Eötvös Loránd University

Faculty of Science

Low-Stretch Spanning Trees

Pál András Papp

BSc thesis

Advisors:

Sándor Kisfaludi-Bak

Zoltán Király

Department of Computer Science

Budapest, May 2014

Contents

1 Introduction **3**

2 Motivation and background **5**
2.1 Related work 5
2.2 Aim of the thesis 6
2.3 Preliminaries and notation 8

3 The star decomposition algorithm **8**
3.1 The paper of Elkin *et al.* 9
3.1.1 Concepts and notions 10
3.1.2 Methods of the algorithm 12
3.1.3 Theorems and claims 14
3.1.4 Improving the stretch 16
3.2 Incorrect statements in the article 16
3.2.1 Inequality on the radius of cones 17
3.2.2 Incorrect star decomposition 17
3.3 Introducing different cones 19
3.3.1 Modified cones 20
3.3.2 Weak Concentric Systems 20
3.3.3 The propositions with the new cones 23
3.3.4 The main statements of Elkin *et al.*'s paper 24

4 Further algorithms **25**
4.1 The decomposition of Alon *et al.* 26
4.1.1 The algorithm 27
4.1.2 Claims of the article 28
4.2 Star decomposition with queues 29
4.2.1 Concept 30
4.2.2 Introducing queues 31
4.2.3 Claims about the algorithm 33

5 Empiric results **34**
5.1 Evaluation framework 35
5.1.1 The LEMON library 35
5.1.2 Graph models examined 36
5.2 Different versions of star decomposition 37
5.3 Comparing the algorithms 39
5.3.1 Average edge stretch 39
5.3.2 Average path stretch 43
5.4 Sublinear bounds for the Kruskal algorithm 46

6 Conclusion **48**

1 Introduction

Finding short paths in graphs and networks is a topic with countless different applications in various fields of Computer Science and Mathematics. Although at first sight the topic seems to cover a small group of homogenous problems only, problems of this kind can in fact be very different in the particular application areas. While some of them need the exact shortest path or all shortest paths to be computed from only a certain vertex to another, others may require a not necessarily shortest, but sufficiently short path between all pairs of nodes, in reasonable time. Whereas the previous example is a well-known and thoroughly studied problem, the latter (and the entire area of approximately shortest paths) usually received much less attention.

The topic of spanning trees is also one of the most important areas of network design. Although a spanning tree in a weighted graph can be developed according to many different objectives, the minimum weight spanning tree is perhaps the only widely known way to do this. However, many application areas require spanning trees that are designed in a different way, according to other aims than minimizing the sum of their weights.

When creating low-stretch spanning trees, our goal is to develop a spanning tree in the graph for which the paths in the tree between the vertices are as short as possible, combining the previous two topics. That is, in our area of application we need a spanning tree in the graph, but we want to select a tree such that when disposing the non-tree edges, the distance of vertices in the graph stays relatively small. Although this thesis mainly focuses on approaches trying to minimize the average length of paths in the developed spanning tree, other algorithms also exist that instead aim to keep the length of the longest path in the resulting tree as small as possible [5].

When evaluating a path between two nodes of the graph in a spanning tree, it is a natural idea to compare this path to the distance of the vertices in the original graph. The ratio of the distance in the spanning tree and the distance in the original graph is known as the *stretch* of the path in the spanning tree. Therefore, the stretch of a path shows how much the shortest path between the nodes is expanded, "stretched", if instead of the entire graph, only the edges of the spanning tree are available. The task is to keep the average value of stretch in the spanning tree as low as possible.

When defining the notion of stretch, there are two slightly different possibilities available; in the thesis, these two versions of stretch are referred to as *edge stretch* and *path stretch*. Given a graph $G(V, E)$ with a spanning tree T, edge stretch is defined for the edges of G; for an edge $(u, v) \in E$, the stretch of the edge is

$$\text{E-stretch}_T(u, v) = \frac{d_T(u, v)}{d_G(u, v)},$$

where $d_G(u, v)$ denotes the distance of vertices u and v in the graph G. Path stretch, on the other hand, is defined with the same formula (the ratio of tree distance to original graph distance), but for every pair of vertices in the graph instead of only the endpoints of edges. Path stretch between the vertices $u, v \in V$ is denoted by $\text{P-stretch}_T(u, v)$.

With these definitions, it is straightforward to express the amount in which a spanning tree expands shortest paths in a graph, by defining the average edge stretch and average

path stretch of a spanning tree in a graph as

$$\text{ave-E-stretch}(G)_T = \frac{1}{|E|} \cdot \sum_{(u,v)\in E} \text{E-stretch}_T(u,v)$$

and

$$\text{ave-P-stretch}(G)_T = \frac{1}{\binom{|V|}{2}} \cdot \sum_{u,v\in V} \text{P-stretch}_T(u,v).$$

Although most statements in the literature are related to the average edge stretch, in the empirical experiments of the thesis, average path stretch values are also investigated.

Spanning trees of low stretch value have proven useful in a number of different areas. One of the most important of these is the solving of symmetric diagonally dominant linear systems, as pointed out by Boman and Hendrickson [6]. Applying the spanning tree found in [1], the authors designed a solver with a running time of

$$m^{\frac{3}{2}} \cdot 2^{O(\sqrt{(\log n \log\log n)})} \cdot \log\left(\frac{1}{\varepsilon}\right),$$

where ε is a parameter that determines the precision of the solution. This result was later improved by Spielman and Teng into a more powerful solver [7], that has already been used to obtain eigenvalues or to approximate maximum flows [8], among many other applications.

One of the first applications of the topic was suggested by Alon *et al.* [1], in connection with a two-player zero-sum game played on a graph. This game was developed to gain deeper understanding of the k-server problem. In this problem, there are k available servers, each of them located at a point in a given metric space M at every point in time in the process, and there is a series of requests at specified points in M. To process a request, one of the servers has to be moved to the position of the request; the task is to process all request while minimizing the sum of distances traveled by the servers in the process. Through the graph-theoretic game of Alon *et al.*, low-stretch trees can be used to devise a randomized on-line algorithm for this problem.

Another application of the topic is the minimum communication cost spanning tree (MCT) problem. In this task, we need to select a spanning tree in a network minimizing the total cost of transmitting a given set of required signals between the nodes. Formally, given a weighted graph $G(V,E)$ and a matrix A such that both the columns and the rows of A correspond to the vertices of G, our aim is to find the spanning tree that minimizes

$$\sum_{x,y} a_{x,y} \cdot d_T(x,y),$$

where $a_{x,y}$ denotes the entry of A for some $x, y \in V$. Since even very special cases of this problem are shown to be NP-hard, it is important to design strong approximation algorithms [9]. One such algorithm was devised by Peleg and Reshef [10], using the low-stretch spanning trees of [1]. Since the lower the stretch value of the spanning trees found, the better approximation their algorithm can give to the MCT problem, finding spanning trees with the lowest possible stretch value is also an important question in this area.

2 Motivation and background

2.1 Related work

There are numerous papers and studies presenting results on low-stretch spanning trees, and in other subjects that are closely related to the topic. This subsection contains a brief overview of these results, while the most important algorithms and statements will later be presented in detail.

The first notable and probably the most cited result in the area is that of Alon *et al* [1]. In their paper, the authors proved that the lower bound of average edge stretch that can be guaranteed by an algorithm is $\Omega(\log n)$, and also presented an algorithm that returns a spanning tree with an average edge stretch of $O(e^{(\sqrt{\log n \cdot \log\log n})})$, which is discussed in detail in Section 4.1.

Another basic result in the topic is the star decomposition algorithm of Emek et al [2], which is also the most thoroughly discussed algorithm of this paper, described in Section 3. This algorithm divides the graph into smaller subgraphs (connected by some edges), then recursively calls itself on these 'components', assembling the final spanning tree from the spanning trees found in the 'components' and the connecting edges. The method finds a spanning tree with an average edge stretch of $O(\log^3 n)$, and an improved version with an average stretch of $O(\log^2 n \cdot \log\log n)$ is also presented. The required running time is $O(m \log n + n \log^2 n)$, or only $O(m \log n)$ if the graph is unweighted.

Many of the further algorithms are based on the star decomposition algorithm in some way. The algorithm of Abraham *et al* [3], described in Section 4.2, adds only a slight modification to the previous algorithm, yielding an average edge stretch of $O(\log n \cdot (\log\log n)^3)$. However, the findings of this paper will also play a crucial part when discussing the original algorithm of Emek *et al.* Another similar, more sophisticated and more complex algorithm is that of Abraham and Neiman [4], who use a different decomposition method called "Petal-decomposition", with which a spanning tree of $O(\log n \cdot \log\log n)$ average stretch is constructed in $O(m \log n \cdot \log\log n)$ time.

Many articles discuss topics that are closely related to finding spanning trees with low average edge stretch. Some papers have also studied a distribution of spanning trees in a graph instead of one single tree; statements about such distributions usually correspond to spanning trees developed by a randomized algorithm. Alon *et al.* [1] also show a distribution of trees, for which the expected stretch on the edges of the graph is $e^{O(\sqrt{\log n \cdot \log\log n})}$. A randomized version of the algorithm of Abraham *et al* [3] is even shown to bound the expected value of the stretch of an edge by $\widetilde{O}(\log n)$. Also, in the case of a relaxed problem, when new nodes can be added to the graph if distances between the original nodes are not reduced, Fakcharoenphol *et al.* [11] showed that a stretch of $O(\log n)$ can be achieved.

While the previous papers aim to minimize average stretch, some authors developed algorithms with the purpose of obtaining a low value for maximum stretch. It is also a common approach to measure the stretch of a spanning tree in maximum stretch, defined naturally as

$$\text{max-E-stretch}(E)_T = \max_{(u,v)\in E} \text{E-stretch}_T(u,v)$$

and

$$\text{max-P-stretch}(G)_T = \max_{u,v \in V} \text{P-stretch}_T(u, v).$$

It has been shown that for a fixed $t > 1$, determining if a weighted graph has a spanning tree with a maximum stretch value below t is NP-complete [12]. As the most important practical results of this approach, Emek and Peleg [13] presented an algorithm for finding a spanning tree in unweighted graphs, the maximum path stretch of which is at most $O(\log n)$ times the optimum.

A closely related area is the analysis of sparse spanners in graphs. A spanner in a connected graph is by definition a connected subgraph that spans the graph. Spanners also have practical applications in a wide variety of areas, ranging from distributed systems through communication networks to genetics and robotics [14]. The previous definitions of stretch are also applicable in this case, with the numerator switched to the distance in the spanner instead of the distance in the spanning tree. In fact, the task of finding spanners is a tradeoff between the number of edges to be included and the stretch value obtained. It should also be noted that for any graph $G(V, E)$, integers $m \geq |V| - 1$ and $t \geq 3$, the problem of deciding if G has a spanner of maximum path stretch t and at most m edges is shown to be NP-complete [15].

One of the most basic results on spanners is connected to Althofer *et al.* [14]. While previous research mostly focused on special (unweighted, Euclidean, etc.) graphs, these authors were working with graphs of arbitrary positive weight. They showed that for any positive constant $t > 1$, a spanner with a maximum edge stretch of $2t + 1$ can be found, which has at most $n^{1+\frac{1}{t}}$ edges and the sum of the weights on these edges is at most $(1 + \frac{n}{2t})$ times the weight of the minimum spanning tree in the graph. The paper also presents some special results for planar and Euclidean graphs. Finally, the paper of Dor *et al.*[16] has to be mentioned, which presents a different algorithm that is known for finding a spanner of maximum path stretch 3 with at most $\widetilde{O}(n^{\frac{3}{2}})$ edges included.

Although not closely related to stretch, it is also important to mention the transformation algorithm of Alon *et al*, also presented in [1], since some of the papers discussing the topic apply this method. This paper describes a linear time transformation algorithm, which receives an arbitrary input multigraph $G(V, E)$, and returns a multigraph with the same set of vertices for which the number of edges is at most $|V| \cdot (|V| + 1)$. The property of this algorithm that makes it useful is that it guarantees that for every spanning tree in the resulting graph, the average edge stretch is at most twice as much as it was for the same tree in the original graph. By applying this transformation before an algorithm that computes a low-stretch spanning tree, the magnitude of the upper bounds proven for the stretch of the method remain unchanged, while the number of edges can be bounded by a function of the number of vertices. This might be required in the case of some multigraphs, because the number of edges in a multigraph can be arbitrarily large. This transformation can be useful in the case when the running time or returned stretch of an algorithm is a function of the number of edges of the input graph.

2.2 Aim of the thesis

The main objective of the thesis is to present a detailed survey about the algorithms that can be applied to find a spanning tree with a low average edge stretch value, and to

discuss the effectiveness of these methods both in theory and in practical use. The thesis focuses on three different algorithms available in the literature for developing spanning trees. These algorithms are all described in detail, and the statements and claims about them are also examined, although proofs of these claims from the original articles are not included.

When thoroughly examining one of these methods, the star decomposition algorithm, I realized that there is a mistake in the proof of one of the small propositions of the article, the claim of which is (indirectly) used in the main statements of the paper. Through this proposition, it is possible to create counterexamples for which the main statements of the article do not hold; therefore, it is a very important question whether it is possible to modify the star decomposition algorithm or some of the claims and proofs of the paper in a way that the stated bound on the average stretch will hold indeed. The thesis also presents a series of alterations in the algorithm, and shows that by slightly modifying some of the required lemmas, it can be proven that the original bound on average stretch does hold for this new version of the method, and so the idea of star decompositions is not in fact a failed concept to develop low-stretch spanning trees.

Also, whereas for each of the examined algorithms, a theoretic upper bound on the returned stretch is proven, not much is known about empiric results achieved by these methods. In the papers, no information is available about the effectiveness of the algorithms when applied in practice on input graphs of different kind and size, or when compared to one another. Collecting such empiric results about the methods is of great importance; although the proven upper bound for some algorithm A may be of larger magnitude than that of another algorithm B, it is still possible that in fact A usually yields much better stretch values in practice than B.

Therefore, I also implemented the examined algorithms and run them on a variety of different graphs, drawing conclusions about the behavior of the methods in practice. Besides the methods developed especially to find low-stretch spanning trees, a standard minimum spanning tree algorithm (the Kruskal algorithm) is also included in the comparison; while it has not been created for this purpose, it is still possible that it yields better results in practice than some of the previous methods.

Thus, the main objectives of the thesis are the following:

- to give an overview of the most important algorithms for finding spanning trees with low average edge stretch, with a detailed discussion of the way they operate and the related statements
- to analyze and provide counterexamples to the incorrect claims about the star decomposition algorithm, and by modifying the method and the proofs of the claims, to develop a version of the algorithm for which the statements of the article hold
- to compare the different stretch values achieved by the algorithms on a range of different graphs, and hence determine which of the methods is most suitable for practical use

The next subsection introduces the definitions and notations used throughout the thesis. Section 3 presents and analyses the star decomposition algorithm of Emek et al. Section 4 discusses two other algorithms of the literature that find spanning trees of low

stretch. Section 5 presents the empiric results of comparing these algorithms. Finally, Section 6 concludes the paper.

2.3 Preliminaries and notation

Throughout the thesis, the graphs $G = (V, E, \ell)$ are connected undirected weighted graphs, where $\ell : E \to \mathbb{R}^+$ is a length function. The number of nodes will usually be denoted by n, and the number of edges by m. Although the claims are stated for graphs, they are also true for multigraphs.

For two vertices u and v, the distance between u and v, denoted $\mathrm{d}(u, v)$ or $\mathrm{dist}(u, v)$ in some of the examined papers, is the length of the shortest path between u and v. If we want to emphasize that the path is in the graph G, we write $\mathrm{d}_G(u, v)$. For a subset $X \subseteq V$ of the vertices, $\mathrm{d}_X(u, v)$ means the distance of u and v in the graph induced by X (therefore, $\mathrm{d}_V(u, v)$ is another notation for $\mathrm{d}_G(u, v)$). The $\mathrm{d}_G(u, X)$ distance of a vertex u and a subset $X \subseteq V$ denotes the minimum of the $\mathrm{d}_G(u, x)$ values for all $x \in X$.

For a subset X of vertices, $G[X]$ denotes the graph induced by these vertices, and $E[X]$ denotes the set of edges with both of their endpoints in X. The boundary of X, written $\partial(X)$, means the set of edges with exactly one endpoint in X. The volume of X, or $\mathrm{vol}(X)$ is the number of edges with at least one endpoint in X. A subset of the vertices X that induce a connected subgraph is also referred to sometimes as a cluster. For a set of edges F, $|F|$ (named the volume of F in one of the examined papers) is simply the number of edges in the set.

A partition of V is a set of pairwise disjoint subsets $\{V_1, V_2, \ldots, V_n\}$ of the vertices such that their union is V. The subsets are referred to as the components of the partition. The boundary of a partition means the set of edges with endpoints in different components, denoted by $\partial(V_1, V_2, \ldots, V_n)$.

Let $v \in V$. The radius of the graph from v, written $rad_G v$, denotes the value of $\max_{x \in V} \mathrm{d}(v, x)$. For a subset of vertices X, the notation $rad_X v$ means $\max_{x \in X} \mathrm{d}(v, x)$. The ball of radius r around v is the set of those vertices $x \in V$ for which $\mathrm{d}(v, x) \leq r$, denoted $B(v, r)$. The ball shell of radius r around v, written $BS(v, r)$, is the set of vertices s in $V - B(v, r)$ that have a neighbor w in $B(v, r)$ such that $\mathrm{d}(v, w) + \ell(w, s) = \mathrm{d}(v, s)$.

Numbered theorems or lemmas of examined articles are referred to and cited with their original number in the source paper.

3 The star decomposition algorithm

While the thesis investigates a number of different algorithmic results for finding a low-stretch spanning tree in a given graph, it primarily focuses on the star decomposition algorithm introduced by Michael Elkin, Yuval Emek, Daniel A. Spielman and Shang Hua-Teng [2]. This section discusses their paper, "Lower-Stretch Spanning Trees", in which their algorithm is presented. Note that there are multiple versions of this paper available online, published between 2005 and 2008, with significant differences. We chose

the version discussed here because it is the latest and it has fixed some previous errors, and it is also the version linked from the homepage of the authors.

