Searching in Constant Time and Minimum Space* †

(Minimæ Res Magni Momenti Sunt)

by

Andrej Brodnik

Waterloo, Ontario, Canada, 1995

©Andrej Brodnik 1995

*This report is based on the author's PhD thesis. Many results are joint work with J. Ian Munro.

†Supported in part by the Natural Science and Engineering Research Council of Canada under grant number A-8237 and the Information Technology Research Centre of Ontario.

Abstract

This report deals with techniques for minimal space representation of a subset of elements from a bounded universe so that various types of searches can be performed in constant time. In particular, we introduce a data structure to represent a subset of N elements of $[0, \ldots, M-1]$ in a number of bits close to the information-theoretic minimum and use the structure to answer membership queries in constant time. Next, we describe a representation of an arbitrary subset of points on an $M \times M$ grid such that closest neighbour queries (under L_1 and L_∞) can be performed in constant time. This structure requires $M^2 + o(M^2)$ bits. Finally, under a byte overlap model of memory we present an $M + o(M)$ bit, constant time solution to the dynamic one-dimensional closest neighbour problem (hence, also union-split-find and priority queue problems) on $[0, \ldots, M-1]$.

Acknowledgements

Since a joint authorship of theses is impossible, all I can do is most sincerely thank Prof. Ian Munro for all his time, patience, encouragement, and unlimited help. I will be always indebted to him not only for being an excellent mentor, but also for being a very generous mæcenas. Thank you, Ian!

I am also grateful to my other committee members: Prof. Anna Lubiw, Prof. Joseph Cheriyan, Prof. Frank Dehne, and Prof. Prabhakar Ragde, for their thorough reading and valuable suggestions, and David Clark and Alfredo Viola for their comments. The Department of Computer Science, its head Prof. Frank Tompa, and all its faculty and staff, especially Wendy, deserve a special thanks for an inspiring research environment. Thanks to the Math Faculty Computing Facility for a comfortable computing environment.

The number of those who through their devoted teaching showed me "how to inflate the balloon of knowledge" is too large for all of them to be listed here. However, I would like to mention Dr. Marjan Špegel and Prof. Boštjan Vilfan whose help was crucial when I decided to seek knowledge outside my home country of Slovenia.

For all those YABSs and :-), thanx to Claudia, Ka Yee, Marzena, Mei, Alex, Dexter, Dmitri, Igor, Kevin, Naji, Pekka, Peter, Piotr, Robert, Rolf, and all of my other friends at the Department, and especially to the "WE-THEY ♠♡◇♣ gang": Glenn, David, and Alfredo. Further help came from numerous friends outside the Department and especially from a small local Slovene community and their association *Sava*.

Last, but not least, words are not enough to thank my daughter Helena, my wife Pija, and my mother Dr. Tatjana Brodnik, who all the time believed in and stood by me.

To my grandfather
Mojemu dedku

Contents

1 Introduction 1

1.1 Organization and Guide for the Reader 2

2 The Name of the Game 5

2.1 Notations and Definitions 6

2.2 The Spirit and the Body of a Computation 7

2.3 The Random Access Machine 8

2.3.1 Putting More Muscles on the Body 9

2.4 Associative Memory 9

2.5 The Extended Random Access Machine: ERAM 10

2.6 The Random Access Machine with Bytes Overlapping: RAMBO 11

3 Membership in Constant Time and Minimum Space 15

3.1 Introduction 16

3.2 Notation, Definitions and Background 17

3.2.1 A Short Walk Through the Literature 18

3.3 Solution for the Static Case 20

3.3.1 Indexing – Solution for $r \geq \log_\lambda M$ 20

3.3.2 Complete Solution 22

3.4 Static Solution Using $B + o(B)$ Space 27

3.4.1 Sparse Subsets 27

3.4.2 Dense Subsets 27

3.5 Dynamic Version 29

3.5.1 Memory Management Scheme 29

3.5.2 Solution 31

3.6 Two “Natural” Examples 33

3.7 Discussion and Conclusions 33

4 Word-Size Parallelism 35

4.1 Introduction . . . 36

4.2 Linear Registers . . . 36

4.3 The Technique . . . 38

4.4 Rectangular Registers . . . 43

4.5 Multidimensional Registers . . . 50

4.6 Extremal Bits in Linear Registers . . . 54

4.7 Extremal Bits in Rectangular Registers . . . 57

4.8 Extremal Bits in Multidimensional Registers . . . 61

4.9 Conclusions . . . 62

5 The Closest Neighbour in Constant Time 63

5.1 Introduction . . . 64

5.2 Definitions and Background . . . 65

5.2.1 Literature Background . . . 66

5.3 One Dimension . . . 68

5.4 Two Dimensions . . . 71

5.4.1 Circles and Some More Circles . . . 71

5.4.2 L_∞, or How Circles Became Squares . . . 75

5.4.3 L_1, or How Circles Became Diamonds . . . 83

5.4.4 L_2, or the Trouble with Curved Circles . . . 91

5.4.5 Conclusion . . . 93

5.5 Many Dimensions . . . 94

5.5.1 Introduction . . . 94

5.5.2 Search for the Closest Neighbour . . . 95

5.6 Conclusions and Discussion . . . 101

6 One-Dimensional Dynamic Neighbour Problem in Constant Time 103

6.1 Introduction and Motivation . . . 104

6.2 Related Problems and Literature . . . 104

6.2.1 Related Problems 105
6.2.2 And the Last Stroll Through the Literature 106
6.3 Stratified Trees and Binary Tries 107
6.3.1 Tagging Trees 108
6.3.2 Where to Look for Neighbours 109
6.4 The Solution 111
6.4.1 The Data Structure 111
6.4.2 Finding Internal Nodes 113
6.4.3 The Algorithms 114
6.4.4 Final Improvements and Simplifications 116
6.5 Conclusion, Discussion and Open Questions 118

7 Summary and Conclusions **121**

Bibliography **125**

A Glossary of Definitions **135**

B Notation **137**

Tables

3.1 Ranges of N and structures used to represent set $\mathcal{N}$. 22

3.2 Space usage for sets of primes and SINs for various data structures. 33

6.1 A mapping between the neighbourhood and interval sets problems. . . . 105

6.2 A mapping of the priority queue problem onto the neighbourhood problem. 106

Figures

2.1 Taxonomy of computer models. 8

2.2 Initial content of a memory. 11

2.3 Difference between updated memory contents of the usual RAM and implicit RAMBO. 12

2.4 Directed graph representing an implicit RAMBO from Example 2.1. 13

4.1 Graphic illustration of word-size parallelism. 38

4.2 Distribution of a block across a linear register. 40

4.3 Spreading of bits across a linear register. 41

4.4 Two step compression of an (s,k)-sparse register. 42

4.5 Column- and row-stripe registers $S^C_{i,j}$ and $S^R_{i,j}$. 44

4.6 Spreading of a stripe across a rectangular register. 47

4.7 Graphical representation of rectangular registers $P^{\diagup}_{c,\delta}$ and $P^{\diagdown}_{c,\delta}$ for $\delta = \frac{3c}{r} = \frac{3c^2}{b}$. 48

4.8 Shifting of $P^{\diagup}_{r,\delta}$ and $P^{\diagdown}_{r,\delta}$ k bits to the right. 49

4.9 Shifting of $P^{\diagup}_{r,\delta}$ and $P^{\diagdown}_{r,\delta}$ k bits to the left. 50

4.10 Results of `ShiftRightColumns(`$P^{\diagup}_{r,\delta}$`, ` k`, FALSE)` and `ShiftRightColumns(`$P^{\diagdown}_{r,\delta}$`, ` k`, TRUE)`. 51

4.11 Rectangular interpretation of a hyper-cubic register with bits set if $\pi_k = \pi$ is a column-stripe register. 52

4.12 Relation between k^{th} and l^{th} index of $H_{k,l}$ and $H_{k,-l}$. 53

5.1 Regular hexagons as tiling polygons. 72

5.2 Query point T, empty circles, a circle of candidates, and an enclosing polygon in two dimensions. 73

5.3 Wedge and a corner area of a polygon as defined by empty circles $\mathscr{C}_1$ and $\mathscr{C}_3$. 74

5.4 Four search regions in a plane. 76

5.5 Query point T on a tile $\mathscr{T}_0$ with its direct neighbour tiles $\mathscr{T}_1$, $\mathscr{T}_2$, $\mathscr{T}_3$ and $\mathscr{T}_4$, corresponding empty circles $\mathscr{C}_1$, $\mathscr{C}_2$, $\mathscr{C}_3$ and $\mathscr{C}_4$, an enclosing rectangle $\mathscr{P}_0$, and corner areas $\mathscr{A}_1$, $\mathscr{A}_2$, $\mathscr{A}_3$ and $\mathscr{A}_4$. 79

5.6 Corner area $\mathscr{A}_1$ limited by a distance between centres of empty circles $\mathscr{C}_1$ and $\mathscr{C}_2$. 80

5.7 Small universe under the norm L_1. 84

5.8 Mapping of points from the local coordinate system into bits of a register pair. 85

5.9 Big universe tiled by diamonds. 88

5.10 Placement of corner areas under the norm L_1. 90

5.11 Corner area $\mathscr{A}_4$ limited by the distance between centres of empty circles $\mathscr{C}_4$ and $\mathscr{C}_1$. 91

5.12 Circles under L_2. 93

6.1 Underlying complete binary tree (trie) of the stratified tree for a subset of the universe of size 16. 108

6.2 A tree with boxed splitting nodes. 109

6.3 Element x as the smallest in many subtrees. 110

6.4 Pointers to both neighbours of element $e \notin \mathcal{N}$ are either at the lowest left (n_l) or right (n_r) splitting node. 111

6.5 Overlapped memory, modeled as a complete binary tree of height $m = 5$, with marked `reg[3]`. 112

6.6 The situation after an insertion of e in $\mathcal{N}$. 116

Algorithms

3.1 Membership query if `elt` is in $\mathcal{N} \subseteq \mathcal{M}$, where $\mathcal{N}$ is represented by a data structure `data` and the $|\mathcal{M}| = M$. 25

