Matrix Reordering by Hypertree Decomposition

Wilfried Gansterer[*]
Thomas Korimort[†]

AURORA TR2003-19

[*]Department of Computer Science and Business Informatics
University of Vienna
Lenaugasse 2/8, A-1080 Vienna, Austria
e-mail: `wilfried.gansterer@univie.ac.at`

[†]Institute for Applied Mathematics and Numerical Analysis
Vienna University of Technology
Wiedner Hauptstrasse 8-10, A-1040 Wien, Austria
e-Mail: `korimort@aurora.anum.tuwien.ac.at`

Supported by the SFB AURORA of the Austrian Science Fund FWF.

Abstract

Recently, in the area of constraint satisfaction a new hypergraph decomposition technique has been invented by Gottlob et al. [7] called *hypertree decomposition*.

In this report hypertree decomposition is applied to matrix reordering. A new structure of a given matrix is obtained by reording rows and columns according to a special hypertree decomposition computed for a hypergraph related to the non-zero structure of the matrix.

A comparison with well-known matrix reordering techniques like reverse Cuthill-McKee and approximate minimum degree reordering is given.

Contents

Introduction

In this report standard reordering techniques for sparse matrices (see Chapter 1) are compared with a new technique based on hypertree decomposition (see Chapter 2).

Given a sparse matrix A, an orthogonal matrix $C^\top$ of column permutations and an orthogonal matrix R of row permutations are constructed. These row and column permutations define a reordering of the matrix A:

$$A' = RAC^\top.$$

If A is symmetric and the reordering is required to preserve symmetry, then $R = C$ has to be chosen.

Objectives

This report compares the effectiveness of various techniques for reducing the bandwidth of a given sparse matrix. A forthcoming project will also investigate a different, but related objective—decoupling or "almost" decoupling of a given sparse matrix. Efficient approaches to achieve one or even both of these objectives can be utilized in many situations, such as in a new framework for computing approximate spectral information of symmetric matrices [2].

Bandwidth Reduction

The *bandwidth* b of a matrix A is defined as

$$b := \max\left\{|i-j|, |A(i,j)| \neq 0\right\}, \quad i = 1, 2, \ldots, n, \quad j = 1, 2, \ldots, n.$$

The bandwidth of the reordered matrix A' is b'. A bandwidth reduction is achieved, whenever $b' < b$. Obviously, the goal is to reduce the bandwidth as much as possible, such that $b' \ll b$.

Chapter 1

Matrix Reordering Techniques

Many reordering techniques for sparse matrices have been developed, such as minimum degree, approximate minimum degree reordering [6], etc. This report deals primarily with techniques for reducing the bandwidth or the profile of a given matrix, and relevant methods are summarized in the following.

Not all reordering methods are primarly designed for reducing the bandwidth of a sparse matrix. Some of them are intended to reduce the *profile* p of a symmetric sparse matrix, i. e.,

$$\begin{aligned} f &:= \sum_{i=1}^{n} (i - g(i)), \quad \text{where} \\ g(i) &:= \min \{ j | A(i,j) \neq 0 \vee j = i \}, \quad i = 1, 2, \ldots, n. \end{aligned}$$

Although a profile reduction usually leads to a reduction in bandwidth, it is possible that a near minimum profile corresponds to a large bandwidth, for example, if there is a single row with a nonzero element far away from the diagonal.

With respect to the new matrix reordering method based on hypertree decomposition (hypertree reordering) we can define a *single sided profile* as follows:

$$\begin{aligned} f &:= \sum_{i=1}^{n} (g(i) - i), \quad \text{where} \\ g(i) &:= \max \{ j | A(i,j) \neq 0 \vee j = i \}, \quad i = 1, 2, \ldots, n. \end{aligned}$$

1.1 Cuthill-McKee Reordering

The Cuthill-McKee (CMK) and the reverse Cuthill-McKee (rCMK) algorithm [3, 5, 10, 11] are based on the simple technique of traversing the adjacency graph of the matrix. CMK and rCMK produce a row respectively column reordering of the matrix by traversing the nodes of the graph in a breadth first manner. The nodes of each level set are ordered with respect to their degree.

1.2 Column Approximate Minimum Degree Ordering

The *approximate minimum degree* (AMD) ordering algorithm [1] for a symmetric matrix A produces a reordering P such that factorizing the resulting permuted matrix produces much less fill-in than factorizing the original matrix A. It is based on the quotient graph for matrix factorization that allows to obtain computationally cheap bounds for the minimum degree of the nodes of the corresponding graph.

Column approximate minimum degree (COLAMD) ordering is a column ordering method based on a symbolic LU factorization of a non-symmetric matrix A [4]. The column ordering Q resulting from COLAMD, which is based solely on the nonzero pattern of A, is designed such that AQ tends to have sparser LU factors than A, regardless of the subsequent choice of the row interchanges P from standard partial pivoting. Also, the Cholesky factorization of $(AQ)^\top AQ$ tends to be sparser than that of $A^\top A$.

Although minimum degree ordering algorithms in general do not result in a bandwidth reduction of A, the corresponding reorderings are shown in Chapter 3 for several reasons: (i) COLAMD is a onesided reordering and therefore it does not preserve symmetry (similar to the more efficient variant of hypertree reordering); (ii) it forms the basis of the *symmetric approximate minimum degree* (SYMAMD) ordering, which is not applicable to the test cases of this report that are not positive definite; (iii) in further research activities, we are planning to investigate the applicability of hypertree reordering to the computation of fill-in reducing reorderings when factorizing non-symmetric linear systems.

Chapter 2

Hypertree Reordering

2.1 Hypertree Decomposition

Hypertree decomposition is a decomposition method for hypergraphs, which was invented by Gottlob et al. [7].

Definition 2.1.1 (Hypertree Decomposition) *A hypertree decomposition of a hypergraph $G = (V, E)$ is a triple (T, χ, λ) such that:*

1. *$T = (N, A)$ is a (rooted) tree,*
2. *T_n is the subtree from T, which is rooted at n,*
3. *$\chi : N \to P(V)$,*
4. *$\lambda : N \to P(E)$,*
5. *$\forall r \in E \quad \exists n \in N : r \subseteq \chi(n)$,*
6. *$\forall v \in V$ the subgraph of T induced by $\{n \in N | v \in \chi(n)\}$ is a subtree of T,*
7. *$\forall n \in N : \chi(n) \subseteq \bigcup_{r \in \lambda(n)} r$,*
8. *$\forall n \in N : \chi(T_n) \cap \bigcup_{r \in \lambda(n)} r = \chi(n)$.*

The width of a hypertree decomposition of G is $w = \max_{n \in N} |\lambda(n)|$.

One can see from the definition that a hypertree decomposition is essentially a decomposition tree which additionally satisfies Axiom 8. This axiom has been added to the definition of a decomposition tree to be able to prove the polynomiality of the fixed width algorithm to compute hypertree decompositions of minimal width w.

