

École polytechnique de Louvain

Analysis of a nearly-linear time Laplacian solver and its application in finite element method

Author: **Christophe HENEFPE**

Supervisor: **Jean-Charles DELVENNE**

Readers: **Jean-Francois REMACLE, Vincent LEGAT**

Academic year 2022–2023

Master [120] in Mathematical Engineering

Contents

1	Introduction	3
1.1	Structure	4
2	Background	5
2.1	Laplacian of graphs	5
2.1.1	Test graphs	6
2.2	General symmetric diagonal dominant matrices	7
2.3	Traditional solvers	8
2.3.1	Direct solvers	8
2.3.2	Iterative solvers	11
2.3.3	Preconditioning	12
2.3.4	Multigrid method	13
3	Combinatorial preconditioner	14
3.1	Vaidya’s support graph preconditioner	14
3.2	Low-stretch spanning tree	15
3.3	Spielman-Teng solver	17
3.4	Spielman and Srivistava	20
3.4.1	Effective resistance	20
3.4.2	Sampling algorithm	21
3.5	Koutis-Miller-Peng’s first (KMP)-solver	23
3.6	Other related works	24
4	KMP solver analysis	26
4.1	KMP solver improvements	26
4.2	Test graphs properties	31
4.3	Sampling procedure	32
4.3.1	Sampling based on effective resistance	32
4.3.2	Sampling based on stretch	34
4.4	Spine-heavy graphs	37
4.5	Improvement (top level)	39

4.6	General graphs	40
4.7	Error accumulation	42
4.8	Graphs with lognormal weight distribution	45
5	Low-stretch spanning tree	47
5.1	Low-stretch spanning tree algorithm	47
5.2	Finite difference problem	50
5.3	Time needed to generate the preconditioner	52
6	Application in finite element method	54
6.1	Poisson problem	54
6.2	A magnetostatics problem	57
7	Conclusion	61
7.1	Future works	62

Chapter 1

Introduction

Solving linear system is of fundamental importance in science and engineering. Recent researches lead to the development of fast algorithms to solve symmetric diagonally dominant (SDD) linear systems. A matrix A is SDD if

$$A(i, i) \geq \sum_{j \neq i} |A(i, j)| \quad \forall i.$$

One extremely important class of matrices that is SDD are called Laplacian matrices of graphs, which will be formally defined in Chapter 2.

There are many real life applications where such systems need to be solved. These come, for example, from solving partial differential, applications in networks (Link prediction, recommendation system, protein interaction networks, ...), applications in computer vision (image denoising, image inpainting, image segmentation, ...) and many others.

There exists various SDD solvers that work empirically well for specific applications. However, they often do not generalize well for other problems. Moreover, the existing SDD solvers lack theoretical guarantees on their performances.

The recent researches lead to the creation of algorithms called combinatorial preconditioner with provably good running time. Very briefly, a combinatorial preconditioner of the Laplacian matrix L_G of a graph G is the Laplacian matrix L_H of a subgraph H of G with some additional edges. It was a breakthrough from Vaidya in 1990 that allowed to design an algorithm with provably good bounds !

There have been many combinatorial preconditioners since the first idea emerged. The most important one came from Spielman and Teng, who were able to create the first ever almost-linear time algorithm to solve SDD systems, ie the solver has a total complexity of

$$\mathcal{O}(n \log^c n),$$

for some huge constant c . All the solvers that followed then optimized the value of this constant. These were crucial in the design of the solver of Koutis, Miller and Peng (KMP) which has a theoretical complexity of

$$\mathcal{O}(m \log n).$$

This solver has never been implemented and will be the main focus of this thesis.

The algorithms that generate the preconditioner work as follows: they start by generating a good subtree of the system and then add some of the important edges to the tree based

on some algorithm. The subtree is called a low-stretch spanning tree. It is an important notion that will be defined in Chapter 3 and is the key to obtaining the almost-linear run-time. The preconditioner is easier to solve and leads to the resolution of a much smaller system, which is then solved recursively. By controlling the quality and the size of the successive smaller systems, it is possible to prove that the solver runs in almost-linear time.

1.1 Structure

Chapter 2 will formally define Laplacian matrices of graphs as well as their properties. It will be followed by a general outlook of the different current solvers used to solve such systems.

Chapter 3 will be purely theoretical and will give an overview of the important tools that have been developed by the different preconditioners in order to understand the KMP-solver. It will be a condensed summary of several papers.

Chapter 4 will in a first part demonstrate the proofs needed to obtain the theoretical bound of the KMP-solver and will closely follow their paper. Then, it will be extensively tested in practice using my implementation on various test graphs in order to understand the strengths and the weaknesses of the solver. Each part of the algorithm will be tested and analyzed. The goal is to determine whether the practical performances follow the theoretical ones or not. It will then be assessed whether the hidden constant is large or not to determine if the KMP-solver is competitive against classical solvers, such as the conjugate gradient.

Chapter 5 will test the affects of the low-stretch spanning trees on the solver. The time needed to generate the preconditioners will also be assessed.

Finally, the solver will be tested on three concrete problems. The first one is a fluid simulation problem that is discretized using finite differences.

The second one will be the resolution of a Poisson equation using finite elements method.

Finally, the last test consists of the simulation of a magnetostatics problem with finite element aswell.

The goal will be to test the practical performances on real problems and to test its competitiveness against other algorithms.

Chapter 2

Background

In this chapter, we first define the Laplacian matrices of graphs and their properties. It is mainly such matrices that will be used to extensively test the KMP solver in this thesis. Then, it will be shown that the general symmetric diagonally dominant matrix can always be reduced to a Laplacian system with a linear cost.

After that, a variety of traditional solvers used to solve these type of systems will be provided. There are two main approaches to solve linear systems. The first are direct methods and compute the exact solution of the system. The second are iterative methods that generate a sequence of successively better approximations of the solution.

2.1 Laplacian of graphs

A graph is a triplet $G = (V, E, \phi)$, where

- V is the set of nodes (or vertices).
- E is the set of edges.
- ϕ is an incident function and associates every edge to a pair of vertices.

A graph is weighted if a value $w(e) = w(u, v)$ is assigned to every edge $e = (u, v)$. From now on, all graphs will be considered as connected weighted undirected graphs with positive weights $w(e) > 0$ and will be described by $G = (V, E, w)$. If n and m denote the number of nodes and the number of edges of G respectively, then the adjacency matrix $A \in \mathbb{R}^{n \times n}$ of the graph is defined by

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

The degree matrix of a weighted graph is a diagonal matrix such that

$$D(u, u) = \sum_v w_{uv}.$$

The Laplacian of a weighted graph G can then be defined as

$$L_G = D - A.$$

By construction, a Laplacian matrix of a graph has the following properties:

1. It is symmetric.
2. It has non positive off-diagonal entries.
3. Each row sums to zero.

It is therefore a weakly diagonal dominant matrix. More generally, any matrix satisfying these three properties are called a Laplacian matrix, as it is always possible to create a weighted graph for which it is the Laplacian.

By assigning an arbitrary orientation to each edge e of the undirected graph, we can define the directed incidence matrix $B \in \mathbb{R}^{m \times n}$ of the graph by

$$B(e, v) = \begin{cases} 1 & \text{if } v \text{ is the head of } e \\ -1 & \text{if } v \text{ is the tail of } e \\ 0 & \text{otherwise} \end{cases}$$

We can also define the diagonal matrix $W \in \mathbb{R}^{m \times m}$ such that

$$W(e, e) = w(e) \leq 0.$$

Then, the Laplacian matrix can be factored as

$$L = B^T W B.$$

Using this factorisation, it can be proven that the Laplacian matrix is semi-positive definite. Indeed, we have

$$x^T L x = x^T B^T W B x = \|W^{\frac{1}{2}} B x\|_2^2 \geq 0, \forall x \in \mathbb{R}^n.$$

Moreover, Laplacian matrices of connected graphs are singular. It has a unique zero eigenvalue with the constant vector as an eigenvector. Indeed, we have

$$\begin{aligned} x^T L x = 0 &\Leftrightarrow \|W^{1/2} B x\|_2^2 = 0 \\ &\Leftrightarrow \sum_{(u,v) \in E} w(u, v) (x_u - x_v)^2 = 0 \\ &\Leftrightarrow x_u - x_v = 0 \quad \forall \text{ edges } (u, v). \end{aligned}$$

x is thus a constant vector since the graph is connected. More generally, if a graph is not connected but instead has k connected components, then the Laplacian matrix has exactly k zero eigenvalues, with a constant eigenvectors on each connected component [Mar13].

2.1.1 Test graphs

This short section will list a series of test graphs that will be used to test the solver. These different models each have different unique properties (degree heterogeneity, density heterogeneity, planarity, ...) that are present in real life systems, such as matrices arising in social networks or matrices arising from discretising PDE equations. The considered graphs in this thesis will be modelled using

- Erdős–Rényi random graph $G(n, m)$ (ER, see [Wik23c]). These are obtained by choosing a graph uniformly at random from the collection of all graphs of n nodes with m edges. These type of graphs serves as a 'null model' for social network graphs. They do not have most of the properties of typical social network graphs but are useful as a test case for its simplicity.
- The Barabási–Albert model (BA [Wik23b]), which generates graphs as follows: it starts from a small network and then n nodes are added to the graph using the preferential attachment process. This means that each new nodes gets connected to k existing nodes with a probability that is proportional to the degree of the existing nodes (ie, $p_i = \frac{\deg(i)}{\sum_j \deg(j)}$). This model generates scale-free networks, which are graphs whose node degree distribution follows a power-law distribution. As a result, the degree distribution is heterogeneous, some nodes are much more connected than others. This is typical of many social networks.
- The stochastic-block model (SBM) partitions n nodes into k blocks. Each pair of nodes from the block i and block j are then connected with probability p_{ij} . This allows to model graphs with heterogeneous density. If the probability to connect nodes from two different blocks are small compared to the probability of connecting nodes from the same block, then the graph will be constituted of k communities with a few edges in-between. Then, the Laplacian matrix will have a unique zero eigenvalue (with constant eigenvectors) and $k - 1$ eigenvalues that are close to zero with eigenvectors approximately constant on each connected component. This allows to have bad condition numbers.
- The last graphs used are two dimensional rectangular grids. Laplacian systems on grid appear when solving boundary value problems on rectangular domain using for example finite differences. The degree distribution of such graphs is extremely homogeneous. Almost all nodes have degree 4. These graphs are known to have condition number evolving as $\kappa \in \mathcal{O}(n)$, which makes the conjugate gradient method unaffordable for big problems.

These graphs will be used to understand the strengths and the weaknesses of the solver.

2.2 General symmetric diagonal dominant matrices

The solver works for more general matrices than only Laplacian ones. The techniques used can directly be adapted to symmetric diagonal dominant matrices with non-positive off diagonal elements that are positive definite. (SDDM-matrix). These can be decomposed as

$$A = A_L + A_D.$$

where A_L is a Laplacian matrix and A_D is a diagonal matrix with non-negative elements. If B_L is a good preconditioner for A_L , then using $B_L + A_D$ as a preconditioner for A will give good performances. This is characterized by the following property:

$$\kappa(A, B_L + D) \leq \kappa(A_L, B_L),$$

where $\kappa(A, B)$ designates the generalized condition number and is equal to the ratio of the largest and smallest eigenvalues of $AB^\dagger$ ($\dagger$ denotes the Moore–Penrose inverse).

This allows to use the preconditioner of Laplacian matrices to easily create good ones for SDDM systems.

For the general symmetric diagonal dominant systems (SDD), it was proven in [Gre96] that solving these systems can always be reduced to solving a Laplacian system that is twice as large. Indeed, let $A \in \mathbb{R}^{n \times n}$ be an SDD matrix. Then A can be decomposed as

$$A = D + A_+ + A_-.$$

where D contains the diagonal entries of A , while A_+ and A_- contains the positive and negative off-diagonal entries of A respectively. The matrix D can then be further decomposed into

$$D = D_1 + D_2.$$

where $(D_1)_{ii} = \sum_{j \neq i} |A_{ij}|$ and $D_2 = D - D_1$. The matrix $\tilde{A} \in \mathbb{R}^{2n \times 2n}$ defined by

$$\tilde{A} = \begin{bmatrix} D_1 + \frac{D_2}{2} + A_- & -\frac{D_2}{2} - A_+ \\ -\frac{D_2}{2} - A_+ & D_1 + \frac{D_2}{2} + A_- \end{bmatrix}$$

is a Laplacian matrix. Let $x \in \mathbb{R}^{2n}$ and $b \in \mathbb{R}^{2n}$ such that

$$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \quad \tilde{b} = \begin{bmatrix} b \\ -b \end{bmatrix}.$$

If $\tilde{x}$ is the solution to the system $\tilde{A}\tilde{x} = \tilde{b}$, then the vector $x = \frac{x_1 - x_2}{2}$ is solution to $Ax = b$.

Finally, Laplacian systems are singular with a unique zero eigenvalue. A system $Lx = b$ possess a solution if $b \in \text{Im}(L)$, which is respected if the sum of the entries of b gives 0, since the eigenvector associated to the zero eigenvalue is the constant vector. In such systems, one has to impose the solution on one of the variables to remove the degree of freedom. In this thesis, the degree of freedom is removed by posing $x_1 = 0$. Let x_2 be the vector x without the first entry and $L_{2:,2:}$ be the submatrix of L containing all but the first row and column of L . Then the system

$$L_{2:,2:}x_2 = b_2.$$

is an SDDM system. These type of matrix are the ones that appear in the Finite element methods.

To summarize, general SDD systems can be converted to Laplacian systems, which in turn give rise to a SDDM system after fixing the degree of freedom.

2.3 Traditional solvers

2.3.1 Direct solvers

Direct solvers allows to solve linear systems exactly under infinite precision. For Laplacian matrices, a Cholesky factorisation is performed beforehand since these are symmetric positive definite. The matrix L is factored into

$$A = CC^T,$$

with C being a lower triangular matrix. Solving this is then done by a forward and a backward substitution ($Cy = b$ and $C^T x = y$). The factoring cost is in $\mathcal{O}(n^3)$, which is prohibitively expensive. The cost of the backward and forward substitution is in $\mathcal{O}(n^2)$, which becomes extremely useful in cases where several linear systems with the same matrix must be solved.

The Cholesky factorization algorithm can be described by a recursive series of step as follows: it start with the matrix $A_1 = A$. Then, the first row and column is factored following

$$A_1 = \begin{bmatrix} a_{11} & b_1^T \\ b_1 & B_1 \end{bmatrix} = C_1 A_2 C_1^T,$$

$$\text{where } C_1 = \begin{bmatrix} \sqrt{a_{11}} & 0 \\ \frac{b_1^T}{\sqrt{a_{11}}} & B_1 \end{bmatrix} \quad \text{and} \quad A_2 = \begin{bmatrix} 1 & 0 \\ 0 & B_1 - \frac{1}{\sqrt{a_{11}}} b_1 b_1^T \end{bmatrix}.$$

This step is then recursively applied on the remaining matrix A_1 . More generally, at the i^{th} step of the factorization, the $i - 1$ first rows and columns have already been factored, and the remaining matrix is factored by

$$A^{(i)} = \begin{bmatrix} \mathbf{I}_{i-1} & 0 & 0 \\ 0 & a_{ii} & b_i^T \\ 0 & b_i & B_i \end{bmatrix},$$

$$\text{where } C_i = \begin{bmatrix} \mathbf{I}_{i-1} & 0 & 0 \\ 0 & \sqrt{a_{ii}} & 0 \\ 0 & \frac{1}{\sqrt{a_{ii}}} b_i & I_{n-i} \end{bmatrix} \quad \text{and} \quad A_{i+1} = \begin{bmatrix} \mathbf{I}_{i-1} & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & B_i - \frac{1}{a_{ii}} b_i b_i^T \end{bmatrix}.$$

This process is repeated until $i = n$. The lower triangular matrix is given by $C = C_1 C_2 \dots C_n$. The problem that can arise with this algorithm is the so called fill-in, which occurs when a zero element of A becomes a non-zero entry in one of the Cholesky factors. The step $B_i - \frac{1}{a_{ii}} b_i b_i^T$ will most likely destroy the sparse structure of B_i . In fact, this step has a clean graph theoretic interpretation by viewing the rows and columns as the edges of the underlying graph. Consider for example the matrix given in 2.1, with its corresponding graph.



Figure 2.1. Example

The resulting matrix obtained by eliminating the first row and column is given in 2.2. From a graph theoretic point of view, we observe that the remaining matrix is Laplacian matrix of the graph in which that node has been removed, and in which all its neighbours have been connected. The fill-in are thus the newly created edges. It is therefore a bad

idea to factor nodes with high degree, since they have a high number of neighbours which all become connected.

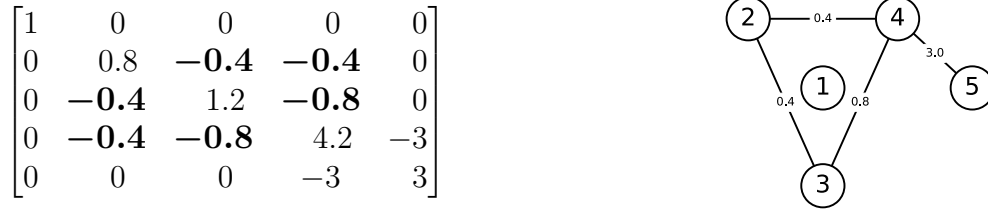


Figure 2.2. Cholesky factorization of node 1. The fill-in is shown in bold

The choice of which row and column to factorize can thus have a huge impact on the performance of the solver. For example, the result of factoring the node number 5 first is given in 2.3. We observe that factoring a node of degree one does not create any fill-in. From a graph theoretical point of view, the remaining system is the graph where the node has simply been removed.



Figure 2.3. Factoring a node of degree 1.

Factoring a node of degree two is also efficient. For example, factoring the node number 4 gives the result in 2.4. It is equivalent to remove the node and to add single new edge between its two neighbours. Thus, at most two new nonzero elements are added to the matrix. More generally, consider a node of degree two which has weight w_1 and w_2 for its edges. Then eliminating this node using Cholesky factorization will create a new edge of weight $\frac{1}{\frac{1}{w_1} + \frac{1}{w_2}}$ between its two neighbours. If there were already a connection between the neighbours, then this value is added to it. In such case, there would be no fill-in at all.

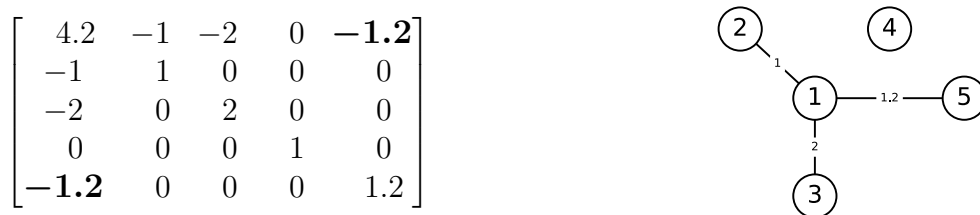


Figure 2.4. Factoring a node of degree 2.

A good reordering of the variables must be applied to reduce the fill-in as much as possible. The factorization with the reordering then respects $PAP^T = LL^T$. The problem of finding the permutation that gives the least amount of fill-in is NP-complete, but several good heuristics exist, such as the Reverse-Cuthill-MacKee algorithm ([LS76]), that performs well in practice. The algorithms in the combinatorial preconditioners do not use such algorithms. Instead, they make use of the two properties mentioned above. They will factor all nodes of degree first and then all nodes of degree 2.

2.3.2 Iterative solvers

Instead of solving the system exactly, iterative methods generate a sequence of improving approximate solutions to the linear system.

The Conjugate Gradient, given in Algorithm 1, is a Krylov subspace method that is well suited to solve Laplacian systems. The Krylov subspaces are spanned by

$$K_n = \{b, Ab, A^2b, A^3b, \dots, A^{n-1}b\}.$$

The CG algorithm generates a unique sequence of iterates $x_n \in K_n$ such that the A -norm ($\|x\|_A = \sqrt{x^T A x}$) of the residual, $\|x_n - x^*\|_A$, is minimized at each iteration. Under exact arithmetic, ie if the orthogonality of the Krylov subspaces is strictly maintained at each iteration, then the CG method is guaranteed to converge in at most n iterations.

However, in practice this is never the case because the Krylov subspaces obtained during the successive iteration are not orthogonal due to numerical errors. As a result, the number of iteration is in practice given by an upper bound instead of the ideal value of n . It is derived from the convergence rate of the CG method, which follows from

$$\frac{\|e_n\|_A}{\|e_0\|_A} \leq 2 \left(\frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1} \right)^n,$$

with

$$\kappa = \frac{\lambda_{max}}{\lambda_{min}}$$

being the condition number of the system. Thus, the maximum number of iterations required to obtain solutions with relative error smaller than some precision ϵ is given by

$$n \leq \frac{1}{2} \sqrt{\kappa} \log \left(\frac{2}{\epsilon} \right).$$

Algorithm 1 Conjugate Gradient

```

 $x_0 = 0$ 
 $r_0 = b$ 
 $p_0 = r_0$ 
for  $n = 1, 2, \dots$ , until convergence do
   $\alpha_n = \frac{r_{n-1}^T r_{n-1}}{p_{n-1}^T A p_{n-1}}$ 
   $x_n = x_{n-1} + \alpha_n p_{n-1}$ 
   $r_n = r_{n-1} - \alpha_n A p_{n-1}$ 
   $\beta_n = \frac{r_n^T r_n}{r_{n-1}^T r_{n-1}}$ 
   $p_n = r_n + \beta_n p_{n-1}$ 
end for

```

The cost of each iteration is dominated by the cost of the matrix vector product Ap_{n-1} , which requires $\mathcal{O}(m)$ flops for sparse matrices with m non zero elements. The overall complexity of the conjugate gradient is thus $\mathcal{O}(\sqrt{\kappa}m)$.

2.3.3 Preconditioning

The convergence of the CG algorithm grows with the condition number of the system, which can increase drastically for large systems. The convergence can thus be extremely slow as a result. In practice, CG is never used alone but with a preconditioner.

The basic principle of preconditioning an iterative solver is to modify an ill-conditioned system

$$Ax = b$$

in such a way that the condition number of the modified system is better. For symmetric matrices, the modified equations are obtained using a preconditioner matrix M :

$$M^{-1/2}AM^{-1/2}x = M^{-1/2}b.$$

Applying the conjugate gradient algorithm on the modified system is called the preconditioned conjugate gradient method (see algorithm 2), and the convergence rate is then bounded by the generalized condition number

$$\kappa(A, M) := \kappa(M^{1/2}AM^{1/2}).$$

A good preconditioner must respect the following properties:

1. $\kappa(A, M)$ must be much smaller than $\kappa(A)$.
2. Solving the system $Mx = b$ must be relatively cheap compared to solving $Ax = b$.
3. The cost to obtain the preconditioner must be reasonable as well.