4.1 Generic form of word-size parallelism. 38

4.2 Distribution of a block of at most s bits l times across the register. . . . 40

4.3 Spreading of a block of $k \leq s$ bits across the register in a reverse order creating an $(s-k, k)$-sparse register. 41

4.4 Compression of an (s, k)-sparse register. 42

4.5 Generation of $S^C_{i,j}$ from $S^C_{0,1}$. 45

4.6 Generation of $S^R_{i,j}$ from $S^R_{0,1}$. 46

4.7 Spreading of columns across the register in a reverse order creating an s-column sparse register. 48

4.8 Shifting of the rectangular register x k columns to the right. 50

4.9 Searching for the left most set bit in an s-small linear register. 54

4.10 Searching for the left most set bit in the linear register x. 56

4.11 Counting the number of set bits in individual columns of the s-column sparse rectangular register x. 58

4.12 Searching for the left most non-empty column in the s-column sparse rectangular register x. 59

4.13 Searching for the left most non-empty column in the $r \times c$ rectangular register x. 60

4.14 Searching for the left most set bit in the $r \times c$ rectangular register x. . . 61

4.15 Searching for the left most set bit in the kth dimension of the hyper-cubic register x. 61

5.1 Searching for the closest neighbour of point $T = (x)$ in an m-point universe `domain`. 69

5.2 Generalized searching for the closest neighbour of the point T in a small universe. 70

5.3 Searching for the closest neighbour of the query point T in one dimension. 70

5.4 General version of a search for the closest neighbour of T in a small universe under the norm L_∞. 78

5.5 Searching for the closest neighbour of T in a corner area starting from `tile` under the norm L_∞. 81

5.6 Searching for the closest neighbour of the query point T under the norm L_∞. 82

5.7 Searching for the closest set bit to the `bit` in the square `register` under the norm L_∞. 86

5.8 Searching for the closest neighbour of the query point T in a small domain under the norm L_1. 87

5.9 Computation of the tile on which the point T lies and of the origin of that tile. 89

5.10 Searching for the closest neighbour to the query point T under the norm L_1. 92

5.11 Generation of mask $\mathscr{H}_{k,l}$ or $\mathscr{H}_{k,-l}$. 97

5.12 Searching for the closest neighbour in a small d-dimensional universe under the norm L_∞. 98

5.13 Searching for the closest neighbour of the query point T in the d-dimensional universe under the norm L_∞. 100

6.1 Memory representation of a split tagged tree (data structure) used for the dynamic neighbourhood problem. 113

6.2 The lowest left and right splitting nodes of e. 114

6.3 The lowest common node of e and f, and an appearance of a corresponding bit in overlapped registers. 114

6.4 The neighbours of $e \in \mathcal{M}$ in $\mathcal{N}$. 115

6.5 Searching for the left and the right neighbour of e in $\mathcal{N}$. 115

6.6 Insertion of e into $\mathcal{N}$. 117

6.7 Deletion of e from $\mathcal{N}$. 118

Chapter 1

Introduction

Zrno na zrno – pogača,
kamen na kamen – palača.

slovenski pregovor

A grain on a grain – a cake,
a stone on a stone – a castle.

Slovene proverb

The number of data items computers deal with continues to increase rapidly. This is made possible by the increasing size of memory on the typical computer. Demand, however, continues to outpace physical capacity. In principle one could add disk space as needed, but in practice this is not always possible. For example, if we want to produce a product on CD-ROM, we are limited by the size of a disk, as requiring a user to repeatedly change disks has a catastrophic effect on performance.

The most natural approach to overcoming this problem is to "compress" the data by some standard method. However, such compressions, because of complex internal structure, usually increase the operation time dramatically. Returning to our CD-ROM example: 30 instead of 4 or 5 accesses to the disk, because of the format of the compressed data, can be prohibitively time consuming.

In this thesis we develop techniques that not only store data succinctly, but also permit basic operations to be performed quickly. The common thrust of all techniques presented here is their concern with the individual bits of the representation. Efficient encoding at the bit level permits us to decrease not only space requirements, but also the time bounds for the complete data structure – because we handle small stones better, the whole castle is in a better shape.

As a case study we use several simple data types. First, we study a simple membership problem in which the only operation is to determine whether a query value is a member of a subset of an underlying finite universe. Later, we extend the study to finding the closest element in the structure to a query point.

1.1 Organization and Guide for the Reader

This and the following chapter contain introductory material. The results of the thesis are presented in Chapters 3 through 6. Chapter 7 provides a brief summary and conclusions.

Chapter 2 presents the notation, several machine models, and how these models interrelate. Most of the material is fairly standard and is included primarily for the sake of completeness and to provide the reader with a self contained document. Lists of all terms that are defined and all notation that is used are given in Appendices A and B, respectively.

In Chapter 3 we focus on the problem of representing an arbitrary subset of size N chosen from a universe of size M. The method given permits constant time searches in a representation of size close to the information-theoretic lower bound. The result is then extended to the dynamic version of the problem.

Chapter 4 presents some preparatory results for the following two chapters. In particular this chapter contains a number of algorithms for manipulating individual words to perform tasks such as searching for extremal set bits in various forms of registers.

Chapter 5 deals with the problem of succinctly representing a subset of a finite grid so that the closest element to a query point can be found in constant time. Chapter 6 is concerned with the dynamic version of this problem in one dimension.

Chapter 2

The Name of the Game

名符其实

The beginning of wisdom is to name things by their right name.

Chinese wisdom

Models of computation help us to better understand the complexity of problems and to better describe classes of algorithms. The most popular and widely accepted models in computer science, in increasing order of power, are finite automata, pushdown machines, and Turing machines (for further references cf. [21, 62, 90, 107, 115]).

The third model, the Turing machine, has been proven to be equal in power to a number of other models (cf. [71, 72, 98, 101]) though the equivalence usually introduces a non-constant, but polynomial, factor in a run time (cf. [44]). The model used throughout this thesis is a *random access machine* (RAM) which is presented in § 2.3. The model is further generalized in § 2.5 and in § 2.6. Before discussing these models we first introduce some common mathematical notation and definitions also used in this work.

2.1 Notations and Definitions

To establish asymptotic bounds, the following standard notation is used (cf. [66, 69, 112]):

Definition 2.1 *Let $f(n)$ and $g(n)$ be two non-negative functions. Then:*

- $g(n) = o(f(n)) \iff \forall \delta > 0, \exists n_0 > 0 : \forall n > n_0, g(n) < \delta f(n)$
- $g(n) = O(f(n)) \iff \exists \delta > 0, \exists n_0 > 0 : \forall n > n_0, g(n) \leq \delta f(n)$
- $g(n) = \Omega(f(n)) \iff \exists \delta > 0, \exists n_0 > 0 : \forall n > n_0, g(n) \geq \delta f(n)$
- $g(n) = \omega(f(n)) \iff \forall \delta > 0, \exists n_0 > 0 : \forall n > n_0, g(n) > \delta f(n)$
- $g(n) = \Theta(f(n)) \iff g(n) = O(f(n)) \wedge g(n) = \Omega(f(n))$

Definition 2.2 *Let "$\log_\lambda$" denote the logarithm with base λ, then the function "$\log_\lambda^{(i)}$" is the so called iterated logarithm defined as (cf. [34, p.36])*

$$\log_\lambda^{(i)} x = \begin{cases} \log_\lambda x & \textit{if } i = 1 \\ \log_\lambda \log_\lambda^{(i-1)} x & \textit{otherwise} \end{cases} .$$

Function "$\log_\lambda^*$" is defined as the minimal number of iterations of a logarithm function such that the result is at most 1

$$\log_\lambda^* x = \min_i \left(\log_\lambda^{(i)} x \leq 1 \right) . \quad (2.1)$$

We use $\lg x$ to denote $\log_2 x$ and $\ln x$ for $\log_e x$.

Definition 2.3 *Let A denote Ackermann's function (cf. [34, p.450])*

- $A(1, j) = 2^j$ *for* $j \geq 1$
- $A(i, 1) = A(i-1, 2)$ *for* $i \geq 2$
- $A(i, j) = A(i-1, A(i, j-1))$ *for* $i, j \geq 2$

then its functional inverse is defined as

$$\alpha(m, n) = \min_{i \geq 1} \left(A \left(i, \left\lfloor \frac{m}{n} \right\rfloor \right) > \lg n \right) \quad .$$

2.2 The Spirit and the Body of a Computation

Kolmogorov and Uspensky in [71, 72] based their definition of an algorithm on the notion of state. A distinguished part of each state is its active part and the transformation to the following state depends solely on the information contained in that part. In particular, the active part consists of variables and relations among them that define the next step of the algorithm. This notion is similar to that of Turing machines ([107]) in which the finite control and data in the cell currently under a read head determine the next move. However, there is a major difference between the two models: Kolmogorov and Uspensky allow nodes of the graph to be connected arbitrarily. Therefore, the advantage of their model is a concept of locality of data and clearer representation of variables – i.e. the issue of *random access*.

The algorithm, in a sense, represents the "spirit" while the computer represents the "body" of a computation. To describe building blocks of computer models considered in this work we use a common taxonomy, shown in Figure 2.1. The taxonomy is similar to the one mentioned by Akl in [9] and consists of:

- an active block (processor),
- a passive block (memory, storage), and
- a communication channel (the path between processor and memory).

This is also a taxonomy of a so called *cell probe model* introduced by Yao in [116] and later extended by Fredman in Saks in [52] (see also [85, 86]). Under this model time is measured by the number of probes to memory.