The exact algorithm for computing hypertree decomposition has only polynomial running time, if one fixes the width to a constant k in advance. Then the space and time complexity of the algorithm is about $\Omega(n^{ck})$ where n is the problem size and c is a constant. This motivates the development of a heuristics to compute hypertree decompositions as is done in [8]. An example for a hypertree decomposition is given in Figs. 2.1 and 2.2. The hypergraph to be decomposed is given in Fig. 2.1. A hypertree decomposition for the hypergraph in Fig. 2.1 is given in Fig. 2.2. Note that for the sake of readability, all variables from the hyperedges not in the χ-set are replaced by an underscore.

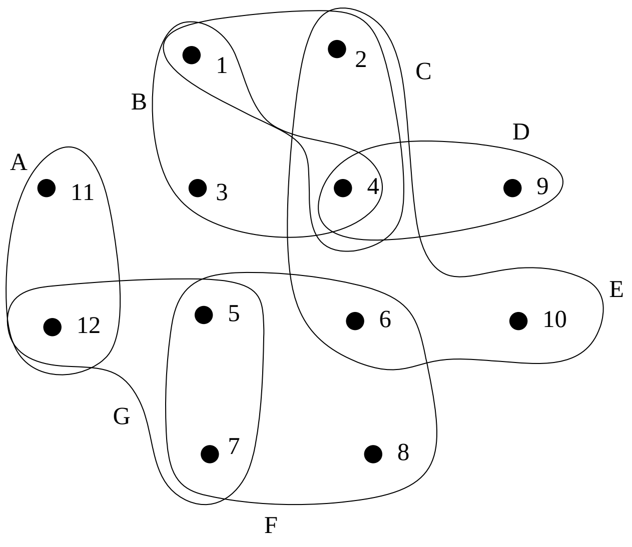

Figure 2.1: The hypergraph to be decomposed by a hypertree decomposition.

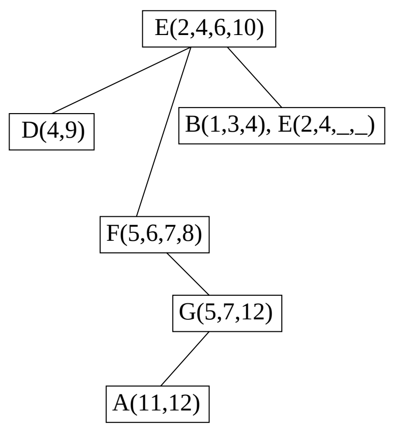

Figure 2.2: A hypertree decomposition of the hypergraph in Fig. 2.1.

2.2 Computing Matrix Reorderings from Hypertree Decompositions

This section deals with methods for computing a matrix reordering from a hypertree decomposition. Given a hypertree decomposition the methods proceed recursively from the root as follows. Pass to the root the set of all hyperedges

```
procedure $compute\_row\_permutation(n, notyetchosen, permulist)$
$n$ ... a hypernode of the hypertree under consideration.
$notyetchosen$ ... contains the rows (hyperedges)
        that have not yet been chosen.
$permulist$ ... contains sets of rows (hyperedges)
        that constitute the row permutation.
Build the set $\mu$ that contains the hyperedges in $notyetchosen$
        that are covered by $\chi(n)$.
Append $\mu$ to $permulist$.
Let $notyetchosen = notyetchosen - \mu$.
for each son $n_i$ of $n$ do
        Call $compute\_row\_permutation(n_i, notyetchosen, permulist)$.
next
end
```

Figure 2.3: Recursive procedure for computing a row partitioning.

of the hypergraph. Collect all hyperedges contained in the χ-set of the root and remove them from the set of hyperedges. Pass the remaining set of hyperedges to the sons of the root and construct in this way an ordered partition of all the hyperedges, where the order of the partition set is given by the depth first traversal of the hypertree decomposition. Concerning the matrix under consideration we obtain a partitioning of the rows into blocks. Since the partition sets are ordered, a row permutation up to row order inside one row partition is introduced. For each row partition set we determine a corresponding column partition set. This set is computed during the computation of the row partition set and consists of all columns that correspond to non-zero entries in one of the rows in the row partition set and have not yet been assigned to a row permutation set.

In Fig. 2.3 the construction of the row partition *permulist* for a hypergraph $H = (V, E)$ from a hypertree T is shown. To initiate the construction of the row partition we call

$$notyetchosen = E.$$
$$permulist = \emptyset.$$
$$compute\_row\_permutation(root(T), notyetchosen, permulist).$$

After processing this sequence of calls, *permulist* is a set of sets containing hyper-

```
procedure $compute\_column\_permutation(permulist)$
$permulist$ ... contains sets of rows (hyperedges)
        that constitute the row permutation.
Let $notyetchosen$ contain all columnindices (hypernodes).
Let $permulist2$ be the empty set to be intended
        to contain sets of columnindices (hypernodes).
for each $\mu$ in $permulist$ do
        Let $\nu$ be the set of hypernodes in $notyetchosen$
                that occur in elements of $\mu$.
        Let $notyetchosen = notyetchosen - \nu$.
        Append $\nu$ to $permulist2$ and
        let $\nu$ be the column partition corresponding to $\mu$.
next
return permulist2.
end
```

Figure 2.4: Procedure for computing a column partition set.

edges (rows), which will be used to compute a column partition. The computation of the column partition works as given in Fig. 2.4.

By doing the following calls to procedures we complete our set of row partitions with a set of corresponding column partitions:

$$permulist2 = compute\_column\_partition(permulist).$$

By using the above procedures an ordered partitioning of rows and columns of the matrix is returned. This partitioning may be interpreted as a unique permutation modulo the exchange of rows and columns inside a partition set.

2.3 Comparison

How does the effort for computing hypertree reorderings relate to the effort for computing (r)CMK? The effort for (r)CMK usually consists in a short time for finding a point to start the breadth first search from, followed by the breadth first search itself. This is rather cheap compared to the effort for computing a heuristic hypertree decomposition. One can see in the section “Reorderings Produced” tables with times and options used by the heuristic for computing hypertree decompositions. By adjusting the optional settings of the heuristic, one can choose between computing times of days or minutes. For our examples it was tried to reduce the computing time to minutes on common PC hardware. From extended experiments it turned out, that either the algorithm is able to find a good hypertree decomposition in a few minutes or it would find no good decomposition at all.

The timing results for the test instances are all in the range of less than five minutes on a Pentium 4 machine with 512 MB of main memory running at 2 GHz. Detailed infos on the timing can be found in the corresponding chapters describing the test cases.

The quality of the solutions returned can be measured by assigning the result to one of two classes. Either a rather continuous profile is obtained or some big block destroys the continuity of the profile. Which of the two cases actually happens can be decided already by using (computationally) inexpensive methods.

Chapter 3

Numerical Experiments

This chapter summarizes some numerical results.