Algorithm 2 Preconditioned Conjugate Gradient

```

 $x_0 = 0$ 
 $r_0 = b$ 
for  $n = 0, 1, \dots$ , until convergence do
   $z_n = M^{-1}r_n$ 
   $\beta_{n-1} = \frac{z_n^T r_n}{z_{n-1}^T r_{n-1}}$  ( $\beta_{-1} = 0$ )
   $p_n = z_n + \beta_{n-1}p_{n-1}$ 
   $\alpha_n = \frac{z_n^T r_n}{p_n^T Ap_n}$ 
   $x_{n+1} = x_n + \alpha_n p_n$ 
   $r_{n+1} = r_n - \alpha_n Ap_n$ 
end for

```

One popular choice for M is given by the incomplete Cholesky factorisation of A . It is obtained by performing the Cholesky factorisation with the additional constraint of preserving the nonzero structure of the initial matrix so as to not create any fill-in. In practice, this works extremely well but there are no theoretical guarantee on the condition number of the preconditioned system.

2.3.4 Multigrid method

Multigrid methods are classical methods for solving linear systems arising in elliptic PDEs. They can be used as an "accelerator" (i.e. as an iterative solver) or as a preconditioner, the difference being that a reduced tolerance is used when a multigrid method is used as a preconditioner (less cycles or less levels) [Wes95]. Multigrid methods use several "grid levels". Geometric multigrid methods geometrically partition the computational grid, while algebraic multigrid methods [Stü01] use the same multigrid principles but in a way that only depends on the coefficients in the underlying matrix. The multigrid approach used as an accelerator has the advantage of having an optimal complexity which is sometimes $\mathcal{O}(n)$, which is obviously remarkable. We used the word "sometimes" on purpose because this optimal complexity is only observed in real life applications when the grid levels are "natural" i.e. when structured discretizations are used.

Chapter 3

Combinatorial preconditioner

The breakthrough that lead to the conception of almost linear-time algorithms to solve SDD system was to precondition the Laplacian matrix of a graph by the Laplacian matrix of a subgraph of the matrix. This idea originated from Vaidya in 1991. This was followed by the conception of a family of preconditioners that are called subgraph preconditioners or combinatorial preconditioners.

This chapter will present the tools that have been used to create the various combinatorial preconditioners. Each new solver improved upon its predecessor and introduced crucial techniques that were needed in the conception of the solver by Koutis, Miller and Peng, which is the solver of interest in this thesis. This section follows the work of [KMP12]

3.1 Vaidya's support graph preconditioner

This section briefly explain the first combinatorial preconditioner and mainly follows the paper [BGH⁺06].

For a long time, preconditioning Laplacian matrices was based essentially on heuristics. Vaidya, in 1990, was the first to use spectral graph theoretic properties to design good graph preconditioners with rigorous bounds on running times [Vai91]. It uses support theory [BH03] to prove inequalities on Laplacian matrices. Let A and B be semi-definite matrices. Then the inequality

$$A \succeq B.$$

means that $B - A$ is positive definite, which is equivalent to

$$x^T A x \leq x^T B x \quad \forall x \in \mathbb{R}^n.$$

Then, the following implication is the main tool used to bound the generalized condition number between the Laplacian matrix L_G of a graph G and its preconditioner L_H , which is the Laplacian of a graph H :

$$\alpha L_H \preceq L_G \preceq \beta L_H \Rightarrow \kappa(L_G, L_H) \leq \frac{\beta}{\alpha}.$$

H is called a t -approximate of G if $\kappa(L_G, L_h) \leq t$.

Vaidya's original idea was to simply use a spanning tree as a preconditionner of an iterative solver because the Cholesky factorisation of a Tree creates no fill-in. This means that

solving a system in a tree is easy and can be done in $\mathcal{O}(n)$. Vaidya used a maximum-weight spanning tree as a preconditioner and proved using support-graph theory that such a preconditioner is an $\mathcal{O}(mn)$ -approximate of a matrix A whose underlying graph has n nodes and m edges. This doesn't improve the complexity of the conjugate gradient. Indeed, using PCG with such a preconditioner gives a total complexity of $\mathcal{O}(\sqrt{mn}m)$. On the other hand, CG ends in n iteration under perfect arithmetic, which thus requires a total complexity of $\mathcal{O}(nm)$.

Vaidya then proposed to add some of the most important edges back to the spanning tree so as to improve the generalized condition number. However, this comes with the cost of increasing the complexity of solving the system. To decide which edges to add, Vaidya defines a parameter t and decomposes the graph into t connected subgraphs $V_1, V_2, \dots, V_t$ of roughly the same size (each has at most $\frac{m}{t}$ nodes). The heaviest edge between every pair of subgraphs is then added back to the tree. The generalized condition number of such a preconditioner is bounded by $\kappa(A, B) \leq \mathcal{O}(\frac{n^2}{t^2})$.

The cost to factor this matrix is inevitably higher since it isn't a simple tree anymore. For general graphs, this is done in two steps. First, all degree 1 and 2 nodes are repeatedly factored using a partial Cholesky on those nodes until all nodes in the graph have degree at least 3. This takes $\mathcal{O}(n)$ time and only creates a fill-in of $\mathcal{O}(n)$. They proved that the remaining smaller graph has at most $2t(t-1)$ nodes in the worst case. This is factored in at most $\mathcal{O}((t^2)^3)$ time.

The parameter t represents the compromise between the number of iterations needed to converge and the cost of factorisation. A high value of t gives a small generalized condition number but with considerable fill-in to factor the system, whereas a small value of t gives a bigger generalized condition number with low fill-in during the factorisation. For general graph, the optimal value of t is $\Theta(n^{0.25})$ and the total cost of the solver is $\mathcal{O}(n^{1.75})$.

For a planar graph, the smaller matrix is factored using nested dissection. The total cost of the solver when t is chosen optimally is $\mathcal{O}(n^{1.2})$.

3.2 Low-stretch spanning tree

In 2001, Boman and Hendrickson observed that using low-stretch spanning trees as preconditioner inside the conjugate gradient algorithm to solve Laplacian matrices gives better performances than using a maximum weight spanning tree [BH01].

This section will formally define the notion of stretch and its impact on combinatorial preconditioners. It will mainly follow [SW09].

Let G be a graph and T be a spanning tree of G . Then, for any edge $e \in E$, if the only path connecting the endpoints of e in T consists of $e_1, \dots, e_k$, then the stretch of e with respect to T is defined by

$$st_T(e) = \frac{\sum_{i=1}^k \frac{1}{w(e_i)}}{\frac{1}{w(e)}} = w(e) \left(\sum_{i=1}^k \frac{1}{w(e_i)} \right).$$

The stretch of an edge is a measure of the penalty incurred by traversing the spanning tree instead of the edge itself. An edge with a high stretch means that it represents a significant shortcut with respect to the tree. Therefore, the stretch of an edge offers an estimation of its importance in relation to the tree.

The stretch of the graph G with respect to T is then defined by

$$st_T(G) = \sum_{e \in E} st_T(e).$$

A low-stretch spanning tree (LSST) is a spanning tree that minimises this value. A LSST thus has the property of providing good detours for edges of the graph on average. Low-stretch spanning trees were first developed by Alon et al. for the k-server problem [AKPW95]. The breakthrough of using these spanning tree to solve Laplacian systems lead to the development of many theoretical algorithms to create such trees. These are summarized in Table 3.1. The state of the art LSST algorithm [AN19] allows to compute trees with total stretch $\mathcal{O}(m \log n \log \log n)$. However, the main goal of all these algorithms was to improve the theoretical bound on the total stretch and are not practical to implement.

	Time	Average stretch
[AKPW95]	$\mathcal{O}(m^2)$	$\exp(\mathcal{O}(\sqrt{\log n \log(\log n)}))$
[EEST05]	$\mathcal{O}(m \log^2(n))$	$\mathcal{O}(\log^2 n \log \log n)$
[ABN08]	$\mathcal{O}(m \log^2 n)$	$\mathcal{O}(\log n (\log \log n)^3)$
[AN19]	$\mathcal{O}(m \log n \log \log n)$	$\mathcal{O}(\log n \log \log n)$

Table 3.1. Overview of the low stretch spanning tree algorithms

Alon et al. proved that there exist a lower bound on the average stretch of $\Omega(\log n)$ for a variety of graphs such as the 2 dimensional grid. The algorithm of Abraham and Neiman is thus tight up to a polylogarithmic factor for these worst case graphs.

An illustration of low-stretch spanning tree is given in Figure 3.1. It shows the difference between a maximum weight spanning tree and a low stretch spanning tree for an unweighted 2d grid. The maximum weight spanning tree here is any arbitrary spanning tree since all weights are equal. The total stretch is $\mathcal{O}(n^{1.5})$. The low-stretch spanning tree on the right has total stretch $\mathcal{O}(\log n)$.

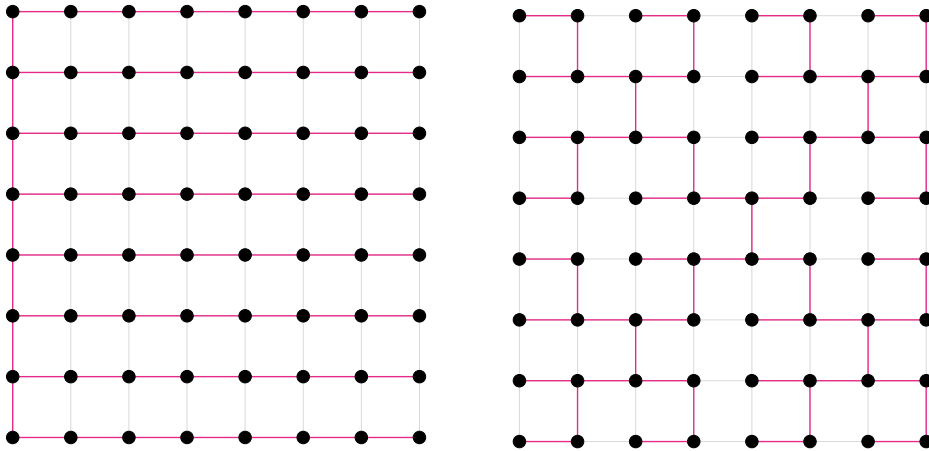


Figure 3.1. Two spanning trees for the unweighted grid. On the left, the spanning trees with high (poor) stretch. On the right a spanning tree has a low stretch. Figure from [McK15]

Let L_G and L_T be the Laplacian matrices associated with G and T respectively. Boman and Hendrickson proved that

$$\text{Tr}(L_G L_T^\dagger) = st_T(G).$$

This means that the maximum eigenvalue of $L_G L_T^\dagger$ is bounded by $st_T(G)$. Furthermore, the minimal eigenvalue is bounded by 1 because T is a subgraph of G . Thus, the generalized condition number of the preconditioned system is bounded by $\mathcal{O}(st_T(G))$, which gives a maximal complexity of $\mathcal{O}(\sqrt{st_T(G)}m) = \mathcal{O}(m^{\frac{4}{3}} \log^{\frac{1}{2}} n \log \frac{1}{\epsilon})$ with the state of the art algorithm for the LSST.

Spielman and Woo then made a more careful analysis of the distribution of the eigenvalues distribution of the preconditioned systems [SW09]. They bounded the number of iterations for solving Laplacian matrices with LSST with PCG by $\mathcal{O}(st_T(G)^{\frac{1}{3}})$. Using the bound from the best theoretical LSST algorithm, they obtain a total complexity of $\mathcal{O}(m^{\frac{4}{3}} \log^{\frac{1}{3}} n \log \frac{1}{\epsilon})$ to solve Laplacian systems. This is a stronger bound than $\mathcal{O}(\sqrt{\kappa})$ and relies on the effect of the distribution of the eigenvalues on conjugate gradient.

Papp et al. [PKBK14] tested several of the low-stretch spanning tree algorithms with provably low total stretch and has shown that in practice, they often provide spanning trees with worse stretch than those generated using simply Kruskal's algorithm.

Kruskal's maximum weight spanning tree does not have any theoretical guarantee on the average stretch of the graph since it was not designed to do so. However, it intuitively makes sense to use such trees. Let e be a heavy edge of the graph. Then, from the formula $st_T(e) = w_e \left(\sum_{i=1}^k \frac{1}{w_{e_i}} \right)$, we see that if e is not included in the tree, then it will greatly impact the total stretch. On the contrary, if it is included in the tree, then its contribution to the total stretch for each edge f not in the tree and whose path connecting its endpoints passes through e in the tree will be $w_f \frac{1}{w_e}$, which gets smaller as w_e gets bigger.

3.3 Spielman-Teng solver

From 2003, Spielman and Teng built upon these concepts further and created the first theoretical nearly linear time SDD solver. They brought two important modifications over the previous approaches. First, the procedure to add edges to the spanning tree is randomized. Second, the edges of the preconditioner are potentially re-weighted, such that the expected value of the preconditioner is the initial graph itself on an edge to edge basis. They defined the notion of spectral sparsifier [ST14], which are graphs that are good spectral approximation of the original graph (i.e, the generalized condition number is low) while having only a nearly-linear number of edges. The set of edges of the ultra-sparsifier is a subset of the edges of the original graph (but with potentially different weights). A (k, h) -ultra-sparsifier of a matrix A with m edges is formally defined as a matrix S such that

- $S \preceq A \preceq k \cdot S$.
- S has at most $n - 1 + hm/k$ edges.
- $S_{ij} \neq 0$ only if $A_{ij} \neq 0$.

To construct such an ultra-sparsifier, they combine a low stretch spanning tree with a sophisticated algorithm that adds edges to this tree. This involves the crucial use of

local clustering methods to create an approximation of the graph by sampling edges with different probabilities in each cluster, in order to keep many edges where it matters (low density parts between clusters) and discard edges where it can be allowed (dense clusters). To do so, Spielman and Teng developed an algorithm to identify the clusters in a graph using random walks. They used the property that by randomly walking in a graph, once the walker is inside a cluster, it is much more likely to stay inside it and continue to explore it than to stumble upon the few edges that leave the cluster. This clustering part, although theoretically interesting, was also a complex bottleneck in terms of implementations. Figure 3.2 shows an example of the sparsification of a graph.

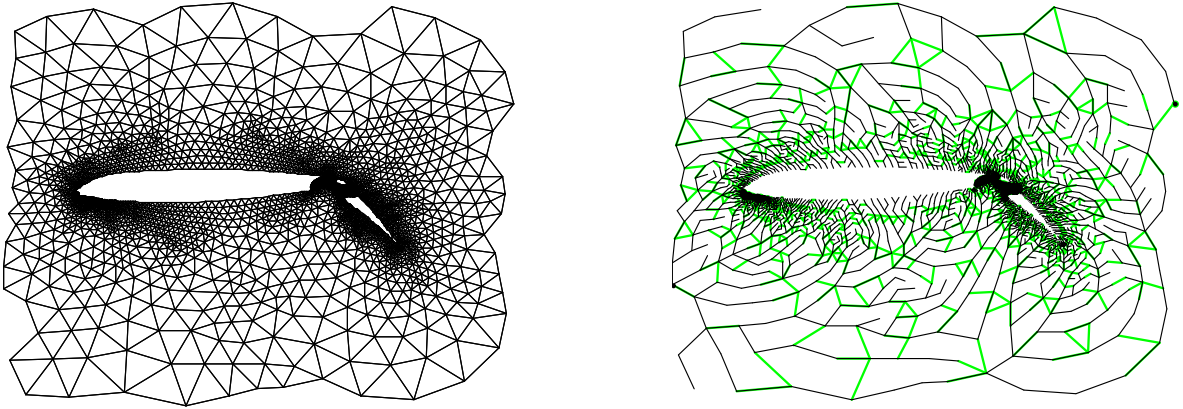


Figure 3.2. The preconditioner from Spielman and Teng (right) replaces the original graph (left) with a spanning tree (black) to which are added a few ‘important’ edges (green). The new graph, while much sparser than the original, has a similar structure as the original graph, in particular in terms of spectrum. Figure from [Kla12]

To obtain a true linear time algorithm, it is required that the preconditioner is as spectrally close as possible to the original graph while simultaneously only requiring a linear time to solve when used inside an iterative method such a preconditioned Chebyshev or conjugate gradient. Spielman and Teng’s ultra-sparsifier allows to obtain really good preconditioners but they can’t be solved in linear time. To remedy this problem, they decided to use recursion.

The solving is done in two steps. First, since the preconditioner is a tree with a linear number of off-tree edges, it most likely has several tree-like parts with many degree 1 and 2 nodes. A partial Cholesky factorisation is thus first applied on all rows and columns corresponding to degree 1 and 2 nodes because this is can be done in linear time.

Algorithm 3 PartialCholesky

Require: Graph $H = (V, E, w)$

Ensure: Graph $\hat{G} = (\hat{V}, \hat{E}, \hat{w})$, Partial factorization C , permutation matrix P .

$\hat{G} = H$

while there are nodes of degree 1 or 2 in $\hat{G}$ **do**

 Remove all degree-1 nodes from $\hat{G}$ and update the Cholesky factorization

if $\deg_{\hat{G}}(v) = 2$ and $(u_1, v), (u_2, v) \in E_{\hat{G}}$ **then**

$$w' = \frac{1}{\frac{1}{w(u_1, v)} + \frac{1}{w(u_2, v)}}$$

 Add edge (u_1, u_2) with weight w' to $\hat{G}$

 Update the Cholesky factorization

end if

end while

This step is described in algorithm 3 with the graph theoretic interpretation. It factors the system into

$$S_i = PC_i \begin{bmatrix} I & 0 \\ 0 & A_i \end{bmatrix} C_i^T P^T.$$

If A is an irreducible *SDDM*-matrix, then the result of the partial Cholesky factorization has the following properties:

Lemma 3.3.1. *Let $H, C, P = \text{PartialCholesky}(G)$, then*

1. A_i is an irreducible *SDDM*-matrix.
2. If H is planar, then so is $\hat{G}$.
3. C has at most $3n$ nonzero entries.
4. If H has $n - 1 + j$ edges, then $\hat{G}$ has at most $2j - 2$ vertices and $3j - 3$ edges.

The triangular matrix C_i is solved using backward and forward substitution. The remaining smaller system A_i preserves the properties of A (planarity, *SDDM*) and can then be solved recursively using the ST-solver. The last property is used to bound the size of the smaller system and applies directly to ultra-sparsifiers who have at most $n - 1 + hm/k$ edges.

In order for this algorithm to give an almost-linear running time, they proved the following lemma:

Lemma 3.3.2. *Let $\mathcal{C} = \{A = A_1, S_1, A_2, S_2, \dots, A_d\}$ be a chain graphs and $K = \{\kappa_1, \kappa_2, \dots, \kappa_{d-1}\}$ be a list of numbers. Denote n_i and m_i as the number of nodes and edges of A_i respectively. If*

1. $A_i \leq S_i \leq \kappa(n_i)A_i$.
2. $A_{i+1} = \text{PartialCholesky}(S_i)$.
3. $\frac{m_i}{m_{i+1}} \geq c\sqrt{\kappa(n_i)}$.
4. $\kappa_i \geq \kappa_{i+1}$.

Then the system $Ax = b$ can be solved in time $\mathcal{O}(m\sqrt{\kappa_1 \log(\frac{1}{\epsilon})})$ using the hierarchy of smaller graph as preconditioners in the iterative method used recursively.

At each level i of the recursion, solving the system in A_i using PCG with S_i as a preconditioner requires $\mathcal{O}(\sqrt{\kappa_i})$ iterations. Each of these iterations are comprised of matrix-vector multiplication with A_i as well as other simpler operations (vector-vector product, ...). This requires $\mathcal{O}(m_i + n_i)$ flops for each iteration. The main cost comes from solving the system with B_i recursively. They proved that if the hierarchy of graphs get smaller sufficiently fast (condition on m_i/m_{i+1}), then the total complexity of the recursive algorithm is asymptotically almost linear, although the hidden constant is huge.

Spielman-Teng's sparsification algorithm allows to compute an ultra-sparsifier with $\kappa = \mathcal{O}(\log^c(n))$ for any graph. Overall, this gives a total complexity of $\mathcal{O}(m \log^c(n))$ to solve Laplacian matrices. It was a theoretical breakthrough but in practice the constant c was huge ($c > 25$). They did not attempt to optimize the value of the exponent c since their goal was to prove the first ever almost linear time solver for SDD matrices. The algorithm is in fact too difficult to implement in practice, and nobody has attempted to do it, especially since simpler techniques were developed after that.

3.4 Spielman and Srivistava

Five years later, Spielman and Srivistava designed a simple sparsification algorithm [SS08]. They assign a probability p_e to each edge of the graph, and then proceed to sample q edges randomly with probability p_e and with replacement. When an edge is picked multiple time, each new copy is added as a parallel edge. They found after some experimentation that the probabilities that give the sparsifier the least amount of edge is by sampling edges based on their effective resistance.

Algorithm 4 Sample

Require: Graph $G = (V, E, w)$, values p'_e

Ensure: Graph $H = (V, E', w')$

$t = \sum_e p'_e$

$q = C_s t \log n$ (C_s is a known constant)

$p_e = p'_e / t$

$H = (V, E', w')$ with $E' = \emptyset$

for q times **do**

 Sample one $e \in E$ with probability p_e

 Add e to E' with weight $w'(e) = \frac{w(e)}{qp_e}$

end for

3.4.1 Effective resistance

To define the notion of effective resistance, it is first needed to view the graph as an electrical circuit, where each edge e represents a wire with a resistance of $\frac{1}{w(e)}$. When this circuit is subject to an external current or voltage, it will automatically adopt the flow of current that minimizes the power dissipation. Intuitively, an edge is important if there aren't easy alternate way for the current to flow through between its two endpoints. If an

important edge that connect two different clusters is removed, then the flow of current will significantly change. On the contrary, if an edge inside a cluster is removed, then the current will simply flow through its neighbouring edges instead.

Let $i_{ext}(u) \in \mathbb{R}^n$ be the vector of currents entering and leaving the vertices, let $i(e) \in \mathbb{R}^m$ be the vectors of current induced at the edges and let $v(u) \in \mathbb{R}^n$ be the vector of potential induced at the vertices. Then by Kirchhoff's law, we have

$$B^T i = i_{ext}.$$

By Ohm's law, we have

$$i = W B v.$$

Thus combining these two equations, we obtain

$$L v = i_{ext}.$$

The effective resistance R_e of an edge e is the potential difference induced across it when a unit of current is injected at one end of e and extracted at the other end of e . Let u and w be the endpoint of e , then the difference of potential between u and w is obtained by solving $L v = b_e$, with b_e being the external current injected through e :

$$b_e(x) = \begin{cases} 1 & \text{if } x = u \\ -1 & \text{if } x = w \\ 0 & \text{otherwise} \end{cases}$$

The effective resistance of e is then obtained by measuring the potential difference across u and w :

$$R(e) = v(u) - v(w) = b_e^T v = b_e L^\dagger b_e^T.$$

The construction of the preconditioner is then conceptually easy, they repeat several time the process of randomly picking an edge of the original graph to add the the preconditioner with a probability proportional to its effective resistance. This allows to add the most important edges with high probability. They proved that if the number of sample is sufficiently large, then the preconditioner and the original graph have close spectral properties.

3.4.2 Sampling algorithm

They proved the effectiveness of their algorithm using tools from matrix theory [RV07]. The method is based an random samplings.

