Graph Model and Sparse Representation

David Hardy

(Material from George & Liu, part of chapter 2 and all of chapter 3.)

1 Background

Problem: $Ax = b$, A is symmetric positive definite (SPD).

Why? This is the most important family of (sparse) matrices in practice. Also, many sparse direct techniques for SPD give insight into general matrices. A is symmetric means that $A = A^{\mathrm{T}}$. A is positive definite means that for all $x \neq 0$, $x^{\mathrm{T}}Ax > 0$.

Use Cholesky factorization for $A = LL^{\mathrm{T}}$. It is stable without pivoting for all SPD matrices. Computational complexity (for dense Cholesky) is about $1/6N^3$, half that for normal LU decomposition.

For sparse matrices, we reduce the cost of Cholesky by exploiting sparsity, pivoting to reduce *fill-in*. Using a permutation matrix P, $Ax = b$ is transformed to solving $(PAP^{\mathrm{T}})(Px) = Pb$. This works because $PAP^{\mathrm{T}} = LL^{\mathrm{T}}$ is also SPD. (Holds iff P is nonsingular.)

1.1 Outline of solution methods

There are various kinds of sparse storage schemes which differ in the way that zeros are exploited. In general, there are four distinct phases:

1. *Ordering* — Find a "good" ordering (permutation P) for A wrt chosen storage method.
2. *Storage allocation* — Determine the necessary information about Cholesky factor L of PAP^{T} to set up storage scheme.
3. *Factorization* — Factor PAP^{T} into LL^{T}.
4. *Triangular solution* — Obtain solution for factored system $LL^{\mathrm{T}}Px = Pb$.
   (a) Solve $Ly = Pb$ for y (forward substitution).
   (b) Solve $L^{\mathrm{T}}z = y$ for z (backward substitution).
   (c) Compute $x = P^{\mathrm{T}}z$.

Solution Method = (sparse storage system) + (ordering/allocation/factorization/solution)

Objectives for choosing a solution method:

(a) reduce computer storage

(b) reduce computer execution time

(c) reduce some combination of storage and execution time

1.2 Storage requirements and execution time

Storage is divided into two components:

- primary — holds numerical values
- overhead — holds matrix structure and facilitates access to primary storage

Note that we have to take both into account. Also, there might be a tradeoff between the two; for example, a method that reduces primary storage by fully exploiting zeros might increase the overhead storage by an amount that makes the total storage suboptimal.

Execution time is considered for each step of the solution method:

- ordering — Execution time might vary dramatically to find different orderings.
- allocation — Time varies in difficulty depending on our storage scheme and determined by the structure of L.
- factorization/solution — Differences in ordering schemes can lead to substantial differences in the arithmetic operations per second. Although execution is roughly proportional to the amount of arithmetic performed, differences in orderings and data structures can lead to large differences in the constant of proportionality.

Also, depending on what problems we are solving, we might be able to neglect the cost of one or more steps:

- Many systems with identical structure: neglect 1 & 2; only 3 & 4 matter.
- One system solved with many right-hand sides: neglect 1, 2, & 3; only 4 matters.

2 Graph Model

Use undirected graph to model the nonzero structure of an SPD matrix. Some results from graph theory and computer representation of graphs will help us in developing sparse matrix techniques.

2.1 Basic terminology and definitions

A *graph* $G = (X, E)$ consists of a finite nonempty set X of *nodes* or *vertices* and a possibly empty set E of *unordered pairs* (i.e. 2-element sets) of nodes. An *ordering* (*labeling*) α of G is a bijection of $\{1, 2, \ldots, N\} \rightarrow X$ where $|X| = N$, notated $G^\alpha = (X^\alpha, E^\alpha)$.

If A is $N \times N$ SPD matrix, the nonzero structure of A induces an ordered graph, the ordered graph of A, denoted by $G^A = (X^A, E^A)$. The vertices are labeled 1 to N, $X^A = \{x_1, \ldots, x_N\}$ corresponding to the diagonal elements, and $\{x_i, x_j\} \in E^A$ iff $a_{ij} = a_{ji} \neq 0$, $i \neq j$. x_i denotes the node of X^A with label i.

For any permutation $P \neq I$, the unlabeled graphs of A and PAP^{T} are the same, but the associated labelings are different. Thus, the unlabeled graph of A represents the equivalence class of matrices PAP^{T} for any permutation matrix P.

2.2 Graph theory definitions (that apply to unlabeled graphs)

Nodes x and y in G are *adjacent* if $\{x, y\} \in E$, notated $x \sim y$.

Let $Y \subseteq X$. The *adjacency set* of Y is

$$\mathrm{Adj}(Y) = \{x \in X - Y \,|\, x \sim y \text{ for some } y \in Y\}$$

$\mathrm{Adj}(Y)$ is the set of nodes not in Y but adjacent to some node that is in Y. If Y is a singleton set $\{y\}$, the notation is $\mathrm{Adj}(y)$. (Note: It is more common to refer to the adjacency set of a single node rather than a set of nodes.)