The active block of all models we consider can perform a *constant* number of different h-bit operations, each of which takes unit time, and has a constant sized internal memory – i.e. a fixed number of h-bit arithmetic registers. Thus, if we define the capacity of a communication channel to be k bits, we can measure the time in the model as

$$\# \text{ of operations} \cdot \frac{\text{bits per operation}}{\text{capacity of channel}} = \# \text{ of operations} \cdot \frac{h}{k} = \text{number of probes} \quad . \tag{2.2}$$

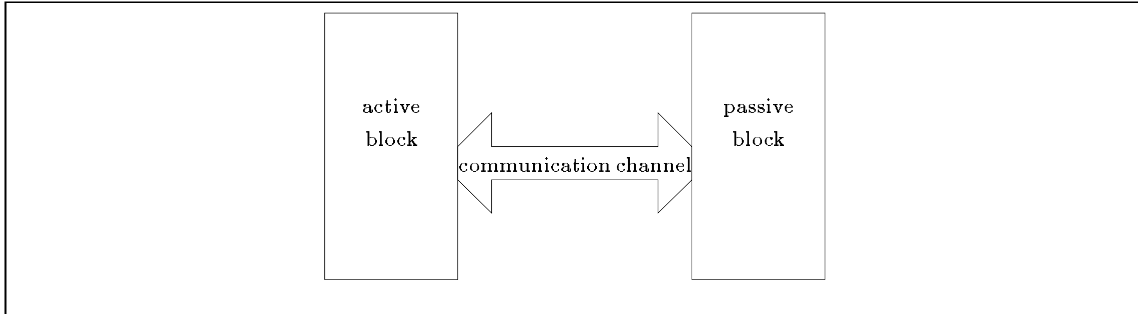

Figure 2.1: *Taxonomy of computer models.*

This is the same measure as in a cell probe model. Moreover, in most cases $k = \Theta(m)$ and $h = \Theta(m)$ where m is a size of a word (memory register) and will be defined later. On the other hand, the organization and accessibility of a passive block varies among the models. In fact, the differences in organization of the passive block are the key issue in this work.

2.3 The Random Access Machine

We use the *random access machine* (cf. [44, p.24], [5, pp.5-11]), as an idealized model of a von Neumann computer. In this model the static block consists of an unbounded number of memory registers labeled by non-negative integers, while the active block can perform the usual arithmetic operations (addition and subtraction), and comparison operations, compute addresses of memory registers, and move data between the registers.[1]

The computational power of a random access machine is the same as that of a pointer machine ([101]) or Turing machine. However, as shown by Ben-Amram and Galil in [14], a step by step emulation of a random access machine by a pointer machine can introduce a logarithmic slow-down factor which is matched by balanced trees (e.g. [2, 3, 12, 34, 74, 112]).

The logarithmic factor occurs only when dealing with so called *incompressible* data types which are defined analogously to incompressible strings in theory of Kolmogorov complexity (cf. [76, 77, 78]). On the other hand, Ben-Amram and Galil in the same work also proved the upper bound $O(t\alpha(s))$ for dealing with compressible data types (t is time spent by random access machine and s the size of its data structure). They show this by emulating a random access machine with many registers by a random access machine with a single register. However, their instruction set includes additional instructions such as integer multiplication and division.[2] This brings us to the next model.

---

[1] Neither integer multiplication nor division are included in this model.

[2] Shifting of the register is just a special case of multiplication or division by a power of 2.

2.3.1 Putting More Muscles on the Body

The main advantage of the cell probe model is its ability to do *any* operation on m bits in one unit of time (cf. uniform cost criterion [5, 12]). This notion is useful when considering lower bounds, but when we design an algorithm we have to fix the instruction set – the instruction set heavily influences the power of the machine model (cf. [15, 16, 100]).

For example, consider a subset of N elements from a universe of size M and store them in a table of $O(N)$ registers. For a model which includes integer division and multiplication, Fredman, Komlós and Szemerédi in [51] present a solution which permits a constant access time. On the other hand for the instruction set lacking these two operations Miltersen and Fich in [49] show a $\Omega(\log N)$ lower bound.

In [44] van Emde Boas discusses two particular random access machine models: MRAM and MBRAM. The former is a random access machine with integer multiplication, and the latter is an MRAM with bitwise Boolean operations (cf. [95]). Moreover, if we allow exponentially large registers and width of communication channel, one can use the MBRAM as the parallel processing model PRAM (cf. [61, 68, 110]) – MBRAM is considered a member of a second machine class ([44, p.6]). Such a model misses the notion of a serial processing computer and hence we *restrict the register size to m bits.* The value of m is likely to be 32 or 64 on processors available today (cf. [56]).

2.4 Associative Memory

In theoretical computer science, machines with *associative memory*, or "content addressable memory", are not usually considered. However, to avoid certain problems later in the definition of a general model, we briefly touch on them here. The notion of associative memory we use is primarily due to Potter ([92], see also [73] or the whole issue [1]).

In terms of our taxonomy (Figure 2.1), the interesting aspect is the static block. It consists of unlabeled (i.e. without addresses) memory locations which are accessed in two steps. In the first step a number of locations are tagged using mask and data registers, and in the second step all tagged locations are either modified or the content of an *arbitrary* such location is sent through the communication channel (cf. [91]).

The emulation of a random access machine by an associative memory machine is straightforward: a register, a, containing a datum, d, in a random access machine is stored as a pair, (a, d), in the associative memory, and there is no time loss in the emulation.

The problems arise with the inverse emulation. A machine with an associative memory can change its entire memory in a couple of steps. This is not possible on a random access machines. Hence, the worst case delay in a step-by-step emulation is as large as the memory. We consider this *possibility of changing the entire memory in a constant number of steps* to be unrealistic and thus undesirable under the general model of serial computation. Therefore, we explicitly forbid such a possibility.

2.5 The Extended Random Access Machine: ERAM

Our definition of a general model of computation to be used in this work is based on the taxonomy from Figure 2.1 and is derived from the definition of a random access machine. We want to bound the amount of information the active block can handle at a time and do this in two ways: first we limit the size of a register and the width of a communication channel to m bits (cf. § 2.3.1), and second we permit only m bits of a passive block to be changed at a time (cf. § 2.4).

Definition 2.4 *The* ***extended random access machine***, ***ERAM***, *consists of (cf. Figure 2.1):*

- *an active block, which performs the following augmented set of unit-time operations on b-bit* ***arithmetic registers*** *($b = \Theta(m)$):*
  - *comparisons;*
  - *branching (conditional and unconditional);*
  - *integer arithmetic (addition, subtraction, multiplication, and division); and*
  - *bitwise Boolean operations (conjunction, disjunction, and negation).*
- *a passive block, which consists of bounded, m-bit* ***memory registers*** *each of which is accessible by the active block, and*
- *a communication channel, which transports data between the active and the passive block and is m bits wide.*

The transportation of m bits of data through the communication channel is called a ***probe****. A single probe influences at most one memory register – it can change at most m bits in a passive block.*

Since widths of the channel and the memory registers are the same in the model, we can measure time, as mentioned in eq. (2.2), by the number of probes (cf. cell probe model in [52, 116]). Further, the only difference between the aforementioned MBRAM and ERAM is that ERAM has memory registers of a bounded width.

The notion of a memory register in Definition 2.4 is vague in the sense that it only says that predetermined aggregates of bits are somehow accessible. We will explore this notion further in § 2.6. However, in most of the thesis, we assume that passive block consists of a linear array of non-overlapping m-bit memory registers.

Finally, the bound on the register size and on the width of the transportation channel inherently limits the size of the set of all possible objects. If a model can transport m bits at a time, it is reasonable to assume that the set of all possible objects is no larger than 2^m. More precisely, we assume that the model can handle at least one and at most m objects at a time. This brings us to the notion of a bounded universe used throughout the thesis:

Definition 2.5 *The set of different objects or* ***universe*** $\mathcal{M} = \{0, 1, \ldots, M-1\}$ *has size* $M = 2^m$*, where* m *is the width of transportation channel and the size of memory registers.*

2.6 The Random Access Machine with Bytes Overlapping: RAMBO

By Definition 2.4, the passive block consists of a set of m-bit memory registers which can be accessed by the active block. So far we have assumed that the individual bits (not their values!) appearing in registers are distinct. We now drop this assumption and permit registers to *share* bits. In other words, a bit can now appear *simultaneously* in several registers, and so the update of a register will also change the contents of those registers which share bits with the updated register. As Fredman and Willard put it in [52], we speak of overlapping of registers (bytes) and so of a "random access memory with bytes overlapping" or RAMBO. Such memory would appear quite implementable in practice ([102]). We feel this has not been done because it has not been shown to give significantly faster solutions to important problems.

When the overlapping of registers is predefined, it is an attribute of a passive block, we speak of an *implicit RAMBO*. Conversely, when the active block can explicitly define the overlapping and hence the overlapping can be changed during the execution of a program, we speak of an *explicit RAMBO*. To illustrate the difference between these versions of RAMBO and regular RAM, we investigate in each model the influence of the same operation on passive blocks (memories) with the identical initial content:

EXAMPLE 2.1: Let the memories consist of eight three-bit registers ($m = 3$) with the initial content shown in Figure 2.2, and let `reg[`i`]` denote the ith register. The operation we consider is `write 3, "111"` which sets all bits in register `reg[3]` to "1": `reg[3].b[0] := 1`, `reg[3].b[1] := 1` and `reg[3].b[2] := 1`. The result of this operation on the usual RAM is shown in the first diagram of Figure 2.3 with the updated bits boxed.

register	b[0]	b[1]	b[2]
0	1	0	1
1	0	1	0
2	0	0	1
3	1	0	0

register	b[0]	b[1]	b[2]
4	0	1	0
5	0	0	1
6	1	0	0
7	0	1	0

Figure 2.2: *Initial content of a memory.*

In a RAMBO machine model, the same bit can appear in different registers. We model this by *appearance sets*:

register	b[0]	b[1]	b[2]
0	1	0	1
1	0	1	0
2	0	0	1
3	[1]	[1]	[1]
4	0	1	0
5	0	0	1
6	1	0	0
7	0	1	0

register	b[0]	b[1]	b[2]
0	1	0	1
1	$\boxed{1}_1$	1	0
2	$\boxed{1}_2$	$\boxed{1}_1$	1
3	$\boxed{1}_3$	$\boxed{1}_2$	$\boxed{1}_1$
4	0	$\boxed{1}_3$	$\boxed{1}_2$
5	0	0	$\boxed{1}_3$
6	1	0	0
7	0	1	0

