Stat260/CS294: Spectral Graph Methods **Lecture 26 - 04/28/2015**

Lecture: Laplacian solvers (2 of 2)

Lecturer: Michael Mahoney *Scribe: Michael Mahoney*

Warning: these notes are still very rough. They provide more details on what we discussed in class, but there may still be some errors, incomplete/imprecise statements, etc. in them.

26 Laplacian solvers, cont.

Last time, we talked about a very simple solver for Laplacian-based systems of linear equations, i.e., systems of linear equations of the form $Ax = b$, where the constraint matrix A is the Laplacian of a graph. This is not fully-general—Laplacians are SPSD matrices of a particular form—but equations of this form arise in many applications, certain other SPSD problems such as those based on SDD matrices can be reduced to this, and there has been a lot of work recently on this topic since it is a primitive for many other problems. The solver from last time is very simple, and it highlights the key ideas used in fast solvers, but it is very slow. Today, we will describe how to take those basic ideas and, by coupling them with certain graph theoretic tools in various ways, obtain a "fast" nearly linear time solver for Laplacian-based systems of linear equations.

In particular, today will be based on the Batson-Spielman-Srivastava-Teng and the Koutis-Miller-Peng articles.

26.1 Review from last time and general comments

Let's start with a review of what we covered last time.

Here is a very simple algorithm. Given as input the Laplacian L of a graph $G = (V, E)$ and a right hand side vector b, do the following.

- Construct a sketch of G by sampling elements of G, i.e., rows of the edge-node incidence matrix, with probability proportional to the leverage scores of that row, i.e., the effective resistances of that edge.
- Use the sketch to construct a solution, e.g., by solving the subproblem with a black box or using it as a preconditioner to solve the original problem with an iterative method.

The basic result we proved last time is the following.

Theorem 1. *Given a graph G with Laplacian L, let x_{opt} be the optimal solution of $Lx = b$; then the above algorithm returns a vector $\tilde{x}_{opt}$ such that, with constant probability,*

$$\|x_{opt} - \tilde{x}_{opt}\|_L \leq \epsilon \|x_{opt}\|_L. \tag{1}$$

The proof of this result boiled down to showing that, by sampling with respect to a judiciously-chosen set of nonuniform importance sampling probabilities, then one obtains a data-dependent subspace embedding of the edge-incidence matrix. Technically, the main thing to establish was that, if U is an $m \times n$ orthogonal matrix spanning the column space of the weighted edge-incidence matrix, in which case $I = I_n = U^T U$, then

$$\| I - (SU)^T (SU) \|_2 \leq \epsilon, \tag{2}$$

where S is a sampling matrix that represents the effect of sampling elements from L.

The sampling probabilities that are used to create the sketch are weighted versions of the statistical leverage scores of the edge-incidence matrix, and thus they also are equal to the effective resistance of the corresponding edge in the graph. Importantly, although we didn't describe it in detail, the theory that provides bounds of the form of Eqn. (2) is robust to the exact form of the importance sampling probabilities, e.g., bounds of the same form hold if any other probabilities are used that are "close" (in a sense that we will discuss) to the statistical leverage scores.

The running time of this simple strawman algorithm consists of two parts, both of which the fast algorithms we will discuss today improve upon.

- Compute the leverage scores, exactly or approximately. A naive computation of the leverage scores takes $O(mn^2)$ time, e.g., with a black box QR decomposition routine. Since they are related to the effective resistances, one can—theoretically at least compute them with any one of a variety of fast nearly linear time solvers (although one has a chicken-and-egg problem, since the solver itself needs those quantities). Alternatively, since one does not need the exact leverage scores, one could hope to approximate them in some way—below, we will discuss how this can be done with low-stretch spanning trees.
- Solve the subproblem, exactly or approximately. A naive computation of the solution to the subproblem can be done in $O(n^3)$ time with standard direct methods, or it can be done with an iterative algorithm that requires a number of matrix-vector multiplications that depends on the condition number of L (which in general could be large, e.g., $\Omega(n)$) times m, the number of nonzero elements of L. Below, we will see how this can be improved with sophisticated versions of certain preconditioned iterative algorithms.

