algorithms

MDPI

Article

Engineering a Combinatorial Laplacian Solver: Lessons Learned †

Daniel Hoske [1,2], Dimitar Lukarski [3], Henning Meyerhenke [1,*] and Michael Wegner [1]

1 Institute of Theoretical Informatics, Karlsruhe Institute of Technology (KIT), Am Fasanengarten 5, D-76131 Karlsruhe, Germany; daniel.hoske@gmail.com (D.H.); michael.t.wegner@gmail.com (M.W.)
2 Google Inc., 1600 Amphitheatre Parkway, Mountain View, CA 94043, USA
3 Paralution Labs UG & Co. KG, D-76571 Gaggenau, Germany; dimitar@paralution.com
* Correspondence: meyerhenke@kit.edu; Tel.: +49-721-608-41876
† This paper is an extended version of our paper published in The proceedings of the 14th International Symposium on Experimental Algorithms (SEA 2015).

Academic Editor: Qianping Gu
Received: 9 September 2016; Accepted: 20 October 2016; Published: 31 October 2016

Abstract: Linear system solving is a main workhorse in applied mathematics. Recently, theoretical computer scientists contributed sophisticated algorithms for solving linear systems with symmetric diagonally-dominant (SDD) matrices in provably nearly-linear time. These algorithms are very interesting from a theoretical perspective, but their practical performance was unclear. Here, we address this gap. We provide the first implementation of the combinatorial solver by Kelner et al. (STOC 2013), which is appealing for implementation due to its conceptual simplicity. The algorithm exploits that a Laplacian matrix (which is SDD) corresponds to a graph; solving symmetric Laplacian linear systems amounts to finding an electrical flow in this graph with the help of cycles induced by a spanning tree with the low-stretch property. The results of our experiments are ambivalent. While they confirm the predicted nearly-linear running time, the constant factors make the solver much slower for reasonable inputs than basic methods with higher asymptotic complexity. We were also not able to use the solver effectively as a smoother or preconditioner. Moreover, while spanning trees with lower stretch indeed reduce the solver's running time, we experience again a discrepancy in practice: in our experiments, simple spanning tree algorithms perform better than those with a guaranteed low stretch. We expect that our results provide insights for future improvements of combinatorial linear solvers.

Keywords: Laplacian linear systems; graph algorithms; low-stretch spanning trees; electrical graph flows; algorithm engineering

1. Introduction

Solving square linear systems $Ax = b$, where $A \in \mathbb{R}^{n \times n}$ and $x, b \in \mathbb{R}^n$, is one of the most important problems in applied mathematics. It has widespread applications in science and engineering since it is at the heart of numerical simulations, which are in widespread use in academia and in industry, e.g., in tools like OpenFOAM [1]. In practice, system matrices are often sparse, i.e., they contain $O(n)$ nonzeros. Ideally, the required time for solving sparse systems would grow linearly with the number of nonzeros $2m$. Most direct solvers, however, show cubic running times and do not exploit sparsity. Furthermore, approximate solutions usually suffice due to the imprecision of floating point arithmetic. Exploiting this fact with sparse iterative solvers, such as conjugate gradient (CG), still yields a running time that is clearly superlinear in n.

Spielman and Teng [2], following an approach proposed by Vaidya [3], achieved a major breakthrough in this direction by devising a nearly-linear time algorithm for solving linear systems

in symmetric diagonally-dominant matrices. Nearly-linear means $\mathcal{O}\big(m \cdot \mathrm{polylog}(n) \cdot \log(1/\epsilon)\big)$ here, where $\mathrm{polylog}(n)$ is the set of real polynomials in $\log(n)$ and ϵ is the relative error $\|x - x_{\mathrm{opt}}\|_A / \|x_{\mathrm{opt}}\|_A$ 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 $x_{\mathrm{opt}} := A^+ b$ is an exact solution (where A^+ refers to the pseudoinverse of A). A matrix $A = (a_{ij})_{i,j \in [n]} \in \mathbb{R}^{n \times n}$ is diagonally dominant if $|a_{ii}| \geq \sum_{j \neq i} |a_{ij}|$ for all $i \in [n]$.

Symmetric matrices that are diagonally dominant (SDD matrices) have many applications, not only in applied mathematics, such as elliptic PDEs [4] and numerical simulations. They also play a major role in graph-based problems from (theoretical) computer science, such as, e.g., maximum flows [5], graph drawing [6] and graph sparsification [7]; also see Kelner and Madry [8]. Thus, the problem INV-SDD of solving linear systems $Ax = b$ for x on SDD matrices A is of significant importance. We focus here on Laplacian matrices (which are SDD) due to their rich applications in algorithms for undirected graphs, e.g., load balancing [9,10] and image segmentation [11], but this is no major limitation [12].

1.1. Related Work

Spielman and Teng's seminal paper [2] requires much sophisticated machinery: a multilevel approach [3,13] using recursive preconditioning, preconditioners based on low-stretch spanning trees [14] and spectral graph sparsifiers [7,15]. Later papers extended and improved upon this approach, both by making it simpler and by reducing the exponents of the polylogarithmic time factors, e.g., [16]. Spielman provides a comprehensive overview of later work online [17]. We focus on an algorithm by Kelner et al. [12] that reinterprets the problem of solving an SDD linear system as finding an electrical flow in a graph. It only needs low-stretch spanning trees and achieves in its best variant $\mathcal{O}\big(m \log^2 n \log\log n \log(1/\epsilon)\big)$ time. Another interesting nearly-linear time SDD solver is the recursive sparsification approach by Peng and Spielman [18]. With a parallel sparsification algorithm (e.g., [19]), it yields nearly-linear work in parallel.

Spielman and Teng's algorithm crucially uses the low-stretch spanning trees first introduced by Alon et al. [20] (for a definition of stretch, see Section 2). Elkin et al. [21] 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. [22,23] later showed how to get rid of some of the logarithmic factors in both stretch and time. Papp [24] 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 and that the original algorithm of Elkin et al. [21] achieves the best results among the provably good approaches. We use these low-stretch spanning trees in our implementation of the algorithm of Kelner et al. [12] and compare their effectiveness for the solver.

It should also be noted that there are a few methods available for the problem with fast empirical running times; but with no equivalent guarantee on the theoretical worst-case running time: combinatorial multigrid (CMG) [25] and lean algebraic multigrid (LAMG) [26] are, as the names suggest, multigrid schemes; the work by Dell'Acqua et al. [27,28] has evolved into a multi-iterative scheme combining multigrid and graph-based preconditioners (among others using spanning trees).

1.2. Motivation, Outline and Contribution

Although several extensions and simplifications to the Spielman–Teng nearly-linear time solver [2] have been proposed, there is a lack of results for how they all perform in practice. We seek to fill this gap by implementing and evaluating an algorithm proposed by Kelner et al. [12] that is fascinating due to its simple description and easier to implement (and thus, more promising in practice) than the original Spielman–Teng algorithm. Recently, Boman et al. [29] have presented an experimental study of the Kelner et al. algorithm. Their results partially extend ours and provide additional insights. However, their focus differs from ours and does not consider running times with natively-compiled code.

In this paper, which extends the previous conference version [30], we implement the KOSZ solver (the acronym follows from the authors' last names) by Kelner et al. [12] and investigate its practical performance. To this end, we start in Section 2 by describing important notation and outlining KOSZ. In Section 3, we elaborate on the design choices one can make when implementing KOSZ. In particular, we explain when these choices result in a provably nearly-linear time algorithm. Section 4 contains the experimental evaluation of the Laplacian solver KOSZ. We consider the configuration options of the algorithm, its asymptotics, its convergence and its use as a preconditioner or as a smoother. As opposed to Boman et al. [29], we focus on real execution performance with natively-compiled code. Moreover, we also explore implementation choices not pursued in [29]. Our results confirm a nearly-linear running time, but are otherwise not very promising from a practical point of view: the asymptotics hide very high constant factors, in part due to memory accesses. We conclude the paper in Section 5 by summarizing the results and discussing future research directions.