3.1 The Implementation

All experiments concerning hypertree decomposition were carried out in the following way: First a matrix was generated and saved in a sparse matrix file format, which is structured as follows.

```
<dimension of matrix>
<row index 1> <column index 1> <value of entry 1>
.
.
.
<row index m> <column index m> <value of entry m>
```

The first line contains the dimension of the matrix. Every consecutive line contains the row index and the column index of an element of the matrix as well as its value.

The sparse matrix file can be converted to a hypergraph, which can be used as an input for hypertree decomposition. This conversion consists of the following steps. Let the column indices of the matrix be the nodes of the hypergraph. For each row generate a hyperedge by simply including all column indices that correspond to non-zero entries in the row under consideration. The corresponding code `sparse2hg.cc` is given below.

```
#include <iostream>
#include <map>
#include <list>
#include <unistd.h>

using namespace std;

int main(int argc, char **argv)
{ int c, transpose = 0;
  while((c = getopt(argc, argv, "t")) != -1)
  { switch(c)
```

```
        { case 't': transpose = 1; break;
          default: break;
        }
      }
      long n,m,size = 0,k,l;
      double d;
      cin >> n;
      m = n;
      bool *test = new bool[n+1];
      for(long i = 0; i <= n; i++)test[i] = false;

      map<long, list<long> > matrix;
      for(; !cin.eof(); )
      { cin >> k >> l >> d;
        list<long> liste;
        if(!cin.eof())
        { if(transpose == 0)
          { matrix.insert(pair<long,list<long> >(l,liste));
            map<long, list<long> >::iterator it = matrix.find(l);
            test[l] = true;
            it->second.push_back(k);
          }
          else
          { matrix.insert(pair<long,list<long> >(k,liste));
            map<long, list<long> >::iterator it = matrix.find(k);
            test[k] = true;
            it->second.push_back(l);
          }
        }
      }

      cout << n << endl;
      map<long, list<long> >::iterator f2 = matrix.begin(), l2 = matrix.end();
      for(; f2 != l2; f2++)
      { list<long>::iterator f1 = f2->second.begin(), l1 = f2->second.end();
        cout << f2->first << " " << f2->second.size();
        for(; f1 != l1; f1++)cout << " " << (*f1);
        cout << endl;
      }
      for(long i = 1; i<n; i++)if(test[i] == false)cout << i <<" 1 "<<i<<endl;
      delete test;
    }
```

After being converted to a hypergraph the new method can be applied with different options to the matrix. If option $-b$ is used, the new method yields as an output two permutation vectors for the hypergraph, i. e., the input matrix, in the following simple file format:

```
//Hyperedgepermutation:
<old index 1> <new index 1>
```

```
.
.
.
<old index n> <new index n>
//Hypernodepermutation:
<old index 1> <new index 1>
.
.
.
<old index n> <new index n>
```

For each row index there is exactly one entry in the section "Hyperedgepermutation" of the file and for each column index there is exactly one entry in the section "Hypernodepermutation".

The permutation file together with the matrix file can be used to generate visual representations using MATLAB.

3.2 The Test Matrices

The experiments of this report were performed using three types of test matrices:

1. Random matrices from a special generator (see Section 3.2.1).
2. Random matrices from MATLAB (see Section 3.2.2).
3. Structured matrices motivated by quantum chemistry applications (see Section 3.2.3).

3.2.1 Random Matrix Generator

For creating the first type of test matrices the following random matrix generator was developed:

```
#include <iostream>
#include <stdlib.h>
#include <limits.h>
#include <set>

using namespace std;

long ran(int n)
{ double d = (double)random();
  d /= 2147483648;
  d *= n;
  return long(d);
}

int main()
{ srandom(time(0));
  long n, maxsize, trials;
  cin >> n >> maxsize >> trials;
  if(maxsize > n)maxsize = n;
  set<long> x[n],y[n];
  cout << n << endl;
  for(long i = 0; i < trials; i++)
  { long j,k;
    j = ran(n);
    k = ran(n);
    if(x[j].size() < maxsize && y[k].size() < maxsize)
    { x[j].insert(k);
      y[k].insert(j);
      x[k].insert(j);
      y[j].insert(k);
    }
  }
  for(long i = 0; i < n; i++)
  { set<long>::iterator f = x[i].begin(), l = x[i].end();
    for(; f != l; f++)cout << i+1 << " " << *f+1 << " 1" << endl;
  }
  return 0;
}
```

This generator produces symmetric matrices of arbitrary size, whose maximum number of non-zero entries in each row and in each column is bounded by a given constant.

One can pass the size of the matrix, the bound of the non-zero entries in each row/column and the number of trials. In each trial a new pair of row and column indices is randomly generated. If the index pair can contain a non-zero element without hurting the non-zero bound, then the corresponding element of the matrix is set to one, otherwise the next trial—if any—is carried out.

The first series of 5 matrices has a bound of 3 for the maximum of non-zeros in each row/column and 300 trials are done. The next series of 5 matrices also has 3 as a bound for the non-zeros of and makes $10,000$ trials. Compared to the first series, this series contains more entries and constitutes a symmetric matrix whose adjacency graph is (almost) 3-regular. This kind of matrices idecompose well and the resulting structure after reordering is an almost triangular matrix. Choosing the bound of non-zeroes in each row and column to be 5 and using $100,000$ trials increases the connectivity of the matrices considerably and the matrix after reordering does not look very promising.

3.2.2 MATLAB's Random Matrices

The second series of test matrices was generated using MATLAB's command *sprandsym*, which produces sparse symmetric random matrices. The matrices have a density of 0.03 and 0.05, which is the number of non-zeros in relation to the number of elements in the matrix. In contrast to the matrices in the first series, there is no restriction on the number of elements in each row and column. It turns out that the computation time required to compute solutions to problems based on these matrices is similar to the computation time for the first test series.

3.2.3 Structured Matrices from Applications

The last class of matrices was motivated by a single symmetric matrix from quantum chemistry. Since this matrix could not be decomposed sufficiently well because of its "density", its structural features were used to constructed a new matrix, which can be decomposed.

The original matrix is symmetric and can be divided into four blocks, where the upper left block and lower right block are square. The upper left block is approximately four times the size of the lower left block. The lower right block is essentially a random symmetric matrix. In the upper left block the dominant elements of the matrix are concentrated around the main diagonal. In the upper right block, the dominant entries are also concentrated around the main diagonal. The abstraction of this matrix was a diagonal matrix as its upper left block, a random matrix as its lower right block, and a "diagonal" matrix as its upper right block as can be seen in Fig. 3.2.3.

The original symmetric matrix has a size of 2176×2176, whereas the test matrix used is of size 1000×1000 and has a much lower density than the original matrix.