Figure 2.3: *Difference between updated memory contents of the usual RAM and implicit RAMBO.*

Definition 2.6 *$\mathcal{B}$, the **appearance set** of a bit, consists of locations of a single bit in the memory registers – $\mathcal{B}$ essentially **is** this bit.*[3]

Thus, using the three-bit-register memory as above we define the implicit RAMBO with the following appearance sets:

$$\mathcal{B}_i = \{\texttt{reg[}(i+j) \bmod 2^m\texttt{].b[}j\texttt{]} \mid j = 0, 1 \ldots, m-1\} \tag{2.3}$$

where $i = 0, 1, \ldots, m-1$. That is, bit $\mathcal{B}_i$ appears as `reg[`i`+`j` MOD `2^m`].b[`j`]` simultaneously for all $0 \leq j < m$ (e.g. $\mathcal{B}_0 = \{$`reg[0].b[0]`, `reg[1].b[1]`, `reg[2].b[2]`$\}$, etc.). On this RAMBO we apply the same operation, `write 3, "111"`, which now assigns value 1 to appearance sets $\mathcal{B}_1$, $\mathcal{B}_2$, and $\mathcal{B}_3$. The result of the operation is shown in the second diagram of Figure 2.3, with updated bits boxed and subscripted with their respective appearance sets. $\mathcal{EF}$

The memory of an explicit RAMBO is also defined using appearance sets. However, in this case contents of appearance sets can change dynamically during the execution of the program. This brings us to the formal definition of RAMBO:

Definition 2.7 *The **random access machine memory model with bytes overlapping, RAMBO**, has the same active block and the communication channel as ERAM from Definition 2.4. However, its passive block consists of bits each of which appears one or more times in m-bit memory registers. The appearances of individual bits are specified by their respective appearance sets. The appearance sets are formed either statically, in the implicit RAMBO, or dynamically, in the explicit RAMBO.*

Note that a RAMBO, whose appearance sets each contain a single element, is an ERAM.

[3] We freely interchange bits and their appearance sets when this does not introduce an ambiguity.

A more "data-structure-flavored" description of a RAMBO's passive block is given by modeling appearance sets as vertices of a directed graph. The vertices (bits) are connected by multiple labeled directed edges representing registers. More precisely, if `reg[i].b[j]` $\in \mathcal{B}_k$ and `reg[i].b[j+1]` $\in \mathcal{B}_l$, then the graph has the edge $(\mathcal{B}_k, \mathcal{B}_l)$ labeled i. Therefore, any operation on register `reg[`i`]` affects m consecutive nodes (bits) connected by the edges labeled i. For example, the directed graph in Figure 2.4 represents the passive block of the implicit RAMBO from Example 2.1. The shaded area in the figure indicates the register `reg[3]`, studied in the example.

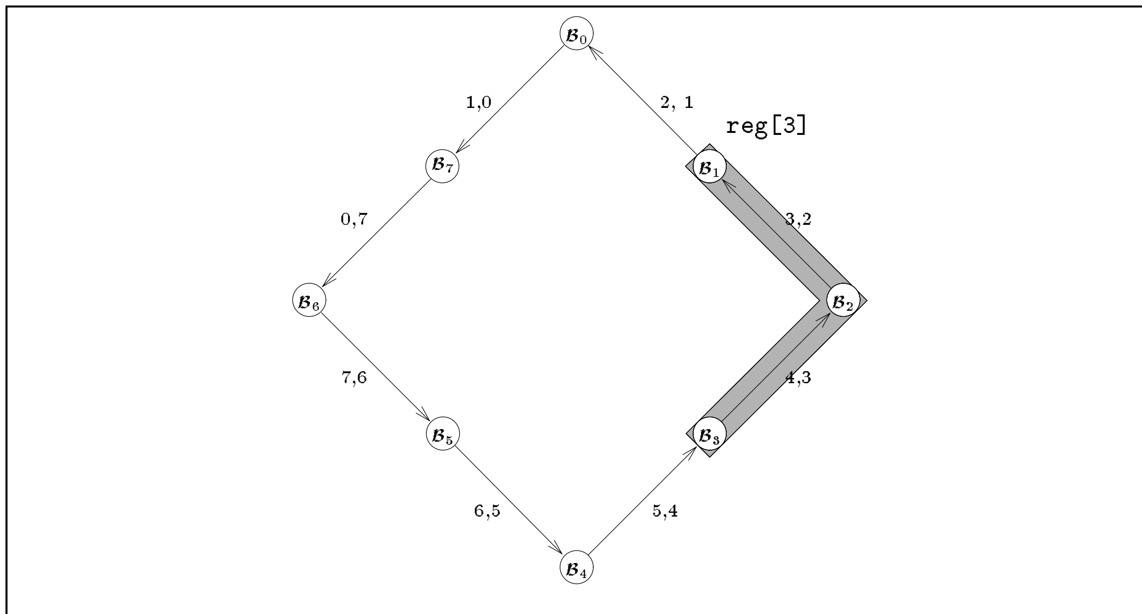

Figure 2.4: *Directed graph representing an implicit RAMBO from Example 2.1.*

In this thesis we use only an implicit RAMBO and this only in § 6 to solve the dynamic closest neighbour problem in one dimension. However, RAMBO itself is a source of countless open questions. For example, how does the topology of bits in the the passive block influence the power of the model? Or, what is the power of the explicit RAMBO (e.g. in comparison with PRAM or even with machines with an associative memory – cf. § 2.4)? The most interesting question is that of finding other problems in which the model is helpful.

Chapter 3

Membership in Constant Time and Minimum Space

Yet it isn't the gold that I'm
wanting
So much as just finding the
gold.

Robert Service, The Spell of the Yukon

This chapter deals with the problem of storing a subset of elements from the bounded universe so that membership queries can be performed in constant time and the space used for the data structure is small (cf. [24]).

The chapter starts with a brief introduction, followed by definitions, notation and background literature. In § 3.3 we present a constant time solution with space bound within a constant factor of the minimum required. In § 3.4 we improve the space bound to the minimum required plus some lower order terms. The results of § 3.3 and § 3.4 are extended in § 3.5 to the dynamic case. Finally, § 3.6 contains two explicit examples in which our technique is used.

3.1 Introduction

The membership problem, that is, maintaining a data structure to answer the question as to whether a certain element is in a given (static) set, is a fundamental problem in computer science. According to Yao ([116]), Minsky and Papert were the first to address the problem, which they called exact match ([87]). The problem also appears implicitly, hidden in problems including *search and retrieval* ([70]), *set manipulation* ([5]), *dictionary implementation* ([108]). Although these problems seem diverse, the solution to any of them first requires the answer to the membership problem. In most, cases only after a successful answer to this query can we proceed with the rest of the work.

Because the set membership problem is so central to the computer science, it has a great deal of work related to it. It is outside the scope to give a comprehensive overview of this work, and so, we address only those references which are closely related to our solution.

Since our first goal is a constant time bound per operation, we will use random access machines with integer division ([49]), and discard all classical tree-like structures, which have logarithmic worst case performance (cf. [2, 3, 34, 63, 64, 74, 112, 114]). This leaves us with perfect hash tables (functions) and bit maps, both of which have constant time worst case behaviour. However, hash tables and bit maps solve the membership problem using almost minimum space only when the set at hand contains, respectively, either only a few (or almost all) or about a half the elements of the universe.[1] In this chapter we address specifically the range between these two cases with the goal of introducing a data structure whose size is close to that minimum.

Roughly speaking, our basic approach is to use either perfect hashing or a bit map whenever one of them achieves the optimum space bound, and otherwise to split the universe into subranges of equal size. We discover that, with care, after a couple of iterations of this splitting, the subranges are small enough so that succinct indices into

[1] The meaning of "minimum space" is formally defined in eq. (3.1).

a single table of all possible configurations of these small ranges (*table of small ranges*) permit the encoding in the minimal space bound. This is an example of what we call *word-size truncated recursion* (cf. [54, 60]). Namely, the recursion continues only to a level of "small enough" subproblems, at which point indexing into a table of all solutions suffices. We can do this because, at this level, a single word in the machine model is large enough to encode a complete solution to each of these small problems.

3.2 Notation, Definitions and Background

Definition 3.1 *Given a universal set $\mathcal{M} = \{0, 1, \ldots, M-1\}$ with an arbitrary subset $\mathcal{N} = \{e_0, e_1, \ldots, e_{N-1}\}$, where N and M are known, the* ***static membership problem*** *is to determine whether a query value $x \in \mathcal{M}$ is in $\mathcal{N}$.*

It has an obvious dynamic extension:

Definition 3.2 *The* ***dynamic membership problem*** *is the static membership problem extended by two operations: insertion of an element $x \in \mathcal{M}$ into $\mathcal{N}$ (if it is not already in $\mathcal{N}$); and deletion of x from a set $\mathcal{N}$ (if it is in $\mathcal{N}$).*

Since solving either problem for $\mathcal{N}$ trivially gives a solution for $\overline{\mathcal{N}}$ we assume $0 \leq N \leq \frac{M}{2}$.

We use the ERAM machine model from Definition 2.4 (also cf. [44, 52, 116]) which performs the usual operations (including integer multiplication and division) on words (registers) in unit time. The size of memory registers is m=$\lceil \lg M \rceil$ bits (cf. Definition 2.5), which means that one memory register (word) can be used to represent a single element of $\mathcal{M}$, specify an arbitrary subset of a set of m elements, refer to some portion of the data structure, or have some other role that is a m-bit blend of these. For convenience we measure space in bits rather than in words.

We take as parameters of our problem M and N. Hence, the information-theoretic lower bound says that we need

$$B = \left\lceil \lg \binom{M}{N} \right\rceil \tag{3.1}$$

bits to describe any possible subset of N elements chosen from M elements. Since we are interested only in solutions which use $O(B)$ or $B + o(B)$ bits for a data structure, we will omit the ceiling and floor functions altogether.