More generally, here are several issues that arise.

- Does one use exact or approximate leverage scores? Approximate leverage scores are sufficient for the worst-case theory, and we will see that this can be accomplished by using LSSTs, i.e., combinatorial techniques.
- How good a sketch is necessary? Last time, we sampled $\Theta\left(\frac{n \log(n)}{\epsilon^2}\right)$ elements from L to obtain a $1 \pm \epsilon$ subspace embedding, i.e., to satisfy Eqn. (2), and this leads to an ϵ-approximate solution of the form of Eqn (1). For an iterative method, this might be overkill, and it might suffice to satisfy Eqn. (2) for, say, $\epsilon = \frac{1}{2}$.
- What is the dependence on ϵ? Last time, we sampled and then solved the subproblem, and thus the complexity with respect to ϵ is given by the usual random sampling results. In particular, since the complexity is a low-degree polynomial in $\frac{1}{\epsilon}$, it will be essentially impossible to obtain a high-precision solution, e.g., with $\epsilon = 10^{-16}$, as is of interest in certain applications.

- What is the dependence on the condition number $\kappa(L)$? In general, the condition number can be very large, and this will manifest itself in a large number of iterations (certainly in worst case, but also actually quite commonly). By working with a preconditioned iterative algorithm, one should aim for a condition number of the preconditioned problem that is quite small, e.g., if not constant then $\log(n)$ or less. In general, there will be a tradeoff between the quality of the preconditioner and the number of iterations needed to solve the preconditioned problem.
- Should one solve the subproblem directly or use it to construct a preconditioned to the original problem? Several of the results just outlined suggest that an appropriate iterative methods should be used and this is what leads to the best results.

Remark. Although we are not going to describe it in detail, we should note that the LSSTs will essentially allow us to approximate the large leverage scores, but they won't have anything to say about the small leverage scores. We saw (in a different context) when we were discussing statistical inference issues that controlling the small leverage scores can be important (for proving statistical claims about unseen data, but not for claims on the empirical data). Likely similar issues arise here, and likely this issue can be mitigated by using implicitly regularized Laplacians, e.g., as as implicitly computed by certain spectral ranking methods we discussed, but as far as I know no one has explicitly addressed these questions.

26.2 Solving linear equations with direct and iterative methods

Let's start with the second step of the above two-level algorithm, i.e., how to use the sketch from the first step to construct an approximate solution, and in particular how to use it to construct a preconditioner for an iterative algorithm. Then, later we will get back to the first step of how to construct the sketch.

As you probably know, there are a wide range of methods to solve linear systems of the form $Ax = b$, but they fall into two broad categories.

- **Direct methods.** These include Gaussian elimination, which runs in $O(n^3)$ time; and Strassen-like algorithms, which run in $O(n^\omega)$ time, where $\omega = 2.87\ldots2.37$. Both require storing the full set of in general $O(n^2)$ entires. Faster algorithms exist if A is structured. For example, if A is $n \times n$ PSD with m nonzero, then conjugate gradients, used as a direct solver, takes $O(mn)$ time, which if $m = O(n)$ is just $O(n^2)$. That is, in this case, the time it takes it proportional to the time it takes just to write down the inverse. Alternatively, if A is the adjacency matrix of a path graph or any tree, then the running time is $O(m)$; and so on.
- **Iterative methods.** These methods don't compute an exact answer, but they do compute an ϵ-approximate solution, where ϵ depends on the structural properties of A and the number of iterations, and where ϵ can be made smaller with additional iterations. In general, iterations are performed by doing matrix-vector multiplications. Advantages of iterative methods include that one only needs to store A, these algorithms are sometimes very simple, and they are often faster than running a direct solver. Disadvantages include that one doesn't obtain an exact answer, it can be hard to predict the number of iterations, and the running time depends on the eigenvalues of A, e.g., the condition number $\kappa(A) = \frac{\lambda_{max}(A)}{\lambda_{min}(A)}$. Examples include the Richardson iteration, various Conjugate Gradient like algorithms, and the Chebyshev iteration.

