Graph Clustering with Limited Marked Connections

ICA-4141202
Ferenc Balla

Utrecht University

Utrecht University
Department of Information and Computing Sciences
Computing Science

September 11, 2019

Abstract

Cluster editing, also called graph clustering, of graphs with full information is a broadly researched topic. It has its roots in network science and computational biology. Closely related to this area is the research on graph clustering on graphs with missing information, or otherwise marked connections. This research aims to explore whether these marked connections can be used in a way to improve solutions of graph clustering. In this paper a new algorithm is proposed and tested against an existing algorithm for graphs with missing information. Their quality is compared, and the run-time of the existing algorithm will be improved upon with the new algorithm. In the end a new benchmark for algorithms concerning themselves with clustering graphs with marked connections will be given.

1 Introduction

Studying the structures of graphs and how to manipulate them is a broadly studied topic, which has a lot of relevance in todays society. Graphs can represent many things, like traffic networks or social networks. When taking the example of social networks one might be interested in certain kinds of groupings within such a network. Here the concept of cluster editing of graphs, also called graph clustering comes into play. Graph clustering is the act of taking a graph and by changing some connections creating a graph consisting entirely of clusters. By clustering a graph one is able to extract information about the most tightly linked groups. Another example of where graph clustering is a vital task is gene expressions. When information from a gene is used in creating a product in the body, this is called gene expression. When different genes have similar expressions, they should be placed in the same cluster, signifying that the genes have a similar task. The genes do not have to have exactly the same expressions, but if the expressions are close enough an algorithm will be able to group them correctly.

Graph clustering algorithms are usually applied to observed data, which may or may not be accurate. In the case of social networks one might not be able to tell whether certain people have befriended each other if that information is kept private, and in the case of gene expression the similarity between genes concerning certain expressions may simply be unavailable. The degree of accuracy may be available to some extent, but it is usually unused when trying to extract information from the graph. This is the origin of the idea explored in this paper that the existing uncertainty could actually be used to improve available algorithms that deal with uncertain or incomplete information represented in graphs.

This research aims to explore whether using circumstantial information, in this case the notion of uniform distribution of missing information, can improve an algorithm for clustering marked graphs. Furthermore it seeks to improve on the practical run-time of the currently existing algorithm for clustering marked graphs. Finally it seeks to create a benchmark for quality for marked graph clustering problems that currently can not be analyzed.

For this research the focus was limited to undirected graphs with no parallel edges and no loops, called simple graphs.

In Section 2 definitions used in this paper will be explained. Section 3 will delve into the background of this research by relating it to previous work on this subject area. The approach to the problem statement and the hypothesis will be discussed in Section 4. The design and implementation of the algorithms used in this research is detailed in Section 5. In Section 6 the details of the tests chosen to be performed will be laid out. Section 7 will give an overview of the results obtained from running the tests. Furthermore it will explain the implications of these results, and they will be compared to the expectations created in Subsection 4.2. Section 8 will contain the conclusion of the research, give an answer to the problem statement, and summarize the vital points of this paper. In the end there will be discussion of future research material in Section 9, which seeks to lay a foundation for future work in this area, and state additional unanswered questions about the research topic.

2 Definitions

In this paper we consider graphs. Whenever a pair of vertices $\{u, v\}$ is mentioned it will be assumed without further stating this explicitly that $u \neq v$. This paper deals with two types of graphs. The first type consists of a set of vertices V, a set of pairs of vertices E and a set of pairs of vertices N. The set E is defined as the set of pairs of vertices $\{u, v\}$ for which there is a connection between u and v. This is what usually is considered the set of edges. The set N is defined as the set of pairs of vertices $\{u, v\}$, such that $\{u, v\} \notin E$. Pairs of vertices $\{u, v\}$ in E represent an explicit connection between the two vertices u and v. Pairs of vertices $\{u, v\}$ in N represent an explicit disconnection between the two vertices u and v. To make this clear the pairs of vertices in E will be called real edges, and the pairs of vertices in N will be called non-edges.

Throughout the paper this type of graph will be called an unmarked graph, to contrast with the second type of graph that will be discussed later in this section. For unmarked graphs the most well studied graphs are unweighted graphs, but since weights will come into play in some of the used algorithms, any unweighted graph will be turned into a weighted graph. Weights are elements of $\mathbb{Z}$ and are initially set to 1 for every pair of vertices in E and N. This weight is the cost that it takes to transform a real edge into a non-edge, and vice versa.

The second type of graph that is used is a graph similar to the first type, but with an extra set F. Every pair of vertices $\{u, v\}$ belongs to exactly one of the sets E, N or F. Bodlaender et al. [2] calls the pairs of vertices in F fuzzy edges. In that paper they are defined as pairs of vertices about which it is not known whether they are real edges or non-edges. This research builds upon that concept, but tries to generalize the concept of special edges. As such

the general concept of the set of pairs of vertices F is simply defined as the set of pairs of vertices $\{u, v\}$ that have a special marking. That marking could be anything that sets it apart from pairs of vertices that do not receive such a marking, such as the quality of being fuzzy. In this research these pairs of vertices will thus be called marked edges. They will however still be treated as fuzzy edges, in the sense that it is not known whether they are real edges or non-edges and can not have a weight other than 0. Thus for the purpose of this paper a more refined definition will be used. The set of pairs of vertices F is defined as the set of pairs of vertices $\{u, v\}$, such that $\{u, v\}$ is not in E or N and the weight of $\{u, v\}$ is 0. Equivalently the type of graph that contains fuzzy edges was called a fuzzy graph in [2], but in this paper the type of graph that is studied containing marked edges will be called a marked graph.

For unmarked graphs the meaning of common neighbors and non-common neighbors will be defined next. For marked graphs these terms will not be used.

For a pair of vertices $\{u, v\}$ a common neighbor is defined as a vertex w where $w \neq u$, $w \neq v$ and for which the pair of vertices $\{u, w\}$ is in E and also the pair of vertices $\{v, w\}$ is in E.

For a pair of vertices $\{u, v\}$ a non-common neighbor is defined as a vertex w where $w \neq u$, $w \neq v$ and for which one of two situations hold. Situation 1 is: The pair of vertices $\{u, w\}$ is in E and the pair of vertices $\{v, w\}$ is in N. Situation 2 is: The pair of vertices $\{u, w\}$ is in N and the pair of vertices $\{v, w\}$ is in E.

For unmarked graphs a connected component is defined as the maximal set of vertices S for which there exists a path of only real edges between every two vertices in the set. For marked graphs a connected component is defined in exactly the same way.

For unmarked graphs a cluster is defined as a connected component, for which between every two vertices within the connected component there exists a real edge. So a cluster in an unmarked graph is a maximal set of vertices C, for which for all $\{u, v\} \in C$ it holds that $\{u, v\}$ is in E. Such a cluster is more colloquially known in mathematics as a clique. Clusters in marked graphs are defined in a somewhat different way. Fur unmarked graphs a cluster is defined as a connected component, for which between every two vertices within the connected component there exists a real edge or a marked edge. So a cluster in a marked graph is a maximal set of vertices C, for which for all $\{u, v\} \in C$ it holds that $\{u, v\}$ is in $E \cup F$. This is because there is no cost to turn all of the marked edges between vertices in a connected component into real edges, thus forming a cluster as for unmarked graphs.

A clustering algorithm takes as input an unmarked or a marked graph, and gives as an output a graph consisting of only connected components that are clusters. To achieve this edges can be modified. The cost of transforming a real edge to a non-edge is the weight of that edge. The type of graph clustering looked at in this paper focuses on turning non-cluster graphs into cluster graphs

with as little as possible cost.

This research focuses on the fixed parameter tractable approach to the problem like described in [11]. As such every run of an algorithm will be done with a parameter k, which is the most cost that can be expended to find a solution. If an algorithm has determined that the cost must exceed k the algorithm terminates and returns that it has failed to find a solution within k cost.

During the execution of the algorithms pairs of vertices might want to be fixed to being a real edge or a non-edge until the end of the execution of the algorithm. When a pair of vertices is fixed to being a real edge, this fixing is called making them permanent. From that point on that pair of vertices will be called a permanent edge, meaning the connection between those two vertices can not be broken anymore. When a pair of vertices is fixed to being a non-edge, this fixing is called making them forbidden. From that point on that pair of vertices will be called a forbidden edge, meaning no connection can be established between those two vertices.

The quality of a solution for the problem of graph clustering on a marked graph is defined in relation to the solution for graph clustering on the same but unmarked graph. This quality can only be measured if the marked graph is created from an originally unmarked graph, so a comparison can be created. If in one of the solutions a pair of vertices is a real edge and in the other solution a non-edge (or vice versa), that pair of vertices is considered different in the solutions. The quality of a solution for the problem of graph clustering on a marked graph is defined as the sum of the weight of the pairs of vertices that are considered different by the aforementioned definition. This quality can be measured in two different kind of ways. The first and most important measure is when comparing the two solutions looking at the weights in the unmarked graph. In this graph no pair of vertices is marked, and every difference will have a weight greater than 0. The second measure is when comparing the two solutions looking at the weights in the marked graph. In this graph the marked edges have weight 0, so if an pair of vertices that is considered different was a marked edge in the marked graph, this difference will not add to or subtract from the quality. What this second measure does is forgive an algorithm for marked graphs somewhat for differing in its solution with the algorithm for unmarked graphs on pairs of vertices that were marked.

