Philipps-Universität-Marburg

Fachbereich Mathematik und Informatik

Algorithm Engineering for the Triangle-2-Club Problem

Bachelor Thesis

zur Erlangung des akademischen Grades eines

Bachelor of Science

im Studiengang

Informatik

vorgelegt von:

Philipp Heinrich Keßler

am 27.11.2020

Matrikelnr.: 3074703
Betreuer:
Prof. Dr. Christian Komusiewicz
M.Sc. Niels Grüttemeier
M.Sc. Frank Sommer

Contents

1 Introduction

Real world data is often represented as a graph. Finding structures in graphs is part of many wellknown problems in computer sience. The algorithms to find such structures are used for example in social media or for online marketing. Detected structures can help to make suggestions for friends, products, and almost every other point of intrest for users, companies or scientists. Friendships in social media, for example, can be depicted in a graph by making a vertex for each person [4]. If two persons are friends, their vertices are connected by an edge. Other use cases in social media are for example event summarization or spam detection. Schinas et al. [13] proposed a summarization framework, that makes use of a graph-based algorithm to summarize the most relevant social media posts of an event. In regard of spam detection, graphs are used for example by Zhang et al. [14] to detect so called spamming groups in social networks.

One of these well known graph problems is the Clique problem [10]. The Clique problems purpose is to find a clique of maximum size in a graph. A clique is a subset of vertices of a graph, where every vertex in the subset is adjacent to every other vertex in the subset. In social media a clique could be a group of friends, where everyone is friends with every other person of the group. For some applications the requirement of a graph to be a clique may be too restrictive. Consider a group of friends on social media that fulfills the requirement to be a clique. Now, even if an addidtional person p is friends with everyone in the clique except one, then this person is not considered part of the clique. Regardless of how many persons of the clique p is friends with, a single missing friendship—which is a single missing edge in the graph—excludes p from the clique. More relaxed problems have been defined in the past to find various kinds of structures.

The s-Club problem can be used as a more relaxed version of Clique. s-Club was first proposed by Mokken [8]. Instead of requiring all vertices in the subset to be neighbors, an s-club allows a distance of maximum s between these vertices. Note that, therefore 1-Club is equivalent to Clique. Detecting large 2-clubs is not only interesing for graphs of social networks. They also can be used for biological oder financial networks [9]. Algorithms for finding 2-clubs in graphs do already exist and were topic of other research. Pajouh et al. [9] proposed an algorithm to find 2-clubs in large-scale networks to reveal information about the structure of their underlying systems. Other exact algorithms were proposed by Hartung et al. [5] and Schäfer [12]. To come back to the social media example, for a group of friends to be a 2-club, not everyone needs to know every other person. It would be enough to have one friend who knows everyone else to be part of the 2-club. This can lead to the biggest 2-club just being the person that has the most friends, together with all his friends, like shown in Figure 1a.

To add more restriction, additional requirements for structures can extend the 2-Club problem definition. Komusiewicz et al. [6] presented multiple variations with different structure requirements for robust 2-clubs, for example hereditary or well-connected robust 2-clubs. This includes for example a requirement for a certain number of neighbours or paths between the vertices of a 2-club. Another restriction to s-Club is a triangle property which forces vertices or edges of an s-club to be part of a triangle, which was first introduced by Car-

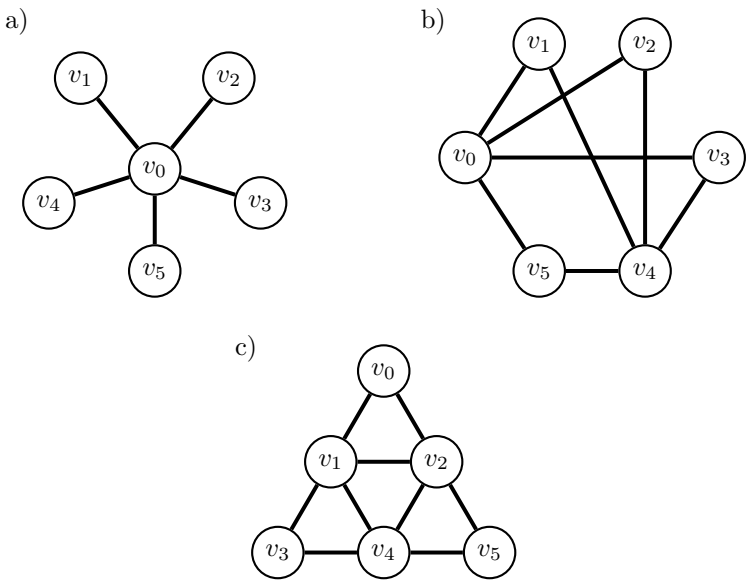

Figure 1: Both graphs in this figure are 2-clubs. Graph a) shows how a centered vertex forms a 2-club with his neighbors. Graph b) has no center vertex. Since both do not have a triangle formed by its vertices, neither of them is a triangle-2-club. On the other hand, graph c) is a triangle-2-club. All vertices and edges in graph c) are part of a triangle.

valho and Almeida [3]. This requirement forces the solution subset to form triangles in its induced subgraph in addition to the distance requirement of a s-club. Figure 1b shows a graph that fulfills the requirements of a 2-club, but not the requirements of a triangle-2-club. The triangle property requires a higher connectivity in the graph but also allows multiple groups to just be connected over one common neighbour. An example for a triangle-2-club is shown in Figure 1c. To come back to our social media example, a single person who is part of multiple friendship triangles, forms a triangle-2-club with all of the triangles he is part of. There is no need of friendships between other persons of these distinct triangles. Carvalho and Almeida [3] introduced a solution for the Triangle-s-Club problem in form of an integer linear programm (ILP) formulation. A graph can be transformed into constraints using an ILP formulation, an then be solved by an ILP solver. Another paper proposed by Almeida and Brás thematizes the complexity and properties of the ℓ-triangle-k-club problem [1]. An ℓ-triangle-k-club requires that each vertex is part of ℓ distinct triangles in addition to the k-club distance requirement. They also provided algorithms based on ILPs and results of their computational experiments.

We implemented a branching algorithm to solve the Triangle-2-Club problem. In Section 3, we propose a basic algorithm, as well as methods to improve running time of the algorithm. This includes data reduction rules that can be applied to problem instances. We provide details of our algorithm implementation in Section 4. The experimental results of our implementation and the proposed data reduction rules are shown in Section 5. We also compare the

running time of our results with the ILP results from Carvalho and Almeida [3] and Almeida and Brás [1].

2 Preliminaries

2.1 Graphs

An undirected graph G is a tuple (V, E) with a set of *vertices* V and a set of *edges* $E \subseteq \{\{u, v\} : u, v \in V\}$. In addition, $n := |V|$ is the number of vertices and $m := |E|$ is the number of edges in a graph. If an edge $\{u, v\}$ exists between two *vertices* u and v, these vertices are called *adjacent*. The number of vertices adjacent to a vertex v in a graph G is called the *degree* $\deg_G(v)$ of v. A vertex v is *isolated* in graph G if $\deg_G(v) = 0$. A graph $G' := (V', E')$ is an induced subgraph of a graph $G = (V, E)$, if $V' \subseteq V$ and $E' \subseteq E$ with E' only containing edges between vertices in V'. Formally, for a set of vertices $S \subseteq V$, $G[S] := (S, \{\{u, v\} \in E : u, v \in S\})$ is the induced subgraph of graph G by vertex set S. To denote the removal of a single vertex v of a graph $G = (V, E)$ we define $G - v := G[V \setminus v]$.

A *path* between two vertices v_0, v_ℓ is a squence $v_0, v_1, ..., v_{l-1}, v_\ell$ of vertices where any successive vertices in a sequence are adjacent in the graph. The length of a path is the number of edges on the path. If all vertices in a path are distinct, the path is called *simple*. A graph is called *connected* if there is a path between any pair of vertices in the graph. The *distance* $\text{dist}_G(u, v)$ between two vertices u, v in a graph G is the length of the shortest path between them. The diameter $\text{diam}(G)$ of a graph G is the maximum distance between any two vertices of the graph. A *cycle* is a path $v_1, \ldots, v_{k-1}, v_k$, where $k > 2$, the first $k - 1$ vertices are distinct, and $v_1 = v_k$. A graph G is called a tree if G is *connected* and does not contain a cycle.

