浙江大学

本科生毕业论文（设计）

文献综述和开题报告

姓名与学号 3110104942 唐周若愚

指导教师王何宇

年级与专业 2011 信息与计算科学

所在学院数学系

一、题目：稀疏矩阵存储的 C++实现

二、指导教师对文献综述和开题报告的具体内容要求：

按照规定的日程进度，全面阅读国内外相关文献，并挑选与本项目密切相关且有指导意义的文献一至三篇重点精读，节选重要部分作文献翻译。在文献研究的基础上，对项目的可行性进行分析论证，对项目的预期目标做合理的调整，形成完整的项目研究计划，并对各主要步骤的具体研究内容、理论基础、技术路线已经重点和难点做较全面的论证，形成书面报告。

指导教师（签名） 王何宇

2015 年 **3** 月 **17** 日

目录

2 开题报告

2.1 问题提出的背景

2.1.1 背景介绍

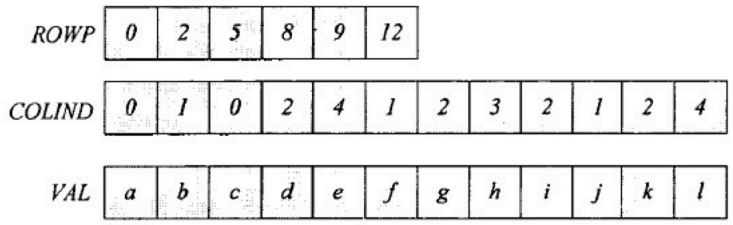
近年来，在工程应用中，求解高阶矩阵的需求日益增长，全矩阵运算脱离了实际的硬件限制，为了满足这一日益增长的需求，同时这些矩阵通常都有着一个特征——非零元远少于零元，稀疏矩阵这门学科便应运而生。在 20 世纪 60 年代研发电子网络的电子工程师们是最早的去利用稀疏性来应用稀疏矩阵进行工程上的计算的。[1] 而在微分方程数值解、线性规划等的有限元分析中，经常出现求解高阶稀疏线性方程组，如利用全矩阵进行存储，则需要 n^2 的空间复杂度和 n^3 的乘法运算时间复杂度，显然，这种程度的运算量是无法被微型计算机，甚至是工作站所接受的。而利用矩阵的稀疏性，可以有效地减小消耗很多无谓的存储空间以及无谓的计算，在很大的程度上降低了时间和空间复杂度，降低了计算对硬件的需求，使计算成为可能。

2.1.2 本研究的意义和目的

在实际工程计算中，尤其是在微分方程数值解、线性规划等的有限元分析中，经常出现高阶的矩阵运算，经常出现百万阶、千万阶的矩阵运算，而如果使用全矩阵运算，假设是百万阶的 float 类型的矩阵，则需要 $4 * 10^{12} B$ 来存储，也就是 4TB 的空间，这显然是无法被微型计算机甚至是工作站所接受的。但是通常这些矩阵具有稀疏的性质，拥有着大量的零元，而且通常阶数越高稀疏度越高。这就可以在相当大的程度上减少了存储对于硬件的压力。同时，在计算中，大量对于 0 的运算是五位的消耗资源的操作，也可以利用稀疏矩阵来避免这些无谓的操作。而在本研究中编写的稀疏矩阵库可以用来方便快捷的实现稀疏矩阵的存储以及一些基本运算，避免了使用者大量编写存储底层的代码，提高开发效率。

2.1.3 研究现状

在稀疏矩阵这些年的发展中，出现了很多的存储方法，比如：对角线存贮法、对称矩阵的变带宽存贮法、坐标存贮法、Elipack-Itpack 存贮法、CSR 存贮法、Shermans 存贮法、超矩阵存贮法、动态存贮方案等。
如行压缩存储方式 (Compressed Row Storage): CRS 存储可以高效地存取任意一行非零元素，但存取任意一列非零元则需要遍历整个 CRS 存储结构。相应地，与 CRS 存储的稀疏矩阵相关的算法要高效的编程实现，算法的计算顺序必须按行来进行。[1] 下面对应于 CRS 存储：



我们可以发现，ROWP 数组存储的时行非零元的增长量。COLIND 则存储的是列索引值，典型的 C 语言结构实现为

```
struct cr_matrix{
        int *Ri;/*col index*/
        int *Rp;/*length nrow+1*/
        double *Rx;
        int Rncol;
        int Rnrow;
};
```

在早期计算机时，串行的算法占到主流，自然，稀疏矩阵的存储方式以及稀疏矩阵的计算都是为了串行算法服务的。但是随着计算机的发展，计算机集群、多核 CPU、GPU 并行等的出现，让并行算法在计算时间上远远地超越了串行算法，为了更好的适应并行计算中的稀疏矩阵计算，出现了许多新的算法以及存储方式。例如对于非结构化的矩阵，基于 CUDA 框架下的 SCOO 形式的 SpMV 算法比利用基于 Cusp 库的 SCOO 具有更高的效率。[2]

而在做稀疏矩阵的计算时，通常都是做一系列的基本运算，如：矩阵转置、矩阵向量乘法、矩阵矩阵乘法、数乘等。为了能得到更好的效率，许多研究者致力于寻找对于这些计算最优的存储结构及计算算法，同时提供了许多类库供科学计算使用，如：Portable, ExtensibleToolkit for Scientific Computation (PETSc)、Boost、GNU Scientific Library (GSL) 等。
例如，在 Boost-uBLAS 中有着稀疏矩阵的模板 mapped_matrix<T, F, A>（元素映射矩阵存储形式）、compressed_matrix<T, F, IB, IA, TA>（压缩存储格式）、coordinat_matrix<T, F, IB, IA, TA>（坐标存储格式）。分别有着如下示例：[3]
mapped_matrix:

```
#include <boost/numeric/ublas/matrix_sparse.hpp>
#include <boost/numeric/ublas/io.hpp>

int main () {
    using namespace boost::numeric::ublas;
    mapped_matrix<double> m (3, 3, 3 * 3);
    for (unsigned i = 0; i < m.size1 (); ++ i)
        for (unsigned j = 0; j < m.size2 (); ++ j)
            m (i, j) = 3 * i + j;
    std::cout << m << std::endl;
}
```

compressed_matrix:

```
#include <boost/numeric/ublas/matrix_sparse.hpp>
#include <boost/numeric/ublas/io.hpp>

int main () {
    using namespace boost::numeric::ublas;
    compressed_matrix<double> m (3, 3, 3 * 3);
    for (unsigned i = 0; i < m.size1 (); ++ i)
        for (unsigned j = 0; j < m.size2 (); ++ j)
            m (i, j) = 3 * i + j;
    std::cout << m << std::endl;
}
```

coordinate_matrix:

```
#include <boost/numeric/ublas/matrix_sparse.hpp>
#include <boost/numeric/ublas/io.hpp>

int main () {
    using namespace boost::numeric::ublas;
    coordinate_matrix<double> m (3, 3, 3 * 3);
    for (unsigned i = 0; i < m.size1 (); ++ i)
        for (unsigned j = 0; j < m.size2 (); ++ j)
            m (i, j) = 3 * i + j;
    std::cout << m << std::endl;
}
```

在多年的研究中，各种相关的研究成果页层出不穷，例如，根据 Michele Martone 的研究成果，在对称矩阵乘法，或者转置的矩阵向量乘法中，RSB 格式的迭代算法效率是比较高的 [4]。而在李佳佳, 张秀霞, 谭光明, 陈明宇的研究中，得到了影响矩阵性能的参数集，可以利用该参数集提取矩阵特征，并输出最优存储格式，供数值解法器和上层应用调用。[5] 为了适用于大数据集计算以及克服现有稀疏矩阵乘法算法低效的问题，郑建华，朱蓉，沈玉利提出了一种基于向量线性组合（VLC）的矩阵乘法处理模式，同时采用 MapReduce 计算模型实现了基于 VLC 模式的并行矩阵乘法算法。[6] 而在集群计算方面，负载平衡是不能不提到的，为了更好地发挥集群计算的计算能力，付朝江博士给出了基于贪婪分配的稀疏矩阵与向量乘的负载平衡的解决方案。[7]

2.2 论文的主要内容和技术路线

2.2.1 主要研究内容

在 linux 平台下，基于 g++ 编译器，编写稀疏矩阵存储的 C++ 库，实现稀疏矩阵的行压缩存储方式 (Compressed Row Storage)，同时实现一些稀疏矩阵的

基本运算，如矩阵乘法、数乘等。为了更好地发挥 CPU 性能，需要实现多线程并行计算。

2.2.2 技术路线

实现稀疏矩阵的行压缩存储方式 (Compressed Row Storage)，通过数组存储非零元的增长量、列索引值和非零元值来实现稀疏矩阵的存储。为了实现可变长数组的存储，利用 C++ 的 STL 中的 vector 来存储数据。
而为了更好地利用 CPU 的性能，利用 thread 库实现多线程并行计算，更为有效地利用 CPU 的并行性能。而为了防止多线程并行计算中"脏数据"的出现，利用 mutex 互斥锁保障进程安全性。
为了实现稀疏矩阵的矩阵矩阵乘法和矩阵向量乘法，可以参考全矩阵矩阵矩阵乘法和矩阵向量乘法算法，利用稀疏矩阵的零元不参与计算的特性，按照稀疏矩阵存储的结构，较全矩阵计算省略大量计算，实现稀疏矩阵计算的优势——更为高效的计算。