Since the running time of iterative algorithms depend on the properties of A, so-called *preconditioning methods* are a class of methods to transform a given input problem into another problem such that the modified problem has the same or a related solution to the original problem; and such that the modified problem can be solved with an iterative method more quickly.

For example, to solve $Ax = b$, with $A \in \mathbb{R}^{n \times n}$ and with $m = \mathbf{nnz}(A)$, if we define $\kappa(A) = \frac{\lambda_{max}(A)}{\lambda_{min}(A)}$, where λ_{max} and λ_{min} are the maximum and minimum non-zero eigenvalues of A, to be the condition number of A, then CG runs in

$$O\left(\sqrt{\kappa(A)}\log\left(1/\epsilon\right)\right)$$

iterations (each of which involves a matrix-vector multiplication taking $O(m)$ time) to compute and ϵ-accurate solution to $Ax = b$. By an ϵ-accurate approximation, here we mean the same notion that we used above, i.e., that

$$\|A\tilde{x}_{opt} - b\|_A \leq \epsilon\|b\|_A,$$

where the so-called A-norm is given by $\|y\|_A = \sqrt{y^T A y}$. This A-norm is related to the usual Euclidean norm as follows: $\|y\|_A \leq \kappa(A)\|y\|_2$ and $\|y\|_2 \leq \kappa(A)\|y\|_A$. While the A-norm is perhaps unfamiliar, in the context of iterative algorithms it is not too dissimilar to the usual Euclidean norm, in that, given an ϵ-approximation for the former, we can obtain an ϵ-approximation for the latter with $O\left(\log\left(\kappa(A)/\epsilon\right)\right)$ extra iterations.

In this context, preconditioning typically means solving

$$B^{-1}Ax = B^{-1}b,$$

where B is chosen such that $\kappa\left(B^{-1}A\right)$ is small; and it is easy to solve problems of the form $Bz = c$. The two extreme cases are $B = I$, in which case it is easy to compute and apply but doesn't help solve the original problem, and $B = A^{-1}$, which means that the iterative algorithm would converge after zero steps but which is difficult to compute. The running time of the preconditioned problem involves

$$O\left(\sqrt{\kappa\left(B^{-1}A\right)}\log\left(1/\epsilon\right)\right)$$

matrix vector multiplications. The quantity $\kappa\left(B^{-1}A\right)$ is sometimes known as the *relative condition number of A with respect to B*—in general, finding a B that makes it smaller takes more initial time but leads to fewer iterations. (This was the basis for the comment above that there is a tradeoff in choosing the quality of the preconditioner, and it is true more generally.)

These ideas apply more generally, but we consider applying them here to Laplacians. So, in particular, given a graph G and its Laplacian L_G, one way to precondition it is to look for a different graph H such that $L_H \approx L_G$. For example, one could use the sparsified graph that we computed with the algorithm from last class. That sparsified graph is actually an ϵ-good preconditioned, but it is too expensive to compute. To understand how we can go beyond the linear algebra and exploit graph theoretic ideas to get good approximations to them more quickly, let's discuss different ways in which two graphs can be close to one another.

26.3 Different ways two graphs can be close

We have talked formally and informally about different ways graphs can be close, e.g., we used the idea of similar Laplacian quadratic forms when talking about Cheeger's Inequality and the quality of spectral partitioning methods. We will be interested in that notion, but we will also be interested in other notions, so let's now discuss this topic in more detail.

- **Cut similarity.** One way to quantify the idea that two graphs are close is to say that they are similar in terms of their cuts or partitions. The standard result in this area is due to BZ, who developed the notion of *cut similarity* to develop fast algorithms for min cut and max flow and other related combinatorial problems. This notion of similarity says that two graphs are close if the weights of the cuts, i.e., the sum of edges or edge weights crossing a partition, are close for all cuts. To define it, recall that, given a graph $G = (V, E, W)$ and a set $S \subset V$, we can define $\text{cut}_G = \sum_{u \in S, v \in \bar{S}} W_{(uv)}$. Here is the definition.

