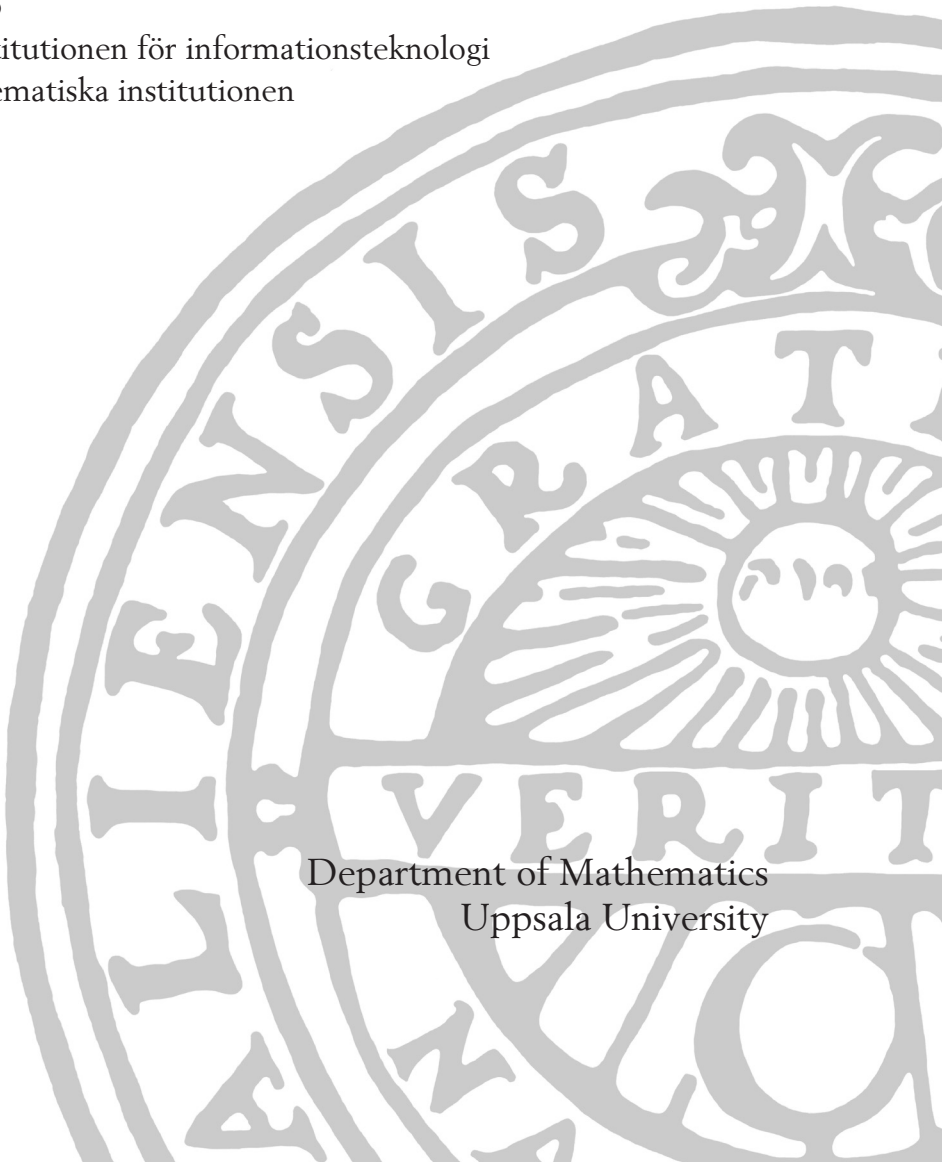
UPPSALA
UNIVERSITET

U.U.D.M. Project Report 2014:2

Programming aspects in a combinatorial sparse linear solver

Aron Rindstedt

Examensarbete i matematik, 15 hp
Handledare: Dimitar Lukarski, Institutionen för informationsteknologi
Examinator: Warwick Tucker, Matematiska institutionen
Januari 2014



