CSC 5160 Spectral Algorithms  
Week 13: Laplacian Solver, Multiplicative Update Method

Today we will finally complete the near linear time Laplacian solver, by putting the techniques developed in the last three weeks, including the conjugate gradient method, preconditioning, low stretch spanning trees, and sampling by effective resistances.

Then we will switch topic, to discuss a simple technique that is useful in various areas, called the multiplicative weight update method. Next week we will apply it to the maximum flow problem, along with electric flow and Laplacian solver.

Overview

Our goal is to solve $L_G x = b$ efficiently.

This can be done in $O(\sqrt{n} m)$ time by the conjugate gradient method (see week 10), where $\kappa = \frac{\lambda_{\text{max}}}{\lambda_{\text{min}}}$ is the condition number of the matrix $L_G$, but $\kappa$ could be large.

The next idea is to do preconditioning: solve the equivalent system $L_H^{-1} L_G x = L_H^{-1} b$. This can be done more efficiently than solving $L_G x = b$ if the following three conditions are satisfied (see week 11):

1. $H$ can be constructed efficiently.
2. $L_H^{-1} y = y$ can be solved efficiently for any $y$.
3. $\kappa(L_H^{-1} L_G)$ is (much) smaller than $\kappa(L_G)$.

For Laplacians, we can choose $H$ to be a low stretch spanning tree (see week 11), then $H$ can be constructed in $O(m)$ time, $L_H^{-1} L_G$ can be solved in $O(n)$ time, and $\kappa(L_H^{-1} L_G) \leq O(1)$. Putting together gives an $O(m^{1.5})$ time algorithm for solving Laplacian systems (can be improved to $O(m^{1.3})$).

Last week we talked about spectral sparsification by effective resistances. This gives $H$ with $O(n \log n)$ edges and $(1+\epsilon)G \preceq H \preceq (1+\epsilon)G$ which implies that $\kappa(L_H^{-1} L_G) \leq O(1)$. If we use $H$ and solve $L_H^{-1} L_G x = L_H^{-1} b$ then the conjugate gradient method will converge in a constant number of iterations.

There are two problems, however, in using such a spectral sparsifier:

- We don’t know how to construct $H$ efficiently. We need to know the effective resistances in order to do
the random sampling, but computing effective resistances seems to be as least as hard as solving Laplacian systems.

- We don't know how to solve $LW = 0$ efficiently, although we know that $H$ has $O(n \log n)$ edges.

In hindsight, we already have all the ingredients are there, and we just need two more ideas to put them together: $H$ will be a low stretch spanning tree plus some random edges.

---

**Sparsification by Oversampling [1]**

In the random sampling algorithm for constructing spectral sparsifiers, we set the sampling probability of an edge $e$ to be $p_e = \frac{w_e}{R_e}$, where $w_e$ is the weight of edge $e$ and $R_e$ is the effective resistance of the edge $e$. In the analysis, this is to guarantee that $\sqrt{\frac{2}{\pi}} \frac{w_e}{R_e} \geq \sqrt{n-1} = n$ and hence we can set the number of samples $g = O(n \log n)$ to ensure that the error is small. (Recall that the error is bounded by $O(M \sqrt{\frac{\log k}{g}})$ by the matrix concentration result.)

A simple but important observation is that "oversampling" would also work. Suppose we set $p_e = \frac{R_e}{w_e}$ and let $t = \frac{2}{\pi} p_e$. Then it follows immediately that $M \leq \sqrt{t}$ and thus $g = O(t \log t)$ samples would make the error small.

**Theorem (Oversampling)** Let $G$ be a weighted undirected graph. Assume $p_e = \frac{R_e}{w_e}$. Then the random sampling algorithm can be used to construct a graph $G'$ satisfying $G \leq G' \leq 4G$, and $G'$ has $O(t \log t)$ edges with high probability where $t = \frac{2}{\pi} p_e$.

So, if we could find $p_e$ for each edge $e$ efficiently such that $p_e \geq \frac{R_e}{w_e}$ and at the same time $t = \frac{2}{\pi} p_e$ is small, then it will give an efficient algorithm to find a sparse spectral sparsifier.