The *closed neighborhood* $N[v] \subseteq V$ of a vertex v in a graph G contains v and every vertex that is adjacent to v in the graph. Further the *closed k-neighborhood* $N_k[v] \subseteq V$ includes all vertices whose distance to v is at most k. Note that $N[v]$ is equivalent to $N_1[v]$. The *open k-neighborhood* $N_k(v)$ of a vertex v isdefined analogously, but does not contain v itself: $N_k(v) := N_k[v] \setminus v$.

2.2 Triangle Properties

Recall, that a 2-club is a subset of vertices S in a graph G, such that the distance between any pair of vertices in $G[S]$ is at most 2. The maximum triangle-2-club for a given graph G is the maximum cardinality subset $S \subseteq V$ that induces a 2-club in $G[S]$ and in addition satisfies a *triangle-property*. There are two differenttTriangle-propertys, where either each vertex (*vertex-triangle-property*) or each edge (*edge-triangle-property*) in a 2-club $G[S]$ needs to be part of a triangle.

A vertex v is part of a triangle in a graph G, if and only if at least two of the neighbours of v are adjacent in G.

An edge $\{v, w\}$ is part of a tringle in a graph, if and only if another vertex u exists in the graph which is adjacent to both v and w.

Both formulations sound may similar at first. But consider, that a single non-triangle edge between the vertices in a graph can result in the graph not

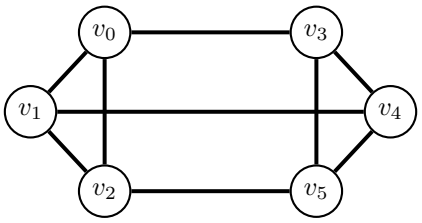

Figure 2: The graph shown in this figure is a 2-Club that satisfies the vertex-triangle-property. Each vertex is part of a triangle. On the other hand the edges $\{v_0, v_3\}$, $\{v_1, v_4\}$ and $\{v_2, v_5\}$ are not part of an edge triangle. Therefore, the graph in this figure has two edge-triangle-2-clubs of size 3 each, but is a vertex-triangle-2-club of size 6 as a whole.

being a edge-triangle-2-club. Figure 2 shows a graph that fulfills the vertex-triangle property as a whole. However, not every edge in Figure 2 is part of a triangle. These edges result in the maximum edge-triangle-2-club being of a smaller size than the maximum vertex-triangle-2-club. Each edge triangle connects three vertices and therefore also forms a vertex triangle. While additional edges between vertex triangles do not interfere with the vertex-triangle-property, they may do with the edge-triangle-property. Therefore the edge triangle requirement is more restrictive than the vertex triangle requirement and hence solutions can be smaller. On the other hand, every edge-triangle-2-club is also a vertex-triangle-2-club.

2.3 Branching Rule

Let us first define the Triangle-2-Club problem as a decision problem.

Triangle-2-Club
Input: A graph $G = (V, E)$ and $k \in \mathbb{N}$.
Question: Does G contains a subset S of vertices with $|S| \geq k$, such that $G[S]$ is a 2-club and every vertex in S is part of a vertex-triangle in $G[S]$?

A *branching rule* transforms a decision problem instance $I = (G, k)$ with a graph G and a natural number k to multiple instances $I_1 = (G_1, k_1)$, ... ,$I_\ell = (G_\ell, k_\ell)$ in polynomial time. Also, (G, k) is a YES-instance if and only if at least one of the transformed instances has to be a YES-instance. The application of a branching rule to an instance (G, k) can be displayed by a *search tree*. The original instance becomes the root of the tree. The transformed instances are children (branches) of their original instance. The search tree can get aborted as soon as a YES-instance is found.

2.4 Data Reduction Rule

A *data recution rule* is a computable function that transforms a given problem instance I into another instance I' in polynomial time. The transformed instance I' is a YES-Intstance if and only if the original instance I was a YES-Instance. In addition, the transformed instance must not be bigger than the original one.

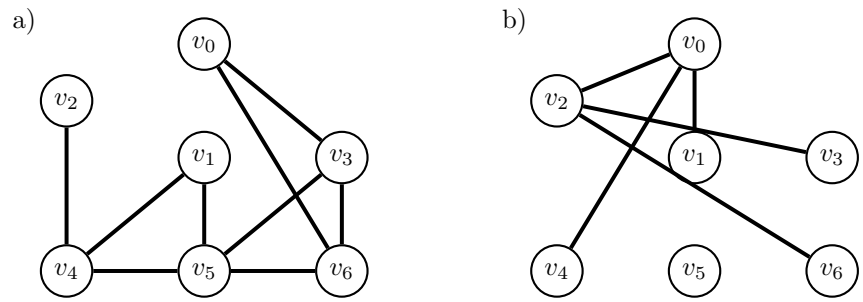

Figure 3: Figure a) shows a graph G. The corresponding conflict graph G_C is shown in Figure b). Any two vertices with a distance greater than two are considered as incompatible. In G the distance between v_0 and v_2 ($\text{dist}(v_0, v_2)$) is equal to four. Because of that the conflict graph G_C contains an edge between v_0 and v_2. On the other hand, vertices that are close enough to each other (for example v_4 and v_5 or v_1 and v_6) are not adjacent in the conflict graph.

Data reduction rules can be used for problems of any complexity. However, if a problem has a time complexity that is higher than polynomial, then the benefits of a smaller problem instance often exceed the costs of data reduction rules. The computation of a data reduction rule on such a problem is cheaper than solving the larger instance.

2.5 Problem Specific Definitions

Recall that a graph can only be a 2-club or a triangle-2-club if the distance between any pair of vertices is at most two. Therefore, if a graph G contains two vertices v and w whose distance is greater than two, then v and w are called *incompatible*. Otherwise, these vertices are called *compatible*. Note, a graph G in which every pair of vertices is compatible is further called *conflict free*. As long as there are incompatible vertices in a graph, it can not form a triangle-2-club. In the algorithm, we can make use of known incompatibilities between vertices. For example, we can specifically remove vertices with incompatibilites instead of random vertices while branching.

To keep track of incompatibilites we use a so called *conflict graph* analogously to the one described by Komusiewicz et al. [6]. The conflict graph $G_C = (V, E_C)$ of a graph G contains the same set of vertices as G itself. The conflict graph contains an edge $\{v, w\} \in E_C$ if and only if v and w are incompatible in G. Figure 3 shows a graph and its corresponding conflict graph. It can be used during the algorithm to determine or count incompatibilities in a graph faster than it could be done in the original graph. If using a adjacency matrix to represent edges in the conflict graph, then checking whether two vertices are compatible takes $\mathcal{O}(1)$ time. In order to construct a conflict graph G_C of a graph G, we need to detwermine if there is a path with a length of at most two between every pair of vertices in G. For each of these pairs we need to check at most m edges to determine if they are incompatible, leading to a cost of $\mathcal{O}(n^2 m)$ time to construct the conflict graph. Since we are already checking compatibility for all pairs of vertices, we can also count the number of incompatibilities of each

vertex. If a vertex v gets removed, it can cause new incompatibilities between vertices of $N_2(v)$. Therefore, updating the conflict graph after a vertex removal also requires $\mathcal{O}(n^2m)$ time. On sparse graphs or graphs with a large diameter, it is unlikely that the two-neighbourhood $N_2(v)$ of a removed vertex v contains every other vertex in the graph. Hence maintaining the conflict graph is cheaper than reconstructing it in general.

Further, may $\text{incom}_G(v)$ of a vertex v in a graph G be the number of vertices v is incompatible with in G. Note that for any graph G and any vertex v in G $\text{incom}_G(v) = \deg_{G_C}(v)$.

3 Algorithm

To solve the Triangle 2-Club problem we implemented a branching algorithm. In our algorithm, we use the vertex-triangle-property, described in Section 2.2. Further we extended the problem formulation by adding a set of marked vertices $M \subseteq V$. Marked vertices are vertices of the graph $G = (V, E)$ that are a mandatory part of the solution.

Marked Triangle-2-Club
Input: (G, M, k), where G is a graph, M a subset of vertices of G and $k \in \mathbb{N}$.
Question: Does G contains a subset S of vertices of size at least size k, such that $G[S]$ is a triangle-2-club and $M \subseteq S$?

Recall, that two vertices of G are called *incompatible* if they can not be part of the same triangle-2-club because of their distance bein greater than two.

3.1 Basic Algorithm

The algorithm builds a solution by marking some vertices in the graph and deleting others until the remaining graph fulfills the requirements to be a Triangle 2-Club.