The paper presents a recursive algorithm named star decomposition algorithm, which is used to find a spanning tree in the graph with an average edge stretch value of $O(\log^3 n)$. A small improvement to the algorithm is also discussed, that yields an average stretch of $O(\log^2 n \cdot \log\log n)$. However, the paper uses a slightly different definition for edge stretch than the usual one; for an edge (u, v) in the graph, previous literature usually defines the stretch of the edge as $d_T(u, v)/d_G(u, v)$, while this paper considers the stretch to be $d_T(u, v)/\ell(u, v)$ (that is, instead of dividing the tree distance of the endpoints by the original distance, it divides it by the length of the edge). This can result in significant differences from some classes of graphs; even the article notes that for some of the application areas, it is necessary to use the original definition of stretch. Nevertheless, the algorithm presented in the paper can be modified to work with the other definition, but this requires a preprocessing stage that calculates the $d_G(u, v)$ distance for every edge (u, v), that may dominate the running time of the method.

Besides the previously introduced definitions, the article defines the cost of an edge, being the reciprocal of its length. For an edge set $F \subseteq E$, let

$$\text{cost}(F) = \sum_{f \in F} \text{cost}(f).$$

We also take the following notations from the paper: $\widetilde{n}$ will be used to denote the number of nodes of the original graph (the input of the first call of the algorithm on the highest recursion level), while n will be the number of nodes of the input graph in the present recursion level. Similarly, $\widetilde{m}$ will mean the number of edges of the original graph on the first recursive call, and m will denote the number of edges in the graph at the present level of recursion.

3.1 The paper of Elkin *et al.*

For a connected graph G, a star decomposition is a partitioning of its vertex set V into vertex subsets $\{V_0, V_1, ..., V_k\}$ such that the subgraph induced by each V_i $(i \in 0, 1, ..., k)$ is connected, and for each $i \geq 1$, there exists a vertex x_i in the set V_i that is connected by an edge to a vertex y_i in the set V_0. In other words, a star decomposition means partitioning the set of vertices into connected components in such a way that there is a selected component to which all other components are connected by at least one edge. V_0 is sometimes referred to as the central component, the node x_i is often called the anchor vertex in V_i, y_i is called the portal to component V_i, and the edge (x_i, y_i) is known as the bridge to V_i. An illustration of a star decomposition can be seen on *Figure 1.*

The star decompositions in the paper also have a selected vertex x_0, named the central vertex. This x_0 can be an arbitrary vertex of G; whichever vertex is chosen, the star decomposition created in the article guarantees to contain this vertex in the central component. In fact, it is also contained in the definition of star decomposition in the paper that the vertex x_0 is in component V_0. The main idea of the article is to recursively apply a star decomposition algorithm to construct a spanning tree of the input graph G. First, a star decomposition of the input graph is created with a method designed for this purpose.

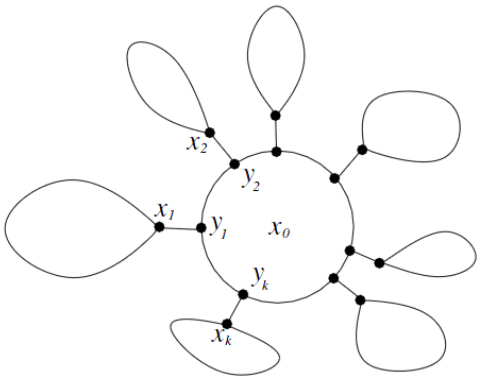

Figure 1: *An illustration of a star decomposition (from [2])*

Then the algorithm is recursively called on all the components of the decomposition, returning a spanning tree of each component of the star. Finally, these spanning trees are connected with the bridge edges of the decomposition; the edges of the spanning trees and the bridges will together form a spanning tree in the original graph. The main statement of the article is that if certain conditions hold for the star decompositions applied, then a number of statements can be proven on the properties on the spanning trees returned.

3.1.1 Concepts and notions

The star decomposition algorithm of the paper finds a star partition of given properties to ensure that the radius and stretch of the resulting spanning tree will be sufficiently small. These properties are collected in the notion of (δ, ε)-star-decomposition. Let δ and ε be real numbers between 0 and $\frac{1}{2}$; then, cited from the article, the definition of (δ, ε)-star-decomposition is as follows:

Definition ((δ, ε)-star-decomposition from [2]). *Let $r = rad_G(x_0)$, and $r_i = rad_{V_i}(x_i)$ for each $0 \leq i \leq k$. For $\delta, \varepsilon \leq \frac{1}{2}$, a star-decomposition $\{V_0, V_1, ..., V_k\}$ is a (δ, ε)-star-decomposition if*

a) $\delta \cdot r \leq r_0 \leq (1 - \delta) \cdot r$, and

b) $d(x_0, x_i) + r_i \leq (1 + \varepsilon) \cdot r$, for each $i \geq 1$.

The algorithm of the paper is claimed to return a (δ, ε)-star-decomposition in each case, and uses its properties to prove statements about the spanning tree.

To create a star decomposition, the algorithm first selects the vertices to sort into the central component, which will be a ball of certain radius around the central node. Thus, as a first step, a radius value is calculated, and all vertices that are in at most that much distance from x_0 are put into the central component. After this, the remaining components are formed one after another. In each step, a vertex x_i is chosen that is not yet sorted into any of the previous components, and is connected by an edge to the central component, and therefore it can be the anchor vertex of the next component. Then, a connected subset of the yet unsorted vertices is chosen, which contains the previously

selected anchor vertex; this subset becomes the next component of the star decomposition. This procedure is continued until all vertices are sorted into one of the components.

One of the key steps in this process is the method of choosing the set of nodes (a connected neighborhood of x_i) to be sorted into the next component. This method has to ensure that all nodes will belong to one of the components, and all the components can really be formed in a connected way (so in the process, every time after a component is created, each connected component of the remaining points contains at least one possible candidate to be an anchor vertex). It also has to make sure that the radius of the formed components are kept under some limits, and hence the inequalities required for the (δ, ε)-star-decomposition property hold. In order to do this, the notion of cones are introduced in the article.

The definition of cone supposes that in the graph in which we want to develop a cone, there is a selected set of vertices S. While creating the star decomposition, the graph will be the induced subgraph on that points that are not yet assigned to any component, and the set S will be the possible anchor vertex candidates (that is, the points in the ball shell of V_0 that are not yet assigned to any component). With this, forward edges, ideals and finally, cones can be defined; the definitions are cited from the paper.

Definition 4.4 of [2] (Ideals and Cones). *For any weighted graph $G = (V, E, \ell)$ and $S \subseteq V$, the set of forward edges induced by S is*

$$F(S) = \{(u \to v) : (u, v) \in E,\ \mathrm{dist}(u, S) + \ell(u, v) = \mathrm{dist}(v, S)\}$$

For a vertex $v \in V$, the ideal of v induced by S, denoted $I_S(v)$, is the set of vertices reachable from v by directed edges in $F(S)$, including v itself.

For a vertex $v \in V$, the cone of width l around v induced by S, denoted $C_S(v, l)$, is the set of vertices in V that can be reached from v by a path, the sum of the lengths of whose edges e that do not belong to $F(S)$ is at most l. Clearly, $C_S(v, 0) = I_S(v)$ for all $v \in V$.

That is, forward edges are the directed version of those edges in the graph that are on a shortest path from S to some node. The ideal of a node v is the set of points that have a shortest path from S which contains v. Finally, a cone of size l around v is the set of points that have a path from v with of length at most l, if forward edge lengths are not considered. When creating a star decomposition, all components except for the central component will be formed as cones. The set S for these sets will be the ball shell of V_0, and the component will be selected as $C_S(v, l)$ in the graph induced by unsorted vertices, for a certain radius l, around some vertex v in the ball shell S of V_0.

Finally one more definition is required to present the algorithms and outline the proofs of the claims in the article. This definition is also cited from the paper.

Definition 4.1 of [2] (Concentric System). *A concentric system in a weighted graph $G = (V, E, \ell)$ is a family of vertex sets $L = \{L_r \subseteq V : r \in \mathbb{R}^+ \cup \{0\}\}$ such that*

a) $L_0 \neq \emptyset$,

b) $L_r \subseteq L_{r'}$ for all $r < r'$, and

c) if a vertex $u \in L_r$ and (u, v) is an edge in E then $v \in L_{r+d(u,v)}$.

It is easy to see that the set of balls $\{B(v,r)\}$ form a concentric system for any vertex v of a graph. It can also be shown that for any vertex v and vertex subset S of the graph, the system of cones $C_S(v,l)$ is also a concentric system. In the paper, a claim about a method of cutting concentric systems is proven, and then applied both for families of balls and cones.

3.1.2 Methods of the algorithm

As mentioned before, the algorithm to find a spanning tree is a recursive method that partitions the graph into a star decomposition, then calls itself on each of the components of the decomposition to find spanning trees in these parts, and connects these spanning trees with the bridge edges to form a spanning tree in the original graph. The pseudocode for this method (LowStretchTree) of the paper is shown below.

T =LowStretchTree$(G = (V, E, \ell), x_0)$.

0. If $|V| \leq 2$, return G. (If G contains multiple edges, return the shortest copy.)
1. Set $\varrho = rad_G(x_0)$.
2. Let $\widetilde{G} = (\widetilde{V}, \widetilde{E})$ be the graph obtained by contracting all edges in G of length less than $\beta \cdot \varrho/\widetilde{n}$.
3. $(\{\widetilde{V}_0, ..., \widetilde{V}_k\})$ =StarDecomp$(\widetilde{G}, x_0, \frac{1}{3}, \beta)$.
4. For each i, let V_i be the preimage under the contraction of step 2 of vertices in $\widetilde{V}_i$, and $(x_i, y_i) \in V_0 \times V_i$ be the edge of shortest length for which x_i is a preimage of $\widetilde{x}_i$ and y_i is a preimage of $\widetilde{y}_i$.
5. For $0 \leq i \leq k$, set T_i =LowStretchTree$(G = G[V_i], x_i)$.
6. Set $T = \bigcup_i T_i \cup \bigcup_i (y_i, x_i)$.

As mentioned before, $\widetilde{n}$ denotes the number of nodes of the original graph (the input of the first call of the algorithm on the highest recursion level), and β is a fixed constant throughout the execution of the algorithm, set to the value

$$\beta = \frac{1}{2 \cdot \log_{\frac{4}{3}}(\widetilde{n} + 32)}.$$

The only part of the method not yet mentioned is the step of contracting vertices. In each recursive call the smallest edges are contracted; their two endpoint vertices are united in one vertex, and the edge set of this vertex will be the union of edges adjacent to any of the previous vertices. Any occurring loops are removed. The fourth step defines the way bridges edges are chosen if any of their endpoints is a contracted node. The StarDecomp method called by LowStretchTree is the star decomposition algorithm previously described. It first calculates a radius value, and forms the central component

V_0 as a the ball with this radius around x_0. It then forms the remaining components by repeatedly choosing an available node from the ball shell of V_0, and forming a cone of certain radius around it. The pseudocode of StarDecomp, cited from the paper:

$(\{V_0, ..., V_k, x, y\})$ =StarDecomp$(G = (V, E, l), x_0, \delta, \varepsilon)$

1. Set $\varrho = rad_G(x_0)$; Set r_0 =BallCut$(G, x_0, \varrho, \delta)$ and $V_0 = B(x_0, r_0)$;
2. Let $S = BS(x_0, r_0)$;
3. Set $G' = (V', E', l') = G[V - V_0]$, the weighted graph induced by $V - V_0$;
4. Set $(\{V_1, ..., V_k\}, x)$ =ConeDecomp$(G', S, \frac{\varepsilon \cdot \varrho}{2})$;
5. For each $i \in [1 : k]$, set y_k to be a vertex in V_0 such that $(x_k, y_k) \in E$ and y_k is on a shortest path from x_0 to x_k . Set $y = (y_1, ..., y_k)$.

$(\{V_1, ..., V_k\}, x)$ =ConeDecomp(G, S, Δ)

1. Set $G_0 = G$, $S_0 = S$, and $k = 0$.
2. While S_k is not empty
   (a) Set $k = k + 1$; Set x_k to be a vertex of S_{k-1};
   Set r_k =ConeCut$(G_{k-1}, x_k, 0, \Delta, S_{k-1})$
   (b) Set $V_k = C_{S_{k-1}}(r_k, x_k)$; Set $G_k = G\left[V - \bigcup_{i=1}^{k} V_i\right]$ and $S_k = S_{k-1} - V_k$.
3. Set $x = (x_1, ..., x_k)$.

The StarDecomp method returns the components of the decomposition, and the x_i, y_i endpoints of the bridge edges collected into two vectors, x and y. The BallCut method is invoked to develop the central ball component, the ConeDecomp is called to sort the remaining vertices into the other components of the star decomposition. Finally, endpoints of the bridges in the central component are selected.

The ConeDecomp method repeatedly chooses an anchor vertex x_i from the ball shell of the central component, and calls ConeCut to calculate a radius, then forms the next component as a cone of this radius around the chosen vertex. When developing the cone of the next iteration, the previously sorted vertices are not considered anymore.

The pseudocode of the BallCut method can be seen below, cited from the paper.

r =BallCut$(G, x_0, \varrho, \delta)$

1. Set $r = \varrho \cdot \delta$.
2. While $\text{cost}(\partial(B(x_0, r))) > \frac{\text{vol}(B(x_0, r)) + 1}{(1 - 2 \cdot \delta) \cdot \varrho} \cdot \log_2(m + 1)$

(a) Find the vertex $v \notin B(x_0, r)$ that minimizes $\text{dist}(x_0, v)$ and set $r = \text{dist}(x_0, v)$.

Recall that m represents the edges of the graph that is the input to the LowStretchTree method on the present recursive level. As can be seen, BallCut continues to add vertices to the developed ball until the sum of the cost of the edges coming out of the ball is larger than the expression on the right side.

Finally, the pseudocode of the ConeCut method is shown.

r =ConeCut$(G, v, \lambda, \lambda', S)$

1. Set $r = \lambda$ and if $|E[C_S(v, \lambda)]| = 0$, then set $\mu = (\text{vol}(C_S(v, r)) + 1) \cdot \log_2(m + 1)$. Otherwise, set $\mu = (\text{vol}(C_S(v, r))) \cdot \log_2\left(\frac{m}{|E[C_S(v,\lambda)]|}\right)$
2. While $\text{cost}(\partial(C_S(v, r))) > \frac{\mu}{\lambda' - \lambda}$
   (a) Find the vertex $w \notin C_S(v, r)$ minimizing $\text{dist}(w, C_S(v, r))$ and set $r = r + \text{dist}(w, C_S(v, r))$.

Again the method is adding vertices to the cone until the sum of the cost of leaving edges becomes larger than a certain expression.

By calling these methods as described, the algorithm finds a spanning tree of the original input graph, the radius and average stretch of which is claimed to have certain properties. These statements (and the lemmas they are built on) are presented in the next section.

3.1.3 Theorems and claims

When analyzing the behavior and performance of the algorithm, one of the most basic requirements is that each method presented terminates after some point, which is not trivial at all for the BallCut and ConeCut algorithms. For these methods, it needs to be proven that there exists a point in the process of adding nodes where the examined cost becomes smaller than the other expression. This is done with the help of a claim on concentric systems, which appears in the article as cited below.

Lemma 4.2 of [2] (Concentric System Cutting). *Let $G = (V, E, \ell)$ be a connected weighted graph and let $L = L_r$ be a concentric system. For every two reals $0 \leq \lambda \leq \lambda'$, there exists a real $r \in [\lambda, \lambda')$ such that*

$$cost(\partial(L_r)) \leq \frac{\text{vol}(L_r) + \tau}{\lambda' - \lambda} \cdot \max\left[1, log_2\left(\frac{m + \tau}{|E[L_\lambda]| + \tau}\right)\right].$$

where $m = |E|$,
and $\tau = 1$ if $|E[L_\lambda]| = 0$, and $\tau = 0$ otherwise.

Using this statement and the fact that both balls and cones form concentric systems, it is proven in the paper that both BallCut and ConeCut will find a subgraph, for which the cost of leaving edges is below the expression it is compared to.

Two more lemmas are required in the process of proving that the star decomposition returned has the claimed properties. One of these lemmas, proposition 4.7 of the paper, will be analyzed deeper in Section 3.2. The other one is briefly mentioned here, cited below; it points out that removing a cone does not increase the radius of the remaining graph.

Proposition 4.8 of [2] (Deleting Cones). *Consider a connected weighted graph $G = (V, E, \ell)$, a vertex subset $S \subseteq V$, a vertex $x \in S$ and a real $l \geq 0$ and let $V' = V - C_S(x, l)$, $S' = S - C_S(x, l)$ and $\Psi = \max_{v \in V} \operatorname{dist}(v, S)$. Then*

$$\max_{v \in V'} \operatorname{dist}_{V'}(v, S) \leq \Psi.$$

In our next citation, the required properties of the decomposition are stated, with a proof built on these claims.

Lemma 3.2 of [2] (Low-Cost Star Decomposition). *Let $G = (V, E, \ell)$ be a connected weighted graph and let x_0 be a vertex in V. Then for every positive $\varepsilon \leq \frac{1}{2}$,*

$$(\{V_0, ..., V_k, x, y\}) = \mathit{StarDecomp}(G, x_0, \frac{1}{3}, \varepsilon)$$

in time $O(m + n \log n)$ returns a $(\frac{1}{3}, \varepsilon)$-star-decomposition of G with center x_0 of cost

$$\mathit{cost}(\partial(V_0, ..., V_k)) \leq \frac{6 \cdot m \cdot \log_2(m+1)}{\varepsilon \cdot \mathit{rad}_G(x_0)}.$$

On unweighted graphs, the running time is $O(m)$.

In the proof, propositions 4.7 and 4.8 are used to ensure that the returned decomposition is in fact a $(\frac{1}{3}, \varepsilon)$-star-decomposition, while the terminating conditions of BallCut and ConeCut are used to show that the inequality on the cost of the decomposition holds. A careful analysis of the algorithm yields the statements on the runtime.

