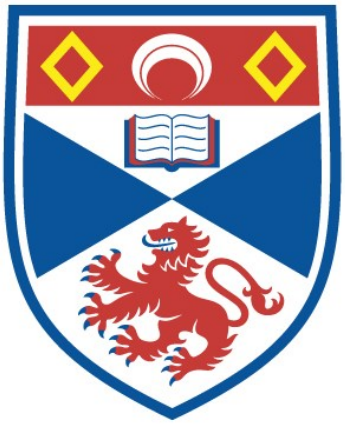
FINDING "SMALL' MATRICES P,Q SUCH THAT PDQ = S

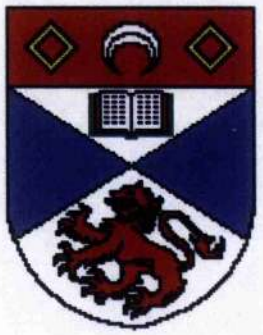
Robert J. Wainwright

A Thesis Submitted for the Degree of PhD
at the
University of St Andrews



2002

Finding 'small' matrices P, Q such that $PDQ = S$



A thesis submitted to the
UNIVERSITY OF ST ANDREWS
for the degree of
DOCTOR OF PHILOSOPHY

by

Robert J. Wainwright

School of Computer Science
University of St Andrews

March 19, 2002

I, Robert J. Wainwright, hereby certify that this thesis, which is approximately 76000 words in length, has been written by me, that it is the record of work carried out by me, and that it has not been submitted in any previous application for a higher degree.

date 05/04/02 *signature of candidate* _____

I was admitted as a research student in October 1996 and as a candidate for the degree of Doctor of Philosophy in October 1997; the higher study for which this is a record was carried out in the University of St Andrews between 1996 and 1999.

date 05/04/02 *signature of candidate* _____

I hereby certify that the candidate has fulfilled the conditions of the Resolution and Regulations appropriate for the degree of Doctor of Philosophy in the University of St Andrews and that the candidate is qualified to submit this thesis in application for that degree.

date 05/04/02 *signature of supervisor* _____

In submitting this thesis to the University of St Andrews I understand that I am giving permission for it to be made available for use in accordance with the regulations of the University Library for the time being in force, subject to any copyright vested in the work not being affected thereby. I also understand that the title and abstract will be published, and that a copy of the work may be made and supplied to any bona fide library or research worker.

date 05/04/02 *signature of candidate* _____

Abstract

Given an integer matrix A, there is a unique matrix S of a particular form, called the Smith Normal Form, and non-unique unimodular matrices P and Q such that $PAQ = S$.

It is often the case that these matrices P and Q will be used for further calculation, and as such it is desirable to find P and Q with small entries. In this thesis we address the problem of finding such P and Q with small entries, in particular in the case where A is a diagonal matrix, which arises as a final step in many published algorithms.

Heuristic algorithms are developed which appear to do well in practice and some theory is developed to explain this behaviour.

We also give an account of the implementation of an alternative algorithm which bypasses this intermediary diagonal form. The basic theoretical development of this is work by Storjohan.

Acknowledgements

I would especially like to thank my supervisor, Dr. Steve Linton, for his invaluable advice and encouragement, and for providing an inspiring research environment.

I would also like to thank my parents for their support and encouragement, and my lovely wife for understanding.

A final special thanks to my office mate, Andrew Cutting, who supported my chess and coffee habits, and who helped me retain my sanity as a counterbalance to his own particular brand of lunacy.

Contents

1 **Introduction** 1

2 **Decomposition and Examples** 5

2.1 A Worked Example 6

3 **Quality of Solution** 10

3.1 Quality of a Single Matrix 10

3.2 Quality of a Pair of Matrices 12

3.3 Bounds for Some Measures 14

3.3.1 $\mathcal{Q}_{[1,1,\times]}(P,Q)$ 14

3.3.2 $\mathcal{Q}_{[x,x,+]}(P,Q)$ 16

3.3.3 $\mathcal{Q}_{[x,x,Max]}$ and $\mathcal{Q}_{[\infty,\infty,Max]}$ 17

3.3.4 Relationships Between $\mathcal{Q}_{[x,x,]}$ and $\mathcal{Q}_{[y,y,]}$ 18

3.4 Conclusions 18

4 **Directed Graphs** 19

4.1 Basic Concept 19

4.1.1 Definitions 19

4.1.2 Choices 22

4.2 Properties 23

4.2.1 $\Gamma_2(\mathrm{D})$ 23

4.2.2 $\Gamma_3(\mathrm{D})$ 24

4.2.3 $\Gamma_x(\mathrm{D})$ 24

4.3 Some Number and Graph theory 25

4.3.1 Gcd Free Basis 25

4.3.2 Isomorphism of Graphs 28

4.4 Input Shape 29

4.4.1 A Worst Case Input 30
4.4.2 State Space of Inputs 31
4.5 Coprime Entries 35
4.6 Conclusions 36

5 The 2 x 2 Problem 37
5.1 The Extended Euclidean Algorithm 38
5.2 Basic Procedure 41
5.3 Optimizing the Multiplier Matrices 43
5.3.1 Brute Force 43
5.3.2 More Intelligent 43
5.3.3 Factorization Based Methods 44
5.4 As an Isolated Problem 44
5.4.1 $\mathcal{Q}_{[2,2,+]}$ 45
5.4.2 $\mathcal{Q}_{[2,2,\times]}$ 45
5.5 As an Intermediary Step 46
5.5.1 $\mathcal{Q}_{|+|}$ 48
5.5.2 $\mathcal{Q}_{|\times|}$ 49
5.6 Conclusions 49

6 Strategies 50
6.1 Testing and Comparing Algorithms 53
6.1.1 Permutation of Input 57
6.2 Positional Heuristics 58
6.2.1 The Standard Algorithm 59
6.2.2 Divide and Conquer 64
6.3 Power Growth 69
6.3.1 Maximum Powers for Particular Heuristics 71
6.3.2 Random 79
6.3.3 Relationship Between Theory and Practice 82
6.4 Comparisons of Positional Algorithms 85
6.5 Structural Heuristics 93
6.5.1 Overview of the Structural Algorithm 94
6.5.2 Random 95
6.5.3 MinGcd 95

6.5.4 MaxLcm 96
6.5.5 Edge Creation in $\Gamma_2(D)$ 96
6.5.6 Height / Weight 96
6.5.7 Proximity 98
6.5.8 Smallest / Largest Pair 98
6.5.9 RowColSize 101
6.5.10 Full Lookahead 101
6.5.11 Structural Conclusions 103
6.6 Overall Strategy Conclusions 103

7 Bypassing the Diagonal Form **105**
7.1 Overview of the Algorithm 106
7.1.1 split 106
7.1.2 rgcd 107
7.1.3 2×2 HNF 108
7.1.4 The modulo N extended gcd problem 109
7.1.5 SNF with Transforms 111
7.2 Performance 115

8 Conclusion and Further Notes **116**
8.1 Conclusions 116
8.2 Closing Notes and Further Work 117
8.2.1 Bounds revisited 117
8.2.2 Parallelisation 119
8.2.3 Improving the Solution 119
8.2.4 Average SNF 120

Appendix 1 **121**

Appendix 2 **128**

References **130**

Chapter 1

Introduction

A relatively common problem in computational linear algebra is to find a transformation of some input matrix into an "equivalent" but simpler canonical form. For input matrices with integer entries (or in fact entries from any Euclidean domain) one of the most useful forms is the Smith Normal Form (hereafter SNF). The existence and uniqueness of the SNF is one of the most important results in elementary matrix theory. Algorithms to reduce a matrix to SNF provide a constructive proof of the basis theorem for finitely generated abelian groups (see e.g. [HS79, HHR93, Sim94]).

Converting a matrix to SNF is usually achieved by a sequence of *elementary row / column operations*, namely :

- Negating a row / column (generally multiplying by an invertible element, but the only invertible elements of $\mathbb{Z}$ are ± 1).
- Adding a multiple of one row / column to another row / column.
- Swapping two rows / columns.

To each of these elementary row(column) operations there corresponds an *elementary matrix*, that is an invertible non-singular integer matrix. An elementary row(column) operation can be applied to a matrix by pre(post)-multiplication by the corresponding elementary matrix.

By application of **either** elementary row or elementary column transformations any matrix over $\mathbb{Z}$ can be reduced to a triangular form. Further operations then permit the reduction of the off-diagonal entries modulo the diagonal entry in each column(row). If row operations

are used to produce an upper triangular matrix with this property we call this the Hermite Normal Form (HNF). Hermite first proved the existence of the HNF in 1851 [Her51]. By application of **both** elementary row and column operations any matrix over $\mathbb{Z}$ can be reduced to a diagonal form. Further application of elementary row and column transformations allow the entries along the diagonal to be adjusted such that each entry divides the next. A matrix in this form is said to be in Smith Normal Form. Smith gave a construction for the SNF diagonalization in a paper of 1861 [Smi61].

Two matrices A and B are are said to be *equivalent* if there exist unimodular matrices P and Q such that $PAQ = B$.

Definition 1.1. *An integer matrix A is in Smith Normal Form if for some $r \geq 0$ the entries $s_i = A_{ii}, 1 \leq i \leq r$, are non-negative, A has no other nonzero entries and s_i divides $s_{i+1}, 1 \leq i \leq r$.*

$$\begin{bmatrix} s_1 & 0 & \cdots & 0 \\ 0 & s_2 & & \\ \vdots & & \ddots & \\ 0 & & & s_n \end{bmatrix}.$$

Smith's paper effectively showed :

- Every integer matrix A is equivalent over $\mathbb{Z}$ to a unique matrix S in Smith Normal Form.
- There exist unimodular matrices P and Q such that $PAQ = S$.

There has been much previous work on developing good strategies for the calculation of both the HNF and SNF of matrices over Principal Ideal Domains. This work has been mainly aimed at combatting "intermediate expression swell", the problem of having to deal with exceedingly large values during the calculation rather than reducing any measure of the final answer. In the case of the HNF, for non-singular matrices at least, intermediate expression swell is the main issue as given $UA = H$ where A is non-singular and H is in HNF, U is uniquely determined as HA^{-1}. In the case of the SNF however, where $UAV = S$, the matrices U and V are not determined uniquely and different algorithms may produce matrices U and V with larger or smaller entries. It turns out that the methods which aim to keep intermediate entries in the work matrix small also tend to keep the final transformation matrices relatively small as only small multiples of rows or columns

need be added to others. Algorithms for computing the SNF generally proceed by finding U_*, V_* such that U_*AV_* is diagonal but may not have the divisibility property required for SNF. The final step in finding the SNF is then to perform row and column operations to repeatedly replace pairs with their greatest common divisor (gcd) and lowest common multiple (lcm). In fact the consequences of this final step upon the multiplier matrices can be quite severe and the investigation of this problem will form the main subject of this thesis.

An important situation arises from the natural correspondence between $\mathbb{Z}-$modules and abelian groups. The fundamental theorem of finitely generated abelian groups classifies all such groups by giving a canonical decomposition. One version of the theorem states that :

Theorem 1.1. *Let G be a finitely-generated abelian group. Then G has a direct decomposition*

$$G = G_1 \oplus \ldots \oplus G_r \oplus G_{r+1} \oplus \ldots \oplus G_{r+f}$$

where:

1) G_i is a nontrivial finite cyclic group of order l_i for $i = 1, \ldots, r$;

2) G_i is an infinite cyclic group for $i = r+1, \ldots, r+f$;

3) $l_1 | l_2 | \ldots | l_r$.

The integers f and $l_1, \ldots, l_r$ occurring in such a decomposition are uniquely determined. A finitely presented abelian group G, written additively, may be given as a set of n generators $x_1, \ldots, x_n$ and m relations of the form $\sum_{j=1}^{n} a_{i,j}x_j = 0$. Such presentations arise from a variety of natural computations, for example from computation of subgroup presentations by the Reidemeister-Schreier process. We associate with G its relation matrix, the $m \times n$ integer matrix A. Performing row or column operations upon A correspond to Tietze transformations (see [Tie08] or [Joh90] for details) of the group presentation, leaving the associated group unchanged up to isomorphism, so that abelian groups with equivalent relation matrices are isomorphic. Column operations correspond to operations on the group generators, row operations to the relations. The SNF gives us the direct decomposition. The column transformation matrix and its inverse provide us with a way of writing the generators of the original group in terms of the generators of the group defined

by using the SNF as the relation matrix and vice versa. The row transformation matrix allows us to rewrite the relations and effectively provides a proof of correctness. It is valuable to obtain matrices with small entries for use in further calculations.

The algorithms that are described in this thesis have been developed and implemented in the GAP computational algebra system, see [S+95] . GAP (Groups, Algorithms and Programming) is a system for computational discrete algebra with particular emphasis on, but not restricted to computational group theory. GAP was developed at Lehrstuhl D für Mathematik (LDFM), RWTH Aachen, Germany from 1986 to 1997. After the retirement of J. Neubüser from the chair of LDFM, the development and maintenance of GAP has been coordinated by the Schools of Mathematics and Computer Science at the University of St Andrews, Scotland. List manipulation and large integer and rational arithmetic are all built into the language.

Chapter 2

Decomposition and Examples

This chapter will provide an overview of the main problem, and an explanation of how this problem is then broken down into several parts, each of which will be the subject of a later chapter.

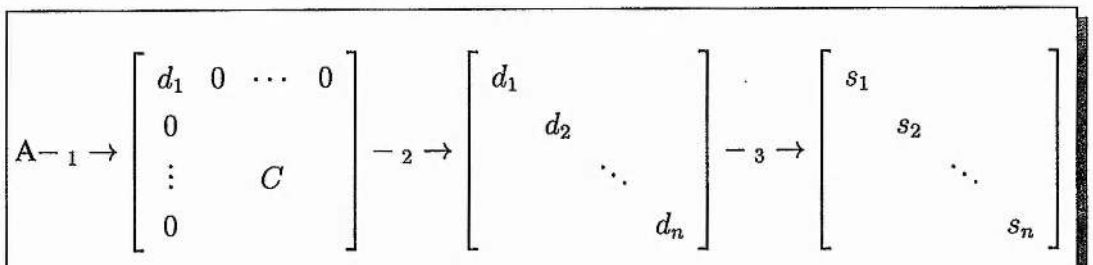


Figure 2. The basic procedure for finding the SNF of a matrix, A.

Figure 2 shows the general method by which the SNF of a matrix is usually computed, each step being achieved by some sequence of elementary operations. First the gcd of a row and column is obtained. This element is then used to zero the rest of the entries in that row and column. This appears as step 1 in the figure. This process is then applied repeatedly to the submatrix remaining until a diagonal form is reached (step 2). The divisibility requirement of the SNF is then finally tackled (step 3). Each of these steps is achieved by some sequence of

- premultiplying by unimodular matrices (row operations) and
- postmultiplying by unimodular matrices (column operations).

The final step, step 3 in figure 2, from a general diagonal matrix D to one in Smith Normal Form S is the step we will investigate further in the main part of this thesis. Much work has been done on the diagonalization problem (steps 1 and 2), see e.g. [HM97] for work in this area. The problem of converting a diagonal form to SNF, and especially of finding small multiplier matrices, has been little studied however.

2.1 A Worked Example

Consider the matrix

$$A = \begin{bmatrix} 42 & 24 & 18 & 12 & 78 \\ 177 & 737 & -238 & 71 & 491 \\ 294 & 168 & 256 & 84 & 676 \\ 639 & 692 & 90 & 203 & 1248 \\ 1260 & 951 & 930 & 360 & 2961 \end{bmatrix}.$$

Using a norm driven diagonalization procedure [HM97] we find :

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 13 & 1 & 4 & -1 & -1 \\ -7 & 0 & 1 & 0 & 0 \\ -5 & 0 & 0 & 1 & 0 \\ -9 & 0 & -3 & 0 & 1 \end{bmatrix} \times A \times \begin{bmatrix} 1 & 2 & -3 & -2 & 2 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & -1 \\ -3 & -4 & 9 & 7 & -12 \\ 0 & -1 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 6 & 0 & 0 & 0 & 0 \\ 0 & 78 & 0 & 0 & 0 \\ 0 & 0 & 130 & 0 & 0 \\ 0 & 0 & 0 & 143 & 0 \\ 0 & 0 & 0 & 0 & 231 \end{bmatrix}. \quad (2.1)$$

Now we have a diagonal matrix, D, we calculate the SNF by, for example, repeating the following basic step on selected 2x2 submatrices $\begin{bmatrix} a & 0 \\ 0 & b \end{bmatrix}$:

- Perform a single column operation to obtain
$$\begin{bmatrix} a & 0 \\ b & b \end{bmatrix}.$$
- Followed by row operations corresponding to the steps of the euclidean algorithm to obtain
$$\begin{bmatrix} gcd(a,b) & bt \\ 0 & lcm(a,b) \end{bmatrix}.$$
- And finally one more column operation to zero out the upper corner entry and leave the SNF,
$$\begin{bmatrix} gcd(a,b) & 0 \\ 0 & lcm(a,b) \end{bmatrix}.$$

So at each 2x2 step we can replace a pair of entries from the diagonal by the gcd and the lcm thereof. By repeating this step we will eventually produce the SNF (termination of this process is proved in chapter 4).

A standard approach is to take pairs of positions in the order (reading left to right, top to bottom):

$$\begin{array}{cccccc} [1,2] & [1,3] & & \dots & [1,n] & (s_1) \\ & [2,3] & [2,4] & \dots & [2,n] & (s_2) \\ & & \ddots & & \vdots & \vdots \\ & & & & [n-1,n] & (s_{n-1}) \end{array}$$

The subsequence (s_1) guarantees that we obtain the gcd of $d_1, \dots, d_n$ in position 1. And similarly the $n-2$ steps of (s_2) obtain the gcd of $d_2, \dots, d_n$ in position 2. This continues until finally with the subsequence (s_{n-1}) consisting of a single pair of positions we must definitely obtain the last two entries of the SNF.

Applying this naive procedure to the above diagonal matrix, D we find transformation matrices, U and V such that we obtain the Smith normal form, S.

$$UDV = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 78 & 0 & 0 \\ 0 & 0 & 0 & 858 & 0 \\ 0 & 0 & 0 & 0 & 30030 \end{bmatrix}$$

where

$$U = \begin{bmatrix} -1562 & 0 & -71 & 1 & 0 \\ -251680 & 320 & -11440 & 160 & 1 \\ -9568 & 11 & -435 & 6 & 0 \\ -6684249 & 8481 & -303831 & 4248 & 26 \\ -33373340 & 42350 & -1516977 & 21210 & 130 \end{bmatrix}, V = \begin{bmatrix} 1 & 143 & -1781 & 83941 & -620620 \\ 0 & 1 & -11 & 572 & -4235 \\ -1 & -143 & 1782 & -83952 & 620697 \\ 1 & 142 & -1704 & 82644 & -611310 \\ 0 & -9 & 0 & -4160 & 31200 \end{bmatrix}.$$

Thus, combining these with the multipliers shown in Equation 2.1, we obtain overall multipliers

$$XAY = S$$

where

$$X = \begin{bmatrix} -1070 & 0 & -71 & 1 & 0 \\ -168249 & 320 & -10163 & -160 & -319 \\ -6410 & 11 & -391 & -5 & -11 \\ -4468653 & 8481 & -269985 & -4233 & -8455 \\ -22311171 & 42350 & -1347967 & -21140 & -42220 \end{bmatrix},$$

and

$$Y = \begin{bmatrix} 2 & 272 & -3741 & 163333 & -1206161 \\ 0 & 1 & -11 & 572 & -4235 \\ -1 & -133 & 1771 & -79220 & 585262 \\ -5 & -618 & 9497 & -381251 & 2811503 \\ 0 & -10 & 11 & -4732 & 35435 \end{bmatrix}.$$

These multiplier matrices have much larger entries than either the original matrix or the SNF. The magnitude of the largest entry is of the order of 1000 times the magnitude of the largest entry in the SNF itself, S_{nn}, or roughly of the order of S_{nn}^2. If we are a little more careful when selecting the order of pairs upon which to perform the basic gcd-lcm step then the calculation produces much better multipliers. Performing the gcd-lcm step on the sequence of pairs $[[1,4],[3,5]]$ we obtain

$$U_2 D V_2 = S$$

where

$$U_2 = \begin{bmatrix} 24 & 0 & 0 & 1 & 0 \\ 0 & 0 & 16 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 143 & 0 & 0 & 6 & 0 \\ 0 & 0 & 2079 & 0 & 130 \end{bmatrix}, \text{ and } V_2 = \begin{bmatrix} 1 & 0 & 0 & -143 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & -231 \\ -1 & 0 & 0 & 144 & 0 \\ 0 & -9 & 0 & 0 & 2080 \end{bmatrix}.$$

So the overall multipliers we obtain using this diagonal to SNF transformation would be

$$X_2 A Y_2 = S$$

where

$$X_2 = \begin{bmatrix} 19 & 0 & 0 & 1 & 0 \\ -121 & 0 & 13 & 0 & 1 \\ 13 & 1 & 4 & -1 & -1 \\ 113 & 0 & 0 & 6 & 0 \\ -15723 & 0 & 1689 & 0 & 130 \end{bmatrix}, \text{ and } Y_2 = \begin{bmatrix} 3 & -21 & 2 & -431 & 4853 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 10 & 1 & 0 & -2311 \\ -10 & 117 & -4 & 1437 & -27039 \\ 0 & -9 & -1 & 0 & 2080 \end{bmatrix}$$

which appears to be a 'better' solution as it is sparser and contains smaller numbers. Notably we can see that the magnitude of the largest entry in either of the multiplier matrices is actually less than S_{nn}. We can assign other measures of quality to each of these solutions by for example, examining the sum of the squares of all the elements of the transformation matrices. Then we have that

$$Quality(X, Y) \approx 2^{48}$$

and

$$Quality(X_2, Y_2) \approx 2^{29}.$$

If we use the method described in Chapter 7, implemented in GAP4, which computes the SNF directly without the intermediate diagonal form then we produce the following :

$$\begin{bmatrix} 54090 & 1695 & -3799 & -3868 & 807 \\ 48702 & 1529 & -3387 & -3482 & 718 \\ 38250 & 1199 & -2680 & -2735 & 569 \\ 15562 & 484 & -1139 & -1114 & 244 \\ 36723 & 1155 & -2529 & -2625 & 535 \end{bmatrix} \times A \times \begin{bmatrix} 1 & 0 & -56 & -525 & -13988 \\ 1 & 1 & -78 & -781 & -41195 \\ 0 & 1 & -21 & -265 & -27474 \\ 0 & 0 & 0 & 1 & -16 \\ 0 & 0 & 0 & 1 & -15 \end{bmatrix} = S.$$

Here these multiplier matrices though not particularly sparse are quite well balanced and the largest entry in the multiplier matrices is of magnitude only roughly twice S_{nn}. The sum of the squares of all the elements of the transforming matrices is $\approx 2^{33}$. It appears probable that we can compete favourably with such 'combined techniques' and gain noticeable improvements over naive methods by making some intelligent choices but still using the simple pairwise techniques.

Three main points arising from this example will form the subjects of later chapters.

- How to measure the quality of the solution?
- How best to perform each 2×2 step?
- How to select pair order?

Chapter 3

Quality of Solution

We are looking for 'good' or 'small' solutions to the problem of finding unimodular integer matrices P and Q given a diagonal integer matrix D such that $PDQ = S$ is the Smith Normal Form of D. We need to define what we actually mean by a 'good' or 'small' solution. In this chapter we shall describe various ways to measure the quality of a particular solution to the above problem and in doing so we will provide a range of methods to compare in a quantitative fashion two such solutions and decide which is the 'better', or 'smaller' solution.

3.1 Quality of a Single Matrix

Our metrics for solutions P, Q to the problem $PDQ = S$ will combine measures for two individual matrices. Note that a transforming matrix T is a square, non-singular (unimodular in fact) matrix over $\mathbb{Z}$ - our definitions will reflect this fact and could be readily extended to more general matrices over any euclidean domain, however we do not do this here. One natural measure with useful invariance properties is the determinant but as the matrices we are interested in are unimodular the determinant will always be ± 1. For the applications for which these matrices are useful we are primarily interested in the absolute magnitudes of the entries T_{ij} of T and so this suggests we need to examine some function on these elements T_{ij}. The most obvious option is a sum of powers of the absolute values of the entries of T. Since we will only be comparing different solutions of $n \times n$ problems for fixed n we can also safely introduce any form of scaling depending only on n. We shall for the moment examine the average (absolute) magnitude of the sum of some power

of the entries which will help simplify formulae later in this chapter. This scaling appears as a simple multiplier of $\frac{1}{n^2}$ to the sum. We have :

Definition 3.1. *Let T be an $n \times n$ integer matrix and τ be a non-negative real number. We define a function $\|\cdot\|_\tau$ from T to $\mathbb{R}^{\geq 0}$ by*

$$\|T\|_\tau := \frac{1}{n^2} \sum_{i=1}^{n} \sum_{j=1}^{n} |T_{ij}|^\tau.$$

Note that we are abusing standard notation here as this function is not a matrix norm. The only requirement of a matrix norm it fails to meet, however, is the linear scalability requirement which is irrelevant here since we are dealing with unimodular matrices and multiplying them by any non-unit would mean they were no longer unimodular. We will extend (and abuse) this notation slightly as we are quite often interested in the largest entry appearing in a matrix :

Definition 3.2. *We define $\|\cdot\|_\infty$ from T to $\mathbb{R}^{\geq 0}$ to be*

$$\|T\|_\infty := \textit{Magnitude of largest entry of} T.$$

Note that we are choosing not to scale $\|\cdot\|_\infty$. Note also that the function $\|\cdot\|_0$ is simply the number of non-zero entries of T divided by n^2. And finally note that the above definitions imply that $\|\cdot\|_\alpha$ will be greater than $\frac{1}{n}$ for the problem in which we are interested, as any unimodular matrix must have at least one non-zero entry in each row.

An immediate question is then what is a sensible value of τ to select if we wish to use $\|\cdot\|_\tau$ as a measure for comparing two matrices in the context in which we are interested i.e. as multiplier matrices. By using $\|\cdot\|_\infty$ we would be ignoring a lot of the potential information which we might prefer to utilise. For example this would give preference to a much denser matrix over a very sparse matrix with only slightly larger entries. At the other end of the scale $\|\cdot\|_0$ is simply a measure of sparsity and is not of any use for selecting the better of two solutions with respect to the size of the entries of the matrix. $\|\cdot\|_1$ is the sum of the absolute magnitudes of the entries and if we consider two $n \times n$ matrices X, Y such that X has n^2 entries, each of size k, and Y has $n^2 - 1$ zero entries and a single entry of size n^2k we see that $\|X\|_1 = \|Y\|_1 = n^2k$. So the ratio of the largest entries of two solutions with the same size could be of the order of n^2. We note that applying a similar argument shows that under $\|\cdot\|_2$, the ratio of the largest entries in the matrices is now at worst linear in the number of rows of the matrix. For our purposes this seems sensible and so $\|\cdot\|_2$ will be the measure upon which we will mostly concentrate in this thesis.

Note that if we have two $n \times n$ matrices X, Y such that $\|X\|_\tau = \|Y\|_\tau$ for some τ and we wish to select between them, we can do so by examining the measures $\|X\|_{\tau+1}, \|Y\|_{\tau+1}$. It should be clear that this will differentiate between the matrices according a smaller value for the measure on the matrix which is better balanced i.e. having entries of a more similar size.

We will also make a couple of useful simple points,

Lemma 3.1. *Pre or post multiplication by a permutation matrix does not change the size of a matrix.*

Proof: The size of a matrix is defined to be a sum of powers. Note that we can take that sum in any order. Hence swapping rows and/or columns has no effect on size. ∎

Lemma 3.2. *Transposition does not change the size of a matrix.*

Proof: As for lemma 3.1. ∎

3.2 Quality of a Pair of Matrices

As already mentioned we generally wish to consider the pair, $\{P, Q\}$ together when assessing the quality of a solution $PDQ = S$. We shall consider various functions $\sigma : \mathbb{R}^2 \to \mathbb{R}$ for combining two matrix norms. All of these functions will satisfy the weak monotonicity property,

$$y > z \quad \Rightarrow \quad \sigma(x, y) \geq \sigma(x, z) \wedge \sigma(y, x) \geq \sigma(z, x).$$

We will write $\|x, y\|_\sigma$ for $\sigma(x, y)$. Further, we will allow binary operators such as $+$ and $\times$ in the place of σ.

Definition 3.3. *Given $PAQ = S$ where P and Q are unimodular matrices the quality, $\mathcal{Q}_{[\alpha,\beta,\sigma]}$, of this solution is*

$$\mathcal{Q}_{[\alpha,\beta,\sigma]}(P, Q) := \Big\| \|P\|_\alpha, \|Q\|_\beta \Big\|_\sigma,$$

where the triple $[\alpha, \beta, \sigma]$ denotes the respective metrics being used for P, Q and the combination function σ.

Some obvious candidates for the combination function σ for our purposes are Minimum (Min), Maximum (Max), $+$, or $\times$. Many combination functions such as "root mean square" are actually redundant as $\mathcal{Q}_{[\alpha,\beta,rms]} = \sqrt{\mathcal{Q}_{[2\alpha,2\beta,+]}}$ and so no new comparisons are revealed.

Using Min as the combination function suffers from the same potential pitfalls as $\|\cdot\|_\infty$, i.e. we are ignoring information, albeit to a much lesser extent. In particular we need to be very careful if $\alpha \neq \beta$, otherwise incredibly disparate solutions may well have similar qualities, which does not really seem sensible. However in some circumstances we are only interested in minimizing one of the multiplier matrices. In this case the measure $\mathcal{Q}_{[\tau,\tau,Min]}$ appears to be quite a good choice, as it does not actually matter which of the multiplier matrices we minimize since

$$PDQ = S = S^T = (PDQ)^T = Q^T D P^T \tag{3.1}$$

and transposition has no effect on the size of the matrix (lemma 3.2).

Using Max as the combination function suffers from similar problems, though to an even lesser extent. We will generally consider either $+$ or $\times$ as our combination function.

So for example we denote by $\mathcal{Q}_{[2,2,+]}$ the quality metric

$$\frac{1}{n^2}\left(\sum_{i=1}^{n}\sum_{j=1}^{n}|P_{ij}|^2 + \sum_{i=1}^{n}\sum_{j=1}^{n}|Q_{ij}|^2\right). \tag{3.2}$$

and we denote by $\mathcal{Q}_{[2,2,\times]}$ the quality metric

$$\frac{1}{n^4}\left(\sum_{i=1}^{n}\sum_{j=1}^{n}|P_{ij}|^2\right) \times \left(\sum_{i=1}^{n}\sum_{j=1}^{n}|Q_{ij}|^2\right). \tag{3.3}$$

Note that though we have no idea of the exact relationship between the elements in the multiplier matrices that form solutions to our overall problem we can get an idea of what solutions with small metrics should look like by considering the possible distributions of entries that may occur in pairs of matrices with a given quality. For example we can look at minimizing the largest entry occuring in either matrix.

Under $\mathcal{Q}_{[2,2,+]}$, it is clear that if we are allowed to distribute the entries as we choose then assuming an even distribution in each of the matrices leads to the matrices being well balanced as well. That is to say the entries in both matrices will be of the same size.

Under $\mathcal{Q}_{[2,2,*]}$, it is interesting to note that if the matrices P, Q have equal sized entries of magnitude X say then $\mathcal{Q}_{[2,2,*]}(P,Q) = X^2$. However we get the same result if the entries in P are all of magnitude kX and all the entries in Q are of magnitude $\frac{X}{k}$.

So it appears that $\mathcal{Q}_{[2,2,+]}$ is a good metric to use if we wish to find small well balanced multiplier matrices. If we are not too worried about the balance between the matrices then we may find that $\mathcal{Q}_{[2,2,*]}$ is a good choice that allows a greater variety of solutions of a particular quality.

We have defined here measures which we can use to analyze the behaviour of various algorithms. We will now discuss briefly some bounds on these measures.

3.3 Bounds for Some Measures

The main aim of this thesis is practical rather than analytical, but there are some interesting results to be stated and connections between various norms that illustrate the complexity of the global minimization problem at hand. We will calculate some lower bounds but it should be noted that it is not meaningful to consider upper bounds for the general problem as given A and S there will exist arbitrarily large U and V such that $UAV = S$. We will return to this in chapter 8 and prove some upper bounds under certain constraints on various parts of the problem.

3.3.1 $\mathcal{Q}_{[1,1,\times]}(P,Q)$

It is possible to derive a lower bound for the metric $\mathcal{Q}_{[1,1,\times]}$ immediately from the description of the basic problem. Given $PDQ = S$ where D is a 'sorted' diagonal matrix, i.e. $D_{11} \leq D_{22} \leq \ldots \leq D_{nn}$, and P and Q are unimodular multiplier matrices such that S is the Smith Normal Form of D we have that,

$$\mathcal{Q}_{[1,1,\times]}(P,Q) = \frac{1}{n^4}\left(\sum_{i=1}^{n}\sum_{j=i}^{n}|P_{ij}|\right) \times \left(\sum_{i=1}^{n}\sum_{j=i}^{n}|Q_{ij}|\right). \tag{3.4}$$

Now,

$$S_{kk} = \sum_{i=1}^{n} P_{ki} D_{ii} Q_{ik} \tag{3.5}$$

$$|S_{kk}| \leq \sum_{i=1}^{n} |P_{ki} D_{ii} Q_{ik}| \tag{3.6}$$

$$\leq \sum_{i=1}^{n} |P_{ki} D_{nn} Q_{ik}| \tag{3.7}$$

$$\left|\frac{S_{kk}}{D_{nn}}\right| \leq \sum_{i=1}^{n} |P_{ki}||Q_{ik}| \tag{3.8}$$

$$\leq \sum_{i=1}^{n} |P_{ki}| \sum_{i=1}^{n} |Q_{ik}| \tag{3.9}$$

$$\frac{1}{|D_{nn}|} \sum_{k=1}^{n} |S_{kk}| \leq \sum_{k=1}^{n} \left(\sum_{i=1}^{n} |P_{ki}| \sum_{i=1}^{n} |Q_{ik}| \right) \tag{3.10}$$

$$\leq \sum_{i=1}^{n} \sum_{k=1}^{n} |P_{ki}| \sum_{i=1}^{n} \sum_{k=1}^{n} |Q_{ik}|. \tag{3.11}$$

Giving,

$$\mathcal{Q}_{[1,1,\times]}(P,Q) \geq \frac{1}{|D_{nn}|\, n^4} \sum_{k=1}^{n} |S_{kk}|\,. \tag{3.12}$$

Remarks on the Derivation

The above bound is tight for a diagonal matrix that is already in SNF, when P, Q could be $n \times n$ identity matrices.

Note also that at two points, [3.9 and 3.11] in the above derivation, use is made of the following fact :

$$\sum_{i=1}^{n} (x_i y_i) \leq \left(\sum_{i=1}^{n} x_i \right) \left(\sum_{i=1}^{n} y_i \right).$$

for non-negative x_i, y_i.

In the 'average' case we actually have that

$$\sum_{i=1}^{n} (x_i y_i) \approx \frac{1}{n} \left(\sum_{i=1}^{n} x_i \right) \left(\sum_{i=1}^{n} y_i \right).$$

(To see this note that $\sum_{i=1}^{n} x_i$, where $x_i \in [0..m] = \frac{nm}{2}$ average and $\sum_{i=1}^{n} (x_i y_i)$, where $x_i, y_i \in [0, \ldots, m] = \frac{nm^2}{4}$ average. Let m tend to ∞. The result follows.)

Thus in 'random' situations we would expect the inequality

$$\mathcal{Q}_{[1,1,\times]}(P,Q) \geq \frac{1}{|D_{nn}|\, n^2} \sum_{k=1}^{n} |S_{kk}| \,.$$

to hold.