At the beginning of the algorithm each vertex in the graph is considered a potential start point for branching. Therefore, each potential start vertex is the first marked vertex in its instance. So every vertex is considered for a solution at least once during the algorithm. The algorithm branches until the remaining marked and unmarked vertices form a triangle-2-club according to the triangle-property or no vertices are left to remove or mark. The algorithm always saves the current best solution. Whenever branching finds a new triangle-2-club, that larger than our current best solution, the old best solution gets discarded and replaced by the new one. As long as the current graph is not a triangle-2-club or too small, an unmarked vertex v gets selected from the remaining vertices. The algorithm branches into two cases:

- $(G - v, M, k)$, and
- $(G, M \cup \{v\}, k)$.

The first way of branching removes the selected vertex. After branching finishes for a start vertex the current graph gets rolled back to the state before the branching started. Each vertex is either part of the solution or not. If it is not,

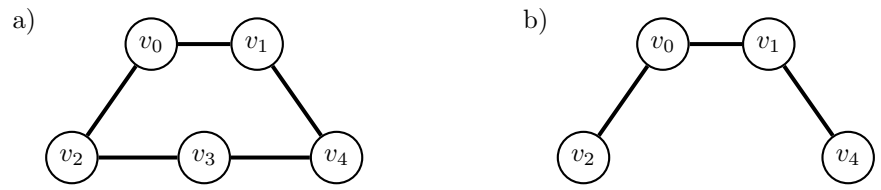

Figure 4: In graph a), the vertices v_2 and v_4 are compatible, because their distance is two. In the graph shown in b), vertex v_3 is removed. The removal of v_3 destroys the shortest path between v_2 and v_4 with a distance of three. The vertices v_2 and v_4 have become incompatible through the removal of v_3.

the first branching case will lead to a solution, otherwise the second one will. Every possible combination of vertices is covered by this branching rule. The algorithm can always find a correct solution.

3.2 Data Reduction Rules

Data reduction rules reduce the size of a problem instance. A smaller problem instance requires less time to compute. Komusiewicz et al. [6] presented data reduction rules for different robust variations of the 2-Club problem. Some of these rules can be used as data reduction rules for Triangle-2-Club, too, while others need minor changes. These rules can improve algorithm running time by reducing the size of the graph, marking vertices or identifying NO-instances early.

A data reduction rule for the Triangle-2-Club problem has to satisfy the following requirements for reducing an instance (G, M, k).

- The transformed graph G' does not contain more vertices and edges than G,
- the transformed set of marked vertices $M' \supseteq M$, and
- $k' \leq k$.

Before defining the reduction rules, some observations can be made, that help us formulate and discuss the actual reduction rules and their time complexity.

Observation 1. *If a graph G contains a pair of vertices that are incompatible to each other, then G is not a triangle-2-club.*

Proof. If two vertices in the graph are incompatible, their distance is greater than two. Therefore the graph is not a 2-club and hence no triangle-2-club. □

Observation 2. *Removing vertices from the graph can make vertices become incompatible.*

The correctness of Observation 2 is demonstrated as an example in Figure 4.

Observation 3. *Two incompatible vertices can not become compatible by removing other vertices.*

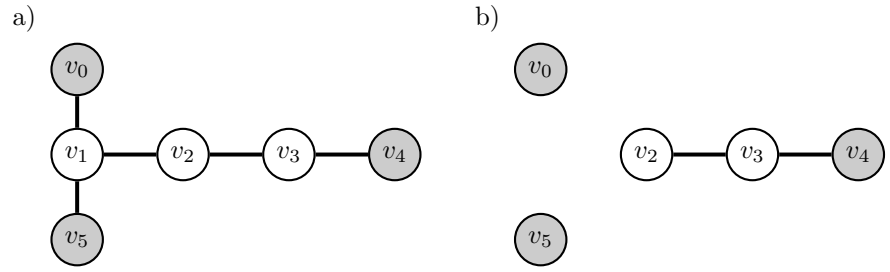

Figure 5: Considering filled vertices as marked, the application of Reduction Rule 2 on marked vertex v_4 in graph a) causes the removal of vertex v_1. This removal makes the marked vertices v_0 and v_5 become incompatible.

Proof. Let G be a graph with two incompatible vertices v and w, hence a distance $\text{dist}_G(v, w) > 2$. Deleting vertices can only remove edges with their adjacent vertices and not add new ones. So no other shorter path can be created by removing vertices and therefore no vertices can become compatible. □

Let us get to the actual reduction rules now. Some of the reduction rules use the size of a solution the algorithm already found beforehand, to identify NO-instances.

The first rule provides an important criterion for identifying NO-instances early. It can be applied at any point during branching.

Reduction Rule 1 (Marked Incompatible Rule)**.** *If at any momemt during the algorithm, the instance contains two incompatible vertices that are both marked, then this instance is a NO-Instance.*

Lemma 1. *Reduction Rule 1 is correct and can be computed to exhaustion in* $\mathcal{O}(n^2)$ *time.*

Proof. Marked vertices are a mandatory part of the solution. From Observation 1 we conclude that a graph that contains two incompatible vertices, can not form a triangle-2-club. Therefore this instance is a NO-instance.

In order to check if any pair of marked vertices are incompatible, we need to check all pairs of marked vertices. If every vertex in the graph is marked, then we need to check $\mathcal{O}(n^2)$ pairs of vertices. Since a constant time is required to check incompatibilieties in the conflict graph, this leads to a time complexity of $\mathcal{O}(n^2)$.

This reduction rule only needs to be applied once a time and only after marking new vertices. □

The next rule can be used to reduce the graph size based on the vertices that are already marked. Whenever new vertices get marked, this rule can help reduce the rest of the graph size. This data reduction rule needs to be applied multiple times at once, since a single application of it can result in new incompatibilities as Figure 5 shows.

Reduction Rule 2 (Incompatible Resolution)**.** *Remove all vertices that are incompatible to a marked vertex.*

Lemma 2. *Reduction Rule 2 is correct and can be computed to exhaustion in* $\mathcal{O}(n^3)$ *time.*

Proof. Let (G, M, k) be an instance and let $v, w \in V$ be two incompatible vertices where $v \in M$ and $w \notin M$. Since v is marked, it needs to be part of the solution built up by this branch. Vertex w is incompatible to v and can therefore not be part of solution containing v. The instance can be reduced to $(G - w, M, k)$.

Similar to Reduction Rule 1, checking all pairs of vertices requires $\mathcal{O}(n^2)$ time. As shown in Figure 5 a vertex removal can cause other incompatibilities. This makes $\mathcal{O}(n)$ applications of this rule necessary, leading to an overall time complexity of $\mathcal{O}(n^3)$. □

With help of the next rule, the graph size can be further reduced by removing vertices that can not be part of a solution. This rule can be applied after every vertex removal in the graph but especially before the branching starts. On sparse graphs, this rule can remove a large number of vertices at the beginning, which reduces the start vertices the algorithm has to branch on. By making use of the triangle-property, many verticesmay be removed before and during the branching.

Reduction Rule 3 (Triangle Rule)**.** *Remove every vertex that is not part of any triangle.*

Lemma 3. *Reduction Rule 3 is correct and can be computed to exhaustion in* $\mathcal{O}(nm)$ *time.*

Proof. To satisfy the vertex-triangle-property each vertex of the solution has to be part of a triangle. If a vertex is not part of any triangle, it can not be part of any solution and can be deleted from the graph.

To check if a vertex is part of a triangle requires searching over at most m edges. If the graph does not contain any triangle vertices, then this reduction rule needs to go along these m edges for every of the n vertices. This leads to a time complexity of $\mathcal{O}(nm)$ for a single application of this rule.

Deleting a non-triangle vertex can not destroy any triangle in the graph. Therefore, applying this reduction rule to an instance multiple times at once will not further reduce it. □

The next reduction rule allows us to delete vertices and identify NO-instances based on the number of incompatibilities these vertices have. Recall that in a problem instance (G, M, k), parameter k is the size of the triangle-2-club we want to find in the graph.

Reduction Rule 4 (Low Compatibility Rule)**.** *Remove vertices whose number of compatible vertices is smaller than* k*. If a removed vertex was marked, this instance is a NO-instance.*

Lemma 4. *Reduction Rule 4 is correct and can be computed to exhaustion in* $\mathcal{O}(n^2)$ *time.*

Proof. Let (G, M, k) be an instance of Marked Triangle-2-Club and $v \in V$ a vertex in this instance. At least $\text{incom}_G(v)$ vertices need to be removed in order to make v part of a sulotion. If $|V| - \text{incom}_G(v)$ is smaller than k, then

there can not be a solution of size k that contains v. Therefore, a branch where v is marked can not lead to a solution greater than $k-1$. On the other hand, if v is not marked, v can be removed from the graph since every solution resulting from this instance that is better than the current one, will not contain v.

