Karlsruhe Institute of Technology

An experimental study of a nearly-linear time Laplacian solver

Master's thesis of

Daniel Hoske

At the Department of Informatics
Institute of Theoretical Informatics

Reviewers:	Prof. Dr. Henning Meyerhenke
	Prof. Dr. Peter Sanders
Advisors:	Prof. Dr. Henning Meyerhenke
	Dr. Dimitar Lukarski

Time period: June 2014 to November 2014

Acknowledgements

I would like to express my deepest gratitude to my advisors Prof. Dr. Henning Meyerhenke and Dr. Dimitar Lukarski for their continued support and advice.

All the contributors to NetworKit also deserve my deepest thanks for building such a powerful and efficient toolkit for graph algorithms, without which this thesis would not have been possible.

Special thanks also go to every other proofreader of this thesis and anyone else who supported me during its writing in any shape or form.

Statutory Declaration

I hereby declare that this thesis is the result of my own work, that I used no other than the indicated references and resources, that all the information that has been taken directly or indirectly from other sources is indicated as such, and that I have regarded the statute of the Karlsruhe Institute of Technology on securing good scientific practice in its currently applicable version.

Karlsruhe, December 3, 2014

Abstract

Solving sparse Laplacian systems is a problem of significant practical importance. Spielman and Teng's [ST04] nearly-linear time Laplacian solver was, therefore, an important theoretical breakthrough that spawned a series of extensions and simplifications.

Although these solvers are an enormous theoretical achievement, as of this writing they have not been extensively validated in practice. In this thesis we seek to fill this gap by implementing and benchmarking the nearly-linear time Laplacian solver proposed by Kelner et al. [Kel+13] that is much simpler than Spielman and Teng's [ST04] original algorithm.

While we confirm that its running time grows nearly-linearly, the constant factor is so large that the solver performs poorly compared with common iterative solvers. In particular, we find that the convergence of the solver strongly depends on the stretch of a chosen spanning tree. As Papp [Pap14] observed, known spanning tree algorithms with provable stretch give poor stretch in practice and, therefore, result in slow convergence of the solver.

In addition, we show that using this solver as a preconditioner in common solvers causes convergence problems. However, it quickly dampens the high-frequency components of the error and could, therefore, work well as a smoother.

Overall, Spielman and Teng's [ST04] Laplacian solver did not prove to be competitive against common iterative solvers.

Deutsche Zusammenfassung

Das Lösen von linearen Gleichungssystemen auf dünnbesetzten Laplacematrizen ist von enormer praktischer Bedeutung. Der von Spielman und Teng [ST04] vorgestellte Laplacelöser mit fast-linearer Laufzeit war deshalb ein wichtiger theoretischer Durchbruch, dem viele Erweiterungen und Vereinfachungen folgten.

Obwohl diese Löser mit fast-linearer Laufzeit ein enormer theoretischer Erfolg waren, wurden sie bisher noch nicht ausführlich praktisch getestet. In dieser Abschlussarbeit füllen wir diese Lücke indem wir den Laplacelöser von Kelner et al. [Kel+13], eine wesentliche Vereinfachung von Spielman und Tengs [ST04] originalem Algorithmus, implementieren und benchmarken.

Wir bestätigen das fast-lineare Wachstum der Laufzeit des Lösers. Der konstante Faktor ist allerdings so groß, dass der Laplacelöser nicht mit üblichen Lösern mithalten kann. Insbesondere hängt die Konvergenz des Lösers stark von der Streckung eines gewählten Spannbaums ab. Wie Papp [Pap14] beobachtet hat, liefern die bekannten beweisbar guten Spannbaumalgorithmen in der Praxis schlechte Streckung. Dies resultiert in langsamer Konvergenz des Laplacelösers.

Ferner zeigen wir, dass die Verwendung des Lösers als Präkonditionierer Konvergenzprobleme verursacht. Trotzdem dämpft der Löser schnell hochfrequente Komponenten des Fehlers und könnte deshalb gut als Glätter fungieren.

Insgesamt hat sich Spielman und Tengs [ST04] Laplacelöser nicht als konkurrenzfähig gegenüber üblichen iterativen Lösern erwiesen.

Contents

1 Introduction

Solving linear systems has been one of the most important and well-studied problems in mathematics as it is widely applicable in engineering and the sciences. It was one of the central problems that electronic computers were first used for. As early as 1948 Alan Turing [Tur48] adapted classical approaches for solving linear systems to computers.

The most basic algorithm he adapted is the LU-decomposition. It takes $\mathcal{O}(n^3)$ time for solving a linear system $Ax = b$ where $A \in \mathbb{R}^{n \times n}$, $b \in \mathbb{R}^n$. While this approach works well for small systems, it quickly becomes infeasible as n grows. It is particularly problematic if we need to deal with matrices that are large but have few nonzero entries. We call a matrix with $o(n^2)$ nonzeroes *sparse*. Ideally, the required time for solving sparse systems would grow linearly with the number of nonzeroes m.

Research has been working towards this goal ever since. As a slight simplification we are usually happy with approximate solutions since the precision of numbers stored in a computer and the experimental data we use as input are limited anyway. Spielman and Teng [ST04], following an approach proposed by Vaidya [Vai90], achieved a breakthrough in this direction by devising a nearly-linear time algorithm for solving linear systems in symmetric diagonally dominant matrices.

Nearly-linear means in $\mathcal{O}\big(m \cdot \operatorname{polylog}(n) \cdot \log(1/\epsilon)\big)$ in this thesis, where $\operatorname{polylog}(n)$ is the set of real polynomials in $\log(n)$ and ϵ is the relative error $\|x - x_{\text{opt}}\|_A / \|x_{\text{opt}}\|_A$ we want for the solution $x \in \mathbb{R}^n$. Here $\|\cdot\|_A$ is the norm $\|x\|_A := \sqrt{x^T A x}$ given by A and $x_{\text{opt}} := A^+ b$ is an exact solution of the problem. A matrix $A = (a_{ij})_{i,j \in [n]} \in \mathbb{R}^{n \times n}$ is *symmetric* if $a_{ij} = a_{ji}$ for all $i, j \in [n]$; it is *diagonally dominant* if $|a_{ii}| \geqslant \sum_{j \neq i} |a_{ij}|$ for all $i \in [n]$.

Matrices that are both symmetric and diagonally dominant (SDD matrices) have a lot of practical applications: In elliptic PDEs [BHV08], maximum flows [Chr+11], sparsifying graphs [SS08] and many other areas [KM09; KMP12]. Thus, the restricted problem INV-SDD of solving linear systems on SDD matrices is still of significant importance.

Problem: INV-SDD
Given: SDD matrix $A \in \mathbb{R}^{n \times n}$ and vector $b \in \operatorname{im}(A)$.
Problem: Find an $x \in \mathbb{R}^n$ with $Ax = b$.

Although quite a few extensions and simplifications to Spielman and Teng's [ST04] nearly-linear time solver have been proposed, none of them has been validated in practice so far.

We seek to fill this gap by implementing and thoroughly analysing a variant of the algorithm proposed by Kelner et al. [Kel+13] that is easier to describe and implement than Spielman and Teng's [ST04] original algorithm.

In the remainder of this chapter we first describe related work (Section 1.1) and then outline how this thesis is structured (Section 1.2).

1.1 Related work

The work on nearly-linear time SDD solvers was started by Spielman and Teng's seminal paper [ST04]. It required a lot of sophisticated machinery: a multi-level approach [Vai90; Rei98] using recursive preconditioning, preconditioners based on low-stretch spanning trees [SW09] and spectral graph sparsifiers [SS08; KLP12].

Later papers extended this approach, both by making it simpler and by reducing the exponents of the polylogarithmic time factors.[1]

We focus on a simplified algorithm by Kelner et al. [Kel+13] that reinterprets INV-SDD as the problem of finding an electrical flow in a graph. It only needs low-stretch spanning trees and achieves $\mathcal{O}\left(m \log^2 n \log\log n \log(1/\epsilon)\right)$ time.

Spielman and Teng's algorithm crucially uses the low-stretch spanning trees first introduced by Alon et al. [Alo+95]. Elkin et al. [Elk+05] provide an algorithm for computing spanning trees with polynomial stretch in nearly-linear time. Specifically, they get a spanning tree with $\mathcal{O}(m \log^2 n \log\log n)$ stretch in $O(m \log^2 n)$ time. Abraham et al. [ABN08; AN12] later showed how to get rid of some of the logarithmic factors in both stretch and time.

Papp [Pap14] tested these algorithms in practice and showed that they do not usually result in spanning trees with lower stretch than a simple minimum-weight spanning tree computed with Kruskal's algorithm [Kru56] and that Elkin et al.'s original algorithm [Elk+05] achieves the best results among the provably good approaches. We use these low-stretch spanning trees in our implementation of Kelner et al's. [Kel+13] algorithm and compare their effectiveness.

Furthermore, we also compare our implementation to the classical conjugate gradient method [She94].

[1] Spielman provides a comprehensive overview of later work at http://www.cs.yale.edu/homes/spielman/precon/precon.html (accessed on September 14, 2014).

One of the many other variations on nearly-linear time SDD solvers we consider particularly interesting is the recursive sparsification approach by Peng and Spielman [PS14]. Together with a parallel sparsification algorithm, such as the one given by Koutis [Kou14], it yields a nearly-linear work parallel algorithm. Since we show in this thesis that Kelner et al.'s [Kel+13] algorithm is hard to parallelise and does not converge particularly fast, it would be interesting to benchmark how well this parallel approach performs.

1.2 Contributions and outline

From the literature analysis above we can see that there are several nearly-linear time SDD solvers. In this thesis we want to implement the solver by Kelner et al. [Kel+13] and analyse its practical performance:

Chapter 2 We start by giving basic definitions from spectral graph theory and from optimization theory that we need in the rest of the thesis. We also show that we can reduce INV-SDD to solving linear systems on Laplacians, a subclass of the SDD matrices. We then only consider Laplacian matrices.

Chapter 3 We continue by introducing the idea of the algorithm to interpret a linear system as an electrical flow problem, and we show how this interpretation leads to an algorithm for solving the linear system. Based on this interpretation we can give an overview of Kelner et al.'s proof [Kel+13] that the resulting algorithm converges in a number of steps that depends on the stretch of a spanning tree of the graph.

Chapter 4 In the following chapter we elaborate on the decisions we can make when implementing Kelner et al.'s [Kel+13] algorithm. In particular, we explain when these decisions result in a provably nearly-linear time algorithm.

Chapter 5 This chapter contains the heart of this thesis, the experimental evaluation of the Laplacian solver. We consider the configuration options of the algorithm, its asymptotics, its convergence and its use as a preconditioner or smoother. Furthermore, we explore other practical aspects such as the performance of the solver on a modern computer and whether it can be parallelised.

Chapter 6 We conclude the thesis by summarising the experimental results and discussing viable future research directions.

2 Preliminaries

The algorithm for solving INV-SDD investigated in this thesis combines knowledge from linear algebra, graph theory and optimization theory. In this chapter we introduce some basic definitions and notations from these fields. We will look at graphs and their matrices (Section 2.1), cycles and spanning trees (Section 2.2) as well as duality (Section 2.3). Section A also contains a list of more general notations.

Furthermore, we show that solving a INV-SDD system can, in fact, always be reduced to solving a Laplacian system (Section 2.4). We conclude by fixing some conventions for the rest of this thesis (Section 2.5).

2.1 Graphs and their matrices

A *graph* is a pair $G = (V, E)$ where V is a finite set and $E \subseteq \binom{V}{2}$. That is, we only consider undirected simple graphs. A graph is *weighted* if we have an additional function $w\colon E \to \mathbb{R}_{>0}$, i. e. the assigned weights need to be positive. When necessary we consider unweighted graphs to be weighted with $w_e = 1$ for every $e \in E$.

Conventions: We usually write an edge $\{u, v\} \in E$ as uv and its weight as w_{uv} instead of $w(uv)$. We denote the *order* $|V|$ of G by $|G|$. We also define the set operations $\cup$, $\cap$ and $\setminus$ on graphs by applying them to the set of vertices and the set of edges separately.

A *path in* G is a sequence of nodes $v_0, \dots, v_k \in V$ such that $v_{i-1}v_i \in E$ for all $i \in [k]$. It is *simple* if edges do not repeat. G is called *connected* if there is a path between any two nodes in V.

For every node $u \in V$ its *neighbourhood* $N_G(u)$ is the set $N_G(u) := \{v \in V : uv \in E\}$ of vertices v with an edge to u and its *degree* d_u is $d_u = \sum_{v \in N_G(u)} w_{uv}$.

For each graph $G = (V, E)$ we define the following matrices:

- The *adjacency matrix* $A(G) \in \mathbb{R}^{V \times V}$ of G is given by

$$A_{u,v} := \begin{cases} w_{uv} & \text{if } uv \in E \\ 0 & \text{otherwise} \end{cases}$$

  for all $u, v \in V$.

- The *degree matrix* $D(G) \in \mathbb{R}^{V \times V}$ of G is $D(G) := \operatorname{diag}(d_u)_{u \in V}$.
- The *Laplacian* $L(G) \in \mathbb{R}^{V \times V}$ of G is $L(G) := D(G) - A(G)$.

More generally, we call a square matrix $L \in \mathbb{R}^{n \times n}$ *Laplacian* if there is a graph G' with $L = L(G')$. This is equivalent to:

- L is symmetric
- The diagonal elements of L are nonnegative and the off-diagonal elements of L are nonpositive.
- Each row of L sums to 0.

In particular, a Laplacian matrix is always an SDD matrix. Another useful property of the Laplacian is the factorisation $L = B^T R^{-1} B$ where $B \in \mathbb{R}^{E \times V}$ is the *incidence matrix* and $R \in \mathbb{R}^{E \times E}$ is the *resistance matrix* (see Section 3.1 for why this name makes sense) defined by

$$B_{ab,c} = \begin{cases} 1 & a = c \\ -1 & b = c \\ 0 & \text{otherwise} \end{cases} \qquad R_{e_1,e_2} = \begin{cases} 1/w_{e_1} & \text{if } e_1 = e_2 \\ 0 & \text{otherwise} \end{cases}$$

for all $e_1, e_2 \in E$ and $a, b, c \in V$ where we arbitrarily fixed a start and end node for each edge when defining B.

Since R is diagonal, this factorisation is very useful and easy to work with. With

$$x^T L x = (Bx)^T R^{-1} (Bx) = \sum_{e \in E} \underbrace{(Bx)_e^2 \cdot w_e}_{\geqslant 0} \geqslant 0,$$

we can, for example, conclude that L is positive semidefinite. (A matrix $A \in \mathbb{R}^{n \times n}$ is *positive semidefinite* if $x^T A x \geqslant 0$ for all $x \in \mathbb{R}^n$.)

2.2 Cycles, spanning trees & stretch

A *cycle* in a graph is usually defined as a simple path that returns to its starting point and a graph is called *Eulerian* if there is a cycle that visits every edge exactly once.

In this thesis we will interpret cycles somewhat differently: We say that a cycle in G is a subgraph C of G such that every vertex in G is incident to an even number of edges in C, i. e. a cycle is a union of Eulerian graphs. It is useful to define the addition $C_1 \oplus C_2$ of two cycles C_1, C_2 to be the set of edges that occur in exactly one of the two cycles, i. e. $C_1 \oplus C_2 := (C_1 \setminus C_2) \cup (C_2 \setminus C_1)$.

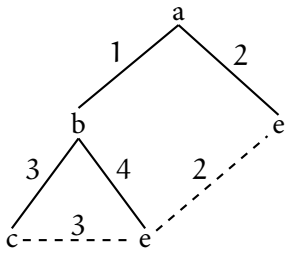


Figure 2.1: The solid edges form a spanning tree T, while the dashed edges are the off-tree-edges. We have $\mathrm{st}(ce) = (3+4)/3$ and $\mathrm{st}(T) = 4 + (3+4)/3 + (4+1+2)/2$.

In the language of linear algebra we can regard a cycle as a vector $C \subseteq \mathbb{F}_2^E$ such that $\sum_{v \in N_C(u)} 1 = 0$ in $\mathbb{F}_2$ for all $u \in V$ and the cycle addition as the usual addition on $\mathbb{F}_2^E$. We call the resulting linear space of cycles $\mathcal{C}(G)$.

A *tree* T is a connected graph without cycles. In T there is a unique path $P_T(u, v)$ from every node u to every node v. A *spanning tree (ST) of a graph* G is a subgraph $T = (V_T, E_T)$ of G with $V_T = V$ that is a tree. For any edge $e = uv \in E \setminus E_T$ (an *off-tree-edge with respect to* T) the subgraph $e \cup P_T(u, v)$ is a cycle, the *basis cycle induced by* e. One can easily show that the basis cycles form a basis of $\mathcal{C}(G)$. Thus, the basis cycles are very useful in algorithms that need to consider all of the cycles of a graph.

Another notion we need is a measure of how well a spanning tree approximates the original graph. We capture this by the *stretch* $\mathrm{st}(e) = \left(\sum_{e' \in P_T(u,v)} w_{e'}\right)/w_e$ *of an edge* $e = uv \in E$. This stretch is the detour you need in order to get from one endpoint of the edge to the other if you stay in T, compared to the length of the original edge. In the literature the stretch is sometimes defined with the length of the shortest path between u and v in the denominator instead of w_e, but we follow the definition in Kelner et. al.'s [Kel+13] paper using w_e.

The *stretch of the whole tree* T is the sum of the individual stretches $\mathrm{st}(T) = \sum_{e \in E} \mathrm{st}(e)$. See Figure 2.1 for an example. Finding a spanning tree with low stretch is crucial for proving the fast convergence of the Laplacian solver. The *condition number* $\tau(T) := \mathrm{st}(T) + |E| - 2 \cdot |V| + 2$ *of the tree* T, also plays a role in the convergence analyses.