**Sampling by Stretch**

But how should we choose $p_e$? Recall that $R_e = \frac{w_e}{w_e}$ is defined as the resistance of the edge $e$, upon which effective resistance is defined. By the Rayleigh's monotonicity law, $R_e \leq \frac{w_e}{w} \frac{w}{w}$, where $p$ is a path connecting the two endpoints of $e$.

So, $\frac{w_e}{w} = \frac{w_e}{w_e} \frac{w}{w} = \frac{w_e}{w_e} \frac{w}{w}$, which is very similar to the definition of stretch.

This observation leads us to the following sparsification algorithm. Find a low stretch spanning tree $T$,
with total stretch $\tilde{O}(m)$ in near linear time, using $r_e$ as the length of edge $e$. Set $p_e = \text{stretch}_k(e) = \frac{2(\text{che})}{r_e} \geq k \cdot r_e$. Then, by the oversampling theorem, the random sampling algorithm will produce a subgraph $G'$ which is a constant approximation of $G$ with $\tilde{O}(\text{total stretch}) = \tilde{O}(m)$ edges. This is not useful since $G'$ is not sparser than $G$. Here is the modification that trades condition number for sparsity.

**Incremental Sparsifier**

1. Use $r_e$ as length to find a spanning tree $T$ with total stretch $O(m \log n \log \log n)$ in $O(n \log n \log n)$ time.
2. Let $T'$ be $T$ scaled by a factor of $k$, i.e., $w' = kw$ and thus $r_e' = \frac{r_e}{k}$ for every edge $e$ in $T$.
3. Let $G'$ be the graph obtained from $G$ by replacing $T$ by $T'$.
4. For each edge $e \in E'$, set $p_e = \text{stretch}_k(e)$ and run the random sampling algorithm to obtain a subgraph $H$ of $G'$.
5. Return $2H$.

**Theorem** The algorithm returns $H$ such that $G \preceq H \preceq 3kG$ with $n-1 + O(\frac{\log n}{k})$ edges in near linear time.

**Proof** First we bound the condition number.

Since we scale the tree edge weight by a factor of $k$, we have $G \preceq G' \preceq kG$.

Since $\text{stretch}_k(e) \geq kr_e$ in $G'$, by the oversampling theorem, $G \preceq G' \preceq H \preceq 3G' \preceq 3kG$.

So, $\chi(L_H L_G) \leq 3k$.

Next we bound the number of edges in $H$. Each edge in $T$ is sampled with probability one.

Each non-tree edge is sampled with probability $\text{stretch}_k(e) = \frac{\text{stretch}_k(e)}{k}$, by the oversampling theorem, the number of non-tree edges is $O\left(\frac{\sum \text{stretch}_k(e)}{k}\right) = \tilde{O}(\frac{\log n}{k})$.

Therefore, the total number of edges in $H$ is $n-1 + O(\frac{\log n}{k})$.

Finally, note that all the stretches can be computed by Tarjan's lowest common ancestor algorithm in $O(m)$ time, and thus the whole algorithm can be implemented in $\tilde{O}(m)$ time.

The sparsifier $H$ is just the union of a low stretch spanning tree plus some random edges.

---

**Near Linear Time Laplacian Solver** [7]

Using the above theorem, we can construct a sparsifier $H$ with $\chi(L_H L_G) = k$ and $O(n-1 + \frac{\log n}{k})$.
edges in near linear time. So, by allowing a larger condition number, in return we gain that the
construction time is near linear, satisfying the first condition of a good preconditioner.

But the second problem is still there: how to solve \( L_{12} y = \tilde{z} \) efficiently?

The final idea is to solve \( L_{13} y = \tilde{z} \) recursively, just like what we did for \( L_{4} x = b \).

Recall that to solve \( L_{4} x = b \), we solve \( L_{13} L_{4} x = L_{13} b \) instead, in \( \sqrt{K(L_{13} L_{4})} \) iterations.

Every time we need to compute \( L_{13} \), we solve \( L_{13} y = \tilde{z} \), and how we solve this problem recursively,

i.e. find a sparsifier \( H_{1} \) of \( H_{1} \), solve \( L_{13} L_{4} y = L_{13} \tilde{z} \), in \( \sqrt{K(L_{13} L_{4})} \) iterations. In every iteration,

we need to solve \( L_{13} \tilde{z} \), and we solve \( L_{13} \tilde{z} \) recursively.

