Data Structures II

CPSC 482

Dr. Hossain

DSM Project Report

Name
Simon Kraft
Joshua Payne
El Sall

April 2, 2026

Abstract

The Design Structure Matrix (DSM) is a compact representation of task dependencies in large-scale engineering and software systems. Efficient reordering of DSM tasks to minimize feed-back dependencies is a computationally challenging problem with significant practical importance. In this report, we present a multi-stage pipeline for DSM partitioning using the Compressed Sparse Column (CSC) data structure. The pipeline decomposes the DSM into its block lower triangular form (BLTF) through strongly connected component (SCC) decomposition and topological block ordering, intra-block tearing, and bandwidth reduction. Two classical algorithms are used for finding the strongly connected components, Tarjan's algorithm and the Sharir-Kosaraju algorithm. We evaluate both algorithms on a suite of benchmark sparse matrices and compare their performance in terms of running time, number of SCCs identified, and solution quality as measured by the number of feed-back marks and total feed-back distance. Our results demonstrate that the full pipeline achieves an average FBM reduction of 84.15% and TFBD reduction of 94.54% relative to baseline orderings across 10 benchmark matrices, with both SCC algorithms producing identical decompositions and Tarjan's algorithm running approximately 1.77× faster than Kosaraju-Sharir.

Contents

1 Introduction **3**
1.1 Motivation . . . 3
1.2 Design Structure Matrix . . . 3
1.3 DSM Partitioning . . . 4
1.4 Existing Approaches . . . 4

2 Description of Solution **5**
2.1 Compressed Sparse Column Representation . . . 5
2.2 Strongly Connected Component Decomposition . . . 5
2.3 Block Ordering . . . 6
2.4 Tearing . . . 6
2.5 Banding . . . 7
2.6 Permutation Application . . . 8

3 Numerical Results **8**
3.1 Experimental Setup . . . 9
3.2 Stage-wise Optimization Quality . . . 9
3.3 Algorithmic Runtime Comparison . . . 10
3.4 Qualitative Results . . . 10

4 Conclusion **11**

5 Acknowledgments **11**
5.1 Artificial Intelligence Disclosure . . . 11

References **13**

Appendix **15**

1 Introduction

1.1 Motivation

Modern software systems and engineering projects are constantly growing in complexity, often comprising hundreds or even thousands of interdependent tasks that require the collaboration of many participants from diverse backgrounds. Traditional management tools such as Gantt charts and Critical Path Method (CPM) networks are well-suited for modeling sequential and parallel task executions. However, they fail for interdependent tasks, i.e., when the output of one task is needed as input for another task, and vice versa. As these tasks are especially common in complex product development (PD), project managers require tools capable of modeling interdepdencies and identifying a task ordering that minimizes potential conflicts in information flow [16].

1.2 Design Structure Matrix

An effective tool for representing the pairwise information flow between a set of tasks is a square matrix called the *design structure matrix* or *dependency structure matrix* (DSM) [9]. If a project is composed of n tasks, a DSM models the information flow between these tasks in a binary $n \times n$ matrix, where the rows and columns are both labeled by the same ordered list of tasks. Marks in the DSM indicate the relationship between two tasks. The marks in a single row of the DSM show all the tasks that provide input that is needed to perform the task in the corresponding row. Similarly, marks in a single column display all the tasks that receive information from the corresponding task in that column [16]. For instance, the (i, j)-th entry of the DSM is 1, if task i depends on task j, and it is 0 if task i can be completed independent of task j. Furthermore, the diagonal entries do not represent dependencies with other tasks and are therefore often disregarded. Equivalent to its matrix representation, a DSM can also be displayed as a directed graph [4].

If the ordering of tasks along the axes is interpreted as the sequence in which the tasks are executed, then three different types of dependencies can be identified for a pair of tasks [16].

- **Feed-Forward**: Marks that represent the forward transfer of information. Task j supplies information for task i and is scheduled before it, i.e., $j < i$. The pair (i, j) is located below the diagonal. These dependencies are easy to solve as task i just waits for task j to finish.
- **Feed-Back**: Marks that represent the backward transfer of information. Task j supplies information for task i but is scheduled afterwards, i.e., $i < j$. The pair (i, j) is located above the diagonal. This means that the required inputs (task j) are not available at the time of executing task i, making it necessary to make assumptions about the inputs that may not hold if task j is finally executed. In the worst case, the dependent task i needs to be re-executed.
- **Coupled**: Task i and j are mutually dependent and both require input from each other, i.e., the pairs (i, j) and (j, i) both exist in the DSM. These tasks require iteration as neither task can be started first without making assumptions about the output of the other task.

1.3 DSM Partitioning

An important objective from a project manager's perspective is reorder the tasks to reduce the number of feed-back marks to minimize the potential conflicts that could arise during the execution stage. This process is regarded as *partitioning* [13, 17].

In the ideal scenario, the permutation of the rows and columns leads to a lower triangular form in the DSM, i.e., all feed-back marks get eliminated. However, not all DSMs can be permuted to lower triangular form [8, 16]. Especially for complex engineering systems it is highly unlikely that all feed-back marks can be moved underneath the diagonal. If the underlying directed graph describing the DSM contains at least one cycle, the best achievable result is the *block lower triangular form* (BLTF), in which all vertices participating in cycles are gathered into diagonal blocks [8]. Furthermore, the second objective is to minimize the total distance of the remaining feed-back marks from the diagonal, as smaller and more tightly packed diagonal blocks cause fewer tasks to be involved in each iteration cycle, leading to a faster development process [4].

