ON THE COMPLEXITY OF SOME HIERARCHICAL STRUCTURED MATRIX ALGORITHMS

JIANLIN XIA†

Abstract. In recent years, hierarchical structured matrices have been widely used in fast solutions of integral equations, PDEs, structured matrix (such as Toeplitz) problems, companion eigenproblems, etc. In this paper, we systematically study the complexity of some hierarchical structured matrix algorithms, in terms of hierarchically semiseparable (HSS) matrices. Several important aspects are considered. We perform detailed complexity analysis for some typical HSS algorithms, with the aid of certain graph techniques. This analysis helps us provide some significant improvements to classical HSS methods. One improvement is to propose more efficient HSS construction and solution algorithms. Another improvement is a recompression procedure which reorthonormalizes some HSS generators and converts noncompact HSS forms to compact ones. A precise theoretical justification of the compactness is also given. The third improvement is to relax the rank requirement in HSS operations. Unlike many classical HSS methods where the appropriate off-diagonal (numerical) ranks are often required to be bounded, we allow the ranks to increase. Certain general rank patterns are proposed, so that similar performance can be achieved with the maximum rank unbounded. These improvements significantly enhance both the efficiency and the applicability of HSS methods. Numerical examples from some applications are included to support the analysis.

Key words. rank structure, hierarchically semiseparable (HSS) matrix, HSS construction and ULV factorization, recompression, relaxation of rank requirement

AMS subject classifications. 65F05, 65F30

DOI. 10.1137/110827788

1. Introduction. In recent years, rank structured matrices have attracted much attention and are widely used in fast solutions of many numerical problems, including certain PDEs, integral equations, special structured (such as Toeplitz) matrices, companion eigenvalue problems, etc. See [1, 4, 6, 15, 16, 19, 24, 22, 36, 37, 40] for a partial list of references. It is observed that some dense intermediate matrices in these problems have a low-rank property, or their off-diagonal blocks have small ranks or numerical ranks. For example, in the direct solution of elliptic PDEs, the Schur complements in the factorization have this property [2, 11, 33, 35, 41]. Many rank structured matrix representations have been proposed to make use of this property. They provide data-sparse representations or approximations for dense matrices. An important class of rank structured representations are hierarchical structured matrices, such as \( \mathcal{H} \)-matrices [7, 26, 27, 29], \( \mathcal{H}^2 \)-matrices [5, 28, 30], and hierarchically semiseparable (HSS) matrices [9, 14, 42]. With these structures, a matrix is hierarchically partitioned into blocks at multiple levels. Hierarchical structured representations and operations enable one to reuse and share information across different levels so as to achieve high efficiency.

In this paper, we focus on HSS representations, which have a nice binary tree structure called the HSS tree. HSS forms have been used to develop highly efficient structured solvers and effective preconditioners for both dense and sparse linear sys-
tems [9, 14, 33, 39, 43]. For example, based on HSS methods and the multifrontal method [18, 32], a structured sparse direct solver is proposed in [41] for solving two-dimensional (2D) Poisson equations and 2D elasticity equations with nearly linear complexity and storage.

Classical HSS methods for dense matrices rely on an assumption that appropriate off-diagonal blocks (called HSS blocks) of the dense matrix have small bounded (numerical) ranks. Let $n$ be the order of the matrix and let $r$ be the maximum rank (called HSS rank) of all the HSS blocks. Then the matrix can be converted into an HSS matrix in $O(rn^2)$ flops or less, and a linear system with this HSS coefficient matrix can be solved in $O(r^2n)$ flops with the aid of ULV-type factorizations [9, 14, 42]. Many other fast HSS algorithms are also developed.

However, existing studies on HSS matrices generally do not include detailed complexity analysis. In fact, different HSS algorithms with the same order of complexity may have significantly different prefactors in the flop counts. This indicates the possibility of improving standard HSS algorithms. In particular, HSS construction algorithms as in [42] involve HSS blocks across different levels which makes the flop count nontrivial. Here, we provide a graph technique where an HSS tree is converted into a Pascal triangle so as to count the number of HSS blocks across different levels during the HSS construction. Then, systematic flop counts for a sequence of HSS algorithms can be conveniently conducted. We can also make some useful observations. For example, the HSS construction scheme in [42] for a general dense matrix with certain block sizes costs $6rn^2$ flops, and then it needs $42r^2n$ to compute a ULV factorization and $37rn$ to solve the ULV system. In contrast, while a triangular HSS factorization of the dense matrix costs $11rn^2$ flops, it needs only $10rn$ to solve a triangular HSS system. Such a comparison provides a way to choose different algorithms in different circumstances. These counts can also help study the performance of generalizations of HSS methods such as the sparse structured direct solver in [41].

We then provide several useful strategies for improving existing HSS methods and for developing new fast ones. Based on different internal operations and different ways of accessing matrix data, we can get more efficient HSS construction and factorization algorithms. For example, a modified ULV factorization scheme can improve the cost from $42r^2n$ (as in [14]) to $\frac{74}{3}r^2n$. We also show some HSS methods which specifically take advantage of symmetry, such as a ULV factorization scheme with $9r^2n$ cost.

As another improvement, a procedure for reorthonormalizing and recompressing noncompact HSS forms is proposed. It is useful in improving the efficiency and reliability of HSS forms arising from indirect construction methods such as those based on randomized sampling [31, 34] or strong rank-revealing LU factorizations [25], and those in sparse factorizations [41]. The recompression procedure also provides a precise theoretical justification of the compactness of HSS representations.

Furthermore, we enhance the flexibility and applicability of HSS methods by relaxing the requirement of bounded HSS ranks. In practical problems such as direct solutions of Helmholtz equations and Toeplitz linear systems, related off-diagonal ranks depend on the size ($n$) of appropriate dense matrices. In these two cases, the maximum HSS rank is actually $O(\log n)$ [20, 37] under certain conditions. In this paper, instead of requiring the HSS rank to be bounded by a constant, we allow the ranks of the HSS blocks to increase following certain function patterns of the block sizes so that the HSS ranks depend on $n$. We show that with these rank patterns, we can still get the same or similar order of complexity as before. This allows us to apply regular HSS methods to broader classes of applications. It is also possible to
generalize such rank relaxation techniques to other structured methods.

The remaining sections are organized as follows. In section 2, we briefly review HSS structures. Section 3 presents systematic flop counts for some HSS algorithms with the aid of some graph techniques. Section 4 shows some useful improvements to existing HSS algorithms and demonstrates the performance with numerical examples. An HSS recompression procedure is given in section 5. Techniques for relaxing classical rank requirements in HSS methods are presented in section 6. We draw some concluding remarks in section 7.

2. Review of HSS structures. We first briefly review some concepts of HSS structures. The following notation is used in this paper:

1. Assume $\mathcal{T}$ is a full binary tree in its postordering with $2k - 1$ nodes labeled as $i = 1, 2, \ldots, 2k - 1$. That is, each node $i$ of $\mathcal{T}$ is either a leaf, or is a nonleaf node with the left and right children $c_1$ and $c_2$, respectively, which are postordered so that $c_1 < c_2 < i$. Use $\text{sb}(i)$ and $\text{par}(i)$ to denote the sibling and the parent of a node $i$, respectively. The root node of $\mathcal{T}$ is $2k - 1$, which is written as $\text{root}(\mathcal{T})$.

2. Let $A$ be an $n \times n$ real matrix and let $\mathcal{I} = \{1, 2, \ldots, n\}$ be the set of all row/column indices. Assume $t_i \subset \mathcal{I}$ is a subset of $\mathcal{I}$ with contiguous indices. We let $\mathcal{I} = t_i^\uparrow \cup t_i \cup t_i^\wedge$, where all indices in $t_i^\uparrow$ are smaller than those in $t_i$, and all indices in $t_i^\wedge$ are larger than those in $t_i$.

3. $A|_{t_i \times t_j}$ is the submatrix of $A$ with row index set $t_i$ and column index set $t_j$.

4. By the compression of a block $A|_{t_i \times t_j}$, we mean a rank-revealing QR (RRQR) factorization or a truncated SVD of $A|_{t_i \times t_j}$. For simplicity, the former is used mostly in this paper, and sometimes, we write $A|_{t_i \times t_j} \approx U_i A|_{\hat{t}_i \times \hat{t}_j}$, which means that the second factor is still stored in $A|_{t_i \times t_j}$ with row index set $\hat{t}_i$.

5. $\text{diag}(\ldots)$ represents a block diagonal matrix formed by the blocks in $(\ldots)$. An HSS representation is generally defined recursively [9, 14, 42]. Here, we use a postordered HSS form defined in [42].

**Definition 2.1.** Assume $A$ is an $n \times n$ matrix and $\mathcal{I} = \{1, 2, \ldots, n\}$. Let $\mathcal{T}$ be a full binary tree with $2k - 1$ nodes labeled as $i = 1, 2, \ldots, 2k - 1$, and let $t_i \subset \mathcal{T}$ be an index set associated with each node $i$ of $\mathcal{T}$. We say $\mathcal{T}$ is a postordered HSS tree and $A$ is in an HSS form if the following conditions hold:

1. $\mathcal{T}$ is a postordered full binary tree (with the root $2k - 1$ at level 0).

2. There is one index set $t_i$ associated with each node $i$ of $\mathcal{T}$ which is defined hierarchically. For each nonleaf node $i$, $t_{c_1} \cup t_{c_2} = t_i$, $t_{c_1} \cap t_{c_2} = \emptyset$, and $t_{2k-1} = \mathcal{I}$.

