A FAST MEMORY EFFICIENT CONSTRUCTION ALGORITHM FOR HIERARCHICALLY SEMI-SEPARABLE REPRESENTATIONS

K. LESSEL[†], M. HARTMAN[‡], AND S. CHANDRASEKARAN[§]

Abstract. Existing Hierarchically Semi-Separable (HSS) construction algorithms for dense $n \times n$ matrices require as much as $O(n^2)$ peak workspace memory, at a cost of $O(n^2)$ flops. An algorithm is presented which requires $O(n^{1.5})$ peak worskpace memory in the worst case, while still requiring only $O(n^2)$ flops.

Key words. fast multipole method, hierarchically semiseparble representations, fast algorithms, orthogonal factorizations, low-rank structures, memory efficient

AMS subject classification. 65F05

1. Introduction. The fast multipole method (FMM) was originally introduced by Greengard and Rokhlin [5] and since their contribution, it has become clear that such matrices do arise commonly in practice [14], [17]. In this paper we will present a memory efficient construction algorithm for $n \times n$ matrices with off diagonal blocks that have low rank. The memory consumption of the HSS representation itself is $O(n)$ if the rank of the off-diagonal blocks is small. If the user is not required to store the matrix A, but instead only provides a functional interface in order to access the elements of the matrix, it is worthwhile to ask for the algorithm which computes the HSS representation to be memory efficient as well. Previous algorithms, [2], [16], have shown the HSS representation can be computed in $O(n^2)$ flops. Randomized algorithms also exist [10], [13], [15]. However, the memory requirements of these algorithms can be excessive, requiring as much as $O(n^2)$ peak workspace memory [2], [10], [13], [16]. In this paper, we deal with this issue and present an algorithm that requires $O(n^{1.5})$ peak workspace memory in the worst case, while still requiring only $O(n^2)$ flops. In section 2 we will briefly cover the HSS representation, followed by the memory efficient HSS construction algorithm in section 3. Memory and flop counts are discussed in section 4 and 5, respectively, followed by some experimental results of our construction algorithm in section 6.

2. HSS Representation. Define a **regular binary tree** to be an ordered rooted tree in which each parent node has two children. Denote the set of regular binary trees as $\mathcal{T}_{reg}$. Specifically each child of a parent node is either a **left child** or a **right child**. A subtree rooted at the left (right) child of a given node, r, is called r's **left (right) subtree** [1]. Each node in this regular binary tree is associated with an index $(k; i)$, where k denotes the node's depth from the root node, and i denotes left to right ordering in that level. Label the root node as $(0; 1)$. Then every parent node has a labeling $(k; i)$, with left child labeled as $(k + 1; 2i - 1)$ and right child labeled as $(k + 1; 2i)$.

Define a **partition tree** to be a regular binary tree that has an integer, $n_{k;i}$ at every node, $(k; i)$, which satisfy the property that $n_{k;i} = n_{k+1;2i-1} + n_{k+1;2i}$. A partition tree, T, of depth d, is **complete**, if all levels $k \in \{0, 1, ..., d - 1\}$ of T have 2^k vertices [1].

We use partition trees to hierarchically partition a matrix $A \in \mathbb{R}^{m \times n}$. Let T_R and T_c be partition trees. Let $m_{k;i}$ and $n_{k;i}$ denote the integer at node $(k; i)$ in T_R and T_c, respectively. Let $A_{0;1,1} = A$, $m_{0;1} = m$ and $n_{0;1} = n$. Partitioning the rows and columns of the matrix A according to the integers at the first level of T_R and T_c,

$$A_{0;1,1} = \begin{matrix} \\ m_{1;1} \\ m_{1;2} \end{matrix} \begin{matrix} \begin{matrix} n_{1;1} & n_{1;2} \end{matrix} \\ \begin{pmatrix} A_{1;1,1} & A_{1;1,2} \\ A_{1;2,1} & A_{1;2,2} \end{pmatrix} \end{matrix}. \tag{2.1}$$

[†]Department of Electrical and Computer Engineering, University of California at Santa Barbara, Santa Barbara, CA 93106 (klessel@engineering.ucsb.edu). The reasearch of this author was supported in part by the National Science Foundation under Grant nos. CCF-0830604 and CCF-1450321, and by DARPA under contract no. W31P4Q-15-C-0074.

[‡]Department of Electrical and Computer Engineering, University of California at Santa Barbara, Santa Barbara, CA 93106 (mhartman@umail.ucsb.edu)

[§]Department of Electrical and Computer Engineering, University of California at Santa Barbara, Santa Barbara, CA 93106 (shiv@ece.ucsb.edu). The research of this author was supported in part by the National Science Foundation under Grant nos. CCF-0830604 and CCF-1450321, and by DARPA under contract no. W31P4Q-15-C-0074.

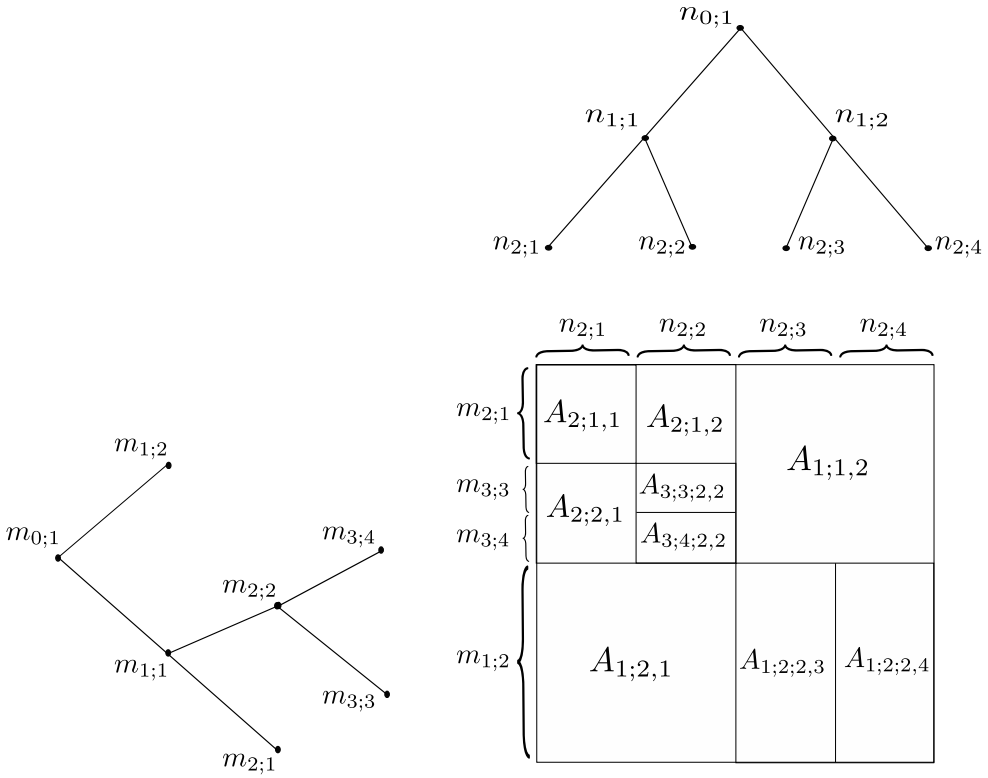


Fig. 2.1: Example Block Partitioning of a Matrix, A, with Corresponding Partition Trees, T_R (bottom left) and T_C (top right)

