18.409 An Algorithmist's Toolkit December 3, 2009

Lecture 23

Lecturer: Jonathan Kelner

1 Outline

Last lecture discussed the conjugate gradient algorithm for solving linear systems $Ax = b$. This lecture will discuss *preconditioning*, a method for speeding up the conjugate gradient algorithm for specific matrices.

2 Last Lecture

Last lecture we described the conjugate gradient algorithm for solving linear systems $Ax = b$ for positive definite matrices A. The time bound for conjugate gradient depends on fitting low-degree polynomials to be small on the eigenvalues of A and large at 0. Using Chebyshev polynomials, this gives a running time of $\tilde{O}(\kappa^{1/2})$, where $\kappa = \frac{\lambda_{\max}}{\lambda_{\min}}$ is the *condition number* of A.

For certain matrices A, tighter bounds can be achieved. For example, if the eigenvalues of A tend to be clustered together, then polynomials can more easily be small at all the eigenvalues. But in the worst case and the general case, conjugate gradient takes $\tilde{\Theta}(\kappa^{1/2})$ time. For *ill-conditioned* (or *badly-conditioned*, *poorly-conditioned*, etc.) matrices, this can be quite slow. What do we do then?

3 Preconditioning

3.1 Motivating example

Some matrices A with terrible condition number can still be solved fairly easily. For example, consider the matrix

$$A = \begin{pmatrix} 10000 & 777 & 123 \\ 0.1 & 1 & 0.2 \\ 0.002 & 0.001 & 0.01 \end{pmatrix}.$$

This has condition number $\kappa \approx 1000000$, and condition number 10^{12} once you compute A^TA to get a positive definite matrix to perform conjugate gradient on. But if you normalize the diagonal to get

$$D^{-1}A = \begin{pmatrix} 1 & 0.0777 & 0.0123 \\ 0.1 & 1 & 0.2 \\ 0.2 & 0.1 & 1 \end{pmatrix},$$

you find a well-conditioned matrix. So you can use conjugate gradient to quickly solve $D^{-1}Ax = D^{-1}b$ instead. When we do this, we call D a "preconditioner" for A.

There's no reason that preconditioners have to be diagonal; they just have to be easily invertible. The next section discusses the general problem of finding preconditioners.

3.2 In general

The problem is that we want to solve $Ax = b$ but A is ill-conditioned, so conjugate gradient doesn't work directly. However, we also know some other positive definite matrix M that approximates A and is easy to invert. Then we instead use conjugate gradient on $M^{-1}Ax = M^{-1}b$. If M approximates A, then $M^{-1}A$ should have low condition number and conjugate gradient will be fast. This idea has few complications:

- **How do we find M?** There's no general answer to this question, since it's impossible for most A. However, most problems you want to solve have structure, which often allows you to find a good M. The second part of this lecture discusses how to find a good M when A is a Laplacian.
- **It could hard to compute $M^{-1}A$.** If M and A are sparse, you don't want to compute the dense matrix $M^{-1}A$. Fortunately, you don't need to. Conjugate gradient only computes vector products, which you can compute in succession.
- **$M^{-1}A$ may not by symmetric or positive definite.** You need it to be positive definite for conjugate gradient to be proven correct. Fortunately, this can be worked around, as shown below:

3.3 Dealing with $M^{-1}A$ being asymmetric

While $M^{-1}A$ may not be symmetric, both M and A are. So we can factor $M = EE^T$. Then $E^{-1}AE^{-T}$ has the same eigenvalues as $M^{-1}A$, since if $M^{-1}Av = \lambda v$, then

$$E^{-1}AE^{-T}(E^Tv) = E^TM^{-1}Av = \lambda E^Tv.$$

So rather than solving $M^{-1}Ax = M^{-1}b$, we can solve $E^{-1}AE^{-T}\hat{x} = E^{-1}b$ and return $x = E^{-T}\hat{x}$. This can be done with conjugate gradient, since it uses a positive definite matrix.

Now, we might not know how to factor $M = EE^T$. Fortunately, if we look at how conjugate gradient works, it never actually requires this factorization. Every time E is used, it will come in the pair $(aE^{-T})(E^{-1}b) = aM^{-1}b$.

This completes our sketch of how preconditioning algorithms work, once you find a preconditioner. We will spend the rest of lecture on finding preconditioners for Laplacians.

4 Preconditioners on Laplacians

Recall from previous lectures that any graph G can be *sparsified.* This means that we can find a graph H with $\tilde{O}(n)$ edges such that

$$(1-\epsilon)L_h \preceq L_G \preceq (1+\epsilon)L_h.$$

Then L_h is a good preconditioner for L_G. This is because all the eigenvalues of $L_H^{-1}L_G$ lie in $[1-\epsilon, 1+\epsilon]$, so $L_H^{-1}L_G$ has constant condition number.

We can use this to solve Laplacian linear systems for all graphs as if they are sparse and only multiply the number of iterations by log factors. Each step of conjugate gradient requires solving a sparse linear system on H, and it only takes logarithmically many iterations to converge.