3 Previous work

The grouping of data points comes in many shapes and sizes. In [6] graphs were partitioned into groups such that as few as possible edges cross between groups. Note that the vertices in these groups do not all have to be connected. This task is equivalent to creating a graph consisting of only connected components by removing as few real edges as possible. The paper had the extra constraint of having to create l equal sized groups. Moving away from the grouping of

points to specifically the clustering of points, the clustering of graphs has been described as an interesting topic in computational biology [1]. In this and other studies like in [12] the topic of approximating a solution to the graph clustering problem using heuristics is touched upon. However, as Gramm et al. [10] expressed, for biological problems where full knowledge is available and the underlying structures a perfect solution would give are sometimes too valuable to approximate. The area of exact solutions is thus of more interest for problems without marked edges.

In this area there has been some research on kernelization [5, 11]. Gram et al. [11] give a simple branching strategy with running time $\mathcal{O}(2.27^k)$. There is also some research towards explicitly improving the branching of the algorithm [10], where a branching strategy with running time $\mathcal{O}(1.92^k)$ was developed. The core of these papers is the basis for the unmarked algorithm implemented in this paper.

There has been research about the implementation of this algorithm as well [7]. The most important conclusion from this paper is that an algorithm based on [11, 10] is as expected faster than "*LP based and greedy based methods for cluster editing*". Another area in this research that became relevant later on in this research is the clustering of weighted version of unmarked graphs. The papers [3, 8] discuss this problem. [3] gives a branching strategy with $\mathcal{O}(1.82^k)$ running time. However, since the starting graphs in this paper are unweighted, this research forgoes implementing this improved running time, since it only focuses on weighted graphs, which is not always used in this line of research.

Then there is research on the topic of clustering graphs with missing information. There has been research [4] that explored among other things the notion of graphs containing uncertainty or missing information, referring to them as general graphs (as opposed to complete graphs where the presence or absence of every edge at the start is known). The foundation for the research in this paper was laid by Bodlaender et al. [2]. It discusses the kernelization of graphs with missing information. It focuses on producing a good kernel for branching towards a cluster graph. However, this research does not consider the impact of the missing information on the result, which this paper aims to do. Furthermore, the result given, while having a strictly deterministic kernel is very slow, due to a very quickly growing exponential running time which limits its practical implementation, as will be shown in this research.

4 Problem approach

The research question seems to lend itself to be answered well through experimentation, since the use of circumstantial information is hard to quantify only theoretically. The initial approach was to replicate the existing algorithm for marked graphs, and to improve upon it by extra restrictions, trying to force the result in the direction of what is considered by this research to be a result of higher quality.

During testing however it came to light that the existing algorithm for marked graphs can in practice only be run with very low parameters. The results in Gramm et al. [11] concerning the clustering algorithm for marked graphs that are presented as interesting are the polynomial time kernel reduction, and the kernel size of $\mathcal{O}(k^2 + r)$. And while mathematically that is a neat result, the branching process that would follow, which is necessary to actually get a cluster graph, is not really highlighted in the paper, but tucked away at the very end of the third section of that paper. There the authors acknowledge a run-time of $\mathcal{O}((k^2 + r)^{2k})$ for the entire process, where $k^2 + r$ is the kernel size, the 2 in the exponent is for getting the number of edges (real edges, non-edges and marked edges) in the kernel, and the k in the exponent is the input parameter.

In the problem with unmarked graphs the total running time was $\mathcal{O}(2.27^k + |V|^3)$, where $|V|$ was the number of vertices, 2.27 was the branching factor, and the k in the exponent was the input parameter. The difference between the two running times is the way in which branching is done. In the problem with unmarked graphs the branching was done by picking one of three pairs of vertices (where the possibilities were reduced to 2.27 per three pairs of vertices with clever math) per branching step, where in the problem with marked graphs the branching is done by picking one random pair of vertices out of the kernel per branching step, giving $\mathcal{O}((k^2 + r)^2)$ options per branching step.

Two points can be made about the algorithm for unmarked graphs not being much faster than the algorithm for marked graphs. Firstly, having a parameter k in the exponent does prevent either of the algorithms from ever solving problems where many steps are required. Secondly the $\mathcal{O}(|V|^3)$ does prevent the algorithm for unmarked graphs from running on graphs with very many vertices because of the long kernelization time.

However, as for the second point: extremely large graphs with very few edits required to create cluster graphs is not a very interesting problem. This is because assuming the pairs of vertices to be changed are distributed uniformly most cases would consist of having a few clusters with one non-edge, or having one real edge between two otherwise (almost) complete clusters. This is demonstrated by the following randomly generated graphs with a large number of vertices and a small number of steps to a solution:

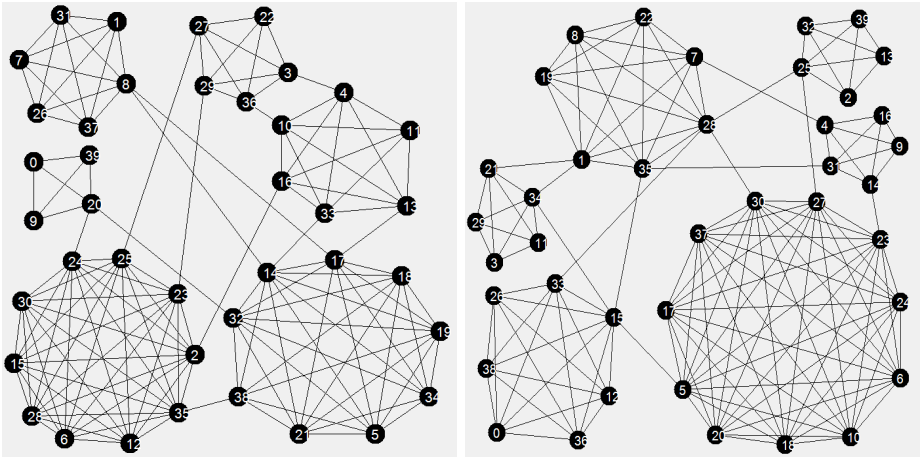

Figure 1: 2 randomly generated graphs with each having 40 vertices divided into six clusters. Graphs are generated by same algorithm used for testing. Visualization is by a self-written force-spring model.

The graph on the left requires fourteen steps to make into a cluster graph, the right graph requires twelve steps. Parameters of $k = 12$ are quite high as will be visible in the results in Section 7. Both graphs are not even that large, and already it is clearly visible that even when they are quite some steps away from a cluster graph, clustering is only a trivial matter, and does not reveal new and interesting information. The interesting problems are where multiple cluster graphs can be made with the same order of magnitude number of steps, so not being able to run the algorithm on graphs with many vertices with a low parameter is not an issue.

The first point of not being able to solve problems where many steps are required is somewhat valid since as Section 7 will show both algorithms fall of at a parameter k that is not too high. However, there is still a significant difference in the parameters with which the algorithms can still be run. After some experimenting it has been established that using the algorithm for unmarked graphs about 500 branching operations per second could be done, and using the algorithm for marked graphs about 270 branching operations per second could be done, which was later reduced to 20. This reduction will be elaborated upon in Subsection 5.4. The difference between 500 and 270 comes from the overhead of setting every marked edge at the end of each branch, and is thus somewhat dependent on the number of marked edges left before starting branching. For the sake of simplicity 270 and for later runs 20 will be used in calculating the cutoff point instead of setting a hard time limit, since most graphs will not perform orders of magnitude slower or faster. Since the branching number of 2.27 is larger than 2 (and the branching number for the algorithm for the marked graphs is definitely larger than 2), the number of branches approaches the num-

ber of branching operations, so it can be assumed that respectively about 500, 270 and 20 branches per second can be checked. The following table gives the time both algorithms would take for their branching phase for a few parameters that might want to be used, given that respectively 500, respectively 270 and 20 branches per second can be visited.

k	run-time in s		k	run-time in s		k	run-time in s
2	0.01		8	1.41		14	192.92
3	0.02		9	3.20		15	437.94
4	0.05		10	7.26		16	994.13
5	0.12		11	16.49		17	2256.67
6	0.27		12	37.44		18	5122.65
7	0.62		13	84.98		19	11628.44

Table 1: Expected run-time of branching in algorithm for unmarked graphs

k	r	run-time in s		k	r	run-time in s		k	r	run-time in s
2	4	15.17		3	0	1968.30		4	0	15907286.28
2	5	24.30		3	1	3703.70		4	1	25836138.67
2	6	37.03		3	2	6561.33		4	2	40814668.80
2	7	54.22		3	3	11059.20		4	3	62902085.34
2	8	76.80		3	4	17877.07		4	4	94814814.81
2	9	105.78		3	5	27887.17		4	5	140084664.30

Table 2: Expected run-time of branching in algorithm for marked graphs in runs with lower parameters

k	r	run-time in s		k	r	run-time in s		k	r	run-time in s
2	4	204.80		3	0	26572.05		4	0	214748364.80
2	5	328.05		3	1	50000.00		4	1	348787872.10
2	6	500.00		3	2	88578.05		4	2	550998028.80
2	7	732.05		3	3	149299.20		4	3	849178152.10
2	8	1036.80		3	4	241340.45		4	4	1280000000.00
2	9	1428.05		3	5	376476.80		4	5	1891142968.00

Table 3: Expected run-time of branching in algorithm for marked graphs in runs with higher parameters

From Table 2 and Table 3 it is clearly visible that the pool of problems that can be tackled with the current algorithm for marked graphs is very limited in theory, while Table 1 shows that problems with a respectable size (considering it is an NP-complete problem) can be solved by branching with the algorithm for unmarked graphs. This was the inspiration to not only make a new algorithm which performs better in quality of its result, but also orders of magnitude faster

than the current existing algorithm for marked graphs, so it can be applied to small, but real problems, and not only to toy problems of very small size.