2.3 Lagrangian duality

We now briefly look at the concept of duality in optimization theory. Take an arbitrary minimization problem P with $x \in \mathbb{R}^n$ and $o, f_i, g_j \colon \mathbb{R}^n \to \mathbb{R}$:

minimize	$o(x)$
subject to	$f_i(x) \leqslant b_i$ for $i = 1, \ldots, m$
	$g_j(x) = c_j$ for $j \in 1, \ldots, l$

We can systematically determine lower bounds on the optimal value $o^\star := o(x^\star)$ by pricing constraint violations heavily instead of disallowing vectors $x \in \mathbb{R}^n$ that violate any constraint. That is, we turn the hard constraints into soft constraints. The inequality constraints $f_i(x) \leqslant b_i$ get a price $\lambda_i \in \mathbb{R}_{\geqslant 0}$ and the equality constraints $g_j(x) = c_j$ get a price $\mu_j \in \mathbb{R}$.

Then can we find a lower bound on $o^\star$ by solving the unconstrained problem

$$l(\lambda, \mu) := \min \left\{ o(x) + \sum_{i=1}^{m} \lambda_i \cdot \big(b_i - f_i(x)\big) + \sum_{j=1}^{l} \mu_j \cdot \big(c_j - g_j(x)\big) \right\}$$

The property $l(\lambda, \mu) \leqslant o^\star$ is called *weak duality*. Thus, for fixed λ and μ we find a single lower bound on $o^\star$ (it can trivially be $-\infty$).

The *Lagrangian dual* $P^\star$ is then the problem of determining the best lower bound possible using this construction:

maximize $l(\lambda, \mu)$
subject to $\lambda \geqslant 0$

Let $l^\star$ be the optimal value of $P^\star$. If $l^\star = o^\star$, we say that *strong duality* holds. The Slater conditions [Sla50] are the most frequently used sufficient conditions for strong duality. One important special case of them is when P only has equality constraints.

2.4 SDD to Laplacian

The core problem of this thesis is to solve SDD systems. In this section we show that we can always reduce an SDD system to a Laplacian system using a construction by Gremban [Gre96]. We only consider Laplacian systems in the remainder of the thesis.

The construction increases the size of the matrix by a factor of 2, so it imposes a significant cost in practice. Still, since Laplacian systems also occur in many practical applications, focusing on them is not an unrealistic restriction.

Let $Ax = b$ be a linear system where $A \in \mathbb{R}^{n \times n}$ is SDD and $b \in \mathbb{R}^n$. Decompose the matrix A into its positive off-diagonal entries A_p, its negative off-diagonal entries A_n and its diagonal entries D, i. e. $A = D + A_p + A_n$. Then further decompose D into two nonnegative diagonal matrices $D = D_1 + D_2$ via

$$D_{ii} = \underbrace{D_{ii} - \sum_{j \neq i} |A_{ij}|}_{=:(D_2)_{ii} \geqslant 0 \text{ since } A \text{ is SDD}} + \underbrace{\sum_{j \neq i} |A_{ij}|}_{=:(D_1)_{ii}}$$

Define

$$\widetilde{A} = \begin{pmatrix} D_1 + D_2/2 + A_n & -D_2/2 - A_p \\ -D_2/2 - A_p & D_1 + D_2/2 + A_n \end{pmatrix} \in \mathbb{R}^{2n \times 2n}$$

and $\widetilde{b} = \begin{pmatrix} b \\ -b \end{pmatrix} \in \mathbb{R}^{2n}$.

Since the off-diagonal entries in $\widetilde{A}$ come from the sum of the nonpositive matrices A_n, $-A_p$ and $-D_2/2$, they must be nonnegative. We have $D_1 + D_2/2 + A_n - D_2/2 - A_p = D_1 - (-A_n + A_p) = 0$ by the definition of D_1, i. e. the rows of $\widetilde{A}$ sum to zero. The symmetry of the matrix $\widetilde{A}$ follows from the symmetry of the matrices D_1, D_2, A_n and A_p. Thus, $\widetilde{A}$ is a Laplacian matrix.

We now show that $\widetilde{A}\widetilde{x} = \widetilde{b}$ is equivalent to $Ax = b$ in the sense that for every solution $\widetilde{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$ the vector $(x_1 - x_2)/2$ is a solution to $Ax = b$.

$$\begin{aligned} A \cdot (x_1 - x_2)/2 &= \frac{1}{2}(D_1 + D_2 + A_p + A_n)(x_1 - x_2) \\ &= \frac{1}{2}\Big[(D_1 + D_2/2 + A_n)x_1 + (-D_2/2 - A_p)x_2 \\ &\qquad - (-D_2/2 - A_p)x_1 - (D_1 + D_2/2 + A_n)x_2\Big] \\ &= \frac{1}{2} \cdot (b + b) \\ &= b \end{aligned}$$

Thus, we reduced solving an SDD system to solving a Laplacian system.

Problem: INV-LAPLACIAN
Given: Laplacian matrix $L \in \mathbb{R}^{n \times n}$ and vector $b \in \mathrm{im}(L)$.
Problem: Find an $x \in \mathbb{R}^n$ with $Lx = b$.

Kelner et al. [Kel+13] quantified this reduction further since a more precise statement of the equivalence is necessary to prove the time bound of their solver.

Theorem 2.1 (Appendix A in [Kel+13]).
Let $\widetilde{y} := \widetilde{A}^{-1}\widetilde{b}$, $y := A^{-1}b$ *and* $\widetilde{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \in \mathbb{R}^{2n}$. *If* $\|\widetilde{x} - \widetilde{y}\|_{\widetilde{A}} \leqslant \epsilon \cdot \|\widetilde{y}\|_{\widetilde{A}}$ *for some* $\epsilon > 0$, *then* $\|x - y\|_A \leqslant \epsilon \cdot \|y\|_A$ *where* $x = (x_1 - x_2)/2$.

2.5 Conventions & notations

In the remainder of this thesis we will use some common conventions: $G = (V, E)$ is a weighted undirected graph with vertices V, edges E and edge weights $w_e > 0$ for

each $e \in E$. We use $n := |V|$ for its order and $m := |E|$ for its size. Every function that is parametrised by a single graph will implicitly use G, e. g. $A = A(G)$.

We also assume that G is connected. This is not a significant restriction since we can just apply the solver to every component. Of course, in our actual implementation we first decompose the graph into components.

Furthermore, whenever we talk about the residual of a vector y with respect to a linear system $Ax = b$ we refer to the *relative residual* $\|Ay - b\|_2 / \|b\|_2$.

3 Nearly-linear time solver

Using the prerequisites from Chapter 2, in this chapter we present the basic idea of the Laplacian solver by Kelner et al. [Kel+13].

We first show how to intuitively interpret INV-LAPLACIAN as the problem of iteratively finding an electrical flow on the graph corresponding to the Laplacian (Section 3.1). Then we show how to quantify the solutions in this iterative scheme by introducing energies (Section 3.2). Finally, we use the energies to show how the flow interpretation results in a solver whose convergence depends on the stretch of a spanning tree of the graph (Section 3.3).

3.1 Laplacians and electrical flows

In this section we first interpret the operation of the Laplacian L on vectors in terms of electrical engineering (Section 3.1.1) and then rephrase INV-LAPLACIAN using this interpretation (Section 3.1.2).

3.1.1 Operation of a Laplacian

L operates on every vector $x \in \mathbb{R}^n$ via

$$\begin{aligned}(Lx)_u &= -x_u \cdot \sum_{v \in N(u)} w_{uv} + \sum_{v \in N(u)} x_v \cdot w_{uv} \\ &= \sum_{v \in N(u)} (x_v - x_u) \cdot w_{uv}\end{aligned}$$

for each $u \in V$.

As illustrated in Figure 3.1, we can regard G as an electrical network where each edge uv corresponds to a resistor with conductance w_{uv} and x as an assignment of potentials to the nodes of G.

Then $x_v - x_u$ is the voltage across uv and $(x_v - x_u) \cdot w_{uv}$ is the resulting current along uv. Thus, $(Lx)_u$ is the current flowing out of u that we want to be equal to the right-hand side b_u. These interpretations are summarised in Table 3.1.

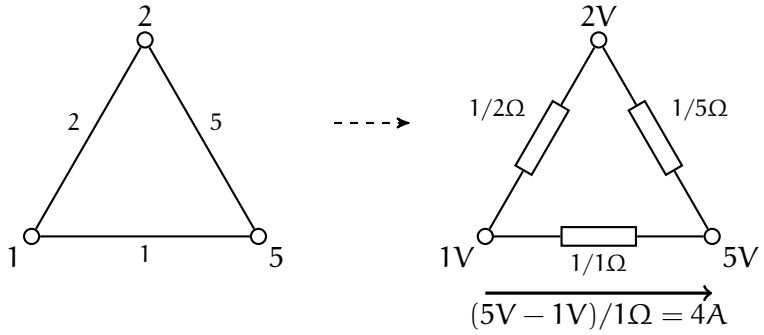


Figure 3.1: Transformation into an electrical network.

e	edge/resistor e
w_e	conductance of resistor e
$r_e := 1/w_e$	resistance of resistor e
x_u	potential at node u
$(Lx)_u$	current flowing out of node u
b_u	current required to flow out of node u

Table 3.1: Interpretations given to a Laplacian $L = L(G) \in \mathbb{R}^{n \times n}$ and a vector $x \in \mathbb{R}^n$ where the w_e for each $e \in E$ are the edge weights.

Thus, INV-LAPLACIAN can be considered as a problem INV-LAPLACIAN-CURRENT of assigning potentials that result in a given flow out of each node.

Problem: INV-LAPLACIAN-POTENTIAL
Given: Laplacian $L = L(G)$ and vector $b \in \mathrm{im}(L)$.
Problem: Assign potentials $x \in \mathbb{R}^n$ to the nodes in G such that the current flowing out of u is b_u for each $u \in V$.

3.1.2 Dualising INV-LAPLACIAN-POTENTIAL

If we look at the structure of the *induced currents* $f_{uv} := (x_v - x_u) \cdot w_{uv}$ instead of the potentials x_u, we get a graph flow. A *(valid) graph flow* on G with demand vector $x \in \mathbb{R}^V$ is a function $f\colon \widetilde{E} \to \mathbb{R}$ on a directed copy $\widetilde{E} := \{(u,v) : uv \in E\}$ of the edges E with the following two properties:

1. $f(u,v) = -f(v,u)$ for all $uv \in E$
2. $\sum_{v \in N(u)} f(u,v) = b_u$ for all $u \in V$

Computing a graph flow f is not equivalent to INV-LAPLACIAN-POTENTIAL since not every flow is induced by potentials. We call the flows induced by potentials *electrical*

flows. Searching for valid graph flows that are also electrical flows gives us the problem INV-LAPLACIAN-CURRENT equivalent to INV-LAPLACIAN-POTENTIAL.

Problem: INV-LAPLACIAN-CURRENT
Given: Laplacian $L = L(G)$ and vector $b \in \mathrm{im}(L)$.
Problem: Compute a function $f\colon \widetilde{E} \to \mathbb{R}$ with:

(1) f is a graph flow on G with demand b

(2) f is induced by a potential vector $x \in \mathbb{R}^V$, i.e. $f(u,v) = \big(x(v) - x(u)\big) \cdot w_{uv}$ for all $(u,v) \in \widetilde{E}$.

It is not at all clear how you can guarantee or even check property (2). Fortunately, a classical result from electrical engineering comes to the rescue. Kirchhoff's voltage law [Kir45] states that (2) is equivalent to

(2') The potential drop along every cycle in G is zero.

Input: Laplacian $L = L(G)$ and vector $b \in \mathrm{im}(L)$.
Output: Solution x to $Lx = b$.
1 $f \leftarrow$ any graph flow on G with demand b
2 **while** *there is a cycle* c *with potential drop* $\neq 0$ *in* f **do**
3 Add multiple of c to f such that the potential drop along c becomes 0
4 **return** *vector of potentials in* f *with respect to an arbitrary node in* G

Algorithm 1: Basic approach of the INV-LAPLACIAN-CURRENT solver.

Unlike (2), the property (2') can be used pragmatically to compute an electrical flow. The idea is to start with any valid flow and then successively adjust it so that every cycle has potential zero. The basic idea is given in Algorithm 1. Note that we need to transform the flow back to potentials at the end. This can be done consistently since all potential drops along cycles are zero.

In the following sections we elaborate on how to actually choose the cycles in order to get fast convergence with this approach.

3.2 Energies

We reinterpreted INV-LAPLACIAN as the problem INV-LAPLACIAN-CURRENT of iteratively finding an electrical flow. In every iterative method we need a notion of how good the current solution is. We use the *energy* $\xi_r(f) := \sum_{e \in E} r_e f(e)^2$ *of a flow* $f \in \mathbb{R}^E$ from electrical engineering to derive such a measure of goodness. (Note that this is actually the electric power $P = \text{voltage} \cdot \text{current} = \text{resistance} \cdot \text{current}^2$.)

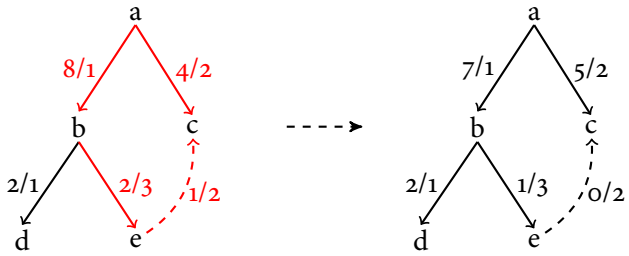


Figure 3.2: Repairing a single cycle in a tree flow. The edges of the tree are labelled with f_e/r_e. The initial potential drop on the cycle $C := ecabe$ is $1 \cdot 2 - 4 \cdot 2 + 8 \cdot 1 + 2 \cdot 3 = 8$. If we add $-8/(2+3+1+2) = -1$ to each edge in C in the same direction as in C, we get a potential drop of $0 \cdot 2 - 5 \cdot 2 + 7 \cdot 1 + 1 \cdot 3 = 0$ on C.

For the following computations we need that this energy can be written as $\xi_r(f) = f^T R f$, that $(B^T f)_u = \sum_{v \in N(u)} f(uv)$ for all $u \in V$ and that the potential drop in f along a cycle c in G is

$$\Delta_c(f) = \sum_{uv \in c} f(uv)/w_{uv} = f^T R c.$$

Here we interpret c both as an ordered sequence of edges and a vector as well as f as both a flow and a vector.

The definition of $\xi_r(f)$ is consistent with the goal of avoiding cycles with nonzero potential drop. Assume there is a cycle c with $\Delta_c(f) \neq 0$ and consider modifying the flow by subtracting a multiple of c, i. e. $f' = f - \lambda \cdot c$ for a $\lambda \in \mathbb{R}$.

The choice $\lambda_{opt} := c^T R f / c^T R c$ both guarantees that the potential drop

$$\begin{aligned} \Delta_c(f') &= (f - \lambda \cdot c)^T R c \\ &= f^T R c - \lambda \cdot c^T R c \end{aligned}$$

along c is zero and that the energy

$$\begin{aligned} \xi_r(f') &= (f - \lambda \cdot c)^T R (f - \lambda \cdot c) \\ &= f^T R f - 2\lambda \cdot c^T R f + \lambda^2 \cdot c^T R c \end{aligned}$$

of f' is minimized. Figure 3.2 gives an example for repairing a single cycle.

Thus, minimizing the energy ξ_r coincides with avoiding nonzero potential drops and we can rephrase INV-LAPLACIAN-CURRENT as the optimization problem

minimize $\xi_r(f) = f^T R f$
subject to $B^T f = b$

By applying Lagrangian duality we get the dual problem

maximize $\zeta_r(x) := \min_{f \in \mathbb{R}^E} m_x(f)$ where $m_x(f) := \left(f^T R f + x^T (b - B^T f)\right)$
subject to $x \in \mathbb{R}^V$

Since $\nabla_f m_x(f) = 2Rf - Bx$, the optimum flow is $f_{\min} = 1/2 \cdot R^{-1} Bx$ and by using $B^T R^{-1} B = L$ we get

$$\begin{aligned}\zeta_r(x) &= \frac{1}{4} \cdot x^T B^T R^{-1} B x - \frac{1}{2} \cdot x^T B^T R^{-1} B x + b^T x \\ &= -\left(\frac{1}{2} \cdot x^T L x - b^T x\right).\end{aligned}$$

We call the value $\zeta_r(x)$ the *dual energy of* x.

By negating the objective function and turning the problem into a minimization problem, we get the equivalent formulation

minimize $E(x) := \frac{1}{2} x^T L x - b^T x$
subject to $x \in \mathbb{R}^V$

We have $\nabla E(x) = Lx - b$ and $H(E) = L \geqslant 0$, i.e. E is convex and its minima are at $\nabla E(x) = 0 \Leftrightarrow Lx = b$. Thus, solving the linear system $Lx = b$ is equivalent to minimising $E(x)$. This standard observation is at the core of many iterative methods for linear systems, most prominently gradient descent and conjugate gradients.

With weak duality we have $\xi_r(f) - \xi_r(f_{\text{opt}}) \leqslant \xi_r(f) - \zeta_r(x) =: \text{gap}(f, x)$ where $x \in \mathbb{R}^V$ is some assignment of potentials. Thus, $\text{gap}(f, x)$ can serve as the desired measure of goodness of f.

In fact, in our case even strong duality holds as we only have equality constraints. Thus, minimising $\xi_r(f)$ (avoiding cycles with nonzero potential) yields the same value as maximising $\zeta_r(x)$ (solving $Lx = b$).

3.3 Cycle selection and convergence

The basic approach presented in Algorithm 1 leaves open the crucial question of what flow to start with and how to choose the cycle to be repaired in each iteration. Kelner et al. [Kel+13] suggest using the cycle basis induced by a spanning tree T of G and prove that the convergence of the resulting solver depends on the stretch of T. More

Input: Laplacian $L = L(G)$ and vector $b \in \mathrm{im}(L)$.
Output: Solution x to $Lx = b$.

$T \leftarrow$ a spanning tree of G
$f \leftarrow$ unique flow with demand b that is only nonzero on T
while *there is a cycle with potential drop* $\neq 0$ *in* f **do**
  $c \leftarrow$ a cycle in $\mathcal{C}(T)$ where cycles are chosen randomly weighted by their stretch
  $f \leftarrow f - \frac{c^T R f}{c^T R c} c$
return *vector of potentials in* f *with respect to the root of* T

Algorithm 2: Refined INV-LAPLACIAN-CURRENT solver.

specifically, they suggest starting with a flow that is nonzero only on T and weighting the basic cycles by their stretch when sampling them.

