The next steps in implementation of linked list representations of sparse matrices are the algorithms used to solve them. One of these algorithms well suited to computing solutions to systems of linear equations with matrix operations lies in the conjugate gradient method.

5 Real World Applications for Data Structures Storing Sparse Matrices and Matrix Operations

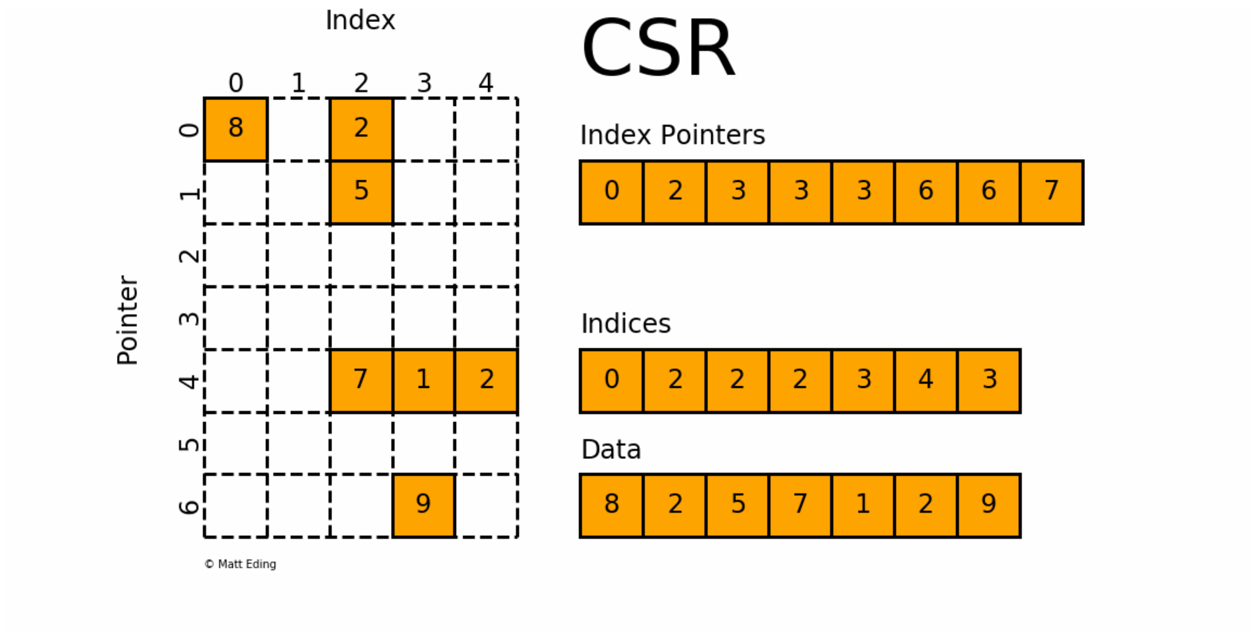
Sparse matrix algorithms lie in the intersection of graph theory and numerical linear algebra and are the natural continuation of the project outlined in this paper. The next section of this report briefly pursues the sparse matrix representation of systems of linear equations and what algorithms can be used to solve them in the most efficient ways possible. This application is essential to relating the current project directly to real-world uses and bridge some gaps of how we go from matrix operations to solving some of the complex real world problems.

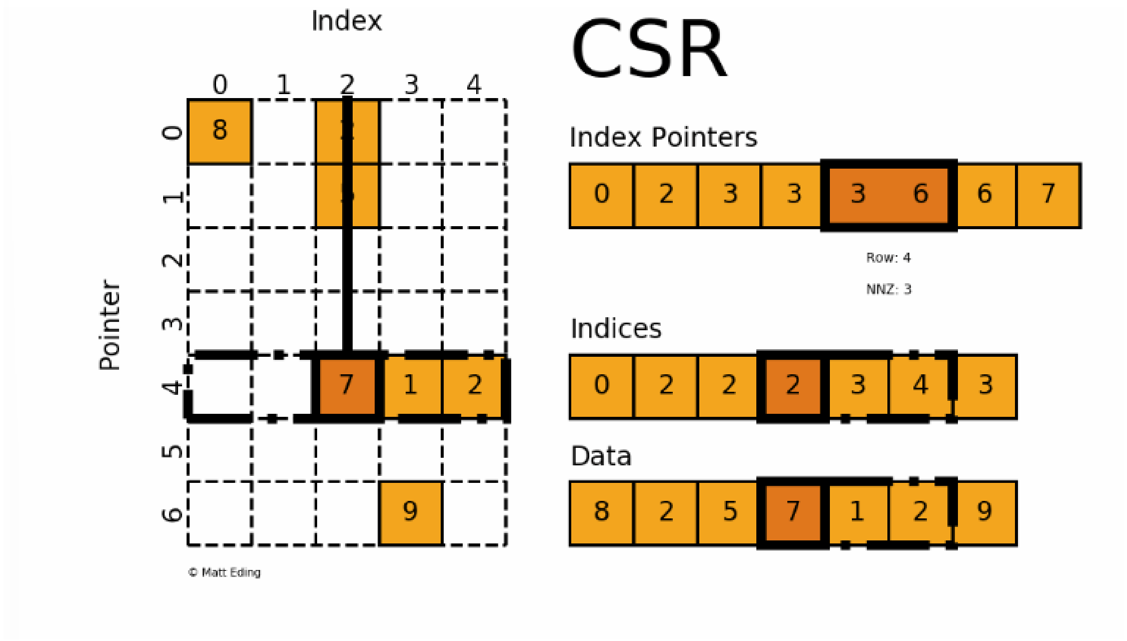
5.1 Sparse Matrices Solving System of Linear Equations Algorithms