The newly developed algorithm is created with a run-time in the order of magnitude of the algorithm for unmarked graphs. As such besides comparative testing with the current algorithm for marked graphs a new benchmark will be created for the solving of marked graphs of larger size. In the following subsection the tests to be run will be laid out. Section 5 will then spell out the exact design and implementation of the algorithms.

4.1 Methodology

As mentioned, the main goal was to improve on the quality of the result of the algorithm, and to vastly improve its run-time. The quality of the result has been described in Section 2, but to shortly reiterate: The quality of a solution to the problem of clustering a marked graph, is the distance in cost from the solution to the problem of clustering the same but unmarked graph. This can of course only be measured if the unmarked graph is available. In reality this will not be the case, but that is what this experimentation is for, to test how well the algorithms perform, so when this information is not known there is an indication of how close the solution is to the optimal one (on average). To get this measure the marked graphs must be created from unmarked graphs, so the results from two different clustering algorithms can be compared.

For different input parameters k and different graph sizes tests will be run. For each combination of k and graph sizes multiple tests will be run. For lower k and lower graph sizes more of these tests will be run, and for higher k and graph sizes fewer. This is because for higher k and graph sizes it is harder for the algorithms to find solutions. To give the algorithms a chance to find a solution at higher k and graph sizes more time per run will be available to the algorithm, but as a trade-off fewer tests are performed to compensate for the extra time this costs.

A single test consists of six main parts. The first is the generation of an unmarked graph. This is done with an approximate target distance d from a cluster graph, which will be explained in Subsection 5.1. Then this graph is marked by a marking method that is explained in Subsection 5.2. Then the three algorithms are run, with parameter k being $d+2$. This difference between the expected distance and parameter is so more tests actually succeed, because for this research results where a result could not be found because k was too low are not usable, since they can not be compared, nor say something about the running time of the algorithms. The algorithm for unmarked graphs is run on the unmarked graph, which will be elaborated on in Subsection 5.3, and the existing and new algorithms for marked graphs, which are going to be laid out in Subsections 5.4 and 5.5, are run on the marked graph.

Finally the resulting cluster graphs will be analyzed in comparison to each other and the input graphs. The information gathered from each test consists

of the following:

- Whether both algorithms for marked graphs were compared to the algorithm for unmarked graphs. This is only applicable when all three algorithms successfully found a cluster graph. The next items are only considering these tests:
- The number of marked edges in the marked graph
- The number of clusters produced by the algorithm for unmarked graphs
- The cost of the steps taken by the algorithm for unmarked graphs to find a cluster graph
- The cost of the steps taken by the existing algorithm for marked graphs to find a cluster graph
- The cost of the steps taken by the new algorithm for marked graphs to find a cluster graph
- The cost of the difference between the result of the algorithm for unmarked graphs and the result of the existing algorithm for marked graphs, when taking the weights from the unmarked graph
- The cost of the difference between the result of the algorithm for unmarked graphs and the result of the existing algorithm for marked graphs, when taking the weights from the marked graph
- The cost of the difference between the result of the algorithm for unmarked graphs and the result of the new algorithm for marked graphs, when taking the weights from the unmarked graph
- The cost of the difference between the result of the algorithm for unmarked graphs and the result of the new algorithm for marked graphs, when taking the weights from the marked graph

---

- Whether the new algorithm was compared to the algorithm for unmarked graphs. This is applicable when at least the algorithm for unmarked graphs and the new algorithm for marked graphs found a cluster graph. Whether the old algorithm for marked graphs successfully found a cluster graph is irrelevant here. The next items are only considering these tests:
- The number of marked edges in the marked graph
- The number of clusters produced by the algorithm for unmarked graphs
- The cost of the steps taken by the algorithm for unmarked graphs to find a cluster graph

- The cost of the steps taken by the new algorithm for marked graphs to find a cluster graph
- The cost of the difference between the result of the algorithm for unmarked graphs and the result of the new algorithm for marked graphs, when taking the weights from the unmarked graph
- The cost of the difference between the result of the algorithm for unmarked graphs and the result of the new algorithm for marked graphs, when taking the weights from the marked graph

---

- Whether the algorithm for unmarked graphs failed because no solution could be found.
- Whether the existing algorithm for marked graphs failed because no solution could be found.
- Whether the new algorithm for marked graphs failed because no solution could be found.
- Whether the algorithm for unmarked graphs failed because it ran out of time.
- Whether the existing algorithm for marked graphs failed because it ran out of time.
- Whether the new algorithm for marked graphs failed because it ran out of time.

Then the average (where applicable) of each test performed per k and input size is calculated and stored. This data will then be analyzed to conclude answers for the three questions of this research. Firstly whether the new algorithm performed better in accordance with the given quality measure than the existing algorithm. Secondly whether larger parameters and thus larger and more complex graphs can be clustered. And if so thirdly what the minimum quality is expected of future algorithms on problems with those k and graph size, by giving a benchmark.

4.2 Hypothesis

There are two questions this paper aims to answer. The first one being whether making use of available information in the form of marked edges can improve the quality as defined in this paper of cluster graphs created from marked graphs. Since the decision has been made to also make the algorithm an approximation algorithm, the question the results will more specifically answer is whether making use of available information in the form of marked edges can improve the

quality as defined in this paper of cluster graphs created from marked graphs **even with an approximation algorithm**.

The expectation is that the new algorithm will perform quite well. Because few marked edges will end up in clusters on average, very few clusters will be formed that are not present in the solution of the algorithm for unmarked graphs, and thus since the distance between the results will be small, the quality will be considered very good, even though the algorithm is done through approximation.

However, since the parameters k used to compare the existing algorithm for marked graphs and the new algorithm for marked graphs are expected to be quite low because of the expected intolerance of the existing algorithm for marked graphs for high parameters k, it is also expected that the existing algorithm for marked graphs will perform decently well. With low parameters simply few mistakes can be made. The strongest result would come from a lot of marked edges that get set differently, which could make a difference in quality, but this could very well average out to a low amount of incorrectly determined marked edges with such small graphs.

In conclusion, the hypothesis about the quality difference is that the new algorithm for marked graphs will perform only slightly, maybe negligibly better than the existing algorithm for marked graphs on such small parameters.

The second question is whether the new algorithm for marked graphs can take on larger problems, in this case larger graph sizes and higher parameters k.

The hypothesis for this is that the new algorithm for marked graphs will vastly outclass the existing algorithm for marked graphs, and will be able to solve a lot more problems, from low to high parameters k and graph sizes, and that there will be a limit where the existing algorithm for marked graphs will simply not be able to solve anything, where the new algorithm for marked graphs can still come up with solutions.

5 Design and implementation

There are five main parts of the implementation that are relevant for this research. In this section the how and why of the design of each of the parts will be explained, and a description of the practical implementation for each of the designs will be given.

5.1 Generating unmarked graphs

Generating unmarked graphs is done according to four given parameters. These are:

- Number of vertices in the graph n
- Number of clusters in the graph c

- Probability of a pair of vertices belonging to the same cluster being a real edge a
- Probability of a pair of vertices belonging to different clusters being a real edge b

The clusters are created using a bucket system. In this system c buckets are created, where each bucket represents a cluster. Each bucket gets assigned at least one element. Then the rest of the elements are randomly divided over the buckets. This is somewhat resemblant of the $G(n, M)$ variant of the Erdős-Rényi random graph model as defined in [9].

Next the edges are created according to the $G(n, p)$ variant of the Erdős-Rényi random graph mode as defined in [9]. In this model each edge has a probability of existing independently from the other edges. This translates to our model in the way that each pair of vertices has a probability of being in the set E or N independently from the other pairs of vertices. In this case the probability of whether a pair of vertices $\{u, v\}$ is in E depends on whether u and v are put in the same bucket in the previous step, representing whether they are supposed to be in the same cluster or not. The probability that a pair of vertices belonging to the same cluster are initialized as a real edge is a, and the probability that they are initialized as a non-edge is $1 - a$. Correspondingly the probability that a pair of vertices belonging to different clusters are initialized as a real edge is b, and the probability that they are initialized as a non-edge is $1-b$.

This way of creating graphs ensures control over the approximate number of connected components, and thus clusters in the graphs (whether there are a lot or a few) without exactly predetermining the graph. Furthermore it gives some control over how far the generated graph should stray from a cluster graph. If the first probability would be set to 1 and the second to 0 one would end up having generated a cluster graph. The further those probabilities are moved from those values, the more steps it will take to find the set of clusters determined during the bucketing process, and even a different cluster graph within fewer steps might be found, which enforces the idea that this generation is not too deterministic.

To evenly distribute the pairs of vertices that are keeping the graph from being a cluster graph, all pairs of vertices should have equal odds of not being as would be expected in a cluster graph based on the initial buckets. This means that the odds of a pair of vertices that are not in the same cluster being a real edge should be the same as the odds of a pair of vertices that are in the same cluster being a non-edge, thus a must be $1 - b$. Since a and b can now be expressed in one another, the number of steps that would be necessary to turn the graph into a cluster graph can now simply be approximated, which is $n * b$ (or $n * (1 - a)$). Reversely, if the graph size and desired testing parameter k is known, the probabilities of a and b can be calculated such that the resulting graph has on average k pairs of vertices that need to be changed to get a cluster graph based on the initial buckets. However, since this is a probability distribution, sometimes less and sometimes more k is desired. As mentioned in Section

4.1, because of this the probabilities are calculated with expected distance d as $a = 1 - (d/n)$ and $b = d/n$, and for the corresponding test set the parameter k will be set as $k = d + 2$ to ensure that more of the graphs are solvable with k or fewer steps.

The generator does have a small fail-safe built in, where the probability of a pair of vertices belonging to the same cluster can not be set lower than the probability of a pair of vertices belonging to different clusters, because that would result in all of the clusters becoming loosely connected components, which would be connected to most or all of the other loosely connected components, resulting in a tangled mess where no cluster graph could be found in a reasonable number of steps.