Department of Mathematics
Uppsala University

Programming aspects in a combinatorial sparse linear solver

Aron Rindstedt

Uppsala University, Department of Mathematics
aron@absentia.se
Supervisor: Dimitar Lukarski, Department of Information Technology, Division of Scientific Computing
Examiner: Warwick Tucker, Department of Mathematics, Division of Mathematics

January 2014, Uppsala

Contents

I Introduction **3**
I Combinatorics/Graph theory . . . 3
II Subgraphs and connected components . . . 4
III Flows and potential . . . 4
IV Linear algebra . . . 5
V Electricity . . . 5
VI Optimal Flow . . . 6
VII Summary . . . 6

II Basic solver **7**
I Compact form . . . 7
II Detailed form . . . 7

III Star-decomposition **9**

IV Conclusions and future work **15**

Abstract

This bachelor thesis is to present a method for solving linear SDD matrix equations based on a combinatorial approach following the work of [1], well as provide a basic framework for efficiently implementing this method (or variants on them) in actual programs. Based on previously existing combinatorial solvers, this method has a significant speed advantage compared to more well known iterative matrix solvers when applied to large problems, for sparse matrices increasing in complexity at a nearly linear pace with matrix size. It also has a slight edge in speed compared to older combinatorial methods. This bachelor thesis will attempt to present this method in a compact yet accessible manner, simplifying its implementation in a programming environment.

I. Introduction

The purpose of this bachelor thesis is to demonstrate a fast, algorithm that solves the equation

$$A\vec{x} = \vec{b}$$

where A is a symmetric diagonally dominant matrix (for simplicity named the SDD equation). A symmetric diagonally dominant matrix (SDD matrix) is a matrix that is equal to its own transpose and whose diagonal elements are greater than or equal to the sum of the absolute values found in their respective columns

$$a_{ii} \geq \sum_{j \neq i} |a_{ij}|.$$

Additionally, it is sparse when most of its elements equal zero. Sparse SDD equations arise in a variety of important problems, including but not limited to numerical solutions of many differential equations and simulations of electrical systems. Thus, it is useful to have a fast and somewhat easy to understand algorithm to solve it.

Most well known "classical" methods for matrix equations are effective for small problems, but scale very poorly as the size of the problem increases and have trouble taking advantage of sparsity. There does however exist a wide range of combinatorial methods that are exceedingly good at solving large, sparse SDD equations. Most are incredibly complex and take significant effort to understand, but one- the one presented first in [1] and now here- is comparatively simple, though a full understanding of its underlying mechanisms is beyond the scope of this work.

Though far simpler than previous methods, this algorithm is still quite the complex machinery. Grossly, it operates using knowledge from three different fields, combinatorics/graph theory, electrical circuitry and linear algebra.

First, the SDD problem is changed to a combinatorial graph theory problem. Then, one perceives this problem in the form of an electrical circuit and uses the information from these two areas to establish the linear algebra system where much of the calculations take place.

I. Combinatorics/Graph theory

Combinatorics is the mathematic of finite structures. It deals with structures which are limited to a finite number of possible expressions (albeit an often very high number). A combinatorial algorithm is an algorithm that exploits the laws of combinatorics by establishing some finite structure and finding a solution somewhere within its set of expressions.

In this particular case, the finite structure we are dealing with is called a directed weighted graph. A directed weighted graph $G = G(V, E, w)$ is a common mathematical structure used to represent systems where objects move along fixed lines, including rivers, electric circuits, and roads. It is composed of the vertex set V, the edge set E and the set of edge weights w.

Vertices (also known as nodes) form the underpinnings of the graph, the points that connect edges to each other. In a road system, they represent the crossings. In terms of data structures, vertices typically don't have very many properties, offloading their data onto the edges.

