Excercise: Building Augmented Maximum Spanning Tree Preconditioners

Sivan Toledo and Haim Avron

March 22, 2007

The goal of this question is to implement Vaidya's augmented maximum-spanning tree preconditioner in MATLAB. We provide a skeleton of the implementation in the file `ex_vaidya.m`. In each part of this assignment you will develop another part of the algorithm.

The file `ex_vaidya.m` contains several internal functions (already implemented). The functions are fully documented inside the file, but we also provide here a brief description of them.

- `convert_to_graph`. This function takes the input matrix A (or any symmetric diagonally-dominant matrix and returns a weighted undirected graph G_A and a vector of the row sums of A. We represent a graph G with a MATLAB structure with exactly two fields: `G.n` and `G.edges`. The field `G.n` is the number of vertices, which we take to be the set $\{1, \ldots, G.n\}$. The field `G.edges` is an $|E|$-by-3 matrix that represents the graph's edges. Each row represents a different edge. Row $[i\, j\, w]$ represents the edge (i, j) with weight w.

- `convert_to_matrix`. The other direction, for converting a sparsified graph G_B to a preconditioner B.

- `makesets`, `unionsets`, and `findset` implement efficiently a data structure that represents a collection disjoint sets that can be unified and queried.

Your overall goal in this exercise is to implement the function `sparsify_graph`, which should construct the graph G_B of the preconditioner. We have provided an almost empty skeleton of this function that simply returns the input graph G_A.

1. The first phase of building Vaidya's preconditioner is to build a maximum spanning tree. Implement a function `kruskal` that computes the maximum spanning of the input graph. You can use the disjoint-sets data structure that we provide. You can also sketch the implementation of Prim's algorithm, another famous spanning tree algorithm, but you do not have to implement it.

2. Integrate `kruskal` into `sparsify_graph` to form a maximum-spanning tree preconditioner. You can use the following MATLAB code to generate the Laplacian of a two dimensional mesh, to make it strictly diagonally-dominant, and to visualize it. Test your code on such matrices: use them to precondition MINRES and try to plot the graph of the preconditioner to make sure that it is a tree.

```
>> U = mesh2d(10,10);
>> U = add_vertex_vectors(U,[1]);
>> A=U*U';
>> mesh2d_plot(10,10,U)
...
>> x = rand(size(A,1),1);
>> b = A*x;
>> xhat = minres(A,b,1e-8,1000,B);
```

3. When you call `minres` and pass a matrix B as a preconditioner, MATLAB factors B inside the `minres` function. Use `symmmd` to compute a fill-reducing ordering `p` for B and use `chol` to factor the matrix `B(p,p)`. Do you get any fill?

4. To augment the tree with extra edges, we will need to convert it into a rooted tree, with parent/child relationships between neighbors. Write a function `form_rooted_tree` that converts a tree `G` to a rooted tree. The output should be an array `parent` and vertex number `root`. Choose the root arbitrarily.

5. Write a function `build_child_arrays` that uses the `root` and `parent` to compute arrays `first_child` and `next_child`. If vertex `j` has children, then `first_child(j)` should be one of them, otherwise it should be `-1`. The child `i=first_child(j)` should be head of a linked list of the children of `j`. That is, `next_child(i)` should be the index of the next child of `j`, and so on. The list should end with the index `-1`. In the next phases of the algorithm it will be easier to work with this data structure than with the `parent` array alone.

6. Write a function `tree_partition` that receives a tree and a value $e \in [0, 1]$ and partitions the tree into a set of connected trees. Each subtree should have at least n/t vertices and at most $dn/t + 1$, where d is the maximum vertex degree in the tree and $t = n^e$. We use e rather than t as the argument to allow the same parameter value to work with matrices of different sizes. You are free to return the vertexes that are in the subtrees in any format you wish.

7. Implement a function `augment` that given a graph and a partition of its vertices finds the subset of edges that either connect vertices in the same subset, or are the heaviest among those that connect two specific subsets. These edges form the augmented-spanning-tree preconditioner.

8. Integrate the augmentation functions into `sparsify_graph` (instead of the simple maximum-spanning tree).

9. Test your preconditioner on two-dimensional meshes and plot its performance as a function of the density parameter e. Hint: to get reasonable performance you may need to preorder both A and B using a fill-reducing ordering computed for the preconditioner B. You can improve performance further by factoring the reordered B yourself and passing to `minres` or `pcg` the factor and its transpose instead of passing the reordered B.

10. (This part is optional.) You can also compare the performance of the preconditioner to that of a Joshi preconditioner and to that of `cholinc`. Can you estimate operation counts?