The resulting refinement of Algorithm 1 is given in Algorithm 2. Note that we may stop before all potential drops are zero and we can consistently compute the *potentials induced by* f at the end by only looking at T.

We can prove that the energy of the starting flow f_0 is at most a factor of $\mathrm{st}(T)$ larger than the energy of an optimal flow f_{opt}.

Lemma 3.1 (Lemma 6.1 from [Kel+13]).
We have $\mathcal{E}_r(f_0) \leqslant \mathrm{st}(T) \cdot \mathcal{E}_r(f_{opt})$.

If we weight each cycle by its stretch, we can prove that each iteration decreases $\mathcal{E}_r(f)$ by a factor of $1 - 1/\tau(T)$ on average.

Lemma 3.2 (Lemma 4.5 from [Kel+13]).
Every iteration i *computes a feasible* $f_i \in \mathbb{R}^E$ *such that*

$$\mathbb{E}\left[\mathcal{E}_r(f_i)\right] - \mathcal{E}_r(f_{opt}) \leqslant \left(1 - 1/\tau(T)\right) \cdot \left(\mathcal{E}_r(f_{i-1}) - \mathcal{E}_r(f_{opt})\right).$$

To prove convergence of the potentials (the vector we are actually interested in) we also need a statement on how the energy of the flow f corresponds to the distance of the potentials x to $L^{-1}b$.

Lemma 3.3 (Lemma 6.2 from [Kel+13]). *Let* $f \in \mathbb{R}^E$ *be a feasible flow with demand* b *and* $\mathcal{E}_r(f) \leqslant (1+\alpha)\mathcal{E}_r(f_{opt})$ *for an* $\alpha > 0$. *Then we have*

$$\|x - L^{-1}b\|_L \leqslant \sqrt{\alpha\tau(T)}\|L^{-1}b\|_L$$

for the potentials $x \in \mathbb{R}^V$ *induced by* f *on* T.

By choosing $\alpha = \epsilon^2/\tau(T)$ in Lemma 3.3, we see that $\xi_r(f_i) \leqslant \big(1 + \epsilon^2/\tau(T)\big)\xi_r(f_{opt})$ is sufficient to ensure $\|x - L^{-1}b\|_L/\|L^{-1}b\|_L \leqslant \epsilon$ for an $\epsilon > 0$. Lemmas 3.1 and 3.2 together then give the central convergence result of the Laplacian solver.

Theorem 3.4 (Convergence of Algorithm 2, Theorem 3.2 in [Kel+13]).
Let $\epsilon > 0$ *and* $x_{opt} := L^{-1}b$. *Then we have*

$$\mathbb{E}\big[\|x - x_{opt}\|_L/\|x_{opt}\|_L\big] \leqslant \epsilon$$

for the potentials $x \in \mathbb{R}^V$ *induced by* f *on* T *after* $\tau\big(T\big)\log\big(st(T)\tau(T)/\epsilon\big)$ *iterations.*

4 Implementation

While Algorithm 2 provides the basic idea of Kelner et al.'s [Kel+13] Laplacian solver, it leaves open several implementation decisions that we elaborate on in this chapter.

The solver crucially depends on the spanning tree T for forming the cycle basis. We discuss possible spanning tree algorithms (Section 4.1). Since Papp [Pap14] showed that, in practice, the algorithms with provably good stretch do not yield better stretch than simpler approaches, we particularly look at simple spanning tree algorithms.

We continue by looking at how to store and repair the current flows (Section 4.3). You could trivially store the flow directly on T. Unfortunately, repairing a basis cycle with this scheme takes $\mathcal{O}(n)$ worst-case time, which does not suffice for the desired nearly-linear running time. We, therefore, also look at an improved data structure described by Kelner et al. [Kel+13] that only needs $\mathcal{O}(\log n)$ time for repairing a basis cycle.

While the provably good algorithm given in Algorithm 2 requires weighting the randomly chosen cycles by their stretch, it could also be worthwhile taking a look at what happens when we choose a basis cycle uniformly at random. We describe both implementation choices in Section 4.3.

The only parts of Algorithm 2 that remain open are how to find the initial flow and how to get the dual potential at the end. Both can be implemented optimally with a recursive traversal of T by directly using the definition of a valid graph flow $\sum_{v \in N(u)} f(uv) = b_u$ and the potential drop $x_u := \sum_{e \in P_T(u,r)} r_e f(e)$, respectively.

We conclude with a summary of the choices and their running times in Section 4.4. If we use a spanning tree with low stretch and weight the cycles by their stretch, we can then infer a running time bound for the whole solver with Theorem 3.4.

4.1 Spanning trees

As suggested by the convergence result in Theorem 3.4, the Laplacian solver depends on low-stretch spanning trees. The notion of stretch was first introduced by Alon et al. [Alo+95] along with an algorithm to compute a spanning tree with low stretch. Unfortunately, the guaranteed stretch with their algorithm is super-polynomial.

	Time	Stretch
[Alo+95]	$\mathcal{O}(m^2)$	$m \cdot \exp\left(\mathcal{O}(\sqrt{\log n \log\log n})\right)$
[Elk+05]	$\mathcal{O}(m \log^2 n)$	$m \cdot \mathcal{O}(\log^2 n \log\log n)$
[ABN08]	$\mathcal{O}(m \log^2 n)$	$m \cdot \mathcal{O}(\log n (\log\log n)^3)$
[KMP11]	$\mathcal{O}(m \log n \log\log n)$	$m \cdot \mathcal{O}(\log n (\log\log n)^3)$
[AN12]	$\mathcal{O}(m \log n \log\log n)$	$m \cdot \mathcal{O}(\log n \log\log n)$
Dijkstra [Dij59]	$\mathcal{O}((m+n) \log n)$	No guarantee
Kruskal [Kru56]	$\mathcal{O}(m\alpha(n) \log n)$	No guarantee

Table 4.1: Spanning trees and their guaranteed stretch

Elkin et al. [Elk+05] presented an improved algorithm requiring nearly-linear time and yielding nearly-linear average stretch. The basic idea is to recursively form a spanning tree using a star of balls in each recursion step, but the specifics are not particularly important for us. We just note that we use Dijkstra with binary heaps for growing the balls and that we take care not to need more work than necessary to grow the ball. In particular, ball growing is output-sensitive and growing a ball $B(x, r) := \{v \in V : \text{Distance from } x \text{ to } v \text{ is } \leqslant r\}$ should require $\mathcal{O}(d \log n)$ time where d is the sum of the degrees of the nodes in $B(x, r)$.

The exponents of the logarithmic factors of the stretch of this algorithm were improved by subsequent papers (see Table 4.1), but Papp [Pap14] showed experimentally that these improvements do not yield better stretch in practice. In fact, his experiments suggest that the stretch of the provable algorithms is usually not better than just taking a minimum-weight spanning tree.

Therefore, we additionally use two simpler spanning trees without stretch guarantees: A minimum-distance spanning tree with Dijkstra's algorithm and binary heaps; as well as a minimum-weight spanning with Kruskal's algorithm using union-find with union-by-size and path compression.

4.2 Flows on trees

We now show how to store and update the flow (the currents) in the graph. The goal is to be able to efficiently get the potential drop of every basis cycle and to be able to add a constant amount of flow to it.

Since every basis cycle contains exactly one off-tree-edge, the flows on off-tree-edges can simply be stored in a single vector. The core problem is then to efficiently store and

update flows in T. More formally, we want to support the following two operations for all $u, v \in V$ and $\alpha \in \mathbb{R}$ on the flow f:

- query(u, v): return the potential drop $\sum_{e \in P_T(u,v)} f(e) r_e$
- update(u, v, α): set $f(e) := f(e) + \alpha$ for all $e \in P_T(u, v)$

} (1)

We can simplify the operations by fixing v to be the root r of T:

- query(u): return the potential drop $\sum_{e \in P_T(u,r)} f(e) r_e$
- update(u, α): set $f(e) := f(e) + \alpha$ for all $e \in P_T(u, r)$

} (2)

The two-node operations (1) can then be supported with

$$\text{query}(u, v) := \text{query}(u) - \text{query}(v)$$

and

$$\text{update}(u, v, \alpha) := \{\text{update}(u, \alpha) \text{ and } \text{update}(v, -\alpha)\}$$

since the changes on the subpath $P_T\big(r, \text{LCA}(u, v)\big)$ cancel out. Here $\text{LCA}(u, v)$ is the *lowest common ancestor* of the nodes u and v in T, the node farthest from r that is an ancestor of both u and v.

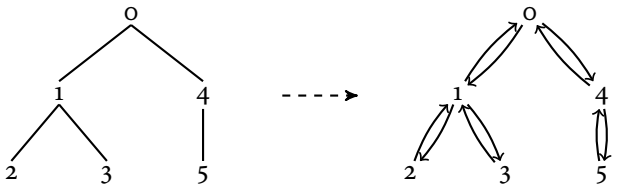
We provide two approaches for implementing the operations. Firstly, in Section 4.2.1 we present a trivial implementation of (2) that stores the flow directly on the tree and uses the definitions of the operations without modification. Obviously, these operations require $\mathcal{O}(n)$ worst-case time and $\mathcal{O}(n)$ space. If we have an LCA data structure, we can implement the operations in (1) without the simplification (2). This does not improve the worst-case time, but helps in practice. Also, many graphs in the real world have low diameter and, correspondingly, the depth of T may be low. Thus, the LCA approach could work very well. We check this in Section 5.2.4.

Secondly, we briefly describe the improved data structure by Kelner et al. [Kel+13] that guarantees $\mathcal{O}(\log n)$ worst-case time but uses $\mathcal{O}(n \log n)$ space. In this case the operations (2) boil down to a dot product (query) and an addition (update) of a dense vector and a sparse vector.

4.2.1 Linear time updates

Trivial approach

The trivial implementation of (2) directly stores the flows in the tree and implements each operation in (2) with a single traversal from the node u to the root r.



T

	0	1	2	3	4	5	6	7	8	9	10
`tour`	0	1	2	1	3	1	0	4	5	4	0
`depths`	0	1	2	1	2	1	0	1	2	1	0
`idx`	0	1	2	4	7	8					

Figure 4.1: LCA to RMQ: `tour` is the Eulerian tour of T, `depths` stores the depths of the nodes in `tour`, `idx` maps each node in T to its first occurrence in `tour`.

.

Improvement with LCA

We can improve this implementation by only traversing up to the the lowest common ancestor of u and v in (1). Of course, this does not help with the worst-case time $\mathcal{O}(n)$, but could be quite significant in practice since basis cycles are often short.

Data structures that answer LCA queries for pairs of nodes after some precomputation are a classic topic and optimal ($\mathcal{O}(n)$ time precomputation, $\mathcal{O}(1)$ time queries) solutions are known [HT84; BFoo].

In our implementation we used a simpler implementation with $\mathcal{O}\big(n \log(n)\big)$ time for the precomputation and queries in $\mathcal{O}(1)$ time:

1. First, we transform an LCA query into an RMQ query, the problem of determining the minimum in a subrange of an array.

   Problem: RMQ
   Given: Array of numbers $v[1 \dots n]$ and two indexes $l, r \in [n]$ with $l \leqslant r$.
   Problem: Compute $\arg\min_{l \leqslant i \leqslant r} v[i]$.

   To do so we store an Eulerian tour of T, where we imagine every edge of T to be replaced by a forward edge and a backward edge, in an array `tour`. We also store the depths of the nodes visited along the tour in `depths` and for every node in T the index in `tour` that it first appears at in `idx`. Figure 4.1 illustrates these data structures.

   Let u and v be two nodes in T and without loss $\texttt{idx}[u] \leqslant \texttt{idx}[v]$. Then $P := \texttt{tour}\big[\texttt{idx}[u] \dots \texttt{idx}[v]\big]$ is a subpath of the Eulerian tour from u to v and the

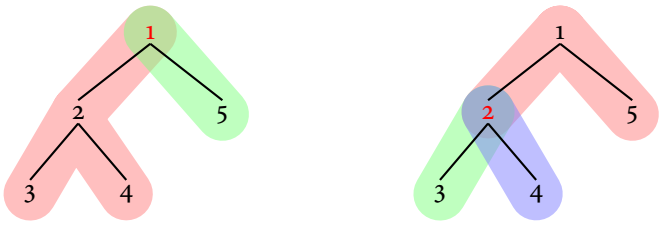


Figure 4.2: The subtrees induced by the nodes 1 and 2. Node 2 is a good vertex separator.

highest node visited in P is exactly $\mathrm{LCA}(u, v)$. Thus, we can transform LCA to RMQ via

$$\mathrm{LCA}(u, v) = \texttt{tour}\left[\operatorname*{arg\,min}_{\texttt{idx}[u] \leqslant i \leqslant \texttt{idx}[v]} \texttt{depths}[i]\right].$$

2. We now solve the RMQ problem by precomputing the RMQ of every range that has a length that is exactly a power of two, i. e. for each i with $2^i \leqslant n$ and every $j \in [n]$ we compute

$$\texttt{precomp}[i, j] := \arg\min v[j \mathinner{..} j + 2^i - 1].$$

This can be done in $\mathcal{O}(n \log(n))$ time with the recurrence

$$\texttt{precomp}[i, j] = \arg\min\left(v[\texttt{precomp}[i-1, j]], v[\texttt{precomp}[i-1, j + 2^i]]\right)$$

for $i > 0$. (We disregard the boundary of the array.)

The crucial observation for answering RMQs on ranges of any length is then that any range can be decomposed into two (possibly overlapping) ranges that have a power of two as length. In particular, for $i, j \in [n]$ with $i \leqslant j$ we have

$$\mathrm{RMQ}(i, j) = \arg\min\left(v[\texttt{precomp}[k, i]], v[\texttt{precomp}[k, j - 2^k + 1]]\right),$$

where $k \in \mathbb{N}$ is the largest number such that $2^k \leqslant j - i + 1 = \left|\{i, \ldots, j\}\right|$.

4.2.2 Logarithmic time updates

While the data structure presented in the last section allows fast repairs for short basis cycles, the worst-case time is still in $\mathcal{O}(n)$.

In this section we briefly describe the data structure by Kelner et al. [Kel+13] with $\mathcal{O}(\log n)$ worst-case time repairs. It is based on the link-cut trees introduced by Sleator and Tarjan [ST83].

The first observation it uses is that every rooted tree T on n nodes can be decomposed into edge-disjoint subtrees intersecting in exactly one node such that each subtree has $\leqslant n/2$ nodes. Equivalently, we find a vertex in T all of whose induced subtrees have size $\leqslant n/2$, as illustrated in Figure 4.2. We call such a vertex a *good vertex separator*.

To see this, start at the root r of T and consider the subtrees $T_1, \dots, T_k$ induced by r (ordered by size: $|T_1| \leqslant \cdots \leqslant |T_k|$). If all subtrees have size $\leqslant \lceil n/2 \rceil$, then we are done. Otherwise, recursively look at the root u of T_k. Since $|T_k| > \lceil n/2 \rceil$, every subtree induced by u must have a size strictly smaller than $|T_k|$. Thus, by continuing this recursion we must eventually get a good vertex separator, i. e. a node whose induced subtrees have size $\leqslant \lceil n/2 \rceil$.

By recursively finding good vertex separators on the subtrees, we get a recursive decomposition of the whole tree into subtrees. Since the size of the trees halves in each step, the depth of this decomposition is at most $\mathcal{O}(\log n)$.

Now consider a tree T at one level of recursion with root r that is split into the subtrees $T_0, \dots, T_k$ at the good vertex separator d. Let T_0 contain r without loss.

We can implement `query` and `update` efficiently by storing several values:

- d_{drop} the total potential drop on the path $P_T(r, d)$
- d_{ext} the total flow contribution to $P_T(r, d)$ from vertices below d
- $\text{height}(u) := \sum_{e \in P_T(r,a) \cap R_T(r,d)} r_e$ for every $u \in V(T)$, i. e. the accumulated resistance in common between the $P_T(r, d)$ path and the $P_T(r, a)$ path.

Then we can compute $\text{query}(u)$ as follows:

- If $u \in T_0$, the potential drop consists of the potential drop $\text{query}_{T_0}(u)$ in T_0 and the part $d_{\text{ext}} \cdot \text{height}(u)$ of the potential drop caused by vertices beyond d.
- If $u \in T_i$ and $u \neq d$, then we have the complete potential drop d_{drop} along $P_T(d, r)$ and a recursive potential drop $\text{query}_{T_i}(u)$.

The $\text{update}(u, \alpha)$ operation can be implemented similarly:

- If $u \notin T_0$, we need to adjust d_{ext} by α.
- In all cases we need update d_{drop} by the $\text{height}(u)$ part of the $P_T(r, u)$ path in common with T_0. Unless $u = d$, we then need to recursively update the tree T_i with $u \in T_i$.

```
if u = r then
    return d_drop
else if |V| = 2 then
    return 0
else if a ∈ T_0 then
    return d_ext · height(u) + query_{T_0}(u)
else
    return d_drop + query_{T_i}(u) where u ∈ T_i is unique
```

Algorithm 3: Query in LogFlow: $\text{query}_T(u)$

```
d_drop := d_drop + α · height(u)
if |V| > 2 then
    if u ∉ T_0 then
        d_ext := d_ext + α
    if u ≠ d then
        update_{T_i}(u, α) where u ∈ T_i is unique
```

Algorithm 4: Update in LogFlow: $\text{update}_T(u, \alpha)$

While we could directly implement this recursion as in Algorithms 3 and 4, we unrolled the recursion to get a more efficient implementation. We can store the complete state of the data structure in a dense vector x containing the d_{drop} and d_{ext} values for all recursion levels. For each $u \in T$, query is then a dot product $q(u) \cdot x$ with a vector $q(u)$ containing the appropriate coefficients and $\text{update}(u, \alpha)$ is a vector addition $x := x + \alpha l(u)$ with a vector $l(u)$.