For scalar random variables, if Y is a bounded random variable with $|Y| \leq M$, then the classical Law of Large numbers states that the empirical mean of many independent samples $\frac{1}{q} \sum_{j=1}^d Y_j$, strongly concentrates around the true mean $\mathbb{E}[Y]$. This can be quantified by the Chernoff bound as

$$\mathbb{P} \left[\left| \frac{1}{q} \sum_{i=1}^q y_i - \mathbb{E}[Y] \right| > \epsilon \right] \leq \exp \left(\frac{-q\epsilon^2}{4M^2} \right).$$

In such case, taking $q = \mathcal{O}(M^2/\epsilon^2)$ samples is enough to obtain an excellent approximation of $\mathbb{E}[Y]$ with high probability.

Rudelson then discovered that the same phenomenon applies if Y is a random $n \times n$ matrix, but with a blowup of $\mathcal{O}(\log n)$ in the number of sample needed to attain the concentration. This is captured in the theorem (from [Sri10]):

Theorem 3.4.1. *Suppose $y \in \mathbb{R}^n$ is a random vector satisfying $\mathbb{E}[yy^T] = I_n$ and $\|y\|_2 \leq M$. If $y_1, y_2, \dots, y_q$ are independent samples of y , then*

$$\mathbb{P} \left[\left\| \frac{1}{q} \sum_{i=1}^q y_i y_i^T - I_n \right\|_2 \geq \epsilon \right] \leq n \exp \left(\frac{-q\epsilon^2}{4M^2} \right).$$

Taking $q = \mathcal{O}(n \log n / \epsilon^2)$ is needed to obtain a good approximation of I . Spielman and Srivastava applied this theorem for a random vector y such that

$$y = y_e = \frac{1}{\sqrt{p_e}} w(e)^{\frac{1}{2}} (L^\dagger)^{\frac{1}{2}} b_e \quad \text{with probability } p_e = \frac{w(e)R(e)}{n-1}.$$

This random vector respects the hypothesis of the theorem:

$$\begin{aligned} \mathbb{E}[yy^T] &= \sum_e p_e \frac{1}{p_e} w(e) (L^\dagger)^{\frac{1}{2}} (b_e b_e^T) (L^\dagger)^{\frac{1}{2}} \\ &= (L^\dagger)^{\frac{1}{2}} L^\dagger (L^\dagger)^{\frac{1}{2}} \\ &= I_n. \end{aligned}$$

Furthermore, the bound on its norm is given by

$$\begin{aligned} \|y_e\|_2 &= \frac{1}{\sqrt{p_e}} \sqrt{w(e) b_e^T L^\dagger b_e} \\ &= \sqrt{\frac{n-1}{w(e)R(e)}} \sqrt{w(e)R(e)} \\ &= \sqrt{n-1}. \end{aligned}$$

They then linked these results from matrix theory with graph sparsification. First, from the result of the sample procedure, they define the diagonal matrix $S \in \mathbb{R}^{m \times m}$ such that

$$S(e, e) = \frac{\# \text{ of times } e \text{ is sampled}}{qp_e}.$$

Then, they create a sparsified matrix of L by

$$H = B^T W^{\frac{1}{2}} S W^{\frac{1}{2}} B.$$

They proved that such a sparsifier respects

$$\left(1 - \left\| \frac{1}{q} \sum_{i=1}^q y_i y_i^T - I_n \right\|_2 \right) L \leq H \leq \left(1 + \left\| \frac{1}{q} \sum_{i=1}^q y_i y_i^T - I_n \right\|_2 \right) L.$$

To obtain a $\frac{1+\epsilon}{1-\epsilon}$ -approximation of A , which is equivalent to have

$$(1 - \epsilon)L \leq H \leq (1 + \epsilon)L.$$

It is enough to take $q = \mathcal{O}(n \log n / \epsilon^2)$ to obtain such a preconditioner with high probability.

This procedure allows the efficient construction of an excellent preconditioner. However, calculating the effective resistance of the edges require solving Laplacian system, which can thus not be used to create fast Laplacian solvers. Nevertheless, the theoretical ideas developed in this work were extremely useful for the works that followed.

3.5 Koutis-Miller-Peng's first (KMP)-solver

Sampling edges based on probabilities proportional to their effective resistance is inefficient in practice. To use the procedure described by Spielman and Srivastava, it is then needed to sample them based on probabilities that approximate the effective resistance and that are easy to compute. This is exactly what Koutis, Miller and Peng proved in [KMP14]. They showed that randomly sampling edges based on probability that is proportional to their stretch with right to a sparse subgraph instead of their effective resistance allows to create good preconditioner with high probability.

Let $R_G(e)$ denote the effective resistance of e in G and $R_T(e)$ the effective resistance of e in a spanning tree T of the original graph. Since there are no cycle in trees, there only exists a unique path for every pair of nodes u and v . Let this path be composed of $e_1, \dots, e_k$. When a unit of current is injected in u and extracted in w , the flow has to go through them entirely. These different edges thus act as resistors in series. We have

$$R_T(e) = \sum_{i=1}^k \frac{1}{w(e_i)}.$$

We then have

$$st_T(e) = \frac{\sum_{i=1}^k \frac{1}{w(e_i)}}{\frac{1}{w(e)}} = w(e)R_T(e).$$

By Rayleigh's monotonicity law [Spi10]:

$$R_G(e) \geq R_T(e).$$

because T is a subgraph of G . Finally, it gives

$$st_T(e) \geq w(e)R_T(e).$$

The stretch of an edge is therefore an overestimation of the effective resistance but are easy to compute and suffice to obtain good bounds.

Theorem 3.4.1 can then be applied random vector y such that

$$y = y_e = \frac{1}{\sqrt{p_e}} w(e)^{\frac{1}{2}} (L^\dagger)^{\frac{1}{2}} b_e \quad \text{with probability } p_e = \frac{st_T(e)}{st_T(G)}.$$

The only modification appears on the bound on its norm:

$$\begin{aligned} \|y_e\|_2 &= \frac{1}{\sqrt{p_e}} \sqrt{w(e) b_e^T L^\dagger b_e} \\ &= \sqrt{\frac{st_T(G)}{st_T(e)}} \sqrt{w(e) R_e} \\ &\leq \sqrt{st_T(G)}. \end{aligned}$$

The value of M is not equal to $\sqrt{n-1}$ anymore but is bounded by the square root of the total stretch of G with T .

To create their preconditioner, they sample edges to add back to the tree based on their stretch of the tree. To obtain a 3-approximation of L , it is enough to take $C_s M^2 \log n / \epsilon^2$,

with $M^2 = st_T(G)$ and $\epsilon = 1/2$. The consequence of using the approximation leads to a higher number of off-tree edge sampled since it now depends on m instead of n . It is therefore crucial to use a spanning tree with the lowest possible total stretch to lower the number of off-tree edge as much possible. However, even with excellent LSST algorithm, the number of sampled edges is in general too big to create almost linear-time solver.

To remedy this, they control the number of off-tree edge sample by first creating a graph $\tilde{A}$ out of A by scaling every tree edge by a factor κ . The stretch of every off-tree edge is therefore divided by κ , which with the best LSST algorithm gives a total stretch of $\mathcal{O}(\frac{m \log n}{\kappa})$. Using support theory [BH01], one can prove that $\tilde{A}$ is a κ -approximation of A . They then apply the sampling algorithm on $\tilde{A}$ to obtain a sparse graph H which 3-approximate $\tilde{A}$. By transitivity, we have

$$A \preceq \tilde{A} \preceq H \preceq 3\tilde{A} \preceq 3\kappa A,$$

which means that H is a 3κ -approximate of A . The number of off-tree samples is given in expectation by $\mathcal{O}(\frac{m \log^2 n}{\kappa})$. They called their result an incremental sparsifier instead of an ultra-sparsifier.

Algorithm 5 IncrementalSparsify

Require: Graph $G = (V, E, w)$, real κ

Ensure: Graph H

$T = \text{LSST}(G)$

$T' = \kappa T$

$G' = G - T + T'$

for $e \in E$ **do**

$p'_e = st_{T'}(e)$

end for

$H = \text{Sample}(G', p'_e)$

Hence, the KMP algorithm allows to obtain ultra-sparsifier such that

- $A_i \preceq B_i \preceq 3\kappa(n_i)B_i$.
- B_i has $n_i - 1 + \mathcal{O}(\frac{m_i \log^2 n_i}{\kappa(n_i)})$ edges.

To satisfy Spielman and Teng's condition in Lemma 3.3.2 to obtain an almost linear time solver, it is needed to take $\kappa(n_i) = \mathcal{O}(\log^4 n_i)$.

After generating the hierarchy of approximations of the original graph, they use the same algorithm as the ST solver. That is, they use an iterative method, in which they solve the system by first applying a partial Cholesky factorisation on columns and rows corresponding to nodes of degree 1 and 2 and then recursively solve the remaining system. The total complexity of the first KMP solver is $\mathcal{O}(m \log^2 n)$.

3.6 Other related works

Even more efficient theoretical solvers have been developed since then, such as the solver of Cohen et al. that has a complexity of $\mathcal{O}(m \log^{\frac{1}{2}}(n))$ [CKM⁺14].

The most recent results, published in 2021, claims a solver for Laplacian matrices with a run-time of $\mathcal{O}(m(\log \log(n))^c)$ [JS21] ! These solvers are also based on low-stretch trees and are purely theoretical.

Another simpler solver was developed by Kelner et al. [KOSZ13]. It is not a combinatorial preconditioner and instead uses different techniques. However it still also uses low-stretch spanning trees. This solver has a complexity of $\mathcal{O}(m \log^2(n))$ has been implemented and analysed by several researchers [Dew18; HLMW15]. Interesting results have been obtained, such as confirming that the running time indeed scales nearly-linearly as proved theoretically. However, the constant factors make the practical run-time on reasonable inputs slower than basic solvers with higher asymptotic complexity. Moreover, it was also shown that using lower stretch spanning trees do indeed tend to reduce the running time of the solver.

Chapter 4

KMP solver analysis

This chapter will analyse the second version of Koutis, Miller and Peng’s solver, which they created one year after their first work [KMP11]. It is a variant of their first solver that was improved after a more throughout study of the properties of the tree, which allowed it to attain a total running time of $\mathcal{O}(m \log n)$. This version had never been implemented prior to this work.

This section is structured as follows: In a first part, some important theorem will be proven to understand the KMP solver. Then, an in-depth analysis of the practical performance of this solver will be the core of this section. Each part of the algorithm will be analyzed practically and compared to the theoretical guarantees such as to understand the strengths and weaknesses of the solver.

The performances of the solver will then be compared to that of the Conjugate Gradient and PCG with the LSST as a preconditioner on the test graphs. The goal is to estimate the practical asymptotic running time of the KMP solver and to verify if they indeed take almost-linear time.

4.1 KMP solver improvements

Koutis, Miller and Peng brought three major modifications which allowed to improve their first solver. The first modification is that they modify the sampling algorithm by only sampling off-tree edges instead of all edges. All tree edges sampled are removed from the sparsified graph and the original tree is added back. This was done to prove one of their theorems (4.1.3).

The second modification came from the realization that it was not necessary to generate a new LSST from scratch at each level of the recursion and could instead generate each successive tree from the previous one. This allows to keep the sampling based on the same spanning tree at every level of the recursion.

They proved this for Laplacian matrices as follows: At each step during the partial Cholesky factorization of the graph H (Algorithm 3), the edges that are eliminated in H are also eliminated in its spanning tree T_H , and the resulting modifications (new edge added with new weight when factoring a node of degree 2) are also applied in the tree. They proved that it naturally generates a spanning tree for the factored graph that also has low-stretch. This step is given in Algorithm 6 and is called the GreedyElimination step. It factors

nodes of degree 1 and 2 in H , as well as generating the next low stretch spanning tree of H .

The claim is proved in Lemma 4.1.1.

Algorithm 6 GreedyElimination

Require: Graph $H = (V, E, w)$, Spanning tree $T_H = (V, E_{T_H}, w)$ of H

Ensure: Graph $G = (\hat{V}, \hat{E}, \hat{w})$, Spanning tree $T_G(\hat{V}, E_{T_G}, \hat{w})$ of H

$G = H$

$E_{T_G} = E_{T_H}$

while there are nodes of degree 1 or 2 in G **do**

 Remove all degree-1 nodes from G and its corresponding edge in T_G .

if $\deg_G(u) = 2$ and $(u, v_1), (u, v_2) \in E_{\hat{G}}$ **then**

 Remove edges (u, v_1) and (u, v_2) from G

 Add edge (u_1, u_2) and update d_{u_1}, d_{u_2} with weight $\frac{1}{1/w(u, v_1) + 1/w(u, v_2)}$

if (u, v_1) and $(u, v_2) \in E_{T_G}$ **then**

 Add (u_1, u_2) to T_G

end if

 Remove (u, v_1) and (u, v_2) from T_G

end if

end while

Lemma 4.1.1. *Let $(G, T_G) = \text{GreedyElimination}(H, T_H)$. the output T_G is a spanning tree of G and*

$$|st_{T_G}(G)| \leq |st_{T_H}(H)|.$$

Proof. The claim is proved by inductively showing that it holds at every iteration of the loop. Let $(G_i, T_{G,i})$ be the result of the algorithm after the elimination of the i^{th} node. The claim obviously hold for the base case $(G_1, T_{G,1}) = (H, T_H)$.

When a degree-1 node gets factored, its only edge is necessarily in the tree. The removal of this edge doesn't affect the stretch of any other edges. Therefore, if the claim holds for $(G_i, T_{G,i})$, then it will still hold for $(G_{i+1}, T_{G,i+1})$.

When a degree-2 node gets factored, one or both of (u, v_1) and (u, v_2) must be in $T_{G,i}$. Suppose first that only (u, v_1) is the tree. Then, the only non-tree edge removed is the edge (u, v_1) . The contribution of this edge to the total stretch is equal to

$$s_1 = w(u, v_2) R_{T_{G,i}}(u, v_2).$$

After the factorization, the contribution of the new edge (v_1, v_2) to the total stretch is equal to

$$s_2 = \frac{1}{1/w(u, v_1) + 1/w(u, v_2)} R_{T_{G,i+1}}(v_1, v_2).$$

We have

$$\frac{1}{1/w(u, v_1) + 1/w(u, v_2)} < \frac{1}{1/w(u, v_2)} = w(u, v_2).$$

Moreover,

$$R_{T_{G,i+1}}(v_1, v_2) = R_{T_{G,i}}(v_1, v_2) = R_{T_{G,i}}(u, v_2) - \frac{1}{w(u, v_1)} < R_{T_{G,i}}(u, v_2).$$

No other edges stretch is effected by the factorization. Therefore, the claim follows from

$$st_{T_{G,i+1}}(G_{i+1}) - st_{T_{G,i}}(G_i) = s_2 - s_1 < 0.$$

Consider now the case where both (u, v_1) and (u, v_2) are in $T_{G,i}$. Let $T_{G,i}[e]$ denote the unique path in the tree between the endpoints of e in $T_{G,i}$. The factorization will create a new tree edge, which will affect the stretch of every edge e such that $T_{G,i}[e]$ passed through u . We have

$$s_1(e) = w(e)R_{T_{G,i}}(e) = w(e) \left(w(u, v_1)^{-1} + w(u, v_2)^{-1} + \sum_{f \in \tilde{T}_{G,i}[e]} w(f)^{-1} \right).$$

In the new tree, the path in the tree goes through the new edge (v_1, v_2) instead of going through $(v_1, u), (u, v_2)$. Let $\tilde{T}_{G,i}[e] = T_{G,i}[e] - \{(u, v_1), (u, v_2)\}$. The stretch is then given by

$$s_2(e) = w(e)R_{T_{G,i+1}}(e) = w(e) \left(\left(\frac{1}{1/w(u, v_1) + 1/w(u, v_2)} \right) + w(v_1, v_2) \right)^{-1} + \sum_{f \in \tilde{T}_{G,i+1}[e]} w(f)^{-1}.$$

Then, it follows that

$$\begin{aligned} st_{T_{G,i+1}}(e) - st_{T_{G,i}}(e) &= s_2(e) - s_1(e). \\ &= \left(\frac{1}{1/w(u, v_1) + 1/w(u, v_2)} + w(v_1, v_2) \right)^{-1} - w(u, v_1)^{-1} - w(u, v_2)^{-1} < 0. \end{aligned}$$

In such case, individual edge stretch only decreases. The total stretch will also decrease and the claim holds. $\square$

The second improvement came from a better understanding of the sampling algorithm. They proved that when the graph is viewed as a multi-graph (such a graph can have multiple parallel edges connecting the same endpoints), then proved every sampled edge l has the same stretch. This is proven in the following Lemma:

Lemma 4.1.2. *Let $H = \text{IncrementalSparsify}(G, \kappa)$. Then, for each off-tree edges l sampled from the $C_s t \log n$ samples, we have*

$$st_{T_H}(l) = \frac{1}{C_s \log n}.$$

Proof. Let $T' = \kappa T$ and $G' = G - T + \kappa T$. The probability of any non-tree edge e to be sampled is equal to

$$p'_e = \frac{st_{T'}(e)}{st_{T'}(G')} = \frac{w_e R_{T'}(e)}{st_{T'}(G')}.$$

The new edge l then is added to the graph with weight

$$\begin{aligned} w_l &= \frac{w_e}{p'_e} \frac{1}{q} = \frac{w_e}{\frac{w_e R_{T'}(e)}{st_{T'}(G')}} \frac{1}{C_s st_{T'}(G') \log n} \\ &= \frac{1}{C_s R_{T'}(e) \log n}. \end{aligned}$$

The stretch of l with respect to T' is then given by

$$st_{T'}(l) = w_l R_{T'}(e) = \frac{1}{C_s \log n}.$$

□

This is crucial to control the depth of the recursion. Indeed, the number of samples depends on the stretch, which can now be bounded with Lemma 4.1.2. It is so important that they define the notion of spine-heavy graphs, which are graphs that possess a spanning tree that give a total stretch

$$st_T(G) = \mathcal{O}\left(\frac{m}{\log n}\right).$$

With this, they design the following algorithm to generate all the graphs needed in the chain:

Algorithm 7 BuildChain

Require: Graph $G = (V, E, w)$, Spanning tree T of G , constant κ_c

Ensure: Chain of graphs $\mathcal{C} = \{G_1 = G, H_1, G_2, H_2, \dots, G_d\}$

$$\kappa_1 = \frac{st_T(G) - n + 1}{m - n + 1} C_s \log n$$

$$T_1 = \kappa_1 T_G$$

$$H_1 = G - T_G + T_1$$

$$G_2 = H_1$$

$$\mathcal{C} = \emptyset; i = 2$$

while $st_{T_i}(G_i) - n + 1 > \kappa_c$ **do**

$$H_i = \text{IncrementalSparsify}(G_i, \kappa_c) \text{ (Algorithm 5)}$$

$$\mathcal{C} = \mathcal{C} \cup \{G_i, H_i\}$$

$$G_{i+1}, T_{i+1} = \text{GreedyElimination}(H_i, T_i) \text{ (Algorithm 6)}$$

$$i = i + 1$$

end while

$$H_d = \text{PartialCholesky}(T_d) \text{ (Algorithm 3)}$$

$$\mathcal{C} = \mathcal{C} \cup \{G_d, H_d\}$$

Let us first define the following short notation for the total stretch of the off-tree edges only:

$$\tilde{st}_T(G) = st_T(G) - n + 1.$$

In algorithm 7, this first graph is scaled by

$$\kappa_1 = \tilde{st}_T(G) / \frac{m - (n - 1)}{C_s \log n}.$$

to become a spine heavy graph. In doing so, the stretch of every off-tree edge is divided by κ_1 . Indeed, the weight of any off-tree edge e hasn't changed whereas the sum of the inverse of the weight in the path connecting the endpoints of e is κ_1 times smaller. The stretch of every off-tree edges

$$st_{T'}(e) = w(e) \sum_i \frac{1}{w'(e_i)} = w(e) \sum_i \frac{1}{\kappa_1 w(e_i)}.$$

is thus divided by κ_1 .

As a result, the graphs resulting from this algorithm respect the following Lemma:

Lemma 4.1.3. *Let l_1 be the number of edges in G_1 and l_i be the number of off-tree samples for $i > 1$. Then, the chain of graphs from Algorithm 7 has the following properties:*

1. $\mathbb{E}[l_{i+1}] \leq \frac{l_i}{\kappa_c}$.
2. $\tilde{st}_{T_{i+1}(G_{i+1})} \leq \frac{l_i}{C_s \log n_i} = \kappa_c \tilde{st}_{T_i}(G_i)$.

Proof. The proof is done by induction. For the base case $i = 1$, the value of κ_1 is set such that it respects property (2). Then, property (1) holds since the number of samples in IncrementalSparsify is given by $q = C_s \tilde{st}_{T_i}(G_i) \log n_i$, and the expected number of off-tree edges is given by

$$\begin{aligned} \mathbb{E}[l_{i+1}] &= q \frac{\tilde{st}_{T'_i}(G'_i)}{\tilde{st}_{T'_i}(G'_i)} = C_s \tilde{st}_{T'_i}(G'_i) \log n_{i+1} \frac{\tilde{st}_{T'_i}(G'_i)}{\tilde{st}_{T'_i}(G'_i)} \\ &= C_s \log n_{i+1} \frac{l_i}{\kappa_c C_s \log n_i} \\ &\leq \frac{l_i}{\kappa_c} \end{aligned}$$

For the induction, we have

$$\tilde{st}_{T_{i+1}}(G_{i+1}) \leq \tilde{st}_{T_{H_i}}(H_i) \leq \frac{l_i}{C_s \log n_i}.$$

by Lemma 4.1.2 and theorem 4.1.1 for each inequality respectively. Thus Property (2) is satisfied. Property 1 then holds using the same development. $\square$

By correctly choosing κ_c (discussed later), it was proven that recursively solving a Laplacian system using PCG is done in time

$$\mathcal{O}(m\sqrt{\kappa_1}) = \mathcal{O}\left(m\sqrt{\log^{c+1}(n)}\right).$$

The recursive algorithm of the preconditioned conjugate gradient with the hierarchy of graphs is given in Algorithm 8.

Algorithm 8 Recursive Preconditioned Conjugate Gradient - RPCG

Require: Chain of graphs $\mathcal{C} = \{G_1, H_1, G_2, H_2, \dots, G_d, H_d\}$

Require: Partial Cholesky triangular matrices $\{C_1, P_1, C_2, P_2, \dots, C_d, P_d\}$

Require: level (i) of recursion, right hand side b

Ensure: $x = L_{G_i}^{-1}b$

$$x_0^{(i)} = 0$$

$$r_0^{(i)} = b$$

for $n = 0, 1, \dots$, until convergence **do**

$$s_n = \begin{bmatrix} s_{n,0} \\ s_{n,1} \end{bmatrix} = C_i^{-1} P_i^{-1} b$$

if $i \neq d$ **then**

$$s_{n,1} = \text{RPCG}(i+1, s_{n,1})$$

end if

$$z_n = P_i^{-T} C_i^{-T} s_n$$

$$\beta_{n-1} = \frac{z_n^T r_n}{z_{n-1}^T r_{n-1}} \quad (\beta_{-1} = 0)$$

$$p_n = z_n + \beta_{n-1} p_{n-1}$$

$$\alpha_n = \frac{z_n^T r_n}{p_n^T L_{G_i} p_n}$$

$$x_{n+1} = x_n + \alpha_n p_n$$

$$r_{n+1} = r_n - \alpha_n L_{G_i} p_n$$

end for

This recursive algorithm can be converted into an equivalent iterative one. This is an important step because a recursive function is often of higher memory and space complexity than its iterative counterpart. Indeed, a recursive function is often less efficient in practice because each recursive call will add a new frame to the call stack. An iterative function will in contrast only use a single frame stack, which is more memory efficient. The performance of the iterative function is faster since it avoids the overhead of creating and destroying stack frame at each recursive call.