3. There exist matrices $D_i, U_i, V_i, R_i, W_i, B_i$ (called HSS generators) associated with each node $i$ satisfying the following relations for $i = 1, 2, \ldots, 2k - 1$:

   $$(2.1) \quad D_i = A|_{t_i \times t_i} = \begin{pmatrix} D_{c_1} & U_{c_1} B_{c_1} V_{c_1}^T \\ U_{c_2} B_{c_2} V_{c_2}^T & D_{c_2} \end{pmatrix},$$

   $$(2.2) \quad U_i = \begin{pmatrix} U_{c_1} \\ U_{c_2} \end{pmatrix}, \quad R_i = \begin{pmatrix} R_{c_1} \\ R_{c_2} \end{pmatrix}, \quad V_i = \begin{pmatrix} V_{c_1} \\ V_{c_2} \end{pmatrix}, \quad W_i = \begin{pmatrix} W_{c_1} \\ W_{c_2} \end{pmatrix}.$$  

   Here, $D_{2k-1} \equiv A$, and $U_{2k-1}, V_{2k-1}, R_{2k-1}, W_{2k-1}, B_{2k-1}$ are empty matrices.

For a nonleaf node $i$, the generators $D_i, U_i, V_i$ are recursively defined and are not explicitly stored. The HSS form conveniently represents the rank structure of certain off-diagonal blocks as defined next.
**Definition 2.2.** Let

\begin{equation}
A_i^{-} = \begin{pmatrix} A|_{t_i \times t_i} & A|_{t_i \times t_i^c} \\ A|_{t_i^c \times t_i} & A|_{t_i^c \times t_i^c} \end{pmatrix}, \quad A_i^{\dagger} = \begin{pmatrix} A|_{t_i^c \times t_i} \\ A|_{t_i^c \times t_i} \end{pmatrix}.
\end{equation}

The blocks \(A_i^{-}\) and \(A_i^{\dagger}\) are called the HSS block row and column associated with node \(i\), respectively. (See Figure 2.1.) The maximum (numerical) rank of all HSS blocks is called the HSS rank of \(A\).

(i) One level of HSS blocks. (ii) Two levels of HSS blocks.

Fig. 2.1. HSS (off-diagonal) block rows \(A_i^{-}\), \(i = 1, 2, \ldots, 6\).

Clearly, the columns of \(U_i\) form a basis for the column space of \(A_i^{-}\), and the rows of \(V_i^T\) form a basis for the row space of \(A_i^{\dagger}\). Thus, \(U_i\) and \(V_i\) are called basis matrices in this paper, or cluster bases in [5]. They are often made to have orthonormal columns for good stability.

The following is a block \(4 \times 4\) HSS matrix example:

\begin{equation}
A = \begin{pmatrix}
D_1 & U_1B_1V_2^T & U_1R_1B_3W_4^TV_4^T & U_1R_1B_3W_6^TV_6^T \\
U_2B_2V_1^T & D_2 & U_2R_2B_3W_4^TV_4^T & U_2R_2B_3W_5^TV_5^T \\
U_4R_4B_6W_7^TV_7^T & U_4R_4B_6W_7^TV_7^T & D_4 & U_4B_4V_5^T \\
U_5R_5B_6W_7^TV_7^T & U_5R_5B_6W_7^TV_7^T & U_5B_5V_4^T & D_5
\end{pmatrix}.
\end{equation}

See Figure 2.2. If \(A\) is symmetric, we can set [42]

\begin{equation}
D_i = D_i^T, \quad V_i = U_i, \quad B_{\text{sub}(i)} = B_i^T.
\end{equation}

Fig. 2.2. HSS matrix pattern for (2.4) and the corresponding HSS tree.
3. Complexity of hierarchical construction and solution schemes. In
hierarchical compression schemes for computing structured matrix representations
and their factorizations, it often involves simultaneous compression of blocks across
different levels [14, 28, 42, 43]. Here, in the context of HSS representations, we present
an innovative way of counting the detailed complexity of such compression with the
aid of some graph results. These counts are hardly done in [14, 42, 43] and most other
HSS literature. For convenience, the flop counts of some basic matrix operations are
listed in Table 3.1. They can be found, say, in [17, 23].

**Table 3.1**
Flop counts of some basic matrix operations, with low order terms dropped. The RRQR fac-
torization we use is based on the modified Gram-Schmidt algorithm with column pivoting as in
[23].

<table>
<thead>
<tr>
<th>Operation</th>
<th>Flops</th>
</tr>
</thead>
<tbody>
<tr>
<td>QR factorization of an $m \times q$ tall matrix ($m &gt; q$)</td>
<td>$2q^2 (m-\frac{q}{2})$</td>
</tr>
<tr>
<td>Product of the $Q$ factor and an $m \times r$ matrix</td>
<td>$2rq(2m-q)$</td>
</tr>
<tr>
<td>Product of an $m \times q$ matrix and a $q \times r$ matrix</td>
<td>$2mqr$</td>
</tr>
<tr>
<td>LU factorization of an $m \times m$ matrix</td>
<td>$\frac{4}{3}m^3$</td>
</tr>
<tr>
<td>Solution of an order $m$ triangular system $Lx = b$</td>
<td>$m^2$</td>
</tr>
<tr>
<td>RRQR factorization of an $m \times q$ matrix with rank $r$</td>
<td>$4mqr - 2r^2(m+q) + \frac{4}{7}r^3$</td>
</tr>
</tbody>
</table>

3.1. An analytical result for binary trees. In hierarchical compression
schemes, later compression steps are usually designed to take advantage of compressed
forms from previous steps so as to improve the efficiency. Here, we briefly rephrase the
major steps of the HSS construction algorithm in [42] with more systematic notation
in [43] and then show the detailed complexity analysis. This is done with the aid of
the following definition given in [43].

**Definition 3.1.** The visited set $\mathcal{V}_i$ associated with a node $i$ of a postordered
binary tree $T$ is

$$\mathcal{V}_i = \{ j \mid j \text{ is a left node and } \text{sib}(j) \in \text{pred}(i) \},$$

where $\text{pred}(i)$ is the set of predecessors associated with node $i$ given by

$$\text{pred}(i) = \begin{cases} \{ i \} & \text{if } i = \text{root}(T), \\ \{ i \} \cup \text{pred(par}(i)) & \text{otherwise}. \end{cases}$$

See Figure 3.3(b) below for an example. Clearly, $\mathcal{V}_i$ is the set of visited nodes
(before $i$) whose siblings have not been visited. As pointed out in [43], the set $\mathcal{V}_i$ essentially corresponds to a stack which is often used in the postordering traversal of
binary trees such as elimination trees in the multifrontal method [18, 32] and HSS
trees in HSS methods. This stack is obtained as follows. For $i = 1, 2, \ldots, i$ is a left
node, push $i$ onto the stack. Otherwise, pop a node from the top of the stack. Then
before the visit of $i$, the nodes in the stack form $\mathcal{V}_i$. For convenience, we denote the
elements of $\mathcal{V}_i$ by

$$(3.1) \quad \mathcal{V}_i = \{ j_1, j_2, \ldots, j_s \mid j_1 < j_2 < \cdots < j_s \},$$

where $s_i$ is the cardinality of $\mathcal{V}_i$ (later, we sometimes write $s_i$ as $s$ to simplify the
subscripts when no confusion is caused).

The use of $\mathcal{V}_i$ helps clearly describe the HSS construction algorithm, and $s_i$ will be
involved in the flop count. According to the definition, each $j \in \mathcal{V}_i$ corresponds
to a right node \( \text{sib}(j) \in \text{pred}(i) \) and also a right edge in \( P_{\text{root}(T)\rightarrow i} \), the path from root\( (T) \) to \( i \). (A right edge is an edge connecting a nonleaf node with its right child.) Then it is not hard to verify the following lemma.

**Lemma 3.2.** For \( V_i \) in Definition 3.1 with cardinality \( s_i \) as in (3.1), we have the following:

1. \( \text{pred}(i) = \{ v \mid \text{sib}(v) \in V_i \} \cup \text{pred}(j_1) \), where \( j_1 \) is given in (3.1).
2. \( s_i = s_{c_1} = s_{c_2} - 1 \) for \( c_1 \) and \( c_2 \), the left and right children of \( i \), respectively.
3. \( s_i \) is equal to the number of right edges in \( P_{\text{root}(T)\rightarrow i} \).

Based on this lemma, we have the following result.

**Theorem 3.3.** Assume the HSS tree \( T \) is a perfect binary tree with the root at level 0. Let \( s_i \) be the cardinality of \( V_i \) as in (3.1). Then at tree level \( l \), a node \( i \) has possible \( s_i \) value \( j \in \{1, 2, \ldots, l\} \), and the number of nodes \( i \) with \( s_i \) being the same value \( s_i = j \) is \( f_j^l = \binom{l}{j} \) for \( j = 1, 2, \ldots, l \).

**Proof.** For a node \( i \) at level \( l \), there are at most \( l \) right edges in \( P_{\text{root}(T)\rightarrow i} \). Then the third statement in Lemma 3.2 indicates that the possible values of \( s_i \) are \( \{1, 2, \ldots, l\} \).

To find \( f_j^l \) for a fixed \( j = 1, 2, \ldots, l \), we rearrange \( T \) into a new graph \( G \) by merging certain nodes. That is, at each level \( l \), all the nodes with the same \( s_i \) value \( j \) (Figure 3.1) are merged into one single node, which is the \( j \)th node (from the left) at level \( l \) of \( G \). According to Lemma 3.2, this can be performed by merging all nodes \( i \) with the same number of right edges in \( P_{\text{root}(T)\rightarrow i} \). Then there are \( l \) nodes at level \( l \) of \( G \). Note that all paths in \( G \) from its root (the top node) to the \( j \)th node at level \( l \) have the same number of right paths. Thus, \( P_{\text{root}(T)\rightarrow i} \) in \( T \) is transformed into a path in \( G \). See Figure 3.2(a).

![Fig. 3.1. s_i values for the nodes i of an HSS tree, and the nodes to be merged.](image)

Therefore, \( f_j^l \) is simply the number of top-down paths from the root to the \( j \)th node at level \( l \) of the new graph. It is known that this number is the binomial coefficient \( \binom{l}{j} \), with the aid of the corresponding Pascal triangle as illustrated in Figure 3.2(b).