2.2.3 可行性分析

1 linux 平台较 Windows 更为稳定，不容易出现宕机，且被工程界广为使用

2 g++ 是 GNU 的 C++ 编译器，历史悠久，经历过多年考验，有着成熟的文档和社区帮助

3 稀疏矩阵这门学科已存在多年，存在着成熟的存储方案及相关算法

4 多线程的锁机制已出现多年，可以有效地保障进程安全性

5 基于标准 C++ 开发，有着良好的跨平台性

2.3 研究计划进度安排及预期目标

2.3.1 进度安排

2014 年 11 月 24 日 -2014 年 12 月 21 日，文献收集整理，基础知识学习。
2014 年 12 月 22 日 -2015 年 1 月 4 日，讨论算法模型，确定方案和技术路线。
2015 年 1 月 5 日 -2015 年 1 月 29 日，完成文献综述、开题报告和文献资料翻译。
2015 年 3 月 9 日 -2015 年 4 月 5 日，程序编写、调试。
2015 年 4 月 6 日 -2015 年 5 月 3 日，数值实验和数据收集。
2015 年 5 月 4 日 -2015 年 5 月 31 日，毕业论文撰写。
2015 年 6 月 1 日 -2015 年 6 月 14 日，ppt 制作，准备答辩。

2.3.2 2、预期目标

利用 c++ 在 g++ 编译器 linux 平台下实现稀疏矩阵的行压缩存储方式 (Compressed Row Storage) 存储。可以并行计算稀疏矩阵的矩阵乘法、数乘等运算。

2.4 参考文献

[1] 张永杰、孙秦. 稀疏矩阵存储技术 [J]. 长春理工大学学报,2006 年,03 期:38-41.
[2]Hoang-Vu Dang, Bertil Schmidt.CUDA-enabled Sparse Matrix–Vector Multiplication on GPUs using atomic operations[J].Parallel Computing, 2013,Vol.39 (11):737-750.
[3]Joerg Walter and Mathias Koch.Boost 1.57.0 Library Documentation-uBLAS. http://www.boost.org/doc/libs/1_57_0/libs/numeric/ublas/doc/index.html,2011
[4]Michele Martone.Efficient multithreaded untransposed, transposed or symmetric sparse matrix–vector multiplication with the Recursive Sparse Blocks format[J].Parallel Computing, 2014,40:47-58.
[5] 李佳佳, 张秀霞, 谭光明, 陈明宇. 选择稀疏矩阵乘法最优存储格式的研究 [J]. 计算机研究与发展, Journal of Computer Research and Developmen,2014 年,04 期:882-894.
[6] 郑建华, 朱蓉, 沈玉利.Sparse matrix multiplication algorithm based on Map R educe[J]. 仲恺农业工程学院学报,2013 年,03 期:45-50.
[7] 付朝江. 基于贪婪分配的稀疏矩阵与向量乘的负载平衡 [J]. 福建工程学院学报,2010 年,01 期:79-82.

3 文献综述

3.1 背景介绍

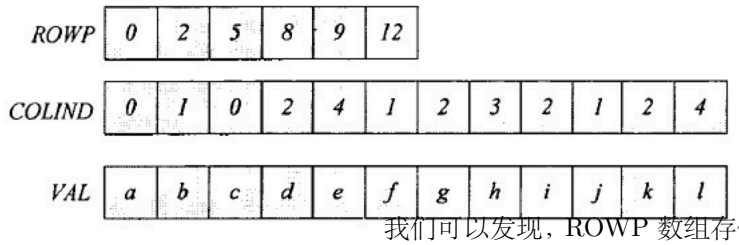
近年来，在工程应用中，求解高阶矩阵的需求日益增长，全矩阵运算脱离了实际的硬件限制，为了满足这一日益增长的需求，同时这些矩阵通常都有着一个特征——非零元远少于零元，稀疏矩阵这门学科便应运而生。在 20 世纪 60 年代研发电子网络的电子工程师们是最早的去利用稀疏性来应用稀疏矩阵进行工程上的计算的。[1] 而在微分方程数值解、线性规划等的有限元分析中，经常出现求解高阶稀疏线性方程组，如利用全矩阵进行存储，则需要 n^2 的空间复杂度和 n^3 的乘法运算时间复杂度，显然，这种程度的运算量是无法被微型计算机，甚至是工作站所接受的。而利用矩阵的稀疏性，可以有效地减小消耗很多无谓的存储空间以及无谓的计算，在很大的程度上降低了时间和空间复杂度，降低了计算对硬件的需求，使计算成为可能。

3.2 国内外研究现状

3.2.1 研究方向及进展

在这些年的发展中，出现了很多的存储方法，比如：对角线存贮法、对称矩阵的变带宽存贮法、坐标存贮法、Elipack-Itpack 存贮法、CSR 存贮法、Shermans 存贮法、超矩阵存贮法、动态存贮方案等 [2]。

如行压缩存储方式 (Compressed Row Storage): CRS 存储可以高效地存取任意一行非零元素，但存取任意一列非零元则需要遍历整个 CRS 存储结构。相应地，与 CRS 存储的稀疏矩阵相关的算法要高效的编程实现，算法的计算顺序必须按行来进行。[3] 下面对应于 CRS 存储：



我们可以发现，ROWP 数组存储的时行非零元的增长量。COLIND 则存储的是列索引值，典型的 C 语言结构实现为

```
struct cr_matrix{
        int *Ri;/*col index*/
        int *Rp;/*length nrow+1*/
        double *Rx;
        int Rncol;
        int Rnrow;
```

```
};
```

在早期计算机时，串行的算法占到主流，自然，稀疏矩阵的存储方式以及稀疏矩阵的计算都是为了串行算法服务的。但是随着计算机的发展，计算机集群、多核 CPU、GPU 并行等的出现，让并行算法在计算时间上远远地超越了串行算法，为了更好的适应并行计算中的稀疏矩阵计算，出现了许多新的算法以及存储方式。例如对于非结构化的矩阵，基于 CUDA 框架下的 SCOO 形式的 SpMV 算法比利用基于 Cusp 库的 SCOO 具有更高的效率。[4]
而在做稀疏矩阵的计算时，通常都是做一系列的基本运算，如：矩阵转置、矩阵向量乘法、矩阵矩阵乘法、数乘等。为了能得到更好的效率，许多研究者致力于寻找对于这些计算最优的存储结构及计算算法，同时提供了许多类库供科学计算使用，如：Portable, ExtensibleToolkit for Scientific Computation（PETSc）、Boost、GNU Scientific Library (GSL) 等。
例如，在 Boost-uBLAS 中有着稀疏矩阵的模板 mapped_matrix<T, F, A>（元素映射矩阵存储形式）、compressed_matrix<T, F, IB, IA, TA>（压缩存储格式）、coordinat_matrix<T, F, IB, IA, TA>（坐标存储格式）。分别有着如下示例：[5]
mapped_matrix:

```
#include <boost/numeric/ublas/matrix_sparse.hpp>
#include <boost/numeric/ublas/io.hpp>

int main () {
    using namespace boost::numeric::ublas;
    mapped_matrix<double> m (3, 3, 3 * 3);
    for (unsigned i = 0; i < m.size1 (); ++ i)
        for (unsigned j = 0; j < m.size2 (); ++ j)
            m (i, j) = 3 * i + j;
    std::cout << m << std::endl;
}
```

compressed_matrix:

```
#include <boost/numeric/ublas/matrix_sparse.hpp>
#include <boost/numeric/ublas/io.hpp>

int main () {
    using namespace boost::numeric::ublas;
    compressed_matrix<double> m (3, 3, 3 * 3);
    for (unsigned i = 0; i < m.size1 (); ++ i)
        for (unsigned j = 0; j < m.size2 (); ++ j)
            m (i, j) = 3 * i + j;
    std::cout << m << std::endl;
}
```

coordinate_matrix:

```
#include <boost/numeric/ublas/matrix_sparse.hpp>
#include <boost/numeric/ublas/io.hpp>

int main () {
    using namespace boost::numeric::ublas;
    coordinate_matrix<double> m (3, 3, 3 * 3);
    for (unsigned i = 0; i < m.size1 (); ++ i)
        for (unsigned j = 0; j < m.size2 (); ++ j)
            m (i, j) = 3 * i + j;
    std::cout << m << std::endl;
}
```

在多年的研究中，各种相关的研究成果页层出不穷，例如，根据 Michele Martone 的研究成果，在对称矩阵乘法，或者转置的矩阵向量乘法中，RSB 格式的迭代算法效率是比较高的 [6]。而在李佳佳, 张秀霞, 谭光明, 陈明宇的研究中，得到了影响矩阵性能的参数集，可以利用该参数集提取矩阵特征，并输出最优存储格式，供数值解法器和上层应用调用。[7] 为了适用于大数据集计算以及克服现有稀疏矩阵乘法算法低效的问题，郑建华，朱蓉，沈玉利提出了一种基于向量线性组合（VLC）的矩阵乘法处理模式，同时采用 MapReduce 计算模型实现了基于 VLC 模式的并行矩阵乘法算法。[8] 而在集群计算方面，负载平衡是不能不提到的，为了更好地发挥集群计算的计算能力，付朝江博士给出了基于贪婪分配的稀疏矩阵与向量乘的负载平衡的解决方案。[9]

3.2.2 存在问题

稀疏矩阵的存储没有一种通用的形式，各种形式都各有利弊 [10], 这也就需要我们对于不同的情况去寻找最合适的存储形式来计算，虽然李佳佳, 张秀霞, 谭光明, 陈明宇 [7] 的研究得到了一个参数集用以寻找最优存储形式，但是比较的存储形式比较少，还有多种形式没有计入考量范围。

而在超大规模的稀疏矩阵计算时，计算机的内存可能不能完全存储这些数据，部分数据将会存储在硬盘中，而硬盘的读取效率远低于内存，如何更好地将硬盘中的数据读取到内存中，减少内存与硬盘交互的时间，同时，如何降低内存与缓存间的交互时间也是一个问题。