Using Stirling's approximation (cf. [88, p.184]) we compute from eq. (3.1) a lower bound on the number of bits required,

$$B = \lg \binom{M}{N} = \lg M! - \lg N! - \lg(M-N)!$$

$$\begin{aligned}
&\approx M \lg M - N \lg N - (M-N) \lg(M-N) && \text{with error} \leq \lg N + O(1) \\
&= M \lg M - N \lg N - (M-N)(\lg M + \lg(1 - \frac{N}{M})) \\
&= N \lg \frac{M}{N} - (M-N) \lg(1 - \frac{N}{M}) && (3.2)
\end{aligned}$$

The error bound is computed from Robbins' approximation of the function $n!$ ([88, p.184]). Further, defining the *relative sparseness* of the set $\mathcal{N}$

$$r = \frac{M}{N} \ , \tag{3.3}$$

and observing that $2 \leq r \leq \infty$, we rewrite the second term of eq. (3.2) into

$$N \leq -N \cdot ((r-1) \lg(1 - r^{-1})) \leq \frac{1}{\ln 2} N \approx 1.4427 \ldots N \ . \tag{3.4}$$

Thus, for the purposes of much of this work, we can use

$$B \approx N \lg \frac{M}{N} \equiv N \lg r \tag{3.5}$$

with an error bounded in eq. (3.4) by $\Theta(N)$ bits. Note, the error is positive and hence eq. (3.5) an underestimate.

An intuitive explanation of eq. (3.5) is that $\mathcal{N}$ is fully described when each element in $\mathcal{N}$ "knows" its successor. Since there are N elements in $\mathcal{N}$, the average distance between them is $r = \frac{M}{N}$ and to encode this distance we need $\lg r$ bits. Moreover, it is not hard to argue that the worst case, and indeed the average one, occurs when elements are fairly equally spaced. This is exactly what eq. (3.5) says.

3.2.1 A Short Walk Through the Literature

We address three aspects: the static case, the dynamic case, and the information-theoretic tradeoffs. In the first two cases it has generally been assumed that there is enough space to list those keys that are present or to list all the answers to queries. The first assumption takes us to hash tables, and the second to a bit map. Here we deal with the situation in which we can not always afford the space needed to use either structure directly.

We start with the static case, and, in particular, with the implicit data structures in Yao's sense. His notion of an implicit structure was that one has room to store only N data items, although these were not constrained to be those in the "logical" table. In other words, by the term implicit structure he meant a structure using only N registers ($N \lg M$ bits) and no additional storage.

For such data structures Yao ([116]) showed that in a bounded universe there always exists some subset for which any implicit data structure requires at least logarithmic

search time. Fiat and Naor ([47]) later improved the bound to N. They proved that there is no constant time implicit solution when N is less than $\lg^{(O(1))} M$, but that there is such a solution when $N = \Omega((\log M)^\epsilon)$.

Adding some storage changes the situation. For example, with one more register ($\lg M$ bits) Yao ([116]) showed that there exists a constant time solution for $N \approx M$ or $N \leq \frac{1}{4}\sqrt{\lg M}$, while Tarjan and Yao ([105]) presented a constant time solution using $O(N \lg M)$ bits of space for $N = O(M^\epsilon)$ and $0 < \epsilon \leq 1$. However, there was still a substantial gap between $\Theta(M)$ and $O(M^\epsilon)$. Fredman, Komlós and Szemerédi ([51]) closed this gap, developing a constant time algorithm with a data structure of $N \lg M$ bits plus $O(N\sqrt{\log N} + \log^{(2)} M)$ additional bits. Fiat, Naor, Schmidt and Siegel in [48] decreased additional bits to $\lceil \lg N \rceil + \left\lceil \log^{(2)} M \right\rceil + O(1)$. Moreover, combining their result with Fiat and Naor's ([47]) construction of an implicit search scheme for $N = \Omega((\log M)^p)$ they had a scheme using fewer than $(1+p)\left\lceil \log^{(2)} M \right\rceil + O(1)$ additional bits.

Mairson in [80] took a different approach. He assumed all structures are implicit in Yao's sense and the additional storage represents the complexity of a searching program. Following a similar path, Schmidt and Siegel in [99] proved a lower bound of $\Omega(\frac{N}{k^2}e^{-k} + \log^{(2)} M)$ bits spatial complexity for k-probe oblivious hashing. In particular, for a 1-probe hashing this gives a spatial complexity of $\Theta(\log^{(2)} M + N)$ bits.

For the dynamic case, Dietzfelbinger et al. ([38]) proved an $\Omega(\log N)$ worst case lower bound for a class of realistic hashing schemes. In the same paper they also presented a scheme which, using results of [51] and a standard doubling technique, achieved constant amortized expected time per operation with a high probability. However, the worst case time per operation (non-amortized) was $\Omega(N)$. Later Dietzfelbinger and Meyer auf der Heide in [39] upgraded the scheme and achieved constant worst case time per operation with a high probability. A similar result was also obtained by Dietzfelbinger, Gil, Matias and Pippenger in [37].

In the data compression technique described by Choueka et al. ([32]), a bit-vector is hierarchically compressed. First, the binary representations of elements stored in the dictionary are split into pieces of equal size. Then the elements with the same value of the most significant piece are put in the same bucket, and the technique is recursively applied within each bucket. When the number of elements which fall in the same bucket becomes sufficiently small, they are stored in a compressed form. The authors experimentally tested their ideas but did not formally analyze them. They claim their result gives a relative improvement of about 40% over similar methods.

Finally, we turn to some information-theoretic work. Elias ([41]) addressed a more general version of the static membership problem which involved several different types of queries. For these queries he discussed a tradeoff between the size of the data structure and the average number of bit probes required to answer the queries. In particular, for the set membership problem he described a data structure of a size $N \lg \frac{M}{N} + O(N)$ (using

eq. (3.5), $B+o(B)$) bits which required an average of $(1+\epsilon)\lg N+2$ bit probes to answer a query. However, in the worst case, the method required N bits. Elias and Flower in [42] further generalized the notion of a query into a database. They defined the set of data and a set of queries, and in a general setting studied the relation between the size of the data structure and the number of bits probed, given the set of all possible queries. Later, the same arrangement was more rigorously studied by Miltersen in [85].

3.3 Solution for the Static Case

As noted earlier, if more than half the elements are present we can solve the complementary problem. The static solution is presented in two parts. The separation point between them is defined in terms of relative sparseness

$$r_{sep} = \log_\lambda M \quad , \tag{3.6}$$

or the size of sets

$$N_{sep} = \frac{M}{r_{sep}} = \frac{M}{\log_\lambda M} \quad . \tag{3.7}$$

where the predefined constant $\lambda > 1$ is used later for fine tuning a practical solution. First we describe a solution for $r \equiv \frac{M}{N} \geq r_{sep}$ and then we deal with the case when $r_{sep} \geq r \geq 2$, that is, with $N \leq N_{sep}$ and then $N_{sep} \leq N \leq \frac{M}{2}$. We refer to the first case as *sparse* and to the second as *dense*.

Both of these cases have an extreme situation. In the dense case this occurs when $N \geq \alpha M$, for some $0 < \alpha \leq \frac{1}{2}$ and requires $\Omega(M)$ bits. Thus, we use a bit map of size M to represent the set $\mathcal{N}$. In the sparse case, a special situation occurs for very sparse sets. When $N \leq M^{1-\epsilon}$ for some $0 < \epsilon \leq 1$, we are allowed $\Theta(N \log M)$ bits which is enough to list all the elements of $\mathcal{N}$. For $N \leq c = O(1)$ we simply list them, beyond this we use a perfect hashing function of some form (cf. [47, 48, 51]). Note, that all these structures allow us to answer a membership query in constant time. Moreover, the constant is at most 3 or 4 using the hashing schemes suggested.

The parameters c, α, ϵ, and λ are used in tuning specific example. They have no particular role in the asymptotic solution as addressed in § 3.4.

3.3.1 Indexing – Solution for $r \geq \log_\lambda M$

We first focus on the sparse case $r \geq r_{sep}$ defined in eq. (3.6). The bottom end of this range, that is $r = r_{sep}$, typifies the case in which both simple approaches require too much space, as $B = \Theta(N \log^{(2)} M)$. Indeed, this solution suggests the first iteration of our general approach for the dense case.

Lemma 3.1 *If $N \leq N_{sep} = \frac{M}{\log_\lambda M}$ (i.e. $\infty > r \geq r_{sep} = log_\lambda M$) for some $\lambda > 1$, then there is an algorithm which answers a membership query in constant time using $O(B)$ bits of space for a data structure.*

Proof: The idea is to split the universe $\mathcal{M}$ into p buckets, where p is to be determined later. The data falling into individual buckets are then organized using perfect hashing. The buckets are contiguous ranges of equal sizes, $M_1 = \frac{M}{p}$, so that a key $x \in \mathcal{M}$ falls into bucket $\left\lfloor \frac{x}{M_1} \right\rfloor$. To reach individual buckets, we index through an array of pointers.

To consider this in detail, let us assume that we split the universe into p buckets. We build an index of pointers to individual buckets, where each pointer occupies $\lceil \lg M \rceil$ bits. Hence, the total size of the index is $p \cdot \lceil \lg M \rceil$ bits.

