigates different ranges of l for fixed $k = 7$ value, and compares the performance of the algorithms.



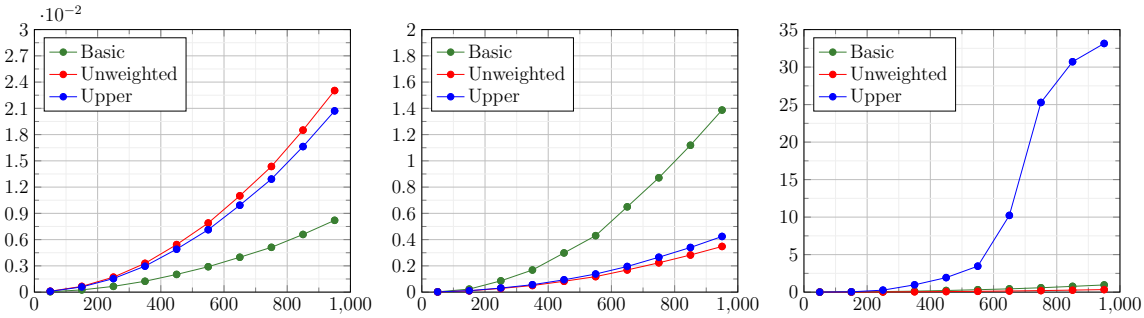
(a) $m \approx n$ *edges.* **(b)** $m \approx n^{\frac{3}{2}}$ *edges.* **(c)** $m \approx n^2$ *edges.*

Figure 7.3: *Running times when* $l = 0$.



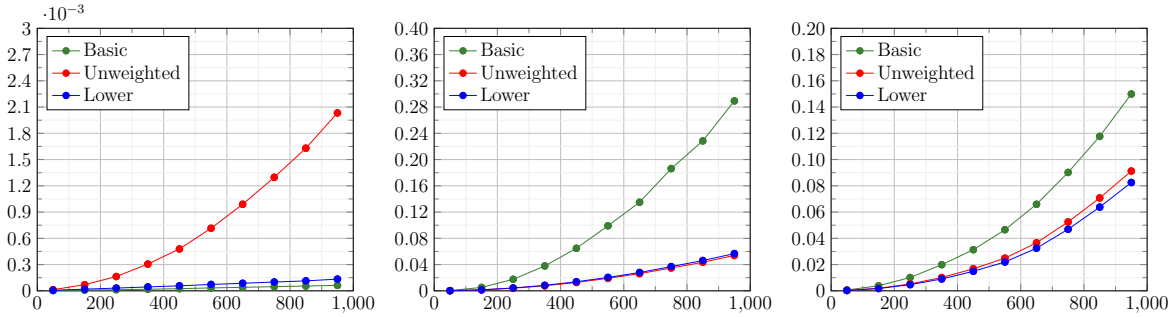
(a) $m \approx n$ *edges.* **(b)** $m \approx n^{\frac{3}{2}}$ *edges.* **(c)** $m \approx n^2$ *edges.*

Figure 7.4: *Running times when* $l = k$.



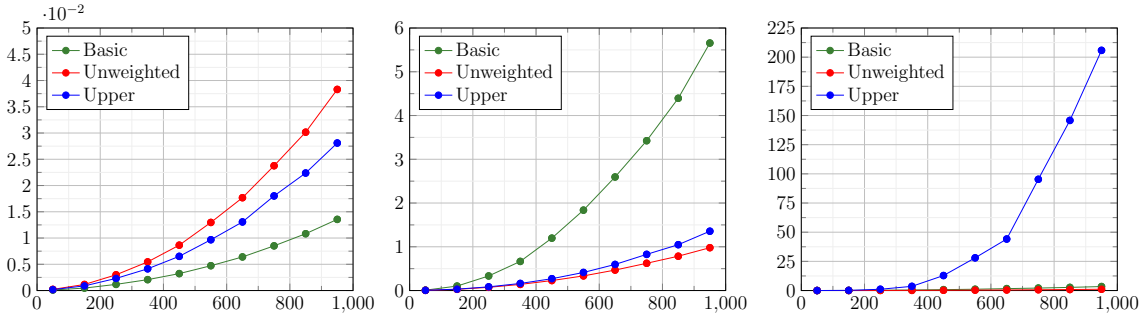
(a) $m \approx n$ *edges.* **(b)** $m \approx n^{\frac{3}{2}}$ *edges.* **(c)** $m \approx n^2$ *edges.*

Figure 7.5: *Running times when* $l = 2k - 1$.