For a vertex v, the number of vertices compatible to v can be determined by subtracting the number of compatibilities $\text{incom}_G(v)$ the vertex v has in G of n. Applying this rule to the graph once, hence takes $\mathcal{O}(n)$ time. But by Observation 2, removing a vertex can cause new incompatiblities, which can reduce the number of compatibilities a vertex has below the treshold of this reduction rule. This can make a repeated application of this rule possible to up to n times. This causes a time complexity of $\mathcal{O}(n^2)$, not considering the time required to update the conflict graph. □

Previous reduction rules focused on deleting vertices and identifying NO-instances. Marking vertices helps these other reduction rules to work better. In some situations marked vertices depend on other (unmarked) vertices to form a solution at all. This allows us to mark vertices based on alread existing marks.

Reduction Rule 5 (No Choice Rule 1)**.** *Let v be a marked vertex. If every triangle that contains v also contains another vertex u, then mark u.*

Lemma 5. *Reduction Rule 5 is correct and can be computed to exhaustion in $\mathcal{O}(nm)$ time.*

Proof. Let S be a solution with a vertex v. Furthermore, let $w \neq v$ be a vetex of the graph, that is part of every triangle that contains v. Since $v \in S$, vertex v is part of at least one triangle. If every triangle that contains vertex v also contains vertrex w, then v can not be part of any solution that does not contain w. Therefore the solution S also contains w.

To determine if a marked vertex v shares all its triangles with another vertex, we need to find all triangles containing v. This means, we need to iterate over all $\mathcal{O}(m)$ edges. Since up to n vertices can be marked, applying this rule once has a time complexity of $\mathcal{O}(nm)$. Fortunately this rule needs to be applied only once every branching step. If a vertex v causes another vertex w to get marked, then w either has no other triangles than those shared with v or not. If w has no other triangles, using the rule again on w has no impact.

Otherwise, if w is part of a triangle that does not contain v, then triangle vertices of this triangle can not be marked by this rule, since w can not share every triangle with them. □

An important special case for this rule emerges whenever a marked vertex is only part of a single triangle. In this case the reduction rule can mark both other vertices since they both satisfy the requirement of Reduction Rule 5.

The next reduction rule also allows marking vertices, but now those vertices that are in between already marked vertices.

Reduction Rule 6 (No Choice Rule 2)**.** *Let v and w be non-adjacent vertices and marked. If v and w share exactly one neighbor, then the common neighbor can be marked.*

Lemma 6. *Reduction Rule 6 is correct and can be computed in $\mathcal{O}(n^3)$ time.*

Proof. Let v and w be two marked vertices that are not adjacent. If these vertices have exactly one common neighbor u, then u is part of the unique path between v and w of length of at most two. By removing u, no other path of length two would exist between v and w, making these marked vertices incompatible. Therefore, a solution containing v and w also needs to contain u.

At most n vertices are marked at any time, leading to $\mathcal{O}(n^2)$ pairs of marked vertices. Since every vertex has at most $n-1$ neighbours, we can find all common neighbor of two marked vertices in $\mathcal{O}(n)$ time. This leads to an overall time complexity of $\mathcal{O}(n^3)$ for one application of this rule. □

From Observation 1 we conduct that the conflict graph (see Section 2.5) of a triangle-2-club does not conatin any edges. In order to resolve an existing conflict without adding edges or vertices, one of the vertices in a conflict has to be removed. The conflict graph of an instance can help us to identify NO-instances early on. For this reduction rule we make use of a *maximum matching* on the conflict graph.

A *matching* A is subset of edges, such that no distinct edges are incident to the same vertex in the graph. Each vertex in a matching is of degree zero. A matching is a maximum matching if there is no other matching of greater size for the graph.

Let us recall, that the a graph $G := (V, E)$ and its corresponding conflict graph $G_C := (V, E_C)$ contains the same set of vertices V.

Parameter k of the problem instance (G, M, k) can be used as upper bound to cancel any maximal matching computation early. If too many vertices would have to be removed, such that $|V| < k$ the computation can be aborted, since the result is already big enough to trigger Reduction Rule 7.

Reduction Rule 7 (Maximum Matching Rule)**.** *Compute the size b of a maximum matching for the conflict graph G_C. If $|V| - b$ is not greater than k, then the instance is a NO-instance.*

Lemma 7. *Reduction Rule 7 is correct and can be computed in $\mathcal{O}(n|E_C|)$ time.*

Proof. Let an instance (G, M, k) be given and G_C be the corresponding conflict graph. From Observation 1 we conduct that G is not allowed to have incompatibilities in order to form a Triangle-2-Club. The maximum matching chooses edges and removes them with their incident vertices. This computes the number of vertices b that need to get removed to make the graph conflict free. If $|V| - b$ is smaller than k, too many vertices of G would need to be removed to make G conflict free.

The time needed to compute a maximal matching is depending on the number vertices n and the number of edges in the conflict graph $|E_C|$. A greedy approach would randomly select an edge and remove both of its incident vertices. With the deletion of a single vertex, up to $|E_C|$ have to be removed too. This leads to a time complexity of $\mathcal{O}(n|E_C|)$.

The maximal matching does not modify the graph and hence only needs to be applied after new vertex deletions. □

3.3 Start Vertex Order

The algorithms running time can be further improved by ordering the start vertices by their neighborhood size. For each start vertex v, the size of its

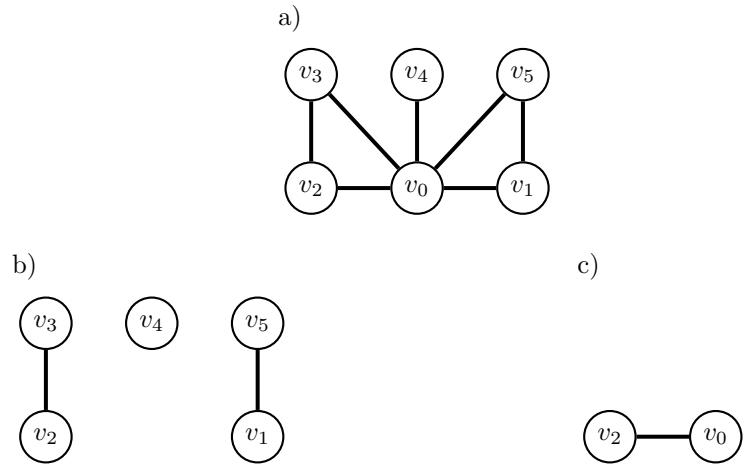

Figure 6: Figure a) shows a graph G. Figure b) shows the subgraph induced by $N(v_0)$ and Figure c) shows the subgraph incuded by $N(v_3)$. By counting the non-isolated vertices (and adding one for vertex v_0 or v_3) a lower bound for a Triangle 2-Club solution can be found.

two-neighborhood $|N_2(v)|$ can be computed beforehand and then used to order the start vertices. Branching over start vertices with small neighborhood sizes requires less time than branching over vertices with large neighborhoods. After a vertex v was used as start vertex, the algorithm considered all possible solutions including v. Therefore, if another vertex w is used as a start vertex at a later time, all solutions that would contain v has already been considered at this point. Since there cannot be a better solution containing vertex v, it can be removed from the graph permanently. This results in a decrease of size of the graph over the course of the algorithm. The removal of vertices also decreases the size of the remaining vertices neighborhoods, and hence, speeds up their branching.

3.4 Lower Bound

A *lower bound* for a problem instance shows how big an optimal solution has to be at a minimum. These lower bounds can be computed faster than optimal solutions and can help reducing the running time of the algorithm. If the size of the current graph is smaller than the lower bound, then this branch can be aborted. In our algorithm the global lower bound gets determined before the branching begins. Instead of looking at the two-neighborhood ($N_2[v]$) of a vertex v like in the actual algorithm, only direct neighbors are used to compute a lower bound. A vertex v forms a triangle-2-club with its neighbors that are not isolated in $G[N(v)]$. Therefore, a lower bound for v can be determined by counting these non-isolated vertices. Every non-isolated vertex u in $N(v)$ is adjacent to at least one other vertex w. Therefore the vertices v, u and w form a triangle in the graph. Figure 6 shows an example where vertex v_0 is the center of the lower bound computation. By applying this method once with every graph vertex as center a global lower bound can be determined.