Finally, the main theorem of the paper can be stated, which gives a bound on the radius and average edge stretch of the returned spanning tree. The proofs of the inequalities rely on the decomposition properties, while the runtime claims can again be proved through a careful study of the algorithm. This theorem is also cited from the article.

Theorem 3.5 of [2] (Low-Stretch Spanning Tree). *Let $G = (V, E, \ell)$ be a connected weighted graph and let x_0 be a vertex in V. Then*

$$T = \mathit{LowStretchTree}(G, x_0),$$

in time $O(\widetilde{m} \log \widetilde{n} + \widetilde{n} \log^2 \widetilde{n})$, returns a spanning tree of G satisfying

$$\mathit{rad}_T(x_0) \leq 2 \cdot e \cdot \mathit{rad}_G(x_0)$$

and

$$AveStretch_T(E) \leq O(\log^3 \widetilde{m}).$$

The paper also presents an improved version of the algorithm, for which a tighter bound on the stretch can be proven. This modification is summarized in the next section.

3.1.4 Improving the stretch

As mentioned, the article also presents an improved version of the algorithm, which uses a different version of the ConeDecomp method, named ImpConeDecomp, which repeatedly applies ConeCut, and only accepts a cone if the number of its edges falls under a certain limit. The pseudocode for this method, cited from the paper, is the following:

$(\{V_1, ...V_k, x\})$ =ImpConeDecomp$(G, S, \Delta, t, \widetilde{m})$

1. Set $G_0 = G$, $S_0 = S$, and $j = 0$.
2. While S_j is not empty
   (a) Set $j = j + 1$; Set x_j to be a vertex of $S_j - 1$ and set $p = t - 1$;
   (b) While $p > 0$
      i. r_j =ConeCut$(G_{j-1}, x_j, \frac{(t-p-1)\cdot\Delta}{t}, \frac{(t-p)\cdot\Delta}{t}, S_{j-1})$
      ii. if $\left|E\left[C_{Sj-1}(r_j, x_j)\right]\right| \leq \frac{m}{2^{\log\frac{p}{t}\,\widetilde{m}}}$ then exit the loop; else $p = p - 1$;
   (c) Set $V_j = C_{Sj-1}(r_j, x_j)$ and set $G_j = G\left[V - \bigcup_{i=1}^{j} V_i\right]$ and $S_j = S_{j-1} - V_j$;
3. Set $x = (x_1, ..., x_k)$.

This algorithm has an extra parameter t, set by the authors of the paper to the value $\log\log n$, hence in this version of the algorithm, ImpConeDecomp$(G', S, \frac{\varepsilon\cdot\varrho}{2}, t, \widetilde{m})$ is invoked in StarDecomp, with this t value. It can also be proven for this decomposition method to return a star decomposition with the required properties, and a series of inequalities (different from that of Lemma 3.2) can be shown to hold for the cost of the edges leaving the components of the decomposition. Using these bounds, it can be proven that this improved version of the LowStretchTree algorithm yields a spanning tree for which an average stretch value of $O(\log^2 \widetilde{n} \log\log \widetilde{n})$ can be achieved, which is a slightly better bound than the $O(\log^3 \widetilde{n})$ obtained for the previous version of the algorithm. Therefore, although both the algorithm itself and the related proofs are more complicated, this version can guarantee better average edge stretch for the spanning tree it finds.

3.2 Incorrect statements in the article

Unfortunately, I have found that some statements presented in the article are not valid. Although most of the proofs in the article seem to be correct, some of these proofs (directly

or indirectly) use the statement of a minor claim of the article, proposition 4.7, which does not seem to hold. Since the most important statements of the article (Lemma 3.2, Theorem 3.5) are also built on this proposition, this raises the question whether the specified bound on the average stretch of the spanning tree returned by the algorithm holds or not. This section is to provide counterexample to some of these incorrect statements.

3.2.1 Inequality on the radius of cones

All the problematic statements of the article are based on proposition 4.7, which is the following:

Propisition 4.7 of [2] (Radius of Cones). *Let $G = (V, E, \ell)$ be a connected weighted graph, let $x_0 \in V$, and let $\varrho = rad_G(x_0)$. Let $r_0 < \varrho$ and let $V_0 = B(x_0, r_0)$, $V' = V - V_0$ and $S = BS(x_0, r_0)$. Let x be any vertex in S and let $\psi = \varrho - \text{dist}_G(x_0, x)$. Then the cones $C_S(x_1, l)$ in the graph $G[V']$ satisfy $rad_{C_S(x,l)}(x) \leq \psi + 2 \cdot l$.*

Unfortunately, a counterexample can be found to this statement relatively easily.

The graph for the concrete counterexample is shown in *Figure 2*. In this example graph G, $rad_G(x_0) = \varrho = 9.2$; let $r_0 = 3.1$, so $V_0 = B(x_0, r_0) = \{x_0, b_1, b_2, b_3\}$ and $S = BS(x_0, r_0) = \{s_1, s_2, s_3\}$. Let us consider the cone $C_S(s_1, 0.2)$ in the graph spanned by $V - V_0$. The forward edges for S are the directed edges marked with an arrow on the figure. This means that $C_S(s_1, 0.2)$ consists of s_1, c_1, c_2, c_3, and $rad_{C_S(s_1,0.2)}(s_1) = 6.2$. In the graph, $\psi = \varrho - \text{dist}_G(x_0, s_1) = 9.2 - 6.1 = 3.1$. This way, proposition 4.7. of the article states that $rad_{C_S(s_1,0.2)}(s_1) \leq \psi + 2 \cdot l$, however, in our current setting this is $6.2 \leq 3.1 + 2 \cdot 0.2 = 3.5$, which is not the case. Therefore, proposition 4.7 does not hold in this case.

Intuitively, the main problem with the proposition is that the shortest path from x_0 to a node (in this case, c_3) is not necessarily going through s_1, and so the sum of the shortest path lengths between x_0 and s_1 and between s_1 and c_3 can be significantly larger than the shortest path length between x_0 and c_3, which can be equal to the radius if no point is further away from x_0 than c_3. The proof included in the paper also suggests that the authors assumed that the shortest path will certainly go through the vertex chosen in the ball shell; the concrete statement built on this assumption is the line $\text{dist}_G(x_0, b_{i+1}) = \text{dist}_G(x_0, a_i) + \text{dist}_{G(V')}(a_i, b_{i+1})$ in the technical proof of the article, which is not a valid claim.

3.2.2 Incorrect star decomposition

The claim of Proposition 4.7 is applied in the proof of lemma 3.2, which states that the star-decomposition algorithm of the article always returns a $(\frac{1}{3}, \varepsilon)$-star-decomposition if called with the specified parameters. However, since Proposition 4.7 is incorrect, so is the part of the statement it is applied in, which is the proof of the second property of a $(\frac{1}{3}, \varepsilon)$-decomposition, stating that for each component the inequality $\text{dist}(x_0, x_i) + r_i \leq (1 + \varepsilon) \cdot \varrho$ holds, where r_i denotes the radius of the i^{th} component from x_i. We show that in some cases where proposition 4.7 does not hold, this property does not hold either, and therefore the star decomposition returned by the algorithm of the article is not necessarily a $(\frac{1}{3}, \varepsilon)$-star-decomposition.

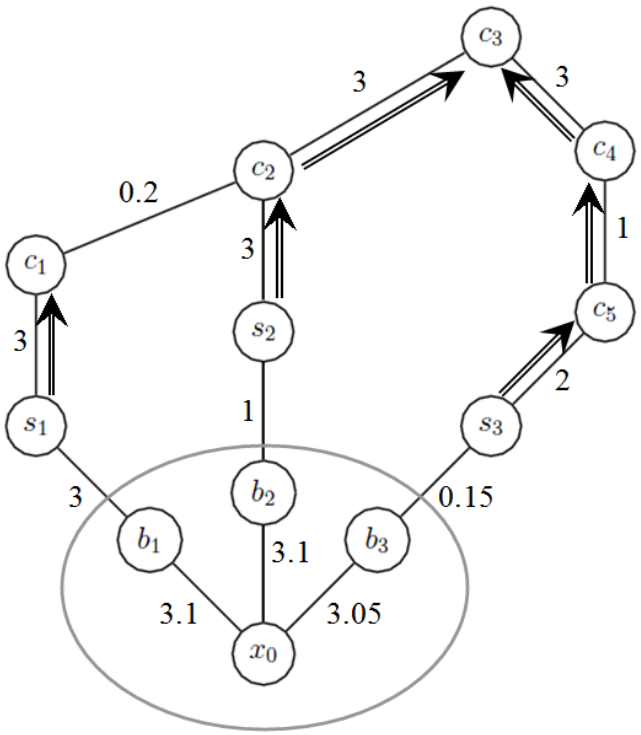

Figure 2: *Counterexample to Proposition 4.7. The encircled part indicates the developed V_0, and the arrows mark the forward edges in the graph.*

For the counterexample, consider the graph on *Figure 2* again. As before, $rad_G(x_0) = \varrho = 9.2$. Clearly $|E| = 13$, and thus $\log(|E|+1) \approx 3.8074$. Let's assume $\varepsilon = 0.322$ all through this counterexample. As in the article, the cost of an edge is defined to be the reciprocal of its length.

As in the previous example, let $r_0 = 3.1$, $V_0 = B(x_0, r_0) = \{x_0, b_1, b_2, b_3\}$, $S = BS(x_0, r_0) = \{s_1, s_2, s_3\}$, and as before, $C_S(s_1, 0.2)$ in the graph spanned by $V - V_0$ will contain the nodes $\{s_1, c_1, c_2, c_3\}$. We first show that it is possible for the star decomposition algorithm of the article to return these components when decomposing a graph. For this, let us simulate the execution of the StarDecomp method on this graph with $\delta = \frac{1}{3}$ and the previously specified ε. This begins by the StarDecomp method first calling BallCut with parameters $(x_0, \varrho = 9.2, \frac{1}{3})$.

The first step in the execution of BallCut is setting the value of r to $\frac{\varrho}{3} = \frac{9.2}{3} \approx 3.0667$. At this point, $B(x_0, r) = \{x_0, b_3\}$, which contains the edge (x_0, b_3) entirely and one endpoint of three other edges (edges (x_0, b_1), (x_0, b_2) and (b_3, s_3)), so $\mathrm{vol}(B(x_0, r)) = 4$. This means that the value of the expression in the loop is $\frac{3\cdot(4+1)\cdot\log(|E|+1)}{\varrho} \approx 6.2077$. At this stage, $\partial(B(x_0, r))$ is made up of 3 edges, and the cost in question is the sum of the reciprocals of the lengths belonging to these edges, which is $\frac{1}{0.15} + \frac{1}{3.1} + \frac{1}{3.1} \approx 7.3118$. Since this is larger than the previous expression, r will be assigned a new value in the loop, $r = 3.1$. In the second execution of the loop, there will be 3 edges (those that have x_0 as an endpoint) with both endpoints in $B(x_0, r)$, and another 3 (edges (b_1, s_1), (b_2, s_2) and

(b_3, s_3)) with one endpoint in it, hence $\mathrm{vol}(B(x_0, r)) = 6$. Thus at the next evaluation, $\frac{3\cdot(6+1)\cdot\log(|E|+1)}{\varrho} \approx 8.6908$ will be compared to $\partial(B(x_0, r))$, which is $\frac{1}{0.15} + \frac{1}{1} + \frac{1}{3.1} \approx 7.9892$ at this point. Since this time the value of $\partial(B(x_0, r))$ is smaller, the loop will terminate, and BallCut will indeed return $V_0 = B(x_0, 3.1)$, with the appropriate set S of vertices.

StarDecomp then calls the ConeDecomp method. Suppose that while executing ConeDecomp, we choose the node x to be s_1; then ConeCut is called. The method ConeCut will first set $r = 0$, that creates the Cone $\{s_1, c_1\}$, in which the edge (s_1, c_1) is already included, so $|E[C_S(s_1, 0)]| = 1$. Thus the "otherwise" branch is chosen in ConeCut, and the value of μ is set to $\mathrm{vol}(C_S(s_1, 0)) \cdot \log\left(\frac{m}{|E[C_S(s_1,0)]|}\right) = 2 \cdot \log\left(\frac{m}{1}\right) \approx 7.4009$. With this, we get the value $\frac{\mu}{\lambda'-\lambda} = \frac{\mu}{\Delta-0} = \frac{\mu}{\Delta} = \frac{2\cdot\mu}{\varepsilon\cdot\varrho} = \approx \frac{14.8018}{0.322\cdot 9.2} \approx 4.9965$ to compare to boundary costs in the loop. When we first examine it, only edge (c_1, c_2) leaves the cone, the cost of which is $\frac{1}{0.2} = 5$, which is slightly bigger than the previous value, so we enter the loop, where the parameter of the cone will be increased by 0.2, to $C_S(s_1, 0.2)$. This new cone will contain the vertices $\{s_1, c_1, c_2, c_3\}$. The edges leaving this cone are (c_2, s_2) and (c_3, c_4), the sum of their costs is $\frac{1}{3} + \frac{1}{3} = \frac{2}{3}$, which is smaller than 4.9959, therefore the loop will terminate, and return $C_S(s_1, 0.2)$, the radius of which is 6.2 from node s_1.

Although ConeDecomp continues running after this, this is irrelevant for the present counterexample; it is enough to see that the decomposition returns the components $V_0 = B(x_0, 3.1)$ and $V_1 = C_S(s_1, 0.2)$. Also, note that y_1 has to be a node on the shortest path between $x_1 = s_1$ and x_0, therefore it can only be b_1.

Finally, we are able to show that the statement of Lemma 3.2 is incorrect. The lemma claims that the decomposition obtained is a $\left(\frac{1}{3}, \varepsilon\right)$-star-decomposition, which seems to be false in our current case. According to property b), $\mathrm{dist}(x_0, x_1) + r_1 \leq (1+\varepsilon)\cdot\varrho$ must hold for such decompositions. However, in our current case we get $6.1 + 6.2 \leq (1 + 0.322) \cdot 9.2$, so $12.3 \leq 12.1624$, which is a contradiction. Thus the part of Lemma 3.2 relying on Proposition 4.7 is incorrect, and the decomposition returned by the algorithm will not necessarily be a $\left(\frac{1}{3}, \varepsilon\right)$-star-decomposition.

Although it is important to know how the claims article are affected by the incorrectness of Lemma 3.2, assembling the complete list of consequences of the invalid proposition in the article is beyond the scope of this thesis. However, the bound on the cost of the star decomposition plays a key role in the proof of the main statement of the article. Therefore, with Proposition 4.7 being incorrect, the proof of the upper bound on the average edge stretch of the returned spanning tree is also invalid.

3.3 Introducing different cones

With these statements of the paper being invalid, it is an important question whether the algorithm can be modified in a way such that all the essential claims of the article become valid. A good opportunity for this is offered by the more recent article of Abraham *et al.* [3], presented in detail later in Section 4.2, that adds a number of major improvements to the LowStretchTree algorithm, and therefore may be considered as a completely different method than the original one. Although this paper also works with star decompositions, which are created by first forming a ball around a central vertex, and then repeatedly cutting cones from the remaining vertices, it defines cones in an entirely different way. In

this section, we show that if we also use this new definition to form cones in our current algorithm, then the main theorem of the original paper becomes valid for this new version of the algorithm, that is, the proven bounds indeed hold on the radius and average stretch of the returned spanning tree.

3.3.1 Modified cones

The new definition we introduce for cones, based on the paper by Abraham *et al.*, is as follows:

Definition (Cones, new definition). *Consider a weighted graph $G = (V, E, \ell)$ and subset $Y \subseteq V$, and any given points $x \in V - Y$ and $y \in Y$. For any real l, the modified cone or cone according to new definition of radius l around node y, denoted $\widetilde{C}_{Y,x}(y, l)$, is the set of vertices $z \in Y$ such that $d_V(x, y)+d_Y(y, z)-d_V(x, z) \leq l$.*

That is, the cone is the set of points to which the shortest path from x that intersects node y is at most l longer than the shortest path from x. When applying this new definition to the article, we will set $Y = G\left[V - \bigcup_{i=0}^{j} V_i\right]$ (the points not yet sorted into any component), $x = x_0$ and as y, the currently selected anchor vertex (in the ball shell of V_0) will be chosen. Since the basic problem with the original claim of the article was that the radius of the decomposed graph depends on the distance from x_0, not on the distance from the ball shell S of V_0, introducing a new definition that sorts vertices into cones according to distance from x_0 rather than according to distance from S seems to be a promising way to correct the invalid statements.

Hence, we can modify ConeCut to work with this new definition of cones, using the previously specified parameters. In this version, every time the method adds a new vertex to the cone around the anchor vertex y, it increases the radius to the smallest value for which a new vertex is added to the cone, and therefore chooses the vertex (or vertices) z that minimizes the expression $\mathrm{d}_V(x_0, y)+\mathrm{d}_Y(y, z)-\mathrm{d}_V(x_0, z)$, and this minimum becomes the new radius of the cone being created.

Note that this modification does not affect the running time of the star decomposition significantly. When developing the central component in the BallCut method invoked by StarDecomp, the distance of the vertices from x_0 is calculated, and can then be passed on as a parameter to the modified ConeCut method. This way, ConeCut only needs to calculate the distance of every vertex from the chosen anchor vertex in the graph it receives as a parameter, using Dijkstra's algorithm. However, the previous version of ConeCut also had to apply a modified version of Dijkstra's algorithm to determine the cone distance of each vertex from the anchor vertex (this version calculated as if the length of forward edges was 0), hence running the new ConeCut method does not require more time than running the original method.

3.3.2 Weak Concentric Systems

Naturally, if modified cones are to be used in the new version of the method, it has to be proven that the cost of the edges leaving the cone will certainly become smaller at

some point than the loop condition expression, thus this new ConeCut method will also terminate. In the case of the original version of ConeCut, this is done using Lemma 4.2 for concentric system cutting. Unfortunately, it can be seen that the newly introduced cones do not necessarily form concentric systems, since the third property of concentric systems will not hold in some cases.