Additionally, step 3.6 in the derivation

$$\sum_{i=1}^{n} P_{ki} D_{ii} Q_{ik} \leq \sum_{i=1}^{n} |P_{ki} D_{ii} Q_{ik}|.$$

could hide a large amount of cancellation but unfortunately there appears to be no way of estimating this

On the other hand, step 3.7

$$\sum_{i=1}^{n} |P_{ki} D_{ii} Q_{ik}| \leq \sum_{i=1}^{n} |P_{ki} D_{nn} Q_{ik}|$$

will hide a factor of less than n, a worst case example being a diagonal with an unusually large entry which is coprime to all the other entries. Again it is difficult to anticipate what will occur in practise and how closely we could hope to approach the bound in general.

3.3.2 $\mathcal{Q}_{[x,x,+]}(P,Q)$

We can derive upper and lower bounds for $\mathcal{Q}_{[x,x,+]}(P,Q)$ in terms of $\mathcal{Q}_{[x,x,\times]}(P,Q)$.

Recall that

$$\mathcal{Q}_{[x,x,\times]}(P,Q) = \frac{1}{n^4} \left(\sum_{i=1}^{n} \sum_{j=i}^{n} |P_{ij}|^x \right) \times \left(\sum_{i=1}^{n} \sum_{j=i}^{n} |Q_{ij}|^x \right).$$

And

$$\mathcal{Q}_{[x,x,+]}(P,Q) = \frac{1}{n^2} \left(\sum_{i=1}^{n} \sum_{j=i}^{n} |P_{ij}|^x + \sum_{i=1}^{n} \sum_{j=i}^{n} |Q_{ij}|^x. \right)$$

Note that $\mathcal{Q}_{[x,x,+]} = R + S$, and $\mathcal{Q}_{[x,x,\times]} = R \times S$. If we assume that $\mathcal{Q}_{[x,x,\times]}$ is fixed then a simple differentiation reveals that $\mathcal{Q}_{[x,x,+]}$ is minimized when R and S are both equal to $\sqrt{\mathcal{Q}_{[x,x,\times]}}$. Hence

$$\mathcal{Q}_{[x,x,+]}(P,Q) \;\geq\; 2\sqrt{\mathcal{Q}_{[x,x,\times]}(P,Q)} \tag{3.13}$$

And similarly we can see from the symmetry and smoothness of the function that in order to maximise $\mathcal{Q}_{[x,x,+]} = R + S$ for a fixed $\mathcal{Q}_{[x,x,\times]} = R \times S$, we require one of R, S to be as small as possible. The smallest possible value we could have is $\frac{1}{n}$, the quality of an identity matrix. We then immediately have that

$$\mathcal{Q}_{[x,x,+]}(P,Q) \leq n\mathcal{Q}_{[x,x,\times]}(P,Q) + \frac{1}{n} \tag{3.14}$$

3.3.3 $\mathcal{Q}_{[x,x,Max]}$ and $\mathcal{Q}_{[\infty,\infty,Max]}$

We can quickly derive lower bounds for either the largest matrix or the largest entry occuring in either multiplier matrix in terms of either $\mathcal{Q}_{[x,x,+]}(P,Q)$ or $\mathcal{Q}_{[x,x,\times]}(P,Q)$.

If $\mathcal{Q}_{[x,x,+]}(P,Q) \geq L_+$ then it follows immediately that

$$\mathcal{Q}_{[x,x,Max]} \geq \frac{L_+}{2} \tag{3.15}$$

(by noting that the best we can do is split the L_+ evenly over the two matrices).

Similarly we can see that

$$\mathcal{Q}_{[\infty,\infty,Max]} \geq \sqrt[x]{\frac{L_+}{2}} \tag{3.16}$$

If $\mathcal{Q}_{[x,x,\times]}(P,Q) \geq L_\times$ then it again follows immediately that

$$\mathcal{Q}_{[x,x,Max]} \geq \sqrt{L_\times} \tag{3.17}$$

(by noting again that the best we can do is split the $L_\times$ equally).

And similarly we have

$$\mathcal{Q}_{[\infty,\infty,Max]} \geq \sqrt[2x]{L_\times}. \tag{3.18}$$

We can also derive a lower bound for $\mathcal{Q}_{[\infty,\infty,Max]}$ directly. We have

$$S_{nn} = \sum_{i=1}^{n} P_{ni} D_{ii} Q_{in}. \tag{3.19}$$

Hence there exists an m such that

$$P_{nm} D_{mm} Q_{mn} > \frac{S_{nn}}{n}, \tag{3.20}$$

and so

$$P_{nm} \text{ or } Q_{mn} > \sqrt{\frac{S_{nn}}{nD_{nn}}}. \tag{3.21}$$

3.3.4 Relationships Between $\mathcal{Q}_{[x,x,]}$ and $\mathcal{Q}_{[y,y,]}$

We will assume $y \geq x \geq 1$. There exist simple relationships between $\mathcal{Q}_{[x,x,\times]}(\tilde{P}, Q)$ and $\mathcal{Q}_{[y,y,\times]}(P, Q)$ and between $\mathcal{Q}_{[x,x,+]}(P, Q)$ and $\mathcal{Q}_{[y,y,+]}(P, Q)$.

Note first that, since we have products (or sums) of sums of powers of absolute magnitudes, we have

$$\mathcal{Q}_{[x,x,+]} \leq \mathcal{Q}_{[y,y,+]}.$$

$$\mathcal{Q}_{[x,x,\times]} \leq \mathcal{Q}_{[y,y,\times]}.$$

Now given a particular $\mathcal{Q}_{[x,x,+/\times]}$, we can see that $\mathcal{Q}_{[y,y,+/\times]}$ will be minimized when all entries are of similar size, X. Then we have that

$$\mathcal{Q}_{[x,x,\times]} = X^{2x}\ ,\ \mathcal{Q}_{[y,y,\times]} = X^{2y}$$

Hence

$$\mathcal{Q}_{[y,y,\times]} \ \geq \ \left(\mathcal{Q}_{[x,x,\times]}\right)^{\frac{y}{x}}. \tag{3.22}$$

and similarly,

$$\mathcal{Q}_{[x,x,+]} = 2X^{x}\ ,\ \mathcal{Q}_{[y,y,+]} = 2X^{y}$$

And so

$$\mathcal{Q}_{[y,y,+]} \ \geq \ \frac{\left(\mathcal{Q}_{[x,x,+]}\right)^{\frac{y}{x}}}{2^{\frac{y}{x}-1}}. \tag{3.23}$$

3.4 Conclusions

We have defined here measures by which we can assess the quality of a solution to the problem of finding unimodular matrices P, Q such that $PAQ = S$. We have provided lower bounds for some of these measures and also derived some relationships between them. We have seen that, apart from the "extreme" cases, the measures all broadly rise or fall together. Thus algorithms that perform well for one measure should perform reasonably well for other measures.

Chapter 4

Directed Graphs

The algorithms we are studying in chapters 4 to 6 proceed from an $n \times n$ diagonal integer matrix $D_0 = D$ to its SNF, S by repeatedly 'moving' subsections towards SNF. This is done by taking a (not necessarily contiguous) $k \times k$ diagonal submatrix of D and finding $k \times k$ multiplier matrices that transform that subsection into SNF. After η applications of this process we will produce unimodular integer matrices P_η, Q_η such that $P_\eta D Q_\eta = D_\eta$ and we will reach a point where $D_\eta = S$ for some η. We shall now formalise this process.

4.1 Basic Concept

4.1.1 Definitions

We first define the restriction of the problem to a submatrix.

Definition 4.1. *Let D be an $n \times n$ diagonal integer matrix and let I be the set $\{i_1, \ldots, i_k\}$, where $i_x \in [1, \ldots, n]$. We define $(D|_I)$ to be the $k \times k$ matrix M such that $M_{ab} = D_{i_a i_b}$.*

We now define the embedding of a matrix into a larger identity matrix.

Definition 4.2. *Let M be a $k \times k$ matrix. We define $\left({}^nM^I\right)$ to be the $n \times n$ matrix such that*

$$\left({}^nM^I\right)_{xy} = \begin{cases} M_{ab} & \text{if } x = i_a, y = i_b, \\ 1 & \text{if } x = y \notin I, \\ 0 & \text{otherwise.} \end{cases}$$

We will generally omit the ^{n} as it will be clear from the context. We are also going to need some notion of how 'close' a diagonal matrix is to SNF. For this we will use the notion of a *divisibility graph*.

Definition 4.3. *The **directed divisibility graph** of a length n list of integers $L = [L_1, \ldots, L_n]$ is the graph obtained by taking the entries $L_1, \ldots, L_n$ as vertices and adding a directed edge from L_i to L_j if $L_i | L_j$.*

Definition 4.4. *The **divisibility graph** of a length n list of integers $L = [L_1, \ldots, L_n]$ is the graph obtained by taking the directed divisibility graph, replacing all directed edges by undirected edges and ignoring multiple edges.*

Lemma 4.1. *The divisibility graph of the diagonal of an $n \times n$ matrix in SNF has $\binom{n}{2}$ edges.*

Proof: It is the complete graph. ■

This now allows us to formalise a notion of 'closeness' to SNF, and justify our basic 2×2 step.

Definition 4.5. *Given two lists, A and B whose divisibility graphs have E_A and E_B edges respectively, we say that A is closer to SNF than B if $E_A > E_B$.*

Proposition 4.1. *Given an $n \times n$ diagonal matrix, D, and a set of two elements, $I = \{x, y\}$, such that $D|_I$ is not in SNF then if P and Q are 2×2 unimodular multiplier matrices such that $PD|_IQ$ is in SNF then $P^I DQ^I$ is closer to SNF than D.*

Proof:

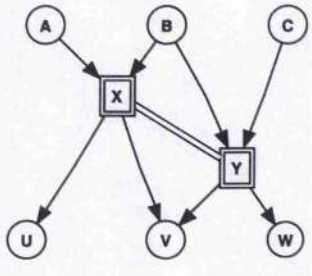


Figure 4.1 : Possible changes in divisibilites.

Consider the directed divisibility graph of the diagonal D, denoted by $ddg(D)$. For each x in $\{1, \ldots, n\}$ denote D_x by X. We will also define $X' = Gcd(X, Y)$ and $Y' = Lcm(X, Y)$.

Then we have

$$PD|_IQ = \begin{pmatrix} \text{X'} & 0 \\ 0 & \text{Y'} \end{pmatrix}.$$

The diagonal of D is

$$[1, 2, \ldots, N],$$

and the diagonal of $P^I DQ^I$ is

$$[1, \ldots, X-1, X', X+1, \ldots, Y-1, Y', Y+1, \ldots, N].$$

So $ddg(P^I DQ^I)$ differs from $ddg(D)$ at least in that there is an edge $x \to y$. Also other edges in the graph may differ. We will consider those edges in $ddg(D)$ which may be absent in $ddg(P^I DQ^I)$. We shall denote by $\overline{AB}$ the edge from A to B, $(D_a \to D_b)$.

The only types of edges which could be affected are those shown in Figure 4.1. Two of these, ($\overline{XU}$ and $\overline{CY}$), are easily seen to be unaffected by our operation as, if $X|U$ then clearly $X'|U$ and similarly, if $C|Y$ then $C|Y'$. There are then 4 cases to deal with:

Case 1 : There exists an entry A such that $\overline{AX}$ is an edge but $\overline{AY}$ is not an edge in $ddg(D)$. Then since $X|Y'$, there will be an edge $\overline{AY'}$ in $ddg(P^I DQ^I)$ and possibly an edge $\overline{AX}$. Specifically, there will be no further gain if $A \not| X'$ or a gain of one more edge if $A|X'$.

Case 2 : There exists an entry B such that both $\overline{BX}$ and $\overline{BY}$ are edges in $ddg(D)$. Clearly if B divides both X and Y then $X = jB$ and $Y = kB$, i.e. B divides both X' and Y'. We gain no further edges.

Case 3 : There exists an entry W such that $\overline{YW}$ is an edge but $\overline{XW}$ is not an edge in $ddg(D)$. Clearly X' divides Y and hence divides W. We have no further gain if $Y' \not| F$ or one more edge if $Y'|W$.

Case 4 : There exists an entry V such that both $\overline{XV}$ and $\overline{YV}$ are edges $ddg(D)$. Clearly X' divides Y and hence divides V. Also note that since $V = kX = jY$ then $V = \frac{Gcd(k,j)*XY}{Gcd(X,Y)}$ i.e. $V = Gcd(k,j) * Y'$. We gain no further edges.

In every case the number of edges in the graph increases. The list moves closer to SNF. ■

Corollary 4.1. *Given an $n \times n$ diagonal matrix, D and a set of k elements, $I = \{x_1, x_2, \ldots, x_k\}$, such that $D|_I$ is not in SNF, then if $PD|_I Q$ is in SNF then $P^I DQ^I$ is closer to SNF than D.*

Proof: A length k list may be put into SNF by repeated applications of 2×2 steps. Hence in terms of divisibilities gained, performing a single $k \times k$ step is equivalent to performing between 1 and $\binom{k}{2}$ steps on 2×2 subsections on that $k \times k$ set. ■

4.1.2 Choices

At step η of the overall calculation we have a diagonal matrix D_η. We select some set I_η of size k giving us $M = (D_\eta|_{I_\eta})$ and find $k \times k$ unimodular integer multiplier matrices U_η, V_η such that $U_\eta M V_\eta$ is a diagonal matrix that is in SNF. Note that $D_\eta = P_\eta D Q_\eta$ where

$$P_\eta = U_{\eta-1}^{I_{\eta-1}} \times U_{\eta-2}^{I_{\eta-2}} \times \ldots U_0^{I_0}.$$

and

$$Q_\eta = V_0^{I_0} \times V_1^{I_1} \times \ldots V_{\eta-1}^{I_{\eta-1}}.$$

So $D_{\eta+1} = (U_\eta^{I_\eta} P_\eta) \times D_\eta \times (Q_\eta V_\eta^{I_\eta})$ is a diagonal matrix which is closer to SNF than D_η.

It should be clear that since we make progress at each stage towards SNF we can write down an acyclic directed graph whose vertices are the various intermediary matrices and whose edges are associated with particular I. Note that these intermediary matrices are all diagonal. Note also that since by lemma 3.1 neither pre- or post-multiplication by a permutation matrix affects the size of a matrix we can,where convenient, assume each of these diagonal matrices D to be 'sorted' i.e. $D_{ii} \leq D_{jj}\ \forall i \leq j$.

Definition 4.6. *Let D be an $n \times n$ diagonal matrix. Then (D_{ii}) is the length n list consisting of the diagonal entries of D.*

Definition 4.7. *We define $\Gamma_x(D)$ to be the directed graph associated with (D_{ii}). The vertices of $\Gamma_x(D)$ are those length n lists reachable from (D_{ii}) by repeatedly replacing length x subsections with their SNF and then sorting. The edges of $\Gamma_x(D)$ correspond to sets of positions. We do not include self-edges.*

In further discussion we will use the square bracket notation $I = [I_1, I_2 \ldots I_k]$ to denote the ordered set of positions I. We will sometimes abuse this notation to refer also to the entries, d_{I_i} at those positions. We shall denote a path through the graph Γ_x(D) as a sequence of k such pairs $[[I_{1_1}, \ldots, I_{1_x}], [I_{2_1}, \ldots, I_{2_x}], \ldots, [I_{k_1}, \ldots, I_{k_x}]]$.

Note that we shall further systematically abuse notation and refer to $\Gamma_x((D_{ii}))$ when we mean Γ_x(D) as in the example shown in Figure 4.2.

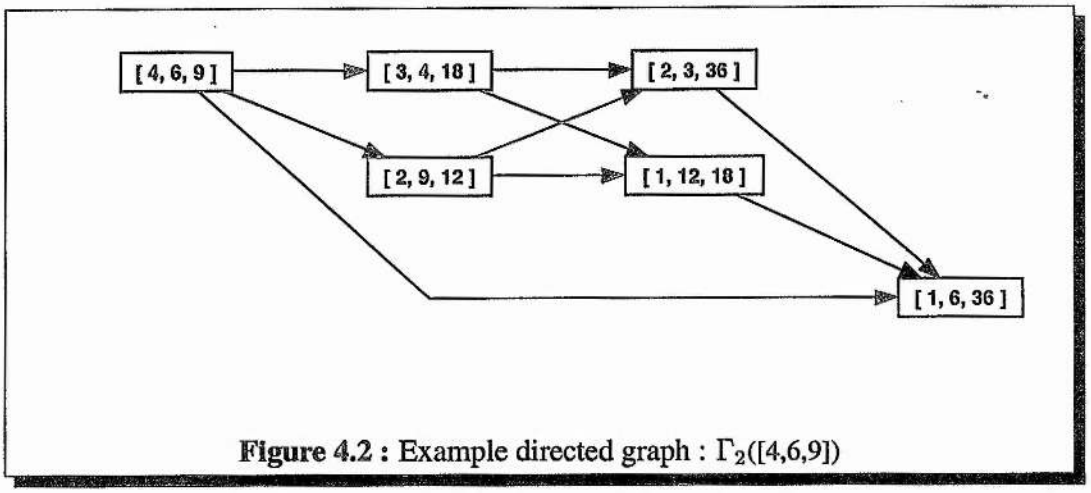


Figure 4.2 : Example directed graph : Γ_2([4,6,9])

4.2 Properties

We are interested primarily in $\Gamma_2(\mathrm{D})$ as we can find good multiplier matrices P and Q for the 2×2 problem (chapter 5). Note that any directed graph $\Gamma_x(\mathrm{D})$ has one source vertex,(D_{ii}), and one sink vertex,(S_{ii}).

Each heuristic we employ serves to traverse this graph from source to sink. As can be seen from example Figure 4.2 not all paths have the same length. Choice of path has a large effect on the (quality of) the transformation matrices obtained, and has some interaction with the best method of performing the SNF calculation asociated with each edge. This forms the subject of chapter 5. Since every step takes us closer to SNF the path length is clearly bounded.

4.2.1 $\Gamma_2(\mathrm{D})$

Theorem 4.1. *For $\Gamma_2(D)$ the path length can be no greater than $\binom{n}{2}$ for any graph, although this bound is attainable for many graphs.*

Proof: Follows directly from Lemma 4.1 and Proposition 4.1 ∎

4.2.2 Γ_3(D)

For Γ_3(D) we shall see that the path length can be no greater than $\binom{n-1}{2}$ for any graph, although this bound is attainable for many graphs. For example the following sequence, illustrated in Figure 4.3, will take this many operations.

- $[1,2,3],[1,2,4],\ldots,[1,2,n] = n-2$ operations to get positions 1 and 2 correct.
- $[1,3,4],[1,3,5],\ldots,[1,3,n] = n-3$ operations to get position 3 correct also.
- Continue in this vein obtaining one more correct position each run.

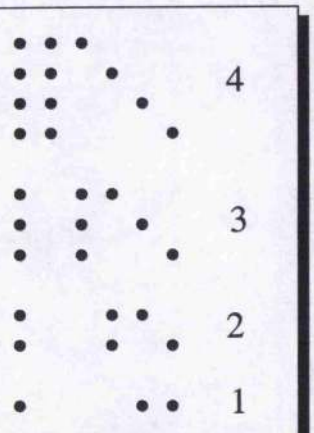


Figure 4.3: A naive algorithm on 6 points using 3x3 steps.

And it is easy to see that

$$\text{Total number of steps} = \sum_{i=1}^{n-2} i = \binom{n-1}{2}.$$

4.2.3 Γ_x(D)

Theorem 4.2. *Path length for $\Gamma_x(D)$ can be no greater than $\binom{n-x+2}{2}$.*

Proof: Starting with a length n list, L such that $ddg(L)$ is completely disconnected, i.e. as far from SNF as possible, the first $x \times x$ step will place those x elements into SNF, i.e. create $\binom{x}{2}$ edges. Each step that connects one of the $n-x$ disconnected vertices to the connected part will create at least $x-1$ edges. Once all vertices are joined by at least one edge each step will increase the number of edges in the graph by at least one, by proposition 4.1.

So the number of steps can be at most :

$$\binom{n}{2} - \binom{x}{2} - (n-x)(x-1) + n - x + 1 = \binom{n-x+2}{2}$$

■

4.3 Some Number and Graph theory

If we examine Γ_2[6,10,35], Γ_2[18,112,168], or even Γ_2[111,815,26233] we notice that these graphs are all isomorphic.

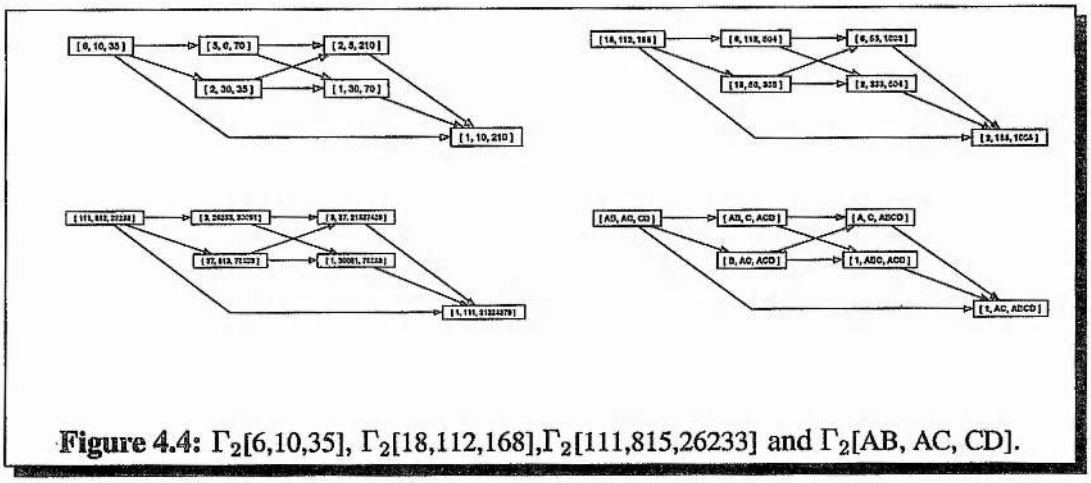


Figure 4.4: Γ_2[6,10,35], Γ_2[18,112,168],Γ_2[111,815,26233] and Γ_2[AB, AC, CD].

One immediate question is then what do these length 3 sets have in common? We can perform the 'translations' from any of these lists to any other by noting that all of these sets of numbers are of the form [AB,AC,CD] for some restricted choices of A,B,C,D and all have SNF [1,AC,ABCD].

$6 = 2 * 3$	$18 = 6 * 3$	$111 = 3 * 37$	$A * B.$
$10 = 2 * 5$	$168 = 6 * 28$	$813 = 3 * 271$	$A * C.$
$35 = 5 * 7$	$112 = 28 * 4$	$26233 = 37 * 709$	$C * D.$

It should be intuitively obvious that the shape of the above graphs is determined by the numbers A,B,C,D and the 'signature' [AB,AC,CD]. What restrictions must we place on the values of A,B,C,D to retain isomorphism? What can we say about the graph associated with an arbitrary set of numbers? Can we find some canonical representation of an arbitrary list of numbers? We shall now develop fuller theory to explain what is going on here and provide some answers to these questions.

4.3.1 Gcd Free Basis

We are interested in investigating how relatively coprime factors of various numbers 'move around' under each $k \times k$ step, or rather how the initial arrangement of relatively coprime

parts in the list of n numbers that is the diagonal affects the graph. The first thing to note is that any common integer multiplier can be ignored, or at least brought out of the representational shape since $Gcd(kx, ky) = k * Gcd(x, y)$.

Definition 4.8. *Let $A = (a_1, a_2, \ldots, a_m)$ be a nonempty list of m positive integers, not necessarily distinct. Let $B = (b_1, b_2, \ldots, b_n)$ be a set of integers, each ≥ 2. We say that B is a **gcd-free basis** for A if*

1. *$gcd(b_i, b_j) = 1 \quad \forall i \neq j$; and*
2. *there exist mn non-negative integers e_{ij} such that $a_i = \prod_{1 \leq j \leq n} b_j^{e_{ij}} \quad \forall i, 1 \leq i \leq m$.*

One immediate consequence is the following.

Theorem 4.3. *Let the set B be a gcd-free basis for $A = (a_1, a_2, \ldots, a_m)$. Then each element of A can be expressed uniquely as the product of non-negative powers of elements of B (up to order of the factors).*

A proof of this can be found in [BS96a].

Note however that this does not in any way define a 'minimal' gcd-free basis. We could for instance take as B the set containing the complete prime factorization of each element of A, or even add 'redundant' primes which don't divide any of the elements of A. However the algorithm developed in [BS96a] to compute a gcd-free basis does in fact efficiently compute such a minimal gcd-free basis.

Now we can consider the cases involving powers.

Lemma 4.2. *Let p, q, r, s be coprime and let $m, n \geq 0$ be integers. Then*

$$Gcd(p^m qr, p^m rs) = p^m r, \tag{4.1}$$

$$Gcd(p^{m+n} r, p^m rs) = p^m r. \tag{4.2}$$

And similarly,

$$Lcm(p^m qr, p^m rs) = p^m qrs, \tag{4.3}$$

$$Lcm(p^{m+n} r, p^m rs) = p^{m+n} rs. \tag{4.4}$$

The factor p^n and the factor q behave identically with respect to the Gcd and Lcm operations. In particular the p^m and p^n factors do not interact.

Thus we can effectively treat certain powers p^k as single factors, q. So we can rewrite our list in terms of linear factors. We can then utilise the concept of the gcd-free basis to find a canonical representation.

Given this machinery, it is possible to write down a 'signature' for any list of integers by using the following rewriting mechanism :

Signature Algorithm

- Remove any common factor, i.e. divide through by the gcd of the list.
- Replace any powers of elements with new coprime elements.
- Find a minimal gcd-free basis, and label each element of this GFB uniquely.
- Write each element of A in terms of (labels of) elements of the GFB.

Using this signature algorithm we can rewrite an arbitrary list D as another, simpler, list D_R. We will point out here that we can actually provide a much stronger rewriting. It should be obvious that we can associate with each element B_i of our gcd-free basis, B a term from another basis C. It should now be simple to see how we can perform the rewriting from $[4, 6, 9]$ to $[6, 10, 35]$, say - each list being $[AB, AC, CD]$. We also point out that by performing this further rewriting we can convert our potentially large integer problem over an arbitrary set of basis elements to another over a smaller set, e.g. the first k primes. This process provides us with a relatively simple method of checking similarity of two problems. To see this we will now define some concepts of 'shape'. If we assign a unique letter as a label to each element of the GFB and then we have

Definition 4.9. *The signature of a length n list, A, is the length n list $Sig(A)$ where each element is the string corresponding to rewriting the element of A as described in the above Signature Algorithm.*

We regard signatures differing only by a permutation of letters or positions to be equal. Determining this equality in general may be difficult, but the examples we shall use will be small enough to allow such observation. A partial solution is repeated sorting and application of the signature algorithm described above. It should be simple to see that this process will always converge to a stable solution and we can thus utilise this idea to check if two lists define the same digraph

Definition 4.10. *The height of a length n list, A, is the number of distinct letters in Sig(A).*

Definition 4.11. *The weight of a length n list, A, is the total number of letters in Sig(A).*

Note that we have some restrictions on $height(A)$ and $weight(A)$. It is obvious that the minimum value for both is zero i.e. $A = \{1, 1, \ldots, 1, 1\}$, although this is not really a valid input for our purposes. We will explain further the constraints on height and weight in subsection 4.4.2

4.3.2 Isomorphism of Graphs

Theorem 4.4. *The signatures define isomorphism classes of graphs (up to permutation of letters).*

Proof: If two lists have the same signature, they clearly have the same graph associated with them. ∎

Note that we have not precluded two lists with differing signatures having the same graph associated with them. This is for the simple reason that under our definitions we can find such a case. For example Γ_2([a,b,c]) and Γ_2([ac,ab,bc]) are isomorpic. The reason for this is that $[ac, ab, bc]$ and $[a, b, c]$ are complements of each other. We can interpret this behaviour as meaning that in the first case we are tracking the elements a, b and c, in the second we are following their absence. This of course implies that further rewriting is possible in certain circumstances.

In fact,

Theorem 4.5. *Given a list A and denoting it's complement by A^c, we have that if*

$$\frac{weight(A)}{height(A)} \geq \frac{n}{2}$$

then

$$weight(A^c) \leq weight(A).$$

Proof: Let H be the height of A, and let W be the weight of A. We will denote the height of A^c by H^c, and the weight of A^c by W^c. We will denote by k_i the number of positions in which element i appears. Firstly recall that the height is the number of distinct letters

of A. The same number of distinct letters appear in A^c, hence $H^c = H$. Now, recall that the weight is the total number of letters of A, and we have :

$$W = \sum_{i=1}^{H} k_i. \tag{4.5}$$

$$W^c = \sum_{i=1}^{H} (n - k_i). \tag{4.6}$$

$$= \sum_{i=1}^{H} n - \sum_{i=1}^{H} k_i. \tag{4.7}$$

$$= nH - W. \tag{4.8}$$

Now, if

$$\frac{W}{H} \geq \frac{n}{2}. \tag{4.9}$$

$$\frac{W}{H} = \frac{n}{2} + \delta, \qquad \delta \geq 0. \tag{4.10}$$

$$nH = 2W - \epsilon, \qquad \epsilon \geq 0. \tag{4.11}$$

And hence by equations 4.8 and 4.11 we have that $W^c = W - \epsilon$, i.e. $W^c < W$ if ϵ is not zero. If ϵ is zero then we have that $W^c = W$. ∎

This idea can be used in conjunction with the other rewriting ideas to produce equivalent smaller (both in number and magnitude of coprime entries) lists in order that digraph calculation (a breadth first search algorithm) will proceed more speedily.

4.4 Input Shape

The previous definitions (4.10 and 4.11) of the height and weight of the representational signature of an input diagonal allow us to discuss broad classes of problems, and compare more easily the similarity of given input diagonals. We will utilise the term 'shape' of an input to refer to the particular distribution of a given input rather than its signature class. Note that we have to be quite careful when utilising these terms as the height and weight themselves do not uniquely identify the digraph associated with a list. For instance the two lists $L_1 = [A, B, C, ABD, ABE]$, and $L_2 = [A, B, AC, BE, ABD]$ both have height 5, weight 9 and SNF $[1, 1, AB, AB, ABCDE]$, but the associated digraphs differ (L_1 has 17 vertices, 45 edges and L_2 has 18 vertices and 52 edges).

Even though height and weight do not uniquely define the problem, we can use the concepts to get an idea of the nature of the problem. We will utilise these concepts to describe a 'statespace' of possible inputs. Firstly, however, we will discuss the shape of the input in the worst case scenario.

4.4.1 A Worst Case Input

A worst case input, W_n, generating a directed graph with the maximum number of vertices, edges and the longest paths can be computed quite easily. It should be obvious that an input with a signature that has exactly one letter in every proper set of places must generate such a graph. This can be achieved as follows: Take all combinations of $[1, \ldots, n]$ and discard the two trivial ones ([ ] and $[1, \ldots, n]$). Then assign a distinct prime to each particular combination. Then each of the n numbers can be constructed by taking the product of the relevant primes belonging to the combinations in which the number appears. This covers all possibilities for any coprime part appearing in any selection of the positions and so we have a worst case input. This gives a 'Universal Graph', for this problem with a length n input, of which all directed graphs of other inputs of length n will be quotients, since removal of a particular set of elements from W_n corresponds to some collapse of vertices in the associated graph.

An example construction of the worst case of length 3 is shown in Table 4.1.

pos	a	b	c	d	e	f
1	1	0	0	1	1	0
2	0	1	0	1	0	1
3	0	0	1	0	1	1

Table 4.1: Generation of a worst case length 3 diagonal

i.e. $[ade, bdf, cef]$ is as bad an input as possible for the length 3 problem, having height 6, weight 9.

The worst case diagonal matrix of length 4 can be similarly constructed as shown in Table 4.2 so here a worst case diagonal looks like

$$[acegikm, bcfgjkn, defglmn, hijklmn]$$

pos	a	b	c	d	e	f	g	h	i	j	k	l	m	n
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0
2	0	1	1	0	0	1	1	0	0	1	1	0	0	1
3	0	0	0	1	1	1	1	0	0	0	0	1	1	1
4	0	0	0	0	0	0	0	1	1	1	1	1	1	1

Table 4.2: Generation of a worst case length 4 diagonal

with height 14 and weight 28. The SNF of this input is

$$[1, gkmn, cefgijklmn, abcdefghijklmn].$$

It is simple to see that the worst case diagonal of length n has height $2^n - 2$ and weight $n * (2^{n-1} - 1)$.

We are interested in the structure of these worst case graphs. Enumeration of the first few $\Gamma_2(W_i)$ where W_i is the worst case of length i reveals the details in table 4.3.

Length	2	3	4	5
Vertices	2	11	261	43337
Edges	1	15	633	154570
Paths	1	6	708	3269040

Table 4.3: Various details of $\Gamma_2(W_i)$

It should be obvious that these numbers are growing very rapidly, and it has in fact been beyond our ability to produce such complete results for larger worst case scenarios.

4.4.2 State Space of Inputs

In order that we will be able to pick a reasonable selection of cases upon which to test our algorithms we first need to understand the set from which we will be selecting these cases. We will refer to this set, or state space, as ${}^n\Upsilon$, the set of all inputs of length n. We can get a good idea of the characteristics of this set from the concepts of height and weight we have already descibed.

We have seen that we can 'rewrite' an arbitrary diagonal in terms of its coprime parts. Also as mentioned the worst case described above 'contains' all other possible diagonals.

We can take this worst case, containing $2^n - 2$ coprime factors and look at the 2^{2^n-2} cases of various elements being present or not. Clearly there are a large number of possible different diagonals and enumerating them all and plotting the results is unfeasible. There is however also a large amount of symmetry, and by examining the restrictions on the possible weights for a given height we can produce a 'statespace' diagram of weight vs height, as for example in Figure 4.5 which shows the boundaries of the state space of inputs for integer lists of various length.

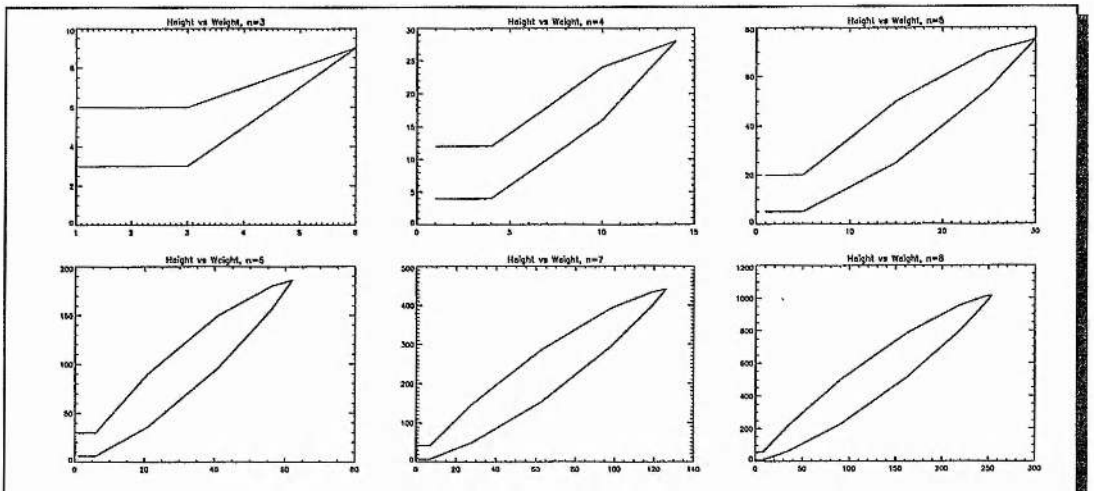


Figure 4.5 : Height / Weight State Space boundaries for length n lists, n=3 to n=8

Given the evident shape of the statespace, we conclude that it should be reasonably simple to sample effectively for algorithm testing.

