University of Amsterdam

MSc Mathematics

Master Thesis

Quantum algorithms for vertex sparsification

Author:
Lorenzo Grevink

Supervisor:
prof. dr. Ronald de Wolf

Examination date:
July 17, 2024

Korteweg-de Vries Institute for
Mathematics

Abstract

The goal of graph sparsification is to compress large graphs into smaller ones. In vertex sparsification we are given a graph and a set of terminals. We aim to find a graph with the terminals as vertices, that preserves some property of interest. For vertex-flow sparsification we aim to preserve flows. Moitra proved that vertex-flow sparsifiers exist with quality $O\left(\frac{\log k}{\log \log k}\right)$, where k is the number of terminals. There is a classical algorithm with runtime $\tilde{O}(m^2)$ to find such vertex-flow sparsifier, where m is the number of edges of the original graph. In this work we present a new quantum algorithm with runtime $\tilde{O}(m^{11/6}n^{1/6})$ that finds the same quality vertex-flow sparsifier, where n is the number of vertices of the original graph. This is a small polynomial speedup for dense graphs. The quantum speedup is based on the classical algorithm, but uses various quantum subroutines from the literature.

Title: Quantum algorithms for vertex sparsification

Author: Lorenzo Grevink, lorenzogrevink@student.uva.nl, 12390089
Supervisor: prof. dr. Ronald de Wolf
Second Examiner: dr. Maris Ozols

Examination date: July 17, 2024

Korteweg-de Vries Institute for Mathematics
University of Amsterdam
Science Park 105-107, 1098 XG Amsterdam
`http://kdvi.uva.nl`

Contents

1 Introduction

1.1 Vertex sparsification

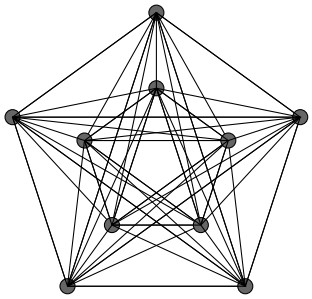
Graph theory is essential in understanding communication networks. If, for example, we are interested in transferring data between data centers, then we would like to know how to do that efficiently. In this example, our graph could form a gigantic communication network. For solving problems in graph theory, we typically need a runtime which depends on the number of edges and the number of vertices of our graph. If the graph is very large, then we would need a lot of storage space and time to solve these problems.

Graph sparsification offers a solution to this problem. In graph sparsification we try to make our graph smaller, while approximately preserving some properties of interest of the graph. This reduces the storage space needed to store the graph and it reduces computational time for various problems, at the expense of having only approximate solutions. Broadly speaking, there are two types of graph sparsification: edge sparsification and vertex sparsification.

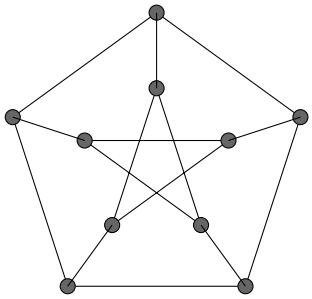
In **edge sparsification** we are only allowed to reduce the number of edges, while we keep all vertices. There are different formal definitions of edge sparsification, depending on what features of the graph we want to preserve. Examples are edge sparsifiers that ϵ-approximate cuts [17, 2] or exactly preserve cuts smaller than some constant [24]. Special interest goes to spectral sparsification, introduced by Spielman and Teng [27]. Given a graph G with graph Laplacian L, we want to find a graph on the same set of vertices that has a Laplacian that ϵ-approximates L. Spielman and Teng proved such spectral sparsifiers exist with a number of edges nearly linear in the number of vertices of the graph, and that it can be found in time nearly linear in the number of edges. An example of spectral sparsification is given in Figure 1.1. We see the complete graph on $n = 10$ vertices, K_{10}. This is a dense graph, and we want to find a sparse graph that approximates the Laplacian of K_{10}. The Petersen graph depicted in Figure 1.1b is a 3-regular graph that $\sqrt{5/2}$-approximates the Laplacian of K_{10}, as argued by Spielman and Teng [27] in their Example 1. The sparse graph will now be easier to work with than the original dense graph, but we have to pay an expense in

Apers and de Wolf [1] showed that the runtime to find spectral sparsifiers can be improved to $\tilde{O}(\sqrt{mn}/\epsilon)$ using quantum algorithms, where n is the number of vertices and m the number of edges of the original graph. Note that this is a polynomial speedup in the case that $\epsilon > \sqrt{n/m}$.

This quantum speedup directly gives speedups for a number of other problems from graph theory, for example Laplacian solving. In this problem we are given a graph Laplacian L and a vector $b \in \mathbb{R}^n$, and we want to find a vector $x \in \mathbb{R}^n$ such that $Lx = b$. If we allow ourselves a small error, then we could look at the problem of finding an x



(a) The complete graph on $n = 10$ vertices, K_{10}. Every edge has weight 1.



(b) A d-regular spectral sparsifier with $d = 3$. Every edge has weight $n/d = 10/3$. This graph is known as the Petersen graph.

Figure 1.1: Spectral sparsification of K_{10}

such that $||x - L^+b||_L \leq \epsilon||L^+b||_L$, where $\epsilon > 0$ is some real number, $||b||_L \coloneqq \sqrt{b^T Lb}$ and L^+ is the pseudo-inverse of L. There is a classical algorithm that finds such an x in runtime $\tilde{O}(m \log(1/\epsilon))$ [30, Theorem 3.1]. If $\epsilon > \sqrt{n/m}$, then we could apply the quantum spectral sparsification algorithm of Apers and de Wolf beforehand, giving us a runtime of $\tilde{O}(\sqrt{mn}/\epsilon)$.

Inspired by the work of Apers and de Wolf on spectral sparsification, in this thesis we want to see if we could speed up algorithms for vertex sparsification as well.

In **vertex sparsification** we are not only allowed to remove edges, but we are also allowed to reduce the number of vertices. To go back to our example of the data centers: maybe we are only interested in how we can communicate between a few data centers. Many of the vertices in the graph describing the communication network will thus not be of interest to us. In a vertex-sparsification problem we are typically given a graph $G = (V, E)$ and a subset of the vertices $K \subset V$, that we call the terminals. We want to return a graph with fewer vertices than G, but still containing all vertices in K. While making the graph smaller, we want to preserve certain properties of the graph.

Just like in edge sparsification, there are different notions of vertex sparsification, depending on which properties we want to preserve. In Chapter 2 we discuss three different definitions. We will discuss vertex-flow sparsifiers (Section 2.1) where we want to approximately preserve flows, vertex-cut sparsifiers (Section 2.2) where we want to approximately preserve cuts, and connectivity-c mimicking networks (Section 2.4) where we want to exactly preserve small cuts.

In Chapter 3 we discuss a classical algorithm to find vertex-flow sparsifiers. Moitra [22] showed that vertex-flow sparsifiers on vertex set K exist with quality

$$q = O\left(\frac{\log |K|}{\log \log |K|}\right),$$

i.e. flows get preserved up to a factor q. Note that because $|K|$ tends to be small, often a constant independent of the number of vertices, this quality is quite good. The classical algorithm we discuss in Chapter 3 finds a vertex-flow sparsifier with this quality.

It was first stated by Englert, Gupta, Krauthgamer, Räcke, Talgam-Cohen and Talwar [11], and a complexity analysis was given by Moitra [23]. This classical algorithm has runtime $\tilde{O}(m^2)$.

Chapter 4 is about quantum algorithms.

1.2 Quantum algorithms

Classical computers store information using bits, that can be either in the state 0 or 1. Quantum computers use **qubits**, which are different than bits in the sense that they can be in a **superposition** of 0 and 1. A superposition can be interpreted as being both 0 and 1 "at the same time". We formally define the **state** of a qubit as a vector in $\mathbb{C}^2$

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$$

where $\alpha, \beta \in \mathbb{C}$. We also require that $|\alpha|^2 + |\beta|^2 = 1$. If we measure the qubit in the standard basis, then we find $|0\rangle$ with probability $|\alpha|^2$ and $|1\rangle$ with probability $|\beta|^2$. After the measurement, the state collapses to the state that we received as the outcome of the measurement. For example, if we measure the state $|\psi\rangle = \frac{1}{\sqrt{2}}\,|0\rangle + \frac{1}{\sqrt{2}}\,|1\rangle$ in the computational basis, and we obtained outcome $|0\rangle$, then the state of the qubit after the measurement will be $|0\rangle$, and not the state that we started with.

If we have n qubits, then a state looks like

$$|\psi\rangle = \sum_{i=0}^{2^n-1} \alpha_i\,|i\rangle\,,$$

where $\alpha_i \in \mathbb{C}$ for every i, and $\sum_{i=0}^{2^n-1} |\alpha_i|^2 = 1$. We note that if we wanted to store this state on a classical computer, then we would need to store 2^n complex numbers. This suggests that quantum computers have the potential to perform certain computations much faster than classical computers.

To make use of these qubits, we need the ability to implement some elementary gates on them. In quantum computers, gates correspond to unitary matrices that we can apply to the states. Some common examples of quantum gates on one qubit are

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix},\ H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}.$$

The X-gate flips the $|0\rangle$ and $|1\rangle$ states, while the H-gate brings the $|0\rangle$ or $|1\rangle$ state into a superposition. A common example of a quantum gate on two qubits is the CNOT (controlled not) gate

$$\text{CNOT} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}.$$

The CNOT gate is a two qubit gate that applies the X gate to the second qubit iff the first qubit is in the $|0\rangle$ state.

There are several computational problems where quantum algorithms offer significant speedups. We will outline two of the most famous examples.

Shor's algorithm [26] is a polynomial-time quantum algorithm for factoring an integer into its prime factors. This algorithm is exponentially faster than the best known classical algorithm. This speedup could have far-reaching implications in cryptography [28].

Grover's algorithm [15] is a polynomial speedup for searching a marked element in a set of size N. The fastest classical algorithm to do so runs in time roughly N, but Grover's algorithm runs in time roughly $\sqrt{N}$. We will discuss some applications of Grover's algorithm in Section 4.2.

Today's computers are classical computers, but a lot of research is being done in building a well functioning quantum computer. The quantum computers that exist today are not yet powerful enough to perform useful computations: they do not have many qubits yet, and both the qubits and the gates are too noisy. However, we hope that in the future quantum computers will be made that can actually be used. It is therefore useful to study what classical computations could be sped up using quantum computers. We refer to the lecture notes of Ronald de Wolf [8] for more information on quantum algorithms.

In Chapter 4 we discuss our own quantum algorithm. It is a polynomial speedup for vertex sparsification, based on the classical algorithm by Englert et al. [11]. The main theorem of the thesis is as follows:

Theorem 1. *Let $G = (V, E, c)$ be a capacitated graph, and $K \subset V$ a set of terminals. Let $k = |K|, n = |V|$ and $m = |E|$. There exists a quantum algorithm that outputs a vertex-flow sparsifier of G on vertex set K with quality*

$$q = O\left(\frac{\log k}{\log \log k}\right),$$

in runtime

$$\tilde{O}\left(m^{11/6} n^{1/6}\right),$$

with success probability $\geq 2/3$.

The classical runtime is $\tilde{O}\left(m^2\right)$. Therefore, if m is larger than n by more than a polylogarithmic factor, then this quantum algorithm achieves a better runtime than the best known classical algorithm.

Our quantum algorithm is one of the first quantum optimization algorithms to combine an iterative algorithm with a dynamic data structure. This approach has become very prominent in classical algorithms in recent years, as evidenced by the breakthrough by Chen, Kyng, Liu, Peng, Gutenberg and Sachdeva [7], who found an almost-linear time algorithm that computes exact maximum flows and minimum-cost flows on directed graphs.

2 Different types of vertex sparsification

In this chapter we discuss vertex-flow sparsification and vertex-cut sparsification. We also discuss the relation between the two. Then we discuss a third definition of vertex sparsification: the connectivity-c mimicking networks.

2.1 Vertex-flow sparsification

Let $G = (V, E, c)$ be a graph, where $c : E \to \mathbb{R}_+$ is a capacity function. The capacity function can be interpreted as the amount of flow that can pass an edge. We will sometimes write the capacity function as a map $c : V \times V \to \mathbb{R}_+$, which gives $c(u, v) = 0$ if $(u, v) \notin E$. We are interested in calculating the maximum flow between various vertices. However, we are not interested in all vertices. We are only interested in a subset of the vertices $K \subset V$, which we will call the **terminals**. Instead of calculating all the flows separately, we could also choose to create a smaller graph. This smaller graph should still contain the terminals, and should have approximately the same flows between the terminals. We can then use this smaller graph as a proxy for the bigger graph, giving us a more efficient way to store the relevant aspects of our network. This smaller graph, called a **vertex-flow sparsifier**, will be defined more exactly in this section.

Let us say that we are interested in routing multiple flows simultaneously in graph G. We are given pairs of vertices $s_i, t_i \in V$, and a demand $d_i \in \mathbb{R}_+$, for $i = 1, \ldots, m$ for some $m > 0$. In this case the terminals will be the set $K = \cup_{i \in \{1,\ldots,m\}} \{s_i, t_i\} \subset V$. For every i we are now looking for a set P_i of paths from s_i to t_i, and weights on the paths $x : \cup_i P_i \to \mathbb{R}_+$ that satisfy

$$\sum_{P \in P_i} x(P) \geq d_i, \ \forall i \in \{1, \ldots, m\} \tag{2.1}$$

$$\sum_{P:\ e \in P} x(P) \leq c(e), \ \forall e \in E. \tag{2.2}$$

Equation (2.1) enforces that the demands are met. Equation (2.2) enforces that the capacities of the edges are not exceeded. If the two equations are satisfied, then we say that the paths $\cup_i P_i$ and x form a **flow** satisfying the demands d_i.

