CS 798 - Algorithmic Spectral Graph Theory, Fall 2015, Waterloo

Lecture 13: Laplacian solver

Today we discuss a near linear time algorithm for solving Laplacian system.

Simple SDD solver

Given a system of linear equations $Ax=b$, if A is symmetric and diagonally dominant ($a_{ii} > \sum_{j \neq i} |a_{ij}| \; \forall i$), then there is a simple reduction from this problem to solving a Laplacian system $Lx=b$.

The first near linear time solver is designed by Spielman and Tang, who along the way introduced the tools for spectral sparsification, local graph partitioning, low-stretch spanning trees and combinatorial preconditioning.

The resulting algorithm is quite complicated and the hidden polylog factor is quite large.

The algorithm that we discuss today doesn't need to do spectral sparsification and local graph partitioning, and its numerical steps are very simple. Crucial in the analysis is the low-stretch spanning tree only.

Problem setup

Given a Laplacian system $Lx=b$, as we have seen before, it corresponds to computing the voltage vector for some electric flow problem.

The simple algorithm tries to compute the flow vector (on the edges) for the electric flow problem instead.

Recall that electric flow is the unique flow that minimizes the energy subject to the flow conservation constraints.

So, the problem of computing the electric flow can be formulated as

$$\min \quad f^T R f = \sum_{e \in E} r_e f_e^2 \qquad \text{where } r_e \text{ is the resistance and } f_e \text{ is the flow on edge } e$$

$$\text{s.t.} \quad B^T f = b \qquad \text{where } B \text{ is the } m \times n \text{ edge-vertex incidence matrix} \quad m\overset{n}{\begin{pmatrix} + & - \\ - & + \\ + & - \end{pmatrix}}$$

(each row two nonzeros)

Note that $b \in \mathbb{R}^n$ is the flow-demand vector, where $b_i > 0$ if i is a source, $b_i < 0$ if i is a sink.

The constraint $B^T f = b$ enforces that the total outgoing flow minus the total incoming flow at each vertex i is exactly equal to b_i in the flow-demand vector. So, these are the flow conservation constraints.

In the simple algorithm, we starts with an initial flow, satisfying the flow conservation constraints.

Then, we keep doing some simple operations to drive down the total energy, while satisfying conservation.

Recall from L12 that a flow minimizes the total energy if and only if it satisfies the potential law.

So, we can also try to maintain the flow conservation constraints, while driving the flow to satisfy the potential law (Ohm's law), i.e. there exists $v \in \mathbb{R}^n$ such that $v_i - v_j = f_{ij} \cdot r_{ij}$

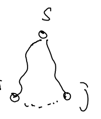
It doesn't seem easy to measure the progress of satisfying the potential law, but there is an equivalent way of looking at the potential law that is easier to work with.

<u>Claim</u> (cycle law) The potential law is satisfied if and only if the following cycle law is satisfied: For any cycle $C = i_1, i_2, \ldots, i_{|C|} = i_1$, we have $\sum_{j=1}^{|C|-1} f_{i_j i_{j+1}} \cdot r_{i_j i_{j+1}} = 0$.

<u>proof</u> One direction is easy. If there exists $v \in \mathbb{R}^n$ such that $f_{ij} \cdot r_{ij} = v_i - v_j$ for all $i, j \in V$, then $\sum_{ij \in C} f_{ij} \cdot r_{ij} = \sum_{ij \in C} (v_i - v_j) = 0$.

The other direction is not hard. Given the cycle law is satisfied, we just pick an arbitrary (root) vertex s and set $v_s = 0$, say use a BFS tree to define the voltages for other vertices, so that the potential law is satisfied for all the tree edges.

To see that the potential law is also satisfied for non-tree edges, consider the cycle in the figure and see that edge ij satisfies the potential law because the cycle law is satisfied (as each path going to i defines the same voltage on i). $\square$



Okay, with the cycle law in mind, the strategy of the simple algorithm is as follows.

- Compute an initial flow f_0 (satisfying the conservation constraints)
- Find a cycle that violates the cycle law. Fixes it to obtain f_t. Repeat.

This is the algorithm on a high level. First, we specify how to fix a cycle.

Recall that we want to maintain the flow conservation constraints. So, if we find a violating cycle, we just add or subtract the flow on the cycle by the same amount to satisfy the cycle.

Given a cycle C, let $\Delta_C(f) = \sum_{e \in C} r_e f_e$.