而随着计算机的普及，计算机进入了家家户户，分布式计算逐渐成为可能，让普通的计算机使用者参与到科学计算中，贡献 GPU 资源来帮助研究人员更快地完成计算，但是这尚未得到普及，同时，中国尚未形成一个成熟的分布式计算平台，也没有培养出普通用户参与到分布式计算项目的习惯。

3.2.3 研究展望

为了更好的适应现代的工程计算需求，计算更大规模的稀疏矩阵，以及适应新的 CPU 指令集和 GPU 计算框架，稀疏矩阵的存储形式值得进一步的研究。同时，伴随着分布式计算的发展，更为高效的冗余计算机制和任务分配机制也将会带给稀疏矩阵计算新的研究方向。按照摩尔定律，计算机的计算能力将每隔

18-24 个月翻一番，那么计算稀疏矩阵的能力也将相应提高。同时随着量子计算机、光计算机等的出现，更高德计算能力也将随之到来。而为了更好地发挥这些硬件的能力，以往的算法可能并不再适合了，需要开发新的算法以适应新的硬件。而对于硬件商而言，如 Intel、AMD 等 CPU 厂商而言，封装更多的 CPU 指令，让用户可以直接调用 CPU 指令来进行科学计算，更好地发挥硬件的性能。对于普通计算机用户，培养参与到分布式计算的习惯，通过建立一个国家级的分布式计算中心，建立奖励机制，鼓励普通计算机用户参与到分布式计算中，为研究计划中的稀疏矩阵计算贡献自己的计算资源。

3.3 参考文献

[1]Yousef Saad.Iterative Methods for Sparse Linear Systems[M].SECOND EDITION.USA:Society for Industrial and Applied Mathematics,2003 年.68.

[2] 张永杰、孙秦. 稀疏矩阵存储技术 [J]. 长春理工大学学报,2006 年,03 期:38-41.

[3] 冯广祥. 大型稀疏矩阵直接求解算法的研究及实现 [8]. 东北大学：系统工程，2010.

[4]Hoang-Vu Dang, Bertil Schmidt.CUDA-enabled Sparse Matrix–Vector Multiplication on GPUs using atomic operations[J].Parallel Computing, 2013,Vol.39 (11):737-750.

[5]Joerg Walter and Mathias Koch.Boost 1.57.0 Library Documentation-uBLAS. http://www.boost.org/doc/libs/1_57_0/libs/numeric/ublas/doc/index.html,2011

[6]Michele Martone.Efficient multithreaded untransposed, transposed or symmetric sparse matrix–vector multiplication with the Recursive Sparse Blocks format[J].Parallel Computing, 2014,40:47-58.

[7] 李佳佳, 张秀霞, 谭光明, 陈明宇. 选择稀疏矩阵乘法最优存储格式的研究 [J]. 计算机研究与发展, Journal of Computer Research and Developmen,2014 年,04 期:882-894.

[8] 郑建华, 朱蓉, 沈玉利.Sparse matrix multiplication algorithm based on Map R educe[J]. 仲恺农业工程学院学报,2013 年,03 期:45-50.

[9] 付朝江. 基于贪婪分配的稀疏矩阵与向量乘的负载平衡 [J]. 福建工程学院学报,2010 年,01 期:79-82.

[10] 张永杰,孙秦. 稀疏矩阵存储技术 [J]. 长春理工大学学报,2006 年,03 期:38-41.

4 文献翻译

4.1 译文

就像在上一节描述的一样，标准的离散化的偏微分方程往往会伴随着一个庞大的且稀疏的矩阵。稀疏矩阵可以被模糊的描述为一个具有非常少的非零元的矩阵。但是，事实上，当特殊的技巧需要利用到大量的非零元以及它们的位置时，一个矩阵是可以被稀疏化的。这些稀疏化矩阵的技巧是从不储存零元的想法开始的。一个关键的问题是制定能够适合于高效地使用不论是直接还是迭代的标准计算方法的存储稀疏矩阵的数据结构。这一章节将简介稀疏矩阵，它们的属性、呈现，以及用以存储它们的数据结构。

3.1 介绍

利用一个矩阵中的零元以及它们的位置的自然的想法最初是由在不同学科的工程师们提出的。在涉及带状矩阵的简单地例子中，特殊的技巧直接的被发明了。在 20 世纪 60 年代研发电子网络的电子工程师们是最早的去利用稀疏性来对于具有特殊结构的矩阵解决一般稀疏线性系统。对于稀疏矩阵技巧而言，最主要也是最早需要解决的问题是去设计一个在线性系统中得直接求解算法。这些算法需要是可以接受的，在存储和计算效率上。直接的稀疏算法可以被用于计算那些庞大的难以被稠密算法来实现的问题。

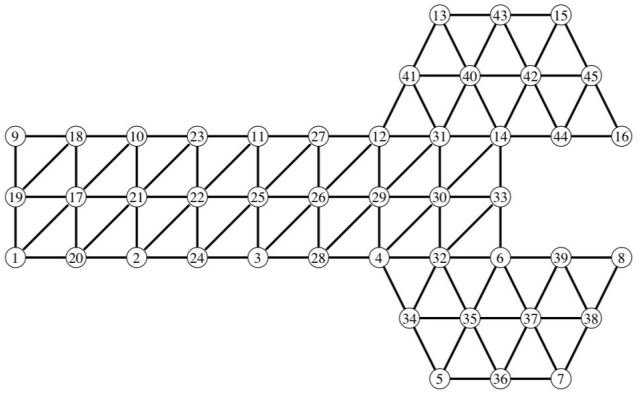


Figure 3.1 *A finite element grid model.*

基本上，有两个明显的类别的稀疏矩阵，结构化的和非结构化的。一个结构化的矩阵是指一个非零元的位置形成某个规律的矩阵，通常这些非零元在对角线附近。要不然，这些非零元会在相同大小的块内（稠密子矩阵），而这也会形成一个规律，通常这些非零元在对角线（块）附近。一个具有着不规则位置的非零元的矩阵会被称作是非结构化的。最好的一个结构化的矩阵的例子是一个只

有着少量对角元的矩阵。网格上的有限差分矩阵，就像上一节中提到的，是典型的具有着规律结构的例子。大部分的对于复杂几何的有限元和有限体积技巧会导致非结构化的矩阵。图 3.2 展示了一个与图 3.1 所呈现的有限元网格问题的一个小规模的非结构化的矩阵。

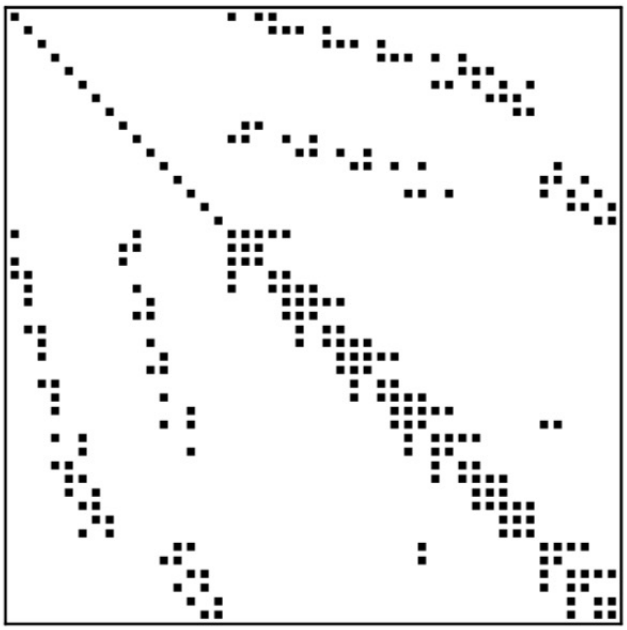


Figure 3.2 *Sparse matrix associated with the finite element grid of Figure 3.1.*

3.2 图论

图论是用来表示稀疏矩阵结构的一个理想的工具，因此，在稀疏矩阵技巧中，它扮演着一个主要的角色。例如，图论是用于解决并行稀疏高斯消除和预处理技术的关键。在下一节中，将讨论图的一般特性，以及它们在有限元和有限差分矩阵中得应用。

3.2.1 图与邻接图

记住一个图由两个集合定义，一个顶点集合 $V = \{v_1, v_2, \cdots, v_n\}$ 和一个边的集合 E, E 是由点对 (v_i, v_j) 组成的, v_i, v_j 都是 V 中的元素, 换而言之, $E \subseteq V \times V$。这个图 $G = (V, E)$ 通常被平面内的一系列的被边联系的点的向量来表示。这个

图被用来描述集合 V 中元素间的关系。例如，V 可以被用来描述世界上的主要城市。线就是两个城市间的直达航线。那么这个图就会描述这样一个关系“在城市 A 和城市 B 间存在一条直达航线”。在这个特殊的例子中，二元关系很可能是对称的，换而言之，如果有一条 A 到 B 的直达航线，那么也有一条 B 到 A 的直达航线。在这样的情形中，图被称作是无向的，用以与通常的有向图相对。

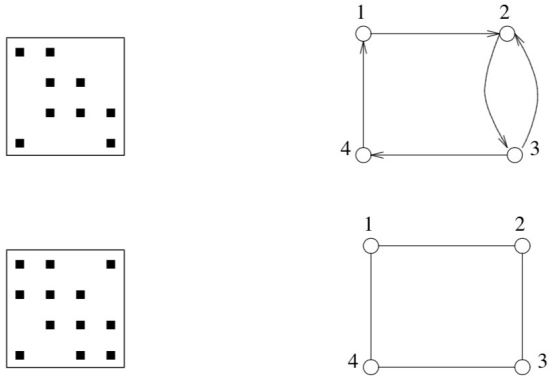


Figure 3.3 *Graphs of two* 4×4 *sparse matrices.*

回到稀疏矩阵，稀疏矩阵的邻接图是一个图 $G = (V, E)$，V 中得 n 个顶点代表 n 个未知数。它的边是按照以下规则建立的方程式建立的二元关系：当 $a_{ji} \neq 0$ 时，有一条从节点 i 指向节点 j 的边。而这条边将因此描述包含未知量 j 的二元关系方程式 i。注意，这个图是有向的，除非这个矩阵是有对称结构的（对任意的 $1 \leq i, j \leq n$，若 $a_{ji} \neq 0$，那么 $a_{ij} \neq 0$ ）。