2. Preliminaries

2.1. Fundamentals

We consider undirected simple weighted graphs $G = (V, E)$ with n vertices and m edges. A graph is weighted if we have an additional function $w \colon E \to \mathbb{R}_{>0}$. Where necessary, we consider unweighted graphs to be weighted with $w_e = 1 \ \forall e \in E$. We usually write an edge $\{u, v\} \in E$ as uv and its weight as w_{uv}. Moreover, we define the set operations $\cup$, $\cap$ and $\setminus$ on graphs by applying them to the set of vertices and the set of edges separately. For every node $u \in V$, its neighborhood $N_G(u)$ 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 $d_u = \sum_{v \in N_G(u)} w_{uv}$. The Laplacian matrix of a graph $G = (V, E)$ is defined as: $L_{u,v} := -w_{uv}$ if $uv \in E, \sum_{x \in N_G(u)} w_{ux}$ if $u = v$, and 0 otherwise for $u, v \in V$. A Laplacian matrix is always an SDD matrix. Another useful property of the Laplacian is the factorization $L = B^T R^{-1} B$, where $B \in \mathbb{R}^{E \times V}$ is the incidence matrix and $R \in \mathbb{R}^{E \times E}$ is the resistance matrix defined by $B_{ab,c} = 1$ if $a = c, = -1$ if $b = c$ and zero otherwise; $R_{e_1,e_2} = 1/w_{e_1}$ if $e_1 = e_2$ and zero otherwise. This holds for all $e_1, e_2 \in E$ and $a, b, c \in V$, where we arbitrarily fix a start and end node for each edge when defining B. With $x^T L x = (Bx)^T R^{-1} (Bx) = \sum_{e \in E} (Bx)_e^2 \cdot w_e \geq 0$ (every summand is non-negative), one can see that L is positive semidefinite. (A matrix $A \in \mathbb{R}^{n \times n}$ is positive semidefinite if $x^T A x \geq 0$ for all $x \in \mathbb{R}^n$.)

Some conventions used throughout the paper: Every function that is parametrized 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 $\|Ay - b\|_2 / \|b\|_2$.

2.2. Cycles, Spanning Trees and 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 work, 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 occurs in exactly one of the two cycles, i.e., $C_1 \oplus C_2 := (C_1 \setminus C_2) \cup (C_2 \setminus C_1)$. In algebraic terms, we can regard a cycle as a vector $C \subseteq \mathbb{F}_2^E$ ($\mathbb{F}_2$ is the finite field of order two), 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 $\mathcal{C}(G)$.

In a spanning tree (ST) $T = (V, E_T)$ of G, there is a unique path $P_T(u, v)$ from every node u to every node v. 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 slightly differently, but we follow the definition in Kelner et al. [12] using w_e. The total stretch of the whole tree T is the sum of the individual stretches $\mathrm{st}(T) = \sum_{e \in E} \mathrm{st}(e)$. Finally, we define the average stretch as the total stretch divided by the total edge weight. Finding a spanning tree with low stretch is crucial for proving the fast convergence of the KOSZ solver.

2.3. Electrical Network Analogy

We can regard G as an electrical network where each edge uv corresponds to a resistor with conductance w_{uv} and resistance $1/w_{uv}$, and x as an assignment of potentials to the nodes of G (cf. Figure 1). L operates on every vector $x \in \mathbb{R}^n$ via $(Lx)_u = \sum_{v \in N(u)} (x_u - x_v) \cdot w_{uv}$ for each $u \in V$. Then, $x_u - x_v$ is the voltage across uv, and $(x_u - x_v) \cdot w_{uv}$ 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. Furthermore, one can reduce solving SDD systems to the related problem INV-LAPLACIAN-CURRENT [12]: Given a Laplacian $L = L(G)$ and a vector $b \in \mathrm{im}(L)$, compute a function $f\colon \widetilde{E} \to \mathbb{R}$ with (i) f being a valid graph flow on G with demand b and (ii) the potential drop along every cycle in G being zero, where a valid graph flow means that the sum of the incoming and outgoing flow at each vertex respects the demand in b and that $f(u,v) = -f(v,u)\ \forall uv \in E$. Furthermore, $\widetilde{E}$ is a bi-directed copy of E, and the potential drop of cycle C is $\sum_{e \in C} f(e) r_e$.

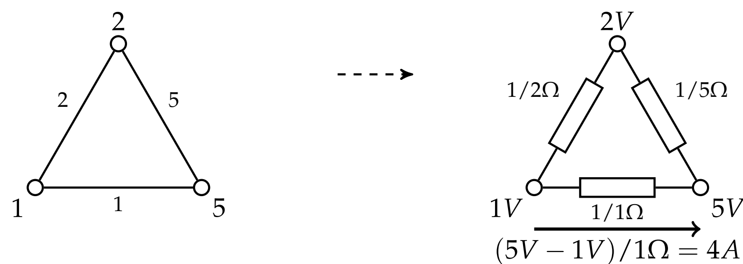


Figure 1. Transformation into an electrical network.

2.4. KOSZ (Simple) Solver

The idea of the algorithm is to start with any valid flow and successively adjust the flow, such that every cycle has potential zero. We need to transform the flow back to potentials at the end, but this can be done consistently, as all potential drops along cycles are zero.

Regarding the crucial question of what flow to start with and how to choose the cycle to be repaired in each iteration, Kelner et al. [12] 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 specifically, they suggest starting with a flow that is nonzero only on T and weighting the basis cycles proportionately to their stretch when sampling them. The resulting algorithm is shown as Algorithm 1; note that we may stop before all potential drops along cycles are zero, and we can consistently compute the potentials induced by f at the end by only looking at T.

The solver described in Algorithm 1 is actually just the SimpleSolver in the Kelner et al. [12] paper. They also show how to improve this solver by adapting preconditioning to the setting of electrical flows. In informal experiments, we could not determine a strategy that is consistently better than the SimpleSolver, so we do not pursue this scheme any further here. Eventually, Kelner et al. derive the following running time for the KOSZ (simple) solver:

Theorem 1. *(Theorem 3.2 in [12]) SimpleSolver can be implemented to run in time $O(m \log^2 n \log \log n \log(\epsilon^{-1} n))$ for computing an ϵ-approximation of x.*

The improved running time of their FullSolver to compute an ϵ-approximation of x is $O(m \log^2 n \log \log n \log(\epsilon^{-1}))$ (Theorem 7.4 in [12]).

Algorithm 1: INV-LAPLACIAN-CURRENT solver KOSZ.

Input: Laplacian $L = L(G)$ and vector $b \in \mathrm{im}(L)$.
Output: Solution x to $Lx = b$.
1 $T \leftarrow$ a spanning tree of G
2 $f \leftarrow$ unique flow with demand b that is only nonzero on T
3 **while** *there is a cycle with potential drop* $\neq 0$ *in* f **do**
4   $c \leftarrow$ cycle in $\mathcal{C}(T)$ chosen randomly weighted by stretch
5   $f \leftarrow f - \frac{c^T R f}{c^T R c} c$
6 **return** *vector of potentials in f with respect to the root of T*

3. Implementation

While Algorithm 1 provides the basic idea of the KOSZ solver, it leaves open several implementation decisions we had to make and that we elaborate on in this section.

3.1. Spanning Trees

As suggested by the convergence result in Theorem 1, the KOSZ solver crucially depends on low-stretch spanning trees. The notion of stretch was introduced by Alon et al. [20] along with an algorithm to compute a spanning tree with low stretch. Unfortunately, the stretch guaranteed by their algorithm is super-polynomial. Elkin et al. [21] presented an 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. We use Dijkstra with binary heaps for growing the balls and take care not to need more work than necessary to grow the ball. In particular, ball growing is output-sensitive, and growing a ball $B(x,r) := \{v \in V : \text{distance from } x \text{ to } v \text{ is} \leq r\}$ 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, but [24] showed experimentally that these improvements do not yield better stretch in practice. In fact, his experiments suggest that the stretch of the provably good 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 (the tree built implicitly during the search) using binary heaps; as well as a minimum-weight spanning tree with Kruskal's algorithm using union-find with union-by-size and path compression.