If $\Delta_C(f) \neq 0$, we choose ε so that $\sum_{e \in C} r_e (f_e - \varepsilon) = 0 \Rightarrow \varepsilon = \Delta_C(f) / \sum_{e \in C} r_e$.

The good news is that by doing this simple operation, the total energy is decreasing.

<u>Lemma</u> (energy improvement) If $\Delta_C(f) \neq 0$, let $\varepsilon = -\Delta_C(f) / \sum_{e \in C} r_e$, then $\mathcal{E}(f + \varepsilon \chi_C) - \mathcal{E}(f) = -\Delta_C^2(f) / \sum_{e \in C} r_e$, where $\mathcal{E}(f)$ denotes the total energy of flow $\vec{f}$.

then $\mathcal{E}(f+\varepsilon\chi_c) - \mathcal{E}(f) = -\Delta_c^2(f) / \sum_{e\in C} r_e$, where $\mathcal{E}(f)$ denotes the total energy of flow $\vec{f}$.

<u>proof</u> $\mathcal{E}(f+\varepsilon\chi_c) - \mathcal{E}(f) = \sum_{e\in C} r_e(f_e+\varepsilon)^2 - \sum_{e\in C} r_e f_e^2 = \sum_{e\in C} 2\varepsilon r_e f_e + \varepsilon^2 \sum_{e\in C} r_e$.

Putting $\varepsilon = -\Delta_c(f) / \sum_{e\in C} r_e$ gives the result.

(In fact, setting ε this way minimizes the energy among all ε, so that's why the algorithm enforces to satisfy C.) $\square$

The above lemma shows that the algorithm makes sense. By fixing a violated cycle, we are guaranteed that the total energy decreases while the flow conservation maintained, so intuitively it will eventually converge to the electric flow.

To bound its running time, we need to specify which violated cycles to fix and bound the convergence rate of the algorithm, and also discuss efficient implementations.

<u>Tree and cycles</u>

There are too many cycles to worry about. In principle, from the energy improvement lemma, it would be good to choose the cycle with largest violation. but it is computationally too expensive.

One important idea in the simple algorithm is to fix a (good) tree and only keep track of a small number of cycles.

Given a spanning tree T, for any edge e, it defines a unique path P_e on the tree

For $e\in G-T$, let $C_e := P_e \cup \{e\}$ be the fundamental cycle involving e, and $R_e := \sum_{f\in C_e} r_e$.

For our problem, they are fundamental because one can show that if all the fundamental cycles are fixed (satisfying the cycle law), then all cycles are fixed.

(We don't need this fact explicitly, so we just sketch the argument. Consider any cycle. Replace each edge by the tree path. By the cycle law on the fundamental cycles, the paths have the same "imbalance" as the edges. Summing over tree paths over a cycle will give us zero, and so is the sum of the original cycle.)

The simple algorithm in a higher resolution is as follows.

① Find an initial flow solution f_0.

② Choose a (good) tree T.

③ repeat the following steps for K steps

- choose a fundamental cycle C_e with $\Delta_{C_e}(f_t) \neq 0$.

- set $f_{t+1} = f_t + \varepsilon \chi_{C_e}$ where $\varepsilon = -\Delta_{C_e}(f_t) / R_e$ where $R_e := \sum_{f \in C_e} r_f$.

④ return f_K and approximate voltage vector.

Dual Problem

To bound the convergence rate, we define a dual program to measure the progress to optimality. It also turns out that the dual is useful in defining the voltage vector in our final solution.

Lemma (dual) The program $\max_{v \in \mathbb{R}^n} 2v^Tb - v^TLv$ is dual to $\min_{f \in \mathbb{R}^{|E|}} f^TRf$ s.t. $B^Tf = b$,

and their optimal values are the same.

~~proof~~ Consider $\min_{f \in \mathbb{R}^m} f^TRf$ s.t. $B^Tf = b$, the primal program.

The Lagrangian function $g(u) : u \in \mathbb{R}^n$ is $\min_{f \in \mathbb{R}^m} f^TRf - u^T(B^Tf - b)$.

Note that $g(u) \leq OPT_{primal}$ because the primal optimal solution satisfies $B^Tf - b = 0$,

and so $g(u)$ is a lower bound on the optimal primal value.