This is an optimisation purely in term of running time. The analysis of the recursive solver is done in term of FLOPS, which doesn't change between the recursive and the iterative versions.

4.2 Test graphs properties

In this chapter, the four test graphs described in section 2.1.1 will be extensively tested to obtain a good picture of the practical performance of the KMP solver.

In this chapter, Kruskal's maximum weight spanning trees are used instead of true LSST. It was done so because it was shown in [PKBK14; HLMW15] that in practice, they perform better than the algorithms with the proven theoretical bounds (Table 3.1). This hypothesis will be tested again in the next chapter.

The number of iteration is theoretically bounded by $\mathcal{O}(\log^{0.5c+0.5})$, with c being the exponent of the logarithmic term in the stretch value. This means that the number of iteration will not be bounded by $\mathcal{O}(m \log n)$ using the theoretical best bounds on LSST but will instead depend on the values of c obtained with Kruskal's algorithm. This section will thus test if they respect these bounds instead. All weights are randomly drawn from a uniform distribution $U(0, 10]$.

The number of iteration of the conjugate gradient depends on the condition number of the matrices, whereas KMP's solver depends on the logarithmic term of the stretch. Figure 4.1 shows the evolution of these two quantities for the four test graphs.

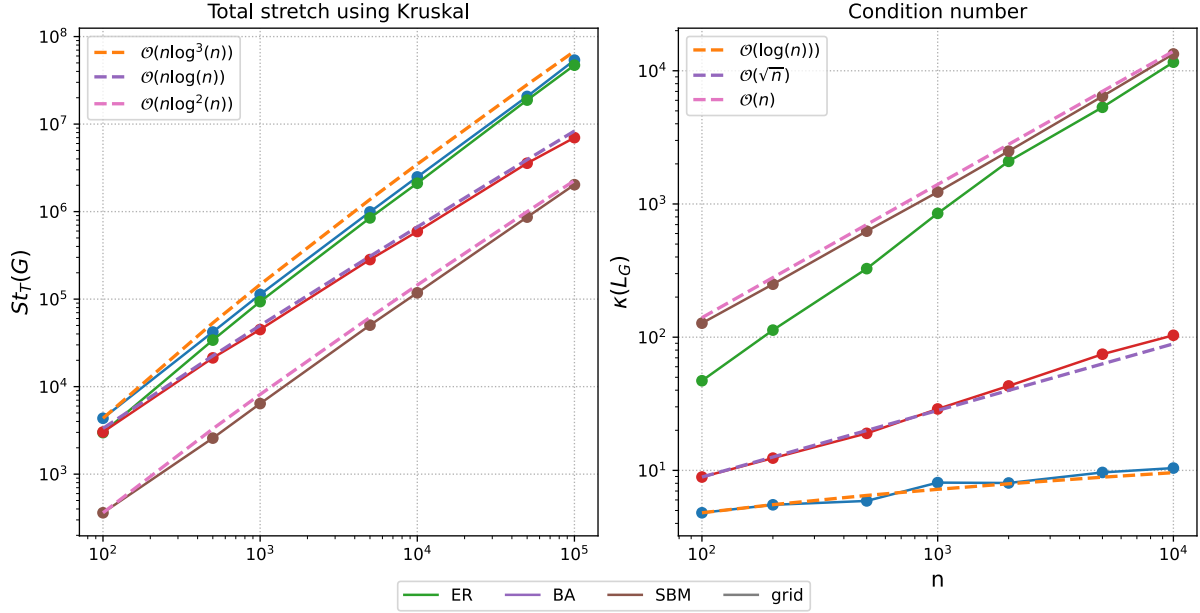


Figure 4.1. Condition number and total stretch of Kruskal's maximum weight spanning tree. The dashed lines represent the fitted asymptotic behavior of the practical curves.

From this Figure we observe that the ER graphs will perform poorly with KMP compared to CG since they have low condition number (good for CG) and high stretch (bad for KMP).

On the contrary, the grid graphs are the most promising test candidate for the KMP against CG solver as these have condition number in $\mathcal{O}(n)$ and average stretch $\mathcal{O}(\log^2 n)$. The BA and SBM graphs will have performance in between the other two.

4.3 Sampling procedure

This section will analyze the result of the sampling procedure in practice. The goal is to determine whether the number of samples in the theory is a tight bound or not in practice. Both the sampling based on effective resistance and stretch will be tested to visualize the impact of the approximation. Then, the hidden constant will be computed based on the results to obtain the exact number of samples needed in practice.

4.3.1 Sampling based on effective resistance

In a first part, the optimal sampling algorithm of Spielman and Srivastava (see section 4) will be tested.

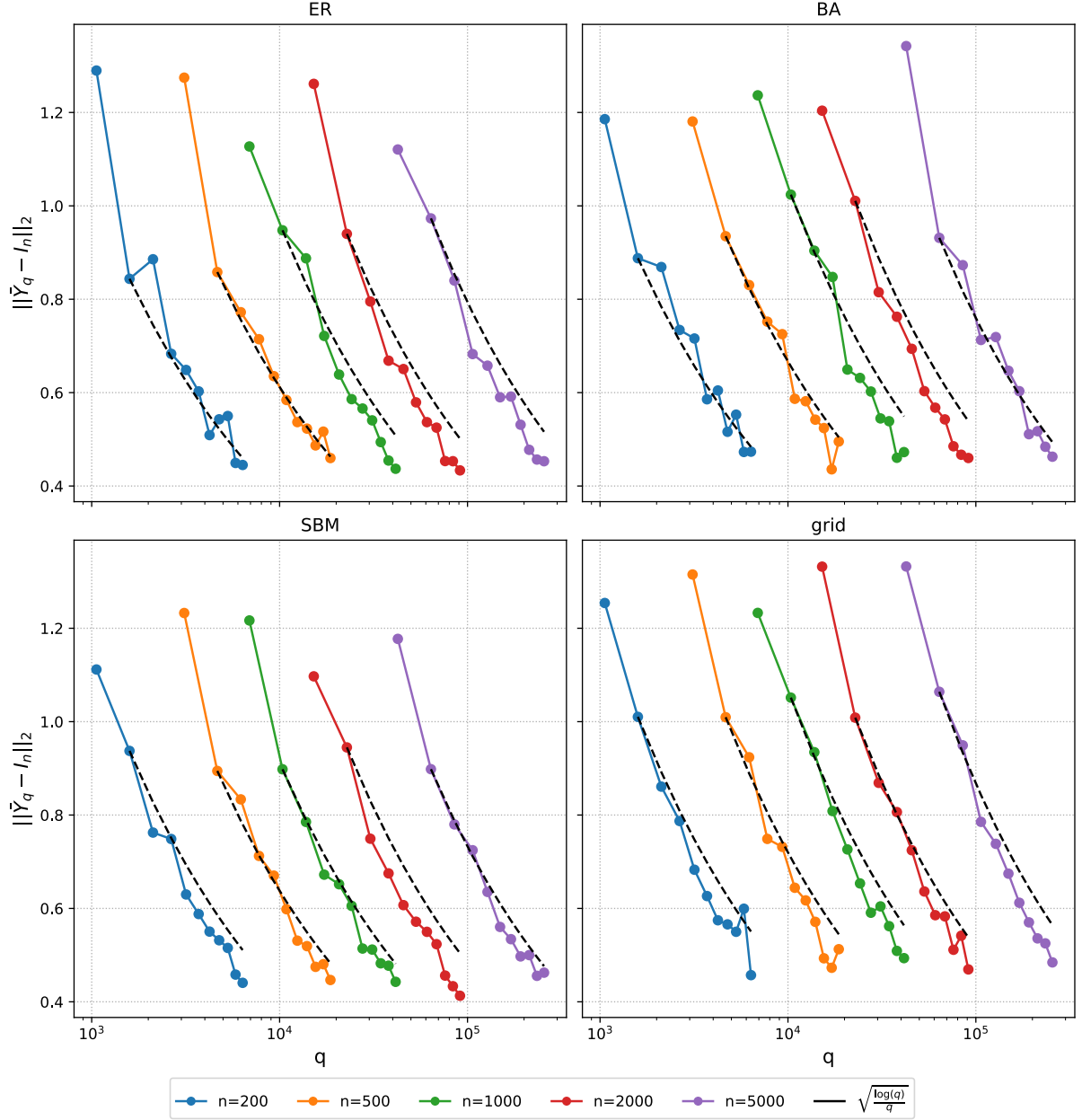


Figure 4.2. Practical results of the sampling algorithm for the test graphs based on the effective resistances. $\bar{Y}_q = \frac{1}{q} \sum_{i=1}^q y_i y_i^T$ is a realisation of the sampling algorithm, with y_i being the vector defined in section 3.4. The dashed lines are the fitted asymptotic behavior of the curves

In their method, the number of samples needed is determined by Rudelson and Vershynin's theorem:

Theorem 4.3.1. *let $\mathbf{p}$ be a probability distribution over $\Omega \in \mathbb{R}^n$ such that $\sup_{y \in \Omega} \|y\|_2 \leq M$ and $\|\mathbb{E}_{\mathbf{p}}[yy^T]\| \leq 1\|_2$. Let $y_1, \dots, y_q$ be independent samples drawn from $\mathbf{p}$. Then*

$$\mathbb{E} \left[\left\| \frac{1}{q} \sum_{i=1}^q y_i y_i^T - \mathbb{E}[yy^T] \right\| \right] \leq \min \left(CM \sqrt{\frac{\log q}{q}}, 1 \right).$$

where C is an absolute constant.

This bound is known to be generally optimal. This first practical experiment is there to confirm whether this bound is also optimal for the distribution of the test matrices or if it is an overestimation, in which case the algorithm could perform better in practice than originally thought. The results for the four test graphs for various sizes is shown in Figure 4.2. We observe that the practical results closely follow the theoretical bound of theorem 4.3.1 for every test graphs.

From these results, the asymptotic number of samples needed to obtain a 3-approximation of the test graphs can be extracted. This is shown in Figure 4.3. We see that taking $5n \log n$ samples is enough to obtain 3-approximate for every case.

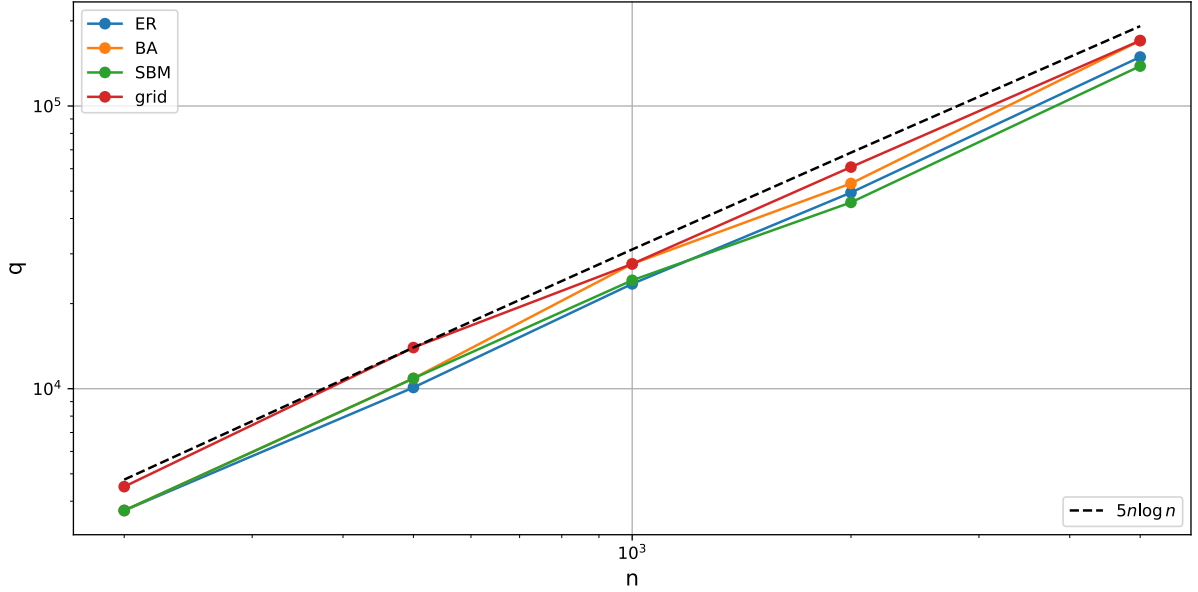


Figure 4.3. Number of samples needed to obtain a $\kappa(L_G, L_H) \leq 3$ for the test graphs by sampling based on effective resistance. The dashed line reveals the hidden constant in the number of sample q (as defined in Algorithm 4) needed in practice to obtain the preconditioner.

The result for these plots were obtained by sampling based on the effective resistance, which were obtained by solving Laplacian systems. It is therefore unpractical but could prove to be useful if efficient methods to estimate those values arise in future researches.

4.3.2 Sampling based on stretch

Using the stretch instead of the effective resistance to sample will require more samples to obtain the same quality of approximation. When applied on the spine-heavy version of the graph, the sampling algorithm still closely follows the bound from Theorem 4.3.1, as shown in Figure 4.4.

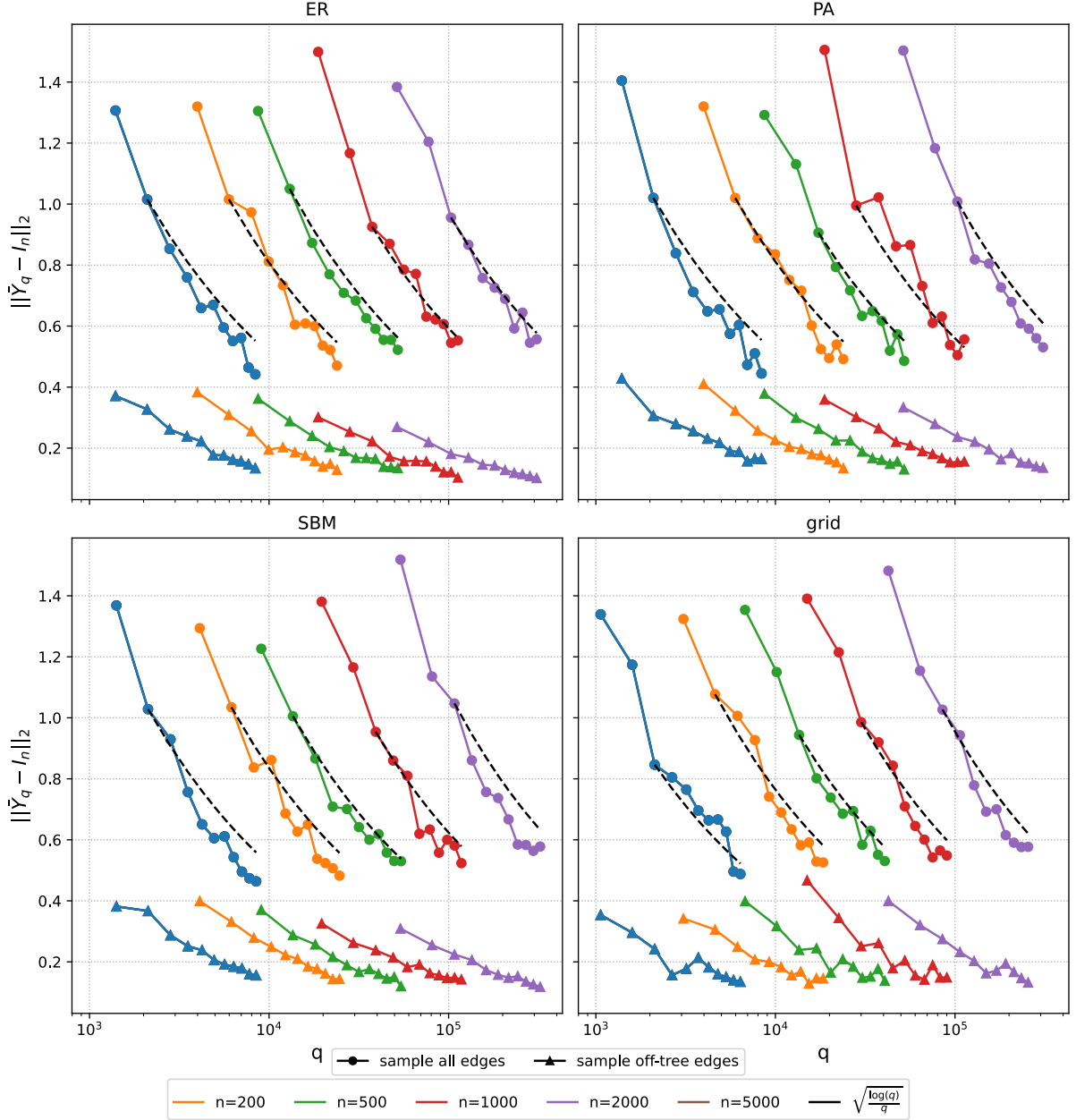


Figure 4.4. Practical results of the sampling algorithm for the test graphs.

The exact number of samples needed to obtain a 3-approximate of the spine heavy graphs now depends on the stretch of the graphs instead of the number of nodes (shown in Figure 4.5).

Since the spine-heavy graphs are themselves a κ_1 -approximate of their original graphs, it means by transitivity that the sparsified spine-heavy graphs are $3\kappa_1$ -approximate of the original graphs.

In the second version of the solver, they modified the sampling algorithm to only sample off-tree edges and keep the tree edges intact. This was done in order to prove Lemma 4.1.1, which allows to generate a hierarchy of tree from their predecessor in linear time instead of finding a new tree at each level of the chain. It was also done to prove Lemma 4.1.2 which was crucial to upgrade their solver.

However, it was not done to increase the quality of the preconditioners. Replacing a sampled

edge by its original edge is not guaranteed to improve the quality of the approximations. Nevertheless, the practical results shows that doing so significantly improves the quality of the approximations. As a result, the number of samples needed in practice can be lower than predicted in theory with this modified sampling algorithm.

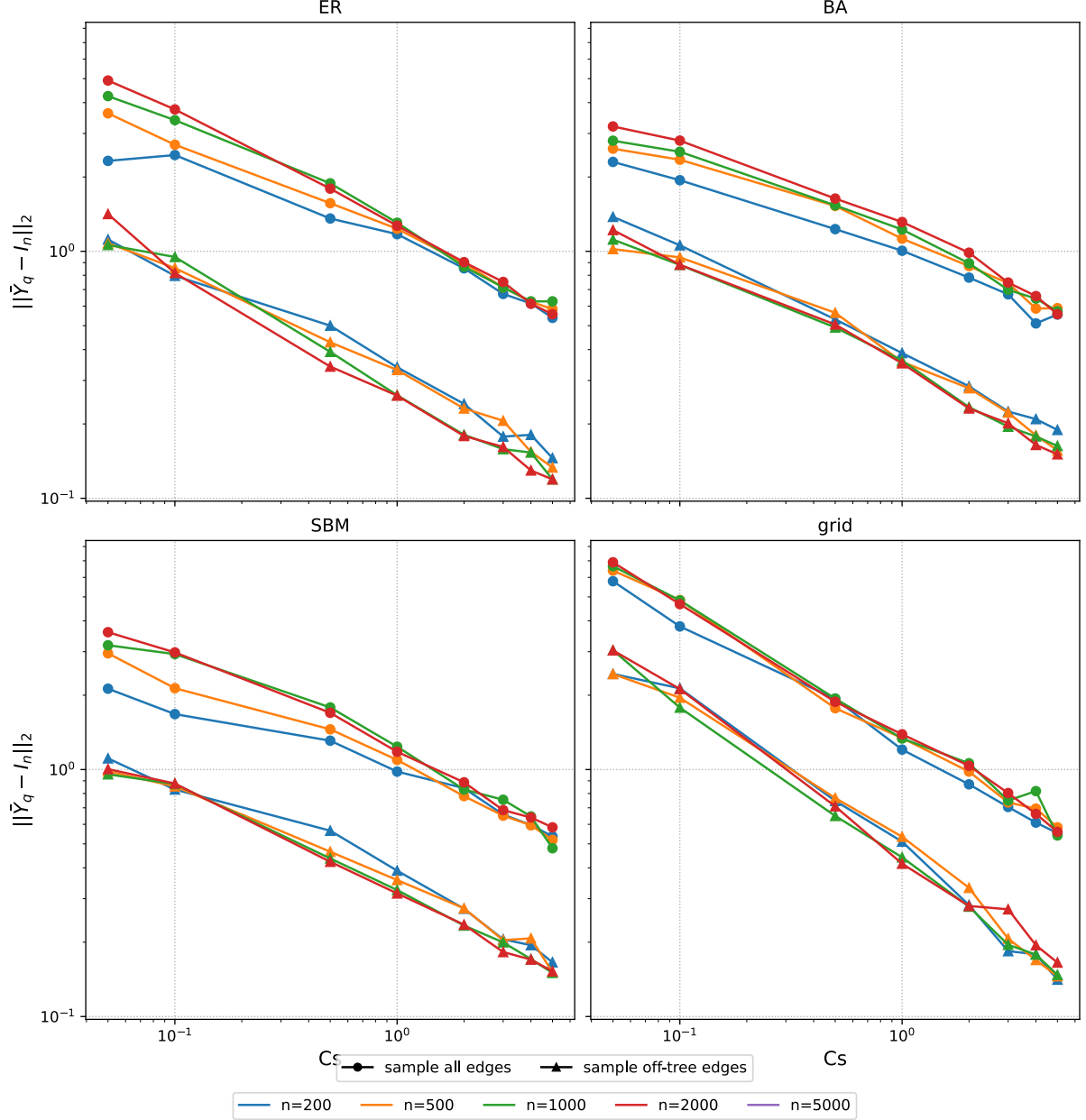


Figure 4.5. Number of samples needed to obtain a 3-approximate of spine-heavy graphs. C_s is the constant in the number of sample $q = C_s st_{T'}(G') \log n$. $\bar{Y}_q$ is a realization of the sample algorithm with y_i as defined in section 3.5.

Figure 4.5 shows the result of the sampling algorithm applied on spine-heavy graphs for various value of C_s . The first KMP sampling algorithm require $C_s = 5$ to obtain 3κ -approximates of the graphs while $C_s = 0.5$ suffices with the second version. The value of κ_1 is thus 10 times smaller which is quite significant because the running time evolves as $\sqrt{\kappa_1}$.

4.4 Spine-heavy graphs

Spine-heavy graphs are a special kind of graph because they can be solved in linear time (at least in theory) with the KMP solver. Indeed, spine-heavy graphs have by definition total-stretch $\mathcal{O}(\frac{m}{\log n})$ and therefore don't need to be scaled down by a first factor κ_1 . Instead, the run time will when used inside the recursive conjugate gradient will be bounded by

$$\mathcal{O}\left(\sqrt{\kappa_c} \log \frac{1}{\epsilon_c}\right).$$

with κ_c being a constant. This follows from the fact that the graphs obtained by buildChain (Algorithm 7) respect:

1. $\kappa(G_i, H_i) \leq 3\kappa_c$.
2. $\mathbb{E}[\frac{l_i}{l_{i+1}}] = \kappa_c$.