When a system of cones with parameter l includes a vertex u from which an edge (u, v) of length d is leaving, it is not certain that the cone with parameter increased by d will contain the vertex v and so $\mathrm{d}_V(x_0, y)+\mathrm{d}_Y(y, v)-\mathrm{d}_V(x_0, v) \leq l + d$ will hold. The problem is that it is possible that the new node v will be both further from y by d and closer to x_0 by d, that is, if compared to the original expression $\mathrm{d}_V(x_0, y)+\mathrm{d}_Y(y, u)-\mathrm{d}_V(x_0, u) \leq l$, both $\mathrm{d}_Y(y, v)$ will be larger than $\mathrm{d}_Y(y, u)$ by d and $\mathrm{d}_Y(x_0, v)$ will be smaller than $\mathrm{d}_Y(x_0, u)$ by d, and therefore the left hand side will be increased by $2 \cdot d$, while the right side is only increased by d.

A concrete example for this is illustrated in *Figure 3*. Let $Y = \{y, v_1, v_2\}$ in the figure, and consider the cone $\widetilde{C}_{Y,x_0}(y, l)$ for $l = 0$. This cone contains y (obviously) and also v_1, since $\mathrm{d}_V(x_0, y)+\mathrm{d}_Y(y, v_1)-\mathrm{d}_V(x_0, v_1) = 2 + 1 - 3 = 0$. Now, because of the edge (v_1, v_2), if the third property of concentric system was true for modified cones, then v_2 should be included in $\widetilde{C}_{Y,x_0}(y, l')$ for $l' = l + \ell(v_1, v_2) = 1$. However, for v_2, $\mathrm{d}_V(x_0, y)+\mathrm{d}_Y(y, v_2)-\mathrm{d}_V(x_0, v_2) = 2 + 2 - 2 = 2$, which is larger than l', so v_2 is not contained in $\widetilde{C}_{Y,x_0}(y, l')$. Note that the reason for this is indeed the fact that both $\mathrm{d}_Y(y, v_2)$ is bigger than $\mathrm{d}_Y(y, v_1)$ by 1 and $\mathrm{d}_V(x_0, v_2)$ is smaller than $\mathrm{d}_V(x_0, v_1)$ by 1, while the difference between l and l' is 1 (that is, only one times the length of the edge (v_1, v_2)).

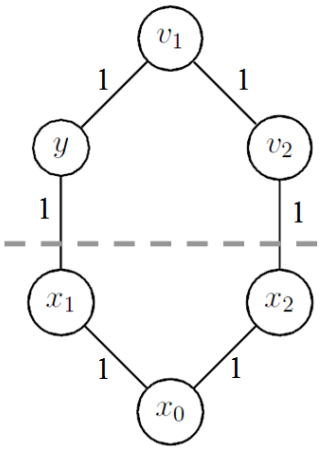

Figure 3: *Example graph to illustrate that modified cones are not concentric. The subset Y consist of the vertices above the dashed line.*

Hence, a new definition for concentric systems is required which also suits the notion of modified cones. These new systems will be called weak (or weakly) concentric systems, and they will be designed to handle the problem that the length of a new edge can completely be included in two terms on the left side of the inequality, and therefore the containment of the endpoint of the edge can only be guaranteed if the parameter of the cone is increased by twice the length of the edge. For this, it is only the third property of concentric systems that has to be modified.

Definition (Weak Concentric System). *A weak concentric system in a weighted graph* $G = (V, E, \ell)$ *is a family of vertex sets* $L = \{L_r \subseteq V : r \in \mathbb{R}^+ \cup \{0\}\}$ *such that*

a) $L_0 \neq \emptyset$,

b) $L_r \subseteq L_{r'}$ *for all* $r < r'$, *and*

c) if a vertex $u \in L_r$ *and* (u, v) *is an edge in* E *then* $v \in L_{r+2 \cdot d(u,v)}$.

It is easy to see that this new definition applies to a system of modified cones.

Claim (Modified Cones are Weakly Concentric). *Consider a weighted graph* $G = (V, E, \ell)$, *a subset* $Y \subseteq V$, *and any given points* $x \in V - Y$ *and* $y \in Y$. *For the real numbers, the system* $L = \{\widetilde{C}_{Y,x}(y, l) : l \in \mathbb{R}^+ \cup \{0\}\}$ *of modified cones is a weak concentric system in* $G[Y]$.

Proof. The first property obviously holds; node y will always be contained in $\widetilde{C}_{Y,x}(y, 0)$, since $\mathrm{d}_V(x, y) + \mathrm{d}_Y(y, y) - \mathrm{d}_V(x, y) = 0 \leq 0$.
The second is also trivial, if $r < r'$ and $\mathrm{d}_V(x, y) + \mathrm{d}_Y(y, z) - \mathrm{d}_V(x, z) \leq r$, then $\mathrm{d}_V(x, y) + \mathrm{d}_Y(y, z) - \mathrm{d}_V(x, z) \leq r'$.
Finally, suppose that $z, z' \in Y$, $(z, z') \in E$ and $z \in \widetilde{C}_{Y,x}(y, l)$ for some l; we have to show that in this case, $\mathrm{d}_V(x, y) + \mathrm{d}_Y(y, z') - \mathrm{d}_V(x, z') \leq l + 2 \cdot \mathrm{d}_Y(z, z')$. Because z is in the cone, we know that $\mathrm{d}_V(x, y) + \mathrm{d}_Y(y, z) - \mathrm{d}_V(x, z) \leq l$. Since $\mathrm{d}_Y(y, z') \leq \mathrm{d}_Y(y, z) + \mathrm{d}_Y(z, z')$, this implies

$$\mathrm{d}_V(x, y) + \mathrm{d}_Y(y, z') - \mathrm{d}_V(x, z) \leq l + \mathrm{d}_Y(z, z'),$$

and because $\mathrm{d}_V(x, z) \leq \mathrm{d}_V(x, z') + \mathrm{d}_Y(z, z')$, the inequality

$$\mathrm{d}_V(x, y) + \mathrm{d}_Y(y, z') - \mathrm{d}_V(x, z') \leq l + 2 \cdot \mathrm{d}_Y(z, z')$$

holds indeed. □

Since the original proof of the claim that the algorithm terminates used the lemma on concentric system cutting and the fact that cones of the original definition formed concentric systems, these part of the paper all have to be adapted to the new concept of weak concentric systems. Fortunately, the lemma for concentric system cutting only requires minimal modifications to work for weak concentric systems: the expression on the right side only has to be multiplied by a factor of 2.

Lemma (Weakly Concentric System Cutting). *Let* $G = (V, E, \ell)$ *be a connected weighted graph and* $L = \{L_r\}$ *be a weak concentric system. For every two reals* $0 \leq \lambda \leq \lambda'$, *there exists a real* $r \in [\lambda, \lambda')$ *such that*

$$cost(\partial(L_r)) \leq \frac{2 \cdot \mathrm{vol}(L_r) + \tau}{\lambda' - \lambda} \cdot \max\left[1, log_2\left(\frac{m + \tau}{|E\left[L_\lambda\right]| + \tau}\right)\right],$$

where $m = |E|$, *and* $\tau = 1$ *if* $|E\left[L_\lambda\right]| = 0$, *and* $\tau = 0$ *otherwise.*

To prove this, only minimal modifications are required in the (rather complicated) proof of the original 'concentric system cutting' claim. It can be noted that the modified property of concentric systems is only applied at two points in the proof. The first is in the case when there is no vertex between norm λ and λ', where the norm of a vertex means the smallest r value for which the vertex is in L_r. In this case, all the edges in the graph leaving $L_{(\lambda+\lambda')/2}$ have to be at least $\lambda - \lambda'$ long, since otherwise the third property would guarantee that a new vertex is added to the system between λ and λ'. This means that the cost of each such edge is at most $\frac{1}{\lambda-\lambda'}$, and this implies that the sum of the cost of edges leaving $L_{(\lambda+\lambda')/2}$ is at most

$$|\partial(L_{(\lambda+\lambda')/2})| \cdot \frac{1}{\lambda - \lambda'} \leq \mathrm{vol}(L_{(\lambda+\lambda')/2}) \cdot \frac{1}{\lambda - \lambda'}.$$

In the new lemma using weakly concentric systems, very similar statements can be made: since this way, we can only state that endpoints will certainly be included when the increase is twice their lengths, we can only claim that every such edge in weak concentric systems leaving $L_{(\lambda+\lambda')/2}$ is at least $(\lambda+\lambda')/2$ long, and thus has a cost of at most $2/(\lambda+\lambda')$. With this, the sum of costs can be bounded by

$$|\partial(L_{(\lambda+\lambda')/2})| \cdot \frac{2}{\lambda - \lambda'} \leq \mathrm{vol}(L_{(\lambda+\lambda')/2}) \cdot \frac{2}{\lambda - \lambda'}.$$

This way, the new form of the lemma is established in this case. The other point in the proof using the third property has to be modified in a completely identical way. This point is the second case of the proof, when for some vertices v_i and v_{i+1} with norms r_i and r_{i+1}, there are no vertices with norm between r_i, and r_{i+1}, and $r_{i+1} - r_i \geq (\lambda - \lambda')/\eta$, where η denotes the logarithmic expression in the lemma. The original lemma used that since the difference of parameters is at least $r_{i+1} - r_i$, then all the edges stepping out of L_{r_i} are at least $(\lambda - \lambda')/\eta$ long, and thus have costs of at most $\eta/(\lambda - \lambda')$. By this, the sum of costs is at most

$$|\partial(L_{(\lambda+\lambda')/2})| \cdot \frac{\eta}{\lambda - \lambda'} \leq \mathrm{vol}(L_{(\lambda+\lambda')/2}) \cdot \frac{\eta}{\lambda - \lambda'}.$$

In the new lemma, the difference is that we only know that each edge leaving L_{ri} is at least $(\lambda - \lambda')/2 \cdot \eta$ long, and thus have costs of at most $2 \cdot \eta/(\lambda - \lambda')$. This bounds the sum of costs by

$$|\partial(L_{(\lambda+\lambda')/2})| \cdot \frac{2 \cdot \eta}{\lambda - \lambda'} \leq \mathrm{vol}(L_{(\lambda+\lambda')/2}) \cdot \frac{2 \cdot \eta}{\lambda - \lambda'}.$$

Since the proof of the lemma only uses the third property of concentric systems at these two points, and after switching concentric systems to weak concentric systems, the double of the original bound still holds for the cost, the new lemma for weakly connected system cutting is established.

3.3.3 The propositions with the new cones

To create star decompositions with modified cones, we have to show that these cones also have the basic properties that were proven in Propositions 4.7 and 4.8 for the original cones.

New Propisition 4.7 (Radius of Modified Cones). *Let $G = (V, E, \ell)$ be a connected weighted graph, let $x_0 \in V$, and let $\varrho = rad_G(x_0)$. Let $r_0 < \varrho$ and let $V_0 = B(x_0, r_0)$, $Y = V - V_0$ and $S = BS(x_0, r_0)$. Let y be any vertex in S and let $\psi = \varrho - \mathrm{dist}_G(x_0, y)$. Then the modified cones $\widetilde{C}_{Y,x_0}(y, l)$ in the graph $G[Y]$ satisfy $rad_{\widetilde{C}_{Y,x_0}}(y, l)(y) \leq \psi + 2 \cdot l$.*

Proof. **Proof.** We have to show that for every point z of the modified cones, $\mathrm{d}_Y(y, z) \leq \varrho - \mathrm{d}_V(x_0, y) + 2 \cdot l$ holds, so $\mathrm{d}_Y(y, z) + \mathrm{d}_V(x_0, y) - \varrho \leq 2 \cdot l$. Since ϱ is at least $\mathrm{d}_V(x_0, z)$, the left hand side is only increased if altered to $\mathrm{d}_Y(y, z) + \mathrm{d}_V(x_0, y) - \mathrm{d}_V(x_0, z)$. This is true according to the definition of the modified cone; it would even hold if there was only one l on the right side. □

New Propisition 4.8 (Deleting Modified Cones). *Consider a connected weighted graph $G = (V, E, \ell)$, a vertex $x_0 \in V$, a real $r < rad_G(x_0)$, the vertex subset $Y = B(x_0, r)$, a vertex $y \in Y$, a real $l \geq 0$ and let G' be the graph spanned by $V' = V - \widetilde{C}_{Y,x_0}(y, l)$. Also, let $\varrho = rad_G(x_0)$. Then*

$$rad_{G'}(x_0) \leq \varrho.$$

Proof. This is a direct consequence of the fact that after deleting a modified cone, for each of the remaining nodes z, all shortest paths from x_0 to z are entirely included in the remaining graph. The proof of this more general claim is also shown in [3]. Suppose that that the modified cone contains some point v in Y of a shortest path from x_0 to z, then $\mathrm{d}_V(x_0, y) + \mathrm{d}_Y(y, v) - \mathrm{d}_V(x_0, v) \leq l$. Using the facts that $\mathrm{d}_Y(y, z) \leq \mathrm{d}_Y(y, v) + \mathrm{d}_Y(v, z)$ and $\mathrm{d}_V(x_0, z) = \mathrm{d}_V(x_0, v) + \mathrm{d}_Y(v, z)$, we can see that node z will certainly be contained in the cone, since

$$\mathrm{d}_V(x_0, y) + \mathrm{d}_Y(y, z) - \mathrm{d}_V(x_0, z) \leq$$

$$\leq \mathrm{d}_V(x_0, y) + \mathrm{d}_Y(y, v) + \mathrm{d}_Y(v, z) - \mathrm{d}_V(x_0, v) - \mathrm{d}_Y(v, z) \leq l.$$

Thus, the maximum of the distance of vertices in Y from x_0 can not increase by deleting a cone, so the proposition holds. □

We have seen that both propositions are true for modified cones, therefore these cones have the same basic properties as the original ones had (or, in fact, supposed to have had). Note that Proposition 4.7 not only holds in this case, but also holds with only one l on the right side, which will be the key to a possible improvement to the algorithm in the next section.

3.3.4 The main statements of Elkin *et al.*'s paper

With the previously presented properties, it is clear that the algorithm can also work with the new definition of cones, and it is easy to see that the proofs in the paper can be adjusted to this new case with only minimal modifications. Since these cones are still

weakly concentric, the claim that ConeCut will return a cut with cost under a certain limit will hold if we take the double of the original limit, based on the lemma of cutting weakly concentric systems. The first part of Lemma 3.2 in the article (stating that StarDecomp returns a $(\frac{1}{3}, \varepsilon)$-star-decomposition) only relies on Propositions 4.7 and 4.8, and thus will hold in this new case. The proof of the second part (bounding the cost of the decomposition) simply uses the sum of the limits on the cuts to bound the cost of the partition, so with the bounds on the cuts being doubled, the resulting bound on the cost will also be doubled. Hence, Lemma 3.2 will hold in this case, with a twice as large upper bound on the cost as the original one.

However, this does not have any effect on the proof of the main theorem of the paper, Theorem 3.5, in which the previous lemma is applied. In this proof, the cost of the decomposition edges is bounded by the lemma, then this is summed up for all recursion levels. Multiplying the bound by 2 only results in the bound for the sum of the stretch for all edges being also doubled, but this does not change the fact that the resulting formula is in $O(\widetilde{m} \log^3 \widetilde{m})$, and so the average stretch is still $O(\log^3 \widetilde{m})$. Therefore, although the modifications change the upper bound given on the average stretch by a factor of 2, its magnitude will remain the same, and hence the statement of the theorem will indeed hold for this new version of the star decomposition algorithm.

Note that another slight modification can correct the algorithm without increasing the upper bound to twice as large as before. Since Proposition 4.7 with modified cones is also valid with only $\psi + l$ on the right side, it is possible to call ConeDecomp with $\varepsilon \cdot \varrho$ as a last parameter instead of $\varepsilon \cdot \varrho/2$, and the claim on the returned decomposition being a $(\frac{1}{3}, \varepsilon)$-star-decomposition will remain true. Because of this, the upper bounds on the boundary costs of cones returned by ConeCut will all be halved, which compensates for the factor 2 increase generated by the weaker bounds on cutting weakly concentric systems. Hence, if ConeDecomp is invoked with its last parameter doubled, the algorithm will still work flawlessly, while producing the original bounds on the cost of edges leaving the cones. Therefore, the proof of Lemma 3.2 and Theorem 3.5 will require no alteration at all, and the main statements of the article will be true in their original form.

To sum it up, we have successfully altered the star decomposition algorithm with the concept of modified cones, creating a version for which the claims of the paper on the average edge stretch and radius of the returned spanning tree really hold. This is important because it also proves the correctness of other algorithms of the area that rely on the correctness of the star decomposition algorithm, for example, the method of Abraham *et al* [3].

4 Further algorithms

Besides the star decomposition algorithm, two other methods for creating low-stretch spanning trees are discussed in the thesis.

One of these is the algorithm of Alon *et al.* [1], presented in Section 4.1. This algorithm is one of the very first methods introduced for developing spanning trees with low stretch, returning a tree with an average edge stretch of at most $O(e^{(\sqrt{\log n \cdot \log \log n}})$. An-

other important result in the article is that the best achievable upper bound on average edge stretch for any algorithm is of magnitude $O(\log n)$. The paper also presents one of the application areas of the topic in detail, and studies randomized algorithms for the problem.

The other presented method is the algorithm of Abraham *et al.* [3], which is an improved version of the star decomposition method. This paper is important because it introduces the new definition of cones that was used in Section 3.3 to correct the star decomposition algorithm, and also shows an example of more advanced algorithms based on the star decomposition method. The algorithm of the paper, which is discussed in Section 4.2, returns a tree with an average stretch of at most $O(\log n \cdot (\log \log n)^3)$.

4.1 The decomposition of Alon *et al.*

The first notable algorithm to develop low-stretch spanning trees was devised by Noga Alon, Richard M. Karp, David Peleg and Douglas West in their paper "A graph-theoretic game and its application to the k-server problem". At first sight, the paper seems to examine a different topic, a two player zero-sum game played on a weighted connected graph, in which one of the players chooses a spanning tree in the graph and the other chooses an edge, an then the amount they win is determined by the relationship between this tree and edge. However, the main result of the paper is in fact an algorithm that finds a low-stretch spanning tree, and then the bound on the average stretch of this tree is used to prove statements about the previous game.