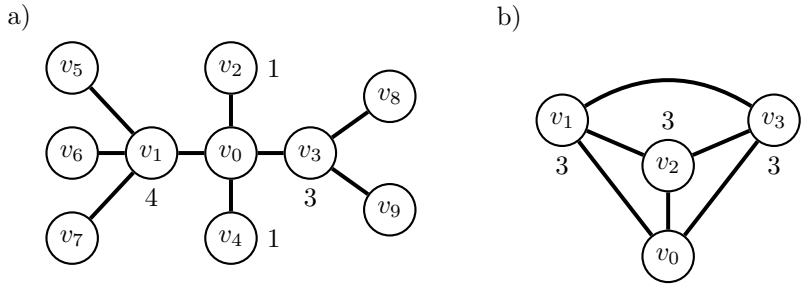

Figure 7: In Figures a) and b) the vertex labeled v_0 is considered the center/start vertex for upper bound computation. All vertices adjacent to these centers have their degree written next to them. For Figure a), the upper bound for vertex v_0 is the sum of the degrees of vertices v_1, v_2, v_3 and v_4 and one for v_0 itself. Therefore the upper bound in a) is $ub'(v_0) = 10$, which is equivalent to $|N_2(v_0)|$. In Figure b) multiple edges between vertices of $N[v_0]$ result in the vertices v_1, v_2 and v_3 getting counted multiple times. The upper bound computed $ub'(v_0) = 10 \geq |N_2(v_0)| = 4$

During the algorithm, the lower bound computed at the beginning gets updated to the current best solution. If the size of the current best solution exceeds the neighborhood size of the next potential start vertex, this start vertex can not lead to a larger solution.

3.5 Upper-Bound

In Subsection 3.3 we described how start vertices can be excluded based on their initial neighborhood size. But removing used start vertices reduces the number of remaining vertices over the course of the algorithm. To handle these changes in the graph, an upper bound can be computed for a vertex before using it as start vertex.

Formally, an upper bound ℓ for a start vertex v in an instance is a natural number, such that every solution s for this instance that contains vertex v satisfies $|s| \leq \ell$.

For the algorithm, the upper bound is the number of vertices that are compatible to a potential start vertex v:

$$ub(v) := |N_2[v]|$$

If the computed upper bound for a start vertex is smaller than the current best solution, this start vertex can be skipped.

In terms of correctness, considering a vertex v as center of a solution, the greatest possible solution containing v is v itself and every vertex that is compatible to v, which is $N_2[v]$.

Actually counting all vertices in a two-neighborhood is expensive. Instead, we may also use the degree of the vertices that are adjacent to the vertex v only:

$$ub'(v) := 1 + \sum_{w \in N(v)} \deg(w)$$

This approach can lead to counting vertices multiple times if they are adjacent in the subgraph induced by v's open two-neighborhood. Since the results of the

second approach—using the vertex degrees—are at least as big as the results of the first one, the computed upper bound can not be smaller than the real one and can therefore be used for skipping start vertices. On the other hand, fewer start vertices may be skipped due to the potentially higher upper bound. Since $ub' \geq ub$, ub' is also an upper bound. Figure 7 shows both upper bounds on different graphs.

4 Implementation

We used Java SE version 1.7 to implement our algorithm. No additional packages were required.

4.1 Graph

To represent a graph $G = (V, E)$ in our implementaion we use of the classes `HashSet` and `HashMap`. For the set of vertices V—labeled with integer numbers—a `HashSet` provides a fast check for presence of an element. The set of edges E is organized in an adjacency list. A `HashMap` uses the integer label of vertices to assign each vertex a distinct `HashSet`, containing its neighbors. This `HashSet` contains the integer labels of every adjacent vertex. This approach causes redundancy by saving every edge twice but it also gives us the ability to use a simple implementation for undeleting vertices during the algorithm.

4.2 Graph Rollback

Making copys of the input graph in order to modify it needs a lot of time and space. The graph implementation provides functionality to delete and undelete vertices.

The actual rollback in the implementation is handled by the *Rollback Stack*. It contains every deleted vertex in the reverse order of deletion. At any point in an algorithm run, a rollback point can be set by saving the element on top of the Rollback Stack. After any number of deletions the graph can be rolled back to the state of the rollback point by undeleting the top element on the Rollback Stack until the saved element appears as top element again.

As described in Section 4.1, the graph uses a `HashMap` to represent adjacency. The `HashMap` contains a `HashSet` of neighbor vertices for each vertex. By retaining the `HashSet` of neighbors for deleted vertices, these deleted vertices can be undeleted in reverse order.

Let $v, w \in V$ be vertices and $\{v, w\} \in E$. When deleting v, its integer label will be removed from w's adjacency `HashSet` and deleted from the `HashSet` of vertices. After the deletion of vertex v, the `HashMap` entry of v remains untouched. To undelete v, its unchanged integer label can be brought back into the vertex `HashSet`. Now, everything left to do is iterating over the neighbors of v and adding v to of their adjacency `HashSet` each.

4.3 Marked Vertices

The set of marked vertices is represented by a `HashSet`. Vertices get marked over the course of the algorithm by the branching or by data reduction rules. For the

marks made by reduction rules, another stack is needed to reverse these changes on rollback. Every vertex that gets marked, also is pushed onto the *Rollback Stack for Marked Vertices* to unmark vertices back to a certain moment.

4.4 Conflict Graph

By computing all incompatabilities between vertices in the two-neighborhood of the start vertex beforehand and updating the conflict graph dynamically during the algorithm, the time required for each compatibility check is constant. In order to construct the conflict graph we need to know which vertices are compatible, hence are close enough in the original graph.

4.4.1 Constructing the Conflict Graph

The construction of the conflict graph has already been described in Subsection 2.5. Recall, that for the construction, we need to determine the compatibility for every pair of vertices of the original graph. For two vertices v and w we can determine if they are incompatible by iterating over the adjacency `HashSet` of v. If one of the vertices in the adjacency `HashSet` of v is also present in the adjacency `HashSet` of w, then v and w are compatible.

4.4.2 Updating the Conflict Graph during the Algorithm

The conflict graph G_C needs to be dynamically updated during the algorithm every time a vertex gets deleted or undeleted in the graph. Since the set of vertices in the conflict graph is the same as in the original graph G, we can imitate every delete or undelete operation that is performed on G in G_C.

Deleting a vertex can cause new conflicts, which leads to new edges in G_C. For the Triangle-2-Club problem the deletion of an vertex v can cause new incompatibilities between vertices in $N_2[v]$. These new incompatibilities for every deleted vertex need stored and then discarded when undeleting v. For this purpose a `HashMap` contains a list of every edge that was added on v's deletion. The integer label of v is used as key in the `HashMap`. As soon as v gets undeleted, every added edge gets removed. This method requires vertices to get undeleted in the exact reverse order they were deleted. For this reason a *Rollback Stack* is added to the graph. The correct order for undeletion is guaranteed through the graph rollback described in Section 4.2.

4.5 Maximal Matching

Recall that a maximal matching chooses edges, and removes them together with their incident vertices until no edges are left in the graph. Actually removing edges for every maximum matching computation is expensive. Instead, we use another `HashSet` to keep track of the vertices of choosen edges. Edges can be found by iterating over the adjacency `HashMap` until a stored `HashSet` is not empty. The key v of this `HashMap` entry and a value w in the stored `HashSet` are two adjacent vertices of the graph. If v and w are not marked yet, then our matching bound algorithm marks them, and therefore considers the edge $\{v, w\}$ as chosen for the maximum matching. The iteration continues until all vertices are marked over their incident edges or the number of marked vertices exceeds the maximum number of vertex deletions described in Section 3.2.

5 Computational Experiments

In this section, we show and evaluate the experimental results of our implementation on real-world graphs. This evaluation includes overall algorithm running time and the benefits of introduced data reduction rules (see Section 3.2). We used graphs from the 10th DIMACS challenge [2], publicly available at http://dimacs.rutgers.edu/Challenges/, the Koblenz network collection [7] and the Network Data Repository with Interactive Graph Analytics and Visualization [11].