当一个矩阵的非零元总有一个对称非零元，换而言之 a_{ij} 和 a_{ji} 总是同时为非零元，那么这图就是无向的。因此，对于无向图，每条边都有两个方向。因此，无向图可以用无向边来表现。

作为利用图模型的例子，并行高斯消去法可以通过寻找在指定消去阶段的未知数来获得。根据以上的二元关系，这些未知数两两独立。这些与未知数一致的行可以被用作基。因此，在一个极端情况下，当一个矩阵是对角阵，那么所有的未知数是独立的。与之相反的是，当一个矩阵是稠密的，那么每一个未知量都与其他未知量相关。稀疏矩阵则介于这两种极端情况之间。

邻接图有一些有趣的简单性质。A^2 的图可以被解释成一个 n 顶点图，对每条边的点对 (i,j)，表示在原图 A 中至少存在一条长度确切的说是 2 的从节点 i 到节点 j 的路径。与之相似的时，A^k 的图包含的时用以描述从节点 i 到节点 j 的至少存在一条长度为 k 的路径的二元关系的边。欲知详情，请看练习 4.

3.2.2PDE 矩阵的图

对于在每个网格点只涉及一个屋里未知量的偏微分方程，离散矩阵的邻接图通常就是用来描述网格的图。但是，在每个网格点上有着多个未知量是很常见的。例如，模拟流体流动的方程可能涉及流体的两个速度分量（二维）以及在每个

网格点的能量和动量。在这样的情况下，有两种用来标记未知量的选择。在每个网格点，它们可以被连续的标记。因此，在刚才的例子中，我们可以在一个指定的网格点例如 u(k),$\cdots$,u(k+3) 上标记所有的四个未知量（两个速度的分量，动量以及压力）。另外，所有的与一类变量相关的未知量可以最先被标记（比如，第一个速度分量），接下来是第二类的变量（比如，第二个速度分量）等等。在任意情况下，很明显邻接矩阵是有冗余信息的。物理网格的商图可以被用来替代使用。这将节约大量的存储量和计算量。在上述的流体流动的例子中，用以描述图的整数数组的存储可以被缩小到接近 1/16。这是因为边的数量被所见到了大约这么多，但是通常很小的顶点数却保持着不变。

3.3 置换和重新排序

对于稀疏矩阵而言，重排序行或列，或者行和列是一个常见操作。事实上，重排序行和列是一个用于直接求解法和迭代法并行实现的一个最重要的部分。本节介绍这些重排技术和矩阵的邻接图的相关思想的关系。记得在第一章中，矩阵的第 j 列记作 a_{*j}，第 i 行则记作 a_{i*}。

3.3.1 基础概念

我们先开始一个定义与符号。

定义 3.1 有一个矩阵 A，以及 $\pi = \{i_1, i_2, \cdots, i_n\}$ 的一个交换集合 $\pi = \{i_1, i_2, \cdots, i_n\}$。那么矩阵

$$A_{\pi,*} = \{a_{\pi(i),j}\}_{i=1,\cdots,n;j=1,\cdots,m}$$

$$A_{*,\pi} = \{i, a_{\pi(j)}\}_{i=1,\cdots,n;j=1,\cdots,m}$$

就分别被称作 A 的行 π-交换和列 π-交换。

广为周知的时，最多 n 个交换（换而言之，就是只互换两项的基本排列）可以产生集合 $\{1, 2, \cdots, n\}$ 的任意置换。一个交换矩阵就是一个两行互换了得单位矩阵。用 X_{ij} 来表示第 i 和 j 行交换了的单位矩阵。注意到，为了交换矩阵 A 的第 i 和 j 行，我们可以用矩阵 X_{ij} 来左乘矩阵 A。让 $\pi = \{i_1, i_2, \cdots, i_n\}$ 为一个任意序列。这个置换就是一系列连续的交换矩阵 $\sigma(i_k, j_k), k = 1, \cdots, n$ 的乘积。那么，我们就可以通过交换矩阵的 i_1 和 j_1 行，然后再结果矩阵的基础上交换 i_2 和 j_2 行，以此类推，最后交换 i_n 和 j_n 行。每一步我们都可以通过左乘矩阵 X_{i_k,j_k} 来实现。同样的，对于矩阵的列也是一样的：为了交换矩阵的第 i 和 k 列，通过右乘矩阵 X_{i_k,j_k} 可以实现。从上我们可以得到下述命题。

命题 3.1 让 π 是交换 $\sigma(i_k, j_k), k = 1, \cdots, n$ 的乘积得到的置换。那么，$A_{\pi,*} = P_\pi A$，$A_{*,\pi} = AQ_\pi$，当

$$P_\pi = X_{i_n,j_n} X_{i_{n-1},j_{n-1}} \cdots X_{i_1,j_1} \tag{3.1}$$

$$Q_\pi = X_{i_1,j_1} X_{i_2,j_2} \cdots X_{i_n,j_n} \tag{3.2}$$

这些交换矩阵的乘积被称作置换矩阵。显然，一个置换矩阵只不过是进行了行列交换的单位矩阵。

注意到 $X_{i,j}^2 = I$，换而言之，置换矩阵的平方是一个单位矩阵，或者等价地，置换矩阵的逆等于它本身，这是很显然的一个属性。易见，矩阵（3.1）和（3.2）满足

$$P_\pi Q_\pi = X_{i_n,j_n} X_{i_{n-1},j_{n-1}} \cdots X_{i_1,j_1} \times X_{i_1,j_1} X_{i_2,j_2} \cdots X_{i_n,j_n}$$

表示了矩阵 Q_π 和 P_π 都是非退化的，且互为另一个的逆。换而言之，用同一个置换矩阵来交换一个矩阵的行和列事实上做了类似的变换。因为定义 (3.1) 和（3.2）中得 P_π 和 Q_π 的乘积是相反的顺序，另一个推论就显而易见了。由于每一个基矩阵 XX_{i_k,j_k} 是对称的，那么 Q_π 是 P_π 的转置。因此

$$Q_\pi = P_\pi^T = P_\pi^{-1}$$

因为矩阵 P_π 的逆矩阵是它的转置，置换矩阵就是唯一的。
另一个用来推出上述关系的方法是用置换矩阵 P_π 和 P_π^T 来代表行列交换了的单位矩阵。（在练习 3 中）显而易见

$$P_\pi = I_{\pi,*}\ P_\pi^T = I_{*,\pi}$$

那么，接下来就可以直接验证

$$A_{\pi,*} = I_{\pi,*}A = P_\pi A\ A_{*,\pi} = AI_{*,\pi} = AP_\pi^T$$

这对于在线性系统中解释置换操作很重要。当矩阵的行交换了，方程的顺序就改变了。换而言之，当列交换了，那么未知量就会对应的改变标记或者改变顺序。
例子 3.1 思考，比如，线性系统 $Ax = b$，当

$$A = \begin{pmatrix} a_{11} & 0 & a_{13} & 0 \\ 0 & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & 0 \\ 0 & a_{42} & 0 & a_{44} \end{pmatrix}$$

以及 $\pi = \{1, 3, 2, 4\}$，那么（列）交换线性系统是

$$\begin{pmatrix} a_{11} & a_{13} & 0 & 0 \\ 0 & a_{23} & a_{22} & a_{24} \\ a_{31} & a_{33} & a_{32} & 0 \\ 0 & 0 & a_{42} & a_{44} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix}$$

注意到，不只是未知量交换了，方程也是，特别的，右边没有变。
在上述例子中，只有 A 的列交换了。在稀疏矩阵技术中，这样的单侧变换不如两侧变换寻常。事实上，这通常与线性系统中得对角元起着一个明显且重要的角色的事实有关。比如，在偏微分应用中，对角元通常很大，而且，在交换矩阵中可能很需要去保留这一性质。为了达到这一目的，很典型的是去同时对 A 的行和列进行相同的交换。这样的操作被叫做对称置换，若果用 $A_{\pi,\pi}$ 来表示，那么，这样的对称置换的结果就满足这一关系

$$A_{\pi,\pi} = P_\pi^T A P_\pi$$

对称置换的解释很简单。由此产生的矩阵用相同的方式重命名，或重标记，或重排序未知量和重排序等式。

例子 3.2 对于前面的例子，如果行和列用相同的置换矩阵来置换，那么线性系统可以这样来获得

$$\begin{pmatrix} a_{11} & a_{13} & 0 & 0 \\ a_{31} & a_{33} & a_{32} & 0 \\ 0 & a_{23} & a_{22} & a_{24} \\ 0 & 0 & a_{42} & a_{44} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix}$$

注意到，对角元是原来的矩阵的对角元在主对角线上德一个不一样的顺序。

3.3.2 与邻接图的关系

从图论的角度，另一个对于对称置换的重要解释是这相当于不不改变边来重新标记顶点。事实上，(i,j) 是原矩阵 A 的邻接图的边，A' 是交换了得矩阵。当且仅当 $(\pi(i), \pi(j))$ 是原始矩阵 A 的图中的一条边时，那么 $a'_{ij} = a_{\pi(i),\pi(j)}$，结果 (i,j) 是交换了的矩阵 A' 的邻接图中一条边。因此，交换了的矩阵的图没有改变；甚至，顶点的标记也是。与之相对的是，不对称置换就不会保存好图。事实上，它们可以将一个无向图转换为一个有向图。尽管邻接矩阵的一般图是相同的，对称置换可能会对矩阵的结构造成一些重大的影响。

