CHAPTER 7

Augmented Spanning-Tree Preconditioners

It's time to construct a preconditioner! This chapter combinatorial algorithms for constructing preconditioners that are based on augmenting spanning trees with extra edges.

1. Spanning Tree Preconditioners

We start with the simplest support preconditioner, a maximum spanning tree for weighted Laplacians. The construction of the preconditioner B aims to achieve three goals:

- The generalized eigenvalues (A, B) should be at least 1.
- The preconditioner should be as sparse as possible and as easy to factor as possible.
- The product of the maximum dilation and the maximum congestion should be low, to ensure that generalized eigenvalues of the pencil (A, B) are not too large. (We can also try to achieve low stretch.)

These objectives will not lead us to a very effective preconditioner. It usually pays to relax the second objective and make the preconditioner a little denser in order to achive a smaller $\kappa(A, B)$. But here we strive for simplicity, so we stick with these objectives.

We achieve the first objective using a simple technique. Given A, we compute its canonical incidence factor U. We construct V by dropping some of the columns of U. If we order the columns of U so that the columns that we keep in V appear first, then

$$V = U \begin{pmatrix} I \\ 0 \end{pmatrix} .$$

This is a subset preconditioner, so

$$\lambda(A, B) \geq \sigma^{-1}(B, A) \geq \left\| \begin{pmatrix} I \\ 0 \end{pmatrix} \right\|_2^{-2} = 1^{-2} = 1 .$$

We now study the second objective, sparsity in B. By Lemma **??**, we should G_B as connected as G_A. That is, we cannot drop so many

columns from U that connected components of G_A become disconnected in G_B. How sparse can we make G_B under this constraint? A spanning forest of G_A. That is, we can drop edges from G_A until no cycles remain in G_B. If there are cycles, we can clearly drop an edge. If there are no cycles, dropping an edge will disconnect a connected component of G_A. A spanning subgraph with no cycles is called a spanning forest. If G_A is connected, the spanning forest is a spanning tree. A spanning forest G_B of G_A is always very sparse; the number of edges is n minus the number of connected components in G_A. The weighted Laplacian B of a spanning forest G_B can be factored into triangular factors in $\Theta(n)$ time, the factor requires $\Theta(n)$ words of memory to store, and each preconditioning step requires $\Theta(n)$ arithmetic operations.

Which edges should we drop, and which edges should we keep in the spanning forest? If G_B is a spanning forest of G_A, than for every edge (i_1, i_2) there is exactly one path in G_B between i_1 and i_2. Therefore, if G_B is a spanning forest, then there is a unique path embedding π of the edges of G_A into paths in G_B. The dropping policy should try to minimize the maximum congestion and dilation of the embedding (or the stretch/crowding of the embedding). The expressions for congestion, dilation, stretch, and crowsing are sums of ratios whose denominators are absolute values or squares of absolute values of entries of B. Therefore, to reduce the congestion, dilation, and stretch, we can try to drop edges (i_1, i_2) that correspond to $A_{i_1,i_2}$with small absolute values, and keep "heavy" edges that correspond to entries of A with large absolute values.

One class of forests that are easy to construct and that favor heavy edges are *maximum spanning forests*. A maximum spanning forest maximizes the sum of the weights $-A_{i_1,i_2}$ of the edges of the forest. One property of maximum spanning forest is particularly useful for us.

LEMMA 1.1. *Let G_B be a maximum spanning forest of the weighted (but unsigned) graph G_A, and let $\pi(i_1, i_\ell) = (i_1, i_2, \ldots, i_\ell)$ be the path with endpoints i_1 and i_ℓ in G_B. Then for $j = 1, \ldots, \ell - 1$ we have $|A_{i_1,i_2}| \leq |A_{i_j,i_{j+1}}|$.*

PROOF. Suppose for contradition that for some j, $|A_{i_1,i_\ell}| > |A_{i_j,i_{j+1}}|$. If we add (i_1, i_ℓ) to G_B, we create a cycle. If we then drop (i_j, i_{j+1}), the resulting subgraph again becomes a spanning forest. The total weight of the new spannign forest is $|A_{i_1,i_\ell}| - |A_{i_j,i_{j+1}}| > 0$ more than that of G_B, contradicting the hypothesis that G_B is a maximum spanning forest. □

It follows that in all the summations that constitute the congestion, dilation, stretch and crowding, the terms are bounded by 1. This yields the following result.

