Pushing Points, and Curves on Grid:
Geometric Characterization and Algorithms

Shah Rushin Navneet

Pushing Points, and Curves on Grid: Geometric Characterization and Algorithms

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of

Master of Technology
in
Computer Science and Engineering

by

Shah Rushin Navneet

03CS3012

under the guidance of

Dr. Arijit Bishnu

Department of Computer Science and Engineering
Indian Institute of Technology
Kharagpur
May, 2008

CERTIFICATE

This is to certify that the thesis entitled **Pushing Points, and Curves on Grid: Geometric Chracterization and Algorithms**, submitted by **Shah Rushin Navneet**, to the department of Computer Science and Engineering in partial fulfillment for the award of the degree of **Master of Technology**, is a bonafide record of work carried out by him under my supervision and guidance. The thesis has fulfilled all the requirements as per the regulations of this institute and, in my opinion, has reached the standard needed for submission.

Dr. Arijit Bishnu
Dept. of Computer Science & Engg.
Indian Institute of Technology,
Kharagpur — 721302, INDIA.
May, 2008.

Acknowledgments

I wish to thank my parents. I would like to thank my guide Dr. Arijit Bishnu for his help and support. I also wish to thank my faculty advisor Dr. Abhijit Das and Dr. Indranil Sengupta. Finally, my sincere thanks to all the faculty members of the Department of Computer Science & Engineering at IIT Kharagpur.

Shah Rushin Navneet
Department of Computer Science and Engineering
Indian Institute of Technology
Kharagpur 721302, India
May, 2008

Contents

1 Moving Red and Blue Cells on a Checkerboard with Empty Spaces **1**
1.1 Motivation 1
1.2 Problem Definition 3
1.3 Lower Bound for N_e Required to Empty a Matrix 6
1.4 Complexity of The Problem 9
1.5 Non-Optimal Transitions 11
1.6 Optimal Algorithm 14
1.7 Proof of Optimality 19
1.8 Implementation and Results 21
1.9 Conclusion 26

2 Digital Curve on Grid: Curve Fitting using Linear Programming **29**
2.1 Introduction 29
2.2 Problem Definition 30
2.2.1 Linear Programming 30
2.3 Formulation of Curve Fitting as Linear Programming 32
2.4 Implementation and Solutions 37
2.5 Alternative Formulation 38
2.6 Conclusion 39

References **42**

List of Tables

1.1 Results for the case $4 \times 4, N_e = 1$ 23
1.2 Results for the case $7 \times 7, N_e = 1$ 24
1.3 Results for the case $7 \times 7, N_e = 5, 10$ 25
1.4 Results for $N = 10, 20 \ldots 50, N_e z = 1, 0.05 \times N^2 \ldots 0.25 \times N^2$ 25
1.5 Results for $N = 60, 70 \ldots 100, N_e z = 1, 0.05 \times N^2 \ldots 0.25 \times N^2$ 25

List of Figures

1.1 $N = 10$ . . . 26
1.2 $N = 20$ . . . 26
1.3 $N = 30$ . . . 26
1.4 $N = 40$ . . . 26
1.5 $N = 50$ . . . 27
1.6 $N = 60$ . . . 27
1.7 $N = 70$ . . . 27
1.8 $N = 80$ . . . 27
1.9 $N = 90$ . . . 28
1.10 $N = 100$ . . . 28
1.11 $N_e = 0.10 \times N^2$ . . . 28

2.1 $N = 5$, Initial Set . . . 37
2.2 $N = 5$, Target Sequence . . . 37
2.3 $N = 7$, Initial Set . . . 37
2.4 $N = 7$, Target Sequence . . . 37
2.5 $N = 10$, Initial Set . . . 38
2.6 $N = 10$, Target Sequence . . . 38
2.7 $N = 12$, Initial Set . . . 38
2.8 $N = 12$, Target Sequence . . . 38
2.9 $N = 15$, Initial Set . . . 39
2.10 $N = 15$, Target Sequence . . . 39
2.11 $N = 20$, Initial Set . . . 39
2.12 $N = 20$, Target Sequence . . . 39

Chapter 1

Moving Red and Blue Cells on a Checkerboard with Empty Spaces

1.1 Motivation

Many real-world problems can be modeled as matrices consisting of C different types of cells. We refer to such matrices as C coloured matrices. In addition to the C colours, a cell might also be of no colour at all, a situation where we refer to the cell as being empty. For all such matrices with various values of C, of particular interest to us is the case $C = 2$.

The most obvious examples of these matrices are **microarrays** [2]. We consider a particular sub-type: *DNA* microarrays. These are small, solid supports onto which the sequences from thousands of different genes are immobilized, or attached, at fixed locations. Microarrays are very important in the study of areas such as gene expression, genomic gains and losses, as well as mutations in *DNA* [3]. They allow scientists to analyze expression of many genes in a single experiment quickly and efficiently. Microarray technology is used to try to understand fundamental aspects of growth and development as well as to explore the underlying genetic causes of many human diseases.

For an example of how microarrays are used in studying gene expression, consider two cells: cell type 1, a healthy cell, and cell type 2, a diseased cell. For each cell type, scientists isolate messenger *RNA* (*mRNA*) from the cells and use this *mRNA* as a template to generate a strand of *DNA* complementary to the original *DNA* of the cell. This type of newly produced *DNA* is known as *cDNA* [5]. Different fluorescent tags are attached to the *cDNA* produced from the different cell types, so that the samples can be differentiated in subsequent steps. The two labeled samples are then mixed and incubated with a microarray. The labeled molecules bind to the sites on the array cor-

responding to the genes expressed in each cell. This microarray is then scanned in a microarray scanner to visualize fluorescence of the two fluorophores [4]. Thus it can be represented as a matrix with colour $C = 2$. The relative intensities of each fluorophore may then be used in ratio-based analysis to identify up-regulated and down-regulated genes.

Many sophisticated statistical algorithms need to be performed on microarrays in order to analyze the vast amounts of biological data present on them. These include permutation analysis, *k-means* clustering, t-tests, etc. The results of these operations correspond to important biological data, for example, the level of expression of a gene, or the sequence of nucleotides that are part of a gene, or finding the centers of gene expression [2]. One would also like to extract specific gene sequences, which is the same as extracting specific coloured chains.

Other examples of such matrices include imeages consisting of only black and white pixels in image processing, as well as memory storage devices, in which cells correspond to individual memory locations, and cell transitions represent applications of different voltages at different cells.

There are many tasks we wish to perform efficiently in these situations, and this problem corresponds to finding efficient algorithms for manipulating the cells of such matrices. Such manipulations might include, as mentioned earlier, *k-means* clustering, t-tests, and also sorting the cells, permuting them to achieve or to remove homogeneity, or emptying cells of a particular colour without affecting those of other colours, using specially designated receptor cells. The previous work done in this area indicates that any two configurations with the same distribution of cells of different colours can be transformed into each other by a sequence of moves so that all intermediate configurations are connected [1]. We consider the problem of emptying all cells from a matrix with colour $C = 2$, using the minimum possible number of transitions.

1.2 Problem Definition

We are given an $M \times N$ matrix of cells. Each such cell can have 3 possible values:

- R - Red
- B - Blue
- E - Empty

Let N_r denote the number of red cells, N_b denote the number of blue cells and N_e denote the number of empty cells.

Then $M \times N = N_r + N_b + N_e$

We denote the cell in the ith row and j^{th} column as (i, j).

For a cell (i, j), we define its neighbourhood to be the cells $\{(i - k_1, j - k_2) \mid k_1, k_2 \in \{-1, 0, 1\}, i - k_1 \in \{1, 2, \ldots N\}, j - k_2 \in \{1, 2, \ldots N\}, k_1$ and k_2 not both $0\}$.

A cell with value E can exchange its value with any of its neighbouring cells. A cell with value R or B can exchange its value only with a neighbouring E cell.

Such exchanges are achieved in real-world examples, such as microarrays, with the help of different voltages being applied at different positions of the microarray.

An R receptor is located outside the matrix, adjacent to the cell $(1, 1)$. Any R in this cell is removed by the R receptor, and hence can be replaced in cell $(1, 1)$ by an E.

Similarly, a B receptor is located outside the matrix, adjacent to the cell $(1, N)$. Any B in this cell is removed by the B receptor, and hence can be replaced in cell $(1, N)$ by an E.

Thus, these receptors provide a mechanism for emptying R and B cells from the matrix.

For example, consider the following matrix and its associated receptors:

$$A = \begin{matrix} R \sqsupset \\ \\ \\ \\ \end{matrix} \begin{pmatrix} E & B & B & R \\ B & R & R & B \\ B & B & R & B \\ R & R & B & R \end{pmatrix} \begin{matrix} \sqsubset B \\ \\ \\ \\ \end{matrix}$$

Here, $M = 4, N = 4, N_e = 1, N_r = 7, N_b = 8$

For the sake of simplicity, we assume from now onwards that $M = N$. We do so because this assumption does not affect the complexity of emptying the matrix, nor does it affect the complexity of the algorithm that we propose to do so. We shall omit representations of the R and B receptors and implicitly assume their existence unless stated otherwise.

