Fast Memory Efficient Construction Algorithm for Hierarchically Semi-separable Representations

University of California Santa Barbara

Department of Electrical and Computer Engineering

Kristen Lessel, Matthew Hartman, Shivkumar Chandrasekaran

October 29, 2015

Overview

- Review of Hierarchically Semi-Separable (HSS) Representation
  - Notation
- Previous HSS Algorithm Complexities
- Memory Efficient Algorithm
  - Phase 1
  - Phase 2
- Memory Consumption
- “A Fast Memory Efficient Construction Algorithm for Hierarchically Semi-Separable Representations” submitted for publication

Use a Partition Tree to Block Partition a Matrix

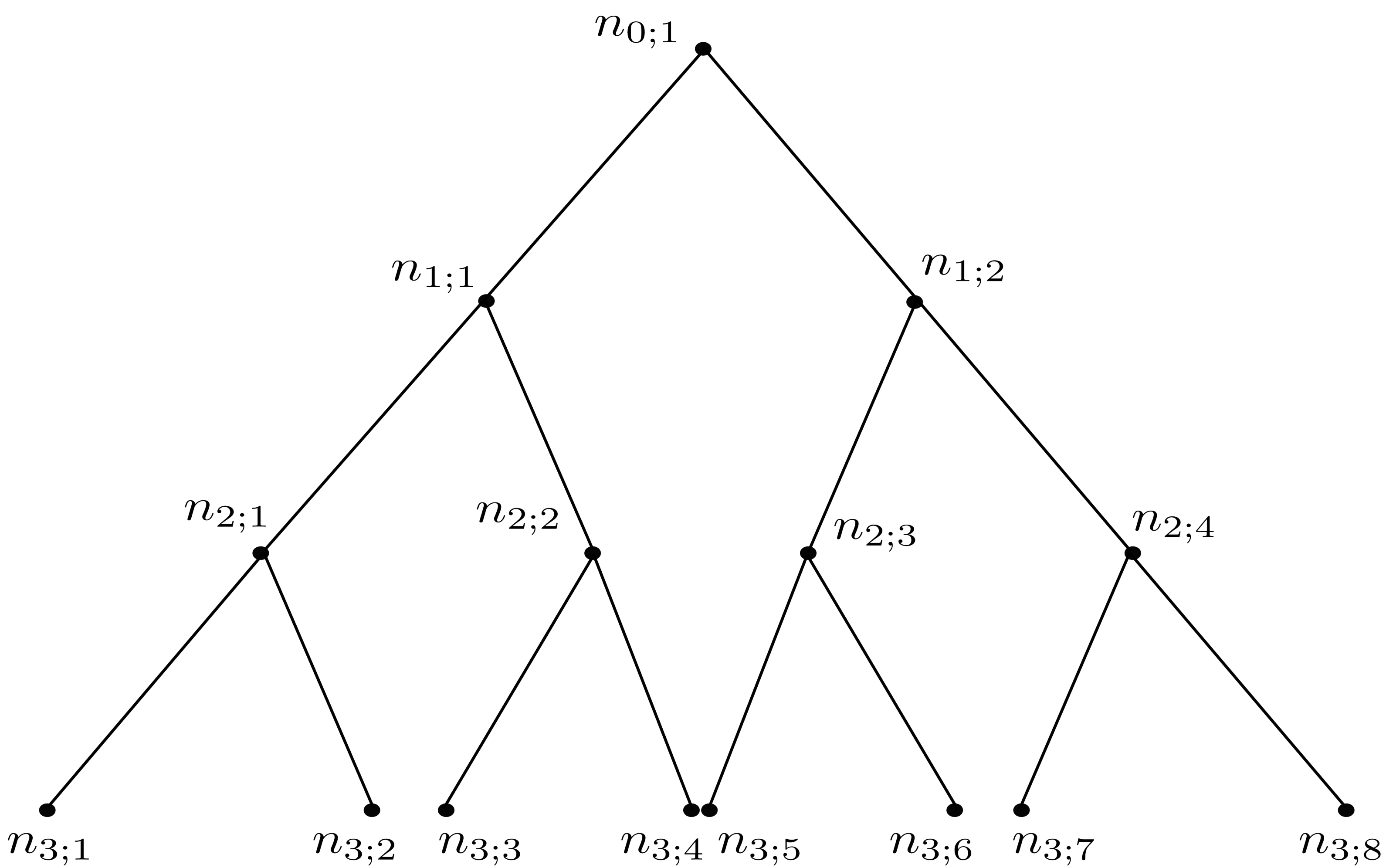
- Partition A according to the integers at the first level of the partition tree

$$\begin{array}{cc} & \begin{array}{cc} n_{1;1} & n_{1;2} \end{array} \\ A_{0;1,1} = \begin{array}{c} m_{1;1} \\ m_{1;2} \end{array} & \begin{pmatrix} A_{1;1,1} & A_{1;1,2} \\ A_{1;2,1} & A_{1;2,2} \end{pmatrix} \end{array}$$

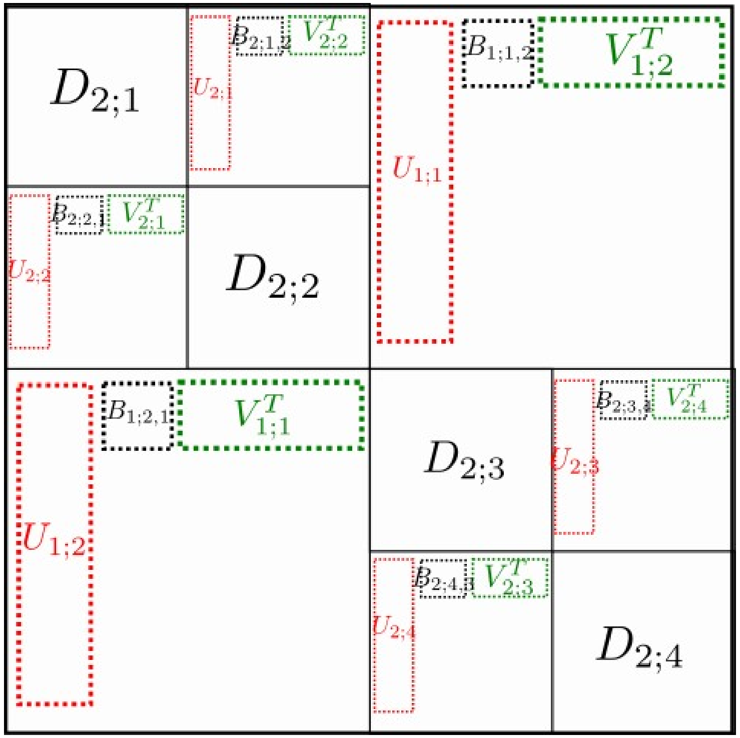
- Recursively partition the block rows and columns of A

$$\begin{array}{cc} & \begin{array}{cccc} n_{2;1} & n_{2;2} & n_{2;3} & n_{2;4} \end{array} \\ A_{0;1,1} = \begin{array}{c} m_{2;1} \\ m_{2;2} \\ m_{2;3} \\ m_{2;4} \end{array} & \begin{pmatrix} A_{2;1,1} & A_{2;1,2} & A_{2;1,3} & A_{2;1,4} \\ A_{2;2,1} & A_{2;2,2} & A_{2;2,3} & A_{2;2,4} \\ A_{2;3,1} & A_{2;3,2} & A_{2;3,3} & A_{2;3,4} \\ A_{2;4,1} & A_{2;4,2} & A_{2;4,3} & A_{2;4,4} \end{pmatrix} \end{array}$$

Definition – Complete Partition Tree

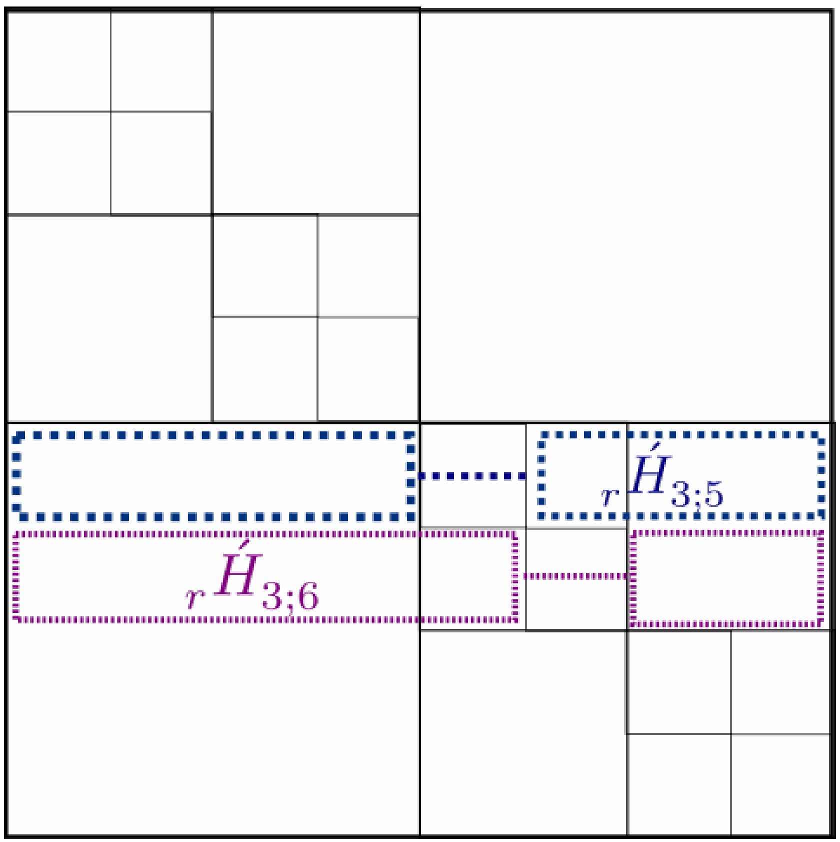


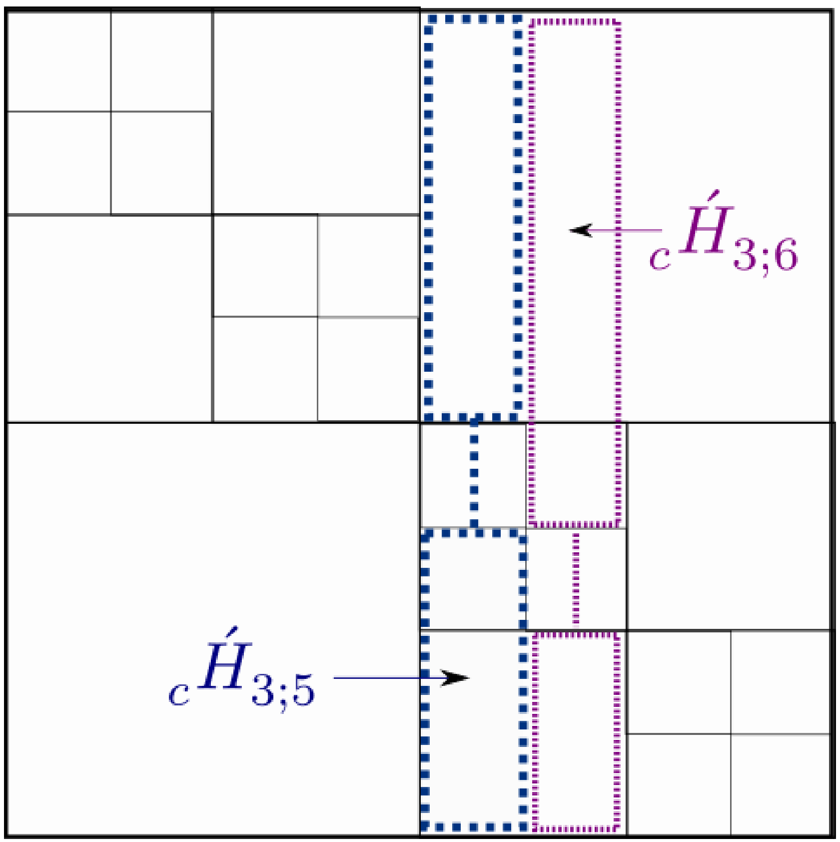
HSS Representation



- Off-Diagonal blocks thus have low rank and can be compressed
- Only store smaller basis matrices ($U_{k;i}, V_{k;i}$) and translation operators ($R_{k;i}, W_{k;i}$)

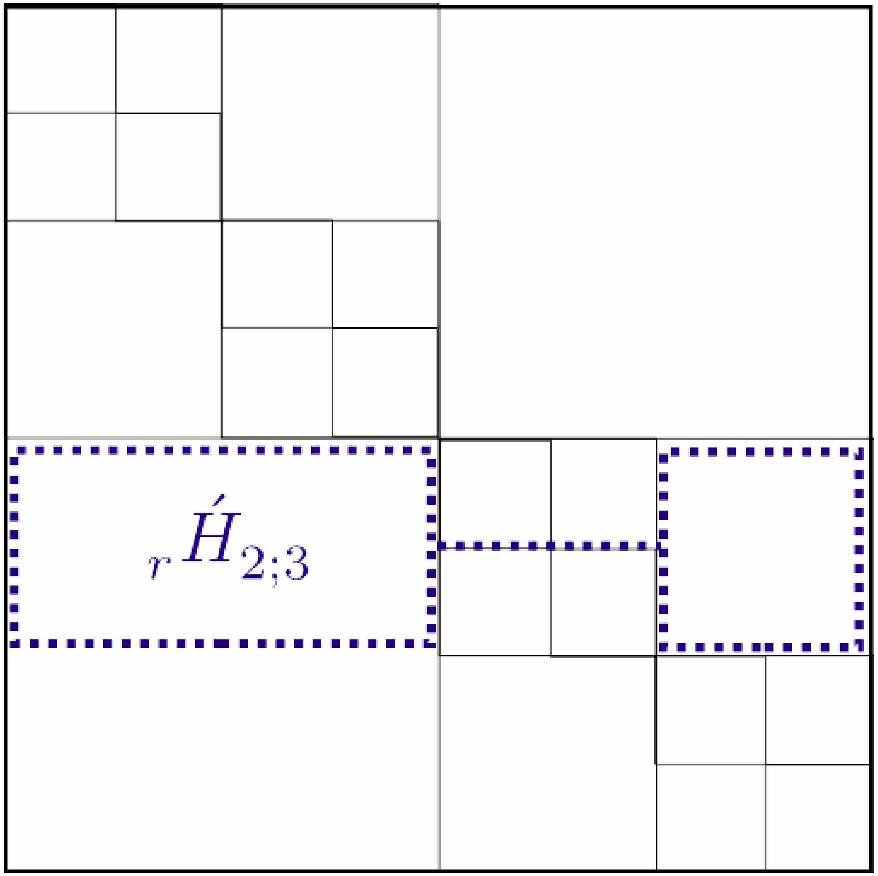
Definition - Hankel Blocks

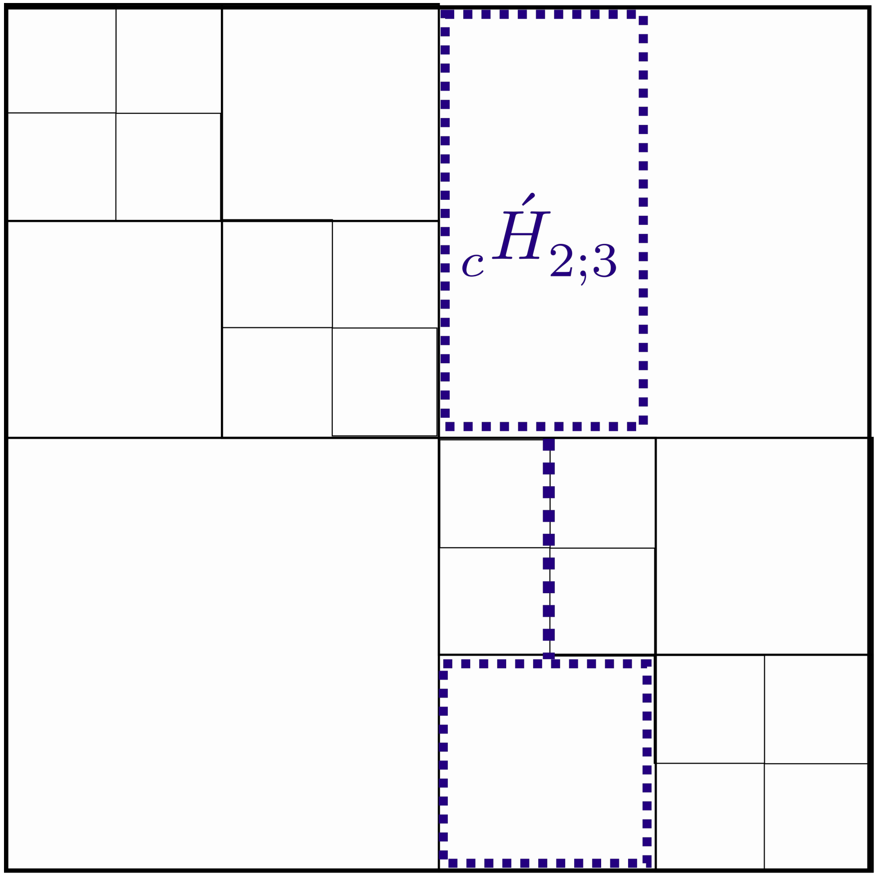




Block rows/columns of A, excluding diagonal blocks

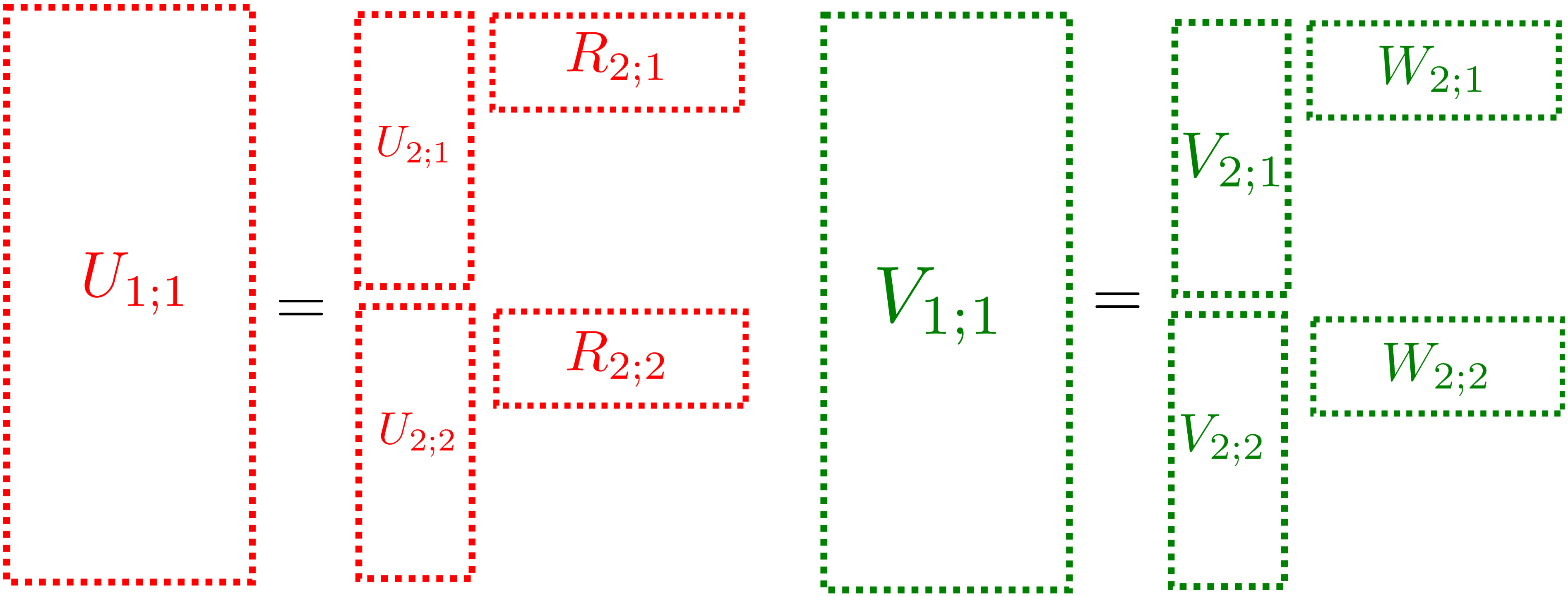
Definition - Hankel Blocks





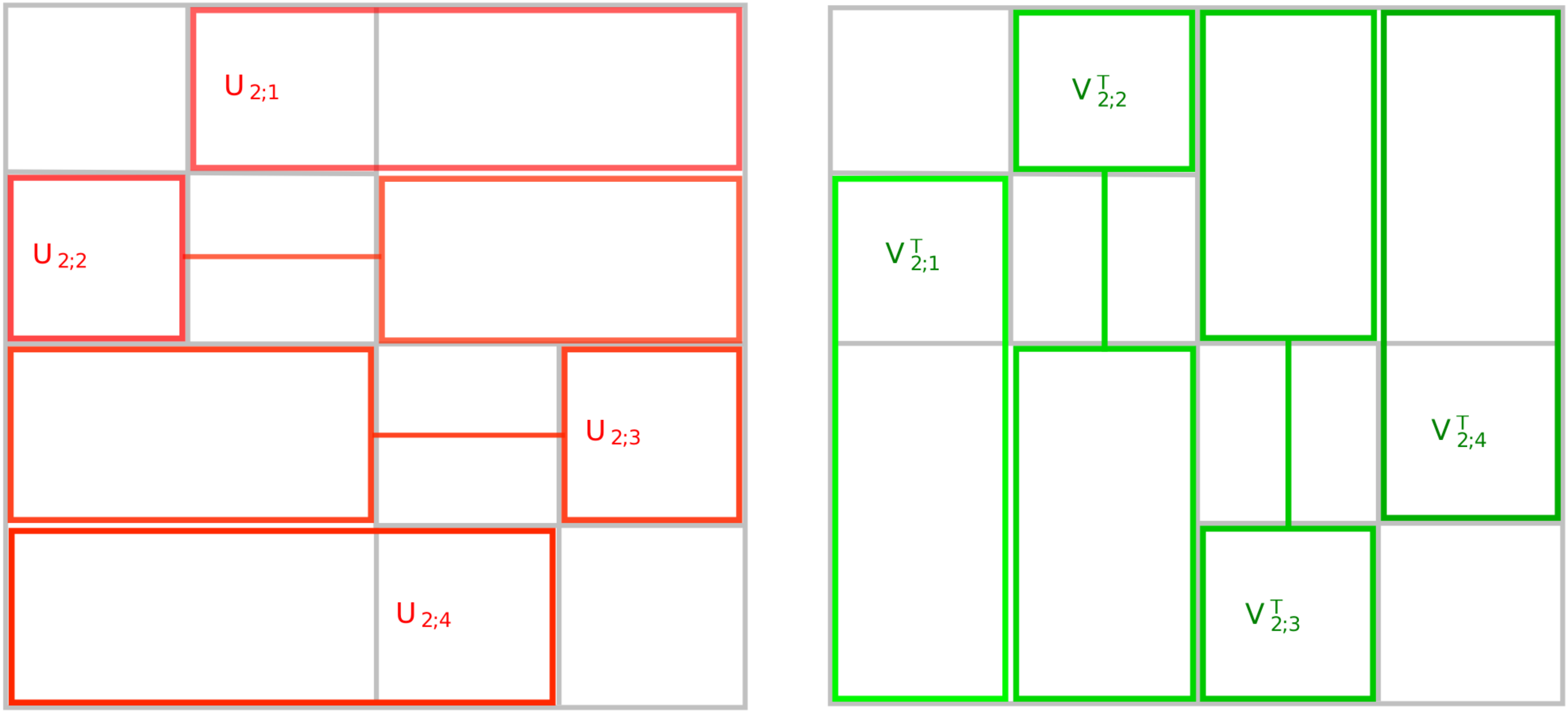
Block rows/columns of A, excluding diagonal blocks

Larger Basis Matrices Can Be Stored as Translated Versions of Smaller Basis Matrices

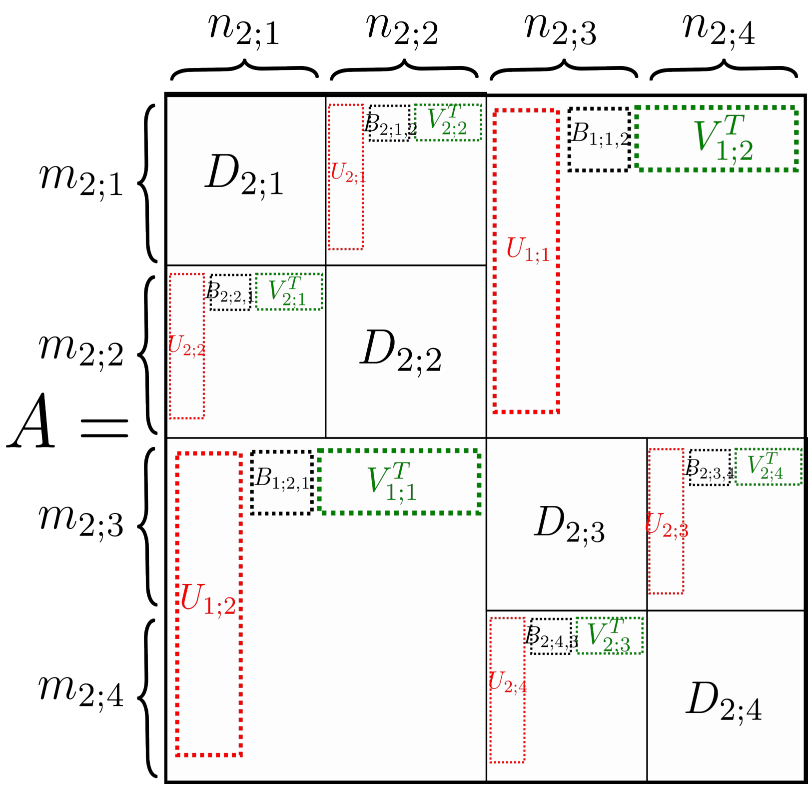


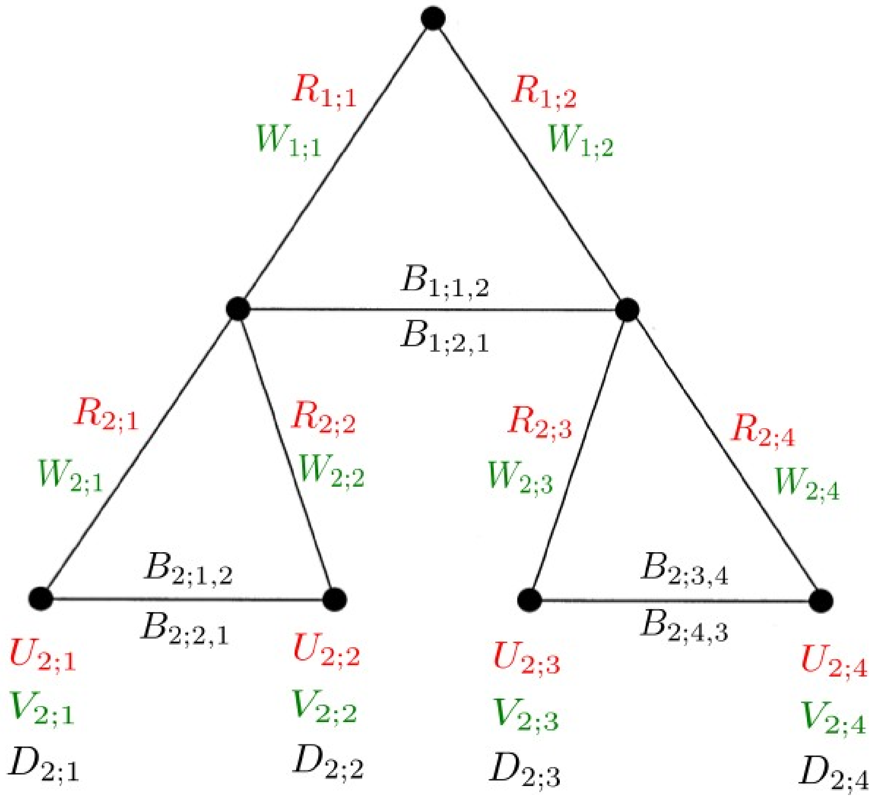
$$U_{1;2} = \begin{matrix} U_{2;3}\, R_{2;3} \\ U_{2;4}\, R_{2;4} \end{matrix} \qquad V_{1;2} = \begin{matrix} V_{2;3}\, W_{2;3} \\ V_{2;4}\, W_{2;4} \end{matrix}$$

Example 2 Level Column and Row Bases



Example 2 Level HSS Representation and Corresponding HSS Tree





Note: Notice the larger U's and V's are not stored, and do not appear in the HSS Tree

Definition - HSS Tree

- An HSS tree of a matrix is the corresponding partition tree decorated with $U_{k;i}, V_{k;i}, D_{k;i}, R_{k;i}, W_{k;i}$ and $B_{k;i,j}$.
  - The matrices $U_{k;i}, V_{k;i}, D_{k;i},$ are stored at each leaf node $(k;i)$.
  - The matrices $R_{k;i}$ and $W_{k;i}$ are stored at each edge which connects parent to child node, $(k;i)$.
  - We add edges to the partition tree from node $(k;i)$ to node $(k;j)$ corresponding to $B_{k;i,j}$.

