# Final Report: H-matrix solver and H-matrix invertsion

Chao Chen
ICME, Stanford University

March 19, 2015

## 1 Introduction

This work started as a class project for CME 335 (Advanced Topics in Numerical Linear Algebra) at Stanford. The original plan was to compute the explicit inverse of an h-matrix under standard admissibility, but I became interested in (preconditioned) iterative solvers after implementing the h-matrix solve algorithm. The work for h-matrix inversion is still going on and I will show some preliminary results in this report.

To be brief, I have implemented an h-matrix solver for week admissible matrices and used it as preconditioner for iterative methods, including fix-point iteration and conjugate gradient. The algorithm for the h-matrix solver is not original, please refer to [cite Amir] for the details.

The solver is the crucial part in week h-matrix inversion. Once the week-admissible matrix is inversed, we want to use it as the initial guess for Newton-Schulz and hopefully obtain the inversion of standard h-matrix.

## 2 Introduction

This report summarizes the project progress and the plan for the rest of the quarter. A fast solver for weak-admissible matrices with complexity $O(n\log^2(n))$ has been implemented. The code takes a (weak) hierarchical matrix, stores it in sixteen-nary tree structure (every node has at most 16 children) and solves the linear system. The c++ code can be found at my github H-Matrix-Inverse repository. This implementation uses the Eigen linear algebra library. When I first started, I have referred to the following two codes for the hierarchical tree structure:DMHM and HMat.jl. In the rest of the report, Matlab notation will be used in a couple of places for convenience.

## 3 Performance Results

First of all, the solver can be used as a direct solver meaning that the result is exact (except for round-off error). And the test matrix is a five-point finite difference matrix. Since the rank of the off-diagonal blocks are $\sqrt{a}$ where $a$ is the size of the block, the solver should scale as $O(n^3)$ where $n$ is the total number of grids. The following picture shows the timing results.

**Machine** The experiments are done on the cs sapling, below are the processor information.

| Grid $(n * n)$ | 128 | 64 | 32 | L | F | A | Pts |
|---|---|---|---|---|---|---|---|
| Time | 3.01 | 0.109 | 0.0067 | 10 | 5 | 12 | |

# 4 Preconditioned Conjugate Gradient

The original conjugate gradient method [cite ?] is as follows

**Algorithm 1** CG

1. Compute $r - 0 = b - Ax_0, p_0 = r_0$
2. For $j = 0, 1, ...$ until convergence
3. $\quad \alpha = r_j^T r_j / p_j^T A p_j$
4. $\quad x_{j+1} = x_j + \alpha p_j$
5. $\quad r_{j+1} = r_j - \alpha A p_j$
6. $\quad \beta = r_{j+1}^T r_{j+1} / r_j^T r_j$
7. $\quad p_{j+1} = r_{j+1} + \beta p_j$
8. EndFor

Suppose we have the cholesky factorization $M = LL^T$, the split preconditioned system [cite saad] would be

$$L^{-1} A L^{-T} y = L^{-1} b$$

where $y = L^T x$ and using algorithm **??** yields the following procedure

**Algorithm 2** split preconditioned CG

1. Compute $\hat{r}_0 = L^{-1} b - L^{-1} A x_0, \hat{p}_0 = \hat{r}_0$
2. For $j = 0, 1, ...$ until convergence
3. $\quad \hat{\alpha} = \hat{r}_j^T \hat{r}_j / \hat{p}_j^T L^{-1} A L^{-T} \hat{p}_j$
4. $\quad \hat{x}_{j+1} = \hat{x}_j + \hat{\alpha} \hat{p}_j$
5. $\quad \hat{r}_{j+1} = \hat{r}_j - \hat{\alpha} L^{-1} A L^{-T} \hat{p}_j$
6. $\quad \hat{\beta} = \hat{r}_{j+1}^T \hat{r}_{j+1} / \hat{r}_j^T \hat{r}_j$
7. $\quad \hat{p}_{j+1} = \hat{r}_{j+1} + \hat{\beta} \hat{p}_j$
8. EndFor

almost every character is in hat, meaning that they are not the same value as their counterpart in algorithm **??**. Note algorithm **??** requires the explicit cholesky decomposition of the preconditioner, but we will show this is actually not necessary and the hat symbols are closely related to the unhat values.