Edges represent the roads, the rivers and the pieces of wire that things travel upon. Each edge $e = (v_1, v_2) \in E$ is defined by the two vertices (v_1 and v_2) it connects together. Edges are one-way, $(v_1, v_2) \neq (v_2, v_1)$. A graph is *symmetric* if for every edge (v_1, v_2) there exists an edge (v_2, v_1) going in the other direction (this is equivalent to what is commonly called an undirected graph). It is *antisymmetric* if there are no such edges (i.e. $\forall (v_1, v_2) \in E, (v_2, v_1) \notin E$). A graph is *weakly connected* if for any two vertices v_1 and v_2 there exists a path between them in at least one direction, i.e. one can reach one from the other by stringing together edges:

$$\forall v_1, v_2 \in V \exists n, \{u_i\}_{1 \leq i \leq n, i \in \mathbb{N}} \begin{cases} u_1 = v_1, u_n = v_2, \forall i \exists (u_i, u_{i+1}) \in E & \text{or} \\ u_1 = v_2, u_n = v_1, \forall i \exists (u_i, u_{i+1}) \in E \end{cases}$$

and *disconnected* if this condition is not fullfilled. In these ways, the structure of the edge set defines much of a graph's nature.

Finally, the set of edge weights w is composed of real positive scalars $w_e, e \in E$, typically representing in some way the amount of effort required to traverse an edge, such as the length of a road, the breadth of a river or the conductivity of an electrical wire. As a note to the confused, much of graph theory (but not this paper) deals with *unweighted* graphs, these are simply graphs with all weights set to one.

II. Subgraphs and connected components

Given a graph $G = G(V, E, w)$ and a vertex subset $V^* \subset V$, the *induced subgraph* $H = G(V^*, E^*, w^*)$ is the graph with the vertices V^*, the edges $E^* = \{(v_1, v_2) | v_1, v_2 \in V^*, (v_1, v_2) \in E^*\}$ and the edge weights $w^* = \{w_e \in w | e \in E^*\}$.

A graph's *connected components* is the set of induced subgraphs $H_i = G(V_i^*, E_i^*)$ such that for each i, H_i is weakly connected and for each $v \in V \setminus V_i^*$, the induced subgraph of $V_i^* \cup \{v\}$ is disconnected.

III. Flows and potential

Given an antisymmetric graph, a graph *flow* f is a set of scalars f_e associated to each edge $e \in E$. Flows can take many forms, and are usually used to represent the passage of something through the edges of the graph, such as cars, water or electricity. In this paper, we will primarily visualize our flows as electric current. In a slight heterodoxy, this flow can be negative for some edges, allowing the flow to move backwards along those edges (in this case, edge direction is less about making sure things go one way only and more a convenient way of keeping track of which direction they go).

A *source voltage* χ is a set of scalars χ_v associated with each vertex $v \in V$. The source voltage represents the amount of flow entering or leaving the graph at that vertex (presumably into a larger system). If for every vertex $v \in V$ by summing the flow of each edge (u, v) going to v and subtracting the flow of each edge (v, u) going from it one gets χ_v, then the flow is said to *fulfill the requirements for* χ.

$$\forall v \in V, \sum_{(u,v) \in E, u \in V} f_{(u,v)} - \sum_{(v,u) \in E, u \in V} f_{(v,u)} = \chi_v \tag{1}$$

A *potential set* ϕ (in the original paper confusingly referred to as v) is a set of scalars ϕ_e assigned to each vertex $v \in V$. Representing electric potential, potential sets are strongly related to flow. Each potential set generates a flow over the edges of its graph by the rule

$$f_{(v_1,v_2)} = (\phi_{v_1} - \phi_{v_2}) / w_{(v_1,v_2)} \tag{2}$$

Though this procedure is neither injective nor surjective, some flows (like the ones we seek) can be mapped onto potential sets by the rule

$$\phi_{v_1} = f_{(v_1,v_2)} \cdot w_{(v_1,v_2)} + \phi_{v_2} \tag{3}$$

This process does not actually define the potential set absolutely, merely a linear relation between its scalars. This will have to be dealt with but is by no means an insurmountable problem

since one can get any potential set corresponding to a given flow from any other such merely by adding a single constant from each of its scalars.