To test how dependent the algorithm is on the stretch of the spanning tree (ST), we also look at a special ST for $n_1 \times n_2$ grids. As depicted in Figure 2, we construct this spanning tree by subdividing the $n_1 \times n_2$ grid into four subgrids as evenly as possible (the subgrid sizes are shown in Figure 2a), recursively building the STs in the subgrids (the termination of the recursion is shown in Figure 2b) and connecting the subgrids by a U-shape in the middle.

Proposition 1. *Let G be an $n_1 \times n_2$ grid with $n_1, n_2 \geq 4$. Then, the special ST has $\mathcal{O}\left(\frac{(n_1+n_2)^2 \log(n_1+n_2)}{n_1 n_2}\right)$ average stretch on G.*

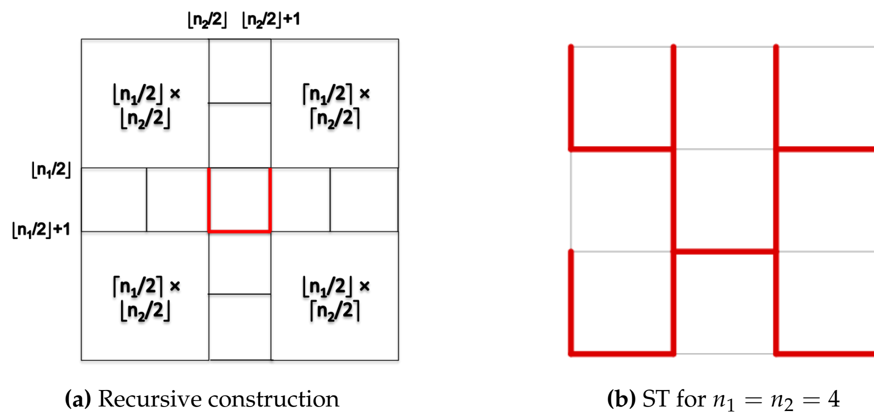
Proof. First note that, by the recursive construction, the total stretch of the four subgrids remains the same if such a subgrid is treated separately. Moreover, the stretches of the $\mathcal{O}(n_1 + n_2)$ off-tree edges between the rows $\lfloor n_1/2 \rfloor$ and $\lfloor n_1/2 \rfloor + 1$, as well as the columns $\lfloor n_2/2 \rfloor$ and $\lfloor n_2/2 \rfloor + 1$ are

in $\mathcal{O}(n_1 + n_2)$ each. To see this, let s and t be the source and target vertices of such an off-tree edge, respectively. Then, by construction, it is possible to reach the center from s in $\mathcal{O}(n_1 + n_2)$ steps and t from the center likewise. Consequently, $S(n_1, n_2) = 4 \cdot S(n_1/2, n_2/2) + \mathcal{O}(n_1 + n_2)^2$ when disregarding rounding. After solving, this recurrence (note that $S(n_1/2, n_2/2)$ is essentially one fourth in size compared to $S(n_1, n_2)$ as long as $n_1, n_2 \geq 4$); we get:

$$S(n_1, n_2) = \mathcal{O}\big((n_1 + n_2)^2 \log(n_1 + n_2)\big).$$

Since the number of edges is $\Theta(n_1 n_2)$, the claim for the average stretch follows. □

In case of a square grid ($n_1 = n_2$) with $N = n_1 \times n_2$ vertices, we get $S(N) = 4S(N/4) + \mathcal{O}(N) = \mathcal{O}(N \log N) = \mathcal{O}(n_1^2 \log(n_1))$ and, thus, $\mathcal{O}(\log n_1)$ average stretch. A logarithmic average stretch (and thus, detour) is noteworthy since the average distance between a random pair of nodes in the square grid is $\Omega(n_1)$. Furthermore, for this special case, our result slightly improves on the general low-stretch spanning tree algorithms. Later on in this paper, we will use it in comparison to other spanning trees to assess their effect on the KOSZ solver.



(a) Recursive construction **(b)** ST for $n_1 = n_2 = 4$

Figure 2. Special spanning tree (ST) with $\mathcal{O}\left(\frac{(n_1+n_2)^2 \log(n_1+n_2)}{n_1 n_2}\right)$ average stretch for the $n_1 \times n_2$ grid.

3.2. Flows on Trees

Since every basis cycle contains exactly one off-tree-edge, the flows on off-tree-edges can simply be stored in a single vector. 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, the core problem is to efficiently store and update flows in T. We want to support the following 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 $\sum_{e \in P_T(u,r)} f(e) r_e$ and
- update(u, α): set $f(e) := f(e) + \alpha$ for all $e \in P_T(u, r)$. (2)

The itemized two-node operations can then be supported with query(u, v) := query(u) − query(v) and update$(u, v, \alpha) := \{$update(u, α) and update$(v, -\alpha)\}$, since the changes on the subpath $P_T(r, \mathrm{LCA}(u, v))$ cancel out. Here, $\mathrm{LCA}(u, v)$ 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; they are described next in some detail.

3.3. Linear Time Updates

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. We can improve this implementation by only traversing up to $\text{LCA}(u, v)$ 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 lowest common ancestor queries for pairs of nodes after some precomputation are a classical topic; optimal solutions are known [31,32]. In our implementation, we use a simpler implementation with higher (but still insignificant) preprocessing time that transforms an LCA query into a range minimum query (RMQ), the problem of determining the minimum in a subrange of an array. We can then solve the RMQ problem by precomputing the RMQ of every range whose length is a power of two, i.e., for each i with $2^i \leq n$ and every $j \in [n]$, we compute $\texttt{prec}[i, j] := \arg\min v[j \dots j + 2^i - 1]$. This can be done in $\mathcal{O}(n \log n)$ time.

3.4. Logarithmic Time Updates

While the data structure presented above allows fast repairs for short basis cycles, the worst-case time is still in $\mathcal{O}(n)$. We therefore also implement the data structure by Kelner et al. [12] with $\mathcal{O}(\log n)$ worst-case time repairs. It is based on link-cut trees [33]. 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 $\leq n/2$ nodes. Equivalently, we find a vertex in T, all of whose induced subtrees have size $\leq n/2$. We call such a vertex a good vertex separator. 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.

Remark 1. *We can implement query and update efficiently by storing several values: (i) d_{drop}: the total potential drop on the path $P_T(r, d)$, (ii) d_{ext}: the total flow contribution to $P_T(r, d)$ from vertices below d, and (iii) for every $u \in V(T)$: $\text{height}(u) := \sum_{e \in P_T(r,u) \cap P_T(r,d)} r_e$, i.e., the accumulated resistance in common between the $P_T(r, d)$ path and the $P_T(r, u)$ 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, however, $u \in T_i$ 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 to 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 $u \in T_i$. While we could directly implement this recursion, we unroll it 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 one). Kelner et al. [12] provide more details about the unrolling.

3.5. Results

In our preliminary experiments, in order to evaluate the flow data structures, the cost of querying the LCA-based data structure (LCAFlow) strongly depends on the structure of the used spanning tree, while the logarithmic-time data structure (LogFlow) induces costs that 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 LogFlow in later results.

3.6. Remarks on Initial Solution and Cycle Selection

Given x, we can compute a flow f via $f_{uv} := (x(u) - x(v)) \cdot w_{uv}$. The potential drop of each cycle in this flow f is zero. Unfortunately, this flow is not a valid graph flow with demand b, unless x already fulfills $Lx = b$. In contrast, in the solver, we iteratively establish the zero-cycle-sum property from the flow originally induced by the spanning tree T. There is an important consequence: we cannot start from an arbitrary vector x, which may make it harder to use the solver in a larger context.