Not every demand can be satisfied. To formalize vertex-flow sparsification we define a **demand vector** for a set of terminals K as a vector $\vec{d} \in \mathbb{R}_+^{\binom{k}{2}}$, where $k = |K|$ and in which each coordinate corresponds to a pair of terminals $x, y \in K, \ x \neq y$. The **congestion** of a demand vector $\text{cong}_G(\vec{d})$ is now defined as the smallest $C \geq 0$ such that $\vec{d}/C$ can be satisfied in G by some flow. The congestion is thus a measure to

quantify how well a demand can be routed in a graph. In vertex-flow sparsification we want to approximately preserve this quantity.

Definition 2. *Let $G = (V, E, c)$ be a capacitated graph with terminals $K \subset V$. A graph $H = (V_H, E_H, c_H)$ is a **vertex-flow sparsifier** of G if $K \subseteq V_H$ and for every demand vector $\vec{d} \in \mathbb{R}_+^{\binom{k}{2}}$ we have*

$$\operatorname{cong}_H(\vec{d}) \leq \operatorname{cong}_G(\vec{d}). \tag{2.3}$$

*The **quality** q of a vertex-flow sparsifier is defined as*

$$q := \max\left\{ \frac{\operatorname{cong}_G(\vec{d})}{\operatorname{cong}_H(\vec{d})} : \vec{d} \in \mathbb{R}_+^{\binom{k}{2}} \right\}. \tag{2.4}$$

We note that the quality q is always at least 1. The closer q is to 1, the better of an approximation the vertex-flow sparsifier gives for the congestion. We will therefore say that a vertex-flow sparsifier is of better quality that another vertex-flow sparsifier if its quality is closer to 1. We typically also want to keep $|V_H|$ small, since the idea of a vertex sparsifier is to reduce the number of vertices.

We can look at a first easy example.

Example 3. *Let $G = (V, E, c)$ be a capacitated graph and let $K = \{s, t\} \subset V$ contain exactly two vertices. In this case a demand vector $\vec{d} \in \mathbb{R}_+^1$ is just a non-negative real number d. The congestion will now simply be $\operatorname{cong}_G(\vec{d}) = d/b$, where b is the maximum flow between s and t in G. We can construct a "trivial" vertex-flow sparsifier $H = (V_H, E_H, c_H)$, where $V_H = K = \{s, t\}$ and E_H just consists of one edge connecting s and t with capacity b. This trivial vertex-flow sparsifier has quality 1.*

Of course this example is not very exciting. If we allow K to have more than 2 elements, then it gets more interesting.

In the case where $|K| = 2$ it is easy to calculate the quality of a vertex-flow sparsifier, but it gets more complicated when $|K| > 2$. There is a lemma that will help us to calculate the quality.

Lemma 4 (Claim 1 by Leighton and Moitra [20])**.** *Let $G = (V, E, c)$ be a capacitated graph, $K \subset V$ a set of terminals, $k = |K|$, and $H = (K, E_H, c_H)$ a vertex-flow sparsifier on vertex-set K. If $\vec{H} \in \mathbb{R}_+^{\binom{k}{2}}$ is the demand vector such that the entry corresponding to every pair (a, b) of terminals is exactly $c_H(a, b)$, then the quality q of H is*

$$q = \operatorname{cong}_G(\vec{H}).$$

Proof. Since $\vec{H}$ can be routed in H by saturating all edges, we have that $\operatorname{cong}_H(\vec{H}) = 1$. It follows directly from Equation (2.4) that $q \geq \operatorname{cong}_G(\vec{H})$.

Now let $\vec{d}$ be a demand vector and assume without losing generality that $\operatorname{cong}_H(\vec{d}) = 1$. If we prove that $\operatorname{cong}_G(\vec{d}) \leq \operatorname{cong}_G(\vec{H})$, then we are done. It thus suffices to find a flow $\vec{f}_G$ in G that satisfies the demand $\vec{d}/\operatorname{cong}_G(\vec{H})$.

Let $\vec{f}_H$ be a flow in H that satisfies the demand $\vec{d}$. For $a, b \in K$ let

$$P_{a,b} = (p_0, p_1), (p_1, p_2), \ldots, (p_\ell, p_{\ell+1})$$

be a path connecting $p_0 = a$ to $p_{\ell+1} = b$ that has weight $x_H(P_{a,b}) = \delta > 0$ in $\vec{f}_H$. Let P_i be the set of paths in G connecting p_i to p_{i+1} in the flow in G that satisfies demand vector $\vec{H}/\operatorname{cong}_G(\vec{H})$. We multiply the weights in P_i by $\delta/c_H(p_i, p_{i+1})$. We define $P'_{a,b}$ as being the union over all i of these reweighed flow paths. We note that $P'_{a,b}$ sends δ units of flow from a to b in G.

Let $\vec{f}_G$ be the flow that we get by taking the union over all these flow paths $P'_{a,b}$ that were defined from all flow paths $P_{a,b}$ in $\vec{f}_H$. By construction it is now possible to route $\vec{f}_G/\operatorname{cong}_G(\vec{H})$ in G. Combining this with the fact that $\vec{f}_G$ satisfies the demand vector $\vec{d}$, we proved that $q \leq \operatorname{cong}_G(\vec{H})$. □

2.2 Vertex-cut sparsification

In Section 2.1 we discussed vertex-flow sparsification. In this section we discuss a similar, yet different, notion of vertex sparsification, called vertex-cut sparsification.

Let $G = (V, E, c)$ be a capacitated graph and suppose we are given a set of terminals $K \subset V$. We want to find a graph on fewer vertices than G, but still containing K, that approximately preserves the minimum cuts between subsets of K. To formally define these sparsifiers we first define the **cut function** $h_G : 2^V \to \mathbb{R}_+$ of a graph G as

$$h_G(U) = \sum_{e \in \delta(U)} c(e),$$

where $\delta(U)$ denotes the set of all edges crossing the cut $(U, V \setminus U)$ in G. We can now define the **terminal cut function** $h_{G,K} : 2^K \to \mathbb{R}_+$ as

$$h_{G,K}(A) = \min_{U \subset V, U \cap K = A} h_G(U).$$

The interpretation of $h_{G,K}(A)$ is that it is the value of the minimum cut separating A from $K \setminus A$ in graph G, where $A \subset K$ is a subset of the terminals. We want to approximately preserve the terminal cut function for vertex-cut sparsification.

Definition 5. *Given a capacitated graph $G = (V, E, c)$ and a set of terminals $K \subset V$, a **vertex-cut sparsifier** is a graph $H = (V_H, E_H, c_H)$, where $K \subset V_H$, such that the cut function $h_{H,K}(A) : 2^K \to \mathbb{R}_+$ of H satisfies*

$$h_{G,K}(A) \leq h_{H,K}(A)$$

*for every $A \subset K$. The **quality** q of a vertex-cut sparsifier is defined as*

$$q = \max_{A \subsetneq K, A \neq \emptyset} \frac{h_{H,K}(A)}{h_{G,K}(A)}.$$

We note that $h_{G,K}(K) = h_{G,K}(\emptyset) = 0$. Assuming G is connected will make sure that $h_{G,K}(A) \neq 0$ for $A \neq \emptyset, K$, and that the quality is well defined.

Just like in the case of vertex-flow sparsification, the quality is always at least 1. The interesting cases, again, are the cases in which H has much fewer vertices than G.

2.3 Relation

There is a famous theorem relating flows and cuts.

Theorem 6 (mincut-maxflow, 11.3 in [3])**.** *If $G = (V, E, c)$ is a capacitated graph and $s, t \in V$ are two distinct vertices, then the maximum flow passing from s to t equals the minimum cut separating s and t.*

Using Theorem 6 we see that Example 3 is also an example of a vertex-cut sparsifier, of the same quality. In this section we discuss the relation between the two definitions of vertex sparsification further.

It turns out that in an important case vertex-cut sparsification is a relaxation of vertex-flow sparsification.

Theorem 7 (Leighton and Moitra [20])**.** *Let $G = (V, E, c)$ be a capacitated graph with terminals $K \subset V$. If a graph $H = (K, E_H, c_H)$ is a vertex-flow sparsifier of quality q, then H is a vertex-cut sparsifier of quality at most q.*

Proof. Let $H = (K, E_H, c_H)$ be a vertex-flow sparsifier and let q be its quality. We know that for every demand vector $\vec{d} \in \mathbb{R}_+^{\binom{k}{2}}$ we have

$$\text{cong}_H(\vec{d}) \leq \text{cong}_G(\vec{d}) \leq q \cdot \text{cong}_H(\vec{d}). \tag{2.5}$$

Our goal is to prove that $h_{G,K}(A) \leq h_{H,K}(A) \leq q \cdot h_{G,K}(A)$ for all $A \subset K$.

We assume, to get to a contradiction, that there exists some $A \subset K$ such that $h_{G,K}(A) > h_{H,K}(A)$. The mincut-maxflow theorem tells us that there exists a flow $\vec{r}$ feasible in G such that the total flow crossing the cut $(A, K \setminus A)$ is exactly $h_{G,K}(A)$. But $\vec{r}$ cannot be feasible in H, since the cut $(A, K \setminus A)$ has capacity $h_{H,K}(A)$ in H which is strictly smaller than $h_{G,K}(A)$. We get that $\text{cong}_G(\vec{r}) \leq 1 < \text{cong}_H(\vec{r})$, which contradicts Equation (2.5). We thus proved that $h_{G,K}(A) \leq h_{H,K}(A)$ for all $A \subset K$.

Now we assume, to get to a contradiction, that there is a $A \subset K$ such that $h_{H,K}(A) > q \cdot h_{G,K}(A)$. We define $\vec{H} \in \mathbb{R}_+^{\binom{k}{2}}$ as in Lemma 4. As we saw in the proof of the lemma, we have that $\text{cong}_H(\vec{H}) = 1$. Now let $U \subset V$ be such that $U \cap K = A$ and $h_G(U) = h_{G,K}(A)$, such U exists by the definition of $h_{G,K}$. The sum of the capacities of the edges crossing the cut $(U, V \setminus U)$ is $h_{G,K}(A)$, but the demand crossing the cut is $h_{H,K}(A)$. We find that

$$\text{cong}_G(\vec{H}) \geq \frac{h_{H,K}(A)}{h_{G,K}(A)} > q.$$

Therefore, $\text{cong}_G(\vec{H}) > q = q \cdot \text{cong}_H(\vec{H})$, which contradicts Equation (2.5). We thus proved that $h_{H,K}(A) \leq q \cdot h_{G,K}(A)$.

We conclude that $h_{G,K}(A) \leq h_{H,K}(A) \leq q \cdot h_{G,K}(A)$ for all $A \subset K$, which means that H is a vertex-cut sparsifier of G of quality at most q. □

Note that in Theorem 7 we are looking at the case in which the set of vertices of the vertex sparsifier is exactly the set of terminals, i.e. $V_H = K$. We could wonder if

vertex-flow sparsifiers and vertex-cut sparsifiers are the same in this case. However, this is not the case; Leighton and Moitra [20] proved that there exists an infinitely large family of capacitated graphs $G = (V, E, c)$ and sets $K \subset V$ with $k = |K|$ such that there exists a vertex-cut sparsifier of quality $O(1)$, but there exists no vertex-flow sparsifier with quality less than $\Omega(\log \log k)$.

So vertex-cut sparsifiers and vertex-flow sparsifiers are really different concepts. In Section 2.6 we will see that vertex-flow sparsifiers of good quality exist. This directly implies the same for vertex-cut sparsifiers.

2.4 Connectivity-c mimicking network

In this section we discuss a form of vertex sparsification introduced by Chalermsook, Das, Kook, Laekhanukit, Liu, Peng, Sellke and Vaz [5]. Just like for vertex-flow sparsification and vertex-cut sparsification we are only interested in a subset of the vertices of our graph, called the terminals. For connectivity-c mimicking networks we are interested in preserving small cuts exactly, but we do not care about large cuts. We first make this precise. Recall the definition of the terminal cut function $h_{G,K}$ in Section 2.2. In this section all graphs are multigraphs, which can be interpreted as weighted graphs with integer weights.

Definition 8. *Let $G = (V, E)$ be a multigraph with $K \subset V$ a set of terminals, $k = |K|$, $n = |V|$, $m = |E|$ and c an integer. A multigraph $H = (V', E')$, with $K \subset V'$, is (K, c)-equivalent to G if for every $A \subset K$ we have that*

$$\min\{c, h_{G,K}(A)\} = \min\{c, h_{H,K}(A)\}.$$

We say that H is a connectivity-c mimicking network of G.

A special case of (K, c)-equivalency is if we do not want to decrease the number of vertices, i.e. we have that $V = V' = K$. If we also require that $E' \subset E$, then this is a form of edge sparsification, distinct from spectral sparsification. There is a theorem from the literature about this.