IV. Linear algebra

The link between graphs and SDD matrices is called a Graph Laplacian

$$L_{v_1,v_2} = \begin{cases} \sum_{(v,u)\in E} w_{(v,u)} & v = v_1 = v_2 \\ -w_(v_1, v_2) & (v_1, v_2) \in E \\ 0 & \text{otherwise} \end{cases} . \tag{4}$$

For symmetric graphs, Laplacian construction is a bijective function onto the set of symmetric matrices M with all negative non-diagonal entries and the property $L_{v,v} = \sum_{j\neq i} L_{i,j}$. This means that if we have such a matrix, we can construct a graph G with properties strongly linked to those of M by inverting the process. This connection is one we can exploit, so long as our matrix is of the proper form. Fortuitously, we can convert any SDD equation into one with a Laplacian matrix and (when we are done) extract from the solution to that equation the solution to the original equation. Thus, to solve any SDD equation, we need merely be able to solve "Graph Laplacian equations", allowing us to exploit many graph-theoretical principles in our algorithm.

Like potentials, flows can be seen as vectors, and every operation mentioned above can be seen as an algebraic operation. Summing flow over each vertex can be accomplished by multiplying leftwise with an easily constructed *transpose incidence matrix* and the energy is an easily computed norm. Additionally, we can see the set of flow vectors with zero sum in- or output into individual vertices as a vector space (since the zero flow naturally inputs zero and sums and multiples of flow vectors generate sum and multiple inputs).

V. Electricity

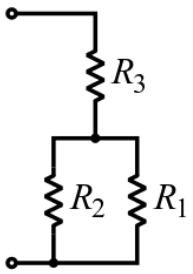


Figure 1: *An electric circuit. Where lines meet or end is a vertex, the lines themselves are edges. Creative commons <Wikipedia-Omegatron>*

To visualize a graph as an electrical circuit (fig 1), one sees each edge as a wire with a conductivity equal to its weight in Ω^{-1} (or in other words a resistance equal to $\frac{1}{w_e}\Omega$). In this model, our equation $L\vec{\phi} = \vec{\chi}$ becomes a representation of an electrical flow problem, where the elements of $\vec{\phi}$ represent the potential at the respective vertices and the elements of $\vec{\chi}$ represent the voltage going in or out of the system at the same.

Rather than calculating the potentials directly, however, we will concern ourselves with the flow of current they induce. This flow has two important properties. First, it must always fulfill the

requirements for $\vec{\chi}$. Second, we can with a bit of effort construct the sum energy of a given current flow, and since all systems naturally seek to minimize their energy, we need then simply find the least energetic flow which fulfills the requirements. As it turns out, this is a simple problem, as we can get any flow which fulfills the requirements by taking any other such vector and adding an appropriate flow from the set of flows that in- or outputs zero flow into each vertex, i.e.

$$\forall v \in V \sum_{e \in E} f_v(e) = 0. \tag{5}$$

VI. Optimal Flow

In order to find the optimal flow vector, we take a flow which fulfills the requirements and minimize its energy in the directions defined by the space of zero-input flows. To do this we need to find a basis for this space, and we need to define it such that we can reduce energy by using as few basis vectors as possible. This can be accomplished by creating a *spanning tree* (fig 2) over the circuit graph, specifically a *low-stretch spanning tree* and taking the flow around its *tree cycles* (the cycles formed by adding a single edge to the tree). Formed properly, this special tree ensures that minimizing need only be done with a pre-determined number of basis vectors.