We can derive an upper bound $\frac{2^{2^n-2}}{n!}$ for the number of non-isomorphic digraphs for any given length of problem by considering a worst case input of length n. Each possible input is a quotient of the worst case and can be viewed as a binary representation of an integer between 1 and 2^{2^n-2}. The full group of symmetries of the n positions acts. The result follows. Some of these selections are not valid for all purposes as they actually define problems on smaller diagonals for example $[1, a, b]$, $[1, a, a]$, or $[b, ab, a]$. It is not immediately obvious how to count these precisely, or whether they should be included, but they are few in number for moderate or large n.

It is worth mentioning here that height and weight are not invariant as we progress through the digraph of a given problem. A worst case SNF

$$[1, p_1, p_1p_2, \ldots, \prod_{i=1}^{n} p_i]$$

has height $n-1$ and weight $\binom{n}{2}$ compared to the worst case input which has height 2^n-2 and weight $n*(2^{n-1}-1)$. The point, in the state space of $\Gamma(D)$, associated with D_η will be below and to the left of the point associated with $D_{\eta-1}$. We will return to this idea as the basis of an algorithm in section 6.5.

Figure 4.6 shows the five possible 'different' inputs for the 3×3 case with their graphs, excluding the degenerate cases. For the 4×4 case a simple enumeration using GAP shows there to be at least 418 different graphs.

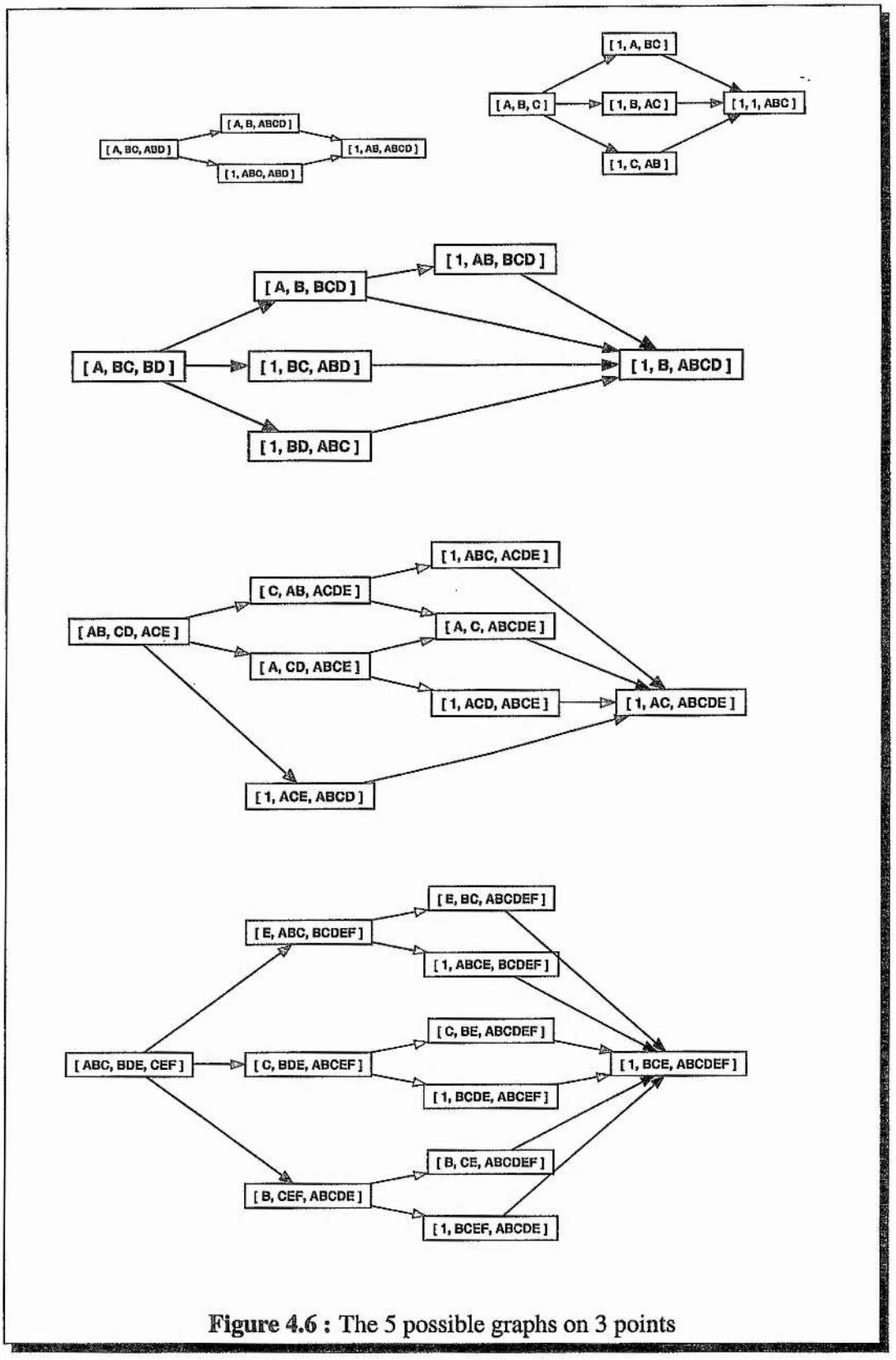


Figure 4.6 : The 5 possible graphs on 3 points

4.5 Coprime Entries

In some sense the opposite of the worst case is the coprime case, i.e. $D = [d_1, d_2, \ldots, d_n]$ where $\text{Gcd}(d_i, d_j) = 1, \forall i, j : i \neq j$. In this case we have n coprime entries, i.e. the problem has height and weight both equal to n. We find that the structure of the associated directed graph is particularly limited. It is not the smallest of the graphs in terms of number of vertices or edges (see e.g. Figure 4.6), but it does permit some pleasing observations.

Since at each pairwise Gcd-Lcm step we 'generate' a 1, it follows that the length of each path through the graph from D to S is $n - 1$. We can readily split this graph into various 'levels', where the number of steps taken to reach a level from the input is the number of 1's in each vertex of that level.

We have that at each level of this graph the number of vertices is equal to the number of ways to partition a set of n elements into k pairwise disjoint nonempty subsets and so we see that the number of vertices in the graph is equal to the Bell number $B(n)$ i.e. it is the sum of the Stirling numbers of the second kind.

$$\begin{aligned}
\text{Number of Vertices} &:= \text{Bell}(n). && (4.12)\\
&:= \sum_{m=1}^{n} \left(\text{Stirling2}(n, x)\right). && (4.13)\\
&:= \sum_{m=1}^{n} \left(\frac{1}{m!} \sum_{k=0}^{m} \left((-1)^{m-k} \binom{m}{k} k^n \right) \right). && (4.14)
\end{aligned}$$

A formula for the number of edges follows readily from the fact that within each level each vertex has the same number of edges to new vertices beneath it being $\binom{k}{2}$, and so

$$\begin{aligned}
\text{Number of Edges} &:= \sum_{m=2}^{n} \left(\text{Stirling2}(n, x) \binom{x}{2} \right). && (4.15)\\
&:= \sum_{m=2}^{n} \left(\frac{1}{m!} \sum_{k=0}^{m} \left((-1)^{m-k} \binom{m}{k} k^n \binom{x}{2} \right) \right). && (4.16)
\end{aligned}$$

We can also simply derive an expression for the number of paths through through this graph. Since, as mentioned previously, within each level each vertex has $\binom{k}{2}$ edges to vertices on the next level then,

$$\begin{aligned}
\text{Number of Paths} &:= \prod_{x=2}^{n} \binom{x}{2}. && (4.17)\\
&:= \frac{(n-1)!n!}{2^{n-1}}. && (4.18)
\end{aligned}$$

It is obvious from this simple analysis that enumerating all possible paths, even in this relatively simple case, and selecting the optimal solution is not a practical option for a diagonal with more than a few entries.

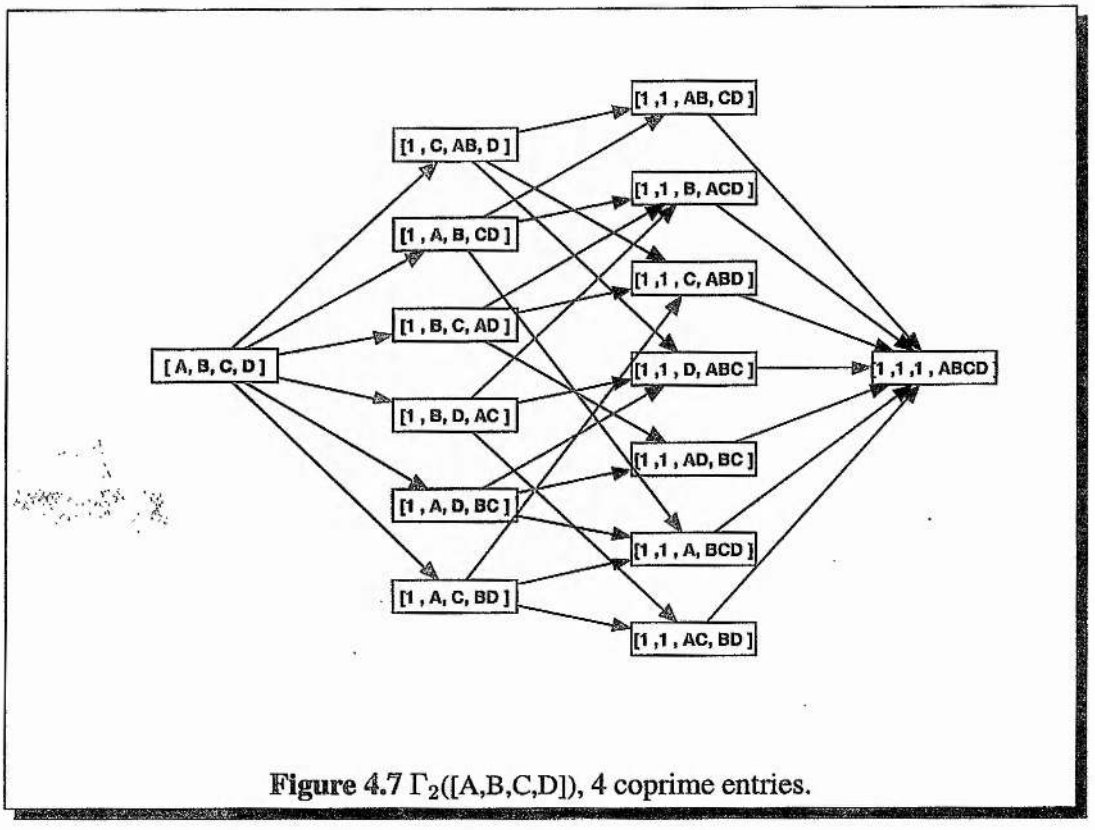


Figure 4.7 Γ_2([A,B,C,D]), 4 coprime entries.

4.6 Conclusions

We have seen in this chapter that the graph structures associated with even apparently simple diagonal inputs can be incredibly complex. Enumerating all possible paths is clearly not an option except in very small cases. In the next two chapters we will examine first how to perform each 2×2 step and then describe and analyze some algorithms for selecting paths through these digraphs.

Chapter 5

The 2 x 2 Problem

As we have seen the solution of the overall problem can be naturally broken down into the solution of smaller problems. A natural line of investigation is then to consider in detail the subproblem which we will take as the base case.

If S is the SNF of a diagonal matrix D then S_{11} is the gcd of all the d_{ii}, and if $S = PDQ$ then the first row of P and the first column of Q give an integer multiplier vector V such that $\sum_{i=1}^{n} v_i d_{ii} = s_{11}$ where $v_j = p_{1j}q_{ji}$.

Finding 'good' solution vectors for arbitrary length sets of integers is difficult, see for example [MH94] wherein it is shown (Corollary 7) that, given a positive integer K and a sequence of n positive integers $A = \{a_1 \ldots a_n\}$, the task of expressing $g = gcd(a_1 \ldots a_n)$ in the form

$$\sum_{i=1}^{n} x_i a_i = g$$

with $|x_i| \leq K$, is NP-complete. i.e. it is difficult to find an optimal solution with respect to the L_∞ (or max) norm.

For the case $n = 2$ however the extended euclidean algorithm provides us with an efficient method of finding good vector multipliers for the gcd of a pair of integers.

We are, of course, actually interested in the more complex construct of a pair of multiplier matrices rather than just a multiplier vector. We will take the euclidean algorithm as our starting point and see how it relates to the problem of finding good multiplier matrices for the SNF of a 2×2 diagonal integer matrix. We shall also mention in passing a few points about the 3×3 case and the further problems associated with finding solutions.

We need to find unimodular integer matrices P and Q such that $PDQ = S$ is in Smith Normal Form, where D is a 2×2 diagonal integer matrix. We only need to consider the case with coprime diagonal entries, as any common divisor will not affect the multiplier matrices since $PkDQ = kS$. From section 5.2 on we will deal only with this case.

$$P \begin{bmatrix} a & 0 \\ 0 & b \end{bmatrix} Q = \begin{bmatrix} gcd(a,b) & 0 \\ 0 & lcm(a,b) \end{bmatrix} \tag{5.1}$$

Note that we will generally assume $a > b$. We will write $x \equiv y\,(a)$ to mean x is equivalent to y modulo a.

5.1 The Extended Euclidean Algorithm

As we will be building our techniques on the extended euclidean algorithm we will first recall some elementary facts.

Theorem 5.1. *Let a and b be integers, and let $d = gcd(a,b)$. Then there exist integers s and t such that $as + bt = d$. The extended euclidean algorithm computes d, s and t from a and b.*

Proof: See e.g. [BS96b]. ∎

Note that the coefficients s and t of the vector multiplier are not uniquely defined, but we can describe all possible pairs of coefficients.

Theorem 5.2. *Let $d = gcd(a,b) = as + bt = au + bv$, then*

$$u = s + \frac{kb}{d} \qquad \text{and} \qquad v = t - \frac{ka}{d}$$

for some integer k.

Proof: We have $a(s-u) = b(v-t)$ which must equal some multiple k of the $lcm(a,b) = \frac{ab}{d}$. Hence $s - u = \frac{kb}{d}$ and $v - t = \frac{ka}{d}$. ∎

Since we can 'move' easily amongst the solutions it should be easy to see that we can find small solutions. In fact we have

Theorem 5.3. *If $gcd(a,b) = as + bt$, then s and t can be chosen such that $|s| \leq |b|$ and $|t| \leq |a|$.*

Proof: Clearly by theorem 5.2 we can find s such that $|s| \leq \frac{1}{2}\left|\frac{b}{d}\right| \leq |b|$. Since $bt = d - as$, we have that $|bt| \leq \frac{1}{2}\left|\frac{ab}{d}\right| + d$. Since $|d| \leq |b|$ it follows that $|t| \leq \frac{1}{2}\left|\frac{a}{d}\right| + 1$. Finally if $a = 1$ we can take $t = 0$, and otherwise we have $\frac{1}{2}\left|\frac{a}{d}\right| + 1 \leq |a|$. ■

This will suffice for some of our purposes but we can quite easily deduce much tighter results which will in turn provide tighter bounds for some upper bound derivations in chapter 8.

Theorem 5.4. *If $a > b > g$ where $g = gcd(a, b) = as + bt$, then s and t can be chosen such that $|s| \leq \left|\frac{b}{2g}\right|$ and $|t| < \left|\frac{a}{2g}\right|$.*

Proof: Denoting $\frac{a}{g}$ by α and $\frac{b}{g}$ by β we have

$$\alpha s + \beta t = 1. \tag{5.2}$$

We can find t such that $|t| \leq \left|\frac{\alpha}{2}\right|$ using theorem 5.2. Now we can rearrange equation 5.2 and substitute for t :

$$\alpha s = 1 - \beta t, \tag{5.3}$$

$$|\alpha s| \leq |1| + |\beta t|, \tag{5.4}$$

$$\leq |1| + \left|\frac{\alpha\beta}{2}\right|, \tag{5.5}$$

$$|s| \leq \left|\frac{1}{\alpha}\right| + \left|\frac{\beta}{2}\right|. \tag{5.6}$$

But $a > b > g$ i.e. $\alpha > \beta > 1$ and so $\alpha > 2$ so $\left|\frac{1}{\alpha}\right| < \frac{1}{2}$ and it therefore follows that $|s| \leq \left|\frac{\beta}{2}\right|$, which gives us

$$|s| \leq \left|\frac{b}{2g}\right| \tag{5.7}$$

as required. Assume further that we have $|t| = \left|\frac{\alpha}{2}\right|$, then substituting for t gives

$$\alpha s \pm \frac{\alpha\beta}{2} t = 1, \tag{5.8}$$

$$\Rightarrow \quad 2s \pm \beta = \frac{2}{\alpha}. \tag{5.9}$$

Which implies that $\alpha = 2$ since $2s \pm \beta \in \mathbb{Z}$, but we have $\alpha > 2$, so we have a contradiction and we can therefore select $|t| < \left|\frac{\alpha}{2}\right|$, which is to say

$$|t| < \left|\frac{a}{2g}\right|. \tag{5.10}$$

■

Corollary 5.1. *Note that if $b > 2$ then by a similar argument to steps 5.8- 5.10 in the proof of theorem 5.4 we have $|s| < \left|\frac{b}{2g}\right|$.*

We make one further observation about the coefficients computed by the extended euclidean algorithm,

Theorem 5.5. *If $as + bt = gcd(a, b)$, then $gcd(s, t) = 1$.*

Proof: Dividing through we have $\frac{a}{gcd(a,b)}s + \frac{b}{gcd(a,b)}t = 1$. ∎

Note that the effect of premultiplying a column vector $[a, b]^T$ by a unimodular matrix is equivalent to performing some sequence of row operations upon that vector. Now, starting with a 2×2 identity matrix and performing row operations corresponding to the steps of the euclidean algorithm, and applying those same operations to that identity matrix, will yield a unimodular multiplier matrix and a column vector $[gcd(a, b), 0]^T$.

Then the multiplier matrix is of the form

$$\begin{bmatrix} s & t \\ x & y \end{bmatrix}$$

where $as+bt = gcd(a, b)$. Note that the determinant of the matrix is $sy-tx = 1$, showing again that $gcd(s, t) = 1$.

This is equivalent to the statement that we can find a unimodular 2×2 matrix P such that

$$P \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} gcd(a, b) \\ 0 \end{bmatrix}$$

and gives a constructive method of finding such a matrix.

We will also need some elementary lemmata about gcds.

Lemma 5.1. *Let a, b, k be integers. Then $gcd(a, bk) = gcd(a, b) * gcd(\frac{a}{gcd(a,b)}, k)$.*

Proof: The factors of $gcd(a, bk)$ common to both a and b appear in $gcd(a, b)$. The further factors of $gcd(a, bk)$ are those common to a and k not in a and b i.e. $gcd(\frac{a}{gcd(a,b)}, k)$. ∎

Corollary 5.2. *$gcd(a, bk) = gcd(a, b)$ iff $gcd(\frac{a}{gcd(a,b)}, k) = 1$.*

Corollary 5.3. *If $gcd(a, b) = 1$, then $gcd(a, bk) = gcd(a, b)$ iff $gcd(a, k) = 1$.*

With these results we can now describe the 2×2 SNF algorithm.

5.2 Basic Procedure

Given the 2×2 diagonal integer matrix $D = \begin{bmatrix} a & 0 \\ 0 & b \end{bmatrix}$ with $gcd(a, b) = 1$. As previously noted we will examine the case $a > b$. We can now consider the following basic procedure.

1. Column operation - We begin by adding some multiple k of the second column to the first, with the restriction that $gcd(a, k) = 1$, to obtain
$$\begin{bmatrix} a & 0 \\ bk & b \end{bmatrix}.$$
2. Row operations. Applying theorem 5.5 to the first column we can obtain
$$\begin{bmatrix} gcd(a, bk) & bt \\ 0 & lcm(a, b) \end{bmatrix}.$$
3. And then with one more column operation we can zero out the upper corner entry and obtain the SNF.

A Couple of Notes

In step one we restrict ourselves to multiples k such that $gcd(a, k) = 1$. This allows us to apply Lemma 5.1 and so be sure that step 3 is possible and the operation proceeds smoothly to SNF. If we were to allow any k then the algorithm could fail.

For example, taking $a = 6, b = 5$ and $k = 2$ and following the procedure outlined above we would proceed as follows :

$$\begin{bmatrix} 6 & 0 \\ 0 & 5 \end{bmatrix} \rightarrow \begin{bmatrix} 6 & 0 \\ 10 & 5 \end{bmatrix} \rightarrow \begin{bmatrix} 6 & 0 \\ 4 & 5 \end{bmatrix} \rightarrow \begin{bmatrix} 2 & -5 \\ 4 & 5 \end{bmatrix} \rightarrow \begin{bmatrix} 2 & -5 \\ 0 & 15 \end{bmatrix}.$$

Example 5.1 : Why the choice of k is restricted.

Here we reach a situation where we cannot simply apply a single column operation to zero out the off-diagonal entry. Most, if not all, current implementations of this diagonal to SNF sub-step do not encounter this problem as they simply add a single copy of one column to the other (that is to say they choose $k = 1$), and then proceed with the euclidean

algorithm. A simple example that demonstrates that adding a single multiple of a column is not optimal is the problem of finding multiplier matrices P, Q such that

$$P \times \begin{bmatrix} 97 & 0 \\ 0 & 2 \end{bmatrix} \times Q = \begin{bmatrix} 1 & 0 \\ 0 & 194 \end{bmatrix}.$$

Applying the standard procedure, i.e. selecting $k = 1$ we find the solution

$$\begin{bmatrix} 1 & -48 \\ -2 & 97 \end{bmatrix} \times \begin{bmatrix} 97 & 0 \\ 0 & 2 \end{bmatrix} \times \begin{bmatrix} 1 & 96 \\ 1 & 97 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 194 \end{bmatrix}.$$

This solution has $\mathcal{Q}_{[2,2,+]} = 7586$ and $\mathcal{Q}_{[2,2,\times]} = 13641949$. However we find, by a simple enumeration of the solution pairs P, Q arising from applying the basic procedure described above for various k, that $k = 7$ appears to be the best choice. The solution we obtain in this case is

$$\begin{bmatrix} -1 & 7 \\ -14 & 97 \end{bmatrix} \times \begin{bmatrix} 97 & 0 \\ 0 & 2 \end{bmatrix} \times \begin{bmatrix} 1 & -14 \\ 7 & -97 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 194 \end{bmatrix},$$

which appears to be the best quality solution obtainable under this procedure. The qualities associated with this solution are $\mathcal{Q}_{[2,2,+]} = 4827$ and $\mathcal{Q}_{[2,2,\times]} = 5826189$.

Now we will consider the impact of the above procedure on the actual multiplier matrices. We have $PDQ = S$ where P is a unimodular matrix computable with the extended euclidean algorithm (Theorem 5.5). It is the product of a sequence of elementary matrices corresponding to the steps of the euclidean algorithm. It should be clear from Theorem 5.3 or 5.4 that we can determine simple upper bounds for the magnitude of the entries of P (assuming we are careful). In fact it can be arranged so that the largest entry in P has magnitude equal to the greater of $|a|$ and $|bk|$. The column multiplier matrix, Q, produced by following the basic procedure is simply the product of the two elementary matrices,

$$\begin{bmatrix} 1 & 0 \\ k & 1 \end{bmatrix} \times \begin{bmatrix} 1 & -bt \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & -bt \\ k & 1 - bkt \end{bmatrix}.$$

Note that we can similarly arrange, by choice of k and care in step 2, that $|kt|$ is bound by $|a|$ and so the largest entry in Q is then bound by $|ab|$.

We can actually write down explicit multiplier matrices for the Smith Normal Form of an arbitrary 2×2 diagonal matrix. Specifically we have:

$$\begin{bmatrix} \frac{1-bkt}{a} & t \\ -bk & a \end{bmatrix} \times \begin{bmatrix} a & 0 \\ 0 & b \end{bmatrix} \times \begin{bmatrix} 1 & -bt \\ k & 1-bkt \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & ab \end{bmatrix}. \tag{5.11}$$

Figure 5.2 : Explicit multipliers for the 2×2 problem

where we have $as + bkt = 1$, rather than the usual $as + bt = 1$. i.e.

$$bkt \equiv 1(a). \tag{5.12}$$

As we remarked these are the multiplier matrices assuming a, b to be coprime. However note that applying the algorithm to ga, gb will give the same result.

5.3 Optimizing the Multiplier Matrices

Given a 2×2 diagonal matrix D we now have explicit 2×2 multiplier matrices P, Q such that $PDQ = S$ (Equation 5.11). We investigate how to select k and t under the restriction provided by equation 5.12 in order to minimize some particular measures on these multiplier matrices. Each measure reduces to a function of a, b, k and t. There are several strategies we can employ to proceed with our search:

5.3.1 Brute Force

Run through all values of k in some range and for each k calculate $t \equiv (bk)^{-1}\ (a)$. Although there are an infinite number of possible t the one with the smallest magnitude is usually the one that will minimize our measure for that value of k. Do this for each k and then select the k, t pair that is optimal. The problem, of course, with this exhaustive method is that a can be rather large, so that the search space becomes excessive.

5.3.2 More Intelligent

In this search we are interested in minimizing some $\mathcal{Q}(P, Q)$ which will be a function of k and t, $f(k, t)$, which is generally increasing in both k and t. One immediate improvement

is to search in order of increasing k and stop the search when $f(k,0)$ exceeds the smallest $f(k',t')$ already found.

A further improvement along these lines is to let $k = 1$, calculate t and then $f(1,t)$, then since we can just as easily calculate k given t, let $t = 1$, calculate $f(k,1)$. We then alternately increase each of k,t, continuing until we reach a point where $f(k_\kappa, t_\tau)$ is greater than our best $f(k,t)$ to that point.

5.3.3 Factorization Based Methods

We can start by considering possible values for kt, which is known modulo a from $as + bkt = 1$. Taking v to be between 0 and $a-1$ such that $as+bv = 1$ our candidates for kt are $v + ma$ and, while the exact bound varies for different $\mathcal{Q}$, typically only $m \in [-2,\ldots,2]$ need be considered.

For fixed kt we can then consider the possible factorizations into k and t.

Even this method is expensive and examines many options that are unlikely to be relevant. For fixed values of kt we can examine and possibly simplify the measure $f(k,t)$ and see heuristically where good choices of k and t may lie. Generally we will show that we wish $\frac{k}{t}$ to be close to some value, r depending on the metric and the pair a,b. Once we know r and have factorized kt as $v_1 \ldots v_\omega$, where $v_i \geq v_{i+1}$ we can then arrange $\frac{k}{t}$ to be as close as possible to r by starting with $k_1 = 1, t_1 = 1$ and for each v_i in turn, multiplying either k_i or t_i by v_i in order to best preserve the ratio r.

We will now consider two scenarios and examine some of the possible measure functions that arise with a view to selecting k in order to find a good quality solution, i.e. one which hopefully minimizes the measure.

These scenarios are :

- we wish to minimize the solution to this 2×2 problem in isolation;
- or the problem is part of a larger $n \times n$ calculation.

5.4 As an Isolated Problem

Considering this step in isolation we seek to optimise the quality of P,Q by our choice of k,t under condition 5.12. We can consider the problem for particular fixed values of kt

(which we can assume to be $O(a)$ by theorem 5.3) and then we can investigate the manner in which various ratios of $\frac{k}{t}$ affect the approximate order of the function we are examining. We shall examine in more detail the two likely measures of choice $\mathcal{Q}_{[2,2,+]}$ and $\mathcal{Q}_{[2,2,\times]}$.

5.4.1 $\mathcal{Q}_{[2,2,+]}$

Recall $\mathcal{Q}_{[2,2,+]}$ is:

$$\frac{1}{4}\left(\sum_{i=1}^{n}\sum_{j=1}^{n}|P_{ij}|^2+\sum_{i=1}^{n}\sum_{j=1}^{n}|Q_{ij}|^2\right)$$

Minimizing this should give a solution where all the entries of both matrices are of roughly similar magnitude. For fixed kt and ignoring the constant multiplier $\frac{1}{4}$ the function we are trying to minimize is:

$$f(k,t) := \left(\frac{1-bkt}{a}\right)^2 + t^2 + b^2k^2 + a^2 + 1 + b^2t^2 + k^2 + (1-bkt)^2 .$$

Note kt is constant, hence $\left(\frac{1-bkt}{a}\right)$ is constant. The variable part of f is then

$$g(k,t) := \left(t^2+k^2\right)\left(b^2+1\right).$$

And so it is clear that to minimize this function we require $\frac{k}{t} \approx 1$. Note that it may be impossible to achieve this ratio exactly as we are working over the integers.

5.4.2 $\mathcal{Q}_{[2,2,\times]}$

Recall $\mathcal{Q}_{[2,2,\times]}$ is

$$\frac{1}{16}\left(\sum_{i=1}^{n}\sum_{j=1}^{n}|P_{ij}|^2\right)\times\left(\sum_{i=1}^{n}\sum_{j=1}^{n}|Q_{ij}|^2\right).$$

Attempting to find a solution that minimizes this metric will hopefully lead to a solution in which the entries of P and Q respectively are of roughly equal magnitude. Ignoring the constant factor of $\frac{1}{16}$, we have

$$f(k,t) := \left(\left(\frac{1-bkt}{a}\right)^2 + t^2 + b^2k^2 + a^2\right)\times\left(1 + b^2t^2 + k^2 + (1-bkt)^2\right).$$

Recall that $(1 - bkt)^2 = (as)^2$ for some non-zero constant s. And so, for constant kt we see that the variable part of f is

$$g(k,t) := b^2\left(t^4 + k^4\right) + t^2\left(s^2b^2 + 1 + a^2s^2 + a^2b^2\right) + k^2\left(s^2 + b^2 + a^2b^2s^2 + a^2\right)$$

If we make the reasonable assumption that kt is of $O(a)$ and that $s = c + hb$ is $O(b)$ then we can take k and t such that both are less than or equal to $O(a)$. We can see that g is no worse than $O(a^4b^4)$. From the 4th powers it would appear that $\frac{k}{t} \approx 1$ is again a sensible choice i.e. k and t would be roughly $O(\sqrt{a})$, and in this case we can see that g would be no worse than $O(a^3b^4)$. In fact it appears that the quadratic term is likely to dominate. This term is itself dominated by :

$$h := t^2\left(a^2s^2 + a^2b^2\right) + k^2\left(a^2b^2s^2\right).$$

Of which the variable parts are roughly (assuming $s = c + hb$ is $O(b)$ and so ignoring the constant multiplier a^2b^2)

$$z := 2t^2 + b^2k^2$$

Which suggests that $\frac{k}{t} \approx \frac{2}{b}$ is a sensible ratio. This in turn suggests that k is $O(\sqrt{\frac{a}{b}})$ and t is $O(\sqrt{ab})$. We can see that using this ratio we have g of $O(a^3b^3)$ at worst, which appears to be about as good as we can get.

5.5 As an Intermediary Step

In fact of course this step is not usually isolated but is part of an ongoing calculation and our goal is to optimize the final $n \times n$ multiplier matrices. The first stage in our calculation is to select a path in $\Gamma(D)$ which minimizes the power build up to be discussed in the next chapter. The effect of this is likely to be that the entries of P_η and Q_η are of uniform magnitude and then we can start to select optimization strategies based on selecting k and t to minimize the effect of these multiplier matrices on our current transformation matrices.

We need to consider the likely form of the multiplier matrices before our 2×2 step. There are two cases:

- If we are in the early stages of the diagonal to SNF process, having started with P_0, Q_0 identity matrices then P_η, Q_η will still be sparse. In this case the conclusions of the isolated step analysis should be appropriate.

- If we have P_0, Q_0 from a diagonalization process or we have already performed substantial work in the diagonal to SNF process (especially if we use the "power growth" heuristics of the next chapter) , then it is likely to be that the entries of P_η and Q_η are of uniform magnitude. This is the situation we will now investigate.

Given an $n \times n$ diagonal matrix D at some stage of the calculation we have multiplier matrices P_η, Q_η. We then calculate matrices U, V such that $UD|_{\{i,j\}}V$ is in SNF. We will examine how $P_{\eta+1} = U^{\{i,j\}}P_\eta$ and $Q_{\eta+1} = Q_\eta V^{\{i,j\}}$ differ from P_η, Q_η. Let U, V be as follows with entries of magnitude $u_{\{1\ldots4\}}, v_{\{1\ldots4\}}$.

$$\begin{pmatrix} \boxed{u_1 \quad u_2} \\ \\ \boxed{u_3 \quad u_4} \end{pmatrix} \times \begin{pmatrix} D_i & 0 \\ \\ 0 & D_j \end{pmatrix} \times \begin{pmatrix} \boxed{v_1} & \boxed{v_3} \\ \boxed{v_2} & \boxed{v_4} \end{pmatrix} = \begin{pmatrix} gcd(D_i, D_j) & 0 \\ \\ 0 & lcm(D_i, D_j) \end{pmatrix}.$$

If we assume, as previously mentioned, that the entries in the multiplier matrices P_η, Q_η are similar in magnitude, and bound by x then we can estimate the damage done by the step involving $U^{\{i,j\}}$ and $V^{\{i,j\}}$. We shall examine the damage done to P but note that the analysis for Q is very similar, differing only in the order of multiplication of the matrices. We can see that premultiplication by $U^{\{i,j\}}$ will change the bound for the entries in only two rows.

$$U^{\{i,j\}} \times \begin{bmatrix} x & \cdots & x \\ \\ \\ \\ \vdots & & \vdots \\ \\ \\ \\ x & \cdots & x \end{bmatrix} = \begin{bmatrix} x & \cdots & x \\ \vdots & & \vdots \\ x & \cdots & x \\ x(u_1+u_2) & \cdots & x(u_1+u_2) \\ x & \cdots & x \\ \vdots & & \vdots \\ x & \cdots & x \\ x(u_3+u_4) & \cdots & x(u_3+u_4) \\ x & \cdots & x \\ \vdots & & \vdots \\ x & \cdots & x \end{bmatrix}$$

And so the 'damage' done by this step is $\mathcal{Q}(P_{\eta+1}) - \mathcal{Q}(P_\eta)$ for some measure $\mathcal{Q}$. If we examine our usual sum of squares metric $\|P\|_2 = \frac{1}{n^2}\sum_{i,j=1}^{n} |P_{ij}|^2$ then, ignoring the $\frac{1}{n^2}$ constant, we see that the damage is at most

$$n(n-2)x^2 + (u_1+u_2)^2x^2 + (u_3+u_4)^2x^2 - n^2x^2$$

$$= x^2 \left((u_1 + u_2)^2 + (u_3 + u_4)^2 - 2n\right).$$

And so it appears that we should be looking at the contributions of $(u_1+u_2)^2$ and $(u_3+u_4)^2$. Similarly for V. The obvious measures to now attempt to minimize are

$$\left((u_1 + u_2)^2 + (u_3 + u_4)^2\right) + \left((v_1 + v_2)^2 + (v_3 + v_4)^2\right) \tag{5.13}$$

and

$$\left((u_1 + u_2)^2 + (u_3 + u_4)^2\right) \times \left((v_1 + v_2)^2 + (v_3 + v_4)^2\right) \tag{5.14}$$

We will denote the measures defined by equations 5.13 and 5.14 by $\mathcal{Q}_{|+|}$ and $\mathcal{Q}_{|\times|}$ respectively.