Finally after this generation is complete all the real edges and non-edges get the weight 1. This represents an unweighted graph, the only starting unmarked graphs that will be experimented with. However, having weighted pairs of vertices will be necessary in algorithms later on, so they are created in advance.

5.2 Marking unmarked graphs

In this research a pair of vertices being marked represents a lack of knowledge about whether the connection between those two vertices would be observed as present or as absent if it would be possible to observe it, so whether they should be real edges or non-edges in the graph. If a pair of vertices is marked in an otherwise tight cluster, it might seem obvious that it should be a real edge. If a pair of vertices is connecting two disjointed clusters, it might seem obvious that it should be a non-edge. Later on it will be shown that this 'obviousness' is handled by the kernelization part of the algorithms. However, whether you can easily guess the true state of a marked pair of vertices or not, it does not affect the reason that it is marked, namely that somewhere in the graph there is missing data. In this research the assumption was made that the lack of knowledge is uniformly distributed, so every pair of vertices has equal odds of being marked.

From this assumption the choice was made to pick a certain percentage of pairs of vertices to be marked, and to mark that number of pairs of vertices picked randomly. This is again akin to the $G(n, p)$ variant of the Erdős-Rényi random graph mode as defined in [9].

5.3 Algorithm 1.

The first algorithm, which solves the problem of graph clustering on unmarked graphs has its implementation based on Gramm et al. [11]. The paper describes a kernelization algorithm consisting of three rules. It goes on to describe a simple branching strategy yielding a run-time together with the kernelization of $\mathcal{O}(3^k + |V|^3)$ and a more refined branching strategy yielding a run-time together with the kernelization of $\mathcal{O}(2.27^k + |V|^3)$.

5.3.1 Kernelization

The implemented algorithm firstly performs the kernelization as described by the three rules. Their exact definitions and the proof for their correctness and time bound of these rules can be found in the paper they originate from. Here only a rough idea of their working and their implementation will be given. These rules are designed for unweighted graphs, but they can be used for weighted graphs as well. Wherever the description of the rules speak of a number of pairs of vertices, for weighted graphs you could replace the value with the total weight of those pairs of vertices. This is also how every rule was implemented.

Design of Rule 1. *Rule 1* states that if two vertices u and v have more than k common neighbors, which as mentioned in Section 2 are the vertices they have in common, it would take more than k steps to separate them for the end result, so the two vertices would have to be connected, and thus the pair of vertices $\{u, v\}$ should be a real edge. If $\{u, v\}$ is a non-edge, turn it into a real edge, and reduce k by the weight of the pair. Then, when they are connected the pair of vertices should be marked as permanent, indicating the pair of vertices can not be changed anymore. The same goes for two vertices $\{u, v\}$ that have more than k non-common neighbors, which as mentioned in Section 2 are vertices being neighbors of only one or the other of the two vertices. These two vertices can not be connected in the end result, as it would take more than k steps to join all the neighbors of one vertex to the other and vice versa. In the same way as previously, the pair of vertices $\{u, v\}$ should be turned into a non-edge if it is a real edge (again reducing k by the weight of the pair of vertices), and in any case be made forbidden. If both of the cases would apply it is obvious that a conflict is reached, and there is no solution for the entire graph for the given k.

Implementation of Rule 1. This rule is implemented by creating two queues in which initially pairs of vertices are placed that either have more than k common neighbors, or have more than k non-common neighbors. Then a loop is performed until both queues are empty. In each loop an element from a non-empty queue is taken, and the appropriate above described action is performed on it. Then a check is done if turning pairs of vertices from real edges into non-edges and vice versa results in more instances of Rule 1 than in the initial assessment. If these occur, the pairs of vertices for which the rule newly holds are placed in the queue. These pairs of vertices however can not be permanent or forbidden. If a pair of vertices is already permanent, and according to this newly triggered rule it should be made forbidden (or vice versa), then there is a contradiction, and the algorithm fails, since no solution can be found within k steps.

Design of Rule 2. Rule 2 states firstly that for every triangle of vertices, if two pairs of vertices are known to be permanent, the last pair of vertices must be permanent, since otherwise those three vertices will be part of a connected component that is impossible to make into a cluster. Secondly it states that

for every triangle of vertices, if one pair of vertices is known to be permanent and one pair of vertices is known to be forbidden, the last pair of vertices can not be permanent (and thus must be forbidden), because if it were to be made permanent those three vertices would be part of a connected component that would be impossible to make into a cluster. The rule does not say anything about triangles of vertices for which two pairs of vertices are known to be forbidden, since the last pair of vertices could still be a real edge or a non-edge (so forbidden or permanent), while all of the vertices could still end up in valid clusters.

Implementation of Rule 2. For this rule to trigger some pairs of vertices will already have to be permanent or forbidden. So in the implementation there is no preprocessing for this rule. This rule is implemented within the loop mentioned in the implementation of the first rule. When the first or the second rule triggers and indicate that a pair of vertices will have to be made permanent or forbidden, it is placed in a queue. When it is dequeued and the appropriate action is taken, every triangle of three vertices that this newly set permanent of forbidden pair of vertices is in is checked, to see if Rule 2 holds in any way. If it does, the pair of vertices in the triangle of three vertices that must be made permanent or forbidden is placed in the corresponding queue that was created for Rule 1. Here the same contradiction rule holds, if after checking Rule 2 a pair of vertices can be found in both queues no valid end result can be found, and the algorithm terminates. Incidentally this check is performed after the check for Rule 1, but since after setting the pair of vertices no other action to change a pair of vertices is performed in the loop, these two could just as well have been swapped up.

Rule 3. Rule 3 states that all connected components which are clusters can safely be deleted from the graph.

The implementation of this rule is quite straightforward. After all actions for Rule 1 and 2 have finished a depth first search is performed from every (not yet visited) vertex, and by counting the vertices and checking if the degree of every vertex in the search tree matches the number of vertices in the search, it can be determined if a connected component is a cluster, and if so its vertices are marked to be removed from the graph.

5.3.2 Branching

After the kernelization is done and some vertices are not yet marked to be removed from the graph it means that according to Rule 3 not every connected component is a cluster yet. A branching will have to be performed to see if in the remaining k steps a cluster graph can be found. Before the branching is performed a new graph is created and every vertex from the kernel that is not marked to be removed is copied to it, preserving the state of the pairs of vertices. In practice this speeds up the branching process, compared to performing the branching algorithm on the entire graph, because every pair of vertices that is not in the kernel anymore does not have to be checked anymore. However,

for the end result all of the vertices are still required, so the removed clusters are saved separately, and if branching has successfully terminated, they are all reinserted, giving the final cluster graph.

Design The branching process in [11] is described as follows: While k is larger than 0 and the graph is not a cluster graph yet, at least one set of three vertices must be present for which two of the pairs of vertices are real edges, and one is a non-edge. This structure being present is preventing a connected component from being a cluster. When such a triplet is found, a branching should be done. Two different branching structures are described. The first branching described is a branching in three ways. The first branch, called B1, is to turn one of the pairs of vertices that is a real edge into a non-edge and make it forbidden. The second branch, called B2, is to turn the other pair of vertices that is a real edge into a non-edge and make it forbidden. Furthermore in this branch the pair of vertices that is still a real edge has to be made permanent, and the other pair of vertices that is a non-edge also has to be made forbidden. The third and last branch, called B3, is to turn the pair of vertices that is a non-edge into a real edge, and make all three pairs of vertices permanent. This gives a branching factor of 3, and thus results in a run-time of $\mathcal{O}(3^k)$ for the branching part of the algorithm. This branching was initially implemented.

The paper also gives a more elaborate branching structure, resulting in a lower branching factor. This branching structure still branches on the same structure of three vertices, for which two of the pairs of vertices are real edges, and one pair of vertices is a non-edge. However, it takes in consideration a shortcut that can be taken. The branching is divided in three cases, of which only one can occur at a time. The first case is that the two vertices that form a non-edge do not share any common neighbor other than the one in the triangle they are found in. In this case only Branch B1 and B2 are considered, so a branching number of two applies for this case. In the second case the two vertices u and v that form a non-edge do share at least one common neighbor x other than the one in the triangle they are found in, and that common neighbor is connected to the third vertex in the triangle w. In this case five branches can be identified. The exact layout and proof for these branches can be found in the paper by Gramm et al. [11]. In short, the five branches consist of respectively 1, 2, 2, 3 and 3 steps each. For some branches multiple steps are taken, because taking one step, when considering the fourth vertex x leads to forced extra steps. The last case looks similar the second, except the fourth vertex x is only connected to the two vertices u and v in the triangle that form a non-edge, and not to the other edge in the triangle w. Here a different branching is done, but it still results in five branches which take 1, 2, 2, 3 and 3 steps each respectively. From a simple recurrent relation, $a^k = a^{k-1} + a^{k-2} + a^{k-2} + a^{k-3} + a^{k-3}$, the branching number for this branching can be calculated to be 2.27, as indicated by the paper. This branching was eventually implemented for this research and used in the testing.

Implementation The implementation of both branchings looked very similar. The very first thing is to check whether k is larger than 0. If not, the branching terminates and returns a value indicating the branch has failed. If it holds the branching on a particular branch actually starts. The branching starts with a check whether all vertices are in clusters yet. If so the branching on this branch terminates and returns the remaining k and the solution found. If not a search is started over all triplets of vertices until a conflict triple is found, a set of three vertices for which two pairs of vertices are real edges and the third pair of vertices is a non-edge. As mentioned in the design such a conflict triple must exist, otherwise all vertices would be in clusters. In the improved version at this point it is determined which of the three cases is occurring. After the case is determined (or simply right away for the simpler branching) the algorithm makes the branching change as described in the design, it changes the appropriate pair(s) of vertices, and lowers k by the correct amount for that branching step. After this it recursively calls itself. When a branch is done and returns, it saves the result to be used later if it is an improvement over the best one so far, then sets the changed values back to before the branching, so the graph looks as if that branch has not been gone into. After this it performs the same actions on the remaining branches.