Without the receptors shown, a typical example case looks like:

$$A = \begin{pmatrix} E & B & B & R \\ B & R & R & B \\ B & B & R & B \\ R & R & B & R \end{pmatrix}$$

Within this framework, we examine a number of interesting problems. For example, consider the problem of finding a lower bound on the number of E cells required in the matrix, in order to ensure that all the non - E cells can be emptied. We prove that this lower bound is $N_e = 1$, i.e. even if the matrix contains just 1 E cell, it is possible to empty all the R and B cells from it, using this E cell.

The proof for this particular lower bound is constructive in the sense that it also yields a strategy to empty all the coloured cells from the matrix. However, this strategy doesn't account for the possibility that emptying cells might become progressively easier as the number of E cells in the matrix increases. Hence, in order to devise an optimal algorithm, we must consider a strategy that takes advantage of the inherent structure of the problem.

It is our intuition that the problem of emptying all coloured cells from the matrix requires $\Omega(n^3)$ time. We produce a proof for this particular lower bound.

Having obtained such a proof, we describe an algorithm that empties the entire matrix, which in addition to having optimal asymptotic time complexity, also minimizes the number of cell transitions required to achieve its task. We also prove the optimality of this algorithm.

In addition, we have discussed our implementation of this algorithm, and its application to matrices of various sizes and distributions. We start with a fixed number of empty cells N_e and vary the matrices according to the difficulty of emptying them

greedily. We then repeat this procedure over different values of N_e. We have presented the results obtained for these cases and tried to infer the influence of factors such as N_e and the distribution of R and B cells on the cost of emptying the matrix.

1.3 Lower Bound for N_e Required to Empty a Matrix

We wish to prove that if $N_e \geq 1$, all the coloured cells in the matrix can be emptied.

In order to prove this, if we can give a procedure for 1 E to remove a coloured cell from any hardest possible position X, which does not depend on removing any cells prior to removing the cell at X, we can use that procedure to remove all other coloured cells subsequently.

Let us elaborate on the notion of hardest possible position. Consider the following 4×4 matrix:

$$R \sqsupset \begin{pmatrix} E & B & B & R \\ B & B & B & B \\ B & B & B & B \\ B & B & B & R \end{pmatrix} \sqsubset B$$

The B receptor is blocked by an R cell, so no B cells can be emptied in this configuration.
Now, consider the R cell at position (N, N):

- This R is separated from the cell $(1, 1)$ adjacent to the R receptor by only B cells.
- Moreover, the lone E cell in the matrix is the maximum possible distance away from this R cell, i.e. at cell $(1, 1)$.

It is clear that if we can show a procedure to remove this R cell, which doesn't depend on removing any cell prior to it, we can then use this procedure to empty any remaining R or B cell from the matrix, no matter how many opposite coloured cells separate it from its own receptor or how far it is from the nearest E cell. Thus we can see that this R cell at (N, N) in this example matrix is at a hardest possible position, and providing a strategy for removing it, without requiring the removal of any other cell, is equivalent to proving that 1 E cell suffices to empty any given matrix.

Lemma 1 *For any given matrix A, $\exists$ always a strategy for emptying A, if $N_e \geq 1$*

Proof: Consider the following 4×4 matrix A:

$$A = \begin{matrix} R \sqsupset \\ \\ \\ \\ \end{matrix} \begin{pmatrix} E & B & B & R \\ B & B & B & B \\ B & B & B & B \\ B & B & B & R \end{pmatrix} \begin{matrix} \sqsubset B \\ \\ \\ \\ \end{matrix}$$

Here, the R cell at $(4, 4)$ is at the hardest possible position to empty.

Now, for a 2×2 subset

$$\begin{pmatrix} R & B \\ x & E \end{pmatrix}$$

of some given matrix, where X represents the "don't care" condition and can be R or B or E, consider the following series of transitions:

$$\begin{pmatrix} R & B \\ x & E \end{pmatrix} \longrightarrow \begin{pmatrix} R & E \\ x & B \end{pmatrix} \longrightarrow \begin{pmatrix} E & R \\ x & B \end{pmatrix} \longrightarrow \begin{pmatrix} B & R \\ x & E \end{pmatrix}$$

These transitions prove that any RB duo can be transformed to a BR duo and vice-versa using only 1 adjacent E cell, and without disturbing any other cell.

We denote any transformation of this type as $\frac{RB}{E} \leftrightarrow \frac{BR}{E}$.

Now, consider the following series of transitions:

$$\begin{pmatrix} E & B & B & R \\ B & B & B & B \\ B & B & B & B \\ B & B & B & R \end{pmatrix} \longrightarrow \begin{pmatrix} B & B & B & R \\ B & B & B & B \\ B & B & E & B \\ B & B & B & R \end{pmatrix} \longrightarrow \begin{pmatrix} B & B & B & R \\ B & B & B & B \\ B & B & E & R \\ B & B & B & B \end{pmatrix}$$

$$\longrightarrow \begin{pmatrix} B & B & B & R \\ B & B & E & B \\ B & B & B & R \\ B & B & B & B \end{pmatrix} \longrightarrow \begin{pmatrix} B & B & B & R \\ B & B & E & B \\ B & B & R & B \\ B & B & B & B \end{pmatrix} \longrightarrow \begin{pmatrix} B & B & B & R \\ B & B & E & B \\ B & R & B & B \\ B & B & B & B \end{pmatrix}$$

$$\longrightarrow \begin{pmatrix} B & B & B & R \\ B & R & E & B \\ B & B & B & B \\ B & B & B & B \end{pmatrix} \longrightarrow \begin{pmatrix} B & B & E & R \\ B & R & B & B \\ B & B & B & B \\ B & B & B & B \end{pmatrix} \longrightarrow \begin{pmatrix} B & R & E & R \\ B & B & B & B \\ B & B & B & B \\ B & B & B & B \end{pmatrix}$$

$$\longrightarrow \begin{pmatrix} B & R & B & R \\ E & B & B & B \\ B & B & B & B \\ B & B & B & B \end{pmatrix} \longrightarrow \begin{pmatrix} R & B & B & R \\ E & B & B & B \\ B & B & B & B \\ B & B & B & B \end{pmatrix} \longrightarrow \begin{pmatrix} E & B & B & R \\ E & B & B & B \\ B & B & B & B \\ B & B & B & B \end{pmatrix}$$

Thus we have removed the R cell at the hardest possible position (N, N), and initially separated from its receptor entirely by B cells, using a sequence of $\frac{RB}{E} \leftrightarrow \frac{BR}{E}$ transitions. Hence every other R or B cell in A can be removed using a sequence of such transitions. Also, it is clear that all the R or B cells in any other given matrix where $N_e \geq 1$ can be removed using similar sequences of $\frac{RB}{E} \leftrightarrow \frac{BR}{E}$ transitions. □

It is also apparent upon inspection that removing an R or B cell in this manner takes $O(N)$ time. In many cases, a coloured cell will have a clear path of E cells to its receptor, and will not be separated thus by oppositely coloured cells, and emptying it will require fewer transitions. Thus the procedure we have shown accounts for the worst case scenario.

1.4 Complexity of The Problem

For a cell at position (i, j), at least $\lceil \sqrt{i^2 + j^2} \rceil$ transitions will be required to empty it. Thus, the minimum number of transitions required to empty all such cells is given by:

$$S = \sum_{i=1}^{N} \sum_{j=1}^{N} \lceil \sqrt{i^2 + j^2} \rceil$$

The order of this double summation S is the worst case lower bound of the problem of emptying all cells from the matrix.

Let $S' = S + T$, where $T = \sum_{i=1}^{N} \sqrt{i^2 + i^2}$

Now, $S \geq N^2$, so adding T to S will not change the order of S.

Hence $S = \theta(S')$.

It is trivial to see that $S = O(N^3)$:

$S \leq \sqrt{2} \sum_i \sum_j \sqrt{\text{ max } (i,j)^2}$

$S \leq 2\sqrt{2}N * N + (N-1) * (N-1) + \ldots$

$S = 2\sqrt{2} \sum N^2$

Hence, $S = O(N^3)$

However, we are much more interested in obtaining a *lower bound* for S. Consider:

$S' \geq \sqrt{2} \sum_{i,j} \sqrt{\text{ min } (i,j)^2} + \sum_{t=1}^{N} \sqrt{t^2 + t^2}$

$S' \geq 2\sqrt{2}N * 1 + 2 * (N-1) + 3 * (N-2) \ldots$

$S' = \Omega(\sum_{r=1}^{N} r(N-r))$

$S' = \Omega(N^3)$

Thus, $S = \Omega(N^3)$ □

(We have proved earlier that $S = O(N^3)$, hence, $S = \theta(N^3)$)

Thus, we have shown that the complexity of the problem of emptying all the cells from the matrix has a lower bound $\Omega(N^3)$. There are many algorithms that succeed in performing the task with asymptotically optimal time complexity. For example, consider the following algorithms:

Algorithm 1

---

while $\exists$ any R or B cell in the matrix **do**
  Empty the closest R or B cell by brute force, using $\frac{RB}{E} \leftrightarrow \frac{BR}{E}$ transformations.
end while

Since there are $O(N^2)$ cells, and removing each such cell takes $O(N)$ time, this algorithm runs in $O(N^3)$ time.

Algorithm 2

---

repeat
  Greedily remove all the R or B cells that can be removed.
  Remove the closest R or B cell using $\frac{RB}{E} \leftrightarrow \frac{BR}{E}$ transformations.
until $\nexists$ any R or B cell in the matrix

The cells that are removed greedily utilize minimum number of transitions by definition. Hence this algorithm too runs in $O(N^3)$ time.