Definition of HSS Representation

- If $(k;i)$ is a leaf node, $D_{k;i} = A_{k;i,i}$
- If $(k;i)$ is not a leaf node,

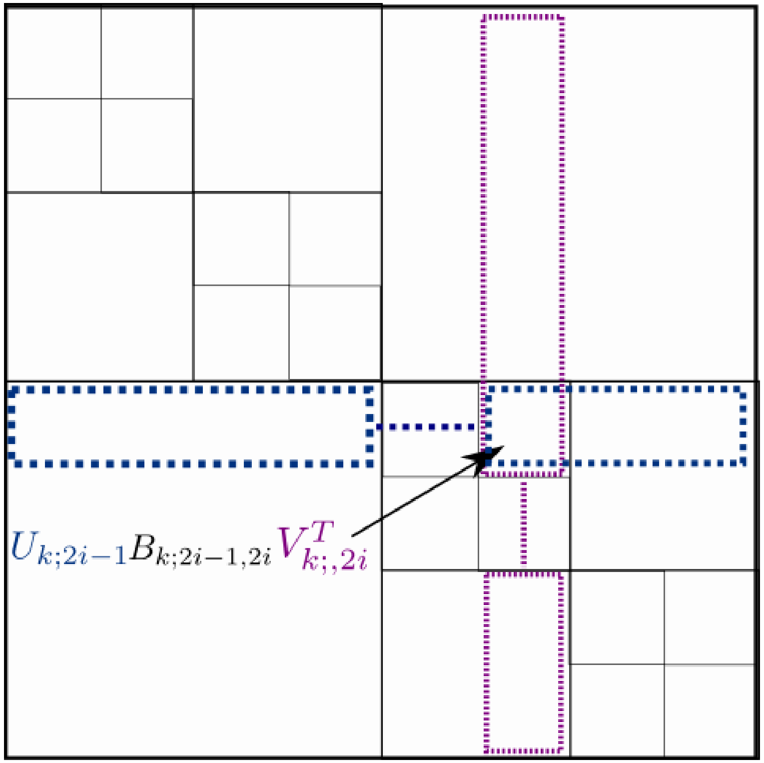
$$A_{k;2i-1,2i} = U_{k;2i-1} B_{k;2i-1;2i} V_{k;2i}^T,$$
$$A_{k;2i-1,2i} = U_{k;2i} B_{k;2i;2i-1} V_{k;2i-1}^T,$$

- Where,

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

An Inefficient Method to Compute the HSS Representation

- One obvious way to form the HSS representation of a matrix would be to take a Singular Value Decomposition (SVD) all Hankel blocks at each level of the HSS representation.
- This is extremely slow, $O(n^3)$ flops.
- Not memory efficient, $O(n^2)$ memory



A More Efficient Way to Compute the HSS Representation

- Previous HSS construction algorithms (Xia, Chandrasekeran, Martinsson) focused on speed, requiring $O(n^2)$ flops.
- It seems they were unaware they require $O(n^2)$ memory in the worst case.
- We present an HSS construction algorithm which requires $O(n^{1.5})$ peak workspace memory in the worst case, while still requiring only $O(n^2)$ flops.
- We require only $O(n \log n)$ memory for a complete tree.

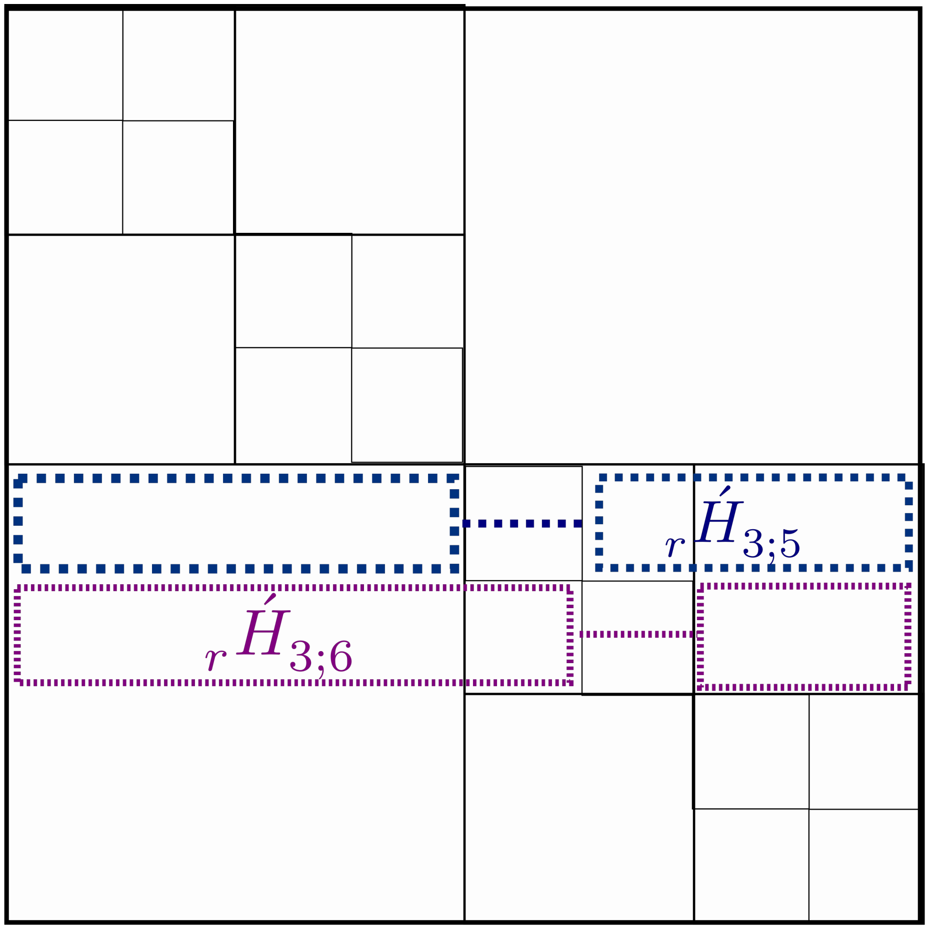
Main Points of this Talk

- Basic building blocks for $O(n^2)$ flop construction algorithm
- Peak memory consumption

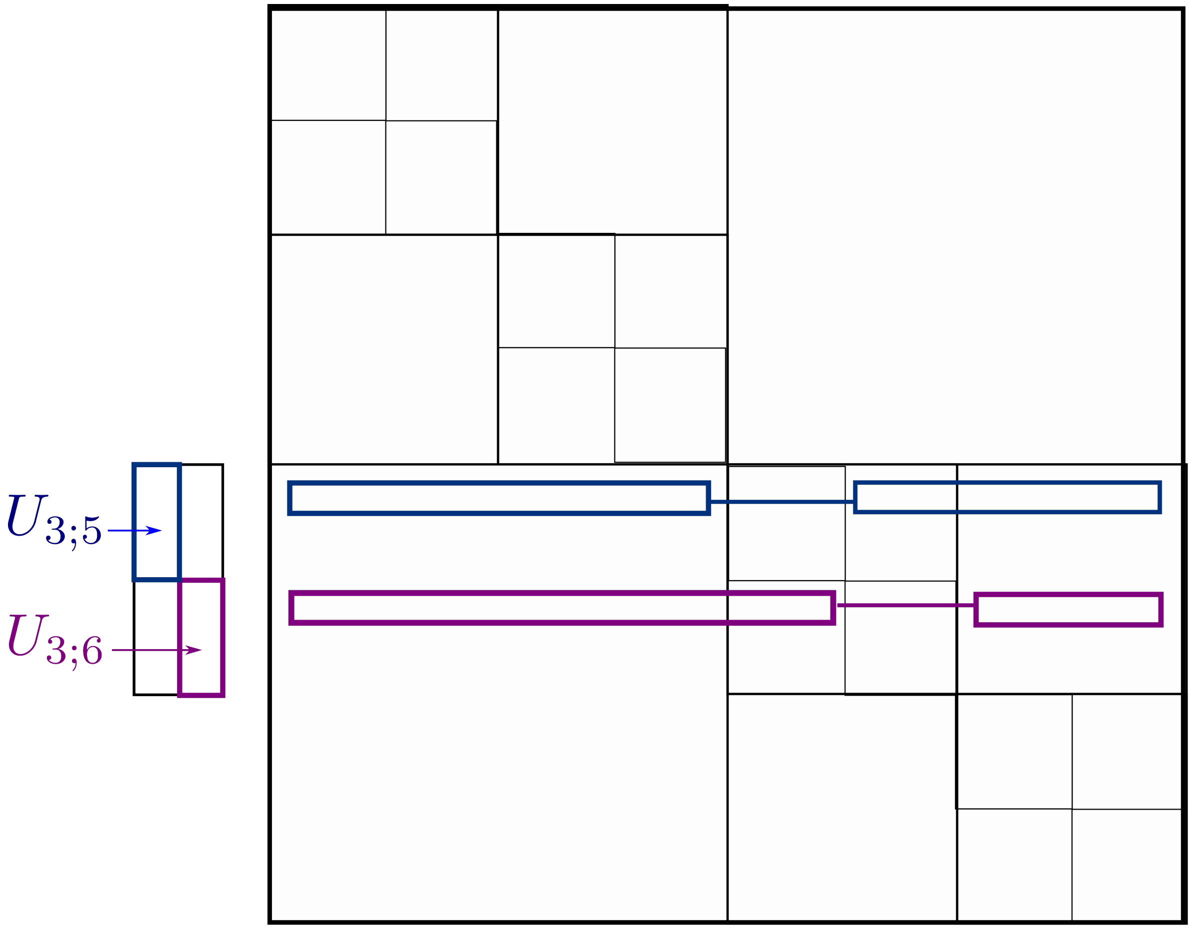
Phase 1 & Phase 2 of our Construction Algorithm

- Phase 1 – Computation of Basis Matrices, $U_{k;i}$, and $V_{k;i}$, as well as Translation Operators $R_{k;i}$ and $W_{k;i}$
- Phase 2 – Computation of Expansion Coefficients $B_{k;i,j}$

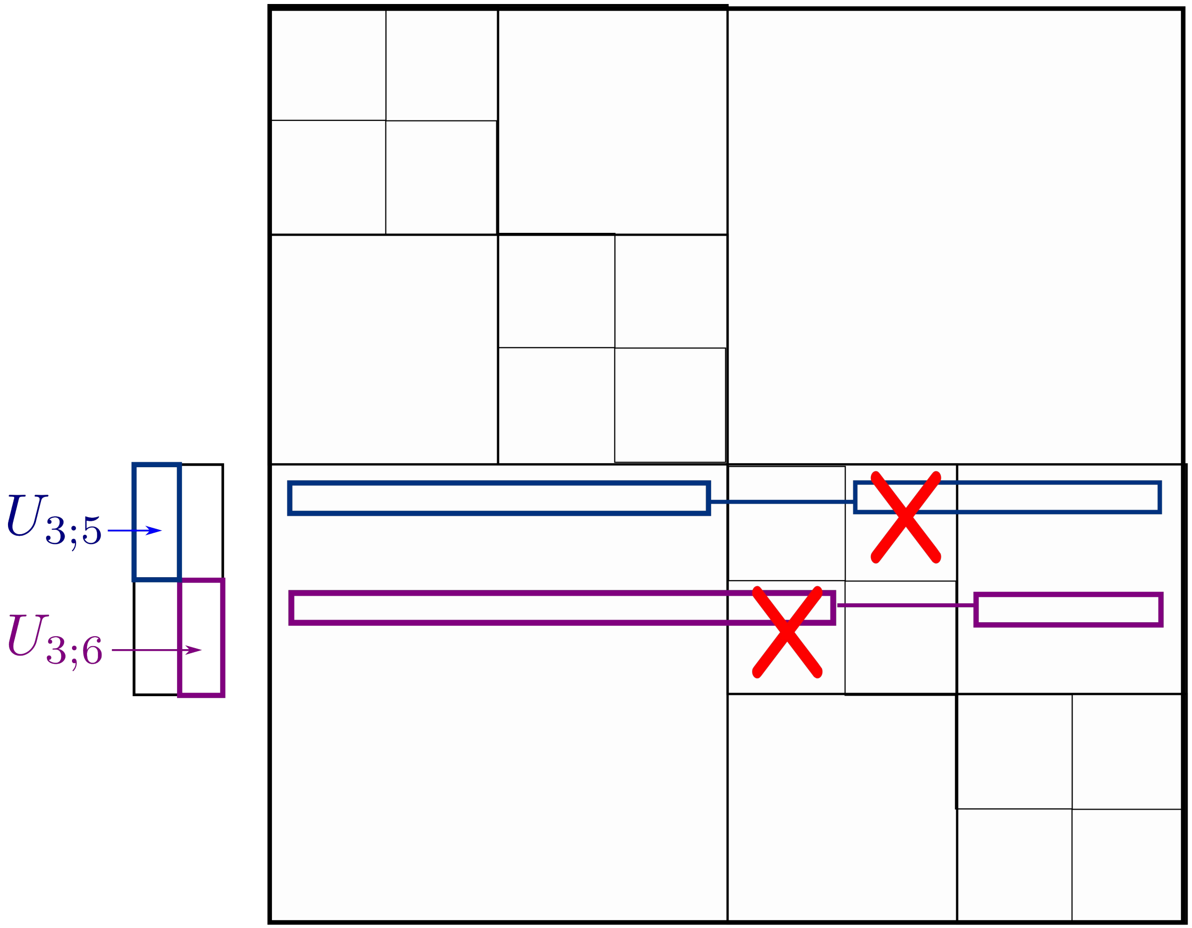
Phase 1 - Take SVD's Hankel Blocks at Leaf Nodes