With these two objectives at hand, the DSM partitioning problem can be stated as follows. Let $A \in \{0,1\}^{n\times n}$ be a binary matrix representing the DSM and let $P \in \{0,1\}^{n\times n}$ be a permutation matrix obtained by permuting the columns/ rows of the identity matrix I_n. The symmetrically permuted matrix $A' = P^\top AP$ represents the reordered DSM, where the goal is to find the permutation matrix P that minimizes two quantities [8]:

1. The number of feed-back marks |FBM|, where
$$\text{FBM} = \{a'_{ij} \neq 0 \mid j > i,\ i,j = 1,2,\ldots,n\}$$
2. The total feed-back distance TFBD of feed-back marks from the diagonal
$$\text{TFBD} = \sum_{a'_{ij}\in\text{FBM}} (j-i),$$

1.4 Existing Approaches

Over the years, several algorithmic approaches have been proposed in the literature to solve the DSM partitioning problem [4, 16]. However, most methods share the same structure and only differ in the way they identify *information cycles*, i.e. cycles of two or more mutually dependent tasks. Usually, the partitioning starts by moving tasks with no incoming dependencies (empty rows) to the top of the DSM, and moving tasks with no outgoing dependencies to the bottom of the DSM. Once the these tasks are rearranged, they are removed from the DSM with all their corresponding marks. These steps are repeated until no further reordering is possible [16]. Any remaining task in the DSM must then be part of at least one information cycle, which must be identified explicitly. Three classical methods have been proposed for this [4, 17]:

- **Path Searching**: Every remaining activity has an off-diagonal mark [4]. An activity is arbitrarily chosen and a list of successors is generated by tracing the information flow until a task in encountered twice [11, 13]. All tasks between the first and second occurrence constitute an information cycle and are grouped together [16]. The approach can also be applied recursively to reveal sub-cycles [4].

- **Powers of the Adjacency Matrix**: The binary DSM is raised to the n-th power. A non-zero diagonal entry indicates that the corresponding task can reach itself in n steps, which confirms a cycle [15, 16]. However, finding blocks of coupled activities through this method is computationally expensive, especially for large matrices [4].
- **Depth-First Search**: The most efficient way to identify subsets of coupled activities is Tarjan's depth-first search algorithm [4, 14]. Following the outputs of each task detects dependency paths that cycle back to the same task.

When all information cycles are identified, the involved tasks are grouped together to be treated as a single representative block [16]. Then the procedure restarts on the remaining tasks, which ultimately yields the block lower triangular form of the DSM.

The remainder of this report is structured as follows: Section 2 describes CSC matrix data structure and the algorithms that were used to permute the DSM, including their asymptotic runtime and space complexities. Section 3 presents the results of our numerical experiments on benchmark matrices in terms of solution quality and running time. Section 4 summarizes our findings and provides future research directions of this work. Finally, the Appendix contains graphical representations of the matrices before and after partitioning.

2 Description of Solution

Not every DSM can be permuted to strict lower triangular form, when the underlying graph contains cycles, the best achievable result is the block lower triangular form (BLTF) [8]. We compute a BLTF of the input DSM in six stages: CSC storage, SCC decomposition, topological block ordering, tearing, banding, and permutation application.

2.1 Compressed Sparse Column Representation

Since DSMs are sparse ($\texttt{nnz} \ll n^2$), we store the matrix in compressed sparse column (CSC) format [1, 8] using a column pointer array $\texttt{col_ptr}[0 \dots n]$ and a row array $\texttt{row_ind}[0 \dots \texttt{nnz}-1]$. The nonzero row indices of column j occupy $\texttt{row_ind}[\texttt{col_ptr}[j] \dots \texttt{col_ptr}[j+1)-1]$. Since DSMs are binary, no value array is needed. Reading the DSM as a directed graph $G(A)$, column j lists the out-neighbours of vertex j, so iterating them costs $O(\deg^+(j))$, this is the access pattern required by DFS. Total storage is $O(n + \texttt{nnz})$. The transposed graph needed by Kosaraju-Sharir is also computed in $O(n + \texttt{nnz})$ time.

2.2 Strongly Connected Component Decomposition

Each SCC in the directed graph of the DSM corresponds to a maximal set of mutually dependent tasks that cannot be decoupled by reordering alone. We implement two SCC algorithms, Tarjan's and Kosaraju-Sharir, which share an iterative DFS over the CSC structure, using a stack instead of recursion. Following Erickson's DFS framework with PreVisit and PostVisit hooks [5], our implementation adds PreEdge and PostEdge callbacks to handle edge-level events.

Tarjan's Algorithm. Tarjan's algorithm [14] finds all SCCs in a single DFS pass. Each vertex v gets a starting time $v.pre$ and a value $low(v)$: the smallest starting time reachable from v by tree edges followed by at most one non-tree edge [5]. On first visit, v is pushed onto a stack S. When an edge $v \to w$ leads to a vertex still on S, we set $low(v) \leftarrow \min\{low(v), w.pre\}$. After returning from a child w, we propagate $low(v) \leftarrow \min\{low(v), low(w)\}$. When a vertex finishes with $low(v) = v.pre$, it is the root of a sink component [5]: all vertices above it on S are popped to form the SCC. Each vertex is pushed and popped exactly once, so the algorithm runs in $O(V + E)$ time and $O(V)$ auxiliary space [5, 14]. Pseudocode is given in Erickson [5] (Figure 6.19).

Kosaraju-Sharir Algorithm. The Kosaraju-Sharir algorithm [12] uses two DFS passes. Phase 1 runs DFS on the original graph, pushing each vertex onto a stack at finish time. Phase 2 builds $rev(G)$ and runs DFS in reverse postorder; each tree yields exactly one SCC [5]. Both passes give $O(V + E)$ time and $O(V + E)$ space [5, 12]. Pseudocode is in Erickson [5] (Figure 6.16).