Set \( k = \log^3 n \).

Then \( K(L_{13} L_{4}) \leq \log^3 n \), and thus there are \( O(\log^3 n) \) iterations in the highest level.

Then \( H_{1} \) has only \( n-1 + O \left( \frac{\log^2 n}{\log^3 n} \right) = n-1 + O(\frac{m}{\log^2 n}) \) edges.

By doing Gaussian elimination on degree one and degree two vertices, we can eliminate the degree one
and degree two vertices while not increasing the number of nonzeros in the resulting matrix.

Assume there are \( n-1+j \) edges in \( H_{1} \). Then, after eliminating degree one and degree two vertices, there
are at most \( 2j-3 \) edges and \( 2j-2 \) vertices. So, after Gaussian elimination, we can reduce \( H_{1} \)
to a graph with \( O(\frac{m}{\log^3 n}) \) vertices and \( O(\frac{m}{\log^2 n}) \) edges, with only \( O(m) \) additional work.

Then we solve this modified \( H_{1} \) recursively.

So, the total running time is \( T(m) = \sum_{k} \left( T \left( \frac{m}{\log^3 n} \right) + O(m) \right) = \log^3 n \left( T \left( \frac{m}{\log^2 n} \right) + O(m) \right) = O(m) \).

Remarks

- The whole algorithm is not that complicated. The most difficult step is in finding low stretch spanning
trees, for which we did not explain, as the details are quite involved and the techniques are not that
relevant to this course.

- There is a downloadable code in the homepage ofannis Kontis, called "combinatorial multigrid." 9

- Local graph partitioning was also invented for designing near linear time Laplacian solver.

- The solver is further improved to have complexity essentially \( O(m \log n) \).

- Any symmetric and diagonally dominant system \( \left( \begin{array}{c} 1 & a_{12} & a_{13} \\ a_{21} & 2 & a_{23} \\ a_{31} & a_{32} & 1 \end{array} \right) \) can be reduced to a Laplacian system,

  by doubling the variables (each \( x_{i} \) becomes \( x_{i}^{+} \) and \( x_{i}^{-} \)); details omitted.
The Multiplicative Weights Update Method \([2,3]\)

First we illustrate the multiplicative update method by a simple example.

Consider the following setting: There are \(n\) experts \(E_1, \ldots, E_n\) who predict the stock market every day. The predictions are binary valued.

Suppose we do not know the stock market. We can only hope to do as well as the best expert \((in hindsight)\).

Notice that we need to make an online decision every day.

Of course it is not possible, but surprisingly we can do almost as well as the best expert.

**Weighted Majority Algorithm**

Initially \(w_i = 1\) for each expert \(i\).

For each day

- predict according to the weighted majority of the experts \((i.e.\ make\ the\ decision\ with\ more\ weights)\).
- for each expert who predicts wrongly, update \(w_i \leftarrow (1-\varepsilon)w_i\).

This is the algorithm. The idea of the analysis is simple: every time we make a mistake, the total weight decreases by a constant factor. So, if we make many mistakes, then the total weight is small, which also implies that the best expert’s weight is small, and so even the best expert made a few mistakes.

Let \(W = \sum_i w_i\) be the total weight. Let \(M\) be the number of mistakes we made. Let \(m\) be the number of mistakes that the best expert made.

**Claim** \(M \leq 2(1+\varepsilon)M + 2\ln n / \varepsilon\).

**Proof** Every time we make a mistake, at least half the total weight is decreased by a factor of \((1-\varepsilon)\).

So the total weight is decreased by a factor of \((1-\varepsilon)^M\).

Thus the total weight after \(M\) mistakes by the algorithm is at most \((1-\varepsilon)^M\).

On the other hand, if the best expert made only \(m\) mistakes, then the total weight \(\geq (1-\varepsilon)^m\).

Therefore, \((1-\varepsilon)^m \leq (1-\varepsilon)^M\).