5.5.1 $\mathcal{Q}_{|+|}$

Substituting the relevant terms for u_i, v_i we see that the function we wish to minimize subject to $bkt \equiv 1\ (a)$ is:

$$f(k,t) := \left(\left|\frac{1-bkt}{a}\right| + |t|\right)^2 + (|bk| + |a|)^2 + (1 + |bt|)^2 + (|k| + |1 - bkt|)^2.$$

We will assume that both a and b are positive. We will take a positive value for $b^{-1}\ (a)$ and hence k, t will both either both be positive, or both be negative. We shall assume that both are positive. Calculation actually showsthat inverting any of these decisions makes no difference to the final result about the (magnitude of the) suggested ratio $\frac{k}{t}$.

$$f(k,t) := \left(\frac{bkt-1}{a} + t\right)^2 + (bk + a)^2 + (1 + bt)^2 + (k + bkt - 1)^2.$$

Observe that $(bkt - 1) = (as)$ for some positive constant s. And so, for constant kt we have the variable parts of f are:

$$g(k,t) := (t^2 + k^2)(b^2 + 1) + 2(t + ak)(s + b).$$

Here the quadratic powers suggest that $\frac{k}{t} \approx 1$ would be a sensible choice, i.e. selecting k and t both of $O(\sqrt{a})$. Then the quadratic powers would be $O(ab^2)$ and the linear powers would be $O(a^{\frac{3}{2}}b)$, so $O(g)$ depends upon the exact relationship between a and b.

To minimize the contribution of the linear powers of k, t, we see that we require the ratio $\frac{k}{t} \approx \frac{1}{a}$, i.e. selecting k to be $O(1)$ and t of $O(a)$. Then the linear powers would be $O(ab)$ but the quadratic powers would be $O(a^2b^2)$ and hence g would be $O(a^2b^2)$.

It therefore appears that the ratio, $\frac{k}{t} \approx 1$ is the best choice, and we can, in fact, see no way to improve upon the order of g by using a different ratio.

5.5.2 $\mathcal{Q}_{|\times|}$

$$f(k,t) := \left(\left(\left| \frac{1-bkt}{a} \right| + |t| \right)^2 + (|bk| + |a|)^2 \right) \times \left((1+|bt|)^2 + (|k| + |1-bkt|)^2 \right).$$

Under the same positivity criteria we used for $\mathcal{Q}_{|+|}$ and again replacing $(bkt-1)$ by (as) for some positive constant s, we have:

$$f(k,t) := \left((s+t)^2 + (bk+a)^2 \right) \times \left((1+bt)^2 + (k+as)^2 \right).$$

And so, for constant kt, the variable part of f is :

$$\begin{aligned} g(k,t) &= \left(k^4 + t^4\right) b^2 \\ &+ \left(ak^3 + t^3\right) 2b\,(bs+1) \\ &+ t^2\left(a^2b^2 + a^2s^2 + b^2s^2 + bs + 1\right) + k^2\left(a^2b^2s^2 + a^2 + b^2 + s^2\right) \\ &+ t \end{aligned}$$

Here the quartic powers suggest $\frac{k}{t} \approx 1$, i.e. k and t both $O(\sqrt{(a)}$, giving g to be $O(a^3b^4)$.

The cubic powers suggest $\frac{k}{t} \approx \frac{1}{a^3}$ is a good ratio, i.e. k is $O(a^{\frac{1}{3}})$ and t is $O(a^{\frac{2}{3}})$, giving g to be $O(a^{\frac{10}{3}}b^2)$, or $O(a^{\frac{8}{3}}b^4)$ depending upon the exact relationship between a and b.

The quadratic powers, approximating slightly, suggest that $\frac{k}{t} \approx \frac{1}{b}$, i.e. k is $O(\sqrt{a\frac{a}{b}})$ and t is $O(\sqrt{ab})$, leading to g being at worst $O(a^3b^3)$.

From these it appears that the best ratio for $\frac{k}{t} \approx \frac{1}{b}$.

5.6 Conclusions

Explicit multiplier matrices have been given for the 2×2 problem. It has been shown that the 'standard' method does not always produce the best result and an improvement has been provided. Heuristic methods have been described by which we can hope to produce small multipliers and some suggestions have been made for various general cases.

Chapter 6

Strategies

Once again we will take D to be an $n \times n$ diagonal integer matrix and we shall write d_i for d_{ii} and S for the SNF of D. We shall continue to use the notation developed in chapter 4. In particular we will use the notation Γ_2(D) for the graph of intermediate diagonal matrices reached by solving 2×2 subproblems. In this chapter we will often refer to pairs of positions or elements, or sequences of pairs. We shall use the square bracket notation $[x, y]$ to denote the ordered pair of positions x and y. We will sometimes abuse this notation to refer also to the entries, d_x and d_y, at those positions. We shall denote a sequence of k such pairs by $[[x_1, y_1], [x_2, y_2], \ldots [x_k, y_k]]$. We shall be using this notation in the context of performing a 2×2 step on each pair where, after the operation on $[x, y]$ we will have $Gcd(d_x, d_y)$ in position x and $Lcm(d_x, d_y)$ in position y.

The problem we are dealing with is solved in two separate stages :

1 Pick a path from D to S in Γ_2(D). See Chapter 4 for discussion of the graph Γ_2(D).

2 Perform the 2×2 step corresponding to each edge of that path as well as possible. See Chapter 5 for this analysis.

We need to select a path through Γ_2(D). In principle we could enumerate all possible paths and take the best result. One efficient method of performing this exhaustive enumeration begins by calculating Γ_2(D) and for each edge, calculating an associated good quality matrix multiplier. All that then remains is to perform all the matrix multiplication chains corresponding to each path. Unfortunately, as we have seen in chapter 4, the size of Γ_2(D) grows very rapidly with n. The number of possible paths through any particular graph is usually far too large to allow exhaustive enumeration.

We therefore need some strategy to allow us to select one, or a few, promising paths without having to examine the entirety of $\Gamma_2(\mathrm{D})$. In this chapter we shall examine some heuristics and develop some effective strategies for selecting paths that generally lead to good overall multiplier matrices.

We should note that there are basically two different types of algorithm we will be considering:

- Structural algorithms in which the next pair is selected according to some virtue of the elements within the current state of the overall problem or by some further lookahead strategy.

  For example selecting the pair $[d_i, d_j]$ with the least gcd (mingcd strategy).

- Positional algorithms in which the next pair is selected with no regard to the actual values of elements.

  For example performing operations on the sequence of positions

  $$[1,2],[1,3],\ldots,[1,n],[2,3],[2,4],\ldots,[2,n],\ldots,[n-1,n].$$

  This obtains the gcd of all n elements in position 1, then the gcd of the remaining $n-1$ elements in position 2 and so on (This is the standard implementation).

Recall our initial example 2.1 with diagonal $D = [6, 78, 130, 143, 231]$. We can see in Example 6 the effect of the two different example strategies outlined above upon the path through $\Gamma_2(\mathrm{D})$.

In this example the structural heuristic finds a minimal length path (two steps) through $\Gamma_2(\mathrm{D})$ and, as we have seen, produced smaller multipliers than the positional heuristic which takes a longer path (6 steps). When actually applying this algorithm some care must be taken; in the example shown the starred steps should have no effect, that is to say that a sensible implementation will check for divisibility before applying a 2×2 step and either do nothing or simply permute the entries. One immediate improvement, especially in this example, would be to precalculate the SNF and reduce the size of our problem by removing any entries of the SNF that appear in the input, the 78 in the example shown. We have of course been deliberately remiss with this degenerate example thus far. With this ostensibly minor improvement the standard positional heuristic would still produce a path of length 5 through $\Gamma_2(\mathrm{D})$ - in fact, depending upon the initial ordering of D, the standard

heuristic produces paths of length as little as 2, or as much as 7 with the 78 included (120 orderings - Path length distribution : $2^8 3^4 4^{24} 5^{42} 6^{26} 7^{16}$, where a^b denotes b paths of length a) or between 2 and 5 without the 78 (24 orderings - Path length distribution : $2^4 4^{12} 5^8$). Note also that in this case we appear to have improved our chances of randomly selecting one of the shortest possible paths from 1 in 15 to 1 in 6, as well as cutting down on the amount of work we need to perform.

However, when we have larger or more complex problems where we cannot see quite so immediately the best course to take, it is useful to fall back on a good positional strategy. For this we shall rely on a positional algorithm which we will proceed to develop in this chapter.

```
Standard (Positional):

  [ 6, 78, 130, 143, 231 ]   Start.
  [ 6, 78, 130, 143, 231 ]   *Pair [1,2] has no effect.
  [ 2, 78, 390, 143, 231 ]   Pair [1,3] makes a little progress.
  [ 1, 78, 390, 286, 231 ]   Pair [1,4] supplies us with the 1st entry of the SNF.
  [ 1, 78, 390, 286, 231 ]   *Pair [1,5] has no effect.
  [ 1, 78, 390, 286, 231 ]   *Pair [2,3] has no effect.
  [ 1, 26, 390, 858, 231 ]   Pair [2,4] moves us 'closer', but 'damages' the 78.
  [ 1, 1, 390, 858, 6006 ]   Pair [2,5] gives us the 2nd entry of the SNF.
  [ 1, 1, 78, 4290, 6006 ]   Pair [3,4] returns the 78, the 3rd entry of the SNF.
  [ 1, 1, 78, 4290, 6006 ]   *Pair [3,5] has no effect.
  [ 1, 1, 78, 858, 30030 ]   Pair [4,5] gives us the 4th and 5th entries.

Mingcd (Structural):

  [ 6,  78, 130, 143, 231 ]   Start - Note gcd(6,143)=1, and gcd(130,231)=1.
  [ 1,  78, 130, 858, 231 ]   Pair [1,4] actually gives two entries of the SNF.
  [ 1,  78, 1, 858, 30030 ]   Pair [3,5] then gives us the remaining two entries.

  and we can then easily permute to obtain the SNF.
```

Example 6: A little intelligence goes a short way

During the rest of this chapter we will first describe the results of some experiments in order to get a better feeling for the problem at hand and to provide a background for the development of the algorithms in the rest of the chapter . We will examine some positional heuristics and develop a good positional heuristic for selecting a path through $\Gamma_2(D)$ in this setting. We will then describe several structural heuristics and then compare these various ideas with a view to deriving a good strategy for general employment.

6.1 Testing and Comparing Algorithms

Our algorithms will be attempting to minimize $\mathcal{Q}_+$ or $\mathcal{Q}_*$ by choosing a path through $\Gamma_2(\mathrm{D})$. A useful experiment is to consider random paths for various D and evaluate the corresponding $\mathcal{Q}$ to provide a background for assessing the utility of an algorithm. In section 8.2.4 we will provide some thoughts on an 'average' case. However, since we will be looking for algorithms that do well in difficult situations, we will have to be a little careful when constructing our test cases.

Path Length / Quality Correlation

We will examine some experimental evidence which suggests a correlation between the length of the path through $\Gamma_2(\mathrm{D})$ and the quality of the solution.

First we will define the parameters of our experiment. Recall from section 4.4.1 that to create a worst case diagonal of length n we require 2^n-2 coprime elements. Each diagonal entry is then the product of half of these. Though the 2×2 multipliers are dependent upon the exact entries, we can try to minimize the effect of large differences between the sizes of the elements, and thus hopefully minimize the effects of ordering, by choosing all our coprime elements to be of similar magnitude.

Recall that W_n is the worst case input of length n, as defined in section 4.4.1. In order to select specific sets of examples from the entire statespace for a length n problem we fix a height, H, generally 10%, 20%, ..., 100% of $height(W_n) = 2^n - 2$. Recall that the height is the number of distinct coprime elements. We then select randomly a set of H elements (by selecting a set of H integers from 1 to $2^n - 2$), each of which could appear in from 1 to $n-1$ positions in the list (determined by examining the binary representation of each element h, i.e. where there's a one there's a h). Thus we have fixed heights and various associated, but random, weights. This procedure therefore allows us to effectively sample slices of the statespace. We perform the tests on length 10 diagonals, using the set of 1022 primes $\{997, \ldots, 9643\}$. We also tried other proportions and sets of elements and obtained similar results.

Having built our example diagonals, we then perform 200 runs through each problem, choosing the next pair at each stage randomly. Scatter plots of $\mathcal{Q}_+$, $\mathcal{Q}_*$ vs number of edges in the path follow in Figure 6.1.

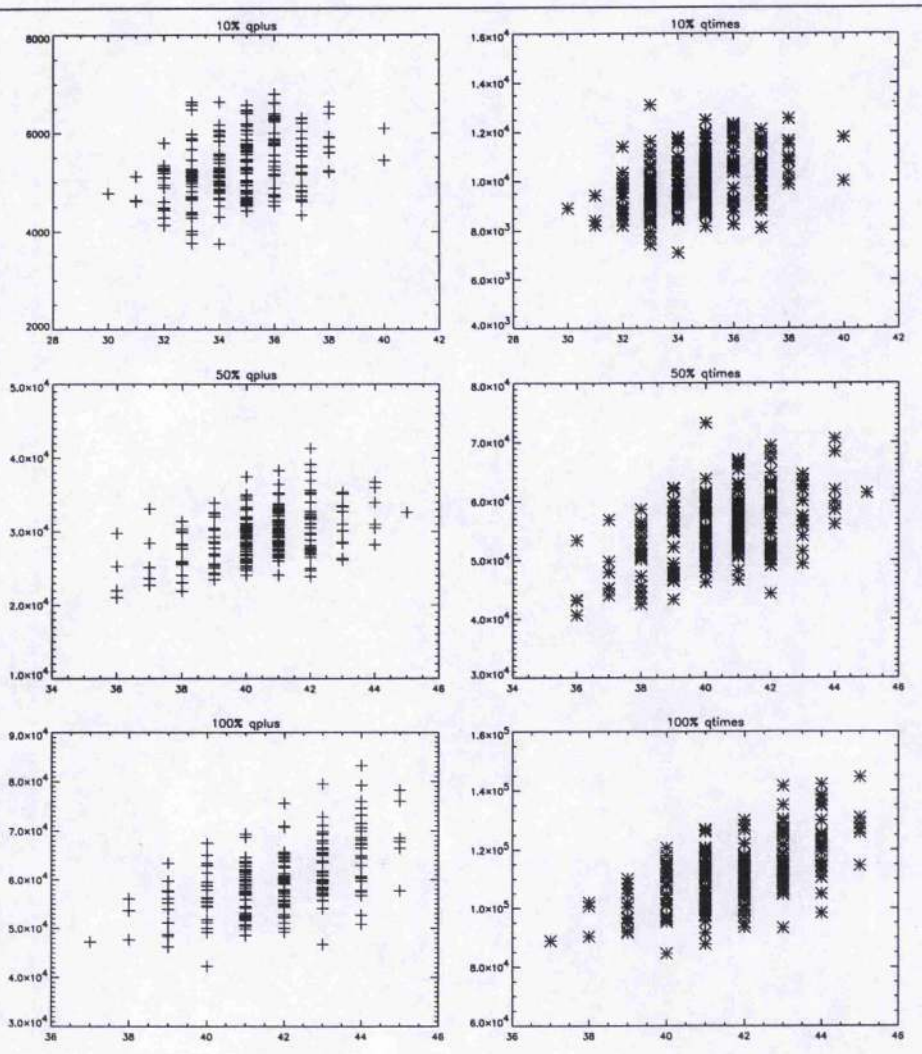


Figure 6.1: Scatter plots of 200 random runs
In each case the abscissa is path length, ordinate is $log_2(quality)$.

It appears from the generally rising trend of these pictures that algorithms which find short paths through the graph are likely to produce smaller Q. We will later examine various positional algorithms in this regard.

Distribution of Path Lengths

As we have seen from the experimental results in Figure 6.1 there appears to be some correlation between short paths and small solutions. One interesting question to investigate is the distribution of path lengths through $\Gamma_2(D)$. We can get an idea of the distribution

of path lengths through Γ_2(D) experimentally by randomly selecting paths and plotting the resulting distribution statistics. Note that we need to be slightly careful about what constitutes a valid step at each stage. Also should we 'lock' elements of the SNF as we discover them, or should we make completely random selections at each stage? In either case, of course, we need to ensure that each step would actually make some progress i.e. $d_x \not| d_y$ and $d_y \not| d_x$.

There are two possibilities of interest depending upon whether or not we detect and 'lock' SNF entries, removing them from further consideration, when we find them. Locking the SNF entries will typically produce slightly shorter paths as we saw in the discussion of example 6. In fact, in the worst case, it appears to make only a fairly small difference that does not affect the overall shape of the distribution at all significantly. We will therefore 'lock out' particular elements as by so doing we can produce much faster algorithms for our experiments.

Experimental distribution of path lengths from 10000 random runs through worst case diagonals for various lengths are shown as bar charts in figures 6.2 through 6.7 :

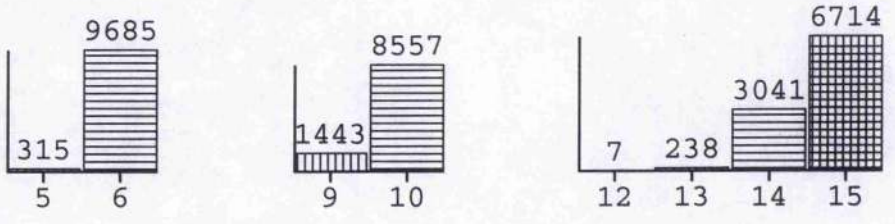


Figure 6.2 : Distribution of random path lengths in $\Gamma_2(W_4)$, $\Gamma_2(W_5)$ and $\Gamma_2(W_6)$.

From these smaller cases it appears that we are most likely to produce maximal length paths, with the number of paths of any given shorter length tailing off quickly, however :

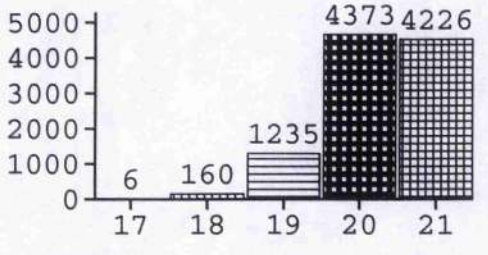


Figure 6.3 : Distribution of random path lengths in $\Gamma_2(W_7)$.

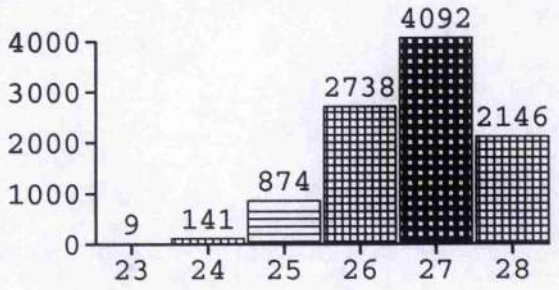


Figure 6.4 : Distribution of random path lengths in $\Gamma_2(W_8)$.

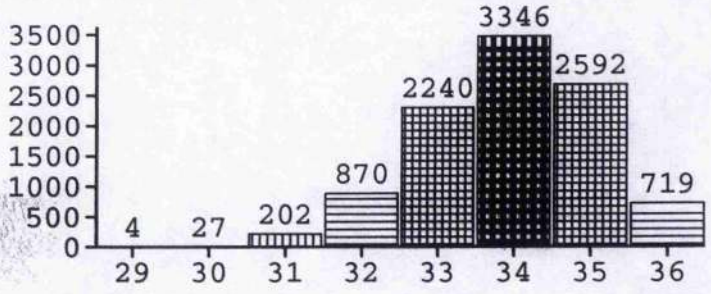


Figure 6.5 : Distribution of random path lengths in $\Gamma_2(W_9)$.

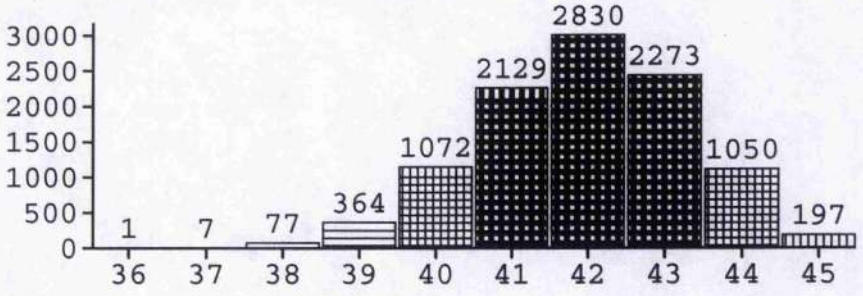


Figure 6.6 : Distribution of random path lengths in $\Gamma_2(W_{10})$.

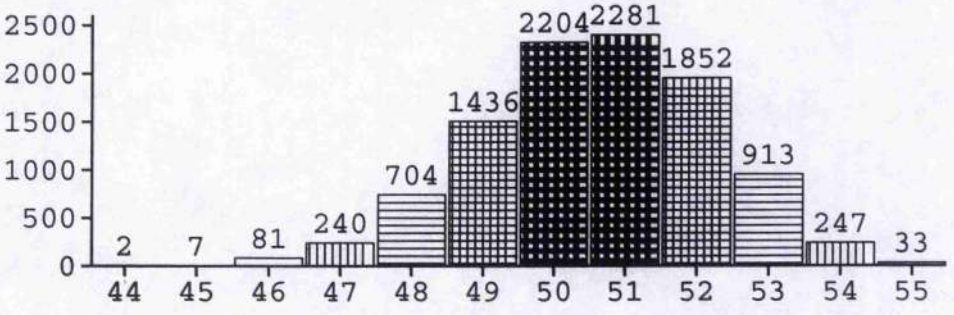


Figure 6.7 : Distribution of random path lengths in $\Gamma_2(W_{11})$.

There is a twist in the tale; it appears that this distribution of path lengths in $\Gamma_2(W_n)$ is actually a relatively smooth single peaked distribution with the maximum near, but not actually at, the longest possible length (i.e. $\binom{n}{2}$) for a given problem. The bell curve has quite a long tail to the left, i.e. the shorter paths. This is the area in which we would like to find solutions, but we would like to find a better way of doing this than performing several thousand random runs.

Studying the behaviour of this distribution may be an interesting area for future study, but since the heuristics we will develop find significantly shorter paths we will leave this as a potential area for further work.

6.1.1 Permutation of Input

Note that if we had applied the standard algorithm to the diagonal $[130, 231, 6, 143, 78]$ without presorting we would have produced exactly the same result as applying the mingcd structural algorithm. For 5 distinct elements there are 120 possible orderings. We plot solution size vs path length for applying the standard approach to each of these orderings in figure 6.8. As already mentioned this is a degenerate case, the 78 being in the SNF, and we plot solution size vs path length for each of the 24 permutations of 'reduced' input also in figure 6.8. Recall that the standard approach first performs operations on the sequence of pairs $[[1, 2], [1, 3], \ldots, [1, n]]$ to produce the gcd of all the entries in position 1. We could at this point attempt to examine all orderings of the n-1 remaining entries before embarking on each next major step. However this would then no longer be the standard algorithm as described. We will therefore only consider orderings of the initial entries when examining the effect of such orderings for any given positional algorithm. Clearly permutation of input can have a large effect and it would be beneficial to develop an algorithm that minimizes the effect of this ordering with respect to both the number of such orderings and the range of sizes associated. When discussing positional algorithms we will often refer to the range of qualities associated with them for a particular input. These ranges have been estimated from the results of several random permutations of input and while they may not actually correspond to the entire range they should be sufficient to give a good idea.

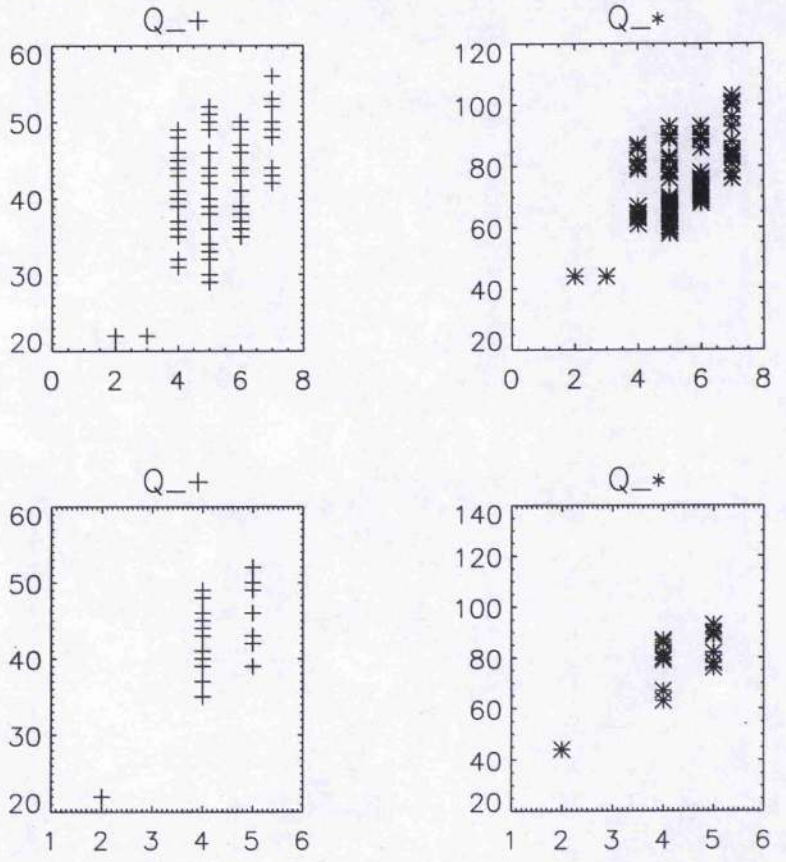


Figure 6.8: Scatter plots of results of applying standard positional algorithm :
Top : to all 120 orderings of [6, 78, 130, 143, 231].
Bottom : to all 24 orderings of [6, 130, 143, 231].
In each case the abscissa is path length, ordinate is $log_2(quality)$.

Most implementations sort the diagonal D into ascending order before applying the standard algorithm. These experiments show that the ordering of the input can have a significant effect on performance. We will investigate in section 6.2.1 whether sorting is actually beneficial in the standard implementation.

6.2 Positional Heuristics

The basic idea behind the positional algorithms is to develop a strategy that preselects a path (sequence of pairwise steps), P through the worst case graph, $\Gamma_2(W_n)$. Since

any other input D of length n has an associated graph, Γ_2(D), that is a quotient graph of $\Gamma_2(W_n)$ it follows that by applying the same sequence of steps, P, to D we will find a route through Γ_2(D). We will develop in this section 'better' positional algorithms for this task. By 'better', we mean heuristics which find shorter paths through $\Gamma_2(W_n)$, and which appear to generally produce better quality solutions in both the worst case and also for more general input diagonals. We will also provide some theoretical foundation that will help explain the behaviour of these heuristics, and that permits a more general comparison of positional algorithms.

6.2.1 The Standard Algorithm

The algorithm (sequence of steps / position pairs) that is usually implemented to convert a diagonal matrix to SNF is as shown in Figure 6.9.

```
1. for i in [1, ..., n − 1] do
2.         for j in [i + 1, ..., n] do
3.                 m[i][i]:=gcd, m[j][j]:=lcm.
4.         od;
5. od;
```

Figure 6.9 : Pseudocode for the standard algorithm.

This procedure takes up to $n-1$ steps to find the gcd of n numbers, another $n-2$ to find the gcd of the remaining $n-1$ numbers and so on. It is easy to see that in the worst case this takes $\sum_{i=1}^{n-1} = \frac{n(n-1)}{2}$ steps.

So here we repeatedly obtain gcds until the final step at which point we also obtain the lcm of all of the numbers. There are many variants of this algorithm, e.g. to produce Lcms, (naive-lcm) or to alternately produce some number of Gcds followed by some number of Lcms. In all cases the worst case still takes $\sum_{i=1}^{n-1} = \frac{n(n-1)}{2}$ steps.

To Sort or not to Sort

This procedure is obviously sensitive to input ordering. Exchanging d_1 and d_2 has no effect on later diagonals, but any other change may do so, giving $\frac{n!}{2}$ possible cases. Even

exchanging d_1 and d_2 will have an effect upon the multiplier matrices, but this can be made quite small by utilising the methods of chapter 5 to 'balance' the entries of the multipliers.

Examining the effect of applying the algorithm described above to various random diagonals (10, 20...100 percent of worst case) of various lengths it appears fairly clear that sorting into increasing order is unhelpful when we are applying the standard algorithm. Sorting into decreasing order on the other hand seems to generally produce better quality results. These effects are more pronounced the closer the problem is to worst case i.e. the larger the percentage of coprime elements that occur.

n	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
4	[NA,NA]	[NA,NA]	[25,28]	[25,30]	[16,16]	[8,14]	[8,14]	[16,17]	[1,2]	[0,0]
5	[21,21]	[12,16]	[19,19]	[10,14]	[4,7]	[7,11]	[2,7]	[3,8]	[2,2]	[0,0]
6	[19,25]	[12,18]	[4,9]	[4,5]	[1,5]	[3,8]	[2,5]	[2,3]	[0,2]	[0,0]
7	[7,17]	[2,12]	[1,4]	[1,8]	[0,3]	[2,3]	[0,1]	[1,4]	[0,1]	[0,0]
8	[4,11]	[0,5]	[2,5]	[0,0]	[0,0]	[1,2]	[1,2]	[0,0]	[0,0]	[0,0]
9	[0,5]	[0,2]	[0,3]	[0,0]	[1,1]	[0,0]	[0,0]	[0,0]	[0,0]	[0,0]
10	[0,0]	[0,0]	[0,1]	[0,0]	[0,0]	[0,0]	[0,0]	[0,0]	[0,0]	[0,0]

Table 6.1: Results of applying Standard algorithm to 1000 random diagonals, showing the percentage of cases where sorting produced a better quality result than reverse sorting - In each case the pair relates as $[Q_*, Q_+]$.

Note that the small percentages for $n = 4$ have not been given values. This is because the height of W_4 is only 14, and so 10 or 20% of this is too small for sensible comparison (The cases are so degenerate as to either be in SNF, or be within a single step).

In fact it appears that in these cases (10,20,...,100 % of W_n) sorting into increasing order generally gives a result that is comparatively bad whilst sorting into decreasing order generally gives a good quality result in comparison to other permutations of the input ordering. This of course depends heavily upon the exact nature of the problem, but the results shown in Tables 6.2 and 6.3 support this view. In each case, 10 random inputs (each k% of W_n) were created. 100 random permutations of each of these inputs were selected with the proviso that the sorted input and the reverse sorted input were included in those 100 permutations. The figures shown in the table are the sums of final positions of these ten runs of the sorted and reverse sorted inputs in terms of the final qualities achieved by applying the standard algorithm. (i.e. a 10 would imply that that permutation of inputs

produced a better final result in all 10 cases, a 1000 would imply every other permutation was better.) In the case of the length 4 input, all 24 permutations were selected, rather than 100 random. We give only the results for $\mathcal{Q}_*$ as $\mathcal{Q}_+$ shows almost identical behaviour.

n	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
4 (240 runs)	NA	NA	117	169	183	209	183	146	217	180
5 (1000 runs)	289	330	729	696	694	609	856	803	850	976
6 (1000 runs)	483	563	803	830	866	842	868	807	885	995
7 (1000 runs)	584	860	768	869	821	891	951	912	935	993
8 (1000 runs)	771	885	940	972	901	929	869	966	950	990

Table 6.2: Results of applying standard algorithm to 100 different sortings of 10 inputs. The value shown is the sum of the positions in which the 'sorted' input placed (in terms of $\mathcal{Q}_*$). i.e. 10= first place every time, 1000=last place each run

n	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
4 (240 runs)	NA	NA	82	67	68	64	47	75	56	30
5 (1000 runs)	179	201	118	178	342	182	199	94	84	60
6 (1000 runs)	284	198	201	163	152	115	157	110	135	11
7 (1000 runs)	137	60	61	183	175	32	98	173	117	17
8 (1000 runs)	86	38	33	47	82	62	66	37	28	12

Table 6.3: Results of applying standard algorithm to 100 different sortings of 10 inputs. The value shown is the sum of the positions in which the 'reverse sorted' input placed (in terms of $\mathcal{Q}_*$). i.e. 10= first place every time, 1000=last place each run

The experimental evidence from tables 6.2 and 6.3 definitely appears to show that sorting into reverse order is not only better than sorting into increasing order, but also that it appears to generally produce results that are in the top quartile. Sorting into increasing order on the other hand generally appears to produce results that are deep in the fourth quartile of quality.

Another case we are interested in that is not really covered by the above is the coprime case, with height (and weight) equal to n. This case arises relatively often in practice and

as we can see from the data in table 6.4 it appears that here sorting into increasing order generally produces better results than sorting into decreasing order.

n	3	4	5	6	7	8	9	10
Q_*	72	89	97	98	99	100	100	100
Q_+	75	91	98	99	99	100	100	100

Table 6.4: Results of applying Standard algorithm to 1000 random diagonals of coprime elements, showing the percentage of cases where sorting produced a better quality result than reverse sorting.

In fact we can also see that in this case sorting into increasing order performs very well in comparison to other random orderings, whereas sorting into decreasing order appears to be a particularly bad way of proceeding. Table 6.5 shows the results of summing the final positions of these two sortings in comparison with 98 random sortings, over 100 runs of random coprime inputs.

n	3	4	5	6	7	8	9	10
range (00)	1-6	1-24	1-100	1-100	1-100	1-100	1-100	1-100
Increasing	208	528	1523	939	712	390	269	148
Decreasing	486	2060	9147	9522	9787	9859	9962	9967

Table 6.5: Results of applying Standard algorithm to 100 random diagonals of coprime elements, showing the sum of positions over 100 runs wrt Q_* i.e. 100-always best, 10000-always worst.

This behaviour becomes more exaggerated the greater the length of the input. For length 20 lists of coprime elements sorting into increasing order appears to generally produce solutions with qualities in the top percentile of all qualities for input orderings under application of the standard algorithm. It also appears that, for lists of this length, sorting into decreasing order generally produces solutions with qualities in the bottom percentile.

It appears then that there is some relation between the height of the input and whether sorting is beneficial. In fact the relation appears to be somewhat more complex as if we examine the the complement of the coprime case, i.e. where every element appears in $n-1$ positions (hence Height n, Weight $n(n-1)$), it appears, from table 6.6, to benefit from sorting into decreasing order. And we can again examine how these orderings

n	3	4	5	6	7	8	9	10
Q_*	18	7	2	1	0	0	0	0
Q_+	19	8	2	1	0	0	0	0

Table 6.6: Results of applying Standard algorithm to the complement of 1000 random diagonals (complement of coprime), showing the percentage of cases where sorting produced a better quality result than reverse sorting.

perform in comparison to other random orderings, as shown in Table 6.7. In this case the opposite result of that in the coprime case is true, and for the longer inputs here we would recommend, in virtually all circumstances, sorting into decreasing order. There is

n	3	4	5	6	7	8	9	10
range (00)	1-6	1-24	1-100	1-100	1-100	1-100	1-100	1-100
Increasing	457	1967	8911	9355	9514	9757	9896	9929
Decreasing	241	549	1242	697	505	316	216	173

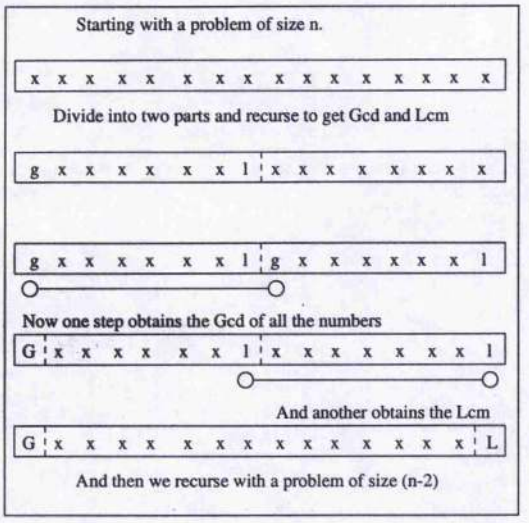
Table 6.7: Results of applying Standard algorithm to 100 random diagonals (complement of coprime), showing the sum of positions over 100 runs wrt Q_* i.e. 100-always best, 10000-always worst.

presumably some connection between height / weight and how best to sort our input. In the absence of a better strategy we suggest that whilst sorting into reverse order appears to be the best strategy in the general case, and sorting into increasing order appears best in the coprime case, it is probably worthwhile trying both orderings in each case. In fact we suggest trying a small number of pairs of random orderings and their reverses in order to attempt to find a good quality solution. This approach will then require only a linear increase in the amount of work to be performed, which amount can then be decided depending upon whatever external criteria are most pressing.