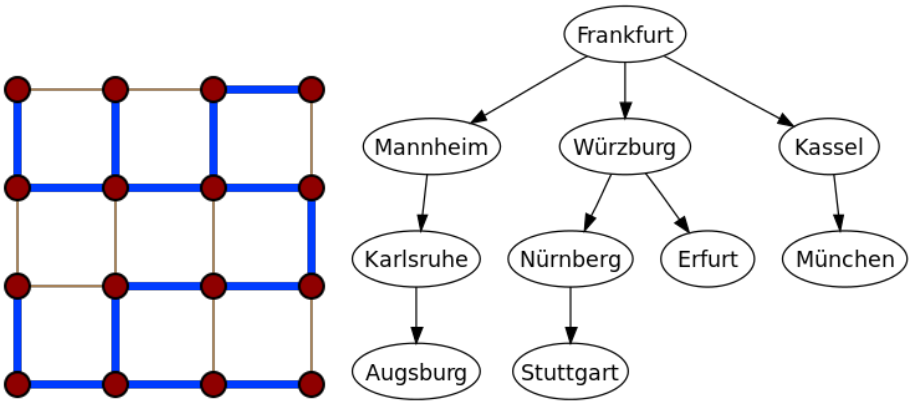


Figure 2: *Spanning trees*

VII. Summary

To solve an SDD problem, we first transform it into a more manageable problem with a Graph Laplacian-like matrix. Then, we extract from this matrix a corresponding graph and divide this graph into its connected components. For each such component, we construct incidence and resistance matrices and a low-stretch spanning tree and set up the vectors for its tree cycles. Finally, we subtract tree cycle vectors a preset number of times and revert the system (now including a solution) back into its original form.

II. Basic solver

I. Compact form

This section details the algorithm as used to solve an SDD equation $A\vec{x} = \vec{b}$ of size $n/2 \times n/2$ to within precision ϵ. It begins with a small overview, then goes into greater detail for each step along the way. One part of the algorithm, the Star-decomposition method for finding low-stretch spanning trees, is not found here, instead placed in the next section for purposes of text flow.

1 Convert A to Laplacian L.

2 Transform L into a graph.

3 Subdivide the graph into its weakly connected components, run subsequent steps on individual subgraphs.

4 Create incidence and resistance matrices.

5 Construct a low-stretch spanning tree and tree cycle vectors. Calculate the probability for each tree cycle and the total number of necessary loops K.

6 Iteratively calculate succesively better flow approximations K times.

7 Turn the resulting flow vector back into a potential vector.

8 Reassemble the full potential vector solving L and convert it back to a solution for A.

II. Detailed form

Input: A,$\vec{b}$
Output: $\vec{x}$

1. Let A be an SDD matrix of size $n/2 \times n/2$, $\vec{b}$ a vector of length $n/2$ and ϵ be a positive real number. This process is designed to find a vector $\vec{x}$ such that if $\vec{x}_{opt}$ solves

$$A\vec{x}_{opt} = \vec{b},$$

then

$$(\vec{x} - \vec{x}_{opt})^T A(\vec{x} - \vec{x}_{opt}) \leq \epsilon \cdot (\vec{x}_{opt}^T A\vec{x}_{opt}).$$

Divide A into A_+, containing all positive non-diagonal entries; A_-, containing all negative non-diagonal entries; D_1, the diagonal matrix defined by

$$D_1(i,i) = \sum_{j=1}^{n} A(i,j);$$

and D_2, the difference between A and the other three matrices,

$$D_2 = A - A_+ - A_- - D_1.$$

Now, define L and $\vec{\chi}$ by

$$L = \begin{pmatrix} D_1 + D_2/2 + A_- & -D_2/2 - A_+ \\ -D_2/2 - A_+ & D_1 + D_2/2 + A_- \end{pmatrix} \text{ and} \tag{6}$$

$$\vec{\chi} = \begin{pmatrix} b \\ -b \end{pmatrix}. \tag{7}$$

2. Create graph G with n vertices, ordering them between 1 and n. For each non-zero non-diagonal entry in the lower triangle of L, add an edge going between the vertex corresponding to that entry's column index to the vertex corresponding and its row index and give it a random direction. Order the edges and call the total number of edges m.

3. Select a vertex and use a depth-first search to find all vertices connected to it. Run steps 4 - 7 on the subgraph made from those vertices and all edges attached to them. Repeat this process (ignoring any vertices that have already been run) until no vertices remain.

4. Create the incidence matrix B as an $m \times n$ matrix with the entries

$$B_{i,j} = \begin{cases} -1 & \text{If the edge with index } i \text{ goes from the vertex with index } j \\ 1 & \text{If the edge with index } i \text{ goes to the vertex with index } j \\ 0 & \text{Otherwise} \end{cases}$$

and the resistance matrix R as an $m \times m$ matrix which has the diagonal entries

$$R_{i,i} = \frac{1}{w_{e_i}}$$

and is zero elsewhere.