The two implementations of the branching algorithm are very similar, only with the second one requiring a few extra checks, and a lot of extra branches. The order in which the vertices are checked remains the same. However, during testing it has come to light that the fourth vertex in the second and third case of the more advanced branching, which is in the implementation always chosen as the highest numbered vertex in the cases that it presents itself, influences the order in which the branches are traversed. In the current implementation a new result that is equal to the previous best result is not considered, whether the resulting graphs are the same or different, only improvements are considered. The influence on the order was visible from the fact that running the (deterministic) algorithms back to back on the same graph gave equally good but different solutions. Not all graphs will have different solutions that can be found in the same amount of steps, but for the ones that do the choice of branching might influence the outcome. This is expected to affect the outcome of this study, since the measure of accuracy does depend on the closeness to the solution found by this algorithm. However, as will be elaborated on more later on, the newly designed algorithm does save all of the different solutions with the lowest cost, so if it performs well, at least one of the solutions should resemble the solution chosen by the algorithm for unmarked graphs.

5.4 Algorithm 2.

The second algorithm, which solves the problem of graph clustering on marked graphs has its implementation based on the paper by Bodlaender et al. [2]. The paper describes the graph clustering again through a kernelization based on three rules, and an unmentioned branching process. The run-time the paper mentions is $n^{\mathcal{O}(1)} + \mathcal{O}((k^2+r)^{2k})$. The polynomial time factor at the start of the

expression signals that all operations in the kernelization process can be done in polynomial time, and the second term indicates how long the branching will take, considering the kernel size.

5.4.1 Kernelization

The first thing to consider for this algorithm is its kernelization. This consists of two simple but nevertheless important rules, and a rule on which the kernel size hinges.

Rule 1. The first rule states "***Rule 3.1.1.*** *If there is a connected component C with no non-edges, remove C*". The implementation of this rule is done by checking for connected components connected by real edges, and then checking whether these components had any non-edges between any two vertices. If not the connected component is a cluster as defined for marked graphs, so it can be removed. This is done by storing all of the vertices in the cluster in a list of vertices not to be considered for the rest of the kernelization.

Rule 2. The second rule states "***Rule 3.1.2.*** *If Rule 3.1.1 does not apply and there are more than $k+1$ connected components, then answer NO*". This rule is implemented by counting all connected components that do have non-edges between their vertices. If this number is greater than k, the algorithm terminates.

Rule 3. The third rule is a way to merge inseparable vertices. This is very reminiscent of the first part of Rule 1 in the kernelization part of the algorithm for unmarked graphs. However, because this algorithm is dealing with marked vertices that have weight and thus cost zero, the requirement and execution is defined somewhat differently. The requirement for the rule to trigger is the presence of two vertices u and v where $u \neq v$ which have a minimum cut between them of at least $k+1$. If that is the case the two vertices are fused together to be represented by only one vertex for the rest of the algorithm. Lets call this vertex z. The connection of z to all of the other vertices is handled in one of three ways, depending on the situation.

For every vertex w where $w \neq u$ and $w \neq v$ one of the following three situations can occur. Situation 1: The pair of vertices $\{u, w\}$ is marked. In this situation the new pair $\{z, w\}$ gets put into the same set that $\{v, w\}$ is in (E, N or F) and inherits the weight of $\{v, w\}$. If instead $\{v, w\}$ is marked the same happens the other way around. Situation 2: The the pairs of vertices $\{u, w\}$ and $\{v, w\}$ both are real edges or non-edges. In that case $\{z, w\}$ will be of the same type, and the weight of $\{z, w\}$ will be the sum of the weights of $\{u, w\}$ and $\{v, w\}$. Situation 3: $\{u, w\}$ is a real edge and $\{v, w\}$ is a non-edge or the other way around. In this case the values are subtracted from each other, and the result will be the weight of $\{z, w\}$. The pair of vertices $\{z, w\}$ will be a real edge if the weight of the real edge is higher than the weight of the non-edge and

vice versa. Furthermore the weight of the pair of vertices with the lower weight is subtracted from k as in essence this is a number of steps taken towards a solution. If however $\{u, w\}$ and $\{v, w\}$ have equal weight they cancel each other out. In this case $\{z, w\}$ becomes a marked edge, since it would get a cost of 0. In this case also steps are taken towards the solution, and the weight of $\{u, w\}$ (or of $\{v, w\}$ since they are equal) is subtracted from k.

After the new edge is established, the merger of the two old vertices will have to be acknowledged. If they formed a non-edge before the merger, the weight of the pair of vertices them is subtracted from k, since the merger represents the pair becoming a real edge.

If now k has become lower than 0 the algorithm fails, since a necessary step was taken that turned out to cost too many steps.

The newly developed algorithm uses these rules as well, but also concerns itself with the number of marked edges. The last situation in rule 3 can add marked edges to the graph, which at first glance might look to influence the new algorithm that is going to use this kernelization. However, this addition of marked edges is called for. One of the two pairs of vertices will definitely have to be changed, but at this point it is impossible to tell which one. So together the pairs of vertices are in a state of possibly both being turned into a real edge or both into a non-edge, for no cost. Since this was also the definition of a fuzzy edge and thus for this research a marked edge, the inclusion of these newly created marked edges in the rest of the algorithm is completely justified.

Implementation of the rules The implementation of this kernelization is quite straightforward. A loop is performed until no more rules can be applied. In this loop clusters are attempted to be removed as per Rule 1, a check is done whether more than k connected components that are not clusters are present, and finally min-cuts larger than k are looked for, and if found handled according to the description in the design. The min-cut is looked for by performing the Edmonds-Karp Algorithm for finding maximum flow. The handling of the min-cuts is done exactly as prescribed by the paper, as mentioned in the design.

After Rule 1, 2 and 3 have been applied exhaustively, if k is still above 0 and no solution has been found branching is executed.

5.4.2 Branching

Design The paper unfortunately does not give a branching algorithm. The best indication it gives is in the line: "*We have thus proved that the (k, r)-WFCE problem has a kernel of size $\mathcal{O}(k^2+r)$, where it is then possible to find a solution just by trying all possible ways of editing its edges*". From this expression the aforementioned $\mathcal{O}((k^2+r)^{2k})$ is deduced. as said before, in this expression the 2 in $2k$ stands for getting the number of edges (real edges, non-edges and marked edges, so every pair of vertices) from the number of vertices, and the k stands for trying every possible combination of the pairs of vertices of total cost k.

Implementation The branching briefly mentioned suggests an implementation where every combination of changing the sets the pairs of vertices are in are tried, as long as the total cost of the changes remains below k. This turns out however to be a very costly branching algorithm. The implementation for this research uses a slight improvement, that can implicitly be deduced from the paper. Instead of looping over all of the pairs of vertices and branching them in the two directions of making them a real edge or a non-edge, we loop over all of the pairs of vertices not in F and branch on those. This results in somewhat of the same behavior: For some branches the algorithm will reach an endpoint because it has run out of cost to spend, for some it will reach an endpoint because all of the pairs of vertices have been branched on, even though not all k is spent. This occurs in branches where along the way many pairs of vertices remain unchanged. This way we now for sure that every possible branch has been visited (only considering the real edges and non-edge), and we know for sure no solution has been skipped, if one is available. This last statement holds because we can make the marked edges either real edges or non-edges for free. When a branch has terminated all the connected components as defined in Section 2 are checked for any non-edges between the vertices in that connected component. If none are present all of the connected components can be turned into clusters by changing the remaining marked edges that are connecting vertices within that connected component to real edges for free, and changing all of the marked edges that are connecting different connected components into non-edges for free. If at least one connected component still contains a non-edge, the branch returns that it has failed. This is because either there is no more k to spend, or this particular branch has already made a decision for all of the edges whether to change or not change them.

This alteration to the branching does not improve the worst case run-time, namely when there are no marked edges, but it greatly improves the run-time in practice, especially when there are lots of marked edges, which otherwise would have all been tried both ways, resulting in a lot of extra branches on average.

As can be imagined trying every pair of vertices that is not a marked edge as a real edge and non-edge still takes a lot of time. For the implementation the design choice has been made to cap the maximum k to be branched on at 2. This is because if it was capped at 3, each branching would take over half an hour. A time limit was placed on the run-time, which will be elaborated upon in Section 6, and with a branching taking over half an hour the algorithm would always return that it has not found a result because of running out of time. The preference went to more often finding a result with the cap of 2 and sometimes not finding a result because the cap was reached over not finding a result at all because the time limit was reached every time. This does not mean that the parameter k that the algorithm started with was capped to 2, only that after the kernelization if the remaining k was still higher than 2, the branching would only try to branch up until a cost of 2.