Recursively partitioning the block rows and block columns of A according to the integers stored at the corresponding nodes of T_R and T_C, respectively,

$$A_{k-1;i,i} = \begin{matrix} \\ m_{k;2i-1} \\ m_{k;2i} \end{matrix} \begin{matrix} \begin{matrix} n_{k;2i-1} & n_{k;2i} \end{matrix} \\ \begin{pmatrix} A_{k;2i-1,2i-1} & A_{k;2i-1,2i} \\ A_{k;2i,2i-1} & A_{k;2i,2i} \end{pmatrix} \end{matrix}. \tag{2.2}$$

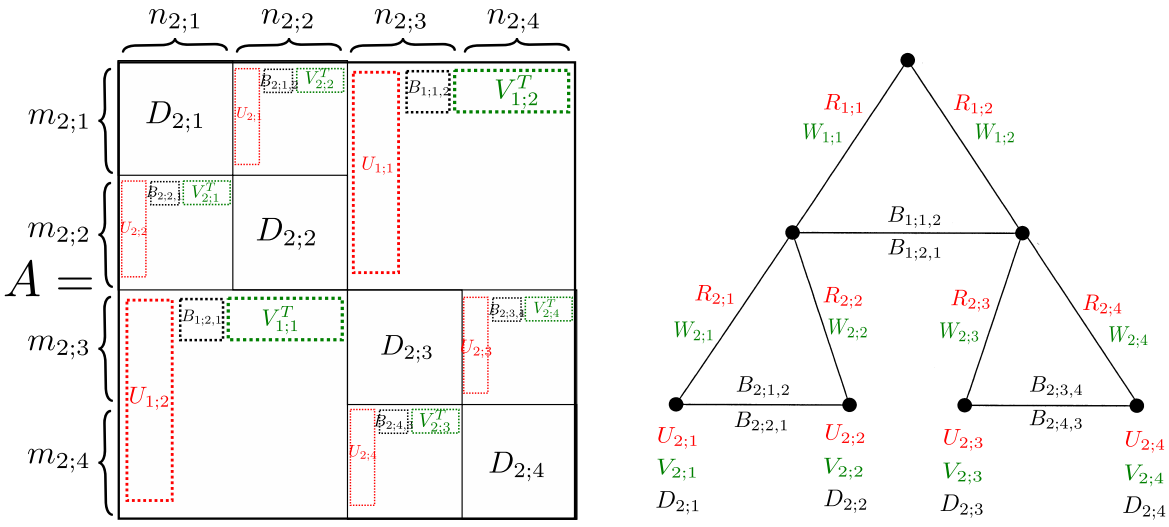
Figure 2.1 shows an example of a matrix which has rows partitioned according to a partition tree T_R, and columns partitioned according to a partition tree T_C. In any level, when it is the case that the row partitions are the same as the column partitions, we use the more simplified notation, $A_{k;i,j} = A_{k;i,k;j}$ (Figure 2.1). In this case, it is the convention that the partition tree corresponding to the row partitions and the partition tree corresponding to the column partitions are merged. For the purpose of this paper we will focus on this case. Any matrix has an HSS representation, but if the matrix has off diagonal blocks with low numerical rank, this representation will consume less memory, and is defined as follows. Let T be a partition tree for the matrix A. Partition the rows and columns of A according to T as defined above. Suppose, $D_{k;i} = A_{k;i}$ if $(k;i)$ is a leaf node and if $(k;i)$ is not a leaf node,

$$\begin{aligned} A_{k;2i-1,2i} &= U_{k;2i-1} B_{k;2i-1;2i} V_{k;2i}^T, \\ A_{k;2i-1,2i} &= U_{k;2i} B_{k;2i;2i-1} V_{k;2i-1}^T, \end{aligned} \tag{2.3}$$

such that equation (2.4) holds.

$$U_{k;i} = \begin{pmatrix} U_{k+1;2i-1} R_{k+1;2i-1} \\ U_{k+1;2i} R_{k+1;2i} \end{pmatrix}, \quad V_{k;i} = \begin{pmatrix} V_{k+1;2i-1} W_{k+1;2i-1} \\ V_{k+1;2i} W_{k+1;2i} \end{pmatrix}. \tag{2.4}$$

Then, the **HSS representation** consists of **basis matrices**, $U_{k;i}$, and $V_{k;i}$, as well as $D_{k;i}$ for all leaf nodes, $(k;i)$. Further, for all $(k;i)$, which are not leaf nodes, the representation consists of **translation operators**,



(a) 2 level HSS (b) 2 level HSS tree

Fig. 2.2: 2 level HSS Matrix and Binary Tree

$R_{k;i}$, $W_{k;i}$, and **expansion coefficients**, $B_{k;i-1,i}$, for i odd, and $B_{k;i;i-1}$ for i even. The existence of this factorization is established [2], and this will become more obvious as we proceed.

Define an **HSS tree** of the matrix A to be the corresponding partition tree of A decorated with the matrices $U_{k;i}$, $V_{k;i}$, $D_{k;i}$, $R_{k;i}$, $W_{k;i}$, $B_{k;i,j}$. We store $U_{k;i}$, $V_{k;i}$, $D_{k;i}$ at each leaf node $(k;i)$. $R_{k;i}$ and $W_{k;i}$ are stored at each edge connecting parent to child node, $(k;i)$. Further we add edges to the partition tree from node $(k;i)$ to node $(k;j)$ corresponding to $B_{k;i,j}$. Since $U_{k;i}$ $(V_{k;i})$ are only stored at leaf nodes, only the smaller $R_{k;i}$ $(W_{k;i})$ are stored at each subsequent level. Basis matrices at higher levels are obtained via matrix multiplication with corresponding translation operators (2.4). In figure 2.2 we give an example of a 2-level HSS representation of a matrix, along with its corresponding HSS tree. In order that the translation operators $R_{k;i}$ and $W_{k;i}$ be as defined above, we must have that each $U_{k;i}$ be a column basis for the corresponding row Hankel block, which is

$$ {}_r\acute{H}_{k;i} = \begin{pmatrix} A_{k;i,1} & A_{k;i,2} & \dots & A_{k;i,i-1} & A_{k;i,i+1} & \dots & A_{k;i,end} \end{pmatrix}. \tag{2.5} $$

And likewise, we must have that each $V_{k;i}$ be a row basis for its corresponding column Hankel block, which is

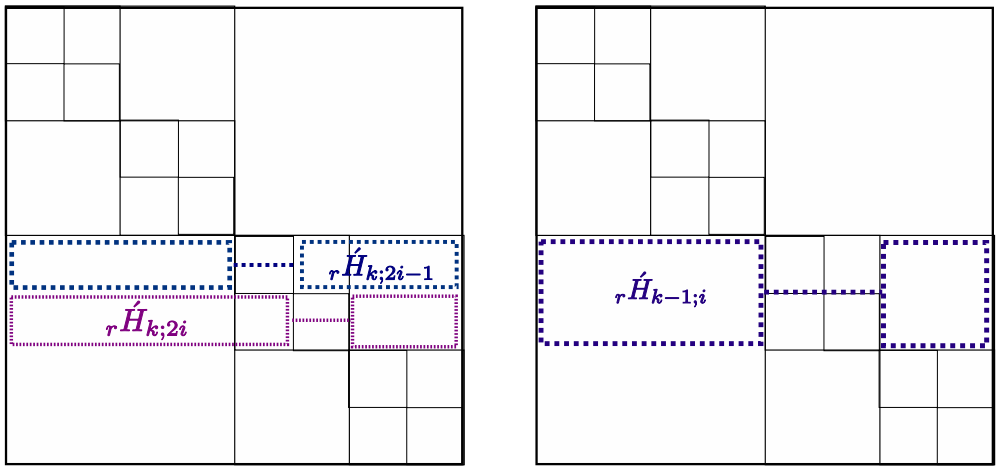
$$ {}_c\acute{H}_{k;i} = \begin{pmatrix} A_{k;1,i}^T & A_{k;2,i}^T & \dots & A_{k;i-1,i}^T & A_{k;i+1,i}^T & \dots & A_{k;end,i}^T \end{pmatrix}. \tag{2.6} $$