**3.2. Complexity of HSS construction.** Here, we show systematic complexity analysis for HSS constructions using the results in the previous subsection. We reformulate the major steps of the algorithm in [42] with clearer notation. The algorithm traverses the HSS tree \( T \) following the postordering and hierarchically compresses the HSS blocks \( A^-_t \) and \( A^+_t \) in (2.3), where \( A|_{t \times t'} \) and \( A|_{t' \times t} \) have been partially compressed in previous steps. By ignoring appropriate \( U, V \) basis matrices, we need only to compress certain blocks \( \Psi_i \) and \( \Phi_i \) as in Figure 3.3 instead of \( A|_{t \times t'} \) and \( A|_{t' \times t} \), respectively. This is explained as follows. For simplicity, we mainly consider \( A^-_t \). We assume \( r \) is the maximum numerical rank of all the HSS blocks of \( A \) for a
given relative tolerance $\tau$. (Note that, for different $\tau$, the numerical rank $r$ may vary. Sometimes, we may also specify a fixed $r$ when we compress an HSS block or truncate off-diagonal singular values. In this paper, for convenience, we use a single symbol $r$.)

For node $i = 1$, compute the following compression or RRQR factorization:

$$A_i^- = A|_{t_i \times t'_i} \approx U_1 A^{(1)}|_{t_i \times t'_i}, \quad A_i^+ = A^T|_{t_i \times t_i} \approx V_i A^{(1)}|_{t_i \times t_i}^T.$$

For node $i > 1$, the compression is done hierarchically. If $i$ is a leaf, by recursion, $A|_{t_i \times t'_i}$ is in a compressed form $A|_{t_i \times t'_i} \approx \Psi_i \tilde{V}_i^T$, where $\tilde{V}_i$ is given by existing $V$ basis matrices, and $\Psi_i$ is the contribution from the previous $i - 1$ compression steps, associated with nodes $j_1, j_2, \ldots, j_s$ in (3.1) due to the hierarchy (Figure 3.3):

$$\Psi_i = \left( A^{(i-1)}|_{t_i \times t_{j_1}} \cdots A^{(i-1)}|_{t_i \times t_{j_s}} \right).$$

Thus, the compression of the HSS block row $A_i^- \approx (\Psi_i \tilde{V}_i^T \ A|_{t_i \times t'_i})$ can be effectively done on $(\Psi_i \ A|_{t_i \times t'_i})$, where we ignore a matrix diag$(\tilde{V}_i, I)$ which has orthonormal columns. Compute an RRQR factorization

$$\left( \Psi_i \ A|_{t_i \times t'_i} \right) \approx U_i \left( A^{(i)}|_{t_i \times t_{j_1}} \cdots A^{(i)}|_{t_i \times t_{j_s}} \right) A|_{t_i \times t'_i}.$$
where the column partition of \( A^{(i)}|_{i_1 \times i_1} \cdots A^{(i)}|_{i_1 \times i_s} \) follows that of \( \Psi_i \) in (3.2). The corresponding HSS block is similarly compressed to get \( V_i \).

If \( i \) is a nonleaf node, suppose \( c_1 \) and \( c_2 \) are the left and right children of \( i \), respectively. Let

\[
(3.4) \quad \Psi_i = \left( \begin{array}{c} \Psi_{c_1} \\ \Psi_{c_2} \end{array} \right) = \left( \begin{array}{ccc} A^{(i-1)}|_{i_1 \times i_1} & \cdots & A^{(i-1)}|_{i_1 \times i_s} \\ A^{(i-1)}|_{i_2 \times i_1} & \cdots & A^{(i-1)}|_{i_2 \times i_s} \end{array} \right).
\]

Compute the compression

\[
(3.5) \quad \left( \begin{array}{c} \Psi_{c_1} \\ \Psi_{c_2} \end{array} \right) A^{(i-1)}|_{i_1 \times t_j'} \Psi_{c_1} = R_{c_1} \left( \begin{array}{ccc} (A^{(i)}|_{i_1 \times i_1} & \cdots & A^{(i)}|_{i_1 \times i_s} \\ A^{(i)}|_{i_2 \times i_1} & \cdots & A^{(i)}|_{i_2 \times i_s} \end{array} \right) R_{c_2},
\]

where the column partition of \( A^{(i)}|_{i_1 \times i_1} \cdots A^{(i)}|_{i_1 \times i_s} \) follows that of \( \Psi_i \) in (3.4). Similarly, we compress the associated HSS block column to get \( (W_{c_1} \quad W_{c_2} \quad W_{c_1}) \). In addition, we set

\[
(3.6) \quad B_{c_1} = A^{(i-1)}|_{i_1 \times i_1}, \quad B_{c_2} = A^{(i-1)}|_{i_1 \times i_1}.
\]

Note \( c_2 = i - 1 \).

The algorithm is summarized in Algorithm 1.

**Algorithm 1 (HSS construction).**

For nodes (separators) \( i = 1, 2, \ldots, \text{root}(T) \),

if \( i \) is a leaf

- (Row compression) Form \( \Psi_i A|_{i_1 \times t_j'} \) with (3.2) and compute \( U_i \) as in (3.3)
- (Column compression) Compute \( V_i \) similarly

Else

- (Row compression) Form \( \Psi_{c_1} A^{(i-1)}|_{i_1 \times t_j'} \Psi_{c_1} \) with (3.4) and compute

\[
\left( \begin{array}{c} R_{c_1} \\ R_{c_2} \end{array} \right)\]

as in (3.5)
- (Column compression) Compute \( (W_{c_1} \quad W_{c_2} \quad W_{c_1}) \) similarly
- Form \( B_{c_1} \) and \( B_{c_2} \) as in (3.6) for the children \( c_1, c_2 \) of \( i \)

End

To simplify the complexity count, we assume that the HSS tree \( T \) is a perfect binary tree with \( 2k - 1 \) nodes (\( k \) leaves) and \( L \approx \log_2 k \) levels, the numerical rank of any HSS block is \( r \ll n \), and any bottom/leaf level HSS block row dimension is \( m = O(r) \). To avoid unnecessary compression for full rank blocks at the leaf level, we let \( m > r \). Also let \( n_i \) be the column size of \( A|_{i_1 \times t_j'} \).

We count the costs levelwise. Two useful formulas are needed:

\[
(3.7) \quad \sum_{i \text{ at level } l} s_i = \sum_{j=1}^{l} \binom{l}{j} \frac{1}{2} 2^j,
\]

\[
(3.8) \quad \sum_{i \text{ at level } l} n_i = \sum_{j=1}^{2^l} \left( k - j \frac{k}{2^l} \right) m = \frac{1}{2} km(2^l - 1).
\]
The formula (3.7) is a direct result of Theorem 3.3, and (3.8) is based on the fact that each level \( l \) has \( 2^l \) nodes with \( n_i \) values \( (k-j\frac{1}{2^j})m \), \( j = 1, 2, 3, \ldots, 2^l \).

First, we count the costs associated with all leaves. Clearly, \((\Psi_i \ A_{i \times r})\) in (3.3) has size \( m \times (rs_i + n_i) \). From Table 3.1, (3.3) costs \( 4m(s_ir + n_i)r - 2r^2(m + (s_ir + n_i)) + \frac{4}{3}r^3 \) flops for each leaf \( i \). The total for all leaves is then

\[
C_1 = \sum_{i: \text{leaf}} \left( 4m(s_ir + n_i)r - 2r^2(m + (s_ir + n_i)) + \frac{4}{3}r^3 \right)
= (4mr^2 - 2r^3) \sum_{i: \text{leaf}} s_i + (4mr - 2r^2) \sum_{i: \text{leaf}} n_i + \left( \frac{4}{3}r^3 - 2mr^2 \right) 2^L
\approx 2rk^2m^2 - r^2k^2m,
\]

where (3.7) and (3.8) are used together with \( k \approx 2^L \), and some low order terms are dropped since \( r, m \ll n \).

Next, consider nonleaf nodes. The left-hand side of (3.5) has size \( 2r \times (rs_i + n_i) \).

Thus, (3.5) costs \( 8r^2(rs_i + n_i) - 2r^2(2r + (rs_i + n_i)) + \frac{4}{3}r^3 \) for each nonleaf node \( i \). The total block-row compression cost for all nonleaf nodes can be similarly counted:

\[
C_2 = \sum_{l=1}^{L-1} \sum_{i: \text{at level } l} \left( 8r^2(rs_i + n_i) - 2r^2(2r + (rs_i + n_i)) + \frac{4}{3}r^3 \right) \approx 3r^2k^2m.
\]

The cost for all block-column compression or the computation of \( V, W \) generators is also \( C_1 + C_2 \). Thus, the total HSS construction cost is

\[
C_{\text{constr}} = 2(C_1 + C_2) \approx 2(2rk^2m^2 - r^2k^2m + 3r^2k^2m) = 4 \left( 1 + \frac{r}{m} \right) rn^2.
\]

For example, if \( m = 2r \), we have

\[
C_{\text{constr}} \approx 6rn^2.
\]

Note that the storage of this HSS matrix can be easily verified to be

\[
S_{\text{mem}} \approx (m^2 + 2mr + 2r^2)\frac{n}{m}.
\]

If \( m = 2r \), we have \( S_{\text{mem}} \approx 5rn \).

### 3.3. Complexity of HSS factorization and solution

For a given HSS form, we can quickly compute ULV-type factorizations and solutions [14, 42]. The fundamental idea of ULV HSS factorizations can be illustrated in terms of a block \( 2 \times 2 \) HSS form

\[
(3.9) \quad \begin{pmatrix} D_j & U_jB_jV_j^T \\ U_iB_iV_i^T & D_i \end{pmatrix} m \quad m,
\]

where \( j = \text{sib}(i) \), the generators \( U_i \) and \( V_i \) have sizes \( m \times r \), and \( B_i \) has size \( r \times r \).