This recursive implementation checked 270 branches as mentioned in Section 4. However, with larger k and larger graphs this recursion eventually led to stack overflows, since it is possible to recurse $n*n$ times, where n is the graph size. To

keep every k consistent, for larger k for all graph sizes the choice was made to use a different implementation. In this implementation instead of recursing two ways, recursion was only done when a pair of vertices was changed from a real edge to a non-edge, and vice versa. To still look at all the possible variations of changing the graph a loop was implemented. For every branch it looped over all remaining pairs of vertices which had not been looked at yet, and for each it did recursion by changing that pair of vertices from a real edge to a non-edge or the other way, and leaving the other pairs of vertices unchanged. To clarify: When in this loop the pair of vertices $\{i, j\}$ had been branched on and returned, assuming i and j are both smaller than the graph size, the pair of vertices $\{i, j\}$ would be set back to its original state (in addition to the pairs of vertices deeper in the recursion of course), and then the same branching would be done on the pair of vertices $\{i, j+1\}$, thus effectively symbolizing the branch where $\{i, j\}$ would not be changed, but without having to go in recursion. This branching method making use of a loop never encountered stack overflow issues as expected from this implementation. However, since it does use a large loop it severely decreased the speed of the branching algorithm, from 270 branches per second to 20 branches per second, which is why these two values are mentioned in Section 4. But as will be visible from the graphs in Section 7, this did not affect the potency of this algorithm too much, since at lower k it was already starting to perform bad.

5.5 Algorithm 3.

The final algorithm to be discussed is the newly created algorithm for the clustering of marked graphs. It borrows elements from the first two algorithms, at places simply copying existing parts of the algorithms, and together with some additions it weaves them into a new algorithm that performs differently, gives different results, and has different run-times. The algorithm is divided into three distinct phases instead of two like the previous algorithms. There is a preprocessing phase, a kernelization phase, and a branching phase.

5.5.1 Preprocessing

Design To start off there is a preprocessing phase. One of the problems this algorithms aims to solve is the long run-time of the existing algorithm for marked graphs. This is realized by using a heuristic. The algorithm for unmarked graphs is deterministic, but as already mentioned previously, multiple equally good solutions depending on the order in which branching is traversed can be found. The same is even more so true for marked graphs. As an example: when two vertices are each separately in a cluster of size 1, and there is a marked edge between them, they can freely be transformed into one cluster of size 2. This means that even following a perfect mathematically sound algorithm like the second presented algorithm might not lead to the same solution as or one even close to one the unmarked algorithm leads to. For this reason using a heuristic to improve speed is deemed acceptable, as the non-deterministic nature of using

random guesses does not worsen the result more than the previous algorithm does with its uncertainty in the order of traversing the branches. This is also complemented by completing multiple different searches using the heuristics, and by having multiple results, one or more are bound to be closer to the original solution.

Implementation The preprocessing consists of two steps. The first step is to create a number of unmarked graphs from the marked graphs. This is done by assigning the marked edges to be real edges or non-edges, completely at random. Each such unmarked graph is created separately, so the unmarkings happen independent from each other, producing different unmarked graphs. The second step is to run the unmarked algorithm on these unmarked graphs. What this does is give a very broad approximation of a solution. The clusters that are more likely to be present in the solution of the unmarked algorithm on the original unmarked graph, based on the fact that the marking happened at random as well, have a higher chance of appearing in these solutions. So when these approximations are used, on average there should be a result between them approaching the result of the unmarked algorithm on the original unmarked graph. These approximations will then be used later on, in between the kernelization and the branching.

5.5.2 Overarching design

After this preprocessing the second and third phase of the algorithm begin. Because there is uncertainty, multiple solve attempts are being made. Each such solve attempt goes through the kernelization phase. When the kernelization phase is done, if k is still larger than 0, the kernelization has not terminated with the message that no solution can be found, and there are still connected components left that are not clusters, the approximations come into play. The outcome of the kernelization is copied as many times as there are approximations. For every marked edge in the results from the kernelizations that are going to be branched on, a choice is made whether it is a real edge, or a non-edge. This choice depends on their state in the solution of the approximations. If the approximation did not deliver a solution, then the states of the pairs of vertices in the unmarked graph that was used as input for the approximated solution is used as the approximation instead.

All of these changes from marked edges to real and non-edges are, as the name of the graph used for this suggests, approximations. This choice is made so the edges have a cost higher than 0, so the branching terminates much faster. However, it is a good design choice to not have them get the same weight as the other edges, seeing as in reality their weight was lower (namely 0). Thus the weight of every other edge gets inflated by a given number, and k gets inflated by the same number. When branching this ensures that even though the algorithm is bound in time by the number of marked edges that can be branched on, they are cheaper to branch on than the other pairs of vertices.

When the algorithm terminates, the cost of changing the marked edges gets added to k, and k is deflated again, so the final k is as if this inflation never happened. What has to be accounted for though is that if you input a certain parameter k, and the cost would be exactly k, the algorithm will terminate without finding that answer if any marked edge is altered, because that would lead during branching to a value of k of lower than 0. So when running a test with this algorithm, if a certain input parameter k is estimated to be required, it is wise to raise it with a certain value, depending on the inflation and the expected number of marked edges to be changed to account for this. In the implementation this is also covered somewhat by the difference between d, the expected distance from a cluster graph, and input parameter k being equal to $k+2$. After all of the pairs of vertices are given a state of real edge or non-edge after the kernelization, a branching is performed for each such approximation.

5.5.3 Kernelization

But firstly, the kernelization works in the following way. Kernelization is performed as previously for unmarked graphs, as described in the second algorithm. However there is an extra step after finding cluster. When a cluster is found the number of marked edges in it are counted. If there are more marked edges in a cluster than a certain given number, one random marked edge from that cluster is picked, and stored in memory. For this implementation the choice is made to make that number 2. This number is large enough to tear up too large clusters where many fuzzy edges are potentially present. However, if a lot of real edges are present in the same cluster this prevents the forming of a good cluster early on. This improvement is mainly for smaller clusters where it could go either way. In these cases relatively a lot of marked edges could sway the algorithm to make them a cluster. With not too large graph sizes there are a lot of such smaller clusters expected, and as such for these specific experiments the number 2 was chosen. Other choices will be discussed in Section 9. It is important to note here that this number that is preventing certain clusters that really want to form from forming is only holding those clusters back in the kernelization process. During the branching process larger clusters still have a chance to form if their structure really indicates that all of their vertices should be connected. When the kernelization is done, and one or more marked edges have newly been put in storage this run because of too many marked edges being in a cluster, every pair of vertices is reset to the state they were just before the kernelization. Then the marked edges that are in storage are all changed to non-edges, and the kernelization is run again. This storage is not emptied, but compounds with each retry of the kernelization. This forces the number of marked edges to be on average lower in the end result than without this step, without being extremely strict on which marked edges can absolutely not be in clusters. The aim for the algorithm is to push the search for a good cluster graph in the right direction. If it would force the search in a tight corridor, good answers might be missed in some cases, seeing as the marking has occurred randomly. This resetting continues until after the rules have been applied exhaustively no clusters found

during the kernelization contain too many marked edges.

5.5.4 Branching

After the kernelization is done and all of the marked edges are assigned to be real edges or non-edges as mentioned in Subsubsection 5.5.2, a branching is performed for each of the approximations. Since the deliberate action has been taken to unmark the graphs, the branching itself is taken exactly from the algorithm for unmarked graphs, specifically the improved branching algorithm that gives a run-time of $\mathcal{O}(2.27^k)$.

5.5.5 Finding a solution

After the branching is done on all approximations, the best solution is selected. Then finally the results from every attempt are compared. As said these can vary, because of random pairs of vertices being made non-edges during kernelization, and because of the random unmarkings before branching. From these results all of the ones with the lowest steps taken are saved. In all of the tests performed this however turned out to be just one solution, since the steps all led to the same solution, even considering the random unmarking, or led to a worse solution.

6 Experiments

As mentioned in Section 4.1, tests were performed for mainly the input parameters k and graph size s. For each combination of k and s a number of tests were run, which depended on the parameters k. Also depending on this parameter was the number of approximations in the preprocessing of the new algorithm for marked graphs, the number of solve attempts for the kernelization process and the inflation of k and the weights of the pairs of vertices that are not marked edges between the kernelization and the branching in the new algorithm for marked graphs. Finally the maximum number of branches that can be traversed during the branching part before the algorithms return that they have ran out of time were dependent on the input parameter k. These dependencies exist because of longer run-times at larger graph sizes for increasing parameters k. To keep consistency within all runs per parameter k no other factors than s were varied. In the next subsection the specific parameters will be mentioned.

6.1 Parameters

For $k = 4$ and $k = 5$ the following parameters were used:

- Number of tests performed and taken average out of = 10
- Number of approximations = 5
- Number of solve attempts = 5

- Size of inflation between kernelization and branching = 5
- Maximum number of branching steps to be taken by the algorithm for unmarked graphs = 60000
- Maximum number of branching steps to be taken by the existing algorithm for marked graphs = 32400

For $k = 6$, $k = 7$ and $k = 8$ the following parameters were used:

- Number of tests performed and taken average out of = 5
- Number of approximations = 4
- Number of solve attempts = 4
- Size of inflation between kernelization and branching = 3
- Maximum number of branching steps to be taken by the algorithm for unmarked graphs = 30000
- Maximum number of branching steps to be taken by the existing algorithm for marked graphs = 16200

For $k = 6$, $k = 7$ and $k = 8$ some of the parameters were lowered so the algorithms would terminate in less time.

For $k = 9$ the following parameters were used:

- Number of tests performed and taken average out of = 5
- Number of approximations = 4
- Number of solve attempts = 4
- Size of inflation between kernelization and branching = 3
- Maximum number of branching steps to be taken by the algorithm for unmarked graphs = 30000
- Maximum number of branching steps to be taken by the algorithm for unmarked graphs in the run of the new algorithm for unmarked graphs = 3750
- Maximum number of branching steps to be taken by the existing algorithm for marked graphs = 1200

For $k = 9$ the limit of recursion for the existing algorithm for marked graphs was reached, and the switch was made to the new variant using a loop as mentioned in Subsubsection 5.4.2. This is the reason for the steep drop in the last parameter. Furthermore from here on the new algorithm for marked graphs was branching deeper, and the choice was made to lower the time for the new algorithm for marked graphs to be more in line with the other algorithm. Since it performs multiple branchings dependent on the number of approximation and number of solve attempts, the choice was made to give the branching algorithm it borrows from the algorithm for unmarked graphs twice the time as the unmarked graph divided by the number of approximations and the number of solve attempts.