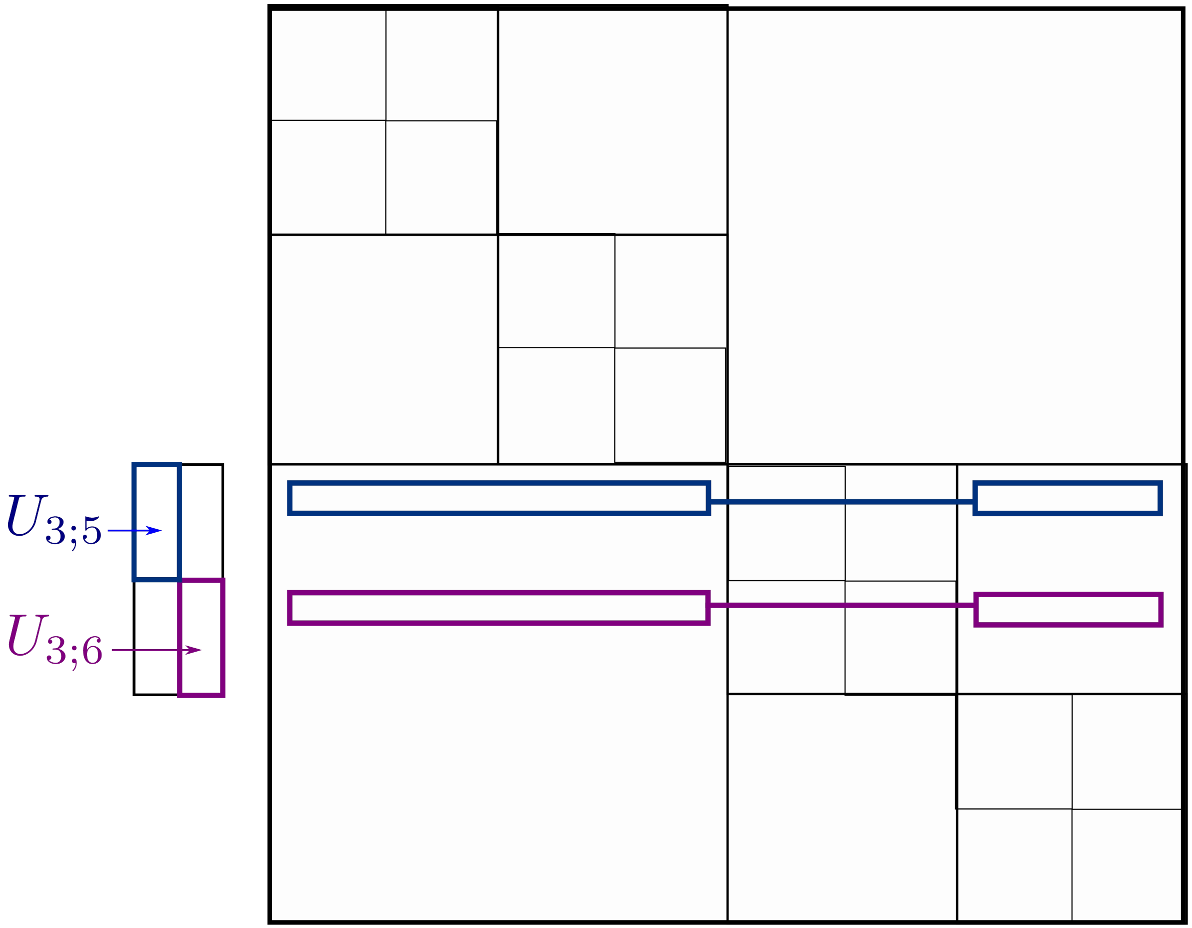
Construction Algorithm - Phase 1



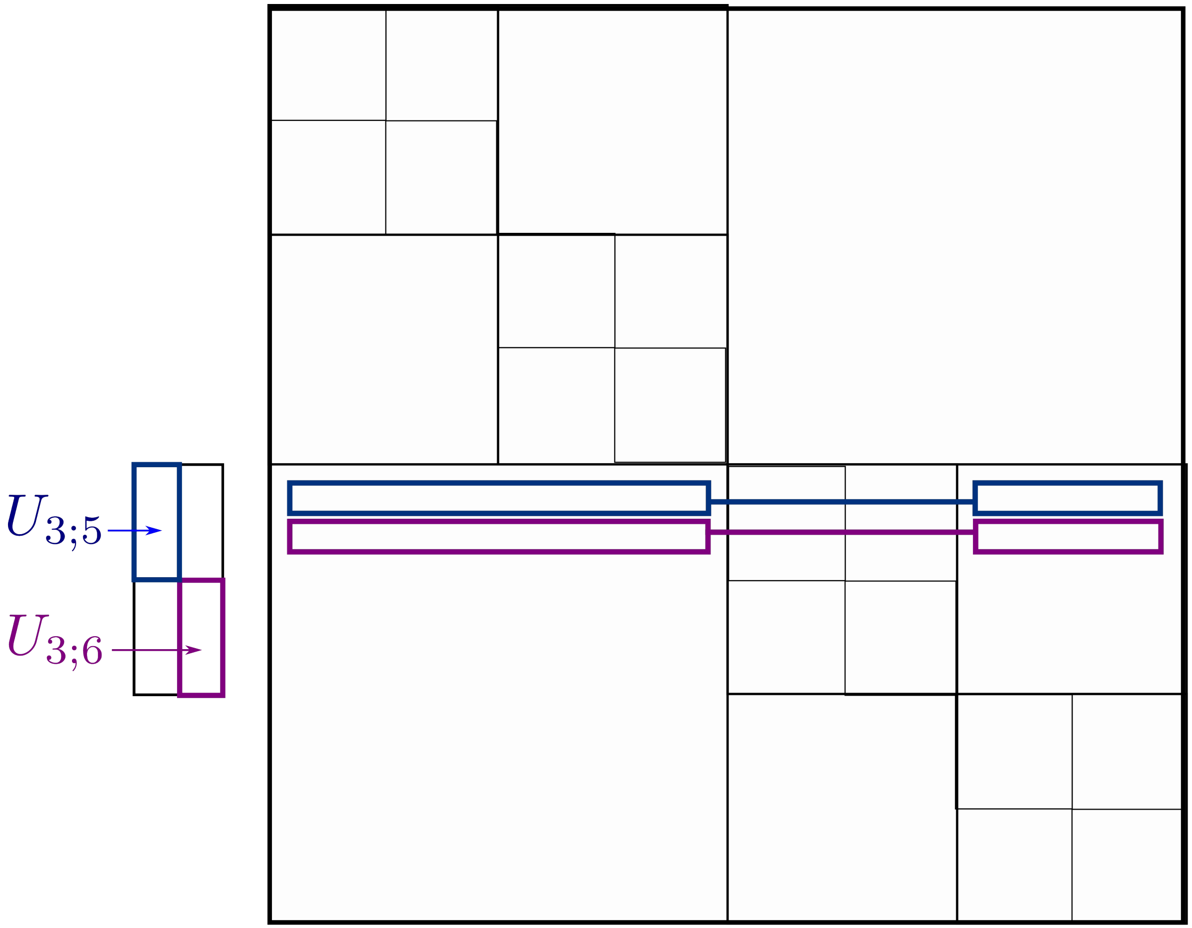
Construction Algorithm - Phase 1



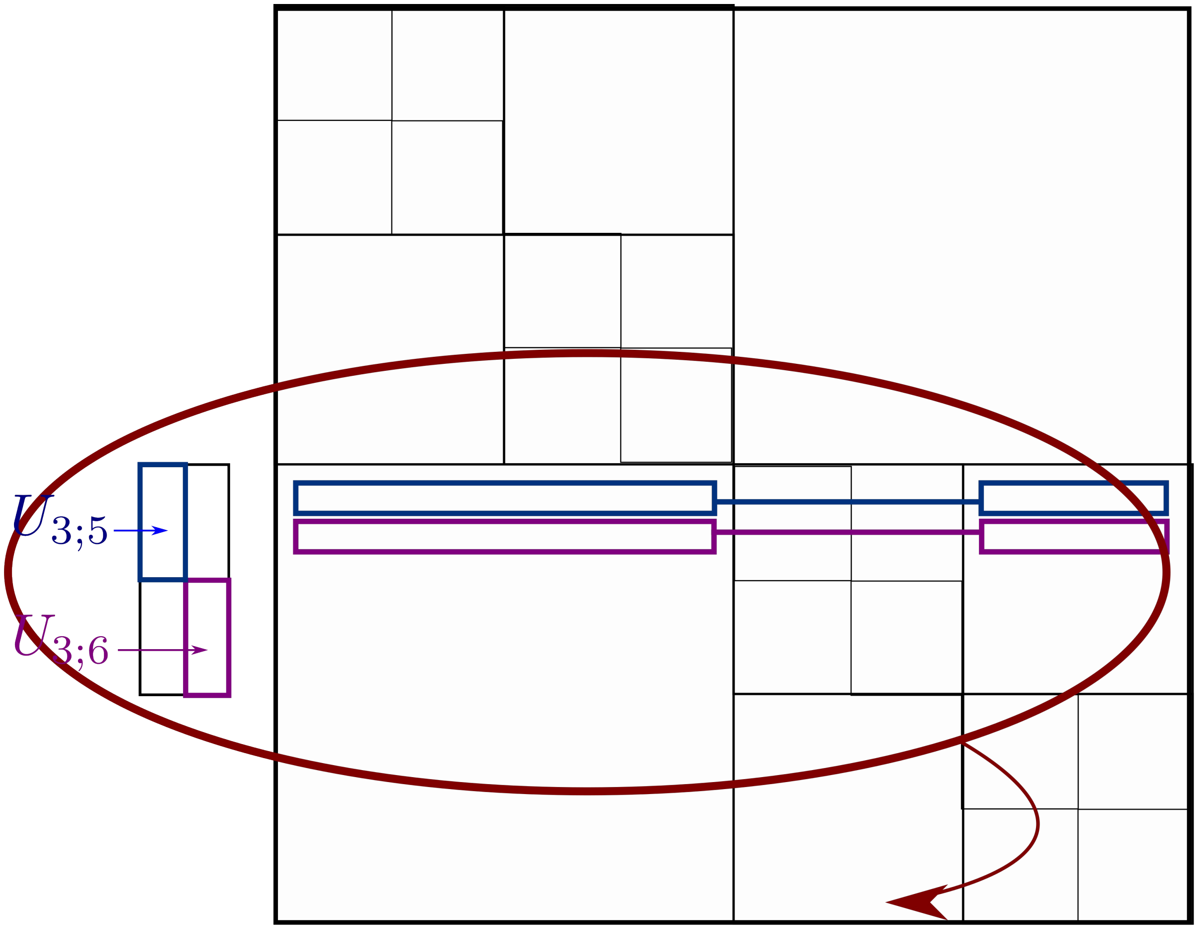
Construction Algorithm - Phase 1



Construction Algorithm - Phase 1



Construction Algorithm - Phase 1



Construction Algorithm - Phase 1

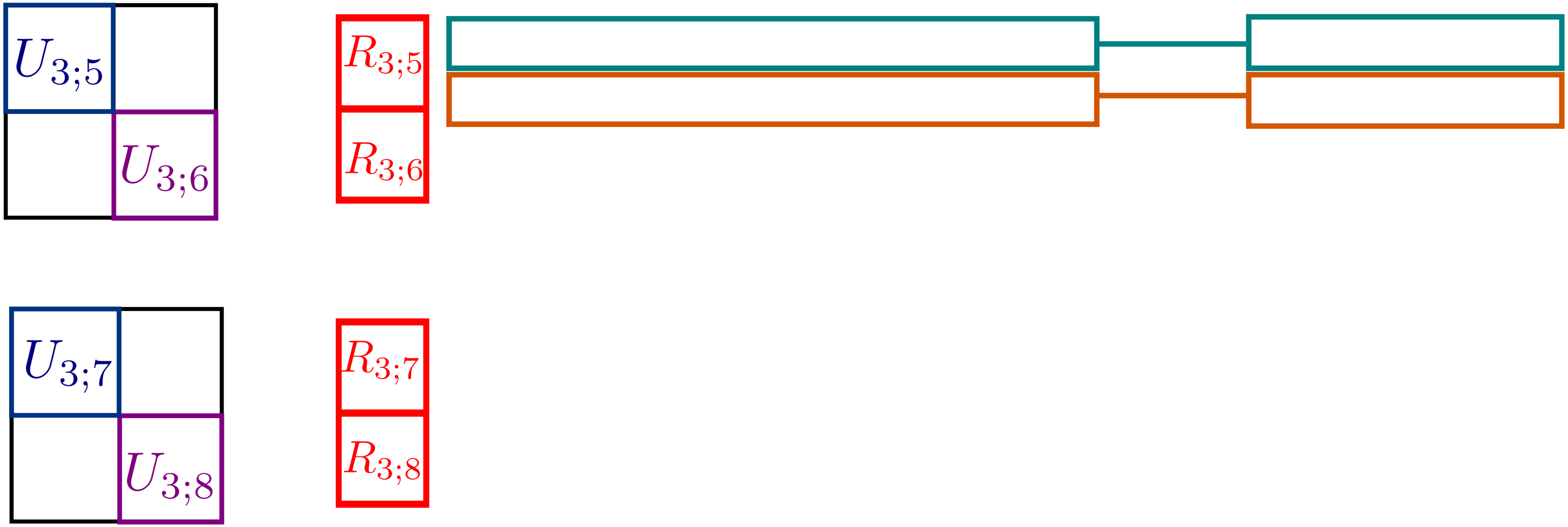
Construction Algorithm - Phase 1

Construction Algorithm - Phase 1

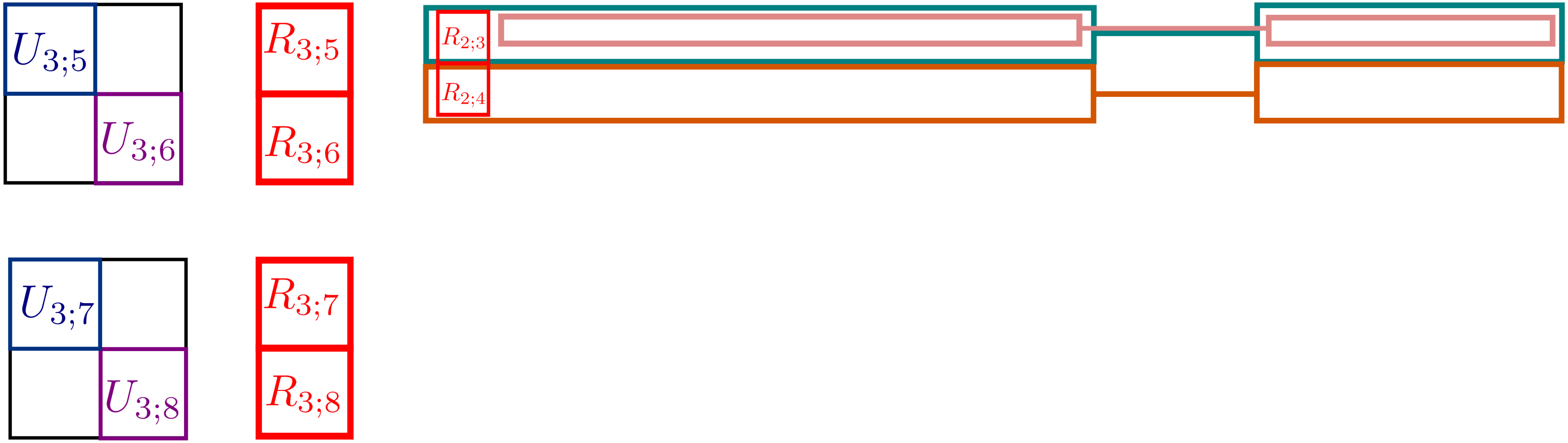

$U_{3;5}$
$U_{3;6}$
$R_{3;5}$
$R_{3;6}$


$U_{3;7}$
$U_{3;8}$
$R_{3;7}$
$R_{3;8}$

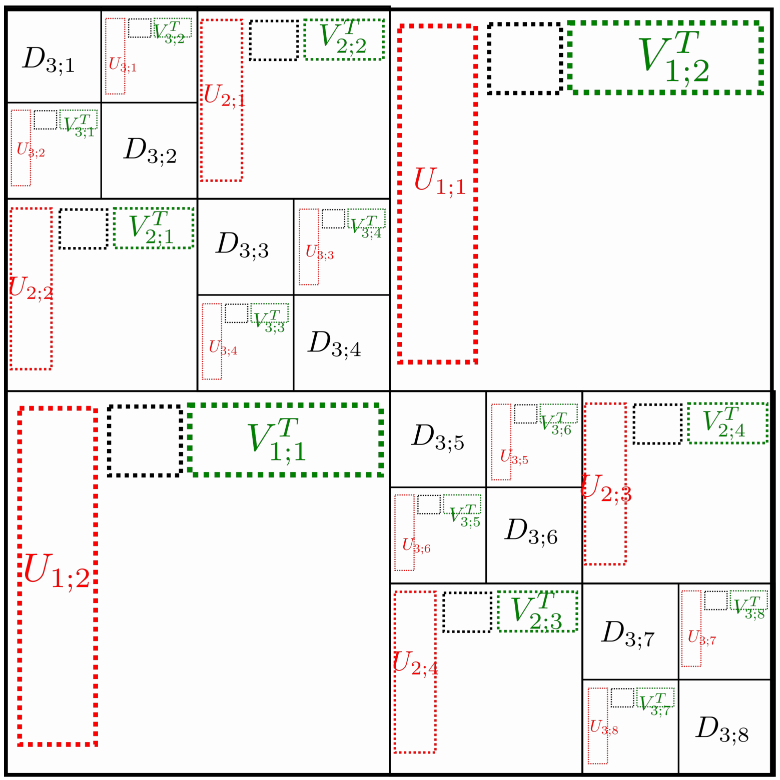
Construction Algorithm - Phase 1

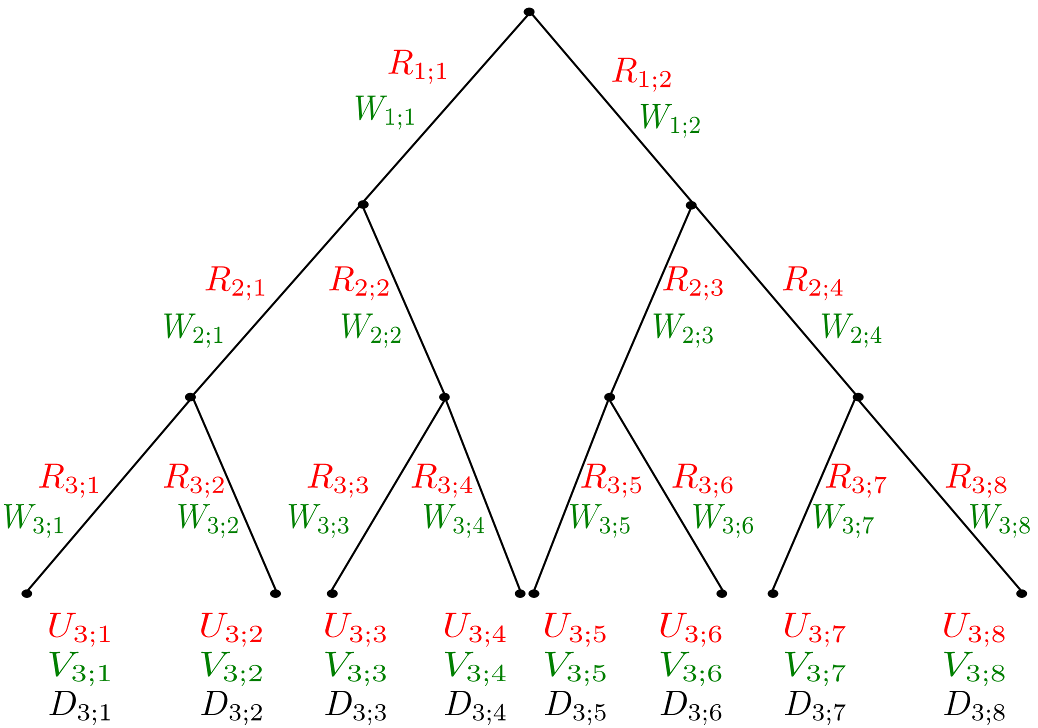


Construction Algorithm - Phase 1

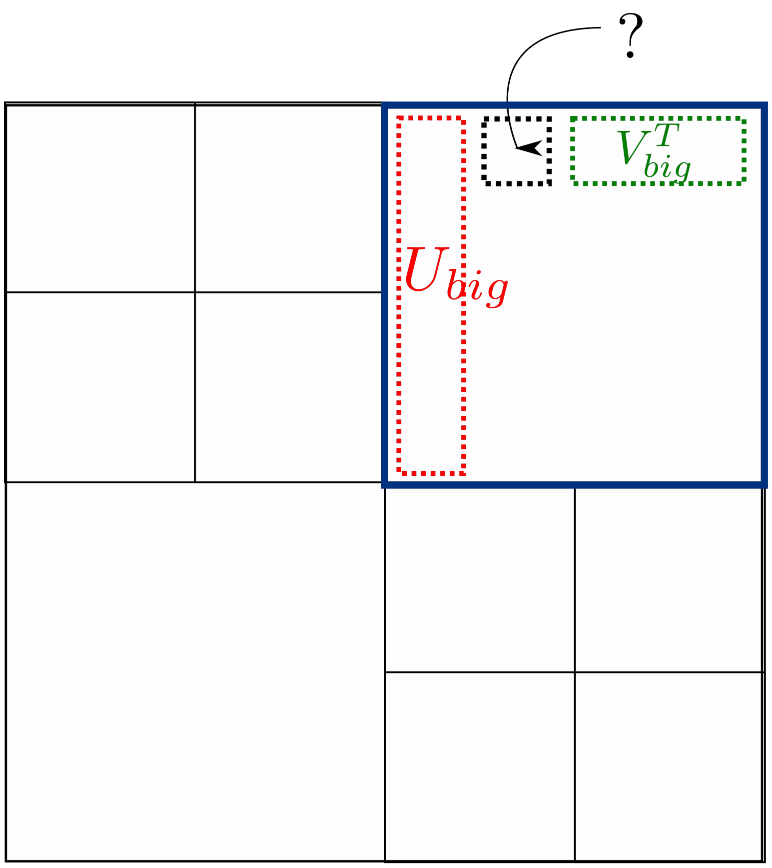


Phase 1 Complete

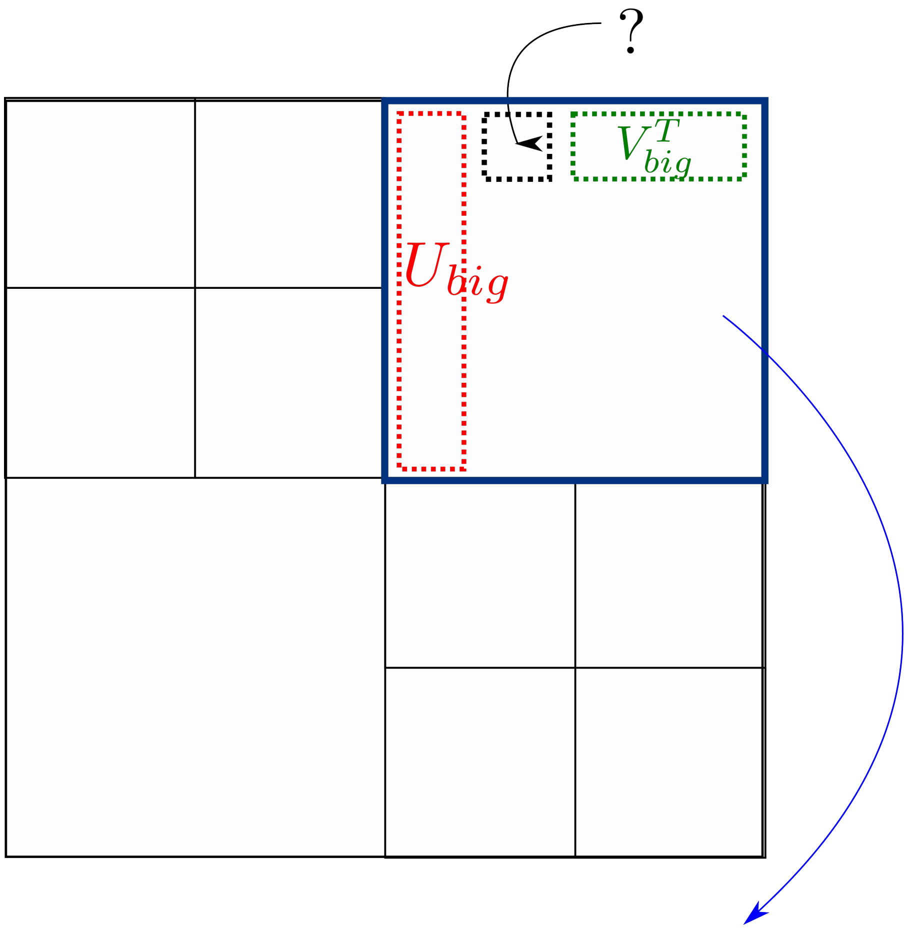




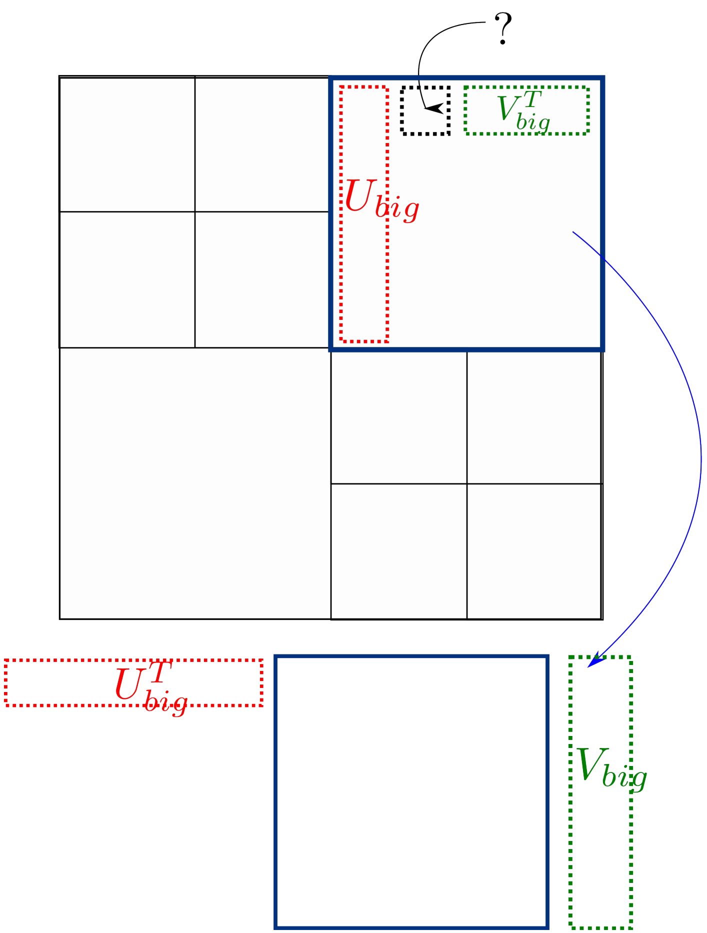
Construction Algorithm - Phase 2 – The Naive Way



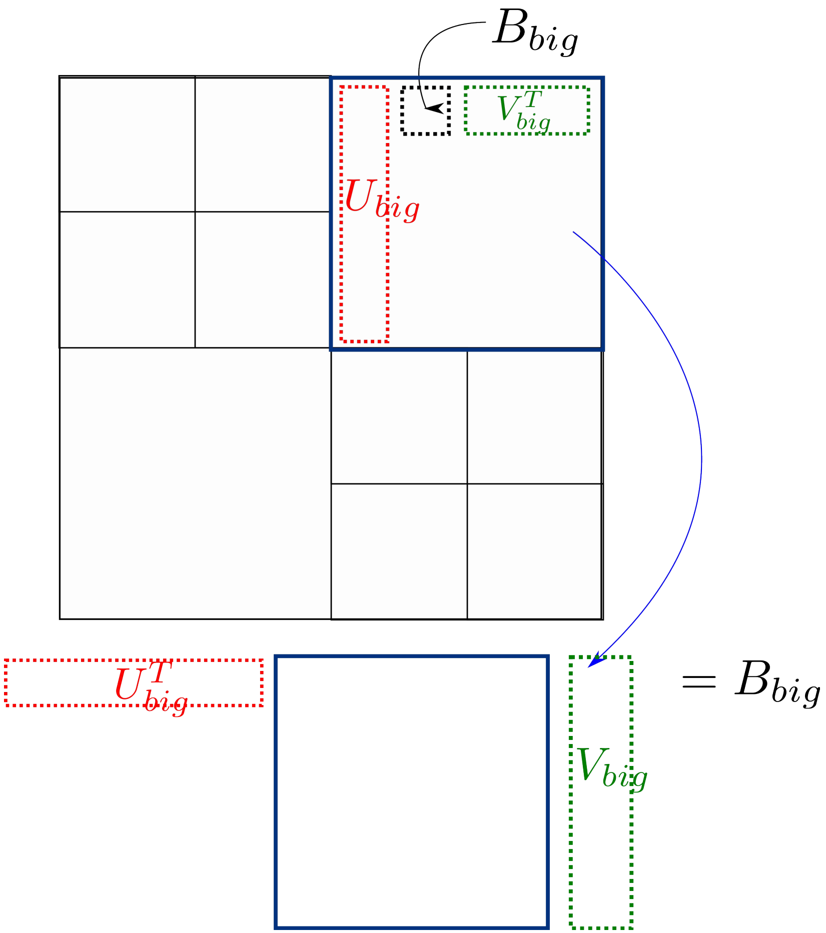
Construction Algorithm - Phase 2 – The Naive Way



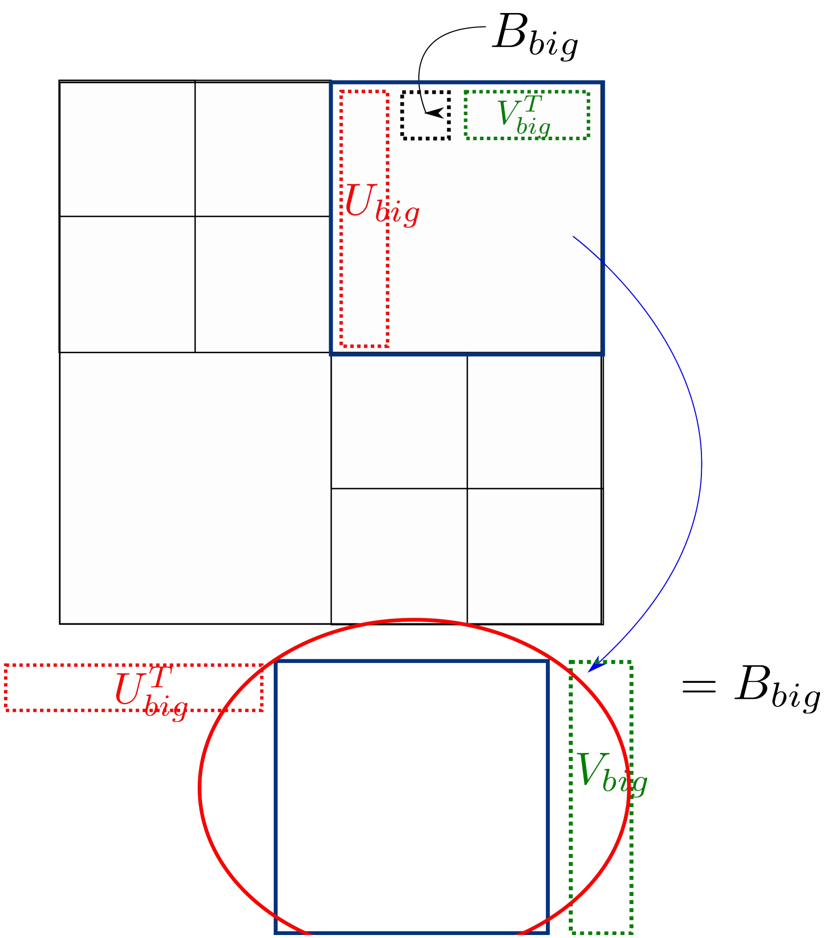
Construction Algorithm - Phase 2 – The Naive Way



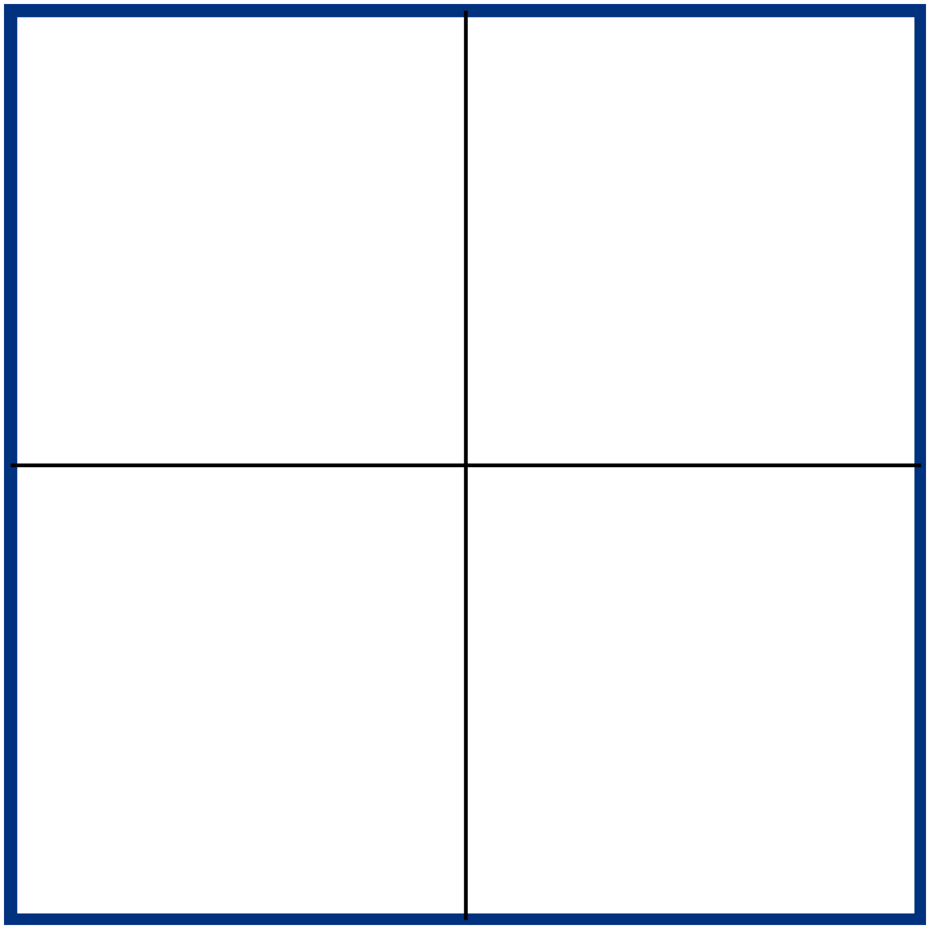
Construction Algorithm - Phase 2 – The Naive Way



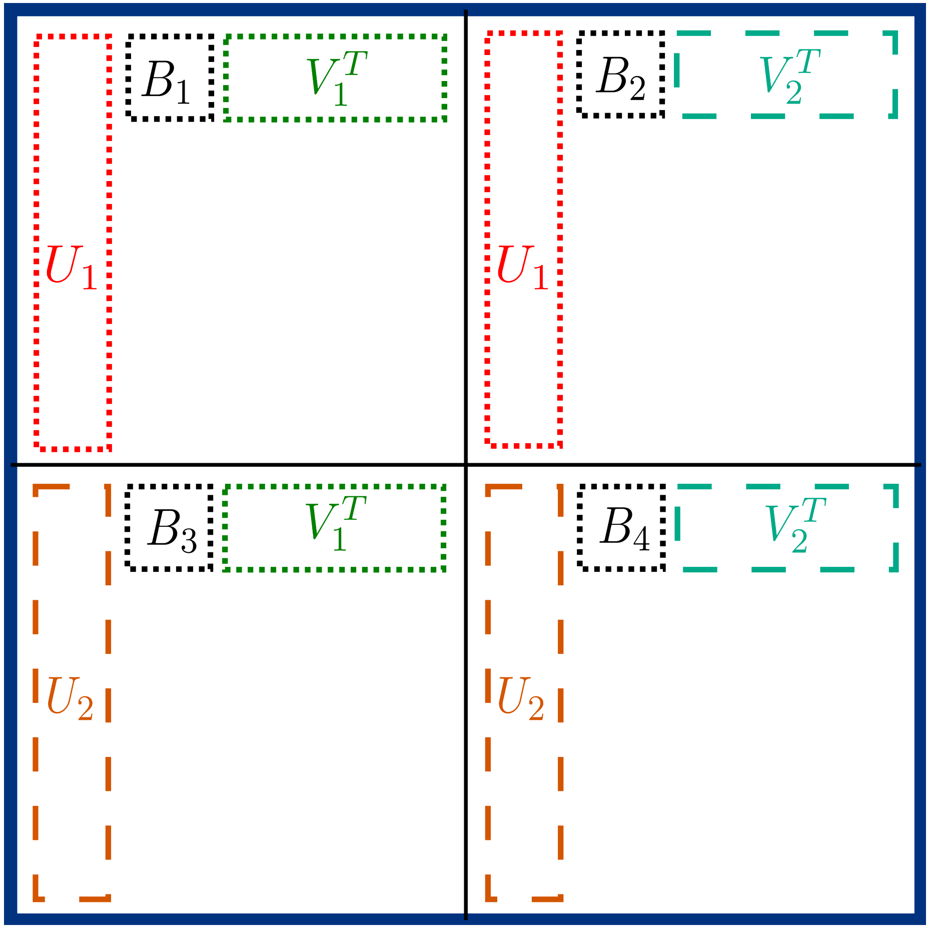
Construction Algorithm - Phase 2 – The Naive Way