Compute a QL factorization of \( U_i \) and update \( D_i \) as \( \tilde{D}_i \):

\[
(3.10) \quad U_i = Q_i \begin{pmatrix} 0 \\ \tilde{U}_i \end{pmatrix} \begin{pmatrix} m - r \\ r \end{pmatrix}, \quad \tilde{D}_i = Q_i^TV_i^TD_i.
\]
U_j is similarly factorized and D_j is updated. We have

\[(3.11) \begin{pmatrix} Q_j^T & Q_i^T \end{pmatrix} \begin{pmatrix} D_j & U_j B_i V_i^T \\ U_i B_i V_i^T & D_i \end{pmatrix} = \begin{pmatrix} \hat{D}_j & 0 \\ 0 & \hat{U}_j \end{pmatrix} \begin{pmatrix} 0 & B_j V_i^T \\ B_i V_i^T & \hat{D}_i \end{pmatrix}.
\]

Then compute a QR factorization of \(\hat{D}_i^T\) and update \(V_i\) as

\[(3.12) \quad \hat{D}_i^T = W_i \hat{D}_i^T, \quad \hat{V}_i = W_i^T V_i.
\]

\(\hat{D}_j^T\) is similarly factorized and \(V_j\) is updated. We have

\[(3.13) \quad \begin{pmatrix} Q_j^T & Q_i^T \end{pmatrix} \begin{pmatrix} D_j & U_j B_i V_i^T \\ U_i B_i V_i^T & D_i \end{pmatrix} \begin{pmatrix} W_j & W_i \end{pmatrix} = \begin{pmatrix} \tilde{D}_j;1,1 & 0 \\ \tilde{D}_j;2,1 & \tilde{D}_j;2,2 \end{pmatrix} \begin{pmatrix} 0 & \tilde{U}_j \end{pmatrix} \begin{pmatrix} B_j & B_j^T \tilde{V}_{i;1} \\ \tilde{V}_{i;2} \tilde{V}_{i;2}^T & \tilde{D}_{i;1,1} & 0 \\ \tilde{D}_{i;2,1} & \tilde{D}_{i;2,2} \end{pmatrix},
\]

where \(\tilde{D}_i, \tilde{V}_i, \tilde{D}_j,\) and \(\tilde{V}_j\) are partitioned conformably. Then the blocks \(\tilde{D}_j;1,1\) and \(\tilde{D}_{i;1,1}\) can be eliminated, and the rest of the blocks are merged into a new matrix. That is, nodes \(j\) and \(i\) are eliminated, and their parent node \(p\) becomes a leaf with generators

\[(3.14) \quad D_p = \begin{pmatrix} \tilde{D}_j;2,2 & \tilde{U}_j B_j \tilde{V}_{i;2} \\ \tilde{U}_i B_i \tilde{V}_{i;2} & \tilde{D}_{i;2,2} \end{pmatrix}, \quad U_p = \begin{pmatrix} \tilde{U}_j R_j \\ \tilde{U}_i R_i \end{pmatrix}, \quad V_p = \begin{pmatrix} \tilde{V}_{j;2} W_j \\ \tilde{V}_{i;2} W_i \end{pmatrix}.
\]

After this step, we have a smaller HSS form and the elimination continues recursively. This is summarized in Algorithm 2.

**Algorithm 2 (HSS ULV factorization).**

\[
\text{for nodes (separators) } i = 1, 2, \ldots, \text{root}(\mathcal{T}) - 1 \quad \text{do}
\]

\[
\text{if } i \text{ is a leaf}
\]

\[
- \text{(Introducing zeros into off-diagonal blocks) Compute a QL factorization of } U_i \text{ and update } D_i \text{ as in (3.10)}
\]

\[
- \text{(Introducing zeros into diagonal blocks) Compute a QR factorization of } \hat{D}_i^T \text{ and update } V_i \text{ as in (3.12)}
\]

\[
- \text{(Partial elimination) Eliminate } \tilde{D}_{i;1,1} \text{ in (3.13)}
\]

\[
\text{else}
\]

\[
- \text{(Merging) Form } D_i, U_i, V_i \text{ as in (3.14) (with } j, i, \text{ and } p \text{ in (3.14) replaced by } c_1, c_2, \text{ and } i, \text{ respectively, where } c_1, c_2 \text{ are the children of } i) \quad \text{end}
\]

\[
\text{end}
\]

To get the complexity of this ULV factorization, we list the individual costs of the operations associated with each node in Table 3.2, where Table 3.1 is used. Then it can be easily verified that the total ULV factorization cost is about

\[(3.15) \quad C_{\text{fact}} \approx \frac{4}{3} m^2 n + 6 mn + \frac{148}{3} kr^3.
\]
Table 3.2
Operations associated with each leaf of $T$ in the HSS factorization scheme described in this section, where some low order terms are dropped. For a nonleaf node, replace $m$ in the table by $2r$.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Flops</th>
<th>Times needed</th>
</tr>
</thead>
<tbody>
<tr>
<td>QR factorization of a size $m \times r$ matrix</td>
<td>$2r^2 (m - \frac{r}{2})$</td>
<td>$x1$</td>
</tr>
<tr>
<td>Product of this $Q$ factor and an $m \times m$ matrix</td>
<td>$2rm(2m - r)$</td>
<td>$x1$</td>
</tr>
<tr>
<td>QR factorization of a size $m \times m$ matrix</td>
<td>$\frac{5}{2}m^3$</td>
<td>$x1$</td>
</tr>
<tr>
<td>Product of this $Q$ factor and an $m \times r$ matrix</td>
<td>$2r m^2$</td>
<td>$x1$</td>
</tr>
<tr>
<td>Product of two $r \times r$ matrices in the merge process</td>
<td>$2r^3$</td>
<td>$x4$</td>
</tr>
</tbody>
</table>

When $m = 2r$, we have $C_{\text{fact}} \approx 42r^2n$, which is slightly tighter than the bound $46r^2n$ in [14]. In such a situation, it can be similarly shown that the ULV factors can be used to solve a system with one right-hand side in $C_{\text{sol}} = 37rn$ flops. (Later, the complexity of other algorithms is similarly obtained and the details are omitted.)

Another HSS solution method is to directly factorize a dense matrix into triangular HSS factors. The case of symmetric positive definite matrices is discussed in [43]. The case of general nonsymmetric matrices can be considered in a way similar to the HSS construction algorithm. The costs of the dense-to-triangular-HSS factorization and the triangular HSS solution by substitution are

$$C_{\text{fact}} \approx 6rn^2 + mn^2 + 6kr^2n \quad \text{and} \quad C_{\text{sol}} \approx mn + 4rn + 8kr^2,$$

respectively.

The complexity of these algorithms with $m = 2r$ is summarized in Table 3.3.

Table 3.3
Complexity of some HSS algorithms (with the low order terms dropped) when $m = 2r$, where all leaf level HSS block rows have row sizes $m$, and the HSS rank of the matrix is $r$.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSS construction (bottom-up)</td>
<td>$6rn^2$</td>
</tr>
<tr>
<td>ULV HSS factorization</td>
<td>$42r^2n$</td>
</tr>
<tr>
<td>ULV HSS solution</td>
<td>$37rn$</td>
</tr>
<tr>
<td>Dense-to-triangular-HSS factorization</td>
<td>$11rn^2$</td>
</tr>
<tr>
<td>Triangular HSS solution</td>
<td>$10rn$</td>
</tr>
</tbody>
</table>

4. Selected improvements.

4.1. HSS construction. One improvement to the HSS construction algorithm is to use a top-down construction procedure. For node $i$ with children $c_1$ and $c_2$, the HSS block associated with $i$ is given by

(4.1) \[ \left( \begin{array}{cc} A_{t_i \times t_{c_1}} & A_{t_i \times t_{c_2}} \\ \end{array} \right) = \left( \begin{array}{cc} A|_{t_{c_1} \times t_{c_1}} & A|_{t_{c_1} \times t_{c_2}} \\ A|_{t_{c_2} \times t_{c_1}} & A|_{t_{c_2} \times t_{c_2}} \end{array} \right). \]

Thus, $A_{t_i \times t_{c_1}}$ is first compressed; then its compressed form participates in the compression of the HSS blocks associated with $c_1$ and $c_2$ which include the subblocks on the right-hand side of (4.1).

The cost can be similarly counted with (3.7)–(3.8). When $m = 2r$, the complexity is $4rn^2$ flops, which is less than the bottom-up construction in the previous section. However, this top-down procedure accesses the matrix entries globally and may not be suitable for parallelization.
4.2. ULV HSS factorization. For a given HSS matrix, an improvement to the
ULV factorization algorithm is to replace the full QR factorization of \( \tilde{D}_i^T \) in (3.12) by a
partial one, as in [42]. That is, partition \( \tilde{D}_i \) in (3.11) as 
\[
\tilde{D}_i = \begin{pmatrix}
\tilde{D}_{i,1,1} & \tilde{D}_{i,1,2} \\
\tilde{D}_{i,2,1} & \tilde{D}_{i,2,2}
\end{pmatrix}
\]
and compute an LQ factorization
\[
( \begin{pmatrix}
\tilde{D}_{i,1,1} & \tilde{D}_{i,1,2} \\
\tilde{D}_{i,2,1} & \tilde{D}_{i,2,2}
\end{pmatrix}) = ( \begin{pmatrix}
\tilde{D}_{i,1,1} & 0
\end{pmatrix}) W_i^T.
\]
Also let
\[
( \begin{pmatrix}
\tilde{D}_{i,1,1} & \tilde{D}_{i,1,2} \\
\tilde{D}_{i,2,1} & \tilde{D}_{i,2,2}
\end{pmatrix}) = ( \begin{pmatrix}
\tilde{D}_{i,1,1} \\
\tilde{D}_{i,2,1} \end{pmatrix} ) W_i.
\]
Then we also get a form (3.13). It can be verified that the cost of such a factorization
is \( C_{\text{fact}} \approx \frac{4}{3} km^3 + \frac{128}{3} kr^3 + 6km^2r \) flops. With \( m = 2r \), we have \( C_{\text{fact}} \approx \frac{116}{3} r^2n \).