The following generator `scfstep.cc` performs this task. One can pass as parameters the size of the matrix and the size of the four blocks of the matrix as well as the density of the random block.

```
#include <iostream>
#include <stdlib.h>
#include <limits.h>
#include <set>

using namespace std;

long ran(int n)
{ double d = (double)random();
  d /= 2147483648;
  d *= n;
  return long(d);
```

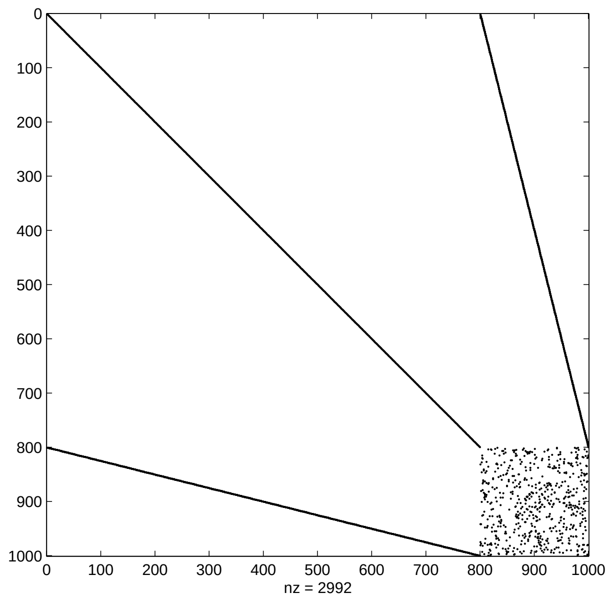

Figure 3.1: The abstraction of the matrix from quantum chemistry

```
}

int main()
{ srandom(time(0));
  long n, m, trials;
  double ratio;
  cin >> n >> m >> trials;
  if(m > n)m = n;
  set<long> x[n],y[n];
  cout << n << endl;
  for(long l = 0; l < m; l++){ x[l].insert(l); y[l].insert(l); }
  ratio = (n-m)/double(m);
  for(long r = 0; r < m; r++)
  { long index = m+long(ratio*r);
    x[r].insert(index);
    y[index].insert(r);
    x[index].insert(r);
    y[r].insert(index);
  }
  for(long i = 0; i < trials; i++)
  { long j,k;
    j = ran(n-m);
    k = ran(n-m);
    x[j+m].insert(k+m);
    y[k+m].insert(j+m);
    x[k+m].insert(j+m);
    y[j+m].insert(k+m);
    if(i > 0 && i <= m){ x[i].insert(i); y[i].insert(i); }
    }
    for(long i = 0; i < n; i++)
    { set<long>::iterator f = x[i].begin(), l = x[i].end();
      for(;f!=l;f++)cout << i+1 << " " << *f+1 << " 1" << endl;
    }
    return 0;
  }
```

3.3 The Reorderings Produced

The test instances obey the following naming system: $gss-100-3-300.x$ denotes the x-th instance in a series of instances generated using the random generator producing (almost) regular symmetric matrices of size 100×100. The matrices are almost 3-regular and 300 trials have been used. Similarly $gss-100-3-10000.x$ matrices have been generated with 10000 trials. $gss-100-5-100000.x$ matrices are almost 5 regular 100×100 matrices generated with 100000 trials. $sprs-100-y.x$ matrices have been generated using MATLAB's command *sprandsym*. Those matrices are symmetric 100×100 matrices having a density of y percent. Finally $scfstep-1000-800-300$ denotes the quantum chemistry matrix abstraction of size 1000×1000 with the upper left block of size 800×800. The lower right block will be filled with 300 trials.

The options used for calling the new reordering algorithm for different instances were the following: The option $-b$ just indicates to the heuristic to output permutation vectors rather than hypertree decompositions. The option $-u$ indicates to the heuristic to use the "single node split" split procedure, when searching for a separator. This method works by isolating single nodes in the hypergraph from the rest of the hypergraph. The second option used in the context of the test cases in this report is $-d$ which means: use dual vertex connectivity. In this method the dual graph of the hypergraph is computed. Then the vertex connectivity of the dual graph is determined. One can give a threshold for the size of a cut which is feasible with the option $-vnn$, where nn is the size of a minimum vertex cut with which we are satisfied. For a more detailed exposition on the subject consult [8, 9].

The running times listed in the last column in each table were measured on a dual Pentium 4 system running at 2 GHz, having 512 MB of RAM. The operating system on this machine is SuSE Linux 8.1.

3.3.1 Generated Random Matrices

This series of matrices originates from the C++ - random generator producing (almost) regular matrices.

Name of test case	Options	Runtime [s]
gss-100-3-300.1	budv4	109
gss-100-3-300.2	budv4	36
gss-100-3-300.3	budv4	32
gss-100-3-300.4	budv4	35
gss-100-3-300.5	budv4	32
gss-100-3-10000.1	budv4	173
gss-100-3-10000.2	budv5	28
gss-100-3-10000.3	budv5	49
gss-100-3-10000.4	budv5	39
gss-100-3-10000.5	budv5	55
gss-100-5-100000.1	budv18	112
gss-100-5-100000.2	budv18	90
gss-100-5-100000.3	budv20	162
gss-100-5-100000.4	budv20	119
gss-100-5-100000.5	budv20	79

Whenever the run times in the table are significantly higher than the standard deviation and average suggest, more time has been spent on computing solutions to linear programs arising in connection with the computation of the vertex connectivity of a hypergraph. Usually, higher running times indicate a higher connectivity of the hypergraph under consideration. One can see in Figs. 3.2-3.5 that the `gss-100-3-x.y` instances are well decomposable. The profile ist continuous and stays close to the diagonal. In Figs. 3.6-3.7 there are already big blocks that cannot be decomposed very well.

3.3.2 MATLAB's Sparse Random Matrices

This section contains results for test matrices generated using MATLAB's function *sprandsym*.

Name of test case	Options	Runtime [s]
sprs-100-3.1	budv10	81
sprs-100-3.2	budv10	71
sprs-100-3.3	budv10	78
sprs-100-3.4	budv10	71
sprs-100-3.5	budv10	81
sprs-100-5.1	budv10	379
sprs-100-5.2	budv10	231
sprs-100-5.3	budv10	220
sprs-100-5.4	budv10	167
sprs-100-5.5	budv10	171