As a result, each level needs $\frac{\sqrt{3\kappa_c}}{2} \log \frac{2}{\epsilon_c}$ iterations to solve the subsystems in the KMP solver. At each iteration of each level i , if n_i and m_i are the number of nodes and edges in G_i respectively, then the number of FLOPS is in order $\mathcal{O}(m_i + n_i)$. Indeed, each iteration of CG is comprised of vector-vector scalar product ($\mathcal{O}(n_i)$ FLOPS), and of a single matrix-vector product $\mathcal{O}(m_i)$. With PCG, the triangular matrices from the partial Cholesky factorization of nodes of degree 1 and 2 have at most $3n_i$ non-zero elements. Thus, the forward and backward resolution of these requires $\mathcal{O}(n_i)$ FLOPS each. At the last level, the factorization of the tree creates a triangular matrix with exactly $2n_d$ elements.

Therefore, at every other level i , the total number of FLOPS is given by

$$\left(\frac{\sqrt{3\kappa_c}}{2} \log \frac{2}{\epsilon_c}\right)^i \cdot \mathcal{O}(m_i + n_i).$$

The accumulated total amount of work is thus given by

$$\sum_{i=1}^d \left(\frac{\sqrt{3\kappa_c}}{2} \log \frac{2}{\epsilon_c}\right)^i \mathcal{O}\left(\frac{m_1}{\kappa_c^{i-1}}\right) = \kappa_c \sum_{i=1}^d \left(\frac{\sqrt{3}}{2\sqrt{\kappa_c}} \log \frac{2}{\epsilon_c}\right)^i \mathcal{O}(m_1).$$

In order to obtain a total work in $\mathcal{O}(m_1)$, it is enough to impose

$$\frac{\sqrt{3}}{2\sqrt{\kappa_c}} \log \frac{2}{\epsilon_c} < 1.$$

In this work, the values is fixed at 0.5

$$\Rightarrow \kappa_c = 3 \log^2 \frac{2}{\epsilon_c}.$$

The maximum depth d of the chain is attained when the off-tree stretch can no longer be reduced by κ_c . When this happens, the value of the off-tree stretch are so small that they are not sampled anymore:

$$\begin{aligned} \tilde{st}_{T_i}(G_i) &\leq \kappa_c \\ \frac{m_1 - n_1 + 1}{C_s \log n_1} \frac{1}{\kappa_c^d} &\leq \kappa_c \end{aligned}$$

$$\Rightarrow d = \log_{\kappa_c} \left(\frac{m_1 - n_1 + 1}{C_s \log n_1} \right). \quad (4.1)$$

By fixing $\epsilon_c = 10^{-1}$ (this choice will be motivated in a later section), the solver performs as shown in Figure 4.6:

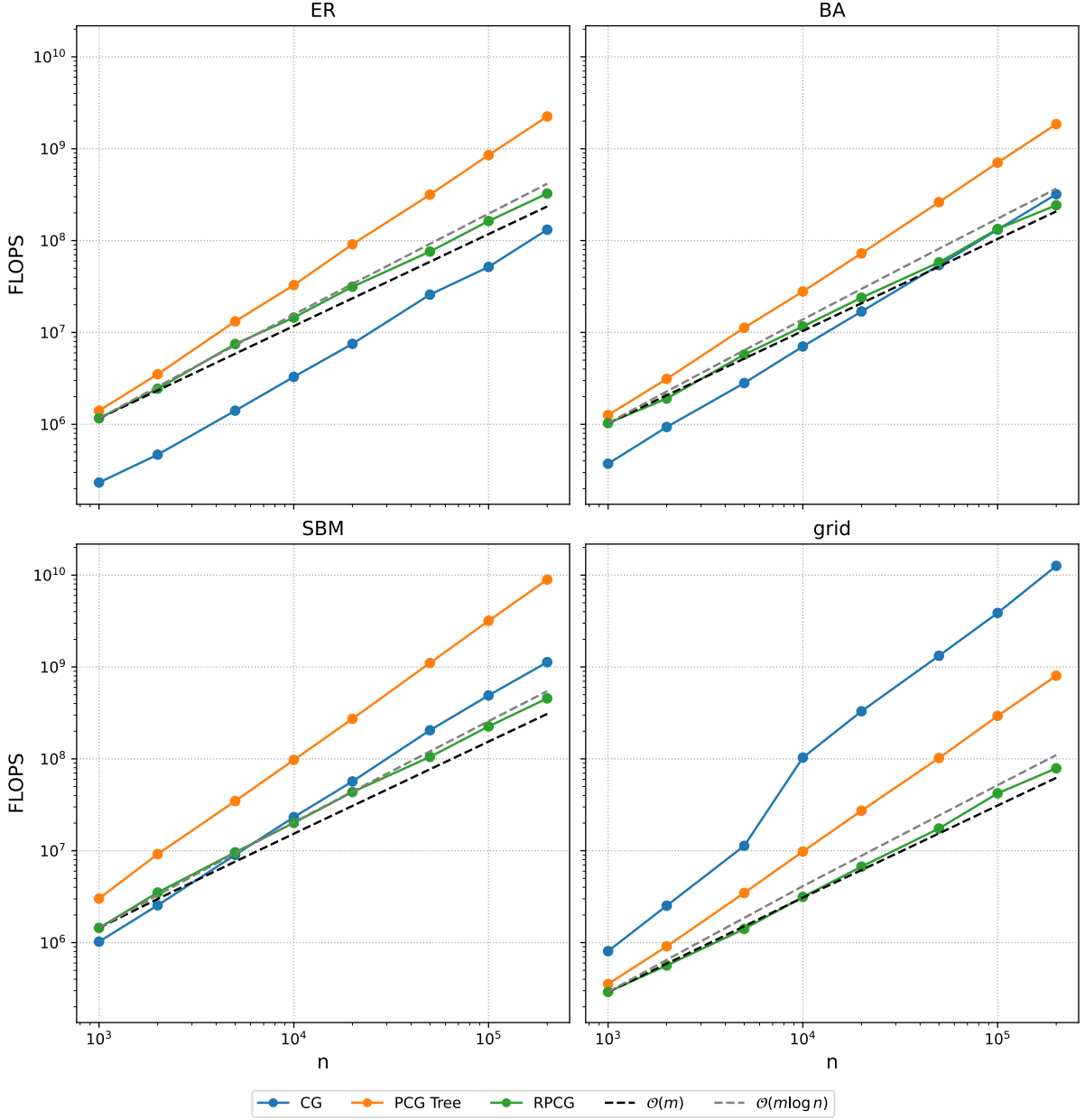


Figure 4.6. Total amount of work to solve spine-heavy graphs. The dashed line are asymptotic curves used to obtain the complexity of the KMP solver.

Figure 4.6 shows the exact number of FLOPS performed by the algorithms when applied on spine-heavy graphs. First of all, it shows that the solver indeed improves upon the simple solver of using CG with the spanning tree as a preconditioner. Then, we see that the running time is almost linear for such graphs. The asymptotic run-time is between $\mathcal{O}(m)$ and $\mathcal{O}(m \log n)$. This is explained because the condition 4.1 gets harder and harder to respect as the size increases. If the value of 4.1 tends to 1, then the accumulated total

amount work tends to

$$\kappa_c \sum_{i=1}^d \left(\frac{\sqrt{3}}{2\sqrt{\kappa_c}} \log \frac{2}{\epsilon_c} \right)^i \mathcal{O}(m_1) \longrightarrow \log_{\kappa_c}(m) \mathcal{O}(m).$$

Nevertheless, we can confirm that the run-time is indeed almost linear for the spine heavy graphs. This means that the KMP solver does not depend on the condition number as predicted in the theory, whereas the classical CG does.

The grids graphs have the worst condition number and is beaten by KMP for every graph size. For the BA and SBM graphs, CG is initially better but gets beaten by KMP after a certain size. Finally, the ER graphs have the lowest condition numbers and are thus well solved by CG. We still see that KMP will eventually require less FLOPS.

4.5 Improvement (top level)

General Laplacian matrices whose corresponding graph is not spine-heavy are solved by the algorithm created by KMP using PCG with the Laplacian matrix H_1 of a spine-heavy graph as a preconditioner. This is obtained by multiplying every tree-edge of the original graph G by $\kappa_1 = \frac{st_T(G)-n+1}{m-n+1} \frac{1}{C_s \log n}$. This preconditioner is solved in time $\mathcal{O}(\sqrt{\kappa_c} m \log \frac{1}{\epsilon_c})$, which is linear as confirmed empirically in the previous section. Therefore, their solver has a run-time of

$$\mathcal{O} \left(\sqrt{\kappa_1} \left(m + \sqrt{\kappa_c} m \log \frac{1}{\epsilon_c} \right) \log \frac{1}{\epsilon} \right).$$

However, there exists a way to improve this solver with a small modification. Indeed, at the two top level, we have $\kappa(G = G_1, H_1) = \kappa_1$ and $\kappa(G_2 = H_1, H_2) = 3\kappa_c$. Since the first preconditioner H_1 is just a scaled version of G_1 , they have the same dimensions. Therefore, one could also directly use H_2 as a preconditioner for G_1 to obtain

$$\kappa(G_1, H_2) \leq \kappa(G_1, H_1) \kappa(H_1, H_2) = 3\kappa_1 \kappa_c.$$

By using this modified solver, the run-time would be proportional to

$$\mathcal{O} \left(\sqrt{\kappa_1 \kappa_c} \left(m + \sqrt{\kappa_c} \frac{m}{\kappa_c} \log \frac{1}{\epsilon_c} \right) \log \frac{1}{\epsilon} \right).$$

They are asymptotically the same without the constant $\log \frac{1}{\epsilon_c}$ and $\frac{1}{\epsilon}$. However, the hidden constant will potentially be better depending on the values of κ_c , ϵ_c and ϵ .

Figure 4.7 shows that using H_2 directly as a preconditioner is consistently better than the original solver of KMP.

Both have the same asymptotic behaviors but the modified version has a lower hidden constant for every test graphs. This version will therefore be the one used to test the solver on the general graphs.

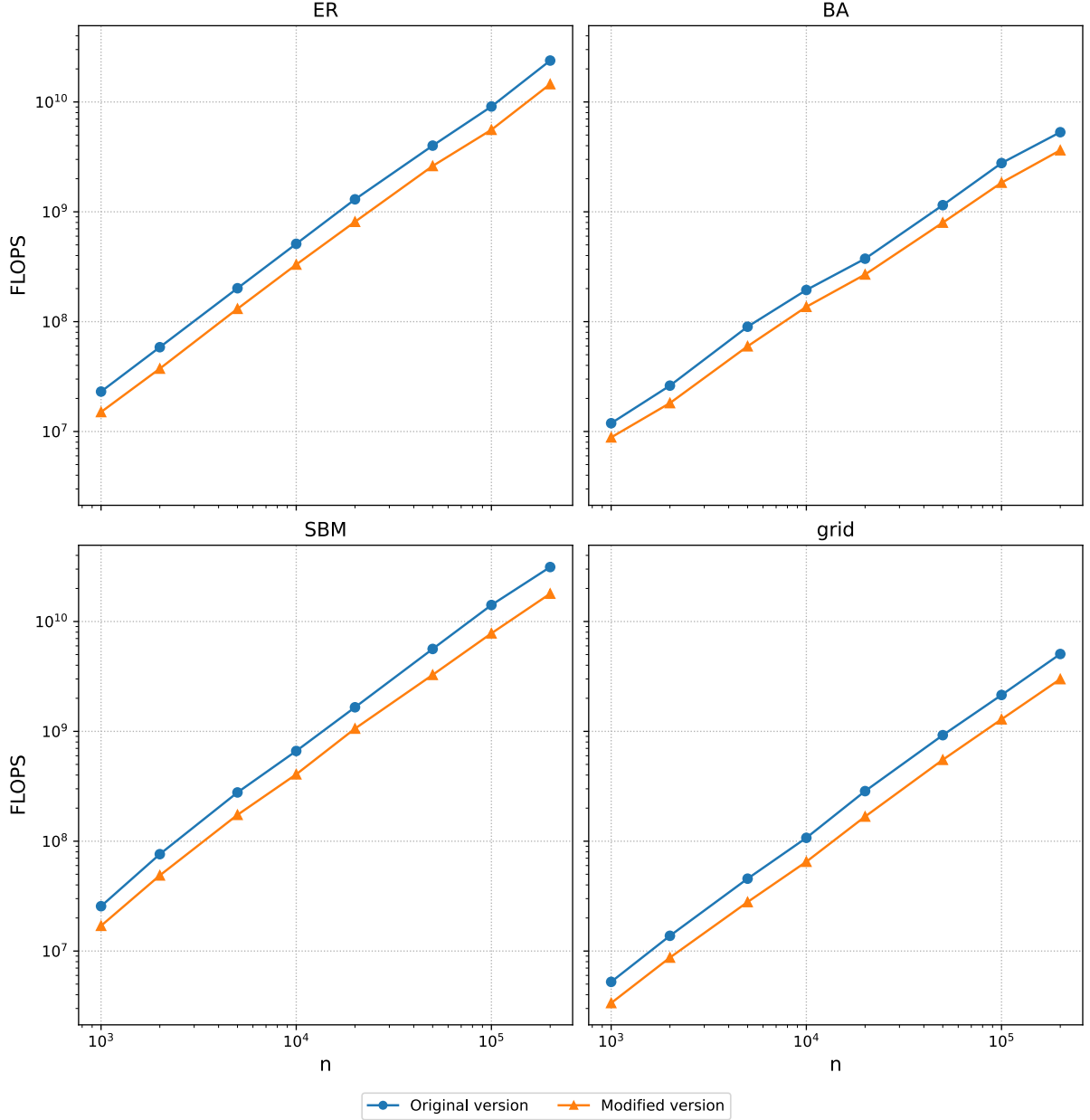


Figure 4.7. Effect of combining the two first level in the chain of graphs.

This test was only performed with $\epsilon_c = 10^{-1}$ but remains true for the other values of ϵ_c . This will in be shown in section 4.7.

4.6 General graphs

The solver is now ready to be tested on the original test graphs. Figure 4.8 shows the result of solving the Laplacian matrices with CG, PCG with the LSST as a preconditioner and the modified KMP solver. The goal of this section is to confirm whether the asymptotic bound respect the theoretical one, namely

$$\mathcal{O}\left(m \log^{0.5c+0.5} n\right). \quad (4.2)$$

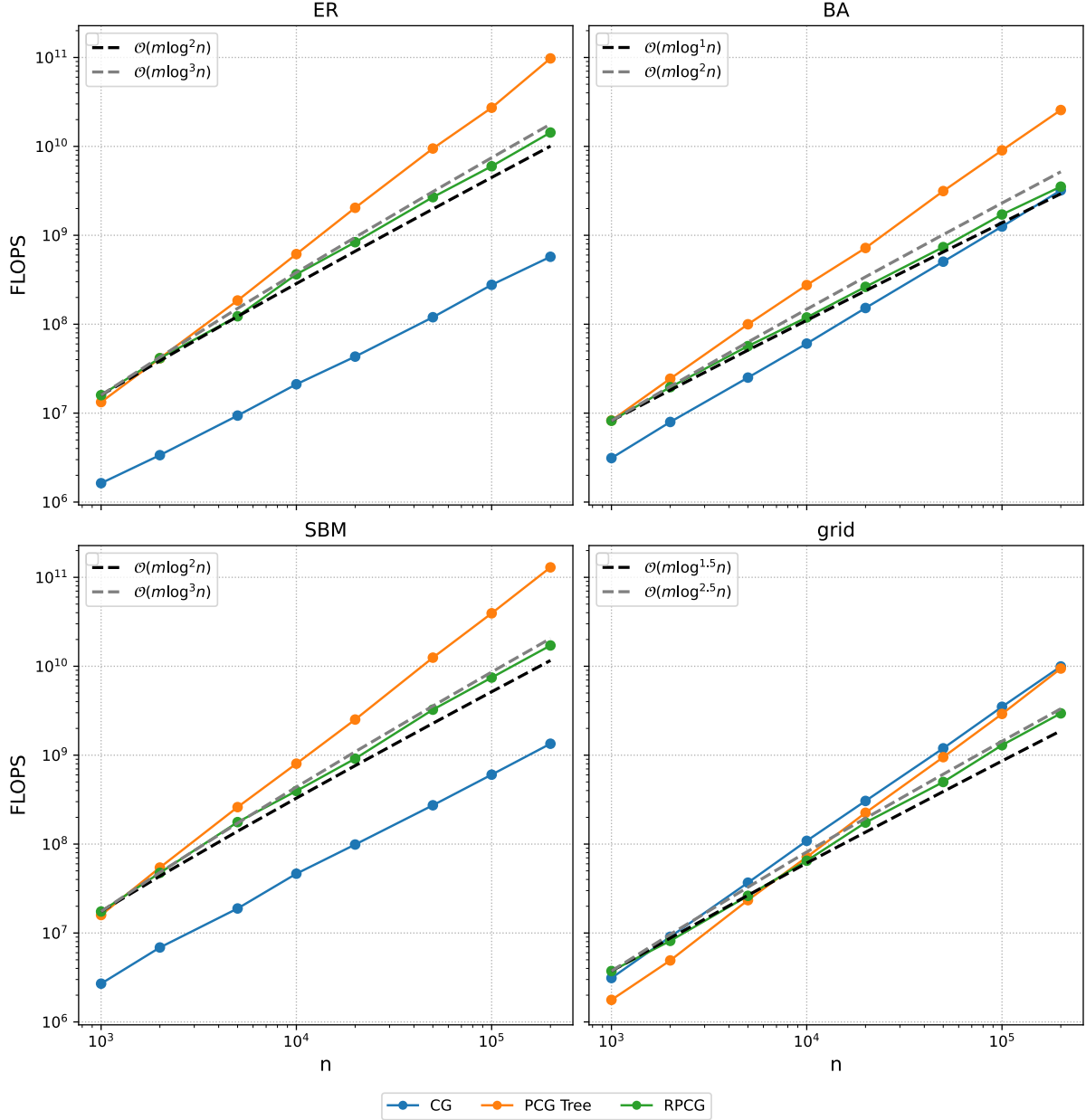


Figure 4.8. Total amount of work to solve spine-heavy graphs.

For the ER graphs, the CG is better than KMP as predicted since the condition number was $\mathcal{O}(\log n)$. The Kruskal tree had stretch $\mathcal{O}(\log^3 n)$ and could thus at best lead to a complexity of $\mathcal{O}(m \log^2 n)$. The practical result is close to the theoretical bound.

For the BA graphs, which had stretch $\mathcal{O}(\log^2 n)$, the predicted asymptotic run-time is $\mathcal{O}(m \log^{1.5} n)$. This is perfectly respected in practice. As the size increases, the solver eventually gets better than CG.

For the SBM graphs, they were supposed to be a good candidate for the KMP solver against CG since they had condition number in $\mathcal{O}(n)$ and total stretch $\mathcal{O}(\log^3 n)$. However, in practice the CG performs extremely well. This is a good reminder that the bound on the condition number is not necessarily tight. The number of iterations actually depends on the distribution of the eigenvalues. As for the KMP solver, the bound is close to the theoretical one.

Finally, the grid graphs were indeed the best candidates for the KMP solver against CG. They had condition number $\mathcal{O}(n)$, which leads to a maximum number of operations of $\mathcal{O}(n^{1.5})$. This is indeed the complexity of CG in practice. On the other hand, the grid graphs had good stretch ($\mathcal{O}(\log^2 n)$), which leads to a theoretical bounds of $\mathcal{O}(m \log^{1.5} n)$. In practice, the run-time has an additional log factor. Nevertheless, the KMP solver is here in fact clearly better than CG.

For all the graphs, the practical performance lies between the theoretical limit and the latter multiplied by a logarithmic factor. We can thus confirm that the run-time of the KMP solver is independent of the condition number of the graph and is instead dictated by the logarithmic term in the stretch. This additional term is certainly the result of imperfect arithmetic operation, which will be analyzed more in details in the next section. As for PCG with the spanning tree as a preconditioner, it should in theory lead to a complexity of

$$\mathcal{O}(m^{4/3} \log^{c/3} n).$$

In practice, it is the worst performing solver for every test graphs apart from the grid graphs. It should therefore never be used without sampling edges back into the tree.

4.7 Error accumulation

One crucial question one might have with this solver is how does the recursion affect the performances. The proof in the papers were done with the assumption of perfect arithmetic operation, which is obviously not the case. But in theory, the recursion lead to no problems at all. The goal of this section will be to understand the effect of the recursion.

In practice, the solution obtained in by the recursive solver is not the true solution that one would obtain with a direct solver. Instead, the solution has a relative error of ϵ_c compared to the error of the initial guess of the solution. In this work, the starting point was always the zero vector, so the obtained solution of the subsystem $L_{G_i} x_i = b_i$ respects:

$$\frac{\|b_i - L_{G_i} x_i\|_2}{\|b_i\|_2} \leq \epsilon_c.$$

The value ϵ_1 is fixed by the precision the user wants. However, the quality of the solutions of the subproblems is in theory arbitrary. Depending on its value, it may have a significant effect on the overall performance of the solver.

Figure 4.9 shows the evolution of the number of iterations needed at the top level for the KMP solver and the modified version for various values of ϵ_c . In this test, the value of κ_c was fixed at the value obtained with $\epsilon_c = 10^{-1}$. The other levels in the hierarchy do not matter here, the interest of this first test is to see the number of iteration needed at the top level based on the quality of the solution of the preconditioned system.

The analysis is done solely based on the grid graphs instead of every graphs since they all respect behave similarly (Complexity within the theoretical bounds of 4.2).

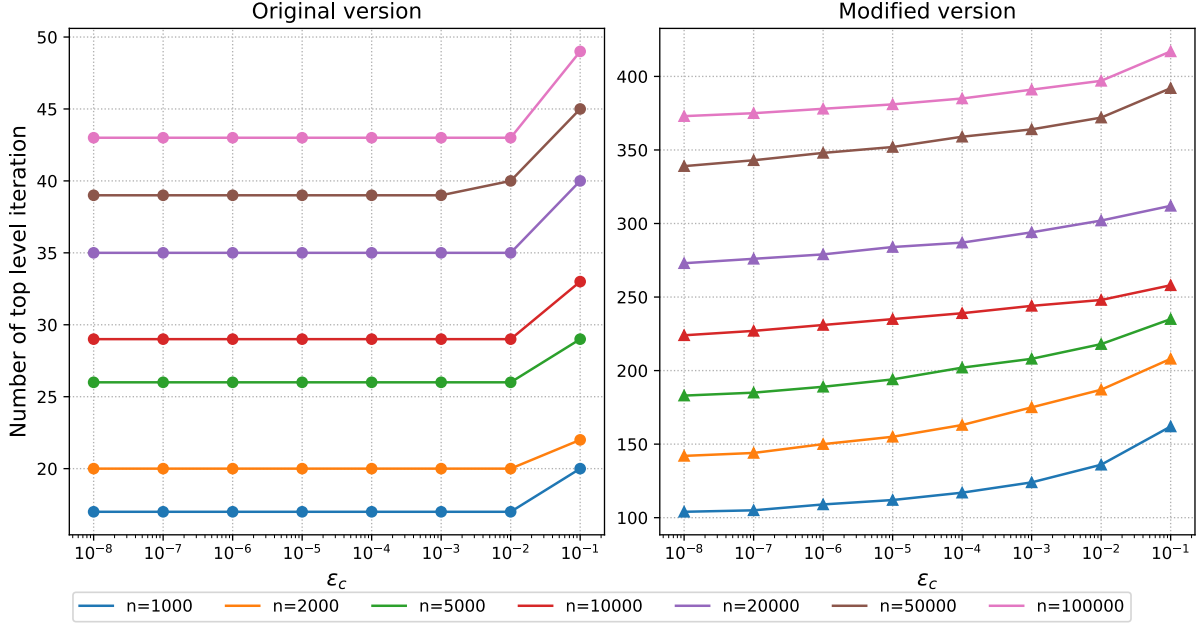


Figure 4.9. Number of iterations needed at the top level to solve the system for various quality of the solution to the preconditioner.