Lemma 9 (Nagamochi and Ibaraki [24])**.** *For a multigraph $G = (V, E)$ with $n = |V|$ there exists a sub-multigraph $H = (V, E')$ such that G and H are (K, c)-equivalent, and such that the number of edges $|E'|$ is upper bounded by $O(nc)$.*

Furthermore, Chalermsook et al. [5, Lemma 2.6] showed that this sparse graph can be found in time $O(cm)$. This can be useful for an algorithm to find connectivity-c mimicking networks on fewer vertices, because we can do this edge-sparsification beforehand. Thus, we can assume that the graph is sparse.

A natural way to construct connectivity-c mimicking networks is by **contractions**. For an edge $e \in E$ we define G/e as the graph obtained from G identifying the two endpoints of e as a single vertex. We say that we have contracted the edge e. If at least one of the two endpoints was a terminal, then we mark the new vertex as a terminal

as well. For a set of edges $\hat{E} \subset E$ we let $G/\hat{E}$ denote the graph obtained from G by contracting all edges in $\hat{E}$.

We note that contractions do not decrease the terminal cut function.

Lemma 10. *If $A \subset K$ and $\hat{E} \subset E$ is such that no terminal is identified with another terminal in $G/\hat{E}$, then we have that*

$$h_{G,K}(A) \leq h_{G/\hat{E},K}(A).$$

This gives an idea for an algorithm to find connectivity-c mimicking networks. For an edge $e \in E$, check if contracting it preserves (K, c)-equivalency, i.e. if

$$\min\{c, h_{G,K}(A)\} = \min\{c, h_{G/e,K}(A)\}$$

for every $A \subset K$. If this is the case, then we call this edge **contractible**.

A high-level approach for an algorithm is now to enumerate over the edges and contract an edge if it is contractible. The problem with this approach is that there is no known algorithm that runs in polynomial time that checks if an edge is contractible. An algorithm presented by Chalermsook et al. [5] works with this approach, but their algorithm does not run in polynomial time in c. However, there exists a classical algorithm that finds connectivity-c mimicking networks in polynomial time.

Lemma 11 (Liu [21])**.** *For a multigraph $G = (V, E)$ with $K \subset V$, $n = |V|$, $k = |K|$ there exists a connectivity-c mimicking network $H = (V', E')$ with $|E'| = O(kc^3)$ edges. There is a classical algorithm that runs in time $n^{O(1)}$ that finds a connectivity-c mimicking network with $O(kc^3 \log^{3/2} n \log\log n)$ edges.*

In the rest of this thesis we will focus on vertex-flow sparsifiers and vertex-cut sparsifiers.

2.5 Various directions of study

In Section 2.3 we looked at vertex sparsifiers on vertex-set $V_H = K$, but in Definition 2 and Definition 5 we only required that $K \subset V_H$. The vertices $V_H \setminus K$ are called **Steiner vertices**. Taking into account that the set of Steiner vertices does not need to be empty, we can look at the following question.

What happens if we fix $q \geq 1$, and we look for vertex sparsifiers $H = (V_H, E_H, c_H)$ that have quality at most q? The interesting question now is: how small can we keep $|V_H|$? In the answer to this question there turns out to be a clear distinction between the two types of vertex sparsification, just like in the case in which we do require that $K = V_H$.

Let us for example take $q = 1$, which means that we are looking for vertex sparsifiers that exactly preserve either cuts or flows. It was shown by Hagerup, Katajainen, Nishimura and Ragde [16] that such vertex-cut sparsifiers exist with $|V_H| = O(2^{2^k})$. This is remarkable, since this value does not depend on the size of the original graph, only on the number of terminals. On the other hand, Krauthgamer and Mosenzon [19]

showed that such a statement cannot be made concerning vertex-flow sparsifiers. More precisely, they showed that there exists no number $M > 0$ such that all graphs with $k = 6$ terminals have a quality $q = 1$ vertex-flow sparsifier with $|V_H| \leq M$.

There is also research being done on vertex sparsification in which some requirement is put on the input graph. For example, Goranci and Räcke [14] proved that trees and quasi-bipartite graphs admit vertex-flow sparsifiers of quality 2, without using Steiner vertices. Goranci, Henzinger and Peng [13] looked at planar graphs where all terminals lie on the same face. They proved that in this case there exists a vertex-cut sparsifier of quality $q = 1$ with $O(|K|^2)$ vertices, that is still planar and all terminals still lie on the same face. Leighton and Moitra [20] showed that if a graph excludes a fixed minor, then there exists a $O(1)$-quality vertex-flow sparsifier on the terminal set.

Steiner vertices are an interesting field for future research, and it is also interesting to look at specific classes of graphs. However, from now on in this thesis we will assume that $V_H = K$, and we will not assume that our input graph is of any specific class.

2.6 0-extensions

The class of all graphs that satisfy Definition 2 is hard to characterize, so let us look at ways to create these vertex-flow sparsifiers in a structured way. It turns out there is an easy way to find graphs that satisfy Equation (2.3), using 0-extensions. Let us discuss the formal definition.

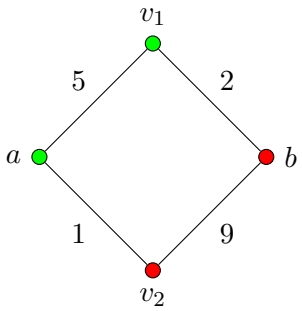
Definition 12. *Let $G = (V, E, c)$ be a capacitated graph, and $K \subset V$ a set of terminals. A **0-extension** is a map $f : V \to K$ such that $f(a) = a$ for all $a \in K$.*

We can interpret a 0-extension as a partition of the vertices of the graph into $k = |K|$ sets, with exactly one terminal in each set of the partition. Given a 0-extension we can define a vertex-flow sparsifier by contracting all vertices to their image by f. The vertices of our new graph are thus exactly the terminals. The capacity of the edge connecting two terminals in the new graph will now be the sum of all capacities of edges between elements of the two sets, where a capacity of 0 means that there is no edge. If we define c_f as the capacity function of this new graph, then we have

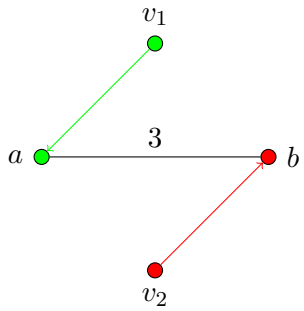
$$\forall a, b \in K,\ c_f(a, b) = \sum_{u \in f^{-1}(a), v \in f^{-1}(b)} c(u, v). \tag{2.6}$$

We let $G_f = (K, E_f, c_f)$ denote the capacitated graph on vertices K that we get with these capacities. A small example is shown in Figure 2.1. In Figure 2.1a we see a graph with 4 vertices and two terminals. We see that the vertices are partitioned in red and green vertices, by their image under the 0-extension f. In Figure 2.1b we see the graph G_f induced by the 0-extension f. The capacity of the one remaining edge is $1 + 2 = 3$, calculated using Equation (2.6).

Note that we can interpret G_f as the graph G after having contracted some of its edges. This makes it clear that if we can route a flow in G, then we can also route it in G_f. We get the following lemma.



(a) An example of a graph with four vertices and $k = 2$ terminals a and b. We define the 0-extension f by $f(v_1) = f(a) = a$ and $f(v_2) = f(b) = b$. The vertices are colored red or green depending on their image under the 0-extension.



(b) The 0-extension f sends the green vertices to a and the red vertices to b. The capacity of the edge between a and b is now the sum of the capacities of the edges in the original graph that crossed from the green to the red vertices.

Figure 2.1: A small example of a 0-extension

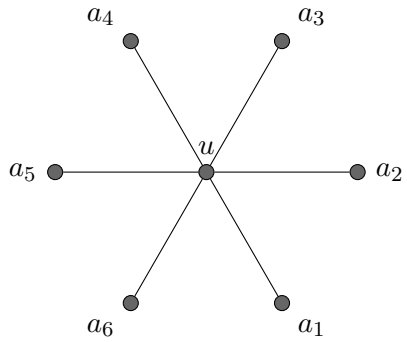
Lemma 13. *For every 0-extension $f : V \to K$, the graph G_f is a vertex-flow sparsifier of G on terminals K.*

Now G_f is a vertex-flow sparsifier of G, but possibly not a good one, i.e. it could be of bad quality $q \gg 1$. We can now look at the problem: can we always find a 0-extension that has a good quality? If this would be the case, then our problem would become much simpler, since there exists only a finite number of 0-extensions. The answer to the question, however, is no, as we will see in Example 14.

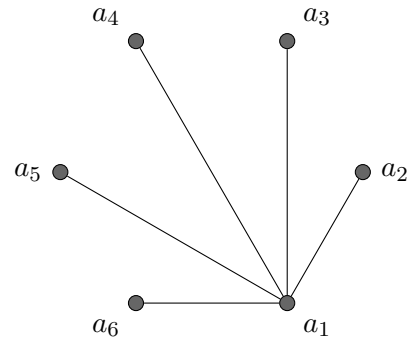
We thus need to find more ways to construct vertex-flow sparsifiers. We can do that by considering convex combination of 0-extensions. A convex combination in this context means that we have a vector $\vec{\mu} \in \mathbb{R}_+^{\mathcal{H}}$ where $\mathcal{H}$ is the set of all 0-extensions and $\sum_i \mu_i = 1$. We can define a vertex-flow sparsifier $H = (K, E_H, c_H)$ by $c_H = \sum_{f \in \mathcal{H}} \mu_f c_{G_f}$. We will look at an example that shows why this can be useful.

Example 14. *We consider the star graph G depicted in Figure 2.2a. The terminal set $K = \{a_1, \ldots, a_k\}$ is defined as all the vertices in G except for vertex u in the center. In Figure 2.2b we see G_f for a 0-extension f, which sends u, without loss of generality, to terminal a_1. We can now define the demand vector $\vec{G_f} \in \mathbb{R}_+^{\binom{k}{2}}$ such that the (a_1, a_i) entry is 1 for all $i \neq 1$, and all other entries are 0. This demand is easily routed in G_f, since we can canonically send a flow by saturating every edge. However, in graph G, we will for sure get congestion $k - 1$ in the edge (u, a_1). According to Lemma 4 we have quality $q = k - 1$, which is a really bad result since it increases linearly in k.*

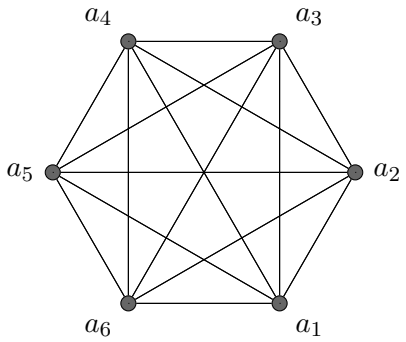
However, we can take a convex combination of 0-extensions. There are k different choices for what 0-extension to choose, i.e. we have to send u to a terminal, and we can choose between k terminals. We can make a convex combination of all these 0-extensions, giving every one the same weight $\frac{1}{k}$. We will get a complete graph on k



(a) The star graph G with $k = 6$ terminals a_i and one non-terminal u in the center. Every edge has capacity 1.



(b) The graph G_f where f is the 0-extension defined by $f(u) = a_1$. Every edge has capacity 1.



(c) The convex combination of all 0-extensions with weight $\frac{1}{k} = \frac{1}{6}$. Every edge now has capacity $\frac{2}{k} = \frac{1}{3}$.

Figure 2.2: An example of a graph where the best vertex-flow sparsifier is not a 0-extension

vertices such that each edge has capacity $\frac{2}{k}$. Using Lemma 4 this vertex-flow sparsifier has quality $q = \frac{2(k-1)}{k}$. This is a really good result since $q = O(1)$ is small, also for large values of k. We see that this convex combination of 0-extensions gives a much better vertex-flow sparsifier than a single 0-extension does.

Note, however, that not all vertex-flow sparsifiers on vertex set $V_H = K$ are convex combinations of 0-extensions. This is shown in the next example.

Example 15. *We again look at the star graph of Example 14, but choosing $k = 3$. The convex combination of all 0-extensions gives us the graph K_3 with edge capacities $\frac{2}{3}$. This vertex-flow sparsifier has quality $q = \frac{4}{3}$. However, if we take K_3 with edge capacities $\frac{1}{2}$, then we get a vertex-flow sparsifier with quality $q = 1 < \frac{4}{3}$. This vertex-flow sparsifier of quality 1, however, is not a convex combination of 0-extensions.*

So there are cases in which there exist vertex-flow sparsifiers of better quality than the best convex combination of 0-extensions. However, considering only convex combinations

of 0-extensions, we can still find vertex-flow sparsifiers of reasonable quality, as was proven by Moitra using a zero-sum game.

Theorem 16 (Moitra [22])**.** *Let $G = (V, E, c)$ be a capacitated graph, and $K \subset V$ a set of terminals. There exists a convex combination of 0-extensions that is a vertex-flow sparsifier of quality $O\left(\frac{\log k}{\log \log k}\right)$, where $k = |K|$.*

This result is remarkable, since it states that for every graph, no matter how many vertices it contains, there exists a vertex-flow sparsifier of quality that only depends on k. In Chapter 3 we discuss a classical algorithm to find vertex-flow sparsifiers that match the quality promised by Moitra. By constructing it we also prove the theorem.