Thus we can see that there are many algorithms possible that would run in $O(N^3)$ time. However, we want to find an algorithm that achieves the task by using minimum possible number of transitions. We define such an algorithm to be an optimal algorithm. We will present such an algorithm in the following sections.

1.5 Non-Optimal Transitions

Definition 1 *We define a non-optimal transition to mean moving an* R *cell such that its distance from the* R *receptor increases, or a* B *cell such that its distance from the* B *receptor increases. Any transition that is not a non-optimal transition is defined as an optimal or greedy transition.*

If no non-optimal transitions occur in an algorithm that solves the problem, then by definition, such an algorithm would be optimal. However, we now prove that in some problem configurations, non-optimal transitions are unavoidable.

Result 1 $\frac{RB}{E} \leftrightarrow \frac{BR}{E}$ *transformations contain non-optimal transitions.*

$\frac{RB}{E} \leftrightarrow \frac{BR}{E}$ transformations are defined as the following series of transitions:

$$\begin{pmatrix} R & B \\ x & E \end{pmatrix} \longrightarrow \begin{pmatrix} R & E \\ x & B \end{pmatrix} \longrightarrow \begin{pmatrix} E & R \\ x & B \end{pmatrix} \longrightarrow \begin{pmatrix} B & R \\ x & E \end{pmatrix}$$

Here, we can see that B temporarily moves from its current row to the next lower one. Hence, its distance from the B receptor increases. Thus, even though this B might have moved closer to the B receptor by the end of this series of transitions, $\exists$ one transition in this series which causes it to move further from its receptor, and is hence by definition, non-optimal.

Lemma 2 *Some matrices cannot be emptied without* $\frac{RB}{E} \leftrightarrow \frac{BR}{E}$ *transformations.*

Proof: Consider the 2-row matrix shown below:

$$R \sqsupset \begin{pmatrix} B & B & B & R & R & R \\ E & B & B & R & R & R \end{pmatrix} \sqsubset B$$

In this case, the greedy strategy has no success in emptying the matrix. Thus in order to start emptying the cells, we need to make some $\frac{RB}{E} \leftrightarrow \frac{BR}{E}$ transformations. $\square$

Thus, by Result 1 and Lemma 1, we can see that there are some matrices that require non-optimal transitions to be completely emptied.

In general, non-optimal transitions will be unavoidable whenever we have collections of cells of the form:

$$R \sqsupset \begin{pmatrix} \ldots & B & X_1 & X_2 & \ldots & X_{l-1} & X_l & R & \ldots \end{pmatrix} \sqsubset B$$

, where any cell of the type X_i can be either R or B.

We shall refer to such collections of cells bounded by a B cell at the left and an R cell at the right simply as a BR **duo**, since to empty these cells we need to make at least a few $\frac{RB}{E} \leftrightarrow \frac{BR}{E}$ non-optimal transformations.

Thus, even the best possible algorithm for solving this problem, i.e. the optimal algorithm would still have to make a few non-optimal transitions for some matrices. To prove the optimality of such an algorithm, we must prove that the number of non-optimal transitions is minimized during its execution.

We now consider the question of whether some matrices are easier to empty than others. Intuitively, matrices in which R cells are closer to the B receptor and B cells to the R receptor require non-optimal transitions to be emptied, and are hence harder to empty i.e. require more transitions, than matrices in which the cells of each colour are closer to their own receptor. Also, we note the following result:

Result 2 *Once the* 1st *row has been emptied, the rest of the rows can be emptied optimally.*

We provide a simple intuitive argument for this result. First, any cell in the 2nd row can be brought to the 1st row by directly swapping it with the E cell above it in the 1st row, and can then be brought to its receptor by successively swapping it with E cells. Once the 2nd rows is emptied in this manner, any cells in the 3rd row can be brought up to the 1st row by swapping it with the 2 E cells above it, and then emptied as described above. In this manner, all the cells in all the rows can be emptied. Since all the transitions here are greedy transitions, and move any coloured cell strictly closer to its receptor, we can see that all the rows from the 2nd row onwards can be emptied optimally. □

Accordingly, we restrict ourselves to considering only the 1st row when considering the difficulty of emptying any given matrix. Here, we must note that once the cells of any one particular colour have been removed, the 1st row only consists of E cells and cells of the other colour, and hence the 1st row can now be entirely emptied using greedy transitions. With these facts in mind, we define the difficulty levels of emptying various matrices as:

- *Easy*:
  The matrices in which the entire 1st row can be emptied greedily, without needing any non-optimal transitions are known as matrices of an Easy level.
- *Medium*:
  The matrices in which a mixture of greedy and non-optimal transitions are required

to empty either all the R or all the B cells in the 1st row are known as matrices of a Medium level.

- *Hard*:
  The matrices in which either the R or B cells in the 1st row can be emptied only using non-optimal transitions are known as matrices of a Hard level.

This classification of matrices does not have an impact on the algorithm that we propose in the next section, nor on the proof of its optimality, however, they will provide a useful framewrok while analyzing the results produced by our implementation of the following algorithm.

1.6 Optimal Algorithm

We present an algorithm, which we claim to be optimal, for emptying all the R and B cells from a matrix. The proof of its optimality follows in the next section.

We use the following data structures in our algorithm:

- *Matrix*:
  We are given an $N \times N$ matrix. Numbering starts from 1 for both rows and columns, i.e. (i, j) refers to the cell in the ith row and j^{th} column.

- *ER* , *EB*:
  These are two variables that indicate how many consecutive cells from the R or B receptors in the 1st row are E cells. (ER is the variable corresponding to the R receptor, and EB is the variable corresponding to the B receptor). For example, ER $= 4$ implies that cells $(1, 1), (1, 2), (1, 3), (1, 4)$ are R. Initially, both of these pointers are set to NULL.

- *Cost*:
  Cost is an integer variable that indicates the number of transitions observed till the present time, i.e. the cost incurred during the execution of the algorithm so far.

We now define some sub-routines that are used in the control loop of our algorithm:

Algorithm 1 CheckBoundary

```
if (1,1) == R then
    (1,1) ⟵ E
    ER ⟵ 1
end if
if (1,N) == B then
    (1,N) ⟵ E
    EB ⟵ N
end if{Any transition from R or B to E is spontaneous, so the value of Cost remains unchanged.}
```

The colour to be used by the RemoveNonOptimally function explained below is decided by this SelectForNonOptimality function.

Algorithm 2 ExpandGreedily

```
repeat
  if ER +1 == R then
    ER +1 ⟵ E
    Cost ⟵ Cost + ER)
    ER ⟵ ER +1
  end if
  if EB −1 == B then
    EB −1 ⟵ E
    Cost ⟵ Cost +N− EB)
    EB ⟵ EB −1
  end if
  if ER +1 == E then
    ER ⟵ ER +1
  end if
  if EB −1 == E then
    EB ⟵ EB −1
  end if
  if ER +1 == EB then
    ER ⟵ N
    EB ⟵ 1 {The 1st row has been emptied, so set both pointers to include the
    whole 1st row}
  end if
until ∄ R or E in contact with ER and ∄ B or E in contact with EB
for i = 1 to N - 1 do
  if (1, i) == B AND (1, i + 1) == E then
    Swap (1, i) and (1, i + 1)
    Cost ⟵ Cost +1
  end if
end for
for i = N - 1 to 1 do
  if (1, i) == E AND (1, i + 1) == R then
    Swap (1, i) and (1, i + 1)
    Cost ⟵ Cost +1
  end if
end for
```

Algorithm 3 SelectForNonOptimality

$SR \longleftarrow$ Sum of distances of R cells in 1st row from R receptor
$SB \longleftarrow$ Sum of distances of B cells in 1st row from B receptor
if $SR \leq SB$ **then**
  return R
else
  return B
end if{Whichever colour has the lesser sum should be eliminated using non optimal transitions so that such transitions are minimized}

Algorithm 4 RemoveNonOptimally

Require: There is at least 1 E in the 2nd row.
C $\longleftarrow$ Colour returned by SelectForNonOptimality
C' $\longleftarrow$ Opposite Colour to C
X $\longleftarrow$ Closest C to EC
Y $\longleftarrow$ Closest E in 2nd row to X
Bring Y to the cell directly below X. Add the cost of this movement to *Cost* {Y is now directly below a BR duo.}
repeat
  Convert BR to RB, using $\frac{RB}{E} \leftrightarrow \frac{BR}{E}$ transformation
  $Cost \longleftarrow Cost + 3$
  Shift Y by one place in the direction that X shifts to
  $Cost \longleftarrow Cost + 1$
until X has been brought to EC
Replace X with E
$Cost \longleftarrow Cost$ + Length(EC)
Increment EC pointer by 1 in its correct direction

Algorithm 5 ClearNextRows

Require: The 1st row has been totally cleared of coloured cells.
for $i = 2$ to N **do**
  for $j = 2$ to N **do**
    Set (i, j) to E
    Add $i+j$ (or $\lceil \sqrt{i^2 + j^2} \rceil$, depending on whether we use L1 or L2 distance) to the value of *Cost*
  end for
end for

Algorithm 6 IsEmpty(row_{max}, col_{max})

```
for i = 1 to row_max do
  for j = 1 to col_max do
    if (i, j) ≠ E then
      return false
    end if
    return true
  end for
end for
```

Having estabilished these sub-routines, we now present our main algorithm:

Algorithm 7 EmptyOptimally

```
CheckBoundary
ExpandGreedily
if IsEmpty(N, N) then
  return Cost
end if{If the whole matrix has been emptied, return the cost}
SelectForNonOptimality
repeat
  RemoveNonOptimally
  ExpandGreedily
until IsEmpty(1,N) {Check if the 1^st row has been emptied}
ClearNextRows
return Cost
```

We first try to empty as many cells as possible in a greedy manner. If all the cells are successfully emptied this way, the algorithm terminates and returns *Cost*. If there are cells left that could not be emptied greedily, we calculate which colour would cause a lower addition to *Cost* while being removed non-optimally from the 1st row, and alternately call the functions RemoveNonOptimally (which removes one cell from the 1st row non-optimally) and ExpandGreedily (which removes as many cells as possible greedily), until the 1st row has been emptied. We then call the function ClearNextRows and empty the remaining cells in the other rows optimally. The algorithm then terminates and returns *Cost*. We shall prove in the following section that this algorithm is optimal.