5.2.1 Other Data Structure Implementations for Sparse Matrices Geared Towards Matrix Operations

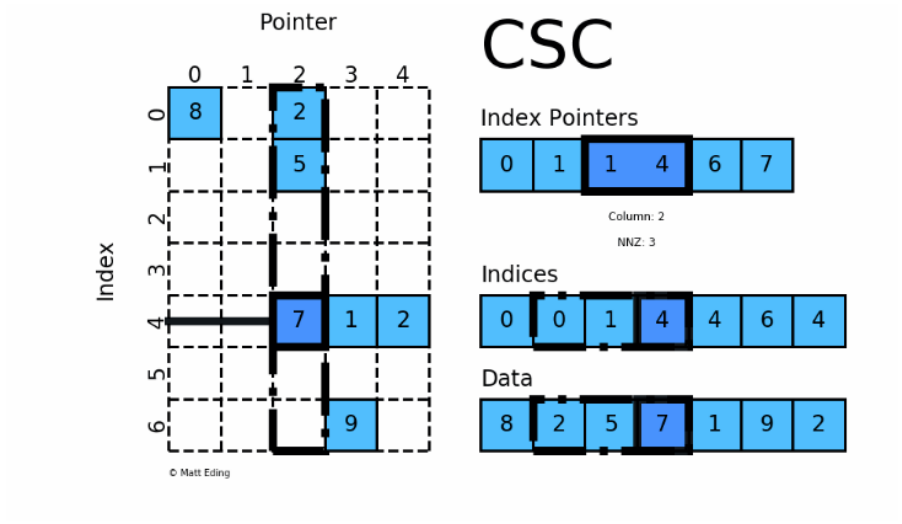
This project's non-theoretical portion included implementation of sparse matrix storage in the COO format. In general both the LIL and COO are forms which involve linked lists and are more geared towards the modification of matrices as in constructing them from the original data source. However, there are alternative implementations of sparse matrix storage which are usually utilized for more efficient matrix operations. So, before diving into an algorithm which heavily utilizes matrix operations to conduct solving a system of linear equations, it is beneficial to make sure that a different container for the sparse matrices is properly defined.

These implementations rely mostly on parallel arrays and are the Compressed Sparse Row (CSR or CRS) and Compressed Sparse Column (CSC or CCS). These are more oriented towards reading with their constant access times stemming from their array data structure utilization. For the Compressed Sparse Row and Column formats you have three arrays. One filled with adjacent pairs of index pointers where their index in the array is the row number and the difference between the two of them is the amount of non-zero elements held in that row. Meanwhile, there are two other arrays one which holds the column indices of where the location of the non-zero data element(s) are in the rows. The number of cells from the array to be read relies on the difference of the above pairs of indices of the first array, and these cells are stored continuously next to each other representing. Then, the last array is simply the data which is accessed in tandem by the same order and number of cells in a row as the column indices array. When, the number of cells which is dictated by that first difference is traversed, whenever the next difference in pairs of indices of the first array is larger than 0, the values in the two other arrays will continue to be read. So, while somewhat convoluted, it's much more space compact compared to a normal 2D array representation.





For the compressed sparse column representation, the first array is in reference to the column indices meanwhile the second and third arrays reference the row indices to point out non-zero values in each column.