A further improvement is to replace the QR factorization in (3.12) by an LU one
and the matrix multiplication in (3.12) by a triangular solution. That is, compute
LU factorizations \( D_i = \tilde{D}_i T_i \), and let \( V_i = T_i^{-1} T_i \). This is done similarly for \( j \). Then
the following operation also yields a form as on the right-hand side of (3.13):
\[
( \begin{pmatrix}
Q_j^T \\
Q_i^T
\end{pmatrix})( D_j \begin{pmatrix}
U_j B_j V_j^T \\
D_i
\end{pmatrix}) ( T_j^{-1} \begin{pmatrix}
T_j^{-1}
\end{pmatrix}).
\]
We can verify that the complexity is
\[
C_{\text{fact}} \approx \frac{2}{3} m^3 n + 5rmn + 24kr^3.
\]
The saving over (3.15) is \( \frac{2}{3} m^2 n + rmn + \frac{100}{3} kr^3 \) flops. When \( m = 2r \), we have \( C_{\text{fact}} \approx \frac{74}{3} r^2n \), as compared with 42r^2n of (3.15).

4.3. Symmetry. Significant improvements can be made specifically for symmetric
matrices. If \( A \) is symmetric, in the HSS construction, only either the block row
compression or block column compression is needed. A straightforward implementa-
tion of the bottom-up construction in section 3.2 as in [42] costs \( 3rm^2 \) flops, and a
top-down procedure costs \( 2rn^2 \), where \( m = 2r \) is assumed.

For a symmetric positive definite HSS matrix, the general ULV factorization
algorithm in [14] does not specifically take advantage of the symmetry. A modified
ULV scheme is proposed in [42], where after introducing zeros into the off-diagonal
blocks just like (3.11), the diagonal blocks \( \tilde{D}_i \) are partially eliminated with Cholesky
factorizations. This version costs \( \frac{116}{3} r^2n \) flops when \( m = 2r \).

It can be further improved. Before the step (3.11), compute a full Cholesky
factorization \( D_i = L_i L_i^T \). In this way, we can avoid the dense matrix product
\( Q_i^T D_i \) in (3.11). That is, by letting \( \tilde{U}_i = L_i^{-1} U_i \) and computing a QR factorization \( \tilde{U}_i = Q_i \tilde{U}_i \),
we have
\[
( Q_i^T L_i^{-1} \begin{pmatrix}
Q_i L_i^{-1}
Q_i L_i^{-1}
\end{pmatrix}) ( D_1 \begin{pmatrix}
D_1 \\
U_2 B_2 U_1^T
\end{pmatrix} D_2 ) ( L_1^{-T} Q_1 \begin{pmatrix}
L_1^{-T} Q_1 \\
L_2^{-T} Q_2
\end{pmatrix})
= \begin{pmatrix}
( I \\
( I \\
( I \\
( I \\
( I \\
( I \\
\end{pmatrix}) \\
( I \\
( I \\
( I \\
( I \\
( I \\
\end{pmatrix}) \\
( I \\
( I \\
( I \\
( I \\
( I \\
\end{pmatrix}) \\
( I \\
( I \\
( I \\
( I \\
( I \\
\end{pmatrix})
\]
\]
Then $U_1B_1U_2^T$ participates in later eliminations. Here, there are no explicit matrix multiplications involving $Q$, and the partial diagonal elimination is straightforward.

In such a scheme, the total complexity is

$$C_{\text{fact}} = \frac{1}{3}m^2n + rmn + \frac{28}{3}kr^3 + 2r^2n.$$ 

If $m = 2r$, this is only about $9r^2n$. Table 4.1 lists the complexity of some symmetric HSS algorithms.

**Table 4.1**

Complexity of some symmetric HSS algorithms (with the low order terms dropped) for symmetric positive definite problems when $m = 2r$, where all leaf level HSS block rows have row sizes $m$ and the HSS rank of the matrix is $r$.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSS construction (bottom-up version)</td>
<td>$3rn^2$</td>
</tr>
<tr>
<td>HSS construction (top-down version)</td>
<td>$2rn^2$</td>
</tr>
<tr>
<td>Dense to triangular HSS factorization</td>
<td>$\frac{11}{2}rn^2$</td>
</tr>
<tr>
<td>ULV HSS factorization</td>
<td>$9r^2n$</td>
</tr>
</tbody>
</table>

**4.4. Numerical experiments.** We demonstrate the performance of some important HSS algorithms for a linear system

$$Ax = b, \quad \text{with } A = \alpha I + B = \left( \alpha + \sqrt{|x_i - x_j|} \right)_{n \times n},$$

where the points $x_i = \cos((2i+1)\pi/2n)$ are the zeros of the $n$th Chebyshev polynomial, and $\alpha$ is chosen to make $A$ positive definite. $B$ is illustrated to have a small numerical HSS rank $r$ in [9, 13]. Thus, $A$ has the low-rank property. The right-hand side $b$ is obtained by $b = Ax^*$ with the exact solution $x^*$ being random. The preliminary implementation is in MATLAB and the flop counts, timing, and accuracies are shown in Tables 4.2, 4.3, and 4.4, respectively.

**Table 4.2**

Complexity (floating point operations or flops) of some HSS algorithms applied to a symmetric system (4.3), where all leaf level HSS block rows have row sizes $m = 30$, and the relative tolerance in the compression is $\tau = 10^{-8}$.

<table>
<thead>
<tr>
<th>$n$</th>
<th>256</th>
<th>512</th>
<th>1024</th>
<th>2048</th>
<th>4096</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct HSS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Construction</td>
<td>2.26E6</td>
<td>1.11E7</td>
<td>4.87E7</td>
<td>1.98E8</td>
<td>7.86E8</td>
</tr>
<tr>
<td>ULV solution</td>
<td>4.55E5</td>
<td>1.08E6</td>
<td>2.34E6</td>
<td>4.40E6</td>
<td>8.67E6</td>
</tr>
<tr>
<td>Triangular HSS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Factorization</td>
<td>4.34E6</td>
<td>2.09E7</td>
<td>9.28E7</td>
<td>3.82E8</td>
<td>1.57E9</td>
</tr>
<tr>
<td>Solution</td>
<td>2.74E4</td>
<td>6.20E4</td>
<td>1.30E5</td>
<td>2.62E5</td>
<td>5.19E5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$n$</th>
<th>8192</th>
<th>16384</th>
<th>32768</th>
<th>65536</th>
<th>131072</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct HSS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Construction</td>
<td>3.09E9</td>
<td>1.18E10</td>
<td>4.47E10</td>
<td>1.67E11</td>
<td>6.17E11</td>
</tr>
<tr>
<td>ULV solution</td>
<td>1.64E7</td>
<td>3.33E7</td>
<td>5.53E7</td>
<td>9.90E7</td>
<td>1.78E8</td>
</tr>
<tr>
<td>Triangular HSS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Factorization</td>
<td>6.29E9</td>
<td>2.52E10</td>
<td>9.90E10</td>
<td>3.89E11</td>
<td>1.52E12</td>
</tr>
<tr>
<td>Solution</td>
<td>1.01E6</td>
<td>1.94E6</td>
<td>3.69E6</td>
<td>6.94E6</td>
<td>1.30E7</td>
</tr>
</tbody>
</table>

The flop counts are consistent with the theoretical counts. For example, the direct HSS construction (symmetric version of Algorithm 1) and the ULV factorization (symmetric version of Algorithm 2) have roughly $O(n^2)$ and $O(n)$ complexities, respectively. Also, the rows in Table 4.3 for the timing follow similar patterns. For
example, the ULV solution time roughly doubles when \( n \) doubles. In addition, both methods give satisfactory accuracies, as in Table 4.4.

From another point of view, we can compare the HSS rank with its estimate based on the actual costs. The flop count of the bottom-up symmetric HSS construction is \( C_{\text{constr}} \approx 2rn^2 + 2r^2n^2/m \), which indicates that \( r \) can be estimated by 
\[
\tilde{r} = \frac{m}{2}(1 + 2\frac{E}{n}) - 1
\]
with the actual cost \( C \). The estimate \( \tilde{r} \) can be viewed as an average rank value, while \( r \) is an upper bound. These results are shown in Table 4.5.

Table 4.5

<table>
<thead>
<tr>
<th>( n )</th>
<th>256</th>
<th>512</th>
<th>1024</th>
<th>2048</th>
<th>4096</th>
<th>8192</th>
<th>16384</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSS rank ( r )</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>Estimated ( \tilde{r} )</td>
<td>13</td>
<td>15</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>

5. HSS recompression. An important improvement to HSS methods is to recompress noncompact HSS representations. During the operations of HSS matrices we may get HSS forms where the \( U, V \) generators are not orthogonal or which are not compact enough even though the actual HSS rank is small. Such HSS matrices arise in sparse factorizations [41] and other construction schemes such as those based on randomized sampling [31, 34] or strong rank-revealing LU factorizations [25].
As an example, we consider adding two HSS matrices $A$ and $C$ with the same HSS tree structure and block partition. Assume that their generators are $D_i(A)$, $U_i(A)$, \ldots, and $D_i(C)$, $U_i(C)$, \ldots, respectively. Then, obviously, the sum $A + C$ is an HSS matrix with generators
\[
D_i(A + C) = D_i(A) + D_i(C), \quad B_i(A + C) = \text{diag} \left( B_i(A), B_i(C) \right),
\]
\[
U_i(A + C) = \left( \begin{array}{cc} U_i(A) & U_i(C) \end{array} \right), \quad R_i(A + C) = \text{diag} \left( R_i(A), R_i(C) \right),
\]
\[
V_i(A + C) = \left( \begin{array}{cc} V_i(A) & V_i(C) \end{array} \right), \quad W_i(A + C) = \text{diag} \left( W_i(A), W_i(C) \right).
\]
We see that the sizes of the HSS generators associated with the off-diagonal blocks of $A + C$ increase additively. However, the HSS rank of $A + C$ may be much smaller, and the HSS form is not sufficiently compact. To enhance the effectiveness of the HSS representation we can use recompression techniques to recover a compact HSS form.