The *degree* of Y is $\mathrm{Deg}(Y) = |\mathrm{Adj}(Y)|$, the number of adjacent nodes to Y. Again for a singleton, the notation is $\mathrm{Deg}(y)$. (Note: This is not the standard graph theoretic definition. The usual definition is for the degree of a node to be the number of edges at that node. The fact that we are working with simple undirected graphs gives the happy coincidence that this definition agrees with the standard one for the degree of a node. Here, the degree of a set of nodes seems less intuitive.)

A *subgraph* $G' = (X', E')$ of G is a graph such that $X' \subseteq X$, X' nonempty, $E' \subseteq E$. For $Y \subseteq X$, the *section graph* $G(Y)$ is the subgraph $(Y, E(Y))$ where

$$E(Y) = \{\{x, y\} \in E \,|\, x, y \in Y\}$$

For a matrix, $G(Y)$ is the graph of the matrix obtained by deleting all rows and columns of A except those corresponding to Y.

A section graph is a *clique* if nodes are pairwise adjacent (i.e. it's a complete graph). This corresponds to a full submatrix.

For (distinct) nodes x and y, a *path* from x to y of length $\ell \geq 1$ is an ordered set of $\ell + 1$ (distinct) nodes $(v_1, \ldots, v_{\ell+1})$ such that $v_i \sim v_{i+1}$, $i = 1, \ldots, \ell$ with $v_1 = x$ and $v_{\ell+1} = y$.

A graph G is *connected* if every pair of distinct nodes is connected by at least one path. Otherwise G is *disconnected* and consists of two or more *connected components*. For a matrix, if G is disconnected and consists of k connected components and each component is labeled consecutively, the corresponding matrix will be block diagonal with each diagonal block corresponding to a connected component.

$Y \subseteq X$ is a *separator* of connected graph G if the section graph $G(X - Y)$ is disconnected.

2.3 Computer representation of sparse graphs

The performance of graph algorithms is quite sensitive to representation. The basic operation is that of retrieving adjacency relations between nodes. We need a representation which provides adjacency properties efficiently while being economical in storage.

Let $G = (X, E)$ be a graph. An *adjacency list* for $x \in X$ is a list containing nodes of $\mathrm{Adj}(x)$. An *adjacency structure* for G is the set of adjacency lists for all $x \in X$.

Three suggested methods of representation:

1. Store the adjacency lists sequentially in one array (`ADJNCY`) arranged in node order. Another array (`XADJ`) of length $|X| + 1$ is used to index into the adjacency structure array, one entry for each node, with the final entry pointing to the next available space in `ADJNCY` to mark the end of the adjacency structure. This method requires $|X| + 2|E| + 1$ storage.

2. Keep a *connection table* having $N = |X|$ rows and $m = \max\{\mathrm{Deg}(x) \mid x \in X\}$ columns. This is easier to create and maintain than the first approach but requires $m|X|$ storage. This might be very inefficient if few nodes have high degrees.

3. Store the adjacency structure as an array of linked lists, one linked list for each node's adjacency list. (George & Liu refer to this as keeping a *link field*.) This is good for dynamically storing the adjacency information (i.e. adding/removing edges), but the locality of reference is worse than the first method. (George & Liu use a cursor list representation in Fortran, with arrays `HEAD`, `NBRS`, and `LINK`. `HEAD` stores an index for each node into the `NBRS` list that gives a neighbor of that node. Each `NBRS` entry has a corresponding entry in `LINK` that indexes the next neighbor of the `HEAD` node in `NBRS`. The list is terminated using negative the `HEAD` node number. This is a cute idea, because it actually provides a circular linked list structure. Of course, the weakness with Fortran is that you have to predetermine the maximum amount of space used.) This representation requires at least $|X| + 4|E|$ storage.

Data structures for later routines described in George & Liu:

- `XADJ` — Index by node number into `ADJNCY`.
- `ADJNCY` — Sequential list of adjacencies arranged in node order. (Are the adjacencies themselves arranged in increasing order?)
- `PERM` — Used to permute the nodes. $\mathtt{PERM}(i) = k$ means that node i is renumbered k.
- `INVP` — The inverse permutation. $\mathtt{INVP}(\mathtt{PERM}(i)) = i$.
- `MASK` — (Boolean) array used to consider only nodes i for which $\mathtt{MASK}(i) \neq 0$.
- `ROOT` — Stores the number of a single node contained in the connected component given by `MASK` and such that $\mathtt{MASK}(\mathtt{ROOT}) \neq 0$.

Basically, XADJ and ADJNCY store the adjacency structure as specified by the first method above. The values in the matrix would need to be stored in arrays that parallel XADJ (for the diagonals) and ADJNCY (for the off diagonals). PERM and INVP correspond to P and P^{T} respectively. MASK allows routines to operate on section subgraphs. Using ROOT with MASK specifies a connected component of a section subgraph. (Note: This approach stores twice as much information as actually required. The structure information gives both L and L^{T}. An array of values that parallels ADJNCY would be storing each value twice.)