例子 3.3 考虑图 3.4 描述的矩阵和它的邻接图。因为它们的形状，这样的矩阵又是被叫做“箭头”矩阵，但是，因为它们图的结构可能更适合把它们叫做“星”矩阵。

如果用置换 $9, 8, \cdots, 1$ 来重排序等式，图 3.5 所描述的矩阵和图就得到了。尽管两张土建的区别看起来很小，但是矩阵可能会有一个对于算法有着重要影响的完全不同的结构。以此为例，如果用高斯消去法来重排序矩阵，那么填充就不会发生。换而言之，LU 分解的 L 和 U 部分会与 A 的下和上两部分有着一样的结构。在另一方面，在原始矩阵上作高斯消去法会导致灾难性的填充。特别的，在高斯消去法第一部以后，LU 分解的 L 和 U 部分是稠密矩阵。用直接稀疏矩阵技术，找到在高斯消去过程中对于较少填充有作用的矩阵的置换很重要。最后这一段，总的来说，两侧非对称置换也可能在实践中出现。然而，在直接法中它们很常见。

3.3.3 常用重排序

在实践中，重排序和置换的种类七绝与直接或者迭代法是否被考虑。下面是一个这样的对赌迭代法更为有用的一个重排序的例子。

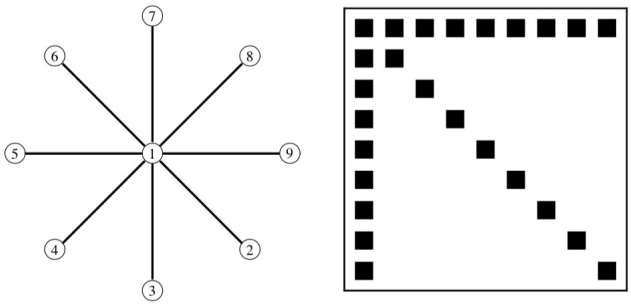


Figure 3.4 *Pattern of a 9 × 9 arrow matrix and its adjacency graph.*

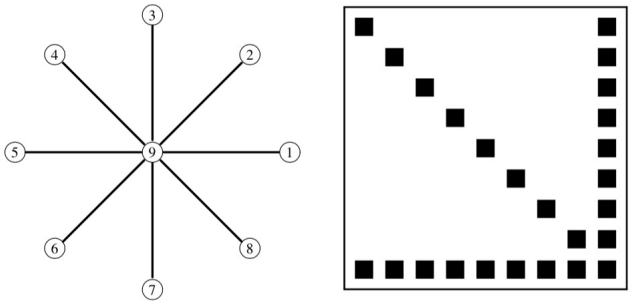


Figure 3.5 *Adjacency graph and matrix obtained from above figure after permuting the nodes in reverse order.*

水平集序这种顺序类型包含了许多基于水平集的图的便利的技巧。水平集是递归定义的上一级的所有节点的所有未标记的邻集。最初，一个水平集有一个节点，虽然有几个将来会被讨论的也重要的起始点。当一个水平集被遍历完成，它的节点就被标记了且排了编号。比如，它们可以按照遍历的顺序来编号。另外，遍历顺序的不同会产生不同的顺序。例如，某一个水平集中的节点可以按照它们列出的自然序访问。然后，可以检查它们每一个的邻节点。每一次，当遇到一个访问过的顶点有一个没有编号的邻节点，那么它就被添加到列表中并标记为下一水平集的下一元素。在图论中，这个简单地策略被称为广度优先搜索。在每个水平集中，顺序取决于节点遍历的方式。在广度优先搜索中，水平集中元素总是以它们列出的自然序来遍历。在 Cuthill-McKee 排序中，水平集的元素被以从最低到最高的顺序来遍历。

4.2 原文

CHAPTER

3

SPARSE MATRICES

As described in the previous chapter, standard discretizations of Partial Differential Equations typically lead to large and *sparse* matrices. A sparse matrix is defined, somewhat vaguely, as a matrix which has very few nonzero elements. But, in fact, a matrix can be termed sparse whenever special techniques can be utilized to take advantage of the large number of zero elements and their locations. These sparse matrix techniques begin with the idea that the zero elements need not be stored. One of the key issues is to define data structures for these matrices that are well suited for efficient implementation of standard solution methods, whether direct or iterative. This chapter gives an overview of sparse matrices, their properties, their representations, and the data structures used to store them.

3.1 INTRODUCTION

The natural idea to take advantage of the zeros of a matrix and their location was initiated by engineers in various disciplines. In the simplest case involving banded matrices, special techniques are straightforward to develop. Electrical engineers dealing with electrical networks in the 1960s were the first to exploit sparsity to solve general sparse linear systems for matrices with irregular structure. The main issue, and the first addressed by sparse matrix technology, was to devise direct solution methods for linear systems. These had to be economical, both in terms of storage and computational effort. Sparse direct solvers can handle very large problems that cannot be tackled by the usual "dense" solvers.

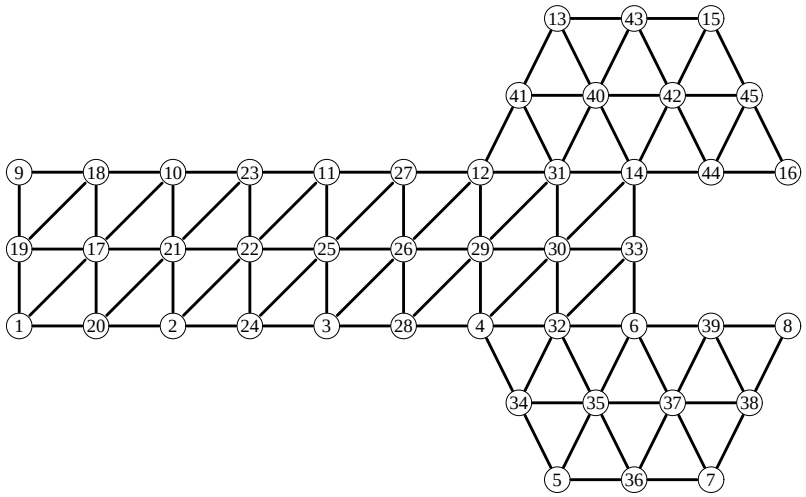


Figure 3.1 *A finite element grid model.*

Essentially, there are two broad types of sparse matrices: *structured* and *unstructured*. A structured matrix is one whose nonzero entries form a regular pattern, often along a small number of diagonals. Alternatively, the nonzero elements may lie in blocks (dense submatrices) of the same size, which form a regular pattern, typically along a small number of (block) diagonals. A matrix with irregularly located entries is said to be irregularly structured. The best example of a regularly structured matrix is a matrix that consists of only a few diagonals. Finite difference matrices on rectangular grids, such as the ones seen in the previous chapter, are typical examples of matrices with regular structure. Most finite element or finite volume techniques applied to complex geometries lead to irregularly structured matrices. Figure 3.2 shows a small irregularly structured sparse matrix associated with the finite element grid problem shown in Figure 3.1.

The distinction between the two types of matrices may not noticeably affect direct solution techniques, and it has not received much attention in the past. However, this distinction can be important for iterative solution methods. In these methods, one of the essential operations is matrix-by-vector products. The performance of these operations can differ significantly on high performance computers, depending on whether they are regularly structured or not. For example, on vector computers, storing the matrix by diagonals is ideal, but the more general schemes may suffer because they require indirect addressing.

The next section discusses graph representations of sparse matrices. This is followed by an overview of some of the storage schemes used for sparse matrices and an explanation of how some of the simplest operations with sparse matrices can be performed. Then sparse linear system solution methods will be covered. Finally, Section 3.7 discusses test matrices.

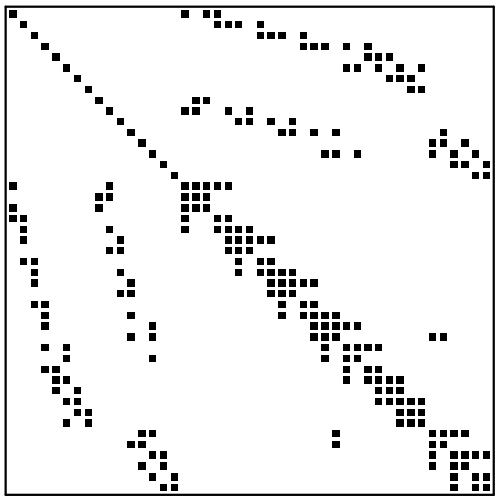


Figure 3.2 *Sparse matrix associated with the finite element grid of Figure 3.1.*

3.2 GRAPH REPRESENTATIONS

Graph theory is an ideal tool for representing the structure of sparse matrices and for this reason it plays a major role in sparse matrix techniques. For example, graph theory is the key ingredient used in unraveling parallelism in sparse Gaussian elimination or in preconditioning techniques. In the following section, graphs are discussed in general terms and then their applications to finite element or finite difference matrices are discussed.

3.2.1 GRAPHS AND ADJACENCY GRAPHS

Remember that a graph is defined by two sets, a set of vertices

$$V = \{v_1, v_2, \ldots, v_n\},$$

and a set of edges E which consists of pairs (v_i, v_j), where v_i, v_j are elements of V, i.e.,

$$E \subseteq V \times V.$$

This graph $G = (V, E)$ is often represented by a set of points in the plane linked by a directed line between the points that are connected by an edge. A graph is a way of representing a binary relation between objects of a set V. For example, V can represent the major cities of the world. A line is drawn between any two cities that are linked by a nonstop airline connection. Such a graph will represent the relation "there is a nonstop flight from city (A) to city (B)." In this particular example, the binary relation is likely to

be symmetric, i.e., when there is a nonstop flight from (A) to (B) there is also a nonstop flight from (B) to (A). In such situations, the graph is said to be undirected, as opposed to a general graph which is directed.