The two versions behave quite differently. The original KMP solver is extremely robust. The number of iterations needed does not depend on the quality of the solution of the subproblems apart for the case $\epsilon_c = 10^{-1}$, where we observe a sudden increase for every graph size. Since the run-time of solving the subproblems increases with value of ϵ_c , the best choice for the original solver is to take either $\epsilon_c = 10^{-2}$ or 10^{-1} depending on the difference in run-time needed to solve at such precision. Indeed, the small difference in the number of iteration at the top level might not be worth it if the increase in the cumulative time needed to solve all the subproblems is bigger than the time saved by the reduced number of top level iteration.

For the modified version, the number of iterations gradually increases as the quality of the solution of the subproblems decreases. The difference in behavior likely comes from the fact that H_1 is a κ_1 -approximate of G_1 , whereas H_2 is a $3\kappa_1\kappa_c$ -approximate. The modified preconditioner is a worse approximate and is therefore less robust. Once again, the choice is dictated by whether the net time gained by the reduced number of iteration out-weights the increased time needed to solve the subproblems. Nevertheless, we see that the value of ϵ_c has an impact on the run-time. It does not simply depends on $\mathcal{O}(\sqrt{\kappa_1\kappa_2})$ and instead has a dependency with ϵ_c .

This explains the additional logarithmic factor that was present in the previous section.

To decide which ϵ_c to choose, the number of flops needed to solve the problems will be computed for various values of ϵ_c .

In this second test, the values of κ_c are now adapted for each ϵ_c to have a chain of graph that respect condition 4.1.

$$\kappa_c = 3 \log^2 \frac{2}{\epsilon_c}.$$

One problem that arises is that this value increases fast as ϵ_c decreases. When it becomes too big compared to the problem size, then the off-tree edges have a really small value compared to the scaled tree-edges. As a result, almost no off-tree edge will be sampled and the preconditioner will thus be the spanning tree, which was shown to be bad in practice. Only the highest value $\epsilon_c = 10^{-1}, 10^{-2}, 10^{-3}$ are thus even worth considering.

Figure 4.10 shows the number of FLOPS needed to solve the system for these values of ϵ_c .

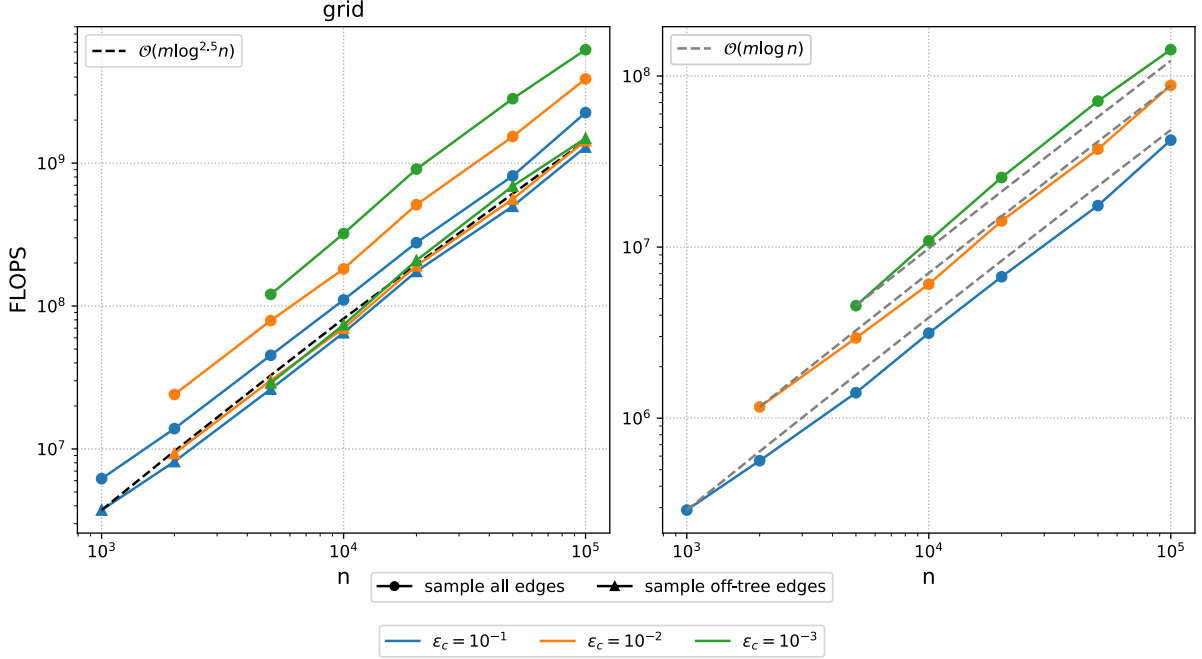


Figure 4.10. Effect of ϵ_c on the total FLOPS needed to solve the problems. The dashed lines are asymptotic curves used for comparison.

We observe that increasing the precision of ϵ_c leads to a higher number of FLOPS for both versions of the solver.

This means that solving the subproblems requires increasingly more time as ϵ_c decreases. This is clearly captured by the right figure. Solving the spine-heavy graph should in theory be done in $\mathcal{O}(m)$. However, as ϵ_c decreases, the required time increases, until it surpasses even $\mathcal{O}(m \log n)$ for $\epsilon_c = 10^{-3}$. This is due to the fact that the condition 4.1 gets harder and harder to satisfy as both the size n and the precision ϵ_c increases. This is more than likely due to the fact that the preconditioners aren't 3-approximate of the spine-heavy tree anymore.

We also observe from the left figure that the gap between the two solvers gets wider and wider as the value ϵ_c decreases.

From the first test, the original solver might have seemed like the better choice since it was more robust in term of the number of iterations needed at the top layer. What really happens is that the problem arises from solving of the spine-heavy graph, which gets worse and worse as ϵ_c decreases. The modified solver is less affected since the recursion begin by solving at the second layer of the hierarchy instead of the first. The small increase in the number of iterations at the top level is not the highest price to pay.

4.8 Graphs with lognormal weight distribution

To end this section, the solver will be tested on the same graph with weights that are randomly chosen from the distribution

$$w \sim \text{Lognormal}(0, 3).$$

This choice is motivated to get edges with high a wide range of values. A few edges will have extremely high and low values.

We expect the following to happen: the few extremely heavy edges will almost certainly all be chosen the Kruskal tree (they are few, so the odds of three extremely high edges creating a cycle is not so probable). Likewise, the very small edges will almost certainly not be in the tree. The average tree edge will thus likely be exponentially heavier than the off-tree ones. As a result, the total stretch would then have a low value and would therefore be well suited to be solved with the KMP solver.

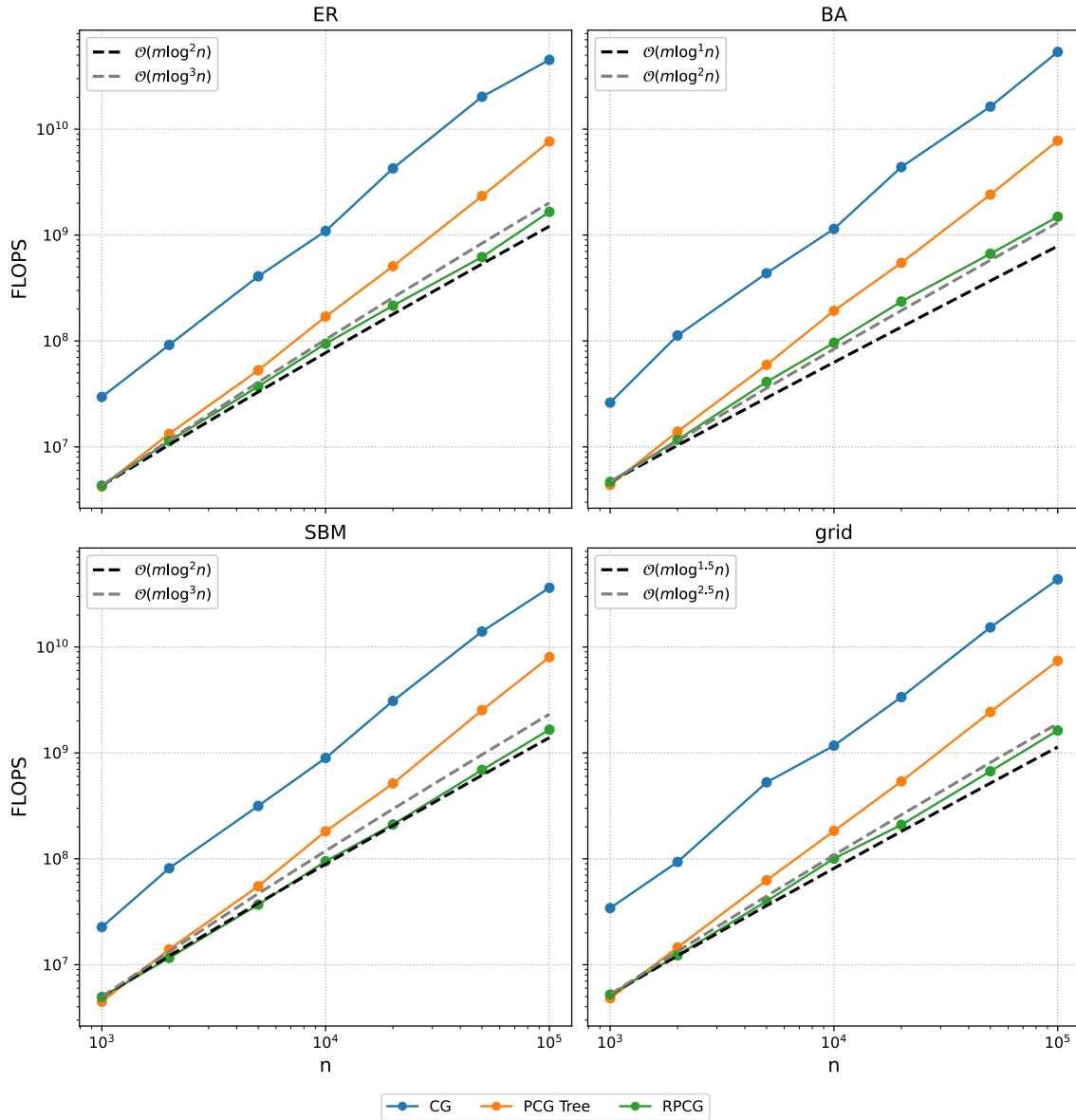


Figure 4.11. Effect of combining the two first level in the chain of graphs.

Figure 4.11 shows the result of the solver on these graphs.

We observe that the hidden constant is indeed lower than in the case without the lognormal weight distribution (see Figure 4.8).

Moreover, the condition number is highly degraded because of the huge discrepancy in the weights. The CG now performs very poorly as a result for every test graphs and is beaten by the KMP solver. This confirms that the KMP solver is completely independent of the condition number of the matrix.

This thus constitutes a family of graphs that are guaranteed (at least in practice based on the four test graphs) to be solved faster with KMP than with CG.

All the test graphs (ER, BA, SBM) were "stylised" graphs, ie they were specifically created to capture qualitative features of real-life graphs. Nevertheless, the results obtained here are general and would apply to any real-life graphs with small and large weights that could be modelled by a lognormal distribution. The KMP solver would then be well adapted to solve such systems.

Chapter 5

Low-stretch spanning tree

The low-stretch spanning tree is a crucial component of the KMP solver since the square root of the average stretch appears on the asymptotic running time

$$\mathcal{O}\left(m\sqrt{st_T(T)\log n}\right).$$

Papp showed experimentally in [PKBK14] that the various algorithms with theoretical bounds (3.1) do not yield better stretch than simply using Kruskal's spanning tree. This was then independently confirmed in [HLMW15]. As a result, Kruskal's spanning trees were used in this work instead of trees generated by LSST algorithm.

This section will test this hypothesis again for completeness since the LSST is the most important part of the solver. Surprisingly, it will be shown that it actually generates better tree than Kruskal's algorithm for every test graphs except the grids. The impact of using such tree will then be evaluated.

Finally, the solver will be tested on a finite difference problem which lead to a Laplacian matrix in a 2d-grid graph. The optimal low-stretch spanning tree is known for these so the solver will be tested with the best possible LSST.

5.1 Low-stretch spanning tree algorithm

The algorithm of Elkin, Emek, Spielman and Teng (see [EEST05]) was available at [Bha20]. I used this with some small modifications for this section. The algorithm gives a theoretical total stretch of

$$\mathcal{O}(m\log^2\log\log n).$$

Without going into the details, the algorithm works by recursively applying a star-decomposition of the graph: the nodes are separated into several sets, one of which is called the central set and is connected to others with an edge called a bridge. These are added to the spanning tree, and the algorithm is then recursively applied on each smaller sets. The radius of the graph induced by the star-decomposition allows to prove the theoretical bounds.

The results of this algorithm when applied on the four test graphs are shown in Figure 5.2.

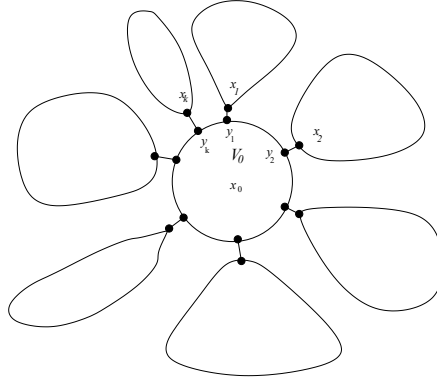


Figure 5.1. Illustration of a star-decomposition of a graph. Figure taken from [EEST05].

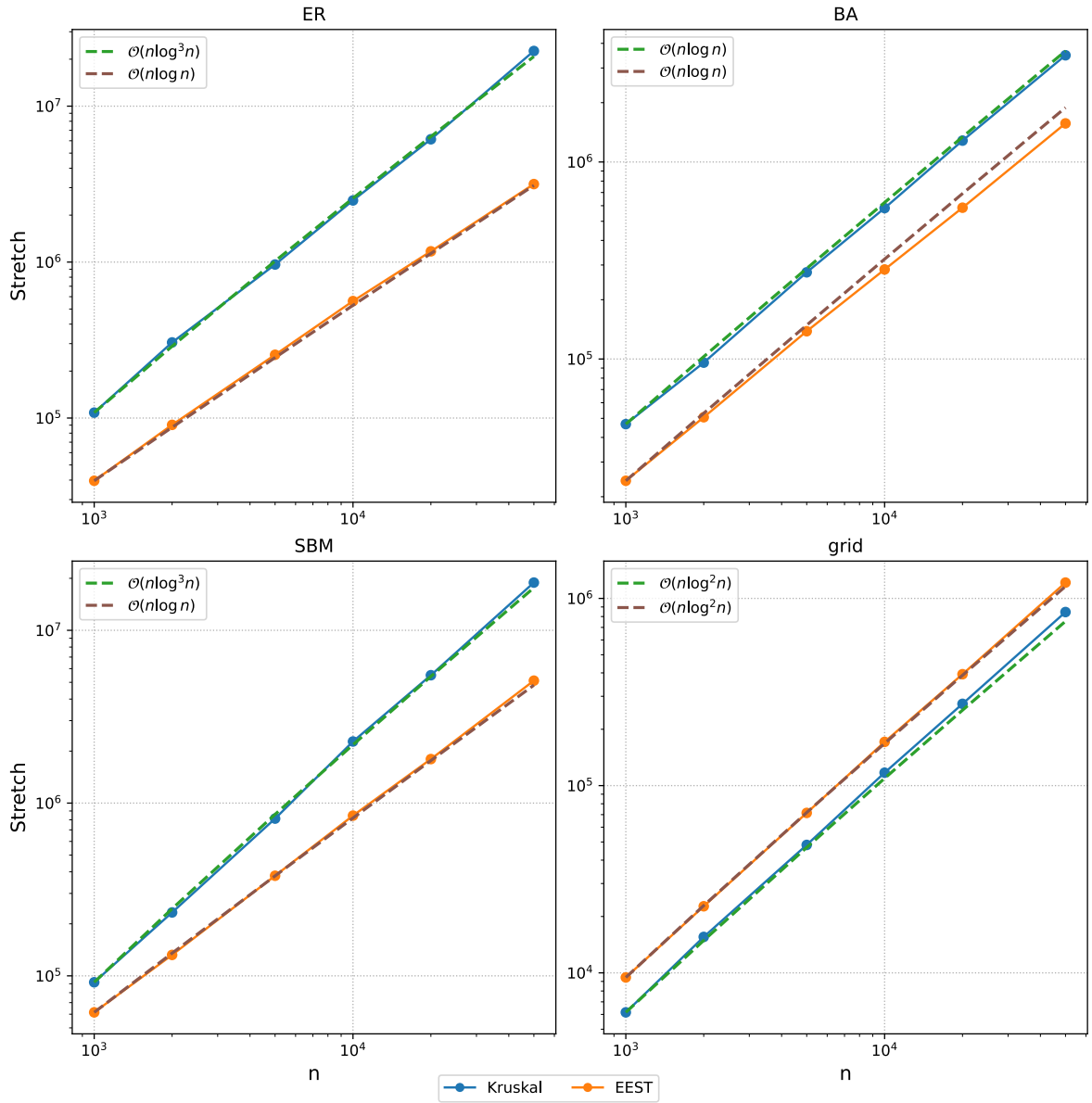


Figure 5.2. Comparison of the total stretch obtained using Kruskal or EEST's algorithm. The dashed curve represents the fitted asymptotic complexity of the KMP solver.

We observe that the EEST algorithm has both a better asymptotic behavior and a lower hidden constant for the ER, BA and SBM graphs. This is quite surprising since the ER and BA graphs were tested in the paper mentioned above and they did not beat Kruskal. Moreover for these three graphs, the total stretch grows as $\mathcal{O}(n \log n)$, which is better than the theoretical bound of $\mathcal{O}(n \log^2 n)$.

As for the grid graphs, the total stretch has the same performance of $\mathcal{O}(n \log^2 n)$ as using Kruskal's tree but has a higher hidden constant.

We can thus expect better performances from the KMP solver for the three first graphs by using the EEST tree instead of Kruskal. This is shown in Figure 5.3.

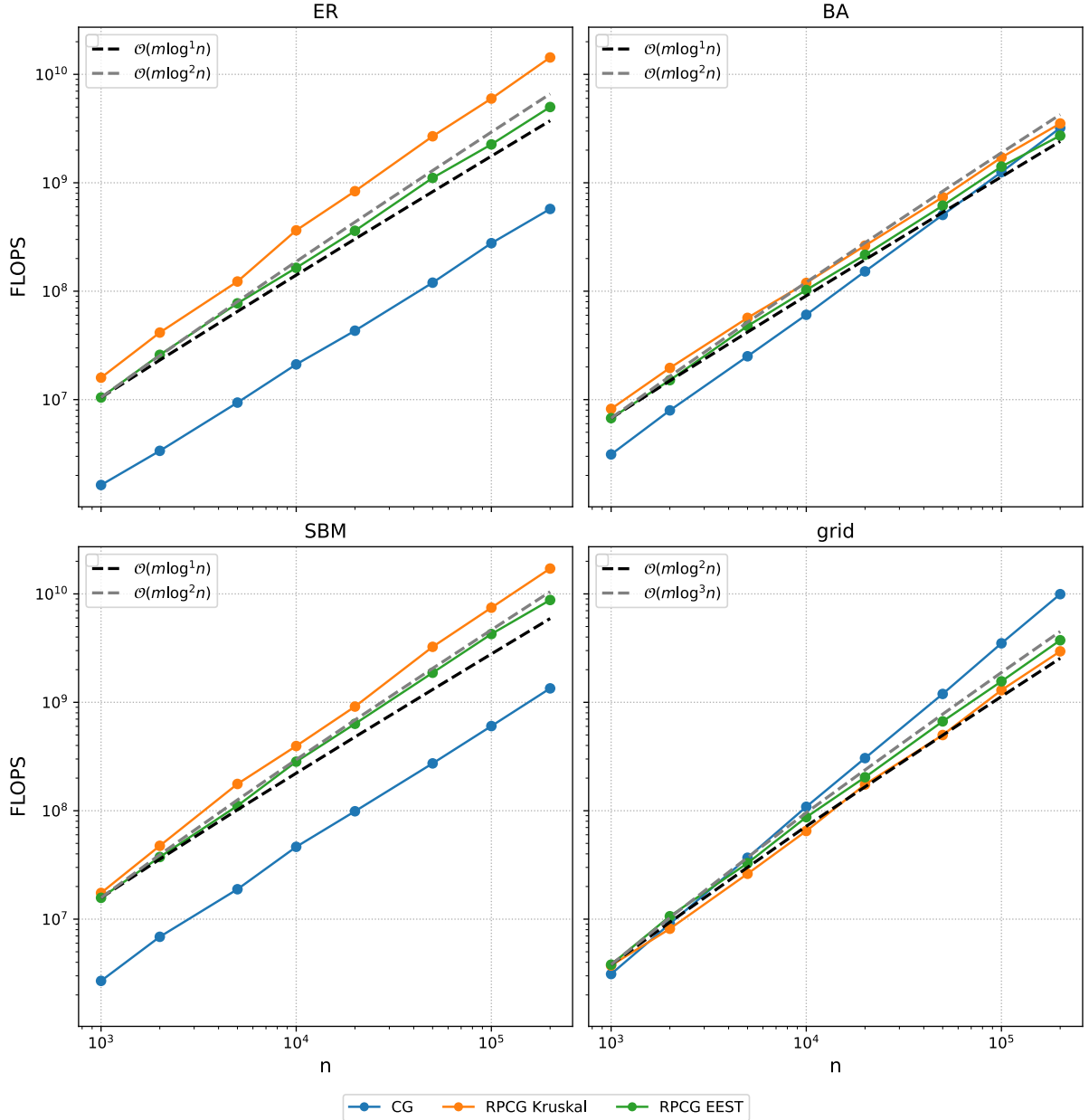


Figure 5.3. Effect of the quality of the spanning tree on the performances of the KMP solver.

For the ER, BA and SBM graphs, we indeed see that the KMP solver performs better with the EEST trees than with Kruskal's. The theoretical bound for the KMP solver with

a total stretch of $\mathcal{O}(n \log n)$ gives a running time of $\mathcal{O}(n \log n)$. Once again, in practice the running time is between the theoretical bound and the latter multiplied by logarithmic factor.