Notice that, ${}_r\acute{H}_{k;i}$ is the i^{th} block row of A in the k level HSS structure, excluding the $A_{k;i,i}$ block. Similarly, ${}_c\acute{H}_{k;i}$ is the i^{th} block column of A corresponding to the k^{th} level of the HSS representation, excluding the $A_{k;i,i}$ block as illustrated by Figures 2.3a, and 2.3b. The i^{th} block row of A in the $(k-1)^{st}$ level HSS structure, excluding the $A_{k-1;i,i}$ block, is ${}_r\acute{H}_{k-1;i}$ (Figure 2.3b) , and is defined as

$$ \begin{aligned} {}_r\acute{H}_{k-1;i} &= \begin{pmatrix} A_{k;2i-1,1} & A_{k;2i-1,2} & \dots & A_{k;2i-1,2i-2} & A_{k;2i-1,2i+1} & \dots & A_{k;2i-1,end} \\ A_{k;2i,1} & A_{k;2i,2} & \dots & A_{k;2i,2i-2} & A_{k;2i,2i+1} & \dots & A_{k;2i,end} \end{pmatrix} \\ &= \begin{pmatrix} A_{k-1;i,1} & A_{k-1;i,2} & \dots & A_{k-1;i,i-1} & A_{k-1;i,i+1} & \dots & A_{k-1;i,end} \end{pmatrix}. \end{aligned} \tag{2.7} $$

The same definition is extended to the column Hankel blocks.

The storage of a matrix $A \in \mathbb{R}^{n\times n}$ in general will consume $O(n^2)$ memory, whereas the HSS representation can potentially require much less than $O(n^2)$ memory. Therefore, we are interested in construction algorithms which are efficient in both peak memory consumption as well as speed. Previoius algorithms, [2]



(a) Hankel Blocks, $_r\acute{H}_{k;2i-1}$ and, $_r\acute{H}_{k;2i}$ with $k = 3$, $i = 3$

(b) Hankel block, $_r\acute{H}_{k-1;i}$, with $k = 2$, $i = 3$

Fig. 2.3: Row Hankel Blocks

and [16], consume at most $O(n^2)$ memory, with a complexity of at most $O(n^2)$ flops. Other construction algorithms use randomized techniques [10], [13], [15], [17]. As no explicit statement is given in regard to peak memory consumption, we have inferred the memory complexity for these algorithms to be at most $O(n^2)$. $\mathcal{H}^2$ matrices are similar to HSS representations in hierarchical structure and the reader is referred to [7] for more details on complexity counts.

Here we introduce a fast algorithm that computes the HSS representation of a matrix while using at most $O(n^{1.5})$ memory. The algorithm we present is very similar to [2] and [16], with key differences introduced to minimize peak memory usage. Specifically, the algorithm separates computations into 4 passes, the traversal of the HSS tree is modified and a new method of computation for the expansion coefficients is introduced. Note that merging the 4 passes would yield a viable algorithm very similar to that given in [16], which would give a negligible savings in flops at the expense of the increase in peak memory by a factor of 4. Also note that any rank revealing factorization can be substituted for the svd in our algorithm. Several such factorizations are rank revealing QR factorization [6], [8], [3], Interpolative Decomposition [9], [10] and rank revealing LU factorization [11], [12]. If a non-orthogonal decomposition is used then the formuals to compute the translation operators must be suitably adjusted as follows,

$$R_{k+1;2i-1} = U^{\dagger}_{k+1;2i-1}\,(U_{k;i})_1, \quad R_{k+1;2i} = U^{\dagger}_{k+1;2i}\,(U_{k;i})_2, \tag{2.8}$$

where,

$$U_{k;i} = \begin{pmatrix} (U_{k;i})_1 \\ (U_{k;i})_2 \end{pmatrix}. \tag{2.9}$$

The same adjustment is extended to the column translation operators.

3. HSS Construction Algorithm. In this section we present a recursive two phase construction algorithm. The first phase (Algorithm 1) is a deepest-first algorithm that computes the basis matrices at leaf nodes and translation operators at non-leaf nodes. In the second phase (Algorithm 2 and 3), the expansion coefficients are computed. In this algorithm, rather than asking the user to give the whole dense matrix as input, the user is asked only to provide a subroutine which generates subblocks of the matrix. Similar algorithm structure has been used by others [13] [16], and has been used since the first implementation of these methods [2]. The user must also give as input the partition tree, as well as either a tolerance, ε, or alternately a fixed rank, p, which will determine the amount of compression in the HSS representation.

3.1. HSS Algorithm - Phase 1.

3.1.1. Leaf Node Computations - Phase 1. In order to obtain the $U_{k;i}$'s at the leaf node we take an SVD of each row Hankel block, ${}_r\acute{H}_{k;i}$,

$$ {}_r\acute{H}_{k;i} = ({}_rU_{k;i})({}_r\Sigma_{k;i})({}_rQ_{k;i}^T) = \begin{pmatrix} U_{k;i} & * \end{pmatrix} \begin{pmatrix} \Sigma_{k;i} & 0 \\ 0 & * \end{pmatrix} \begin{pmatrix} Q_{k;i}^T \\ * \end{pmatrix}. \tag{3.1} $$

where, by definition, both ${}_rU_{k;i}$ and ${}_rQ_{k;i}$ are orthogonal matrices, and ${}_r\Sigma_{k;i}$ is a diagonal matrix. Likewise for $V_{k;i}$ at each leaf node $(k;i)$,

$$ {}_c\acute{H}_{k;i} = ({}_cP_{k;i})({}_c\Lambda_{k;i})({}_cV_{k;i}) = \begin{pmatrix} P_{k;i} & * \end{pmatrix} \begin{pmatrix} \Lambda_{k;i} & 0 \\ 0 & * \end{pmatrix} \begin{pmatrix} V_{k;i}^T \\ * \end{pmatrix}. \tag{3.2} $$

Only the columns of ${}_rU_{k;i}$ and ${}_cV_{k;i}$ which correspond to singular values above the chosen tolerance, ε, or rank, p are kept. The user can choose to give as input a tolerance, ε, such that all singular values smaller than ε are discarded. Alternately, the user can choose to give as input the rank, p, such that both $U_{k;i}$ and $V_{k;i}$ contain at most p columns. We denote TRUNC_SVD() as a function that returns an svd to a tolerance, ε, or alternatively a pre-determined rank, p. $U_{k;i}$, and $V_{k;i}$ are stored at the leaf node $(k;i)$ in our HSS tree, and both $(\Sigma_{k;i})(Q_{k;i}^T)$ and $(P_{k;i})(\Lambda_{k;i})$ are returned to the parent function.