5. Construct a low-stretch spanning tree T (III) over the graph. Create $\vec{f_0}$ as a size m vector which is 0 everywhere except for the edges of T and has the property

$$B^T \vec{f_0} = \vec{\chi}$$

on all relevant vertices. For each $e \in E \setminus T$ identify the tree cycle edge set c_e by taking $T \cup e$ and extricating the cycle (for example by using a variant depth-search, starting at e and probing through $T \cup e$ until you find the other end of e and erasing every path that didn't lead there). For each c_e, create the length m flow vector $\vec{c_e}$ that is either 1 or -1 on $e \in T \cup \{e\}$, 0 elsewhere and has the property $B^T \vec{c_e} = \vec{0}$ (i.e. the flow goes around the cycle in a circle and doesn't leak anywhere else). Create

$$\tau = \sum_{e \in E \setminus T} (\vec{c_e}^T R \vec{c_e}) \cdot w_e,$$

$$st(T) = \sum_{e \in E} w_e \cdot \sum_{e' \in c_e \setminus e} \frac{1}{w_{e'}}.$$

Create the probability vector $\vec{p}$ as a size m vector with the entries

$$P_e = \begin{cases} \frac{1}{\tau} \cdot w_e \cdot \vec{c_e}^T R \vec{c_e} & e \in E \setminus T \\ 0 & e \notin E \setminus T. \end{cases}.$$

Create

$$K = \left\lceil \tau \log(\frac{st(T) \cdot \tau}{\epsilon}) \right\rceil.$$

6. Do this step in a for-loop, with $1 \leq i \leq K$ as the loop variable.

   Use the probability vector to randomly select a tree cycle. Calculate

$$\vec{f_i} = \vec{f}_{i-1} - \frac{\vec{f}_{i-1}^T R \vec{c_e}}{\vec{c_e}^T R \vec{c_e}} \vec{c_e}$$

7. Transform the final flow vector $\vec{f_K}$ into a potential vector $\vec{\phi}$ containing the potentials for each vertex. Suggested method is to fix the potential ϕ_i in one vertex and use that to calculate all adjacent potentials by exploiting the rule

$$\forall 1 \leq i_1, i_2 \leq n, \phi_{i_1} = \phi_{i_2} + \vec{f_K}(e) \cdot r_e$$

   where e is the edge going from v_{i_1} to v_{i_2}. Then, create $\vec{\phi}$ by adding a constant to all potentials (that is, adding a multiple of the length n vector $\vec{I}$ defined by $\forall v, I_v = 1$) until you minimize the difference between $L\vec{\phi}$ and $\vec{\chi}$.

8. Assemble the final $\vec{\phi}$ from the subgraph $\vec{\phi}$ vectors. Finally, extract $\vec{x}$ from $\vec{\phi}$ by

$$\vec{x}(i) = \frac{\phi_i + \phi_{i+n/2}}{2} \tag{8}$$

III. Star-decomposition

The star-decomposition method [2] is an effective method for finding a low-stretch spanning tree. We discovered it when looking through the bibliography of [1] and chose it because it appeared to be the most effective algorithm (apart from the petal-decomposition algorithm [3] appears to be more a more complex machinery). Below is provided a streamlined instruction for convenience.

Here X is a symmetric graph; c is the constant 2^{16}; Q is an arbitrarily ordered queue data structure containing all the vertices of X; $d_X(u, v)$ is a metric formed by taking the length of the shortest path between u and v using only the edges of X; $rad_{x_0}(X)$ is the biggest $d_X(u, x_0)$ for any $u \in X$; $|X|$ is the number of vertices in X; (x, y) is the edge between x and y; and (X, Y, x, y) is just just a handy notation to keep track of two sets and two points.

Star-decomposition sets off the process.