The vectors $q(u)$ and $l(u)$ are sparse with at most $\mathcal{O}(\log n)$ nonzero entries and can be determined directly from the recursive decomposition in $\mathcal{O}(n \log(n))$ time (their entries are either $\text{height}(u)$ or 1). Kelner et al. [Kel+13] provide more details about the unrolling.

4.3 Cycle selection

In Section 4.2 we saw how to repair a cycle, but how do we actually select one? As discussed in Section 3.3, we work on the cycle basis given by a spanning tree, i. e. each of the cycles we want to select is represented by a unique off-tree edge.

Thus, the easiest way to select a cycle is to choose an off-tree edge *uniformly at random* in $\mathcal{O}(1)$ time. However, to get provably good results, we need to weight the off-tree-edges by their stretch.

We can use the data structure from Section 4.2.2 to get the stretches. More specifically, the data structure initially represents $f = 0$. For every off-tree edge uv we first execute update$(u, v, 1)$, then query(u, v) to get $\sum_{e \in P_T(u,v)} r_e$ and finally update$(u, v, -1)$ to return to $f = 0$. This results in $\mathcal{O}(m \log n)$ time to initialise cycle selection.

Once we have the weights, we use *roulette wheel selection* in order to select a cycle in $\mathcal{O}(\log m)$ time after an additional $\mathcal{O}(m)$ time initialisation. Roulette wheel selection is a simple strategy to sample an arbitrary discrete distribution with finite support:

- Let X be a random variable with $\text{Prob}[X = x_i] = p_i$ for $i \in [k]$.
- Precompute the prefix sums $P = (0, p_1, p_1 + p_2, \ldots, p_1 + \cdots + p_k = 1)$.
- To sample, choose a uniform random value $x \in [0, 1)$. Then find the index i with $P_i \leqslant x < P_{i+1}$ using binary search and output x_i. The probability for getting x_i with this scheme is
$$\left| \left[\sum_{j=0}^{i-1} p_i, \sum_{j=0}^{i} p_i \right) \right| = p_i,$$
as desired.

Lipowski and Lipowska [LL12] presented a faster method for discrete sampling that takes $\mathcal{O}(1)$ expected time for a somewhat restricted class of distributions, but requires more randomness. This method did not show significant improvements over binary search in informal tests of our own, so we did not pursue it further.

4.4 Summary

We summarise the possible implementation choices for Algorithm 2 in Table 4.2.

Explanation: The top-level item in each section is the running time of the best sub-item that can be used to get a provably good running time using Theorem 3.4. The convergence theorem requires a low-stretch spanning tree and weighted cycle selection. Note that $m = \Omega(n)$ since G is connected.

Spanning tree	$\mathcal{O}\big(m \log n \log\log n)\big)$ stretch, $\mathcal{O}(m \log n \log\log n)$ time
Dijkstra	no stretch bound, $\mathcal{O}(m \log n)$ time
Kruskal	no stretch bound, $\mathcal{O}(m \log n)$ time
Elkin et. al. [Elk+05]	$\mathcal{O}(m \log^2 n \log\log n)$ stretch, $\mathcal{O}(m \log^2 n)$ time
Abraham et. al. [AN12]	$\mathcal{O}(m \log n \log\log n)$ stretch, $\mathcal{O}(m \log n \log\log n)$ time
Initialise cycle selection	$\mathcal{O}(m \log n)$ time
Uniform	$\mathcal{O}(m)$ time
Weighted	$\mathcal{O}(m \log n)$ time
Initialise flow	$\mathcal{O}(n \log n)$ time
LCA flow	$\mathcal{O}(n)$ time
Log flow	$\mathcal{O}(n \log n)$ time
Iterations	$\mathcal{O}\big(m \log n \log\log n \log(\epsilon^{-1} \log n)\big)$ expected iterations
Select a cycle	$\mathcal{O}(\log n)$ time
Uniform	$\mathcal{O}(1)$ time
Weighted	$\mathcal{O}(\log n)$ time
Repair cycle	$\mathcal{O}(\log n)$ time
LCA flow	$\mathcal{O}(n)$ time
Log flow	$\mathcal{O}(\log n)$ time
Complete solver	$\mathcal{O}(m \log^2 n \log\log n \log\big(\epsilon^{-1} \log n)\big)$ expected time
Improved solver, see 5.2.1	$\mathcal{O}(m \log^2 n \log\log n \log\big(\epsilon^{-1})\big)$ expected time

Table 4.2: Summary of the components of the algorithm

5 Evaluation

Having seen the basic nearly-linear time Laplacian solver (Algorithm 2) and what implementation decision we can make (Chapter 4), we now come to the core of this thesis: An experimental evaluation of the Laplacian solver by Kelner et al. [Kel+13].

We start by describing some low-level implementation issues and how we benchmarked our implementation (Section 5.1).

Then we evaluate and benchmark the different parts of the algorithm in isolation and discuss the sensible choices for the components of the solver (Section 5.2).

Next we put the components together and benchmark the solver as a whole by looking at the convergence behaviour for single graphs (Section 5.3) and the asymptotic running time for graphs of increasing size (Section 5.4). We will see that the Laplacian solver has slow convergence and disappointing performance when compared to existing linear solvers. Even though we confirm that the running time grows nearly-linearly, the constant is too high to be competitive.

We try to save the solver by looking at how the solver behaves in conjunction with another solver as a preconditioner (Section 5.5) or a smoother (Section 5.6).

We conclude this experimental study by looking at other practical considerations: Various small problems that make the solver hard to use (Section 5.7), an evaluation of the micro-performance of the solver (Section 5.8) and a discussion about whether the solver can be parallelised (Section 5.9)

5.1 Benchmarking environment

We implemented the Laplacian solver in NetworKit [SSM14], a toolkit focused on implementing network analysis algorithms with a high degree of parallelism and scalability, and benchmarked it with the hardware and software in Table 5.1.

In some of the following sections we will compare this implementation to existing solvers as implemented by Eigen 3.2.2 [G+10] and Paralution 0.7.0 [Luk14]. Both libraries provide high-performance implementations of various common sparse matrix solvers.

But before continuing with the actual benchmarks, in this section we briefly describe which graphs we tested (Section 5.1.1), how we measured (Section 5.1.2) and how we set up the experiments to reduce errors (Section 5.1.3).

CPU	Intel® Xeon® CPU E5-2680, 2.70 GHz, 6/45/7 Family/Model/Stepping
Threads	2 sockets with 8 cores each, 2 hardware threads per core, 32 hardware threads in total
Cache	32/32 kB data/instruction L1 cache per core, 256 kB unified L2 cache per core, 20 MB unified L3 cache per socket
Memory	2 NUMA nodes with 128 GB memory each
Compiler	`g++ 4.8.3`
Flags	`-Wall -fPIC -std=c++11 -DNDEBUG` `-O3 -flto -ffast-math -fopenmp -mavx`
OS	`Linux 3.11.10 x86_64`

Table 5.1: Benchmarking hardware & software

5.1.1 Graphs

We use two classes of graphs to test the Laplacian solver:

- Rectangular $k \times l$ grids given by $\mathbb{G}_{k,l} := \left([k] \times [l], \left\{\{(x_1, y_1), (x_2, y_2)\} \subseteq \binom{V}{2} : |x_1 - x_2| = 1 \vee |y_1 - y_2| = 1\right\}\right)$. Laplacian systems on grids are, for example, crucial for solving boundary value problems on rectangular domains. Note that $\mathbb{G}_{k,l}$ is very uniform, i. e. most of its nodes have degree 4.

- Barabási–Albert [BA99] random graphs with parameter k. These random graphs are parametrised with a so-called *attachment* k. They are constructed by starting with K_k and iteratively adding $(n - k)$ nodes. We connect a new node to k random existing nodes where each existing node is weighted by its current degree, i. e. nodes are preferentially attached to nodes that already have a high degree. We denote the distribution of Barabási-Albert random graphs with n nodes and attachment k by $\mathrm{Barabasi}(n, k)$.

  This construction models that the degree distribution in many natural graphs is not uniform at all since some nodes are much better connected than others. More specifically, the fraction of nodes with degree l is usually proportional to $l^{-\gamma}$ for

Graph	n	m	*CG, no precond.*	*Laplacian solver*
`airfoil1`[a]	4253	12289	30 ± 0 ms	2039 ± 4 ms
`PGPgiantcompo`[b]	10680	24316	47 ± 0 ms	1152 ± 1 ms
`luxembourg.osm`[c]	114599	119666	22835 ± 3 ms	38624 ± 35 ms
`citationCiteseer`[d]	268495	1156647	54382 ± 361 ms	292574 ± 6350 ms

[a] http://www.cise.ufl.edu/research/sparse/matrices/AG-Monien/airfoil1.html
[b] http://www.cise.ufl.edu/research/sparse/matrices/Arenas/PGPgiantcompo
[c] http://www.cc.gatech.edu/dimacs10/archive/streets.shtml
[d] http://networkrepository.com/citationCiteseer.php

Table 5.2: Running times for reaching residual 10^{-4}. The values after $\pm$ give the standard deviation of the times. We use the Eigen CG implementation and the settings of the Laplacian solver resulting in the best performance when taking LogFlow.

some $\gamma > 0$ (typically $2 < \gamma < 3$). Graphs with a degree distribution following such a power law are called *scale-free*. For example, street graphs and Facebook friendship graphs are almost always scale-free.

For both classes of graphs we consider both unweighted variants (weights are 1) and weighted variants (uniform random weights in $[1, 8)$).

We also did informal tests on 3D grids and graphs that were not generated synthetically. Since these graphs did not exhibit significantly different behaviour than the two graph classes described above, we do not describe them in detail. In particular, they also did not prove to be competitive to CG, as shown in Table 5.2.

5.1.2 Measurements

Performance counters

We measured CPU performance characteristics such as the number of retired instructions, the number of executed FLOPS (floating point operations), etc. using the PAPI library [Bro+00].

While CPU counters can give nondeterministic results for low-level reasons [WTM13], experiments [ZJH09] show that the variance of the counter values is very low (far below 0.1%) if the measurement is long compared to the overhead of setting up and retrieving performance counters (several thousand cycles). Our benchmarking runs each take several seconds (billions of cycles), so we expect the counter values to be quite accurate.

Still, there are several caveats to take into account:

- Recent Linux kernels save and restore the counter registers on context switch, so the values should be accurate independent of the scheduled threads. Since thread

switches lead to other problems such as possible cache invalidation, we avoid them by pinning threads and making sure there is no other work on the machine.

- Due to hardware restrictions not all performance events can be measured at the same time. To circumvent this we use multiplexing, i. e. we partition the events into sets that can be measured at the same time and switch between them. The value of an event in a period where it is not measured must then be extrapolated.

  We choose a time slice of 100 ms to switch between sets and our choice of events resulted in 3 sets. Since the executed benchmarks do not exhibit significant behavioural changes or periodicity in their inner loops, this should not result in significant measurement problems if a run takes significantly longer than 300 ms.

- If not explicitly stated otherwise, a performance counter measures speculative executions that may not be retired at the end if a previous branch has been mispredicted. This behaviour leads to overcount compared to an analysis in a simpler machine model such as the RAM model. Arguably, this behaviour is actually advantageous to account for the complexities of real-world machines.

- On modern `x86` processors there are multiple ways to execute and count floating point operations, so it is not obvious what we mean by FLOPS. In particular, we may use the 128-bit SSE and the 256-bit AVX registers both for scalar and vector operations with single-precision and double-precision floating point numbers. One could also use the legacy `x87`-FPU or additionally count floating point loads and stores, but this is not common and we do not do so.

  Our approach: We only use double-precision floating point numbers and count the number of scalar double SSE operations ($\mathrm{SSE}_{\mathrm{SD}}$), vector double SSE operations ($\mathrm{SSE}_{\mathrm{VD}}$) as well as vector double AVX operations ($\mathrm{AVX}_{\mathrm{VD}}$).

  We then define FLOPS as

  $$\mathrm{FLOPS} = \mathrm{SSE}_{\mathrm{SD}} + 2 \cdot \mathrm{SSE}_{\mathrm{VD}} + 4 \cdot \mathrm{AVX}_{\mathrm{VD}}.$$

  Thus, we assume that every vector operation uses all available entries of the vector. This may overcount operations on the boundaries of data structures, e. g. on the boundary of a matrix row or a vector.

Number of iterations

Aside from the actual machine performance we also look at more abstract performance measures of the algorithm such as its number of iterations.

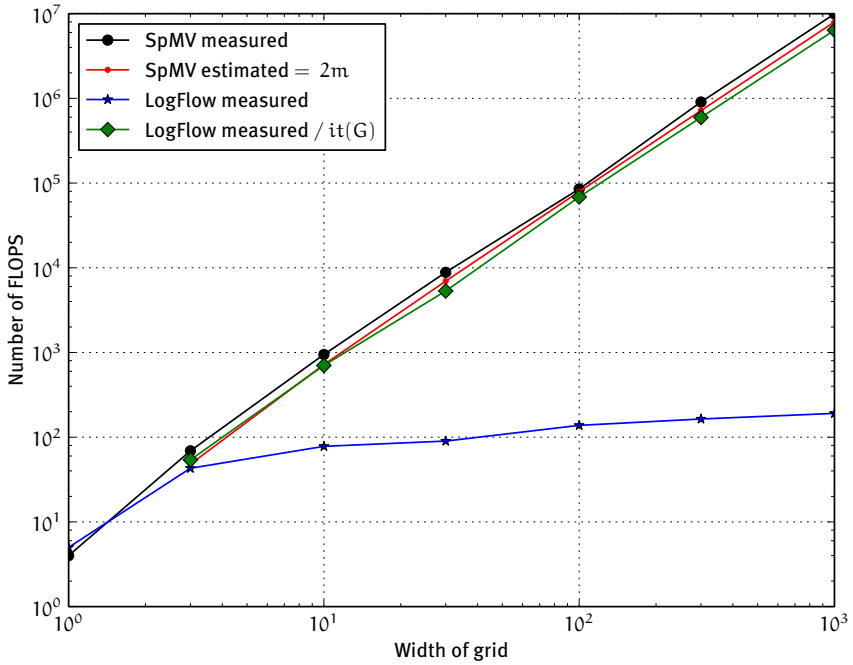


Figure 5.1: Flops for an SpMV and for repairing a cycle with LogFlow on a $k \times k$ grid.

It is hard to compare this number of iterations to more common iterative solvers since these solvers do far more work in a single iteration. Their cost per iteration is usually dominated by a few sparse matrix-vector multiplications (SpMVs), while our solver only locally repairs a single cycle in each iteration. For example, conjugate gradient needs one SpMV per iteration.

We can roughly compare the cost of operations. An SpMV needs $2m + O(1)$ FLOPS, while repairing a cycle with LogFlow needs at most $12 \log_2(n) + O(1)$ FLOPS. Here n is the number of rows of the Laplacian and m is its number of nonzeroes. We get the latter result by noting that a cycle repair requires two queries and two updates. A query is a sparse dot-product, i. e. it costs $2s + O(1)$ FLOPS if the sparsity is s. An update is a sparse vector addition that only needs $s + O(1)$ FLOPS. As we have at most $\log_2(n) + O(1)$ levels in the tree decomposition of LogFlow and we store two values in each level, the sparsity s is in $2 \log_2(n) + O(1)$. Thus, we can roughly estimate that an iteration of the Laplacian solver costs $it(G) := (12 \log_2 n)/(2m)$ SpMVs.

We tested these estimates by measuring FLOPS for an SpMV and a LogFlow cycle repair on $k \times k$ grids. As Figure 5.1 shows, $\mathrm{it}(G)$ is correct within a factor of 2 and slightly overestimates the cost of a cycle repair. So we can use it for rough comparisons.

5.1.3 Experimental setup

In the description of the solver so far we did not state our termination condition and Kelner et al. [Kel+13] only give a theoretical expected number of iterations to achieve a desired error in $\|\cdot\|_L$. We choose, as usual in iterative solvers, to terminate when the relative residual $\|Ax - b\|_2 / \|b\|_2$ is smaller than a given $\epsilon > 0$.

Unfortunately, the solver cannot keep track of the residual. To get it, we must first compute the dual potential vector x. Since this takes $\mathcal{O}(m \log(n))$ time, we cannot update the residual every iteration. Therefore, to still get provably nearly-linear time we heuristically choose to update it every m iterations. Informal experiments show that computing the residuals takes less than 3% of the global time and that only updating every m iterations does not prolong convergence more than 4% in all of our tests.

In the last section we saw how to get accurate performance measurements. Still, these measurements can vary significantly between runs for a number of reasons that we now account for.

The first source of variance is of course that the algorithm is probabilistic. As usual, we avoid this variance by getting the randomness from a pseudorandom generator (MT19937) seeded with a 32-bit value fixed at the start.

The other sources of variance, the hardware and the OS, are much harder to deal with. This system-dependent variance mainly affects the time and cycle counters, while the FLOPS are barely affected by it.

Our most basic choice to reduce these errors is to repeat the benchmark multiple times and average the values gathered. In our case, we repeated each measurement 10 times. This number is quite arbitrary and is mainly motivated by time constraints. Since the resulting measurements are not skewed, we believe that the central limit theorem (an asymptotic theorem) is already applicable for these 10 runs. Given that the measured standard deviations are below 5%, the real counter values are within $-\operatorname{erf}(0.025) \cdot 5\%/\sqrt{10} \approx 3\%$ of the measured mean value with 95% confidence.

In addition, we start each series of runs with a dry run that fills the caches. Thus, we take an optimistic approach with regards to cache usage. Another choice would be to pessimistically flush all caches at the start of each run. With this constant cache usage we can then assume that the runs are independent.

We also account for some of the hardware issues more systematically:

- IEEE floating point numbers provide gradual underflow by using denormalized numbers for values close to zero. Unfortunately, as, for example, Bjørndalen and Anshus [BA07] show, denormalized operations can be up to ten times slower than normal floating point operations. To be able to get consistent results for different inputs and algorithms, we flush denormalized numbers to zero.
- The benchmarking computer has two NUMA nodes. To get consistent memory access times we used the `numactl` command to ensure that the memory was allocated and the threads were scheduled on one of these nodes:

  ```
  numactl --membind 0 --cpunodebind 0
  ```

- Except for testing parallelism, we additionally pinned the threads to a single core to take full advantage of per-core caches:

  ```
  numactl --physcpubind=0
  ```

- We locked all memory pages with the `mlockall(MCL_FUTURE)` library function to avoid swaps to disk.