In general, we consider recompressing an HSS matrix $A$ represented by generators $D_i$, $U_i$, \ldots, $B_i$. The details are explained as follows. The next proposition forms the foundation of the recompression technique here. It shows a nested representation for the HSS block row $A_i^-$ and column $A_i^T$ associated with a node $i$ as in (2.3). Here for convenience, we permute $A_i^-$ and $A_i^T$ and rewrite them as
\[
A_i^- = A_i^- \Pi_i = \left( A_{l_i \times t_i} \ A_{l_i \times (T \setminus \{t_i, \cup_{j \neq i} t_j\})} \right), \quad A_i^T = \Pi_i^T A_i^T = \left( A_{l_i \times t_i} \ A_{(T \setminus \{t_i, \cup_{j \neq i} t_j\}) \times t_i} \right).
\]
That is, $\Pi_i$ permutes $A_{l_i \times t_i}$ to the front of $A_i^-$, and $\Pi_i^T$ permutes $A_{l_i \times t_i}$ to the top of $A_i^T$.

**Proposition 5.1.** Assume $A$ is an HSS matrix as defined in Definition 2.1. For a node $i < \text{root}(T)$ with sibling $j$ and parent $p$, the corresponding HSS blocks $A_i^-$ and $A_i^T$ in (5.2) have the following forms:
\[
A_i^- = U_i S_i \hat{V}_i^T, \quad \text{with} \quad S_i = \left( \begin{array}{cc} B_i & R_i S_p \end{array} \right), \quad \hat{V}_i = \text{diag} \left( V_j, \Pi_p \hat{V}_p \right),
\]
\[
A_i^T = \bar{U}_i T_i V_i^T, \quad \text{with} \quad T_i = \left( \begin{array}{c} B_j \bar{T}_p W_j^T \end{array} \right), \quad \bar{U}_i = \text{diag} \left( U_j, \Pi_p \bar{U}_p \right),
\]
where $\Pi_p$ is a permutation matrix as defined in (5.2), $\hat{V}_i$ and $\bar{U}_i$ have orthonormal columns, and $S_p$, $T_p$, $\bar{U}_p$, $\hat{V}_p$ are empty matrices when $p = \text{root}(T)$.

**Proof.** We prove (5.3) by induction, and (5.4) can be proved similarly. The result is obvious for a node $i$ with $\text{par}(i) = 2k - 1$. Assuming (5.3) holds for a nonleaf node $i < 2k - 1$, we show it also holds for the children $c_1$ and $c_2$ of $i$. Clearly, $A|_{l_{c_1} \times t_{c_2}} = U_{c_1} B_{c_1} V_{c_2}^T$. Since
\[
A_i^- = \left( \begin{array}{cc} A|_{l_{c_1} \times (T \setminus \{t_i, \cup_{j \neq i} t_j\})} & A|_{l_{c_2} \times (T \setminus \{t_i, \cup_{j \neq i} t_j\})} \end{array} \right) \Pi_i, \quad U_i S_i \hat{V}_i^T = \left( \begin{array}{cc} U_{c_1} R_{c_1} & U_{c_2} R_{c_2} \end{array} \right) S_i \hat{V}_i^T,
\]
we have $A|_{l_{c_1} \times (T \setminus \{t_i, \cup_{j \neq i} t_j\})} = U_{c_1} R_{c_1} S_{c_1}^T \hat{V}_{c_1}^T \Pi_i^T$. Thus,
\[
A_i^- = \left( \begin{array}{cc} A|_{l_{c_1} \times t_{c_1}} & A|_{l_{c_2} \times (T \setminus \{t_i, \cup_{j \neq i} t_j\})} \end{array} \right) \left( \begin{array}{cc} U_{c_1} B_{c_1} V_{c_2}^T & U_{c_1} R_{c_1} S_{c_1} \hat{V}_{c_1}^T \Pi_i^T \end{array} \right)
\]
\[
= U_{c_1} \left( \begin{array}{cc} B_{c_1} & R_{c_1} S_{c_1} \end{array} \right) \text{diag} \left( V_{c_2}^T, (\Pi_i \hat{V}_i)^T \right)
\]
\[
= U_{c_1} S_{c_1} \hat{V}_{c_1}^T.
\]
Therefore, (5.3) holds for $c_1$. Similarly, it holds for $c_2$.  

\[\square\]
This proposition also justifies the precise definition of the compactness of an HSS form traditionally used in HSS methods.

**Definition 5.2.** An HSS matrix \( A \) as defined in Definition 2.1 is said to be in a compact form if its HSS rank is small, and the column sizes of all \( U_i \) and \( V_i \) are close to the ranks of \( A_i^- \) and \( A_i^+ \), respectively. Also, we say \( A \) is in a proper form if it is compact and all \( U_i \) and \( V_i \) have orthonormal columns.

According to this proposition, we design a recompression scheme with two stages. In the first stage or a forward stage, we compress each to be orthonormal, which is done in a postordering traversal of the HSS tree. In the second stage or a backward stage, we compress \( S_i \) and \( T_i \) in (5.3)–(5.4) to further make the HSS form a compact one, which is done in a reverse-postordering traversal. This procedure is more systematic than a preliminary one in our technical report [12]. We use tilded notation \( \hat{R}, \hat{W}, \) etc. to denote the generators after compression, and hatted notation \( \hat{R}, \hat{W}, \) etc. to denote intermediate temporary forms of the generators.

**5.1. Forward stage.** In the first stage, we traverse the tree bottom-up for nodes \( i = 1, 2, \ldots \) The generators \( U_i \) and \( V_i \) are compressed and \( R_i, W_i, \) and \( B_i \) are updated. If \( i \) is a leaf, compute QR factorizations

\[
U_i = \hat{U}_i F_i, \quad V_i = \hat{V}_i G_i,
\]

where \( \hat{U}_i \) and \( \hat{V}_i \) are the new generators. That is, after this compression step, \( U_i \) and \( V_i \) are updated to \( \hat{U}_i \) and \( \hat{V}_i \), respectively. The matrices \( F_i \) and \( G_i \) are passed to generators \( R_i \) and \( W_i \), respectively, as in

\[
\hat{R}_i = F_i R_i, \quad \hat{W}_i = G_i W_i.
\]

If \( i \) is a nonleaf node, \( U_i \) and \( V_i \) are compressed indirectly. For example, \( U_i \) is implicitly given by

\[
U_i = \begin{pmatrix} U_{c_1} & R_{c_1} \\ U_{c_2} & R_{c_2} \end{pmatrix} = \begin{pmatrix} \hat{U}_{c_1} & \hat{R}_{c_1} \\ \hat{U}_{c_2} & \hat{R}_{c_2} \end{pmatrix} = \begin{pmatrix} \hat{U}_{c_1} & \hat{U}_{c_2} \\ \hat{R}_{c_1} & \hat{R}_{c_2} \end{pmatrix},
\]

where \( \hat{U}_{c_1} \) and \( \hat{U}_{c_2} \) are compressed by recursion, and have orthonormal columns. Thus it suffices to compute QR factorizations

\[
\begin{pmatrix} \hat{R}_{c_1} \\ \hat{R}_{c_2} \end{pmatrix} = \begin{pmatrix} F_i \\ W_i \end{pmatrix}, \quad \begin{pmatrix} W_{c_1} \\ W_{c_2} \end{pmatrix} = \begin{pmatrix} F_i \\ W_i \end{pmatrix} G_i.
\]

This gives the new generators \( \hat{R}_{c_1}, \hat{R}_{c_2}, \hat{W}_{c_1}, \hat{W}_{c_2} \). Then use (5.6) to update \( R_i, W_i \). If \( i \) is also a right node, update

\[
\tilde{B}_i = F_i B_i G_j^T, \quad \tilde{B}_j = F_j B_j F_i^T,
\]

where \( j = \text{sib}(i) \), and \( F_j \) and \( G_j \) are available from the \( j \)th compression step (associated with node \( j \) which has been visited before).

Repeat these steps along the postordering traversal of \( T \). At the end of this stage, we have \( A \) in a new HSS form with generators \( \tilde{D}_i, \tilde{U}_i, \ldots, \tilde{B}_i \). After this stage, all \( \tilde{U}_i \) and \( \tilde{V}_i \) generators have orthonormal columns (and full column ranks).
5.2. Backward stage. This is a top-down stage, or the reverse-postordering traversal of the nodes \( i = \text{root}(T) − 1, \text{root}(T) − 2, \ldots, 1 \). For convenience, we still use \( D_i, U_i, \ldots, B_i \) to denote the generators of \( A \) after the forward stage and use tilded notation for the new generators. Here, we compress \((5.11)\) \( \hat{\mathbf{R}}_c, \mathbf{W}_c, \mathbf{R}_c, \mathbf{W}_c \) for the children \( c_1, c_2 \) of \( i \) with

\[
(5.10) \quad \begin{pmatrix} \hat{\mathbf{R}}_{c_1} \\ \mathbf{R}_{c_2} \end{pmatrix} = \begin{pmatrix} \mathbf{R}_{c_1} \\ \mathbf{R}_{c_2} \end{pmatrix} P_i, \quad \begin{pmatrix} \hat{\mathbf{W}}_{c_1} \\ \mathbf{W}_{c_2} \end{pmatrix} = \begin{pmatrix} \mathbf{W}_{c_1} \\ \mathbf{W}_{c_2} \end{pmatrix} Q_i.
\]

This implicitly updates \( U_i, V_i \). Accordingly, set \( \hat{B}_i = \hat{S}_i \) and \( \hat{T}_i = B_j \), where \( j = \text{sib}(i) \).