Taking logs, we have \(m \ln (1-\varepsilon) \leq M \ln (1-\varepsilon) + \ln n\)

Using the facts that \(-\varepsilon \leq \ln (1-x) < -x\) for \(x \in [0,1]\), it implies that \(m (-\varepsilon + \varepsilon^2) \leq M (-\varepsilon + \varepsilon^2) + \ln n\)

The factor \(2\) in front of \(m\) is unavoidable. Suppose there are two experts A and B. Expert A
Randomized Expert Algorithm [3,3]

We generalize the setting where the losses suffered by the experts to be real numbers in \([0,1]\) rather than binary values. The loss suffered by the \(i\)-th expert at time \(t\) is denoted by \(\ell_i(t) \in [0,1]\). We also allow the outcome to be an arbitrary set (instead of up/down as in the previous example) and the set of actions recommended by the experts to be an arbitrary set.

We consider the following randomized algorithm.

Initially \(w_i = 1\) for every expert \(i\).

For each step \(t\):
- Follow expert \(i\) with probability \(\frac{w_i}{W}\) where \(W = \sum_j w_j\) is the total weight.
- Update \(w_i = w_i (1 - \varepsilon \ell_i(t))\) for all experts.

Claim. Let \(L\) be the expected loss of the randomized algorithm and \(L^*\) be the loss of the best expert, then \(L \leq (1 + \varepsilon)L^* + \frac{\ln n}{\varepsilon}\).

Note that the factor of 2 is removed. The analysis is almost the same.

Proof. Let \(W_t\) be the total weight at time \(t\).

The expected loss by the algorithm at time \(t\) is \(L_t = \frac{\sum_i w_i \ell_i(t)}{W_t}\).

Now, \(W_{t+1} = \frac{W_t}{\sum_i w_i (1 - \varepsilon \ell_i(t))} = \frac{W_t}{\sum_i w_i (1 - \varepsilon \ell_i(t))} = W_t - \varepsilon \sum_i w_i \ell_i(t) = W_t (1 - \varepsilon \ell_t(t))\).

Initially, \(w_0 = n\). By induction, \(W_t \leq n \prod_{t=1}^{T} (1 - \varepsilon \ell_t(t))\).

Since the total loss of the best expert is \(L^*\), we have \(W_T \geq (1 - \varepsilon)^L\).

Therefore, \((1 - \varepsilon)^L \leq n \prod_{t=1}^{T} (1 - \varepsilon \ell_t(t)) \Rightarrow L^* \ln (1 - \varepsilon) \leq \ln n + \sum_{t=1}^{T} \ln (1 - \varepsilon \ell_t(t))\).

Again, using \(-\varepsilon \ell_t(t) \leq \ln (1 - \varepsilon) < -\varepsilon\), this implies that \(L^* (-\varepsilon) \leq \ln n + \sum_{t=1}^{T} (-\varepsilon \ell_t(t))\) \(\Rightarrow L \leq L^* (1 + \varepsilon) + \frac{\ln n}{\varepsilon}\).
Remarks

- If $\mathbf{g}^{(t)} \in [0, p]$ , then we can reduce to the case when $\mathbf{g}^{(t)} \in [0, 1]$ by using the update rule
  
  \[ w_t = w_t (1 - 1/p) \]
  
  Then the analysis will give $L \leq (1 + \epsilon) L + pln (1/e).

- We can also consider gain $\mathbf{g}^{(t)} \in [0, 1]$ instead of loss. Using the update rule $w_t \leftarrow w_t (1 - \epsilon)$, the
  same analysis gives $G_t \geq (1 - \epsilon) G_{t-1} - \frac{2\epsilon}{p}.

- We can also consider $\mathbf{g}^{(t)} \in [-1, 1]$ (loss and gain) with similar guarantees; see [2].

- Note that we did not assume anything about the losses, except that $\mathbf{g}^{(t)} \in [0, 1]$. One way to think of
  it as getting close to the best expert. An alternative way to use it is to have an adversary to
  choose the worst case for the algorithm in each step, say the loss is always close to one, and in
  this case we use the above result to claim that even the best expert has average loss close to one.
  That is, we control the outcome and make the algorithm perform poorly, and then we claim that even
  the best expert must also perform poorly. This viewpoint is very useful in applying the multiplicative
  update method.