  Definition 1. *Given two graphs, $G = (V, E, W)$ and $\tilde{G} = (V, \tilde{E}, \tilde{W})$, on the same vertex set, we say that G and $\tilde{G}$ are* σ-cut-similar *if, for all $S \subseteq V$, we have that*

  $$\frac{1}{\sigma} cut_{\tilde{G}}(S) \leq cut_G(S) \leq \sigma\, cut_{\tilde{G}}(S).$$

  As an example of a result in this area, the following theorem shows that every graph is cut-similar to a graph with average degree $O(\log(n))$ and that one can compute that cut-similar graph quickly.

  Theorem 2 (BK)**.** *For all $\epsilon > 0$, every graph $G = (V, E, W)$ has a $(1+\epsilon)$-cut-similar graph $\tilde{G} = (V, \tilde{E}, \tilde{V})$ such that $\tilde{E} \subseteq E$ and $|\tilde{E}| = O\left(n \log(n/\epsilon^2)\right)$. In addition, the graph $\tilde{G}$ can be computed in $O\left(m \log^3(n) + m \log(n/\epsilon^2)\right)$ time.*

- **Spectral similarity.** ST introduced the idea of spectral similarity in the context of nearly linear time solvers. One can view this in two complementary ways.

  – As a generalization of cut similarity.

  – As a special case of subspace embeddings, as used in RLA.

  We will do the former here, but we will point out the latter at an appropriate point.

  Given $G = (V, E, W)$, recall that $L : \mathbb{R}^n \to \mathbb{R}$ is a quadratic form associated with G such that

  $$L(x) = \sum_{(uv) \in E} W_{(uv)} \left(x_u - x_v\right)^2.$$

  If $S \subset V$ and if x is an indicator/characteristic vector for the set S, i.e., it equals 1 on nodes $u \in S$, and it equals 0 on nodes $v \in S$, then for those indicator vectors x, we have that $L(x) = \text{cut}_G(x)$. We can also ask about the values it takes for other vectors x. So, let's define the following.

  Definition 2. *Given two graphs, $G = (V, E, W)$ and $\tilde{G} = (V, \tilde{E}, \tilde{W})$, on the same vertex set, we say that G and $\tilde{G}$ are* σ-spectrally similar *if, for all $x \in \mathbb{R}^n$, we have that*

  $$\frac{1}{\sigma} L_{\tilde{G}}(x) \leq L_G(x) \leq \sigma L_{\tilde{G}}(x).$$

  That is, two graphs are spectrally similar if their Laplacian quadratic forms are close.

  In addition to being a generalization of cut similarity, this also corresponds to a special case of subspace embeddings, restricted from general matrices to edge-incidence matrices and their associated Laplacians.

– To see this, recall that subspace embeddings preserve the geometry of the subspace and that this is quantified by saying that all the singular values of the sampled/sketched version of the edge-incidence matrix are close to 1, i.e., close to those of the edge-incidence matrix of the original un-sampled graph. Then, by considering the Laplacian, rather than the edge-incidence matrix, the singular values of the original and sketched Laplacian are also close, up to a quadratic of the approximation factor on the edge-incidence matrix.

Here are several other things to note about spectral embeddings.

– Two graphs can be cut-similar but not spectrally-similar. For example, consider G to be an n-vertex path and $\tilde{G}$ to be an n-vertex cycle. They are 2-cut similar but are only n-spectrally similar.
– Spectral similarity is identical to the notion of relative condition number in NLA that we mentioned above. Recall, given A and B, then $A \preceq B$ iff $x^T A x \leq x^T B x$, for all $x \in \mathbb{R}^n$. Then, A and B, if they are Laplacians, are spectrally similar if $\frac{1}{\sigma} B \preceq A \preceq \sigma B$. In this case, they have similar eigenvalues, since: from the Courant-Fischer results, if $\lambda_1, \ldots, \lambda_n$ are the eigenvalues of A and $\tilde{\lambda}_1, \ldots, \tilde{\lambda}_n$ are the eigenvalues of B, then for all i we have that $\frac{1}{\sigma}\tilde{\lambda}_i \leq \lambda_i \leq \sigma\tilde{\lambda}_i$.
– More generally, spectral similarity means that the two graphs will share many spectral or linear algebraic properties, e.g., effective resistances, resistance distances, etc.