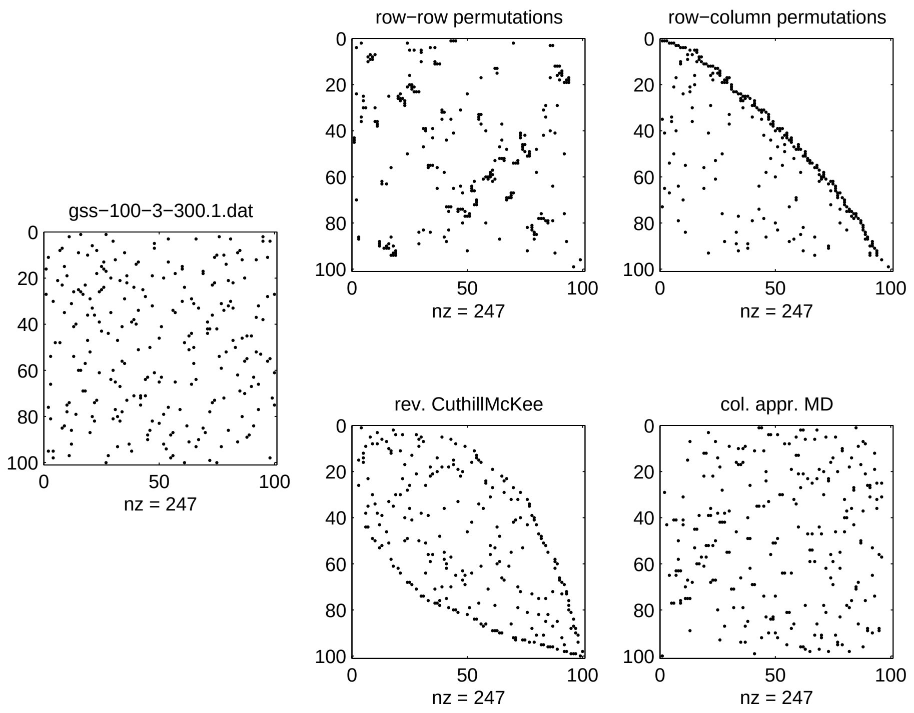

Figure 3.2: Reorderings for gss-100-3-300.1

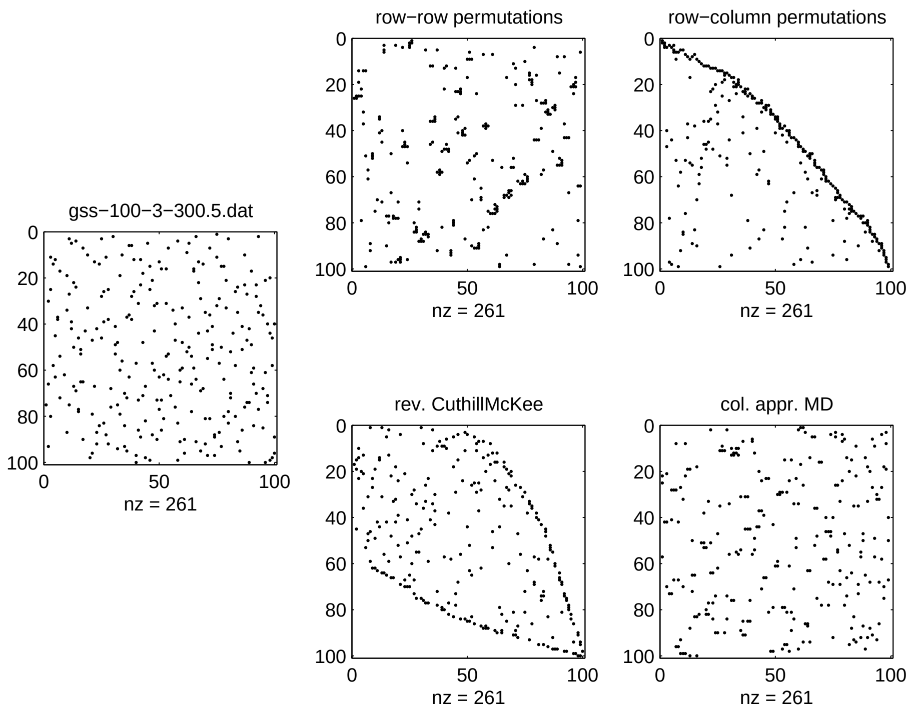

Figure 3.3: Reorderings for gss-100-3-300.5

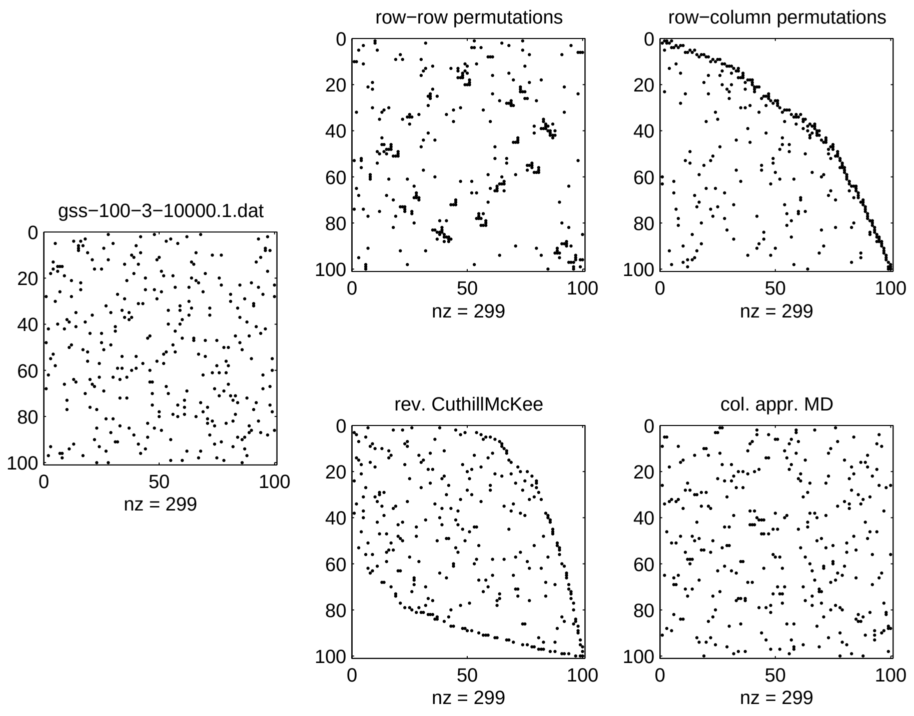

Figure 3.4: Reorderings for gss-100-3-10000.1

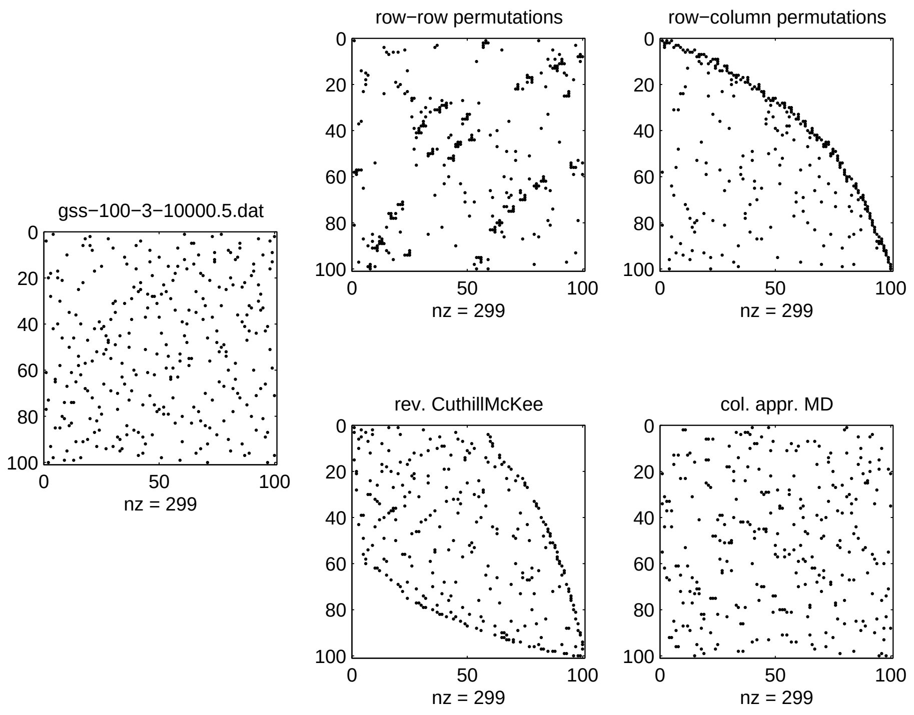

Figure 3.5: Reorderings for gss-100-3-10000.5

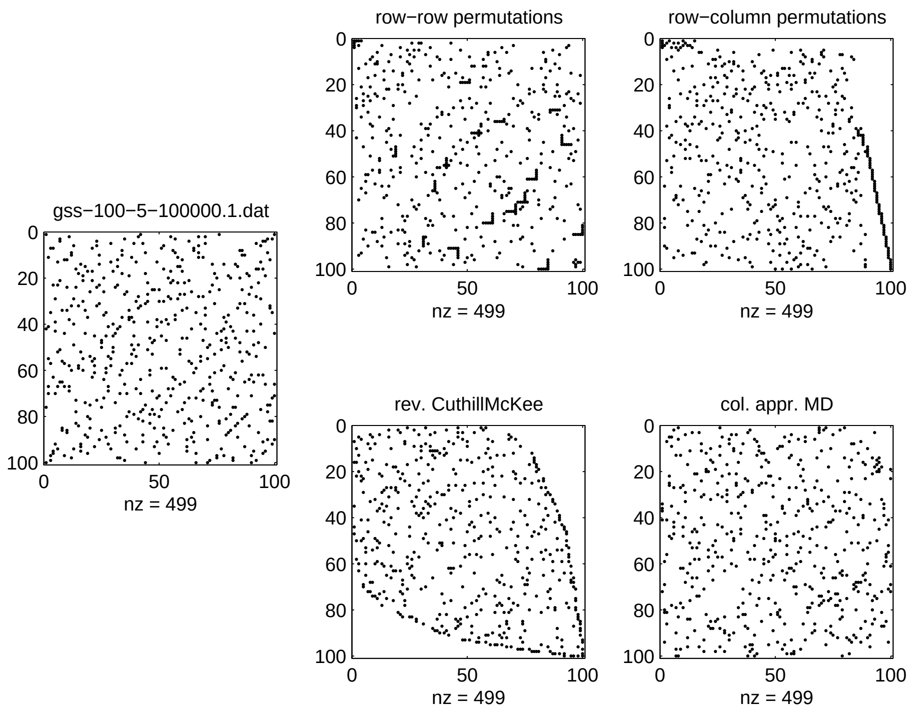

Figure 3.6: Reorderings for gss-100-5-100000.1

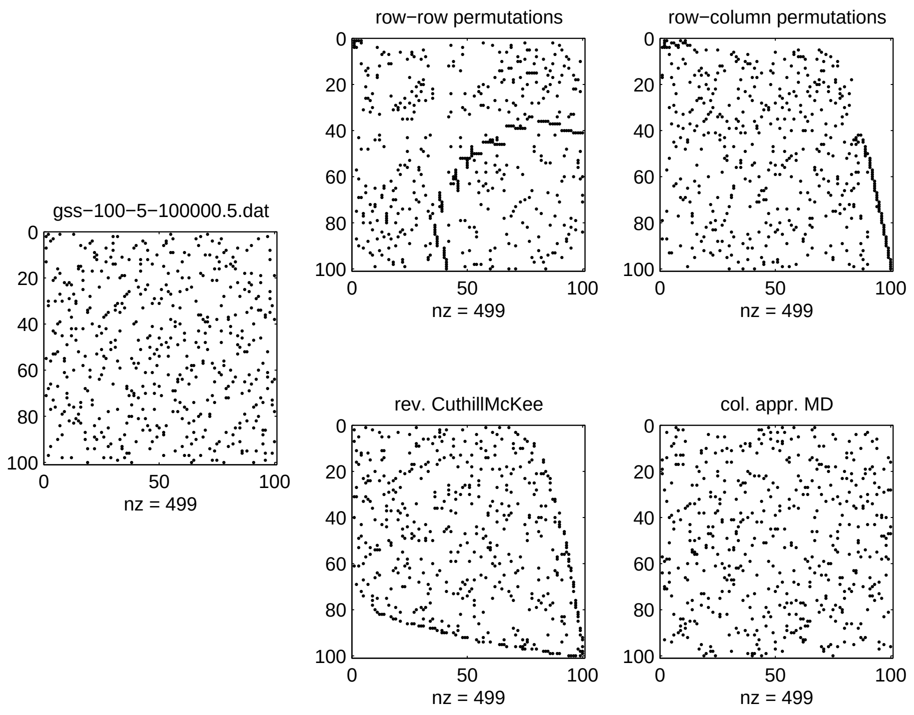

Figure 3.7: Reorderings for gss-100-5-100000.5

Whenever the running times in the table are significantly higher than the standard deviation and average suggest, more time has been spent on computing solutions to linear programs arising in connection with the computation of the vertex connectivity of a hypergraph. Usually, higher running times indicate a higher connectivity of the hypergraph under consideration. One can see in Figs. 3.8-3.11 the results of reordering the `sprs-100-x.y` matrices. The `sprs-100-3.y` matrices are still decomposable quite well and yield rather continuous profiles, although these instances seem to be harder to decompose than the `gss-100-3-x.y` instances. The `sprs-100-5.y` instances are not well decomposable any more. One can see in the corresponding figures that there are big blocks, that destroy the profile.

3.3.3 Matrices from Quantum Chemistry

This section summarizes test results for the matrix class derived from the matrix from quantum chemistry.

Name of test case	Options	Runtime [s]
scfstep-1000-800-300	bu	36

The run time is significantly lower than in the other test cases, because the $-d$ option was not used this time. Therefore no time was spent on computing the dual vertex connectivity of a hypergraph. Only the simplest decomposition method available was used (option $-u$).