For the grid graphs, both trees give the same asymptotic results, with the Kruskal one being a bit better since it has a lower hidden constant.

We have confirmed with this experiment that the quality of the tree has a direct impact on the performance of the KMP solver. A tree with a lower stretch will lead to a faster solver in practice.

5.2 Finite difference problem

Up until now, the tested graphs had weighted edges. However, in practice some problems lead to Laplacian systems with unweighted edges. This is for example the case with the finite difference method.

The solver will be tested on a concrete problem of fluid mechanics that was resolved using finite differences that lead to solving a Laplacian system in an unweighted grid graph. The considered problem study the heating of a liquid in a box and the flow arising from natural convection. It is illustrated in Figure 5.4.

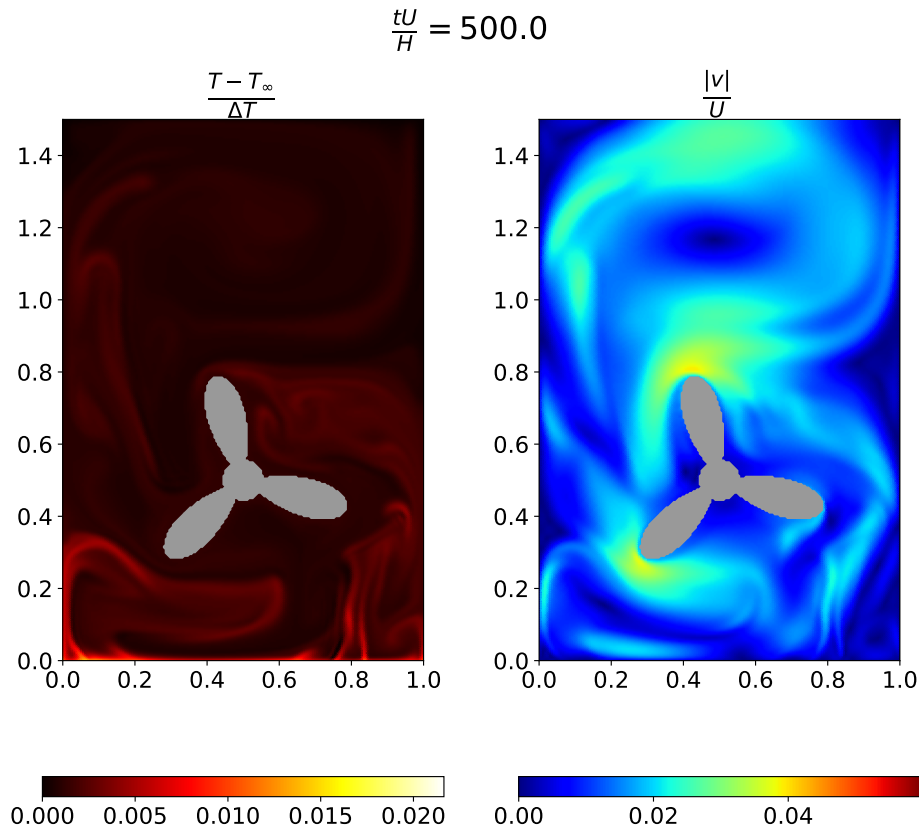


Figure 5.4. Temperature and velocity field of the test problem with finite differences

The Navier-Stokes equation that had to be solved were:

$$\begin{aligned}\nabla \cdot \mathbf{v} &= 0 \\ \frac{D\mathbf{v}}{Dt} &= -\nabla P + \nu \nabla^2 \mathbf{v} - \beta(T - T_\infty)\mathbf{g} - \chi \frac{\mathbf{v} - \mathbf{v}_s}{\Delta\tau} \\ \frac{DT}{Dt} &= \alpha \nabla^2 T - \chi \frac{T - T_s}{\Delta\tau}\end{aligned}$$

Without going into the details, these equations were integrated with a two-step projection scheme combined with a MAC mesh. At each time iteration, a solution $\mathbf{v}^*$ to the momentum equation is first computed. Then, there is a re-projection step, which requires solving a Poisson equation to obtain a velocity $\mathbf{v}$ that is divergence free:

$$\begin{aligned}\frac{1}{\Delta t} \nabla_h \cdot \mathbf{v}^* &= -\nabla_h^2 \phi \\ &= -\frac{\phi_{i+1,j} + \phi_{i-1,j} + \phi_{i,j+1} + \phi_{i,j-1} - 4\phi_{i,j}}{h^2}.\end{aligned}$$

The last line was obtained by using a second order centered finite difference to obtain an approximation of the discrete Laplacian $\nabla_h^2 \phi$. This thus leads to solving a Laplacian matrix of an unweighted 2d-grid.

For such graph, the stretch obtained with Kruskal, EEST as well as the known optimal spanning tree (see Figure 3.1) is shown in Figure 5.5:

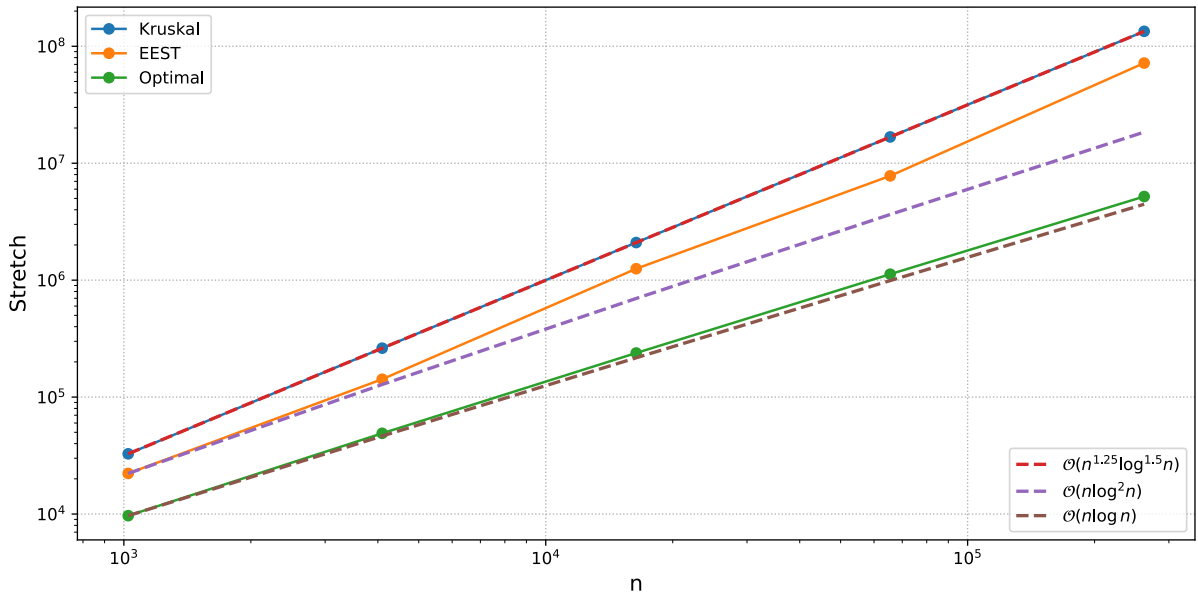


Figure 5.5. Total stretch with Kruskal, EEST's and the optimal spanning tree for unweighted grid graphs.

Kruskal's spanning tree is extremely bad to obtain LSST. Indeed, any tree is a maximum weight spanning tree in an unweighted graphs since every edge has the same weight. Kruskal will thus return any arbitrary spanning tree. The edges's stretch has no impact to Kruskal's algorithm since it was created to generate maximum weight spanning tree and not LSST.

The EEST algorithm gives better spanning trees but still aren't that good. The algorithm performed way better when the edges had heterogeneous weights. Indeed, it gave trees with stretch $\mathcal{O}(n \log^2 n)$ (see Figure 4.1), which is not the case anymore.

As for the optimal tree, it has the theoretical stretch of $\mathcal{O}(n \log n)$. Moreover, it has a smaller hidden constant than Kruskal and EEST's trees.

Using these trees with KMP give the results in Figure 5.6:

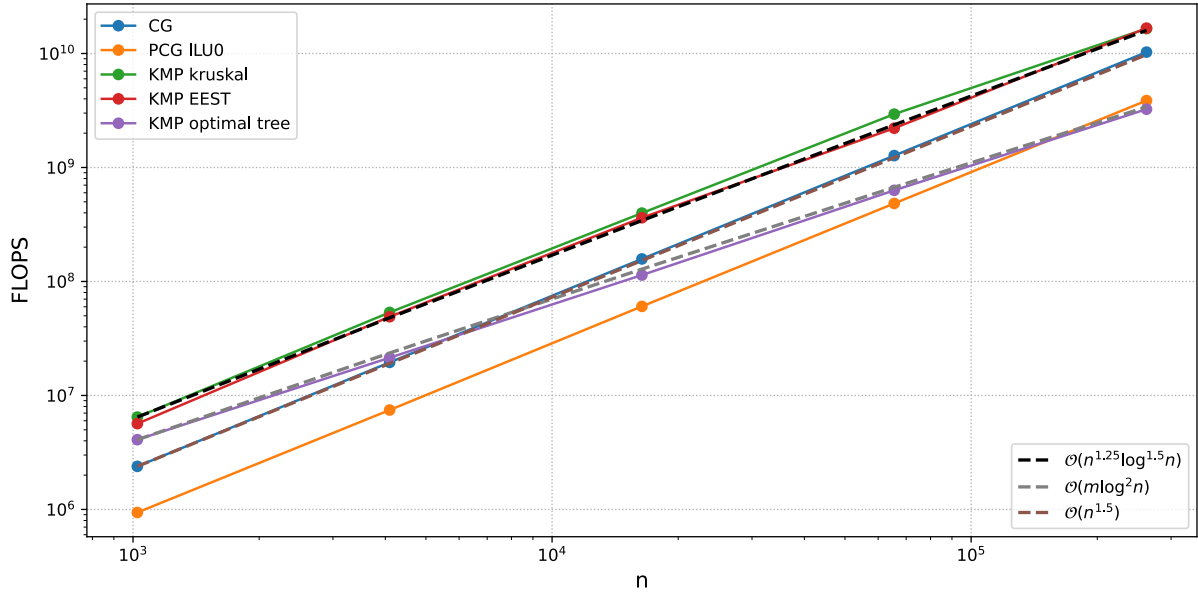


Figure 5.6. Performance of the KMP solver to solve finite difference Laplacian matrices.

Kruskal and EEST's performance are really close to each other. They have a complexity of $\mathcal{O}(n^{1.25} \log^{1.5} n)$. On the other hand, grids are known to have condition number in $\mathcal{O}(n)$, which gives CG a complexity of $\mathcal{O}(n^{1.5})$. This is closely followed in practice. Therefore, Both Kruskal and EEST will eventually be better than CG, even though it is not the case initially because of the big hidden constant.

The KMP solver with the optimal tree performs really well. It follows a complexity of $\mathcal{O}(m \log^2 n)$ and rapidly beats CG. It also beats PCG with ILU0 as a preconditioner after a certain mesh size.

The KMP solver thus works really well in unweighted grid graphs provided that a good low-stretch spanning tree is used. With the optimal tree, it beats both CG and PCG.

Nevertheless, there exists better solver to solve Laplacian on 2d-grid graphs such as Fourier methods or multigrid methods.

5.3 Time needed to generate the preconditioner

Until now, only the performance of the solver was tested. However, it is important to remember that generating the preconditioner also takes time. The time needed to compute these is dominated by the time needed to generate the spanning tree.

Kruskal's algorithm is known to have a complexity of $\mathcal{O}(m \log n)$. As for the EEST algorithm, it has a run-time of $\mathcal{O}(m \log^2 n)$ (See Table 3.1).

The run-time needed to generate the tree is shown in Figure 5.7.

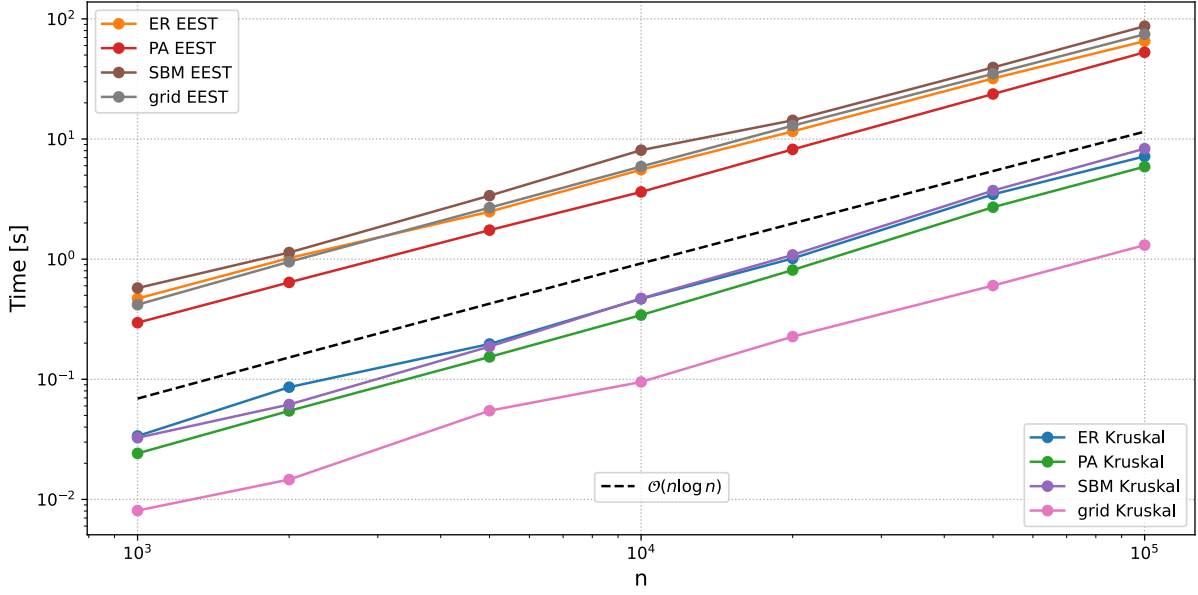


Figure 5.7. Time needed to generate the preconditioner using Kruskal or EEST algorithm to generate the spanning tree. The dashed line represent the asymptotic behavior of the curves.

Both Kruskal and EEST's algorithm have a complexity of $\mathcal{O}(m \log n)$ in practice. Though, Kruskal has a much lower hidden constant. The EEST algorithm that was available is in fact not efficiently implemented at all and could be much faster.

The cost of generating the spanning tree is not significant if the same system has to be solved many times. This happen quite often for fluid simulation problems. Indeed, these requires simulating many time steps. Many numerical schemes then require solving a discretized Poisson equation $Lx = b$ at each iteration. The matrix L remains the same during the whole simulation. In such case, we can precondition the system once and then repeatedly apply the same solver since only the right hand side b changes. The cost of generating the preconditioner is then amortized over the many steps even if it is mildly expensive.

Chapter 6

Application in finite element method

This brief chapter will apply the KMP solver on problems that arise from finite element method. It was proven in [BHV08] that a large class of elliptic partial differential equation discretized using finite differences can be solved to with symmetric, diagonally dominant solver. Such problems have the form:

Find $u : \Omega \Rightarrow \mathbb{R}$ such that

$$\begin{aligned}\nabla \cdot (a \nabla u) + f &= 0 \quad \forall x \in \Omega, \\ \mathbf{n} \cdot (a \nabla u) &= g \quad \forall x \in \Gamma_N, \\ u &= t \quad \forall x \in \Gamma_D.\end{aligned}$$

The function f is a source term, while g and t are respectively the Neumann (imposed flux on Γ_N) and Dirichlet (imposed value on Γ_D) boundary conditions.

This chapter will focus specifically on elliptic problems that are discretized with finite element using triangular elements. It has been proven that such problems give rise to a Laplacian system provided that all the angles of the elements are less than 90° (see section 5.4 [SF86]). They are then automatically suited to be solved with the KMP solver if this condition is respected.

If it is not respected, then the local stiffness matrix S of an element with an obtuse angle may have positive off-diagonal elements. These local matrices will be approximated by an SDD matrix as follows:

$$(S_{sdd})_{ij} = \begin{cases} s_{ij} & \text{if } i \neq j \text{ and } s_{ij} < 0 \\ 0 & \text{if } i \neq j \text{ and } s_{ij} \geq 0 \\ \sum_{k \neq j} -(S_{sdd})_{ik} & i = j \end{cases}$$

Such a matrix is SDD and will approximate the original matrix well in many cases (ie $\kappa(S, S_{sdd})$ is small). The impact of using these approximations will be tested.

6.1 Poisson problem

The first finite element problem that is tested will be the following simple Poisson problem:

$$\begin{aligned}\nabla^2 \cdot u &= -1 \quad \text{on } \Omega \\ u &= 0 \quad \text{on } \partial\Omega\end{aligned}$$

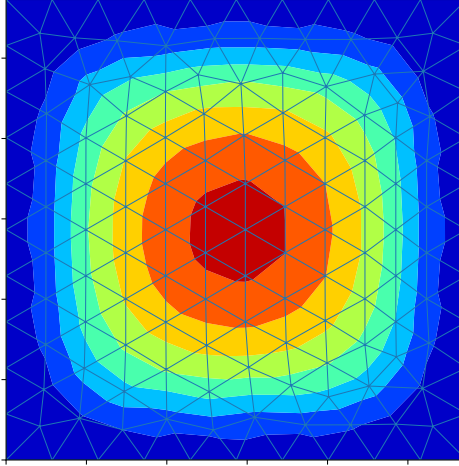


Figure 6.1. Poisson problem.

The number of obtuse triangles are shown in Table 6.1. The true global stiffness matrix K is assembled using every local SDD stiffness matrix S_{sdd} to give the global SDD approximation K_{sdd} . This is the matrix that is then preconditioned. It therefore already starts with a generalized condition number of $\kappa(K, K_{sdd})$ before even scaling it by the factor κ_1 .

n	Number of obtuse triangles	$\kappa(K, K_{sdd})$
511	0	1
1051	0	1
2059	2	1.003
5374	0	1
10889	0	1
21092	3	1.08

Table 6.1. Number of obtuse triangles and its impact on the quality of the preconditioner for the poisson problem.

There are almost never obtuse triangles in this first example. This is due to the fact that the geometry is really easy (a square), and it can easily be meshed without any irregular triangles.

Even in the rare cases where there are some irregular triangles, they have no real impact on the quality of the preconditioner as shown in Table 6.1.

Figure 6.2 shows the stretches as well as the condition number of the matrices obtained for this problem:

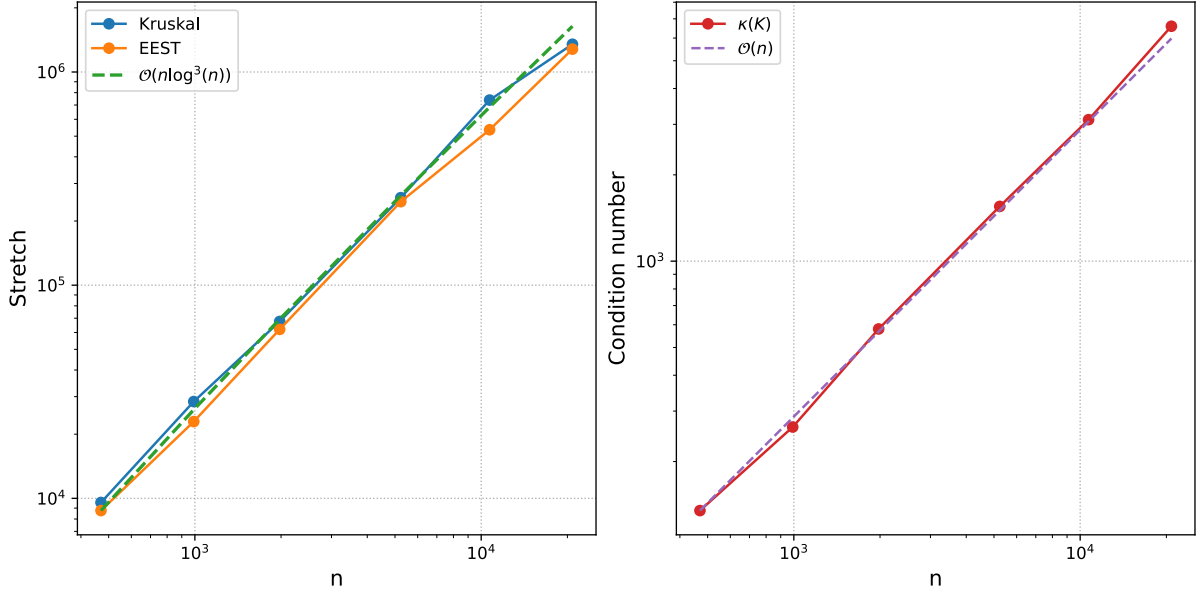


Figure 6.2. Stretch of the spanning trees and condition number of a finite difference Laplacian matrix.

Kruskal and EEST's algorithm have similar performances. It is reminiscent of what happened with the unweighted 2-d grids, although it has a better asymptotic value of the total stretch with $(\mathcal{O}(n \log^3 n))$. Once again, this is because almost every edge has the same weight. Indeed, in a good mesh with a uniform density of triangles like in this case, most of the triangles have the same size and thus have the same values in the local stiffness matrix. However we note that the asymptotic value of $\mathcal{O}(n \log^3 n)$ with triangles is much better than the value of $n^{0.5}$ obtained for the grid graphs.

As for the condition number, it grows linearly with the number of nodes.

Figure 6.3 shows the result of the KMP solver compared to CG and PCG with ILU0:

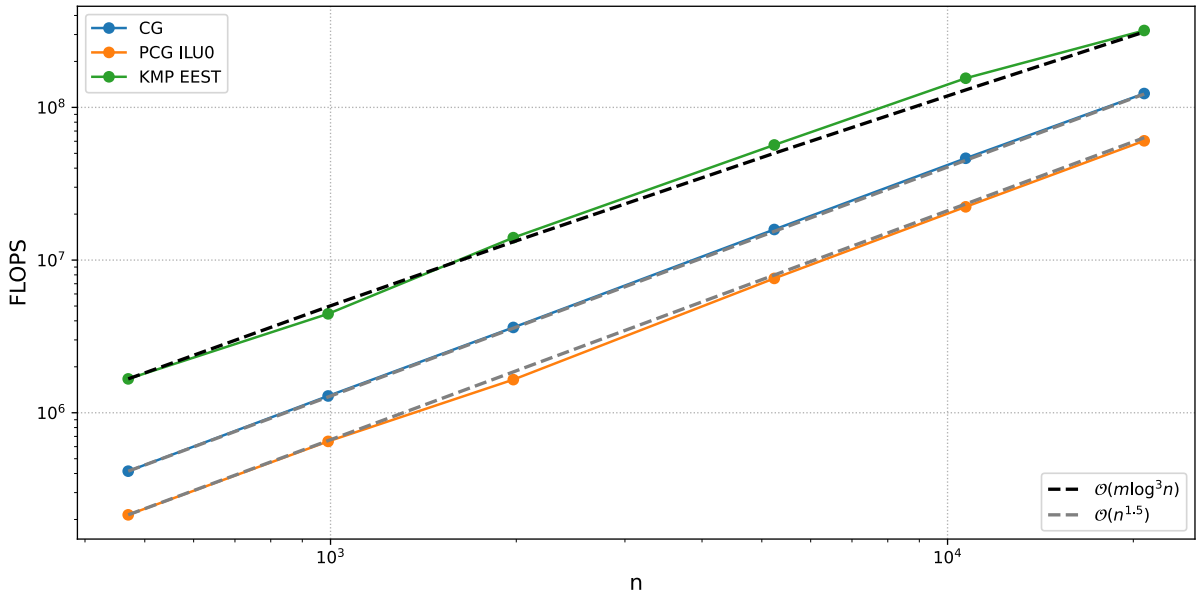


Figure 6.3. Performance of various solvers for the poisson problem in a grid using FEM. The dashed lines are the fitted asymptotic curve of the practical results.