The easiest way to select a cycle, in turn, 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, i.e., edges chosen with a probability proportionate to their stretch. We can use the flow data structure described above 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 initialize 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 initialization. Roulette wheel selection is a simple strategy to sample an arbitrary discrete distribution with finite support: (i) let X be a random variable with $\text{Prob}[X = x_i] = p_i$ for $i \in [k]$; (ii) precompute the prefix sums $P = (0, p_1, p_1 + p_2, \ldots, p_1 + \cdots + p_k = 1)$; (iii) to sample, choose a uniform random value $x \in [0, 1)$, then find the index i with $P_i \leq x < P_{i+1}$ 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.

3.7. Summary

For convenience, we summarize the implementation choices for Algorithm 1 in Table 1. 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. The convergence theorem requires a low-stretch spanning tree and weighted cycle selection. Note that $m = \Omega(n)$, as G is connected.

Table 1. Summary of the components of the KOSZ solver.

Component	Cost
Spanning tree	$\mathcal{O}(m \log n \log\log n))$ 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. [21]	$\mathcal{O}(m \log^2 n \log\log n)$ stretch, $\mathcal{O}(m \log^2 n)$ time
Abraham and Neiman [23]	$\mathcal{O}(m \log n \log\log n)$ stretch, $\mathcal{O}(m \log n \log\log n)$ time
Initialize cycle selection	$\mathcal{O}(m \log n)$ time
Uniform	$\mathcal{O}(m)$ time
Weighted	$\mathcal{O}(m \log n)$ time
Initialize flow	$\mathcal{O}(n \log n)$ time
LCA flow	$\mathcal{O}(n)$ time
Log flow	$\mathcal{O}(n \log n)$ time
Iterations	$\mathcal{O}(m \log n \log\log n \log(\epsilon^{-1} \log n))$ 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	$\mathcal{O}(m \log^2 n \log\log n \log(\epsilon^{-1} \log n))$ expected time
Improved solver	$\mathcal{O}(m \log^2 n \log\log n \log(\epsilon^{-1}))$ expected time

4. Evaluation

4.1. Settings

4.1.1. Software, Hardware and Data

We implemented the KOSZ solver in C++ using NetworKit [34], a tool suite focused on large-scale network analysis. Our code is publicly available [35]. As compiler we use g++ 4.8.3. The benchmark platform is a dual-socket server with two eight-core Intel Xeon E5-2680 at 2.7 GHz each and 256 GB RAM. We present a representative subset of our experiments, in which we compare our KOSZ implementation to existing linear solvers as implemented by the libraries Eigen 3.2.2 [36] and Paralution 0.7.0 [37], both libraries with fast sparse matrix solvers.

We mainly use two graph classes for our tests: (i) rectangular $k \times l$ grids given by $\mathbb{G}_{k,l} := \big([k] \times [l], \{\{(x_1, y_1), (x_2, y_2)\} \subseteq \binom{V}{2} : |x_1 - x_2| = 1 \vee |y_1 - y_2| = 1\}\big)$; 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; (ii) Barabási–Albert random graphs with parameter k [38]. These random graphs are parametrized 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 $\mathrm{Barabasi}(n, k)$. Their construction models that the degree distribution in many natural graphs is not uniform at all.

For both classes of graphs, we consider both unweighted and weighted variants (uniform random weights in $[1, 8]$). We also did informal tests on 3D grids and also real-world graphs, in particular complex networks. These graphs did not exhibit significantly different behavior than the two graph classes above and are therefore omitted from the presentation of the results.

4.1.2. Termination and Performance Counters

In the description of the solver, so far, we did not state our termination condition; Kelner et al. [12] 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 KOSZ 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.

CPU performance characteristics, such as the number of executed FLOP (floating point operations), etc., are measured with the PAPI library [39]. Each of our benchmarking runs takes several seconds (billions of cycles), so we expect the counter values to be quite accurate. Moreover, our most basic choice to reduce the impact of possible measurement 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 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. Moreover, the measured standard deviations are below 5%. We take an optimistic approach with regards to cache usage and start each series of runs with a dry run that fills the caches.

4.2. Results

4.2.1. Spanning Tree

Papp [24] 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 confirm

and extend this observation by comparing our own implementation of the low-stretch ST algorithm by Elkin et al. [21] to Kruskal and Dijkstra in Figure 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 a factor of 20 worse). Interestingly, Kruskal outperforms the other algorithms even on the unweighted Barabási–Albert graphs, where it degenerates to choosing an arbitrary ST. Figure 3 also shows that our special ST (see Section 3.1) yields significantly lower stretch for the unweighted 2D grid, but it does not help in the weighted case.

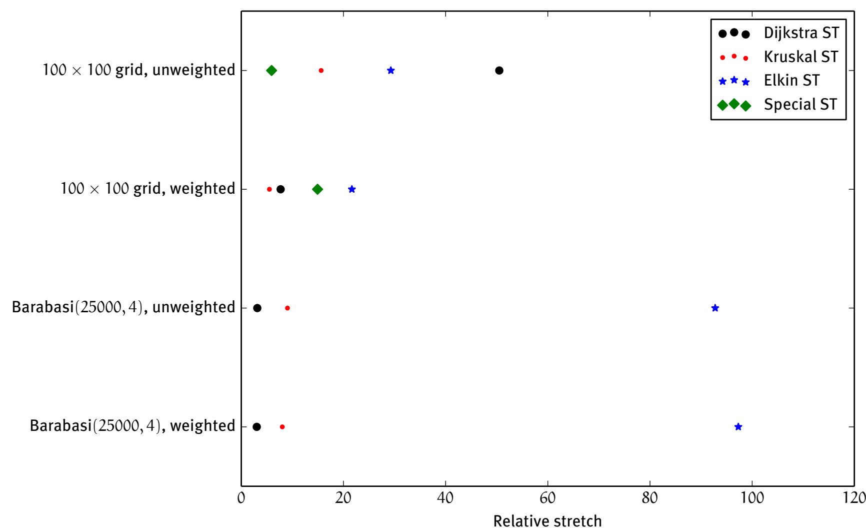
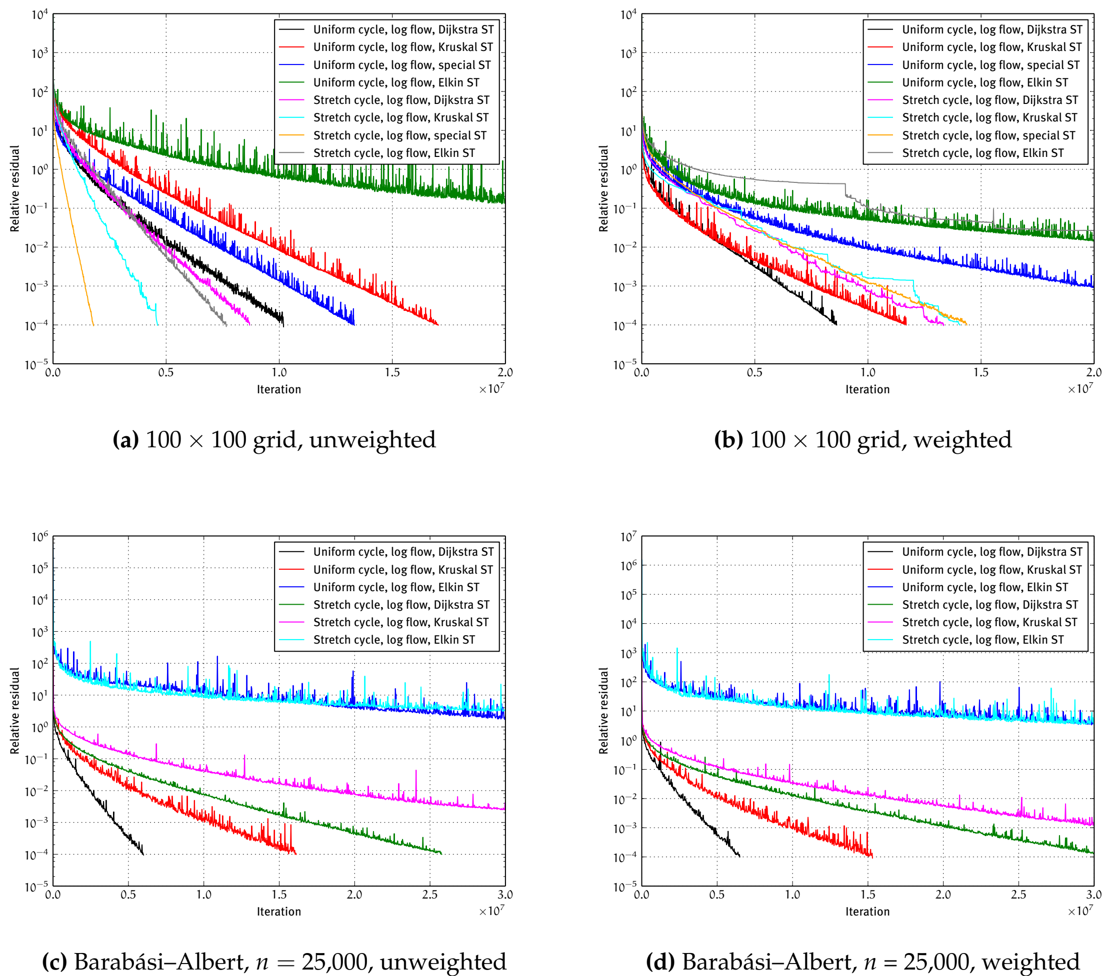