1.7 Proof of Optimality

In the worst case, the 1st row will be of the form:

$$R \sqsupset \left(B \;\; B \;\; \ldots \;\; B \;\; R \;\; R \;\; \ldots \;\; R \right) \sqsubset B$$

In this case, for removing each R or B in the 1_{st} row:
First an E has to reach the target cell. This takes $O(N)$ time.
Then, this cell is brought to the ER or EB boundary (depending on its colour). This also takes $(1 + 3) * O(N) = O(N)$ time.
Thus each R or B is removed in $O(N)$ time, and there are $O(N)$ such cells. Hence the 1st row can be emptied in $O(N^2)$ time.
Once this is done, the rest of the rows are emptied optimally. Hence, the entire algorithm operates in $O(N^3)$ time. Thus the algorithm is asymptotically optimal.

However, in order to prove the optimality of this algorithm, asymptotic optimality is not enough; we must also prove that it makes the minimum possible number of non-optimal transitions.

We evaluate each of the functions used in the algorithm and show that in each function, either only optimal transitions take place, or if non-optimal transitions occur, their number is minimized.

CheckBoundary :

This function is instantaneous, and there are no transitions at all.

ExpandGreedily :

This function empties coloured cells in a greedy manner. Any R or B cell that is removed is done so using the shortest path to its receptor. Also, whenever a BE or ER duo is reached, the coloured cell is moved strictly closer to its receptor. Hence, there are no non-optimal transitions.

SelectForNonOptimality :

This function simply evaluates two mathematical sums and compares them; it does not execute any transitions. Hence, we can easily note that there are no non-optimal transitions here.

RemoveNonOptimally :

As the name suggests, some non-optimal transitions are performed in this function. We will prove that the number of such transitions is kept to a minimum.

In this function, we perform two steps which are sources of non-optimality:

1. Moving E to a BR duo. We shall denote the colour selected by SelectForNonOptimality as C.
2. Repeatedly swapping RB to BR by using a $\frac{RB}{E} \leftrightarrow \frac{BR}{E}$ transformation and propagating the E cell.

We will prove that in our function, the non-optimality in both steps is minimized.

1. First we have to decide whether to remove R or B non-optimally. This decision is made using the SelectForNonOptimality function such that the colour with lesser cumulative distance of its cells from their receptor is chosen, so the non-optimality due to this decision is minimized. We must now move an E in the 2nd row to the BR duo. If there are multiple E cells in this row, we must decide which E to move. We pick the closest E to that BR duo, and hence minimize the number of non-optimal transitions.
2. Prior to calling RemoveNonOptimally, we have arranged the 1st row using the function ExpandGreedily such that all E cells are made part of either the ER list or the EB list., i.e. the BR duos (as defined earlier) are bunched together without any E cells in between them. Thus, in this part of the RemoveNonOptimally funtion, the number of $\frac{RB}{E} \leftrightarrow \frac{BR}{E}$ transformation is minimized, i.e. the number of greedy transitions of C through EC is maximized. Once C is brought to EC, the rest of the function proceeds in a greedy manner, without any further non-optimal transitions.

ClearNextRows :

From Result 2, once the 1st row has been cleared entirely, all cells in subsequent rows can be removed by bringing them strictly closer to their receptor at each step, i.e. only by greedy transitions. This is what is done in this function.

Thus all function except RemoveNonOptimally utilize only greedy transitions, and the number of non-optimal transitions used in the function RemoveNonOptimally is minimized. Hence, our algorithm is optimal.

1.8 Implementation and Results

We implemented our algorithm in C++. The user specifies the size of the matrix, as well as the number of R, B and E cells. The matrix can either be generated randomly, or entered by the user.

There is only one initial condition, namely that there exists at least one E in the matrix.

At each pass of the algorithm, the intermediate matrices are computed, including versions after non-optimal removal and after greedy removal. The total cost for removing all the cells is stored in memory.

We first consider the case of 4×4 matrices, as these are small enough to trace their transitions. We ran our program on various 4×4 matrices of each difficulty level as shown below:

- *Easy*:

$$E1 = \begin{pmatrix} R & R & B & B \\ E & R & B & R \\ B & R & R & B \\ B & B & B & B \end{pmatrix}, \quad E2 = \begin{pmatrix} R & R & R & R \\ R & R & E & B \\ B & B & B & B \\ B & R & B & R \end{pmatrix}$$

- *Medium*:

$$M1 = \begin{pmatrix} R & B & R & B \\ E & R & B & R \\ B & R & R & B \\ B & B & B & B \end{pmatrix}, \quad M2 = \begin{pmatrix} R & B & B & R \\ R & R & E & B \\ B & B & R & R \\ B & R & B & R \end{pmatrix}$$

- *Hard*:

$$H1 = \begin{pmatrix} B & B & R & R \\ E & R & B & R \\ B & R & R & B \\ R & B & B & R \end{pmatrix}$$

We show below some of the intermediate states obtained while running our program on the above matrices:

E1:

$$\begin{pmatrix} R & R & B & B \\ E & R & B & R \\ B & R & R & B \\ B & B & B & B \end{pmatrix} \longrightarrow \begin{pmatrix} E & E & E & E \\ E & R & B & R \\ B & R & R & B \\ B & B & B & B \end{pmatrix} \longrightarrow \begin{pmatrix} E & E & E & E \\ E & E & E & E \\ E & E & E & E \\ E & E & E & E \end{pmatrix}$$

E2:

$$\begin{pmatrix} R & R & R & R \\ R & R & E & B \\ B & B & B & B \\ B & R & B & R \end{pmatrix} \longrightarrow \begin{pmatrix} E & E & E & E \\ R & R & E & B \\ B & B & B & B \\ B & R & B & R \end{pmatrix} \longrightarrow \begin{pmatrix} E & E & E & E \\ E & E & E & E \\ E & E & E & E \\ E & E & E & E \end{pmatrix}$$

M1:

$$\begin{pmatrix} R & B & R & B \\ E & R & B & R \\ B & R & R & B \\ B & B & B & B \end{pmatrix} \longrightarrow \begin{pmatrix} E & B & R & E \\ E & R & B & R \\ B & R & R & B \\ B & B & B & B \end{pmatrix} \longrightarrow \begin{pmatrix} E & E & B & E \\ R & B & E & R \\ B & R & R & B \\ B & B & B & B \end{pmatrix}$$

$$\longrightarrow \begin{pmatrix} E & E & E & E \\ R & B & E & R \\ B & R & R & B \\ B & B & B & B \end{pmatrix} \longrightarrow \begin{pmatrix} E & E & E & E \\ E & E & E & E \\ E & E & E & E \\ E & E & E & E \end{pmatrix}$$

M2:

$$\begin{pmatrix} R & B & B & R \\ R & R & E & B \\ B & B & R & R \\ B & R & B & R \end{pmatrix} \longrightarrow \begin{pmatrix} E & B & B & R \\ R & R & E & B \\ B & B & R & R \\ B & R & B & R \end{pmatrix} \longrightarrow \begin{pmatrix} E & E & B & B \\ R & E & R & B \\ B & B & R & R \\ B & R & B & R \end{pmatrix}$$

$$\longrightarrow \begin{pmatrix} E & E & E & E \\ R & E & R & B \\ B & B & R & R \\ B & R & B & R \end{pmatrix} \longrightarrow \begin{pmatrix} E & E & E & E \\ E & E & E & E \\ E & E & E & E \\ E & E & E & E \end{pmatrix}$$

H1:

$$\begin{pmatrix} B & B & R & R \\ E & R & B & R \\ B & R & R & B \\ R & B & B & R \end{pmatrix} \longrightarrow \begin{pmatrix} B & B & R & R \\ E & R & B & R \\ B & R & R & B \\ R & B & B & R \end{pmatrix} \longrightarrow \begin{pmatrix} E & B & B & R \\ R & E & B & R \\ B & R & R & B \\ R & B & B & R \end{pmatrix}$$

$$\longrightarrow \begin{pmatrix} E & B & B & R \\ R & E & B & R \\ B & R & R & B \\ R & B & B & R \end{pmatrix} \longrightarrow \begin{pmatrix} E & E & B & B \\ R & E & B & B \\ B & R & R & B \\ R & B & B & R \end{pmatrix} \longrightarrow \begin{pmatrix} E & E & E & E \\ R & E & B & B \\ B & R & R & B \\ R & B & B & R \end{pmatrix}$$

$$\longrightarrow \begin{pmatrix} E & E & E & E \\ E & E & E & E \\ E & E & E & E \\ E & E & E & E \end{pmatrix}$$

The costs of emptying these 4×4 matrices in each of these cases are listed in Table 1.1

Index	N_r	N_b	*Cost*
E1	6	9	28
E2	8	7	30
M1	6	9	32
M2	8	7	33
H1	8	7	41

Table 1.1: Results for the case $4 \times 4, N_e = 1$

Based on the results in 1.1, we can see that the distribution of R and B cells in the matrix plays a role in determining the ease of emptying it, since the matrices that are defined as *Easy* accoring to our framework show a lower value of *Cost* than those defined as *Medium*, which in turn show a lower *Cost* than the *Hard* matrix.