The paper also contains as exhaustive discussion of various different related areas. It presents one application, the k-server problem in detail, describing how a strategy for the graph-theoretic game can be used to devise an efficient randomized on-line algorithm for the problem. The transformation method mentioned in Section 2.1, that converts the multigraph into a new one with at most $n \cdot (n+1)$ edges without increasing the magnitude of stretch, is also developed in this paper. Some special cases of the game are analyzed in detail, including grids, hypercubes, complete graphs, cycles and other examples. The paper discusses a wide range of other smaller claims that are connected to the topic in some way.

The main idea of the low-stretch tree algorithm in the paper is somewhat different than that of the star decomposition method, even though the dominant step of both algorithms is to establish a partition of the graph. The star decomposition method can be considered a top-down approach; after sorting the vertices into a components, it first decided the tree structure used to connect these components (a star), and then continues by developing a spanning tree in each of these smaller components. In contrast to this, the method of Alon *et al.* is a bottom-up algorithm: after sorting the nodes into components, it first creates a spanning tree in each of these components (that together form a forest in the original graph), and then contracts each component into a single node and continues to operate in the resulting graph, in order to develop a tree between these components. The next sections outline this algorithm, and briefly present the statements related to its results.

4.1.1 The algorithm

Although less sophisticated than the star decomposition algorithm, the method of Alon *et al.* still seems to be a fairly complex algorithm at first sight. However, while some of the related proofs are in fact complicated, the algorithm itself is not hard to understand.

First, we introduce some notation necessary to discuss the method. Throughout the algorithm, a number of fixed constants are used; these are defined as:

$$\varrho = \left\lceil \frac{3 \cdot \log n}{\log x} \right\rceil, \qquad \mu = 9 \cdot \varrho \cdot \log n, \qquad y = x \cdot \mu$$

where x is a parameter of the algorithm, set in the final version to be $e^{(c \cdot \sqrt{\log n \cdot \log \log n})}$ for some constant c. Also, at the beginning of the process, edge weights are first normalized such that the smallest edge length will be equal to 1 (that is, the length of each edge is divided by the minimal edge length), and then the edges of the graph are sorted into different subclasses according to their weights, where

$$E_i = \left\{ e \mid \ell(e) \in [y^{i-1}, y^i) \right\}.$$

The algorithm works in iterations. In each of these, the graph G_j is partitioned into some clusters, and edges with both endpoints in the same cluster are marked as "covered". At the end of the iteration, each of the clusters are contracted to a single vertex (this way, covered edges are disposed of), and this contracted graph G_{j+1} becomes the initial graph for the next iteration. In fact, the clusters developed in this process represent parts of the graph that will be connected by a shortest path spanning tree in the final spanning tree returned by the method.

A precise definition of the algorithm first requires us to introduce the notation E_i^j, meaning the set of edges that are in E_i and are still uncovered at the beginning of the iteration j (where $i, j \geq 1$). Then, the main idea of the algorithm when forming clusters in the j^{th} iteration, is to partition the vertices into subgraphs that have the following two properties:

- each of the developed clusters must have a spanning tree of radius at most y^{j+1}
- for the first j subclasses of edges, if $E_i \neq \emptyset$ $(1 \leq i \leq j)$, then at most a $\frac{1}{x}$ fragment of the edges of E_i^j can run between different clusters (and therefore remain uncovered by the end of the iteration). That is, $|E_i^{j+1}| \leq \frac{1}{x} \cdot |E_i^j|$

The concrete process of partitioning (one iteration of the algorithm) is divided into steps; in each step, one cluster is developed. In the beginning of a step, first a not yet sorted root vertex u is arbitrarily selected. Then, other unsorted vertices are divided into layers according to their unweighted distance from u (only those that are in the same connected component K in the graph of yet unsorted vertices), with $V(l)$ denoting the nodes at distance l from u. $E_i^j(l)$ will be used to denote the edges from E_i^j that connect two vertices in $V(l)$ or a vertex in $V(l-1)$ and a vertex in $V(l)$. The step proceeds by finding the minimal l^* value of l for which the inequality

$$\forall\ 1 \leq i \leq j, \quad |E_i^j(l+1)| \leq \frac{1}{x} \cdot |E_i^j(1) \cup E_i^j(2) \cup \ldots \cup E_i^j(l)|$$

holds. If there exists such an l^*, then the formed cluster will be $V(1) \cup V(2) \cup ... \cup V(l^*)$; if no such l exist, the cluster will be the entire connected component K. With this, the next cluster is generated, and the step is over; the process is continued with the next step if there is still a vertex not assigned to any cluster.

With the process of developing clusters in each iteration described separately, the entire algorithm can be represented with a relatively short pseudocode:

1. Set $j = 1$ and $G_j = G$.
2. Set $x = x(n)$, ϱ, μ, y and the edge classes E_i as defined previously.
3. While $\bigcup_i E_i \neq \emptyset$ do:
   (a) Partition the vertex set of G_j into clusters, as described above.
   (b) Construct a shortest path spanning tree from the root vertex in each of the developed clusters.
   (c) For each edge e of the constructed trees, add the corresponding edge of the original graph G_1 to the output tree T.
   (d) Construct the next multigraph G_{j+1} by contracting each cluster into a single vertex, discarding covered edges (connecting endpoints in the same cluster) from $\bigcup_i E_i$, and replacing each uncovered edge by a new edge connecting the corresponding contracted vertices.
   (e) Set $j = j + 1$.

The main idea behind the algorithm can be explained intuitively by recalling the two properties of the clustering method. Since the radius of the clusters are bounded, the paths to other vertices from the root vertex are relatively short, and thus the shortest path spanning tree is a good choice for connecting the vertices in the cluster. Of course, the possibly longer edges between the clusters also have to be added to the spanning tree in the next iteration, but since these are only a $\frac{1}{x}$ fraction of the entire edge set, the amount they contribute to the average stretch is still controllable.

The subclasses of edges are required in this process to handle the case of weighted graphs. With this concept, in the first iteration, only the shortest edges E_1 are used, then in the second, the class of a bit longer edges E_2 is also considered, and so on. After we start to consider E_i, the number of uncovered edges in this class will be reduced by a factor of x in each following iteration, until the class finally becomes empty.

4.1.2 Claims of the article

The article contains a huge number of statements, related to various different areas in the topic. These include applications, randomized methods for developing spanning trees, and also the analysis of the problem on some special graphs like grids and hypercubes. Here we only mention the statements connected to the previous algorithm, and one further claim on the best reachable upper bound.

The first two lemmas on the algorithm establish that the clustering method in fact has the aforementioned properties.

Lemma 5.4 of [1] (Bound on E_i^j). *For every $1 \leq i \leq j$, the inequality $|E_i^j| \leq \frac{1}{x^{j-i}} \cdot |E_i|$ holds.*

Lemma 5.5 of [1] (Bound on cluster radius). *In iteration j, the radius of each cluster is bounded by y^{j+1}.*

The former lemma can easily be proven, using the inequality we applied as stopping condition when developing the next cluster. The latter is a more complicated statement; its proof requires a detailed analysis of the algorithm, the former lemma, and also the fact that the number of edges is at most $n \cdot (n+1)$, ensured by applying the transformation algorithm presented.

With these lemmas, the main statement on the stretch can be proven.

Lemma 5.6 of [1] (Bound on the total stretch). *For the tree T returned by the algorithm, $\sum_{e \in E} \textit{E-stretch}_T(e) \leq 4x^2 \cdot \mu^{\varrho+1} \cdot |E|$.*

This lemma and our parameter definitions imply

$$\text{ave-E-stretch}(G)_T = 4x^2 \cdot \mu^{\varrho+1} \leq 4x^2 \cdot \left(\frac{27 \cdot log^2 n}{\log n}\right)^{1+\frac{3 \cdot \log n}{\log x}}.$$

which can optimized by selecting $x = e^{(c \cdot \sqrt{\log n \cdot \log\log n})}$ for some constant c. With this x, the upper bound we get is also $e^{\left(O(\sqrt{\log n \cdot \log\log n})\right)}$. Therefore, the decomposition method returns a spanning tree with an average stretch value of at most $e^{\left(O(\sqrt{\log n \cdot \log\log n})\right)}$.

Although not related to the presented decomposition method, the article also proves the statement that the best magnitude of upper bound on average stretch that can be shown for any algorithm is $\Omega(\log n)$. This is shown through a result introduced earlier in extremal graph theory, stating that for every n, there is an n-vertex graph with $2n$ edges in which every cycle is at least $a \cdot \log n$ long [17]. Obviously, for any spanning tree in this graph, the stretch of non-tree edges is at least $a \cdot \log n - 1$, and since more than half of the edges are not included in the tree, the average at stretch is certainly more than $\frac{1}{2} \cdot (a \cdot \log n - 1)$, which is logarithmic in n. Thus:

Theorem 5.1 of [1] (The best possible upper bound is logarithmic). *Let A be any algorithm that returns a spanning tree. There exist a positive constant c such that, for all n, there exist an n-vertex graph G such that for the tree T returned by A,*

$$\textit{ave-E-stretch}(G)_T \geq c \cdot \log n.$$

4.2 Star decomposition with queues

Another important result in the area is the improved version of the star decomposition method, described in the paper of Ittai Abraham, Yair Bartal and Ofer Neiman, titled "Nearly tight low stretch spanning trees" [3]. The algorithm applies the recursive star

decomposition algorithm of [2], with a major modification and also some minor changes. The main difference is the introduction of queues: the algorithm of Abraham *et al.* assigns priorities to the vertices in the graph, depending on their importance in the process of achieving low stretch. Based on this priority, while creating the star decompositions, the algorithm tries to include a shorter path in the tree for more important vertices. This alteration was enough for the authors to prove a bound of $O(\log n \cdot (\log \log n)^3)$ on the average edge stretch of the returned spanning tree, which is tighter than the one presented in the paper of Elkin *et al.*

The paper of Abraham *et al.* use the new definition of cones, that has already been analyzed in detail in Section 3.3. As already proven there, the bound on the stretch of the original star decomposition method does hold with this new definition, so the improved version of Abraham can indeed rely on the presented claims about the original star decomposition method.

The article also introduces a different, more advanced randomized method for developing spanning trees, for which various other claims are proven, but this version is not discussed in the thesis.

In this thesis, we assumed that the input graph is connected; therefore, claims that are stated in Abraham's paper for a connected component of the graph are declared here for the entire graph.

4.2.1 Concept

The main idea of the authors is to refine the parameters of the original star decomposition algorithm to yield a better stretch value, and then modify the decomposition method to solve the problems arising from their refinements.

In the case of the original star decomposition algorithm (the improved version in Section 3.1.4), when proving that the average stretch is in $O(\log^2 n \cdot \log \log n)$, the total stretch on a certain recursion level was bounded by the product of two things: the radius increase compared to the original recursion level and the cost of edges sorted into different components at the current recursion level. The radius increase (or "radius stretch") was bounded by $O(1)$ in the main theorem (Theorem 3.5) of the paper (specifically, by $2 \cdot e$), and the cost of edges is bounded in Lemma 3.2 about the star decomposition. Then, when summing this for all recursion levels, the cost of edges in different components can be bounded by a $O(m \cdot (\log^2 n \log \log n))$ factor, which stays the same magnitude when multiplied by the $O(1)$ radius stretch, and when divided by the number of edges, yields the mentioned $O(\log^2 n \log \log n)$ average edge stretch bound of [2].

Abraham *et al.* noted that, when showing that the cost of edges add up to $O(m \cdot (\log^2 n \log \log n))$, the paper in fact shows that the sum is $O(m \cdot (\log n \log \log n) \cdot \frac{1}{\beta})$, where β (denoted by ε in Abraham's paper) is one of the main parameters of the original star composition algorithm, set to $\beta = O\left(\frac{1}{\log n}\right)$ by Elkin *et al.* The main idea of Abraham is to modify this parameter to $\beta \approx \frac{1}{\log \log n}$, therefore modifying the sum of edge costs to $O(m \cdot (\log n \cdot (\log \log n)^2))$. The problem with this approach is that if $\beta \approx \frac{1}{\log \log n}$, then this will change the radius stretch that was $O(1)$ in the original case: an attempt to bound the radius in the original way with this β will result in a super logarithmic radius increase in the spanning tree, therefore, the product of the cost of edges and the radius increase

will be of even larger magnitude than before.

To solve this problem, the authors propose an entirely new approach to bound the radius stretch of their algorithm. They introduce the notion of queues, which is essentially a method to pass extra information about the properties of the present decomposition to the next recursion level. With this new concept, Abraham *et al.* successfully bound the radius stretch of their modified star decomposition algorithm by a $O(\log \log n)$ factor. Since their method use $\beta \approx \frac{1}{\log \log n}$, the sum of edge costs for this method is $O(m \cdot (\log n \cdot (\log \log n)^2))$, which, when multiplied by their radius increase of $O(\log \log n)$ yields a $O(m \cdot (\log n \cdot (\log \log n)^3))$, bound on the total stretch, or a $O(\log n \cdot (\log \log n)^3)$, bound on the average edge stretch. Therefore, with a slight alteration of the original parameters and the introduction of queues, the authors have successfully created a version of the star decomposition algorithm which is significantly stronger in terms of asymptotic edge stretch bound.

4.2.2 Introducing queues

As mentioned, the main improvement in the algorithm of Abraham *et al.* is the usage of queues. When calling the main function of the recursive algorithm (named LowStretchTree previously in [2] and hierarchical-star-partition in this paper), besides the input graph G and the central node x_0, another input parameter Q is added. This $Q = (z_1, z_2, ..., z_{n-1})$ is a queue containing all the vertices of the input graph except for the central vertex; that is, Q is an arbitrary ordering of the $V - \{x_0\}$ vertices of the graph. This Q is also passed on to the method creating the star decomposition of the input graph, denoted StarDecomp previously and star-partition in this article; this method, besides returning the $\{V_0, ..., V_k\}$ components and $(y_1, x_1), ..., (y_k, x_k)$ bridge edges of the developed star decomposition as before, also returns a set of queues $\{Q_0, ..., Q_k\}$. Each Q_i of these queues contains a permutation of the vertices $V_i - \{x_i\}$; that is, apart from creating the star decomposition, the star-partition method also splits the nodes in the input queue, arranging the nodes of V_i in the queue Q_i (except for the chosen anchor vertex) in some order.

Intuitively, these queues are used to set the priority of the vertices in the graph. If a node v appears at the beginning of the queue, the algorithm will consider it more important to include a short path to node v in the spanning tree. As a special case, to the first node of the queue, the shortest path will be included in the tree. The basic idea behind these priority lists is that when a graph is decomposed into a star, and then later a spanning tree will be developed in its central component, then some nodes of the central component may be more important to reach through a short path than others. For example, if some node y_i is the portal to a component V_i that contains a large number of nodes, then finding a short path to y_i might be more important than to other vertices, because if the path in the spanning tree to y_i is long, than the stretch value of all vertices in V_i will be bigger, which, due to their large number, will have a significant impact on the average stretch. Therefore, based on different properties of the components developed in the star decomposition, it can be beneficial to organize the nodes of the central component according to their priority.

Since at the point of creating the decomposition, we do not have much information about the non-central components, the subqueues corresponding to these components is

not rearranged at all. For each $i > 0$, the queue Q_i is simply defined as the restriction of the input queue Q on $V_i - \{x_i\}$. However, the queue Q_0 is defined in a more complicated way. First, we add the node z_1 or the portal y_i that is chosen on a shortest path to z_1 to Q_0, where z_1 is the first node in the queue. This ensures that the shortest path to z_1 is included in the tree; this can be proven recursively, if $z_1 \in V_0$ then trivially, and if y_i, then since y_i will be the first node in Q_0 and z_1 will be the first in Q_i. Then, the rest of Q_0 is merged from three different queues, in a way that if a node v is in the i^{th} place of any of these queues, then v is guaranteed to be in one of the first $3i$ elements of the final Q_0 version. The three queues correspond to different reasons to assign high priority to a node in the central component; this way, if v is considered important for any of these reasons, it is also certainly assigned a relatively high priority in Q_0. The three queues are the following:

- $\boldsymbol{Q_0^{(ball)}}$: This is simply the restriction of Q on V_0. If any of the nodes that falls into V_0 has a high priority in the input queue, then it will be located at the front of $Q_0^{(ball)}$, and thus also relatively at the front of Q_0.
- $\boldsymbol{Q_0^{(reg)}}$: This queue contains the portal vertices y_i, their order determined by the input priority of the highest priority vertex in V_i. The aim of this queue is to guarantee that to any vertex in V_i, the part of the path in the central ball is not stretched too much.
- $\boldsymbol{Q_0^{(fat)}}$: This queue contains the portals y_i, for which the corresponding cones contains a relatively large number of vertices. For a bigger cone, even if its nodes have lower priority, it is important to find a short path to the portal vertex in the central ball, since this part of the path has a large effect on the average edge stretch.

The precise definition of the method for creating the queues can be seen below. The operation of enqueueing an element v into a queue Q, used in the description, means placing the element at the end of the queue.

1. Creating the queues $Q_0^{(ball)}, Q_0^{(reg)}, Q_0^{(fat)}, Q_1, ..., Q_k$:
   (a) For $i = 1, ..., n-1$:
      i. If $z_i \in V_0$ then enqueue z_i into $Q_0^{(ball)}$;
      ii. Otherwise let $l \geq 1$ be such that $z_i \in V_l$:
         - If $z_i \neq x_l$ then enqueue z_i into Q_l.
         - If $y_l \notin Q_0^{(reg)}$ then enqueue y_l into $Q_0^{(reg)}$.
         - If $|V_l \cap \{z_1, ..., z_i\}| > \sqrt{i}$ and $y_l \notin Q_0^{(fat)}$ then enqueue y_l into $Q_0^{(fat)}$.
2. Creating the queue Q_0:
   (a) Denote the $Q_0^{(ball)} = z_1^1, .., z_{k_1}^1$, $Q_0^{(fat)} = z_2^1, .., z_{k_2}^2$, $Q_0^{(reg)} = z_3^1, .., z_{k_3}^3$.
   (b) Create Q_0 by interleaving the three queues $Q_0^{(ball)}, Q_0^{(reg)}, Q_0^{(fat)}$ such that:
      - If $z_1 \in V_0$ then z_1 is the first element of Q_0. Otherwise y_1 is the first element of Q_0.