Two Person Zero Sum Game [4]

A two person zero sum game is specified by an $m \times n$ payoff matrix $A$, where $A_{ij}$ is the amount paid by the
row player to the column player if strategies $(i, j)$ are played.

The expected payoff for a pair of mixed strategies $(x,y)$ is $x^T A y$.

The objective of the row player is to minimize payoff while the objective of the column player is to maximize payoff.

If the column player plays first (decide his/her strategies first), then the value of the game is $C = \max_{y} \min_{x} x^T A y$.

If the row player plays first, then the value of the game is $R = \min_{y} \max_{x} x^T A y$.

In particular, there is a column strategy $y^*$ that achieves payoff at least $C$ against any row strategy. Similarly,
there is a row strategy $x^*$ that pays at most $R$ against any column strategy.

Obviously, if the column player plays second, then his payoff is at least as much as he plays first (gaining
at least $C$), and so $C \leq R$.

The von-Neumann's theorem says that $C = R$; it does not matter who starts first.

We give a proof using the multiplicative weights update method.

We assume without loss of generality that $A_{ij} \in [0, 1]$ for all $i, j$.

We think of the row strategies as the experts, and the payoff is their loss.
We think of the row strategies as the experts, and the payoff is their loss.

We will prove that $R < C + 2\varepsilon$ for any $\varepsilon > 0$ by the following experts algorithm.

**Algorithm**

Initially the weights of the row strategies are one: $w_i = 1$ for each row $i$.

Repeat for $T = \frac{\ln n}{\varepsilon}$ rounds:

- The mixed strategy is $x(r) = w_1(r)/w(r)$.
- The column player plays the best response $y(r)$ to the row player's strategy $x(r)$, by setting $y_i(r) = \min_j x_i^T A_j$.
- The row player updates the weights by setting $w_i(r+1) = w_i(r) e^{-y_i(r)}$ where $A_i^T$ is the loss for the $i$-th strategy in round $r$.

Output: The average strategies $x = \frac{1}{T} \sum_r x(r)$ and $y = \frac{1}{T} \sum_r y(r)$ satisfy

1. $C \geq \min_i (A y_i) \geq x^T A y - \varepsilon$
2. $R \leq \max_j (x^T A)_j \leq x^T A y + \varepsilon$

Together they imply $R \leq C + 2\varepsilon$.

**Analysis**

In each round, the column player can guarantee a payoff of at least $R$ (as the row strategy may not be optimal).

So, the total loss $L$ of the row player is at least $R \cdot T$.

By the multiplicative weights update method, we have $L \leq (1 + \varepsilon) L^* + \ln n/\varepsilon$.

In this setting, $L^* = \min_i \sum_r (A y_i(r)) = \min_i (A y_i) \cdot T$.

So, $R \cdot T \geq (1 + \varepsilon) \min_i (A y_i) \cdot T + \ln n/\varepsilon$.

Note that $\min_i (A y_i)$ is at most $C$, as the best response to the column strategy $y$ is at least as good as $x^T A y = C$ (because $x^*$ is just a linear combination of pure row strategies).

Therefore, $R \cdot T \leq (1 + \varepsilon) C \cdot T + \ln n/\varepsilon$.

It implies that $R \leq (1 + \varepsilon) C + \frac{\ln n}{\varepsilon \cdot T} \leq C + \varepsilon + \frac{\ln n}{\varepsilon \cdot T}$ (as $C \leq 1$ as $A_{ij} \in [0, 1]$).

Thus, $R \leq C + 2\varepsilon$ (as $T = \frac{\ln n}{\varepsilon}$).

**Remarks**

- In this problem, we use the approach that an adversary chooses the outcome (and thus the losses).
so that the algorithm performs poorly, and thus the best expert also performs poorly, establishing that the average column strategy is a good mixed strategy. We will use it again next week.

- It is known that this minimax theorem is equivalent to the LP duality theorem. So this result is very general.

- The multiplicative weights update method has many other applications, including machine learning (boosting), online algorithms, solving linear programs efficiently (next week), complexity theory (XOR lemma), network congestion control (TCP/IP), etc. See [2] for references.

References