5.2 Components of the algorithm

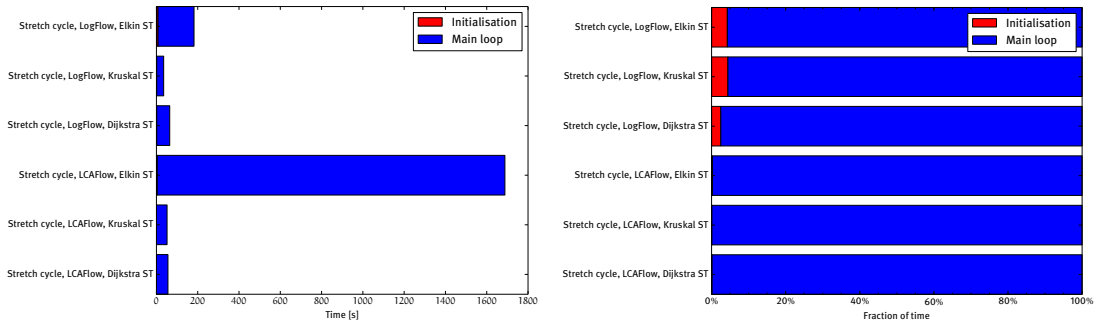
In this section we look at the choices we can make when implementing the solver introduced in Chapter 4.

We first briefly discuss an improvement of the solver (Section 5.2.1). Then we show that the time spent in the initialisation is negligible when compared to the main loop (Section 5.2.2), i. e. we should choose the components based on how they influence convergence and not their initialisation cost. We conclude by discussing the impact of the spanning tree (Section 5.2.3), the flow data structure (Section 5.2.4) and the cycle selection (Section 5.2.5) on convergence.

5.2.1 Improved solver

The solver described in Algorithm 2 is actually just the `SimpleSolver` in Kelner et al.'s [Kel+13] paper. They also show how to improve this solver by adapting preconditioning to the setting of electrical flows.

Their approach is to use multiple runs of the `SimpleSolver` on modified graphs and to take the computed flow of one run as the initial flow of the following run. In each run they modify the graph by scaling down the ST T, i. e. they run the solver on the graph $G - T + T/\alpha$ for some $\alpha \geqslant 1$. This scaling operation improves the stretch of the



(a) Absolute time

(b) Relative time

Figure 5.2: Time spent in the initialisation phase and the main loop on a 200×200 grid.

ST (it is $(st(T) - n + 1)/a + n - 1$) and should lead to faster convergence at the cost of larger error.

Kelner et al. [Kel+13] show how to carefully choose the number of runs, the scaling factors and the random distributions of iteration counts for each run in order to improve the factor $\log(\epsilon^{-1} \log n)$ in the running time to $\log(\epsilon)$ in the `FullSolver`.

When trying to adopt this preconditioning scheme to the practical setting with relative residuals, we were presented with the problem of how to distribute the residual improvements among the runs. Assuming that the initial residual is r_0, we want the residual to be ϵ and we have R runs. Then we could uniformly distribute the residual improvements among the runs, i. e. we stop each run when it has improved the residual by a factor of $(r_0/\epsilon)^{1/R}$. Other possible choices are, for example, to weight the residual improvements by the mean of their number of iterations as chosen by Kelner et al. or to just use a fixed number of iterations with scaled-down ST.

In informal experiments we could not determine a strategy that is consistently better than the `SimpleSolver`, so we did not pursue this preconditioning scheme any further. The problem with preconditioning is compounded by the fact that we also need to rebuild the flow data structure when modifying the graph.

5.2.2 Initialisation

Figure 5.2 shows how much of the time is spent on initialising the data structures and the main loop on a 200×200 grid. We see that the time spent in the initialisation phase is $< 5\%$ of the whole time even when using the sophisticated LogFlow data structure and the Elkin ST.

Since this behaviour is also visible in other experiments, we infer that the speed of the solver is mainly determined by the speed of convergence and it is worthwhile using

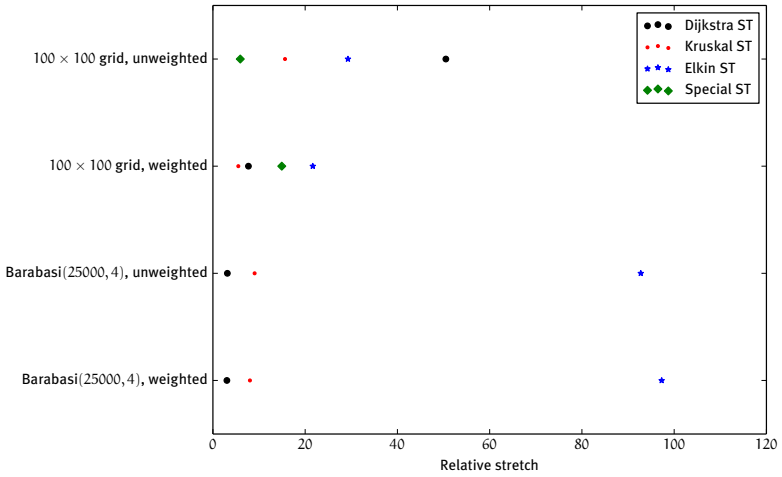


Figure 5.3: Average stretch $\mathrm{st}(T)/m$ with different ST algorithms.

sophisticated approaches if they help convergence. From here on we will not benchmark the initialisation separately.

5.2.3 Spanning tree

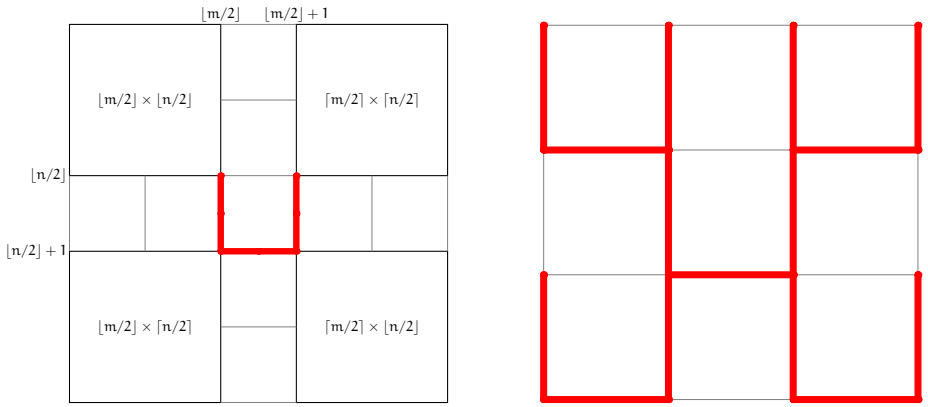
Papp [Pap14] tested various low-stretch spanning tree algorithms and found that in practice the provably good low-stretch algorithms do not yield better stretch than simply using Kruskal. We confirmed this observation by comparing our own implementation of Elkin et al.'s [Elk+05] low-stretch ST algorithm to Kruskal and Dijkstra in Figure 5.3.

Except for the unweighted 100×100 grid, Elkin has worse stretch than the other algorithms and Kruskal yields a good ST. For Barabási-Albert graphs, Elkin is extremely bad (almost factor 20 worse). Interestingly, Kruskal outperforms the other algorithms even on the unweighted Barabási-Albert graphs where it degenerates to choosing an arbitrary ST.

To test how dependent the algorithm is on the stretch of the ST, we also look at a *special ST* for $m \times n$ grids that can easily be shown to have $\mathcal{O}\big(\log(mn)\big)$ average stretch. As depicted in Figure 5.4, we construct this spanning tree by subdividing the $m \times n$ grid into four subgrids as evenly as possible, recursively building the STs in the subgrids and connecting the subgrids by a U-shape in the middle.

Proof sketch for $\mathcal{O}\big(\log(mn)\big)$ *average stretch:* We can inductively show that the stretch $S(m, n)$ of the special ST on the $m \times n$ grid is in $\mathcal{O}\big(mn \log(mn)\big)$.

To do so, we first prove that by the recursive construction the distance of a node on a border of the grid to a corner of the same border is in $\mathcal{O}(m + n)$. Thus, the stretches



(a) Recursive construction (b) ST for $n = m = 4$

Figure 5.4: Spanning tree with $\mathcal{O}(nm \log(nm))$ stretch for the $m \times n$ grid.

of the $m + n - 3$ off-tree edges between the rows $\lfloor n/2 \rfloor$ and $\lfloor n/2 \rfloor + 1$ as well as the columns $\lfloor m/2 \rfloor$ and $\lfloor m/2 \rfloor + 1$ are in $\mathcal{O}(m + n)$. Thus,

$$S(m, n) = 4 \cdot S(m/2, n/2) + \mathcal{O}((m + n)^2)$$

when disregarding rounding. After solving this recurrence we get

$$S(m, n) = \mathcal{O}(mn \log(mn)).$$

In Figure 5.3 we confirm that this special ST yields significantly lower stretch for the unweighted 2D grid, but it does not help in the weighted case.

5.2.4 Flow data structure

In Section 4.2 we introduced two implementations of a data structure for repairing cycles: The trivial LCAFlow that needs $\mathcal{O}(n)$ worst-case time and a sophisticated LogFlow that only needs $\mathcal{O}(\log n)$ time.

While the LogFlow implementation is necessary for good worst-case performance, we now check whether it is worthwhile for practical instances. To compare the data structures independently of low-level details, we introduce two abstract performance measures for updating a cycle between u and v:

1. For LCAFlow the cost is twice the number of nodes in $P_T(u, v)$, once for querying the cycle and once for updating it.
2. For LogFlow the cost is the sum $|q(u)| + |q(v)| + |l(u)| + |l(v)|$ of the sparsities of the update and query vectors.

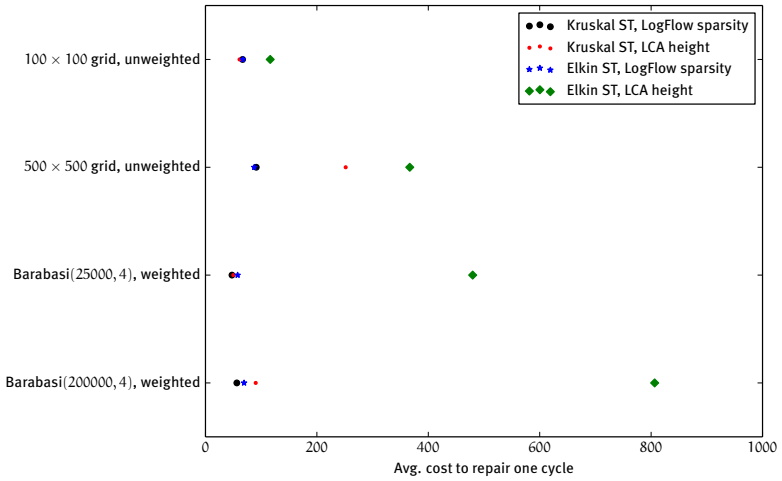


Figure 5.5: Average cost of updating a cycle with the flow data structures.

In Figure 5.5 we compare the average costs of updating a basis cycle using LCAFlow and LogFlow. Unsurprisingly, we see that the cost of LCAFlow significantly depends on the structure of the used spanning tree, while the LogFlow costs stay nearly the same. Similarly, the cost of LCAFlow grows far more with the size of the graph than LogFlow and LogFlow wins for the larger graphs in both classes.

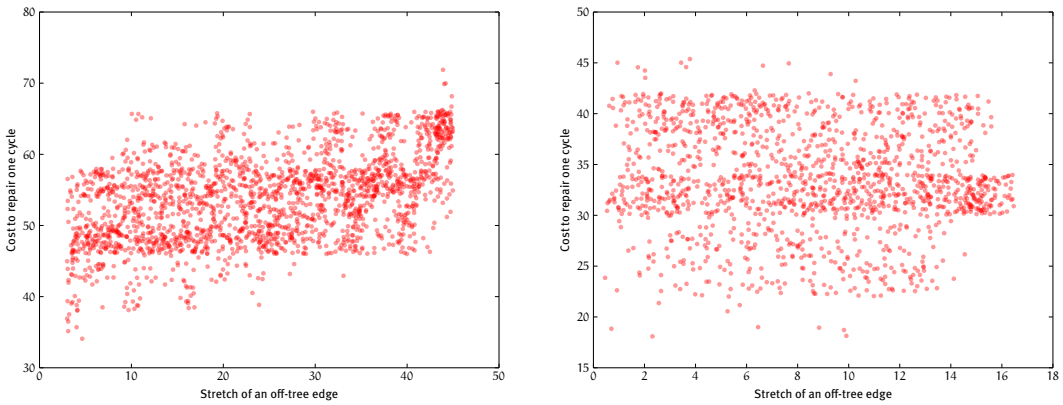
For these reasons, we only use LCAFlow in the following benchmarks.

5.2.5 Cycle selection

The third choice we have to make is how to randomly select the basis cycle to be fixed: We either give every cycle the same weight or we weight them by their stretch.

We expect to get better energy improvement by preferably fixing the cycles with higher stretch. If that is indeed the case, then Figure 5.6 suggests that we can get good running time with this weighting. In this figure we plot the cost of fixing the cycle versus the stretch of the cycle for two graphs. In the unweighted graph the stretch of a cycle is the same as its number of edges. Therefore, the cost of fixing it increases slightly with its stretch. In the weighted case the cost of fixing a cycle is completely independent of its stretch. Thus, we should use whatever cycle results in the best energy improvement and do not need to worry about the cost of fixing it.

We check the energy in Figure 5.7 with a scatter plot of the energy improvement when repairing a basis cycle versus the stretch of the corresponding off-tree-edge. Unfortunately, we cannot make out a clear trend that higher stretch results in better energy improvement. As a matter of fact, for the grid we actually get a slight downward trend.



(a) 30×30 grid, unweighted

(b) Barabasi$(1000, 4)$, weighted

Figure 5.6: Cost of repairing a basis cycle with LogFlow versus the stretch of the corresponding off-tree-edge. We added a jitter of ± 0.5 to every stretch.

Thus, we cannot make a decision about which cycle selection to use just yet and we will test both strategies in the following benchmarks.

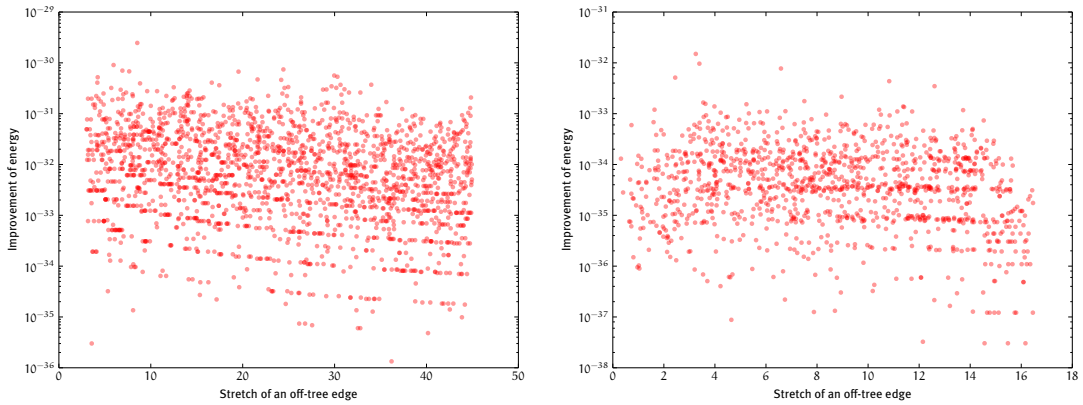
5.3 Convergence

In Section 5.2 we looked at the performance of the components of the algorithm in isolation. In this section we analyse the global convergence behaviour for different choices of the cycles and the spanning trees. We do not vary the flow data structure since it does not affect convergence.

In Figure 5.8 and Figure 5.9 we plot the convergence of the residual and the gap to the optimal energy for different graphs and different algorithm settings. We examined a 100×100 grid and a Barabási-Albert graph with 25,000 nodes. In this experiment we determined the energy gap $\xi_r(f) - \xi_r(f_{opt})$ by fixing the optimal solution x and taking Lx as right hand side, i. e. $\xi_r(f_{opt}) = \zeta_r(x)$.

As expected, the energy in all runs decreases monotonically. While the residuals can increase, they follow the same global downward trend. Also note that the spikes of the residuals are smaller if the convergence is better and that the order (by convergence speed) of the residual curves and the energy curves is the same.

In all cases the solver converges exponentially, but the convergence speed crucially depends on the solver settings. If we select cycles by their stretch, the order of the convergence speeds is the same as the order of the stretches of the ST (compare Figure 5.3), except for the Dijkstra ST and the Kruskal ST on the weighted grid. In particular, for



(a) 30×30 grid, unweighted

(b) Barabasi$(1000, 4)$, weighted

Figure 5.7: Energy improvement when repairing a basis cycle versus the stretch of the corresponding off-tree-edge. We added a jitter of ± 2 to every cost.

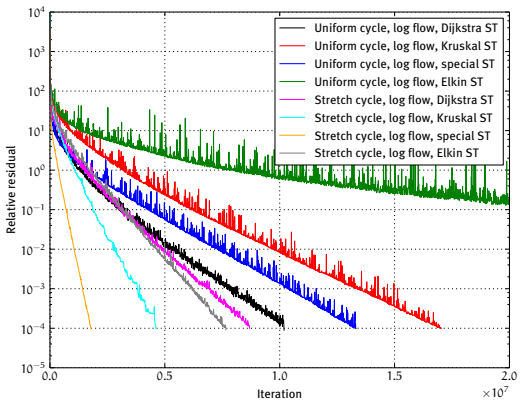
the Elkin ST on Barabási-Albert graphs there is a significant gap to the other settings where the solver barely converges at all and the special ST wins. Thus, low-stretch STs are crucial for convergence. In informal experiments we also saw this behaviour for 3D grids and nonsynthetic graphs Section 5.1.1.