Note

$$\begin{aligned}
\hat{r}_0 &= L^{-1} r_0 \\
z_0 &= L^{-T} \hat{r}_0 = M^{-1} r_0 \\
\tilde{p}_0 &= L^{-T} \hat{p}_0 = M^{-1} p_0
\end{aligned}$$

$$\begin{aligned}
\hat{r}_j &= L^{-1} r_j \\
\tilde{p}_j &= L^{-T} \hat{p}_j = M^{-1} p_j \\
z_j &= L^{-T} \hat{r}_j = M^{-1} r_j \\
\hat{r}_j^T \hat{r}_j &= r_j^T L^{-T} L^{-1} r_j = r_j^T M^{-1} r_j = r_j^T z_j
\end{aligned}$$

$$\tilde{x}_j = L^{-T} \hat{x}_j = x_j$$

**Algorithm 3** left preconditioned CG

1. Compute $r_0 = b - Ax_0, z_0 = M^{-1} r_0, \tilde{p}_0 = z_0$
2. For $j = 0, 1, ...$ until convergence
3. $\quad \tilde{\alpha} = r_j^T z_j / \tilde{p}_j^T A \tilde{p}_j = \hat{\alpha}$
4. $\quad \tilde{x}_{j+1} = \tilde{x}_j + \tilde{\alpha} \tilde{p}_j \quad$ i.e. $x_{j+1} = x_j + \tilde{\alpha} \tilde{p}_j$
5.1 $\quad r_{j+1} = r_j - \hat{\alpha} A \tilde{p}_j$
5.2 $\quad z_{j+1} = M^{-1} r_{j+1}$
6. $\quad \tilde{\beta} = r_{j+1}^T z_{j+1} / r_j^T z_j = \hat{\beta}$
7. $\quad \tilde{p}_{j+1} = z_{j+1} + \tilde{\beta} \tilde{p}_j$
8. EndFor

The point is CG is still applicable, although it seems $M^{-1}Ax = M^{-1}b$ does not preserve SPD. Also the simplified version is as below

**Algorithm 4** left preconditioned CG

1. Compute $r_0 = b - Ax_0, z_0 = M^{-1} r_0, \tilde{p}_0 = z_0$
2. For $j = 0, 1, ...$ until convergence
3. $\quad \tilde{\alpha} = r_j^T z_j / \tilde{p}_j^T A \tilde{p}_j$
4. $\quad x_{j+1} = x_j + \tilde{\alpha} \tilde{p}_j$
5.1 $\quad r_{j+1} = r_j - \hat{\alpha} A \tilde{p}_j$
5.2 $\quad z_{j+1} = M^{-1} r_{j+1}$
6. $\quad \tilde{\beta} = r_{j+1}^T z_{j+1} / r_j^T z_j$
7. $\quad \tilde{p}_{j+1} = z_{j+1} + \tilde{\beta} \tilde{p}_j$
8. EndFor

## 5 Two Step Solve

First, we quickly review the algorithm to solve a 2x2 block matrix where the off-diagonal blocks are low rank. Given such a matrix $A$, we can factorize it by pulling out the diagonal blocks as follows.

$$A = \begin{pmatrix} D_0 & U_0 V_1^T \\ U_1 V_0^T & D_1 \end{pmatrix} = \begin{pmatrix} D_0 & \\ & D_1 \end{pmatrix} \begin{pmatrix} I & u_0 V_1^T \\ u_1 V_0^T & I \end{pmatrix}$$

where $D_0 u_0 = U_0, D_1 u_1 = U_1$. This factorization is useful because the second part can be inverted easily with Woodbury matrix identity.

$$\begin{pmatrix} I & u_0 V_1^T \\ u_1 V_0^T & I \end{pmatrix}^{-1} = I - \begin{pmatrix} u_0 & \\ & u_1 \end{pmatrix} \begin{pmatrix} I & V_1^T u_1 \\ V_0^T u_0 & I \end{pmatrix}^{-1} \begin{pmatrix} & V_1^T \\ V_0^T & \end{pmatrix}$$

Therefore, we have come to a recursive algorithm to solve linear systems with weak-admissibility.