5.1 Experimental Setup

All experiments were performed on a single thread of an Intel(R) Xeon(R) Silver 4116 CPU with 2.1 GHz, with 24 CPUs and 128 GB of RAM running JAVA SE on the OpenJDK 14.0.1. The source code was written in JAVA SE 9.0.1. All time measurements started on algorithm call and do not contain the time required to read and initialize the graph itself.

5.2 Evaluation of the Lower Bound

Figure 8 shows the relative size of the initial lower bound solution to the final solution after the algorithm terminated. The computation of the initial lower bound was described in Subsection 3.4. In 60 out of 74 graphs the initial lower bound was equal to the final result. The minimum relative size of all computed lower bounds was 50.6%. On average the initial lower bound was 97.29% of the final result. Computing the initial lower bound took the longest on the graph 'soc-youtube-snap' with 27 seconds needed. We observed the greatest difference between lower bound and final result size on graphs with a very high connectivity. The graph 'graphAllActors', on wich the lower bound performed the worst, contains 1986 vertices and 103.121 edges.

5.3 Start Point Statistics

As mentioned in Section 3.3, the order of start vertices can improve the algorithms runnning time. Tables 2 and 3 show some metrics regarding the choice of start vertices. Ordering the start vertices, in most cases, does not require a lot of time. However, on large graphs, for example 'coPapersCiteseer' it took almost four minutes to sort the vertices in ascending order of their two-neighborhood size. Unfortunatly, the algorithm did not terminate on this graph, therefore we can not put this time in relation to the overall runnning time.

Column SP shows how many of the n vertices of the graph were used as start vertices. The differnce between n and SP origin in the initial application of data reduction rules before the branching. For the majority of graphs, the number of potential start vertices is close to n. In some cases, many vertices can be sorted out by Reduction Rule 3, which removes vertices that are not part of a triangle. Column $t_2[s]$ shows how much time was required to perform this initial data reduction. The application of this initial reduction is cheap, considering how many vertices it removes on sparse graphs, for example graph 'soc-youtube-snap'.

Column Sk shows how many of the start vertices SP got skipped. If the inital neighborhood size of a start vertex v is smaller or equal to the current best solution, then the algorithm will not branch with v as first marked vertex. Otherwise, if the initial neighborhood size of vertex v could potentially produce a solution of greater size, the upper bound for this vertex is computed first. In the course of the algorithm the number of vertices in the graph decreases, and therefore a new computation is required for the upper bound. The number of skips caused by the computed upper bound ub', described in Section 3.5, is shown in column USk. Note that the upper bound skips are included in Sk.

5.4 Best Results

We ran all of the shown graphs in multiple settings. These settings included disabling the application of the matching bound rule and the no-choice rule. This results show the fastest run for each graph where the algorithm terminated. The time limit was set to one hour. All times are in seconds and rounded up to the next 0.1 second. Tables 4 and 5 show these results.

5.5 Comparison to an ILP Solution

In the introduction we mentioned that ILP formulations for the Triangle-2-Club problem have already been proposed by Carvalho and Almeida [3] and Almeida and Brás [1]. We used their results to validate the correctness of our results, as well as to compare the running time of the ILPs to our implementation. The runnning times are compared in Table 1. Carvalho and Almeida [3] showed experimental results of their formulation on real-world graphs for different variations of triangle-k-clubs. The [k2T] problem formulation fits with our definition of the problem as well as the choosen vertex-triangle-property. Unfortunately, their running times were rounded up to seconds. More recent results were shown by Almeida and Brás [1]. We compared our results to the ones produced by their 1-triangle-2-club formulation. If results for the same graph were shown in both of the referenced works, we chose the faster of these results for comparison.

For small graphs, the ILP running time is similar to the running time of our implementation. Considering that times below one second were rounded up, there is no noticeable diffrence in graphs with up to 100 vertices. On medium size graphs, between 100 and 2500 vertices, results get more diverse. The ILP performed better on graphs with a high connectivness, while our algorithm performs better on sparse graphs. On the large-scale graphs our implementation outperforms the ILP formulation. It is to mention that the large graph testet in the refereced papers also were relativly sparse—to the favor of our algorithm—but it were the only results available for comparison.

6 Conclusion

The basic algorithm version was not able to solve medium or lage sized graphs within a reasonable time without the use of data reduction rules or the other proposed ways to make it faster. As shown in Section 5, the basic algorithm benefits a lot from ordering the graphs vertices based on their neighborhood

Table 1: Comparison of the algorithm running time of our algorithm with the time of the ILP solutions. All times presented in the columns 'our' and 'ILP' are measured in seconds. In one of the papers used for comparison the times were rounded up to whole seconds. We have no valid result of our implementation for the polblogs graph, scince the algorithm did not terminate in the time limit of one hour.

Graph Name	n	our	ILP
karate	34	0.1	0.00
dolphins	62	0.1	0.01
huck	74	0.1	1.00
lemis	77	0.1	1.00
jean	80	0.2	1.00
3-FullIns_3	80	0.1	1.00
david	87	0.2	1.00
mug88_1	88	0.1	1.00
1-FullIns_4	93	0.2	1.00
mug100_1	100	0.1	1.00
mug100_25	100	0.1	1.00
polbooks	105	0.3	0.05
adjnoun	112	0.3	0.04
4-FullIns_3	114	0.1	1.00
football	115	0.6	2.96

Graph Name	n	our	ILP
games120	120	0.7	11.00
miles500	128	0.7	6.00
anna	138	0.6	1.00
5-FullIns_3	154	0.1	1.00
jazz	198	9.8	0.76
celegansneural	297	13.2	1.02
email	1,133	71.4	10,334.20
polblogs	1,490	-	31.26
netscience	1,589	0.1	0.18
add20	2,395	437.3	172.43
power	4,941	0.1	0.08
add32	4,960	5.5	176.07
hep-th	8,361	5.0	626.66
PGPgiantcompo	10,680	18.3	490.16

size, as well as the initial computation of the lower bound. Besides the rules for data reduction, the lower bound sticks out the most. For the majority of graphs it finds a solution that is close or equal to a maximum solution. This allows the algorithm to skip a lot of branching.

A crucial component of our algorithms implementation was the ability to rollback the graph instead of making physical copies to work on. Even with the rollback stacks, the implementation had a high need of memory to store the original graph, the graph to work on as well as the conflict graph. This is caused by the used data structures. The timely benefits of the used data structures came with a tradeoff in memory requirements.

The data reduction rules further improved the algorithm. Removing non-triangle vertices before branching reduces the size sparse graphs, and therefore also allowed to skip the branching of these vertices. The incompatible resultion rule also further enabled the removal of non-triangle vertices.

Not all rules for data reduction performed as well as we expected. The matching bound rule only improved runtime on a small number of graphs. In most cases, the costs of constructing and maintaining the conflict graph—which was required in order to use the matching bound rule—exceeded the timely benefits of the rules application. On the other hand, the application of the matching bound rule was able to reduce the depth of branching by aborting branches early. The no-choice rule showed similar flaws. Depending on the graphs structure, in some cases the no-choice rule was able to mark many vertices, wich enabled the incompatible resolution rule to remove vertices. Especially on sparse graphs, it was able to improve the algorithms running time. Otherwise, on graphs with a high connectivity, the no-choice rules costs of application exceeded its benefits.

New or more optimized data reduction rules are a possible toppic for further research. This may allow exact algorithms to perform better than their ILP

counterparts, even on graphs with a high connectivity. In addition, heuristics could be developed to dynamically enable and disable the use of the matching bound rule and the no-choice rule depending on the graphs structure.

A possible improvement to the implementational part of our algorithm would be more optimized data structures to store the graphs. The integer labeling of vertices, as well as the fact that no vertices are added over the course of the algorithm, would allow more efficient ways of storing vertex data.

Another toppic for further research may be the implementation of an exact algorithm for the edge-triangle-property, as well as an adaption of the proposed reduction rules for this problem variant. Other problem variants, for example the ℓ-TRIANGLE-k-CLUB problem which was solved using ILPs by Almeida and Brás Almeida and Brás [1] may be intresting for future research.