In contrast, for the uniform cycle selection on the unweighted grid, the special ST is superior over the Kruskal ST, even though its stretch is smaller. This is caused by the fact that the basis cycles with the Kruskal ST are longer than the basis cycles with the special ST and fixing them helps more. Still, the other curves with uniform cycle selection follow the stretch.

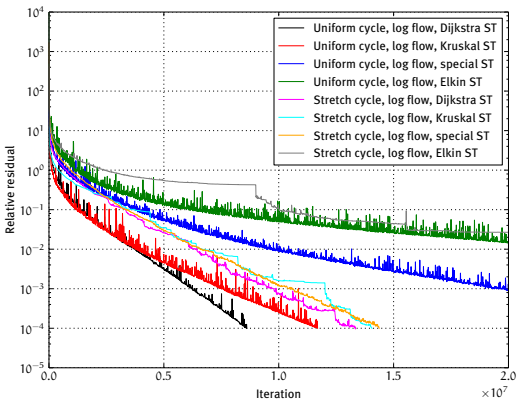
In Section 5.2.5 we already saw that we could not detect any correlation between the energy improvement and the stretch of the cycle. Therefore, we cannot fully explain the different speeds with uniform cycle selection and stretch cycle selection. For the grid the stretch cycle selection wins, while Barabási-Albert graphs favour uniform cycle selection.

Another interesting observation is that most of the convergence speeds stay constant after an initial fast improvement at the start to about residual 1. That is, there is no significant change of behaviour or periodicity.

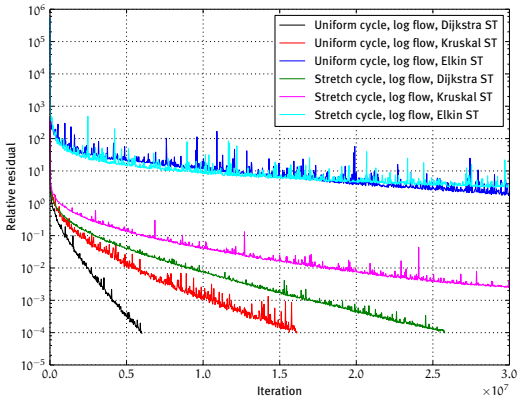
Even though we can hugely improve convergence by choosing the right settings, even the best convergence is still very slow, e. g. we need about 6 million iterations ($\approx$ 3000 SpMVs) on a Barabási-Albert graph with 25,000 nodes and 100,000 edges in order to



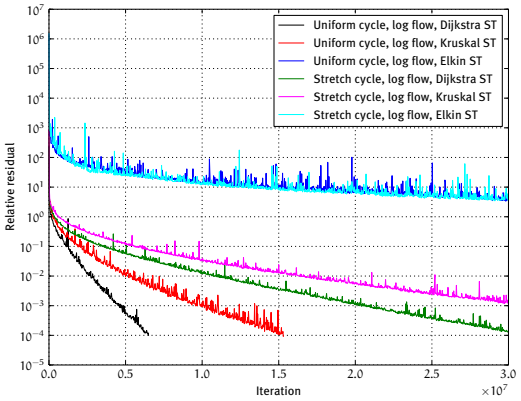
(a) 100×100 grid, unweighted



(b) 100×100 grid, weighted



(c) Barabási–Albert, $n = 25000$, unweighted



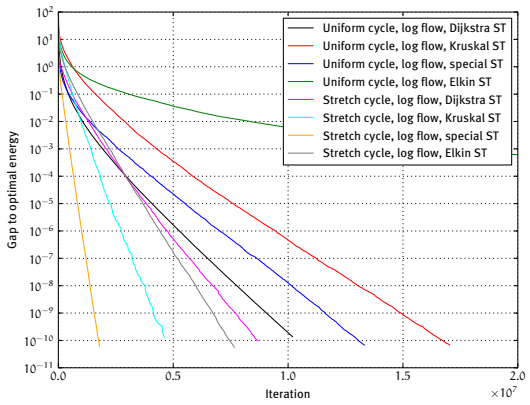
(d) Barabási–Albert, $n = 25000$, weighted

Figure 5.8: Convergence of the residual. Terminate when residual $\leqslant 10^{-4}$.

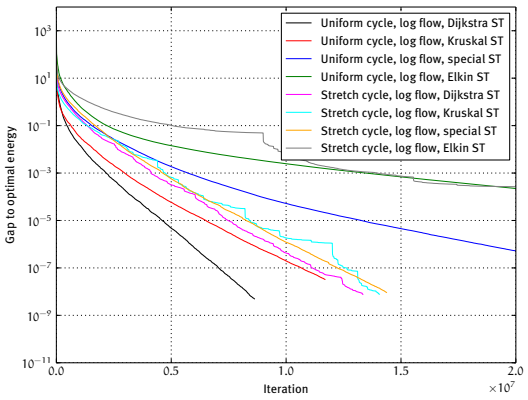
reach residual 10^{-4}. In contrast, CG without preconditioning only needs 204 SpMVs for this graph

5.4 Asymptotics

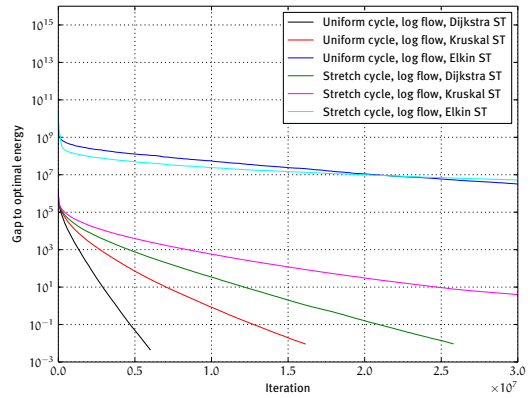
In Section 5.3 we saw which settings of the algorithm yield the best performance for 2D grids and Barabási-Albert graphs. Now we look at how the performance with these settings behaves asymptotically and how it compares to well-established iterative solvers. In particular, we will only compare the algorithm to the conjugate gradient (CG) method without preconditioning, one of the simplest and most popular iterative solvers. Since our solver will turn out to not be competitive at all, we do not need to compare it to more sophisticated algorithms.



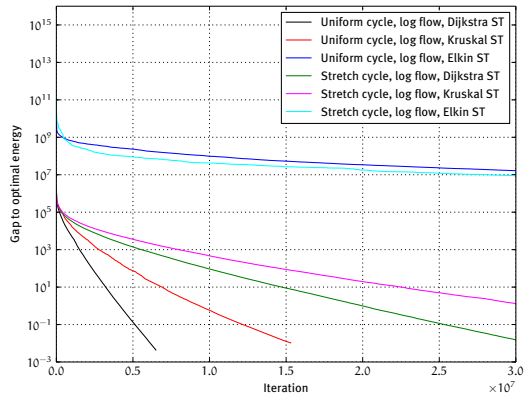
(a) 100×100 grid, unweighted



(b) 100×100 grid, weighted



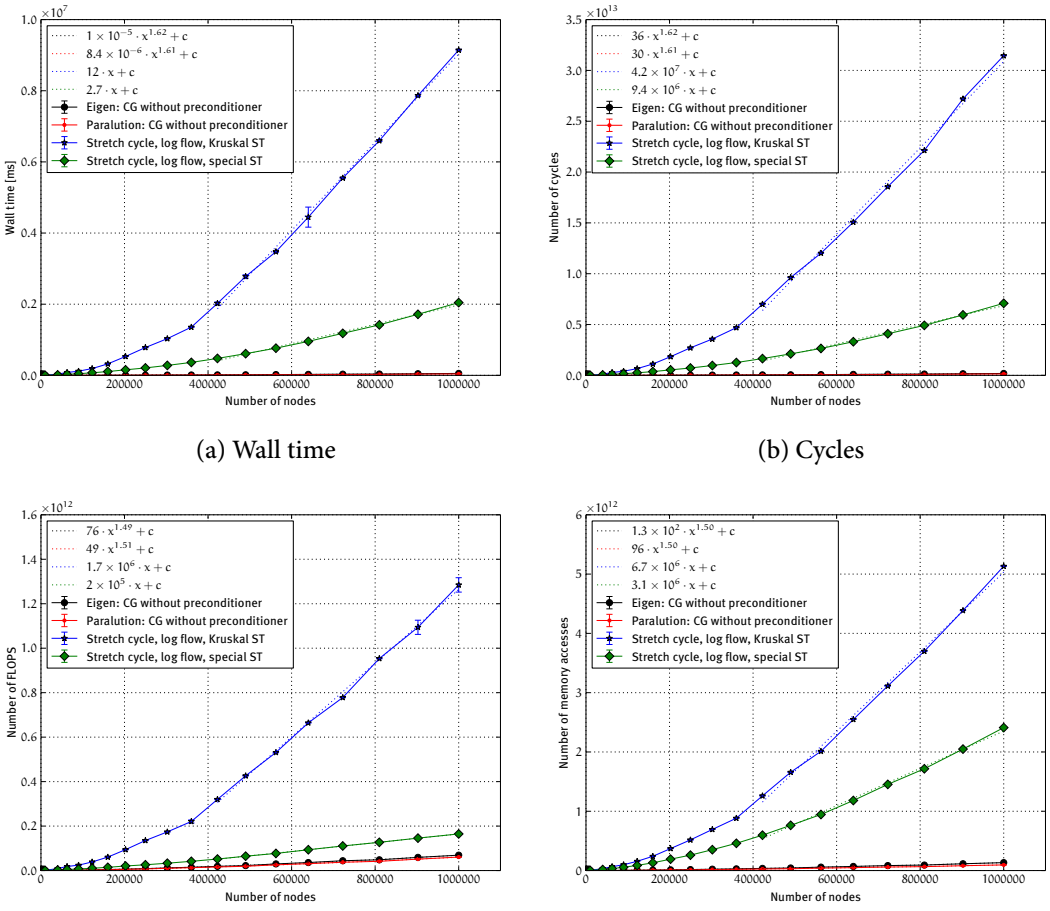
(c) Barabási–Albert, $n = 25000$, unweighted



(d) Barabási–Albert, $n = 25000$, weighted

Figure 5.9: Convergence of the energy. Terminate when relative residual $\leqslant 10^{-4}$.

Let us first look at the 2D grids in Figure 5.10. In this figure each occurrence of c stands for a new instance of a real constant. We expect the cost of the CG method to scale with $\mathcal{O}(n^{1.5})$ on 2D grids [Dem97], while our algorithm should scale nearly-linearly. This expectation is confirmed in the plot: Using Levenberg-Marquardt [Mar63] to approximate the curves for CG with a function of the form $ax^b + c$ we get $b \approx 1.5$ for FLOPS and memory accesses, while the (more technical) wall time and cycle count yield a slightly higher exponent $b \approx 1.6$. We also see that the curves for our algorithm are almost linear from about 650×650. Unfortunately, the hidden constant factor is so large that our algorithm cannot compete with CG even for a 1000×1000 grid.



(a) Wall time

(b) Cycles

(c) FLOPS

(d) Memory accesses

Figure 5.10: Asymptotic behaviour for 2D grids. We terminated when the relative residual was $\leqslant 10^{-4}$. The error bars give the standard deviation.

Note that the difference between the algorithms in FLOPS is significantly smaller than the difference in memory accesses and that the difference in running time is larger still. This suggests that the practical performance of our algorithm is particularly bounded by memory access patterns and not by floating point operations.

This is noteworthy when we look at our special spanning tree for the 2D grid. We see that using the special ST always results in performance that is better by a constant factor. In particular, we save a lot of FLOPS (factor 10), while the savings in memory accesses (factor 2) are a lot smaller. Even though the FLOPS when using the special ST are within a factor of 2 of the CG method, we still have a wide chasm in the running time.

But note that later in Section 5.8 we show that the micro-performance of the solver is actually very competitive with CG. Thus, the bad running time is mainly caused by the very slow convergence that we have already seen in Section 5.3.

The results for the Barabási-Albert graphs in Figure 5.11 are basically the same: Even though the growth is approximately linear from about 400,000 nodes, there is still a large gap between our algorithm and CG since constant factor is enormous. Also, the results for the number of FLOPS are again much better than the result for the other performance counters.

In conclusion, although we have nearly-linear growth, even for 1,000,000 nodes our algorithm is still not competitive with CG because of huge constant factors, in particular a large number of iterations (compare Section 5.3).

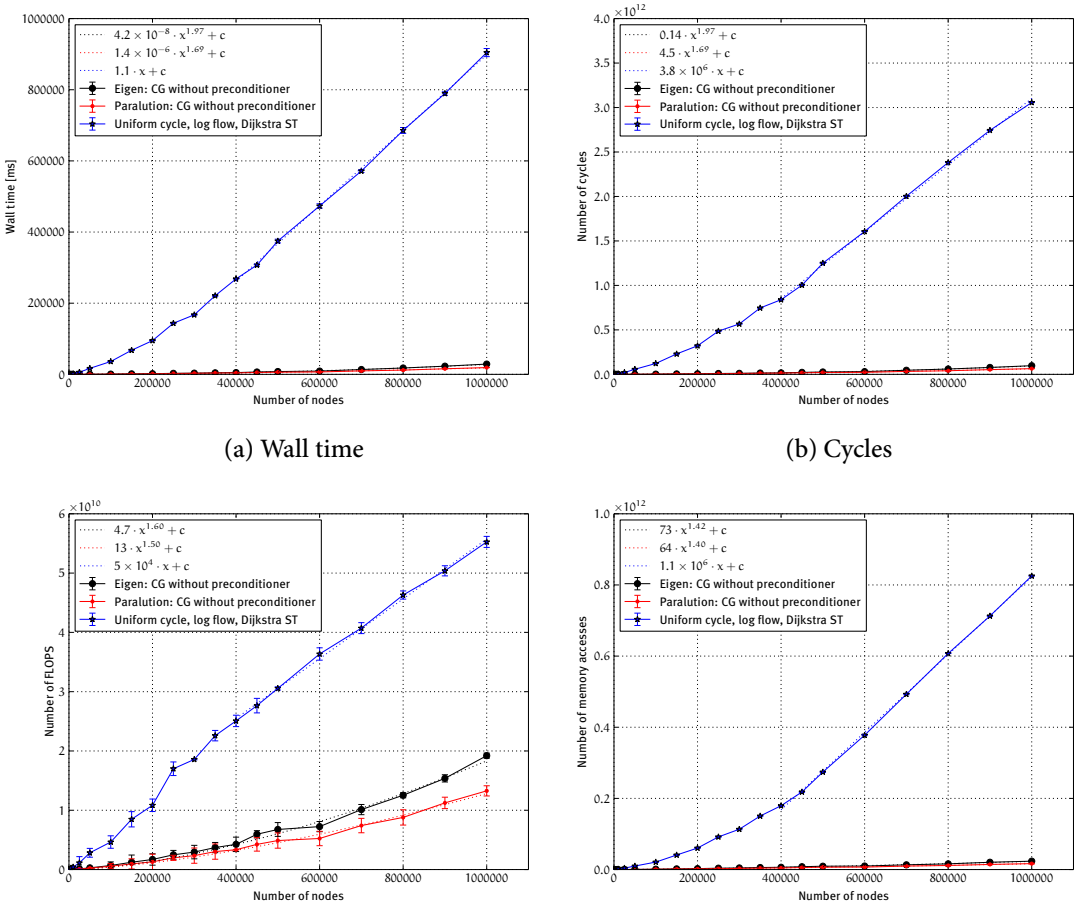
5.5 Preconditioning

Some linear solvers, such as Gauss-Seidel, are good preconditioners even though they are slow when used on their own. In this section we check whether this is the case for our Laplacian solver.

The convergence of most iterative linear solvers on a linear system $Ax = b$ depends on the *condition number* $\kappa(A) := \|A^{-1}\|\|A\|$ of A. The smaller the condition number is, the better the solvers converge. A common way to improve the condition number is to find a matrix P such that $\kappa(P^{-1}A) < \kappa(A)$ and then solve the system $P^{-1}Ax = P^{-1}b$ instead of $Ax = b$.

In iterative methods we usually do not explicitly compute $P^{-1}A$ but apply P^{-1} and A separately to the current vector in each iteration. In our case we use a few iterations of the Laplacian solver as a preconditioner in each iteration instead of taking a fixed matrix P.

Since the solver only works for SDD matrices, we need to use an iterative solver that only passes SDD matrices to the preconditioner. We choose Krylov subspace methods. In



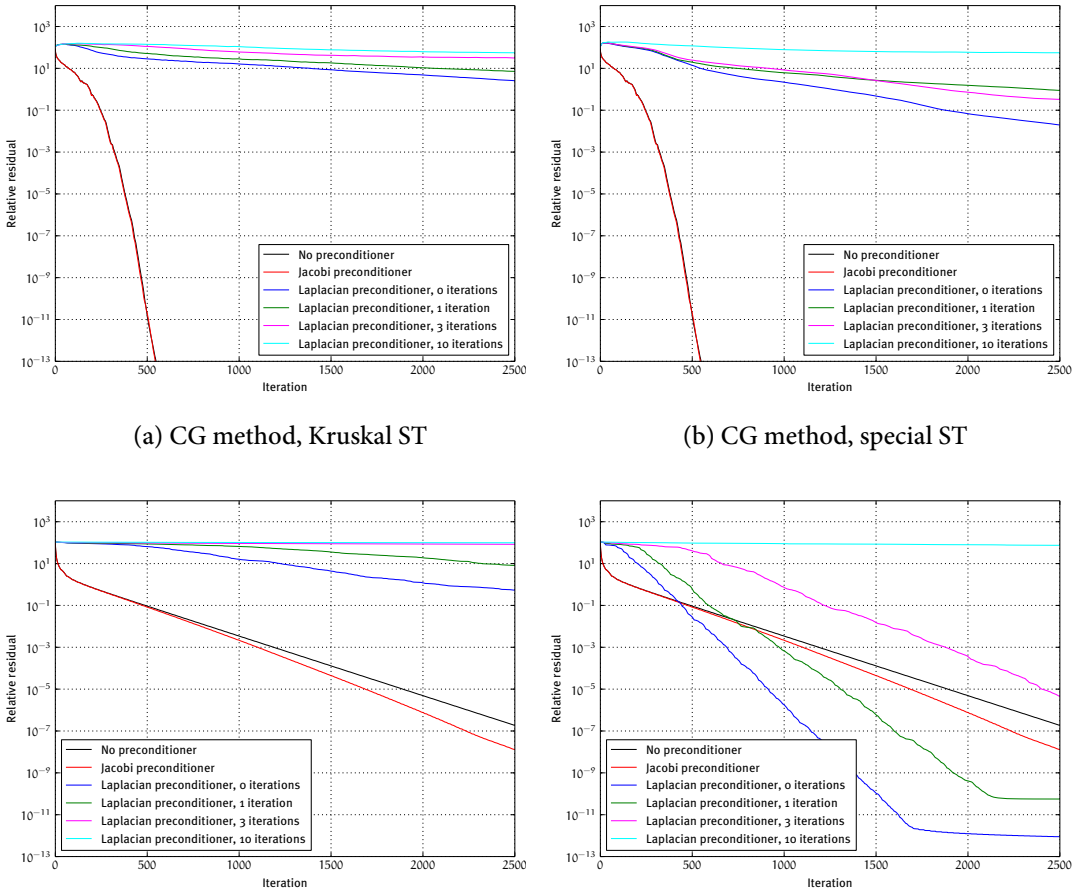
(a) Wall time

(b) Cycles

(c) FLOPS

(d) Memory accesses

Figure 5.11: Asymptotic behaviour for Barabási–Albert graphs. We terminated when the relative residual was $\leqslant 10^{-4}$. The error bars give the standard deviation.