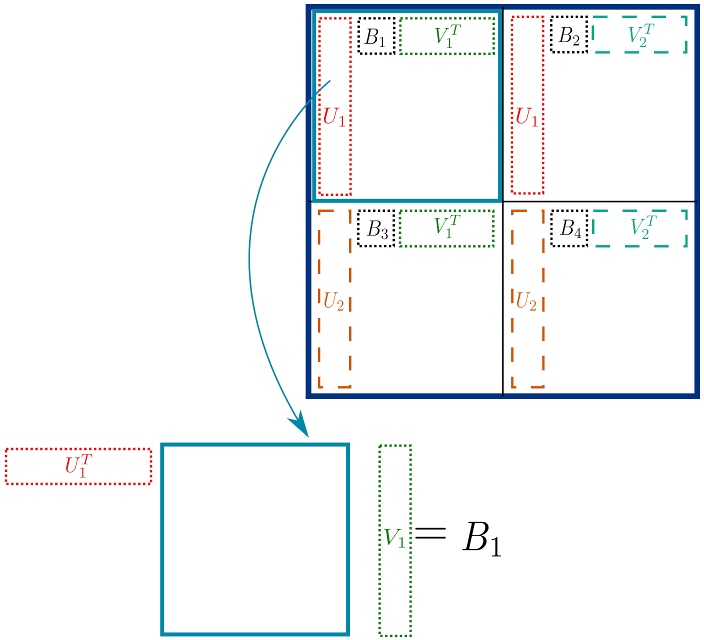
Construction Algorithm - Phase 2 – The Naive Way



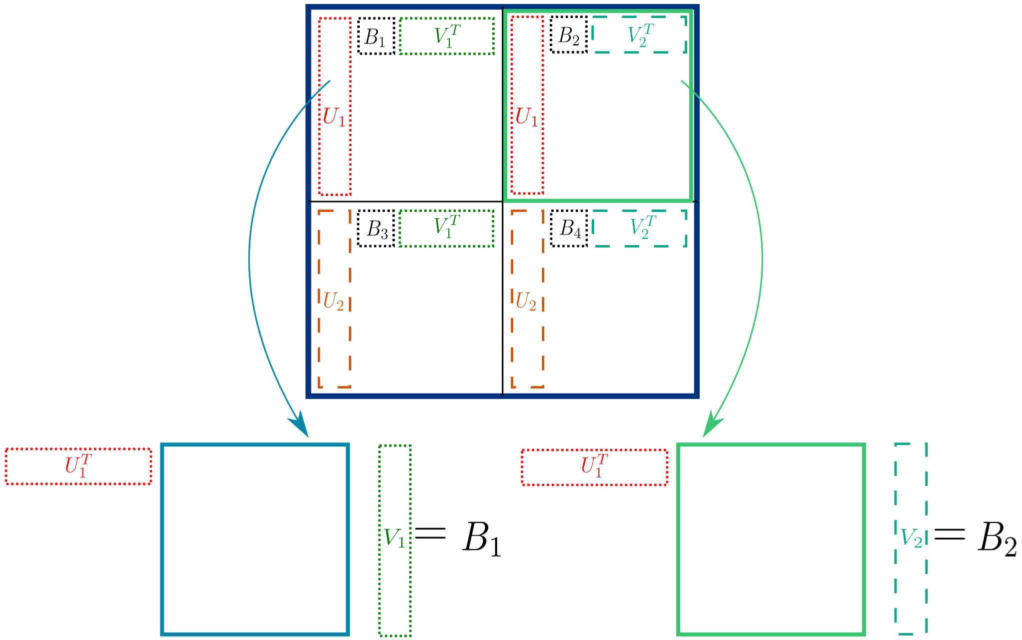
Phase 2 – A Better Way: Divide and Conquer



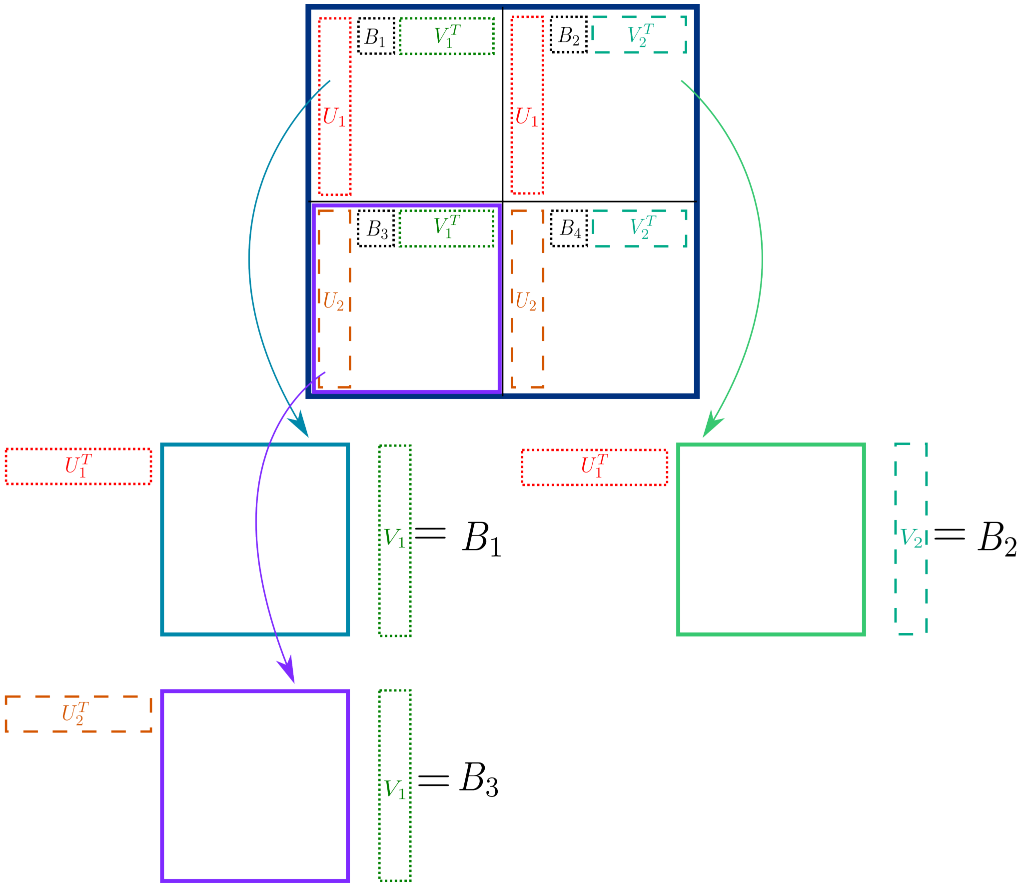
Phase 2 – A Better Way: Divide and Conquer



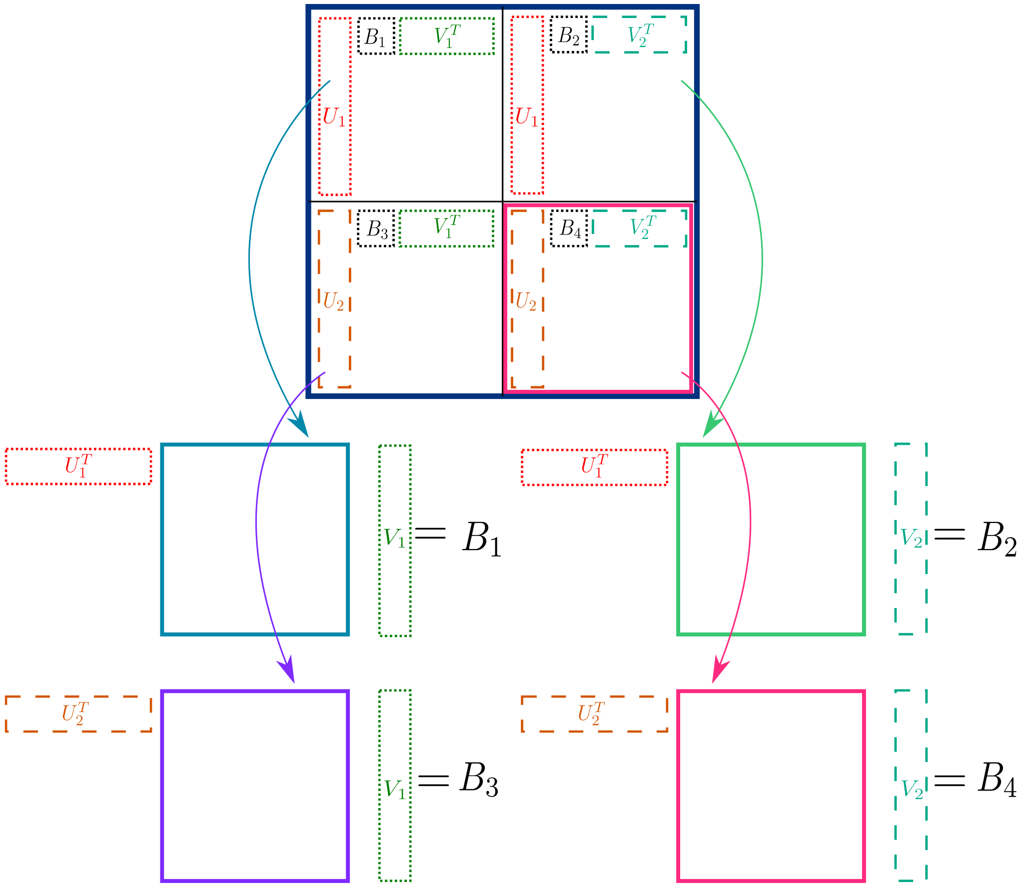
Phase 2 – A Better Way: Divide and Conquer



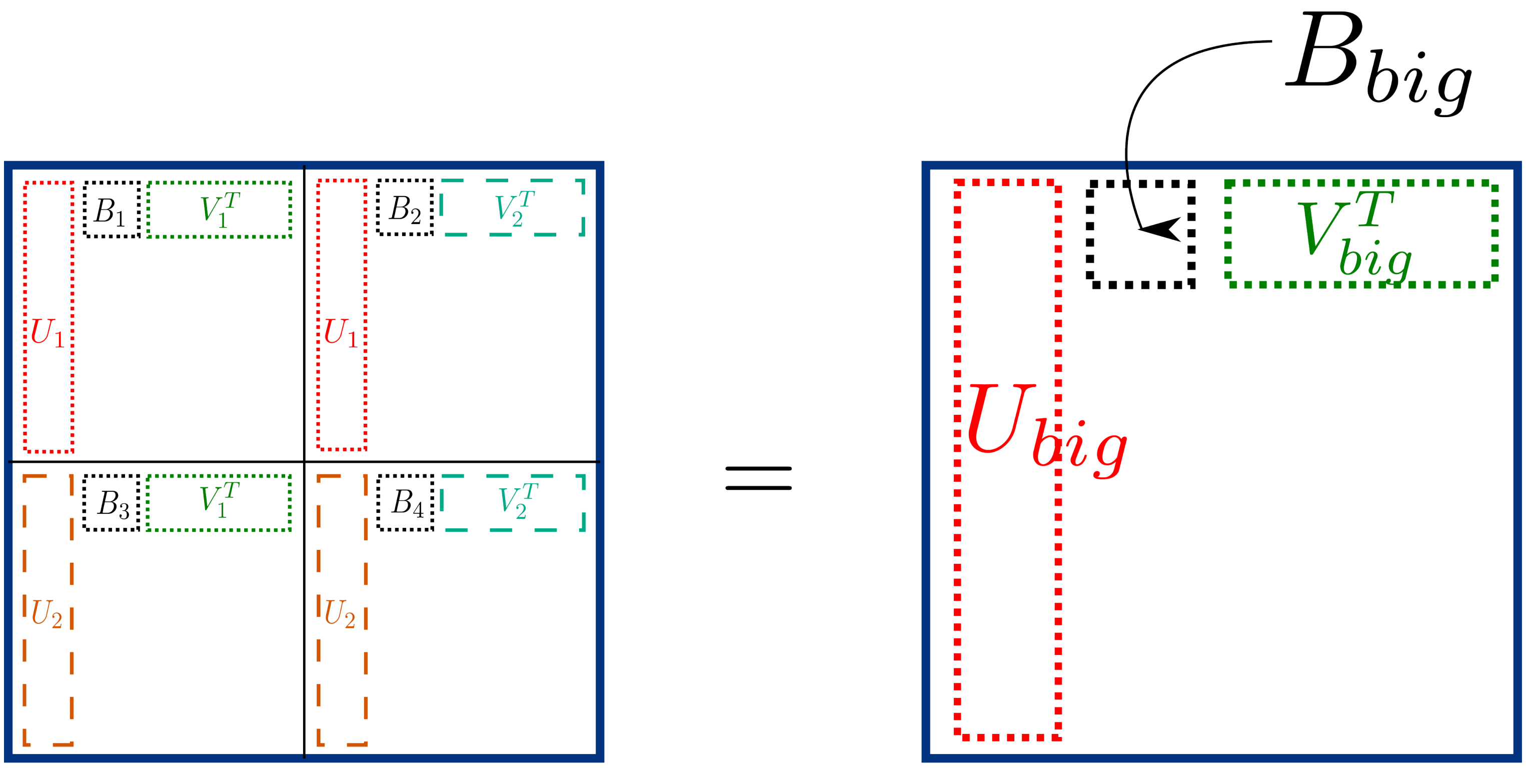
Phase 2 – A Better Way: Divide and Conquer



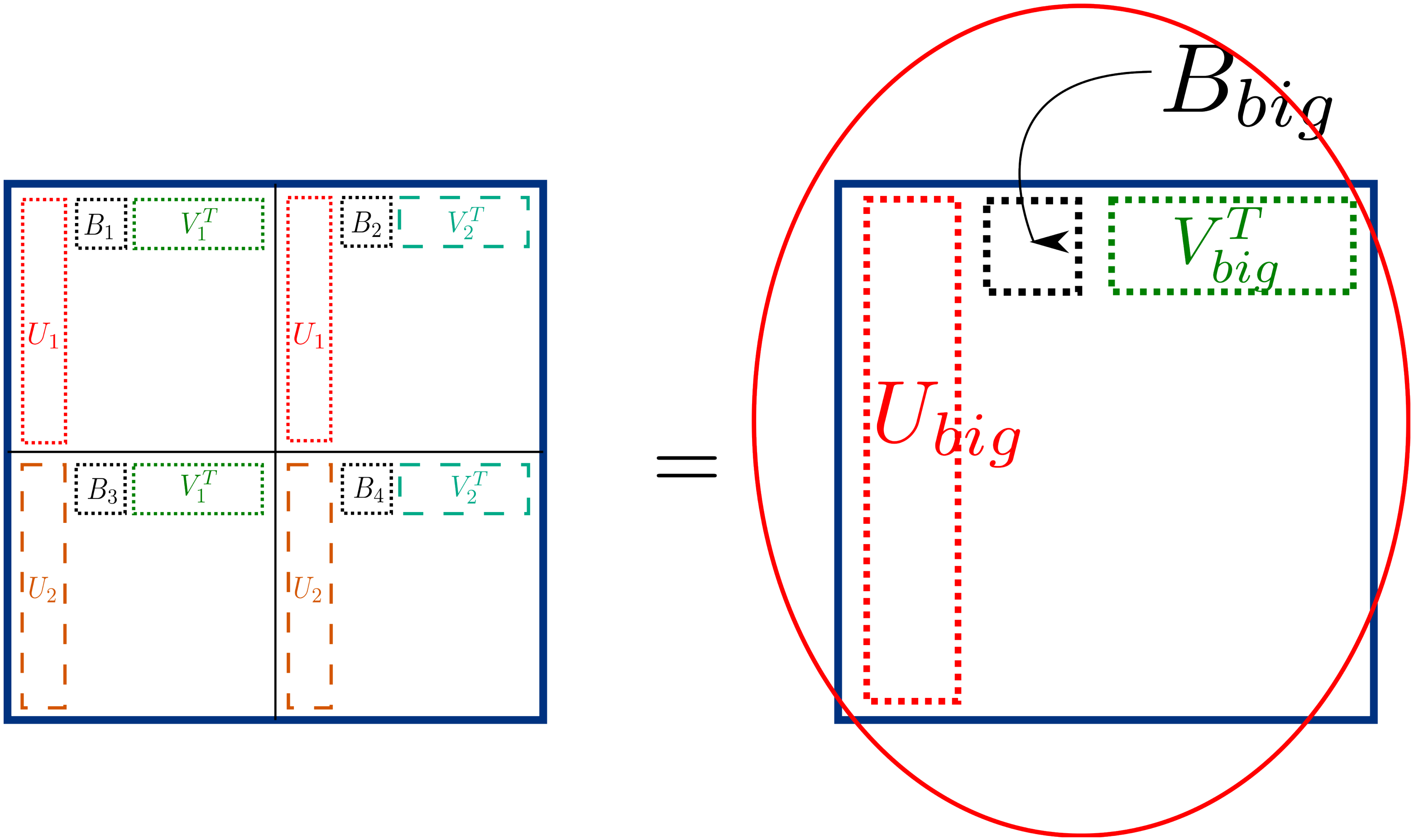
Phase 2 – A Better Way: Divide and Conquer



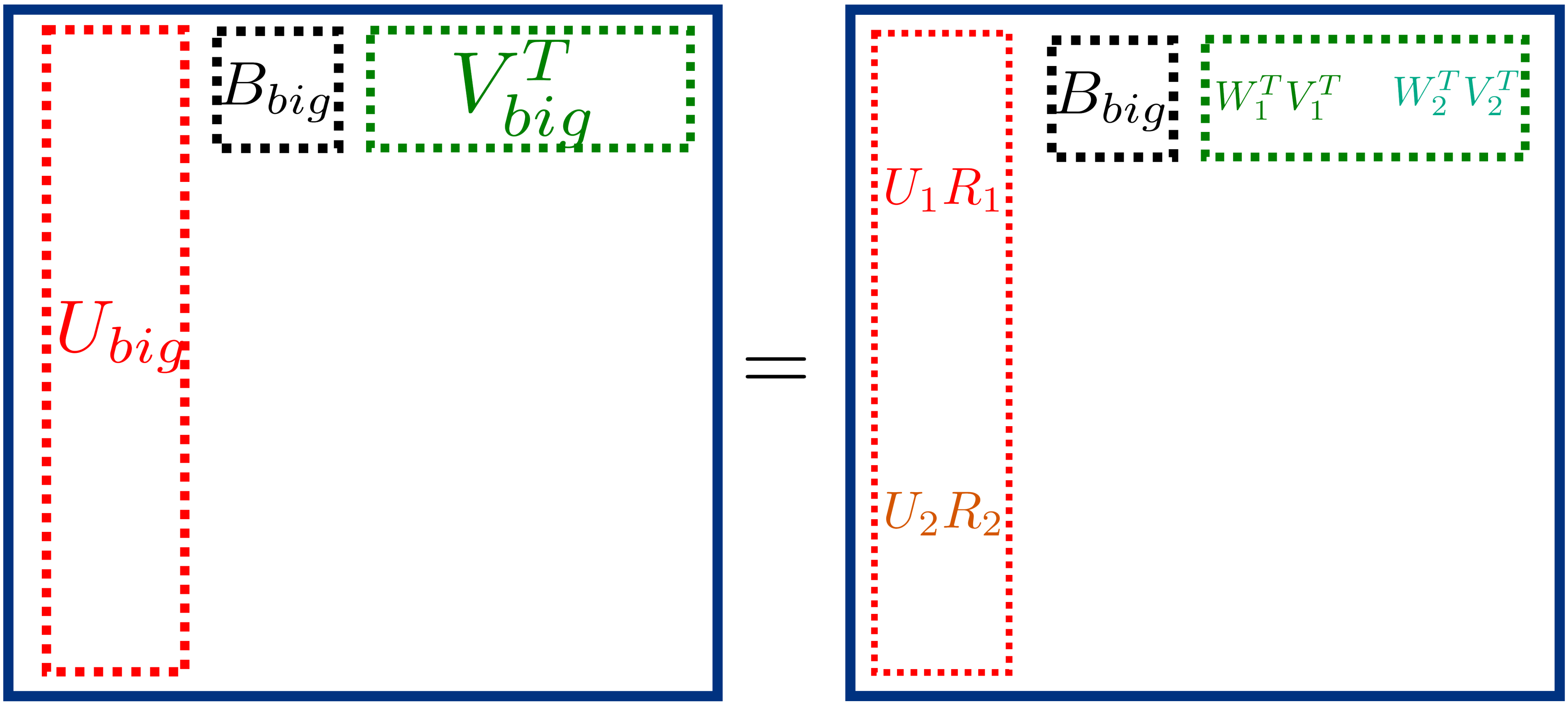
Phase 2 – A Better Way: Divide and Conquer



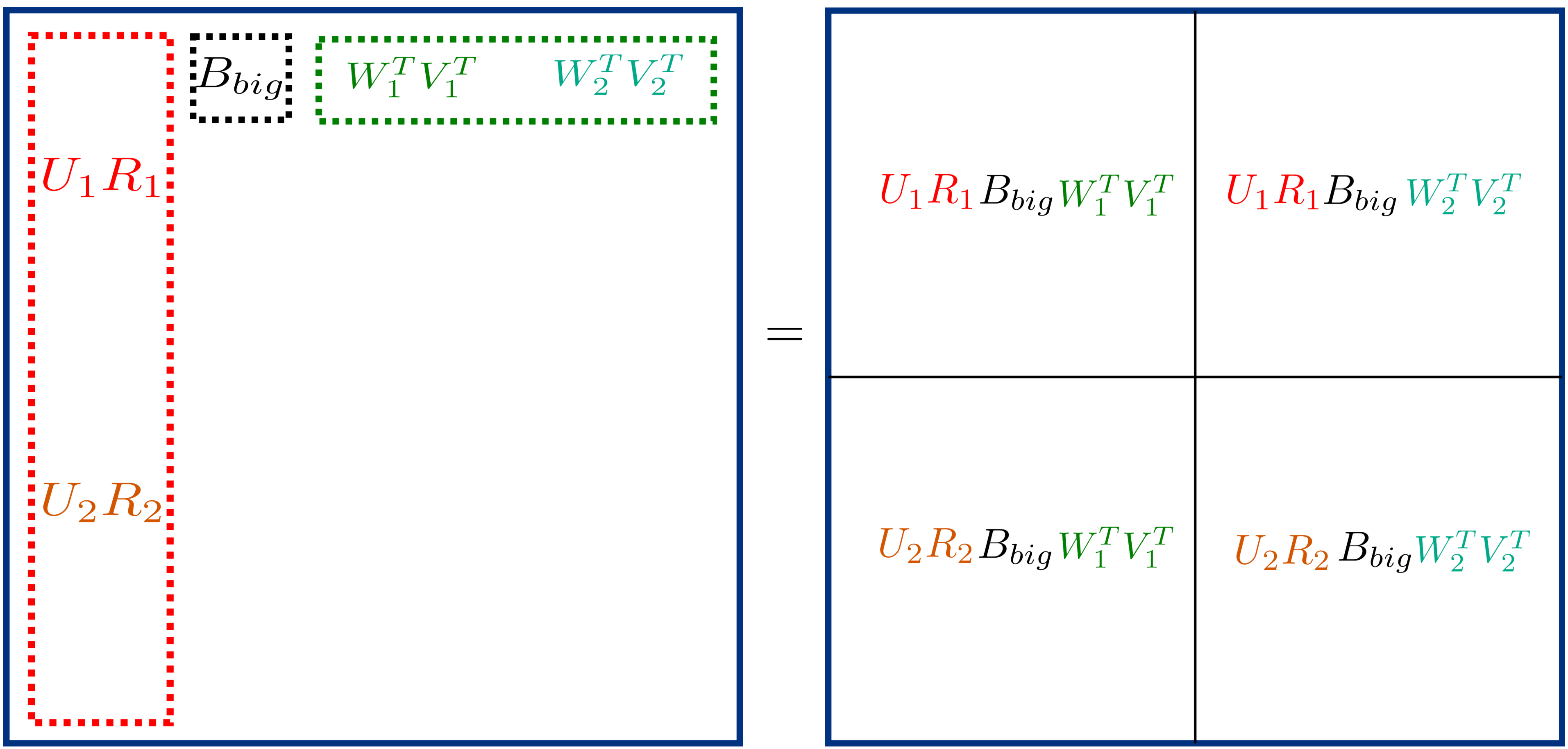
Phase 2 – A Better Way: Divide and Conquer



Phase 2 – A Better Way: Divide and Conquer

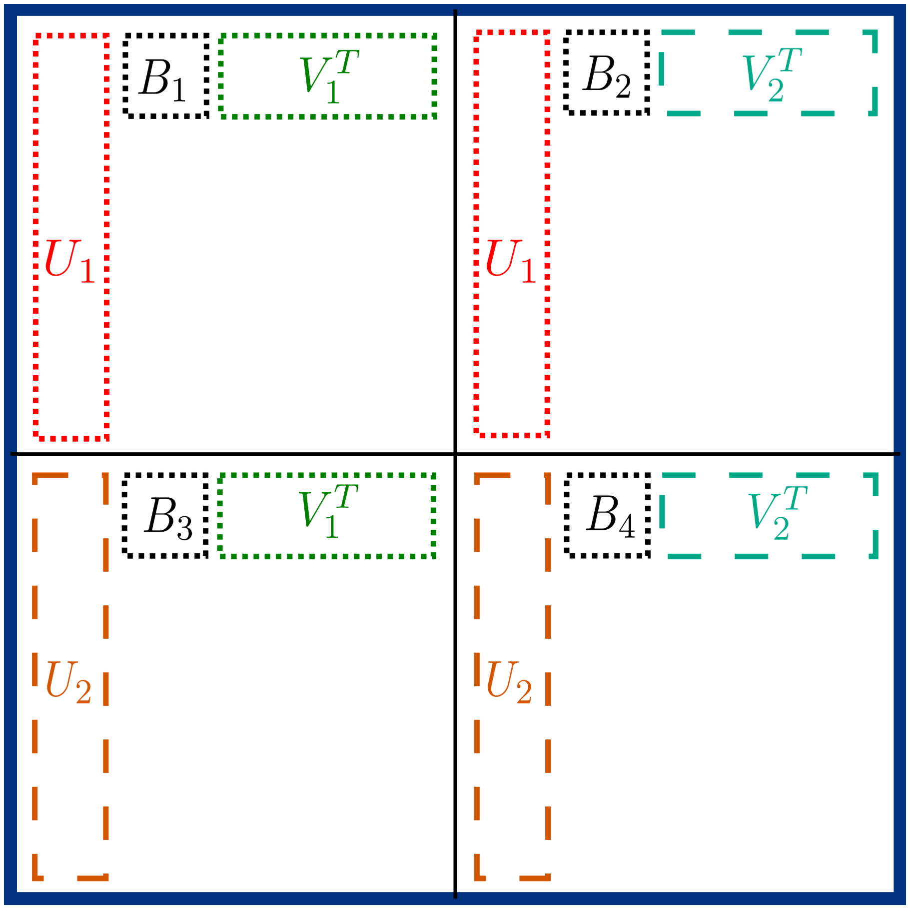


Phase 2 – A Better Way: Divide and Conquer



Phase 2 – A Better Way: Divide and Conquer

$$\left[\begin{array}{c|c} U_1 R_1 B_{big} W_1^T V_1^T & U_1 R_1 B_{big} W_2^T V_2^T \\ \hline U_2 R_2 B_{big} W_1^T V_1^T & U_2 R_2 B_{big} W_2^T V_2^T \end{array}\right] =$$