For $k = 10$, $k = 11$, $k = 12$, $k = 13$, $k = 14$ and $k = 15$ the following parameters were used:

- Number of tests performed and taken average out of = 3
- Number of approximations = 3
- Number of solve attempts = 3
- Size of inflation between kernelization and branching = 2
- Maximum number of branching steps to be taken by the algorithm for unmarked graphs = 120000
- Maximum number of branching steps to be taken by the algorithm for unmarked graphs in the run of the new algorithm for unmarked graphs = 26667
- Maximum number of branching steps to be taken by the existing algorithm for marked graphs = 2400

For all of the parameters k mentioned the tests were performed on graph sizes of 9 up until and including 24.

Another less obvious parameter used is the number of pairs of vertices to be marked. This number is between 20% and 30% of all pairs of vertices in any particular graph.

7 Results

From all of these runs a lot of data was gathered, over 4000 data points. From all of this data some will not be discussed, like the average number of marked edges or average number of clusters produced by the algorithm for unmarked graphs, since those number did not behave unexpectedly, they stayed around

their expected values. The results that are of most interest to the problem statement are which of the algorithms for marked graphs performed better according to the quality measure, which of the algorithms for marked graphs performed faster, or phrased differently, was completed more often within the given time limit, and a benchmark given by the new algorithm for marked graphs.

The next two graphs answer the first question:

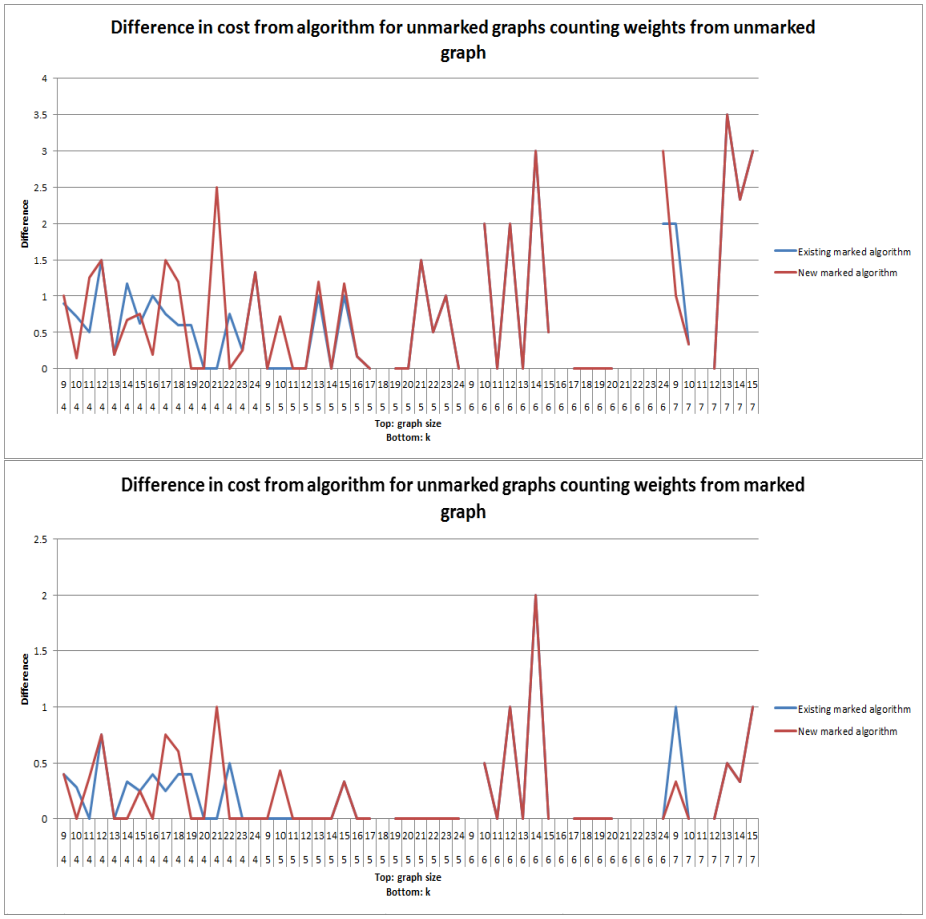

Figure 2: Differences in costs.

These two graphs only focus on $k = 4$, $k = 5$, $k = 6$ and $k = 7$. This is because from $k = 8$ on there are very few results for the existing marked algorithm, and the ones that are present appear to be in line with the findings for the first four values for k.

In these graphs wherever no values could be found the line is simply left out. From these graphs it can be observed that already the existing marked algorithm

is struggling to find solutions. When it does find solutions it is however on par with the existing algorithm. The hypothesis about the quality difference was that the new algorithm for marked graphs would perform only slightly, maybe negligibly better than the existing algorithm for marked graphs on such small parameters.

The results are close to the hypothesis. The new marked algorithm performs only slightly, almost negligibly worse than the existing algorithm for marked graphs. The parameters these can be compared on are even smaller than expected, so it falls well within the confines of the small parameters mentioned in the hypothesis. Unfortunately from these results we can not conclude that the new algorithm performs better in quality than the existing algorithm. What can be concluded from the difference between the first and the second graph is that both algorithms have about an equal decrease in mistakes made when discounting the marked edges, since the second graph looks almost like a down-shifted version of the first graph.

The second question about which algorithm performed better overall can be answered by the following graph:

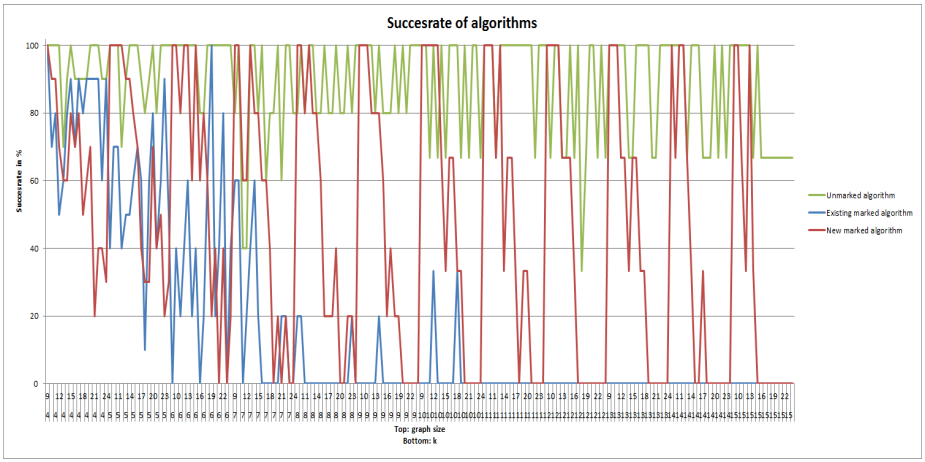

Figure 3: Success rate of the algorithms.

The green line indicates the success rate of the algorithm for unmarked graphs. This algorithm sometimes failed because the solution took more than k steps, and very occasionally since it ran out of time during branching. But otherwise it had a very high success rate, which is not unexpected since that was what the runs were tailored for.

The blue line indicates the success rate of the existing algorithm for marked graphs. This algorithm had a lot of success for smaller k, but as k increased it only had very limited success for small graph sizes while having no success for large graph sizes, and from a certain point it had no success at all anymore.

The red line indicates the success rate of the new algorithm for marked graphs. This algorithm exhibits a very graph size dependent behavior. For every k it performed very well for small graph sizes, even going so far as to occasionally have more success than the unmarked algorithm, but performing very poorly at larger graph sizes.

For low k the existing algorithm for marked graphs finds solutions a little more often for large graphs, but otherwise the new algorithm for marked graphs finds decisively more solutions, and can run for far larger k, thus vastly improving on the usability in practice over the existing marked algorithm.

The failure rates of the algorithms for marked graphs are divided in the category of not finding a solution within k steps, and not finding a solution within the time limit. The following two bar graphs indicate in red that the algorithm failed because it ran out of time, and in blue that it could not find a solution:

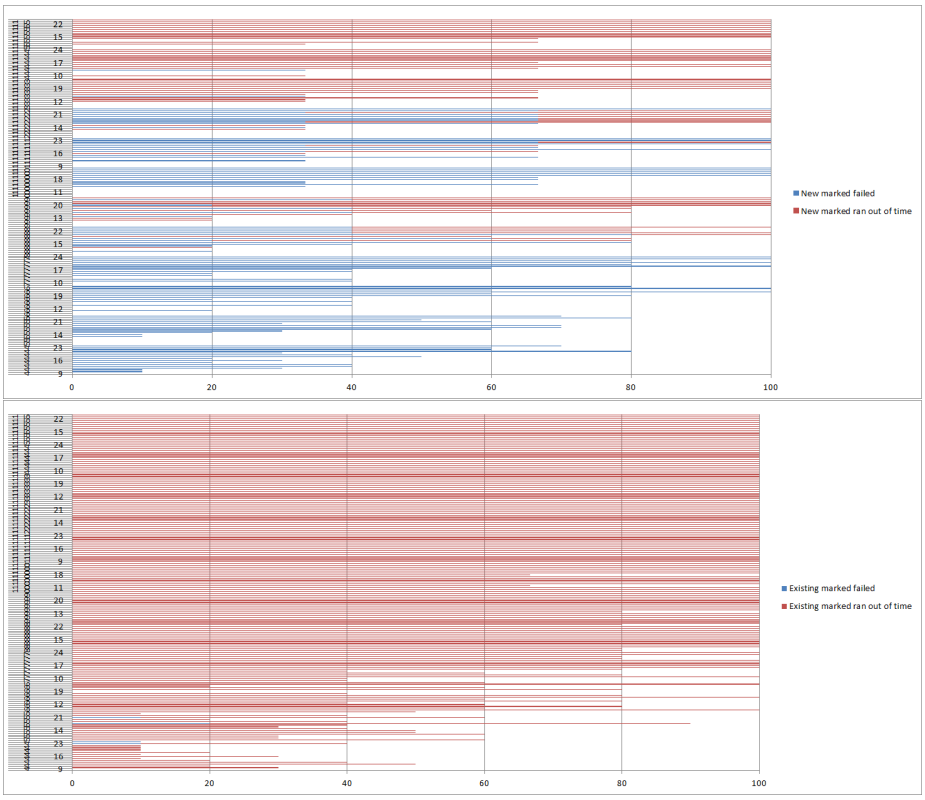

Figure 4: Failure reasons of the algorithms. The top graph is for the existing algorithm for marked graphs. The bottom graphs is for the new algorithm for marked graphs. The y-axis in each of the graphs is increasing k and graph size from bottom to top.