5.2.2 Conjugate Gradient

The following is a mathematical coverage of what goes on with the conjugate gradient algorithm. The conjugate gradient method is simply an algorithm which sets out to solve a system of linear equations in the form **Ax = b**.

The **x** vector representation is a length $\mathbf{d_k}$ vector. The matrix A in this case has to be a symmetric positive definite matrix which simply means that A is equivalent to its transpose ($\mathbf{A}^T = \mathbf{A}$) to be defined as symmetrical, and $\mathbf{x}^T\mathbf{Ax} > \mathbf{0}$ for all non-zero vectors **x** in $\mathbf{R}^n$ in order to be positive definite, and lastly the values have to be real (non-complex/imaginary). In this regard, we are limited, but thanks to things like preconditioners we can make our algorithm work more effectively and in more cases; however, they will not be covered in this report.

This algorithm relies on solving a system of linear equations being parallel to finding the minimum of a function defined as $\mathbf{f(x) = 1/2x^TAx - b^Tx + c}$. This convex function utilizes calculating the gradient along the function till the point where the gradient is equal to 0. This method is similar to how one would find roots of a quadratic equation by setting it to 0. When we find the coordinates within a function which yield to the 0 gradient we have solved the system of linear equations.

The definition of A in this context of f(x) is the Hessian matrix of the function in question we’re solving for, and the gradient is defined by Ax - b. Generally, the Hessian helps determine saddle points of a function and the local extremum whereas the gradient is the main use of this algorithm for finding steepest descent and eventually the overall minimum.

$$A > o \in R^{nxn}, b \in R^n, c \in R$$

$$f(x) = 1/2x^TAx - b^Tx + c. \text{ (Quadratic form)}$$

$$\nabla f(x)=Ax-b \text{ (Gradient)}$$

$$\nabla^2 f(x)=A \text{ (Hessian)}$$

We're attempting to find the minimization of this convex function:

$$f(x) = 1/2x^TAx - b^Tx$$

Since the properties of A are symmetric and positive definite, we can make the leap that only one set of x length vectors which equates to the gradient of that function at those values will equal 0 and be the solution to this equation. More concisely,

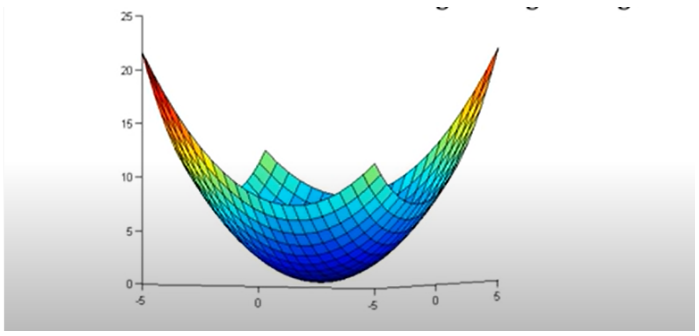
$$\text{there exists only one } x^* \text{ such that } \nabla f(x^*)=0$$

So, when ∇**f(x) =0** we have found the minimum of the function, meaning:

$$\nabla f(x)=Ax-b$$
$$0 = Ax - b$$
$$Ax = b$$

Thus, we can see that finding the gradient at 0 would solve our system and we're set to solve the minimization of this system.

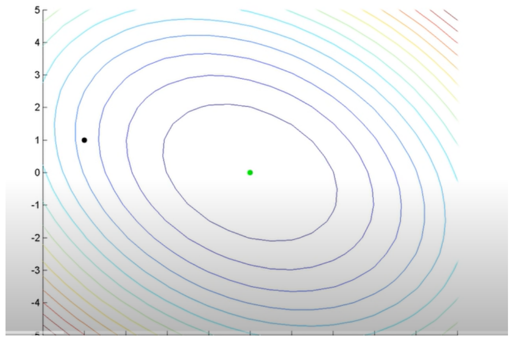
The conjugate gradient method of solving systems of linear equations is done iteratively over multiple points of Ax = b and is actually very similar to that intuitive factorization type of method as multiple points are tried in order to get to the solution of this minimization problem via the gradient.



(Convex function representation)

Now, with the function f(x) we are attempting to solve the minimization problem which can be referred to as the quadratic form within a matrix implementation. The next step would to be define the iterative method for solving ∇**f(x)=Ax-b** into the residual form:

Isocontours



$$r(x) := \nabla f(x) = Ax - b$$

So, when we iterate x_k we have the form of

$$r_k = Ax_k - b.$$

From here, we can take the first step of initially taking the first gradient of an arbitrary point (pictured on the isocontours of the former function graph shown above) which we represent within the length vector x_k. This point will give us an initial direction and function defined by its gradient value which we will conduct a line search along in order to find a conjugate point from that initial gradient to get a direct line to the minimum value of x* along the line of descent.