2.3 Block Ordering

The condensation $scc(G)$ contracts each SCC to a single vertex [5] and is stored as a new CSC matrix. Since it is always a DAG, we topologically sort it with Kahn's algorithm [10] to determine the block order. Both steps run in $O(k + m')$ time, where k is the number of SCCs and m' the number of inter-component edges [10].

2.4 Tearing

Tearing selects feed-back marks whose removal yields a lower triangular block [16]. We approximate the minimum feedback arc set within each non-trivial SCC using Algorithm GR of Eades et al. [3] (Algorithm 1). For each SCC with k vertices, the subgraph is extracted into successor/predecessor lists with in-degree (d^-) and out-degree (d^+) arrays.

Algorithm 1 Algorithm GR [3]

$s_1 \leftarrow [\,], \quad s_2 \leftarrow [\,]$
while $G \neq \emptyset$ **do**
  while G has a sink **do**
    choose a sink u; remove u from G; $s_2 \leftarrow u \cdot s_2$
  end while
  while G has a source **do**
    choose a source u; remove u from G; $s_1 \leftarrow s_1 \cdot u$
  end while
  if $G \neq \emptyset$ **then**
    choose u with maximum $\delta(u) = d^+(u) - d^-(u)$
    remove u from G; $s_1 \leftarrow s_1 \cdot u$
  end if
end while
return $s_1 \cdot s_2$

Edges that go backward in the returned ordering are the feedback arcs. Algorithm GR runs in $O(m)$ time when the max-δ selection at line 10 uses a bucket structure [3]. However, our implementation uses a linear scan instead, giving $O(k^2+m)$ time and $O(k+m)$ space per SCC. Singleton SCCs are returned unchanged.

2.5 Banding

Banding reorders tasks within each non-trivial SCC ($|C| > 2$) to push feed-back marks closer to the diagonal, minimizing TFBD [8].

Reverse Cuthill-McKee Ordering. The RCM algorithm was originally designed to reduce the bandwidth of symmetric sparse matrices [2]. Applied to an SCC subgraph, it moves nonzeros near the diagonal, which directly reduces TFBD. A pseudoperipheral starting vertex is selected via the George-Liu algorithm [7], which runs BFS rounds. From this vertex, a Cuthill-McKee BFS [2] visits neighbours in ascending degree order, the result is then reversed to yield the RCM ordering [6].

Adjacent-Swap Refinement. Both the torn and RCM orderings are further optimized by a greedy adjacent-swap heuristic (Algorithm 2). Each pass scans adjacent pairs and accepts a swap only if it strictly decreases TFBD without increasing FBM. The procedure stops when no improvement is found or after 20 passes.

Algorithm 2 Adjacent-Swap Refinement

Require: ordering $\sigma = [\sigma_0, \ldots, \sigma_{k-1}]$, directed edge set E, max passes P
Ensure: refined ordering σ

1: $pos[\sigma_i] \leftarrow i$ for all i ▷ inverse permutation
2: $inc[v] \leftarrow \{e \in E \mid v \in e\}$ for all v ▷ incidence lists
3: **for** $pass = 1, \ldots, P$ **do**
4: $improved \leftarrow$ **false**
5: **for** $i = 0, \ldots, k-2$ **do**
6: $a \leftarrow \sigma_i, \quad b \leftarrow \sigma_{i+1}$
7: $\Delta\text{FBM} \leftarrow 0, \quad \Delta\text{TFBD} \leftarrow 0$
8: **for** each edge $(s, d) \in inc[a] \cup inc[b]$ **do**
9: compute old and new positions of s, d under the swap $a \leftrightarrow b$
10: update ΔFBM and ΔTFBD from old vs. new contribution
11: **end for**
12: **if** $\Delta\text{TFBD} < 0$ **and** $\Delta\text{FBM} \leq 0$ **then**
13: swap $\sigma_i \leftrightarrow \sigma_{i+1}$; update pos
14: $improved \leftarrow$ **true**
15: **end if**
16: **end for**
17: **if** $\neg\, improved$ **then break**
18: **end if**
19: **end for**
20: **return** σ

The refined ordering with fewer feed-back marks is kept, ties are broken by lower TFBD. If the unrefined RCM ordering increased FBM, it is discarded. Algorithm 3 summarizes the full procedure.

Algorithm 3 Banding an SCC

Require: SCC vertices C with $|C| = k > 2$, DSM matrix A
Ensure: banded ordering of C
build local subgraph G from A restricted to C
$\sigma_{\text{torn}} \leftarrow$ identity ordering (from tearing)
$\hat{\sigma}_{\text{torn}} \leftarrow \textsc{SwapRefine}(\sigma_{\text{torn}}, E(G), 20)$ ▷ Algorithm 2
find pseudoperipheral vertex r in G ▷ George-Liu [7]
$\sigma_{\text{rcm}} \leftarrow \text{RCM}(G, r)$ ▷ Cuthill-McKee [2], reversed [6]
if $\text{FBM}(\sigma_{\text{rcm}}) \leq \text{FBM}(\sigma_{\text{torn}})$ **then**
    $\hat{\sigma}_{\text{rcm}} \leftarrow \textsc{SwapRefine}(\sigma_{\text{rcm}}, E(G), 20)$
    if $\hat{\sigma}_{\text{rcm}}$ has fewer FBM, or equal FBM and lower TFBD **then**
        return $\hat{\sigma}_{\text{rcm}}$
    end if
end if
return $\hat{\sigma}_{\text{torn}}$

RCM runs in $O(k + m)$ time [2, 6]. Overall, banding requires $O(k + m)$ time and space per SCC.