It is unknown if the upper bound given in Theorem 16 is tight. Charikar, Leighton, Li and Moitra [6] found a class of graphs that admit no vertex-cut sparsifiers of quality less than $\Omega(\log^{1/4} k)$. By Theorem 7 this also gives a lower bound on the best possible vertex-flow sparsifiers.

3 A classical algorithm for vertex-flow sparsification

When running a classical algorithm on a graph, we need to access the graph a number of times. We also need to apply elementary classical operations. We use the **word-RAM** (word random access machine) model. This model assumes we have a finite memory and word-length. If we can store $\log N$ bits, then we can store an integer up to N. We can do basic operations on these $\log N$ bits, like summing or multiplying the bits with each other. We assume that doing such a basic operation can be done efficiently, in time $O(1)$. The word-RAM model defines a **runtime**.

It turns out there exist efficient classical algorithms to find vertex-flow sparsifiers that achieve the quality promised by Theorem 16. One such algorithm was presented by Charikar, Leighton, Li and Moitra [6]. This algorithm returns a vertex-flow sparsifier that is not necessarily a convex combination of 0-extensions, but that has a good quality. It has a quality at least as good as the best convex combination of 0-extensions. This algorithm has runtime polynomial in n and k, where n is the number of vertices of the original graph and k is the number of terminals. We will not further discuss this algorithm in this thesis.

In this chapter we discuss another algorithm, presented by Englert, Gupta, Krauthgamer, Räcke, Talgam-Cohen and Talwar [11]. This algorithm returns a convex combination of 0-extensions that has a quality of at most $O\left(\frac{\log k}{\log \log k}\right)$. This second algorithm thus returns a vertex-flow sparsifier of possibly a worse quality than the first algorithm. However, this second algorithm runs faster. We assume that k is small, at most $O(\log n)$. In our runtime-analysis we will absorb all factors of k and of $\log n$ in the $\tilde{O}$-notation.

3.1 The 0-extension problem

Before we discuss the algorithm, we should study the 0-extension problem. In this problem, first proposed by Karzanov [18], we are given a capacitated graph $G = (V, E, c)$ and terminals $K \subset V$. We are also given a metric D on the terminals K. The goal of the 0-extension problem is to find a 0-extension $g : V \to K$ that minimizes

$$\sum_{(u,v)\in E} c(u, v)D(g(u), g(v)).$$

As observed by Karzanov [18], this problem is NP-hard. We will therefore not try to solve this problem itself, but a relaxation of it.

For this we define the concept of a **semi-metric** $d : V \times V \to \mathbb{R}_+$. For d to be a semi-metric we need that

- $d(u,v) \geq 0$ and $d(u,u) = 0$ for all $u \in V$
- $d(u,v) = d(v,u)$ for all $u, v \in V$
- $d(u,w) \leq d(u,v) + d(v,w)$ for all $u, v, w \in V$.

A semi-metric is similar to a metric, but note that in a semi-metric there can be a distance 0 between two non-equal vertices. We can use this property to link the concept of a 0-extension to the concept of a semi-metric on the graph.

Let G, K and D be as above, and f be a 0-extension. We can define a semi-metric $\delta : V \times V \to \mathbb{R}_+$ as $\delta(u,v) = D(f(u), f(v))$. We easily check that this indeed gives a semi-metric, and we see that it coincides with D on the terminals, i.e. $\delta(a,b) = D(a,b)$ for every $a, b \in K$. We will say that the semi-metric δ is **induced** by the 0-extension f.

We can look at the following problem: given G, K and D as above, find a semi-metric δ, under the constraint that $\delta(a,b) = D(a,b)$ for all pairs of terminals $a, b \in K$, that minimizes

$$\sum_{(u,v)\in E} c(u,v)\delta(u,v).$$

Note that if a semi-metric that minimizes this sum is induced by a 0-extension, then it is a solution to the 0-extension problem. However, not all semi-metrics are induced by a 0-extension. But the problem of finding such a semi-metric δ is a relaxation of the 0-extension problem.

The question that we now ask ourselves is: if we find a semi-metric δ that solves the relaxation stated above, can we then efficiently find a 0-extension f that solves the 0-extension problem? It turns out we can!

Theorem 17 (Fakcharoenphol, Harrelson, Rao and Talwar [12])**.** *Let $G = (V, E, c)$ be a capacitated graph and $K \subset V$ a set of terminals, $k = |K|$. Also let D be a metric on the terminals, and δ a semi-metric on V such that $\delta(a,b) = D(a,b)$ for all terminals $a, b \in K$. Let*

$$C := \sum_{(u,v)\in E} c(u,v)\delta(u,v).$$

If we choose parameters $M = \log\log k$ and

$$A_v = \min_{a\in K}\{\delta(a,v)\},$$

then Algorithm 1 outputs a 0-extension f such that

$$\sum_{(u,v)\in E} c(u,v)D(f(u),f(v)) = O\left(\frac{\log k}{\log\log k}\right) C.$$

We refer to Fakcharoenphol, Harrelson, Rao and Talwar [12] for the analysis of Algorithm 1.

Note that this algorithm will assign every vertex to a terminal, since $\delta(v,a) = A_v \leq \alpha A'_v$ for the terminal a closest to v. The algorithm thus outputs a 0-extension. Also

Algorithm 1 by Fakcharoenphol, Harrelson, Rao and Talwar [12]

1: Pick a real number $\gamma \in [1, 2]$ uniformly
2: **for** $v \in V$ **do**
3: $\quad A'_v \leftarrow \min\{2^s \mid 2^s \geq 2A_v/\gamma\}$
4: Choose uniformly at random $i \in \{1, \ldots, M\}$
5: Choose uniformly at random $\alpha \in (2^i, 2^{i+1}]$
6: Choose a random permutation σ of the terminals
7: **for** $\ell := 1 \ldots k$ **do**
8: $\quad$ **for** each unassigned vertex $v \in V \setminus K$ **do**
9: $\quad\quad$ **if** $\delta(v, \sigma(\ell)) \leq \alpha A'_v$ **then**
10: $\quad\quad\quad$ assign vertex v to terminal $\sigma(\ell)$

note that the runtime of the algorithm is $\tilde{O}(kn)$, since we loop over the terminals and over the vertices in Steps 7 and 8. However, we only get this runtime if $\delta(v, a)$ is given as input for every pair $v \in V$, $a \in K$.

We will use Algorithm 1 as a subroutine for the algorithm for finding vertex-flow sparsifiers, which we will discuss in the next section.

3.2 Classical algorithm

We discuss the algorithm found by Englert, Gupta, Krauthgamer, Räcke, Talgam-Cohen and Talwar [11] and discussed by Moitra [23, Section 3.2.4] to construct convex combinations of 0-extensions that give good-quality vertex-flow sparsifiers. For a capacitated graph $G = (V, E, c)$ and $K \subset V$, writing $n = |V|, m = |E|, k = |K|$, this classical algorithm provides a $O\left(\frac{\log k}{\log \log k}\right)$-quality vertex-flow sparsifier on exactly the vertices $K \subset V$ in runtime $\tilde{O}(m^2)$. First we need to discuss some relevant concepts.

We define $\mathcal{R}$ as the set of all sets of $\binom{k}{2}$ simple paths in G such that each path connects a distinct pair of terminals. For $R \in \mathcal{R}$ and $a, b \in K$ we write $R_{a,b}$ for the path in R connecting a to b, so an $R \in \mathcal{R}$ is of the form $R = \{R_{a,b} \mid a, b \in K, a \neq b\}$.

We recall that $\mathcal{H}$ denotes the set of all 0-extensions. For an $H \times R \in \mathcal{H} \times \mathcal{R}$ we define the **load** and the **relative load**, abbreviated by rload, of an edge $e \in E$ as

$$\mathrm{load}_{H\times R}(e) := \sum_{a,b\in K;\ e\in R_{a,b}} c_H(a, b)$$

and

$$\mathrm{rload}_{H\times R}(e) := \frac{\mathrm{load}_{H\times R}(e)}{c(e)}.$$

We can now look at the problem of finding a $\vec{\lambda} \in \mathbb{R}_+^{\mathcal{H}\times\mathcal{R}}$ with $\sum_{H\times R} \lambda_{H\times R} = 1$ such that we have

$$\forall e \in E, \sum_{H\times R} \lambda_{H\times R}\, \mathrm{rload}_{H\times R}(e) \leq \beta, \tag{3.1}$$

for some $\beta \geq 1$. If we find such $\vec{\lambda}$, then we can construct a convex combination of 0-extensions $G' = \sum_{H\times R} \lambda_{H\times R} H$. The demand vector $\vec{G'}$ as defined in Lemma 4 can now be routed in G with congestion at most β, since we can define a flow that routes $c_{G'}(a,b)$ between terminals a, b through the path $R_{a,b}$. Therefore, G' is a vertex-flow sparsifier of quality $q \leq \beta$. If we thus find a $\vec{\lambda}$ that satisfies Equation (3.1), then we find a vertex-flow sparsifier of quality at most β.

The problem of finding an adequate $\vec{\lambda}$ cannot directly be solved efficiently since the set of all 0-extensions contains exponentially many elements, more precisely $|\mathcal{H}| = k^{n-k}$. We thus need some more tricks. The idea is to transform Equation (3.1) into something smooth, so that we can take the derivatives and use gradient descent. For this purpose we define

$$\operatorname{lmax}(\vec{x}) = \ln\left(\sum_e \exp(x_e)\right)$$

for $\vec{x} \in \mathbb{R}_+^E$. We then define an $E \times (\mathcal{H}\times\mathcal{R})$ matrix M such that $M_{e,H\times R} = \operatorname{rload}_{H\times R}(e)$. We thus have

$$(M\vec{\lambda})_e = \sum_{H\times R} \lambda_{H\times R} \operatorname{rload}_{H\times R}(e).$$

We note that $\operatorname{lmax}(\vec{x}) \geq x_e$ for all $e \in E$. Therefore, the following equation implies Equation (3.1),

$$\operatorname{lmax}(M\vec{\lambda}) \leq \beta. \tag{3.2}$$

The idea now is to start with $\vec{\lambda} = 0$, and slowly increase some indices $\lambda_{H\times R}$, without raising $\operatorname{lmax}(M\vec{\lambda})$ too much. To know which indices to raise, we look at

$$\frac{\partial \operatorname{lmax}(M\vec{\lambda})}{\partial \lambda_{H\times R}} \tag{3.3}$$

which we want to be small. For a small $\vec{\delta}$ we will have $\operatorname{lmax}(\vec{x}+\vec{\delta}) - \operatorname{lmax}(\vec{x}) \approx \sum_e \delta_e \frac{\partial \operatorname{lmax}(\vec{x})}{\partial x_e}$. We can make this precise using a lemma from the literature.

Lemma 18 (Lemma 1 by Young [31])**.** *If $\vec{x}, \vec{\delta} \in \mathbb{R}_+^E$ with $0 \leq \delta_e \leq 1$ for all $e \in E$, then*

$$\operatorname{lmax}(\vec{x}+\vec{\delta}) - \operatorname{lmax}(\vec{x}) \leq 2\sum_e \delta_e \frac{\partial \operatorname{lmax}(\vec{x})}{\partial x_e}.$$

We can use this lemma to learn something useful in our case.

Lemma 19. *Let $H \times R \in \mathcal{H}\times\mathcal{R}$. If $\vec{\delta} = \delta_{H\times R}\vec{e}_{H\times R} \in \mathbb{R}^{\mathcal{H}\times\mathcal{R}}$, where*

$$0 \leq \delta_{H\times R} \leq \frac{1}{\max_e\{\operatorname{rload}_{H\times R}(e)\}} \tag{3.4}$$

and $\vec{e}_{H\times R}$ is the unit vector in direction $H \times R$, then we have that

$$\operatorname{lmax}(M(\vec{\lambda}+\vec{\delta})) - \operatorname{lmax}(M\vec{\lambda}) \leq 2\delta_{H\times R}\frac{\partial \operatorname{lmax}(M\vec{\lambda})}{\partial \lambda_{H\times R}}.$$

Proof. We first note that $0 \leq (M\vec{\delta})_e = \delta_{H\times R}\,\mathrm{rload}_{H\times R}(e) \leq 1$ for all $e \in E$, so we can use Lemma 18. We calculate

$$\begin{aligned}\mathrm{lmax}(M(\vec{\lambda}+\vec{\delta})) - \mathrm{lmax}(M\vec{\lambda}) &\leq 2\sum_e (M\vec{\delta})_e \frac{\partial\,\mathrm{lmax}(M\vec{\lambda})}{\partial(M\vec{\lambda})_e} \\ &= 2\delta_{H\times R}\sum_e \mathrm{rload}_{H\times R}(e)\frac{\partial\,\mathrm{lmax}(M\vec{\lambda})}{\partial(M\vec{\lambda})_e} \\ &= 2\delta_{H\times R}\frac{\partial\,\mathrm{lmax}(M\vec{\lambda})}{\partial\lambda_{H\times R}},\end{aligned}$$

where we used the chain rule to see that

$$\frac{\partial\,\mathrm{lmax}(M\vec{\lambda})}{\partial\lambda_{H\times R}} = \sum_e \mathrm{rload}_{H\times R}(e)\frac{\partial\,\mathrm{lmax}(M\vec{\lambda})}{\partial(M\vec{\lambda})_e}.$$

□

If we now find an $H \times R$ such that $\frac{\partial\,\mathrm{lmax}(M\vec{\lambda})}{\partial\lambda_{H\times R}} \leq \beta/2$ for some $\beta > 0$, then we get that

$$\mathrm{lmax}(M(\vec{\lambda}+\vec{\delta})) - \mathrm{lmax}(M\vec{\lambda}) \leq \delta_{H\times R}\beta$$

for $\vec{\delta} = \delta_{H\times R}e_{H\times R}$ for some $\delta_{H\times R}$ in the interval specified in Lemma 19.