Line search:

$$x_{k+1} = x_k + \alpha_k p_k$$

α_k is a positive scalar known as the step length and p_k defines the step direction.

$$\nabla f(x_k + \alpha p_k)^T p_k \geq f(x_k) + c_2 \nabla f_k^T p_k$$

Here, we would define the inexact line search parameters to be advantageous to satisfying the conjugate and keeping in mind Wolfe's linear search conditions that characterize how we step along the initial gradient line.

$$\alpha_k = -\frac{\nabla f(x_k)^T p_k}{p_k^T A p_k} = -\frac{r_k^T p_k}{p_k^T A p_k}.$$

This will only allow us to partly achieve the conjugate gradient's main goal of searching in n conjugate directions where n is the number of variables represented in a system and the directions they span meaning everytime progress is made in one direction, it doesn't interfere with progress in the other directions. In the end, we can make n searches in n directions to find the optimal point.

To make sure we conduct this type of search we must set the direction along the iterations we have to ensure them as conjugates by the condition.

$$p_i^T A p_j = 0 \text{ for all } i \neq j.$$

Where the p_i and p_j are directional vectors whose elements are conjugates with respect to the Hessian.

Each direction of p_k is selected to be a linear combination of the negative residual $-r_k$ (which since r_k is equal to the gradient of f(x) it shows the steepest descent as that is what negative gradient will give you) and the previous direction p_{k-1}.

$$p_k = -r_k + \beta_k p_{k-1},$$

Now, we have been able to define that β_k will hold the requirement of step length to ensure that p_{k-1} and p_k are conjugate with respect to A.

This yields the β_k:

$$\beta_k = \frac{r_k^T A p_{k-1}}{p_{k-1}^T A p_{k-1}}.$$

So, you will simply calculate the direction via p_k and the step via α_k in order to increment the next iteration of the x vector for:

$$x_{k+1} = x_k + \alpha_k p_k.$$

These steps all accumulate to provide an iterative algorithm for the conjugate gradient.

1. Initialize x_0
2. Calculate $r_0 = Ax_0 - b$
3. Assign $p_0 = -r_0$
4. For iterations k = 0,1,2,...:
    a. Calculate
       The step $\alpha_k = -\frac{\nabla f(x_k)^T p_k}{p_k^T A p_k} = -\frac{r_k^T p_k}{p_k^T A p_k}.$

    b. Update $x_{k+1} = x_k + \alpha_k p_k$
       The vector

    c. Calculate $r_{k+1} = Ax_{k+1} - b$
       Calculate the new gradient by adjusting the term Ax -b

    d. Calculate $\beta_k = \frac{r_k^T A p_{k-1}}{p_{k-1}^T A p_{k-1}}.$

       Calculate the beta of linear search to provide a conjugate direction

e. Update $p_{k+1} = -r_{k+1} + \beta_{k+1} p_k$
The new direction and repeat until gradient = 0

https://towardsdatascience.com/complete-step-by-step-conjugate-gradient-algorithm-from-scratch-202c07fb52a8

This proves to be somewhat of an arduous process but provides an optimal way to avoid larger swings of gradient calculations brought on by regular gradient descent and allow for a much shorter path taken to the solution of the system.

5.2.3 Problems related to Systems of Linear Equations

The first example for this application would be GPS systems for a myriad of devices utilizing space satellites. The basic idea behind them is that four separate satellites are used to confirm the coordinates of a person and utilize this system of equations involving the x,y,z coordinates of a cartesian system which tracks a receiver's position and the value d which is the difference in time between the receiver of the satellite signal and the satellites' clocks. A, B, and C shown below are representative of the coordinates of the satellites while t is the travel time for the signal from the satellite to the receiver.

$$\begin{aligned}
(x-A_1)^2+(y-B_1)^2+(z-C_1)^2-(c(t_1-d))^2&=0\\
(x-A_2)^2+(y-B_2)^2+(z-C_2)^2-(c(t_2-d))^2&=0\\
(x-A_3)^2+(y-B_3)^2+(z-C_3)^2-(c(t_3-d))^2&=0\\
(x-A_4)^2+(y-B_4)^2+(z-C_4)^2-(c(t_4-d))^2&=0
\end{aligned}$$

From here, we can imagine the data set used in a conjugate gradient theorem to solve for the approximate location of the receiver.