(a) CG method, Kruskal ST

(b) CG method, special ST

(c) FGMRES method, Kruskal ST

(d) FGMRES method, special ST

Figure 5.12: Convergence of the residual when using the Laplacian solver as a preconditioner on an unweighted 100×100 grid.

particular, we tested the CG method and the FGMRES method on an unweighted 100×100 grid. The convergence of the residual with these solvers is plotted in Figure 5.12.

For the CG method we see that, unfortunately, the more iterations we use, the more slowly the methods converge. Since the cycle repairs depend crucially on the right hand side and the solver is probabilistic, using the Laplacian solver as preconditioner means that the preconditioner matrix is not fixed but changes from iteration to iteration. Axelsson and Vassilevski [AV91] show why this behaviour leads to convergence problems and propose a CG method with variable-step preconditioning to cope with it.

In practice the flexible GMRES method is often more resistant to these convergence problems. Since the initial vector on the special ST is very good, we get good convergence in Figure 5.12 when using zero iterations of the solver in FGMRES, a behaviour that is obviously not generalisable. For more iterations of the Laplacian solver FGMRES still has convergence problems, but it is somewhat better than CG.

We conclude that we cannot use the Laplacian solver as a preconditioner for common iterative methods. It would be an interesting extension to check whether the solver works in a specialised variable-step method.

5.6 Smoothing

Another way to combine the good qualities of two different solvers aside from preconditioning is smoothing. Smoothing means that we use one solver to dampen the low-frequency components of the error and another to dampen the high-frequency components.

In CG and most other solvers we know of the low-frequency components of the error converge very fast, while the high-frequency components converge slowly. Thus, we are interested in finding an algorithm that dampens the high-frequency components, a *good smoother*. This smoother does not necessarily need to reduce the error, it just needs to make its frequency distribution more favourable. Smoothers are particularly often applied at each level of multigrid or multilevel schemes [BHM00] that turn a good smoother into a good solver by applying it at different levels of hierarchy.

To test whether the Laplacian solver is a good smoother, we start with a fixed x with $Lx = b$ and add white uniform noise in $[-1, 1]$ to each of its entries in order to get an initial vector x_0.

Then we execute a few iterations of our Laplacian solver and check whether the high-frequency components of the error have been reduced. Unfortunately, as described later in Section 5.7.1, we cannot directly start at the vector x_0 in the solver. Our solution is to use *Richardson iteration*. That is, we transform the residual $r = b - Lx_0$ back to the

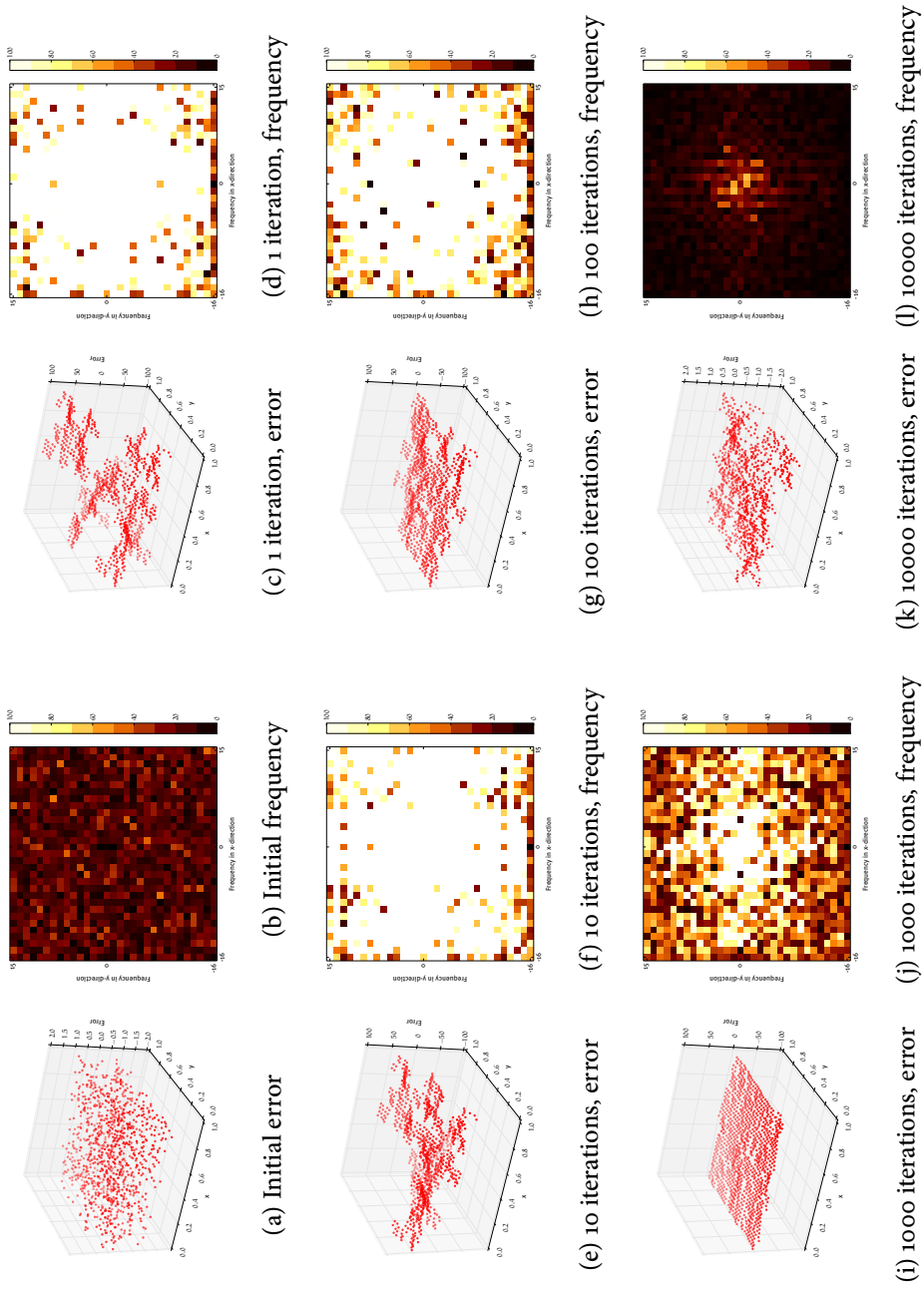


Figure 5.13: The Laplacian solver with the special ST as a smoother on a 32×32 grid. For each number of iterations of the solver we plot the current error and the absolute values of its transformation into the frequency domain. Note that (a) and (k) have a different scale.

source space by computing $L^{-1}r$ with the Laplacian solver, get the error $e = x - x_0 = L^{-1}r$ and then the output solution

$$x_1 = x_0 + L^{-1}r.$$

Figure 5.13 shows the error vectors of the solver for a 32×32 grid together with their transformations into the frequency domain for different numbers of iterations of our solver. We see that the solver is indeed useful as a smoother since the energies for the large frequencies (on the periphery) decrease rapidly, while small frequencies (in the middle) in the error remain.

In the solver we start with a flow that is nonzero only on the ST. Therefore, the flow values on the ST are generally larger at the start than in later iterations, where the flow will be distributed among the other edges. Since we construct the output vector by taking potentials on the tree, after one iteration x_1 will, thus, have large entries compared to the entries of b.

In subplot (c) of Figure 5.13 we see that the start vector of the solver has the same structure as the special ST and that its error is very large. For the 32×32 grid we, therefore, need about 10000 iterations ($\approx$ 150 SpMVs) to get an error of x_1 similar to x_0 even though the frequency distribution is favourable. But note that the number of SpMVs the 10000 iterations correspond to depends on the size of the graph, e. g. for an 100×100 grid the 10000 iterations correspond to 20 SpMVs. In Section 5.4 we also saw that the number of required iterations grows nearly-linearly. Thus, testing the Laplacian solver in a multigrid scheme could be worthwhile.

However, the bad initial vector creates problems when applying the Richardson iteration multiple times with a fixed number of iterations of our solver. In informal tests multiple Richardson steps lead to ever increasing errors without improved frequency behaviour unless our solver already yields an almost perfect vector in a single run.

5.7 Practical problems

In this section we briefly describe several minor problems that hinder the use of the Laplacian solver. We show that we are not able to provide a start vector (Section 5.7.1), we need to deal with the nontrivial kernel of the Laplacian (Section 5.7.2) and we need to heuristically distribute the residual among the components (Section 5.7.3).

5.7.1 Initial solution

One problem is the impedance mismatch between INV-LAPLACIAN-POTENTIAL and INV-LAPLACIAN-CURRENT, i. e. we cannot get from a vector of potentials x to a corresponding graph flow f.

Given x we can compute a flow f via $f_{uv} := x(u) - x(v)$. Since this flow is induced by a vector, the potential drop of each cycle in f is zero (property (2') in Section 3.1.2). Unfortunately, this flow is not a valid graph flow (property (1) in Section 3.1.2) with demand b unless x already fulfils $Lx = b$. In contrast, in the solver we iteratively establish (2') from a flow that has property (1). Thus, f is useless for the solver; the solver cannot make any progress from it.

In particular, this means that we cannot start from an arbitrary vector x in the algorithm, which may make it harder to use the solver in a larger context.

5.7.2 Kernel

Whenever we solve a Laplacian system we need to take into account that the Laplacian $L(G)$ is singular, i. e. $Lx = b$ does not have a unique solution but an affine space $\widetilde{x} + \ker L$ of solutions where $\widetilde{x}$ is an arbitrary solution.

The kernel of L is spanned by the vectors

$$(1_C)_v := \begin{cases} 1 & \text{if } v \in C \\ 0 & \text{otherwise} \end{cases}$$

for every component C of G and $v \in V$. That is, we can add a constant to every component of G. The Laplacian solver yields the solution that has a zero at the root of the spanning tree for each component.

This singularity is not a large problem, but the consumer of the solution needs to take it into account and adjust the solution if necessary.

5.7.3 Connected components

Aside from increasing the dimension of $\ker(L)$, more connected components also create the problem of choosing which residual we want for each component in order for the residual of the whole vector to be $\leqslant \epsilon$.

Let $\mathcal{P}$ be the set of components of G. Furthermore, for each $C \subseteq V$ define L_C to be the submatrix of L with rows and columns in C and v_C to be the subvector of an arbitrary vector $v \in \mathbb{R}^V$ with rows in C.

The simplest way to distribute the residual is to weight each component by its size, i. e. to require $\|L_C x_C - b_C\|_2 \leqslant |C|/|V| \cdot \epsilon$ for each $C \in \mathcal{P}$. With Cauchy-Schwarz we then get the desired bound

$$\begin{aligned}\|Lx - b\|_2^2 = \sum_{C \in \mathcal{P}} \|L_C x_C - b_C\|_2^2 &\leqslant \epsilon^2 \cdot \sum_{C \in \mathcal{P}} \frac{|C|^2}{|V|^2} \\ &\leqslant \epsilon^2 \cdot \left(\sum_{C \in \mathcal{P}} \frac{|C|}{|V|} \right)^2 = \epsilon^2.\end{aligned}$$

Another sensible choice is to weight the components by the stretches of their spanning trees to account for Laplacian problems that are harder to solve. We only implemented the first heuristic. Since we do not believe multiple components bring significant insight, we did not evaluate the solver on disconnected graphs.

We could avoid this problem by working on all of the components at the same time and repairing a random cycle from an arbitrary component in each iteration. But this would result in a significantly more complex implementation.

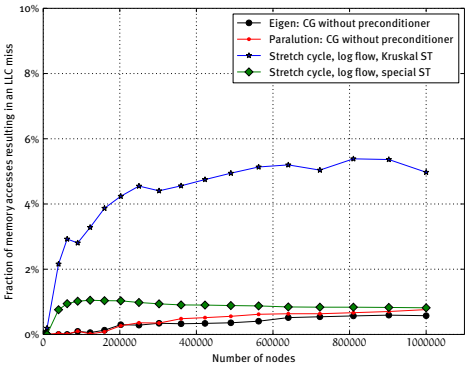
5.8 Micro-performance

The nearly-linear running time of the Laplacian solver was proved in the simplistic RAM machine model. To get good practical performance on modern out-of-order superscalar computers you have to take their complex execution behaviour into account, most prominently the cache hierarchy and data dependencies.

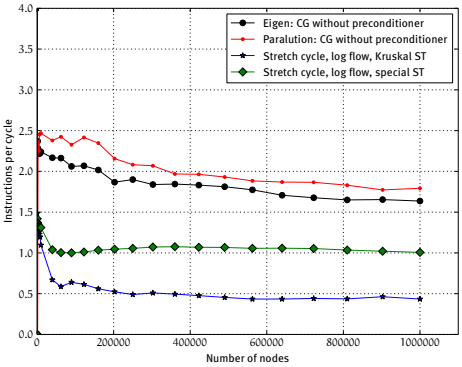
As seen in Figure 5.14, one particular problem when using a bad spanning tree is the number of cache misses in the LogFlow data structure.

Note that querying and updating the flow with this data structure corresponds to a dot product and an addition, respectively, of a dense vector and a sparse vector. The sparse vectors are stored as lists of pairs of indexes (into the dense vector) and values, i. e. you need indirect accesses into the dense vector. The cache behaviour depends on the distribution of the indexes which is determined by the subtree decomposition of the spanning tree and the order of the subtrees. For the CG method we used a compressed CSR representation that also needs indirect accesses for an SpMV.

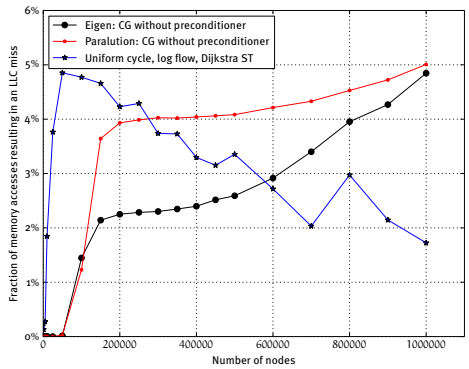
We managed to consistently improve the time by about 6% by doing the decomposition in BFS order, so that the indexes are grouped together at the front of the vector. In contrast, the actual decomposition only depends on the spanning tree. Furthermore, we could save an additional 10% of time by using 256-bit AVX instructions to do four double precision operations at the same time in LogFlow, but this vectorised implementation still uses (vectorised) indirect accesses.



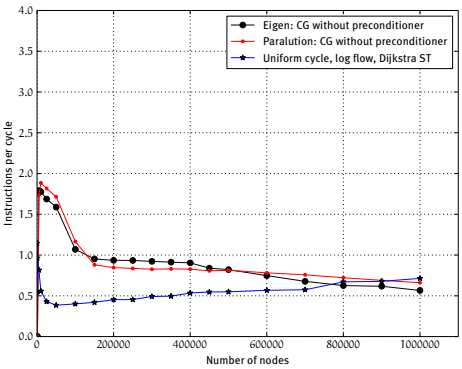
(a) 2D-grid, cache misses



(b) 2D-grid, IPC



(c) Barabási–Albert, cache misses



(d) Barabási–Albert, IPC

Figure 5.14: Last-level cache misses and IPC for 2D grids (unweighted) and Barabási–Albert graphs (weighted).

In Figure 5.14 we see that we get about 5% cache misses by using the minimum weight ST on the 2D grid compared with 1% when using CG. In contrast, the special ST yields competitive cache behaviour.

Interestingly, since the Barabási-Albert graph has a much more complex structure, its cache misses using the sparse matrix representation increase to 5%. In contrast, the cache misses improve for larger graphs with LogFlow since the diameter of the spanning tree is smaller than on grids and the decomposition, thus, groups most indexes at the start of the vector.

Another interesting aspect is the number of instructions issued in each cycle (IPC), a measure of how much of the available superscalar computing power is actually used. The hard limit is that the benchmarked CPU can issue at most 4 instructions per cycle.

Unsurprisingly, we see that for the grid the IPC depends on the cache behaviour and, therefore, the spanning tree. We also see that in the grid case the IPC is significantly better (1.0 vs 1.75) for the CG method. But this can again be blamed on the simple structure of the 2D-grid and for the Barabási-Albert graphs both IPCs are comparable (and much worse!) again.

From the benchmarks we can infer that the micro-performance suffers from indirect accesses just as in the case of the usual sparse matrix representations. Furthermore, the micro-performance crucially depends on the quality of the spanning tree. For good spanning trees or more complex graphs the micro-performance of the Laplacian solver is competitive with CG.

5.9 Parallelisation

While the single-core performance of CPUs is still improving, today most performance improvements can be achieved by putting more cores on a chip [Suto5]. It is, therefore, ever more crucial to use parallel algorithms.