But to do this, we need to find H. Our previous method required a linear system solver to get H, so we can't use it. There is a way to get a slightly weaker spectral sparsifier in nearly linear time, though. We give a sketch of the algorithm, but don't go into much detail:

We know that random sampling does a good job of sparsifying expanders. The problem with random sampling is when cuts have very few edges crossing them. So we first break the graph into well-connected clusters with our fast local partitioning algorithm. Inside each cluster, we randomly sample. We then condense the clusters and recurse on the edges between clusters.

So in nearly linear time, we can get a graph with $\tilde{O}(n)$ edges and a $1+\epsilon$ spectral approximation to G. But for use as a preconditioner, we don't need a $1+\epsilon$ approximation. We can relax the approximation ratio in order to dramatically cut the number of edges in the sparsifier. This will cause us to take more iterations of conjugate gradient, but be able to perform each iteration quickly. We dub these incredibly sparse matrices *ultra-sparsifiers.*

4.1 Ultra-Sparsification

What we cover now is a method to speed up conjugate gradient by using an even sparser H. All the methods discussed henceforth are easily applicable to solving $Mx = b$ for any M that is *weakly diagonally dominant* (not just graph Laplacians), i.e. for all i it holds that $|M_{i,i}| \geq \sum_{j \neq i} |M_{i,j}|$. The H we will precondition with now we call *ultra-sparsifiers* as they will only have $(1 + o(1))n$ edges! You can think of H as essentially being a spanning tree of G with only a few extra edges.

Theorem 1 *Given a graph G with n vertices and m edges, it is possible to obtain a graph H with $n + t \log^{O(1)} n$ edges such that $L_H \preceq L_G \preceq (n/t)L_H$, independent of m.*

We will not prove Theorem 1 here. In the problem set you will show a weaker version where the (n/t) is replaced with (n/t^2). Getting to (n/t) requires similar ideas but gets slightly more complicated.

The main benefit to ultra-sparsification is that for many algorithms, the ultra-sparse graph acts like a graph with many fewer vertices. The ultra-sparse graph is a tree with relatively few additional edges linking nodes of the tree. For intuition on this, note that paths without branching can usually be condensed into a single edge. Furthermore, linear systems on trees can be solved in linear time.

The result will be that we can solve diagonally dominant linear systems in nearly linear time. This lecture will focus on Laplacians, but the problem set has a question on how to extend it to general diagonally dominant systems.

4.1.1 Embedding of graphs

Recall from problem set 1 that:

Lemma 2 *Let $P_{u,v}$ be a path from u to v of length k, and let $E_{u,v}$ be the graph that just has one edge from u to v. Then*

$$E_{u,v} \preceq kP_{u,v}.$$

Now, suppose that we have two graphs G and H and an embedding of G onto H such that each edge in G maps to a path in H. For $(i,j) \in G$, define stretch(i,j) to be the length of (i,j)'s embedded path in H. Then

$$G = \sum_{(i,j)\in E(G)} E_{i,j} \preceq \sum_{(i,j)\in E(G)} \text{stretch}(i,j)\text{image}(i,j) \preceq \sum_{(i,j)\in E(G)} \text{stretch}(i,j)H.$$

If H is a subgraph of G, this means

$$H \preceq G \preceq \sum_{(i,j)\in E(G)} \text{stretch}(i,j)H.$$

4.1.2 Spanning tree preconditioners

For trees T, we can solve $L_T x = b$ in linear time. So it would be really nice if H were a "low average-stretch spanning tree." We could then precondition on H and take $O(m)$ time per iteration.

Turns out that that low average-stretch spanning trees exist and can be found efficiently:

Theorem 3 *Any graph G has a spanning tree T into which it can be embedded such that*

$$\sum_{(i,j)\in E(G)} \textit{stretch}(i,j) \leq m \log^c n.$$

This already is strong enough to give a non-trivial result. If we use such a spanning tree as a preconditioner, we take $O(m)$ per iteration and take $\tilde{O}(m^{1/2}$ iterations (because the condition number is $\tilde{O}(m)$), for $\tilde{O}(m^{3/2})$ time.

Although we won't go into detail, it turns out that this exact algorithm actually runs in $O(m^{4/3})$ time. The eigenvalues of the tree have a structure such that error is halved in $\tilde{O}(m^{1/3})$ iterations, not just $\tilde{O}(m^{1/2})$ iterations as Chebyshev polynomials show.

Instead, we'll add a few more edges to make "Vaidya's augmented spanning trees", which will improve the condition number substantially. In the problem set, you'll go into more detail into this, and into how to apply this recursively to get nearly linear recovery time.

4.1.3 Constructing ultra-sparsifiers

We will take a spanner T of G, and add a small number s more edges to get H. We partition T into t subtrees of balances path lengths. We then add one well-chosen "bridge" edge between every pair of subtrees. This can be done so that

$$\kappa(L_H^{-1/2} L_G L_H^{-1/2}) \leq O(n/t).$$

The ultra-sparsifier H will have $n - 1 + s$ edges, for $s \leq \binom{t}{2}$ in general or $s \leq O(t)$ for planar graphs. With more cleverness, Spielman and Teng showed that s can be improved to $\tilde{O}(t)$ for general graphs.

5 Conclusion

It's interesting that previously we used linear algebra to speed up graph theory, but now we're using graph theory to speed up linear algebra.