LEMMA 1.2. *Let A and B be weighted Laplacians with identical row sums, such that G_B is a maximum spanning forest of G_A. Then*

$$\kappa(A, B) \leq (n-1)m \ ,$$

where n is the order of A and B and m is the number of nonzeros in the strictly upper triangular part of A.

PROOF. Let W correspond to the path embedding of G_A in G_B. For an edge (i_1, i_2) in G_A we have

$$\text{dilation}_\pi(i_1, i_2) = \sum_{\substack{(j_1,,j_2) \\ (j_1,,j_2) \in \pi(i_1,i_2)}} \sqrt{\frac{A_{i_1,i_2}}{B_{j_1,j_2}}} \leq \sum_{\substack{(j_1,,j_2) \\ (j_1,,j_2) \in \pi(i_1,i_2)}} 1 \leq n-1 \ .$$

The rightmost inequality holds because the number of edges in a simple path in a graph is at most $n-1$, the number of vertices minus one. The total number of paths that use a single edge in G_B is at most m, the number of edges in G_A, and hence the number of paths (in fact, its not hard to see that the number of paths is at most $m-(n-2)$). Therefore,

$$\begin{aligned}
\kappa(A, B) &\leq \sigma(A, B)/\sigma(B, A) \\
&\leq \sigma(A, B)/1 \\
&\leq \left(\max \left\{ 1, \max_{(i_1,i_2) \in G_A} \text{dilation}_\pi(i_1, i_2) \right\} \right) \\
&\quad \cdot \left(\max \left\{ 1, \max_{(j_1,j_2) \in G_A} \text{congestion}_\pi(j_1, j_2) \right\} \right) \\
&\leq (n-1)m \ .
\end{aligned}$$

A similar argument shows that the stretch of an edge is at most $n-1$, and since there are at most m edges, $\|W\|_F^2$ is at most $(n-1)m+n$. □

Algorithms for constructing minimum spanning trees and forests can easily be adapted to construct maximum spanning forests. For example, Kruskal's algorithm starts out with no edges in G_B. It sorts the edges of G_A by weight and processes from heavy to light. For each edge, the algorithm determines whether its endpoints are in the same connected component of the current forest. If the endpoints are in the same component, then adding the edge would close a cycle, so the edge is dropped from G_B. Otherwise, the edge is added to G_B. This algorithm requires a union-find data structure to determine

whether two vertices belong to the same connected components. It is easy to implement the algorithm so that it performs $O(m \log m)$ operations. The main data structure in another famous algorithm, Prim's, is a priority queue of vertices, and it can be implemented so that it performs $O(m + n \log n)$ operations. If A is very sparse, Prim's algorithm is faster.

When we put together the work required to construct a maximum-spanning-forest preconditioner, to factor it, the condition number of the preconditioned system, and the cost per iteration, we can bound the total cost of the linear solver.

THEOREM 1.3. *Let A be a weighted Laplacians of order n with $n + 2m$ nonzeros. Then a minimal-residual preconditioned Krylov-subspace method with a maximum-spanning-forest preconditioner can solve a consistent linear system $Ax = b$ using $O((n + m)\sqrt{nm})$ operations.*

PROOF. Constructing the preconditioner requires $O(m + n \log n)$ work. Computing the Cholesky factorization of the preconditioner requires $\Theta(n)$ work. The cost per iteration is $\Theta(n + m)$ operations, because A has $\Theta(n+m)$ nonzeros and the factor of the preconditioner only $\Theta(n)$. The condition number of the preconditioned system is bounded by nm, so the number of iteration to reduce the relative residual by a constant factor is $O(\sqrt{nm})$. Thus, the total solution cost is

$$O(m + n \log n) + \Theta(n) + O((n + m)\sqrt{nm}) = O((n + m)\sqrt{nm}) \ .$$

□

Can we do any better with spanning forest preconditioner? It seems that the congesion-dilation-product bound cannot give a condition-number bound better than $O(mn)$. But the stretch bound can. Constructions for *low-stretch trees* can construct a spanning forest G_B for G_A such that

$$\sum_{(i_1,i_2)\in G_A} \text{stretch}_\pi(i_1, i_2) = O\left((m + n)(\log n \log\log n)^2\right) \ ,$$

where π is the embedding of edges of G_A into paths in G_B. Like maximum spanning forests, lowe-stretch forests also favor heavy edges, but their optimization criteria are different. The number of operations required to construct such a forest is $O((m + n) \log^2 n)$. The constructoin details are considerably more complex than those of maximum spanning forests, so we do not give them here. The next theorem summarizes the total cost to solve a linear system with a low-stretch forest preconditioner.