So far, we have only considered matrices of the form 4×4. We now give some examples of Easy, Medium and Hard matrices (with $N_e = 1$) of the form 7×7:

$$E_7 = \begin{pmatrix} R & R & R & R & B & B & B \\ R & R & B & E & B & B & R \\ R & R & B & B & B & R & R \\ B & B & R & R & B & R & R \\ B & R & R & B & R & B & R \\ R & R & B & B & R & R & B \\ B & R & R & R & B & R & B \end{pmatrix} \quad M_7 = \begin{pmatrix} B & B & R & R & B & B & R \\ R & R & B & E & B & B & R \\ R & R & B & R & B & R & R \\ B & B & R & R & R & B & B \\ B & R & R & R & R & B & R \\ B & R & B & R & B & R & R \\ B & R & R & R & B & B & B \end{pmatrix}$$

$$H_7 = \begin{pmatrix} B & B & B & B & R & R & R \\ B & R & B & E & B & R & R \\ R & R & B & B & B & R & B \\ B & B & R & R & B & B & R \\ B & R & R & B & B & B & R \\ B & R & B & B & R & R & B \\ B & R & R & B & B & R & B \end{pmatrix}$$

For these matrices, we list the costs of emptying them in Table 1.2

Index	N_r	N_b	*Cost*
E_7	27	21	188
M_7	26	22	208
H_7	21	27	230

Table 1.2: Results for the case $7 \times 7, N_e = 1$

From the results of 1.2 too, our intuition that matrices in which R cells are closer to the R receptor and B cells are closer to the B receptor are easier to empty, is confirmed. The *Easy* matrix shows significantly lower time to be emptied than the *Medium* and *Hard* ones, even though all other factors such as N_e, N_r, N_b are equal, or approximately so.

All the matrices that we have considered so far have shared one property: $N_e = 1$. We also ran our code on 7×7 matrices with different values of N_e. The empty cells are interspersed throughout the matrix in question. Consider the following randomly generated examples for $N_e = 5$ and $N_e = 10$:

$$N_{e5} = \begin{pmatrix} B & B & B & B & B & B & R \\ R & R & B & E & B & B & B \\ R & R & E & B & B & R & R \\ B & E & R & R & B & R & R \\ E & R & R & B & R & B & R \\ B & R & B & B & R & R & B \\ E & R & R & R & B & R & B \end{pmatrix} \quad N_{e10} = \begin{pmatrix} B & E & B & B & B & B & B \\ R & R & B & E & B & E & B \\ R & R & E & B & B & R & R \\ E & E & R & R & B & R & R \\ E & R & R & B & R & E & R \\ E & R & B & B & R & R & B \\ E & R & R & B & B & R & B \end{pmatrix}$$

The costs for emptying these matrices are given in Table 1.3

Similarly, we can obtain matrices for higher values of N_e. We now consider matrices of dimension varying from 10 to 100, at intervals of 10. For N_e, we have included as cases the base value $N_e = 1$ as well as the values $N_e = 0.05, 0.10, 0.15, 0.20, 0.25 \times N^2$. The matrices are randomly generated, and we use roughly equal values for N_r and N_b. We present our results for *Cost* in emptying the matrices in all of these cases in Tables

N_e	N_r	N_b	$Cost$
5	22	22	191
10	20	19	153

Table 1.3: Results for the case $7 \times 7, N_e = 5, 10$

1.4 and 1.5.

N_e	$N = 10$	$N = 20$	$N = 30$	$N = 40$	$N = 50$
1	634	5237	17826	42509	82422
$0.05 \times N^2$	609	5050	16719	40119	78195
$0.10 \times N^2$	542	4662	15841	37956	74607
$0.15 \times N^2$	545	4544	15111	36501	69706
$0.20 \times N^2$	507	4095	13999	34140	65391
$0.25 \times N^2$	443	3845	13474	31624	61897

Table 1.4: Results for $N = 10, 20 \ldots 50, N_e z = 1, 0.05 \times N^2 \ldots 0.25 \times N^2$

We show plots of the data in these tables for each value of N in Figures 1.1 - 1.10. We also show a plot of $Cost$ versus N, for the case $N_e = 0.10N^2$, and compare this plot to the curve $y = x^3$ in Figure 1.11.

Our results indicate that the value of N_e, i.e. the number of empty cells in the matrix has a significant impact on the number of transitions needed to remove all the cells. This impact is pronounced if multiple E cells happen to be in the 1st row. Moreover, we can also see, based on comparing the costs of emptying matrices of various sizes N in Figure 1.11, that the number of transitions grows as $O(N^3)$, thus offering experimental validation of the algorithm's complexity.

N_e	$N = 60$	$N = 70$	$N = 80$	$N = 90$	$N = 100$
1	144329	226389	339480	484963	666207
$0.05 \times N^2$	134521	215826	323374	460616	629378
$0.10 \times N^2$	129871	186218	307367	437359	597430
$0.15 \times N^2$	120142	172419	287625	412956	563117
$0.20 \times N^2$	114596	181917	270936	373141	531186
$0.25 \times N^2$	107100	170920	254192	349165	494344

Table 1.5: Results for $N = 60, 70 \ldots 100, N_e z = 1, 0.05 \times N^2 \ldots 0.25 \times N^2$

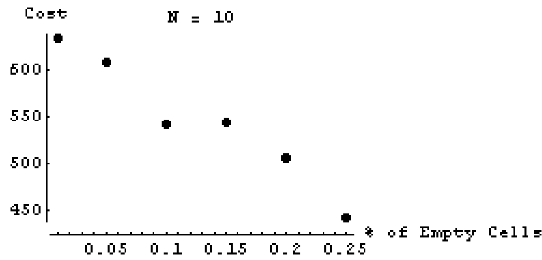

Figure 1.1: $N = 10$

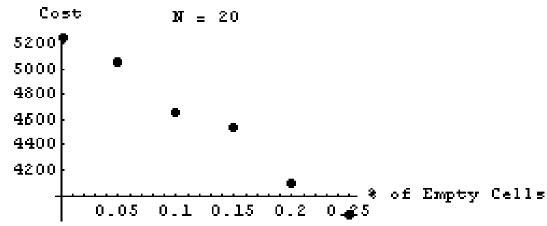

Figure 1.2: $N = 20$

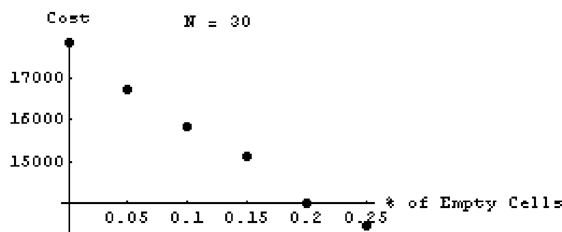

Figure 1.3: $N = 30$

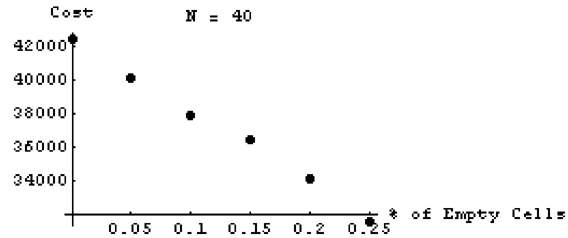

Figure 1.4: $N = 40$

1.9 Conclusion

In this chapter, we introduced the problem of moving red and blue cells in a matrix with empty spaces. We proved that it was possible to empty the entire matrix, no matter what its initial configuration, provided there was at least one empty cell in it initially. We also proved that an optimal algorithm for this task would run in $O(n^3)$ time, where n is the order of the number of rows/columns of the matrix. We then presented various optimal complexity algorithms, and proved that for certain matrix configurations, some non-optimal transitions were necessary. We then presented an algorithm that minimizes these non-optimal transitions, i.e. an optimal algorithm to solve our problem. We then discussed our implementation of this algorithm, and the results obtained by it for matrices of various sizes and with different values of N_e. We considered the impact of the distribution of R and B cells as well as the number of E cells in the matrix, on the total cost of emptying it. The matrices in which R cells are closer to their receptor and similarly so for the B cells have a lower cost of being emptied than those in which R cells are closer to the B receptor and vice versa. In addition, the cost of emptying matrices decreases as the value of N_e increases. This algorithm can be applied in many practical