- **Distance similarity.** If one assigns a length to every edge $e \in E$, then these lengths induce a shortest path distance between every $u, v \in V$. Thus, given a graph $G = (V, E, W)$, we can let $d : V \times V \to \mathbb{R}^+$ be the shortest path distance. Given this, we can define the following notion of similarity.

Definition 3. *Given two graphs, $G = (V, E, W)$ and $\tilde{G} = (V, \tilde{E}, \tilde{W})$, on the same vertex set, we say that G and $\tilde{G}$ are* σ-distance similar *if, for all pairs of vertices $u, v \in V$, we have that*

$$\frac{1}{\sigma}\tilde{d}(u, v) \leq d(u, v) \leq \sigma\tilde{d}(u, v).$$

Note that if $\tilde{G}$ is a subgraph if G, then $d_G(u, v) \leq d_{\tilde{G}}(u, v)$, since shortest-path distances can only increase. (Importantly, this does *not* necessarily hold if the edges of the subgraph are re-weighted, as they were done in the simple algorithm from the last class, when the subgraph is constructed; we will get back to this later.) In this case, a spanner is a subgraph such that distances in the other direction are not changed too much.

Definition 4. *Given a graph $G = (V, E, W)$, a* t-spanner *is a subgraph of G such that for all $u, v \in V$, we have that $d_{\tilde{G}}(u, v) \leq t d_G(u, v)$.*

There has been a range of work in TCS on spanners (e.g., it is known that every graph has a $2t + 1$ spanner with $O\left(n^{1+1/\epsilon}\right)$ edges) that isn't directly relevant to what we are doing.

We will be most interested in spanners that are trees or nearly trees.

Definition 5. *Given a graph $G = (V, E, W)$, a* spanning tree *is a tree includes all vertices in G and is a subgraph of G.*

There are various related notions that have been studied in different contexts: for example, minimum spanning trees, random spanning trees, and low-stretch spanning trees (LSSTs). Again, to understand some of the differences, think of a path versus a cycle. For today, we will be interested in LSSTs. The most extreme form of a sparse spanner is a LSST, which has only $n-1$ edges but which approximates pairwise distances up to small, e.g., hopefully polylog, factors.

26.4 Sparsified graphs

Here is an aside with some more details about sparsified graphs, which is of interest since this is the first step of our Laplacian-based linear equation solver algorithm. Let's define the following, which is a slight variant of the above.

Definition 6. *Given a graph G, a* (σ, d)-spectral sparsifier *of G is a graph $\tilde{G}$ such that*

1. *$\tilde{G}$ is σ-spectrally similar to G.*
2. *The edges of $\tilde{G}$ are reweighed versions of the edges of G.*
3. *$\tilde{G}$ has $\leq d|V|$ edges.*

Fact. Expanders can be thought of as sparse versions of the complete graph; and, if edges are weighted appropriately, they are spectral sparsifiers of the complete graph. This holds true more generally for other graphs. Here are examples of such results.

- SS showed that every graph has a $\left(1+\epsilon, O(\left(\frac{\log(n)}{\epsilon^2}\right)\right)$ spectral sparsifier. This was shown by SS with an effective resistance argument; and it follows from what we discussed last time: last time, we showed that sampling with respect to the leverage scores gives a subspace embedding, which preserves the geometry of the subspace, which preserves the Laplacian quadratic form, which implies the spectral sparsification claim.
- BSS showed that every n node graph G has a $\left(\frac{\sqrt{d}+1}{\sqrt{d}-1}, d\right)$-spectral sparsifier (which in general is more expensive to compute than running a nearly linear time solver). In particular, G has a $\left(1+2\epsilon, \frac{4}{\epsilon^2}\right)$-spectral sparsifier, for every $\epsilon \in (0,1)$.

Finally, there are several ways to speed up the computation of graph sparsification algorithms, relative to the strawman that we presented in the last class.