We store all elements that fall in the same bucket in a perfect hash table ([47, 48, 51]) for that bucket. Since the ranges of all buckets are equal, the space required for these hash tables is $\left\lceil \lg \frac{M}{p} \right\rceil$ bits per element, and so, to describe all elements in all buckets we require only $N \cdot \left\lceil \lg \frac{M}{p} \right\rceil$ bits. In addition to this, we also need some space to describe individual hash tables. If we use a particular implementation due to Fiat, Naor, Schmidt and Siegel ([48]) the additional space for a bucket i is bounded by $\lceil \lg N_i \rceil + \left\lceil \lg^{(2)} M_1 \right\rceil + O(1)$ where N_i is the number of elements in a bucket. Thus, the additional space to describe all hash functions is bounded by $p \cdot (\lg N + O(1)) + \lg^{(2)} M$. Putting the pieces together we get the expression for the size of the structure

$$S = p \cdot \lg M + N \cdot \lg \frac{M}{p} + p \cdot (\lg N + O(1)) + \lg^{(2)} M \quad . \tag{3.8}$$

To minimize it we solve

$$\frac{dS}{dp} = \lg M - N \cdot \frac{\lg e}{p} + \lg N + O(1) = 0$$

and get the minimum at

$$p = \frac{N}{\ln M + \ln N + O(1)}$$

buckets. It turns out that the approximation of

$$p = \frac{N}{\lg M} \tag{3.9}$$

is adequate and simplifies the analysis. So, from eq. (3.8), the size of the data structure is

$$S \; = \; N + N \cdot (\lg \frac{M}{N} + \lg^{(2)} M) + N \cdot \frac{\lg N + O(1)}{\lg M} + \lg^{(2)} M \qquad \text{using eq. (3.3)}$$

$$\begin{aligned}
&\leq \quad N \cdot \lg r + (N \cdot \lg r)\frac{\lg^{(2)} M}{\lg r} + N + N \cdot \frac{\lg N + O(1)}{\lg M} + \lg^{(2)} M \quad \text{using eq. (3.5)} \\
&= \quad B + B \cdot \frac{\lg^{(2)} M}{\lg r} + o(B) \ . \qquad (3.10)
\end{aligned}$$

Hence, for a sparse subset, i.e. $r \geq r_{sep}$, the size of the structure is $O(B)$ bits. It is also easy to see that the structure permits constant time search. $\mathcal{QED}$

Note that if $r_{sep} \geq \lg M$ (i.e. in eq. (3.6) $\lambda < 2$) the lead term of eq. (3.10) is less than $2B$. Moreover, if in eq. (3.8) we do not count the hash table descriptions of individual buckets, the optimal number of buckets becomes $p = \frac{M}{\ln M}$; in which case the lead term of eq. (3.10) is less than $2B$ for $\lambda < e$.

Table 3.1 summarizes the cases covered so far. We still require a solution for

$$\begin{array}{rcccccc}
N_{sep} = \dfrac{M}{\log_\lambda M} & \leq & N & \leq & \alpha M & \leq & \dfrac{M}{2} \\
r_{sep} = \log_\lambda M & \geq & r & \geq & \dfrac{1}{\alpha} & \geq & 2 \ .
\end{array} \qquad (3.11)$$

Next we assume N (and so r) lies in this awkward range, that is, the $\mathcal{N}$ is *moderately dense.*

range of N	B	structure
0	0	nil
1 to c	$\Theta(\log M)$	unordered list
c to $M^{1-\epsilon}$	$\Theta(N \log M)$	hash table
$M^{1-\epsilon}$ to $\dfrac{M}{\log_\lambda M}$	$\Theta(N \log^{(2)} M)$	indexing
$\dfrac{M}{\log_\lambda M}$ to αM	$\Theta(N \log r)$	?
αM to $\frac{1}{2} M$	$\Theta(M)$	bit map

Table 3.1: *Ranges of N and structures used to represent set $\mathcal{N}$.*

3.3.2 Complete Solution

Consider, then, sets $\mathcal{N}$ whose sizes lie in the range given in eq. (3.11). For such moderately dense $\mathcal{N}$ we apply the technique of Lemma 3.1, that is, split the universe $\mathcal{M}$ into equal-range buckets. However, this time the ranges of buckets remain too big to use hash tables, and therefore we apply the splitting scheme again. In particular, we treat each bucket as a new, separate but smaller, universe. If its relative sparseness falls in the range defined by eq. (3.11) (with respect to the size of its smaller universe) we recursively split it.

Such a straightforward strategy leads, in the worst case, to an $\Theta(\log^* M)$ level structure and therefore to an $\Theta(\log^* M)$ search time. However, we observe that at each level the number of buckets with the same range increases and ultimately there must be so many small subsets that not all can be different. Therefore we build a table of all possible subsets of universes of size up to a certain threshold. This *table of small ranges* allows replacement of buckets in the main structure by pointers (indices) into the table. Although the approach is not new (cf. [54, 60]), it does not appear to have been given a name. We refer to the technique as *word-size truncated recursion* because the pointers into the table of small ranges are small enough to be stored in one m-bit word ($m = \lg M$). In our structure the truncation occurs after two splittings.[2] In the rest of this section we give a detailed description of the structure and its analysis.

On the first split we could partition the universe into $\frac{B}{\lg M}$ buckets, but to ensure continuity with the sparse case discussed before we split the universe into

$$p = \frac{N_{sep}}{\lg M} = \frac{\frac{M}{r_{sep}}}{\lg M} = \frac{M}{(\log_\lambda M) \cdot (\lg M)} \tag{3.12}$$

buckets, each of which has a range $M_1 = \frac{M}{p}$. Hence, at the separating point between sparse and dense buckets eq. (3.12) becomes eq. (3.9). At the second level we have again relatively sparse and dense buckets which now separate at the relative sparseness

$$r'_{sep} = \log_\lambda M_1 = \log_\lambda \frac{M}{p} = O(\log^{(2)} M) \ . \tag{3.13}$$

For sparse buckets we apply the solution from § 3.3.1 and for very dense ones with more than the fraction α of their elements present we use a bit map. For the moderately dense buckets, with relative sparseness within the range defined in eq. (3.11), we re-apply the splitting. However, this time the number of buckets is (cf. eq. (3.12))

$$p_1 = \frac{\frac{M_1}{r'_{sep}}}{\lg M_1} = \frac{M_1}{r'_{sep} \cdot (\lg M_1)} \tag{3.14}$$

so that each of these smaller buckets has the same range

$$M_2 = \frac{M_1}{p_1} = O((\log^{(2)} M)^2) \tag{3.15}$$

because $\lg M_1 = O(\log^{(2)} M)$.

At this point we build the table of small ranges, which consists of bit map representations of all possible subsets chosen from the universe of size M_2. Thus we can replace

[2] In fact, because all our second level buckets are of the same range, our table of small ranges consist only of all possible subsets of a single universe.

buckets in the main structure with "indices" (pointers of varying sizes) into the table. We order the table first according to the number of elements in the subset and then lexicographically. If we store, as the representation of a pointer to the table of small ranges, a record consisting of two fields (ν, the number of elements in the bucket, which takes $\lceil \lg M_2 \rceil$ bits; and β, the lexicographic order of the bucket in question among all buckets containing ν elements, which from eq. (3.1) is $\left\lceil \lg \binom{M_2}{\nu} \right\rceil$ bits), then the actual position of the corresponding bit map of the bucket is

$$\sum_{i=1}^{\nu-1} \binom{M_2}{i} + \beta - 1 \ . \tag{3.16}$$

The sum is found by table lookup and so a search is performed in constant time.

This concludes the description of our data structure used by Algorithm 3.1 to answer the membership queries. Probes to the data structure `data` are explicitly denoted by the use of procedure `Probe` with an additional parameter describing how the structure is interpreted and which part is read. The structure allows constant time membership queries, but remains to be seen how much space it occupies. In the analysis we are interested only in moderately dense subsets (cf. eq. (3.11)), as otherwise we use the structure of § 3.3.1.

First we analyze the main structure and begin with the following lemma:

Lemma 3.2 *Suppose we are given a subset of N elements from the universe M and B as defined in eq. (3.1). If this universe is split into p buckets of sizes M_i containing N_i elements each (now, using eq. (3.1), $B_i = \left\lceil \lg \binom{M_i}{N_i} \right\rceil$) then $B + p > \sum_{i=1}^{p} B_i$.*

Proof: If $\sum_{i=1}^{p} M_i = M$ and $\sum_{i=1}^{p} N_i = N$ we know from [88, p.196, inequality 3.1.39] that $0 < \prod_{i=1}^{p} \binom{M_i}{N_i} \leq \binom{M}{N}$ and therefore $\sum_{i=1}^{p} \lg \binom{M_i}{N_i} \leq \lg \binom{M}{N}$. On the other hand, from eq. (3.1) we have $B_i = \left\lceil \lg \binom{M_i}{N_i} \right\rceil$ and therefore $B_i - 1 < \lg \binom{M_i}{N_i} \leq B_i$. This gives us $\sum_{i=1}^{p}(B_i - 1) < B$ and finally $B + p > \sum_{i=1}^{p} B_i$. $\mathcal{QED}$

In simpler terms, Lemma 3.2 proves that if sub-buckets are encoded within the information-theoretic bound then the complete bucket is also within the information-theoretic minimum provided that the number of buckets is small enough ($p = o(B)$) and that the index does not take too much space.