3.1.2. Non-Leaf Node Computations - Phase 1. In order to generate the translation operators $R_{k;2i-1}$, $R_{k;2i}$, $W_{k;2i-1}$ and $W_{k;2i}$, define

$$ {}_r\tilde{H}_{k;2i-1} = \Sigma_{k;2i-1}\, Q_{k;2i-1}^T, \tag{3.3} $$

$$ {}_c\tilde{H}_{k;2i-1} = P_{k;2i-1}\, \Lambda_{k;2i-1}, \tag{3.4} $$

$$ {}_r\tilde{H}_{k;2i} = \Sigma_{k;2i}\, Q_{k;2i}^T, \tag{3.5} $$

$$ {}_c\tilde{H}_{k;2i} = P_{k;2i}\, \Lambda_{k;2i}. \tag{3.6} $$

Vertically concatenate each pair of blocks ${}_r\tilde{H}_{k;2i-1}$ and ${}_r\tilde{H}_{k;2i}$, and remove the portion of the blocks which correspond to the columns that lie in the diagonal block $D_{k-1;i}$ as shown in Figure 3.1a. Likewise, horizontally concatenate each pair, ${}_r\tilde{H}_{k;2i-1}$ and ${}_c\tilde{H}_{k;2i}$ after removing the rows which correspond to the diagonal block $D_{k-1;i}$, as shown in Figure 3.1b. To remove these columns from ${}_r\tilde{H}_{k;2i-1}$ $({}_r\tilde{H}_{k;2i})$ we can instead remove them from $Q_{k;2i-1}$ $(Q_{k;2i})$. Expand $Q_{k;i}$ to obtain

$$ Q_{k;i}^T = \begin{pmatrix} Q_{k;i,1}^T & Q_{k;i,2}^T & \cdots & Q_{k;i,2^k-1}^T \end{pmatrix}, \tag{3.7} $$

where each $Q_{k;i,j}^T$ is of size $m_{k;i} \times n_{k;j}$. Then define

$$ \tilde{Q}_{k;i}^T = \begin{pmatrix} Q_{k;i,1}^T & \cdots & Q_{k;i,i-1}^T & Q_{k;i,i+1}^T & \cdots & Q_{k;i,2^k-1}^T \end{pmatrix}, \tag{3.8} $$

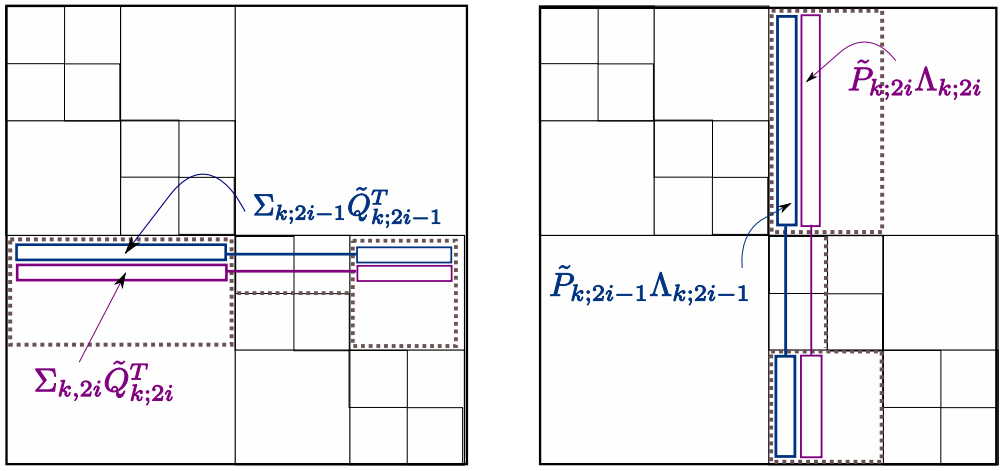
where we have excluded the block $Q_{k;i,i}^T$ from $Q_{k;i}^T$. This process can be repeated similarly for the column blocks, and the compressed row and column Hankel blocks at node $(k-1;i)$ are given by

$$ {}_rH_{k-1;i} = \begin{pmatrix} \Sigma_{k;2i-1}\, \tilde{Q}_{k;2i-1}^T \\ \Sigma_{k,2i}\, \tilde{Q}_{k;2i}^T \end{pmatrix}, \tag{3.9} $$

$$ {}_cH_{k-1;i} = \begin{pmatrix} \tilde{P}_{k;2i-1}\, \Lambda_{k;2i-1} & \tilde{P}_{k;2i}\, \Lambda_{k;2i} \end{pmatrix}. \tag{3.10} $$

A visual representation of (3.9) and (3.10) are shown in Figures 3.1a and 3.1b. The corresponding part of the original Hankel blocks, ${}_r\acute{H}_{k-1;i}$, and ${}_c\acute{H}_{k-1;i}$, with definitions given in (2.7), are outlined in brown dashed lines.

We can then determine the translation operators, $R_{k;2i-1}$, and $R_{k;2i}$, from ${}_rH_{k-1;i}$, by performing a truncated SVD (Algorithm 1, line 16). Likewise, we can determine the translation operators, $W_{k;2i-1}$, and $W_{k;2i}$, from ${}_cH_{k-1;i}$. We then proceed iteratively in order to obtain all translation operators at each node in our HSS structure. At this stage we have stored $D_{k;i}$ and we have computed and stored every basis matrix, $U_{k;i}$ and $V_{k;i}$, at the leaf nodes. At non-leaf nodes, every translation operator, $R_{k;i}$ and $W_{k;i}$, has been computed and stored.



(a) ${}_rH_{k-1;i}$, with $k = 3$, $i = 3$ (b) ${}_cH_{k-1;i}$, with $k = 3$, $i = 3$

Fig. 3.1: Compressed Row and Column Hankel Blocks

Algorithm 1 summarizes the computation of all $U_{k;i}$ at leaf nodes and $R_{k;i}$ at non-leaf nodes. We can compute $V_{k;i}$ and $W_{k;i}$ similarly by applying Algorithm 1 to A^T. Notice that nodes are visited in a **deepest first order**, by which we mean that at each parent node the algorithm will visit the child node which is the root of the subtree with the greatest depth. If the depth of both subtrees are equal, then the left child is visited first.