- For any $v \in V$, $l \in \{1, 2, 3\}$, $1 \leq i \leq k_l$, if $v = z_i^l$ then v is in the first $3i$ elements of Q_0.

Apart from the introduction of queues, the algorithm of Abraham *et al.* is almost identical to the star decomposition algorithm. A minor change is the development of the central component: while the original algorithm used the BallCut method for this purpose, this new algorithm simply chooses an r_0 value between $\frac{1}{16c} \cdot rad_G(x_0)$ and $\frac{1}{8c} \cdot rad_G(x_0)$, where c is some predefined constant, and sets $B(x_0, r_0)$ as the central component. Then, the ImpConeDecomp method of [2] is used to develop all the other components, just like in the improved version of the star decomposition algorithm, with only a slight modification. This modification is the selection of the first anchor vertex; while in the star decomposition algorithm, the next anchor vertex was always an arbitrary unsorted vertex of $BS(x_0, r_0)$, here, the first anchor vertex is chosen to be one that is on a shortest path to the first element of the queue, z_1. Then, all other anchor vertices are chosen arbitrarily like before (if z_1 happens to be in the central component, then this is not required; in this case, ImpConeDecomp can be used without any alteration).

Another minor modification is the terminating condition of the algorithm for recursively calling itself. This happened in the algorithm of Elkin when the input graph contained at most 2 nodes; however, these authors stopped the execution of their algorithm when the radius of the input graph from the central node falls below a predefined constant. When this happens, the shortest path from x_0 is calculated to all other vertices, and the tree consisting of these paths is returned.

Thus, the main difference between the improved StarDecomp of [2] and the star-partition of [3] is application of queues, with which the authors managed to prove a tighter bound on the radius of the developed spanning tree.

4.2.3 Claims about the algorithm

The paper contains numerous smaller claims about the algorithm; discussing them all is beyond the scope of the thesis, but the most important one is worth mentioning specifically. This is in fact a more powerful version of property *b)* of (δ, ε)-star-decompositions. This claim states that when creating a star decomposition with our algorithm, the inequality

$$rad_{V_0}(x_0) + \mathrm{d}(y_j, x_j) + rad_{V_j}(x_j) \leq (1 + \varepsilon) \cdot rad_V(x_0),$$

will hold for each $j \geq 1$. This claim is important because it is used at multiple points in the proofs of the main theorems of the paper, but it can in fact only be proven in the case of unweighted graphs. Because of this, the authors had to find a different approach to prove their main theorems for weighted graphs, which is presented in the paper's appendix. They devised a method for adding imaginary points to their graph in a way that the above inequality does hold for the resulting graph, and therefore the same proof can be applied for this new graph as in the case of unweighted graphs, thus bounding the average edge stretch of the chosen spanning tree in the new graph by $O(\log n \cdot (\log \log n)^3)$. Then the authors show that the average edge stretch of the tree developed in the original graph is the same magnitude as that of the tree in the new graph, this way validating their main theorem for weighted graphs.

A more important statement of the paper is the one that bounds the radius increase in the developed spanning tree. This is the statement showing that with the presented usage of queues, the radius of the spanning tree is at most $O(\log\log n)$ times the radius of the graph, even with the β parameter set to the magnitude $\log\log n$. The proof of this claim is built on different properties of the previously discussed queues, and is not complicated at all, but remarkably long, divided into numerous different cases.

Lemma 4. of [3] (Bounding the radius stretch). *Let $G = (V, E)$ be a connected graph, let $x_0 \in V$ and $Q = (z_1, ..., z_{n-1})$ be any ordering of $V - \{x_0\}$. Let T be the spanning tree of G returned by the algorithm hierarchical-star-partition(G, x_0, Q) with parameter $\beta = \frac{1}{170 \cdot c \cdot \log\log n}$ and $c = 2^{16}$. Then*

$$d_T(x_0, z_i) \leq \begin{cases} d_G(x_0, z_i) & \text{if } i = 1 \\ i \cdot rad_G(x_0) & \text{if } 1 < i < c \\ c \cdot \log\log i \cdot rad_G(x_0) & \text{otherwise} \end{cases}$$

Finally, the statement of the paper on the average edge stretch is cited, although its content has already been discussed.

Corollary 6. of [3] (Average edge stretch of the algorithm). *For any weighted graph $G = (V, E)$, invoking the hierarchical-star-partition algorithm on G where in star-partition we use the*

$$ImpConeDecomp\left(G, BS(x_0, r_0), \frac{rad_G(x_0)}{\log\log n}, \log\log n, m\right)$$

of [2], then we get a single spanning tree T such that

$$ave\text{-}E\text{-}stretch(G)_T \leq O(\log n \cdot (\log\log n)^3).$$

The running time is $O(m \cdot \log n)$ if G is unweighted and $O(m \cdot \log n + n \cdot \log^2 n)$ if G is weighted.

5 Empiric results

One of the main contributions of this thesis is the empirical comparison of the presented algorithms, which, to our knowledge, has not been carried out before. For this, all three algorithms were implemented in the language C++, then executed on a set of input graphs, and the stretch values of the returned spanning trees were calculated. This section presents the results of these experiments. For the sake of reproducibility, the data sheets of results and the source codes of the algorithms are available at *http://www.cs.elte.hu/∼ozdsnn.*

Since different versions of the star decomposition algorithm were discussed, in the first phase, these variants were compared to each other. There were two main differences between the examined versions: the first was whether they use the original definitions of cones or the notion of modified cones introduced to correct the mistakes in the article, and the second is whether they apply the original decomposition method or the improved version briefly presented in Section 3.1.4. The results of these comparisons are shown in Section 5.2.

In the second phase, the different algorithms were compared to each other on various different graphs. In this phase, only one version of the star decomposition algorithm is studied, and it is compared to the decomposition of Alon from Section 4.1, the improved star decomposition method from Section 4.2, and a simple minimum cost spanning tree algorithm. These results are described in Section 5.3.

5.1 Evaluation framework

Before showing the actual results, this section briefly discusses some details of the implementation and evaluation process.

The algorithms have been implemented using the data structures and other features of the LEMON library; the section shortly introduces this framework. The library was applied because it contains effective implementations of various data structures and operations related to graph theory. Since our algorithms were defined in detail and on many occasions the functions of LEMON could be applied, there were only a few parts left that had any degree of freedom in implementation; however, these parts were also written with the intention of minimizing runtime.

Our algorithms have been compared on a set of inputs consisting of various graphs of different size and kind. Before presenting the results obtained, this section also describes the few random graph models that were used to generate the series of input graphs for testing the algorithms.

5.1.1 The LEMON library

The investigated algorithms were implemented using the classes and algorithms provided by the LEMON graph library [18]. LEMON (abbreviating Library for Efficient Modeling and Optimization in Networks) is an open source library with the implementation of various data structures and algorithms in the fields of graph theory and combinatorics, in C++ language. The library is used in a number of different research projects.

LEMON has various features that are very useful when implementing graph algorithms of the more complex kind. The graph data structure in LEMON is designed in a way that makes most of the basic graph operations very effective. Besides the most trivial operations, some more complicated ones are also included, such as the contraction of two vertices, which happens to be extremely useful in our case, since it is regularly executed in the star decomposition algorithm.

Our implementation also makes use of the graph algorithms in the library. LEMON contains an effective implementation of Dijkstra's algorithm using Fibonacci heaps, which is applied at various points in our methods. It also includes a minimum spanning tree al-

gorithm (Kruskal's algorithm), therefore it requires no extra effort to compare our stretch values to those of a minimum spanning tree.

5.1.2 Graph models examined

To compare the results of the algorithms in practice, different kinds of input graphs models were used. For each graph model, a series of graphs with different size have been generated and examined, in order to obtain a picture of the behavior of the algorithms on the model in question. This subsection presents the graph constructions used.

The first model is the well-known $G(n, p)$ random graph model introduced by Erdős and Rényi [19]. In this model, a random graph of n nodes is created in the following way: we take n nodes, and between every pair of nodes, an edge is drawn with probability p, independently from each other. This model is the most widely used for creating random graphs. In our experiments, for some fixed value of p, a series of graphs with different size (different n) are examined.

We also investigate a slightly modified version of the previous series of graphs, since in a series of graphs with a fix p value, the expected number of edges for n nodes is $\binom{n}{2} \cdot p = \Theta(n^2)$. However, in many application areas, the input graphs tend to be of sparse nature, that is, they often contain a number of edges linear in the number of nodes. To generate such graphs with the Erdős-Rényi model, instead of fixing the parameter p, the expected degree of the nodes of the graph is fixed throughout the series. With this, the $G(n, k)$ model can be introduced. In a series of this kind, the value of n is changing, and p is always set so that the expected number of edges will be k; that is, for each n, $p = \frac{k}{n-1}$ is chosen.

For both models, weights for the edges are chosen uniformly from the interval (0,1), independently from each other.

Also, the above models have to be slightly altered to ensure that the generated graph is always a connected one. Although the method of generating new graphs until the graph is connected might seem as a solution to this problem, but is in fact not applicable in practice, due to the long runtime for parameter combinations that have a small chance of the generated graph being connected. Thus, we apply a different method: before creating the edges of the graph, we generate a Hamiltonian cycle in the graph, and then reduce the p or k parameter of the process to keep the expected number of edges the same as in the original case. This method ensures that the graph is connected, with the required expected number of edges, and we believe it has little to no effect on the stretch of the developed spanning tree.

The other inspected random graph model will be the geometric graph model, denoted $U(n, d)$. The parameters of this model are a positive integer n, also denoting the number of nodes in the graph, and a positive real $d \in [0, \sqrt{2}]$ distance. In this model, each node of the graph corresponds to a random point chosen uniformly from the square $[0, 1] \times [0, 1]$ on the plane. Each pair of nodes in the graph are connected by an edge exactly if their Euclidean distance is at most d, and if they are connected, the weight of the edge is the distance of the nodes.

It can be seen that with increasing n for a fixed d, the expected number of edges in the graph will be quadratic in n in this model, too. Therefore, an alternative model $U(n, k)$ is also introduced in this case, with a linear number of edges. In this version, we

start from an empty graph, and at every step we choose the pair of vertices in the graph with minimal distance among the pairs not yet connected by an edge, and draw an edge between them. This procedure is continued until we reach an average degree of k.

We also have to modify the geometric graph model to ensure connectivity. However, in this case, adding a Hamiltonian cycle may result in more significant changes in the structure of the graph, therefore we apply another method. If the graph generated by the model is connected, then no alteration is required. Otherwise, new edges are added to the graph until the graph becomes connected, in the following way: at every step, from the pairs of nodes in different components, we choose the one with minimal distance, and connect them with an edge. Note that since the number of edges added is at most $n-1$, the process will not change the magnitude of the number of edges in the graph; it will still remain linear in the $U(n,k)$ case.

5.2 Different versions of star decomposition

In the first set of experiments, different versions of the star decomposition algorithm were compared. Each graph series in this experiment consisted of five different graphs, of size $n = (100, 200, 400, 800, 1600)$. For all four graph models, I considered three different parameter values: for the $G(n,p)$ model, $p \in \{0.05, 0.1, 0.2\}$, for the $U(n,d)$ model, $d \in \{0.05, 0.1, 0.2\}$, and for the $G(n,k)$ and $U(n,k)$ models, $k \in \{6, 10, 14\}$. For every parameter combination in each of these series, five different graph instances were randomly generated, and all the algorithms were executed for these five inputs. The stretch value returned by an algorithm for a fixed parameter combination was considered to be the average of these five stretch values.

Altogether, there are four available variants of the star decomposition algorithm to be examined, based on the following two choices:

- we can either use the basic version of the algorithm, or the improved version from Section 3.1.4
- we can either use the original version presented in the paper (with the original definition of cones and some invalid proofs), or the modified version of our Section 3.3 that uses modified cones to validate the proofs

First, we compare the basic algorithm with the improved version, both in the case of the original and the modified method. The main conclusion of these experiments was that the improved version does not yield significantly better results than the original one, at least not for graphs of the investigated size. In fact, on many occasions, the two versions of the algorithm returned exactly the same spanning tree; this happens when at every point in the algorithm, the slightly stricter conditions applied by the improved cone decomposition method also hold for the ones formed by the basic ConeCut, and therefore the two procedures develop the same cones.

The results of the comparison can be seen in *Table 1.* For each graph model, the table shows the percentage of average improvement in stretch for all the graphs of the model (that is, the average taken on all three graph series generated in the model). The improvement percentages are shown for both the original and the modified version, for

	Original version		Modified version	
Model	Edge str.	Path str.	Edge str.	Path str.
$G(n,p)$	0.00 %	0.00 %	0.33 %	0.09 %
$G(n,k)$	0.34 %	0.03 %	0.29 %	0.09 %
$U(n,d)$	0.04 %	0.00 %	2.69 %	0.65 %
$U(n,k)$	0.09 %	0.00 %	2.96 %	1.07 %

Table 1: *Improvements of average stretch in percentage when using ImpConeDecomp, as described in Section 3.1.4*

both average path stretch and average edge stretch. It can be seen that the average improvement in stretch is less than 3% even in the best case, and usually much smaller than that. Generally, the method seems to achieve better results in the case of the modified algorithm than in the case of the original one; however, the improvement does not seem to be significant for this version either.

Then, we compare the original and the modified versions of the algorithm. Because we previously concluded that the improved version is not significantly better, we only conduct this analysis with the basic version of the algorithm. The average stretch values obtained from these experiments are shown in *Table 2*. The table shows the percentage of improvement in stretch if we use the modified variant of the method instead of the original one; therefore, positive values indicate that the modified algorithm was more successful, and negative ones indicate the original one being superior. We can see that the differences seem to be much more noteworthy in this case, with the largest value being almost 20%. It is also clear from the table that the original algorithm appears to be more effective on the geometric graph model, while the modified version seems to be slightly better in the case of Erdős-Rényi random graphs. This suggests that our experiments might not end up showing that one of the algorithms is absolutely superior; instead, we may find that for particular graph models, different algorithms are the best choice.

Model	Edge str.	Path str.
$G(n,p)$	3.69 %	4.62 %
$G(n,k)$	3.33 %	4.53 %
$U(n,d)$	-18.65 %	-4.26 %
$U(n,k)$	-9.64 %	-1.24 %

Table 2: *Improvement of average stretch if we use modified version instead of the original one*

Since the result obtained from comparing the different versions are inconclusive, it is not straightforward to decide which version to use in the next set of experiments, when we compare the star decomposition algorithm to other approaches. Because the improved version does not seem to be considerably better than the basic version, but has a higher

runtime, I chose to use the basic version in the following experiments. From the original variant and the modified one, the original version will be used for further investigations, even though no valid bounds are proven for this version. The main motivation behind this is that from the previous tests, it seems that the two versions are effective on different kinds of graphs, and an algorithm based on star decomposition with modified cones (namely, the algorithm of Abraham *et al.*) is already involved in the next phase; therefore, we would also like to include a variant based on the original definition of cones.

5.3 Comparing the algorithms

When comparing the different algorithms, a larger set of input graphs were used than in the previous case. For these experiments, each graph series consisted of 11 graphs, for $n = (100, 150, 200, 300, 400, 600, 800, 1200, 1600, 2400, 3200)$. Four different series were generated for each model; for $G(n, p)$, the values $p \in \{0.02, 0.05, 0.1, 0.2\}$, for $U(n, d)$, the values $d \in \{0.02, 0.05, 0.1, 0.2\}$, and for the $G(n, k)$ and $U(n, k)$ models, the values $k \in \{6, 8, 10, 12\}$. As in the first set of experiments, each stretch value corresponds to the average of the stretch of five different graphs generated with the specified parameters.

The results for each of the graph models in this section are presented in a figure, consisting of four diagrams, corresponding to the parameters applied for the model. These diagrams illustrate the stretch value for the particular graph series, presenting the stretch obtained by the algorithms as a function of n. Each diagram shows four different data plots, belonging to the four algorithms executed and compared on the input graphs. The four algorithms (and their abbreviations used on the plots) are:

- **SD:** the star decomposition algorithm (its original version), presented in Section 3
- **AL:** the decomposition algorithm of Alon *et al.*, shown in Section 4.1
- **AB:** the improved star decomposition method of Abraham, shown in Section 4.2
- **KR:** the Kruskal algorithm for developing a minimum spanning tree

As both the number of nodes in a graph series and the stretch returned by the algorithms vary in magnitude, both axises of the diagrams have a logarithmic scale.

5.3.1 Average edge stretch

The statements of the papers describing the algorithms are all claims on average edge stretch of the returned spanning tree, bounding it by some function of the number of nodes. However, it is also very important to see if the methods with lower proven bound are really better, or are in fact inferior in practical use to another algorithm with a much weaker upper bound. This section presents the average edge stretch values obtained by the algorithms on the described set of graphs.

First, the results for the $G(n, p)$ model are discussed. The diagrams for this model are presented on *Figure 4*.

The figure shows that for the $G(n, p)$ model, the star decomposition algorithm can basically be considered the most successful method, only outperformed on some of the

smallest graphs, usually by the Kruskal algorithm. Even for the largest graphs, the algorithm managed to return a spanning tree with an average edge stretch around 2 to 2.5, and the Kruskal algorithm also found a tree of stretch around 3. The two algorithms also look promising because their stretch seems to be almost completely independent from the size of the input graph, therefore, we can hope that they would remain effective for even larger instances. While the method of Abraham also seems competent for the smallest p, its stretch grows rapidly for the other three parameter values. Finally, the algorithm of Alon is significantly weaker than the others in all of the cases.