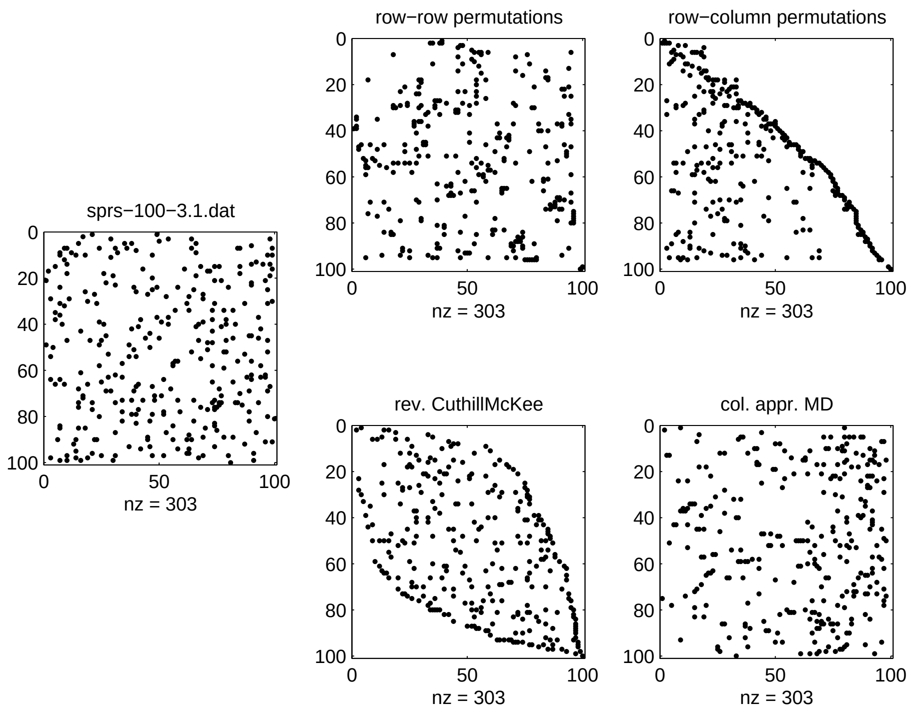

Figure 3.8: Reorderings for sprs-100-3.1

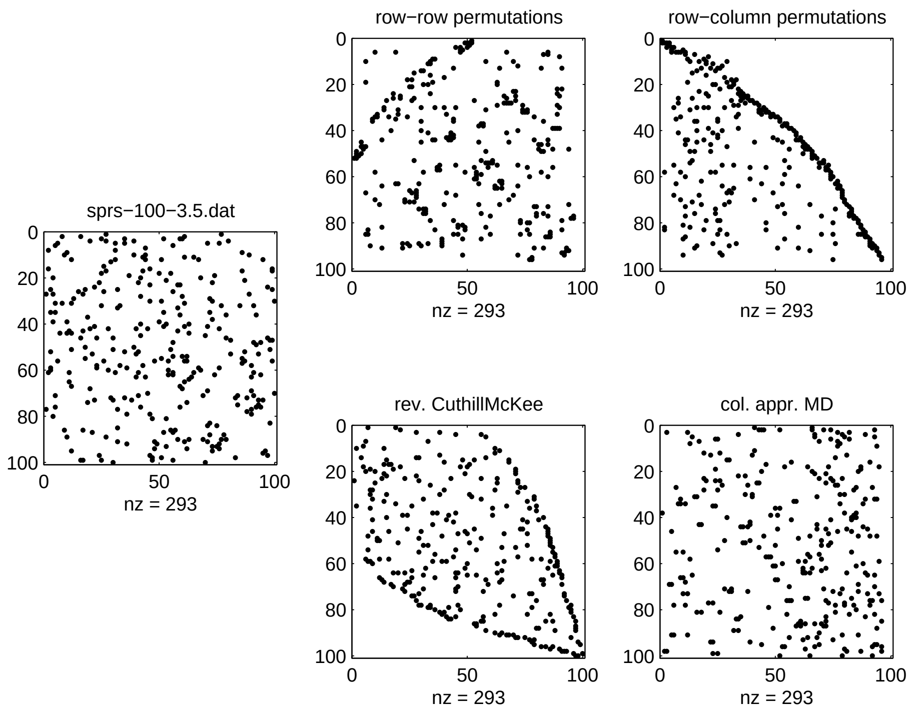

Figure 3.9: Reorderings for sprs-100-3.5

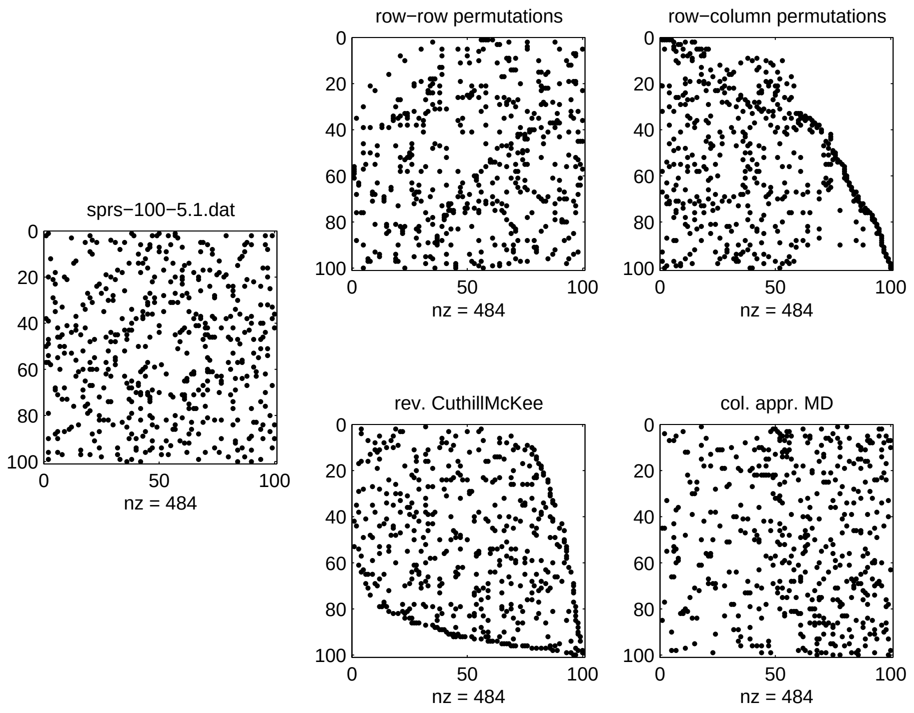

Figure 3.10: Reorderings for sprs-100-5.1

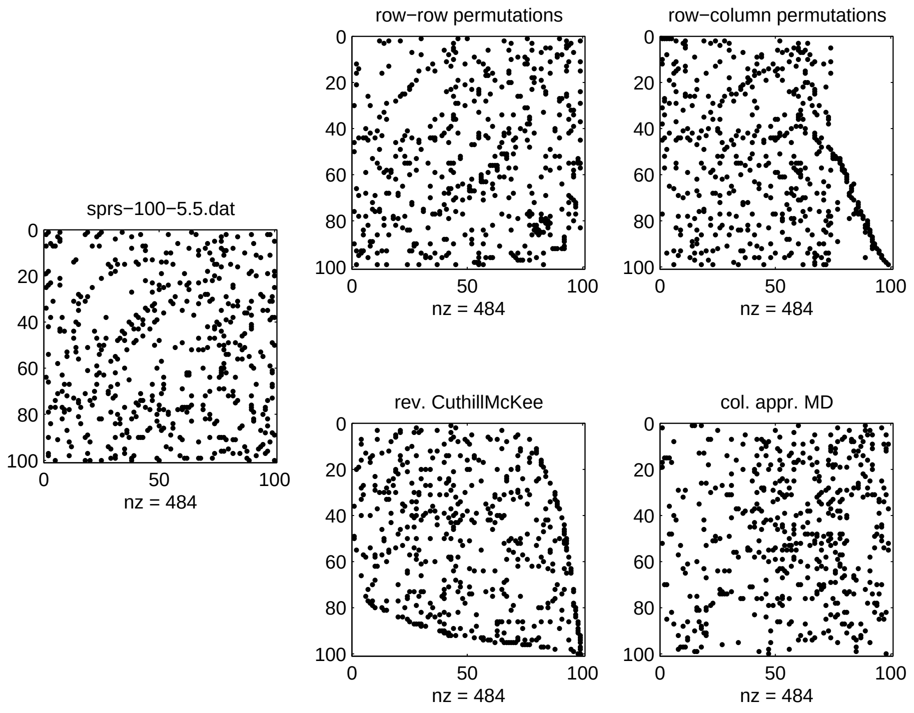

Figure 3.11: Reorderings for sprse-100-5.5

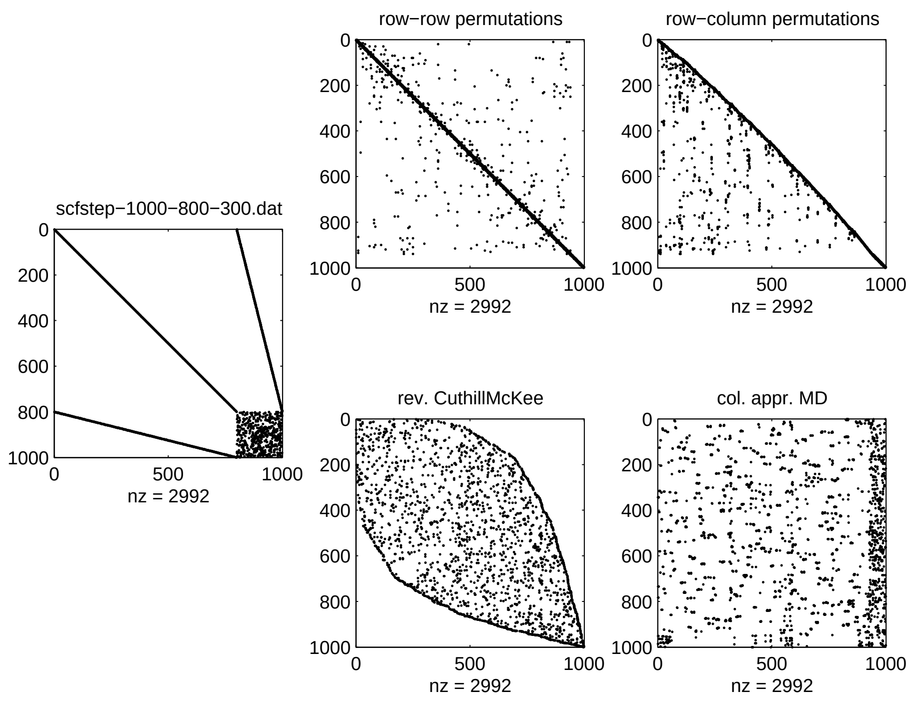

Figure 3.12: Reorderings for scfstep-1000-800-300

Chapter 4

Summary

This report compares the new hypertree reordering method with the reverse Cuthill McKee algorithm and the column approximate minimum degree ordering algorithm. Whereas column approximate degree ordering seems not to be able to cope with the test instances very well, reverse Cuthill McKee yields good results considering the density of the instances. Other methods were not taken into consideration since their primary purpose is not profile reduction or single-profile reduction.

If one compares the single sided profiles of reverse Cuthill McKee and of hypertree decomposition, it turns out that the profile of the hypertree decomposition reordering is much smaller than the profile of reverse Cuthill McKee. This might be due to the fact that the single-sided profile allows a matrix to have non-zeroes everywhere below the diagonal. Reverse Cuthill McKee makes symmetric permutations and therefore the profile fulfills some stronger requirement than being a good single-sided profile.

From the randomly generated matrices it can be conjectured that the decomposability of the matrices follows a threshold phenomenon, where the transition seems to be between 3 and 6 at least for 100×100 matrices. For larger matrices, this threshold might depend on the matrix size. Furthermore it is not clear whether the transition will still be tight enough (sigmoidal) to be considered a threshold transition.