THEOREM 1.4. *Let A be a weighted Laplacians of order n with $n + 2m$ nonzeros. A minimal-residual preconditioned Krylov-subspace method with a low-stretch-forest preconditioner can solve a consistent linear system $Ax = b$ using $O((n + m)^{1.5}(\log n \log\log n)$ operations.*

PROOF. Constructing the preconditioner requires $O((m+n)\log^2 n)$ work. Computing the Cholesky factorization of the preconditioner requires $\Theta(n)$ work. The cost per iteration is $\Theta(n + m)$. The condition number of the preconditioned system is $O((m + n)(\log n \log\log n)^2)$. Thus, the total solution cost is

$$O((m+n)\log^2 n)+\Theta(n)+O\left((n+m)\sqrt{(m+n)(\log n\log\log n)^2}\right) = O\left((n+m)^{1.5}(\log n\log\log n\right) .$$

□

2. Vaiyda's Augmented Spanning Trees

The analysis of spanning-tree preconditioners shows how they can be improved. The construction of the preconditioner is cheap, the factorization of the preconditioner is cheap, each iteration is cheap, but the solver performs many iterations. This suggests that it might be better to make the preconditioner a bit denser. This would make the construction and the factorization more expensive, and it would make every iteration more expensive. But if we add edges cleverly, the number of iterations can be dramatically reduced.

The following algorithm adds edges to a maximum spanning tree in an attempt to reduce the bounds on both the congestion and the dilation. The algorithm works in two phases. In the first phase, the algorithm removes edges to partition the tree into about t subtrees of roughly the same size. In the second phase, the algorithm adds the heaviest edge between every two subtrees. The value t is a parameter that controls the density of the preconditioner. When t is small, the preconditioner remains a tree or close to a tree. As t grows, the preconditioner becomes denser and more expensive to construct and to factor, the cost of every iteration grows, but the number of iterations shrinks.

Partitioning A Tree. The partitioning algorithm, called TREEPARTITION, is shown in Figure 1. The algorithm starts by recursively counting the number of vertices in the subtree rooted at every vertex. Once these counts are computed, the algorithm partitions the tree. It starts at the root. When the partitioning visits vertex i, the first task

Algorithm 1 Partitioning a rooted tree into subtrees with between n/t and $1 + d_{\max}n/t$ vertices. We

```
TreePartition(tree T)
  s ←new integer array
  RecursiveVertexCounts(root of T, s)
  RecursiveTreePartition(root of T,s)

RecursiveVertexCounts(vertex i, integer array s)
  s_i ← 1
  for each child j of i
    RecursiveVertexCounts(j, s)
    s_i ← s_i + s_j

RecursiveTreeParition(vertex i, integer array s)
  s_i ← 1
  for each child j of i
    if (s_j > n/t)
      RecursiveTreePartition(j,s)
    if s_j > n/t
      form a new subtree rooted at j
      disconnect j from i
    else
      s_i ← s_i + s_j
```

is to recursively partition the tree rooted at the children. If the subtree rooted in a child j of i is small, we do not partition it. If it is larger than n/t vertices, we trey to partition it recursively. When the recursive call returns, the tree rooted at j has been partitioned into subtrees no larger than $1 + d_{\max}n/t$ and no smaller than n/t. We test again the size of the subtree rooted at j; it may have shrunk because of the recursive partitioning. If it contains at most n/t vertices, we can keep it connected to i, because i can have at most $d_{\max}$ children. If it is larger, then the subtree rooted at j is large enough to be on its own, so we disconnect it from i. During the process, we recompute s_i, the number of vertices rooted at i *in the partitioned tree*.

Augmenting a Maximum Spanning Tree. Once the maximum spanning tree has been partitioned into subtrees with roughly equal sizes, we augmented the tree with extra edges. We use a simple rule: the heaviest edge that connects two different subtrees that are not connected by a spanning-tree edge is added to the preconditioner.