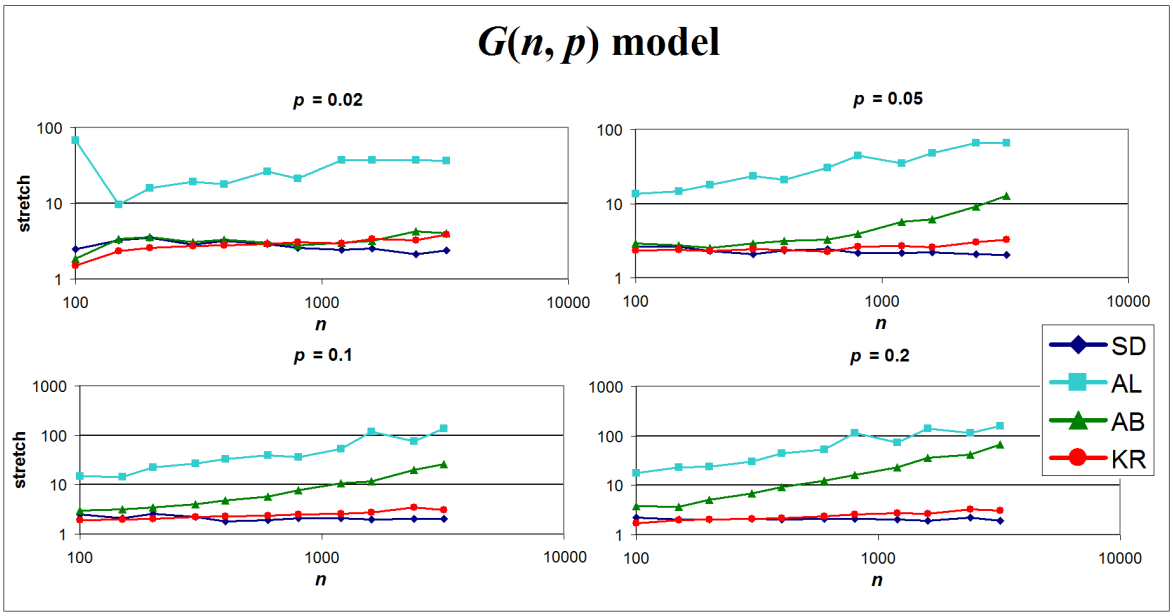

Figure 4: *Average edge stretch diagrams for the $G(n, p)$ model.*

The results for the $G(n, k)$ model can be seen on *Figure 5.* These diagrams seem slightly different than the previous ones; the most notable change is that the algorithm of Abraham also seems to yield as good results as the Kruskal and the star decomposition algorithm, even surpassing them eventually and giving the best stretch for the largest graphs for all four parameter values. However, the difference between the algorithms is not significant in fact; the method of Abraham only outperforms the star decomposition algorithm on the largest graphs by 3%, 6%, 7% and 1% for $k = (6, 8, 10, 12)$, respectively. Still, it is important to note that while Abraham's star decomposition with queues is not a viable option for the dense graphs of the $G(n, p)$ model, it is in fact very effective for sparse graph constructions.

On the other hand, we can also observe that the Kruskal algorithm and the star decomposition algorithm produced weaker results than in the case of dense graphs. While the average of stretch values for all graphs in the $G(n, p)$ model was 2.31 for the star decomposition algorithm and 2.58 for the Kruskal algorithm, these values are 3.20 and 3.23 respectively in the case of the $G(n, k)$ model. It is in fact fairly hard to decide if one should intuitively expect an algorithm to find better or worse trees in dense input graphs; while the higher number of options can help some algorithms make favorable choices when selecting a new edge of the tree, it is also more difficult to find the optimal decision from a

higher number of possibilities. A thorough investigation for the reasons of the algorithms giving better results in sparse or dense graphs is beyond the scope of this thesis.

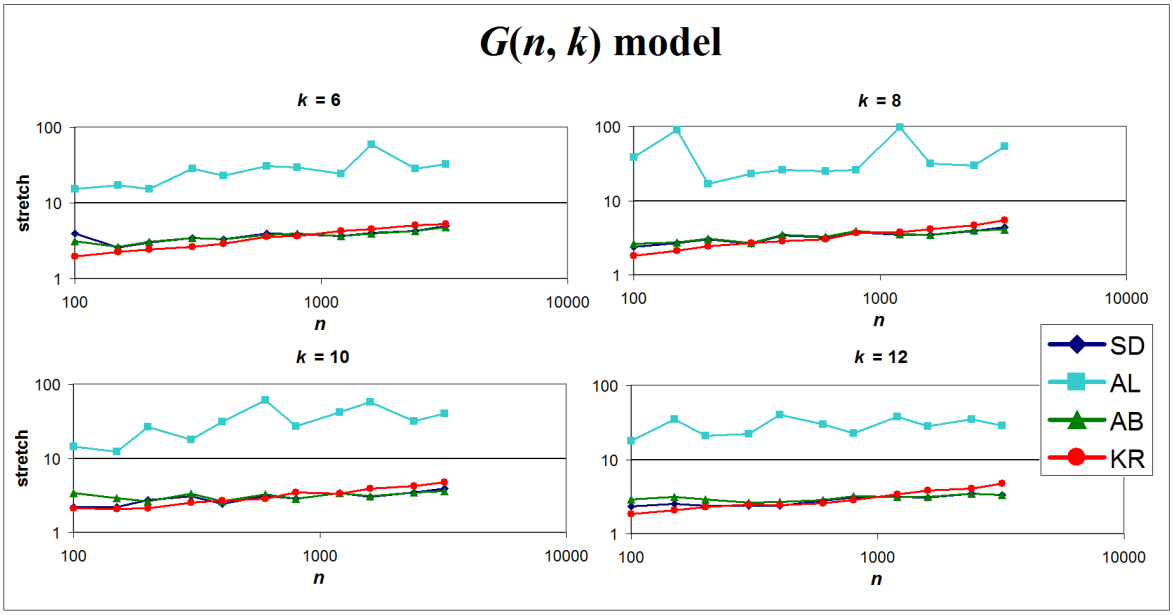

Figure 5: *Average edge stretch diagrams for the $G(n, k)$ model.*

Figure 6 shows the performance of the algorithms on graphs of the $U(n, d)$ model. It is clear at first sight that these diagrams look significantly different than the previous ones. The Kruskal algorithm seems to be the unquestionable winner on instances of this kind, with the other algorithms being considerably weaker for smaller and only slightly weaker for larger d values. However, even this method gives weaker results than in the case of the Erdős-Rényi models: the stretch value for the largest graphs is over 5 for the $d \in \{0.05, 0.1, 0.2, \}$ cases. This points out another important difference between this model and the previous ones: the stretch functions of the algorithms appear to grow much faster than in the previous cases. If this tendency continues for larger graphs, then it is questionable whether any of the investigated methods can find a spanning tree with acceptable stretch for graphs of much bigger size.

Also, note that the other three algorithms show very similar behavior on each of the four diagrams. Even the method of Alon, which gave very weak results previously, yields a stretch value close to those of the star decomposition variants.

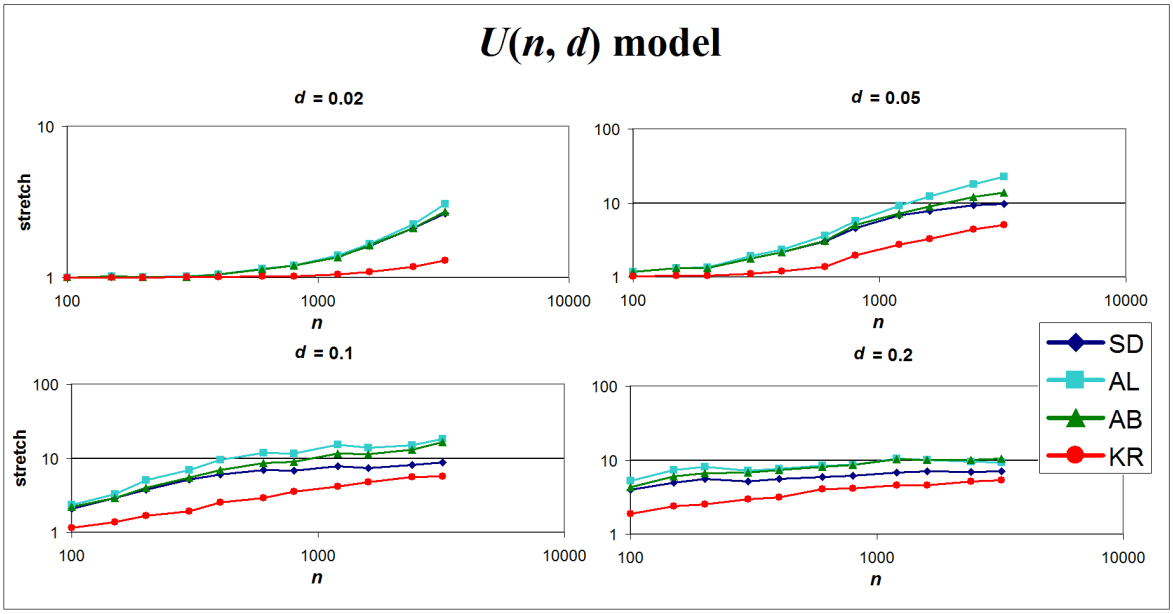

Figure 6: *Average edge stretch diagrams for the $U(n, d)$ model.*

Finally, the results of the $U(n, k)$ model are depicted on *Figure 7*. This diagram suggests that the Kruskal algorithm is in fact the best choice if the input graph was generated with the geometric graph model, regardless of the density of the graph. There seems to be no significant difference between the other tree methods when working on graphs of this kind.

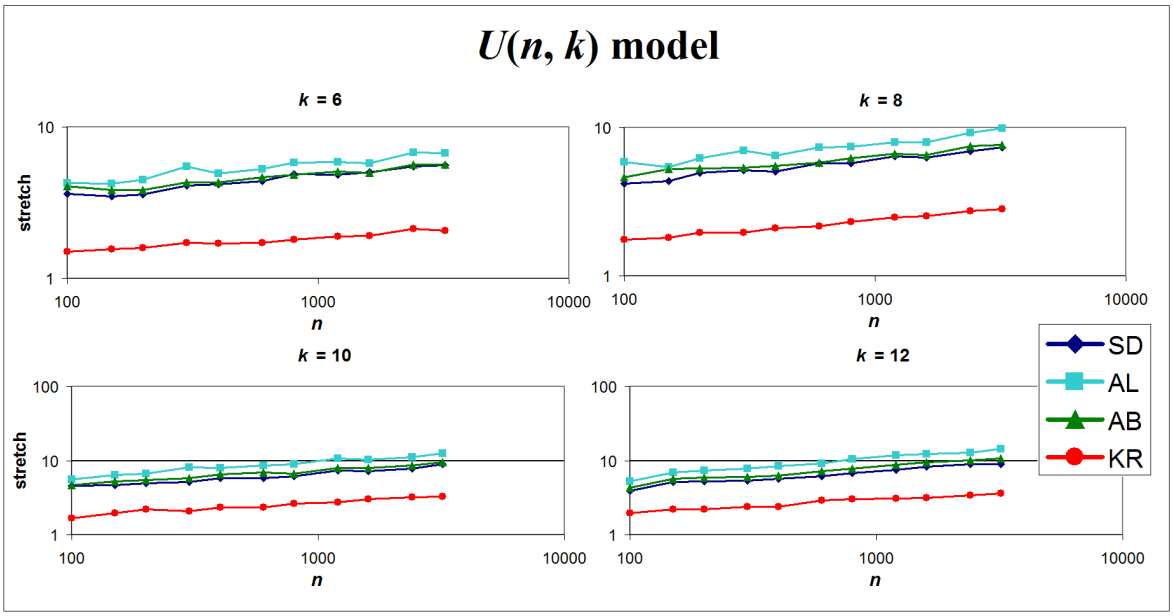

Figure 7: *Average edge stretch diagrams for the $U(n, k)$ model.*

Having examined the average edge stretch of the algorithms on the graph models, it is also interesting to investigate whether these results change significantly if we apply the

slightly modified edge stretch definition of Elkin *et al.* Recall that in [2], an alternative definition for edge stretch was introduced: instead of dividing the distance in the tree by the distance in the graph, the authors divided it by the length of the edge. While it was mentioned that their algorithm can be modified to operate using to the original definition of edge stretch, the statements about the star decomposition (and therefore also about the algorithm of Abraham *et al.*) were in fact stated for this new definition of edge stretch. Therefore, average edge stretch values according to this modified definition of stretch are also worth discussing.

Since graphs in the geometric model are Euclidean, the two definitions give the same stretch value in their case. On the other hand, there might be significant differences in the case of the $G(n, p)$ and $G(n, k)$ models. However, further experiments show that applying this different definition yields very similar results. In the case of the $G(n, k)$ model, even the stretch values returned by the algorithms were almost exactly the same as in the case of the original definition. In the $G(n, p)$ model, stretch values were often different, but the methods did not change relatively to each other; the algorithms that proved to be efficient with the original definition also gave good results with the new one, and the weaker algorithms remained weaker regardless of the change, for every value of p. However, note that with this different definition, stretch values below 1 can also be returned, which might be surprising at first when examining diagrams of this kind.

5.3.2 Average path stretch

Although the papers on the investigated algorithms do not include any claims on the average path stretch of the spanning trees returned, this metric can also be important for some applications, since it is the most natural way to measure how much the paths in a graph were expanded. This subsection gives an overview of the path stretch values obtained on the previously investigated set of graphs.

Figure 8 shows the diagrams for the average path stretch value achieved on the set of graphs in the $G(n, p)$ model. The average path stretch data plots in this case seem to be very similar to the average edge stretch plots. The star decomposition method gives the best results, followed closely by the Kruskal algorithm, and Abraham's method for $p = 0.02$. The method of Abraham for other p values and the decomposition of Alon for any p is considerably weaker than these. Even the magnitude of the path stretch values for the two stronger algorithms is the same as it was for their average edge stretch value.

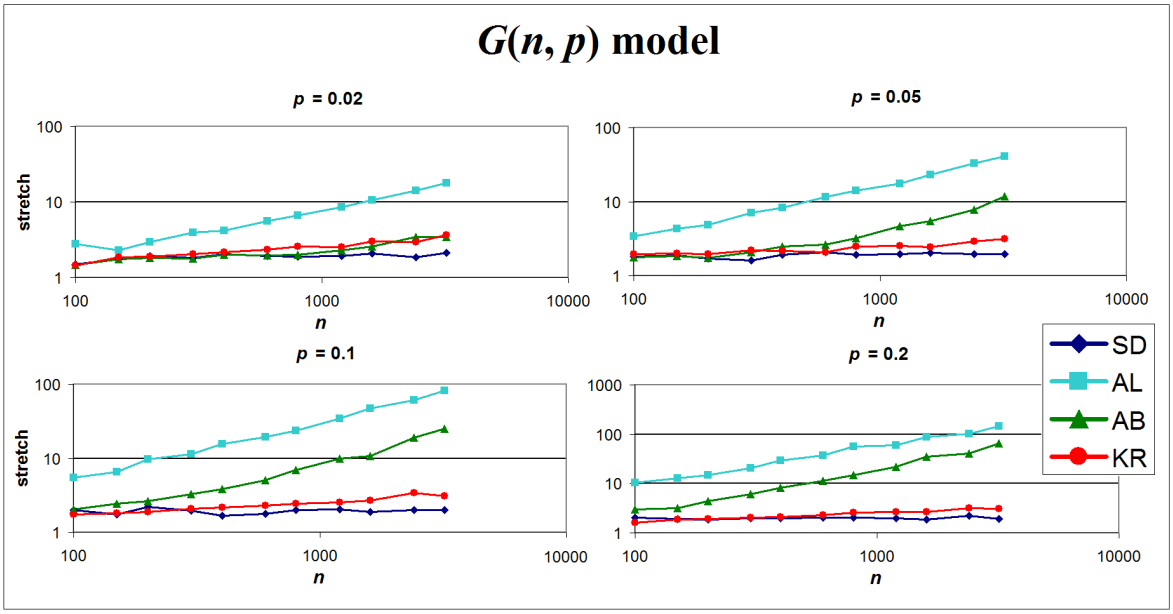

Figure 8: *Average path stretch diagrams for the $G(n, p)$ model.*

However, in the case of the $G(n, k)$ model, there is a significant difference, namely the behavior of the Kruskal algorithm. While this method was only slightly weaker than the star decomposition algorithm when measuring average edge stretch, here it seems to be notably worse than it. Also, if one compares the diagrams of *Figure 9* to those about the average edge stretch of the model, it is observable that the average path stretch values returned by the algorithms (even those of the Kruskal algorithm that became weaker related to the others) are much smaller than the average edge stretch values they previously achieved.