Phase 2 – A Better Way: Divide and Conquer

$$\left[\begin{array}{c|c} U_1 R_1 B_{big} W_1^T V_1^T & U_1 R_1 B_{big} W_2^T V_2^T \\ \hline U_2 R_2 B_{big} W_1^T V_1^T & U_2 R_2 B_{big} W_2^T V_2^T \end{array}\right] = \left[\begin{array}{c|c} U_1 B_1 V_1^T & U_1 B_2 V_2^T \\ \hline U_2 B_3 V_1^T & U_2 B_4 V_2^T \end{array}\right]$$

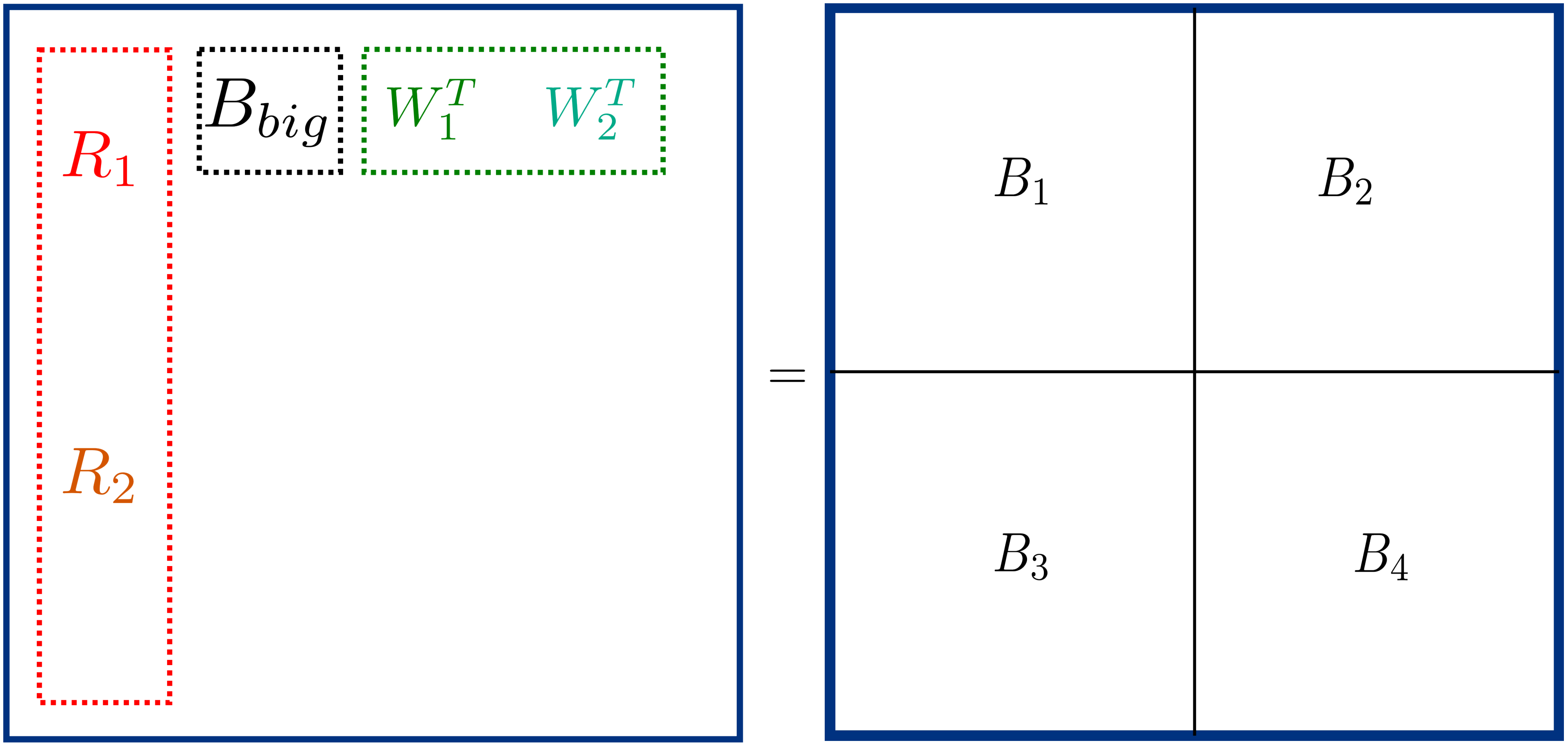
Phase 2 – A Better Way: Divide and Conquer

$$\left[\begin{array}{c|c} R_1 B_{big} W_1^T V_1^T & R_1 B_{big} W_2^T V_2^T \\ \hline R_2 B_{big} W_1^T V_1^T & R_2 B_{big} W_2^T V_2^T \end{array}\right] = \left[\begin{array}{c|c} B_1 V_1^T & B_2 V_2^T \\ \hline B_3 V_1^T & B_4 V_2^T \end{array}\right]$$

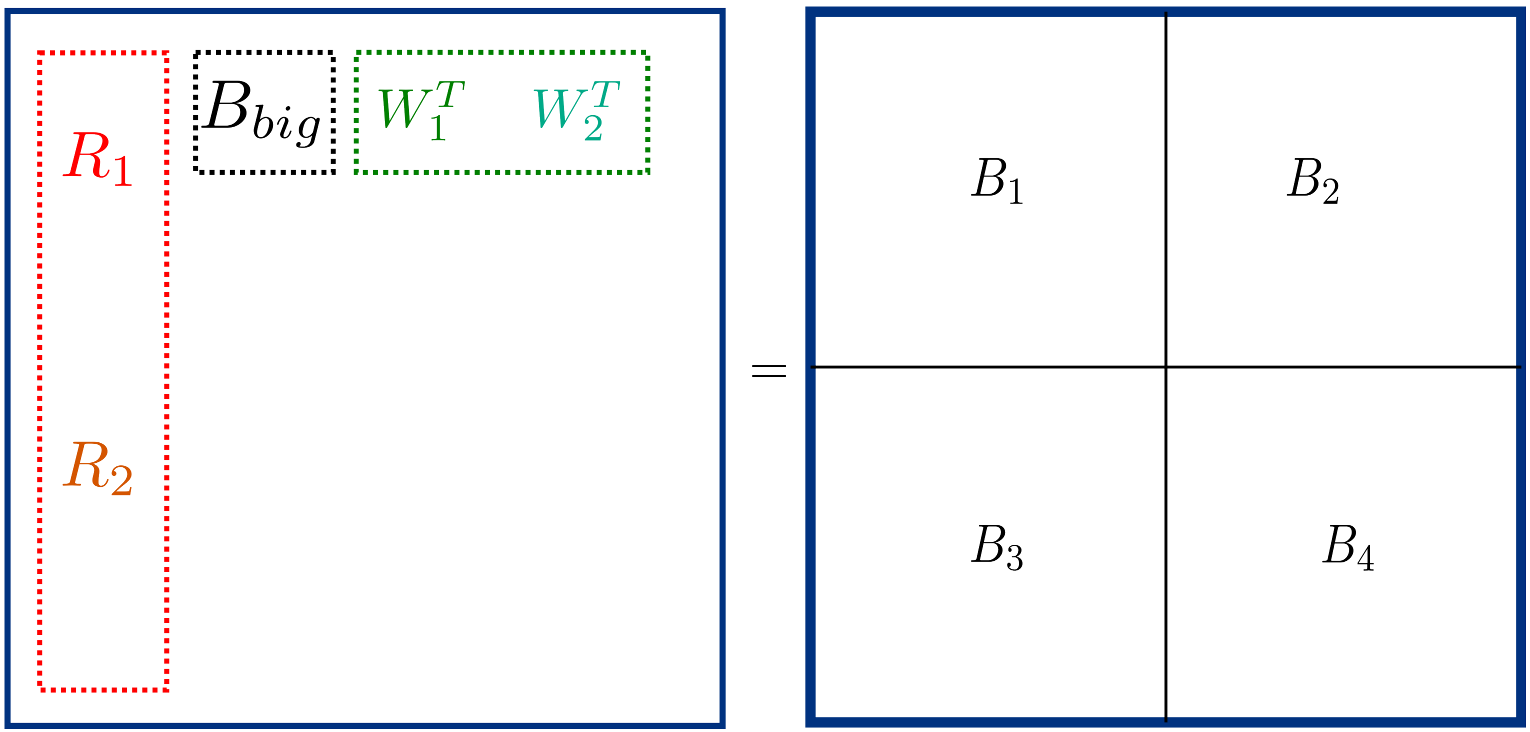
Phase 2 – A Better Way: Divide and Conquer

$$\left[\begin{array}{c|c} R_1 B_{big} W_1^T & R_1 B_{big} W_2^T \\ \hline R_2 B_{big} W_1^T & R_2 B_{big} W_2^T \end{array}\right] = \left[\begin{array}{c|c} B_1 & B_2 \\ \hline B_3 & B_4 \end{array}\right]$$

Phase 2 – A Better Way: Divide and Conquer



Phase 2 – A Better Way: Divide and Conquer



$$\begin{bmatrix} R_1^T & R_2^T \end{bmatrix} \begin{bmatrix} R_1 \\ R_2 \end{bmatrix} = I$$

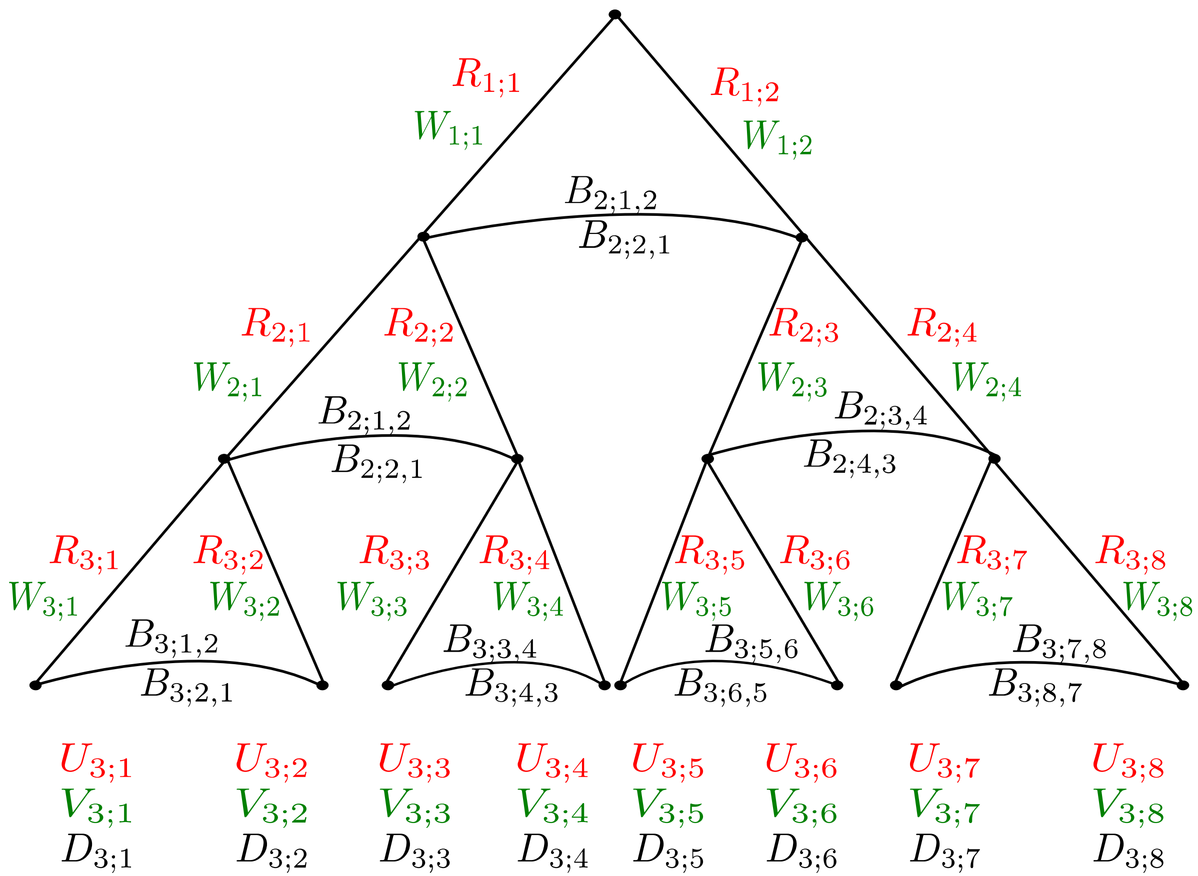
$$\begin{bmatrix} W_1^T & W_2^T \end{bmatrix} \begin{bmatrix} W_1 \\ W_2 \end{bmatrix} = I$$

Phase 2 – A Better Way: Divide and Conquer

$$B_{big} = \begin{array}{cc} & \begin{array}{cc} R_1^T & R_2^T \end{array} \\ & \left[\begin{array}{c|c} B_1 & B_2 \\ \hline B_3 & B_4 \end{array}\right] \begin{array}{c} W_1 \\ W_2 \end{array} \end{array}$$

Phase 1 & 2 Complete

- We have calculated every $U_{k;i}$,$V_{k;i}$, $R_{k;i}$,$W_{k;i}$ and $B_{k;i,j}$ in the HSS Representation



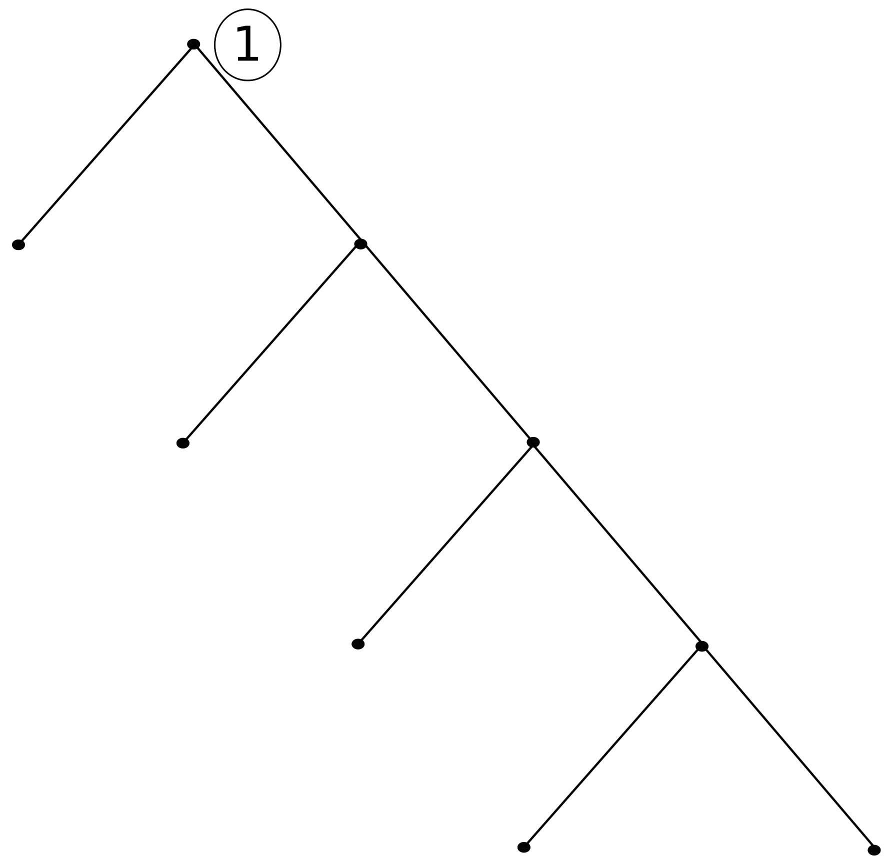
Algorithm Memory Consumption

- Other algorithms can take as much as $O(n^2)$ memory due to a depth first traversal of the HSS tree
- Our algorithm traverses the tree in a deepest first order instead, and takes $O(p^{0.5}n^{1.5})$ memory in the worst case, where p is the rank of the off diagonal blocks of the matrix A, while still taking only $O(n^2p)$ flops

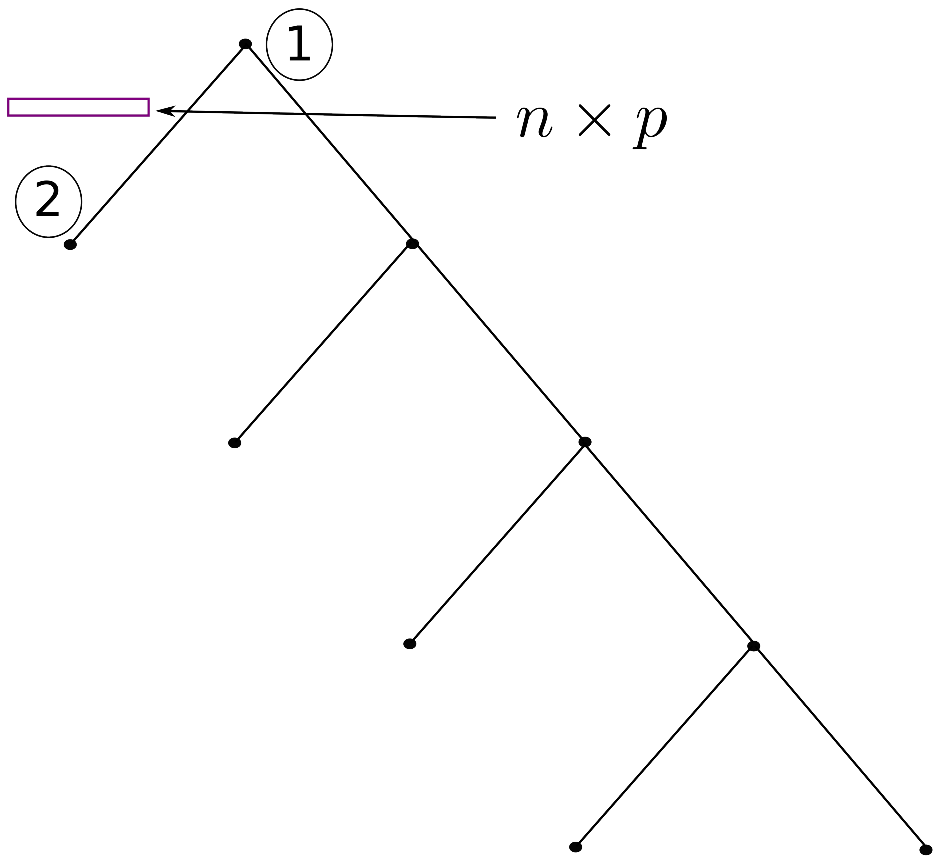
What is the Worst Case Memory Consumption for Our Algorithm?

- Phase 2 of our algorithm (computation of Expansion Coefficients $B_{k;i,j}$) consumes at most $O(pn)$ memory
  - One $p \times p$ block is stored in memory for each recursive call.
  - Tree of max depth is $O(n/p)$
  - This implies $O(pn)$ peak memory consumption for a tree of maximal depth
- We need to focus on Phase 1 (computation of basis matrices, $U_{k;i}$ and $V_{k;i}$, and translation operators $R_{k;i}$ and $W_{k;i}$) of our algorithm in order to determine peak workspace consumption

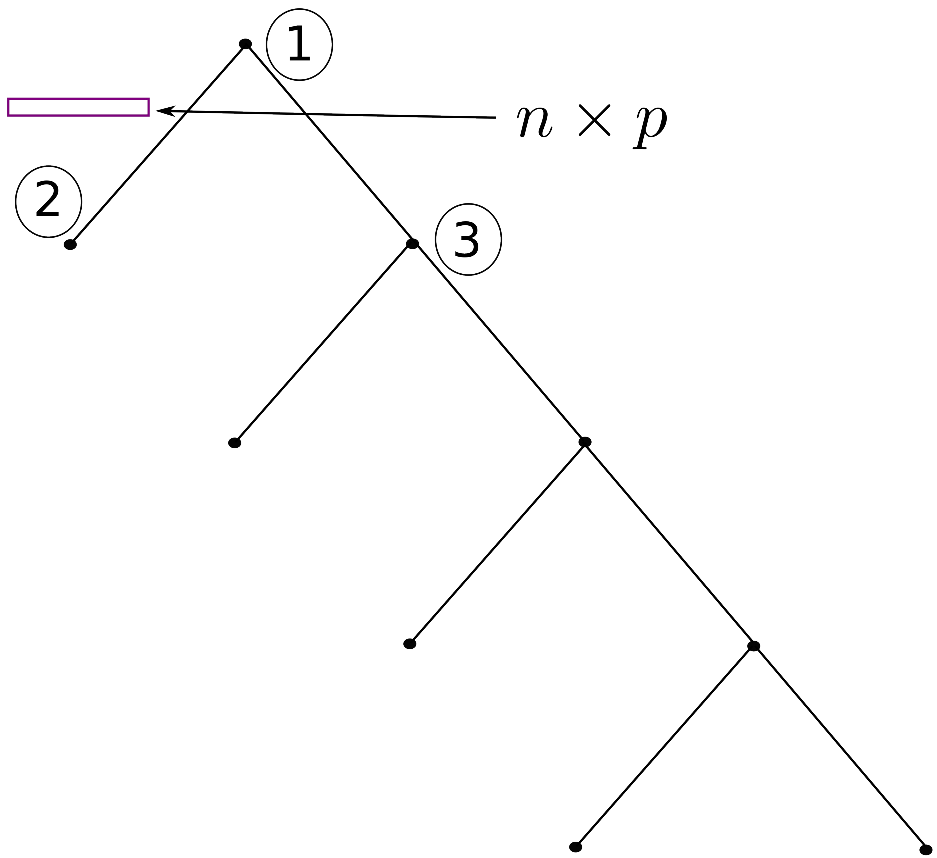
Depth First Traversal is Not Optimal for Peak Memory Consumption



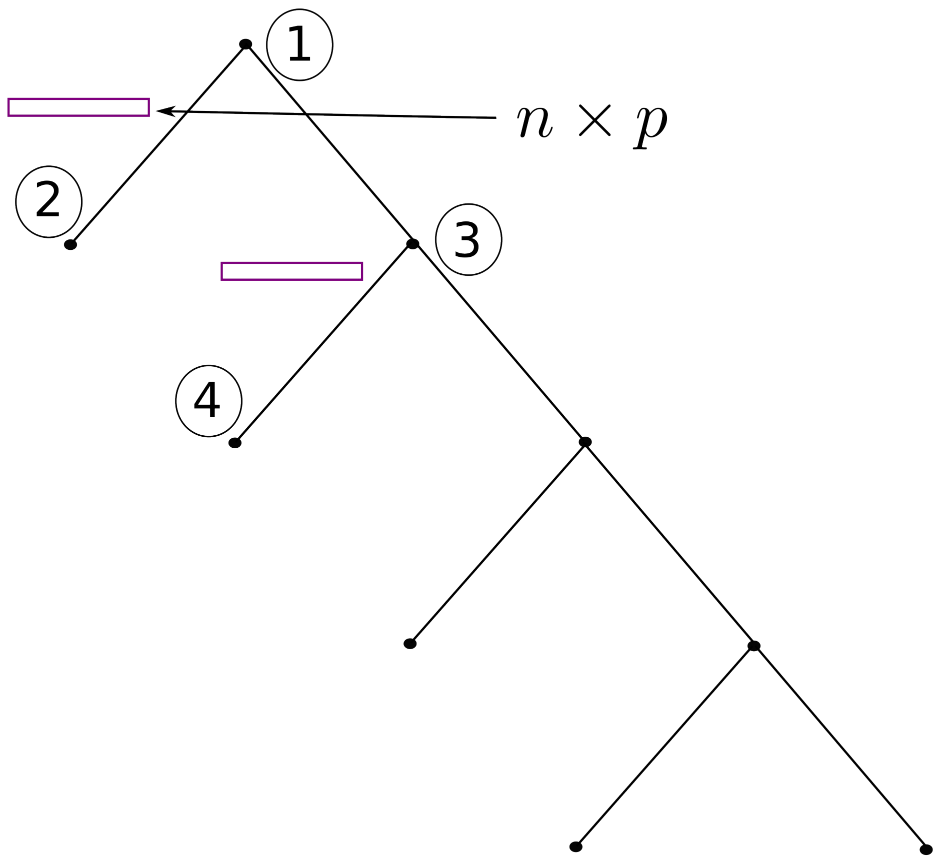
Depth First Traversal is Not Optimal for Peak Memory Consumption



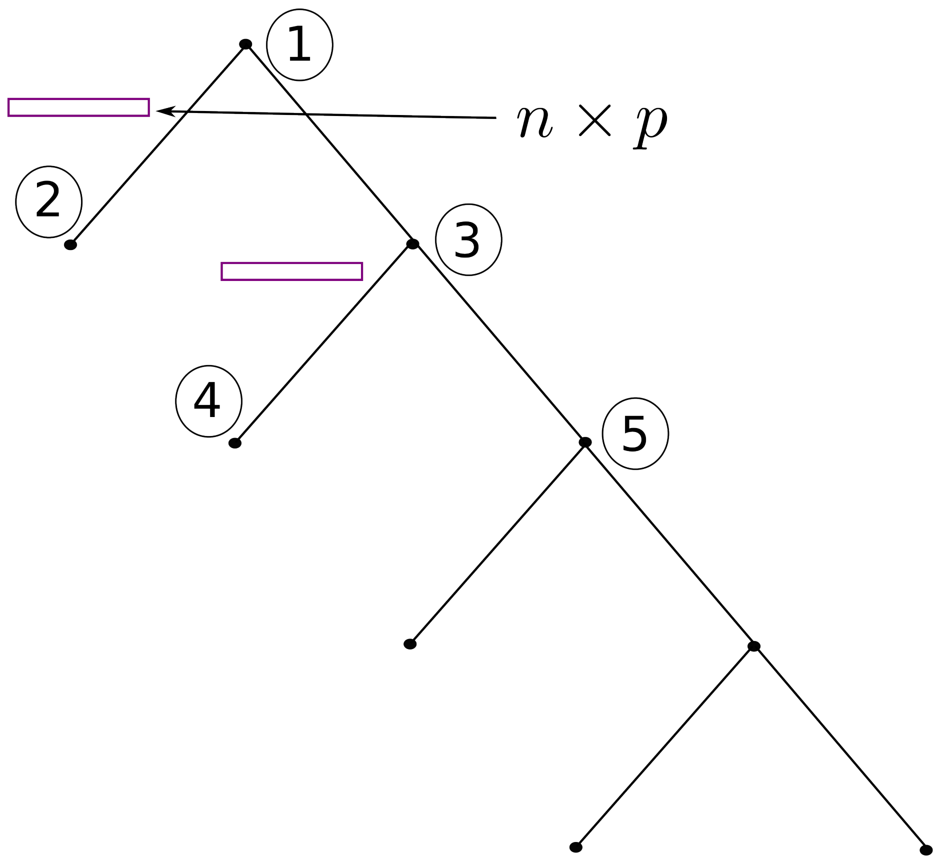
Depth First Traversal is Not Optimal for Peak Memory Consumption