Figure 3. Average stretch $\mathrm{st}(T)/m$ with different ST algorithms.

4.2.2. Convergence

In Figure 4, we plot the convergence of the residual for different graphs and different algorithm settings. We examine a 100 × 100 grid and a Barabási–Albert graph with 25,000 nodes. While the residuals can increase, they follow a global downward trend. Furthermore, 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 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 3), except for the Dijkstra ST and the Kruskal ST on the weighted grid. In particular, on Barabási–Albert graphs, there is a significant gap between the Elkin ST and the other algorithm settings. On these graphs, the Elkin ST barely converges at all. Thus, low-stretch STs are crucial for convergence. In informal experiments we also saw this behavior for 3D grids and non-synthetic graphs. 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. Fixing the shorter basis cycles of the special ST seems to help the algorithm converge quicker. Still, the other curves with uniform cycle selection have the same order as stretch cycle selection.



(a) 100 × 100 grid, unweighted

(b) 100 × 100 grid, weighted

(c) Barabási–Albert, n = 25,000, unweighted

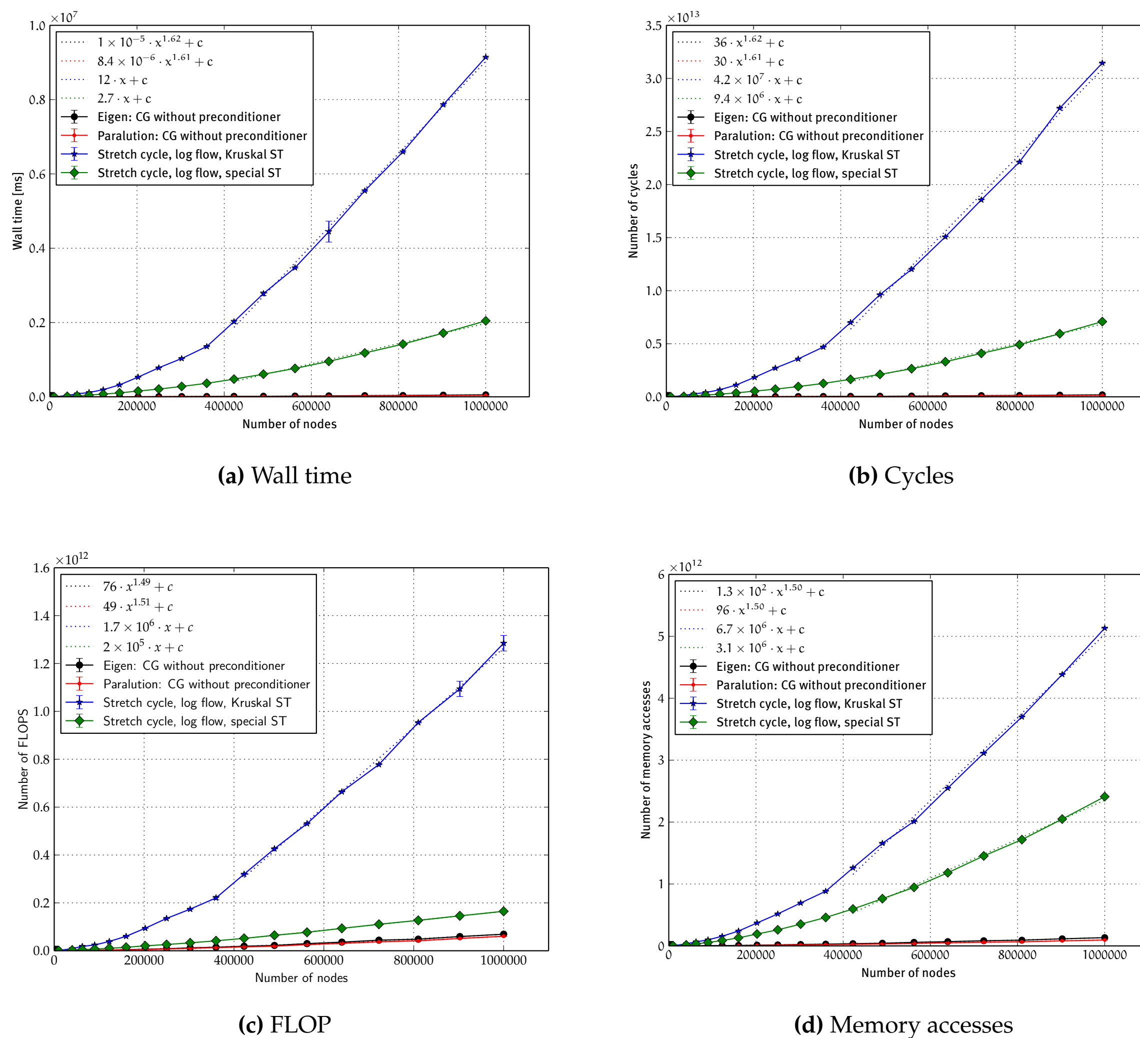
(d) Barabási–Albert, n = 25,000, weighted

Figure 4. Convergence of the residual. Terminate when the residual is $\leq 10^{-4}$.

Using the results of all our experiments, we are not able to detect any correlation between the improvement made by a cycle repair 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 favor 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 a residual of one. That is, there is no significant change of behavior 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 six million iterations ($\approx$3000 sparse matrix-vector multiplications (SpMVs) in time comparison) on a Barabási–Albert graph with 25,000 nodes and 100,000 edges in order to reach a residual of 10^{-4}. In contrast, conjugate gradient (CG) without preconditioning only needs 204 SpMVs for this graph (preconditioning is explained in the corresponding subsection below).

4.2.3. Asymptotics

Now that we know which settings of the algorithm yield the best performance for 2D grids and Barabási–Albert graphs, we proceed by looking at how the performance with these settings behaves asymptotically and how it compares to conjugate gradient (CG) without preconditioning, a simple and popular iterative solver (often used in its preconditioned form). Since KOSZ turns out to be not competitive, we do not need to compare it to more sophisticated algorithms.



(a) Wall time

(b) Cycles

(c) FLOP

(d) Memory accesses

Figure 5. Asymptotic behavior for 2*D* grids. Termination when the relative residual was $\leq 10^{-4}$. The error bars give the standard deviation.