2.6 Permutation Application

The banded block orderings are concatenated in topological order to form a permutation π in $O(n)$ time. The permutation $A' = P^\top AP$ is applied in CSC format by computing π^{-1} in $O(n)$ time, then remapping each edge (i, j) to $(\pi^{-1}(i), \pi^{-1}(j))$ in $O(n + \texttt{nnz})$ time. Row indices within each column are then sorted, if column j has d_j entries, this step costs $O(\sum_j d_j \log d_j) \leq O(\texttt{nnz} \log n)$ since $d_j \leq n$. The total is $O(n + \texttt{nnz} \log n)$ time and $O(n + \texttt{nnz})$ space. The FBM count and total feed-back distance are computed on both the original and permuted DSMs in $O(\texttt{nnz})$ time.

3 Numerical Results

This section evaluates the proposed DSM reordering method on the benchmark matrices included in the project repository. In particular, we measure both *solution quality* and *computational cost* across the stages introduced in Section 2: (i) SCC decomposition and topological block ordering (`TopoOnly`), (ii) intra-block tearing, and (iii) intra-block banding refinement. The final reordered DSM is obtained through the symmetric permutation $A' = P^\top AP$, and all metrics are computed on A'.

3.1 Experimental Setup

All experiments were executed with the benchmark mode of our implementation (`./out/main -bench`) using the same software and hardware environment for all matrices. All runs reported in this section were executed on an Apple M2 system (8 GB RAM), compiled with Apple LLVM 17 under `-O3 -std=c++17`. The M2 cache hierarchy consists of a 64 KB L1 data cache, 128 KB L1 instruction cache, and 4 MB L2 cache per cluster. For each timed algorithm, we use 2 warm-up runs and 5 timed runs, and report the mean, standard deviation, minimum, median, and maximum runtime in microseconds.

The benchmark suite contains 10 sparse matrices with dimensions ranging from $n = 11$ to $n = 27{,}770$ and nonzero counts from 4 to 352,807. This span allows us to examine both small example instances and large, complex graphs that represent realistic dependency systems. In addition to timing, we export matrix-level quality statistics and stage-wise optimization metrics to `out/optimization_metrics.csv`, from which the plots in Appendix Figures 1–4 are generated.

3.2 Stage-wise Optimization Quality

The first observation is that the optimization stages consistently reduce feed-back structure in most matrices, often by a large margin. Averaged across the complete benchmark suite, the final banded ordering reduces the number of feed-back marks by **84.15%** and the total feed-back distance by **94.54%** relative to the baseline ordering. These reductions show that the proposed sequence is effective not only at reducing the count of above-diagonal dependencies, but also at moving the remaining ones much closer to the diagonal.

As shown in Appendix Figure 1, `TopoOnly` already provides a substantial structural benefit for many matrices (average FBM reduction: 76.18%), while tearing accounts for the major additional gain (average FBM reduction after tearing: 84.12%). Banding then provides a smaller but measurable final improvement (84.15% overall), indicating that its contribution is mainly a local refinement step rather than the main source of global FBM reduction.

The TFBD behavior in Appendix Figure 2 adds further detail. On average, `TopoOnly` produces very strong TFBD reduction (95.21%), and tearing may occasionally increase TFBD relative to `TopoOnly` while still remaining far better than baseline. This behavior is expected: tearing mainly targets feedback orientation inside SCCs and does not directly optimize global diagonal distance. Banding then recovers and improves local compactness, yielding the strongest final TFBD values in most cases.

Table 1 summarizes representative matrices from small, medium, and large scales. For `California`, FBM decreases from 6063 to 192 and TFBD from 18,810,564 to 669. For the largest instance, `HEP-th-new`, FBM decreases from 104,435 to 8,281 and TFBD from 1,756,120,879 to 10,054,656. Even for structurally difficult instances such as `wb-cs-stanford`, the final stage reduces FBM by 57.59% and TFBD by 79.71%, which remains substantial in absolute terms. Visualizations of selected benchmark matrices before and after the partitioning process are provided in the Appendix.

Table 1: Representative stage-wise quality metrics (baseline to final).

Matrix	n	nnz	FBM_0	FBM_f	TFBD_0	TFBD_f
Tina_AskCal	11	29	12	4	32	7
California	9664	16150	6063	192	18810564	669
wb-cs-stanford	9914	36854	20131	8537	8773515	1780060
HEP-th-new	27770	352807	104435	8281	1756120879	10054656

3.3 Algorithmic Runtime Comparison

Runtime behavior is evaluated with the benchmark table and scaling plots generated from `out/benchmark.csv`. The measured results indicate that both SCC implementations are efficient at all tested scales, with Tarjan consistently faster than Sharir-Kosaraju on average in our experiments. Across the full benchmark set, mean runtime is approximately 560.76 μs for Tarjan and 992.26 μs for Kosaraju, i.e., a ratio of about 1.77$\times$ in favor of Tarjan.

The scaling trends in Appendix Figure 3 are compatible with the expected near-linear behavior for graph-traversal steps over sparse structures. As anticipated, stages with heavier local refinement logic (especially banding) incur higher absolute runtime than pure traversal-based stages, but this additional cost is justified by the quality gains shown earlier. In practical terms, the optimization trade-off is favorable: banding is computationally heavier, yet it delivers the final quality improvements needed for cleaner diagonal concentration and clearer block structure in the reordered DSM.

Finally, Appendix Figure 4 summarizes the global pattern: topological block ordering provides strong early improvements, tearing drives the main FBM reduction within SCCs, and banding supplies consistent final polishing, particularly in TFBD compactness. Therefore, the empirical evidence supports using the full multi-stage method instead of any single-stage reordering strategy when the objective is robust quality improvement across different DSM instances.