The main structure itself is analyzed from the top to the bottom. The first level index consists of p pointers each of which is of size $\lg M$ bits. Therefore, using eq. (3.12) and eq. (3.7), the size of that complete index for the range defined in eq. (3.11) is

$$p \cdot \lg M = \frac{M}{\log_\lambda M} = N_{sep} = o(B) \ . \tag{3.17}$$

```
PROCEDURE Member (M, data, elt)
  IF M ≤ M₂ THEN            (* Use data as a pointer into a table of small ranges, TSR: *)
    (ν, β):= Probe (data, record);      (* number of elements and lexicographic order, *)
    pointer:= Probe (Binomials[ν], number) + β -1;   (* real pointer by eq. (3.16), *)
    RETURN (elt IN Probe (TSR[pointer], bit map))         (* bit map from the TSR. *)
  ELSE N:= Probe (data, number);                       (* Number of elements in N.  *)
    IF N ≥ M/2 THEN negate:= TRUE; N:= M-N                  (* We are solving N̄.  *)
    ELSE negate:= FALSE END;
                                                           (* How sparse set N is:  *)
    IF N = 0 THEN answer:= FALSE                           (* an empty set - r = ∞; *)
    ELSIF N ≤ c THEN                                (* a very sparse set - r ≥ c/M; *)
      answer:= FindOL (Probe (data, ordered list), elt)
    ELSIF N ≤ M^(1-ε) THEN                               (* less sparse set - r ≥ M^ε; *)
      answer:= FindHT (Probe (data, hash table), elt);
    ELSIF N ≤ M/log_λ(M) THEN                   (* even less sparse set - r ≥ log_λ M; *)
      answer:= Find2L (Probe (data, two level indexing structure from § 3.3.1), elt);
    ELSIF N ≤ α*M THEN                            (* moderately dense set - r ≥ 1/α:   *)
      M₁:= Floor (log_λ(M)*lg(M));      (* Split into buckets of range M₁ by eq. (3.12), *)
      data₁:= Probe (data[elt DIV M₁], index);     (* find bucket in which falls elt, *)
      answer:= Member (M₁, data₁, elt MOD M₁)        (* and recursively search it; *)
    ELSE                                                 (* very dense set - r ≥ 2 *)
      answer:= LookUpBM (Probe (data, bit map), elt);
    END;
    IF negate THEN RETURN NOT answer
    ELSE RETURN answer END;
  END
END Member;
```

Algorithm 3.1: *Membership query if* `elt` *is in* $\mathcal{N} \subseteq \mathcal{M}$, *where* $\mathcal{N}$ *is represented by a data structure* `data` *and the* $|\mathcal{M}| = M$.

For the sparse buckets on the second level we use solution presented in § 3.3.1 and for the very dense buckets ($r \leq \frac{1}{\alpha}$), we use a bit map. Both of these structures guarantee space requirements within a constant factor of the information-theoretic bound on the number of bits (see Table 3.1). If the same also holds for the moderately dense buckets, then, using Lemma 3.2 and eq. (3.17), the complete main structure uses $O(B)$ bits. Note, that we can apply Lemma 3.2 freely because the number of buckets p is $o(B)$.

It remains to see how large the encoding of the second level moderately dense buckets is, that is the encoding of buckets with sparseness in the range of eq. (3.11). For this purpose we first consider the size of bottom level pointers (indices) into the table of small ranges. As mentioned, the pointers are records consisting of two fields. The first field, ν (number of elements in the bucket), occupies $\lceil \lg M_2 \rceil$ bits, and the second field takes

B_ν, as defined in eq. (3.1). Since $B_\nu \geq \lceil \lg M_2 \rceil$, the complete pointer[3] takes at most twice the information-theoretic bound on the number of bits, B_ν. On the other hand, the size of an index is bounded using an expression similar to eq. (3.17). Subsequently, this, together with Lemma 3.2, also limits the size of moderately dense second level buckets to be within a constant factor of the information-theoretic bound. This, in turn, limits the size of the complete main structure to $O(B)$ bits.

It remains to compute the size of the table of small ranges. There are 2^{M_2} entries in the table and each of the entries is M_2 bits wide, where by eq. (3.15) $M_2 = O((\log^{(2)} M)^2)$. This gives us the total size of the table

$$\begin{aligned} M_2 \cdot 2^{M_2} &= O((\log^{(2)} M)^2 \cdot (\log M)^{\log^{(2)} M}) \\ &= O(\frac{\log\log M}{\log M} \cdot (\log^{(2)} M \cdot (\log M)^{1+\log^{(2)} M})) \\ &= o(\frac{\log r_{sep}}{r_{sep}} \cdot M) \qquad \text{by eq. (3.6)} \\ &= o(N_{sep} \cdot \log r_{sep}) = o(B) \qquad (3.18) \end{aligned}$$

for moderately dense sets (cf. eq. (3.11)). Moreover, this also bounds the size of the whole structure to $O(B)$ bits and, hence, proves in a connection with Lemma 3.1 the theorem:

Theorem 3.1 *There is an algorithm which solves the static membership problem in $O(1)$ time using a data structure of size $O(B)$ bits.*

Note the constants in order notation of Theorem 3.1 are relatively small. Algorithm 3.1 performs at most two recursive calls of `Member` and 7 probes of the data structure:

- 2 probes in the first call of `Member` (one to get N, and one to get $\texttt{data}_1$),
- 2 probes in the second call of `Member` (same as above), and
- 3 probes in the last call of `Member` (the first line of Algorithm 3.1) — the initial probe to get the number of elements in the bucket, ν, and the lexicographic order of the bucket, β; the next probe to get the sum in eq. (3.16) by table lookup; and the final probe into the table of small ranges.

The space requirement is certainly less than $2B$ bits, and in the next section we reduce it to $B + o(B)$ bits while retaining the constant query time. For fine tuning of the structure we can adjust the constants c, ϵ, λ, and α of Table 3.1 and Algorithm 3.1.

[3] Note, that the size of a pointer depends on the number of elements that fall into the bucket.

3.4 Static Solution Using $B + o(B)$ Space

The solution presented in the preceding section can easily be adapted to use $(1+\gamma)B + o(B)$ bits of space for arbitrary $\gamma > 0$. We can, in fact, construct a data structure requiring $B + o(B)$ bits.

First, we observe that for very dense sets ($r \leq \frac{1}{\alpha}$) we can not afford to use a bit map because it always takes $B + \Theta(B)$ bits of space. For a similar reason we can not afford to use hash tables for very sparse sets (i.e. $r \geq M^{1-\epsilon}$). Therefore, we categorize sets *only* as sparse or dense (and not moderately dense). Finally, to decrease the space usage to $B + o(B)$ bits at the separation point between sparse and dense sets (cf. eq. (3.10)) we redefine the point (cf. eq. (3.6) and eq. (3.7)) setting it to

$$r_{sep} = (\lg M)^{\lg^{(2)} M} \tag{3.19}$$

or, in other words, using eq. (3.3), to

$$N_{sep} = \frac{M}{(\lg M)^{\lg^{(2)} M}} \quad . \tag{3.20}$$

B mentioned above is the exact one from eq. (3.1), though for sparse sets we can still use the approximation $N \lg r$ from eq. (3.5) since in eq. (3.4) the error is bounded by $\Theta(N) = o(B)$.

3.4.1 Sparse Subsets

Again, sparse subsets are those whose relative sparseness is greater than r_{sep} . For such subsets we *always* apply the two level indexing from § 3.3.1. In fact, all equations from § 3.3.1, and in particular eq. (3.10), remain unchanged. However, this time the second term of eq. (3.10) becomes $o(B)$, because now the relative sparseness r is at least r_{sep} defined in eq. (3.19).[4] This proves the lemma:

Lemma 3.3 *If $N \leq N_{sep}$ defined in eq. (3.20) (i.e. $\infty > r \geq r_{sep}$ defined in eq. (3.19)), then there is an algorithm which answers a membership query in constant time using $B + o(B)$ bits of space for a data structure.*

3.4.2 Dense Subsets

The dense subsets are treated in exactly the same way as were moderately dense subsets in § 3.3.2. Thus most of the analysis can be taken from there with the appropriate change

[4]Indeed, it is sufficient to set $r_{sep} = (\log_\lambda M)^{\omega(1)}$ and still have asymptotically $B + o(B)$ solution but it is easy to see that the larger is r_{sep} the smaller is the second term of eq. (3.10).

of r_{sep} (cf. eq. (3.6)) and r'_{sep} (cf. eq. (3.13)). To compute the size of the main structure, we first bound the size of pointers into the table of small ranges. Recall that each pointer consists of two fields: the number of elements in the bucket, ν, and the lexicographic order of the bucket in question among all buckets with ν elements, β. Although the number of bits necessary to describe ν can be as large as the information-theoretic minimum for some buckets, this is not true on the average. By Lemma 3.2, all pointers together occupy no more than $B + o(B)$ bits.

Furthermore, indices are also small enough so that all of them together occupy $o(B)$ bits (cf. eq. (3.17)). As a result we conclude that the main structure occupies $B + o(B)$ bits of space. It remains to see how big the table of small ranges is.

First, since r_{sep} was redefined in eq. (3.19) we have, by eq. (3.12), on the first level

$$p = \frac{M}{r_{sep} \cdot \lg M} = \frac{M}{(\lg M)^{1+\lg^{(2)} M}} \tag{3.21}$$

buckets each of the range

$$M_1 = \frac{M}{p} = r_{sep} \cdot \lg M = (\lg M)^{1+\lg^{(2)} M} \quad . \tag{3.22}$$

To simplify further analysis we set the redefined separation sparseness between first level sparse and dense buckets (cf. eq. (3.13)) to

$$r'_{sep} = (\lg M_1)^{\frac{\lg^{(2)} M_1 - 5}{6}} \tag{3.23}$$

which is still $(\log_\lambda M_1)^{\omega(1)}$ as required by eq. (3.10) (cf. footnote 4 on page 27). This sparseness r'_{sep} is further bounded by

$$\begin{aligned}
r'_{sep} &= (\lg(r_{sep} \cdot \lg M))^{\frac{\lg^{(2)}(r_{sep} \cdot \lg M) - 5}{6}} && \text{using eq. (3.22)} \\
&< (2 \lg r_{sep})^{\frac{\lg(2 \lg r_{sep}) - 5}{6}} && \text{since } r_{sep} > \lg M \text{ by eq. (3.19)} \\
&< (2(\lg^{(2)} M)^2)^{\frac{\lg((\lg^{(2)} M)^2) - 4}{6}} && \text{again using eq. (3.19)} \\
&< ((\lg^{(2)} M)^3)^{\frac{\lg^{(3)} M - 2}{3}} && \text{since } 2 < \lg^{(2)} M \\
&< \frac{1}{3} \cdot (\lg^{(2)} M)^{\lg^{(3)} M - 1} && \text{since } (\lg^{(2)} M)^{-1} < \tfrac{1}{3} \quad .
\end{aligned} \tag{3.24}$$