In Figure 5, 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 [40], while our KOSZ implementation should scale nearly linearly. This expectation is confirmed in the plot: Using Levenberg–Marquardt [41] to approximate the curves for CG with a function of the form $ax^b + c$, we get $b \approx 1.5$ for FLOP and memory accesses, while the (more machine-dependent) wall time and cycle count yield a slightly higher exponent $b \approx 1.6$. We also see that the curves for our KOSZ implementation 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. Note that the difference between the algorithms in FLOP 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 many FLOP (factor of 10), while the savings in memory accesses (factor of two) are much smaller. Even though the FLOP when using the special ST are within a factor of two of CG, we still have a wide chasm in running time. However, note that later in this section, we see that the micro-performance (in terms of caches) of the solver is actually very

competitive with CG. Thus, the bad running time is mainly caused by memory accesses and the very slow convergence that we have already seen before.

The results for the Barabási–Albert graphs are basically the same (and hence, not shown in detail): Even though the growth is approximately linear from about 400,000 nodes, there is still a large gap between KOSZ and CG since the constant factor is enormous. Furthermore, the results for the number of FLOP are again much better than the results for the other performance counters.

In conclusion, although we have nearly-linear growth, even for 1,000,000 graph nodes, the KOSZ algorithm is still not competitive with CG because of huge constant factors, in particular a large number of iterations and memory accesses.

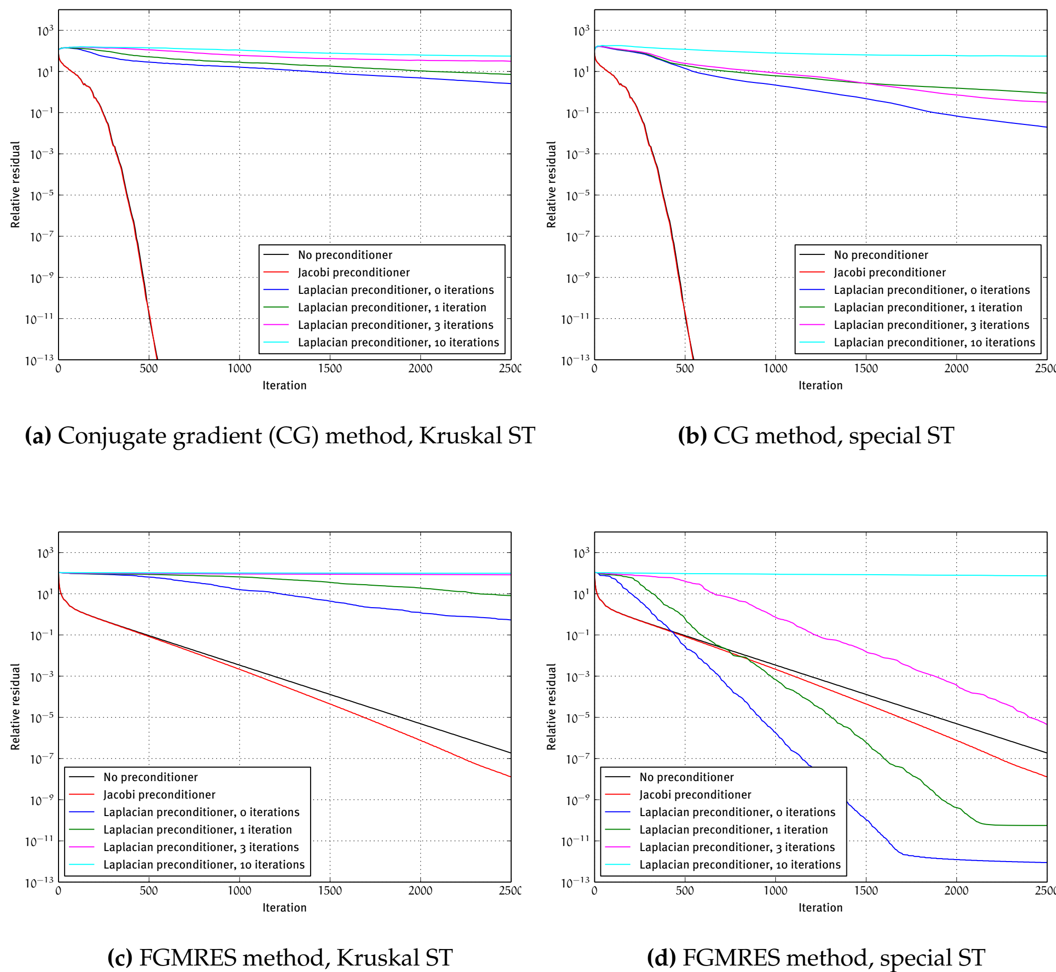
4.2.4. Preconditioning

The convergence of most iterative linear solvers on a linear system $Ax = b$ depends on the condition number $\kappa(A)$ 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 $Ax = b$ (preconditioning). Some linear solvers, such as Gauss–Seidel, are good preconditioners even though they are slow when used on their own. Thus, we check whether this is also true for KOSZ.

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 KOSZ iterations 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 the CG method and the FGMRES method (see Section 9.4 in [42]) on an unweighted 100×100 grid. The convergence of the residual with these solvers is plotted in Figure 6.

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 the preconditioner means that the preconditioner matrix is not fixed, but changes from iteration to iteration. Axelsson and Vassilevski [43] show why this behavior 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 6 when using zero iterations of the solver in FGMRES, a behavior that is obviously not generalizable. For more iterations of the Laplacian solver, FGMRES still has convergence problems, but it is somewhat better than CG.

We conclude that KOSZ is not suitable as a preconditioner for common iterative methods. It would be an interesting extension to check if the solver works in a specialized variable-step method.



(a) Conjugate gradient (CG) method, Kruskal ST

(b) CG method, special ST

(c) FGMRES method, Kruskal ST

(d) FGMRES method, special ST

Figure 6. Convergence of the residual when using the Laplacian solver as a preconditioner on an unweighted 100×100 grid.

4.2.5. Smoothing

One way of combining the good qualities of two different solvers is smoothing. Smoothing means to dampen the high-frequency components of the error, which is usually done in combination with another solver that dampens the low-frequency error components. It is known that in CG and most other solvers, the low-frequency components of the error converge very quickly, 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 reducing the error, it just needs to make its frequency distribution more favorable. Smoothers are particularly often applied at each level of multigrid or multilevel schemes [44] that turn a good smoother into a good solver by applying it at different levels of a matrix 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, 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 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 7 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 may indeed be 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 7, 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 10,000 iterations ($\approx$150 SpMVs in running time comparison) to get an error of x_1 similar to x_0, even though the frequency distribution is favorable. Note that the number of SpMVs to which the 10,000 iterations correspond depends on the graph size, e.g., for an 100×100 grid, the 10,000 iterations correspond to 20 SpMVs.

While testing the Laplacian solver in a multigrid scheme could be worthwhile, the bad initial vector creates robustness 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 behavior unless our solver already yields an almost perfect vector in a single run.