During the rest of this thesis, when we will be comparing the results of applying various algorithms to some given input, we will not be attempting to select the best way of sorting the input. Rather we will generally take several random permutations of the input in order to try and minimize the effects of such input ordering upon our wider comparisons.

6.2.2 Divide and Conquer

If we examine the case $n = 2$, we see that we obtain both the gcd and lcm in a single step i.e. we get the lcm 'for free'. An immediate question is whether we can develop a better algorithm by obtaining both the gcd and lcm of a set of n numbers for $n > 2$. One approach that lends itself to obtaining both gcd and lcm is divide and conquer.



Split the problem into two roughly equal parts and solve recursively as shown on the left. The base case with $n = 2$ is the single step below

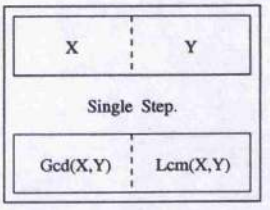


Figure 6.10 : Basic Divide and Conquer Strategy

We can see that we gain by applying this divide and conquer strategy simply by examining the problem of length 4.The naive method can take $\binom{n}{2} = 6$ steps when $n = 4$. The divide and conquer method can be easily seen to take no more than 5 steps to solve the size 4 problem by using the sequence $[[1,2],[3,4],[1,3],[2,4],[2,3]]$ as demonstrated in figure 6.11. Exhaustive enumeration of all 708 paths through $\Gamma_2(W_4)$ where W_4 is a worst case diagonal of length 4 shows that 12 paths have length 5 and 696 paths have length 6. Similar enumeration has shown there to be a total of 3269040 paths through $\Gamma_2(W_5)$, the graph of the length 5 worst case. Space and time constraints have however prevented us from calculating the exact distribution of path lengths in either this or larger worst cases.

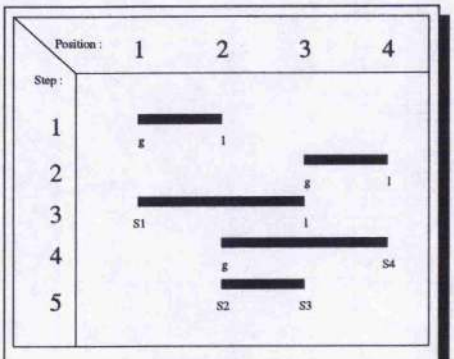


Figure 6.11

We can write down a recursive formula for the maximum number of steps the divide and conquer algorithm could take. As demonstrated in Figure 6.10 this problem has two stages, the recursion to obtain the gcd and the lcm of a length n list of integers and then the overall recursion with a problem of size n-2 to find the SNF. This can be modeled using two linked recursive formulae as follows :

- Maximum number of steps to obtain gcd and lcm of n integers,

$$S_n := S_{\lfloor \frac{n}{2} \rfloor} + S_{\lceil \frac{n}{2} \rceil} + 2.$$

- Maximum number of steps to find SNF of n integers,

$$T_n := T_{n-2} + S_n.$$

With base cases $S_1 = 0, S_2 = 1, T_1 = 0, T_2 = 1$.

And so we have growth as shown in table 6.8:

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S_n	0	1	3	4	6	8	9	10	12	14	16	18	19	20	21	22
T_n	0	1	3	5	9	13	18	23	30	37	46	55	65	75	86	97

Table 6.8: Number of steps required by divide (into 2) and conquer in worst case

It is not particularly illuminating to derive an explicit formula for T_n since the the exact behaviour relies upon the binary representation of n. However we can make some useful observations. Figure 6.12 shows a plot of $2 \times S_n$ (dashes) and a plot of $3 \times n$ (solid).

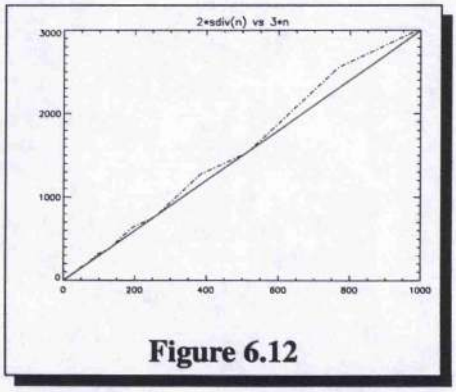


Figure 6.12

From this it appears that S_n is roughly $\frac{3n}{2} + o(log(n))$. This observation is further borne out by the fact that for $n = 2^k$ the recursion to obtain the gcd and lcm at each stage is

$$S_{2^k} = 2 \times S_{2^{k-1}} + 2$$

from which we can readily derive the explicit formula

$$S_{2^k} = 3 \times 2^{k-1} - 2,$$

i.e. S_n is $O(n)$. Hence it appears that T_n is $O(n^2)$.

This prompts us to investigate whether we can find a better divide and conquer algorithm. We assume that the basic premise of recursively finding the gcd and lcm of our list of length $n, n-2, n-4, \ldots, 2$ or 1 cannot be improved upon. We therefore concentrate upon the problem of finding both the gcd and lcm of a list of length n. Note that on the length 6 problem to find the gcd and lcm we split the problem into a pair of problems of size 3, both of which take 3 steps, for 3+3+2=8 steps overall. If instead we split the problem into 3 subproblems each of size 2, taking one step each, then we will require 4 further steps to collect the gcd (2 steps) and lcm (2 steps) for a total of 7 steps in all.

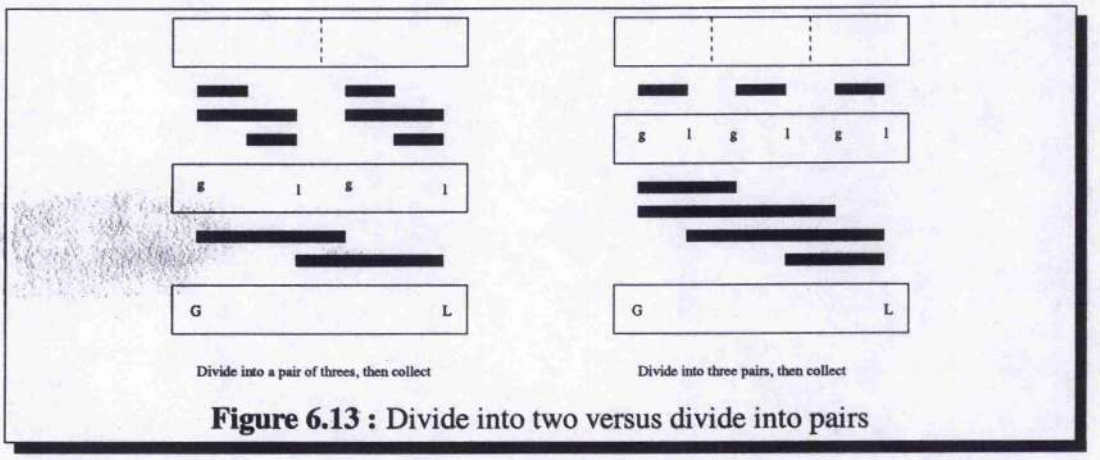


Figure 6.13 : Divide into two versus divide into pairs

Note that this approach splits the gcd-collection and the lcm-collection completely, that is to say after the gcd-lcm operation on each pair, we cannot damage any information we will later require. For this reason this approach appears to have promise. In fact upon examining the various 'divide into k parts and conquer' algorithms we find that the non-recursive variant which divides into $\lceil \frac{n}{2} \rceil$ sub problems of size no more than 2 never takes more steps than any other divide and conquer heuristic.

This 'divide into pairs' algorithm can be analysed by observing that we can set up two linked recursive formulae as follows:

- Maximum number of steps required to obtain gcd and lcm of n integers,

$$S_n := \left\lfloor \frac{n}{2} \right\rfloor + 2(\left\lceil \frac{n}{2} \right\rceil - 1)$$

- Maximum number of steps find SNF of n integers,

$$T_n := T_{n-2} + S_n.$$

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S_n	0	1	3	4	6	7	9	10	12	13	15	16	18	19	21	22
T_n	0	1	3	5	9	12	18	22	30	35	45	51	63	70	84	92

Table 6.9: Number of steps required by divide (into pairs) and conquer in worst case

With base cases $T_1 = 0, T_2 = 1$.

And so we have growth as shown in table 6.9:

For this recursion we can readily derive an explicit formula : note that there are $\lfloor \frac{n}{2} \rfloor$ pairs and then the 'collection' of the gcds to one point and the lcms to another is easily seen to require no more than $2(\lceil \frac{n}{2} \rceil - 1)$ operations.

$$S_n := \left\lfloor \frac{n}{2} \right\rfloor + 2\left(\left\lceil \frac{n}{2} \right\rceil - 1\right), \qquad T_n := T_{n-2} + S_n.$$

For odd n,

$$S_n := \frac{3(n-1)}{2}$$

Let $n := 2m + 1$,

$$T_{2m+1} := \sum_{i=0}^{m} (3i)$$
$$= \frac{3m(m+1)}{2}$$

Now $m = \frac{n-1}{2}$ so

$$T_n := \frac{3(n-1)(n+1)}{8} = \frac{3n^2 - 3}{8}.$$

For even n,

$$S_n := \frac{3n}{2} - 2$$

Let $n := 2m + 2$,

$$T_{2m+2} := \left(\sum_{i=0}^{m} (3i)\right) + m + 1$$
$$= \frac{3m(m+1)}{2} + m + 1$$

Now $m = \frac{n-2}{2}$ so

$$T_n := \frac{3(n)(n-2)}{8} + \frac{n}{2} = \frac{3n^2 - 2n}{8}.$$

And so in general we have

$$T_n = \begin{cases} \frac{3n^2-3}{8} & \text{n odd} \\ \frac{3n^2-2n}{8} & \text{n even} \end{cases} \tag{6.1}$$

This shows us that the divide into pairs allows us to solve the problem in under $\frac{3n^2}{8}$ operations. Recall that the plot in Figure 6.12 shows that standard divide and conquer will generally take a little over $\frac{3n^2}{8}$ operations. This difference also arises in the more general 'divide into k' and conquer heuristics. Note that the 'obtain Gcd-Lcm' sequence i.e S_n for the 'divide into k' and conquer is actually as good as the 'divide into pairs' when the problem size $n = 2k^i$ for some i. In this case, all the base cases are of size 2. However since we then recurse again but with a problem of size $n-2$ we can see that divide into pairs will perform better for problems of size greater than 4.

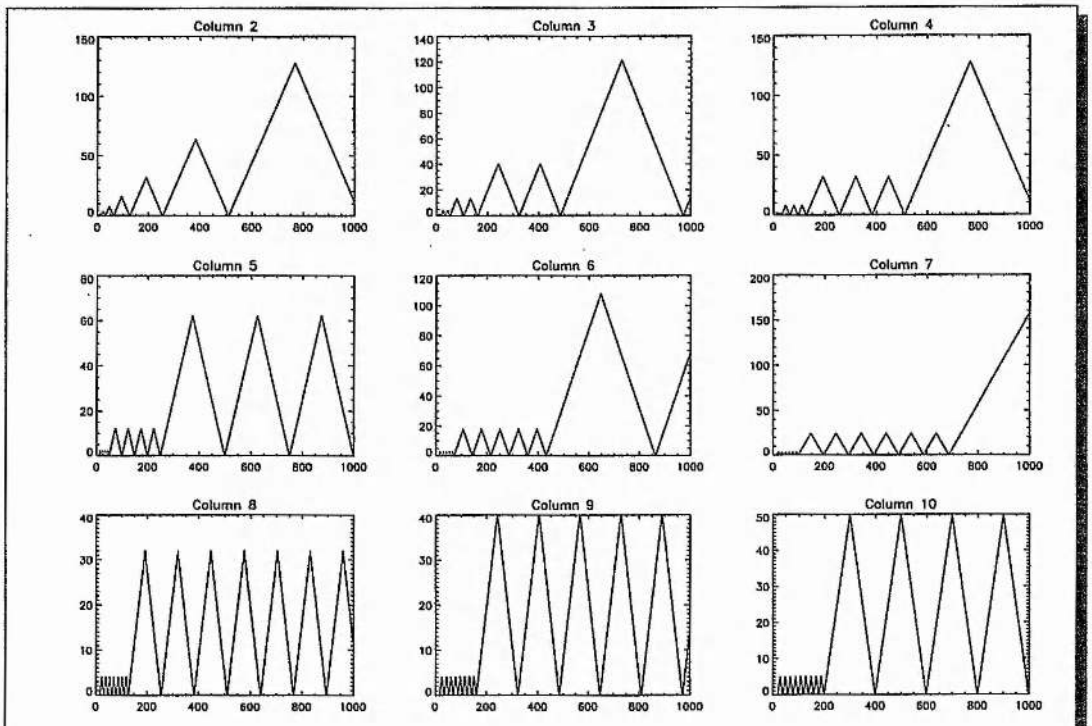


Figure 6.14 : (# steps for divide into k parts) - (# steps for divide into pairs)

Figure 6.14 demonstrates the difference between the 'divide into k parts' heuristics and the 'divide into pairs' heuristic. Each plot demonstrates similar behaviour, i.e. a sawtooth, with sequences of k-1 peaks each of the same height. For $k = 2$ the heights of each successive peak are 2^n, for $k = 3$ the height of each successive pair of peaks appears to

be $\sum_{i=0}^{n} 3^i$, but for larger k there seems to be no readily accessible formula. It is also the case that for larger k the problem of how to perform the base cases rears it's head again, and we need to resort to dividing into smaller k to make further progress.

Having settled upon this strategy of dividing into pairs, collecting up the Gcd and Lcm and recursing, we need to address the problem of exactly how we intend to perform the steps corresponding to the collection of the Gcd and Lcm. We will concentrate on the Gcd collection for the moment. The worst case that can arise for this collection is very similar to the coprime case; we require n-1 operations to obtain the Gcd of n elements. The question is how exactly we perform these n-1 operations. There are many ways of performing these operations (see Section 4.5 for the discussion of the digraph of a coprime selection, and path selection therein) and we shall develop a solution to this problem after developing a little more theory in the next section

6.3 Power Growth

In chapter 5 we looked at how to minimize the impact of each step by looking at the 'damage' that was done by the multiplier matrices for that step. We will now examine another aspect of this 'damage' accumulation.

Recall from sections 4.1.2 and 5.2 that each step through the graph from initial input to SNF consists of pre(post) multiplication of the current row(column) transformation matrices by a multiplier of the form:

$$M = \begin{bmatrix} 1 & 0 & \cdots & & 0 \\ 0 & 1 & & & \\ & & M_{i,i} & M_{j,i} & \\ \vdots & & & 1 & \\ & & M_{i,j} & M_{j,j} & \\ 0 & & & & 1 \end{bmatrix}$$

Note that premultiplication of the current row transformation matrix by such an almost elementary matrix will cause changes to the entries in only two rows, namely rows i and j, of the row transformation matrix. Similarly postmultiplication of the column transformation matrix will cause changes to the entries in only columns i and j of the column transformation matrix.

We could easily keep track of how often each row/column of our evolving row/column transformation matrices have been modified at step η as we progress toward SNF by using a length n vector ${}^{\eta}B$, where each entry ${}^{\eta}B_i, 1 \leq i \leq n$ is initially set to be zero. At each step of the calculation, the only difference between ${}^{\eta}B$ and ${}^{\eta+1}B_i$ would be:

$$ {}^{\eta+1}B_{[i,j]} = {}^{\eta} B_{[i,j]} + [1,1]. $$

This however would not give a good indication of the way in which the magnitudes of the entries of the transformation matrices are likely to change as we progress toward SNF.

A better way of tracking the potential changes to the magnitudes of the entries in each row/column of the transformation matrices is to recall from chapter 5 that we can easily find an upper bound X for the magnitude of every entry in all almost elementary multiplier matrices used in a given SNF calculation. Similarly, at each stage of the calculation we can see that for each row/column of the row/column multiplier matrices we can find some power of X that must be an upper bound for the magnitude of the entries.

We can track these maximum bound powers in a similar manner as described above, i.e. with a sequence of length n vectors ${}^{\eta}B$, relating to step η of the calculation.. In this case it should be clear that, at each step, $\eta + 1$, the potentially largest power of X in each row/column i, j is equal to

$$ Max({}^{\eta}B_i, {}^{\eta} B_j) + 1 $$

i.e. the only difference between ${}^{\eta}B$ and ${}^{\eta+1}B$ would be

$$ {}^{\eta+1}B_{[i,j]} = [Max({}^{\eta}B_i, {}^{\eta} B_j) + 1, Max({}^{\eta}B_i, {}^{\eta} B_j) + 1]. $$

In the rest of this section we will examine the vectors that arise from applying this idea to the positional algorithms discussed previously in this chapter. We will then be able to compare the theoretical maximum power growth (final vector) with those results obtained in practice. Appendix 1 contains a graphical demonstration of this idea, whereby we shade the powers in each position at each stage (darker shading for higher values).

6.3.1 Maximum Powers for Particular Heuristics

We will now examine several heuristic methods in terms of this idea. There are two things which are of interest. The maximum power occurring in any given row or column, and the final distribution of maximum powers across all the rows or columns. We shall demonstrate the ideas in this section upon a length 8 problem. In Appendix 1 the results of applying these ideas to a length 20 problem are demonstrated graphically.

Standard Implementation

We shall begin by examining the usually implemented simple heuristic and tracking the changes in the bounding power of X as we proceed through the steps

$$[1,2],[1,3],\ldots,[1,n],[2,3],[2,4],\ldots,[2,n],\ldots,[n-1,n].$$

```
[ 0,  0,  0,  0,  0,  0,  0,  0 ],
[ 1,  1,  0,  0,  0,  0,  0,  0 ],
[ 2,  1,  2,  0,  0,  0,  0,  0 ],
[ 3,  1,  2,  3,  0,  0,  0,  0 ],
[ 4,  1,  2,  3,  4,  0,  0,  0 ],
[ 5,  1,  2,  3,  4,  5,  0,  0 ],
[ 6,  1,  2,  3,  4,  5,  6,  0 ],
[ 7,  1,  2,  3,  4,  5,  6,  7 ],
```

After this first sequence we have now guaranteed that the gcd of all the elements appears in position 1.

```
[ 7,  3,  3,  3,  4,  5,  6,  7 ],
[ 7,  4,  3,  4,  4,  5,  6,  7 ],
[ 7,  5,  3,  4,  5,  5,  6,  7 ],
[ 7,  6,  3,  4,  5,  6,  6,  7 ],
[ 7,  7,  3,  4,  5,  6,  7,  7 ],
[ 7,  8,  3,  4,  5,  6,  7,  8 ],
```

We now have the gcd of the next 7 elements in position 2. One interesting thing to note is that this sequence of positions automatically ran through using the pair of positions with the smallest powers at each step. This means that the largest power, i.e. the one appearing in position 8 (here, or position n in general) only increases by 1 for each successive sequence after the first. This style of increase then continues in a similar vein :

```
[  7,  8,  5,  5,  5,  6,  7,  8 ],
[  7,  8,  6,  5,  6,  6,  7,  8 ],
[  7,  8,  7,  5,  6,  7,  7,  8 ],
[  7,  8,  8,  5,  6,  7,  8,  8 ],
[  7,  8,  9,  5,  6,  7,  8,  9 ],
[  7,  8,  9,  7,  7,  7,  8,  9 ],
[  7,  8,  9,  8,  7,  8,  8,  9 ],
[  7,  8,  9,  9,  7,  8,  9,  9 ],
[  7,  8,  9, 10,  7,  8,  9, 10 ],
[  7,  8,  9, 10,  9,  9,  9, 10 ],
[  7,  8,  9, 10, 10,  9, 10, 10 ],
[  7,  8,  9, 10, 11,  9, 10, 11 ],
[  7,  8,  9, 10, 11, 11, 11, 11 ],
[  7,  8,  9, 10, 11, 12, 11, 12 ],
[  7,  8,  9, 10, 11, 12, 13, 13 ]
```

We can see that this has a very simple structure for the final distribution of bounding powers. On a length n diagonal the final powers are :

$$[n-1, n, \ldots, 2n-4, 2n-3, 2n-3].$$

It is similarly obvious that the maximum power here for a length n diagonal is :

$$2 * n - 3.$$

So the standard implementation actually seems to do quite well, producing a maximum power of $O(n)$.

Although possibly a slightly odd metric to take we can readily examine the sum of the final powers. This will hopefully give us some indication of the spread. In this case it is easy to see that the sum of the final powers is

$$2n - 3 + \sum_{i=n-1}^{2n-3} i = \frac{1}{2}(3n^2 - 3n - 2).$$

We now have a standard benchmark against which we can compare various other heuristics.

A Worst Case Strategy

It is easy to see that we can develop a worst case strategy that will produce the largest possible powers by ensuring that at each step, one of the (at least two) entries with maximum current bounding power is used. This is achieved by the following sequence

```
[ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 1, 5 ], [ 1, 6 ], [ 1, 7 ], [ 1, 8 ],
[ 2, 8 ], [ 2, 7 ], [ 2, 6 ], [ 2, 5 ], [ 2, 4 ], [ 2, 3 ],
[ 3, 4 ], [ 3, 5 ], [ 3, 6 ], [ 3, 7 ], [ 3, 8 ],
[ 4, 8 ], [ 4, 7 ], [ 4, 6 ], [ 4, 5 ],
[ 5, 6 ], [ 5, 7 ], [ 5, 8 ],
[ 6, 8 ], [ 6, 7 ],
[ 7, 8 ]
```

Here we are effectively chasing the gcd up and down.

```
[  0,  0,  0,  0,  0,  0,  0,  0 ],
[  1,  1,  0,  0,  0,  0,  0,  0 ],
[  2,  1,  2,  0,  0,  0,  0,  0 ],
[  3,  1,  2,  3,  0,  0,  0,  0 ],
[  4,  1,  2,  3,  4,  0,  0,  0 ],
[  5,  1,  2,  3,  4,  5,  0,  0 ],
[  6,  1,  2,  3,  4,  5,  6,  0 ],
[  7,  1,  2,  3,  4,  5,  6,  7 ],
```

After this first sequence we are in exactly the same position as we were with the previous naive strategy.

```
[  7,  8,  2,  3,  4,  5,  6,  8 ],
[  7,  9,  2,  3,  4,  5,  9,  8 ],
[  7, 10,  2,  3,  4, 10,  9,  8 ],
[  7, 11,  2,  3, 11, 10,  9,  8 ],
[  7, 12,  2, 12, 11, 10,  9,  8 ],
[  7, 13, 13, 12, 11, 10,  9,  8 ],
```

We now have the first two entries of the SNF once again. However we can see that the sequence uses the entries with the largest powers of the bound at each step.

```
[  7, 13, 14, 14, 11, 10,  9,  8 ],
[  7, 13, 15, 14, 15, 10,  9,  8 ],
[  7, 13, 16, 14, 15, 16,  9,  8 ],
[  7, 13, 17, 14, 15, 16, 17,  8 ],
[  7, 13, 18, 14, 15, 16, 17, 18 ],
[  7, 13, 18, 19, 15, 16, 17, 19 ],
[  7, 13, 18, 20, 15, 16, 20, 19 ],
[  7, 13, 18, 21, 15, 21, 20, 19 ],
[  7, 13, 18, 22, 22, 21, 20, 19 ],
[  7, 13, 18, 22, 23, 23, 20, 19 ],
[  7, 13, 18, 22, 24, 23, 24, 19 ],
[  7, 13, 18, 22, 25, 23, 24, 25 ],
[  7, 13, 18, 22, 25, 26, 24, 26 ],
[  7, 13, 18, 22, 25, 27, 27, 26 ],
[  7, 13, 18, 22, 25, 27, 28, 28 ]
```

For this worst case strategy the final powers are the worst possible obtainable, being :

$$[n-1, (n-1)+(n-2), \ldots, \sum_{i=1}^{n-1} i, \sum_{i=0}^{n-1} i],$$

where the maximum power after obtaining the kth element of the SNF is given by :

$$\sum_{i=n-k}^{n-1} i$$

and the maximum bound power occurring overall for the length n problem is then :

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}.$$

The sum of the final powers appearing, for the length n problem, is the cubic

$$\frac{1}{3}(n-1)n(n+1).$$

Standard Divide and Conquer

The next heuristic we will examine is the standard divide into two similar sized parts and recurse.

```
[  0,  0,  0,  0,  0,  0,  0,  0 ],
[  1,  1,  0,  0,  0,  0,  0,  0 ],
[  1,  1,  1,  1,  0,  0,  0,  0 ],
[  2,  1,  2,  1,  0,  0,  0,  0 ],
[  2,  2,  2,  2,  0,  0,  0,  0 ],
```

As we can see here, first we work down one half, then the other.

```
[  2,  2,  2,  2,  1,  1,  0,  0 ],
[  2,  2,  2,  2,  1,  1,  1,  1 ],
[  2,  2,  2,  2,  2,  1,  2,  1 ],
[  2,  2,  2,  2,  2,  2,  2,  2 ],
```

And then we collect the gcd and the lcm.

```
[  3,  2,  2,  2,  3,  2,  2,  2 ],
[  3,  2,  2,  3,  3,  2,  2,  3 ],
```

We now recurse with the problem of size n-2. Note that we have managed to find multipliers such that the rows (or columns) corresponding to the first and last entries of the SNF are bound by $log_2(n)$.

```
[  3,  3,  3,  3,  3,  2,  2,  3 ],
[  3,  4,  3,  4,  3,  2,  2,  3 ],
[  3,  4,  5,  5,  3,  2,  2,  3 ],
[  3,  4,  5,  5,  4,  4,  2,  3 ], *
[  3,  4,  5,  5,  5,  4,  5,  3 ], *
[  3,  4,  5,  5,  5,  6,  6,  3 ], *
[  3,  6,  5,  5,  6,  6,  6,  3 ],
[  3,  6,  5,  7,  6,  6,  7,  3 ],
[  3,  6,  8,  8,  6,  6,  7,  3 ],
[  3,  6,  8,  8,  7,  7,  7,  3 ],
[  3,  6,  9,  8,  9,  7,  7,  3 ],
[  3,  6,  9,  9,  9,  9,  7,  3 ],
[  3,  6,  9, 10, 10,  9,  7,  3 ]
```

It is interesting to note that we could improve these slightly by being a little more careful about the order in which we do some of these steps. For instance the starred steps above ignore the fact that if we wish to find the SNF of a length 3 problem where the powers are initially bound by [3,2,2] then we should use positions 2 and 3 for the first of the three moves, which will then lead to the final powers [4,5,5] rather than the [5,6,6] demonstrated above. This is not at all simple to check for generally, however we will make a note of this for later.

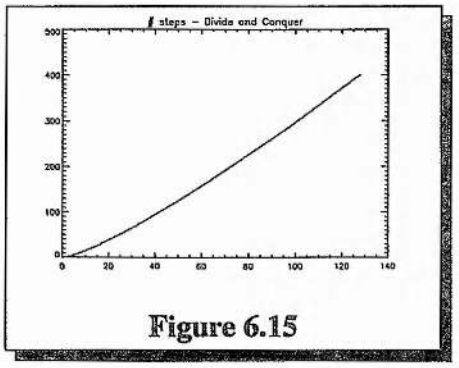


Figure 6.15

Again we note that that exact behaviour of divide and conquer algorithms is difficult to analyse and depends upon the binary representation of n. We can however make some reasonable estimates and assuming we simply apply the basic divide and conquer procedure we find that the maximum powers in any row (or column) increase as shown in Figure 6.15. This seems to suggest that the maximum power occurring in the length n problem is at least $O(n)$, and if we examine the values for various powers of 2, we find that the ratio of $\frac{maxpower(2^k)}{2^k}$ is approximately $\frac{k-1}{2}$. a simple calculation shows that the difference between successive ratios of $\frac{maxpower(2^k)}{2^k}$ appears to be tending towards $\frac{1}{2}$. It appears therefore that the maximum power occurring in the length n problem is governed by an equation of $O(nlogn)$. Similarly it seems that the sum of the final powers is at least $O(n^2)$, but no worse than $O(n^2logn)$.

It appears then that divide and conquer, despite producing shorter paths, is in some sense asymptotically worse than the standard implementation. Divide and conquer produces smaller maximum powers for $n < 16$ however (and smaller sums of powers for $n < 52$) and when we recall that a worst case problem of length n requires each element to be the product of $2^{n-1} - 1$ coprime elements we doubt that many problems will arise where divide and conquer is not competitive with the standard algorithm.

Divide into Pairs with Naive Collection Strategy

As we have seen the divide into pairs heuristic produces generally shorter paths in the worst case than any other heuristic. We will now investigate whether these shorter paths lead to the same powergrowth problems as with standard divide and conquer. One additional problem is of course that there was no indication of how best to perform the gcd-lcm collection step. We shall begin by examining a simple strategy to see the distribution of the powers.

```
[  0,  0,  0,  0,  0,  0,  0,  0 ],
[  1,  1,  0,  0,  0,  0,  0,  0 ],
[  1,  1,  1,  1,  0,  0,  0,  0 ],
[  1,  1,  1,  1,  1,  1,  0,  0 ],
[  1,  1,  1,  1,  1,  1,  1,  1 ],
```

at this point we need to collect the gcds and lcms. These two operations will not interfere with each other in any way. We collect both using a simple chasing strategy i.e. $[\ldots[k, k+2], [k+2, k+4]\ldots]$. We also implemented a standard style strategy i.e. $[[1,3],[1,5],[1,7],\ldots$ and discovered that the results were almost identical.

```
[  1,  1,  1,  1,  2,  1,  2,  1 ],
[  1,  1,  3,  1,  3,  1,  2,  1 ],
[  4,  1,  4,  1,  3,  1,  2,  1 ],
[  4,  2,  4,  2,  3,  1,  2,  1 ],
[  4,  2,  4,  3,  3,  3,  2,  1 ],
[  4,  2,  4,  3,  3,  4,  2,  4 ],
```

We have now reached a point such that we have the first and last entries of the SNF and we can recurse with a problem of size n-2. Note that we have managed to find multipliers such that the rows (or columns) corresponding to the first and last entries of the SNF are bound by $\lceil \frac{n}{2} \rceil$ which is clearly not as good as the bound we discovered for the standard divide and conquer.

```
[  4,  5,  5,  3,  3,  4,  2,  4 ],
[  4,  5,  5,  4,  4,  4,  2,  4 ],
[  4,  5,  5,  4,  4,  5,  5,  4 ],
[  4,  5,  5,  6,  4,  6,  5,  4 ],
[  4,  7,  5,  7,  4,  6,  5,  4 ],
[  4,  7,  6,  7,  6,  6,  5,  4 ],
[  4,  7,  6,  7,  7,  6,  7,  4 ],
[  4,  7,  8,  8,  7,  6,  7,  4 ],
[  4,  7,  8,  8,  8,  8,  7,  4 ],
[  4,  7,  9,  8,  9,  8,  7,  4 ],
[  4,  7,  9,  9,  9,  9,  7,  4 ],
[  4,  7,  9, 10, 10,  9,  7,  4 ]
```

Interestingly, after our apparently poor start, we have not done that much worse than for the standard divide and conquer. However it is now much easier to see where potential improvements can be made. Simply sorting the positions with regards to the power after each collection step leads to a small improvement ([4,6,8,9,9,8,6,4]). We shall utilise this later. Without this improvement however we find that the maximum power for a length n problem is simply

$$\sum_{i=1}^{\lceil \frac{n}{2} \rceil} i.$$

Which is approximately $\frac{n^2}{8}$ so it appears that we have to be very careful with our gcd/lcm collection routine.

Divide into Pairs with Intelligent Collection Strategy

Each pass to extract a gcd or lcm does an uneven amount of damage. The refinement we now consider is to attempt to ensure that at each step the most damage is done where there was least previous damage so that the largest power grows slowly and the damage is more evenly distributed. We do remarkably well by using a simple 'greedy algorithm' wherein we keep track of the powergrowth bound as we proceed, and at each step we are faced with a choice as to which position to use we select the position with the least power. This is as close as we can come to the adaptive (structural) heuristic which would select the position based upon the current state of the multipliers. We then see powergrowth increasing as :

```
[  0,  0,  0,  0,  0,  0,  0,  0 ],
[  1,  1,  0,  0,  0,  0,  0,  0 ],
[  1,  1,  1,  1,  0,  0,  0,  0 ],
[  1,  1,  1,  1,  1,  1,  0,  0 ],
[  1,  1,  1,  1,  1,  1,  1,  1 ],
```

```
[ 2, 1, 2, 1, 1, 1, 1, 1 ],
[ 2, 1, 2, 1, 2, 1, 2, 1 ],
```

Here we see the first implications of this strategy. We now need to use the pair [1,5] in order to collect the overall gcd.

```
[ 3, 1, 2, 1, 3, 1, 2, 1 ],
[ 3, 2, 2, 2, 3, 1, 2, 1 ],
[ 3, 2, 2, 2, 3, 2, 2, 2 ],
[ 3, 2, 2, 3, 3, 2, 2, 3 ],
```

We have now, once again, reached a stage where we have the first and last entries of the SNF and can recurse with a problem of size n-2. We have also managed to find multipliers such that the bounding powers in the rows (or columns) corresponding to the first and last entries of the SNF are bound by $1 + log_2(\lceil \frac{n}{2} \rceil)$. This is as good as the bound we discovered for the standard divide and conquer routine earlier. Now, as we continue we are very careful about the order in which we select our pairs i.e. if given a choice use the position with the least power , and the power growth then proceeds as follows

```
[ 3, 3, 3, 3, 3, 2, 2, 3 ],
[ 3, 3, 3, 3, 3, 3, 3, 3 ],
[ 3, 3, 3, 4, 4, 3, 3, 3 ],
[ 3, 4, 3, 4, 4, 4, 3, 3 ],
[ 3, 5, 3, 5, 4, 4, 3, 3 ],
[ 3, 5, 4, 5, 4, 4, 4, 3 ],
[ 3, 5, 4, 5, 5, 4, 5, 3 ],
[ 3, 5, 5, 5, 5, 5, 5, 3 ],
[ 3, 5, 5, 6, 5, 5, 6, 3 ],
[ 3, 5, 7, 6, 5, 5, 7, 3 ],
[ 3, 5, 7, 7, 5, 7, 7, 3 ],
[ 3, 5, 7, 7, 5, 8, 8, 3 ]
```

This appears to be as well as we can do. Upon examining the maximum powers occurring we find that the maximum power for a problem of length n appears to be $O(n)$.

In this case a simple calculation shows that the difference between successive ratios of $\frac{maxpower(2^k)}{2^k}$ appears to be tending towards zero.

We conjecture that the maximum power that could appear in the length n problem using this strategy is $n\sqrt{2} + O(log(n))$

In fact experiments have shown that the divide into pairs with intelligent collection tglpIQ performs even better than predicted in terms of path length and powergrowth. It turns out

that we don't require ALL the steps suggested by this sequence for W_n with $n > 7$, i.e. we create certain divisibilities early in the sequence, leading to the case that certain later steps would cause no progress to be made. In light of this we can make no certain statements about the shortest path or smallest maximum power arising in any path through $\Gamma_2(W_n)$. The divide into pairs with intelligent collection idea does however provide us with tighter upper bounds than were previously known.

6.3.2 Random

It is interesting to investigate the range of powers produced from following an average, or random, path selection, and to compare these with the previous heuristics. Firstly we provide, for the purposes of comparison, some results relating to the previous positional algorithms described in this chapter.

n	4	5	6	7	8	9	10
Standard	5	7	9	11	13	15	17
Worst Case	6	10	15	21	28	36	45
D&C (into 2)	3	7	7	10	10	14	15
D&C (pairs)-Naive collect	3	6	6	10	10	15	15
D&C (pairs)-Intelligent	3	6	6	9	8	12	11

Table 6.10: Maximum Powers produced by various heuristics for W_n

The following bar charts demonstrate the behaviour of 1000 random runs over various length worst case problems.