- Given the relationship between the leverage scores and the effective resistances and that the effective resistances can be computed with a nearly linear time solver, one can use the ST or KMP solver to speed up the computation of graph sparsifiers.
- One can use local spectral methods, e.g., diffusion-based methods from ST or the push algorithm of ACL, to compute well-balanced partitions in nearly linear time and from them obtain spectral sparsifiers.
- Union of random spanning trees. It is known that, e.g., the union of two random spanning trees is $O(\log(n))$-cut similar to G; that the union of $O\left(\log^2(n)/\epsilon^2\right)$ reweighed random spanning trees is a $1+\epsilon$-cut sparsifier; and so on. This suggests looking at spanning trees and other related combinatorial quantities that can be quickly computed to speed up the computation of graph sparsifiers. We turn to this next.

26.5 Back to Laplacian-based linear systems

KMP considered the use of combinatorial preconditions, an idea that traces back to Vaidya. They coupled this with a fact that has been used extensively in RLA: that only approximate leverage scores are actually needed in the sampling process to create a sparse sketch of L. In particular, they compute upper estimates of the leverage scores or effective resistance of each edge, and they compute these estimates on a modified graph, in which each upper estimate is sufficiently good. The modified graph is rather simple: take a LSST and increase its weights. Although the sampling probabilities obtained from the LSST are strictly greater than the effective resistances, they are not too much greater in aggregate. This, coupled with a rather complicated iterative preconditioning scheme, coupled with careful accounting with careful data structures, will lead to a solver that runs in $O\left(m \log(n) \log(1/\epsilon)\right)$ time, up to $\log\log(n)$ factors. We will discuss each of these briefly in turn.

Use of approximate leverage scores. Recall from last class that an important step in the algorithm was to use nonuniform importance sampling probabilities. In particular, if we sampled edges from the edge-incidence matrix with probabilities $\{p_i\}_{i=1}^m$, where each $p_i = \ell_i$, where ℓ_i is the effective resistance or statistical leverage score of the weighted edge-incidence matrix, then we showed that if we sampled $r = O\left(n \log(n)/\epsilon\right)$ edges, then it follows that

$$\| I - \left(SU_\Phi\right)^T \left(SU_\Phi\right) \|_2 \leq \epsilon,$$

from which we were able to obtain a good relative-error solution.

Using probabilities exactly equal to the leverage scores is overkill, and the same result holds if we use any probabilities p_i' that are "close" to p_i in the following sense: if

$$p_i' \geq \beta \ell_i,$$

for $\beta \in (0, 1]$ and $\sum_{i=1}^m p_i' = 1$, then the same result follows if we sample $r = O\left(n \log(n)/(\beta\epsilon)\right)$ edges, i.e., if we oversample by a factor of $1/\beta$. The key point here is that it is essential not to underestimate the high-leverage edges too much. It is, however, acceptable if we overestimate and thus oversample some low-leverage edges, as long as we don't do it too much.

In particular, let's say that we have the leverage scores $\{\ell_1\}_{i=1}^m$ and overestimation factors $\{\gamma_i\}_{i=1}^m$, where each $\gamma_i \geq 1$. From this, we can consider the probabilities

$$p_i'' = \frac{\gamma_i \ell_i}{\sum_{i=1}^m \gamma_i \ell_i}.$$

If $\sum_{i=1}^m \gamma_i \ell_i$ is not too large, say $O\left(n \log(n)\right)$ or some other factor that is only slightly larger than n, then dividing by it (to normalize $\{\gamma_i \ell_i\}_{i=1}^m$ to unity to be a probability distribution) does not decrease the probabilities for the high-leverage components too much, and so we can use the probabilities p_i'' with an extra amount of oversampling that equals $\frac{1}{\beta} = \sum_{i=1}^m \gamma_i \ell_i$.

Use of LSSTs as combinatorial preconditioners. Here, the idea is to use a LSST, i.e., use a particular form of a "combinatorial preconditioning," to replace $\ell_i = \ell_{(uv)}$ with the stretch of the edge (uv) in the LSST. Vaidya was the first to suggest the use of spanning trees of L as building blocks as the base for preconditioning matrix B. The idea is then that the linear system, if the constraint matrix is the Laplacian of a tree, can be solved in $O(n)$ time with Gaussian elimination.