4.2.6. Cache Behavior

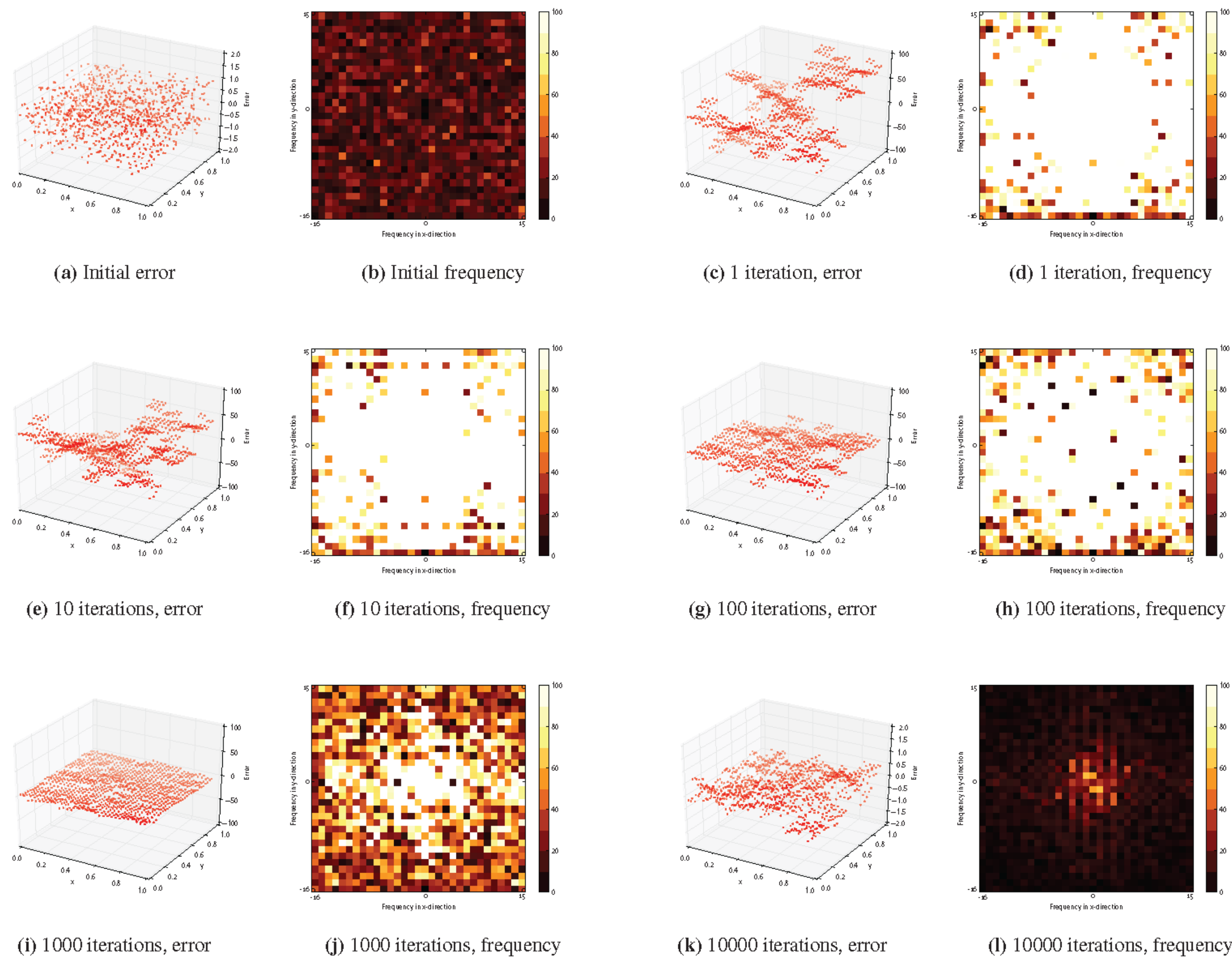
The nearly-linear running time of the Laplacian solver was proven in the RAM machine model. To get good practical performance on modern out-of-order superscalar computers, one also has to take their complex execution behavior into account, e.g., the cache hierarchy.

One particular problem indicated by our experiments is that the number of cache misses increases in the LogFlow data structure when a bad spanning tree is used. 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 sequences of pairs of indices (into the dense vector) and values. Thus, the cache behavior depends on the distribution of the indices, which is determined by the subtree decomposition of the spanning tree and the order of the subtrees.

We managed to consistently improve the time by about 6% by doing the decomposition in breadth-first search order, so that the indices 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 vectorized implementation still uses (vectorized) indirect accesses.

In our experiments, 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 behavior. Not surprisingly, 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 grouping most indices at the start of the vector.

From the benchmarks, we can infer that the micro-performance in terms of cache misses 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.



(a) Initial error **(b)** Initial frequency **(c)** 1 iteration, error **(d)** 1 iteration, frequency

(e) 10 iterations, error **(f)** 10 iterations, frequency **(g)** 100 iterations, error **(h)** 100 iterations, frequency

(i) 1000 iterations, error **(j)** 1000 iterations, frequency **(k)** 10000 iterations, error **(l)** 10000 iterations, frequency

Figure 7. 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.

5. Conclusions

At the time of writing the conference version of this paper, we provided the first comprehensive experimental study of a Laplacian solver with provably nearly-linear running time. In the meantime, our results regarding KOSZ have been recently confirmed and in some aspects extended [29], albeit with a focus on more theoretical performance measures and possibilities for parallelization, not on actual performance with natively-compiled code.

Our study supports the theoretical result that the convergence of KOSZ crucially depends on the stretch of the chosen spanning tree, with low stretch generally resulting in faster convergence. This particularly suggests that it is crucial to build algorithms that yield spanning trees with lower stretch. Since we have confirmed and extended Papp's observation that algorithms with provably low stretch do not yield good stretch in practice [24], improving the low-stretch ST algorithms is an important future research direction.

Even though KOSZ proves to grow nearly linearly as predicted by theory, the constant seems to be too large to make it competitive without major changes in the algorithm, even compared to the CG method without a preconditioner. Hence, we can say that the running time is nearly linear indeed and, thus, fast in the $\mathcal{O}$-notation, but the constant factors prevent usefulness in practice so far. While the negative results may predominate in this study, we expect to deliver insights that lead to further improvements, both in theory and practice. It seems promising to repair cycles other than just the basis cycles in each iteration; as also suggested and performed by Boman et al. [29]. Yet, their results indicate that this alone may not be sufficient. Besides, it would necessitate significantly different data structures to obtain the best possible performance in practice.

Acknowledgments: This work was partially supported by the Ministry of Science, Research and the Arts Baden-Württemberg under the grant "Parallel Analysis of Dynamic Networks—Algorithm Engineering of Efficient Combinatorial and Numerical Methods" and by DFG Grant ME 3619/3-1.

Author Contributions: D.H., D.L. and H.M. designed the research project. D.H. performed the implementation and the experiments, partially assisted by M.W. All authors contributed to the design and evaluation of the experiments. D.H. and H.M. wrote the paper with the assistance of D.L. and M.W.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Jasak, H. OpenFOAM: Open source CFD in research and industry. *Int. J. Nav. Archit. Ocean Eng.* **2009**, *1*, 89–94.
2. Spielman, D.A.; Teng, S. Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems. In Proceedings of the 36th Annual ACM Symposium on Theory of Computing (STOC), Chicago, IL, USA, 13–16 June 2004; pp. 81–90.
3. Vaidya, P.M. *Solving Linear Equations with Symmetric Diagonally Dominant Matrices by Constructing Good Preconditioners*; Technical Report; University of Illinois at Urbana-Champaign: Urbana, IL, USA, 1990.
4. Boman, E.; Hendrickson, B.; Vavasis, S. Solving Elliptic Finite Element Systems in Near-Linear Time with Support Preconditioners. *SIAM J. Numer. Anal.* **2008**, *46*, 3264–3284.
5. Christiano, P.; Kelner, J.A.; Madry, A.; Spielman, D.A.; Teng, S.H. Electrical Flows, Laplacian Systems, and Faster Approximation of Maximum Flow in Undirected Graphs. In Proceedings of the 43rd ACM Symposium on Theory of Computing (STOC), San Jose, CA, USA, 6–8 June 2011; pp. 273–282.
6. Meyerhenke, H.; Nöllenburg, M.; Schulz, C. Drawing Large Graphs by Multilevel Maxent-Stress Optimization. In *Graph Drawing and Network Visualization—23rd International Symposium, GD 2015, Revised Selected Papers*; Springer: Berlin/Heidelberg, Germany, 2015; Volume 9411, pp. 30–43.
7. Spielman, D.A.; Srivastava, N. Graph Sparsification by Effective Resistances. *SIAM J. Comput.* **2011**, *40*, 1913–1926.
8. Kelner, J.A.; Madry, A. Faster Generation of Random Spanning Trees. In Proceedings of the 50th Annual IEEE Symposium on Foundations of Computer Science (FOCS), Los Alamitos, CA, USA, 2009; pp. 13–21.
9. Diekmann, R.; Frommer, A.; Monien, B. Efficient schemes for nearest neighbor load balancing. *Parallel Comput.* **1999**, *25*, 789–812.