Algorithm 1 Pass 1U - Memory Efficient HSS Algorithm

function HSS_BASIS($tree$)
  if $tree$ is a leaf node **then**
    $(U_{k;i}, \Sigma_{k;i}, Q^T_{k;i}) = \text{TRUNC\_SVD}([A_{k;i,1} \;\; A_{k;i,2} \;\; \ldots \;\; A_{k;i,i-1} \;\; A_{k;i,i+1} \;\; \ldots \;\; A_{k;i,end}])$
    return $\Sigma_{k;i}\tilde{Q}^T_{k;i}$ ▷ $\tilde{Q}_{k;i}$ is defined as previously stated in equation (3.8)
  else ▷ $tree$ is **not** a leaf node
    $\_,\ treeL,\ treeR = tree$
    if $\text{DEPTH}(treeL) \geq \text{DEPTH}(treeR)$ **then**
      ${}_rH_{k+1;2i-1} = \text{HSS\_BASIS}(treeL)$
      ${}_rH_{k+1;2i} = \text{HSS\_BASIS}(treeR)$
    else
      ${}_rH_{k+1;2i} = \text{HSS\_BASIS}(treeR)$
      ${}_rH_{k+1;2i-1} = \text{HSS\_BASIS}(treeL)$
    end if
    ${}_rH_{k;i} = \begin{pmatrix} {}_rH_{k+1;2i-1} \\ {}_rH_{k+1;2i} \end{pmatrix}$
    ${}_rH_{k+1;2i-1} = ();\quad {}_rH_{k+1;2i} = ()$
    $\begin{pmatrix} R_{k+1;2i-1} \\ R_{k+1;2i} \end{pmatrix},\ \Sigma_{k;i},\ X_{k;i} = \text{TRUNC\_SVD}({}_rH_{k;i})$
    return $\Sigma_{k;i}\,\tilde{X}^T_{k;i}$ ▷ $\tilde{X}_{k;i}$ is defined as previously stated in equation (3.8)
  end if
end function

3.2. HSS Algorithm - Phase 2. In the second pass of our algorithm we compute all expansion coefficients, $B_{k;i,j}$, via matrix multiplication. Memory consumption for this computation will maximally require $O(p^2 n)$ in the worst case, (as compared with $O(p^2 \log n)$ for a symmetric tree).

3.2.1. Leaf Node Computations - Phase 2. For (k_1, i), (k_2, j) leaf nodes, let us define

$$B_{k_1;i,k_2,j} = U^T_{k_1,i} A_{k_1;i,k_2,j} V_{k_2;j} \tag{3.11}$$

Then, for (k_1, i), a leaf node, and (k_1, j) is not, define

$$\begin{aligned} B_{k_1;i,k_2;j} &= B_{k_1;i,k_2+1;2j-1} W_{k_2+1;2j-1} \\ &\quad + B_{k_1;i,k_2+1;2j} W_{k_2+1;2j}, \end{aligned} \tag{3.12}$$

Now for (k_1, i), not a leaf node, and (k_2, j) is a leaf node, define

$$\begin{aligned} B_{k_1;i,k_2;j} &= R^T_{k_1+1;2i-1} B_{k_1+1;2i-1,k_2,j} \\ &\quad + R^T_{k_1+1;2i} B_{k_1+1;2i,k_2,j}. \end{aligned} \tag{3.13}$$

3.2.2. Non-Leaf Node Computations - Phase 2. For both (k_1, i), and (k_2, j) not leaf nodes, we will have $k_1 = k_2$, and thus we can write

$$B_{k;i,j.} = B_{k_1;i,k_2;j}, \tag{3.14}$$

where $k = k_1 = k_2$. Let us then define

$$\begin{aligned} B_{k;i,j} &= R^T_{k+1;2i-1} B_{k+1;2i-1,2j-1} W_{k+1;2j-1} \\ &\quad + R^T_{k+1;2i-1} B_{k+1;2i-1,2j} W_{k+1;2j} \\ &\quad + R^T_{k+1;2i} B_{k+1;2i,2j-1} W_{k+1;2j-1} \\ &\quad + R^T_{k+1;2i} B_{k+1;2i,2j} W_{k+1;2j}, \end{aligned} \tag{3.15}$$

Thus we have defined a set of recursions with which we can compute any $B_{k;i,j}$ in our HSS structure.

To compute all $B_{k_1;i,k_2;j}$, we present Algorithm 2 and 3. This algorithm specifically computes the expansion coefficients, $B_{k;i-1,i}$. The expansion coefficient $B_{k;i,i-1}$ can be computed analogously by simply interchanging the pair of indices $(k_1; i)$ with $(k_2; j)$ in Algorithms 2 and 3. Note that in the following algorithm a node is a tuple which consists of data, and left and right subtree, though, the details of where we store the data are not relevant here since we access them by index notation. These are implementation dependent details and for more information the reader can refer to the Matlab code that is published on the Scientific Computing Group website [4]. Also, note that we use the notation $B_{k_1;i,k_2;j} = ()$ to mean that the variable $B_{k_1;i,k_2;j}$ is no longer needed and can be explicitly cleared from memory.

4. Memory Consumption. In this section we will compute the peak workspace consumption for Algorithms 1, 2, and 3. The HSS tree itself will consume at most $O(np^2)$ memory. In Phase 1, the main workspace consumption are the compressed Hankel blocks, (3.9), and (3.10). In Phase 2 it is the matrices $B_{k_1;i,k_2;j}$. Let $p = \max_{(k;i) \text{ leaf node}} n_{k;i}$. The assumption here is that p is small compared to n. In particular we will establish,

THEOREM 4.1. *Algorithms 1, 2 and 3 have a memory complexity of at most* $O\left(p^{0.5} n^{1.5}\right)$.

4.1. Memory Consumption - Phase 1. In Phase 1 (Algorithm 1), we visit nodes in a deepest first ordering, and notice that what further complicates our memory complexity calculation is the recursive nature of our algorithm. The worskspace allocated in one call to Algorithm 1 is added to any further recursive calls to Algorithm 1. Therefore, we have to consider the depth of the stack of these recursive calls, which is determined by the HSS tree. Therefore, for a given number of nodes, we must find the HSS tree for which peak memory complexity will be maximum. Without loss of generality, any HSS tree can be re-ordered such that the depth of the left subtree is always larger or equal to the depth of the right subtree, and therefore, in this section we only consider trees which have this property. Notice that we make memory allocations at lines 8 and 9, and lines 11 and 12 of Algorithm 1. Since we are only considering trees which have the property that at each parent node, left subtrees have depth which is larger or equal to that of the right, we will focus on calls to line 8 and line 9 only. When a call is made to line 9 a block of size $n \times p$ is left in memory from line 8. After returning from a call to line 9, line 14 is executed the block is cleared from memory at line 15. Then, for a given number of nodes, we want to find the HSS tree which will result in the maximum number of consecutive calls to line 8 without having returned from a call to line 9. Proposition 1 claims that for a given number of nodes, such a tree satisfies properties $(P1)$ and $(P2)$, which are defined in the following section. An example of a tree which satisfies these properties is given in Figure 4.2a.

Algorithm 2 Pass 2BU - Computation of Expansion Coefficients ($B_{k;i-1,i}$) Corresponding to Diagonal Blocks