If \( i \) satisfies \( \text{par}(i) \neq \text{root}(T) \), according to Proposition 5.1, we compress \( S_i \) and \( T_j \). Let \( p = \text{par}(i), j = \text{sib}(i) \). The proof in Theorem 5.3 shows that the compression of \( S_i \) and \( T_j \) can be done via

\[
(5.11) \quad \hat{S}_i = ( B_i \hat{R}_i \hat{S}_p ) \quad \text{and} \quad \hat{T}_j^T = ( B_j^T \hat{W}_j \hat{T}_p ),
\]

respectively, where \( \hat{R}_i, \hat{S}_p, \hat{W}_i, \) and \( \hat{T}_j \) are results from the recompression step associated with \( p \) by recursion (with \( i \) set to be \( p \) in (5.10), (5.12), and (5.13)). Compute SVDs

\[
(5.12) \quad \hat{S}_i = P_i \hat{S}_i X_i^T = P_i \hat{S}_i \begin{pmatrix} X_{i,1}^T \\ X_{i,2}^T \end{pmatrix}, \quad \text{and} \quad (5.13) \quad \hat{T}_j^T = Q_j \hat{T}_j Y_j^T = Q_j \hat{T}_j \begin{pmatrix} Y_{j,1}^T \\ Y_{j,2}^T \end{pmatrix},
\]

where \( X_i^T \) and \( Y_j^T \) are partitioned according to the column partitions in (5.11). The fact that \( \mathcal{A}_i^- \) and \( \mathcal{A}_j^+ \) share a common block \( A_{|x_i, x_j}, = U_i B_i V_j^T \) yields

\[
B_i = P_i \hat{S}_i X_{i,1}^T = Y_{j,1} \hat{T}_j^T Q_j.
\]

Thus, we set

\[
(5.14) \quad \hat{B}_i = P_i^T B_i Q_j, \quad P_i^T Y_{j,1} \hat{T}_j^T = \hat{S}_i X_{i,1}^T Q_j, \quad (5.15) \quad \hat{R}_i = P_i^T \hat{R}_i, \quad \hat{W}_j = Q_j \hat{W}_j.
\]

Next, if \( i \) is also a leaf node, we update \( U_i \) and \( V_i \) as

\[
(5.16) \quad \hat{U}_i = U_i P_i, \quad \hat{V}_i = V_i Q_i,
\]

respectively. Otherwise, we compute the update (5.10).

The recompression is then done recursively. When it finishes, \( A \) is in a compact HSS form with generators \( \hat{D}_i, \hat{U}_i, \ldots, \hat{B}_i \).

5.3. Algorithm and analysis. We summarized the procedure as follows.

**Algorithm 3 (HSS recompression).**

for nodes (separators) \( i = 1, 2, \ldots, \text{root}(T) − 1 \) (Forward stage)
if \( i \) is a leaf
  \( - \) Compress \( U_i \) and \( V_i \) as in (5.5) and update \( R_i \) and \( W_i \) as in (5.6)
else
  \( - \) Compute QR factorizations in (5.7) and update \( R_i \) and \( W_i \) as in (5.6)
end
if \( i \) is a right node, update \( B_i \) and \( B_j \) as in (5.8) for \( j = \text{sib}(i) \)
end
Set \( D_i, U_i \), etc. to be \( \tilde{D}_i, \tilde{U}_i \), etc., respectively
for nodes (separators) \( i = \text{root}(\mathcal{T}) - 1, \text{root}(\mathcal{T}) - 2, \ldots, 1 \) (Backward stage)
if \( \text{par}(i) = \text{root}(\mathcal{T}) \)
  \( - \) Compute an SVD for \( B_i \) as in (5.9) and update \( R_{c_1}, W_{c_1}, R_{c_2}, W_{c_2} \) as in (5.10) for the children \( c_1, c_2 \) of \( i \)
  \( - \) Set \( \tilde{B}_i = \tilde{S}_i \) and \( \tilde{T}_i = B_j \)
else
  \( - \) Form \( \hat{S}_i \) and \( \hat{T}_j^T \) as in (5.11) and compute SVDs as in (5.12) and (5.13)
  \( - \) Compute \( \tilde{B}_i \) as in (5.14) and \( \tilde{R}_i \) and \( \tilde{W}_j \) as in (5.15)
end
if \( i \) is a leaf, update \( U_i \) and \( V_i \) as in (5.16)
end
Set \( D_i, U_i \), etc. to be \( \tilde{D}_i, \tilde{U}_i \), etc., respectively.

Next, we briefly discuss the effectiveness of this recompression procedure and the complexity.

Theorem 5.3. After the two stages of recompression, the new HSS representation of \( A \) is in a proper form.

Proof. After the first stage, all \( \tilde{U}_i \) and \( \tilde{V}_i \) have orthonormal columns due to (5.5) and (5.7). This property is also preserved in the second stage as in (5.10) and (5.16). We just need to show that the second stage produces a compact HSS form. Let the new generators after the second stage be \( \tilde{D}_i, \tilde{U}_i \), \( \ldots, \tilde{B}_i \).

We show that the second stage recursively yields \( S_p \) in the following form for a node \( p = \text{par}(i) \):

\[
(5.17) \quad S_p = P_p \tilde{S}_p Z_p^T, \quad Z_p = \text{diag}(I, Z_{\text{par}(p)}) X_p,
\]

where \( Z_p \) has orthonormal columns, and \( Z_p = Q_p \) if \( \text{par}(p) = \text{root}(\mathcal{T}) \), or by (5.12) otherwise. We show the result also holds for \( i \). That is,

\[
(5.18) \quad S_i = \begin{pmatrix} B_i & R_i S_p \end{pmatrix} = \begin{pmatrix} B_i & R_i P_p \tilde{S}_p Z_p^T \end{pmatrix} = \begin{pmatrix} B_i & \hat{R}_i \hat{S}_p \end{pmatrix} \text{diag}(I, Z_p^T) = \hat{S}_i \text{diag}(I, Z_p^T) = P_i \tilde{S}_i X_i^T \text{diag}(I, Z_p^T) = P_i \tilde{S}_i Z_i^T,
\]

where (5.12) is used, and \( Z_i = \text{diag}(I, Z_p) X_i \). This verifies the recursion and confirms that the compression of \( S_i \) can be done by compressing \( \hat{S}_i \) in (5.11). Thus,

\[
A_i^- = U_i S_i V_i^T = U_i (P_i \tilde{S}_i Z_i^T) V_i^T = (U_i P_i) \tilde{S}_i (Z_i^T V_i^T) = \tilde{U}_i \tilde{S}_i (\tilde{V}_i Z_i)^T,
\]

where both \( \tilde{U}_i \) and \( \tilde{V}_i Z_i \) have orthonormal columns, and \( \tilde{S}_i \) has full rank. Therefore, the column size of \( \tilde{U}_i \) is the rank of \( A_i^- \), and \( A_i^- \) is in a compact compressed form. A similar result holds for \( A_i^+ \).
The complexity of the algorithm can be conveniently counted. For simplicity, we make assumptions about the generator sizes and ranks as indicated in Table 5.1. Then it can be verified that the total cost of the forward and backward stages is
\[
C_{\text{recom}} = r_1 \left( 8mr_0 + 12r_0^2 + 12r_0r_1 - \frac{8}{3}r_1^2 - 4mr_1 \right) \frac{n}{m} + r \left( 4mr_1 + 24r_1^2 + 28rr_1 + \frac{20}{3}r^2 \right) \frac{n}{m}.
\]

Table 5.1
Generator sizes before and after the two stages of recompression, where \( r_1 \) is the rank of each \( U_i \) and \( V_i \) initially, and \( r \) is the HSS rank satisfying \( r < r_1 \).

<table>
<thead>
<tr>
<th>Generator</th>
<th>Initial size</th>
<th>After first stage</th>
<th>After second stage</th>
</tr>
</thead>
<tbody>
<tr>
<td>( U_i, V_i )</td>
<td>( m \times r_0 )</td>
<td>( m \times r_0 )</td>
<td>( m \times r_1 )</td>
</tr>
<tr>
<td>( R_i, W_i, B_i )</td>
<td>( r_0 \times r_0 )</td>
<td>( r_1 \times r_1 )</td>
<td>( r \times r )</td>
</tr>
</tbody>
</table>

As an example, we consider recompressing an HSS matrix where \( r_0 = 2r_1 = 4r \) and \( m = 2r \). See Table 5.2 for the recompression cost and the comparison of ULV operation costs before and after the recompression.

Table 5.2
Costs of HSS operations for an HSS matrix before and after recompression, where the HSS rank is \( r \), and \( m = 2r \), \( r_0 = 2r_1 = 4r \) (before recompression, as in Table 5.1).

<table>
<thead>
<tr>
<th>Operation</th>
<th>Before recompression</th>
<th>After recompression</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSS recompression</td>
<td>413n ( r^2 )</td>
<td>413n ( r^2 )</td>
</tr>
<tr>
<td>ULV factorization</td>
<td>736n ( r^2 )</td>
<td>41b ( r^2 )</td>
</tr>
<tr>
<td>ULV solution</td>
<td>148n</td>
<td>37n</td>
</tr>
</tbody>
</table>

6. Relaxation of rank requirements in HSS operations. In this section, we propose another useful improvement to classical HSS methods from a different point of view.