10. Meyerhenke, H.; Schamberger, S. A Parallel Shape Optimizing Load Balancer. In Proceedings of the 12th International Euro-Par Conference (Euro-Par 2006), Dresden, Germany, 28 August–1 September 2006; pp. 232–242.
11. Grady, L.; Schwartz, E.L. Isoperimetric graph partitioning for image segmentation. *IEEE Trans. Pattern Anal. Mach. Intell.* **2006**, *28*, 469–475.
12. Kelner, J.A.; Orecchia, L.; Sidford, A.; Zhu, Z.A. 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, Palo Alto, CA, USA, 1–4 June 2013; pp. 911–920.
13. Reif, J. Efficient approximate solution of sparse linear systems. *Comput. Math. Appl.* **1998**, *36*, 37–58.
14. Spielman, D.A.; Woo, J. A Note on Preconditioning by Low-Stretch Spanning Trees. 2009, arXiv:0903.2816. Available online: https://arxiv.org/abs/0903.2816 (accessed on 25 October 2016).
15. Koutis, I.; Levin, A.; Peng, R. Improved spectral sparsification and numerical algorithms for SDD matrices. In Proceedings of the 29th Symposium on Theoretical Aspects of Computer Science (STACS), Paris, France, 29 February–3 March 2012; pp. 266–277.
16. Koutis, I.; Miller, G.L.; Peng, R. Approaching Optimality for Solving SDD Linear Systems. *SIAM J. Comput.* **2014**, *43*, 337–354.
17. Spielman, D.A. Laplacian Linear Equations, Graph Sparsification, Local Clustering, Low-Stretch Trees, etc. Available online: https://sites.google.com/a/yale.edu/laplacian/ (accessed on 26 October 2016)
18. Peng, R.; Spielman, D.A. An Efficient Parallel Solver for SDD Linear Systems. In Proceedings of the 46th Annual ACM Symposium on Theory of Computing, New York, NY, USA, 31 May–3 June 2014; pp. 333–342.
19. Koutis, I. Simple parallel and distributed algorithms for spectral graph sparsification. In Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), Prague, Czech Republic, 23–25 June 2014; pp. 61–66.
20. Alon, N.; Karp, R.M.; Peleg, D.; West, D. A Graph-Theoretic Game and its Application to the k-Server Problem. *SIAM J. Comput.* **1995**, *24*, 78–100.
21. Elkin, M.; Emek, Y.; Spielman, D.A.; Teng, S.H. Lower-stretch Spanning Trees. In Proceedings of the 37th Annual ACM Symposium on Theory of Computing (STOC), Baltimore, MD, USA, 22–24 May 2005; pp. 494–503.
22. Abraham, I.; Bartal, Y.; Neiman, O. Nearly Tight Low Stretch Spanning Trees. In Proceedings of the 49th Annual Symposium on Foundations of Computer Science, Philadelphia, PA, USA, 26–28 October 2008; pp. 781–790.
23. Abraham, I.; Neiman, O. Using Petal-decompositions to Build a Low Stretch Spanning Tree. In Proceedings of the 44th ACM Symposium on Theory of Computing, New York, NY, USA, 20–22 May 2012; pp. 395–406.
24. Papp, P.A. Low-Stretch Spanning Trees. Bachelor Thesis, Eötvös Loránd University, Budapest, Hungary, 2014. Available online: http://www.cs.elte.hu/blobs/diplomamunkak/bsc_alkmat/2014/papp_pal_andras.pdf (accessed on 25 October 2016).
25. Koutis, I.; Miller, G.L.; Tolliver, D. Combinatorial Preconditioners and Multilevel Solvers for Problems in Computer Vision and Image Processing. *Comput. Vis. Image Underst.* **2011**, *115*, 1638–1646.
26. Livne, O.E.; Brandt, A. Lean algebraic multigrid (LAMG): Fast Graph Laplacian Linear Solver. *SIAM J. Sci. Comput.* **2012**, *34*, B499–B522.
27. Dell'Acqua, P.; Frangioni, A.; Serra-Capizzano, S. Accelerated multigrid for graph Laplacian operators. *Appl. Math. Comput.* **2015**, *270*, 193–215.
28. Dell'Acqua, P.; Frangioni, A.; Serra-Capizzano, S. Computational evaluation of multi-iterative approaches for solving graph-structured large linear systems. *Calcolo* **2015**, *52*, 425–444.
29. Boman, E.G.; Deweese, K.; Gilbert, J.R. Evaluating the Dual Randomized Kaczmarz Laplacian Linear Solver. *Informatica* **2016**, *40*, 95–107.
30. Hoske, D.; Lukarski, D.; Meyerhenke, H.; Wegner, M. Is Nearly-Linear the Same in Theory and Practice? A Case Study with a Combinatorial Laplacian Solver. In Proceedings of the 14th International Symposium on Experimental Algorithms (SEA 2015), Paris, France, 29 June–1 July 2015; pp. 205–218.
31. Harel, D.; Tarjan, R.E. Fast Algorithms for Finding Nearest Common Ancestors. *SIAM J. Comput.* **1984**, *13*, 338–355.
32. Bender, M.A.; Farach-Colton, M. The LCA Problem Revisited. In *LATIN 2000: Theoretical Informatics*; Springer: Berlin/Heidelberg, Germany, 2000; Volume 1776, pp. 88–94.

33. Sleator, D.D.; Tarjan, R.E. A data structure for dynamic trees. *J. Comput. Syst. Sci.* **1983**, *26*, 362–391.
34. Staudt, C.L.; Sazonovs, A.; Meyerhenke, H. NetworKit: A Tool Suite for Large-scale Complex Network Analysis. *Netw. Sci.* **2016,** accepted.
35. Hoske, D.; Lukarski, D.; Meyerhenke, H.; Wegner, M. Implementation of KOSZ solver. Information: Available online: http://parco.iti.kit.edu/software-en.shtml, code: Available online: https://algohub.iti.kit.edu/parco/NetworKit/NetworKit-SDD (accessed on 25 October 2016).
36. Guennebaud, G.; Jacob, B. Eigen v3. 2011. Available online: http://eigen.tuxfamily.org (accessed on 25 October 2016).
37. Lukarski, D. Paralution—Library for Iterative Sparse Methods. 2015. Available online: http://www.paralution.com (accessed on 20 November 2015).
38. Barabási, A.L.; Albert, R. Emergence of Scaling in Random Networks. *Science* **1999**, *286*, 509–512.
39. Browne, S.; Dongarra, J.; Garner, N.; Ho, G.; Mucci, P. A Portable Programming Interface for Performance Evaluation on Modern Processors. *Int. J. High Perform. Comput. Appl.* **2000**, *14*, 189–204.
40. Demmel, J.W. *Applied Numerical Linear Algebra*; Society for Industrial and Applied Mathematics: Philadelphia, PA, USA, 1997.
41. Marquardt, D. An Algorithm for Least-Squares Estimation of Nonlinear Parameters. *J. Soc. Ind. Appl. Math.* **1963**, *11*, 431–441.
42. Saad, Y. *Iterative Methods for Sparse Linear Systems*, 2nd ed.; SIAM: Philadelphia, PA, USA, 2003.
43. Axelsson, O.; Vassilevski, P. A Black Box Generalized Conjugate Gradient Solver with Inner Iterations and Variable-Step Preconditioning. *SIAM J. Matrix Anal. Appl.* **1991**, *12*, 625–644.
44. Briggs, W.L.; Henson, V.E.; McCormick, S.F. *A Multigrid Tutorial*; SIAM: Philadelphia, PA, USA, 2000.

© 2016 by the authors; licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC-BY) license (http://creativecommons.org/licenses/by/4.0/).