3.4 Qualitative Results

To complement the quantitative metrics, Figures 5 and 6 in the Appendix show the sparsity structure of four benchmark matrices before and after DSM partitioning. For each matrix, the orange shaded blocks indicate the strongly connected components of the permuted DSM, and the statistics |FBM| and TFBD are reported for both the original and permuted matrix in the top-right corner of each plot.

The four matrices were selected to illustrate a range of structural characteristics. `California` is a large matrix (n = 9,664) that achieves a near-perfect partitioning, with no large SCC blocks visible and almost all feed-back marks eliminated. In contrast, `wb-cs-stanford` (n = 9,914) presents a more structurally difficult instance. Although the partitioning is effective overall, one large SCC block persists along the diagonal, retaining a significant number of feed-back marks with considerable distance from the diagonal, which explains the relatively high residual FBM and TFBD values for this matrix (see Table 1). In comparison the smaller matrix `Tina_AskCal` (n = 11) shows a clean block structure with well-separated components, while

`GD99_c` ($n = 105$) is particularly illustrative in that its permuted form reveals a large number of trivial SCCs, that only contain one isolated node, and two-node SCCs. Furthermore, it contains two larger coupled bocks.

4 Conclusion

We presented a multi-stage method for reordering Design Structure Matrices to minimize feedback marks and total feed-back distance. The approach proceeds through six stages: CSC storage, SCC decomposition, topological block ordering, intra-block tearing via the greedy Algorithm GR, bandwidth reduction using reverse Cuthill-McKee ordering, and adjacent-swap refinement. Two classical SCC algorithms: Tarjan's and Kosaraju-Sharir—were implemented over the CSC data structure and evaluated on a benchmark suite of 10 sparse matrices ranging from $n = 11$ to $n = 27{,}770$.

Our numerical experiments demonstrate that the full pipeline achieves an average FBM reduction of 84.15% and an average TFBD reduction of 94.54% relative to the baseline ordering. Each stage contributes meaningfully: topological block ordering alone accounts for roughly 76% of the FBM reduction on average, tearing drives additional gain, and banding supplies final refinement, particularly in diagonal compactness. Both SCC algorithms produce identical decompositions, with Tarjan's algorithm running approximately 1.77× faster than Kosaraju-Sharir in our experiments. Runtime's across the benchmark suite are consistent with the expected near-linear behavior for graph-traversal-based algorithms over sparse structures.

Several directions for future work are worth considering. First, the tearing stage currently relies on the greedy heuristic of Eades et al., which does not guarantee a minimum feedback arc set, replacing it with a more sophisticated approach could yield further FBM reductions. Second, the current implementation is sequential; parallelizing the independent per-SCC tearing and banding stages would reduce wall-clock time on large matrices with many non-trivial SCCs. Finally, extending the framework to weighted DSMs, where marks carry numerical values reflecting the degree of coupling between tasks, would better capture real-world engineering dependencies and enable optimization of weighted feed-back cost rather than simple mark counts.

5 Acknowledgments

5.1 Artificial Intelligence Disclosure

We hereby declare that we made limited use of generative artificial intelligence during the preparation of this project. Specifically, Claude (Opus 4.6 & Sonnet 4.6) and ChatGPT (GPT-4o) for the following:

1. **Understanding algorithms and terminology.** We asked questions to learn about topics related to DSM reordering. Prompts used:
    - "What are possible solutions for the feedback arc set problem used in DSM tearing, and what are some articles related to this problem?"

- "What is the connection between Reverse Cuthill-McKee algorithm and banding in DSMs?"
- "What is the difference between tearing and banding in the context of DSMs?"
- "Explain how the Eades-Lin-Smyth algorithm works with DSM tearing."

2. **Writing Python plotting scripts.** We asked for help writing python scripts to produce the plots and tables in this report based on the output of our C++ code, such as grouped bar charts comparing FBM and TFBD reductions. All generated code was reviewed and modified by the authors before use.

All C++ algorithm code and report text were written by the authors without AI assistance.

References

[1] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. 2nd. SIAM, 1994 (cit. on p. 5).

[2] E. Cuthill and J. McKee. "Reducing the Bandwidth of Sparse Symmetric Matrices". In: *Proceedings of the 24th National Conference of the ACM*. 1969, pp. 157–172 (cit. on pp. 7, 8).

[3] P. Eades, X. Lin, and W. F. Smyth. "A Fast and Effective Heuristic for the Feedback Arc Set Problem". In: *Information Processing Letters* 47.6 (1993), pp. 319–323 (cit. on pp. 6, 7).

[4] S. Eppinger and T. Browning. *Design Structure Matrix Methods and Applications*. May 2012. ISBN: 9780262301428 (cit. on pp. 3–5).

[5] J. Erickson. *Algorithms*. Self-published, 2019. URL: `http://jeffe.cs.illinois.edu/teaching/algorithms/` (cit. on pp. 5, 6).

[6] A. George. "Computer Implementation of the Finite Element Method". PhD thesis. Stanford University, 1971 (cit. on pp. 7, 8).

[7] A. George and J. W. H. Liu. "An Implementation of a Pseudoperipheral Node Finder". In: *ACM Transactions on Mathematical Software* 5.3 (1979), pp. 284–295 (cit. on pp. 7, 8).

[8] S. Hossain. "Hints on Data Structure and Algorithms for Term Project". Course document, CPSC 482/682, University of Northern British Columbia. 2026 (cit. on pp. 4, 5, 7).

[9] S. Hossain. "Term Project for CPSC482/682 Data Structures II". Course document, CPSC 482/682, University of Northern British Columbia. 2026 (cit. on p. 3).