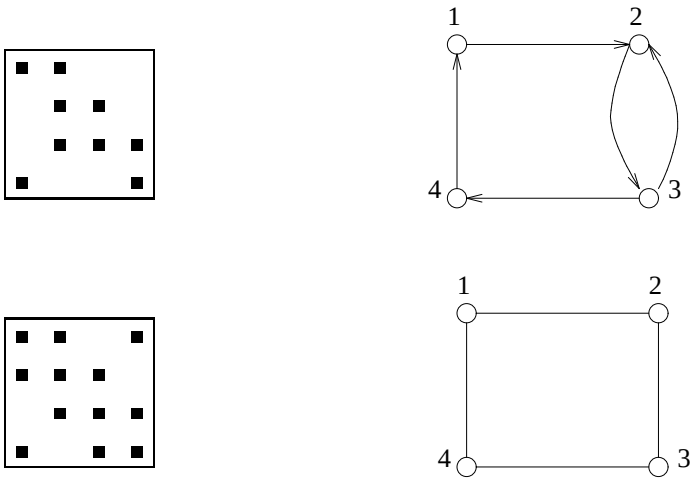


Figure 3.3 *Graphs of two* 4×4 *sparse matrices.*

Going back to sparse matrices, the *adjacency graph* of a sparse matrix is a graph $G = (V, E)$, whose n vertices in V represent the n unknowns. Its edges represent the binary relations established by the equations in the following manner: There is an edge from node i to node j when $a_{ij} \neq 0$. This edge will therefore represent the binary relation *equation* i *involves unknown* j. Note that the graph is directed, except when the matrix has a symmetric pattern ($a_{ij} \neq 0$ iff $a_{ji} \neq 0$ for all $1 \leq i, j \leq n$).

When a matrix has a symmetric nonzero pattern, i.e., when a_{ij} and a_{ji} are always nonzero at the same time, then the graph is *undirected*. Thus, for undirected graphs, every edge points in both directions. As a result, undirected graphs can be represented with nonoriented edges.

As an example of the use of graph models, parallelism in Gaussian elimination can be extracted by finding unknowns that are independent at a given stage of the elimination. These are unknowns which do not depend on each other according to the above binary relation. The rows corresponding to such unknowns can then be used as pivots simultaneously. Thus, in one extreme, when the matrix is diagonal, then all unknowns are independent. Conversely, when a matrix is dense, each unknown will depend on all other unknowns. Sparse matrices lie somewhere between these two extremes.

There are a few interesting simple properties of adjacency graphs. The graph of A^2 can be interpreted as an n-vertex graph whose edges are the pairs (i, j) for which there exists at least one path of length exactly two from node i to node j in the original graph of A. Similarly, the graph of A^k consists of edges which represent the binary relation "there is at least one path of length k from node i to node j." For details, see Exercise 4.

3.2.2 GRAPHS OF PDE MATRICES

For Partial Differential Equations involving only one physical unknown per mesh point, the adjacency graph of the matrix arising from the discretization is often the graph represented by the mesh itself. However, it is common to have several unknowns per mesh point. For example, the equations modeling fluid flow may involve the two velocity components of the fluid (in two dimensions) as well as energy and momentum at each mesh point. In such situations, there are two choices when labeling the unknowns. They can be labeled contiguously at each mesh point. Thus, for the example just mentioned, we can label all four variables (two velocities followed by momentum and then pressure) at a given mesh point as $u(k), \ldots, u(k+3)$. Alternatively, all unknowns associated with one type of variable can be labeled first (e.g., first velocity components), followed by those associated with the second type of variables (e.g., second velocity components), etc. In either case, it is clear that there is redundant information in the graph of the adjacency matrix. The *quotient* graph corresponding to the *physical mesh* can be used instead. This results in substantial savings in storage and computation. In the fluid flow example mentioned above, the storage can be reduced by a factor of almost 16 for the integer arrays needed to represent the graph. This is because the number of edges has been reduced by this much, while the number of vertices, which is usually much smaller, remains the same.

3.3 PERMUTATIONS AND REORDERINGS

Permuting the rows or the columns, or both the rows and columns, of a sparse matrix is a common operation. In fact, *reordering* rows and columns is one of the most important ingredients used in *parallel* implementations of both direct and iterative solution techniques. This section introduces the ideas related to these reordering techniques and their relations to the adjacency graphs of the matrices. Recall the notation introduced in Chapter 1 that the j-th column of a matrix is denoted by a_{*j} and the i-th row by a_{i*}.

3.3.1 BASIC CONCEPTS

We begin with a definition and new notation.

DEFINITION 3.1 *Let A be a matrix and $\pi = \{i_1, i_2, \ldots, i_n\}$ a permutation of the set $\{1, 2, \ldots, n\}$. Then the matrices*

$$A_{\pi,*} = \{a_{\pi(i),j}\}_{i=1,\ldots,n; j=1,\ldots,m},$$
$$A_{*,\pi} = \{a_{i,\pi(j)}\}_{i=1,\ldots,n; j=1,\ldots,m}$$

are called row π-permutation and column π-permutation of A, respectively.

It is well known that any permutation of the set $\{1, 2, \ldots, n\}$ results from at most n interchanges, i.e., elementary permutations in which only two entries have been interchanged. An *interchange matrix* is the identity matrix with two of its rows interchanged. Denote by X_{ij} such matrices, with i and j being the numbers of the interchanged rows. Note that in order to interchange rows i and j of a matrix A, we only need to premultiply it by the matrix X_{ij}. Let $\pi = \{i_1, i_2, \ldots, i_n\}$ be an arbitrary permutation. This permutation is the product of a sequence of n consecutive interchanges $\sigma(i_k, j_k), k = 1, \ldots, n$. Then the rows of a matrix can be permuted by interchanging rows i_1, j_1, then rows i_2, j_2 of the resulting matrix, etc., and finally by interchanging i_n, j_n of the resulting matrix. Each of these operations can be achieved by a premultiplication by X_{i_k,j_k}. The same observation can be made regarding the columns of a matrix: In order to interchange columns i and j of a matrix, postmultiply it by X_{ij}. The following proposition follows from these observations.

PROPOSITION 3.1 *Let π be a permutation resulting from the product of the interchanges $\sigma(i_k, j_k)$, $k = 1, \ldots, n$. Then,*

$$A_{\pi,*} = P_\pi A, \quad A_{*,\pi} = AQ_\pi,$$

where

$$P_\pi = X_{i_n,j_n} X_{i_{n-1},j_{n-1}} \ldots X_{i_1,j_1}, \tag{3.1}$$

$$Q_\pi = X_{i_1,j_1} X_{i_2,j_2} \ldots X_{i_n,j_n}. \tag{3.2}$$

Products of interchange matrices are called *permutation matrices*. Clearly, a permutation matrix is nothing but the identity matrix with its rows (or columns) permuted.

Observe that $X_{i,j}^2 = I$, i.e., the square of an interchange matrix is the identity, or equivalently, the inverse of an interchange matrix is equal to itself, a property which is intuitively clear. It is easy to see that the matrices (3.1) and (3.2) satisfy

$$P_\pi Q_\pi = X_{i_n,j_n} X_{i_{n-1},j_{n-1}} \ldots X_{i_1,j_1} \times X_{i_1,j_1} X_{i_2,j_2} \ldots X_{i_n,j_n} = I,$$

which shows that the two matrices Q_π and P_π are nonsingular and that they are the inverse of one another. In other words, permuting the rows and the columns of a matrix, *using the same permutation*, actually performs a similarity transformation. Another important consequence arises because the products involved in the definitions (3.1) and (3.2) of P_π and Q_π occur in reverse order. Since each of the elementary matrices X_{i_k,j_k} is symmetric, the matrix Q_π is the transpose of P_π. Therefore,

$$Q_\pi = P_\pi^T = P_\pi^{-1}.$$

Since the inverse of the matrix P_π is its own transpose, permutation matrices are unitary.

Another way of deriving the above relationships is to express the permutation matrices P_π and P_π^T in terms of the identity matrix, whose columns or rows are permuted. It can easily be seen (See Exercise 3) that

$$P_\pi = I_{\pi,*}, \quad P_\pi^T = I_{*,\pi}.$$

It is then possible to verify directly that

$$A_{\pi,*} = I_{\pi,*}A = P_\pi A, \quad A_{*,\pi} = AI_{*,\pi} = AP_\pi^T.$$

It is important to interpret permutation operations for the linear systems to be solved. When the rows of a matrix are permuted, the order in which the equations are written is changed. On the other hand, when the columns are permuted, the unknowns are in effect *relabeled*, or *reordered*.

Example 3.1 Consider, for example, the linear system $Ax = b$ where

$$A = \begin{pmatrix} a_{11} & 0 & a_{13} & 0 \\ 0 & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & 0 \\ 0 & a_{42} & 0 & a_{44} \end{pmatrix}$$

and $\pi = \{1, 3, 2, 4\}$, then the (column-) permuted linear system is

$$\begin{pmatrix} a_{11} & a_{13} & 0 & 0 \\ 0 & a_{23} & a_{22} & a_{24} \\ a_{31} & a_{33} & a_{32} & 0 \\ 0 & 0 & a_{42} & a_{44} \end{pmatrix} \begin{pmatrix} x_1 \\ x_3 \\ x_2 \\ x_4 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix}.$$

Note that only the unknowns have been permuted, not the equations, and in particular, the right-hand side has not changed.