We get that $\sum_{H\times R}\lambda_{H\times R}$ increases by $\delta_{H\times R}$ while $\mathrm{lmax}(M\vec{\lambda})$ increases at most by $\delta_{H\times R}\beta$. The idea of the algorithm is to repeatedly find an $H \times R$ such that $\frac{\partial\,\mathrm{lmax}(M\vec{\lambda})}{\partial\lambda_{H\times R}} \leq \beta/2$, and then add $\delta_{H\times R}$ to $\lambda_{H\times R}$.

Let us say that after repeating this procedure a number of times we get

$$\sum_{H\times R}\lambda_{H\times R} = \alpha$$

for some $\alpha > 0$. This will give us $\mathrm{lmax}(M\vec{\lambda}) \leq \mathrm{lmax}(0) + \alpha\beta = \ln m + \alpha\beta$, since at every iteration $\mathrm{lmax}(M\lambda)$ increases at most by $\delta_{H\times R}\cdot\beta$.

If we have $\alpha = \ln m$, then we get $\sum_{H\times R}\lambda_{H\times R} = \ln m$ and $\mathrm{lmax}(M\vec{\lambda}) \leq (1+\beta)\ln m$. Therefore, for every $e \in E$ we have $(M\vec{\lambda})_e \leq (1+\beta)\ln m$. If we now define $\vec{\mu} = \frac{1}{\ln m}\vec{\lambda}$, then we get that $\sum_{H\times R}\mu_{H\times R} = 1$ and $(M\vec{\mu})_e \leq 1+\beta$ for every $e \in E$.

While running the algorithm, we choose $\delta_{H\times R} = 1/\max_e\{\mathrm{rload}_{H\times R}(e)\}$ where possible, since we want to increase $\sum_{H\times R}\lambda_{H\times R}$ as much as possible. Only in the last iteration we do $\delta_{H\times R} \leftarrow \ln m - \sum_{H\times R}\lambda_{H\times R}$ to make sure that the sum ends up at exactly $\ln m$.

To find an $H \times R$ with the required property we will first define a suitable $R \in \mathcal{R}$. For a pair of terminals $a, b \in K$ we define $R_{a,b}$ as the shortest path in G connecting them given edge-lengths defined by

$$d_{\vec{\lambda}}(e) = \frac{\exp\Big((M\vec{\lambda})_e\Big)}{c(e)}. \tag{3.5}$$

Naively we will find that we need a runtime of $\tilde{O}(mS)$ to compute $d_{\vec{\lambda}}$, where S is the support of $\vec{\lambda}$, since we need to compute $(M\vec{\lambda})_e = \sum_{H\times R} \lambda_{H\times R}\, \mathrm{rload}_{H\times R}(e)$ for every edge $e \in E$. However, if we know $(M\vec{\lambda})_e$, then we can efficiently compute $(M(\vec{\lambda}+\vec{\delta}))_e = (M\vec{\lambda})_e + (M\vec{\delta})_e = (M\vec{\lambda})_e + \delta_{H\times R}\, \mathrm{rload}_{H\times R}(e)$, where $\vec{\delta}$ is a vector that has support in exactly one entry. If we thus store all values $(M\vec{\lambda})_e$ and update them every iteration in our algorithm, then we will just need a runtime of $\tilde{O}(m)$ every iteration to update $d_{\vec{\lambda}}(e)$ for every $e \in E$. Once we computed the edge-lengths we can use the following lemma to find an appropriate $H \times R$.

Lemma 20. *If $\vec{\lambda}$ and $d_{\vec{\lambda}}$ as in Equation (3.5) are given, then there is a classical algorithm that finds an $H \times R \in \mathcal{H} \times \mathcal{R}$ with $\frac{\partial \mathrm{lmax}(M\vec{\lambda})}{\partial \lambda_{H\times R}} = O\left(\frac{\log k}{\log\log k}\right)$ in runtime $\tilde{O}(m)$.*

Proof. For every pair of terminals $a, b \in K$ we define $R_{a,b}$ as the shortest path connecting a to b in the graph G with edge-lengths $d_{\vec{\lambda}}$, and we define D as the shortest path metric.

Using Dijkstra's algorithm on a terminal $a \in K$ we can compute $D(a, v)$ and $R_{a,b}$ for every $v \in V, b \in K$ in runtime $\tilde{O}(m)$. Since we absorbed $\mathrm{poly}(k)$ into the $\tilde{O}$-notation we can compute $D(a, v)$ and $R_{a,b}$ for every $a, b \in K, v \in V$ in runtime $\tilde{O}(m)$.

We note that

$$\sum_{u,v\in V} c(u,v) D_{\vec{\lambda}}(u,v) \leq \sum_{e\in E} c(e) d_{\vec{\lambda}}(e).$$

We can use Theorem 17 to find a 0-extension f, $H \coloneqq G_f$ that satisfies

$$\sum_{(u,v)\in E} c(u,v) D_{\vec{\lambda}}(f(u), f(v)) = O\left(\frac{\log k}{\log\log k}\right) \sum_{e\in E} c(e) d_{\vec{\lambda}}(e).$$

Using this 0-extension we can calculate that

$$\begin{aligned} \frac{\partial\, \mathrm{lmax}(M\vec{\lambda})}{\partial \lambda_{H\times R}} &= \sum_e \mathrm{load}_{H\times R}(e) \frac{\exp\Big((M\vec{\lambda})_e\Big)}{c(e)\sum_{e'} \exp\Big((M\vec{\lambda})_{e'}\Big)} \\ &= \frac{\sum_{a,b\in K} c_H(a,b) \sum_{e\in R_{a,b}} d_{\vec{\lambda}}(e)}{\sum_{e'} \exp\Big((M\vec{\lambda})_{e'}\Big)} \\ &= \frac{\sum_{a,b\in K} c_H(a,b) D_{\vec{\lambda}}(a,b)}{\sum_{e\in E} c(e) d_{\vec{\lambda}}(e)} = O\left(\frac{\log k}{\log\log k}\right). \end{aligned} \tag{3.6}$$

The discussion under Theorem 17 states that we can find the 0-extension f in time $\tilde{O}(n)$, so the runtime is dominated by the part finding R and the distances $D(a, v)$ for every $a \in K, v \in V$. The runtime is thus $\tilde{O}(m)$. □

Step 8 of Algorithm 2 can also be computed efficiently.

Lemma 21. *Given $H \times R \in \mathcal{H} \times \mathcal{R}$, there is a classical algorithm that finds $\ell_{H\times R} \coloneqq \max_e\{\mathrm{rload}_{H\times R}(e)\}$ in time $\tilde{O}(m)$.*

Proof. We remember the definition

$$\mathrm{rload}_{H\times R}(e) = \frac{\sum_{a,b\in K;\ e\in R_{a,b}} c_H(a,b)}{c(e)}.$$

Since we are summing over $a, b \in K$ we can calculate $\mathrm{rload}_{H\times R}(e)$ for a given $e \in E$ in time $O(k^2)$. We can thus calculate $\mathrm{rload}_{H\times R}(e)$ for every $e \in E$ in time $O(k^2 m)$, while taking note of the largest such $\mathrm{rload}_{H\times R}(e)$. We can thus compute $\max_e\{\mathrm{rload}_{H\times R}(e)\}$ in time $O(k^2 m) = \tilde{O}(m)$. □

Algorithm 2 Finding a vertex-flow sparsifier

$\vec{\lambda} \leftarrow 0 \in \mathbb{R}_+^{\mathcal{H}\times\mathcal{R}}$
for $e \in E$ **do**
    $d_{\vec{\lambda}}(e) \leftarrow \frac{1}{c(e)}$
while $\sum_{H\times R} \lambda_{H\times R} < \ln m$ **do** ▷ $\tilde{O}(m)$ iterations
    for $a \in K$ **do**
        Compute $D(v,a), R_{a,b}, \forall v \in V, \forall b \in K$ ▷ $\tilde{O}(m)$
    Find $H \in \mathcal{H}$ using Theorem 17 ▷ $\tilde{O}(n)$
    $\ell_{H\times R} \leftarrow \max_e\{\mathrm{rload}_{H\times R}(e)\}$ ▷ $\tilde{O}(m)$
    $\delta_{H\times R} \leftarrow \min\left\{1/\ell_{H\times R},\ \ln m - \sum_{H\times R}\lambda_{H\times R}\right\}$
    $\lambda_{H\times R} \leftarrow \lambda_{H\times R} + \delta_{H\times R}$
    for $e \in E$ **do** ▷ m iterations
        $d_{\vec{\lambda}}(e) \leftarrow d_{\vec{\lambda}}(e) \cdot \exp(\delta_{H\times R}\,\mathrm{rload}_{H\times R}(e))$
return the non-zero entries of $\vec{\mu} = \vec{\lambda}/\ln m$

We combine all the subroutines described in this section in Algorithm 2. Using the following lemma we will upper bound the number of iterations of the algorithm. The lemma and its proof are based on a similar lemma by Räcke [25].

Lemma 22. *Algorithm 2 stops after $\tilde{O}(m)$ iterations.*

Proof. We define the potential function

$$\sum_e \sum_{H\times R} \lambda_{H\times R}\,\mathrm{rload}_{H\times R}(e).$$

We know that $\sum_{H\times R}\lambda_{H\times R}\,\mathrm{rload}_{H\times R}(e) \le \mathrm{lmax}(M\vec{\lambda})$ for every $e \in E$, and that $\mathrm{lmax}(M\vec{\lambda}) = \ln m + O\left(\frac{\log k}{\log\log k}\right)\ln m = O\left(\frac{\log k}{\log\log k}\right)\ln m$ at the end of the algorithm. We can thus upper bound the potential function by $O\left(m\log m\cdot\frac{\log k}{\log\log k}\right) = \tilde{O}(m)$.

During every iteration that is not the last iteration, we choose an $H \times R$ and raise $\lambda_{H\times R}$ by $\frac{1}{\ell_{H\times R}}$, where $\ell_{H\times R} = \max_e\{\mathrm{rload}_{H\times R}(e)\}$. Therefore, the potential function will increase by at least 1. We conclude that the algorithm stops after at most $\tilde{O}(m)$ iterations. □

We have now proved Theorem 23.

Theorem 23. *Algorithm 2 outputs a $O(\log k/\log\log k)$-quality vertex-flow sparsifier in time $\tilde{O}(m^2)$.*

Proof. In every iteration we raise $\sum_{H\times R}\lambda_{H\times R}$ by some $\delta_{H\times R}>0$, and by Lemma 20 we raise $\mathrm{lmax}(M\vec{\lambda})$ by at most $\delta_{H\times R}\cdot O\left(\frac{\log k}{\log\log k}\right)$. The algorithm stops by Lemma 22, so when the algorithm stops we have $(M\vec{\lambda})_e \le \mathrm{lmax}(M\vec{\lambda}) = \ln m + O\left(\frac{\log k}{\log\log k}\right)\ln m$ for every $e\in E$. Therefore, $\vec{\mu}=\vec{\lambda}/\ln m$ has the property that $\sum_{H\times R}\mu_{H\times R}=1$ and

$$\forall e\in E,\ \sum_{H\times R}\mu_{H\times R}\,\mathrm{rload}_{H\times R}(e) = O\left(\frac{\log k}{\log\log k}\right).$$

By the discussion after Equation (3.1) we see that $\vec{\mu}$ is a convex combination of 0-extensions that forms a vertex-flow sparsifier of quality $O\left(\frac{\log k}{\log\log k}\right)$.

Steps 6 and 8 have runtime $\tilde{O}(m)$, Step 7 has runtime $\tilde{O}(n)$ and Step 12 is done m times every iteration. By Lemma 22 the algorithm stops after $\tilde{O}(m)$ iterations, so the total runtime is $\tilde{O}(m^2)$. □

3.3 A speedup by spectral sparsification

In Section 2.3 we saw that vertex-flow sparsifiers are also vertex-cut sparsifiers. Therefore, if we are interested in finding a vertex-cut sparsifier, then we can also find one in time $\tilde{O}(m^2)$ using the same algorithm as for vertex-flow sparsification. Actually, it can be sped up by doing edge sparsification beforehand. Let us define spectral sparsification.

Definition 24. *Let $G=(V,E,c)$ be a capacitated graph. The **graph Laplacian** L_G of G is the $|V|\times|V|$ matrix where the rows and columns are indexed by the vertices V, and the entries are as follows:*

- *For diagonal elements we have that $L_G(u,u)$ is the sum of all the capacities of edges connected to u.*
- *For the off-diagonal elements $u\neq v\in V$ we have that $L_G(u,v)=-c(u,v)$, where we say $c(u,v)=0$ if $(u,v)\notin E$.*

Definition 25. *Let $G=(V,E,c)$ be a capacitated graph, and let L_G be its graph Laplacian. An ϵ-**spectral sparsifier** of G is a sparse graph H on the same vertices such that*

$$(1-\epsilon)L_G \preceq L_H \preceq (1+\epsilon)L_G,$$

where $A\preceq B$ means that $B-A$ is positive semidefinite.