We label every edge (i, j) of the graph with a four-tuple

$$(\min(T(i), T(j)), \max(T(i), T(j), \text{in-the-mst}, |A_{i,j}|) \ ,$$

where $T(i)$ is an integer name of the subree that contains vertex i (say the index of its root) and where in-the-tree is a boolean value that indicates whther the edge (i, j) is in the maximum-spanning-tree. Once all the edges are labeled, we sort them lexicographically. The lexicographic sorting causes all the edges that connect two specific subtrees to be contiguous in the sorted order. If $T(i)$ and $T(j)$ are connected by a maximum-spanning-tree edge, it appears first among them. Also, the heaviest edge appears first. Therefore, if the first edge that links $T(i)$ to $T(j)$ is not in the tree, we add it to the tree and then skip all the other edges that connect $T(i)$ and $T(j)$. Once the sorting is done, we can find all the extra edges in time that is linear in the number m of edges. Sorting costs $O(m \log m)$, but we can use a linear-time radix sort to sort only on $(\min(T(i), T(j)), \max(T(i), T(j), \text{in-the-mst})$. That is, we can ignore the weight in the sort. Once we sort the one of these edges using these three-tuples, we can perform a linear search for the single edge that connects $T(i)$ and $T(j)$. This leads to a total cost of $O(m)$ for the augmentation phase.

Convergence-Rate Analysis of Augmented Maximum Spanning Trees. Our augmentation rule ensures that G_A can be embedded in G_B using short paths.

LEMMA 2.1. *Let G_B be a maximum spanning tree of a weighted graph G_A with maximum degree $d_{\max}$, augmented using the algorithm described above. If the size of subtrees after the partitioning phase is bounded by k, then there is a matrix W that satisfies the following conditions:*

1. *W has with at most $2k - 1$ nonzero entries per column,*
2. *it has at most $d_{\max}k$ nonzero entries per row,*
3. *the magnitude of the elements of W is bounded by 1, and*
4. *$U = VW$, where $A = UU^T$ and $B = VV^T$.*

PROOF. To prove the lemma, we construct an embedding of G_A into G_B. Edges in G_A whose endpoints are in the same subtree are routed using the single path between them in the maximum spanning tree, which are all in G_B. All the vertices along the path must be in the same subtree, so the length of the path is at most $k - 1$ edges. If the two endpoints i and j of an edge (i, j) in G_A are in different subtrees, we find the edge (x, y) in G_B that connects these two subtrees. We route the edge from i to say x (assuming without loss of generality

that i and x are in the same subtree) and then to y and from y to j. The length of this path is at most $2(k-1)+1 = 2k-1$ edges, because the routes from i to x and from y to j remain within a single subtree.

We now claim that the edges of G_B along the path that routes $A_{i,j}$ are all at least as heavy as $A_{i,j}$. This is clearly true if i and j are in the same subtree, since in that case we use only maximum-spanning tree edges. If i and j are in different subtrees but the edge (x, y) is in the maximum spanning tree, then again all the edges of the path are in the maximum spanning tree, so the claim is again true. If the edge (x, y) is not in the tree, it is clearly at least as heavy as $|A_{i,j}|$, otherwise (i, j) would have been selected to augment the tree, not (x, y). We claim that the edges from i to x and from y to j must be at least as heavy as (x, y). If one of them is lighter, we could have dropped it from the maximum spanning tree and included (x, y) instead. This concludes the proof of the claim and therefore, the proof of conditions 1, 3 and 4 of the lemma.

Condition 2 of the lemma follows from the fact that an edge (i, j) of G_B can participate in at most $d_{\max}k$ paths. Every path that it participates in must route an edge of G_A with one endpoint in the subtree that contains vertex i. Since there are at most k vertices in the subgraph and at most $d_{\max}$ edges incident on each such vertex, the total number of paths is bounded by $d_{\max}k$. □

This lemma implies the following one.

LEMMA 2.2. *Let G_B be a maximum spanning tree of a weighted graph G_A, augmented using the algorithm described above. The generalized condition number $\kappa(A, B)$ is bounded by*

$$\kappa(A, B) \leq 2\frac{d_{\max}^3 n^2}{t^2} + 4\frac{d_{\max}^2 n}{t} + 2d_{\max}$$

PROOF. By the construction of G_B, the size of each subgraph is bounded by $1 + d_{\max}n/t$. Therefore, we can substitute $1 + d_{\max}n/t$ for k in the previous lemma. Therefore, the number of nonzeros in a column of W is at most

$$2 + 2\frac{d_{\max}n}{t} - 1 \leq 2 + 2\frac{d_{\max}n}{t} .$$

Similarly, the number of nonzeros per row is at most

$$d_{\max}\left(1 + \frac{d_{\max}n}{t}\right) = d_{\max} + \frac{d_{\max}^2 n}{t} .$$