In the above example, only the columns of A have been permuted. Such one-sided permutations are not as common as two-sided permutations in sparse matrix techniques. In reality, this is often related to the fact that the diagonal elements in linear systems play a distinct and important role. For instance, diagonal elements are typically large in PDE applications and it may be desirable to preserve this important property in the permuted matrix. In order to do so, it is typical to apply the same permutation to both the columns and the rows of A. Such operations are called *symmetric permutations*, and if denoted by $A_{\pi,\pi}$, then the result of such symmetric permutations satisfies the relation

$$A_{\pi,\pi} = P_\pi^T A P_\pi.$$

The interpretation of the symmetric permutation is quite simple. The resulting matrix corresponds to renaming, or relabeling, or reordering the unknowns and then reordering the equations in the same manner.

Example 3.2 For the previous example, if the rows are permuted with the same permutation as the columns, the linear system obtained is

$$\begin{pmatrix} a_{11} & a_{13} & 0 & 0 \\ a_{31} & a_{33} & a_{32} & 0 \\ 0 & a_{23} & a_{22} & a_{24} \\ 0 & 0 & a_{42} & a_{44} \end{pmatrix} \begin{pmatrix} x_1 \\ x_3 \\ x_2 \\ x_4 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_3 \\ b_2 \\ b_4 \end{pmatrix}.$$

Observe that the diagonal elements are now diagonal elements from the original matrix, placed in a different order on the main diagonal.

3.3.2 RELATIONS WITH THE ADJACENCY GRAPH

From the point of view of graph theory, another important interpretation of a symmetric permutation is that *it is equivalent to relabeling the vertices of the graph* without altering the edges. Indeed, let (i, j) be an edge in the adjacency graph of the original matrix A and let A' be the permuted matrix. Then $a'_{ij} = a_{\pi(i),\pi(j)}$ and a result (i, j) is an edge in the adjacency graph of the permuted matrix A', if and only if $(\pi(i), \pi(j))$ is an edge in the graph of the original matrix A. Thus, the graph of the permuted matrix has not changed; rather, the labeling of the vertices has. In contrast, nonsymmetric permutations do not preserve the graph. In fact, they can transform an indirected graph into a directed one. Symmetric permutations may have a tremendous impact on the structure of the matrix even though the general graph of the adjacency matrix is identical.

Example 3.3 Consider the matrix illustrated in Figure 3.4 together with its adjacency graph. Such matrices are sometimes called "arrow" matrices because of their shape, but it would probably be more accurate to term them "star" matrices because of the structure of their graphs.

If the equations are reordered using the permutation $9, 8, \ldots, 1$, the matrix and graph shown in Figure 3.5 are obtained. Although the difference between the two graphs may seem slight, the matrices have a completely different structure, which may have a significant impact on the algorithms. As an example, if Gaussian elimination is used on the reordered matrix, no fill-in will occur, i.e., the L and U parts of the LU factorization will have the same structure as the lower and upper parts of A, respectively. On the other hand, Gaussian elimination on the original matrix results in disastrous fill-ins. Specifically, the L and U parts of the LU factorization are now dense matrices after the first step of Gaussian elimination. With direct sparse matrix techniques, it is important to find permutations of the matrix that will have the effect of reducing fill-ins during the Gaussian elimination process.

To conclude this section, it should be mentioned that two-sided nonsymmetric permutations may also arise in practice. However, they are more common in the context of direct methods.

3.3.3 COMMON REORDERINGS

The type of reordering, or permutations, used in applications depends on whether a direct or an iterative method is being considered. The following is a sample of such reorderings which are more useful for iterative methods.

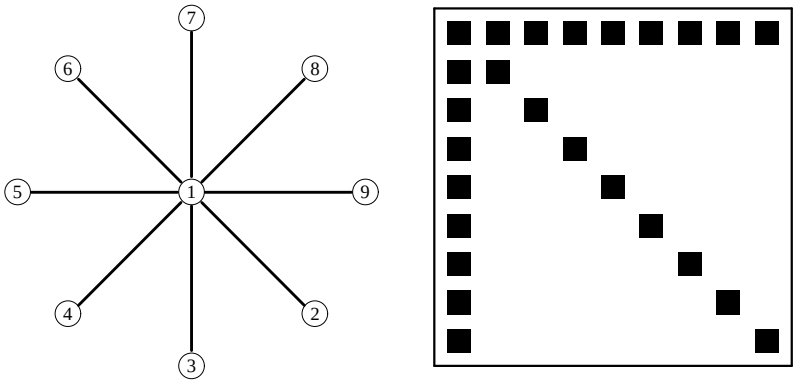


Figure 3.4 *Pattern of a 9 × 9 arrow matrix and its adjacency graph.*

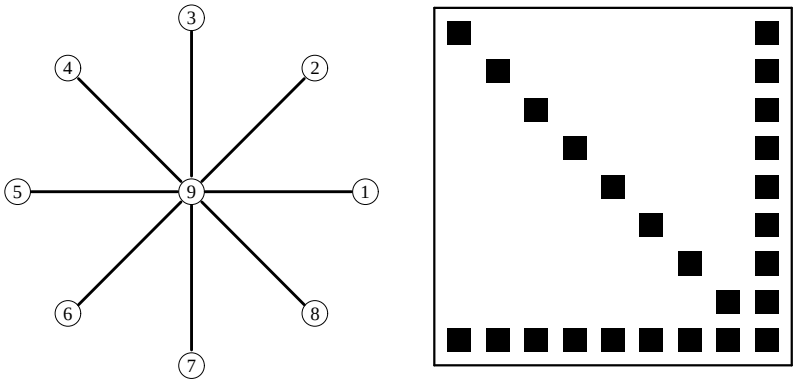


Figure 3.5 *Adjacency graph and matrix obtained from above figure after permuting the nodes in reverse order.*

Level-set orderings. This class of orderings contains a number of techniques that are based on traversing the graph by *level sets*. A level set is defined recursively as the set of all unmarked neighbors of all the nodes of a previous level set. Initially, a level set consists of one node, although strategies with several starting nodes are also important and will be considered later. As soon as a level set is traversed, its nodes are marked and numbered. They can, for example, be numbered in the order in which they are traversed. In addition, the order in which each level itself is traversed gives rise to different orderings. For instance, the nodes of a certain level can be visited in the natural order in which they are listed. The neighbors of each of these nodes are then inspected. Each time, a neighbor of a visited vertex that is not numbered is encountered, it is added to the list and labeled as

the next element of the next level set. This simple strategy is called *Breadth First Search* (BFS) traversal in graph theory. The ordering will depend on the way in which the nodes are traversed in each level set. In BFS the elements of a level set are always traversed in the natural order in which they are listed. In the *Cuthill-McKee ordering* the elements of a level set are traversed from the nodes of lowest degree to those of highest degree.

ALGORITHM 3.1: Cuthill-McKee Ordering

1. *Input: initial node* i_1*; Output: permutation array* `iperm` .
2. *Start: Set* $levset := \{i_1\}$*;* $next = 2$*;*
3. *Set* $marker(i_1) = 1$*;* `iperm` $(1) = i_1$
4. *While* $(next < n)$ *Do:*
5. $Next\_levset = \emptyset$
6. *Traverse* $levset$ *in order of increasing degree and*
7. *for each visited node Do:*
8. *For each neighbor* i *of* j *such that* $marker(i) = 0$ *Do:*
9. *Add* i *to the set* $Next\_levset$
10. $marker(i) := 1$*;* `iperm` $(next) = i$
11. $next = next + 1$
12. *EndDo*
13. *EndDo*
14. $levset := Next\_levset$
15. *EndWhile*

The `iperm` array obtained from the procedure lists the nodes in the order in which they are visited and can, in a practical implementation, be used to store the level sets in succession. A pointer is needed to indicate where each set starts. The array `iperm` thus constructed does in fact represent the permutation array π defined earlier.

In 1971, George [103] observed that *reversing* the Cuthill-McKee ordering yields a better scheme for sparse Gaussian elimination. The simplest way to understand this is to look at the two graphs produced by these orderings. The results of the standard and reversed Cuthill-McKee orderings on the sample finite element mesh problem seen earlier are shown in Figures 3.6 and 3.7, when the initial node is $i_1 = 3$ (relative to the labeling of the original ordering of Figure 2.10). The case of the figure, corresponds to a variant of CMK in which the traversals in Line 6, is done in a random order instead of according to the degree. A large part of the structure of the two matrices consists of little "arrow" submatrices, similar to the ones seen in Example 3.3. In the case of the regular CMK ordering, these arrows point upward, as in Figure 3.4, a consequence of the level set labeling. These blocks are similar the star matrices of Figure 3.4. As a result, Gaussian elimination will essentially fill in the square blocks which they span. As was indicated in Example 3.3, a remedy is to reorder the nodes backward, as is done globally in the reverse Cuthill-McKee strategy. For the reverse CMK ordering, the arrows are pointing downward, as in Figure 3.5, and Gaussian elimination yields much less fill-in.