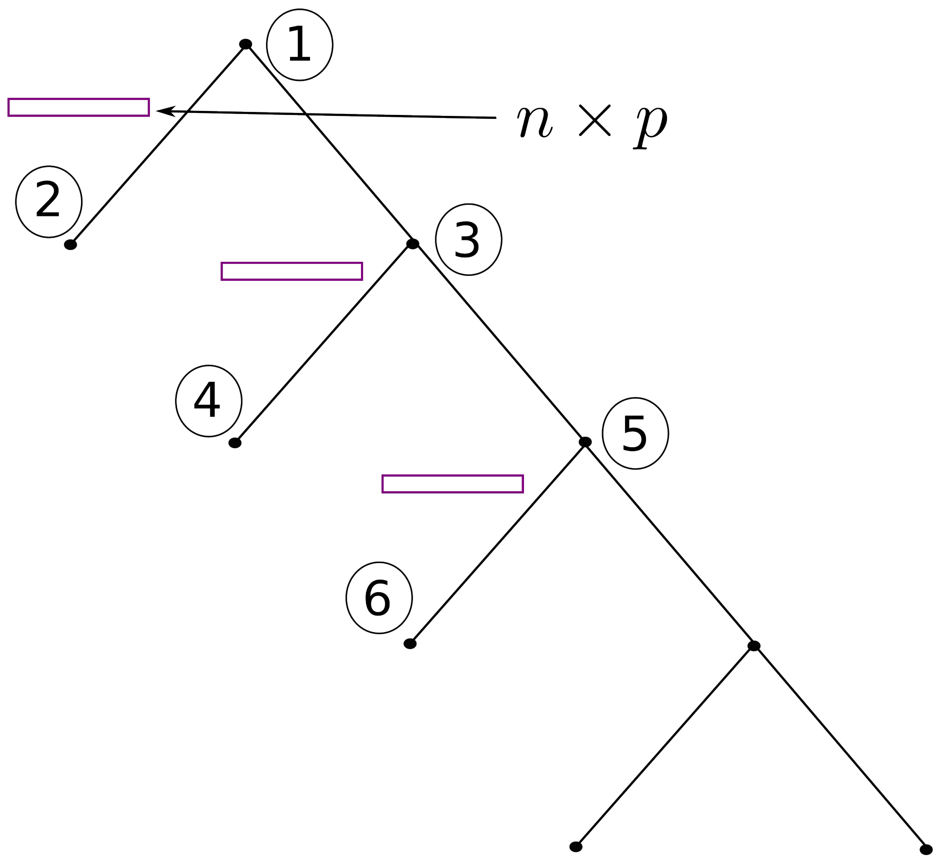
Depth First Traversal is Not Optimal for Peak Memory Consumption



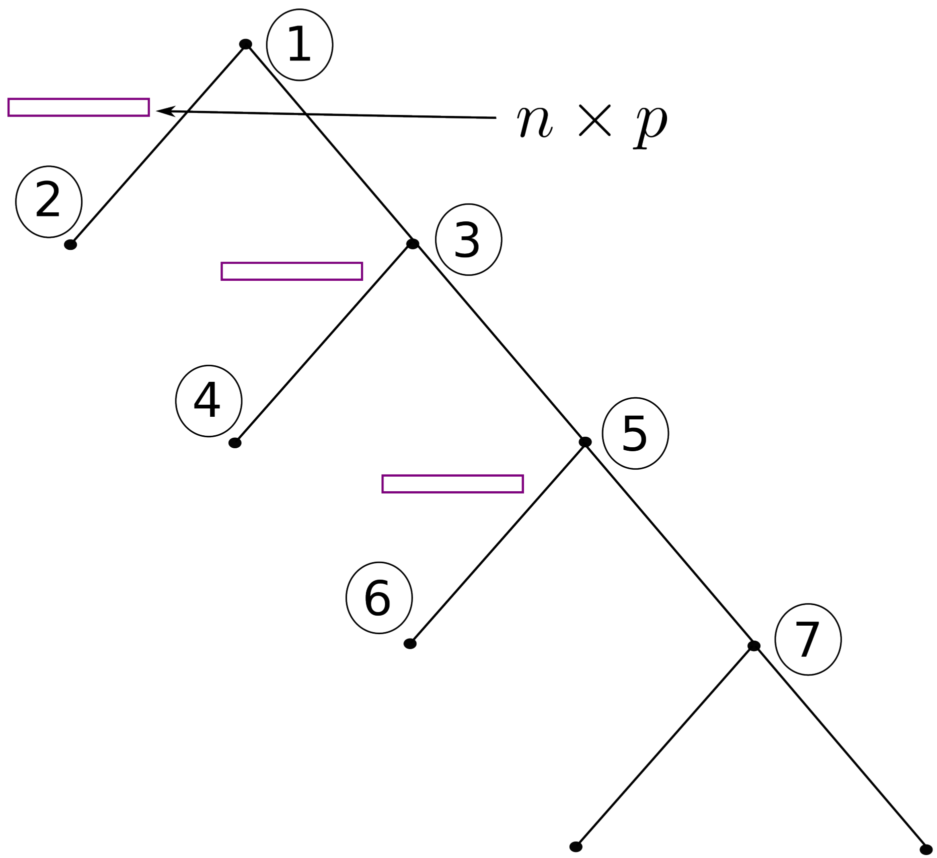
Depth First Traversal is Not Optimal for Peak Memory Consumption



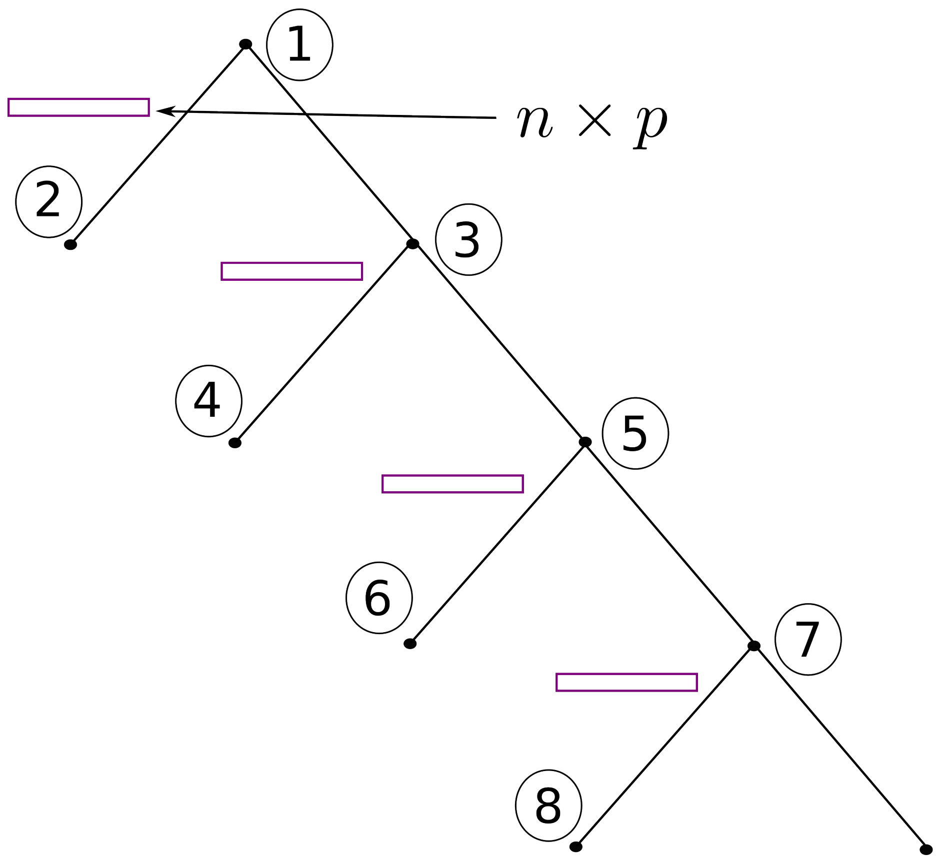
Depth First Traversal is Not Optimal for Peak Memory Consumption



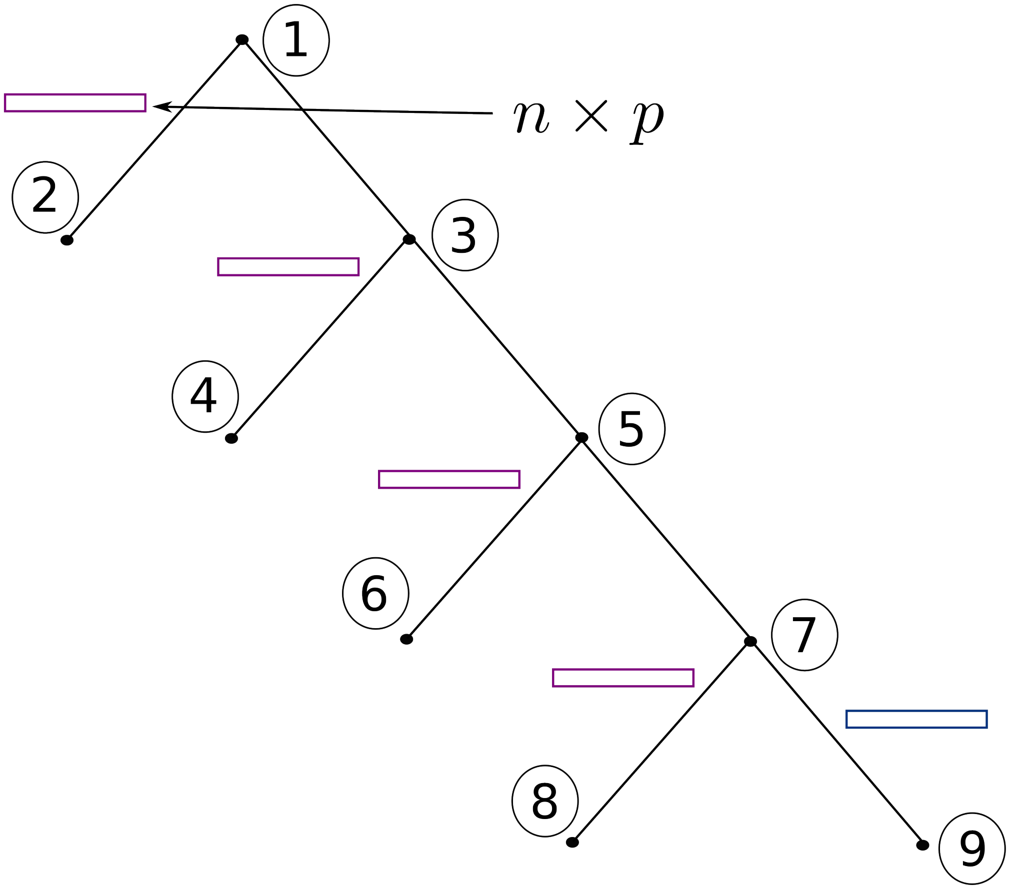
Depth First Traversal is Not Optimal for Peak Memory Consumption



Depth First Traversal is Not Optimal for Peak Memory Consumption



Depth First Traversal is Not Optimal for Peak Memory Consumption



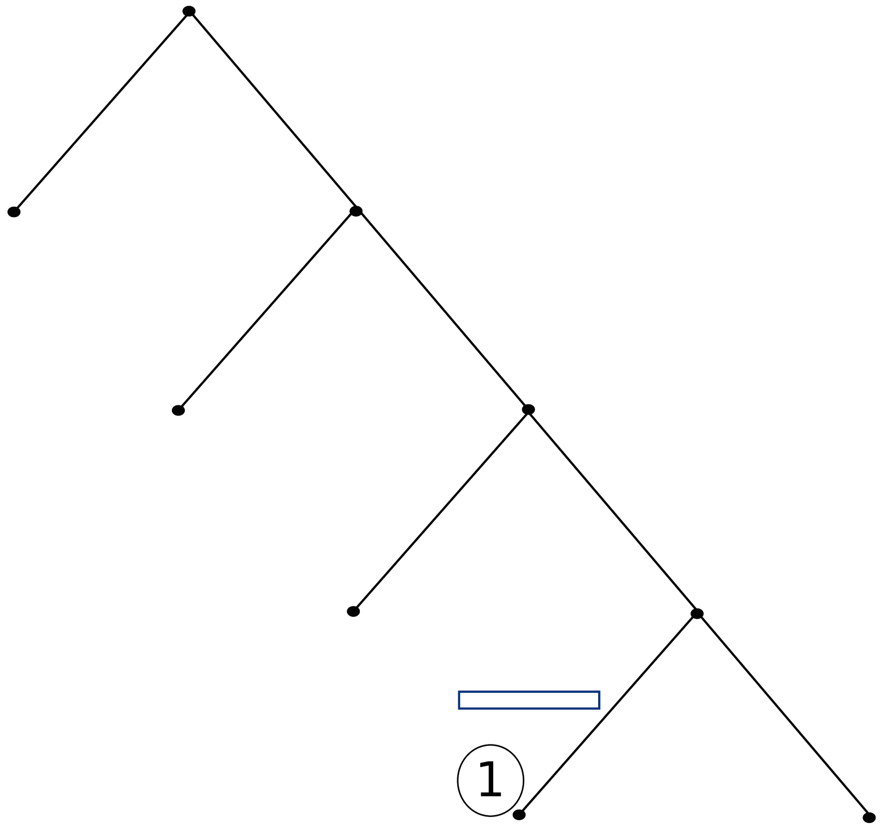
Depth First Traversal is Not Optimal for Peak Memory Consumption

- Each block shown is of dimension $n \times p$, where p is the rank of the off-diagonal blocks of the original matrix
- Maximum Depth of this tree is $O(n/p)$
- This implies a memory consumption of $O(n^2)$

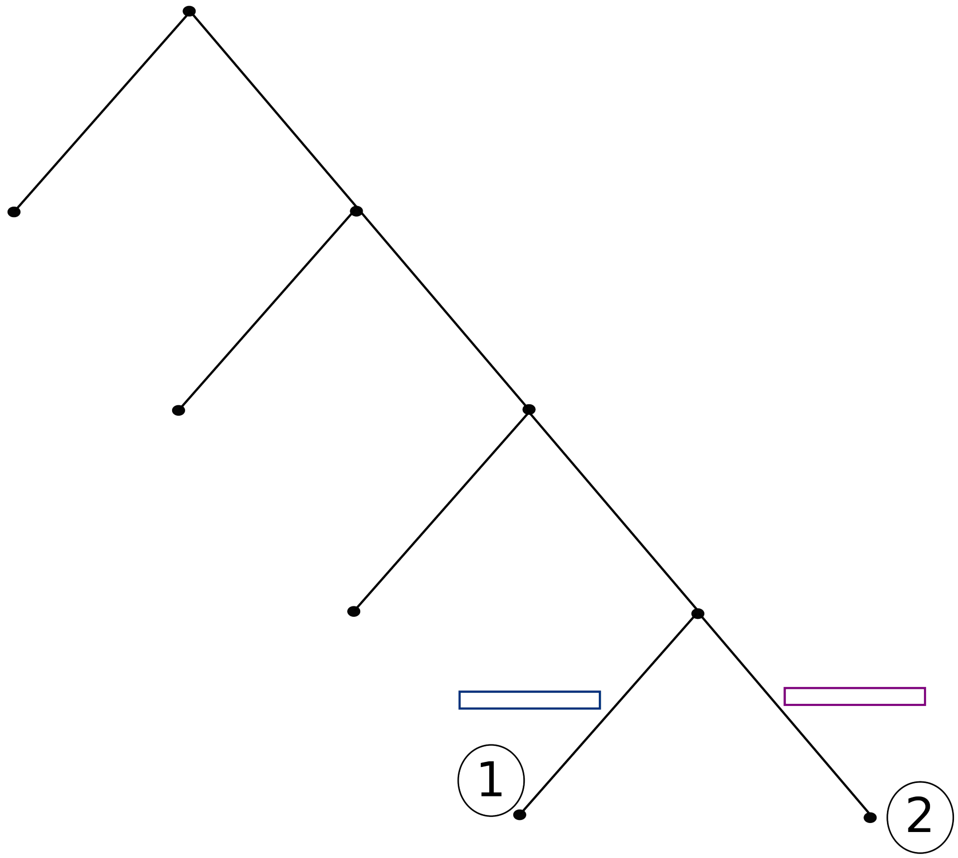
How to fix this: Deepest First Traversal

- Traverse in a Deepest first ordering
- Peak workspace memory consumption of $O(np)$ for a tree of maximal depth