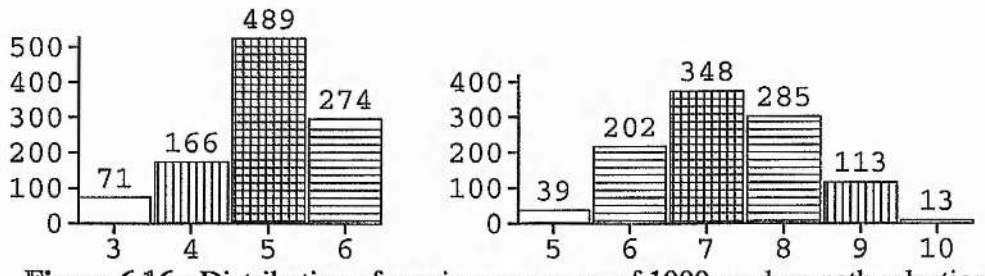


Figure 6.16 : Distribution of maximum powers of 1000 random path selections through $\Gamma_2(W_4)$ and $\Gamma_2(W_5)$.

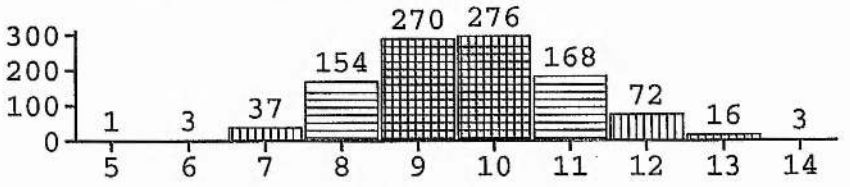


Figure 6.17 : Distribution of maximum powers of 1000 random path selections through $\Gamma_2(W_6)$.

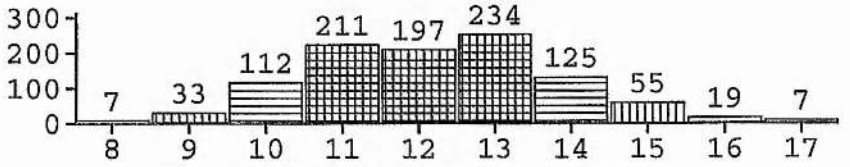


Figure 6.18 : Distribution of maximum powers of 1000 random path selections through $\Gamma_2(W_7)$.

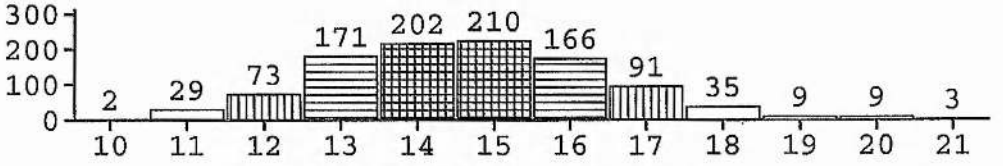


Figure 6.19 : Distribution of maximum powers of 1000 random path selections through $\Gamma_2(W_8)$.

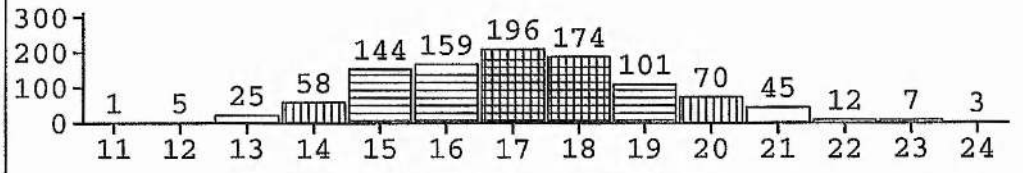


Figure 6.20 : Distribution of maximum powers of 1000 random path selections through $\Gamma_2(W_9)$.

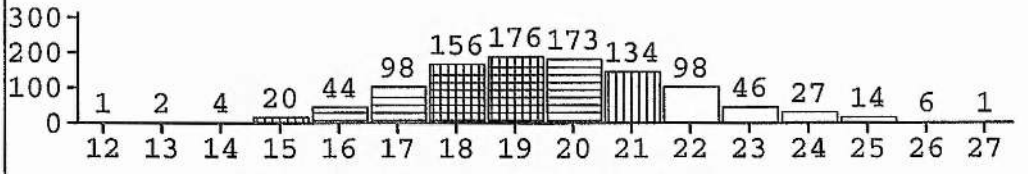


Figure 6.21 : Distribution of maximum powers of 1000 random path selections through $\Gamma_2(W_{10})$.

The range of the sums of the powers appearing from random paths through W_n is a much broader spread. It is not particularly illuminating to show the plots of these wide bell curves here. We will instead briefly describe the salient points for each n. Results of applying the positional algorithms developed here are displayed in Table 6.7 for comparison.

n	4	5	6	7	8	9	10
Standard	17	29	44	62	83	107	134
Worst Case	20	40	70	112	168	240	330
D&C (into 2)	10	27	33	49	57	86	106
D&C (pairs)-Naive collect	10	23	28	51	60	96	110
D&C (pairs)-Intelligent	10	23	28	46	46	76	78

Table 6.11: Sums of Powers produced by various heuristics for W_n

For $n = 4$ the sum of the powers, from a 1000 random paths, ranged from 10 to 21. This had two peaks, at 17 and 21, with over 200 'hits' each.

For $n = 5$ the sum of the powers ranged from 21 to 44. There were peaks at 29 and 32 with 115 hits and127 hits respectively.

For $n = 6$ the sum of the powers ranged from 28 to 75. There were no obvious peaks, but the main plateau was from 45 to 54, where each value had 50 to 60 hits.

For $n = 7$ the sum of the powers ranged from 44 to 106. Again there were no obvious peaks, but again a main plateau could be identified from 64 to 83, wherein each value had at least 30 hits.

For $n = 8$ the sum of the powers ranged from 65 to 146. The values between 92 and 117 all had at least 25 hits.

For $n = 9$ the sum of the powers ranged from 91 to 190. The values between 123 and 145 all had at least 20 hits.

For $n = 10$ the sum of the powers ranged from 115 to 251. The values between 156 and 189 all had at least 15 hits, and there appears to be a slight peak at 171 with a grand total of 30 hits.

From these results it would appear that random selection is generally slightly worse than the standard algorithm, though there is of course the potential to perform much better or much worse. The divide and conquer algorithms appear to compare favourably, at least

over this range of lengths. It is not at all obvious what the expected maximum power or power spread occurring from a random run would be in general.

6.3.3 Relationship Between Theory and Practice

While the power growth idea developed in this section is interesting in it's own right it is obviously much more interesting still if we can show a correlation between theory and practice. In fact we will see that there is indeed such a correlation, not only in the worst case scenario but also in the wider general setting.

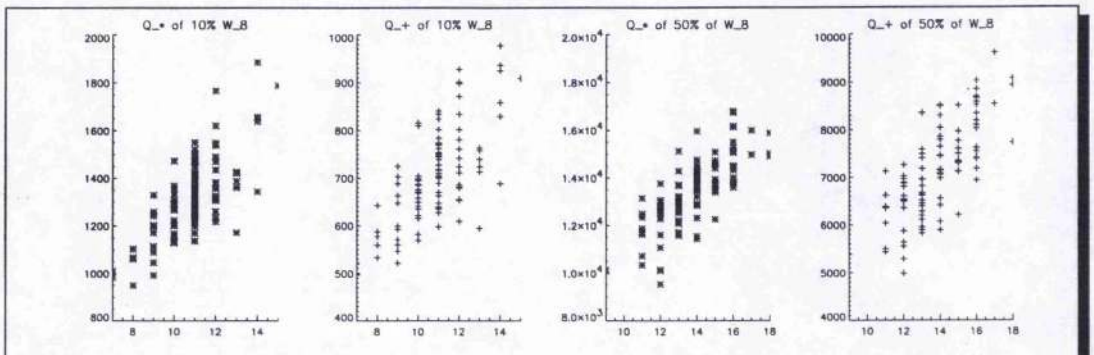


Figure 6.22 : Maximum power occurring in theoretical powergrowth of random runs vs actual qualities in 10% (left pair) and 50% (right pair) of W_8.

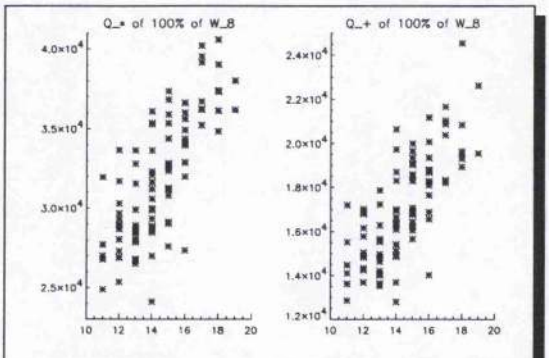


Figure 6.23 : Maximum power occurring in theoretical powergrowth of random runs vs actual qualities in W_8.

Firstly we see a good correlation from plotting quality (in fact log thereof, as usual) versus the theoretical powergrowth of several random runs on various input diagonals, as shown in figures 6.22 and 6.23 These experiments were run several times over various lengths, percentages and sets of elements and in each case a similar correlation was observed.

There is also a strong correlation between the maximum power as calculated from the trace versus our two usual qualities in the coprime case, i.e. where all the elements of the diagonal are coprime. Figure

6.24 demonstrates this correlation in an example arising from attempting to find the SNF of a 20×20 diagonal matrix with prime entries. Over the course of experiments using different diagonals of varying lengths and entries, the same behaviour and strong correlation between maximum power and quality occurred throughout. Recall that in the coprime case it can be seen that every element in either of the multiplier matrices is of the form $ABC\ldots$ where each X is an entry of one of the 2×2 matrices used during the calculation. That is to say there is no interference (addition or subtraction) during this calculation, and so minimizing both powergrowth and also each of the 2×2 matrices will clearly improve the quality of the final multiplier matrices.

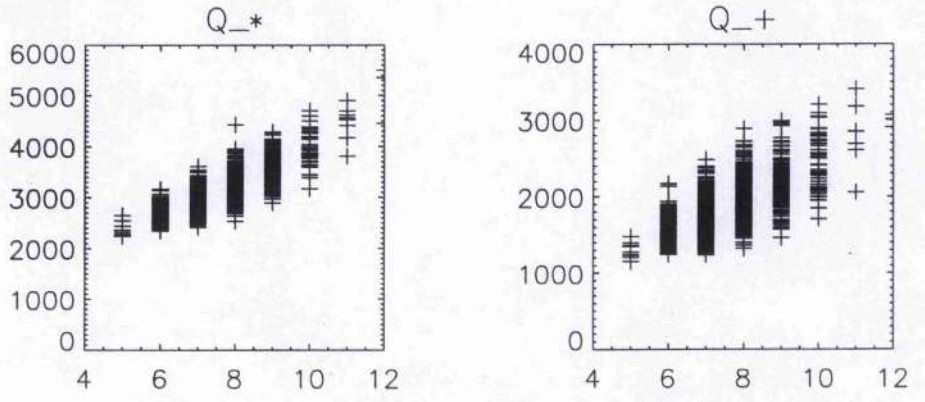


Figure 6.24 : Scatter plots showing the results of plotting log_2 of Quality versus the maximum theoretical power occurring from applying the actual steps taken for 1000 random runs to convert a 20×20 diagonal matrix with prime entries.

This correlation also appears when we examine the ranges of qualities produced by applying the positional algorithms we developed in the previous section to various input diagonals. Of course in this case we are primarily interested in the worst case input as otherwise we have to be careful when assigning our theoretical power; this case is demonstrated in Figure 6.25 . In more general cases where we possibly create a lot more edges in the divisibility graph at each step, negating the need for a later step in a given positional algorithm, we could examine the steps that do get used and compare the power growth with the size for a given sequence.

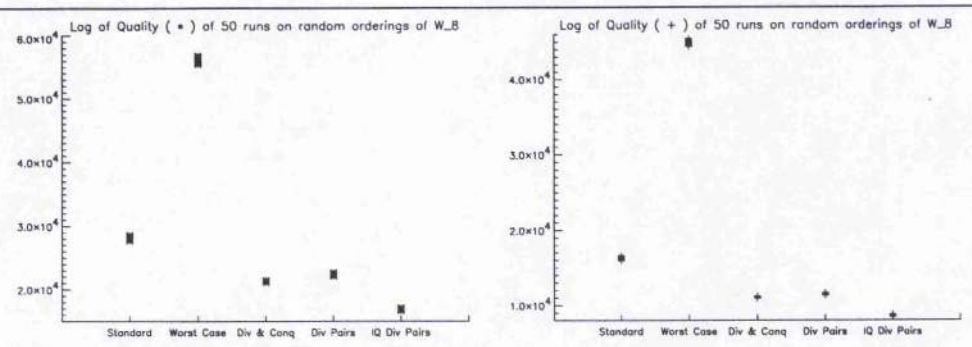


Figure 6.25 : Demonstration of the differences of applying the various positional algorithms described in this chapter to random orderings of a worst case of length 8.

The other main case in which we are interested is the coprime case, however, and it is simple to see that in this case the divide (into 2) and conquer and the divide into pairs with intelligent collection are both effective at spreading the damage, and minimizing the maximum theoretical power appearing. There is again a strong correlation visible, as shown in figure 6.26 :

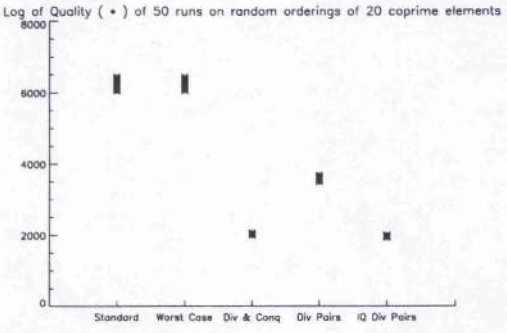


Figure 6.26 : Application to random orderings of a coprime case of length 20.

The largest powers in the theoretical powergrowth here are 19 for both the standard and worst case algorithms, 10 for the divide into pairs, and only 5 for both the divide and conquer and divide into pairs with intelligent collection strategy. The sums of final powers appearing being 209 for both the naive and worst case strategy, 46 for divide into 2 and conquer, and 74 and 40 respectively for the divide into pairs with naive collection and intelligent collection strategies. It appears that the divide into pairs with intelligent collection is the best strategy to choose in either the coprime or worst case scenarios.

6.4 Comparisons of Positional Algorithms

Having developed several positional algorithms by paying attention to the theoretical aspects of the problem at hand we will now provide experimental evidence to confirm that these algorithms do indeed perform well in practice.

We performed experiments upon various percentages (10, 20,. . .,100%) of various length (4,. . .,8) worst cases. In each case we selected 100 random inputs and performed 10 random runs upon each of these inputs. In each of the tables 6.12- 6.21, the values shown are the total number of these random runs that produced a better quality result than the positional algorithm in question. We also performed a similar experiment upon random coprime inputs of length up to 20. In each case the results are given as a pair $[x, y]$ where x is the result for the $\mathcal{Q}_*$ measure and y is the result for $\mathcal{Q}_+$. The values of x and y range between 0 and 1000, where 0 is good, 500 is average (random) behaviour, and 1000 is bad. As usual, these results are indicative of a larger range of experiments, all of which gave similar results.

From these results we can see that the standard algorithm, whilst performing reasonably on larger more complex problems, generally performs worse than a completely random algorithm in most cases we are likely to encounter in practice, especially for coprime input. The 'worst case' algorithm performs as expected, i.e. badly, further strengthening the correlation we have seen between theoretical powergrowth and practice. The results for the divide and conquer heuristics display some interesting properties, performing generally well but giving better results for even length inputs, which split more easily into smaller cases. This appears to be due to the fact that there is a correlation between shorter paths and quality, which is helping, and a correlation between powergrowth and quality, which we are not making the most of here.

It is interesting to note that the divide (into 2) and conquer routine appears to perform much better than the divide into pairs (IQ) for coprime inputs of length 5,7 and 9. This is not an experimental anomaly, rather it is a consequence of applying algorithms designed for use in the worst case to a non-worst case input. In this coprime case, as with any non-worst-case input, many of the steps of these algorithms are not used, and so the theoretical powergrowth, and path length bears less resemblance to practice. We note that while we could have developed algorithms that attempted to minimize powergrowth on general input, these would then be structural algorithms, and as such these will be discussed in the next section. We will however mention here reasons for this behaviour. In the coprime

case, where all path lengths are equal, we can get a slightly clearer view of how damage is accumulated. In this case we can step a little further from the powergrowth general bound and see that at each step the damage that is done is dependent upon how many coprime elements have 'built up' at a given stage. That is to say, we are not only interested in the maximum power of some bound appearing in the multiplier matrices, but also in the values of the pair a and b. In the coprime case we can track the growth of these entries in a similar fashion to the powergrowth tracking. By so doing we discover that the final step in the process of applying the sequence proscribed by divide into 2, upon the length 5 coprime input, demands a 2×2 operation to be performed upon a and b where $a = pqr$ and $b = st$, i.e. a is the product of three of the original entries of the input, b is the product of the other 2. Applying the sequence proscribed by divide into pairs in this case we find the last step is to be performed upon one entry of the original input, and one entry which is the product of the other 4 entries of the input. In general this greater dissimilarity in size will mean larger multipliers, as can be seen by considering the effect of such differences on the explicit multiplier matrices defined in equation 5.11. Hence the divide into pairs heuristic performs worse than the divide into 2, mainly due to this final imbalance. A similar situation appears in both the length 7 and 9 problems, i.e. the divide into 2 performs a final operation upon a reasonably balanced pair, whilst the divide into pairs heuristic generally performs a final operation upon an unbalanced pair. These differences however are far less pronounced for larger coprime problems and we begin to see fairly similar behaviour between both the divide into two and the divide into pairs heuristics.

Though both the divide into two and the divide into pairs heuristics appear to perform similarly well we believe that overall the divide into pairs heuristic is the better of the two. It performs better both in terms of path length found and theoretical (and practical) powergrowth. Tables 6.22 and 6.23 detail the results of an experiment to compare these two algorithms. For each row, 100 random inputs were created, corresponding to the length and percentage height detailed by the first two columns of the table. The two algorithms in question were then applied to each of these inputs; the third and fourth columns detail the sum of all the (logs of) qualities and the fifth and sixth columns detail how the algorithms performed in comparison to one another. As can be seen from table 6.22 the divide into two heuristic appears to perform slightly better for lengths less than 8, though the actual difference is very small, and the trend is far from obvious. For input lengths 8 or more however the divide into pairs heuristic can be seen, from table 6.23, to outperform the

divide into two heuristic considerably over the entire statespace. Recall that for lengths of 8 and greater we found that the divide into pairs, with intelligent collection, performed better than expected finding even shorter paths than predicted. We believe that these results are a consequence of that behaviour and that the divide into pairs heuristic will, for larger length inputs, consistently outperform the divide into two heuristic.

n	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
4	[NA,NA]	[NA,NA]	[327,323]	[552,538]	[581,564]	[586,557]	[521,559]	[480,444]	[443,343]	[348,358]
5	[274,224]	[668,646]	[580,595]	[556,537]	[509,531]	[464,427]	[534,510]	[400,456]	[265,317]	[294,313]
6	[611,598]	[648,623]	[406,417]	[403,447]	[282,347]	[287,333]	[264,365]	[233,324]	[203,342]	[156,285]
7	[797,755]	[524,551]	[465,540]	[246,325]	[237,338]	[161,281]	[110,252]	[124,216]	[133,284]	[185,337]
8	[592,648]	[327,506]	[191,378]	[269,423]	[97,292]	[95,309]	[94,187]	[54,198]	[66,213]	[62,239]

Table 6.12: Sum of final positions ($[\mathcal{Q}_*, \mathcal{Q}_+]$) for standard heuristic on random inputs in comparison to random path selections (0-good,500-avg, 1K-bad)

n	4	5	6	7	8	9	10	11	12	13	14	15	16
Q_*	687	851	933	975	991	1K	1K	999	1K	1K	1K	1K	1K
Q_+	683	866	936	978	991	1K	1K	1K	1K	1K	1K	1K	1K

Table 6.13: Sum of final positions for standard heuristic on coprime inputs in comparison to random path selections (0-good, 500-avg, 1K-bad)

n	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
4	[NA,NA]	[NA,NA]	[345,341]	[568,553]	[560,567]	[598,567]	[597,634]	[673,770]	[692,769]	[780,947]
5	[242,219]	[575,545]	[611,606]	[679,687]	[754,764]	[654,711]	[799,844]	[925,943]	[899,918]	[988,1K]
6	[565,535]	[724,739]	[750,822]	[733,842]	[769,916]	[857,929]	[863,957]	[966,987]	[954,985]	[999,1K]
7	[731,729]	[865,916]	[879,926]	[935,989]	[930,993]	[946,997]	[935,996]	[968,1K]	[962,999]	[1K,1K]
8	[888,950]	[973,997]	[971,999]	[981,1K]	[990,1K]	[1K,1K]	[1K,1K]	[1K,1K]	[1K,1K]	[1K,1K]

Table 6.14: Sum of final positions ($[\mathcal{Q}_*, \mathcal{Q}_+]$) for worst-case heuristic on random inputs in comparison to random path selections (0-good,500-avg, 1K-bad)

n	4	5	6	7	8	9	10	11	12	13	14	15	16
Q_*	684	840	918	974	991	1K	1K	999	1K	1K	1K	1K	1K
Q_+	687	853	919	978	991	1K	1K	999	1K	1K	1K	1K	1K

Table 6.15: Sum of final positions for worst-case heuristic on coprime inputs in comparison to random path selections (0-good, 500-avg, 1K-bad)

n	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
4	[NA,NA]	[NA,NA]	[281,259]	[344,308]	[300,264]	[104,112]	[84,80]	[80,74]	[18,24]	[31,22]
5	[244,224]	[514,525]	[427,462]	[390,378]	[311,289]	[282,228]	[328,240]	[227,149]	[82,52]	[104,65]
6	[399,386]	[288,285]	[95,92]	[54,43]	[22,25]	[16,7]	[2,6]	[8,0]	[0,2]	[0,0]
7	[302,295]	[129,190]	[102,159]	[46,98]	[37,93]	[24,82]	[17,32]	[25,52]	[21,42]	[38,77]
8	[56,66]	[10,8]	[1,2]	[0,2]	[0,0]	[0,0]	[0,0]	[0,0]	[0,0]	[0,0]

Table 6.16: Sum of final positions ($[\mathcal{Q}_*, \mathcal{Q}_+]$) for divide (into 2) and conquer heuristic on random inputs in comparison to random path selections (0-good,500-avg, 1K-bad)

n	4	5	6	7	8	9	10	11	12	13	14	15	16
Q_*	123	0	136	114	49	55	33	2	41	84	31	25	15
Q_+	108	0	60	29	17	0	1	0	2	1	4	0	0

Table 6.17: Sum of final positions for divide (into 2) and conquer heuristic on coprime inputs in comparison to random path selections (0-good, 500-avg, 1K-bad)

n	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
4	[NA,NA]	[NA,NA]	[281,259]	[344,308]	[300,264]	[104,112]	[84,80]	[80,74]	[18,24]	[31,22]
5	[238,215]	[422,415]	[355,385]	[303,311]	[315,289]	[266,244]	[336,259]	[226,153]	[77,49]	[86,56]
6	[371,379]	[191,196]	[65,59]	[30,30]	[19,19]	[6,5]	[0,0]	[0,0]	[0,0]	[0,0]
7	[339,338]	[205,198]	[231,207]	[110,86]	[92,78]	[72,46]	[22,19]	[59,25]	[66,17]	[106,60]
8	[147,130]	[28,21]	[2,4]	[0,0]	[0,0]	[0,0]	[0,0]	[0,0]	[0,0]	[0,0]

Table 6.18: Sum of final positions ($[\mathcal{Q}_*, \mathcal{Q}_+]$) for divide (into pairs) with naive collection strategy heuristic on random inputs in comparison to random path selections (0-good,500-avg, 1K-bad)

n	4	5	6	7	8	9	10	11	12	13	14	15	16
Q_*	123	624	144	593	327	613	508	854	694	928	766	970	936
Q_+	108	590	263	646	403	655	606	870	762	942	809	998	962

Table 6.19: Sum of final positions for divide (into pairs) with naive collection strategy heuristic on coprime inputs in comparison to random path selections (0-good, 500-avg, 1K-bad)

n	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
4	[NA,NA]	[NA,NA]	[281,259]	[344,308]	[300,264]	[104,112]	[84,80]	[80,74]	[18,24]	[31,22]
5	[238,215]	[422,415]	[355,385]	[303,311]	[315,289]	[266,244]	[336,259]	[226,153]	[77,49]	[86,56]
6	[345,309]	[173,167]	[40,38]	[28,30]	[15,27]	[12,15]	[0,6]	[6,0]	[0,8]	[0,9]
7	[287,284]	[147,140]	[162,143]	[85,106]	[56,82]	[51,69]	[18,28]	[32,51]	[55,34]	[98,101]
8	[0,0]	[0,0]	[0,0]	[0,0]	[0,0]	[0,0]	[0,0]	[0,0]	[0,0]	[0,0]

Table 6.20: Sum of final positions ($[\mathcal{Q}_*, \mathcal{Q}_+]$) for divide (into pairs) with intelligent collection strategy heuristic on random inputs in comparison to random path selections (0-good,500-avg, 1K-bad)

n	4	5	6	7	8	9	10	11	12	13	14	15	16
Q_*	123	624	144	593	49	491	9	65	46	15	14	75	15
Q_+	108	590	263	646	17	283	0	30	53	13	12	0	0

Table 6.21: Sum of final positions for divide (into pairs) with intelligent collection strategy heuristic on coprime inputs in comparison to random path selections (0-good, 500-avg, 1K-bad)

n	%	100 Run Sum, $[\mathcal{Q}_*, \mathcal{Q}_+]$ for Divide into two	100 Run Sum, $[\mathcal{Q}_*, \mathcal{Q}_+]$ for Divide into pairs	$\mathcal{Q}_*$ data [d2 , = , dp]	$\mathcal{Q}_+$ data [d2 , = , dp]
5	1	[1354,762]	[1341,756]	[7,86,7]	[7,87,6]
5	2	[6473,3519]	[6114,3338]	[28,13,59]	[33,14,53]
5	3	[14900,8163]	[14331,7889]	[39,6,55]	[40,6,54]
5	4	[24631,13138]	[25462,13765]	[58,2,40]	[61,2,37]
5	5	[38876,20669]	[38504,20593]	[50,0,50]	[49,0,51]
5	6	[52743,27648]	[51566,27365]	[40,0,60]	[41,0,59]
5	7	[66673,34443]	[65954,34419]	[46,0,54]	[49,1,50]
5	8	[82183,42242]	[81097,41737]	[51,1,48]	[40,1,59]
5	9	[97306,49826]	[95501,48516]	[38,2,60]	[29,2,69]
5	10	[112100,57500]	[111900,56800]	[0,0,100]	[0,0,100]
6	1	[6764,3688]	[6477,3506]	[40,4,56]	[36,8,56]
6	2	[26624,14369]	[24211,13023]	[30,1,69]	[28,2,70]
6	3	[52464,27703]	[49225,26172]	[31,2,67]	[30,1,69]
6	4	[82048,42807]	[75338,40147]	[21,1,78]	[26,0,74]
6	5	[116455,60338]	[110301,58427]	[31,0,69]	[42,2,56]
6	6	[146780,75568]	[145841,76840]	[64,0,36]	[66,0,34]
6	7	[179699,92460]	[177446,93769]	[48,1,51]	[64,0,36]
6	8	[213677,109315]	[215912,113848]	[67,0,33]	[84,0,16]
6	9	[248008,125797]	[251376,131833]	[85,0,15]	[92,0,8]
6	10	[288900,144800]	[292300,153100]	[100,0,0]	[100,0,0]
7	1	[28066,15319]	[28853,15477]	[56,0,44]	[52,3,45]
7	2	[101723,56066]	[103549,54591]	[54,0,46]	[37,0,63]
7	3	[187060,104206]	[189817,100834]	[61,0,39]	[38,0,62]
7	4	[284942,160337]	[296947,157783]	[76,0,24]	[38,1,61]
7	5	[387237,218870]	[404984,214365]	[86,0,14]	[42,0,58]
7	6	[486252,274061]	[510853,271611]	[92,1,7]	[43,0,57]
7	7	[593687,333881]	[628570,335093]	[98,0,2]	[51,0,49]
7	8	[697227,391827]	[740288,395250]	[100,0,0]	[67,0,33]
7	9	[811834,457008]	[865304,462022]	[100,0,0]	[77,1,22]
7	10	[928200,521900]	[987000,527500]	[100,0,0]	[100,0,0]

Table 6.22: comparison between divide into two and divide into pairs (part 1)

n	%	100 Run Sum, $[\mathcal{Q}_*, \mathcal{Q}_+]$ for Divide into two	100 Run Sum, $[\mathcal{Q}_*, \mathcal{Q}_+]$ for Divide into pairs	$\mathcal{Q}_*$ data [d2 , =, dp]	$\mathcal{Q}_+$ data [d2 , = , dp]
8	1	[98693,52843]	[84950,44385]	[12,0,88]	[9,0,91]
8	2	[279822,146779]	[223472,115121]	[1,0,99]	[1,0,99]
8	3	[496331,259921]	[388755,199436]	[0,0,100]	[0,0,100]
8	4	[707957,368376]	[552479,282606]	[0,0,100]	[0,0,100]
8	5	[939703,490020]	[734008,374324]	[0,0,100]	[0,0,100]
8	6	[1171915,610498]	[919316,468883]	[0,0,100]	[0,0,100]
8	7	[1401426,731623]	[1105186,562354]	[0,0,100]	[0,0,100]
8	8	[1643429,857198]	[1299643,660246]	[0,0,100]	[0,0,100]
8	9	[1884800,980590]	[1494754,758664]	[0,0,100]	[0,0,100]
8	10	[2144100,1114000]	[1703600,863900]	[0,0,100]	[0,0,100]
9	1	[344782,183828]	[323137,170686]	[25,0,75]	[27,0,73]
9	2	[913715,488584]	[800168,415729]	[3,0,97]	[1,0,99]
9	3	[1548733,831058]	[1331505,690478]	[1,0,99]	[1,0,99]
9	4	[2200584,1179711]	[1888526,975821]	[0,0,100]	[0,0,100]
9	5	[2883841,1544090]	[2471560,1275758]	[0,0,100]	[0,0,100]
9	6	[3585079,1919201]	[3066642,1578929]	[0,0,100]	[0,0,100]
9	7	[4305373,2300976]	[3684830,1893610]	[0,0,100]	[0,0,100]
9	8	[5017584,2681570]	[4304373,2214931]	[0,0,100]	[0,0,100]
9	9	[5775703,3085271]	[4951987,2547283]	[0,0,100]	[0,0,100]
9	10	[6526500,3484700]	[5594500,2878800]	[0,0,100]	[0,0,100]
10	1	[963978,498928]	[798635,411342]	[0,0,100]	[0,0,100]
10	2	[2294931,1174899]	[1869310,952007]	[0,0,100]	[0,0,100]
10	3	[3740858,1909712]	[3048809,1543905]	[0,0,100]	[0,0,100]
10	4	[5245763,2668662]	[4298115,2170625]	[0,0,100]	[0,0,100]
10	5	[6813030,3469217]	[5571534,2807133]	[0,0,100]	[0,0,100]
10	6	[8385058,4260206]	[6889384,3469013]	[0,0,100]	[0,0,100]
10	7	[10015045,5096049]	[8240074,4143892]	[0,0,100]	[0,0,100]
10	8	[11650749,5920169]	[9602865,4827580]	[0,0,100]	[0,0,100]
10	9	[13319112,6773142]	[10979387,5516315]	[0,0,100]	[0,0,100]
10	10	[15042200,7647000]	[12414400,6236700]	[0,0,100]	[0,0,100]

Table 6.23: comparison between divide into two and divide into pairs

6.5 Structural Heuristics

There are far too many potential structural heuristics for us to examine them all. We will attempt to examine a number of the more obvious ideas. The structural heuristics are very tempting as the results are independent of permutation of input and it appears at first sight that we can hopefully produce better final results by making good choices at each stage based on our knowledge at that stage. If we define our heuristics well enough then there are no arbitrary decisions that need be taken (that is to say we can give secondary, tertiary criteria). This can however get rather difficult and we will examine in this section how these heuristics compare in practice as primary criteria. Note also that we need to ensure that we select pairs with which we will actually make progress, so the implementations start to get a little trickier as we don't want to end up in some odd loop, or reselecting the same pair repeatedly.

Note that these are generally one step lookahead algorithms and so we will be making one of $\binom{m}{2}$ choices at each stage. This is where we lose out to the arbitrary positional algorithms, having gained of course from not needing to examine all the permutations of input. However it should be noted that most implementations of positional algorithms do not in fact examine all permutations of input.

We could of course, in many instances, implement two or more step lookahead algorithms. This can be very costly in terms of memory requirements, and obviously speed (to which we have not really paid much attention). We will examine the following heuristics in more detail in this section.

- Mingcd.
- Maxlcm.
- Maximal Edge creation in Γ_2(D).
- Height/Weight of remaining problem.
- Least squares proximity to SNF.
- Smallest pair.
- Largest pair.
- Pair with smallest associated rows of P and columns of Q.
- 'Full' lookahead to minimize the quality of P, Q after each step.

6.5.1 Overview of the Structural Algorithm

The structural algorithms can all be coded within the same basic template, the only thing changing being the method of selecting which pair to use at the next stage. This template is as shown in Figure 6.27.

```
1. While D is not in SNF do
2.       select a pair i,j
3.       D[i][i]:=gcd, D[j][j]:=lcm.
4. od;
```

Figure 6.27 : Pseudocode template for structural algorithms.

This appears to be simple. There are a however a few points of which to be wary.

Firstly the diagonal at each stage has to be checked to decide if the algorithm has completed. The simplest way of performing this check is to precalculate the SNF, sort the diagonal at each stage and compare it to the SNF.

It is also important that progress is made at each stage and no loops are entered into, e.g. when using the MinGcd heuristic described earlier on the diagonal $[2, 4, 12, 20]$ we must not select the pair with the smallest gcd here (2,4), but rather the pair with the smallest gcd that will actually make some progress (12,20). This is easily achieved by 'locking' any elements that are in the SNF, and selecting the pair to use from the others. It should be fairly obvious that this forces progress to always be made. Care must also be taken in this locking out procedure to not lock out elements accidentally, e.g. in the diagonal $[2, 7, 7]$ we wish to lock out only one of the 7's. The locking procedure implemented for the selection routines described hereafter randomly selected from any multiple set of entries which subset to lock out. An improvement may be, if given a choice, to select the elements to which are attached the currently largest multiplying rows / columns in the multiplier matrices, but since this is basically applying a further structural algorithm, in the interests of fair comparison this was not implemented.

In each case it should be noted that the structural heuristic was used as a primary heuristic, all further arbitrary choices being made uniformly at random between any candidates. Further improvements may well follow from implementing a more intelligent decision

procedure to distinguish between any arbitrary choices, i.e. secondary, tertiary, etc heuristics. We will now describe the structural heuristics mentioned above and investigate their potential.

For each heuristic we performed experiments on various percentages (10, 20...,100%) of various length (4...8) worst cases. In each case we selected 20 random inputs and also performed 20 random runs. The values displayed are the total number of random runs, of a maximum of 400, that performed better than the structural algorithm in question. We also performed a similar experiment upon 25 random coprime inputs, with 40 random runs for comparison. In this case then the value in the table is between 0 and 1000.

6.5.2 Random

Not strictly a structural heuristic, but then again not strictly a positional heuristic. The random selection procedure forms the backdrop for our comparisons and is used as the secondary criterion for the following heuristics. As we have seen the random heuristic produces a wide range of qualities.