To obtain the best lower bound, we consider $\max_{u \in \mathbb{R}^n} g(u) = \max_{u \in \mathbb{R}^n} \min_{f \in \mathbb{R}^m} f^TRf - u^TB^Tf + u^Tb$.

Given $u \in \mathbb{R}^n$, $\min_{f \in \mathbb{R}^m} \left(f^TRf - u^TB^Tf\right)$ is attained when the gradient w.r.t. f is zero.

The gradient is $2Rf - Bu$. Rewrite $u = 2v$ so that v satisfies the voltage law.

Then the gradient is zero when $Rf = Bv \Rightarrow f = R^{-1}Bv = WBv$, where W is diagonal with $w_e = \frac{1}{r_e}$.

The dual program is just $\max_{v \in \mathbb{R}^n} g(2v) = f^TRf - 2v^TB^Tf + 2v^Tb$

$$= v^TBWRWBv - 2vB^TWBv + 2v^Tb$$

$$= 2v^Tb - v^TLv \qquad (\text{as } L = B^TWB \text{ and } R = W^{-1}).$$

For this pair of primal dual programs, it is easy to see that the electric flow vector and the corresponding voltage vector are feasible primal and dual solution with the same objective value ($f^TRf = v^TLv$ = total energy). □

Distance to Optimality

To bound the distance of a primal flow solution to optimality, we construct a dual voltage solution and bound the difference between the primal and the dual objective values, which serves as an upper bound to the distance.

From the proof of the cycle law, if all the cycles are satisfied, then we can use a tree to define the voltage vector.

So, the idea here is to use the tree T to define a dual feasible solution.

Fix a (root) vertex s, define $v_s = 0$, and for each vertex $i \neq s$, define $v_i = \sum_{e \in P_{is}} r_e f_e$.

We call this the tree induced voltages, and $gap(f,v)$ the difference beteen the primal and dual values.

Lemma (gap) For flow vector $f \in \mathbb{R}^m$ and its tree induced voltages $v \in \mathbb{R}^n$, we have

$$\mathcal{E}(f) - \mathcal{E}(f_{OPT}) \le gap(f,v) = \sum_{e \in E-T} \frac{\Delta_{C_e}^2(f)}{r_e}.$$

~~proof~~ $gap(f,v) = f^T R f - (2v^T b - v^T L v) = f^T R f - 2v^T B^T f + v^T L v$ (since $B^T f = b$, flow conservation)

$$= \sum_{ij \in E} r_{ij} f_{ij}^2 - 2 \sum_{ij \in E} (v_i - v_j) f_{ij} + \sum_{ij \in E} \frac{1}{r_{ij}} (v_i - v_j)^2$$

$$= \sum_{ij \in E} \frac{1}{r_{ij}} \left(r_{ij} f_{ij} - (v_i - v_j) \right)^2.$$

Recall that $v_i = \sum_{e \in P_{is}} r_e f_e$ by the definition of tree induced voltages,

we have $v_i - v_j = \sum_{e \in P_{is}} r_e f_e - \sum_{e \in P_{js}} r_e f_e = \sum_{e \in P_{ij}} r_e f_e$.

For $ij \in T$, $r_{ij} - (v_i - v_j) = 0$, and for $ij \notin T$, $r_{ij} f_{ij} - (v_i - v_j) = r_{ij} f_{ij} - \sum_{e \in P_{ij}} r_e f_e = r_{ij} f_{ij} + \sum_{e \in P_{ji}} r_e f_e = \Delta_{C_e}(f)$.

Therefore, $gap(f,v) = \sum_{e \in E-T} \frac{\Delta_{C_e}^2(f)}{r_e}$. □

Probability distribution, low-stretch spanning trees

From the energy improving lemma, if we fix the fundamental cycle C_e, then the energy decreases by $\frac{\Delta_{C_e}^2(f)}{R_e}$ where R_e is total resistance along the fundamental cycle.

From the tree gap lemma, the distance to optimality is at most $\sum_{e \in E-T} \frac{\Delta_{C_e}^2(f)}{r_e}$.

In principle, we can check which fundamental cycle contributes most to the summation and fix it, but that takes too much time.

The idea here is to sample from a probability distribution p over the edges so that the expected decrease of energy is proportional to the distance to optimality.

Let $\alpha = \sum_{e \in E} \frac{R_e}{r_e}$ and $p_e = \left(\frac{R_e}{r_e}\right) \cdot \frac{1}{\alpha}$.