[10] A. B. Kahn. "Topological Sorting of Large Networks". In: *Communications of the ACM* 5.11 (1962), pp. 558–562 (cit. on p. 6).

[11] R. Sargent and A. Westerberg. "SPEED-UP in Chemical Engineering Design". In: *Transactions of the Institution of Chemical Engineers* 42 (Jan. 1964), p. 190 (cit. on p. 4).

[12] M. Sharir. "A strong-connectivity algorithm and its applications in data flow analysis". In: *Computers & Mathematics with Applications* 7.1 (1981), pp. 67–72. ISSN: 0898-1221. URL: `https://www.sciencedirect.com/science/article/pii/0898122181900080` (cit. on p. 6).

[13] D. V. Steward. "The design structure system: A method for managing the design of complex systems". In: *IEEE Transactions on Engineering Management* EM-28.3 (1981), pp. 71–74 (cit. on p. 4).

[14] R. Tarjan. "Depth-First Search and Linear Graph Algorithms". In: *SIAM Journal on Computing* 1.2 (1972), pp. 146–160. eprint: `https://doi.org/10.1137/0201010`. URL: `https://doi.org/10.1137/0201010` (cit. on pp. 5, 6).

[15] J. N. Warfield. "Binary Matrices in System Modeling". In: *IEEE Transactions on Systems, Man, and Cybernetics* SMC-3.5 (1973), pp. 441–449 (cit. on p. 5).

[16] A. Yassine. "An Introduction to Modeling and Analyzing Complex Product Development Processes Using the Design Structure Matrix (DSM) Method". In: 2001. URL: https://api.semanticscholar.org/CorpusID:13887392 (cit. on pp. 3–6).

[17] A. Yassine, D. Falkenburg, and K. Chelst. "Engineering design management: An information structure approach". In: *International Journal of Production Research* 37.13 (1999), pp. 2957–2975. eprint: https://doi.org/10.1080/002075499190374. URL: https://doi.org/10.1080/002075499190374 (cit. on p. 4).

Appendix

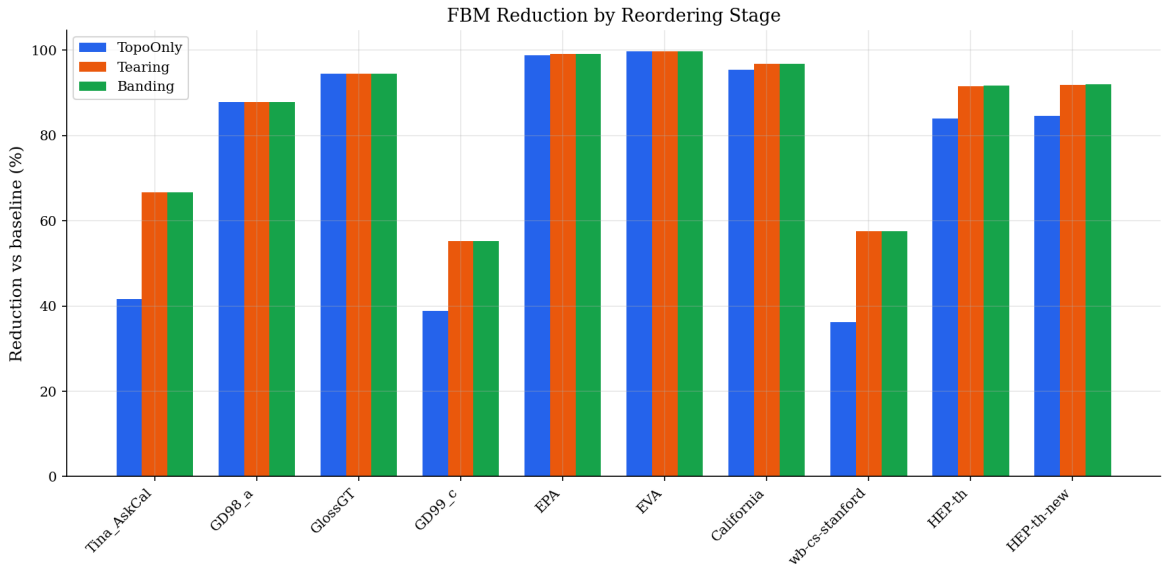


Figure 1: Percentage reduction in FBM at each stage relative to the baseline ordering.



Figure 2: Percentage reduction in TFBD at each stage relative to the baseline ordering.