Require: *tree* is not a leaf node

function B_DIAG(tree)
  $-,\ treeL,\ treeR = tree$ ▷ $\exists\ (k1;i)$ s.t. it is the numbering for the root node of $treeL$.
  ▷ $\exists\ (k2;j)$ s.t. it is the numbering for the root node of $treeR$.
  if $treeL$ is a leaf node and $treeR$ is a leaf node **then**
    $B_{k_1;i,k_2;j} = U^T_{k_1;i} A_{k_1;i,k_2;j} V_{k_2;j}$
  else if $treeL$ is **not** a leaf node and $treeR$ is **not** a leaf node **then**
    $B_{k_1;i,k_2;j}$ = B_OFFDIAG($treeL$, $treeR$)
    B_DIAG($treeL$)
    B_DIAG($treeR$)
  else if $treeL$ is a leaf node and $treeR$ is **not** a leaf node **then**
    $-,\ treeRL\ \ treeRR = treeR$
    $B_{k_1;i,k_2+1;2j-1}$ = B_OFFDIAG($treeL$, $treeRL$)
    $B_{k_1;i,k_2+1;2j}$ = B_OFFDIAG($treeL$, $treeRR$)
    $B_{k_1;i,k_2;j} = B_{k_1;i,k_2+1;2j-1} W_{k_2+1;2j-1} + B_{k_1;i,k_2+1;2j} W_{k_2+1;2j}$
    $B_{k_1;i,k_2+1;2j-1} = ();\qquad B_{k_1;i,k_2+1;2j} = ()$
    B_DIAG($treeR$)
  else if $treeL$ is **not** a leaf node and $treeR$ is a leaf node **then**
    $-,\ treeLL,\ treeLR = treeL$
    $B_{k_1+1;2i-1,k_2,j}$ = B_OFFDIAG($treeLL$, $treeR$)
    $B_{k_1+1;2i,k_2,j}$ = B_OFFDIAG($treeLR$, $treeR$)
    $B_{k_1;i,k_2;j} = R^T_{k_1+1;2i-1} B_{k_1+1;2i-1,k_2;j} + R^T_{k_1+1;2i} B_{k_1+1;2i,k_2;j}$
    $B_{k_1+1;2i-1,k_2;j} = ();\qquad B_{k_1+1;2i,k_2;j} = ()$
    B_DIAG($treeL$)
  end if
end function

4.1.1. Definitions. Denote $V(G)$ as the set of nodes for graph G.

For a set S, $|S|$ is the cardinality of S.

A regular binary tree $T \in \mathcal{T}_{reg}$ has the **P1** property if for each node in T, the depth of the left subtree is greater than or equal to the depth of the right subtree. We denote the set of trees with the P1 property as $\mathcal{T}_{P1}$.

For a regular binary tree $T \in \mathcal{T}_{reg}$, a **root-leaf path** is a sequence of nodes $(u_0, ..., u_k)$ such that u_0 is the root of T, each subsequent node is a child of the preceding node, and u_k is a leaf node. The set of root-leaf paths for T is denoted $p_{rl}(T)$.

For each $T \in \mathcal{T}_{(P1)}$ and root-leaf path $p \in p_{rl}(T)$, the **memory block cardinality** $b_m : p_{rl}(T) \to \mathbb{N}$ maps p to a natural number using the following rule: $b_m(p)$ is equal to the number nodes in p whose right children are also in p, plus one.

For a tree $T \in \mathcal{T}_{P1}$, the **worst-case memory block cardinality**, denoted $b_{wc}(T)$, is defined as $\max_{p \in p_{rl}(T)} b_m(p)$.

For each $T \in \mathcal{T}_{P1}$, the **primary right branch** is the subgraph of T consisting of the root node and subsequent right children. All primary right branches are line graphs. The path along the primary right branch is denoted as $p_{pri}(T)$.

A regular binary tree $T \in \mathcal{T}_{P1}$ has the property **P2** if $b_{wc}(T) = b_m(p_{pri}(T))$. We denote the set of trees with the P2 property as $\mathcal{T}_{P2}$.

4.1.2. Results and Proofs. Our strategy is to determine the fewest number of nodes that a P1 tree can have while generating a fixed number of memory blocks. This results in the following proposition:

PROPOSITION 1. *For all* $d \in \mathbb{N}$,

$$\min_{T \in \mathcal{T}_{P1}, b_{wc}(T) = d+1} |V(T)| = d^2 + d + 1. \tag{4.1}$$

Algorithm 3 Pass 2BU - Computation of Expansion Coefficients ($B_{k;i-1,i}$) Corresponding to Off-Diagonal Blocks

function B_OffDiag(treeL,treeR)
  $-,\ treeLL,\ treeLR = treeL$
  $-,\ treeRL,\ treeRR = treeR$
  if $treeL$ is a leaf node and $treeR$ is a leaf node **then**
    $B_{k_1;i,k_2;j} = U^T_{k_1;i} A_{k_1;i,k_2;j} V_{k_2;j}$
    return $B_{k_1;i,k_2;j}$
  else if $treeL$ is **not** a leaf node and $treeR$ is **not** a leaf node **then**
    $B_{k_1+1;2i-1,k_2+1;2j-1}$ = B_OffDiag($treeLL$,$treeRL$)
    $B_{k_1+1;2i-1,k_2+1;2j-}$ = B_OffDiag($treeLL$,$treeRR$)
    $B_{k_1+1;2i,k_2+1;2j-1}$ = B_OffDiag($treeLR$,$treeRL$)
    $B_{k_1+1;2i,k_2+1;2j}$ = B_OffDiag($treeLR$,$treeRR$)
    $B_{k_1;i,k_2;j} = R^T_{k_1+1;2i-1} B_{k_1+1;2i-1,k_2+1;2j-1} W_{k_2+1;2j-1} + R^T_{k_1+1;2i-1} B_{k_1+1;2i-1,k_2+1;2j} W_{k_2+1;2j}$
      $+ R^T_{k_1+1;2i} B_{k_1+1;2i,k_2+1;2j-1} W_{k_2+1;2j-1} + R^T_{k_1+1;2i} B_{k_1+1;2i,k_2+1;2j} W_{k_2+1;2j}$
    $B_{k_1+1;2i-1,k_2+1;2j-1} = ();$ $B_{k_1+1;2i-1,k_2+1;2j} = ();$
    $B_{k_1+1;2i,k_2+1;2j-1} = ();$ $B_{k_1+1;2i,k_2+1;2j} = ()$
    return $B_{k_1;i,k_2;j}$
  else if $treeL$ is a leaf node and $treeR$ is **not** a leaf node **then**
    $B_{k_1;i,k_2+1;2j-1}$ = B_OffDiag($treeL$,$treeRL$)
    $B_{k_1;i,k_2+1;2j}$ = B_OffDiag($treeL$,$treeRR$)
    $B_{k_1;i,k_2;j} = B_{k_1;i,k_2+1;2j-1} W_{k_2+1;2j-1} + B_{k_1;i,k_2+1;2j} W_{k_2+1;2j}$
    $B_{k_1;i,k_2+1;2j-1} = ();$ $B_{k_1;i,k_2+1;2j} = ()$
    return $B_{k_1;i,k_2;j}$
  else if $treeL$ is **not** a leaf node and $treeR$ is a leaf node **then**
    $B_{k_1+1;2i-1,k_2;j}$ = B_OffDiag($treeLL$,$treeR$)
    $B_{k_1+1;2i,k_2;j}$ = B_OffDiag($treeLR$,$treeR$)
    $B_{k_1;i,k_2;j} = R^T_{k_1+1;2i-1} B_{k_1+1;2i-1,k_2;j} + R^T_{k_1+1;2i} B_{k_1+1;2i,k_2;j}$
    $B_{k_1+1;2i-1,k_2;j} = ();$ $B_{k_1+1;2i,k_2;j} = ()$
    return $B_{k_1;i,k_2;j}$
  end if
end function

To show this, we use the following lemma, which tells us that the class of trees that we need to search in consists of trees with the P2 property:

Lemma 1. *For all $T_1 \in \mathcal{T}_{P1} \setminus \mathcal{T}_{P2}$, there exists $T_2 \in \mathcal{T}_{P1}$ such that $b_{wc}(T_1) = b_{wc}(T_2)$ and $|V(T_2)| < |V(T_1)|$.*