A graph H is sparse if the number of edges in H is low. It turns out spectral sparsifiers exist and they can be found efficiently.

Theorem 26 (Spielman and Teng [27])**.** *Let $G = (V, E, c)$ be a capacitated graph and let $n = |V|, m = |E|$ and $\epsilon > 0$. There exists a ϵ-spectral sparsifier $H = (V, E_H, c_H)$ such that $|E_H| = \tilde{O}(n/\epsilon^2)$. There is a classical algorithm that finds a spectral sparsifier with this property in time $\tilde{O}(m)$.*

Note that the capacities of H could be different than the capacities in G.

If we could do spectral sparsification before doing vertex sparsification, then Algorithm 2 would run in time $\tilde{O}(n^2)$, which is faster than $\tilde{O}(m^2)$. If we are just interested in finding a vertex-cut sparsifier, then we could do this by the following lemma.

Lemma 27. *An ϵ-spectral sparsifier approximates all cuts by a factor $1 \pm \epsilon$.*

Proof. Let G be a graph and H an ϵ-spectral sparsifier of G. We know that

$$(1 - \epsilon)L_G \preceq L_H \preceq (1 + \epsilon)L_G.$$

This implies that for every vector $x \in \mathbb{R}^n$ we have that

$$(1 - \epsilon)x^T L_G x \leq x^T L_H x \leq (1 + \epsilon)x^T L_G x. \tag{3.7}$$

For some $S \subset V$, let us define a vector χ_S such that $\chi_S(u) = 1$ for $u \in S$ and $\chi_S(u) = 0$ for $u \in V \setminus S$. We have that

$$\begin{aligned}
\chi_S^T L_G \chi_S &= \sum_{u \in S} L_G(u, u) + \sum_{u,v \in S; u \neq v} L_G(u, v) \\
&= \sum_{u \in S, v \in V} c_G(u, v) - \sum_{u,v \in S; u \neq v} c_G(u, v) \\
&= \sum_{u \in S, v \in V \setminus S} c_G(u, v) \\
&= h_G(S),
\end{aligned}$$

which is the cut function. By Equation (3.7) this implies that the cuts are ϵ-approximated in H. □

Combining Theorem 7, Theorem 23 and Lemma 27 we find the next theorem.

Theorem 28. *There exists a classical algorithm that computes a vertex-cut sparsifier of quality $O\left(\frac{\log k}{\log \log k}\right)$ in runtime $\tilde{O}(n^2)$.*

4 A quantum speedup

The classical Algorithm 2 finds a vertex-flow sparsifier of quality $O\left(\frac{\log k}{\log\log k}\right)$ in runtime $\tilde{O}\left(m^2\right)$. In this section we will give a quantum speedup. We present a quantum algorithm with runtime $\tilde{O}\left(m^{\frac{11}{6}}n^{\frac{1}{6}}\right)$, which is a small polynomial speedup for dense graphs.

4.1 Computational model

Before discussing our quantum algorithm, we need to specify the computational model we are using. The model we use is a classical computer that can use a quantum computer for subroutines. The input of the algorithm is stored in a **QROM** (quantum-readable classical-writable classical memory). The classical computer can also write bits to a **QRAM** (quantum-readable classical-writable classical memory). The QRAM and QROM allow quantum queries. This means that if an N-bit string $a_0 \dots a_{N-1}$ is stored, then we can apply an oracle O_a such that

$$O_a : \ |i\rangle |0\rangle \to |i\rangle |a_i\rangle \tag{4.1}$$

for every $i \in \{0, \dots, N-1\}$. The number of times such a query is applied in an algorithm is called the **query complexity**. A quantum computer can also implement elementary quantum gates that are not a query. The classical computer sends to the quantum computer what quantum queries and what other quantum gates to implement for a specific subroutine. At the end of the subroutine, the quantum computer measures the qubits in the computational basis and sends the outcome to the classical computer.

We define the **runtime** of an algorithm as the sum of the number of elementary quantum gates, the number of quantum queries and the classical runtime of the classical subroutines.

4.2 Quantum preliminaries

Before we can discuss our quantum algorithm, we need to have a look at some quantum subroutines we want to use. One of the most important quantum algorithms is **Grover's algorithm** [15]. We will discuss a version of this algorithm that is slightly more complicated. We define $[N] = \{0, 1, \dots, N-1\}$.

Theorem 29 (Boyer, Brassard, Høyer and Tapp [4])**.** *Let $F : [N] \to \{0,1\}$ be a map. Let t be the number of entries i such that $F(i) = 1$. There is a quantum algorithm*

that outputs uniformly at random an element from the set $\{i \mid F(i) = 1\}$ *in expected runtime* $O(\sqrt{N/t})$.

Note that Theorem 29 does not require that we know t beforehand. In our quantum algorithm we will need a subroutine to find the minimum of a function. This can be done using Algorithm 3.

Algorithm 3 Finding the minimum of $F : [N] \to \mathbb{R}$

1: Choose $j \in [N]$ uniformly at random
2: **repeat**
3: Find an index $i \in [N]$ such that $F(i) < F(j)$
4: $j \leftarrow i$
5: **until** there is no such i
6: **return** j

Theorem 30 (Dürr and Høyer [10])**.** *If* $F : [N] \to \mathbb{R}$ *is a function that we have query access to, then Algorithm 3 finds the minimum output of* F*. The expected runtime of the algorithm is* $O(\sqrt{N})$.

Proof. It is clear that the j that is returned is a minimum, so we just need to analyze the runtime.

We assume that F is injective. We define the **rank** of an index $j \in [N]$ as the number of indices i such that $F(i) \leq F(j)$. When running the algorithm, there is a probability that at some point in the algorithm we get to an index j that has rank r, we define p_r as this probability.

For some $r \geq 1$, let i be the first index that we find running the algorithm such that $\text{rank}(i) \leq r$. We know with probability 1 that this i exists, since we will end with the index that has rank 1. Since in Step 3 we find an index with lower rank uniformly at random, the probability that i has rank r is $\frac{1}{r}$, so $p_r = \frac{1}{r}$.

If at some point in the algorithm we find an index i that has rank r, then the next step will have expected runtime upper bounded by $C\sqrt{N/(r-1)}$ for some constant $C \geq 0$ by Theorem 29. We can easily find a constant $c > 0$ such that $C\sqrt{N/(r-1)} < c\sqrt{N/r}$ for all $r \geq 2$. We can thus upper bound the total expected runtime by

$$\sum_{r=2}^{N} p_r c\sqrt{N/r} = c\sqrt{N}\sum_{r=2}^{N} r^{-3/2} = O\left(\sqrt{N}\right)$$

where we used the fact that $\sum_{r=1}^{\infty} r^{-p} < \infty$ for all $p > 1$. □

Now that we have discussed some basic quantum algorithms directly based on Grover's algorithm, we can look at more advanced quantum algorithms applied to graph theory problems. In such quantum algorithms we need some type of query access to the graph, similar to Equation 4.1. We discuss two query models.

- In the **adjacency model** the graph is given as the adjacency matrix $A \in \mathbb{R}^{n \times n}$ where n is the number of vertices of our graph. We have $A_{u,v} = c(u,v)$ if there is an edge connecting u, v with capacity $c(u,v)$, and $A_{u,v} = 0$ if there is no edge connecting u, v or if $u = v$. An oracle over the QROM in which such a matrix is stored looks like
$$O_A : \ |i,j\rangle |0\rangle \to |i,j\rangle |A_{i,j}\rangle ,$$
where the second register contains enough qubits to write down $A_{i,j}$ with enough precision.

- In the **adjacency array model** we are given the degrees d_i of the vertices v_i, and for every vertex v_i an array of length d_i of its neighbors v_j and the capacity of the edges connecting v_i to its neighbor v_j. The oracle looks like
$$O_{ad} : \ |i\rangle |a\rangle |0\rangle |0\rangle \to |i\rangle |a\rangle |j\rangle |c(v_i, v_j)\rangle ,$$
where $a \leq \text{Degree}(v_i)$, and the last register contains enough qubits to write down $c(v_i, v_j)$ with enough precision.

We note that, given access to one of these models, we can convert to the other in runtime $\tilde{O}(n^2)$. We can do this before running our algorithm, and our algorithm runs slower than $\tilde{O}(n^2)$, so for the various steps in our algorithm we can choose which one of these models to use.

Dürr, Heiligman, Høyer and Mhalla [9] wrote a useful paper about quantum algorithms in graph theory. They give quantum speedups for various important classical problems from graph theory. One of these speedups will be of use to us, namely the speedup for Dijkstra's algorithm.

Theorem 31 (Theorem 18 by Dürr, Heiligman, Høyer and Mhalla [9])**.** *There is a bounded-error quantum algorithm with worst-case runtime $\tilde{O}(\sqrt{nm})$ that finds the shortest path from one given vertex v to all other vertices. This algorithm uses the array model.*

Dürr, Heiligman, Høyer and Mhalla [9] state in their Theorem 18 that the query complexity of their algorithm is $\tilde{O}(\sqrt{nm})$. In their introduction they state that the runtime is at most a polylogarithmic factor larger than the query complexity.

An algorithm is bounded-error if for every possible input the probability of giving a correct output is at least $2/3$, but this can be improved to any constant in $[2/3, 1)$. We need one more subroutine for our quantum algorithm.

Theorem 32 (van Apeldoorn, Gribling and Nieuwboer [29])**.** *Let $v \in [0,1]^A$. Let $\rho, \Delta \in (0,1)$ be such that $\log(1/\rho)\Delta = O(A)$. Then there is a quantum algorithm that, with probability $\geq 1 - \rho$, finds a multiplicative Δ-approximation of $\sum_{i=1}^{A} v_i$ with query complexity*
$$O\left(\sqrt{\frac{A}{\Delta}\log(1/\rho)}\right),$$
and a runtime which is larger by a factor polylogarithmic in $A, 1/\Delta$ and $1/\rho$.

4.3 Our algorithm

The bottlenecks of Algorithm 2 are Steps 6, 8 and 12, that each have runtime $\tilde{O}(m)$.

Step 8 from the classical algorithm can be sped up easily using Theorem 30, giving an improvement of Lemma 21.

Lemma 33. *There is a quantum algorithm with expected runtime $\tilde{O}(\sqrt{m})$ that finds $\ell_{H\times R} := \max_e\{\text{rload}_{H\times R}(e)\}$.*

Proof. As we saw in the proof of Lemma 21 we can calculate $\text{rload}_{H\times R}(e)$ in time $O(k^2)$ for a given $e \in E$. We now use Theorem 30 to find the largest $\text{rload}_{H\times R}(e)$ over all $e \in E$. This search has runtime $\tilde{O}(\sqrt{m})$. In total we thus get runtime

$$\tilde{O}(k^2\sqrt{m}) = \tilde{O}(\sqrt{m}).$$

□

In Step 6 of the classical algorithm we use Dijkstra's algorithm, which has runtime $\tilde{O}(m)$. This can be improved by Theorem 31, which has runtime $\tilde{O}(\sqrt{nm})$.

The only bottleneck left in the classical algorithm is Step 12. To speed it up, we define an integer A. The idea is to calculate $d_{\vec{\lambda}}(e)$ only once every A iterations. Doing this we get Algorithm 4.

Algorithm 4 Quantum speedup for Algorithm 2

1: $\vec{\lambda} \leftarrow 0 \in \mathbb{R}_+^{\mathcal{H}\times\mathcal{R}}$
2: **for** $e \in E$ **do**
3: $\quad d_{\vec{\lambda}}(e) \leftarrow \frac{1}{c(e)}$
4: $i \leftarrow 0$
5: **while** $\sum_{H\times R} \lambda_{H\times R} < \ln m$ **do** ▷ $\tilde{O}(m)$ iterations
6: $\quad$ **for** $a \in K$ **do**
7: $\quad\quad$ Compute $\delta(v,a), R_{a,b}, \forall v \in V, \forall b \in K$ ▷ $\tilde{O}(A\sqrt{mn})$
8: $\quad$ Find $H \in \mathcal{H}$ using Theorem 17 ▷ $\tilde{O}(n)$
9: $\quad \ell_{H\times R} \leftarrow \max_e\{\text{rload}_{H\times R}(e)\}$ ▷ $\tilde{O}(\sqrt{m})$
10: $\quad \delta_{H\times R} \leftarrow \min\left\{1/\ell_{H\times R},\ \ln m - \sum_{H\times R}\lambda_{H\times R}\right\}$
11: $\quad \lambda_{H\times R} \leftarrow \lambda_{H\times R} + \delta_{H\times R}$
12: $\quad$ **if** $A \mid i$ **then**
13: $\quad\quad$ **for** $e \in E$ **do** ▷ m iterations
14: $\quad\quad\quad d_{\vec{\lambda}}(e) \leftarrow d_{\vec{\lambda}}(e) \cdot \exp\left(\left(\sum_{j=i-A+1}^{i} \delta_j \,\text{rload}_j(e)\right)(1 \pm \Delta)\right)$ ▷ $\tilde{O}(\sqrt{A})$
15: $\quad i \leftarrow i + 1$
16: **return** the non-zero entries of $\vec{\mu} = \vec{\lambda}/\ln m$

Not knowing the edge-lengths at every step forms a problem for Step 7 of the quantum algorithm, where we need $d_{\vec{\lambda}}$ to calculate $D(v,a)$ for all $v \in V, a \in K$. To fix this, we will redefine the query used in Theorem 31.