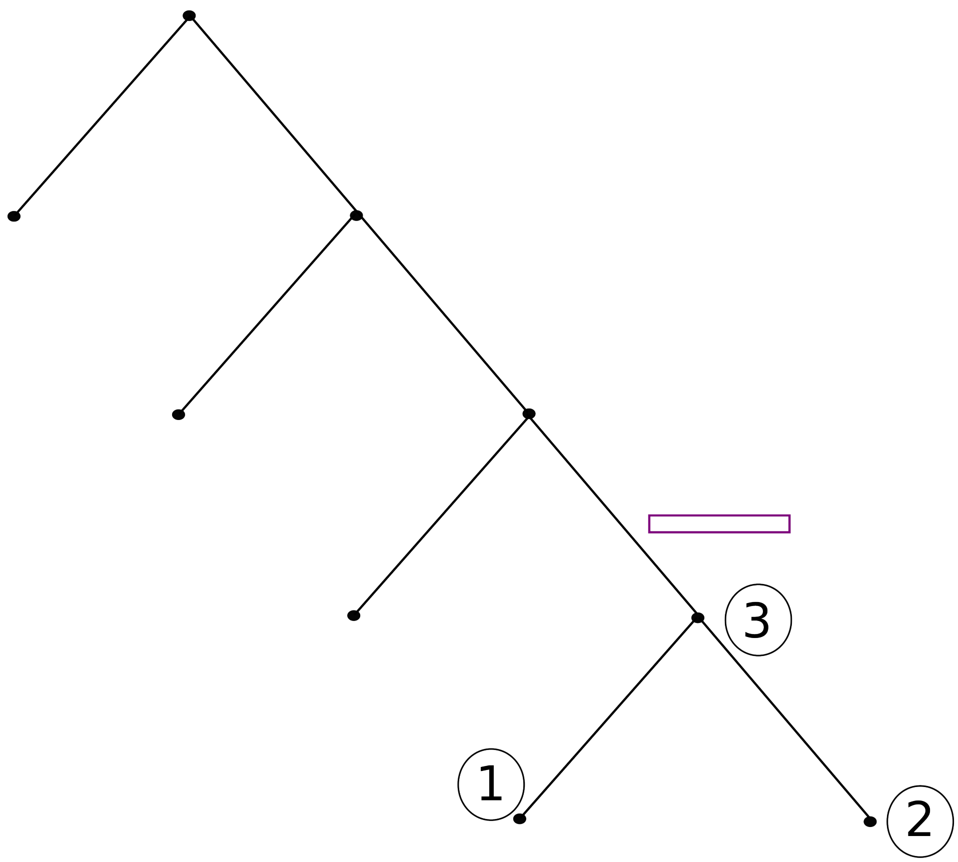
Method We Use: Deepest First Traversal



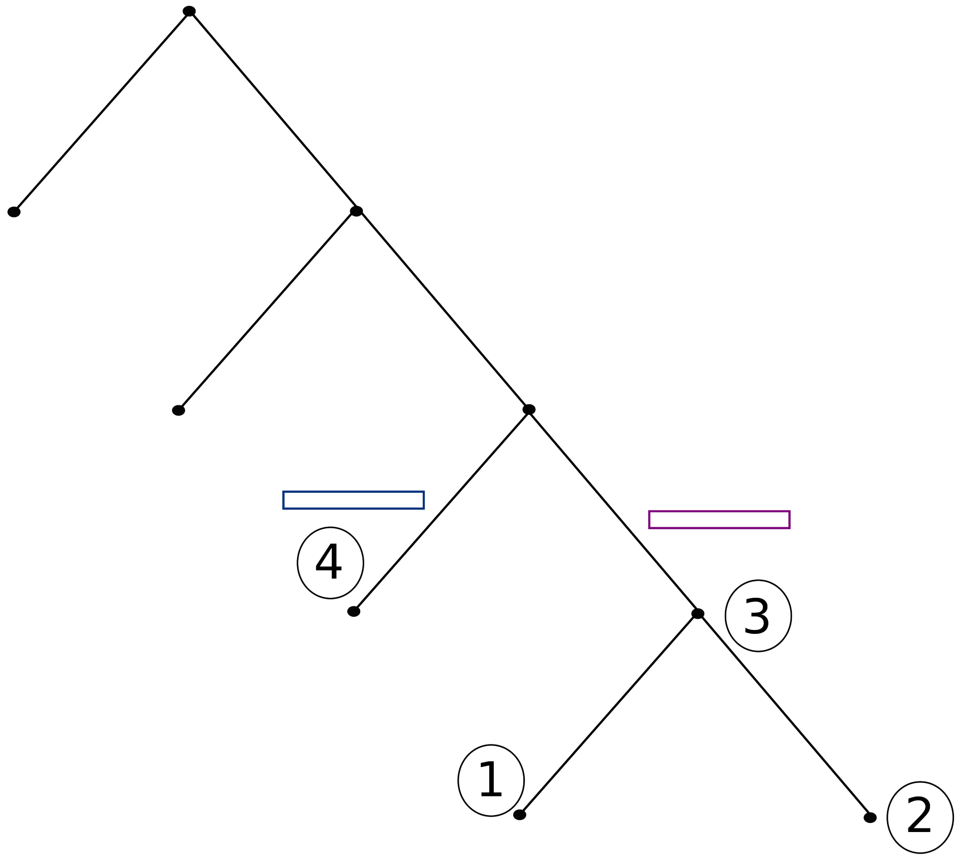
Method We Use: Deepest First Traversal



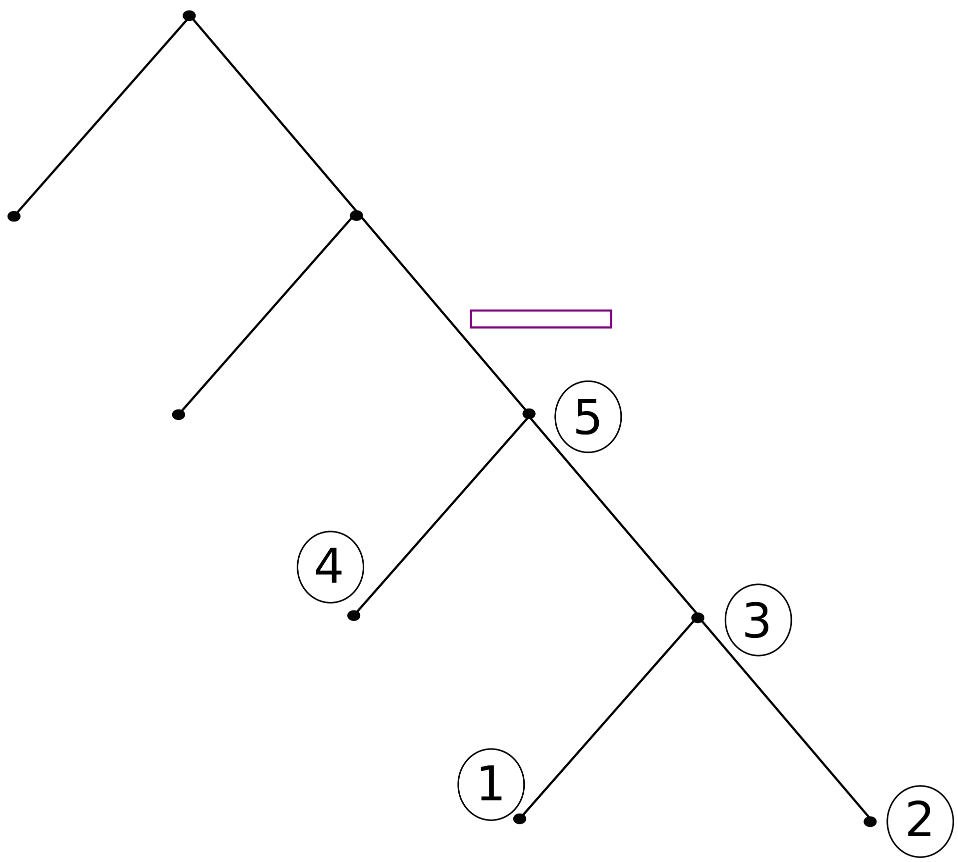
Method We Use: Deepest First Traversal



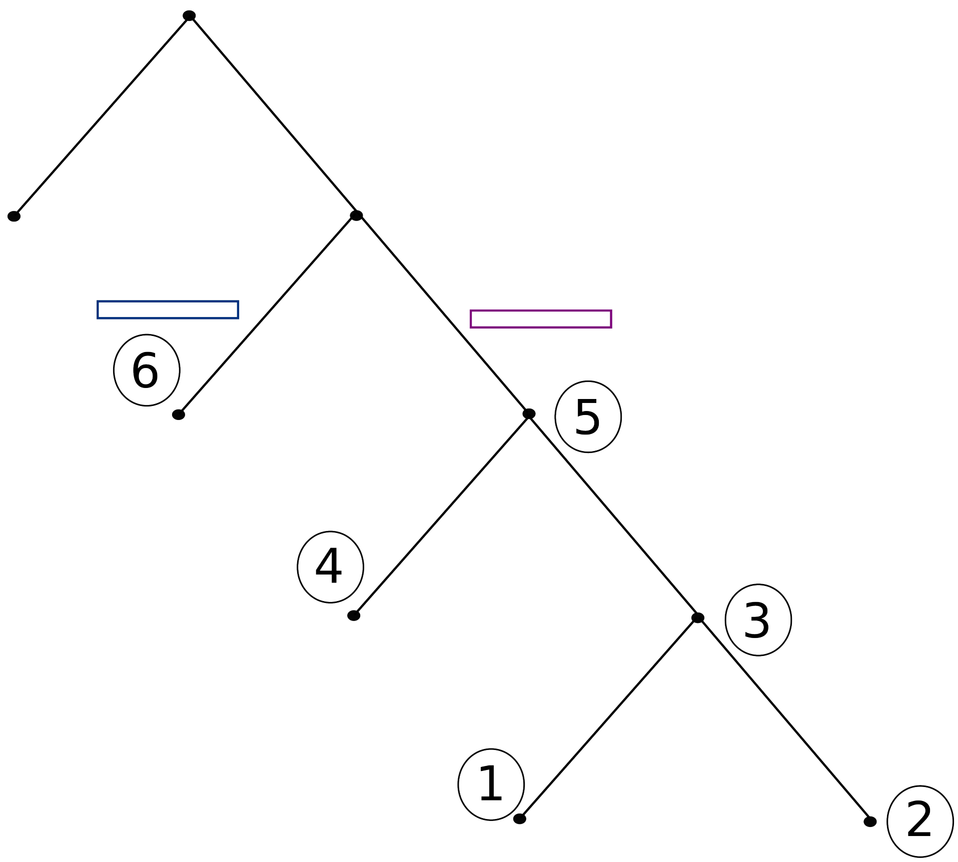
Method We Use: Deepest First Traversal



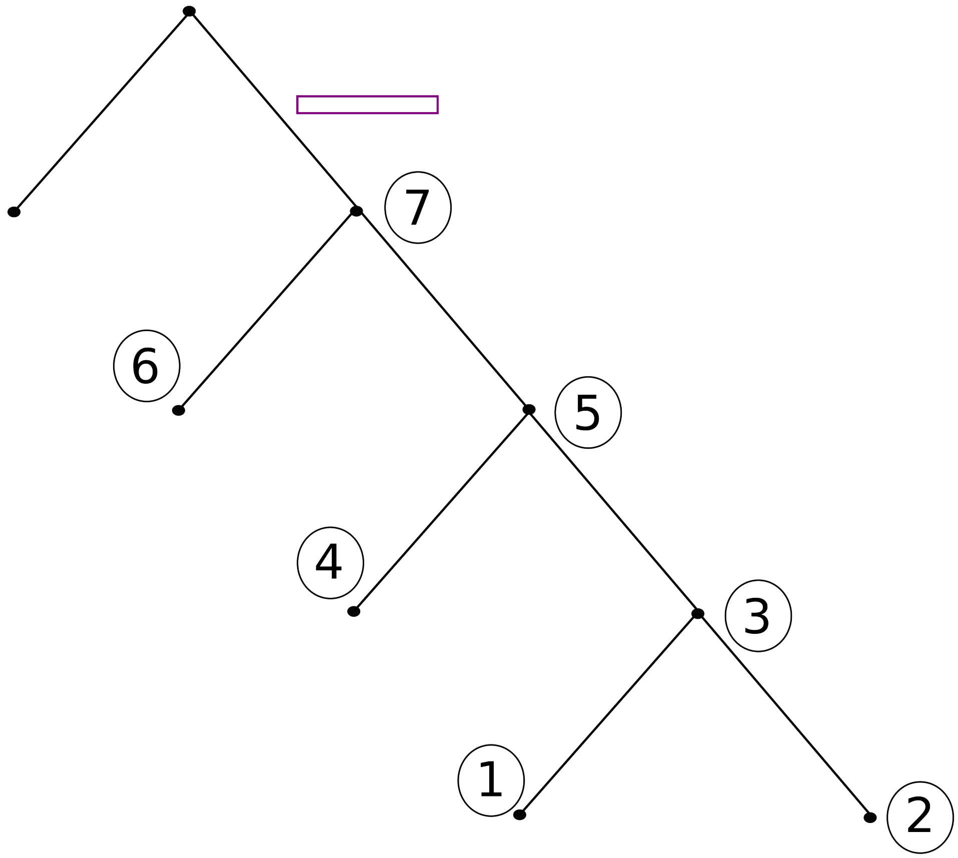
Method We Use: Deepest First Traversal



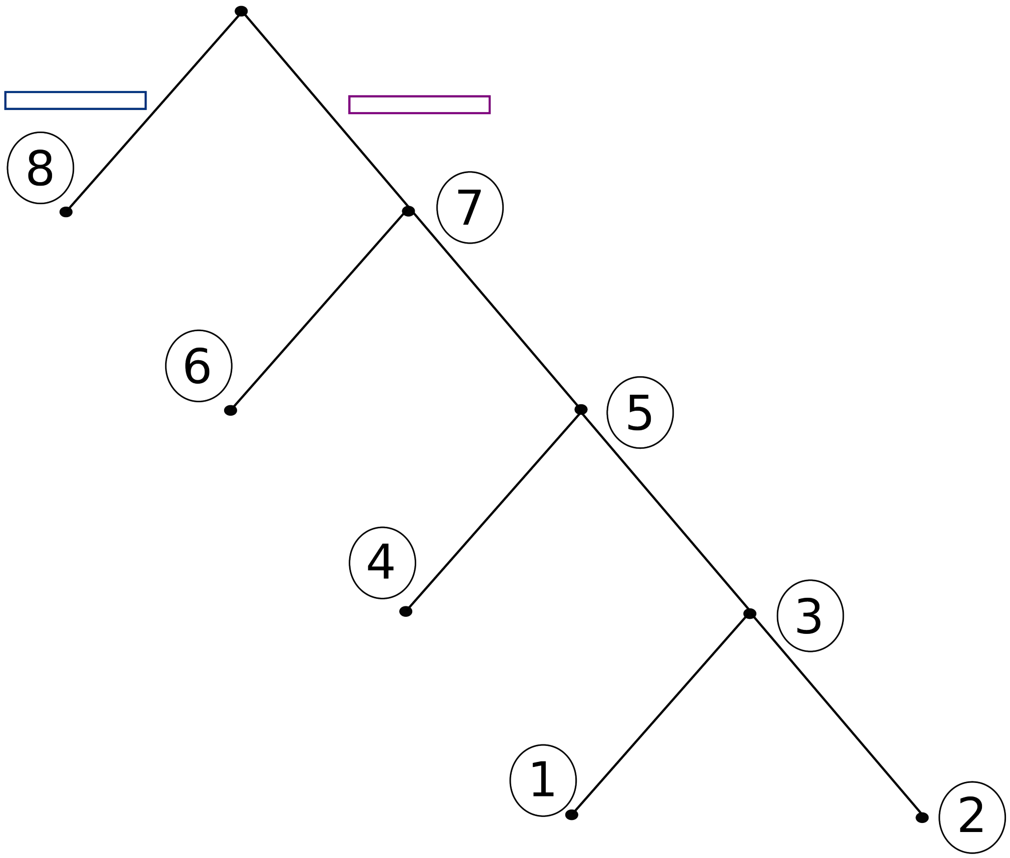
Method We Use: Deepest First Traversal



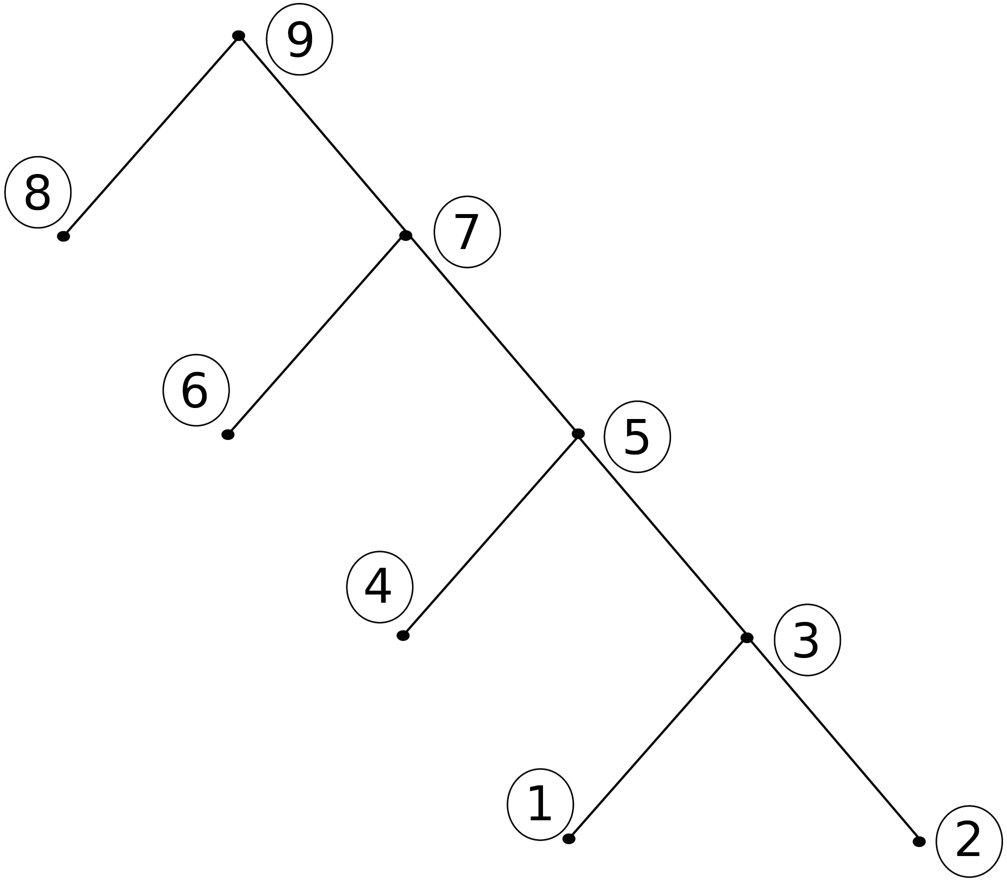
Method We Use: Deepest First Traversal



Method We Use: Deepest First Traversal



Method We Use: Deepest First Traversal



Deepest First Traversal Memory Count

- For the maximal depth tree, only 2 blocks of size $n \times p$ are in memory at any given time
- Peak workspace consumption for a tree of maximal depth is $O(pn)$ using deepest first traversal vs $O(n^2)$ for depth-first traversal
- Further, for a complete tree, the deepest first traversal leads to a peak workspace consumption of $O(pn \log n)$

Worst Case Memory Consumption for our Algorithm

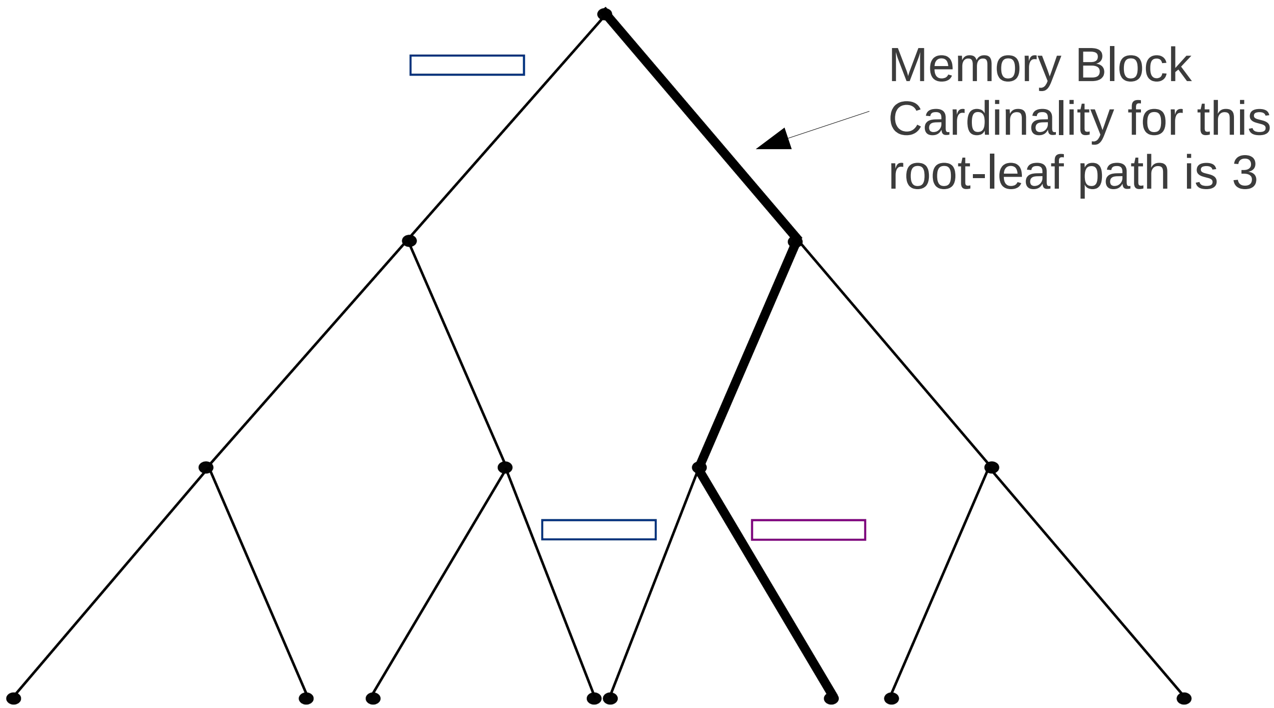
- How does worst case memory usage grow with matrix size, n?
- This can be formulated as a graph theory problem

Memory Block Cardinality for a Root-leaf Path

- Without loss of generality, any HSS tree can be re-ordered such that the depth of the left subtree is always equal to or greater than the depth of the right subtree.

- Block is stored in memory when we return from a left call.

- Memory Block Cardinality for a root-leaf path is equal to the number of right children in that path plus one.

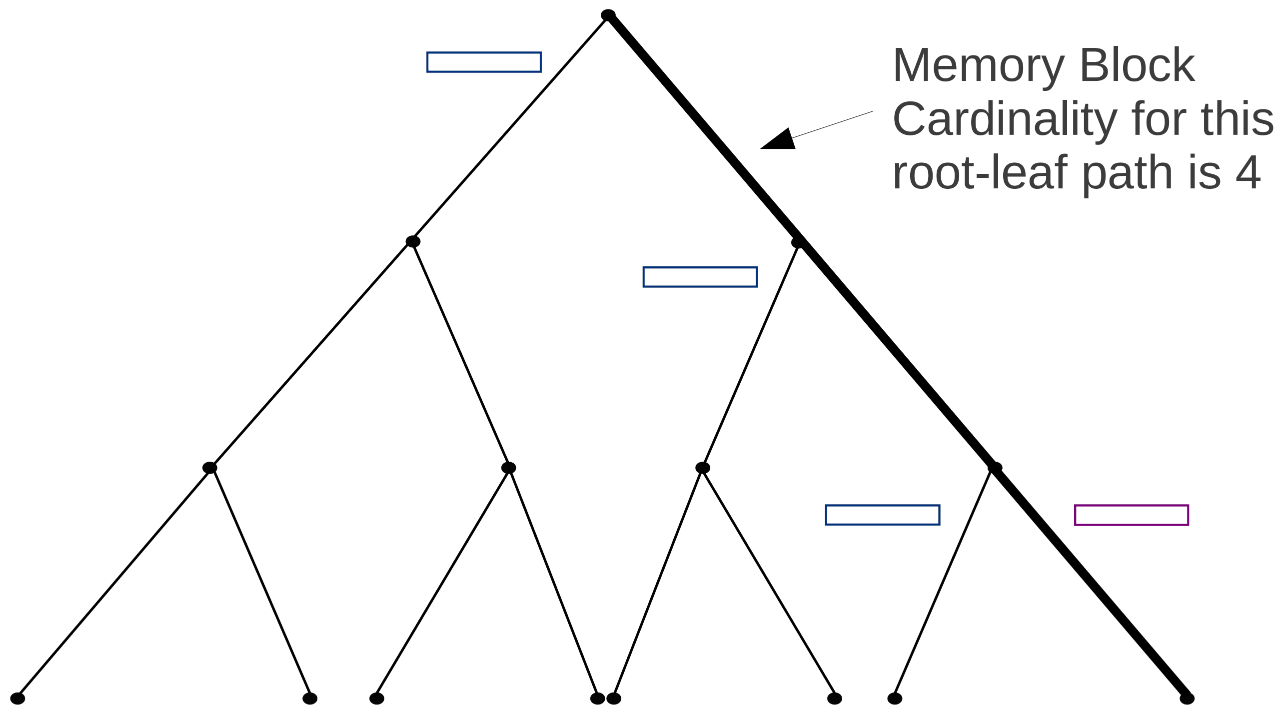


Memory Block Cardinality for a Root-leaf Path

- Without loss of generality, any HSS tree can be re-ordered such that the depth of the left subtree is always equal to or greater than the depth of the right subtree.

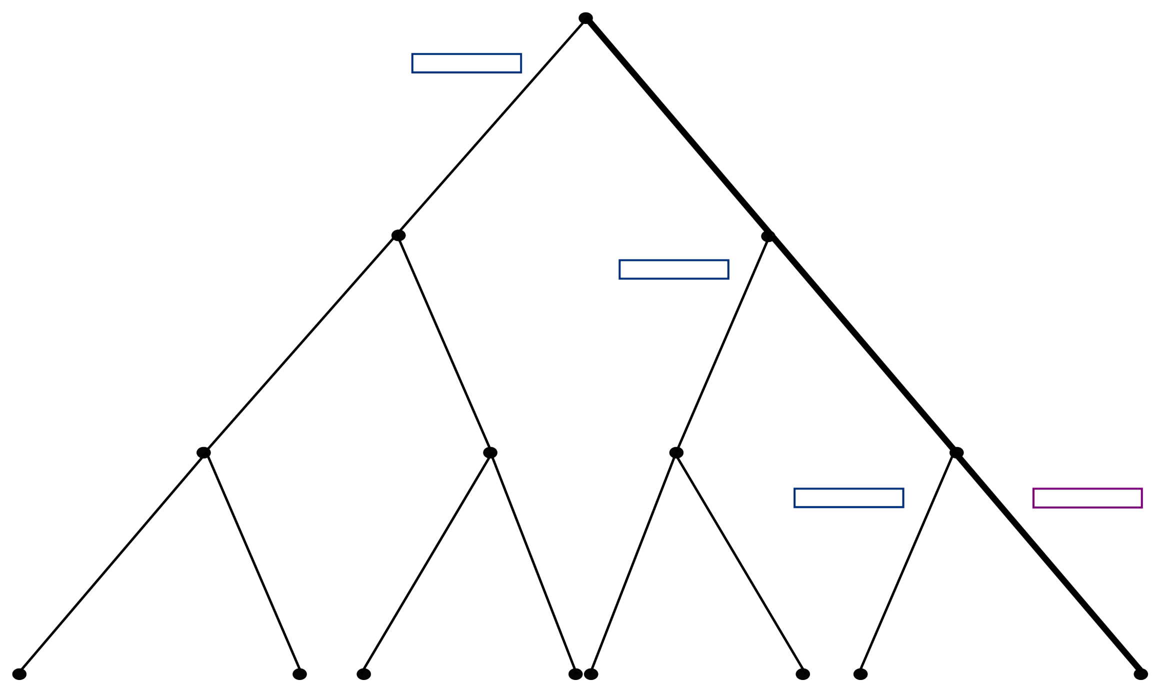
- Block is stored in memory when we return from a left call.

- Memory Block Cardinality for a root-leaf path is equal to the number of right children in that path plus one.



The Search for Maximum Memory Consumption Can Be Formulated as a Graph Theory Problem

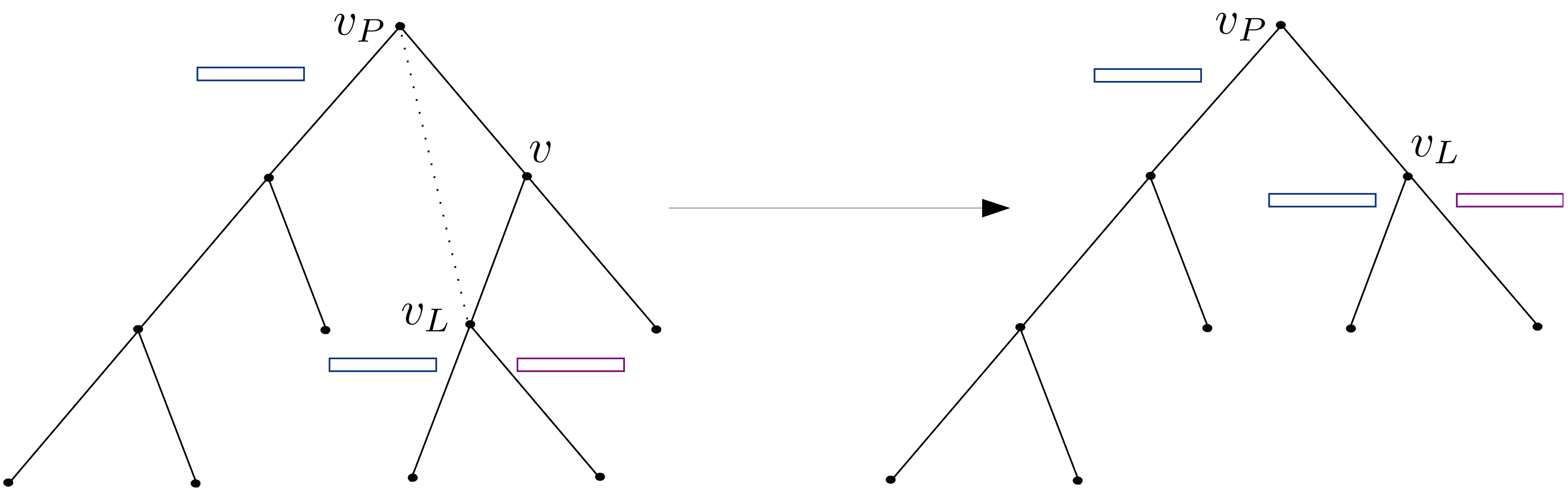
- Branch with maximum memory block cardinality will give peak memory consumption.
- Number of leaf nodes, N_L , is proportional to the size of the matrix n.
- Number of leaf nodes, N_L, is proportional to the number of nodes, N.
- We are looking for a class of trees that maximizes the ratio of the worst case memory block cardinality to number of nodes.



We Can Narrow Down Our Search By Excluding Some Classes of Trees

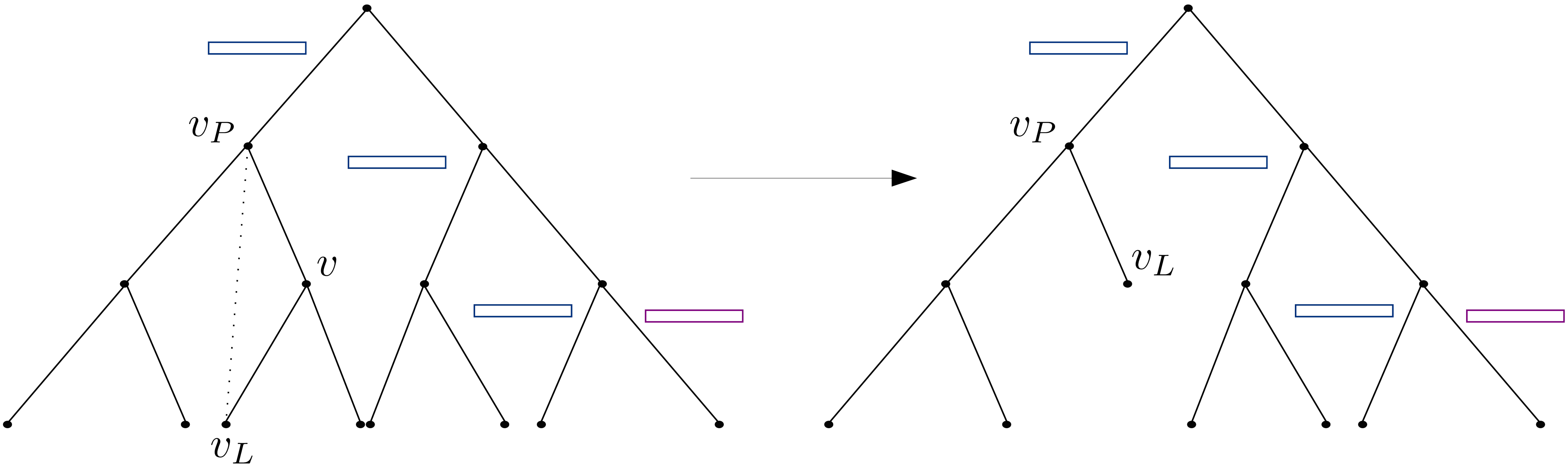
- We can rule out the class of trees that don't have the worst case memory block cardinality along their right-most branch.

- Worst-case memory block cardinality = 3 for both trees shown below.

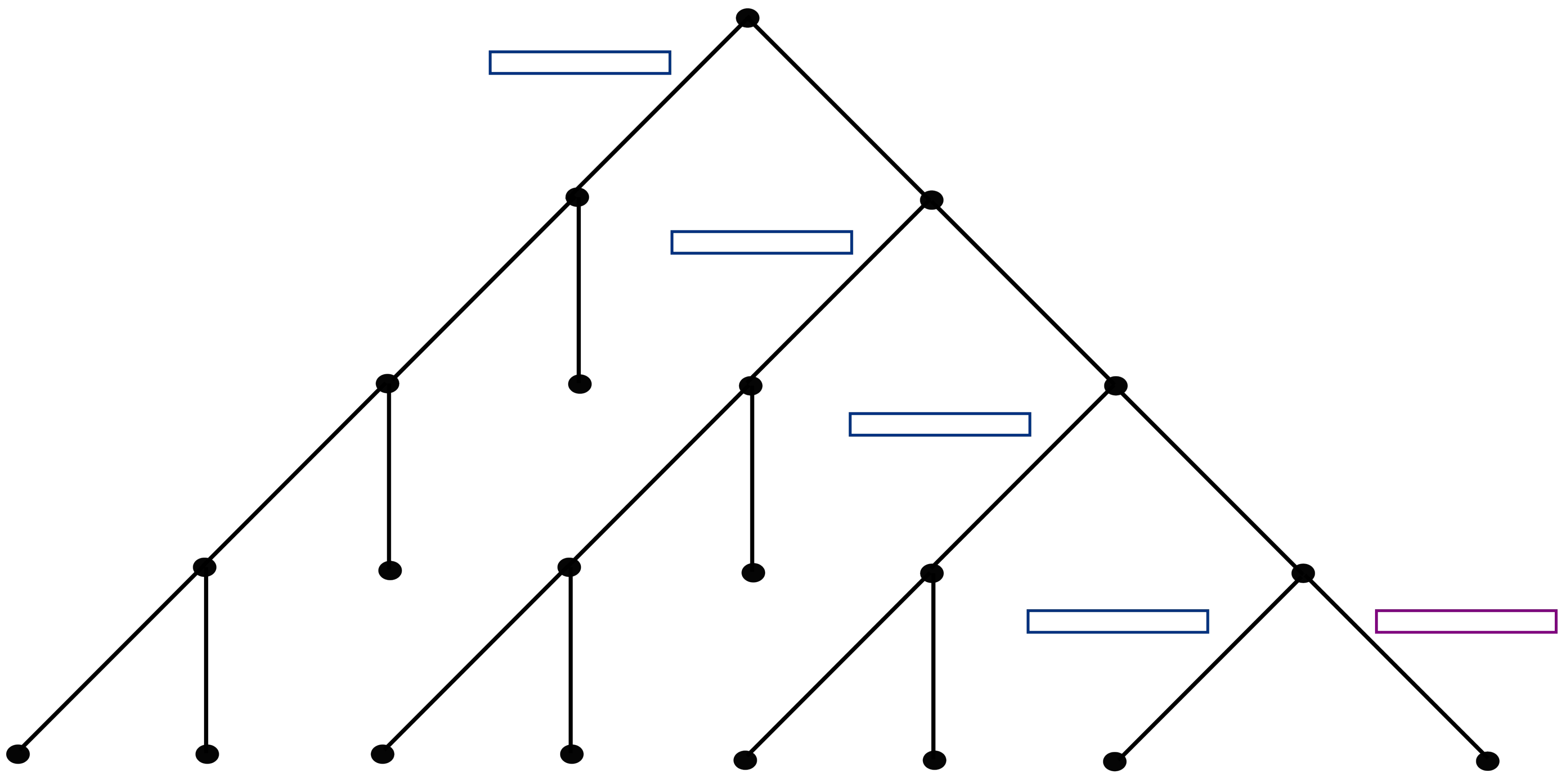


Complete Trees Do Not Give Rise to Worst Case Memory Usage

- Complete trees have the property that the worst case memory block cardinality occurs along the right-most branch.
- Worst-case memory block cardinality = 4 for both trees shown below.