7 Appendix

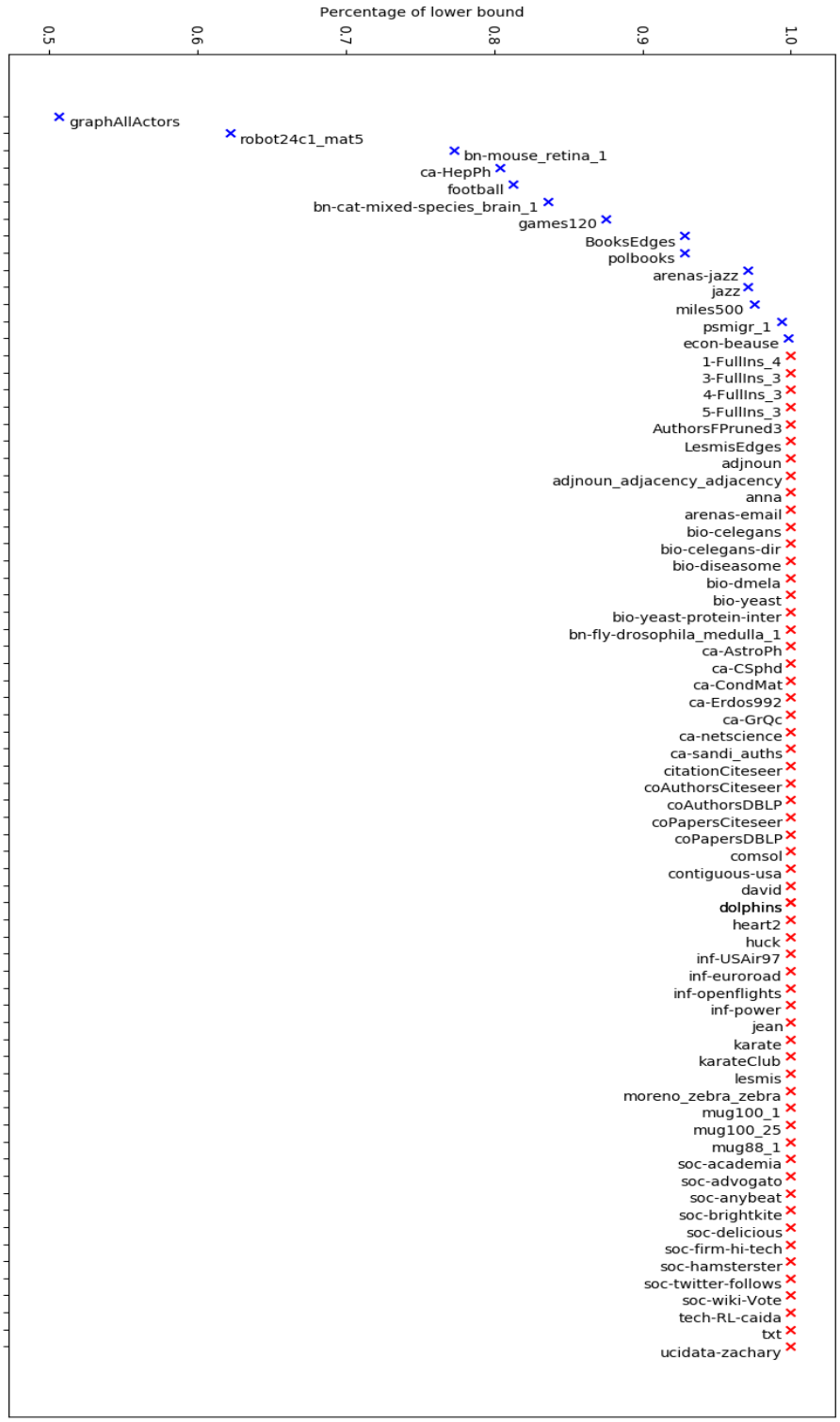

Figure 8: This plot shows the how close the initial lower bound was to the result size. A red marker indicates that the initially computed lower bound is equal to the final result size.

Table 2: Column SP contains the number of vertices that where used as start points. The difference between the number of vertices n and the number of start points SP is caused by the initial application of data reduction rule. This initial data reduction removed non-triangle vertices from the graph. Column Sk is the number of start points that where skipped in any way. USk is how many of the **skipped** start points where skipped because of the computed upper bound ub'. Column t_1[s] contains the time required to sort the start points in ascending order. Column t_2[s] shows how long it took to perform the initial data reduction.

Graph Name	n	SP	Sk	USk	t_1[s]	t_2[s]
moreno_zebra_zebra	27	27	18	9	0.1	0.0
soc-firm-hi-tech	33	29	21	16	0.1	0.1
karate	34	32	23	18	0.1	0.1
ucidata-zachary	34	32	22	17	0.1	0.1
contiguous-usa	49	48	30	16	0.1	0.0
dolphins	62	46	26	13	0.1	0.1
bn-cat-mixed-species_brain_1	65	65	26	21	0.1	0.1
huck	74	67	56	27	0.1	0.1
lesmis	77	57	49	7	0.1	0.1
3-FullIns_3	80	34	34	0	0.1	0.1
jean	80	57	47	7	0.1	0.1
ca-sandi_auths	86	54	51	8	0.1	0.1
david	87	77	55	7	0.1	0.1
mug88_1	88	74	45	16	0.1	0.1
1-FullIns_4	93	77	43	25	0.1	0.1
mug100_25	100	85	47	19	0.1	0.1
mug100_1	100	89	44	24	0.1	0.1
polbooks	105	104	76	63	0.1	0.1
BooksEdges	105	104	76	63	0.1	0.1
adjnoun	112	79	58	25	0.1	0.1
4-FullIns_3	114	40	40	0	0.1	0.1
football	115	115	31	31	0.1	0.1
games120	120	120	35	24	0.1	0.1
miles500	128	128	64	13	0.1	0.1
anna	138	112	97	46	0.1	0.1
5-FullIns_3	154	46	46	0	0.1	0.1
jazz	198	192	92	56	0.1	0.1
inf-USAir97	332	272	189	55	0.1	0.1
ca-netscience	379	351	343	57	0.1	0.1
robot24c1_mat5	404	402	173	140	0.3	0.1
bio-celegans	453	444	373	147	0.1	0.1
bio-celegans-dir	453	444	430	201	0.1	0.1
econ-beause	507	504	39	31	1.5	0.1
bio-diseasome	516	417	415	55	0.1	0.1
soc-wiki-Vote	889	490	414	89	0.1	0.1
bn-mouse_retina_1	1076	1072	2	0	3.0	0.1
arenas-email	1133	840	619	327	0.1	0.1
inf-euroroad	1174	83	81	0	0.1	0.1

Table 3: Column SP contains the number of vertices that where used as start points. The difference between the number of vertices n and the number of start points SP is caused by the initial application of data reduction rule. This initial data reduction removed non-triangle vertices from the graph. Column Sk is the number of start points that where skipped in any way. USk is how many of the **skipped** start points where skipped because of the computed upper bound ub'. Column t_1[s] contains the time required to sort the start points in ascending order. Column t_2[s] shows how long it took to perform the initial data reduction.

Graph Name	n	SP	Sk	USk	t_1[s]	t_2[s]
bio-yeast	1,458	236	230	21	0.1	0.1
comsol	1,500	1,500	0	0	0.6	0.1
bn-fly-drosophila_medulla_1	1,781	1,144	981	520	0.2	0.1
bio-yeast-protein-inter	1,870	262	258	23	0.1	0.1
ca-CSphd	1,882	15	15	0	0.1	0.1
graphAllActors	1,986	1,986	0	0	2.5	0.1
heart2	2,339	2,339	1	0	7.7	0.1
soc-hamsterster	2,426	2,059	1946	567	0.3	0.1
inf-openflights	2,939	2,041	103	0	0.3	0.1
psmigr_1	3,140	3,140	2	0	26.8	0.1
ca-GrQc	4,158	3,348	3,314	347	0.1	0.1
inf-power	4,941	951	948	7	0.1	0.1
ca-Erdos992	5,094	921	856	116	0.1	0.1
soc-advogato	6,551	3,490	3	0	1.1	0.1
bio-dmela	7,393	1,280	988	287	0.1	0.1
AuthorsFPruned3	10,265	9,848	9,833	662	0.3	0.1
ca-HepPh	11,204	9,952	61	0	1.8	0.1
soc-anybeat	12,645	4,931	7	0	2.3	0.1
ca-AstroPh	17,903	16,856	210	0	2.3	0.1
ca-CondMat	21,363	19,474	19,404	2,009	0.7	0.1
soc-brightkite	56,739	25,551	898	0	2.9	0.4
tech-RL-caida	190,914	87,896	87,896	4,087	4.3	0.8
soc-academia	200,169	129,296	1,965	0	25.2	1.0
coAuthorsCiteseer	227,320	196,714	196,688	1128	3.6	0.5
citationCiteseer	268,495	175,480	11	0	12.7	1.1
coAuthorsDBLP	299,067	253,024	252,971	11,557	5.1	0.8
soc-twitter-follows	404,719	16,477	132	0	3.7	2.1
coPapersCiteseer	434,102	428,008	4,194	0	233.2	0.6
soc-delicious	536,108	63,132	9	0	9.2	3.7
coPapersDBLP	540,486	530,300	2,121	0	167.2	0.8
soc-youtube-snap	1,134,890	261,730	261,730	27,660	143.6	3.6

Table 4: Part 1 of best results of all runs. Column T2C is the size of a maximum triangle-2-club in the graph. The algorithm aborted after reaching the set time limit of 3,600 seconds.