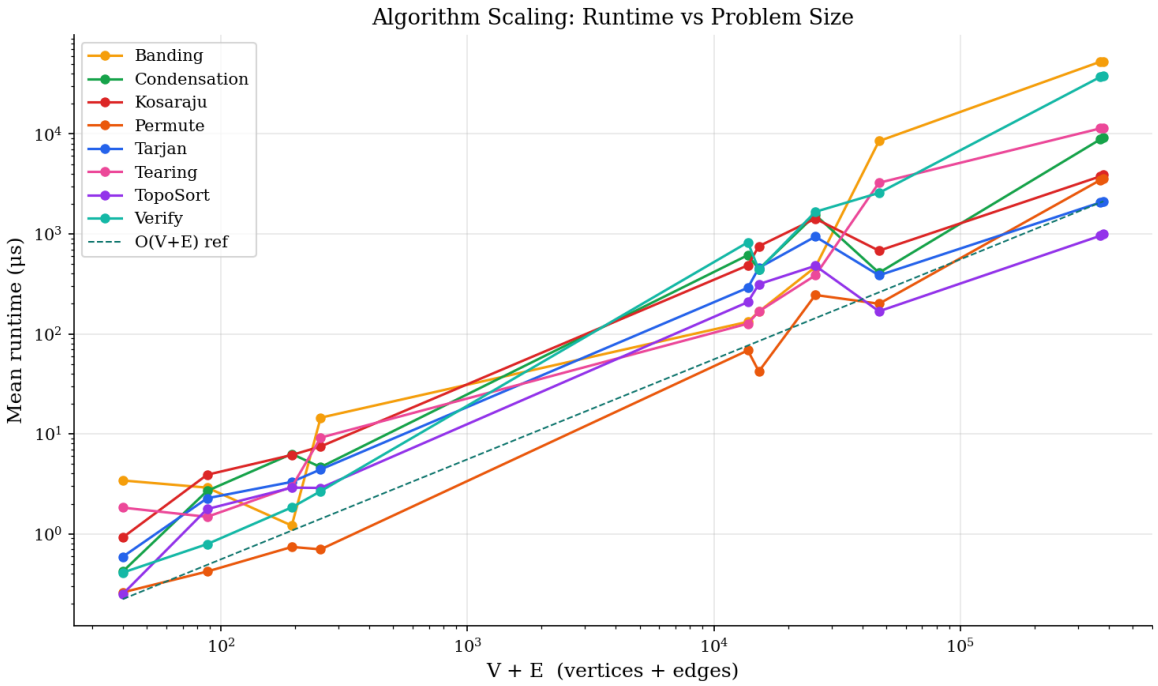


Figure 3: Log-log runtime scaling versus problem size $(V + E)$ for benchmarked algorithms.

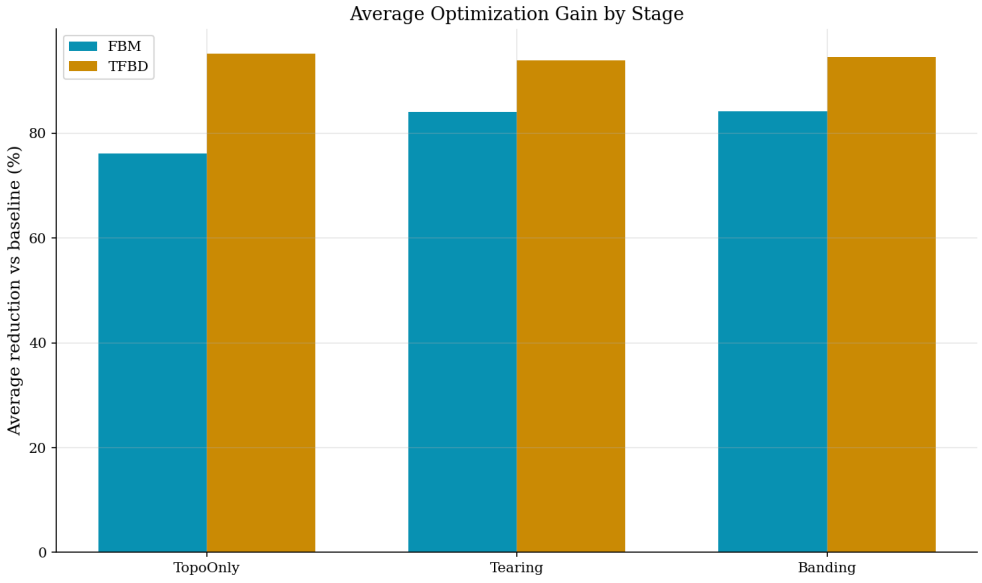


Figure 4: Average FBM and TFBD reduction by stage over all benchmark matrices.