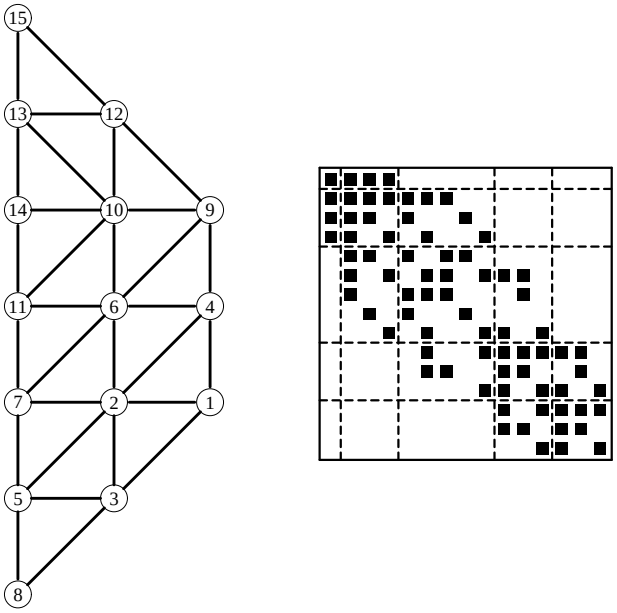


Figure 3.6 *Cuthill-McKee ordering.*

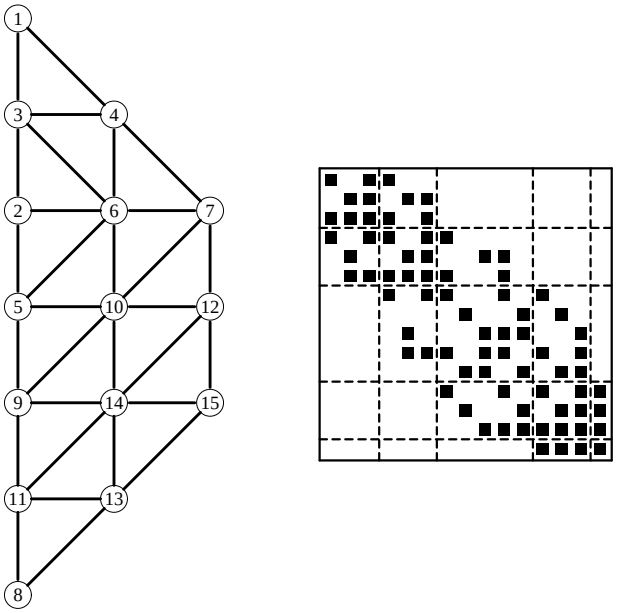


Figure 3.7 *Reverse Cuthill-McKee ordering.*

Example 3.4 The choice of the initial node in the CMK and RCMK orderings may be important. Referring to the original ordering of Figure 2.10, the previous illustration used $i_1 = 3$. However, it is clearly a poor choice if matrices with small bandwidth or *profile* are desired. If $i_1 = 1$ is selected instead, then the reverse Cuthill-McKee algorithm produces the matrix in Figure 3.8, which is more suitable for banded or *skyline* solvers.

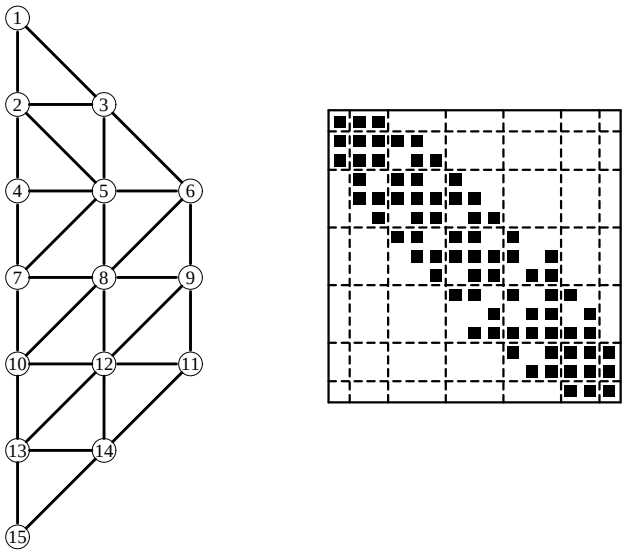


Figure 3.8 *Reverse Cuthill-McKee ordering starting with* $i_1 = 1$.

Independent set orderings. The matrices that arise in the model finite element problems seen in Figures 2.7, 2.10, and 3.2 are all characterized by an upper-left block that is diagonal, i.e., they have the structure

$$A = \begin{pmatrix} D & E \\ F & C \end{pmatrix}, \tag{3.3}$$

in which D is diagonal and C, E, and F are sparse matrices. The upper-diagonal block corresponds to unknowns from the previous levels of refinement and its presence is due to the ordering of the equations in use. As new vertices are created in the refined grid, they are given new numbers and the initial numbering of the vertices is unchanged. Since the old connected vertices are "cut" by new ones, they are no longer related by equations. Sets such as these are called *independent sets*. Independent sets are especially useful in parallel computing, for implementing both direct and iterative methods.

Referring to the adjacency graph $G = (V, E)$ of the matrix, and denoting by (x, y) the edge from vertex x to vertex y, an *independent set* S is a subset of the vertex set V such that

$$\text{if } x \in S, \quad \text{then} \quad \{(x,y) \in E \text{ or } (y,x) \in E\} \rightarrow y \notin S.$$

To explain this in words: Elements of S are not allowed to be connected to other elements of S either by incoming or outgoing edges. An independent set is *maximal* if it cannot be augmented by elements in its complement to form a larger independent set. Note that a maximal independent set is by no means the largest possible independent set that can be found. In fact, finding the independent set of maximum cardinal is NP-hard [132]. In the following, the term *independent set* always refers to *maximal independent set*.

There are a number of simple and inexpensive heuristics for finding large maximal independent sets. A greedy heuristic traverses the nodes in a given order, and if a node is not already marked, it selects the node as a new member of S. Then this node is marked along with its nearest neighbors. Here, a nearest neighbor of a node x means any node linked to x by an incoming or an outgoing edge.

ALGORITHM 3.2: Greedy Algorithm for ISO

1. *Set* $S = \emptyset$.
2. *For* $j = 1, 2, \ldots, n$ *Do:*
3. *If node* j *is not marked then*
4. $S = S \cup \{j\}$
5. *Mark* j *and all its nearest neighbors*
6. *EndIf*
7. *EndDo*

In the above algorithm, the nodes are traversed in the natural order $1, 2, \ldots, n$, but they can also be traversed in any permutation $\{i_1, \ldots, i_n\}$ of $\{1, 2, \ldots, n\}$. Since the size of the reduced system is $n - |S|$, it is reasonable to try to maximize the size of S in order to obtain a small reduced system. It is possible to give a rough idea of the size of S. Assume that the maximum degree of each node does not exceed ν. Whenever the above algorithm accepts a node as a new member of S, it potentially puts all its nearest neighbors, i.e., at most ν nodes, in the complement of S. Therefore, if s is the size of S, the size of its complement, $n - s$, is such that $n - s \leq \nu s$, and as a result,

$$s \geq \frac{n}{1 + \nu}.$$

This lower bound can be improved slightly by replacing ν with the maximum degree ν_S of *all the vertices that constitute* S. This results in the inequality

$$s \geq \frac{n}{1 + \nu_S},$$

which suggests that it may be a good idea to first visit the nodes with smaller degrees. In fact, this observation leads to a general heuristic regarding a good order of traversal. The algorithm can be viewed as follows: Each time a node is visited, remove it and its nearest neighbors from the graph, and then visit a node from the remaining graph. Continue in the same manner until all nodes are exhausted. Every node that is visited is a member of S and its nearest neighbors are members of $\bar{S}$. As result, if ν_i is the degree of the node visited at step i, adjusted for all the edge deletions resulting from the previous visitation steps, then the number n_i of nodes that are left at step i satisfies the relation

$$n_i = n_{i-1} - \nu_i - 1.$$

The process adds a new element to the set S at each step and stops when $n_i = 0$. In order to maximize $|S|$, the number of steps in the procedure must be maximized. The difficulty in the analysis arises from the fact that the degrees are updated at each step i because of the removal of the edges associated with the removed nodes. If the process is to be lengthened, a rule of thumb would be to visit the nodes that have the smallest degrees first.

ALGORITHM 3.3: Increasing Degree Traversal for ISO

1. *Set $S = \emptyset$. Find an ordering $i_1, \ldots, i_n$ of the nodes by increasing degree.*
2. *For $j = 1, 2, \ldots n$, Do:*
3. *If node i_j is not marked then*
4. *$S = S \cup \{i_j\}$*
5. *Mark i_j and all its nearest neighbors*
6. *EndIf*
7. *EndDo*

A refinement to the above algorithm would be to update the degrees of all nodes involved in a removal, and dynamically select the one with the smallest degree as the next node to be visited. This can be implemented efficiently using a min-heap data structure. A different heuristic is to attempt to maximize the number of elements in S by a form of local optimization which determines the order of traversal dynamically. In the following, removing a vertex from a graph means deleting the vertex and all edges incident to/from this vertex.

Example 3.5 The algorithms described in this section were tested on the same example used before, namely, the finite element mesh problem of Figure 2.10. Here, all strategies used yield the initial independent set in the matrix itself, which corresponds to the nodes of all the previous levels of refinement. This may well be optimal in this case, i.e., a larger independent set may not exist.

Multicolor orderings. Graph coloring is a familiar problem in computer science which refers to the process of labeling (coloring) the nodes of a graph in such a way that no two adjacent nodes have the same label (color). The goal of graph coloring is to obtain a colored graph which uses the smallest possible number of colors. However, optimality in the context of numerical linear algebra is a secondary issue and simple heuristics do provide adequate colorings.

Basic methods for obtaining a multicoloring of an arbitrary grid are quite simple. They rely on greedy techniques, a simple version of which is as follows.

ALGORITHM 3.4: Greedy Multicoloring Algorithm

1. *For $i = 1, \ldots, n$ Do: set $Color(i) = 0$.*
2. *For $i = 1, 2, \ldots, n$ Do:*
3. *Set Color$(i) = \min \{k > 0 \mid k \neq$ Color$(j), \forall\, j \in \mathrm{Adj}(i))\}$*
4. *EndDo*

毕业论文（设计）文献综述和开题报告考核

一、对文献综述、外文翻译和**开题报告**评语及成绩评定：

项目选题合理，前期文献调研充分，研究计划切实可行，研究目标恰当且有一定的挑战性。外文文献翻译准确。同意按开题报告进行项目研究。

成绩比例	文献综述 占（10%）	开题报告 占（20%）	外文翻译 占（10%）
分 值			

开题报告答辩小组负责人（签名）_____

年　　月　　日