Function *star-decomposition(X)* **is**
    Pick a vertex x_0 to be the center;
    Put all the vertices in X into the queue Q in a random order;
    n = |X|;
    T = `hierarchical-star-partition`(X,x_0,Q,n);
    Fix any weight changes or edge splits in T;
    return T;
end

The hierarchical star partition is the topmost layer of the actual algorithm. It is a recursive function, either decomposing the received graph using *star-decomposition* and running itself on the pieces or (if the graph is small enough) returning a *Breadth-First Search* of it.

Function *hierarchical-star-partition(X,x_0,Q,n)* **is**
    `/* The following two lines are a modification of the code to allow it to work properly with weighted graphs. */`
    d = $\frac{1}{64c}$;
    Set the weights of all edges with weight less than d to 0;
    if $rad_{x_0}(X) \leq 16c$ **then**
        return `Breadth-First Search` (X);
    end
    $(X_0, \cdots, X_m, (y_1, x_1), \cdots, (y_m, x_m), Q_0, Q_1, \cdots, Q_m)$ =`star-partition`(X, x_0, Q);
    foreach $i \in [0, \cdots, m]$ **do**
        T_i =`hierarchical-star-partition`(X_i, x_i, Q_i, n);
    end
    T=T_0;
    foreach $i \in [0, \cdots, m]$ **do**
        T=T$\cup(y_i, x_i) \cup T_i$;
    end
    return T;
end

The star-partition function divides the received graph into cones surrounding a ball centered on x_0. First it creates X_0 as the central ball, then it makes X_1 and the rest of the X's as cones connected to X_0 by a single edge. Then, it creates queues for each of these sets, taking special care to have Q_0, the queue for X_0, fulfill some special conditions (the special subqueues Q_0^a, Q_0^b, Q_0^c are part of this process).

Function *star-partition(X,x_0,Q)* **is**
- j=2;
- Denote the elements of Q by Q=$(z_1, z_2, \cdots, z_k)$;
- Choose ϵ randomly from $(0, \frac{1}{170c}]$;
- **Create** X_0 **by**
  - Choose r_0 randomly from $(1/(16c), 1/(8c))$;
  - X_0=`Ball(`$X, x_0, r_0 \cdot rad_{x_0}(X)$`)`;
  - Y_0=$X \setminus X_0$;
- **end**
- **Create** X_1 **by**
  - **if** $z_1 \in Y_0$ **then**
    - z=z_1;
  - **end**
  - **else**
    - z=a point in Y_0;
  - **end**
  - (y_1, x_1)=an edge s.t. $y_1 \in X_0, x_1 \in Y_0$ and $d_X(x_0, z) = d_X(x_0, y_1) + d_x(y_1, x_1) + d_{Y_0}(x_1, z)$;
  - r_1=a random number from $[\epsilon/4, \epsilon/2]$;
  - X_1=`Cone(`$(X, Y_0, x_0, x_1), Y_0, x_1, r_1 \cdot rad_{x_0}(X)$`)`;
  - $Y_1 = Y_0 \setminus X_1$;
- **end**
- **Create** $X_2, \cdots, X_m$ **by**
  - **while** $Y_{j-1} \neq 0$ **do**
    - (x_j, y_j, r_j)=`cone-cut(`$X, x_0, X_0, Y_{j-1}, \epsilon$`)`;
    - X_j=`Cone(`$(Y_{j-1} \cup X_0, Y_{j-1}, x_0, x_j), Y_{j-1}, x_j, r_j \cdot rad_{x_0}(X)$`)`;
    - $Y_j = Y_{j-1} \setminus X_j$;
    - j=j+1;
  - **end**
- **end**
- cont. on next page;