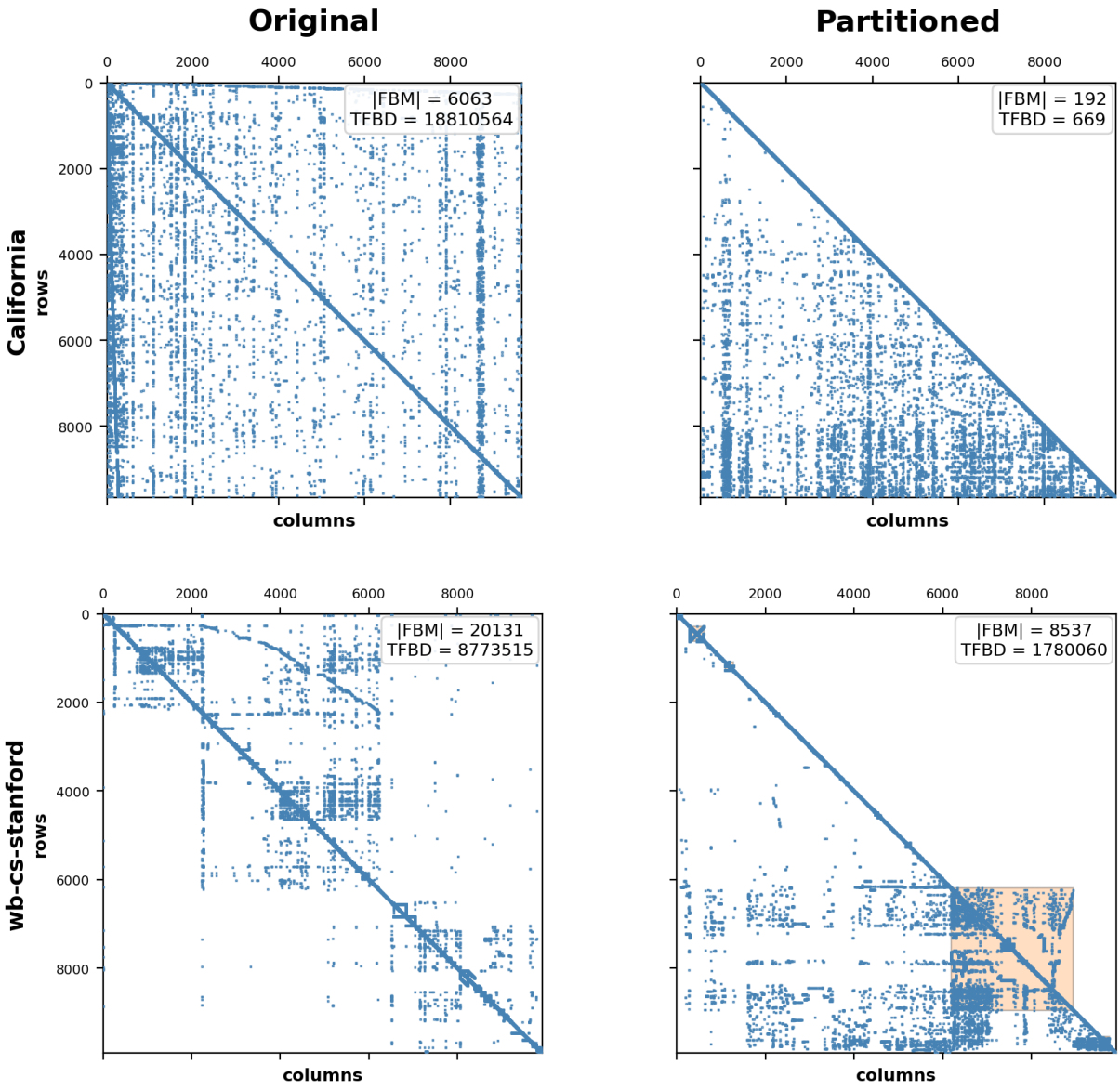


Figure 5: DSM partitioning results (1/2).

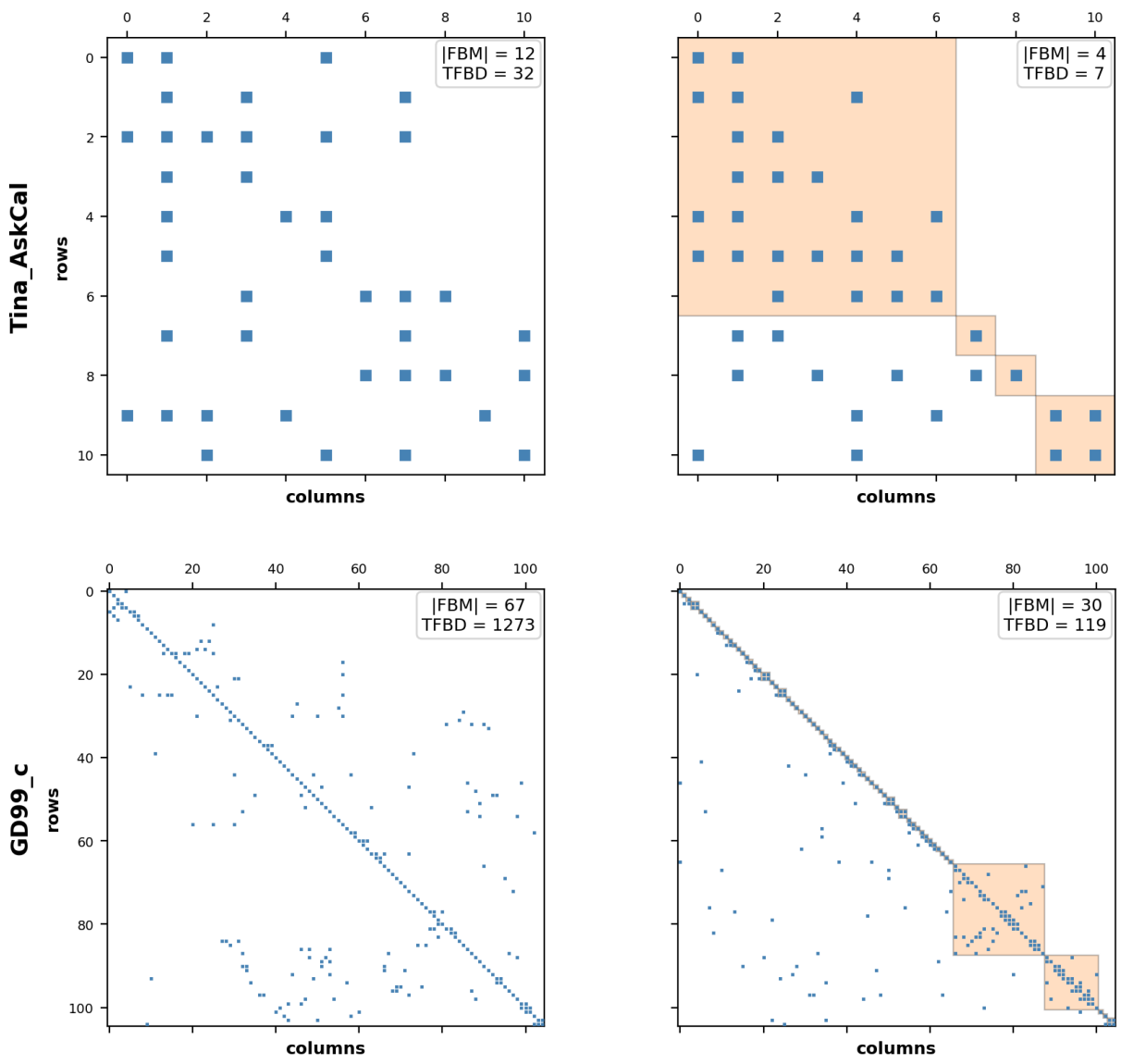


Figure 6: DSM partitioning results (2/2).