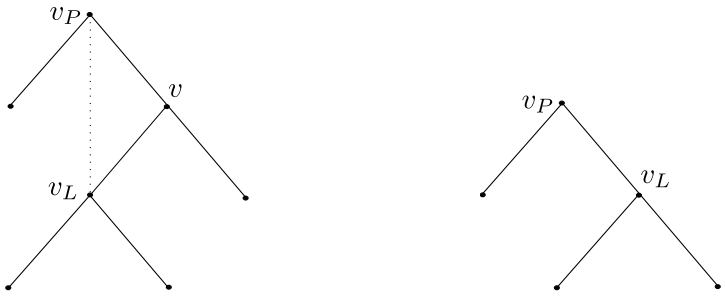
Proof. Let $T_1 \in \mathcal{T}_{P1} \setminus \mathcal{T}_{P2}$. Then there exists $p \in \mathcal{p}_{rl}(T_1)$ such that $b_m(p) > b_m(p_{pri}(T_1))$. Let $v \in p$ be the earliest node in p such that the subsequent node is a left child. We create a new tree T_2 by removing v and its right subtree from T_1, and creating an edge between the parent of v and the left child of v (if v is root, then we skip the latter). Now the path $\hat{p} := p \setminus \{v\}$ is a root-leaf path for T_2 and satisfies $b_m(\hat{p}) = b_m(p)$. Figures 4.1a and 4.1b give examples of such trees T_1 and T_2, respectively.
□

We are now ready to prove Proposition 1.

Proof. We begin with the claim that for all $d \in \mathbb{N}$,

$$\min_{T \in T_{(P1)}, b_{wc}(T) = d+1} |V(T)| \geq d^2 + d + 1. \tag{4.2}$$

Suppose otherwise. Then there exists $d \in \mathbb{N}$ and $T \in \underset{T \in \mathcal{T}_{P1}, b_{wc}(T) = d+1}{\arg\min} |V(T)|$ such that $|V(T)| < d^2 + d + 1$. By Lemma 1, $T \in \mathcal{T}_{P2}$, therefore $b_{wc}(T) = b_m(p_{pri}(T)) = d + 1$. Then, using the P1 property and the definition of b_m, we have that the primary branch of T has depth d.



(a) T_1 decorated with dotted line denoting the newly created root-leaf path which gives rise to T_2

(b) T_2

Fig. 4.1: Example T_1 and T_2 for Lemma 1

Due to the P1 property, the left subtree of the root node must have have a depth of at least d. Such a subtree must contain at least $2d-1$ nodes. Applying the same logic to the right child of the root node, the next left subtree must contain at least $2(d-1)-1$ nodes. We repeat this for the ith right offspring of the root node, obtaining left subtrees consisting of at least $2(d-i)-1$ nodes for $i \in \{0,...,d-1\}$. Including the primary right branch, which consists of $d+1$ nodes, we have

$$\begin{aligned} |V(T)| &\geq d+1+\sum_{k=1}^{d}(2(d-k)+1) \\ &= 2d^2+2d+1-2\sum_{k=1}^{d}k \\ &= d^2+d+1, \end{aligned}$$

which contradicts $|V(T)| < d^2+d+1$. Thus (4.2) is satisfied for all $d \in \mathbb{N}$. Furthermore, by choosing the fewest number of nodes at each step of the construction of T, we obtain a tree that satisfies $T \in \mathcal{T}_{P1}$, $b_{wc}(T) = d+1$, and $|V(T)| = d^2+d+1$. Thus we have the result. □

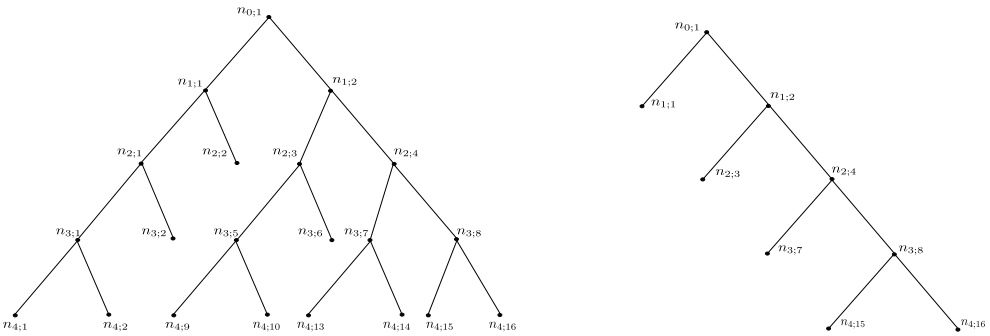
Note that for any binary tree, the number of non-leaf nodes, N_N, will be one less than the number of leaf nodes, N_L, i.e. $N_N = N_L - 1$. Then, since $N_L = n/p$, we have

$$N = \frac{2n}{p} - 1. \tag{4.3}$$

Proposition 1 implies that the worst case number of memory blocks is $O\left(\sqrt{N}\right)$. Since a single memory block consumes $O(pn)$ memory, using (4.3) we have that the worst case memory usage is $O\left(p^{0.5}n^{1.5}\right)$. An example of a tree that generates the worst case number of memory blocks is given in Figure 4.2a. However, for a complete partition tree, Phase 1 will take at most $O(pn\log n)$ memory. Further, for a regular binary tree of maximum depth (see for example Figure 4.2b) memory consumption is only $O(np)$.

4.2. Memory Consumption - Phase 2. In Phase 2 of our algorithm, at most 3 $p \times p$ blocks are stored in memory during each recursive call, and this number can easily be cut down to one $p \times p$ block, though we present the algorithm as is for clarity. Notice that each of the $p \times p$ blocks which are returned on lines 8 through 11 are cleared from memory on lines 13 and 14. This equivalent statement is true for each of the if statement cases in Algorithms 2 and 3. This can clearly be seen by looking at lines 20 and 26 in Algorithm 3, as well as line 15 and 22 in Algorithm 2. Therefore, given a tree of maximal depth, $\frac{n}{p}$ (Figure 4.2b), peak memory consumption for Phase 2 is np.

4.3. Total Memory Consumption. If a deepest first algorithm is not used the memory complexity can become $O(n^2)$, whereas the worst case deepest first search path memory complexity is $O\left(p^{0.5}n^{1.5}\right)$. For a complete partition tree however, the memory complexity is $O(pn\log n)$. As shown in [2], basis matrices and translation operators can be generated for smooth matrices using Chebyshev polynomials. If the basis matrices are computed a priori, it is then only necessary to execute Algorithms 2 and 3 to compute the



(a) Example Worst Case HSS Tree for Memory Consumption

(b) Maximal Depth Partition Tree

Fig. 4.2: Example Partition Trees

expansion coefficients at a cost of at most $O(np)$. In the case of the complete partition tree this cost is reduced to $O(p^2 \log n)$.

5. Flop Count. Let us compute the total number of flops for our algorithm. To begin, we see that for any binary tree with a fixed number of nodes, the number of non-leaf nodes, N_N, will be one less than the number of leaf nodes, N_L. In other words, $N_N = N_L - 1$. We also have that $N_L = n/p$. For Phase 1, we can see that at a leaf node, we generate 2 hankel blocks, and take 2 SVD's at a cost of approximately $2n^2p + 4n^2$. For a non-leaf node in Phase 1, we take 2 SVD's and perform 2 matrix multiplies at a cost of $(2np^2 + 2np)$. So the total flop count for Phase 1 will be $4n^2p + 6n^2$.