6.5.3 MinGcd

Given that the usual implementation (the standard positional approach) proceeds by repeatedly obtaining the gcd of the set of elements remaining at each stage, one natural extension of this is to attempt to obtain the gcd as quickly as possible. One sensible approach then seems to be to always pick the pair with the minimum gcd. Recalling that given a pair of random integers there is a high probability them being coprime we expect this will generally perform well. An immediate drawback is that in the coprime case itself this heuristic gives no indication of which pair to select, and in fact passes transparently to the secondary heuristic, which is in this case completely random.

The results in tables 6.24 and 6.25 show that this is not a good heuristic to use. In fact it appears that selecting a path at random is usually better. For coprime inputs, as mentioned above, the algorithm performs as though it were a random path selection.

6.5.4 MaxLcm

Again, a fairly natural extension of the standard positional procedure, the idea of this heuristic is to try and find the large entries of the SNF first. This appears a reasonable strategy as these entries generally require the largest entries in the multiplier matrices, and so it appears sensible to try and get these out of the way first. Similarly to situation for MinGcd, since two random integers are coprime more often than not, we expect to be able to generate the lcm of all the entries quite quickly. For this heuristic we fall through to the secondary criterion far less than for MinGcd.

The results in tables 6.26 and 6.27 show there to be no merit to this heuristic. The results are worse than those produced by random selection in most cases, including coprime.

6.5.5 Edge Creation in $\Gamma_2(\mathbb{D})$

Whilst we would really prefer to find pairs that directly create entries of the SNF this is not always possible. The immediate extension to the idea of creation of SNF entries is to use the divisibility digraph (Definition 4.3) and to select the pair that will create the most edges in the digraph of the next stage. This suffers from similar problems to MinGcd in that in both the coprime case and the worst case, there are generally multiple options and we fall through to the secondary criteria quite often.

The results in tables 6.28 and 6.29 provide us with no reason to promote this as a good choice of heuristic. In all cases performing one or two random runs should provide a better quality solution.

6.5.6 Height / Weight

Recall that the height of a problem is the number of distinct letters in the signature associated with it, i.e. the number of coprime pieces. The weight is the total number of letters in the signature. As we progress through Γ the height and weight of each successive diagonal will be less than the preceding one. The idea behind these heuristics is to attempt to make large jumps by selecting the pair that will lead to the diagonal with the least height or weight. Unfortunately, as with most other structural heuristics we fall through to the secondary criterion relatively often. Also as we have to totally rewrite each of the potential $\binom{n}{2}$ diagonals at each next stage in order to investigate the height or weight

n	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
4	NA	NA	[94,90]	[147,131]	[198,179]	[180,160]	[206,200]	[257,262]	[313,300]	[376,329]
5	[124,111]	[216,173]	[201,194]	[271,263]	[271,268]	[296,310]	[304,303]	[356,371]	[354,371]	[398,400]
6	[140,135]	[168,155]	[223,229]	[336,352]	[333,335]	[379,383]	[342,351]	[390,395]	[377,388]	[399,400]
7	[183,177]	[310,311]	[342,345]	[375,383]	[391,398]	[396,400]	[398,400]	[399,400]	[400,400]	[400,400]
8	[250,282]	[368,379]	[398,400]	[394,400]	[399,400]	[400,400]	[400,400]	[400,400]	[400,400]	[400,400]

Table 6.24: Sum of final positions ($[\mathcal{Q}_*, \mathcal{Q}_+]$) for MinGcd heuristic on random inputs in comparison to random path selections (0-good,200-avg, 400-bad)

n	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Q_*	418	415	577	525	539	567	540	581	588	494	527	505	471	547
Q_+	405	412	582	494	559	586	531	585	596	500	525	514	491	527

Table 6.25: Sum of final positions for MinGcd heuristic on coprime inputs in comparison to random path selections (0-good, 500-avg, 1K-bad)

n	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
4	NA	NA	[84,85]	[121,133]	[182,176]	[172,184]	[122,121]	[186,235]	[223,257]	[326,390]
5	[109,107]	[158,183]	[151,160]	[212,230]	[178,217]	[208,253]	[240,280]	[293,339]	[331,377]	[394,400]
6	[145,139]	[114,138]	[228,260]	[270,328]	[300,333]	[341,374]	[366,391]	[385,397]	[369,390]	[397,400]
7	[141,157]	[290,326]	[322,375]	[368,392]	[390,396]	[394,400]	[390,400]	[398,400]	[393,400]	[399,400]
8	[279,324]	[384,396]	[394,400]	[400,400]	[395,400]	[398,400]	[393,400]	[399,400]	[400,400]	[400,400]

Table 6.26: Sum of final positions ($[\mathcal{Q}_*, \mathcal{Q}_+]$) for MaxLcm heuristic on random inputs in comparison to random path selections (0-good,200-avg, 400-bad).

n	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Q_*	455	765	961	993	1K	1K	1K	1K	1K	1K	1K	1K	1K	1K
Q_+	488	789	962	994	1K	1K	1K	1K	1K	1K	1K	1K	1K	1K

Table 6.27: Sum of final positions for MaxLcm heuristic on coprime inputs in comparison to random path selections (0-good,500-avg, 1K-bad).

this is a computation heavy heuristic that does not appear justifiable from the experimental evidence.

The results in table 6.30 appear to show that the height heuristic, whilst not drastically outperforming random selection does generally produce results in the second quartile those qualities produced by random behaviour. In the coprime case, table 6.31, it appears to perform on a par with random selection.

The results in table 6.32 and table 6.33 suggest that the weight heuristic generally produces results that are on a par with random selection for any input.

6.5.7 Proximity

An early idea : At some stage we have D. We know S of course. For each possible next sorted diagonal, D^+ examine the least squares measure

$$\sum_{i=1}^{n} (D_i^+ - S_i)$$

and select the pair that will minimize this. Hopefully we can then "home in on" the solution and avoid some of the problems that can occur in the greedier heuristics. In some sense this idea is a precursor to the edge creation and is a refinement of the Mingcd or MaxLcm ideas. A little consideration shows that this heuristic often selects a similar path to that picked by the simpler 'Largest Pair' heuristic.

This heuristic appears to perform quite badly in comparison with random selection, as can be seen from tables 6.34 and 6.35.

6.5.8 Smallest / Largest Pair

Since the size of the multipliers depend upon a and b another potential structural heuristic is to select either the smallest or largest pair with which we can make progress. We therefore require a method to measure the size of a pair of integers. In keeping with the rest of this thesis we consider the sum of squares metric i.e we let $(a, b) < (c, d)$ if $a^2 + b^2 < c^2 + d^2$. As noted previously the Largest Pair heuristic often has the same effect as the Proximity heuristic described earlier, especially in the coprime case.

The 'Smallest Pair' heuristic appears to have the most promise of any of the structural heuristics. Whilst not performing particularly well for general input, table 6.36, it does

n	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
4	NA	NA	[59,60]	[112,123]	[138,134]	[115,135]	[82,84]	[103,149]	[136,203]	[170,261]
5	[107,109]	[166,176]	[109,108]	[169,183]	[139,204]	[197,235]	[195,249]	[213,308]	[249,334]	[307,380]
6	[99,116]	[84,85]	[175,207]	[219,271]	[217,281]	[266,342]	[323,365]	[355,391]	[369,396]	[352,392]
7	[56,70]	[215,269]	[245,319]	[365,393]	[349,390]	[384,396]	[389,398]	[368,390]	[379,400]	[394,400]
8	[266,318]	[318,376]	[384,399]	[388,399]	[397,400]	[399,400]	[400,400]	[400,400]	[400,400]	[400,400]

Table 6.28: Sum of final positions ($[\mathcal{Q}_*, \mathcal{Q}_+]$) for MaxEdgeCreation heuristic on random inputs in comparison to random path selections (0-good, 200-avg, 400-bad).

n	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Q_*	394	448	523	539	415	539	545	510	469	498	582	487	439	504
Q_+	394	455	525	529	409	537	528	484	496	513	535	467	435	512

Table 6.29: Sum of final positions for MaxEdgeCreation heuristic on coprime inputs in comparison to random path selections (0-good, 500-avg, 1K-bad).

n	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
4	NA	NA	[104,98]	[117,122]	[153,143]	[126,121]	[120,155]	[156,156]	[210,212]	[215,222]
5	[79,77]	[179,176]	[125,118]	[184,215]	[95,116]	[153,182]	[119,119]	[167,195]	[197,266]	[130,204]
6	[108,110]	[114,122]	[82,88]	[119,126]	[129,168]	[196,235]	[142,190]	[163,219]	[180,247]	[148,210]
7	[111,118]	[114,138]	[73,111]	[73,114]	[125,177]	[147,221]	[137,211]	[145,265]	[155,247]	[137,214]
8	[85,125]	[94,165]	[125,206]	[102,169]	[150,238]	[143,255]	[147,221]	[162,215]	[183,235]	[196,230]

Table 6.30: Sum of final positions ($[\mathcal{Q}_*, \mathcal{Q}_+]$) for Height heuristic on random inputs in comparison to random path selections (0-good, 200-avg, 400-bad).

n	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Q_*	374	488	425	565	518	504	499	536	556	583	458	508	534	527
Q_+	361	483	407	537	521	504	481	546	546	569	454	492	543	518

Table 6.31: Sum of final positions for Height heuristic on coprime inputs in comparison to random path selections (0-good, 500-avg, 1K-bad).

n	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
4	NA	NA	[76,78]	[166,149]	[171,152]	[180,196]	[185,186]	[243,241]	[232,233]	[243,230]
5	[85,78]	[174,185]	[201,192]	[190,199]	[188,195]	[210,197]	[211,203]	[199,185]	[238,236]	[242,237]
6	[143,136]	[197,185]	[213,214]	[239,240]	[208,219]	[218,250]	[233,225]	[211,216]	[209,220]	[255,243]
7	[159,174]	[193,178]	[210,219]	[247,250]	[214,222]	[211,203]	[235,229]	[231,210]	[227,227]	[233,208]
8	[172,186]	[220,212]	[214,217]	[243,250]	[192,185]	[202,213]	[213,206]	[232,221]	[233,224]	[245,231]

Table 6.32: Sum of final positions ($[\mathcal{Q}_*, \mathcal{Q}_+]$) for Weight heuristic on random inputs in comparison to random path selections (0-good, 200-avg, 400-bad).

n	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Q_*	341	447	510	548	633	443	464	624	451	562	602	561	474	474
Q_+	343	458	517	549	603	447	454	627	426	584	602	571	489	478

Table 6.33: Sum of final positions for Weight heuristic on coprime inputs in comparison to random path selections (0-good,500-avg, 1K-bad).

n	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
4	NA	NA	[108,108]	[170,155]	[226,211]	[221,223]	[310,301]	[352,360]	[350,363]	[396,396]
5	[110,103]	[233,209]	[218,227]	[261,256]	[311,302]	[312,319]	[333,335]	[374,394]	[392,399]	[400,400]
6	[165,158]	[298,303]	[309,305]	[356,365]	[347,357]	[385,385]	[387,397]	[397,398]	[392,398]	[400,400]
7	[191,203]	[340,346]	[346,349]	[385,385]	[393,393]	[400,399]	[399,400]	[397,400]	[399,400]	[400,400]
8	[337,351]	[389,394]	[398,398]	[396,398]	[400,399]	[400,400]	[400,400]	[400,400]	[400,400]	[400,400]

Table 6.34: Sum of final positions ($[\mathcal{Q}_*, \mathcal{Q}_+]$) for Proximity heuristic on random inputs in comparison to random path selections (0-good, 200-avg, 400-bad).

n	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Q_*	455	765	961	993	1K	1K	1K	1K	1K	1K	1K	1K	1K	1K
Q_+	488	789	962	994	1K	1K	1K	1K	1K	1K	1K	1K	1K	1K

Table 6.35: Sum of final positions for Proximity heuristic on coprime inputs in comparison to random path selections (0-good, 500-avg, 1K-bad).

appear to produce reasonably good quality solutions, in comparison to random selection, for the coprime case, table 6.37. As we mentioned when discussing the results for the positional algorithms it appears to be sensible practice to attempt to take entries a and b upon which to perform the next 2×2 step to be of similar size, and specifically to attempt to arrange that the last pair used are of similar magnitude to one another. This heuristic minimizes the differences between the magnitudes of the a and b selected at each stage, and again specifically those of the final step. This appears to be reflected in the results we are seeing here.

The results in table 6.38 and 6.39 appear to show that the largest heuristic is not a sensible choice if we require good quality multipliers.

6.5.9 RowColSize

As we have seen there appears to be a link between powergrowth and quality. The idea of this heuristic is to try and produce an adaptive algorithm that will minimize the largest 'power' appearing in the multipliers. We take the euclidean sizes of the rows and columns respectively of the row and column matrix multipliers, and use these to select which pair we shall use next at each stage. In an attempt to minimize the damage done we select the pair that has the smallest associated rows and columns.

The results in table 6.40 show that this heuristic performs reasonably for most heights and most lengths, producing qualities that are generally somewhat better than those produced by average random behaviour. In the coprime case this heuristic appears to perform on a par with random behaviour as shown by the results in table 6.41. These results may appear a little surprising in light of how well the positional algorithms that attempted to minimize powergrowth performed, but it appears that these results are due to the lack of rigid structure that ensures short paths can be found. Also it appears to be the case that the heuristics which 'try too hard' early on in the calculation simply store up for later problems that can then not be avoided.

6.5.10 Full Lookahead

At some stage we have multiplier matrices P_η, Q_η. We calculate (hopefully) good 2×2 multipliers for each possible next step. Apply each of these, and take the pair that minimises $\mathcal{Q}(P_{\eta+1}, Q_{\eta+1})$. We can improve the speed of this somewhat in general since

n	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
4	NA	NA	[121,118]	[173,150]	[236,212]	[220,223]	[246,241]	[266,267]	[352,329]	[376,329]
5	[87,76]	[221,186]	[269,241]	[302,282]	[292,280]	[324,317]	[332,325]	[368,377]	[388,395]	[398,400]
6	[219,211]	[248,230]	[305,290]	[356,361]	[367,371]	[389,395]	[381,393]	[391,395]	[396,399]	[394,399]
7	[272,262]	[328,320]	[391,397]	[398,400]	[390,395]	[399,400]	[400,400]	[400,400]	[399,400]	[400,400]
8	[351,357]	[389,396]	[400,400]	[400,400]	[400,400]	[400,400]	[400,400]	[400,400]	[400,400]	[400,400]

Table 6.36: Sum of final positions ($[\mathcal{Q}_*, \mathcal{Q}_+]$) for Smallest heuristic on random inputs in comparison to random path selections (0-good, 200-avg, 400-bad).

n	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Q_*	251	93	161	120	58	46	179	159	100	42	24	11	6	7
Q_+	238	134	111	205	72	46	23	80	66	38	30	16	4	2

Table 6.37: Sum of final positions for Smallest heuristic on coprime inputs in comparison to random path selections (0-good, 500-avg, 1K-bad).

n	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
4	NA	NA	[80,86]	[98,101]	[140,156]	[202,242]	[148,158]	[233,298]	[254,312]	[326,390]
5	[134,127]	[142,145]	[178,204]	[245,284]	[207,211]	[286,329]	[289,350]	[323,384]	[347,389]	[394,400]
6	[182,191]	[206,250]	[295,340]	[303,339]	[318,373]	[365,394]	[353,393]	[369,398]	[379,398]	[397,400]
7	[193,224]	[338,372]	[356,372]	[379,396]	[381,399]	[395,400]	[394,400]	[394,400]	[397,399]	[399,400]
8	[337,375]	[392,398]	[396,400]	[400,400]	[399,400]	[400,400]	[400,400]	[400,400]	[400,400]	[400,400]

Table 6.38: Sum of final positions ($[\mathcal{Q}_*, \mathcal{Q}_+]$) for Largest heuristic on random inputs in comparison to random path selections (0-good, 200-avg, 400-bad).

n	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Q_*	455	765	961	993	1K	1K	1K	1K	1K	1K	1K	1K	1K	1K
Q_+	488	789	962	994	1K	1K	1K	1K	1K	1K	1K	1K	1K	1K

Table 6.39: Sum of final positions for Largest heuristic on coprime inputs in comparison to random path selections (0-good, 500-avg, 1K-bad).

only two rows and two columns are affected by a given operation, however there is still a heavy computational burden for this heuristic.

The results in table 6.42 show good behaviour for the full lookahead, which appears to outperform random behaviour in general. It appears that the further from coprime, the better performance in comparison to random path selection we see. In the coprime case, table 6.43, we see that the lookahead performs slightly better than random path selection in general.

6.5.11 Structural Conclusions

Whilst some of the heuristics described here perform relatively well in some particular problems, we have not found a satisfactory structural heuristic, or combination thereof, for general use that can compete with applying the divide into pairs positional heuristic developed earlier. The structural heuristics also generally suffer from the need to perform a large amount of work at each stage, much of which is not reusable, and so they seem to not be suitable for solving real general problems where time is a factor. Potential uses for these smaller solutions being of course so that later calculations can be performed faster. We suggest that certain of these heuristics, in particular the full lookahead and the smallest pair, are however good for small problems (i.e. $n \times n$ matrices, where n is small) or in larger coprime or nearly coprime cases, i.e. where the SNF is mainly 1's where we strongly require good quality solutions.

6.6 Overall Strategy Conclusions

We have provided theoretical ideas and experimental evidence to support the theory that the "divide into pairs with intelligent collection strategy" provides an excellent basis for the solution to the problem of the order of pair selection. The structural heuristics, as well as generally being slow, suffer from several other problems and it seems difficult to arrange cascading criteria so that they perform well and are not relying on a large random element. We believe that the structural heuristics can be made use of however within the positional framework suggested. In particular we note that applying the divide into pairs heuristic could make use of a lookahead algorithm to decide exactly which pairs to use when there is a choice.

n	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
4	NA	NA	[95,90]	[190,184]	[172,182]	[120,134]	[163,169]	[156,131]	[147,123]	[146,182]
5	[100,92]	[179,192]	[124,113]	[166,164]	[132,138]	[115,89]	[81,70]	[81,71]	[94,90]	[109,116]
6	[122,123]	[126,121]	[160,135]	[114,96]	[96,104]	[67,69]	[78,71]	[61,66]	[50,49]	[50,46]
7	[195,205]	[155,166]	[134,134]	[148,145]	[107,104]	[116,114]	[156,153]	[148,163]	[52,50]	[23,19]
8	[175,190]	[157,167]	[87,103]	[122,140]	[115,143]	[125,99]	[87,97]	[131,126]	[84,93]	[79,86]

Table 6.40: Sum of final positions ($[\mathcal{Q}_*, \mathcal{Q}_+]$) for RowColSize heuristic on random inputs in comparison to random path selections (0-good, 200-avg, 400-bad).

n	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Q_*	348	648	662	297	555	314	539	471	606	403	454	483	526	385
Q_+	350	658	656	341	535	339	520	412	546	396	438	474	511	365

Table 6.41: Sum of final positions for RowColsize heuristic on coprime inputs in comparison to random path selections (0-good, 500-avg, 1K-bad).

n	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
4	NA	NA	[184,170]	[126,131]	[137,116]	[143,142]	[93,99]	[140,150]	[113,136]	[341,356]
5	[30,20]	[188,162]	[137,118]	[147,150]	[150,141]	[160,177]	[128,173]	[119,116]	[135,150]	[31,90]
6	[162,165]	[125,144]	[143,166]	[142,183]	[72,97]	[60,64]	[68,76]	[78,99]	[55,95]	[20,30]
7	[168,172]	[74,109]	[52,92]	[42,66]	[97,136]	[50,116]	[44,80]	[13,32]	[23,27]	[48,144]
8	[116,128]	[36,58]	[31,48]	[12,39]	[19,37]	[15,24]	[17,36]	[9,15]	[8,20]	[7,23]

Table 6.42: Sum of final positions ($[\mathcal{Q}_*, \mathcal{Q}_+]$) for Full Lookahead on random inputs in comparison to random path selections (0-good, 200-avg, 400-bad).

n	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Q_*	440	300	485	282	292	251	210	265	167	150	167	165	110	82
Q_+	272	202	377	442	332	80	105	232	192	250	197	180	175	35

Table 6.43: Sum of final positions for Full Lookahead on coprime inputs in comparison to random path selections (0-good,500-avg, 1K-bad).

Chapter 7

Bypassing the Diagonal Form

There are two main routes we can take when attempting to calculate the SNF of a matrix. We can either convert to a diagonal matrix which does not necessarily satisfy the divisibility requirement and deal with this separately, or we can attempt to directly compute the SNF, bypassing the intermediate diagonal form. Most algorithms to directly produce the SNF simply obtain each diagonal entry in turn by performing row and column operations to produce an element which is the gcd of the remaining $n \times n$ submatrix. This element is then used to zero out the other elements in the row and column in which it appears. The computation then proceeds with the same procedure on the $n-1 \times n-1$ submatrix remaining. The drawback of this approach is that it seems particularly difficult to avoid intermediate expression swell, which is why most algorithms convert to an arbitrary diagonal form (where we simply need to ensure that each element we find is the gcd of the row and column in which it appears, rather than the entire submatrix) and then sort out divisibility afterwards. The majority of this thesis studies the problems arising from this approach.

The algorithm described in this chapter also computes the SNF, with transforming matrices, of arbitrary input matrices without going via an intermediate diagonal form. In fact the algorithm implemented in GAP 4.2 is highly flexible and has several options permitting calculation of triangular, hermite or smith forms, with or without relevant transforming matrices. We shall focus upon the parts of the implementation producing the SNF with transforming matrices, but will occasionally comment upon the differences relating to the other options.

7.1 Overview of the Algorithm

At the heart of the algorithm is a routine, very similar to that described in [Sto97], to find a solution to the modulo N extended gcd problem. Unlike the algorithm for SNF with transforms described in [Sto97] however we do not perform modulo arithmetic and hence we do not need to first calculate a triangular form, which is of little further use. We can also construct both transforming matrices as we proceed through the algorithm and not rely upon matrix inversion, with its attendant difficulties, as a last step.

We will first describe the implementations of the lowest level operations we will wish to perform, without describing in too much detail exactly why we require them. Such details will be obvious when describing later algorithms, and the full theoretical development can be found in [Sto97].

7.1.1 split

For two integers N and a we will be interested in finding those divisors of N that are not divisors of a. This function returns the product of all of the prime factors of N which are not factors of a.

The implementation of this is very simple, being a single "while" loop wherein we repeatedly divide out any factors of N_i in a until there are no common factors left. We include the code here, since it is self-explanatory,

```
x:=a;       t:=N;

while x<>1 do
  x:=GcdInt(x,t);
  t:=QuoInt(t,x);
od;

return t;
```

It should be clear that the t returned by this function is exactly that value which we wish to calculate.

Example :

Let $N = 60 = 2^2.3.5$, and $a = 6 = 2.3$. Then the value, t, returned by the algorithm is 5. i.e. all powers of any factors of a are 'removed'.

7.1.2 rgcd

In [Sto97], three algorithmic solutions to the following problem are investigated:

Given integers a, b, N with N positive and $gcd(a, b) = 1$, find the smallest nonnegative integer c that satisfies

$$gcd(a + cb, N) = 1.$$

Three approaches are discussed

- Brute Force.
- Prime Factorization.
- Integer Factor Refinement.

The brute force approach is shown to require at worst $O(log^{3.5}N)$ bit operations. The prime factorization approach is bound by $O(log^2N)$ bit operations, but requires a full prime factorization of N. Integer Factor Refinement is shown to be similarly bound by $O(log^2N)$ bit operations but without the need for any further information. The algorithm of choice would then appear to be Integer Factor Refinement, however we have discovered that whilst Integer Factor Refinement does indeed appear to run faster for larger numbers, the extra complexity of the algorithm, setup overheads and general bookkeeping incurred make brute force a better choice for smaller integers.

It is impossible to produce an exact value for the breakpoint, but by examining the behaviour of the two algorithms on sets of random numbers in various ranges we can estimate where the crossover point is. In GAP 4.2 the break point appears to be when dealing with numbers of around $2^{1200} - 2^{1600}$ or so. For smaller numbers, up to 2^{28} or so, the brute force algorithm runs about 6 or 7 times faster than integer factor refinement, producing far fewer transient results and general garbage. The time difference between the two algorithms narrows as the size of the numbers we are dealing with increases, until the better asymptotic complexity of the integer factor refinement algorithm contributes enough to pay for the various overheads, although the tests we have run on random integers selected from the range $[1 \ldots \approx 2^{27000}]$ only show a roughly 10% difference in favour of the Integer Factor Refinement algorithm.

Since it is actually also quick to get an estimate of the size of a number, we get an overall improvement by first checking the size of the numbers involved and deciding which algorithm to utilise. In this way we can produce faster results, with less garbage creation,

when dealing with numbers of less than 400 decimal digits or so, and also gain for larger numbers due to the better asymptotic behaviour of the implemented algorithm.

7.1.3 2×2 HNF

Another of the simple base routines that we make use of is the transformation of a 2×2 matrix, A, to HNF, H, i.e. by row transforms. It should be noted that it is a simple matter to find a transformation matrix P such that $PA = H$ by use of the extended euclidean algorithm. In particular we actually require

$$\begin{bmatrix} s & t \\ u & v \end{bmatrix} \times \begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} e & f \\ 0 & g \end{bmatrix}$$

where e and g are positive, and $0 \leq f \leq g$. To calculate this we first let P be the 2×2 matrix corresponding to the extended euclidean algorithm upon the pair a, c. That is we compute

$$P = \begin{bmatrix} p & q \\ u & v \end{bmatrix}.$$

such that $P \times [a, c]^T = [e, 0]$, where $e = gcd(a, c)$. All that then remains is to ensure positivity of e and g by multiplication of the corresponding row of P by ± 1 as necessary, and to calculate how many multiples, m of g we need subtract from $bp + dq$ to reduce to the range $0 \leq f \leq g$. We then subtract m copies of the second row of P from the first to arrive at the required solution.

$$P = \begin{bmatrix} s & t \\ u & v \end{bmatrix}.$$

Recall that both the HNF, H and the transform, P are determined uniquely. The above procedure gives us an efficient way of computing P.

7.1.4 The modulo N extended gcd problem

The modulo N extended gcd problem can be stated as follows :

Given a non-negative integer N together with an integer row vector $a = [a_1, a_2, \ldots, a_n]$, find an integer row vector $v = [v_1, v_2, \ldots, v_n]$ such that

$$gcd\left(\sum_{i=1}^{n} v_i a_i, N\right) = gcd\left(a_1, a_2, \ldots, N\right).$$

This problem is solved in [Sto97], and is at the heart of this implementation. The solution described produces a row vector c with particularly small entries, in particular $c_1 = 1$. The procedure we actually implement differs from that described in the paper in several aspects. Firstly we take as input the triple (N, a, v) where N and a are integers, and v is an integer vector, and return a solution vector c such that

$$gcd(N, a + c_1 * v_1 + \ldots + c_k * v_k) = gcd(N, a, v_1, v_2, \ldots, v_k).$$

This is not really a difference as the algorithm described in the paper returns a solution with $c_1 = 1$. Secondly and more importantly, the algorithm described in [Sto97] works by computing $c_1, c_2, \ldots, c_l$ in succession. The algorithm we implement first precomputes the sequence

$$M_i = \frac{gcd(N, v_1, \ldots, v_{i-1})}{gcd(N, v_1, \ldots, v_i)}$$

and then computes the c_i in reverse. This allows to perform operations upon generally smaller numbers, as $M_i \leq N$, while the entries v_i are not bound in any manner. Further we have that M_n is the product of the factors common to $N, v_1, \ldots, v_{n-1}$, that are not to be found in v_n. An immediate consequence of this is if $M_n = 1$ then there will be no useful contribution, and hence we can set $c_n = 0$. Also if M_n has no factors distinct to those of $\frac{v_n}{gcd(N, a, v_1, \ldots, v_n)}$ we can similarly set $c_n = 0$ as our final solution will utilise the 'a' directly.

This method appears to generally produce a solution vector with smaller entries. For comparison we will mention the example given in [Sto97], where $N = 223092870$, and $a = [56039340, 45020850, 114868782, 145800000]$. The algorithm described in the paper produces the solution vector $c = [1, 3, 6, 10]$. Now we can see that $[1, 3, 6, 10] * a = 2338314582$, and $gcd(N, 2338314582) = 6$ as required.

The algorithm we have implemented here produces the solution vector $c = [1, 0, 1, 1]$, (actually $[0, 1, 1]$ as mentioned previously). We then have that $c * a = 316708122$, and the gcd of this value and N is again 6.

We include the GAP code for this function:

```
############################################################
#
#  mgcdex(<N>,<a>,<v>) - Returns c[1], c[2],..., c[k] such that
#    gcd(N,a+c[1]*v[1]+...+c[n]*v[k]) = gcd(N,a,v[1],v[2],...,v[k])
#
BindGlobal("mgcdex", function(N,a,v)
local h,g,M,c,i,d,b,l;
l:=Length(v);  c:=[];  M:=[];  h:=N;

for i in [1..l] do
  g := h;
  h:=GcdInt(g,v[i]);
  M[i]:=QuoInt(g,h);
od;

h:=GcdInt(a,h);          g:=QuoInt(a,h);

for i in [l,l-1..1] do
  b:=QuoInt(v[i],h);
  d:=split(M[i],b);
  if d=1 then
    c[i]:=0;
  else
    c[i]:=rgcd(d,g/b mod d);
    g:=g+c[i]*b;
  fi;
od;

return c;      end);
```

We will also briefly describe here a method by which, given an integer row vector a it is possible to calculate the gcd of that vector, or indeed a vector multiplier v such that

$$\sum_{i=1}^{n} v_i a_i = gcd\left(a_1, a_2, \ldots, a_n\right).$$

By applying the routine described above to the triple $(a_1, a_2, a_{\{3\ldots n\}})$ we can find a vector multiplier c such that

$$gcd\left(a_1, a_2 + \sum_{i=1}^{n-2} c_i a_{i+2}\right) = gcd\left(a_1, a_2, \ldots, a_n\right).$$

We can now apply the extended euclidean algorithm to the pair $\left(a_1, a_2 + \sum_{i=1}^{n-2} c_i a_{i+2}\right)$ and hence easily compute either the gcd or the associated vector multiplier.

In fact, in our implementation we will not need to explicitly calculate such a vector multiplier, but rather we will wish to arrange, by row / column operations, that the gcd of a

given column / row vector appears in a certain position. This routine can be easily adapted for such an arrangement and provides a vector multiplier with generally small entries, so little damage is done to the transforming matrices.

7.1.5 SNF with Transforms

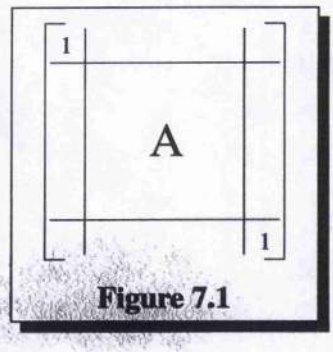


Figure 7.1

We are now ready to put all this together and describe the implementation, in GAP 4.2, that computes the SNF with transforming matrices of an integer matrix A. The first step of the main routine is to embed the input matrix A into a 2 larger identity matrix as described in [Sto96].

This embedding has two main benefits. Firstly it permits the handling of any arbitrary shape of input matrix without the need to check for matrices with no rows or no columns. Secondly, and more importantly, it allows us to apply the main routine without the need to write special code to deal with the edge effects. The first phase of the main routine is to perform row and column operations to convert the matrix to an upper triangular form with the entries of the main diagonal those of the SNF (almost - the final entry actually need not be).

Consider the basic step shown in figure 7.2.

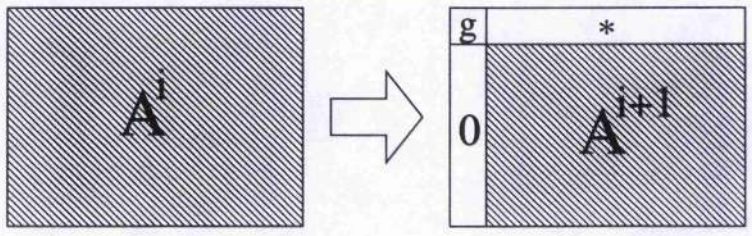


Figure 7.2 : Basic step in producing an upper triangular matrix with diagonal entries equal to those of the SNF; g is the gcd of all the entries of A^i.

It should be clear that by repeatedly applying this procedure we will produce an upper triangular matrix with diagonal entries those of the SNF. Note that the embedding we have already applied means that the initial matrix A^0 is of the correct shape. This basic step can be performed by the following procedure:

- Perform some sequence of column operations to arrange that the gcd of the entries in the first column of A^i is equal to the gcd of all the entries of A^i.
- Perform some sequence of row operations to obtain that gcd and zero out the rest of the column.

The important question in each case is, how do we choose which sequence of row / column operations to apply? We now address this question, firstly for the choice of column operations.

The sequence of column operations we use is simply to add various multiples of columns $2 \ldots m$ to column 1 to arrange that the gcd of column 1 of A^i is equal to the gcd of A^i. The question now becomes how to decide what multiples of the other columns to add to column 1.

The first step is to select a pair of linearly independent columns. We do this by examining the determinant of successive 2×2 submatrices from the two columns; as soon as we discover a non-zero determinant we know that we have found a linearly independent column, and we note the row-column pair for later. Note that due to the embedding such a pair of columns will always exist. We will, for clarity of explanation, assume that the two columns we select are columns 1 and 2 of A^i. Note that in practice, this may not be the case, and we do not in fact bother to permute the columns so that it becomes the case, rather we simply keep track of the columns we are working upon.

Having selected two linearly independent columns we now need to perform some sequence of elementary operations in order that the gcd of the entries in the first of these columns is equal to the gcd of the entire submatrix. We note that it is possible to work modulo N =the gcd of column 2. We could proceed as follows:

- Set $N = gcd(column2)$.
- For each column $j \in \{3 \ldots m\}$
  - Perform row operations to obtain the gcd of column j modulo N in position $A_{1,j}$. i.e. Apply mgcdex to the jth column i.e. $mgcdex(N, A_{1,j}, A_{\{2\ldots n\},j})$.
  - Perform column operations to obtain the gcd of $A_{1,1}$ and $A_{1,j}$ modulo N in position $A_{1,1}$. Again use mgcdex, i.e. $mgcdex(N, A_{1,1}, A_{1,j})$.

Note that this procedure will ensure that column 1 either contain the gcds of columns 3 through m, or entries which divide those gcds. In fact we do not need to actually perform the row operations described in the above sequence. We can simply track the effect that applying the operations would have upon the positions $A_{1,1}$ and $A_{1,j}$, and then the process will continue with each stage being able to work upon far smaller integers, and only upon (copies of) two columns rather than the entire matrix. There is a small proviso that we need to be careful not to change column 1 so that it becomes linearly dependent to column 2. To this extent, if a multiplier is calculated that would cause the determinant of the row-column pair that we calculated earlier to become zero, then the next smallest multiplier is used instead.

All that now remains is to calculate what multiple of column 2 we need to add to column 1 in order that the gcd of column 1 is the gcd of the entire matrix. Consider the following procedure : Perform row operations to obtain the gcd of column 2 in position $A_{1,2}$. Now perform a single column operation to zero out position $A_{1,1}$, i.e. subtract $\frac{A_{1,1}}{A_{1,2}}$ multiples of column 2 from column 1. Now we can let $N = gcd(col1)$, and then we can see that what we wish to do is add some multiple t of column 2 to column 1 such that the gcd of column 1 after the operation is equal to the gcd of column 1 and $A_{1,2}$. i.e. find a t such that

$$gcd(N, A_{1,1}, A_{1,2}) = gcd(N, A_{1,1} + tA_{1,2}).$$

which is easily calculable using the mgcdex routine. Note that we not actually perform these operations, rather we track the effect that finding the gcd of column 2 (denoted by b) will have upon the corresponding entry of the same row in column 1 (denoted by a), we can then calculate N as $gcd(A_{j,1} - \frac{aA_{j,2}}{b})|_j$. These values of a, b and N then permit calculation of the correct multiple of column 2 to add to column 1 to finish this phase of the calculation, arranging that the gcd of column 1 is equal to the gcd of all the entries of A.