These graphs further clarify that the reason the new algorithm for marked graphs failed for larger graphs was not that it ran out of time, but that it did not find a solution. This can be explained by the fact that larger graphs contain more marked edges. In the inflation step of the algorithm these marked edges get a fraction of the cost of a pair of vertices to change. But these fractions can add up, and eventually prevent a solution from being found. The solution to this would be to estimate the number of marked edges, and thus roughly estimate the number of extra steps needed to create a cluster graph because of the weights of the marked edges, and add these extra steps (divided by the inflation factor) to the parameter k before running the graph. This would slow down the branching process somewhat, so this is a consideration that would have to be made when running the algorithm on an unknown problem.

The existing algorithm for marked graphs however clearly fails many times because of its inability to find solutions within the given time limit, since it runs so slow and requires so many branching steps.

Finally a benchmark is created for algorithms for marked graphs. The next graph shows how far currently the found solutions are on average from what the deterministic algorithm for unmarked graphs found. It shows in blue what the real distance is, and in red what the distance is when not considering incorrectly set marked edges to be mistakes:

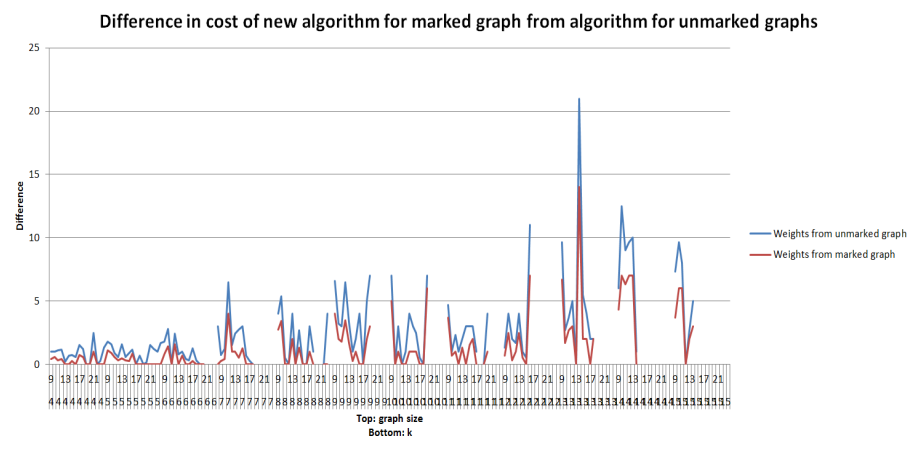

Figure 5: A benchmark for marked graphs.

Except for the spike at $k = 13$ and a graph size of 14, this benchmark indicates that quality of the result is as can be expected dependent on k, but the distance from the solution of the algorithm for marked graphs should not increase too rampantly.

8 Conclusion

This research has aimed to explore whether using circumstantial information, in this case the notion of uniform distribution of missing information, can improve an algorithm for clustering marked graphs. Furthermore it has sought to improve on the practical run-time of the currently existing algorithm for clustering marked graphs. Finally it has sought to create a benchmark for quality for marked graph clustering problems that currently can not be analyzed.

By analyzing the design and implementation of the currently existing algorithms for unmarked and marked graphs their strengths and shortcomings have become apparent. The strengths were used in creating a new design and implementation of an algorithm for clustering marked graphs, while the shortcomings were tried to be avoided.

For the tested parameters the new algorithm does not outperform the currently existing algorithm in means of quality as defined in this paper, but also does not under perform in comparison. Since the new algorithm uses a heuristic this indicates that an improvement on the currently existing algorithm is definitely not out of the question, but this research has not yielded one.

The practical run-time was clearly shown to be reduced in Section 7, since many more graphs were clustered within the time bounds by the new algorithm for marked graphs than by the existing algorithm for marked graphs. Clearly using approximations as a heuristic greatly speeds up a search.

Finally a new benchmark was created for clustering marked graphs, which surely can be improved upon. This benchmark will serve as a sanity check for future development in this area, giving future research a goal to outperform.

9 Further research

Some theoretical and some practical optimizations were not implemented in this research. Future research in this area can definitely benefit from analyzing and implementing these. The main optimizations are:

- The branching strategy for the algorithm discussed in Bodlaender et al. [2] could be further optimized. More research can definitely be done using that algorithm, and better implementations than recursion over all branches or looping as discussed in this paper must be possible.
- A different heuristic could be used for the newly developed algorithm. This heuristic has a relatively long preprocessing, since the unmarked algorithm has to be run multiple times.
- The inflation used in the newly developed algorithm was relatively small. This value is a trade-of between speed, quality, and solvability. Higher inflation values could be experimented with. They would make the marked edges relatively cheaper. This would slow the process down, but perhaps keep less mistakes introduced during the heuristic.

The maximum number of marked edges per cluster during kernelization is capped at 2 in this research. Different values for this cap could be explored, and even a dynamic cap, depending on either the number of marked edges, k, graph size, cluster size, or a combination of those factors could be explored. This cap could make a big difference in larger graph sizes.

Furthermore there is research to be done on how the number of marked edges affects the quality of the results in general.

There might be applications where there are different kind of marked edges, or where marked edges can also be real edges or non-edges. These situations are still left to be explored.

Finally a very interesting technique that was cut from the implementation for this research was the automated branch generation algorithm described in Gramm et al. [10]. This automated branching could further speed up the process when using the heuristic as described, since the automated branch generation would greatly benefit the unmarked branching algorithm that was used in the newly developed marked algorithm.

References

[1] A. Ben-Dor, R. Shamir, Z. Yakhini. *Clustering Gene Expression Patterns.* Proceedings of the Third Annual International Conference on Computational Molecular Biology, pp. 33-42, 1999.

[2] H. L. Bodlaender, M. R. Fellow, P. Heggernes, F. Mancini, C. Papadopoulos, F. Rosamond. *Clustering with partial information.* Theoretical Computer Science, Volume 411, pp. Issues 7-9, 1202-1211, 2009.

[3] S. Böcker, S. Briesemeister, Q. B. A. Bui, A. Truss. *Going weighted: Parameterized algorithms for cluster editing.* Theoretical Computer Science, Volume 410, Issue 52, pp. 5467-5480, 2009.

[4] M. Charikar, V. Guruswami, A. Wirth. *Clustering with qualitative information.* Journal of Computer and System Sciences, Volume 71, Issue 3, pp. 360-393, 2005.

[5] J. Chen, J. Meng. *A 2k kernel for the cluster editing problem.* Journal of Computer and System Sciences, Volume 78, Issue 1, pp. 211-220, 2011.

[6] A. Condon, R. M. Karp. *Algorithms for Graph Partitioning on the Planted Partition Model.* Proceedings of the Third International Workshop on Randomization and Approximation Techniques in Computer Science, and Second International Workshop on Approximation Algorithms for Combinatorial Optimization Problems. Lecture Notes in Computer Science, vol 1671, pp. 221-232, 1999.

[7] F. Dehne, M. A. Langston, X. Luo, S. Pitre, P. Shaw, Y. Zhang. *The Cluster Editing Problem: Implementations and Experiments.* Proceedings of the

Second International Workshop on Parameterized and Exact Computation. Lecture Notes in Computer Science, Volume 4169, pp. 13-24, 2006.

[8] D. Emanuel, A. Fiat. *Correlation Clustering - Minimizing Disagreements on Arbitrary Weighted Graphs*. Proceedings of the 11th Annual European Symposium on Algorithms. Lecture Notes in Computer Science, Volume 2832, pp. 208-220, 2003.

[9] P. Erdős, A Rényi. *On the Evolution of Random Graphs*. Publication of the Mathematical Institute of the Hungarian Academy of Sciences, pp. 17-61, 1960.

[10] J. Gramm, J. Guo, F. Hüffner, R. Niedermeier. *Automated Generation of Search Tree Algorithms for Hard Graph Modification Problems*. Algorithmica, Volume 39, Issue 4, pp. 321-347, 2004.

[11] J. Gramm, J. Guo, F. Hüffner, R. Niedermeier. *Graph-Modeled Data Clustering: Fixed-Parameter Algorithms for Clique Generation*. Theory of Computing Systems, Volume 38, pp. 373-392, 2005.

[12] S. E. Schaeffer. *Graph clustering*. Computer Science Review, 1, pp. 27-64, 2007.