For Phase 2, in which we compute all $B_{k;i,j}$, the number of flops will be maximum for an even tree. The computation of each $B_{k;i,j}$ in the HSS tree requires the multiplication of matrices of size $p \times p$ (Algorithm 3 line 5 and 12), and costs at most $8p^3$. The computation of $B_{1;1,2}$ ($B_{1;2,1}$) begins at the bottom level of the recursion, Algorithm 3 line 5. This line will be executed on $(\frac{n}{2p} \times \frac{n}{2p})$ blocks, at a cost of $2p^3$ flops each. The calculation of $B_{1;1,2}$ ($B_{1;2,1}$) continues at line 12 of Algorithm 3, which will be executed on $\frac{1}{4}(\frac{n}{2p} \times \frac{n}{2p})$ blocks, at a cost of $8p^3$ each. The next stage of calculation for $B_{1;1,2}$ ($B_{1;2,1}$), again on line 12 of Algorithm 2, will be executed on $\frac{1}{16}(\frac{n}{2p} \times \frac{n}{2p})$ blocks, and so on. Following this recursion, the calculation of $B_{1;1,2}$ (or $B_{1;2,1}$) has a cost of n^2p flops. At level 1 in the HSS tree, only $B_{1;1,2}$, and $B_{1;2,1}$ must be computed, and so the flop count for both is $2n^2p$. In a similar fashion, we compute the number of flops required to compute $B_{k;i,j}$ at level 2 in our HSS tree. At this level there are 4 $B_{2;i,j}$, at a cost of n^2p flops. At level 3, we will compute 8 $B_{3;i,j}$ at a cost of $\frac{1}{2}n^2p$. Summing all of these computations the cost of all $B_{k;i,j}$, will be at most $4n^2p$ flops.

Thus for Phase 1 and Phase 2 combined, our algorithm has a complexity of $O(n^2p)$.

6. Numerical Experiments. In this section we will describe some experiments which demonstrate the speed and peak memory consumption of our algorithm. Experiments were carried out on a machine with a 2.5GHz Intel Core i5-3210M processor running Ubuntu with 8GB of RAM.

For both the CPU run-time and the peak memory consumption simulations, the $n \times n$ matrix A was chosen according to the formula $A_{i,j} = \sqrt{|x_i^n - x_j^n|}$, where $x_i^n = i/n$, for $i = 0, 1, \ldots, n$. The rank, p, was fixed across all tests. In the case of the HSS tree which generates the worst case peak memory for a given number of nodes, the structure of the tree was determined by calculating the number of nodes, N, in the HSS tree for a given matrix with dimension, n, via (4.3). We construct the tree for the given dimension, n, ranging from 256 to 1048576, according to the proof of Proposition 1. In table 6.1a each leaf node has a partition size of no more than $2(20p) - 1$, and no less than $20p$. In table 6.1b each leaf node has a partition size of no more than $2(3p) - 1$, and no less than $3p$. The measurement of peak memory given in table 6.1b was obtained from a counter in the code which simply kept track of memory allocations and de-allocations. The numbers listed are the maximum number of floating point allocations in memory during run-time. Note that we only keep track of floating point array allocations, not lower order integer allocations. CPU run-times and peak memory measurements are reported in Table 6.1a and 6.1b.

Table 6.1: CPU run-times in seconds and peak memory consumption in units of floating point allocations for the worst case memory HSS tree with $p = 3$, where p is the rank of the off diagonal blocks.

(a) CPU run-times with minimum partition at leaf nodes being $20p$

n	time(s)
256	0.02
512	0.06
1024	0.25
2048	0.94
4096	3.78
8192	16.51
16384	74.94
32768	347.41
65536	1615.33
131072	8357.98
262144	44011.02
524288	249070.53
1048576	1535229.75

(b) Measurement of peak memory consumption with minimum partition at leaf nodes being $3p$

n	Floating Point Allocations
256	11328
512	26520
1024	51414
2048	95082
4096	163572
8192	329136
16384	723336
32768	1708836
65536	3281298
131072	6035982
262144	10485492
524288	20993676
1048576	46158072

7. Conclusion. In this paper we have focused on the case where the partition tree corresponding to the rows of a matrix is the same as the tree which corresponds to its columns, though we can generalize the algorithm presented in this paper to more complicated partition trees, including FMM representations.

REFERENCES

[1] G. Agnarsson and R. Greenlaw. *Graph theory: Modeling, applications, and algorithms*. Prentice-Hall, Inc., 2006.

[2] S. Chandrasekaran, M. Gu, and T. Pals. A fast ulv decomposition solver for hierarchically semiseparable representations. *SIAM Journal on Matrix Analysis and Applications*, 28(3):603–622, 2006.

[3] S. Chandrasekaran and I. Ipsen. On rank-revealing factorisations. *SIAM Journal on Matrix Analysis and Applications*, 15(2):592–622, 1994.

[4] S. Chandrasekaran and K. Lessel. Scientific computing website. `http://scg.ece.ucsb.edu/software.html`.

[5] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *Journal of computational physics*, 73(2):325–348, 1987.

[6] M. Gu and S. Eisenstat. Efficient algorithms for computing a strong rank-revealing qr factorization. *SIAM Journal on Scientific Computing*, 17(4):848–869, 1996.

[7] W. Hackbusch and S. Börm. Data-sparse approximation by adaptive 2-matrices. *Computing*, 69(1):1–35, 2002.

[8] Y.P. Hong and C.-T. Pan. Rank-revealing qr factorizations and the singular value decomposition. *Mathematics of Computation*, 58(197):213–232, 1992.

[9] E. Liberty, F. Woolfe, Per-Gunnar Martinsson, Vladimir Rokhlin, and Mark Tygert. Randomized algorithms for the low-rank approximation of matrices. *Proceedings of the National Academy of Sciences*, 104(51):20167–20172, 2007.

[10] P.G. Martinsson. A fast randomized algorithm for computing a hierarchically semiseparable representation of a matrix. *SIAM Journal on Matrix Analysis and Applications*, 32(4):1251–1274, 2011.

[11] L. Miranian and M. Gu. Strong rank revealing lu factorizations. *Linear algebra and its applications*, 367:1–16, 2003.

[12] C.-T. Pan. On the existence and computation of rank-revealing lu factorizations. *Linear Algebra and its Applications*, 316(1):199–222, 2000.

[13] F. Rouet, X. Li, P. Ghysels, and A. Napov. A distributed-memory package for dense hierarchically semi-separable matrix computations using randomization. *arXiv preprint arXiv:1503.05464*, 2015.

[14] J Xia. Efficient structured multifrontal factorization for general large sparse matrices. *SIAM Journal on Scientific Computing*, 35(2):A832–A860, 2013.

[15] J. Xia. Randomized sparse direct solvers. *SIAM Journal on Matrix Analysis and Applications*, 34(1):197–227, 2013.

[16] J. Xia, S. Chandrasekaran, M. Gu, and X. Li. Fast algorithms for hierarchically semiseparable matrices. *Numerical Linear Algebra with Applications*, 17(6):953–976, 2010.

[17] J. Xia, Y. Xi, and M. Gu. A superfast structured solver for toeplitz linear systems via randomized sampling. *SIAM Journal on Matrix Analysis and Applications*, 33(3):837–858, 2012.