---

**Algorithm 5** solve 2 by 2 block matrix

---

Problem:

$$\begin{pmatrix} D_0 & U_0 V_1^T \\ U_1 V_0^T & D_1 \end{pmatrix} \begin{pmatrix} x0 \\ x1 \end{pmatrix} = \begin{pmatrix} b0 \\ b1 \end{pmatrix}$$

Algorithm:

(1) recursively solve: $D_0[d0, u0] = [b0, U0]$ and $D_0[d1, u1] = [b1, U1]$

(2) solve a small linear system:

$$\begin{pmatrix} I & V_1^T u_1 \\ V_0^T u_0 & I \end{pmatrix} \begin{pmatrix} \eta_0 \\ \eta_1 \end{pmatrix} = \begin{pmatrix} V_1^T d1 \\ V_0^T d0 \end{pmatrix}$$

(3) return the solution of the original problem:

$$\begin{pmatrix} x0 \\ x1 \end{pmatrix} = \begin{pmatrix} d0 - \eta_0 * u0 \\ d1 - \eta_1 * u1 \end{pmatrix}$$

---

Second, for problems in two dimensions, when the problem domain is sub-divided in every dimension i.e. the domain becomes $2^2 = 4$ pieces and correspondingly, the hierarchical matrix becomes 4 by 4 blocks. As a result, the above algorithm can not be used directly. Fortunately the dense blocks only appear on the diagonal block due to weak admissibility. The other $16 - 4 = 12$ off-diagonal blocks are all low rank and thus the matrix can be view as a two level 2x2 block matrix as shown in figure (1).

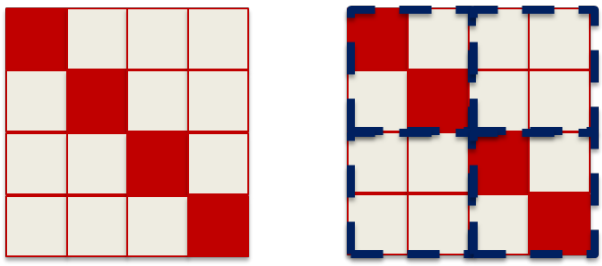

Figure 1: The 4x4 hierarchical matrix can be viewed as a 2x2 block matrix where each diagonal block is again a 2x2 block matrix

Therefore we come to the final algorithm if we apply algorithm 1 multiple times.

**Algorithm 6** solve 4 by 4 block matrix

(1) call algorithm 1 for two 2x2 diagonal blocks
(2) compress two off-diagonal 2x2 low rank blocks
(3) call algorithm 1 again to solve the 2x2 block view of the original matrix

Note that algorithm (1) will call algorithm (2) for solving the diagonal hierarchical blocks, so what happens here is a zig-zag order of function call, as shown in figure (2).

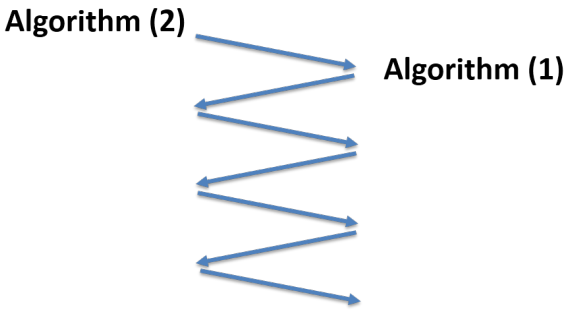


Figure 2: Zig-zag function call

## 6 2 by 2 Low Rank Block Compression

Now we want to address step (2) in algorithm 2 i.e. how to find the low rank representation for a 2x2 low rank blocks. The c++ code uses one of the easiest way by taking advantage of the following equality