The KMP solver performs as predicted with an asymptotic complexity of $\mathcal{O}(m \log^3 n)$. The complexity of the CG also follows the theoretical bound of $\mathcal{O}(n^{1.5})$ very closely. It has a much smaller hidden constant and is thus better than KMP for all the mesh size tested. However, it will eventually be beaten by KMP as the number of elements increases since the KMP solver has a better complexity.

The incomplete Cholesky factorization leads to a drastic reduction in the number of iterations needed to converge. However, it seems to still follow the same complexity as CG so it should also be beaten by KMP eventually.

6.2 A magnetostatics problem

The second problem comes from a concrete magnetostatics problem. It is the simulation of a motor with a variable reluctance in 2 dimensions. The equation of the rotation of the rotor is given by

$$I \frac{d\omega}{dt} = C,$$

where C is the torque that turns the rotor. Its value is obtained using Arkkio's method for torque computation [Wik23a]:

$$C = \frac{L}{\mu_0 d} \int_{\Omega_g} \frac{\partial a}{\partial \theta} \frac{\partial a}{\partial r} d\Omega.$$

This requires the computation of the unique component $A_z = a$ of the potential vector A and is obtained by solving

$$\nabla \cdot \left(\frac{1}{\mu} \nabla a \right) + j_s = 0.$$

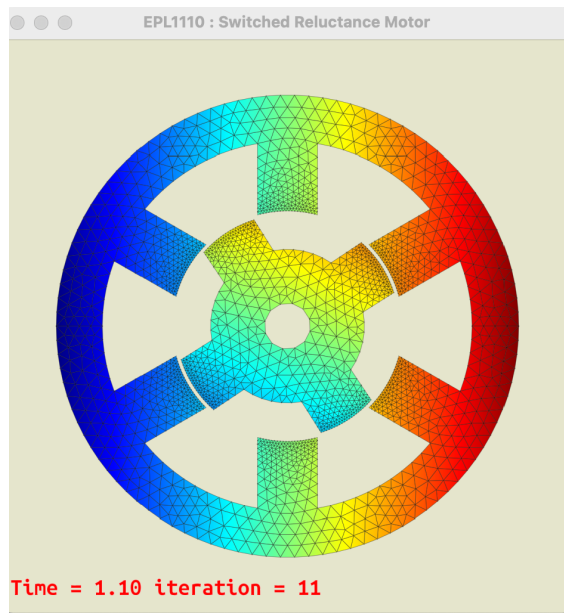


Figure 6.4. Motor problem.

Unlike the precedent example, there are many triangles with obtuse angles because of the geometry of the problem. Many triangles are forced to be irregular to respect the constraints of the geometry. The results are summarized in Table 6.2:

n	Fraction of obtuse triangles	$\kappa(K, K_{sdd})$
400	$263/774 = 34\%$	23.08
838	$422/1638 = 25.8\%$	10.72
1667	$679/3272 = 20.8\%$	2.53
4424	$958/8734 = 11\%$	1.62
14608	$639/28988 = 2.2\%$	1.1

Table 6.2. Number of obtuse triangles and its impact on the quality of the preconditioner for the motor problem.

As the size of the mesh increases, the fraction of elements that are obtuse decreases, which then increases the quality of the approximation K_{sdd} . However, its effect on $\kappa(K, K_{sdd})$ is not negligible for the smaller mesh sizes. The effect will be visible in the results plot.

Before showing it, let us first analyze Figure 6.5 which displays the total stretch for the magnetostatics problem as well as the condition numbers of the matrices.

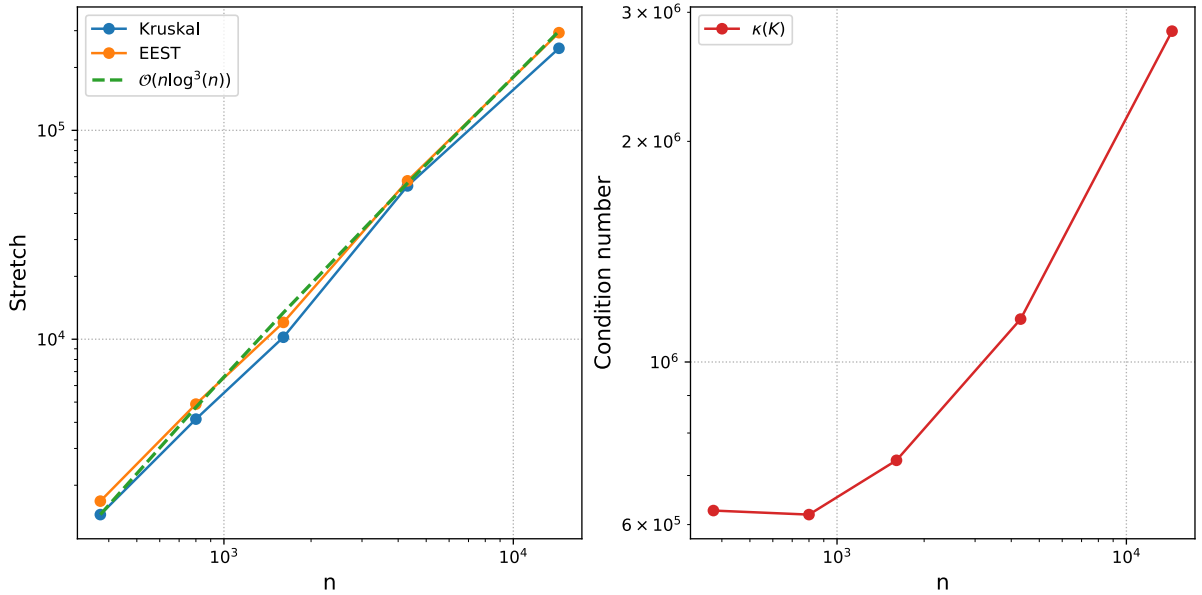


Figure 6.5. Stretch of the spanning trees and condition number of the finite difference Laplacian matrix for the motor problem.

Both Kruskal and EEST's algorithms once again gives stretch in $\mathcal{O}(n \log^3 n)$, though the hidden constant is smaller. This comes from the fact that the weight are less uniform than before. Indeed, the motor is separated into several domain, with each potentially having a different values of j_s . This lead to a smaller total stretch thanks to the weight variation. As for the condition numbers, they are much worse than the ones from the simple Poisson problem.

Figure 6.6 shows the performances of the KMP, CG and PCG with ILU0 for the second problem:

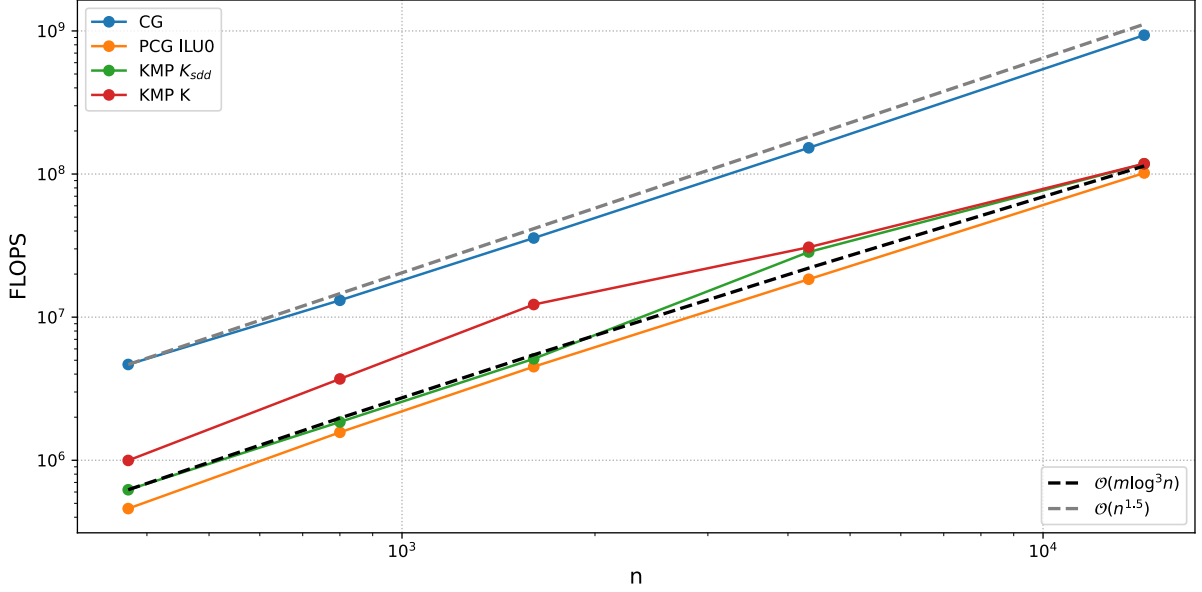


Figure 6.6. Performance of various solvers for the magnetostatic problem. The dashed lines are the fitted complexity of the practical results.

Let us analyze two cases. The first case is if the matrix K_{sdd} was not an approximation and was in fact the true stiffness matrix of the problem. Then, the performance of the KMP solver is extremely close to the predicted complexity of $\mathcal{O}(m \log^3 n)$.

However, this is not the case, and K_{sdd} is a $\kappa(K, K_{sdd})$ -approximate of K . This increases the condition number of the preconditioner and directly affects the performance of the KMP solver. However, it is still faster than CG even with the negative effect of the approximation. Moreover, the additional number of iterations gets smaller as the mesh size increases and eventually becomes negligible.

The KMP solver tends toward the behavior of the solver without the first approximate, which has a performance that is equivalent to that of PCG with ILU0. Note that these results were obtained with spanning trees that has non-optimal stretch, so the solver with a better LSST would beat PCG with ILU0 as well !

Finally, let us analyse the behavior of the solver at each iteration of the iterative method. This is given in Figure 6.7 by showing the residues

$$r_i = \frac{\|b - L_G x_i\|}{\|b\|}.$$

at each step for all three solvers.

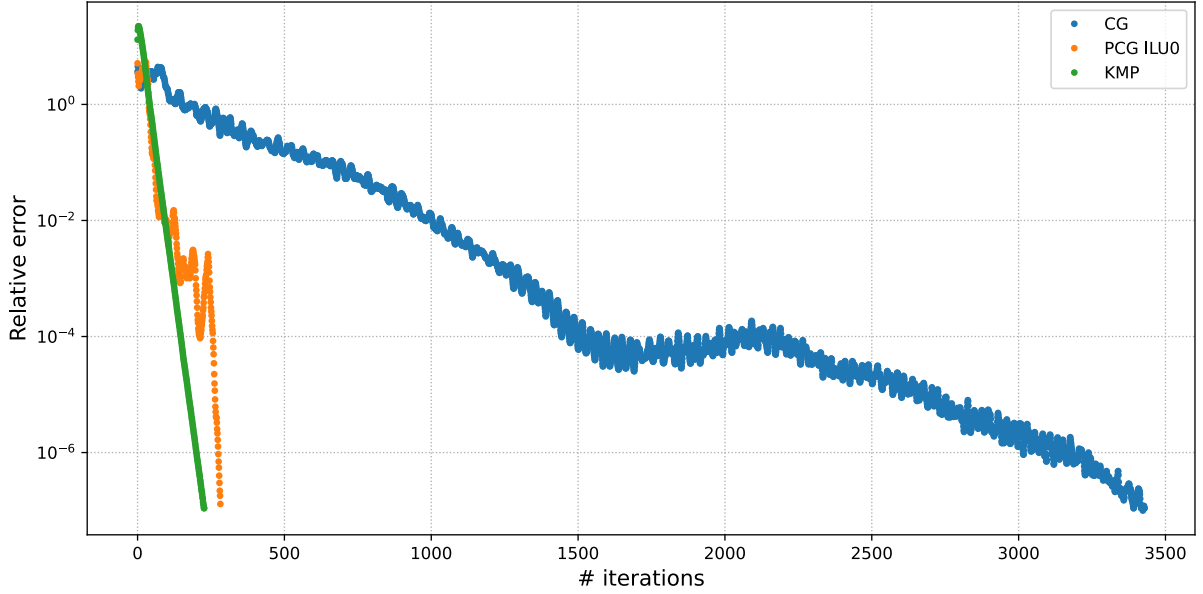


Figure 6.7. Relative error at each iteration for the CG, PCG and KMP solver for the problem with the biggest mesh size.

The KMP solver has an extremely consistent decrease in the relative error at each successive step of the solver. We see that it requires in fact less iterations than PCG with ILU0 to converge. However, the cost of each iteration is higher since it requires solving the spine-heavy graph, which had complexity $\mathcal{O}(m)$. On the other hand, each iteration of PCG with ILU0 only requires a matrix-vector product and the forward and backward resolution of a triangular matrix with exactly m elements (since there is no fill-in with ILU0 by definition). Both are in $\mathcal{O}(m)$ but ILU0 has a lower hidden constant.

All in all, both have really similar performances. The KMP solver thus has the potential to be really good for finite element problems provided that there exist good LSST algorithms for FEM meshes. This is the final bottleneck of this solver.

Chapter 7

Conclusion

To conclude, the various results obtained by the practical tests will be summarized in this section.

First of all, the sampling algorithm of the edges performs better in practice than predicted in theory. This is due to the fact that adding the tree back has the surprising effect of increasing drastically the quality of the approximation. This results in a lower hidden constant in the solver complexity (κ_1 is smaller).

Then, we have confirmed that the resolution of spine-heavy graph is indeed done in linear time. Moreover, the performance of the KMP solver on the general graphs closely followed the theory. The complexity was always within the true theoretical bound of $\mathcal{O}(m \log^{0.5c+0.5})$ and the latter multiplied by a logarithmic factor. This is most likely due to the fact that the arithmetic are not perfect.

We have then tested the impact of the precision required to solve the recursive sub systems and concluded that it was better to take the most crude precision $\epsilon_c = 0.1$. This leads to better performances in practice than using higher values of ϵ_c .

All in all, we have thus confirmed that the practical complexity of the KMP solver is

$$\mathcal{O}(m \log^{0.5c+1.5}).$$

It is confirmed to be independent of the condition number of the original matrix and instead only depend on the quality of the low-stretch spanning tree.

We have then found that graphs with a lognormal weight distribution are well suited for the KMP solver.

After that, we analyzed the effect of the spanning tree on the performance of the solver and confirmed that using spanning trees with lower-stretch gives better performances. The LSST algorithm thus represents the last bottleneck for the solver.

Finally, the solver was tested in three application: a fluid dynamic problem discretized with finite element method, and two finite element problems. For the former, the KMP solver is better than CG and PCG with ILU0 provided that the optimal spanning tree is used.

Then, for the finite element problems, the solver is negatively impacted by the fraction of triangles that are obtuse. However, this effect gets less and less important as the mesh size increases and eventually becomes negligible. The KMP solver will thus perform really well for FEM matrices too.

Problems with fluid simulation represents an extremely good candidate since these often requires solving multiple time the same Laplacian matrix.

7.1 Future works

This small section will list some experiments I would have liked to test but did not have the time to do it.

For future works, it should be a priority to develop LSST algorithms that gives good stretch in practice for meshes that arise from finite element method. Indeed, with better trees, the KMP solver would beat PCG with ILU0.

It would also be interesting to see if ideas from the other almost-linear time Laplacian solver can be combined in some way or another with the KMP solver. This thesis focused solely on the KMP-solver, which was the culmination of many previous combinatorial preconditioners. Nevertheless, there already exists theoretical algorithms that improve upon the KMP solver. ($\mathcal{O}(m \log^{0.5} n)$ in [CKM⁺14] and $\mathcal{O}(m(\log \log n)^c)$ in [JS21]). Though they are much more theoretical and it remains to be seen if they can be practical.

Finally, more finite element problems should be tested in future works. The solver shall be tested with meshes that are not triangular and with nonlinear shape functions. One important test that is also missing from this thesis is 3D problems. These are the most important and should be tested with a high priority.

Another type of FEM problems that are extremely used in real life are linear-elasticity problems. I initially wanted to solve such a problem with KMP but these matrices do not give SDD-matrices. It would be interesting to see if they can be reduced into one or not.

Bibliography

- [ABN08] Ittai Abraham, Yair Bartal, and Ofer Neiman. Nearly tight low stretch spanning trees. In *2008 49th Annual IEEE Symposium on Foundations of Computer Science*, pages 781–790. IEEE, 2008.
- [AKPW95] Noga Alon, Richard M Karp, David Peleg, and Douglas West. A graph-theoretic game and its application to the k-server problem. *SIAM Journal on Computing*, 24(1):78–100, 1995.
- [AN19] Ittai Abraham and Ofer Neiman. Using petal-decompositions to build a low stretch spanning tree. *SIAM Journal on Computing*, 48(2):227–248, 2019.
- [BGH⁺06] Marshall Bern, John R Gilbert, Bruce Hendrickson, Nhat Nguyen, and Sivan Toledo. Support-graph preconditioners. *SIAM Journal on Matrix Analysis and Applications*, 27(4):930–951, 2006.
- [BH01] Erik G Boman and Bruce Hendrickson. On spanning tree preconditioners. *Sandia National Lab*, 2001.
- [BH03] Erik G Boman and Bruce Hendrickson. Support theory for preconditioning. *SIAM Journal on Matrix Analysis and Applications*, 25(3):694–717, 2003.
- [Bha20] Vardhini Bhasker. Lowstretchspanningtree. <https://github.com/vardhinibhasker/LowStretchSpanningTree>, 2020.
- [BHV08] Erik G Boman, Bruce Hendrickson, and Stephen Vavasis. Solving elliptic finite element systems in near-linear time with support preconditioners. *SIAM Journal on Numerical Analysis*, 46(6):3264–3284, 2008.
- [CKM⁺14] Michael B Cohen, Rasmus Kyng, Gary L Miller, Jakub W Pachocki, Richard Peng, Anup B Rao, and Shen Chen Xu. Solving sdd linear systems in nearly $m \log^{1/2} n$ time. In *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*, pages 343–352, 2014.
- [Dew18] Kevin Deweese. *Bridging the Theory-Practice Gap of Laplacian Linear Solvers*. University of California, Santa Barbara, 2018.
- [EEST05] Michael Elkin, Yuval Emek, Daniel A Spielman, and Shang-Hua Teng. Lower-stretch spanning trees. In *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, pages 494–503, 2005.
- [Gre96] Keith D Gremban. *Combinatorial preconditioners for sparse, symmetric, diagonally dominant linear systems*. PhD thesis, Carnegie Mellon University, 1996.

- [HLMW15] Daniel Hoske, Dimitar Lukarski, Henning Meyerhenke, and Michael Wegner. Is nearly-linear the same in theory and practice? a case study with a combinatorial laplacian solver. In *Experimental Algorithms: 14th International Symposium, SEA 2015, Paris, France, June 29–July 1, 2015, Proceedings 14*, pages 205–218. Springer, 2015.
- [JS21] Arun Jambulapati and Aaron Sidford. Ultrasparse ultrasparsifiers and faster laplacian system solvers. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 540–559. SIAM, 2021.
- [Kla12] Erica Klarreich. Network solutions. *Simons foundation*, 2012.
- [KMP11] Ioannis Koutis, Gary L Miller, and Richard Peng. A nearly- $m \log n$ time solver for sdd linear systems. In *2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*, pages 590–598. IEEE, 2011.
- [KMP12] Ioannis Koutis, Gary L Miller, and Richard Peng. A fast solver for a class of linear systems. *Communications of the ACM*, 55(10):99–107, 2012.
- [KMP14] Ioannis Koutis, Gary L Miller, and Richard Peng. Approaching optimality for solving sdd linear systems. *SIAM Journal on Computing*, 43(1):337–354, 2014.
- [KOSZ13] Jonathan A Kelner, Lorenzo Orecchia, Aaron Sidford, and Zeyuan Allen Zhu. 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*, pages 911–920, 2013.
- [LS76] Wai-Hung Liu and Andrew H Sherman. Comparative analysis of the cuthill–mckee and the reverse cuthill–mckee ordering algorithms for sparse matrices. *SIAM Journal on Numerical Analysis*, 13(2):198–213, 1976.
- [Mar13] Anne Marsden. Eigenvalues of the laplacian and their relationship to the connectedness of a graph. *University of Chicago, REU*, 2013.
- [McK15] Rebecca McKnight. *Low-stretch trees for network visualization*. PhD thesis, University of British Columbia, 2015.
- [PKBK14] Pál András Papp, Sándor Kisfaludi-Bak, and Zoltán Király. Low-stretch spanning trees. *Undergraduate thesis, Eötvös Loránd University*, 2014.
- [RV07] Mark Rudelson and Roman Vershynin. Sampling from large matrices: An approach through geometric functional analysis. *Journal of the ACM (JACM)*, 54(4):21–es, 2007.
- [SF86] Gilbert Strang and LB Freund. Introduction to applied mathematics. *Journal of Applied Mechanics*, 53(2):480, 1986.
- [Spi10] Daniel A Spielman. Effective resistance. <http://www.cs.yale.edu/homes/spielman/462/2010/lect13-10.pdf>, 2010.
- [Sri10] Nikhil Srivastava. *Spectral sparsification and restricted invertibility*. Yale University, 2010.

- [SS08] Daniel A Spielman and Nikhil Srivastava. Graph sparsification by effective resistances. In *Proceedings of the fortieth annual ACM symposium on Theory of computing*, pages 563–568, 2008.
- [ST14] Daniel A Spielman and Shang-Hua Teng. Nearly linear time algorithms for preconditioning and solving symmetric, diagonally dominant linear systems. *SIAM Journal on Matrix Analysis and Applications*, 35(3):835–885, 2014.
- [Stü01] Klaus Stüben. An introduction to algebraic multigrid. *Multigrid*, pages 413–532, 2001.
- [SW09] Daniel A Spielman and Jaehoh Woo. A note on preconditioning by low-stretch spanning trees. *arXiv preprint arXiv:0903.2816*, 2009.
- [Vai91] Pravin M Vaidya. Solving linear equations with symmetric diagonally dominant matrices by constructing good preconditioners. *manuscript*, 2(3.4):2–4, 1991.
- [Wes95] Pieter Wesseling. Introduction to multigrid methods. Technical report, Institute for Computer Applications in Science and Engineering, 1995.
- [Wik23a] Wikipedia. Arkkio’s method for torque computation, 2023.
- [Wik23b] Wikipedia. Barabási–albert model, 2023.
- [Wik23c] Wikipedia. Erdős–rényi model, 2023.
- [Wik23d] Wikipedia. Stochastic block model, 2023.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | www.uclouvain.be/epl