Then, the expected decrease of the energy is equal to $\sum_{e \in E-T} p_e \cdot \frac{\Delta^2 c_e(f)}{R_e} = \sum_{e \in E-T} \frac{1}{\alpha} \frac{R_e}{r_e} \cdot \frac{\Delta^2 c_e(f)}{R_e}$

$$= \frac{1}{\alpha} \sum_{e \in E-T} \frac{\Delta^2 c_e(f)}{r_e} = \frac{1}{\alpha} \text{gap}(f, v).$$

Therefore, the gap decreases geometrically, i.e. $E[\text{gap}(f_t, v_t)] \le (1 - \frac{1}{\alpha})^t E[\text{gap}(f_0, v_0)]$.

The convergence rate depends on α, the smaller the faster.

What is α? It is equal to $\sum_{e \in E} \frac{R_e}{r_e}$, where each term is the ratio of the resistance of the path and the resistance of the edge.

Think of r_e as the distance of the edge. Then we want to minimize the average ratio of the distance of the tree paths and the distance of the edges.

It is a natural object that people have studied before.

Given distances d_e on the edges and a tree T, the stretch of an edge is defined as $\sum_{f \in P_e} \frac{d_f}{d_e}$ where P_e is the unique tree path connecting the two endpoints of e.

We want the stretch to be small, so that just using the tree (the simplest topology) doesn't increase the shortest path distance much.

There is a long line of research and the current best known result is the following.

<u>Theorem</u> (Abraham - Neiman) For any graph G, there exists a spanning tree with total stretch $O(m \log n \log\log n)$ and it can be found in $O(m \log n \log\log n)$ time.

Using this theorem with $r_e = d_e$, we get that $\alpha = O(m \log n \log\log n)$.

<u>Initial solution and number of iterations</u>

We are close. Just need to find a good enough initial solution and compute the number of iterations using the convergence rate derived above (using a low stretch spanning tree).

An easy initial solution that is not too bad can be obtained from a low stretch spanning tree.

Given a tree T, we can define a flow solution by using only the edges in T (e.g. if we just send one unit of flow from s to t, just send it along the tree path).

The full algorithm is as follows.

① Compute a low stretch spanning tree T and define f_0 as the flow solution defined on T

② Repeat k steps

- random sample a non-tree edge e with prob. $\frac{1}{\alpha}\frac{R_e}{r_e}$ and fix the fundamental cycle C_e.

③ Return the tree induced voltage solution of f_k.

<u>Lemma</u> $\mathcal{E}(f_0) \le \alpha \cdot \mathcal{E}(f_{OPT})$ where α is the total stretch of the spanning tree T.

<u>proof</u> Let f_{OPT} be the optimal solution, i.e. the electric flow.

To compare it with the tree solution, for each edge ij, just send $f_{OPT}(ij)$ flow from i to j along the tree T.

$$\text{Then } \mathcal{E}(f_0) = \sum_{e\in T} r_e \Big(\sum_{e'\in E:\, e\in P_{e'}} f_{OPT}(e')\Big)^2 \overset{\swarrow\ \text{Cauchy-Schwarz}}{\le} \sum_{e\in T}\Big(\sum_{e'\in E:\, e\in P_{e'}} \frac{r_e}{r_{e'}}\Big)\Big(\sum_{e'\in E:\, e\in P_{e'}} r_{e'} f_{OPT}^2(e')\Big)$$

$$\le \sum_{e\in T}\Big(\sum_{e'\in E:\, e\in P_{e'}} \frac{r_e}{r_{e'}}\Big)\cdot \mathcal{E}(f_{OPT})$$

$$(\text{change of summation}) \quad = \Big(\sum_{e'\in E}\ \sum_{e\in P_{e'}} \frac{r_e}{r_{e'}}\Big)\cdot \mathcal{E}(f_{OPT}) = \alpha\cdot\mathcal{E}(f_{OPT}). \quad \square$$

Therefore, the initial gap is at most $\alpha\cdot\mathcal{E}(f_{OPT})$.

By repeating k times, the expected gap is at most $(1-\frac{1}{\alpha})^k \cdot \alpha\cdot\mathcal{E}(f_{OPT})$.

By setting $k = \alpha \ln\frac{\alpha}{\beta}$ for some $\beta>0$, we get the gap is at most $\beta\cdot\mathcal{E}(f_{OPT})$.