(a) $m \approx n$ *edges.* **(b)** $m \approx n^{\frac{3}{2}}$ *edges.* **(c)** $m \approx n^2$ *edges.*

Figure 7.6: *Running times in the lower range.*



(a) $m \approx n$ *edges.* **(b)** $m \approx n^{\frac{3}{2}}$ *edges.* **(c)** $m \approx n^2$ *edges.*

Figure 7.7: *Running times in the upper range.*

In each case, the following observations can be made. In Figure (a), the running time of the Basic Pebble Game (Algorithm *1*) is proved to be the fastest because there are fewer components in these graph, therefore, maintaining the components is unnecessary. In Figure (b), the running time of the Basic Pebble Game turns out to be the slowest because the graphs have lots of edges which may be rejected in constant time with the concept of components. In these graphs, the running times of the Unweighted Pebble Game (Algorithm *6*) and the Component Pebble Games (Algorithm *2*) are almost the same. In Figure (c), the running times in the lower range is similar to that in Figure (b), however, as these results come from our own algorithm for the updating process (Algorithm *4*), the running time of this algorithm in the upper range is not clear (see, Figure *7.5* (c)).

7.3 Future plans

In this section, we briefly discuss the further plans in our project. As we mentioned, our implementation will be proposed to be part of the LEMON library.

During this summer, we will work on the user interface for all of our algorithms. Our user-friendly intentions regarding the classes that can be used are the following:

1. Basic Pebble Game algorithm, which can also simply handles the optimization problem.
2. A unit of Component Pebble Game algorithms consisting of Algorithm *2* in the weighted case and Algorithm *5* in the unweighted case.
3. Covering edge set E with k edge-disjoint forests algorithm with Finding k edge-disjoint r-arborescences and Pebble Game algorithms.

Each is planned to provide many options, such as, execution control and easy-to-use query functions.

Furthermore, we are also planning to work on a better updating process in the upper range (Algorithm *4*) for Component Pebble Game. In Figure *7.5* (c), the running times in this case are quite unexpected, but the reasons remain unclear. Thus, we would also like to make further tests with more and bigger graphs to understand it.

In addition, we want to examine how the order of the edges affects the running time of the discussed algorithms. As the Basic Pebble Game processes the edges in an arbitrary order, this order can be chosen, except when it solves the optimization problem. We will also try to give a heuristics to find an effective order.

Bibliography

[1] A. Lee and I. Streinu. Pebble game algorithms and sparse graphs. *Discrete Mathematics 308*, 2008.

[2] M. Loréa. On matroidal families. *Discrete Mathematics 28*, 1979.

[3] C. S. A. Nash-Williams. Edge-disjoint spanning trees of finite graphs. *Journal of the London Mathematical Society*, 1961.

[4] W. T. Tutte. On the problem of decomposing a graph into n connected factors. *Journal of the London Mathematical Society*, 1961.

[5] G. Laman. On graphs and rigidity of plane skeletal structures. *Journal of Engineering Mathematics*, 1970.

[6] A. Recski. A network theory approach to the rigidity of skeletal structures i. *Discrete Applied Mathematics 7*, 1984.

[7] D. J. Jacobs, A. Rader, M. Thorpe, and L.A. Kuhn. Protein flexibility predictions using graph theory. *Proteins 44*, 2001.

[8] D. J. Jacobs and B. Hendrickson. An algorithm for two-dimensional rigidity precolation: the pebble game. *Journal of Computational Physics 137*, 1997.

[9] L. Szegő. On constructive characterizations of (k, l)-sparse graphs. *EGRES Technical Report 2003-10*, 2003.

[10] A. Lee, I. Streinu, and L. Theran. Finding and maintaining rigid components. *Canadian Conference on Computational Geometry*, 2005.

[11] A. Schrijver. Combinatorial optimization. *Springer*, 2004.

[12] R. Haas. Characterizations of arboricity of graphs. *Smith College*, 2002.

[13] C. S. A. Nash-Williams. Decomposition of finite graphs into forests. *Journal of the London Mathematical Society*, 1964.

[14] B. Servatius J. Graver and H. Servatius. Combinatorical rigidity. *American Mathematical Society*, 1993.

[15] R. M. Karp. Reducibility among combinatorical problems. *University of California at Berkeley*, 1972.

[16] A. Frank. Connections in combinatorial optimization. *Oxford University Press*, 2011.

[17] Cs. Király. An efficient algorithm for testing (k, l)-sparsity when $l < 0$. *EGRES Technical Report 2019-04*, 2019.

[18] D. Balázs, A. Jüttner, and P. Kovács. Lemon - an Open Source C++ Graph Template Library. *Electron. Notes Theor. Comput. Sci.*, 2011.

[19] LEMON: Library for Efficient Modeling and Optimization in Networks. Repository of sparse graphs: `https://lemon.cs.elte.hu/repos/sparseGraphs`.

[20] P. Erdős and A. Rényi. On random graphs. *Publicationes Mathematicae.*, 1959.