Graph Name	n	m	Time [s]	T2C
moreno_zebra_zebra	27	111	0.1	15
soc-firm-hi-tech	33	91	0.1	17
ucidata-zachary	34	78	0.1	15
karate	34	78	0.1	15
karateClub	34	78	0.1	15
contiguous-usa	49	107	0.1	9
dolphins	62	159	0.1	12
bn-cat-mixed-species_brain_1	65	730	0.2	55
huck	74	301	0.1	53
lesmis	77	254	0.1	32
LesmisEdges	77	254	0.1	32
jean	80	254	0.2	32
3-FullIns_3	80	346	0.1	20
ca-sandi_auths	86	124	0.1	11
david	87	406	0.2	73
mug88_1	88	146	0.1	5
1-FullIns_4	93	593	0.2	33
mug100_1	100	166	0.1	5
mug100_25	100	166	0.1	5
polbooks	105	441	0.3	28
BooksEdges	105	441	0.3	28
adjnoun	112	425	0.3	48
adjnoun_adjacency_adjacency	112	425	0.3	48
4-FullIns_3	114	541	0.1	24
football	115	613	0.6	16
games120	120	638	0.7	16
miles500	128	1,170	0.7	40
anna	138	493	0.6	61
5-FullIns_3	154	792	0.1	28
jazz	198	2,742	9.8	103
inf-USAir97	332	2,126	9.9	137
ca-netscience	379	914	0.1	33
robot24c1_mat5	404	14,261	87.6	344
bio-celegans-dir	453	2,025	13.6	238
bio-celegans	453	2,025	41.8	238
econ-beause	507	39,428	2.3	503
bio-diseasome	516	1,188	0.1	49
soc-wiki-Vote	889	2,914	14.3	91
bn-mouse_retina_1	1,076	90,811	3,600.2	-
arenas-email	1,133	5,451	71.4	69
inf-euroroad	1,174	1,417	0.1	5
bio-yeast	1,458	1,948	0.1	15
comsol	1,500	48,119	1,422.1	180
bn-fly-drosophila_medulla_1	1,781	8,911	25.9	899

Table 5: Part 2 of best results of all runs. Column T2C is the size of a maximum triangle-2-club in the graph. The algorithm aborted after reaching the set time limit of 3,600 seconds.

Graph Name	n	m	Time [s]	T2C
bio-yeast-protein-inter	1,870	2,203	0.1	15
ca-CSphd	1,882	1,740	0.1	6
graphAllActors	1,986	103,121	3,601.6	-
heart2	2,339	340,229	3,618.8	-
soc-hamsterster	2,426	16,630	2.8	270
inf-openflights	2,939	15,677	250.6	242
psmigr_1	3,140	410,781	3,600.9	-
ca-GrQc	4,158	13,422	2.7	82
inf-power	4,941	6,594	0.1	14
ca-Erdos992	5,094	7,515	4.8	46
soc-advogato	6,551	39,432	3,634.0	-
bio-dmela	7,393	25,569	248.7	82
AuthorsFPruned3	10,265	37,086	1.4	224
ca-HepPh	11,204	117,619	3,611.6	-
soc-anybeat	12,645	49,132	3,640.3	-
ca-AstroPh	17,903	196,972	3,620.3	-
ca-CondMat	21,363	91,286	867.2	277
soc-brightkite	56,739	212,945	3,309.7	-
tech-RL-caida	190,914	607,610	6.8	1,039
soc-academia	200,169	1,022,441	3,610.0	-
coAuthorsCiteseer	227,320	814,134	45.7	1,234
citationCiteseer	268,495	1,156,647	3,653.3	-
coAuthorsDBLP	299,067	977,676	1,066.8	333
soc-twitter-follows	404,719	713,319	3,621.8	-
coPapersCiteseer	434,102	16,036,720	3,655.3	-
soc-delicious	536,108	1,365,961	3,610.4	-
coPapersDBLP	540,486	15,245,729	3,607.4	-
soc-youtube-snap	1,134,890	2,987,624	171.3	25,698

References

[1] Maria Teresa Almeida and Raul Brás. The maximum l-triangle k-club problem: Complexity, properties, and algorithms. *Computers & Operations Research*, 111:258–270, 2019.

[2] David A. Bader, Andrea Kappes, Henning Meyerhenke, Peter Sanders, Christian Schulz, and Dorothea Wagner. Benchmarking for graph clustering and partitioning. In *Encyclopedia of Social Network Analysis and Mining. 2nd Ed.* Springer, 2018.

[3] Filipa D. Carvalho and Maria Teresa Almeida. The triangle k-club problem. *Journal of Combinatorial Optimization*, 33(3):814–846, 2017.

[4] Salvatore Catanese, Pasquale De Meo, Emilio Ferrara, and Giacomo Fiumara. Analyzing the facebook friendship graph. *Computing Research Repository*, abs/1011.5168, 2010.

[5] Sepp Hartung, Christian Komusiewicz, and André Nichterlein. Parameterized algorithmics and computational experiments for finding 2-clubs. *Journal of Graph Algorithms and Applications*, 19(1):155–190, 2015.

[6] Christian Komusiewicz, André Nichterlein, Rolf Niedermeier, and Marten Picker. Exact algorithms for finding well-connected 2-clubs in sparse real-world graphs: Theory and experiments. *European Journal of Operational Research*, 275(3):846–864, 2019.

[7] Jérôme Kunegis. KONECT: the Koblenz network collection. In *WWW (Companion Volume)*, pages 1343–1350. International World Wide Web Conferences Steering Committee / ACM, 2013.

[8] Robert J. Mokken. Cliques, clubs and clans. *Quality and Quantity*, 13: 161–173, 1979.

[9] Foad Mahdavi Pajouh, Esmaeel Moradi, and Balabhaskar Balasundaram. Detecting large risk-averse 2-clubs in graphs with random edge failures. *Annals of Operations Research*, 249(1-2):55–73, 2017.

[10] Panos M. Pardalos and Gregory P. Rodgers. A branch and bound algorithm for the maximum clique problem. *Computers & Operations Research*, 19 (5):363–375, 1992.

[11] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *Association for the Advancement of Artificial Intelligence*, pages 4292–4293. AAAI Press, 2015.

[12] Alexander Schäfer. *Exact Algorithms for s-Club Finding and Related Problems*. Dissertation, Friedrich-Schiller-Universität Jena, 2009. URL `https://pure.mpg.de/rest/items/item_1587743_4/component/file_1587742/content`.

[13] Manos Schinas, Symeon Papadopoulos, Yiannis Kompatsiaris, and Pericles A. Mitkas. Mgraph: multimodal event summarization in social media using topic models and graph-based ranking. *International Journal of Multimedia Information Retrieval*, 5(1):51–69, 2016.

[14] Qunyan Zhang, Chi Zhang, Peng Cai, Weining Qian, and Aoying Zhou. Detecting spamming groups in social media based on latent graph. In *ADC*, volume 9093 of *Lecture Notes in Computer Science*, pages 294–305. Springer, 2015.

Selbstständigkeitserklärung

Hiermit versichere ich, Philipp Heinrich Keßler, dass ich die vorliegende Arbeit selbstständig verfasst, ganz oder in Teilen noch nicht als Prüfungsleistung vorgelegt und keine anderen als die angegebenen Hilfsmittel benutzt habe. Sämtliche Stellen der Arbeit, die benutzten Werken im Wortlaut oder dem Sinn nach entnommen sind, habe ich durch Quellenangaben kenntlich gemacht. Dies gilt auch für Zeichnungen, Skizzen, bildliche Darstellungen und dergleichen sowie für Quellen aus dem Internet. Mir ist bewusst, dass es sich bei Plagiarismus um akademisches Fehlverhalten handelt, das sanktioniert werden kann.

Ort, Datum Unterschrift