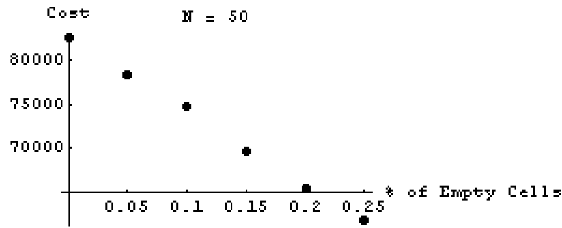

Figure 1.5: $N = 50$

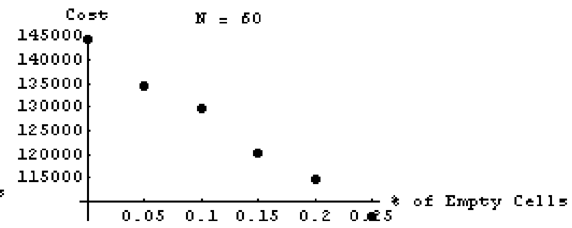

Figure 1.6: $N = 60$

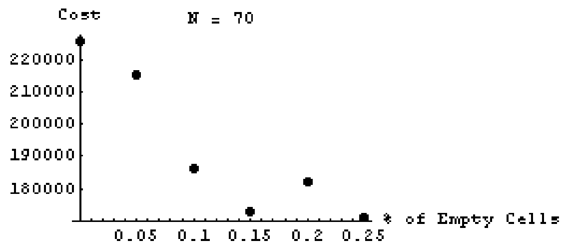

Figure 1.7: $N = 70$

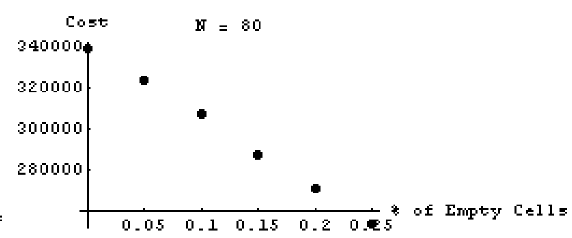

Figure 1.8: $N = 80$

domains. The most important of these is biological microarrays, but they also include other diverse areas such as memory storage devices and image processing.

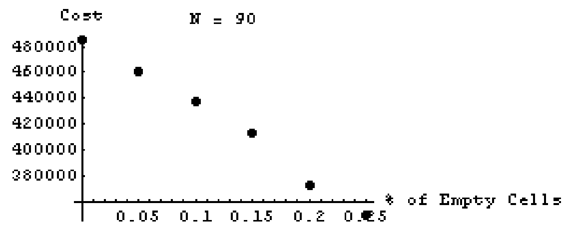

Figure 1.9: $N = 90$

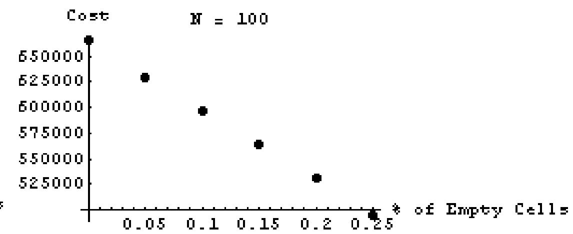

Figure 1.10: $N = 100$

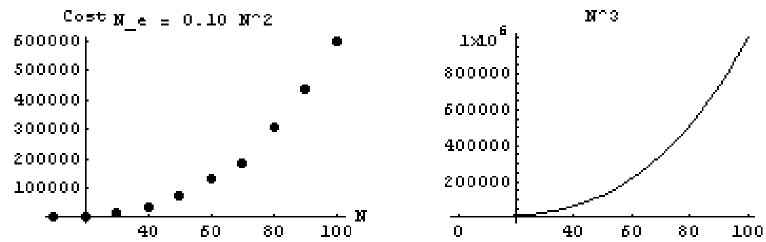

Figure 1.11: $N_e = 0.10 \times N^2$

Chapter 2

Digital Curve on Grid: Curve Fitting using Linear Programming

2.1 Introduction

Curve fitting is a standard problem in Mathematics and Computer Science, and much work has been done in this area. Traditionally, techniques such as interpolation and regression analysis are used, and structures used for approximation are usually 1st or higher order algebraic polynomials. There are many variations to this approach, including splines, trigonometric polynomials, wavelets, etc. Here, we examine a linear programming approach to this problem. We model the problem as finding a target sequence of points in a lattice that has maximal similarity to an initially given set of points in this lattice, subject to various constraints.

The application of linear programming to curve fitting is not a new idea; indeed, it has been investigated earlier[10], [11]. However, we consider a variety of constraints on the fitted curve, such as irreducibility, monotonicity, and an upper bound on the change in slope at any step in the sequence. We first formulate these concepts as linear constraints, and provide a formal procedure to convert an instance of discrete curve fitting into an equivalent integer linear programming problem (The problem of integer linear programming is NP-Hard [7], so we must initially relax some criteria, to obtain a relaxed problem which is an instance of linear programming, which is polynomially solvable). Once we have obtained such a procedure, we obtain solutions for given point sequences of various sizes, using standard linear programming algorithms such as the interior point method. We then use these solutions for the relaxed version of the problem to obtain approximate solutions for instances of our original problem. As we shall see, this approach still manages to produce satisfactory solutions in most cases.

2.2 Problem Definition

We are given a $N \times N$ lattice. Each point (i, j) of the lattice may be either ON or OFF. We associate a value with each such point (i, j). If the point is OFF, its associated value is set to 0, and if the point is ON, its associated value is set to 1.

Initially, a set of points from this lattice is set to be ON. We now wish to compute a target sequence of ON points, which approximates the original set as far as possible, but is also subject to the following constraints:

- The boundary points of the target sequence are the same as those of the original sequence.
- The target sequence is irreducible.
- The change in slope between successive points of the target sequence is bounded.

We wish to formulate this problem using linear programming.

Now, since the lattice points can only take the values 0 or 1, this problem, once formulated as an instance of linear programming, will actually be an instance of integer linear programming, which is known to be NP-hard. Hence, we must initially ignore this constraint, and allow the point to take any real value between 0 and 1.

Once the problem has been formulated as an instance of linear programming, we take various point sets, convert them into equivalent linear programming problems, and solve these problems using a standard mathematical software package [9]. We then round off the values of the points of this solution to either 0 or 1, to obtain a solution to our original problem. There will, of course, be an error in the solution thus obtained.

2.2.1 Linear Programming

Linear Programming problems are optimization problems where the objective function and the constraints are all linear.

Given a set of real numbers $a_1, a_2 \ldots a_n$ and a set of variables $x_1, x_2, \ldots x_n$, a linear function on these variables is defined as:

$$f(x_1, x_2, \ldots x_n) = a_1x_1 + a_2x_2 + \ldots a_nx_n = \sum_{j=1}^{n} a_jx_j$$

In linear programming, we try to maximize or minimize such an objective function $\sum_{j=1}^{n} a_j x_j$, subject to a set of linear constraints, each of which is either of the form $f(x_1, x_2, \ldots x_n) \leq b$ or $f(x_1, x_2, \ldots x_n) = b$, where b is a real number and f is a linear function.

We can restate the linear programming problem in matrix form as:

Given a $n-$dimensional vector C, an $m \times n$ matrix M, and sets of real numbers $b_1, b_2 \ldots b_m$, $l_1, l_2 \ldots l_n$ and $u_1, u_2 \ldots u_n$, find an $n-$dimensional vector x which minimizes $C \cdot x$ subject to the following constraints:

For each row M_i of M,

$$M_i \cdot x \begin{cases} = b_i & \text{if } s_i = 0 \\ > b_i & \text{if } s_i = 1 \\ < b_i & \text{if } s_i = -1 \end{cases}$$

and $\forall i = 1, 2, \ldots n, \qquad l_i \leq x_i \leq u_i$

Hence a linear programming problem can be fully expressed as the following tuple:

$$[C, M, \{(b_1, s_1), (b_2, s_2), \ldots (b_m, s_m)\}, \{(l_1, u_1), (l_2, u_2), \ldots (l_n, u_n)\}]$$

2.3 Formulation of Curve Fitting as Linear Programming

We are given a lattice $L = N \times N$. The point in the ith row and j^{th} column is referred to as (i, j).

Let B_{ij} denote the original status of (i, j). If (i, j) is initally set to ON, $B_{ij} = 1$, else $B_{ij} = 0$. It is clear that all such values B_{ij} are constants.

Let the two boundary points of the initial sequence be denoted by (i_l, j_l) and (i_h, j_h). Hence $B_{i_l j_l} = 1$ and $B_{i_h j_h} = 1$

Let X_{ij} denote the status of (i, j) in the target sequence. If (i, j) is ON in the target sequence, $X_{ij} = 1$, else $X_{ij} = 0$.

Thus, the variables in our linear programming problem are X_{ij}, where

$$i \in \{1, 2, \ldots N\}, j \in \{1, 2, \ldots N\}$$

We want the sequence X_{ij} to approximate B_{ij} as closely as possible. In order to state this formally, we need a definition of the error E between the target sequence and the original point set.

We define this error E as $E = \sum_{i,j}(1 - B_{ij}X_{ij})$. (Here, B_{ij} refers to constants, not variables)

We can see that E as defined above measures the distance between the original and computed point sequences. For a point (i, j), $1 - B_{ij}X_{ij} = 0$ only if the point (i, j) is ON in both sequences, $1 - B_{ij}X_{ij} = 1$ otherwise. Each such point that has differing states in the sequences contributes to the error E by an increment of 1.

