Proj 1: Harmonious Learning

There are many problems that involve optimizing some objective function by making local adjustments to a structure or graph. For example:

- If we want to reinforce a truss with a limited budget, where should we add new beams (or strengthen old ones)?
- After a failure in the power grid, how should lines be either taken out of service or put in service to ensure no other lines are overloaded?
- In a road network, how will road closures or rate-limiting of on-ramps affect congestion (for better or worse)?
- In a social network, which edges are most critical to spreading information or influence to a target audience?

For our project, we will consider a simple method for *graph interpolation*. We are given a (possibly weighted) undirected graph on n nodes, and we wish to determine some real-valued numerical property at each node. Given values at a few of the nodes, how should we fill in the remaining values? A natural approach that is used in some semi-supervised machine learning approaches is to fill in the remaining values by assuming that the value at an unlabeled node i is the (possibly weighted) average of the values at all neighbors of the node. In this project, we will see how to quickly solve this problem, and how to efficiently evaluate the sensitivity with respect to different types of changes in the setup. Of course, in the process we also want to exercise your knowledge of linear systems, norms, and the like!

Logistics

You are encouraged to work in pairs on this project. You should produce short report addressing the analysis tasks, and a few short codes that address the computational tasks. You may use any MATLAB or Octave functions you might want.

Most of the code in this project will be short, but that does not make it easy. You should be able to convince both me and your partner that your code is right. A good way to do this is to test thoroughly. Check residuals, compare cheaper or more expensive ways of computing the same thing, and

generally use the computer to make sure you don't commit silly errors in algebra or coding. You will also want to make sure that you satisfy the efficiency constraints stated in the tasks.

Background

The (combinatorial) *graph Laplacian* matrix occurs often when using linear algebra to analyze graphs. For an undirected graph on vertices $\{1, \ldots, n\}$, the weighted graph Laplacian $L \in \mathbb{R}^{n \times n}$ has entries

$$l_{ij} = \begin{cases} -w_{ij}, & \text{if } (i,j) \text{ an edge with weight } w_i \\ d_i = \sum_k w_{ik}, & \text{i = j} \\ 0, & \text{otherwise.} \end{cases}$$

The unweighted case corresponds to $w_{ij} = -1$ and d equal to the node degree.

In our problem, we seek to solve problems of the form

$$\begin{bmatrix} L_{11} & L_{12} \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \begin{bmatrix} 0 \\ r_2 \end{bmatrix}$$

where the leading indices correspond to nodes in the graph at which u must be inferred (i.e. u_1 is an unknown) and the remaining indices correspond to nodes in the graph at which u is specified (i.e. u_2 is known, though r_2 is not). Note that if i is an index in the first block, then the equation at row i specifies that

$$u_i = \frac{1}{d_i} \sum_{(i,j) \in \mathcal{E}} w_{ij} u_j,$$

i.e. the value at i is a weighted average of the neighboring values.

Your tasks

We will use the California road network data from the SNAP data set; to retrieve it, download the `roadNet-CA.txt` file from the class web page and use the loader script included to read in the topology and form the graph Laplacian. This is a big enough network that you will *not* want to form the graph Laplacian or related matrices in dense form. On the other hand,

because it is a moderate-sized planar graph, sparse Cholesky factorization on L will work fine.

The `README` file included with the code describes the baseline code, with places where you should fill in additional code marked by `TASK` comments. We have provided a testing script as part of the baseline code.

Task 1 Fill in `ginterp_eval0` with code to solve the graph interpolation problem. The code already computes index vectors `Ia` and `Ib` corresponding to the free variables (which must be computed) and the boundary values (which are known). On my laptop, this takes about four seconds. You should be careful with your parentheses, and you should *not* attempt an explicit inverse (which will probably crash your machine).

Task 2 As a set-up for the next steps, we are going to separate the solve in the first task into two components. The `ginterp_factor` routine will compute a sparse Cholesky factorization, while the `ginterp_eval` routine will use that factorization to solve the linear system. Make sure that you use the version of `chol` that returns a permutation for sparsity!

Task 3 Suppose now that we wish to incrementally add new specified values to the system. One way to do this would be to update which nodes are free and which have specified values, then recompute the factorization. We will try a different approach, which is to solve a *bordered system*

$$\begin{bmatrix} L_{11} & L_{12} & B_1 \\ L_{21} & L_{22} & B_2 \\ B_1^T & B_2^T & C \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ w \end{bmatrix} = \begin{bmatrix} 0 \\ r_2 \\ f \end{bmatrix}.$$

To enforce additional boundary conditions, we use each column of B_1 to indicate a node to constrain, and let the corresponding entry of f be the value at that node. The B_2 and C matrices will be zero in this case. You can complete this functionality by filling in the helper function `ginterp_bsys` and extending `ginterp_eval`. Your solution should not do any new sparse matrix factorizations, but will involve $O(k)$ new solves with the pre-computed factorization of L_{11}, where k is the number of new boundary conditions.

Task 4 Beyond using extra variables to enforce new boundary conditions, we can also use them to update the edge weights in the graph. For example,

consider an increase of s to the weight of edge (i, j) in the graph. The Laplacian for the new graph would be

$$L' = L + s(e_i - e_j)(e_i - e_j)^T,$$

and we can write $L'u$ as $Lu + (e_i - e_j)\gamma$ where $\gamma = s(e_i - e_j)^T u$. Using this observation, extend the `ginterp_bsys` command to form a bordered system that incorporates edge weight modifications as well as additional boundary conditions, all without re-computing any large sparse factorizations.

Task 5 Using bordered systems lets us recompute the solution quickly after we adjust the edge weights. But what if we want to compute the sensitivity of the value at some target node to small changes to *any* of the edges? That is, for a target node k, we think of u_k as a function of all the edge weights, and compute the sparse sensitivity matrix

$$S_{ij} = \begin{cases} \frac{\partial u_k}{\partial w_{ij}}, & (i,j) \in \mathcal{E} \\ 0, & \text{otherwise.} \end{cases}$$

Assuming the u vector has already been computed, the sensitivity computation requires constant work per edge after one additional linear solve. Fill in `ginterp_deriv` to carry out this computation. Note that your code should ideally use the bordered system formalism to incorporate any new boundary conditions or edge updates added to the system since the last factorization.

Afternotes

1. Almost all the tasks in this project boil down to block factorization, using sparse factorizations, and sensitivity analysis. Nothing should take many lines of code if you do it right. Nonetheless, the project is not trivial — so ask questions! Office hours and Piazza are your friends.

2. If you follow the intended path, none of the computations should be too expensive. However, the road network is large enough that you will cause yourself serious pain if you attempt to form it as a dense matrix. You may also run into trouble if you attempt a direct factorization without permuting for sparsity.