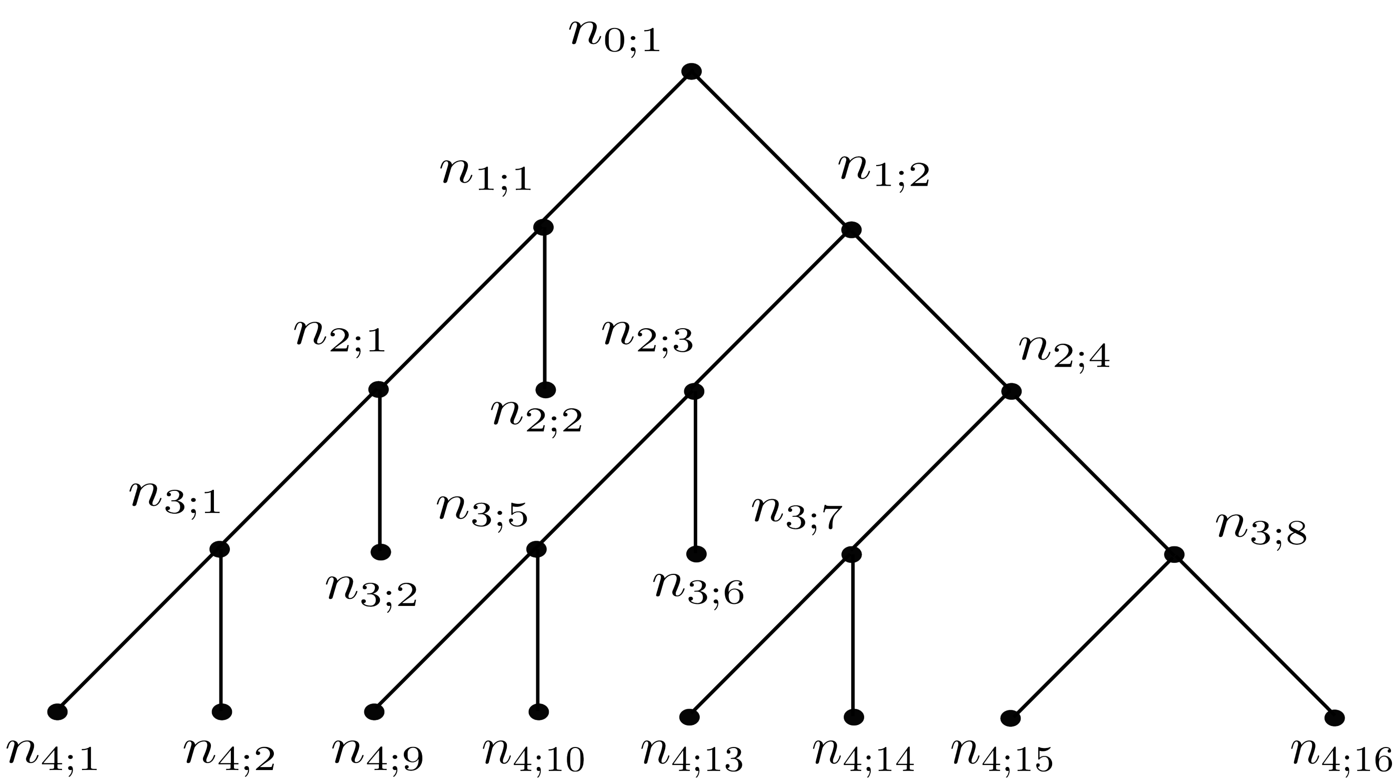
Class of 'Worst Case' Trees* Has a Surprising Structure



* K. Lessel, M. Hartman, and S. Chandrasekaran. A Fast Memory Efficient Construction Algorithm for Hierarchically Semi-Separable Representations. *Submitted to SIAM J. Matrix Analysis and Applications*

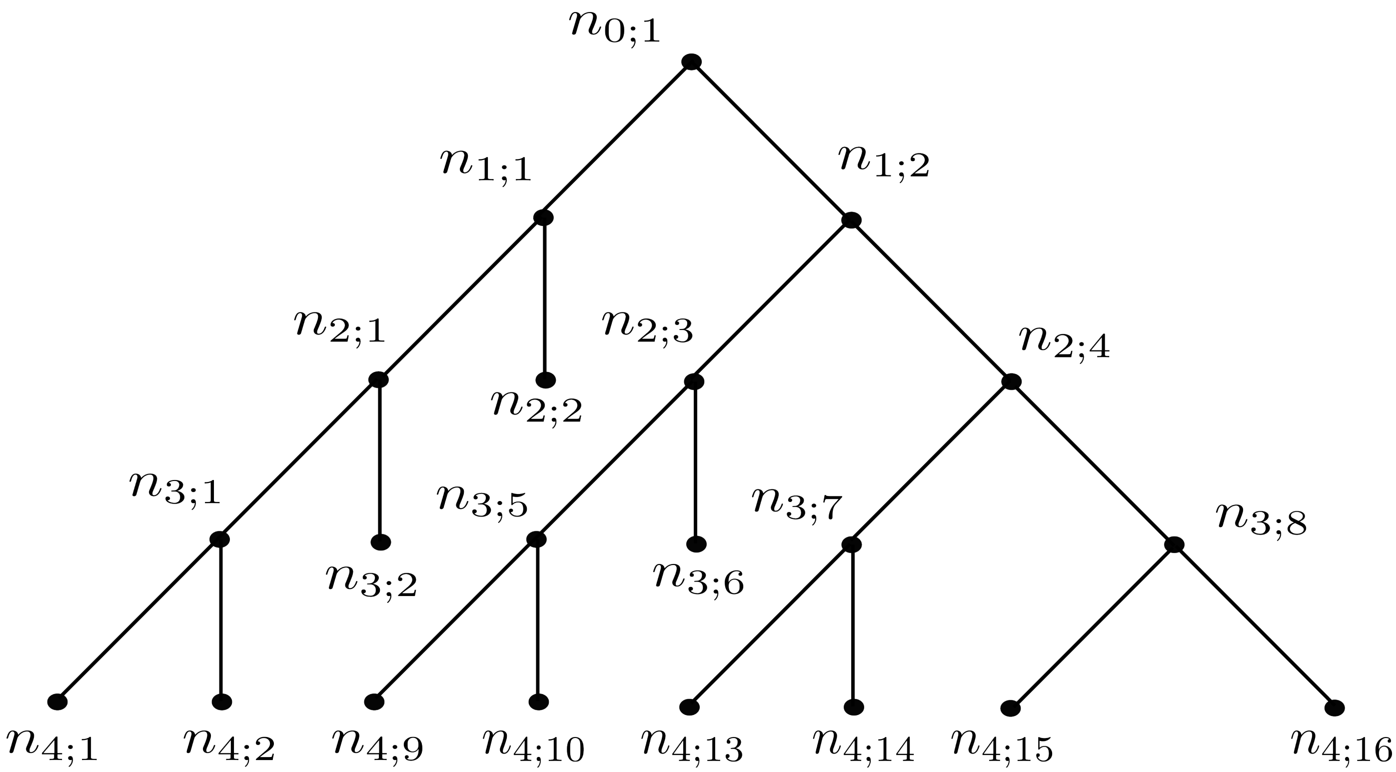
Worst Case Memory Consumption for our Algorithm

- Worst case number of memory blocks we can generate is $O(N^{0.5})$, and is generated by the binary tree with a structure as shown

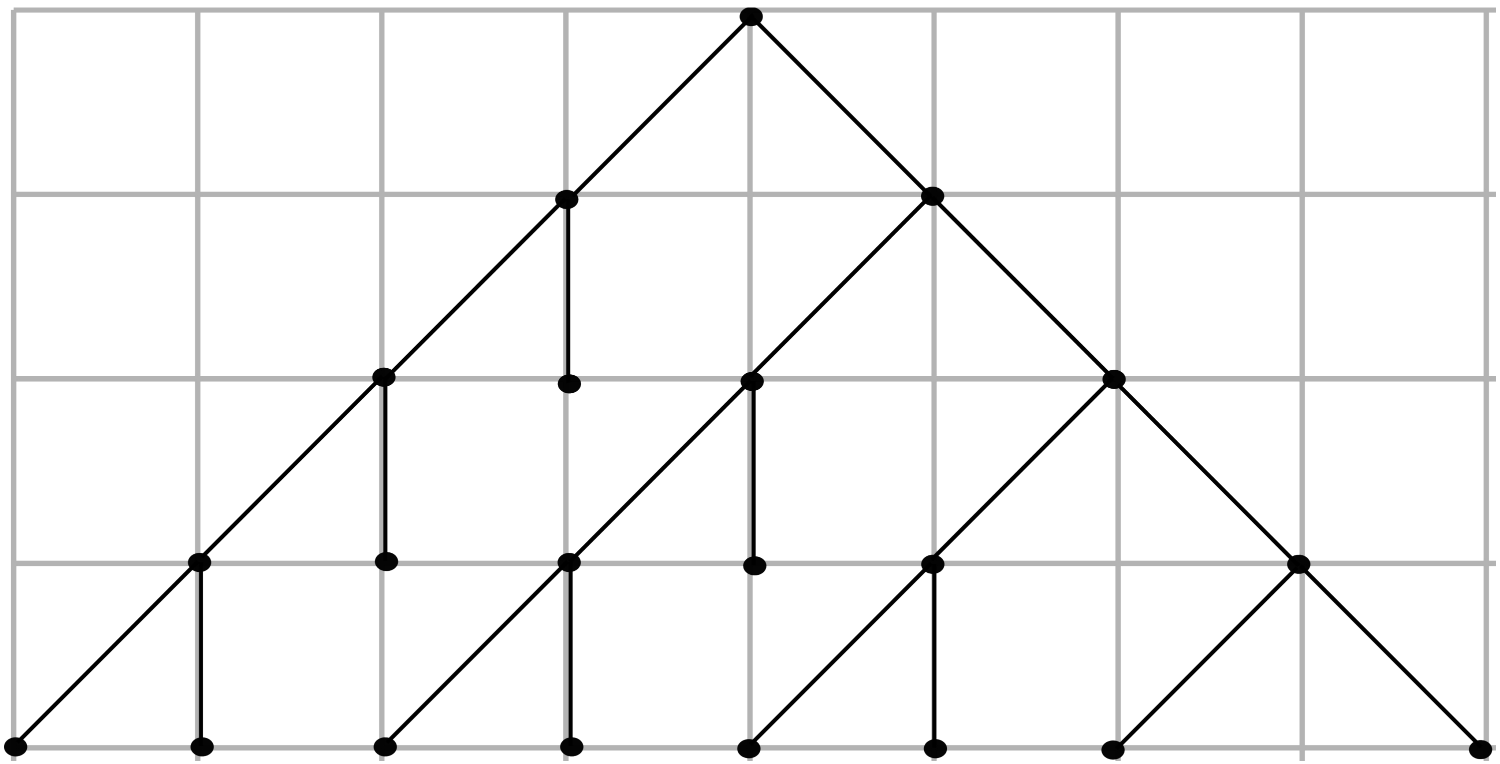


Worst Case Memory Consumption for our Algorithm

- Worst case number of memory blocks we can generate is $O(N^{0.5})$, and is generated by the binary tree with a structure as shown



Worst Case Number of Memory Blocks is $O(N^{0.5})$



Worst case number of memory blocks $= d + 1$

$$N \approx d^2$$

Worst case number of memory blocks $\approx \sqrt{N}$ *

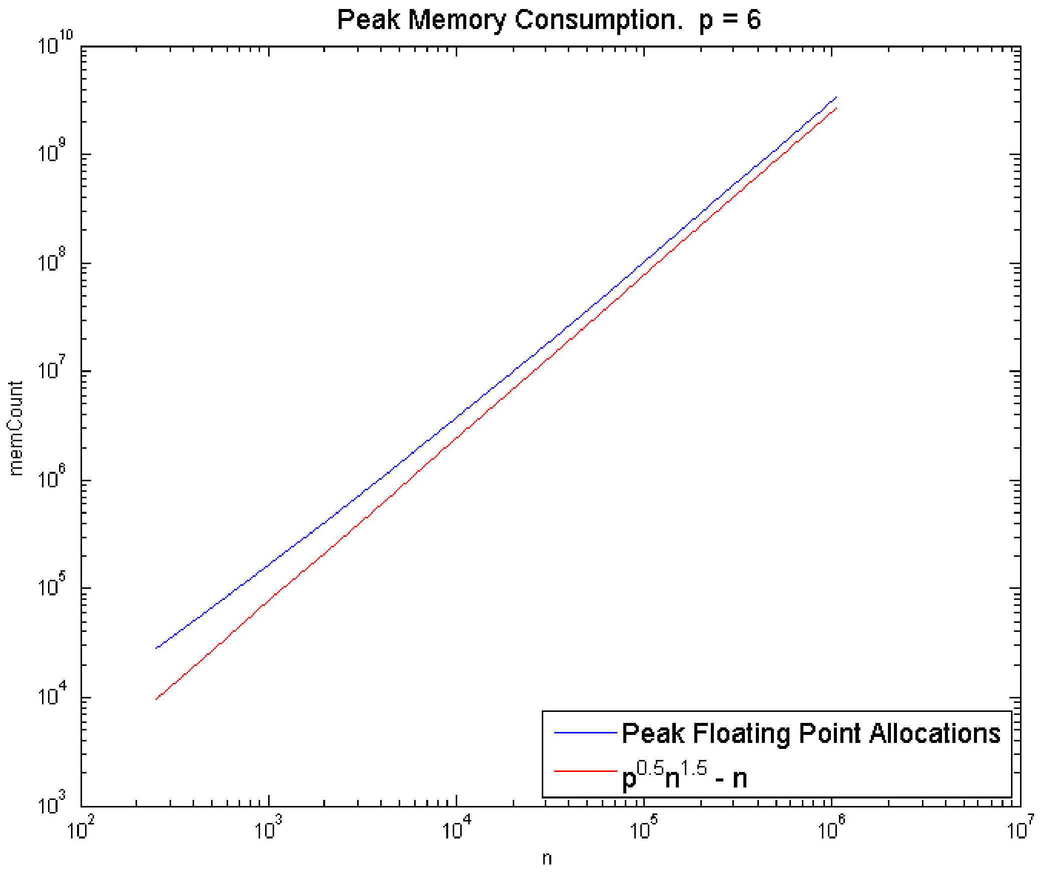
* K. Lessel, M. Hartman, and S. Chandrasekaran. A Fast Memory Efficient Construction Algorithm for Hierarchically Semi-Separable Representations. *Submitted to SIAM J. Matrix Analysis and Applications*

Relationship between the number of nodes, N and the size of our matrix, n

- Number of non-leaf nodes, N_N, is one less than the number of leaf nodes, N_L , i.e,
$$N_N = N_L - 1$$
- $N_L = n/p$
- $N = N_N + N_L$
$$= \frac{2n}{p} - 1$$
- The worst case number of memory blocks is $O(N^{0.5})$
- Peak memory consumption is $O(p^{0.5}n^{1.5})$

Numerical Results

- Upper bound for worst case peak memory consumption is $O(p^{0.5}n^{1.5})$, and we can show this is a tight bound for 'Worst Case' trees.



Conclusion

- Our 2 Phase Algorithm allows for a deepest first traversal of the HSS tree, yielding a reduction in peak memory complexity from $O(n^2)$ to $O(p^{0.5}n^{1.5})$ as compared with previous algorithms, while still taking only $O(n^2)$ flops.

- Open question: Does there exist a 'linear' memory algorithm which does not give up the $O(n^2)$ flop constraint?

References

[1] Shivkumar Chandrasekaran, Ming Gu, and Timothy Pals. A Fast ULV Decomposition Solver for Hierarchically Semiseparable Representations. *SIAM Journal on Matrix Analysis and Applications,* 28(3):603-622, 2006

[2] K. Lessel, M. Hartman, and S. Chandrasekaran. A Fast Memory Efficient Construction Algorithm for Hierarchically Semi-Separable Representations. *Submitted to SIAM J. Matrix Analysis and Applications*

[3] Per-Gunnar Martinsson. A Fast Randomized Algorithm for Computing a Hierarchically Semiseparable Representation Matrix. *SIAM Journal on Matrix Analysis and Applications,* 32(4):1251-1274, 2011.

[4] Jianlin Xia, Shivkumar Chandrasekaran, Ming Gu, and Xiaoye~S Li. Fast Algorithms for Hierarchically Semiseparable Matrices. *Numerical Linear Algebra with Applications,* 17(6):953—976, 2010.

Appendix

Why HSS?

- Matrix vector multiply: $O(n^2)$ flops vs HSS vector multiply: $O(n)$ flops
- Solution, x, of Ax = b. Gaussian Elimination: $O(n^3)$ flops vs Fast HSS solver: $O(n)$ flops

Leaf Node Computations

- For (k_1, i), (k_2, j) leaf nodes define

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

- For (k_1, i) a leaf node, and (k_2, j) is not, define

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

- For (k_1, i) not a leaf node, and (k_2, j) is a leaf node, define

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

Non-Leaf Node Computations

- For (k_1, i), (k_2, j) not leaf nodes, we will have $k_1 = k_2$ and can then write

$$B_{k;i,j} = B_{k_1;i,k_2;j}$$

  where $k = k_1 = k_2$

- Then define

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

Phase 1 - Leaf Node Computations

$$
{}_r\acute{H}_{k;i} = \begin{pmatrix} U_{k;i} & * \end{pmatrix} \begin{pmatrix} \Sigma_{k;i} & 0 \\ 0 & * \end{pmatrix} \begin{pmatrix} Q_{k;i}^T \\ * \end{pmatrix}
$$

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

Phase 1 – Non-leaf Node Computations

$$ {}_r\tilde{H}_{k;2i-1} = \Sigma_{k;2i-1}\, Q_{k;2i-1}^T \qquad {}_c\tilde{H}_{k;2i-1} = P_{k;2i-1}\, \Lambda_{k;2i-1} $$

$$ {}_r\tilde{H}_{k;2i} = \Sigma_{k;2i}\, Q_{k;2i}^T \qquad {}_c\tilde{H}_{k;2i} = P_{k;2i}\, \Lambda_{k;2i} $$

Phase 1 – Non-leaf Node Computations

- Remove block cloumns of $Q_{k;i,j}^T$ which corresond to the columns that lie in the diagonal block $D_{k-1;i}$

$$\mathrm{Q}_{k;i}^T = \left(\begin{array}{cccc} Q_{k;i,1}^T & Q_{k;i,2}^T & \cdots & Q_{k;i,2^k-1}^T \end{array} \right)$$

$$\tilde{Q}_{k;i}^T = \left(\begin{array}{cccccc} Q_{k;i,1}^T & \cdots & Q_{k;i,i-1}^T & Q_{k;i,i+1}^T & \cdots & Q_{k;i,2^k-1}^T \end{array} \right)$$

- Compressed Hankel blocks at node $(k-1;i)$

$${}_rH_{k-1;i} = \left(\begin{array}{c} \Sigma_{k;2i-1}\,\tilde{Q}_{k;2i-1}^T \\ \Sigma_{k,2i}\,\tilde{Q}_{k;2i}^T \end{array} \right)$$

$${}_cH_{k-1;i} = \left(\begin{array}{cc} \tilde{P}_{k;2i-1}\,\Lambda_{k;2i-1} & \tilde{P}_{k;2i}\,\Lambda_{k;2i} \end{array} \right)$$

Algorithm 1 Pass 1U

Algorithm 1 Pass 1U - Memory Efficient HSS Algorithm

function HSS_BASIS($tree$)
    if $tree$ is a leaf node **then**
        $(U_{k;i}, \Sigma_{k;i}, Q^T_{k;i}) = \text{TRUNC\_SVD}([A_{k;i,1} \;\; A_{k;i,2} \;\; \dots \;\; A_{k;i,i-1} \;\; A_{k;i,i+1} \;\; \dots \;\; A_{k;i,end}])$
        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 DEPTH($treeL$) $\geq$ DEPTH($treeR$) **then**
            ${}_rH_{k+1;2i-1}$ = HSS_BASIS($treeL$)
            ${}_rH_{k+1;2i}$ = HSS_BASIS($treeR$)
        else
            ${}_rH_{k+1;2i}$ = HSS_BASIS($treeR$)
            ${}_rH_{k+1;2i-1}$ = 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

Algorithm 2 Pass 2BU

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

Require: *tree* is not a leaf node

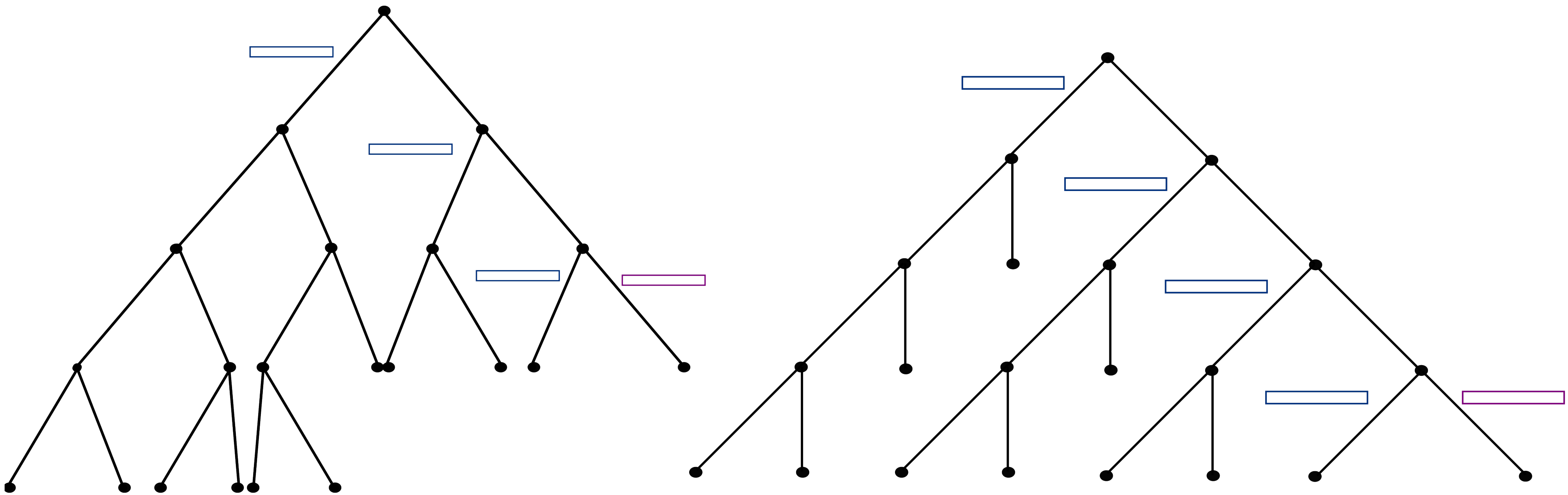
function B_DIAG(tree)
  $-,\ treeL,\ treeR = tree$ ▷ $\exists\ (k1; i)$ s.t. it is the numbering for the root node of $treeL$.
  ▷ $\exists\ (k2; j)$ s.t. it is the numbering for the root node of $treeR$.
  if $treeL$ is a leaf node and $treeR$ is a leaf node **then**
    $B_{k_1;i,k_2;j} = U^T_{k_1;i} A_{k_1;i,k_2;j} V_{k_2;j}$
  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

Algorithm 3 Pass 2BU

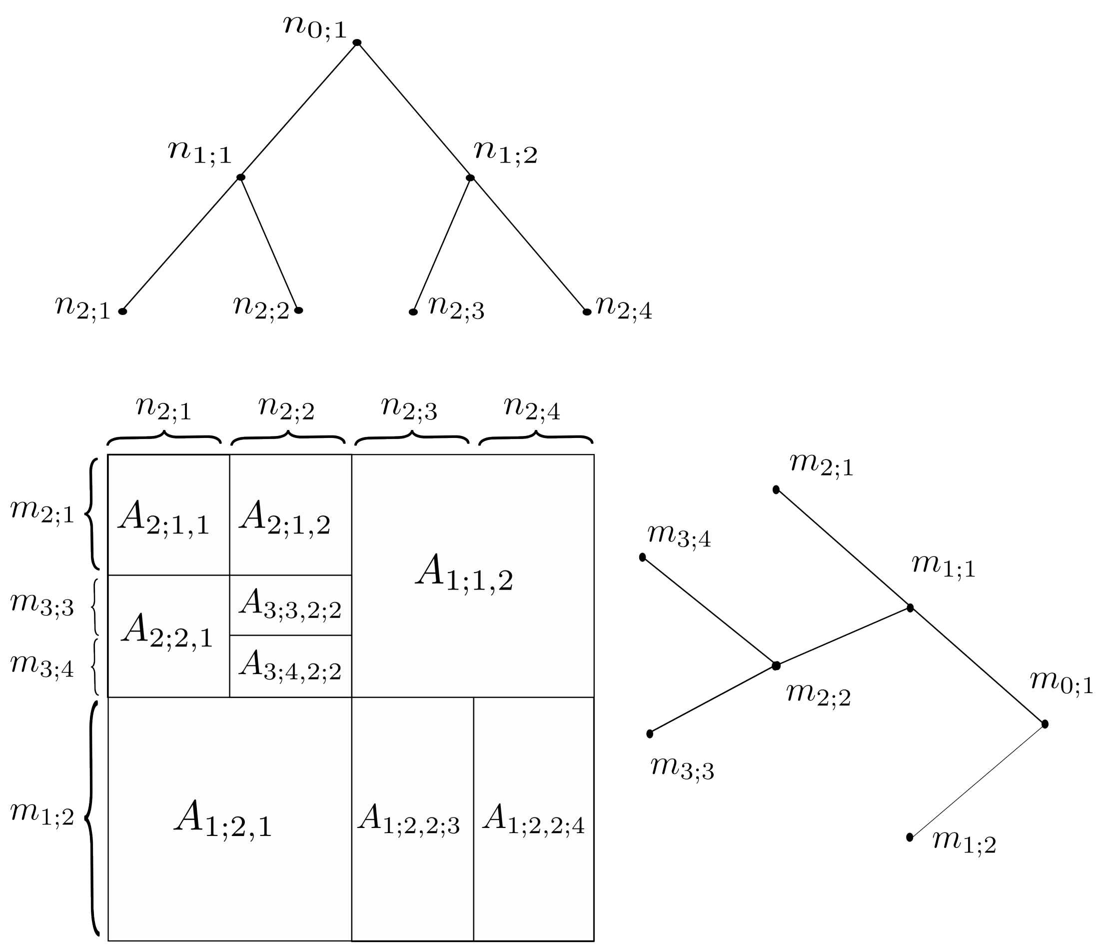
Algorithm 3 Pass 2BU - Computation of Expansion Coefficients ($B_{k;i-1,i}$) Corresponding to Off-Diagonal Blocks

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

Worst Case Memory Consumption for our Algorithm



Example Block Partitioning of a Matrix with Corresponding Partition Trees



Future Work

- Fast Multipole Method (FMM) construction Algorithm
- FMM x FMM
- Application to classical HSS algorithms: HSS Multiply & HSS Solver