Hence the flow is an $(1+\beta)$-approximation in the primal program.

Let $\|x\|_L = \sqrt{x^T L x}$.

$$\text{Then, } \|v_k - v_{OPT}\|_L^2 = \|v_k - L^{\dagger} b\|_L^2 = (v_k - L^{\dagger}b)^T L (v_k - L^{\dagger} b) = \underbrace{v_k^T L v_k - 2 v_k^T L L^{\dagger} b}_{-\text{dual}} + \underbrace{v_{OPT}^T L v_{OPT}}_{OPT}$$

$$\le \text{gap}(f_k, v_k) \le \beta\cdot\mathcal{E}(f_{OPT}) = \beta \|v_{OPT}\|_L^2.$$

So, $\|v_k - v_{OPT}\|_L^2 \le \beta\|v_{OPT}\|_L^2$, after $\alpha\ln\frac{\alpha}{\beta} = O\Big(m\log n \log\log n\cdot \log\Big(\frac{m\log n\log\log n}{\beta}\Big)\Big)$

$= \tilde{O}(m\log n\log\frac{n}{\beta})$ iterations, hiding loglog terms.

<u>One iteration : data structures</u>

To get an $(1+\beta)$-approximation, the algorithm needs $\tilde{O}(m)$ iterations, which looks already too many.

The key feature of this approach is that each iteration can be implemented in $O(\log n)$ time, which makes the whole algorithm still runs in $\tilde{O}(m\log^2 n\log\frac{n}{\beta})$ time.

In each iteration, we sample a non-tree edge with probability P_{ij} (which can be pre-computed), and then query $\sum_{e \in P_{ij}} r_e f_e$ to determine ε and then update $f_e \leftarrow f_e + \varepsilon$ for each $e \in C_{ij}$.

First, we use an array for the flow on the non-tree edge.

The tree data structure is quite simple.

Given the (low-stretch) tree, choose an arbitrary vertex s as root.

We need a data structure to support two operations:

- query(i): return $\sum_{e \in P_{si}} r_e f_e$.
- update(i, ε): set $f_e \leftarrow f_e + \varepsilon$ for every $e \in P_{si}$.

With these two operations, we can compute $\sum_{e \in P_{ij}} r_e f_e$ by returning query(j) − query(i),

and perform $f_e \leftarrow f_e + \varepsilon$ for each $e \in C_{ij}$ by calling update(j, ε) and update$(i, -\varepsilon)$.

Now, suppose for the moment that T is a balanced binary tree with $O(\log n)$ depth to the root.

Then, these two operations can be straightforwardly implemented in $O(\log n)$ time.

But when the tree is not balanced, it could take linear time for this straightforward implementation.

The idea is to use the center c to break the tree into components such that each component size

is at most $n/2$.

Then the center becomes the root of the subtrees, and we find centers in the subtrees to

break them recursively, so that each component is of constant size in $O(\log n)$ depth.

When doing this tree decomposition, we store information about the overlap of the path P_{sc} and

all other paths P_{si}.

After that, when we call query$_s(i)$, we return query$_s(d)$ + query$_d(i)$, and query$_d(i)$ is answered

recursively on the subtree that contains i with d as the root.

Similarly, when we call update$_s(i, \varepsilon)$, we call update$_s(d, \varepsilon)$ and update$_d(i, \varepsilon)$, where update$_d(i, \varepsilon)$

is implemented recursively.

This is the main idea. Details can be found in the paper (< two pages).

<u>References and discussions</u>

The material is from the paper " A simple, combinatorial algorithm for solving SDD systems in near

linear time", by Kelner, Orecchia, Sidford and Zhu.

There is a more general perspective of this algorithm as a special case of the randomized Kaczmarz algorithm, which picks a random (equality) constraint and projects the current solution to the hyperplane that satisfies that constraint.

The convergence analysis of the randomized Kaczmarz method doesn't need to use the dual program, and its convergence rate is proportional to the "average condition number" and the role of the low stretch spanning tree is as a "preconditioner" to decrease the "average condition number".

See the paper and also the monograph "Lx=b" by Vishnoi for more discussions on this general perspective.

There are many papers on Laplacian solvers since Spielman-Teng; see Spielman's homepage for a page about Laplacian solver and related topics.

The faster algorithm only takes $\tilde{O}(m\sqrt{\log n})$ time, even faster than sorting m numbers.