Further research will be carried out to make the hypertree decomposition method available for very large matrices.

References

[1] P. R. Amestoy, T. A. Davis, I. S. Duff: *An Approximate Minimum Degree Ordering Algorithm*. SIAM J. Matrix Anal. Appl. 17 (1996), pp. 886–905.

[2] Y. Bai, R. M. Day, W. N. Gansterer, R. C. Ward: *New Algorithmic Tools for Electronic Structure Computations*. In *Proceedings of the Fourth IMACS Symposium on Mathematical Modelling (4th MATHMOD)*, Vienna University of Technology, Vienna, Austria, 2003.

[3] E. H. Cuthill, J. McKee: *Reducing the Bandwidth of Sparse Symmetric Matrices*. In *Proceedings of the 24th Nat. Conf. ACM*, 1969, pp. 157–172.

[4] T. A. Davis, J. R. Gilbert, S. I. Larimore, E. G. Ng: *A Column Approximate Minimum Degree Ordering Algorithm*. Technical Report TR-00-005, Department of Computer and Information Science and Engineering, University of Florida, 2000.

[5] A. George, J. W. H. Liu: *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[6] A. George, J. W. H. Liu: *The Evolution of the Minimum Degree Ordering Algorithm*. SIREV 31 (1989), pp. 1–19.

[7] G. Gottlob, N. Leone, F. Scarcello: *Hypertree Decompositions and Tractable Queries*. In *PODS'99*, Philadelphia, 1999.

[8] T. Korimort: *Heuristic Hypertree Decomposition*. Ph.D. dissertation, Vienna University of Technology, 2003.

[9] T. Korimort: *Heuristic Hypertree Decomposition*. Technical Report AURORA-TR2003-18, Vienna University of Technology, 2003.

[10] J. W. H. Liu, A. H. Sherman: *Comparative Analysis of the Cuthill-McKee and the Reverse Cuthill-McKee Ordering Algorithms for Sparse Matrices*. SINUM 13 (1976), pp. 198–213.

[11] Y. Saad: *Iterative Methods for Sparse Linear Systems*. PWS Publishing Co., Boston, 1996.