Existing HSS algorithms often assume the HSS ranks to be bounded [39, 41, 42, etc.]. Sometimes, the bound may be pessimistic (see, e.g., \( r \) as a bound for \( \tilde{r} \) in Table 4.5). In [14], a special rank pattern for HSS blocks is used. A concept of rank functions is also given in [8]. Here, we relax the HSS rank requirement by proposing several general rank patterns. For simplicity, assume all HSS block rows corresponding to level \( l \) of the HSS tree have the same row size \( m_{L-l} \), where the bottom/leaf HSS block rows have row dimensions \( m_0 \) (constant) so that there are totally \( L \) levels with
\[
m_0 \approx \frac{n}{2^L}, \quad m_{L-l} = m_0 2^{L-l}.
\]

Also assume \( r_0 \) is maximum rank of bottom level HSS blocks. At level \( l \), we allow the numerical ranks of the HSS blocks to be bounded by
\[
r_{L-l} = (L - l + 1)r_0 = \left( \log_2 \frac{m_{L-l}}{m_0} + 1 \right) r_0.
\]

That is, \( r_{L-l} \) increases as \( m_{L-l} \) increases (and as \( l \) decreases).

For HSS construction algorithms such as the one described in section 3.2, the cost associated with each tree node at level \( l \) is \( O(r_{L-l}^2 n) \) flops. There are \( 2^l \) nodes at level
Using (6.1) and the fact that \( \sum_{l=0}^{L}(L-l+1)^{2^{l}} \approx 2 \times 2^{L} \), we have the total cost for the HSS construction:

\[
C_{\text{constr}} = \sum_{l=0}^{L} O(r_{L-l}^{2})2^{l} = O\left(r_{0}^{2}\sum_{l=0}^{L}(L-l+1)^{2^{l}}\right) = O(r_{0}^{2}2^{L}n) = O(n^{2}).
\]

Thus, after relaxing the rank requirement, we do not increase the order of the complexity.

Similarly, for ULV HSS factorizations, the cost associated with each tree node is \( O(r_{L-l}^{3}) \) flops. Using \( \sum_{l=0}^{L}(L-l+1)^{3}2^{l} \approx 52 \times 2^{L} \), we have the total cost

\[
C_{\text{fact}} = \sum_{l=0}^{L} O(r_{L-l}^{3})2^{l} = O\left(r_{0}^{3}\sum_{l=0}^{L}(L-l+1)^{3}2^{l}\right) = O(n).
\]

The assumption (6.2) indicates \( r_{L-l} = O(r_{0}^{p}m_{L-l}) \), and the HSS rank is bounded by \( O(\log^{2}n) \). More generally, we can allow \( r_{\ell} \) to be

\[
r_{L-l} = O\left((r_{0}^{p}m_{L-l})^{p}\right) = O\left(r_{0}^{p}(L-l+1)^{p}\right),
\]

where \( p \) is a positive integer. For example, when \( p = 2 \) we have

\[
C_{\text{fact}} = \sum_{l=0}^{L} r_{0}^{5}(L-l+1)^{6}2^{l} \approx O(r_{0}^{6}2^{L}) = O(n).
\]

Similarly, the storage remains \( O(n) \) under the new rank condition.

We can further relax the rank requirements so that \( r_{L-l} \) increases as \( O(m_{L-l}^{1/p}) \), where \( p \) is a positive integer. The derivations are similar. For example, for \( p = 3 \) and \( 2 \), the HSS construction costs are now

\[
C_{\text{constr}} = \sum_{l=0}^{L} O\left(m_{L-l}^{1/3}\right)^{2}2^{l} = \sum_{l=0}^{L} O\left((m_{0}2^{L-l})^{2/3}\right) n2^{l} = O(n^{2}) \text{ and}
\]

\[
C_{\text{constr}} = \sum_{l=0}^{L} O\left(m_{L-l}^{1/2}\right)^{2}2^{l} = \sum_{l=0}^{L} O\left(m_{0}2^{L-l}\right) n2^{l} = O(n^{2} \log^{2}n),
\]

respectively.

In addition, we can generalize the rank pattern used in [14]. That is, assume

\[
r_{L-l} = \alpha^{L-l} r_{0}.
\]

\( |\alpha| = \sqrt{2} \) is discussed in [14]. Here, we estimate the counts for a general \( \alpha \). For example,

\[
C_{\text{constr}} = \sum_{l=0}^{L} O(r_{L-l}^{2})2^{l} = n_{0}^{2}\sum_{l=0}^{L} O\left(\alpha^{2(L-l)} \times 2^{l}\right) = n_{0}^{2}2^{L}O\left(\sum_{l=0}^{L} \left(\frac{2}{\alpha^{2}}\right)^{l}\right)
\]

\[
= \begin{cases} 
O(n^{2}) & \text{if } |\alpha| < \sqrt{2}, \\
O\left(n^{2} \log n\right) & \text{if } |\alpha| = \sqrt{2}, \\
O\left(n^{1+\log_{2}\alpha^{2}}\right) & \text{if } |\alpha| > \sqrt{2}.
\end{cases}
\]

Therefore, it is desirable to choose \( |\alpha| \leq \sqrt{2} \).

All of these results are summarized in the following theorem.
THEOREM 6.1. Suppose an order \( n \) matrix \( A \) is recursively partitioned into \( L = O(\log_2 m_0) \) levels of HSS blocks following a perfect binary tree, so that the bottom level HSS block row size is \( m_0 \). Let \( m_{L-l} = O(m_0 2^{2L-l}) \) be the row dimensions of the HSS block rows at level \( l \), and let \( r_l \) be their maximum numerical rank.

1. When \( r_0 = O((\log_2 m_0)^p) \) (so that \( \max_l r_{L-l} = O((\log_2 n)^p) \) ) with an integer \( p \geq 0 \), an HSS form of \( A \) can be constructed in \( C_{\text{constr}} \) flops, a ULV factorization of the HSS form needs \( C_{\text{fact}} \) flops, and the HSS form or its ULV factor needs memory size \( S_{\text{mem}} \), where

\[
C_{\text{constr}} = O(n^2), \quad C_{\text{fact}} = O(n), \quad S_{\text{mem}} = O(n).
\]

(These are the same counts as in the classical situation [14, 42, etc.] where \( r_0 = O(1) \) or \( p = 0 \).

2. When \( r_{L-l} = O(m_{L-l}^{1/p}) \) (so that \( \max_l r_{L-l} = O(n^{1/p}) \) ) with an integer \( p > 0 \), we have the following counts:

(a) If \( p > 3 \), then the counts are

\[
C_{\text{constr}} = O(n^2), \quad C_{\text{fact}} = O(n), \quad S_{\text{mem}} = O(n).
\]

(b) If \( p = 3 \), then the counts are

\[
C_{\text{constr}} = O(n^2), \quad C_{\text{fact}} = O(n \log_2 n), \quad S_{\text{mem}} = O(n).
\]

(c) If \( p = 2 \), then the counts are

\[
C_{\text{constr}} = O(n^2 \log_2 n), \quad C_{\text{fact}} = O(n^{3/2}), \quad S_{\text{mem}} = O(n \log_2 n).
\]

3. When \( r_{L-l} = \alpha^{L-l} r_0 \) with \( \alpha > 0 \), we have the following counts:

(a) If \( |\alpha| < \sqrt[3]{2} \),

\[
C_{\text{constr}} = O(n^2), \quad C_{\text{fact}} = O(n), \quad S_{\text{mem}} = O(n).
\]

(b) If \( |\alpha| = \sqrt[3]{2} \),

\[
C_{\text{constr}} = O(n^2), \quad C_{\text{fact}} = O(n \log_2 n), \quad S_{\text{mem}} = O(n).
\]

(c) If \( \sqrt[3]{2} < |\alpha| < \sqrt{2} \),

\[
C_{\text{constr}} = O(n^2), \quad C_{\text{fact}} = O(n^{\log_2 \alpha^3}), \quad S_{\text{mem}} = O(n).
\]

(d) If \( |\alpha| = \sqrt{2} \),

\[
C_{\text{constr}} = O(n^2 \log_2 n), \quad C_{\text{fact}} = O(n^{3/2}), \quad S_{\text{mem}} = O(n \log_2 n).
\]

These results indicate that we can allow flexible rank patterns to achieve similar results, and enhance the applicability of HSS methods. Moreover, notice that the memory sizes (and also the solution costs) are both nearly linear in \( n \). It is possible to generalize them to other structured matrices such as sequentially semiseparable matrices [10], quasiseparable matrices [3, 19], and \( H^2 \) matrices. Similar results can also be derived for structured sparse solvers based on HSS matrices as in [41].

As a numerical example, we consider some ill-conditioned Toeplitz matrices, which are converted into Cauchy-like matrices with the aid of displacement structures (see, e.g., [21, 38]). It is shown that these Cauchy-like matrices have the low-rank property

\[
\begin{align*}
\text{memory sizes (and also the solution costs) are both nearly linear in n. It is possible to generalize them to other structured matrices such as sequentially semiseparable matrices [10], quasiseparable matrices [3, 19], and H^2 matrices. Similar results can also be derived for structured sparse solvers based on HSS matrices as in [41].}
\end{align*}
\]
and the HSS rank is bounded by $O(\log n)$ [15, 37]. It can also be shown that the ranks satisfy part 1 of Theorem 6.1 with $p = 1$. Here, we just use the low-rank property and ignore any further special structures so as to test the performance of regular HSS algorithms. See Table 6.1, which verifies that with the relaxed rank bounds, the order of the complexity is still similar to that with the classical rank bound, although the prefactor or constant in the count may be larger.

### 7. Conclusions.

This paper gives systematic complexity analysis for some HSS algorithms. Detailed flop counts are conducted. This helps the development of several strategies for improving existing HSS methods and for developing new fast ones, including modified HSS construction and solution and new HSS recompression. We also relax classical rank requirements to allow more flexibility in using HSS methods. The analysis is verified by some numerical examples. The ideas are also useful in sparse structured methods.

**Acknowledgments.** The author is very grateful to the anonymous referees for their valuable suggestions. The author would also like to thank Professors Shivkumar Chandrasekaran, Ming Gu, and Dan Jiao for some useful discussions.

**REFERENCES**