Hence, for the target sequence to be as close to the original sequence as possible, this error E must be minimized, i.e. E or some variation of it is the objective function that we try to minimize using linear programming.

Now, we elaborate on the constraints mentioned earlier on the target point sequence:

Boundary:

The target sequence should have the same boundary points as the original sequence. Therefore, $X_{i_l j_l} = 1$ and $X_{i_h j_h} = 1$.

Irreducibility:

Definition 2 *We define every point of a sequence that is not a boundary point to be an interior point.*

Definition 3 *We define the neighbourhood of a point* (i, j) *to be the set of points* $\{(i-1, j-1), (i-1, j), (i-1, j+1), (i, j-1), (i, j+1), (i+1, j-1), (i+1, j), (i+1, j+1)\}$.

For a point sequence to be irreducible, each interior point of this sequence must have no more than 2 ON points in its neighbourhood, and each boundary point of the sequence must have no more than 1 ON point in its neighbourhood.

Intuitively, if a point sequence is not irreducible, there exists a smaller point sequence that can convey the same amount of information.

Bounded change in slope:

The slope of 2 points $p = (i_1, j_1)$ and $q = (i_2, j_2)$ to be $S_{2,1} = (j_2 - j_1)/(i_2 - i_1)$. When we say that the change in slope at any point $p = (i, j)$ in the sequence is bounded, we mean:

$\mid (S_{p+1,p} - S_{p,p-1}) \mid \leq \beta$, where $p+1$ is the next point and $p-1$ is the previous point in the sequence, and β is some predefined bound.

However, this is not a linear equation. Moreover, it is not an equation in the variables we have designated, X_{ij}.

We need to find constraints in terms of the variables X_{ij}, which are linear, and have an equivalent effect on the target point sequence. It is not immediately obvious how to approach this problem.

Let us define the unacceptable slope transitions as those transitions that involve a change of slope $\beta > 2$. For example, $(0, 0)- > (1, 1)- > (0, 2)$ is an acceptable transition, while $(0, 0)- > (2, 1)- > (0, 2)$ is not. Hence, only 2 out of these 3 points may be ON, and the corresponding slope constraint for this unacceptable transition is:

$X_{00} + X_{21} + X_{02} \leq 2$

Therefore, we must enumerate such constraints for every combination of 3 points from the n^2 points of the given lattice that produces a slope change $\beta > 2$:

$X_{00} + X_{21} + X_{02} \leq 2$

$X_{00} + X_{31} + X_{12} \leq 2$

$X_{11} + X_{31} + X_{12} \leq 2$

$X_{00} + X_{33} + X_{03} \leq 2$

$\vdots$

$\forall (i_1, j_1), (i_2, j_2), (i_3, j_3)$ such that $\mid ((j_2 - j_1)/(i_2 - i_1)) - ((j_3 - j_2)/(i_3 - i_2)) \mid > 2$, $X_{i_1 j_1} + X_{i_2 j_2} + X_{i_3 j_3} \leq 2$

However, each such slope constraint involves a combination of 3 points, and there are N^2 such points, and 3 out of these N^2 points can be chosen in $N^2C_3 = O(N^6)$ ways. We must choose all such combinations that result in a slope transition > 2, and it is clear that there are at least as many unacceptable transitions as acceptable ones, hence the number of unacceptable transitions, i.e. the number of slope constraints according to this formulation would be $O(N^6)$.

This is a prohibitively high asymptotic complexity, and it would make the problem intractable for all but the smallest of lattices. Hence, we need a more efficient way to express the constraint that the change in slope at any step in the target sequence should be bounded.

We make the simplifying assumption that the target sequence is not sparse, i.e. in every column j of the lattice, $\exists$ a row i such that $X_{ij} = 1$. This assumption is validated in practice; target sequences that are usually generated to fit curves, by various techniques, (including our own, as we shall see later) are not sparse.

Now, if a point (i, j) is ON in the target sequence, suppose that the point that is ON in column $j + 1$ is at row i', where:

$i' - i \geq 4$

However, then, for the change in slope at column j to not exceed β, the point that is ON in column $j - 1$ would have to be at a row i'', where $i - i'' \geq 2$. We make the

additional assumption here, that a sharp slope at any point makes it likelier that there will also be a sharp change in slope at that point, and disallowing sharp slopes is equivalent to disallowing sharp changes in slope. Hence, we can now state the slope-related constraints as:

$\forall i, j, k \in 1, 2, \ldots N, X_{ij} + X_{kj+1} \leq 1$, where $\mid k - i \mid \geq 3$

This assumption has the benefit of drastically reducing the number of slope-related constraints required. There are N columns, and in each column, we have N points, each of which have slopes in relation to N other points in the subsequent column, so the total number of such slopes is $O(N^3)$, i.e. the number of slope related constraints, which is some fraction of the total number of slopes, is also $O(N^3)$.

We are now ready to formally state the objective function and constraints in order to state this problem as an instance of linear programming:

Let the objective function be the minimization of $\sum_{i,j}(1 - B_{ij}X_{ij})$.

The constraints are:

$X_{i_l j_l} = 1$ and $X_{i_h j_h} = 1$

$\sum_{k=-1}^{1} X_{i_l - k j_l - k} \leq 2$

$\sum_{k=-1}^{1} X_{i_h - k j_h - k} \leq 2$

If (i, j) is not a boundary point, $\forall (i, j), \sum_{k=-1}^{1} X_{i-k j-k} \leq 3$

$\forall i, j, k \in 1, 2, \ldots N, X_{ij} + X_{kj+1} \leq 1$, where $\mid k - i \mid \geq 3$

Analysis of constraints

Thus the program has been formulated as an instance of linear programming. There are:

- 2 boundary constraints.
- $O(N^2)$ reducibility constraints. There is one such constraint for each point, and there are n^2 number of points.
- $O(N^3)$ slope constraints, as has been mentioned earlier.

Thus, the total number of constraints is $O(N^3)$.

Hence, for any given lattice and initial sequence of points, we can enumerate the various reducibility constraints in $O(N^2)$ time, and the various slope constraints in $O(N^3)$ time. Thus we have a procedure to obtain a formal representation of the curve fitting problem as an instance of linear programming.

2.4 Implementation and Solutions

We used the procedure outlined in the previous section to convert lattices and initial point sequences of various sizes into equivalent linear programming problems. We then solved these problems using the package Mathematica [9] and running the interior point algorithm. We ran our code on randomly generated initial sequences.

We show our results for point sequences of sizes $N = 5, 7, 10, 12, 15, 20$ in Fig. 2.1 - 2.12.

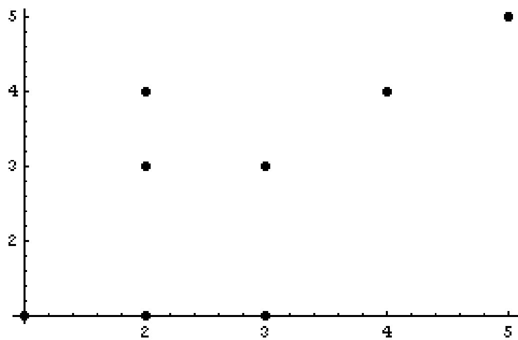

Figure 2.1: $N = 5$, Initial Set

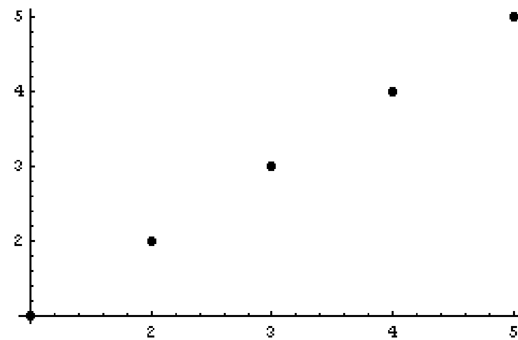

Figure 2.2: $N = 5$, Target Sequence

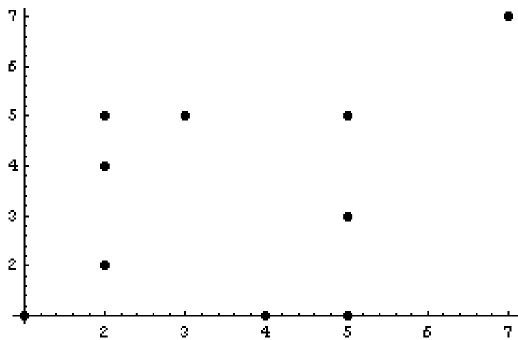

Figure 2.3: $N = 7$, Initial Set

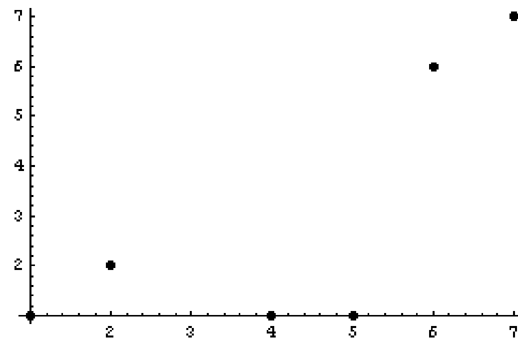

Figure 2.4: $N = 7$, Target Sequence

For each initial sequence, we obtained a set of solutions X_{ij}. However, some of these elements X_{ij} were real numbers in the range $[0, 1]$. In these cases, we rounded off each such X_{ij} to its nearest integer. We thus obtained the target point sequences.