As we see in this section, there are two basic ways of parallelising the solver in a shared memory setting, both of which do not scale very well:

1. We can parallelise each single query/update of the LogFlow data structure. This is easy since a query is just a sparse dot product and an update is a sparse addition.

   Unfortunately, even for larger graphs the vectors are so sparse that parallelising the operations never outweighed the cost of the barrier synchronisation after each operation in our tests. For example, the average density is just ≈ 97 for a 1000×1000 grid.

2. We could also update multiple cycles at the same time. When we store each flow on an edge directly, each update consists of a query phase where we determine the amount of current to add to the cycle and an update phase where we update the cycle.

   Between the phases the flow on the cycle needs to remain fixed. If we do not ensure this, we could, for example, update the same cycle twice and get an increase in energy.

   Thus, we need to lock whole cycles; atomic updates of flow values do not suffice. This would create significant synchronization overhead, but could still result in a viable parallelisation if we manage to find many independent cycles.

But, as we saw in Section 5.2.4, we need to use the LogFlow data structure to get good provable and practical performance. This data structure works by decomposing a tree-path into two root-node paths in the decomposition tree. Since all of these paths intersect in the original tree, we cannot update them in parallel.

In conclusion, the solver cannot be parallelised with good scalability without significantly changing its main loop or the flow data structure.

6 Conclusions

In this thesis we implemented and benchmarked the nearly-linear time Laplacian solver presented by Kelner et al. [Kel+13]. At the time of writing this is the first comprehensive experimental study of a nearly-linear time Laplacian solver.

We were able to support the theoretical result that the convergence of the solver crucially depends on the stretch of the chosen spanning tree, with low stretch generally resulting in faster convergence (Section 5.3). This particularly suggests that it is crucial to build algorithms that yield spanning trees with lower stretch. Since we confirmed Papp's [Pap14] observation that the known algorithms with provably low stretch do not yield good stretch in practice (Section 5.2.3), improving the low-stretch ST algorithms is an important future research direction.

We also observed that convergence varies when changing cycle selection, but we could not determine a single best strategy (Section 5.2.3).

Choosing the solver settings with the best convergence, we then analysed the asymptotic running time of the solver (Section 5.4). Unfortunately, even though it proved to grow nearly-linearly, the constant was still too big to make it competitive, even compared to the CG method without preconditioner. This was also the case when we used a well-suited manually constructed spanning tree on a 2D grid with $\mathcal{O}(|E| \log |V|)$ stretch, i. e. we do not expect better spanning trees alone to make the algorithm competitive. One future research direction to improve competitiveness is to repair cycles other than just the basis cycles in each iteration, but this would necessitate significantly different data structures in the solver.

Then we proceeded by looking at how this solver could be used in conjunction with another solver: as a preconditioner (Section 5.5) or as a smoother (Section 5.6). When using it as a preconditioner in a simple Krylov subspace method we got convergence problems. It could be interesting to investigate whether using it as a preconditioner in a specialised variable-step method could alleviate these issues.

In contrast, the solver smoothed out high-frequency components of the error very fast. The caveat is again that the constant factor in the error of the starting guess is very poor.

Thus, the solver could possibly behave well when embedded into a larger multigrid or multilevel scheme. Checking this is an interesting extension of this work.

In conclusion, the basic solver presented by Kelner et al. [Kel+13] is not competitive in practice as is, but we could improve it with better low-stretch spanning trees and other cycle selections.

Furthermore, the solver is hard to parallelise (Section 5.9) and quite complex to implement compared to standard iterative solvers. Thus, we believe that it probably more worthwhile to instead test how other nearly-linear time solvers perform in practice. In particular, Peng and Spielman [PS14] presented an interesting solver based on recursive sparsification. Together with the parallel sparsification algorithm by Koutis [Kou14] this recursive sparsification yields a nearly-linear work parallel algorithm that could scale well in practice.

Bibliography

[ABN08] I. Abraham, Y. Bartal and O. Neiman. "Nearly Tight Low Stretch Spanning Trees". In: *49th Annual Symposium on Foundations of Computer Science*. 2008, pp. 781–790. DOI: 10.1109/FOCS.2008.62 (cit. on pp. 2, 20).

[Alo+95] Noga Alon et al. "A Graph-Theoretic Game and its Application to the k-Server Problem". In: *SIAM Journal on Computing* (1995), pp. 78–100. DOI: 10.1.1.38.1121 (cit. on pp. 2, 19, 20).

[AN12] Ittai Abraham and Ofer Neiman. "Using Petal-decompositions to Build a Low Stretch Spanning Tree". In: *44th ACM Symposium on Theory of Computing*. 2012, pp. 395–406. DOI: 10.1145/2213977.2214015 (cit. on pp. 2, 20, 27).

[AV91] O. Axelsson and P. Vassilevski. "A Black Box Generalized Conjugate Gradient Solver with Inner Iterations and Variable-Step Preconditioning". In: *SIAM Journal on Matrix Analysis and Applications* 4 (1991), pp. 625–644. DOI: 10.1137/0612048 (cit. on p. 48).

[BA07] John Markus Bjørndalen and Otto J. Anshus. "Trusting Floating Point Benchmarks – Are Your Benchmarks Really Data Independent?" In: *Applied Parallel Computing. State of the Art in Scientific Computing*. Springer, 2007, pp. 178–188. DOI: 10.1007/978-3-540-75755-9_23 (cit. on p. 35).

[BA99] Albert-László Barabási and Réka Albert. "Emergence of Scaling in Random Networks". In: *Science* 5439 (1999), pp. 509–512 (cit. on p. 30).

[BF00] Michael A. Bender and Martin Farach-Colton. "The LCA Problem Revisited." In: *LATIN 2000: Theoretical Informatics*. Springer, 2000, pp. 88–94. DOI: 10.1.1.38.6179 (cit. on p. 22).

[BHM00] William L. Briggs, Van Emden Henson and Steve F. McCormick. *A multigrid tutorial*. SIAM, 2000. DOI: 10.1137/1.9780898719505 (cit. on p. 48).

[BHV08] E. Boman, B. Hendrickson and S. Vavasis. "Solving Elliptic Finite Element Systems in Near-Linear Time with Support Preconditioners". In: *SIAM Journal on Numerical Analysis* 6 (2008), pp. 3264–3284. DOI: 10.1137/040611781 (cit. on p. 1).

[Bro+00] S. Browne et al. “A Portable Programming Interface for Performance Evaluation on Modern Processors”. In: *Int. J. High Perform. Comput. Appl.* 3 (2000), pp. 189–204. DOI: 10.1177/109434200001400303 (cit. on p. 31).

[Chr+11] Paul Christiano et al. “Electrical Flows, Laplacian Systems, and Faster Approximation of Maximum Flow in Undirected Graphs”. In: *Proceedings of the Forty-third Annual ACM Symposium on Theory of Computing*. ACM, 2011, pp. 273–282. DOI: 10.1145/1993636.1993674 (cit. on p. 1).

[Dem97] James W. Demmel. *Applied Numerical Linear Algebra*. Society for Industrial and Applied Mathematics, 1997. DOI: 10.1137/1.9781611971446 (cit. on p. 43).

[Dij59] E. W. Dijkstra. “A note on two problems in connexion with graphs”. In: *Numerische Mathematik* 1 (1959), pp. 269–271. DOI: 10.1007/BF01386390 (cit. on p. 20).

[Elk+05] Michael Elkin et al. “Lower-stretch Spanning Trees”. In: *Proceedings of the Thirty-seventh Annual ACM Symposium on Theory of Computing*. ACM, 2005, pp. 494–503. DOI: 10.1145/1060590.1060665 (cit. on pp. 2, 20, 27, 37).

[G+10] Gaël Guennebaud, Benoît Jacob, et al. *Eigen v3*. http://eigen.tuxfamily.org. 2010 (cit. on p. 29).

[Gre96] Keith Gremban. “Combinatorial Preconditioners for Sparse, Symmetric, Diagonally Dominant Linear Systems”. PhD thesis. Carnegie Mellon University, 1996. DOI: 10.1.1.368.508 (cit. on p. 8).

[HT84] Dov Harel and Robert Endre Tarjan. “Fast Algorithms for Finding Nearest Common Ancestors”. In: *SIAM J. Comput.* 2 (1984), pp. 338–355. DOI: 10.1137/0213024 (cit. on p. 22).

[Kel+13] Jonathan A. Kelner et al. “A Simple, Combinatorial Algorithm for Solving SDD Systems in Nearly-linear Time”. In: *Proceedings of the Forty-fifth Annual ACM Symposium on Theory of Computing*. 2013, pp. 911–920. DOI: 10.1145/2488608.2488724 (cit. on pp. v, vi, 2, 3, 7, 9, 11, 15–17, 19, 21, 23, 25, 29, 34–36, 57, 58).

[Kir45] Gustav Robert Kirchhoff. “Über den Durchgang eines elektrischen Stromes durch eine Ebene, insbesondere durch eine kreisförmige.” In: *Annalen der Physik und Chemie* (LXIV 1845), pp. 497–514 (cit. on p. 13).

[KLP12] Ioannis Koutis, Alex Levin and Richard Peng. “Improved spectral sparsification and numerical algorithms for SDD matrices”. In: *Symposium on Theoretical Aspects of Computer Science*. 2012, pp. 266–277. DOI: 10.1.1.352.9713 (cit. on p. 2).

[KM09] Jonathan A. Kelner and Aleksander Madry. "Faster Generation of Random Spanning Trees". In: *Proceedings of the 2009 50th Annual IEEE Symposium on Foundations of Computer Science*. IEEE Computer Society, 2009, pp. 13–21. DOI: 10.1109/FOCS.2009.75 (cit. on p. 1).

[KMP11] Ioannis Koutis, Gary L. Miller and Richard Peng. "A Nearly-$m \log n$ Time Solver for SDD Linear Systems". In: *Proceedings of the 2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*. IEEE Computer Society, 2011, pp. 590–598. DOI: 10.1109/FOCS.2011.85 (cit. on p. 20).

[KMP12] Jonathan A. Kelner, Gary L. Miller and Richard Peng. "Faster Approximate Multicommodity Flow Using Quadratically Coupled Flows". In: *Proceedings of the Forty-fourth Annual ACM Symposium on Theory of Computing*. ACM, 2012, pp. 1–18. DOI: 10.1145/2213977.2213979 (cit. on p. 1).

[Kou14] Ioannis Koutis. "Simple parallel and distributed algorithms for spectral graph sparsification". In: *Proceedings of the 26th ACM symposium on Parallelism in algorithms and architectures*. ACM. 2014, pp. 61–66. DOI: 10.1145/2612669.2612676 (cit. on pp. 3, 58).

[Kru56] Joseph B. Kruskal. "On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem". In: *Proceedings of the American Mathematical Society* (1956), pp. 48–50. DOI: 10.2307/2033241 (cit. on pp. 2, 20).

[LL12] A. Lipowski and D. Lipowska. "Roulette-wheel selection via stochastic acceptance". In: *Physica A: Statistical Mechanics and its Applications* 6 (2012), pp. 2193–2196. DOI: 10.1016/j.physa.2011.12.004 (cit. on p. 26).

[Luk14] D. Lukarski. "PARALUTION - Library for Iterative Sparse Methods". In: *GPU Technology Conference (GTC)*. 2014 (cit. on p. 29).

[Mar63] D. Marquardt. "An Algorithm for Least-Squares Estimation of Nonlinear Parameters". In: *Journal of the Society for Industrial and Applied Mathematics* 2 (1963), pp. 431–441. DOI: 10.1137/0111030 (cit. on p. 43).

[Pap14] Pál András Papp. "Low-Stretch Spanning Trees". Bachelor thesis. Eötvös Loránd University, Budapest, 2014 (cit. on pp. v, vi, 2, 19, 20, 37, 57).

[PS14] Richard Peng and Daniel A. Spielman. "An Efficient Parallel Solver for SDD Linear Systems". In: *Proceedings of the 46th Annual ACM Symposium on Theory of Computing*. ACM, 2014, pp. 333–342. DOI: 10.1145/2591796.2591832 (cit. on pp. 3, 58).

[Rei98] J.H. Reif. "Efficient approximate solution of sparse linear systems". In: *Computers & Mathematics with Applications* 9 (1998), pp. 37–58. DOI: 10.1016/S0898-1221(98)00191-6 (cit. on p. 2).

[She94] Jonathan Richard Shewchuk. *An introduction to the conjugate gradient method without the agonizing pain*. 1994. DOI: 10.1.1.110.418 (cit. on p. 2).

[Sla50] Morton Slater. "Lagrange Multipliers Revisited". In: *Traces and Emergence of Nonlinear Programming*. Springer Basel, 1950, pp. 293–306. DOI: 10.1007/978-3-0348-0439-4_14 (cit. on p. 8).

[SS08] Daniel A. Spielman and Nikhil Srivastava. "Graph Sparsification by Effective Resistances". In: *STOC*. 2008. DOI: 10.1145/1374376.1374456 (cit. on pp. 1, 2).

[SSM14] Christian L. Staudt, Aleksejs Sazonovs and Henning Meyerhenke. "NetworKit: An Interactive Tool Suite for High-Performance Network Analysis". In: *arXiv:1403.3005* (2014) (cit. on p. 29).

[ST04] Daniel A. Spielman and Shang-Hua Teng. "Nearly-linear Time Algorithms for Graph Partitioning, Graph Sparsification, and Solving Linear Systems". In: *STOC*. 2004, pp. 81–90. DOI: 10.1145/1007352.1007372 (cit. on pp. v, vi, 1, 2).

[ST83] Daniel D. Sleator and Robert Endre Tarjan. "A data structure for dynamic trees". In: *Journal of Computer and System Sciences* 3 (1983), pp. 362–391. DOI: 10.1016/0022-0000(83)90006-5 (cit. on p. 23).

[Sut05] Herb Sutter. "The free lunch is over: A fundamental turn toward concurrency in software". In: *Dr. Dobb's journal* 3 (2005), pp. 202–210 (cit. on p. 54).

[SW09] Daniel A. Spielman and Jaeoh Woo. "A Note on Preconditioning by Low-Stretch Spanning Trees". In: *CoRR* (2009). DOI: abs/0903.2816 (cit. on p. 2).

[Tur48] Alan M. Turing. "Rounding-off errors in matrix processes". In: *The Quarterly Journal of Mechanics and Applied Mathematics* 1 (1948). DOI: 10.1093/qjmam/1.1.287 (cit. on p. 1).

[Vai90] P. M. Vaidya. *Solving linear equations with symmetric diagonally dominant matrices by constructing good preconditioners*. Tech. rep. University of Illinois at Urbana-Champaign, 1990 (cit. on pp. 1, 2).

[WTM13] Vincent M. Weaver, Dan Terpstra and Shirley Moore. "Nondeterminism and overcount on modern hardware performance counter implementations". In: *ISPASS* (2013). DOI: 10.1109/ISPASS.2013.6557172 (cit. on p. 31).

[ZJH09] D. Zaparanuks, M. Jovic and M. Hauswirth. "Accuracy of performance counter measurements". In: *ISPASS* (2009). Boston, pp. 23–32. DOI: 10.1109/ISPASS.2009.4919635 (cit. on p. 31).

Appendix

A Symbols and notations

$\mathbb{F}_k$	finite field of order k
$[n]$	finite set $\{1, \dots, n\}$
$\binom{V}{n}$	set of subsets of size n of set V
$A \subseteq B$	set A is a subset of set B, possibly $A = B$
$\lvert M \rvert$	cardinality of set M
$\mathbb{R}$	set of reals
$\mathbb{R}_{>0}$	set of positive reals
$\mathbb{R}_{\geqslant 0}$	set of nonnegative reals
$\mathbb{R}^{n \times m}$	space of $n \times m$ real matrices
$[a, b)$	half-open interval $\{x \in \mathbb{R} : a \leqslant x < b\}$
$[a, b]$	closed interval $\{x \in \mathbb{R} : a \leqslant x \leqslant b\}$
X^Y	set of functions $f \colon Y \to X$
$M \times N$	set of pairs (m, n) with $m \in M$ and $n \in N$
$\lVert x \rVert_A$	$\sqrt{x^T A x}$ for matrix A and vector x
$\ker(A)$	kernel $\{x \in \mathbb{R}^n : Ax = 0\}$ of matrix $A \in \mathbb{R}^{m \times n}$
$\operatorname{im}(A)$	image $\{Ax : x \in \mathbb{R}^n\}$ of matrix $A \in \mathbb{R}^{m \times n}$
A^+	Moore-Penrose pseudoinverse of matrix A
∇f	gradient of scalar function $f \colon A \to \mathbb{R}$
$H(f)$	Hessian of scalar function $f \colon A \to \mathbb{R}$
$f = \mathcal{O}(g)$	function $f \colon \mathbb{N} \to \mathbb{R}_{\geqslant 0}$ grows asymptotically at most as fast as function $g \colon \mathbb{N} \to \mathbb{R}_{\geqslant 0}$, i. e. there is an $n_0 \in \mathbb{N}$ and a $c \in \mathbb{R}_{\geqslant 0}$ such that $f(n) \leqslant cg(n)$ for all $n \geqslant n_0$
$\alpha(n)$	inverse Ackermann function
G	a weighted, simple graph
$V(G)$	set of nodes of graph G
$E(G)$	set of edges of graph G
K_n	complete graph on n nodes
w_e	weight of edge e
$P_T(u, v)$	unique simple path in tree T from node u to node v
$N_G(u)$	set of neighbours of node u in graph G

B Acronyms

AVX	Advanced vector extensions
BFS	Breadth-first search
CG	Conjugate gradient method
FGMRES	Flexible generalized minimal residual method
FLOPS	Floating point operations
FPU	Floating point unit
IPC	Instructions per cycle
LCA	Lowest common ancestor
NUMA	Nonuniform memory access
PDE	Partial differential equation
RAM	Random access machine
RMQ	Range minimum query
SDD	Symmetric diagonally dominant
SSE	Streaming SIMD extensions
ST	Spanning tree
SpMV	Sparse matrix-vector multiplication

C Figures

D Tables

E Problems

F Algorithms