We keep track of these column operations in our multiplier matrices. If we are not interested in transformation matrices we actually bypass this entire step and simply reduce the input matrix to row echelon form. It is then passed to an implementation of the algorithm described in [Sto98] which allows fast computation of the SNF triangular integer matrices. However that method does not allow us to easily recover transformation matrices.

We then perform row operations upon the matrix so that the gcd of the top two entries of column 1 is equal to the gcd of column 1 (which is equal to the gcd of the entire matrix A).

Again we use mgcdex, i.e. we calculate a multiplier $t = mgcdex(A_{1,1}, A_{2,1}, A_{[3\ldots n],1})$. Furthermore we ensure at this point that the determinant of the top 2×2 submatrix $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$. is non-zero. If we find that it is the case that the determinant is zero, we add a multiple of another lower row to make the determinant non-zero.

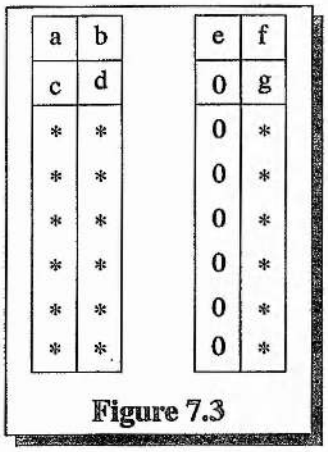


Figure 7.3

This is so that we can proceed with the next step, which is to apply the 2×2 HNF subroutine, to put this top 2×2 section into HNF. We then zero out the entries, below the current row, in column 1, and then reduce the entries below row 2, in column 2 modulo g. This procedure is demonstrated in Figure 7.3, where an * denotes a possibly non-zero entry. Note that the embedding of the input matrix we undertook initially means that the first entry trivially satisfies our criteria, and the first real effect is the reduction of the entries in the first column of our input matrix modulo $A^1_{1,1}$.

We repeat this for each column until we reach the situation where we have an upper triangular form with each successive entry on the diagonal being the corresponding entry of the SNF. Having reached this upper triangular form we can turn our attention to zeroing the off-diagonal entries. We can now make good use of the fact that the entries on the diagonal are those of the SNF. Figure 7.4 demonstrates the situation

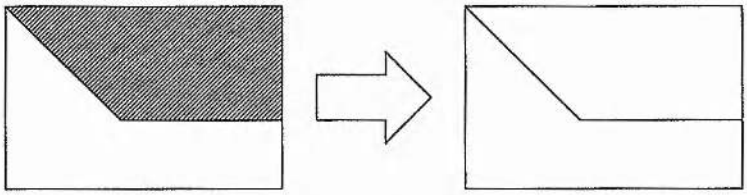


Figure 7.4 : The next step; conversion of upper triangular form to 'almost' SNF.

This conversion is performed by first reducing each off-diagonal element in question modulo the diagonal entry, d_c, in the same column (a row operation), and then reducing it to zero using the diagonal entry, d_r, in the same row (a column operation). We know that $d_c \geq d_r$, and we know that d_r divides all the entries in the submatrix $A[j \ldots n][j \ldots m]$, hence this sequence zeroes each off-diagonal entry, and hopefully keeps the damage to

the multiplier matrices to a minimum. All that then remains is to apply mgcdex and the euclidean algorithm to the last row to obtain the final entry of the SNF and zero off the rest of the tail.

7.2 Performance

In practice this algorithm performs incredibly well, being both very fast and also producing fairly good quality multipliers for the problem of finding the SNF of arbitrary matrices. It is interesting to note here that if we have a matrix with many more rows than columns, or vice versa the performance can be very different depending upon whether we apply the algorithm to the matrix in question, or it's transpose. If we do not require the column transformation multiplier matrix then applying the algorithm to a matrix with a small number of rows and a large number of columns is much faster than applying it to the transpose of that matrix. If we require the column transformation matrix then the opposite is the case. i.e. the algorithm returns a result much faster if we apply it to a matrix with a small number of columns.

Chapter 8

Conclusion and Further Notes

In this chapter we shall first detail the conclusions of this thesis, and then proceed to discuss some aspects that may provide interesting areas for further research.

8.1 Conclusions

In this thesis we have examined the problem of finding good multiplier matrices P and Q such that $PDQ = S$, where D is a diagonal input matrix. We developed, in Chapter 3, practical ways to measure and compare the quality of such solutions. These methods can also be applied to the more general problem $PAQ = S$, where A is an arbitrary matrix. We further showed that there are definite lower bounds for the quality, $\mathcal{Q}(P, Q)$, of these solutions in the case that the input is a diagonal matrix, and found some interesting relationships between various different quality measures.

The basic procedure we investigated was that of converting a diagonal input matrix to Smith Normal Form by a sequence of operations on pairs of entries from the diagonal. This procedure is easily split into two distinct areas, each of which we investigated in this thesis.

- How best to perform each 2×2 step?
- How to select each successive pair?

In Chapter 4 we introduced and developed the concept of the directed graphs associated with the problem of converting a diagonal input matrix to SNF by repeatedly applying

pairwise steps. We investigated the structure of these graphs in certain cases, and saw how we could develop worst case inputs. We further developed the idea of a state space of possible inputs and later utilised this idea to select a representative range of inputs for testing purposes.

We have analysed the 2×2 problem and described explicit matrix multipliers in Chapter 5. We have demonstrated improvements to the usually implemented method and have also provided thoughts upon how best to minimize the impact of these 2×2 multipliers upon the rest of the calculation.

In Chapter 6 we investigated the effects of various heuristic methods of solving the problem in the case of a diagonal input matrix. We discussed two types of algorithms - structural and positional and performed various experiments to investigate the effectiveness of particular algorithms. These experiments allowed us to formulate new positional algorithms which generally produced better quality solutions than we had achieved with the standard implementations.

In chapter 7 we have described an implementation of an algorithm that directly computes the SNF with transforming matrices, and does not proceed via an intermediary diagonal stage. This implementation performs very well in practice.

We believe that a sensible overall approach to the general problem would be to use the "divide into pairs" positional heuristic developed in chapter 6 as a good coarse grain solution to the problem of path selection. We would recommend also using a one step lookahead structural algorithm to decide upon the fine grain structure of exactly which pairs should be used where there is choice. We recommend using a factorization based heuristic to select good multipliers for each of these 2×2 steps, as developed in chapter 5.

8.2 Closing Notes and Further Work

This thesis has laid a solid foundation for further work. We shall note a few potential areas for investigation, and make some brief comments about possible methods therein.

8.2.1 Bounds revisited

By combining the ideas and results of Chapters 5 and 6 we can derive upper bounds for $\mathcal{Q}(P, Q)$ given $PDQ = S$. These bounds assume that we have followed the procedures of

these chapters rigorously.

We know from chapter 4 that each problem has a directed graph associated with it, and from chapter 6 we know that we can find a path through this digraph that has at most $\frac{3n^2}{8}$ steps. Moreover we know that we can minimize the power build up as described in sections 6.3 and 6.3.1 so that the largest power of some bound X occurring is no greater than $\approx \frac{3n}{2}$ for this short sequence of steps.

We also know from chapter 5 that we can put an explicit value on this X, since we can select $|kt| \leq \left|\frac{a}{2}\right|$, and we can always allow either k or t to be 1.

In the case that we set $k = 1$ then the row multiplier matrices are all of the form

$$\begin{pmatrix} \frac{1-bt}{a} & t \\ -b & a \end{pmatrix}$$

where $1 - bt \leq \left|\frac{b}{2}\right|$ and so every entry is bound by $|a|$. Note that for any particular step $a \leq \frac{S_{nn}}{2}$ (we can actually do much better than this if we want to) and so $\|P\|_\infty \leq \left(\frac{S_{nn}}{2}\right)^{\frac{3n}{2}}$. The column multiplier matrices are of the form

$$\begin{pmatrix} 1 & -bt \\ 1 & 1-bt \end{pmatrix}$$

and the largest entry in Q is then bound by $\left|\frac{ab}{2}\right| \leq \left|\frac{a^2}{2}\right|$ giving $\|Q\|_\infty \leq \left(\frac{S_{nn}}{8}\right)^{3n}$.

If we set $t = 1$ then the row multiplier matrices are all of the form

$$\begin{pmatrix} \frac{1-bk}{a} & 1 \\ -bk & a \end{pmatrix}$$

where $1 - bk \leq \left|\frac{b}{2}\right|$ and so every entry is bound by $\left|\frac{ab}{2}\right| \leq \left|\frac{a^2}{2}\right|$ and so $\|P\|_\infty \leq \left(\frac{S_{nn}}{8}\right)^{3n}$. The column multiplier matrices are of the form

$$\begin{pmatrix} 1 & -b \\ k & 1-bk \end{pmatrix}$$

and the largest entry in Q is then again bound by $\left|\frac{ab}{2}\right| \leq \left|\frac{a^2}{2}\right|$ giving $\|Q\|_\infty \leq \left(\frac{S_{nn}}{8}\right)^{3n}$.

These bounds are not very good. We can get improvements by noting for example that by using the divide into pairs with intelligent collection of gcd and lcm, the entries in the 2×2 multiplier matrices for at most $\frac{n}{2} + log(n)$ operations will be bound by S_{nn}. Thereafter the entries will be bound by S_{n-1n-1} for the next few operations and so on. Further improvements along these lines are clearly possible.

8.2.2 Parallelisation

The algorithms to find multiplier matrices for the SNF of a diagonal matrix that we have described in this thesis mostly permit simple parallelisation. For the positional algorithms it is clear that we can select a path through the digraph before we embark upon any further calculation, i.e. we know in advance all the diagonals that will arise along the way and hence we could set the task of finding good 2×2 multiplier matrices for each of these steps running in parallel on as many processors as required (or available). For the majority of the structural algorithms the same is true - we can precalculate the path and hence spread the computationally harder work across parallel processors. Of course, this is not possible for those algorithms wherein we make choices according to the current state of the multiplier matrices.

Pseudocode :

- Calculate a path.
- Parallel : for each step calculate (best) multiplier matrices.
- Parallel : multiply matrices together.

Assuming path length, P, the matrix multiplication can be done using $P/2, P/4, P/8...$ processors in $log_2(P)$ steps.

Also note that depending on exactly how we are finding best multiplier matrices, this step itself could possibly be parallelised further in some fashion, e.g. by splitting the search space across different processors.

8.2.3 Improving the Solution

Assuming we have found matrices P and Q such that $PAQ = S$ is in SNF (and where A need have no special form), then it is possible in some cases to improve the solution we have obtained. If X, Y are unimodular matrices such that $XSY = S$ then

$$(XP)A(QY) = XSY = S.$$

and since S_{ii} divides $S_{jj} \forall i \leq j$ then we can add any multiple, k, of row j to row i (i $\leq$ j) if we also add $-k\frac{S_{jj}}{S_{ii}}$ times column i to column j, to zero out the S_{ij} entry.

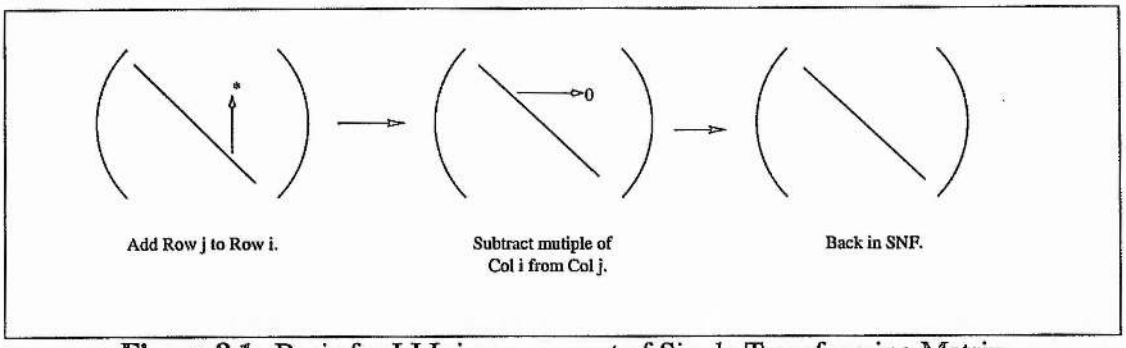


Figure 8.1: Basis for LLL improvement of Single Transforming Matrix

We can either track changes made by this to both transforming matrices, or if for example the column transform Q is required and we only know the row transform, P then we can easily calculate

$$Q := (PA)^{-1}S.$$

Much of the time we are interested only in minimizing one of the transforming matrices, and some experiments suggest that this process can be used to improve the quality of one of the matrices, but at significant cost to the quality of the other. Further investigation of this idea may prove informative.

8.2.4 Average SNF

A question that has frequently arisen during the course of this research has been "What is an average problem?" There is of course no sensible answer to this in general, as many of these matrices arise from algebraic problems with a certain amount of inherent structure. However we can consider the completely general problem of finding the SNF of a random matrix , distinct from any external environment. In this case there are two sensible starting points we can consider :

- The SNF of a matrix of randomly chosen integers.
- The SNF of a diagonal matrix of randomly chosen integers.

In either case we could apply the following reasoning to investigate the question of what constitutes an 'average' SNF. Given an integer matrix,A with k non-zero entries the first

entry in the SNF is the gcd of all of the entries of A. The probability that this is one is

$$\alpha(k) := 1 - \sum_{c=1}^{\infty} \left(\frac{1}{c^k}\right).$$

If we make the assumption that after the row and column operations required to isolate the first entry of the SNF we are left with an essentially random matrix, with one less row and one less column than the original, then we can make a similar estimate for the next entry of the SNF and so on. This method could produce some results about what constitutes an 'average' SNF. The same idea could be applied to a diagonal matrix, i.e. successive lists of essentially random integers of length $n, n-1, \ldots, 2$ to build up an idea of what constitutes an average SNF in this setting.

We are actually more interested in the average difficulty of a problem, i.e. how many steps it takes to produce an SNF in general. We can get an idea of the 'average' complexity by examining the coprime parts of a particular random diagonal matrix. This is a little harder to estimate theoretically, but we can get an idea of the height and weight of random cases by running some experiments. It is interesting to note that we can put immediate bounds upon the height and weight simply from the size of the numbers appearing in the list, although it is likely these will be gross overestimates. By applying the rewriting procedure described in chapter 4 we can examine the height and weight of random lists.

Taking 1000 diagonals of lengths 3 to 20 of randomly chosen integers from the range 1 to 2^{27}, and examining the coprime parts of each of these diagonals reveals that the distribution of height and weight forms a bell curve in each case. The number of distinct coprime parts appearing in a given 'random' diagonal of length n appears to peak at about $\frac{3n}{2}$, whilst the weight appears to be a wider,flatter bell curve with peak increasing as $nlog(n)$. This is clearly a far cry from the worst case scenario described in chapter 4.

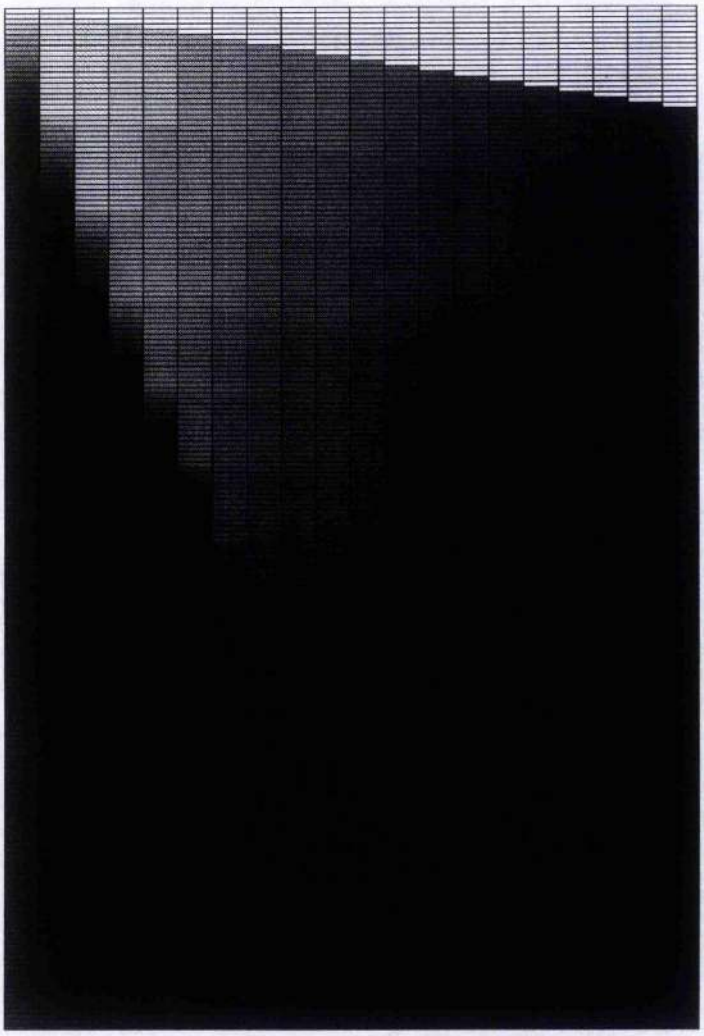
We could also examine the 'complexity' of these random problems by examining the distribution of path lengths through the digraph. However this is difficult, even if we restrict our attention to the shortest paths through each of the digraphs. Preliminary results suggest that the shortest path through the digraph of a random length n input diagonal generally consists of $n-1$ steps i.e. coprime. This is unsurprising given the above probability that k integers are coprime. It would be a useful improvement to be able to analyse in advance the likely complexity of a problem, in order to better select a method of obtaining the multiplier matrices. We leave this as an open problem.

Appendix 1 - Powergrowth Pictures

The following pictures demonstrate graphically the power growth described in section 6.3.1 upon a worst case problem of length 20.

The shading of each picture is individually scaled such that black represents the largest power occurring in the application of that algorithm. White represents a power of zero and the greyscale represents all the values between, with the darker colours representing higher powers.

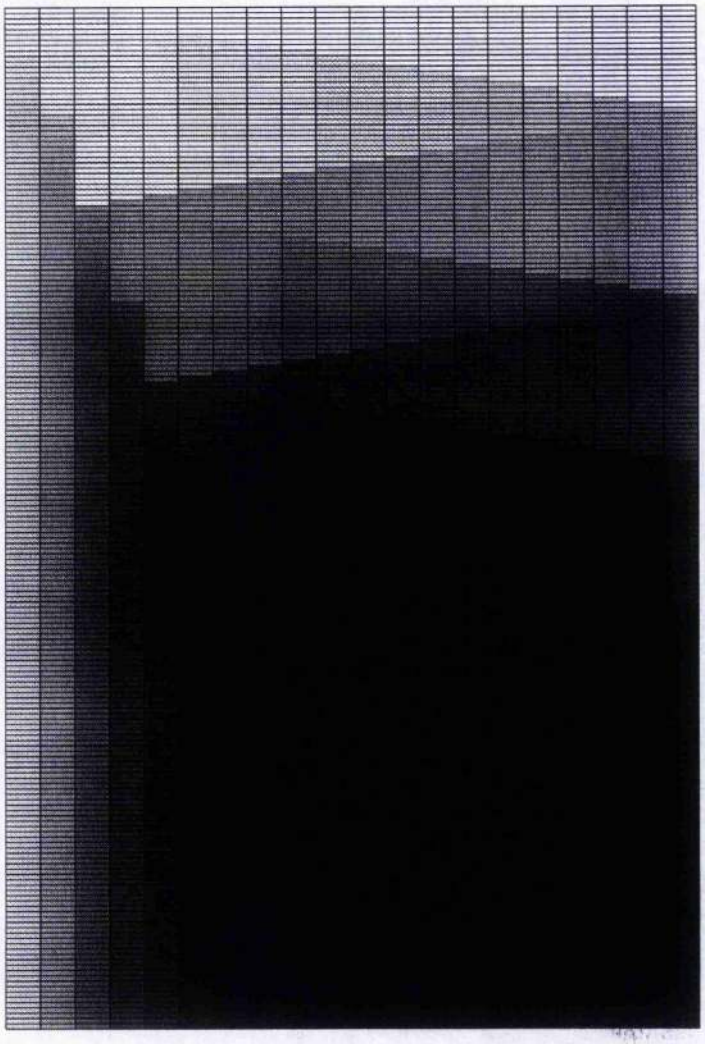
Each row of each picture shows the powergrowth at a particular step in the calculation. The top row of each picture is the powergrowth at the beginning, i.e. all zeroes. The bottom row demonstrates the final powergrowth distribution. Recall that each algorithm can take a different number of steps to complete, and so the number of rows in each of these pictures differs.



Standard algorithm power growth

Here the largest power, represented by the darkest shade, is 37. The final sequence of powers is

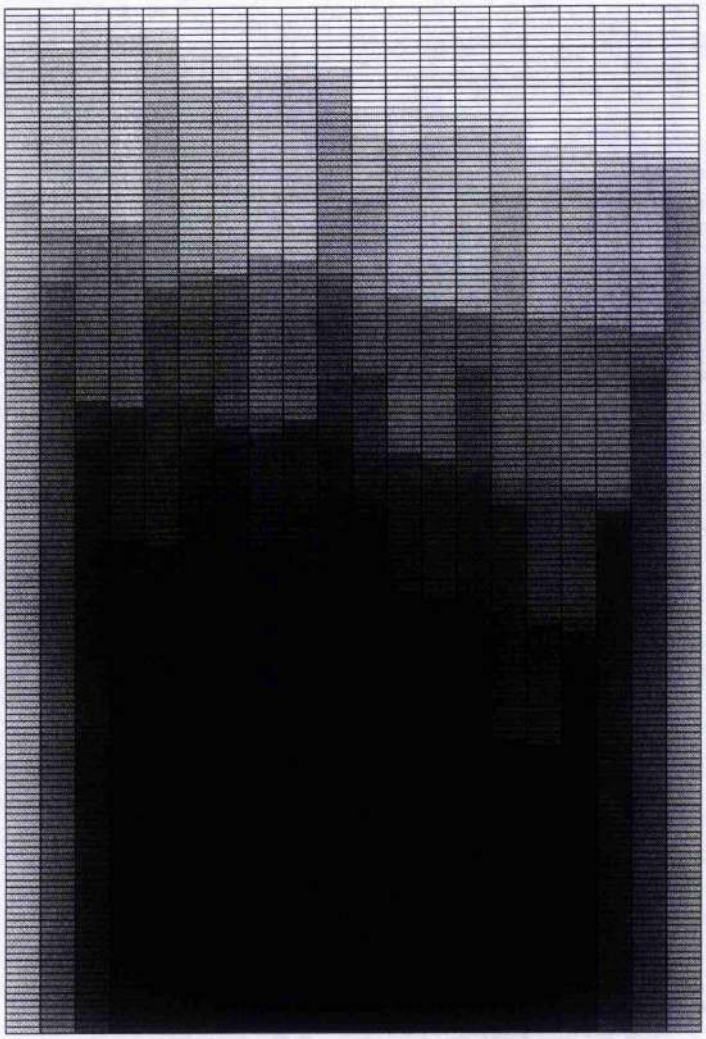
$$[19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 37].$$



Worst case algorithm power growth

Here the largest power, represented by the darkest shade, is 190. The final sequence of powers is

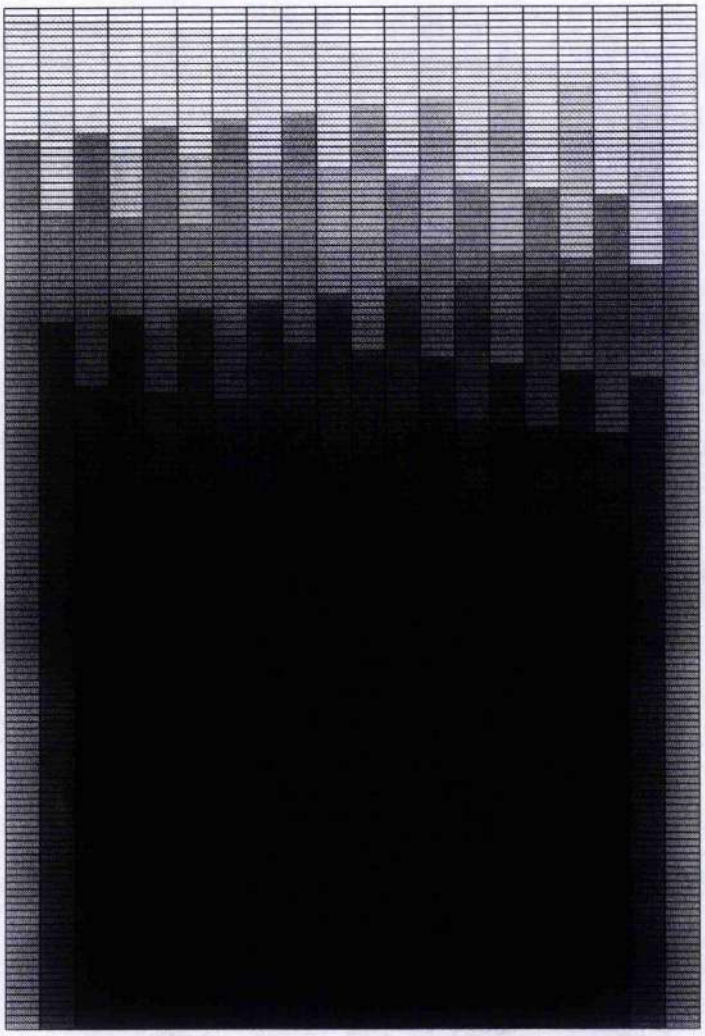
$$[19, 37, 54, 70, 85, 99, 112, 124, 135, 145, 154, 162, 169, 175, 180, 184, 187, 189, 190, 190].$$



Divide and Conquer algorithm power growth

Here the largest power, represented by the darkest shade, is 38. The final sequence of powers is

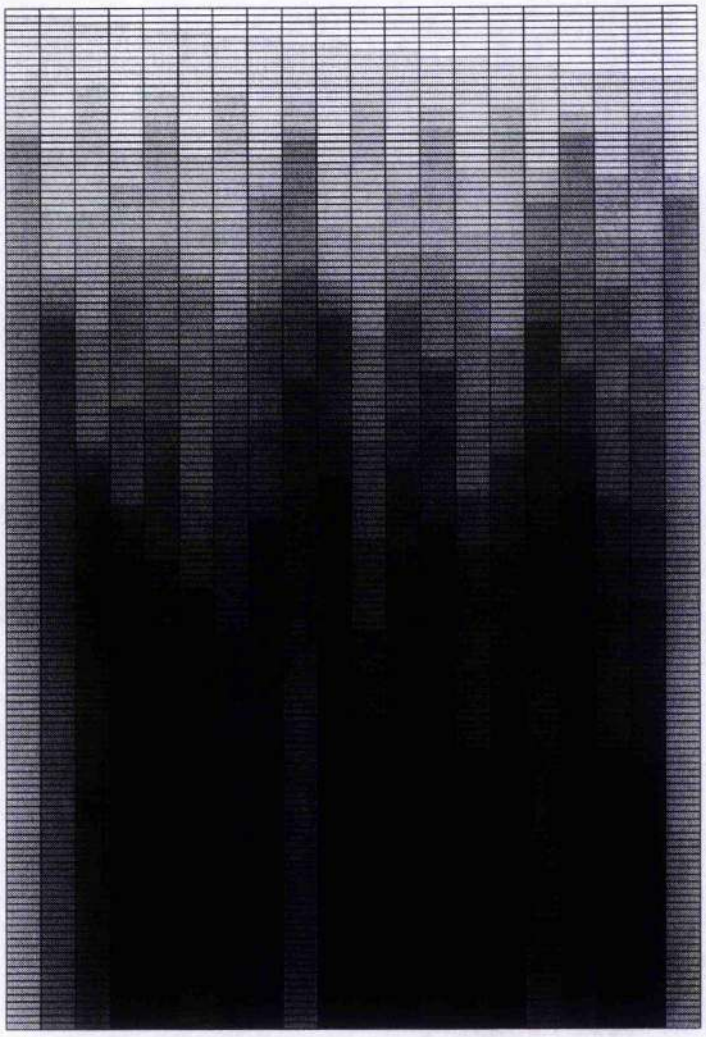
$$[5, 10, 15, 19, 23, 27, 31, 34, 37, 38, 38, 37, 35, 31, 28, 24, 20, 15, 11, 6].$$



Divide into pairs- naive collection strategy, power growth

Here the largest power, represented by the darkest shade, is 55. The final sequence of powers is

$$[10, 19, 27, 34, 40, 45, 49, 52, 54, 55, 55, 54, 52, 49, 45, 40, 34, 27, 19, 10].$$



Divide into pairs- Intelligent collection strategy, power growth

Here the largest power, represented by the darkest shade, is 25. The final sequence of powers is

$$[5, 8, 11, 24, 25, 14, 20, 20, 8, 18, 16, 22, 16, 18, 24, 11, 14, 22, 25, 5].$$

Appendix 2 - daVinci

During the course of this research we desired to understand the properties of the digraph structure generated by an input. We found it very useful to be able to generate pictures and to this purpose we made use of the daVinci system [Wer98]; indeed the figures in chapter 4 were on the whole produced using daVinci. We will briefly describe here some of the capabilities of that system, and some of the tools we developed in GAP to make better use of it.

daVinci is an interactive tool to visualize directed graphs. A graph is a structure with a number of objects (nodes) and relationships between them (edges). For directed graphs, all edges have a direction, i.e. for each edge there is a parent (the source) and a child node (the target). The graph layout in daVinci reflects these hierarchical relationships by arranging the nodes at horizontal levels such that all parent nodes are above their child nodes and all edges point downwards (in a top-down layout; it is possible to arrange other layouts e.g. left-to-right, and in fact we generally use this layout in preference). Further, the direction of an edge is usually visualized with an arrow pointing to the child node. This kind of representation is called hierarchical visualization of a directed graph.

Graphs are loaded in daVinci using a format called term representation. The term representation format supports all kind of directed graphs: cyclic or acyclic graphs, empty graphs, graphs with only one level (a list of nodes without any edges), multi-edges (two or more edges between two nodes) or even self-edges (edges where the parent and child node are the same). The term representation is a plain text ASCII format, so daVinci graphs can even be created with an arbitrary text editor. But normally, one will not do this by hand. Instead, graphs are usually generated automatically by some application program, in our case directly from GAP.

daVinci Term Representation

In general, a term is a structure where a superterm (parent) encloses its subterms (children), e.g. parent[child1,child2,child3]. Brackets [...] are used to get a list of comma-separated elements of the same type. This scheme of expressing parent-child relationships can be applied recursively, so each child may have its own children, and so on. Such a notation allows to represent arbitrary tree structures.

To specify graphs, a mechanism of identifiers and references is used in daVinci. For example, if a child node has more than one parent node, then in the term representation the corresponding subgraph of the child appears in only one of the parents as a subterm. This subterm is marked with an identifier (in fact, all nodes and edges need to be marked with a unique identifier). All the other parents of the same child do not duplicate the subterm. Instead, they point to the child by using a reference to the identifier. Note that this allows the description of cyclic graphs. When loading a term representation, daVinci will construct an internal graph by resolving these references. The linear order of a node's subterm (where the identifier is declared) and a reference to this node (where the identifier is used) is arbitrary in a term representation, so references can be used before the corresponding identifier and subterm appears in the term representation.

Beside the (unique) identifier and the list of child nodes, each node also has a type and a list of attributes which are responsible for the image of a node in the visualization. Between a parent and the corresponding child node, there is an edge in the term representation which also has its unique edge identifier, type and attributes. So, in fact the children of a node are edges and each edge has one node or reference as subterm.

For instance the following term representation was used to generate the directed graph in figure 4.1.2,

```
[l("[ 4, 6, 9 ]",n("node",[a("OBJECT","[ 4, 6, 9 ]")],[l("1_2",e("edge",[],l("[ 2, 9, 12 ]
",n("node",[a("OBJECT","[ 2, 9, 12 ]")],[l("2_5",e("edge",[],l("[ 1, 12, 18 ]",n("node
",[a("OBJECT","[ 1, 12, 18 ]")],[l("5_3",e("edge",[],l("[ 1, 6, 36 ]",n("node",[a("OBJECT
","[ 1, 6, 36 ]")],[])))))))))),l("2_6",e("edge",[],l("[ 2, 3, 36 ]",n("node",[a("OBJECT",
"[ 2, 3, 36 ]")],[l("6_3",e("edge",[],r("[ 1, 6, 36 ]")))]))))))))),l("1_3",e("edge",[],r(
"[ 1, 6, 36 ]"))),l("1_4",e("edge",[],l("[ 3, 4, 18 ]",n("node",[a("OBJECT","[ 3, 4, 18 ]")],
[l("4_5",e("edge",[],r("[ 1, 12, 18 ]"))),l("4_6",e("edge",[],r("[ 2, 3, 36 ]")))]))))))))))]
```

This term representation string was created by starting with the input node [4,6,9] and generating, in GAP, a list of vertices in the digraph, and a list of edges. These are then easily utilised to form the adjacency matrix, each row of which can be labelled (uniquely)

by associating with it the diagonal to which it corresponds. It is then a reasonably simple procedure to create the string above by a procedure which, keeping track of the nodes that have already been added, works upon each node in sequence and either adds all children therefrom, or in the case that the node already appears, simply appends the node identifier and allows daVinci to resolve the internal references and build the graph.

Bibliography

[BS96a] E. Bach and J. Shallit. *Algorithmic Number Theory, Volume 1: Efficient Algorithms*, chapter 4, pages 84–90. The MIT Press, 1996.

[BS96b] E. Bach and J. Shallit. *Algorithmic Number Theory, Volume 1: Efficient Algorithms*, chapter 4, pages 70–71. The MIT Press, 1996.

[Her51] C. Hermite. Sur l'introduction des variables continues dans la theorie des nombres. *J. Reine Angew. Math.*, 41:191–216, 1851.

[HHR93] G. Havas, D.F. Holt, and Sarah Rees. Recognizing badly presented $\mathbb{Z}$-modules. *Linear Algebra Appl.*, 192:137–163, 1993.

[HM97] G. Havas and B.S. Majewski. Integer matrix diagonalization. *Journal Symbolic Computation*, 24:399–408, 1997.

[HS79] G. Havas and L.S. Sterling. Integer matrices and abelian groups. *Lecture Notes in Comput. Sci.*, 72:431–451, 1979.

[Joh90] D.L. Johnson. *Presentations of Groups*. Cambridge University Press, 1990.

[MH94] B.S. Majewski and G. Havas. The complexity of greatest common divisor computations. In *Algorithmic Number Theory, LNCS 877*, pages 184–193, 1994.

[S+95] Martin Schönert et al. *GAP – Groups, Algorithms, and Programming*. Lehrstuhl D für Mathematik, Rheinisch Westfälische Technische Hochschule, Aachen, Germany, fifth edition, 1995.

[Sim94] C. C. Sims. *Computation with Finitely Presented Groups*. Cambridge University Press, 1994.

[Smi61] H.J.S Smith. On systems of linear indeterminate equations and congruences. *Philos. Trans. Roy. Soc. London*, 151:293–326, 1861.

[Sto96] A Storjohann. A fast+practical+deterministic algorithm for triangularizing integer matrices. Technical Report Tech. Rep 256, Departement Informatik, ETH Zurich, Dec. 1996.

[Sto97] A Storjohann. A solution to the extended gcd problem with applications. In *Int'l. Symp. on Symbolic and Algebraic Computation : ISSAC '97*, 1997.

[Sto98] A Storjohann. Computing hermite and smith normal forms of triangular integer matrices. *Linear Algebra and its Applications*, 282:25–45, 1998.

[Tie08] H. Tietze. über die topologischen invarienten mehrdimensionaler mannigfiltigkeiten. *Monatsh. für Math. und Phys.*, 19:1–118, 1908.

[Wer98] M. Werner. davinci v2.1.x online documentation.

'http://www.tzi.de/~davinci/doc_V2.1/' , 1998.