We write d_i, H_i, R_i for respectively the edge-lengths at the start of iteration i, and the 0-extension and shortest paths that we find in Steps 8 and 7 of iteration i. For simplicity we define $\delta_i \coloneqq \delta_{H_i \times R_i}$ and $\text{rload}_i \coloneqq \text{rload}_{H_i \times R_i}$.

Suppose that for some integer N we know d_{NA}, i.e. all edge-lengths at the start of iteration NA. Assume that in Algorithm 4 we are at iteration number $AN + a$ for some $0 \leq a < A$, and we know δ_i, H_i, R_i for integers $NA < i \leq NA + a$. To store H_i for every $i \in \{NA + 1, \ldots, NA + a\}$ it suffices to have storage space $\tilde{O}(na) = \tilde{O}(nA)$ since a 0-extension $H \in \mathcal{H}$ is defined as a map $V \to K$. To store R_i for a specific $i \in \{NA + 1, \ldots, NA + a\}$ we should store k^2 paths. A path has length at most n, so we can store R_i in storage space $\tilde{O}(nk^2) = \tilde{O}(n)$, so to store R_i for every $i \in \{NA+1, \ldots, NA+a\}$ it suffices to have $\tilde{O}(nA)$ storage space. Since the δ_i are just real numbers, $\tilde{O}(nA)$ storage space suffices to store δ_i, H_i, R_i for all $i \in \{NA+1, \ldots, NA+A\}$.

For some edge $e \in E$ we can now calculate

$$d_{NA+a}(e) = d_{NA}(e) \exp\left(\sum_{i=AN+1}^{AN+a} \delta_i \, \text{rload}_i(e)\right) \tag{4.2}$$

in time $\tilde{O}(a) = \tilde{O}(A)$. In Theorem 31 we use a query to access the graph. In our case we can not directly implement this query. We need an extra factor of $\tilde{O}(A)$ in the query complexity, since that is the runtime to calculate $d_{NA+a}(e)$ for a given edge $e \in E$. Finding the shortest paths can thus be done in runtime $\tilde{O}(A\sqrt{nm})$. Note that the calculation done in Equation (4.2) is deterministic, so we do not need to worry about error probabilities here.

Every A iterations we will calculate $d_{\vec{\lambda}}(e) = d_{NA}(e)$ for every edge $e \in E$. We could do this as in Equation (4.2), choosing $a = A$. However, this would result in a runtime of $\tilde{O}(Am)$ to calculate all edge-lengths, which would not give us a speedup. We need to find a smarter way to calculate the exponent $\exp\left(\sum_{i=AN+1}^{AN+A} \delta_i \, \text{rload}_i(e)\right)$. We do this using Theorem 32.

We note that for every $i \in \{AN + 1, \ldots, AN + A\}$ we have that $\delta_i \, \text{rload}_i(e) \in [0, 1]$, since in the algorithm we defined $\delta_i \leq \frac{1}{\max_e \{\text{rload}_i(e)\}}$. We can thus use Theorem 32 to calculate $\sum_{i=AN+1}^{AN+A} \delta_i \, \text{rload}_i(e)$ up to a factor $1 \pm \Delta$ in time $O\left(\sqrt{\frac{A}{\Delta} \log(1/\rho)}\right)$ with error probability ρ. If we already would know $\sum_{i=1}^{AN} \delta_i \, \text{rload}_i(e)$ up to a factor $1 \pm \Delta$, then we could calculate

$$\sum_{i=1}^{AN+A} \delta_i \, \text{rload}_i(e) = \sum_{i=1}^{AN} \delta_i \, \text{rload}_i(e) + \sum_{i=AN+1}^{AN+A} \delta_i \, \text{rload}_i(e)$$

up to a factor $1 \pm \Delta$ as well, since $\delta_i \, \text{rload}_i(e) \geq 0$ at every iteration i.

It turns out that this factor $1 \pm \Delta$ does not affect the outcome of the algorithm in a significant way if we choose Δ appropriately.

In the analysis of the classical algorithm we assumed that $d_{\vec{\lambda}}(e) = \frac{\exp\left((M\vec{\lambda})e\right)}{c(e)}$, but in our case we can have an error. The edge-lengths $d_{\vec{\lambda}}(e)$ that we compute using Theorem 32 will satisfy

$$\frac{\exp\Big((M\vec{\lambda})e(1-\Delta)\Big)}{c(e)} \leq d_{\vec{\lambda}}(e) \leq \frac{\exp\Big((M\vec{\lambda})e(1+\Delta)\Big)}{c(e)}, \forall e \in E. \tag{4.3}$$

This holds at every iteration. A problem arises in Step 8 of the quantum algorithm. In the following lemma we show that the error we get in the edge-lengths does not need to be a problem if we choose Δ small enough.

Lemma 34. *Let $G = (V, E, c)$ be a capacitated graph with terminals $K \subset V$ that we input in Algorithm 4. We define $m = |E|, k = |K|$. There is a constant $C > 0$ such that if we choose*

$$\Delta = C\frac{\log\log k}{\log k \cdot \log m},$$

then Algorithm 4 outputs a vertex-flow sparsifier of quality

$$q = O\left(\frac{\log k}{\log\log k}\right).$$

Proof. Note that the proof of Theorem 22 also holds for Algorithm 4. The algorithm will thus stop after a number of iterations and output a convex combination of 0-extensions. It only remains to compute the quality.

As in Algorithm 2, every iteration i we find an $H \times R \in \mathcal{H} \times \mathcal{R}$ such that $\frac{\partial \operatorname{lmax}(M\vec{\lambda})}{\partial \lambda_{H\times R}}$ is small, where $\vec{\lambda}$ is as we found it at at the start of iteration i. We can upper bound this partial derivative, inspired by Equation (3.6) and using Equation (4.3) as follows

$$\begin{aligned}
\frac{\partial \operatorname{lmax}(M\vec{\lambda})}{\partial \lambda_{H\times R}} &= \sum_e \operatorname{load}_{H\times R}(e)\frac{\exp\Big((M\vec{\lambda})_e\Big)}{c(e)\sum_{e'}\exp\Big((M\vec{\lambda})_{e'}\Big)} \\
&\leq \sum_e \operatorname{load}_{H\times R}(e)\frac{d_{\vec{\lambda}}(e)}{\sum_{e'} c(e')d_{\vec{\lambda}}(e')} \cdot \exp\left(2\max_{e''}\{(M\vec{\lambda})_{e''}\}\Delta\right) \\
&= \frac{\sum_{a,b\in K} c_H(a,b)\sum_{e\in R_{a,b}} d_{\vec{\lambda}}(e)}{\sum_{e'} c(e')d_{\vec{\lambda}}(e')} \cdot \exp\left(2\max_{e''}\{(M\vec{\lambda})_{e''}\}\Delta\right) \\
&\leq \exp\left(2\max_e\{(M\vec{\lambda})_e\}\Delta\right) \cdot S\frac{\log k}{\log\log k}.
\end{aligned} \tag{4.4}$$

In the last step we used that the $H \in \mathcal{H}$ that we used came from the algorithm described in Theorem 17, and we choose the constant $S > 0$ appropriately.

Let us define the map

$$F: \ x \to \exp(x/3).$$

We calculate numerically that $F(x) - x = 0$ has roots $x_0 \approx 1.9$ and $x_1 \approx 4.5$. We note that for every $x < x_0$ we have that $F(x) < x_0$ and $1 < x_0$. Therefore, $F^{\circ i}(1) < x_0$ for every $i \geq 0$, where $F^{\circ i}$ is defined as iterating the map i times.

We choose $C = \frac{1}{12(S+1)}$. We now claim that for every iteration $i \geq 0$ we have that the vector $\vec{\lambda}$ that we find at the start of iteration i satisfies

$$\operatorname{lmax}(M\vec{\lambda}) \leq F^{\circ i}(1)\left(\ln m + 2S\frac{\log k}{\log\log k} \cdot \sum_{H\times R} \lambda_{H\times R}\right). \tag{4.5}$$

We prove the claim using induction on i.

For $i = 0$ we have $\operatorname{lmax}(0) = \ln m$, so the induction basis holds.

We now assume that Equation (4.5) holds at the start of iteration i for given $i \geq 0$. We know that

$$2\max_e\{(M\vec{\lambda})_e\} \leq 2\operatorname{lmax}(M\vec{\lambda}) \leq F^{\circ i}(1)\frac{4(S+1)\log k}{\log\log k}\log m = F^{\circ i}(1)\Delta^{-1}/3. \tag{4.6}$$

Combining Equation (4.4) and Equation (4.6) we find that

$$\begin{aligned}\frac{\partial \operatorname{lmax}(M\vec{\lambda})}{\partial \lambda_{H\times R}} &\leq \exp\bigl(F^{\circ i}(1)/3\bigr) \cdot S\frac{\log k}{\log\log k} \\ &= F^{\circ(i+1)}(1) \cdot S\frac{\log k}{\log\log k}.\end{aligned}$$

If $\vec{\delta} = \delta e_{H\times R}$ is the vector we find in Step 10 at iteration i, then we have that

$$\begin{aligned}\operatorname{lmax}(M(\vec{\lambda}+\vec{\delta})) &\leq \operatorname{lmax}(M\vec{\lambda}) + 2\delta\frac{\partial \operatorname{lmax}(M\vec{\lambda})}{\partial \lambda_{H\times R}} \\ &\leq F^{\circ(i+1)}(1)\left(\ln m + 2S\frac{\log k}{\log\log k} \cdot \left(\delta + \sum_{H\times R}\lambda_{H\times R}\right)\right),\end{aligned}$$

where we used Lemma 19 and the fact that $F^{(i+1)}(1) \geq F^i(1)$. This proves that Equation (4.5) holds at the start of iteration $i+1$, and the induction step holds. We thus proved the claim.

The algorithm ends when $\sum_{H\times R}\lambda_{H\times R} = \ln m$. If we take $\vec{\mu} = \vec{\lambda}/\ln m$, then we get that

$$(M\vec{\mu})_e = F^{\circ I}(1) \cdot O\left(\frac{\log k}{\log\log k}\right) = O\left(\frac{\log k}{\log\log k}\right).$$

for every $e \in E$, where I is the total number of iterations. This proves that $\vec{\mu}$ is a vertex-flow sparsifier with quality $q = O\left(\frac{\log k}{\log\log k}\right)$. □

Let $N = \tilde{O}(m^2/A)$ denote the number of times that Theorem 32 is used in Algorithm 4. We can use the union bound to upper bound the overall error probability by the sum of the individual error probabilities $\sum_{i=1}^{N}\rho = N\rho$. If we want the overall error probability to be smaller than some $\epsilon > 0$, then it suffices to have

$$N\rho < \epsilon.$$

The ρ-dependence of the runtime in Theorem 32 is $O\left(\log^{1/2}(1/\rho)\right)$. To have the total error probability of every iterations of Step 14 upper bounded by ϵ, an extra factor of $O\left(\log^{1/2}\left(\frac{\epsilon}{N}\right)\right)$ thus suffices in the runtime. This term will not make a difference for our final result since we absorb log-factors in the $\tilde{O}$-notation.

The runtime of Algorithm 4 is now dominated by Step 7 and Step 14. Step 7 has runtime $\tilde{O}(A\sqrt{mn})$ and Step 14 has runtime $O\left(m\sqrt{\frac{A}{\Delta}\log(1/\rho)}\right) = \tilde{O}(m\sqrt{A})$, but needs to be done only once every A iterations. There are $\tilde{O}(m)$ iterations in total, so the runtime becomes $\tilde{O}\left(\sqrt{mn}A \cdot m + m\sqrt{A} \cdot m/A\right) = \tilde{O}\left(\sqrt{mn}mA + m^2/\sqrt{A}\right)$. If we choose $A = \left(\frac{m}{n}\right)^{\frac{1}{3}}$, then we get runtime $\tilde{O}\left(m^{\frac{11}{6}} n^{\frac{1}{6}}\right)$. If m is larger than n by more than a polylogarithmic factor, then we have a speedup, and then we have that $\log(1/\rho)\Delta = O(A)$ as was required by Theorem 32. We conclude Theorem 1, which is the main result of this thesis. As we saw in the discussion above Equation (4.2), the space complexity of the algorithm is $\tilde{O}(mA) = \tilde{O}(m^{7/6}n^{-1/6})$. We thus need $O(\log n)$ qubits.

5 Conclusion and future work

5.1 Conclusion

Vertex sparsification is useful when we have a really large graph but we are only interested in a few terminals. In Chapter 2 we discussed three different definitions of vertex sparsifiers.

- Vertex-flow sparsification. The goal in this case is that for every demand vector on the terminals, the congestion is approximately preserved.
- Vertex-cut sparsification. The goal in this case is that the value of every minimum cut between subsets of terminals is approximately preserved.
- Connectivity-c mimicking networks. The goal in this case is that cuts smaller than some constant c are exactly preserved. We do not care about larger cuts.

We focus on vertex-flow sparsifiers and vertex-cut sparsifiers. In the special case where we require the vertex sparsifier to have a vertex set equal to the set of terminals, every vertex-flow sparsifier is also a vertex-cut sparsifier. A theorem by Moitra [22] states that vertex-flow sparsifiers exist with a quality of $q = O\left(\frac{\log k}{\log \log k}\right)$.