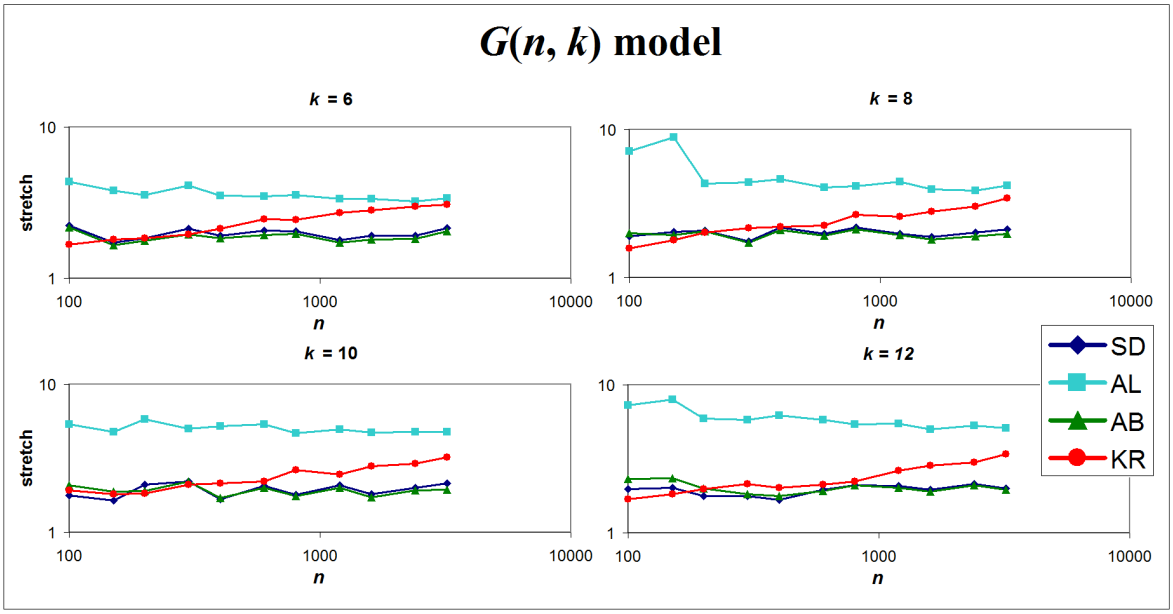

Figure 9: *Average path stretch diagrams for the $G(n, k)$ model.*

The average path stretch results for the $U(n, d)$ model are presented in *Figure 10*. This figure seems to be completely different from its average edge stretch counterpart; while the Kruskal algorithm achieved the best edge stretch value by far, it happens to return the weakest path stretch. If we examine the concrete stretch values, we also find in this case that the path stretch of the Kruskal algorithm is much smaller than its edge stretch value, but for the other three algorithms, the difference between the edge stretch and path stretch returned is a lot larger, and this results in the Kruskal algorithm becoming the worst in terms of average path stretch, even though it was the best in average edge stretch. All three other algorithms obtain a very good path stretch value; however, there is no considerable difference between the three.

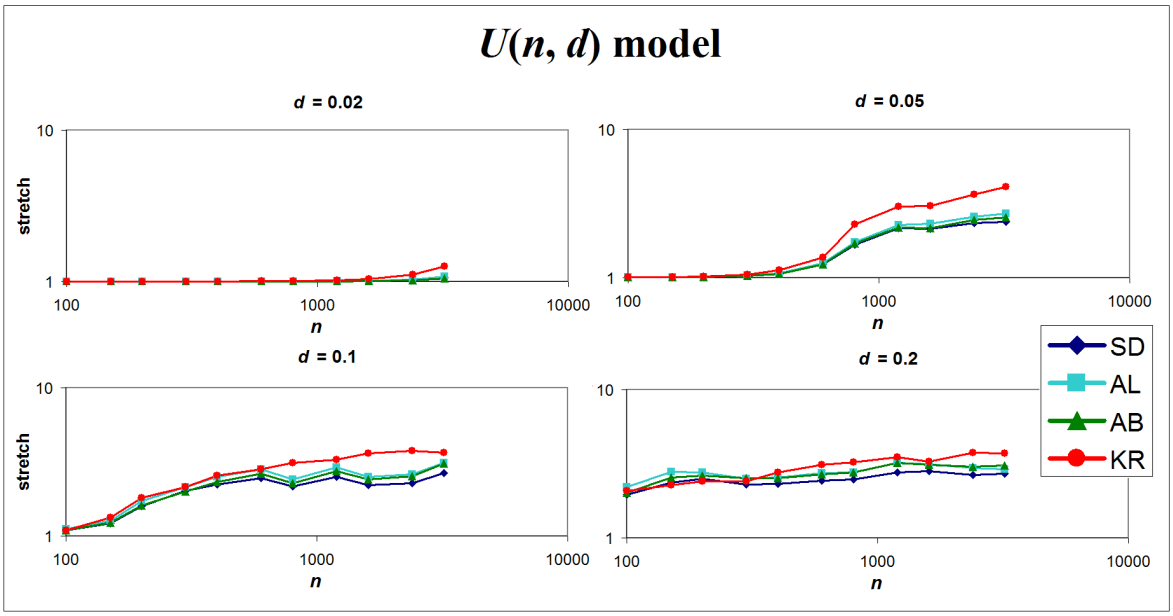

Figure 10: *Average path stretch diagrams for the $U(n, d)$ model.*

Finally, the diagrams of the $U(n, k)$ model are shown on *Figure 11*. This case shows the same phenomenon as the previous one; all the other algorithms yield a much smaller path stretch value than edge stretch value, thus the Kruskal algorithm becomes the weakest, even though it was the best by far in terms of edge stretch. Also, note that the difference between the path stretch of the other three algorithms is extremely small.

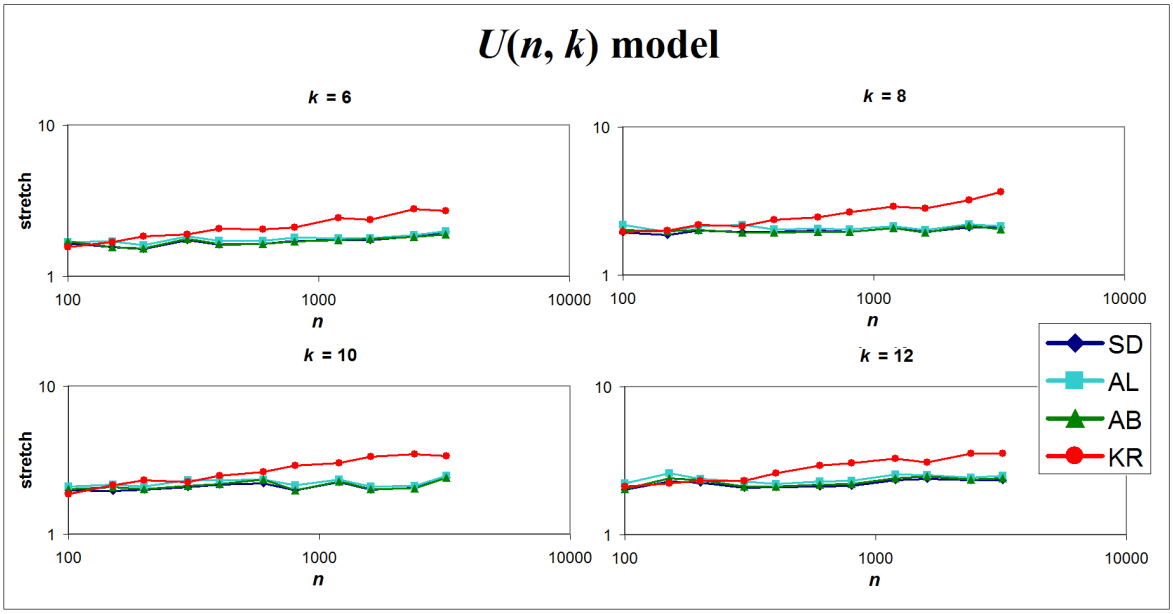

Figure 11: *Average path stretch diagrams for the $U(n, k)$ model.*

Thus, we have seen that the average path stretch of the algorithms can differ significantly from the average edge stretch they obtained, therefore, the optimal choice of algorithm for average path stretch might be different from that for average edge stretch. This is especially true in the case of the geometric graph model, where the Kruskal algorithm proved to be much stronger in terms of edge stretch than the three approaches discussed in the thesis, but all three happened achieve better average path stretch values than the minimum spanning tree algorithm.

5.4 Sublinear bounds for the Kruskal algorithm

Although the empiric results show that the Kruskal algorithm is often superior to all the other inspected methods, it is also important to investigate whether the average stretch of the minimum cost spanning tree can also be bounded by some sublinear function of the number of nodes. This section shows a construction of graphs (named "wheel graphs"), for which it is easy to see that both the average edge stretch and the average path stretch of the minimum spanning tree grows linearly in the number of nodes, and therefore shows that no sublinear bound can be proven on the Kruskal algorithm.

Let the wheel graph of size n be an undirected weighted graph with n nodes and $2 \cdot (n-1)$ edges, where the vertices v_1, v_2, ..., v_{n-1} form a cycle, and the last vertex v_n is connected to all the other vertices. Choose ε to be a positive real number, and assign weights to the edges in the following way. For all $i \in \{1, ..., n-2\}$, let the weight of edge (v_i, v_{i+1}) be $1 - \varepsilon$, and the weight of edge (v_{n-1}, v_1) be 1. Let the weight of edge (v_1, v_n) be $1 - \varepsilon$, and for all $i \in \{2, ..., n-1\}$, the weight of edge (v_i, v_n) be 1. An example for the construction is illustrated on *Figure 12*.

It is clear that the edges (v_1, v_n), (v_1, v_2), (v_2, v_3), ..., (v_3, v_{n-1}) form a minimum cost spanning tree in this graph, with a cost of $(n-1) \cdot (1-\varepsilon)$, and that this is the only

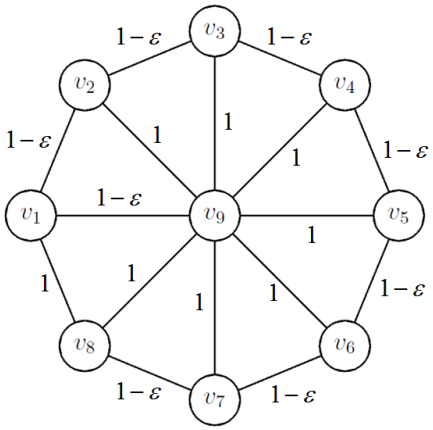

Figure 12: *Illustration of a wheel graph for* $n = 9$.

minimum spanning tree in the graph; therefore, the Kruskal algorithm will always return this spanning tree. We show that both the average edge stretch and the average path stretch in this spanning tree is linear in n, hence the average stretch ratio of the spanning trees returned by the Kruskal algorithm is at least linear in the worst case.

First we analyze the edge stretch of the algorithm. Since we want to give a lower bound on the stretch, we can estimate the average edge stretch from below by adding the edge stretch of only certain edges and dividing it by the number of all edges; if this is still linear in n, then so is the average edge stretch for all edges. This way, for the sake of simplicity, we only take the edges (v_i, v_n) for all $i \in \{2, ..., n-1\}$ values (the edges from the central node, except for (v_1, v_n)). For these edges, the stretch of the edge (v_i, v_n) is $i \cdot (1-\varepsilon)$, therefore, the average edge stretch can be bounded from below by

$$\frac{1}{2 \cdot (n-1)} \cdot \sum_{i=2}^{n-2} i \cdot (1-\varepsilon)$$

which is

$$\frac{(1-\varepsilon) \cdot n \cdot (n-3)}{4 \cdot (n-1)}$$

that can obviously be estimated from below by $\frac{1-\varepsilon}{4} \cdot (n-3)$. Thus the average edge stretch in the graph is $\Omega(n)$.

To calculate path stretch of the algorithm, we have to take the average of path stretch values for each pair of points in the graph. It is also easier in this case to underestimate the average path stretch by adding the stretch for only certain pairs of vertices. Here, we select some pairs of nodes on the "outside" of the wheel not too close to each other: we choose the pairs v_i and v_{i+j} where $i \in 1, ..., n-2$, $j \geq 3$ and $i+j \leq n-1$. For these pairs, assuming $\varepsilon < \frac{1}{3}$, the shortest path between them in the original graph is going through node v_n, having length 2, while the path in the spanning tree is $j \cdot (1-\varepsilon)$ long, therefore the stretch for such a pair is $\frac{j \cdot (1-\varepsilon)}{2}$. Since for a certain value of j, the value of i can range from 1 to $n-1-j$, there are $n-1-j$ pairs of nodes with this stretch value for every j, where j can take values between 3 and $n-2$. Therefore, with these stretch values, the

average path stretch for the graph can be bounded from below by:

$$\frac{1}{\binom{n}{2}} \cdot \sum_{j=3}^{n-2} \sum_{i=1}^{n-1-j} \frac{j \cdot (1-\varepsilon)}{2}$$

This is in fact

$$\frac{2}{n \cdot (n-1)} \cdot \frac{1-\varepsilon}{2} \cdot \sum_{j=3}^{n-2} (n-1-j) \cdot j = \frac{1-\varepsilon}{n \cdot (n-1)} \cdot \frac{1}{6} \cdot (n^3 - 3n^2 - 16n + 48).$$

Clearly, this expression is also linear in n; therefore, the average path stretch in the graph is also $\Omega(n)$.

Hence it is shown that for certain graphs on n points, the average edge stretch and path stretch grows linearly in n. Since all the previously inspected bounds on stretch values were sublinear, this shows that even though the Kruskal algorithm sometimes yielded better empiric results than each previous algorithmic construction that had a bound on the stretch value with a valid proof, but all these algorithms are in fact significantly better in terms of worst-case average stretch on a graph of n vertices than the minimum spanning tree algorithm.

6 Conclusion

In the thesis, we have reviewed the problem of finding spanning trees of low average stretch value in undirected weighted graphs. We presented three different algorithms for the problem: the decomposition method of Alon *et al.*, the star decomposition algorithm of Elkin *et al.* and the improved version of the latter, developed by Abraham *et al.* We have analyzed in detail the way these methods operate, and also discussed the claims about the algorithms to give a sketch of the proofs of the upper bounds on the stretch values they return.

While discussing the star decomposition algorithm, we have found that one of the propositions used in the article is flawed. We have given a counterexample to the proposition, and shown that due to this invalid statement, in some cases an important theorem about the properties of the decomposition does not hold either. We have shown that by using a modified version of cones introduced in the paper of Abraham *et al.*, the previously incorrect proposition holds. We have proven that by altering some of the statements of the paper, the required property of the decomposition can be guaranteed, and therefore the stretch of the modified version of the algorithm can indeed by bounded as shown in the paper of Elkin *et al.*

We have compared the performance of all these algorithms on various different graphs. We have concluded that for different graph models, different algorithms seem to be the most effective. For the Erdős-Rényi graph model, we have found that the star decomposition algorithm returns the best spanning tree, or the upgraded star decomposition method of Abraham in the case of sparse graphs. In the case of the geometric graph model, we have seen that the Kruskal algorithm was the one to produce the best results.

However, we have also noticed that in terms of path stretch, the algorithms presented in this thesis are indeed superior to the Kruskal algorithm. Finally, we have shown that despite its efficiency in practical use, the average stretch values of the minimum spanning tree algorithm grow at least linearly in the worst case.

Although the thesis gave an exhaustive overview of the topic, there are various possibilities for potential future work in the area. The most obvious of these is the overview, implementation and empiric study of further algorithms for finding low-stretch spanning trees, like the Petal-decomposition of Abraham [4]. Naturally, another possibility is the development of other, completely new algorithms for the problem. Also, it is an open question whether the proofs of the claims on the star decomposition method can be corrected with only smaller changes to the algorithm, preferably without altering the original definition of cones. The empiric results can also be easily extended by further experiments on other random graph models, real-world benchmarks, or larger graphs from the models examined in the thesis. Also, the scope of the overview could be extended to spanners, distributions on spanning trees, or other further areas related to the topic.

References

[1] Alon, N., Karp, R. M., Peleg, D., & West, D. (1995). A graph-theoretic game and its application to the k-server problem. SIAM Journal on Computing, 24(1), 78-100.

[2] Elkin, M., Emek, Y., Spielman, D., & Teng, S. (2008). Lower-Stretch Spanning Trees. special issue of SIAM Journal on Computing for STOC'05 conference, 38(2), 608-628. Available at: http://www.cs.bgu.ac.il/∼elkinm/sicomp_final.pdf

[3] Abraham, I., Bartal, Y., & Neiman, O. (2008). Nearly tight low stretch spanning trees. FOCS '08: Proceedings of the 2008 49th annual IEEE Symposium on Foundations of Computer Science, 781-790.

[4] Abraham, I., & Neiman, O. (2012). Using petal-decompositions to build a low stretch spanning tree. In Proceedings of the 44th Symposium on Theory of Computing (pp. 395-406). ACM.

[5] Hassin, R., & Tamir, A. (1995). On the minimum diameter spanning tree problem. Information processing letters, 53(2), 109-111.

[6] Boman, E., & Hendrickson, D. (2001). On spanning tree preconditioners. Manuscript, Sandia National Labs, Livermore, CA.

[7] Spielman, D. A., & Teng, S. H. (2004). Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems. In Proceedings of the thirty-sixth annual ACM symposium on Theory of computing (pp. 81-90). ACM.

[8] Christiano, P., Kelner, J. A., Madry, A., Spielman, D. A., & Teng, S. H. (2011). Electrical flows, laplacian systems, and faster approximation of maximum flow in

undirected graphs. In Proceedings of the 43rd annual ACM symposium on Theory of computing (pp. 273-282). ACM.

[9] Reshef, E. (1999). Approximating minimum communication cost spanning trees and related problems (Doctoral dissertation, Weizmann Institute of Science).

[10] Peleg, D., & Reshef, E. (1998). Deterministic polylog approximation for minimum communication spanning trees (pp. 670-681). Springer Berlin Heidelberg.

[11] Fakcharoenphol, J., Rao, S., & Talwar, K. (2003). A tight bound on approximating arbitrary metrics by tree metrics. In Proceedings of the thirty-fifth annual ACM symposium on Theory of computing (pp. 448-455). ACM.

[12] Cai, L., & Corneil, D. G. (1995). Tree spanners. SIAM Journal on Discrete Mathematics, 8(3), 359-387.

[13] Emek, Y., & Peleg, D. (2004). Approximating minimum max-stretch spanning trees on unweighted graphs. In Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms (pp. 261-270). Society for Industrial and Applied Mathematics.

[14] Althöfer, I., Das, G., Dobkin, D., Joseph, D., & Soares, J. (1993). On sparse spanners of weighted graphs. Discrete & Computational Geometry, 9(1), 81-100.

[15] Cai, L. (1994). NP-completeness of minimum spanner problems. Discrete Applied Mathematics, 48(2), 187-194.

[16] Dor, D., Halperin, S., & Zwick, U. (2000). All-pairs almost shortest paths. SIAM Journal on Computing, 29(5), 1740-1759.

[17] Bollobás, B. (2004). Extremal graph theory. Courier Dover Publications, 107-109.

[18] Dezső, B., Jüttner, A., & Kovács, P. (2011). LEMON–an open source C++ graph template library. Electronic Notes in Theoretical Computer Science, 264(5), 23-45.

[19] Erdős, P., & Rényi, A. (1960). On the evolution of random graphs. Magyar Tud. Akad. Mat. Kutató Int. Közl, 5, 17-61.