(Adding a few edges back into the tree gives a preconditioned that is only better, and it is still easy to solve.) Boman-Hendrickson used a LSST as a stand-along preconditioner. ST used a preconditioner that is a LSST plus a small number of extra edges. KMP had additional extensions that we describe here.

Two question arise with this approach.

- Q1: What is the appropriate base tree?
- Q2: Which off-tree edges should added into the preconditioner?

One idea is to use a tree that concentrates that maximum possible weight from the total weight of the edges in L. This is what Vaidya did; and, while it led to good result, the results weren't good enough for what we are discussing here. (In particular, note that it doesn't discriminate between different trees in unweighted graphs, and it won't provide a bias toward the middle edge of a dumbbell graph.) Another idea is to use a tree that concentrates mass on high leverage/influence edges, i.e., edges with the highest leverage in the edge-incidence matrix or effective resistance in the corresponding Laplacian.

The key idea to make this work is that of *stretch*. To define this, recall that for every edge $(u, v) \in E$ in the original graph Laplacian L, there is a unique "detour" path between u and v in the tree T.

Definition 7. *The* stretch *of the edge with respect to T equals the distortion caused by this detour.*

In the unweighted case, this stretch is simply the length of the tree path, i.e., of the path between nodes u and v that were connected by an edge in G in the tree T. Given this, we can define the following.

Definition 8. *The* total stretch *of a graph G and its Laplacian L with respect to a tree T is the sum of the stretches of all off-tree edges. Then, a* low-stretch spanning tree (LSST) *T is a tree such that the total stretch is low.*

Informally, a LSST is one such that it provides a good "on average" detours for edges of the graph, i.e., there can be a few pairs of nodes that are stretched a lot, but there can't be too many such pairs.

There are many algorithms for LSSTs. For example, here is a result that is particularly relevant for us.

Theorem 3. *Every graph G has a spanning tree T with total stretch $\tilde{O}(m \log(n))$, and this tree can be found in $\tilde{O}(m \log(n))$ time.*

In particular, we can use the stretches of pairs of nodes in the tree T in place of the leverage scores or effective resistances as importance sampling probabilities: they are larger than the leverage scores, and there might be a few that much larger, but the total sum is not much larger than the total sum of the leverage scores (which equals $n - 1$).

Paying careful attention to data structures, bookkeeping, and recursive preconditions. Basically, to get everything to work in the allotted time, one needs the preconditioner B that is extremely good approximation to L and that can be computed in linear time. What we did in the

last class was to compute a "one step" preconditioner, and likely any such "one step" preconditioned won't be substantially easier to compute that solving the equation; and so KMP consider recursion in the construction of their preconditioner.

- In a recursive preconditioning method, the system in the preconditioned B is not solved exactly but only approximately, via a recursive invocation of the same iterative method. So, one must find a preconditioned for B, a preconditioned for it, and so on. This gives s multilevel hierarchy of progressively smaller graphs. To make the total work small, i.e., $O(kn)$, for some constant k, one needs the graphs in the hierarchy to get small sufficiently fast. It is sufficient that the graph on the $(i+1)^{th}$ level is smaller than the graph on the i^{th} level by a factor of $\frac{1}{2k}$. However, one must converge within $O(kn)$. So, one can use CG/Chebyshev, which need $O(k)$ iterations to converge, when B is a k^2-approximation of L (as opposed to $O(k^2)$ iterations which are needed for something like a Richardson's iteration).

So, a LSST is a good base; and a LSST also tells us which off-tree edges, i.e., which additional edges from G that are not in T, should go into the preconditioner.

- This leads to an $\tilde{O}\left(m \log^2(n) \log(1/\epsilon)\right)$ algorithm.

If one keeps sampling based on the same tree and does some other more complicated and careful stuff, then one obtains a hierarchical graph and is able to remove the the second log factor to yield a potentially practical solver.

- This leads to an $\tilde{O}\left(m \log(n) \log(1/\epsilon)\right)$ algorithm.

See the BSST and KMP papers for all the details.