In Chapter 3 we discussed a classical algorithm to find vertex-flow sparsifiers of the quality promised by Moitra. The algorithm was first stated by Englert, Gupta, Krauthgamer, Räcke, Talgam-Cohen and Talwar [11]. The algorithm constructs a convex combination of 0-extensions. It starts with the empty combination of 0-extensions with coefficient-vector $\vec{\lambda} = 0$. In every iteration of the algorithm, one 0-extension is chosen and its index is raised by a small δ. We chose which 0-extension to raise by looking at $\text{lmax}(M\vec{\lambda})$, where M is a matrix defined in a way that the entries of $M\vec{\lambda}$ are exactly the congestion of the various edges. We want to keep $\text{lmax}(M\vec{\lambda})$ small, because it is an upper bound for the congestion on every edge. To find a suitable 0-extension, an algorithm by Fakcharoenphol, Harrelson, Rao and Talwar [12] is used, that finds a 0-extension that keeps the partial derivative of the lmax small enough. The classical algorithm has runtime $\tilde{O}(m^2)$. This can be sped up to $\tilde{O}(n^2)$ in case we are only interested in finding vertex-cut sparsifiers.

In Chapter 4 we presented our own work: a quantum speedup of the algorithm presented in Chapter 3. There are three steps in the classical algorithm that form its bottlenecks.

- Finding the shortest paths between vertices and terminals can be done classically using Dijkstra's algorithm with a runtime $\tilde{O}(m)$. A quantum algorithm by Dürr, Heiligman, Høyer and Mhalla [9] does the same in runtime $\tilde{O}(\sqrt{mn})$.

- Finding the real number δ that we will add to the combination of 0-extensions $\vec{\lambda}$ takes $\tilde{O}(m)$ time classically. This can be sped up using the minimum finding algorithm by Dürr and Høyer [10].

- In the classical algorithm we compute every edge-length, every iteration. This step has a runtime of $\tilde{O}(m)$ every iteration. To speed this up, we compute the edge-lengths only every A iterations, for some integer A. In the meantime, we can still find the shortest paths by calculating the edge-lengths on the spot when needed. This results in a longer runtime for the single-source shortest path step. To calculate all edge-lengths once per every A iterations, we use a subroutine by van Apeldoorn, Gribling and Nieuwboer [29]. This subroutine approximates the sum of some real numbers faster than can be done classically, but it gives an error. This error can be made small enough for the algorithm to still output a vertex-flow sparsifier of good quality.

Combining all these techniques gives us Algorithm 4, the main result of this thesis: a quantum algorithm that finds vertex-flow sparsifiers of good quality with a runtime $\tilde{O}(m^{11/6}n^{1/6})$. This beats the classical $\tilde{O}(m^2)$-runtime algorithm by a small polynomial factor for dense graphs.

5.2 Future work

In this section we state a few ideas for future work.

- As discussed in Section 3.3, we can speed up Algorithm 2 in the case that we are only interested in preserving cuts, doing spectral sparsification beforehand. The question arises if we could do something similar for vertex-flow sparsifiers. We are not aware of a proof that spectral sparsifiers preserve flows, but if it would be the case, then Theorem 28 would hold for vertex-flow sparsifiers as well. If it turns out that spectral sparsifiers do not preserve flows, then it would be interesting to see if there exists another notion of edge sparsification that does, and if implementing it beforehand would give us a speedup.

- The speedup in the main result of this thesis, Theorem 1, was found implementing Theorem 32 while calculating all edge-lengths every A iterations. Perhaps a similar speedup could be found by applying Theorem 32 when calculating the edge-lengths in Step 7 of the quantum algorithm.

  However, the queries in Step 7 happen in superposition over edges. Therefore, subsequent queries would give superpositions over approximations if we use Theorem 32 to approximate the edge-lengths in the query. More analysis of the algorithm is needed to determine whether this forms a problem or not. If, however, it turns out that it works, then implementing the trick would result in Step 7 having runtime $\tilde{O}(\sqrt{Amn})$ instead of $\tilde{O}(A\sqrt{mn})$. In the main Theorem 1 this would result in a runtime of $\tilde{O}\left(m^{7/4}n^{1/4}\right)$ instead of $\tilde{O}\left(m^{11/6}n^{1/6}\right)$, providing a better, but still small, polynomial speedup.

- In Section 2.4 we saw another definition of vertex sparsification: the connectivity-c mimicking networks. There exists a classical algorithm that finds them and runs in time $n^{O(1)}$, where n is the number of vertices of the original graph. It would be interesting to study what this polynomial exactly is and if it could be improved using a quantum algorithm.
- In this thesis we discussed algorithms to find vertex sparsifiers of capacitated graphs, without requiring anything about the graph. It might be possible to achieve a better runtime, either classical or quantum, for specific classes of graphs, such as trees, quasi-bipartite graphs or planar graphs. See the discussion in Section 2.5.

Popular summary

Imagine the train network of Europe. We can represent this train network as a huge **graph**. A graph is a mathematical structure that can be drawn as a collection of **vertices**, which we will draw as points, and **edges**, which we will draw as lines between the vertices.

Let us say that we are interested in the connections between five big European cities: Amsterdam, Paris, Berlin, Milan and Barcelona. In Figure 5.1 we draw a graph representing a part of the train network of Europe, showing the five cities that interest us as vertices of the graph. We draw an edge between cities that are connected by a direct train. There is no direct train connecting Berlin to Milan, so we did not draw an edge between those cities. The figure, however, is deceiving us! It looks like the only way to get from Berlin to Milan is to pass through Amsterdam and Paris. This of course is not true: We could also choose to take the train from Berlin to Basel, and from Basel to Milan. We do not see this in Figure 5.1 because we chose not to show Basel in the graph. We could choose to add Basel to the figure to solve the problem, but then a similar problem arises: do we need to pass through both Paris and Milan if we want to travel from Barcelona to Basel? The only way to solve the problem is to add many, if not all, the train stations of Europe to the graph. However, Germany alone already has about 5000 train stations. If we added all the train stations of Europe to our graph, it would become huge, and we would completely lose the nice overview that Figure 5.1 gives us!

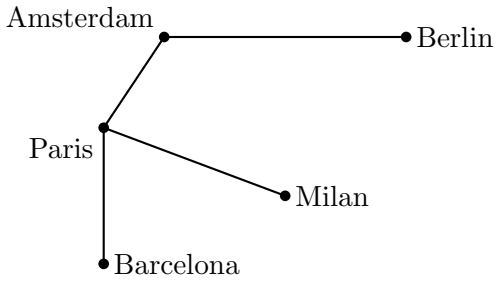


Figure 5.1: Some train connections in Europe

This is where **vertex sparsification** comes in. The goal of vertex sparsification is to represent a really large graph as a small graph. Instead of a large graph with all the train stations of Europe, we would have a small graph that we could put in our pocket if we wanted. Figure 5.1 is an example of such a vertex sparsifier, only showing the five cities we are interested in. However, as we note when traveling from Berlin to Milan, it is not a really good vertex sparsifier. In previous research it was shown that there exists a classical algorithm to find better ones. In this thesis we present a new quantum algorithm that runs faster.

Acknowledgements

I would like to thank my supervisor Ronald de Wolf for all the feedback, ideas and discussions that we have had in the last few months about quantum computing and academic life. More than a few times I thought that I found a mistake that would ruin my entire thesis, but every time, with help from Ronald, we managed to find a solution.

I would also like to thank Ankur Moitra, for helpfully answering a question about his PhD thesis [23].

Bibliography

[1] S. Apers and R. de Wolf. Quantum speedup for graph sparsification, cut approximation, and Laplacian solving. *SIAM Journal on Computing*, 51(6):1703–1742, 2022.

[2] A. A. Benczúr and D. R. Karger. Approximating st minimum cuts in Õ(nˆ2) time. In *Proceedings of the 28th ACM Symposium on Theory of computing*, pages 47–55, 1996.

[3] J. A. Bondy and U. S. R. Murty. *Graph theory with applications*, volume 290. Macmillan London, 1976.

[4] M. Boyer, G. Brassard, P. Høyer, and A. Tapp. Tight bounds on quantum searching. *Fortschritte der Physik: Progress of Physics*, 46(4-5):493–505, 1998. Wiley Online Library.

[5] P. Chalermsook, S. Das, Y. Kook, B. Laekhanukit, Y. P. Liu, R. Peng, M. Sellke, and D. Vaz. Vertex sparsification for edge connectivity. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1206–1225.

[6] M. Charikar, F. T. Leighton, S. Li, and A. Moitra. Vertex Sparsifiers and Abstract Rounding Algorithms. In *2010 IEEE 51st Symposium on Foundations of Computer Science*, pages 265–274.

[7] L. Chen, R. Kyng, Y. P. Liu, R. Peng, M. P. Gutenberg, and S. Sachdeva. Maximum flow and minimum-cost flow in almost-linear time. In *2022 IEEE 63rd Symposium on Foundations of Computer Science (FOCS)*, pages 612–623.

[8] R. de Wolf. Quantum computing: Lecture notes. *arXiv:1907.09415*, 2019.

[9] C. Dürr, M. Heiligman, P. Høyer, and M. Mhalla. Quantum query complexity of some graph problems. *SIAM Journal on Computing*, 35(6):1310–1328, 2006.

[10] C. Dürr and P. Høyer. A quantum algorithm for finding the minimum. *arXiv preprint quant-ph/9607014*, 1996.

[11] M. Englert, A. Gupta, R. Krauthgamer, H. Räcke, I. Talgam-Cohen, and K. Talwar. Vertex sparsifiers: New results from old techniques. *SIAM Journal on Computing*, 43(4):1239–1262, 2014.

[12] J. Fakcharoenphol, C. Harrelson, S. Rao, and K. Talwar. An improved approximation algorithm for the 0-extension problem. In *Proceedings of the 14th ACM-SIAM Symposium on Discrete Algorithms*, pages 257–265, 2003.

[13] G. Goranci, M. Henzinger, and P. Peng. Improved guarantees for vertex sparsification in planar graphs. *SIAM Journal on Discrete Mathematics*, 34(1):130–162, 2020.

[14] G. Goranci and H. Räcke. Vertex sparsification in trees. In *International Workshop on Approximation and Online Algorithms*, pages 103–115. Springer, 2016.

[15] L. K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the 28th ACM Symposium on Theory of computing*, pages 212–219, 1996.

[16] T. Hagerup, J. Katajainen, N. Nishimura, and P. Ragde. Characterizing multiterminal flow networks and computing flows in networks of small treewidth. *Journal of Computer and System Sciences*, 57(3):366–375, 1998. Elsevier.

[17] D. R. Karger. Using randomized sparsification to approximate minimum cuts. In *Proceedings of the 5th Symposium on Discrete Algorithms*, volume 424, page 432, 1994.

[18] A. V. Karzanov. Minimum 0-extensions of graph metrics. *European Journal of Combinatorics*, 19(1):71–101, 1998.

[19] R. Krauthgamer and R. Mosenzon. Exact flow sparsification requires unbounded size. In *Proceedings of the 2023 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2354–2367.

[20] F. T. Leighton and A. Moitra. Extensions and limits to vertex sparsification. In *Proceedings of the 42th ACM Symposium on Theory of computing*, pages 47–56, 2010.

[21] Y. P. Liu. Vertex sparsification for edge connectivity in polynomial time. *arXiv preprint arXiv:2011.15101*, 2020.

[22] A. Moitra. Approximation algorithms for multicommodity-type problems with guarantees independent of the graph size. In *Proceedings of the 50th IEEE Symposium on Foundations of Computer Science*, pages 3–12, 2009.

[23] A. Moitra. *Vertex sparsification and universal rounding algorithms.* PhD Thesis, Massachusetts Institute of Technology, 2011.

[24] H. Nagamochi and T. Ibaraki. A linear-time algorithm for finding a sparse k-connected spanning subgraph of ak-connected graph. *Algorithmica*, 7(1):583–596, 1992. Springer.

[25] H. Räcke. Optimal hierarchical decompositions for congestion minimization in networks. In *Proceedings of the 40th ACM Symposium on Theory of computing*, pages 255–264, 2008.

[26] P. W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing 1997*, 26(5):1484–1509, 1997.

[27] D. A. Spielman and S.-H. Teng. Spectral sparsification of graphs. *SIAM Journal on Computing*, 40(4):981–1025, 2011.

[28] C. Ugwuishiwu, U. Orji, C. Ugwu, and C. Asogwa. An overview of quantum cryptography and shor's algorithm. *International Journal of Advanced Trends in Computer Science and Engineering*, 9(5), 2020.

[29] J. van Apeldoorn, S. Gribling, and H. Nieuwboer. Basic quantum subroutines: finding multiple marked elements and summing numbers. *Quantum*, 8:1284, 2024. Verein zur Förderung des Open Access Publizierens in den Quantenwissenschaften.

[30] N. K. Vishnoi. Lx= b. *Foundations and Trends® in Theoretical Computer Science*, 8(1–2):1–141, 2013. Now Publishers, Inc.

[31] N. E. Young. Sequential and parallel algorithms for mixed packing and covering. In *Proceedings of the 42nd IEEE Symposium on foundations of computer science*, pages 538–546, 2001.