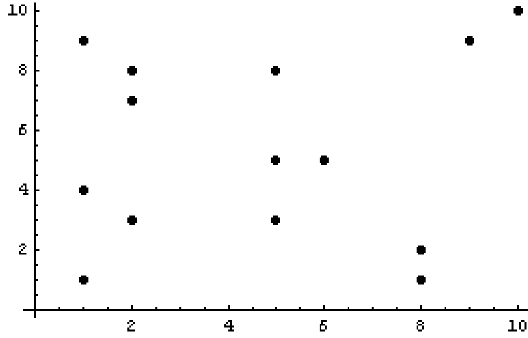

Figure 2.5: $N = 10$, Initial Set

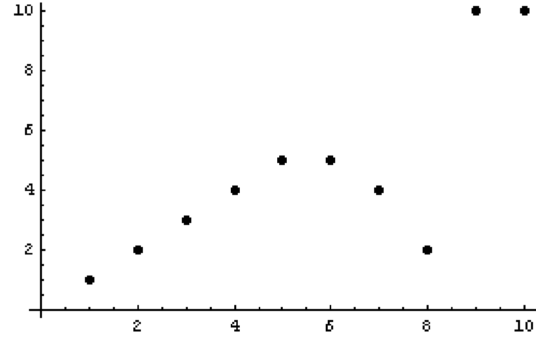

Figure 2.6: $N = 10$, Target Sequence

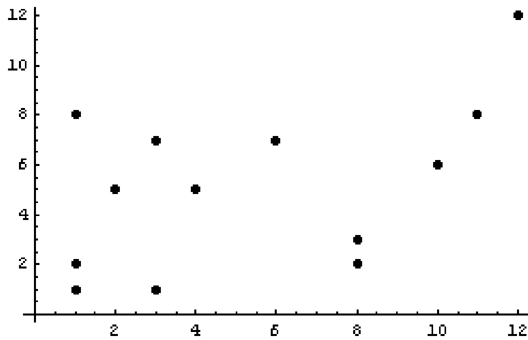

Figure 2.7: $N = 12$, Initial Set

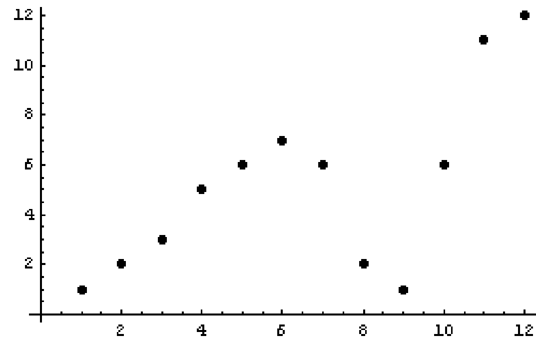

Figure 2.8: $N = 12$, Target Sequence

2.5 Alternative Formulation

We have formulated the problem using variables of the form X_{ij} for every point (i, j) on the lattice. We can instead take advantage of our earlier assumption that in every column j of the lattice, $\exists$ a row i such that $X_{ij} = 1$. From this assumption and the fact that the set of points in question is a sequence, it follow that in each column j, $\exists$ exactly 1 row such that $X_{ij} = 1$.
Hence, the target sequence will have a fixed number of points, i.e. n. We can therefore take as our variables the co-ordinates of these n target points, i.e. $i_1, i_2 \ldots i_n$, and $j_1, j_2 \ldots j_n$. Slope constraints can be expressed more easily than in the earlier formulation; each such constraint is of the form:

$$j_l - j_k \leq \beta \times (i_l - i_k)$$

All such slope constraints can be expressed in $O(n^2)$ time, and without any need for making the assumption that a sharp slope also implies a sharp change in slope.
However, it is not immediately obvious how to express boundary or irreducibility con-

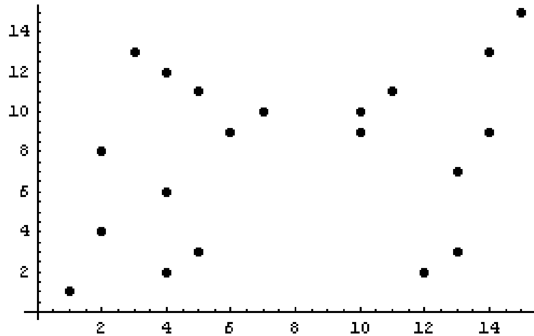

Figure 2.9: $N = 15$, Initial Set

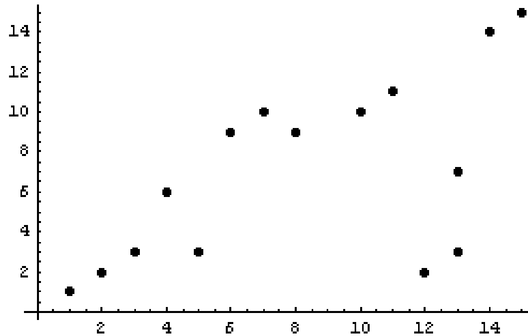

Figure 2.10: $N = 15$, Target Sequence

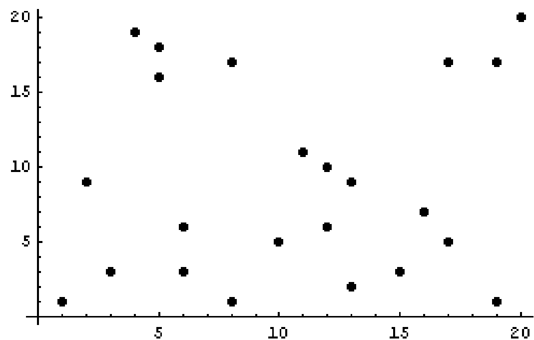

Figure 2.11: $N = 20$, Initial Set

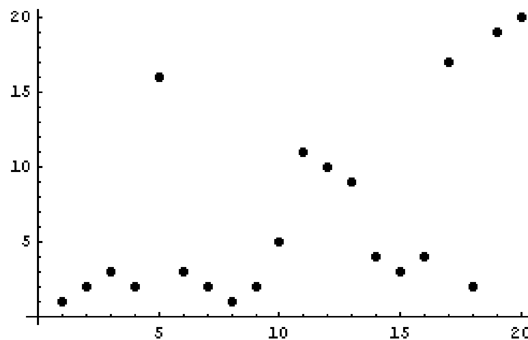

Figure 2.12: $N = 20$, Target Sequence

straints in this formulation.

2.6 Conclusion

We presented the problem of polynomial curve fitting, and the special case of fitting curves in lattices. We examined the use of linear programming as a possible technique to solve this problem, and gave a formal procedure to convert an instance of curve fitting in lattices into an instance of integer linear programming. Since this is NP-Hard, we outlined a procedure to convert such an instance into one of linear programming, by relaxing the integer constraint. We then solved various instances of this type using Mathematica [9], and displayed our results for lattices of various sizes ranging from $N = 5$ to $N = 20$. We also discussed a possible alternative formulation of the problem.

Bibliography

[1] Adrian Dumitrescu, János Pach, “Pushing Squares Around”, Proceedings of the Twentieth Annual Symposium on Computational Geometry, p.116 - 123, 2004,

[2] “Microarrays: Standards and Practices”, **Nature** vol. 442, 2006

[3] Schena M, Shalon D, Davis RW, Brown PO, ”Quantitative monitoring of gene expression patterns with a complementary DNA microarray”, **Science** vol. 270, p.467 - 470, 1995,

[4] Shalon D, Smith SJ, Brown PO, ”A DNA microarray system for analyzing complex DNA samples using two-color fluorescent probe hybridization”, **Genome** Res 6, p.639-645, 1996,

[5] Junaid Ziauddin, David M. Sabatini, “Microarrays of cells expressing defined cD-NAs”, **Nature** vol. 411, p.107 - 110, 2001,

[6] T. Cormen, C. Leiserson, R. Rivest and C. Stein **Introduction to Algorithms**, Prentice-Hall India, 2nd edition, 2004

[7] M. R. Garey, D. S. Johnson , “Computers and Intractability: A Guide to the Theory of NP-Completeness”, W. H. Freeman Publishers, 1979,

[8] M. de Berg, M. van Kreveld, M. Overmars and O. Scwarzkopf, **Computational Geometry, Algorithms & Applications**, Springer-Verlag, 2nd edition, 2000,

[9] Mathematica, v6.0, Wolfram Research,

[10] James E. Kelley Jr., “An Application of Linear Programming to Curve Fitting”, Journal of the Society for Industrial and Applied Mathematics, Vol. 6, No. 1 (Mar., 1958), p. 15 - 22,

[11] Zhibin Lei and David B. Cooper, “Linear Programming Fitting of Implicit Polynomials”, IEEE Transactions on Pattern Recognition and Machine Intelligence, February 1998 (Vol. 20, No. 2) p. 212 - 217,

[12] Daniel, C. and F.S. Wood, “Fitting Equations to Data”, John Wiley & Sons, New York, 1980,

[13] Chambers, J., W.S. Cleveland, B. Kleiner, and P. Tukey, “Graphical Methods for Data Analysis”, Wadsworth International Group, Belmont, CA, 1983,