end

continuation of `star-partition(`X,x_0,Q`);`
Create $Q_0^a, Q_0^b, Q_0^c, Q_1, \cdots, Q_m$ **by**
  for $i = 1, \cdots, |X| - 1$ **do**
    if $z_i \in X_0$ **then**
      enqueue z_i into Q_0^a;
    end
    else
      l =the $l \geq 1$ s.t. $l \in X_l$;
      if $z_i \neq x_l$ **then**
        enqueue z_i into Q_l;
      end
      if $y_l \notin Q_0^c$ **then**
        enqueue y_l into Q_0^c;
      end
      if $|X_l \cap z_1, \cdots, z_i| > \sqrt{i}$ *and* $y_l \notin Q_0^b$ **then**
        enqueue y_l into Q_0^b;
      end
    end
  end
end
Create Q_0 **by**
  Denote $Q_0^a = (z_1^1, \cdots, z_{m_1}^1)$;
  Denote $Q_0^b = (z_1^2, \cdots, z_{m_2}^2)$;
  Denote $Q_0^c = (z_1^3, \cdots, z_{m_3}^3)$;
  Interleave Q_0^a, Q_0^b, Q_0^c **so that**
    if $z_1 \in X_0$ **then**
      z_1 is first in Q_0;
    end
    else
      y_1 is first in Q_0;
    end
    forall the $x \in X$ **do**
      if $x = z_i^l$ *for an* $l \in \{1, 2, 3\}, 1 \leq i \leq n$ **then**
        x is in the first $3i$ elements of Q_0;
      end
    end
  end
end
return $(X_0, \cdots, X_m, (y_1, x_1), \cdots, (y_m, x_m), Q_0, Q_1, \cdots, Q_m)$;

The cone-cut selects appropriate focus points and radii for cones. The while loop creates a pseudo-exponential distribution where the probability of $h = 1$ is 0.5, the probability of $h = 2$ is 0.25 and so on.

Function *cone-cut(X,x_0,X_0,Y,ϵ)* **is**
  p=the point in Y that minimizes $\frac{|X|}{|Ball(Y,p,\epsilon \cdot rad_{x_0}(X)/16)|}$;
  χ=$\frac{|X|}{|Ball(Y,p,\epsilon \cdot rad_{x_0}(X)/16)|}$;
  (y,x)=an edge s.t. $x \in Y, y \in X_0$ and $d_X(x_0, y) + d_X(y, x) + d_Y(x, p) = d_X(x_0, p)$;
  `/* The below line is a modification to allow the code to work with weighted graphs. */`
  Divide (y,x) into two edges (y,y'),(y',x) and a new vertex $y' \in X_0$ such that y' has the property $d_{X_0}(x_0, y') = rad_{x_0}(X_0)$.
  Divide $[\epsilon/4, \epsilon/2]$ into $N = \lceil 2\log(\chi) \rceil$ equal intervals $S_1, \cdots, S_N$;
  h=1;
  while *h*<*N* **do**
    rand = random number from $(0, 1)$;
    if *rand*>0.5 **then**
      break the loop;
    end
    else
      h=h+1;
    end
  end
  r= random number from S_h;
  return *(x,y',r);*
end

The ball function returns a ball with size r, centered on x_0.

Function *Ball(X,x_0,r)* **is**
  return *all points* $x \in X$ *with* $d_X(x, x_0) <$*r*;
end

The cone function returns a cone with size r, centered on x_0, using a cone pseudometric based on the sets X and Y and the points x and y.

Function *Cone((X,Y,x,y),X_0,x_0,r)* **is**
  r_2=d_X(x,x_0)-d_Y(y,x_0);
  return *all points* $u \in X_0$ *with* $|\, d_X$(x,u)-d_Y(y,u)-r_2)$\,| <$r;
end

IV. Conclusions and future work

In this thesis we have presented a combinatoric method for solving sparse linear SDD matrix equations, taking special care to focus on ease of implementation and avoiding going too deep into proof and theory. Now remains the task to convert this algorithm into computer language and implementing it in a program.

References

[1] Jonathan A. Kelner, Lorenzo Orecchia, Aaron Sidford, Zeyuan Allen Zhu. A Simple, Combinatorial Algorithm for Solving SDD Systems in Nearly-Linear Time. arXiv:1301.6628v1 [cs.DS] 2013

[2] Ittai Abraham, Yair Bartal, Ofer Neiman. Nearly Tight Low Stretch Spanning Trees. arXiv:0808.2017 [cs.DS] 2008

[3] Ittai Abraham, Ofer Neiman. Using petal-decompositions to build a Low Stretch Spanning Tree. STOC 2012: 395-406