We can now bound the two-norm of W by

$$\begin{aligned}\|W\|_2^2 &\leq \|W\|_1 \|W\|_\infty \\ &= \left(\max_j \sum_i |W_{ij}|\right)\left(\max_i \sum_j |W_{ij}|\right) \\ &\leq \left(d_{\max} + \frac{d_{\max}^2 n}{t}\right)\left(2 + 2\frac{d_{\max} n}{t}\right) \\ &= 2\frac{d_{\max}^3 n^2}{t^2} + 4\frac{d_{\max}^2 n}{t} + 2d_{\max} \ .\end{aligned}$$

This bounds the largest eigenvalue of (A, B). The smallest eigenvalue is at least 1 because the preconditioner is a subset preconditioner. □

This bound shows that the larger t, the smaller the condition number of (A, B). This makes sense, since by making t larger we partition the spanning tree into more subtrees and therefore add more edges. Of course, as we make t larger, the more expensive it becomes to factor B.

We can bound the cost of factoring B using the number of inter-subtree edges in G_B. Let S be a set of vertices that are incident on inter-subtree edges. We start the factorization of B by eliminating all the degree 1 and 2 in G_B. This phase costs a constant number of operations and fill per elimination step. When this phase ends, all the remaining vertices have degree 3 or more. Moreover, the remaining vertices of a particular subtree still form a tree. The only leaves in this tree (vertices with degree 1) are vertices in S. The number of internal vertices in a tree with k leaves and no other vertices with degree 2 is bounded by the number of leaves, so the total number of vertices that are now left is at most $2|S|$.

Even if we ignore sparsity in the factorization of the remaining vertices, it will cost $O(|S|^3)$. The number of nonzeros in the factor is $O(n + |S|^2)$. In practice, the factorization of the remaining vertices is going to exploit some sparsity, but it is difficult to theoretically estimate this potential benefit.

How large can $|S|$ be? Since all the subtrees except perhaps one have at least n/t vertices, there can be at most $t + 1$ subtrees. Even if we augment the maximum spanning tree with one edge for every two subtrees, the size of $|S|$ is bounded by $2(t + 1)^2$. In some special cases, it is possible to show that the number of inter-subtree edges must be much smaller. For example, if G_B is a planar graph then $|S| = O(t)$.

When we put all of these together, we obtain the follwing result (it ignores the cost of computing the maximum spanning tree, which is almost always negligeable).

THEOREM 2.3. *The cost of solving a linear system using an augmented maximum spanning tree preconditioner is the cost of constructing the maximum spanning tree plus*

$$O\left(m + \left(n + t^6\right) + \left(n + t^4\right)\sqrt{\frac{d_{\max}^3 n^2}{t^2}}\right) .$$

PROOF. The first term inside the O notation bounds the cost of augmenting the spanning tree. The second bounds the cost of factoring the preconditioner, and the third the cost of the iterations. The cost of the iterations is bounded by multiplying the density of the factor of B, which dominates the cost of every iteration, by a bound on the number of iterations. □

If G_A is planar, the cost decreases to

$$O\left(m + \left(n + t^{1.5}\right) + (n + t\log t)\sqrt{\frac{d_{\max}^3 n^2}{t^2}}\right) .$$

These bounds suggest a way to select a theoretically optimal value for t. In the general case, we need to choose a value of t that minimizes

$$O\left(t^6 + \frac{d_{\max}^{1.5} n^2}{t} + d_{\max}^{1.5} n t^3\right)$$

(the other term in the bound are independent of t). Two of the terms grow with t and the third shrinks with t. The sum is minimizes roughly at the point where a growing term and a shrinking term are equal and the third is equal or to them or smaller. At $t \approx n^{1/4}$ the total cost is

$$O\left(n^{1.5} + d_{\max}^{1.5} n^{1.75} + d_{\max}^{1.5} n^{1.75}\right) = O\left(d_{\max}^{1.5} n^{1.75}\right) ,$$

which is asymptotically optimal. If G_A is planar, the minimum is achieved at $t \approx n^{0.8}$ and is $O(n^{1.2})$.

In practice, it is usually not a good idea to pick t according to these formulas. First, we have ignored multiplicative constants, so we know how to set t asymptotically, but this analysis does not give us a concrete formulat for setting t. Second, these formulas were based on a worst-case analysis of the fill in the factorization; if the factorization fills less than these worst-case estimates, making t larger reduces the total amout of work. To achieve high performance, it helps to experiment a bit to find a good value for t.

3. Notes and References

The low-stretch forests are from Elkin-Emek-Spielman-Teng, STOC 2005. This is an improvement over Alon-Karp-Peleg-West.