$$\begin{pmatrix} U_0V_0^T & U_1V_1^T \\ U_2V_2^T & U_3V_3^T \end{pmatrix} = \begin{pmatrix} U_0 & & U_1 & \\ & U_2 & & U_3 \end{pmatrix} \begin{pmatrix} V_0^T & & & \\ V_2^T & & & \\ & & V_1^T & \\ & & V_3^T & \end{pmatrix} = \begin{pmatrix} U_0 & U_1 & & \\ & & U_2 & U_3 \end{pmatrix} \begin{pmatrix} V_0^T & & V_1^T & \\ & V_2^T & & \\ & & & V_3^T \end{pmatrix} \equiv UV^T$$

Note that all the matrices above are in fact skinny matrices, so the shape shown above is not realistic. The the big $U$ matrix should be skinny and the big $V^T$ matrix should be fat. Apparently, this is not an optimal representation without further compressing $[U_0, U_1]$, $[U_2, U_3]$, $[V_0^T; V_1^T]$ and $[V_1^T; V_3^T]$. To achieve a nearly optimal compression, we need fast algorithm to approximate the low rank factorization, which is on the todo list.

## 7 Conclusion and Future Work

I have implemented a fast solver for weak-admissible matrices, which will be used to compute the explicit inverse of such matrices. While the code works well for simple test cases, it definitely needs more furnish and optimization. One thing on the todo list is to have more tests and optimize a couple of important routines e.g. the ComputeLowRank() routine in helper.cpp. This function returns the low rank representation of a give dense/sparse matrix. When the input matrix $A$ is sparse,

$$A = \sum_k A(i_k, j_k) e_{i_k} e_{j_k}^T$$

is a straightforward low rank factorization. But this does not apply to general matrices, where an efficient algorithm is needed for the low rank factorization.

The next step towards the final goal of this project is clear if we review the algorithm in the proposal, which is also listed below. To be brief, I will focus on algorithm 3 step (4) i.e. updating a hierarchical matrix with a low rank matrix. For me, this involves both theoretical and programming work.

**Algorithm 7** Inverse of H-matrix

Problem:

$$\begin{aligned} A^{-1} &= \begin{pmatrix} I & \bar{U}_0 V_1^T \\ \bar{U}_1 V_0^T & I \end{pmatrix}^{-1} \begin{pmatrix} D_0^{-1} & \\ & D_1^{-1} \end{pmatrix} \\ &= \begin{pmatrix} D_0^{-1} - \bar{U}_0 B_{00} V_1^T D_1^{-1} & -\bar{U}_0 B_{01} V_0^T D_0^{-1} \\ -\bar{U}_1 B_{10} V_1^T D_1^{-1} & D_1^{-1} - \bar{U}_1 B_{11} V_0^T D_0^{-1} \end{pmatrix} \end{aligned}$$

where

$$\begin{pmatrix} I & V_1^T \bar{U}_1 \\ V_0^T \bar{U}_0 & I \end{pmatrix}^{-1} = \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix}$$

Proposed algorithm:
(1) Recursively compute $D_0^{-1}$ and $D_1^{-1}$
(2) Compute matrix-matrix product $\bar{U}_0 = D_0^{-1} U_0$, $\bar{U}_1 = D_1^{-1} U_1$, $V_0^T D_0^{-1}$ and $V_1^T D_1^{-1}$
(3) Compute the inverse of a small matrix, i.e. $B_{00}, B_{01}, B_{10}, B_{11}$
(4) Update $D_0^{-1}$ and $D_1^{-1}$ by $-\bar{U}_0 B_{00} V_1^T D_1^{-1}$ and $-\bar{U}_1 B_{11} V_0^T D_0^{-1}$ respectively