Next, the first level dense buckets are further split into p_1 (cf. eq. (3.14)) sub-buckets each of range $M_2 = \frac{M_1}{p_1} = r'_{sep} \cdot \lg M_1$. Finally, since M_2 is also the range of buckets in the table of small ranges, the size of the table is

$$M_2 \cdot 2^{M_2} = r'_{sep} \cdot \lg M_1 \cdot M_1^{r'_{sep}}$$

$$\begin{aligned}
&= r'_{sep} \cdot \lg(r_{sep} \lg M) \cdot (r_{sep} \lg M)^{r'_{sep}} && \text{by eq. (3.22)} \\
&< 2 \lg r_{sep} \cdot r'_{sep} \cdot {r_{sep}}^{2r'_{sep}} && \text{since } r_{sep} = (\lg M)^{\omega(1)} \\
&< \lg r_{sep} \cdot {r_{sep}}^{3r'_{sep}-1} && \\
&< \frac{\lg r_{sep}}{r_{sep}} \cdot ((\lg M)^{\lg^{(2)} M})^{(\lg^{(2)} M)^{\lg^{(3)} M - 1}} && \text{by eq. (3.24) and eq. (3.19)} \\
&< \frac{\lg r_{sep}}{r_{sep}} \cdot (\lg M)^{(\lg^{(2)} M)^{\lg^{(3)} M}} && \\
&= o(\frac{M}{r_{sep}} \cdot \lg r_{sep}) = o(N_{sep} \cdot \lg r_{sep}) && \\
&= o(B) &&
\end{aligned}$$

for $r \leq r_{sep}$. This brings us to the final theorem:

Theorem 3.2 *There is an algorithm which solves the static membership problem in* $O(1)$ *time using data structure of size* $B + o(B)$ *bits.*

Proof: The discussion above dealt only with the space bound. However, since the structure is more or less the same as that of § 3.3 the time bound can be drawn from Theorem 3.1. $\mathcal{QED}$

With Theorem 3.2 we proved only that the second term in space complexity is $o(B)$. In fact, using a very rough estimate from the second term of sparse first level buckets we get the bound $O(\frac{B}{\lg^{(3)} M})$. To improve the bound one would have to refine values r_{sep} and r'_{sep} (cf. footnote 4 on page 27).

3.5 Dynamic Version

The solutions presented in § 3.3 and in § 3.4 deal with a fixed set. It is natural to ask how we can incorporate updates while maintaining the time and space bounds. Before presenting a dynamic solution we describe the memory management scheme used.

3.5.1 Memory Management Scheme

We assume we are operating in an environment in which we have a currently held chunk of memory of size H. If more memory is requested the chunk is extended upward and H is increased. If space requirements reduce we release some from the top of the chunk and H is decreased. Under a slightly different model memory may not be extended in this way, in which case we simply request new space of a desired size and release the old one after copying. All results also hold in the latter model with the caveat that both chunks are required while the copying is being performed.

The basic idea is to operate on a chunk of memory of size, H, by simply allocating new storage from the chunk as required and not reusing storage that has been freed. We say the chunk is the space *held*, while the size of the data structure is amount *in use*. The space that *has been used* refers to that space that has been allocated in the chunk (i.e. it includes the space that has been released). If the amount of memory in use falls below some threshold, E, or if all the storage in the chunk has been used, we compress the data to the bottom of the chunk (by mark & sweep cf. [6]) and perhaps modify the chunk size. The period of time between the point at which we compress our data into the bottom of the chunk until the next such point is called a *phase*. The actual values of H and E are determined (dynamically) at the beginning of each phase and depend on S, the amount of memory in use at that time.

The manipulation outlined above requires the additional variables to keep track of how much of the chunk has been used and how much is actually in use. $2 \lg H$ bits clearly suffice. In the following lemma we bound the amortized cost in a phase:

Lemma 3.4 *Let S be the amount of memory in use in a chunk of size H at the beginning of a phase ($S < H$) and let $E = \frac{S^2}{H}$. We then service a sequence of allocate and deallocate requests, with the assumption that after an allocate or deallocate request the supported algorithms take time at least proportional to that needed to copy data of the size of the request before issuing another space request. Then the amortized computing time of the sequence lasting up to one phase is*

$$1 + O(\frac{H}{H-S}) = 1 + O(\frac{S}{S-E}) \ . \tag{3.25}$$

This time includes all storage management operations including initialization of the next phase.

Proof: Initially observe that equality in eq. (3.25) holds, because $E = \frac{S^2}{H}$. It is easy to see that the worst case occurs when the sequence consists solely of allocate or of deallocate requests. Since the proofs for both cases are almost identical we present only the one for allocate requests.

At the beginning of a phase there are $H - S$ bits of contiguous free space in the chunk. This is also the largest amount of space requested by all but the last allocation request. However, to issue requests for this much space takes at least $T = \Omega(H - S)$ computing time. Thus, the total elapsed time in the phase, excluding the computing time of the last request, is $T + O(H)$. The second term comes from copying data from one chunk of memory to another one. Amortizing over the computing time of the whole phase we get eq. (3.25). $\mathcal{QED}$

Note, that the memory management scheme used in Lemma 3.4 is extremely simple and any improvements (cf. [6, Chapter 12]) would also improve an amortized computing time of eq. (3.25). Lemma 3.4 bounds the amortized computing time while the following theorem relates it to the ratio between the amount of occupied and allocated memory:

Theorem 3.3 *Let D be the amount of memory in use at some moment. If $S = E + \kappa(E)$ ($\kappa(E) = O(E)$ and $\kappa(E) = \omega(1)$) then the amortized computing time is*

$$O(1 + \frac{E}{\kappa(E)}) \tag{3.26}$$

and the chunk in current use is of size

$$D + O(\kappa(D)) \ . \tag{3.27}$$

Proof: The amortized time bound follows directly from Lemma 3.4. The amount of used memory is H, and thus

$$\begin{aligned} H &= \frac{S^2}{E} = \frac{E^2 + 2E\kappa(E) + \kappa(E)^2}{E} \\ &= E + (2\kappa(E) + O(\kappa(E)) && \text{since } \kappa(E) = O(E) \\ &= E + O(\kappa(E)) \\ &\leq D + O(\kappa(D)) && \text{since } E \leq D \end{aligned}$$

$\mathcal{QED}$

One can tighten the analysis in the proof of Lemma 3.4 and consequently relax the condition $\kappa(E) = \omega(1)$ to $\kappa(E) = \Omega(1)$ in Theorem 3.3.

3.5.2 Solution

The dynamic solution is based on the static solution from § 3.4. We present three versions: the first retains the static counterpart's space bound $B + o(B)$, but degrades the time bound to a bit more than constant amortized time with a high probability. The second has constant amortized time with a high probability, but the space bound degrades to $2B + o(B)$ bits. In the last version we further improve the result to the worst case constant time with a high probability but use $O(B)$ bits of space. Indeed we use as a starting point for the third version the solution presented in § 3.3 as it is amenable to fine tuning.

All of our solutions use the standard technique of "doubling". The core of this method is to maintain, at any given time, the most appropriate data structure and when the situation changes, to switch to some other, better structure. In other words, at a certain moment we discard the old structure and build a new one, while the build-up time is amortized over preceding operations. Since the size of a new structure is usually half or double the previous one, hence the name "doubling".

However, it is not clear that the doubling technique is directly applicable to the structure from § 3.4. Clearly, there are no problems with a table of small ranges, because it is built only once — the range of small buckets M_2 is independent of N.

On the other hand, the main structure consists of a number of buckets and to use the technique on them, we have to prove the desired time and space bounds for any sequence of operations on each individual bucket. More precisely, each bucket consists of a number of sub-buckets and even if operations on individual sub-buckets do have the desired bounds and the total space occupied by all sub-buckets does not change, their individual sizes might change and we have to guarantee, for any sequence of operations, that sub-buckets can "live" *inside* a bucket within a desired amortized time. Fortunately, Theorem 3.3 gives exactly such guarantees provided sub-buckets do not change their sizes too quickly. This brings us to the first solution:

Theorem 3.4 *Let $\kappa(B) = \Omega(1)$ and $\kappa(B) = o(B)$; then there is an algorithm which solves the dynamic membership problem in $O(\frac{B}{\kappa(B)})$ amortized time per operation with the high probability using a data structure of size $B + o(\kappa(B))$ bits.*

Proof: First observe that all sub-structures used in the static solution of § 3.4 individually support the claimed time and space bounds — even a hash table if we use the solution due to Dietzfelbinger et al. ([38]). Next, since B changes more slowly than N, it takes $\Omega(R)$ computing time after a memory request for R bits and thus Lemma 3.4 applies. The proof now follows directly from Theorem 3.3. $\mathcal{QED}$

The attribute "high probability" appears in the text of Theorem 3.4 because we use the dynamic perfect hash tables of Dietzfelbinger et al. If we set $\kappa(B)$ to B, the second version of our dynamic solution follows by a similar argument to that of Theorem 3.4:

Theorem 3.5 *There is an algorithm which solves the dynamic membership problem in constant amortized time per operation with high probability using a data structure of size $2B + o(B)$ bits.*

These two solutions presented have an amortized constant time behaviour, but we want to achieve a worst case constant time bound. Indeed, observe we do not need to build a new structure from scratch when a certain threshold is reached, but we can build it *smoothly* through a sufficient number of preceding operations. To put it differently, we always maintain a pair of structures, where one contains "real" data, while the other one is in the process of building-up. In this way we do not have only amortized, but also worst case behaviour provided that all sub-structures support this process.

As mentioned, this final solution is based on a static solution from § 3.3 which employs as sub-structures those listed in Table 3.1. All of them, indeed, support smooth doubling including hash tables (cf. dynamic perfect hashing [37, 39]). This brings us to the last theorem:

Theorem 3.6 *There is an algorithm which solves the dynamic membership problem in $O(1)$ time per operation with high probability using a data structure of size $O(B)$ bits.*

The constant in the space bound is mostly contributed by the dynamic hash tables and is less than 10 (cf. [37, 39]).

3.6 Two "Natural" Examples

There are many situations in which we are dealing with a subset of a bounded universe in which the size of the subset is relatively large but not big enough to justify a bit map representation. We give two such examples. The first is the set of primes less than some number M, hence N is of size approximately $\frac{M}{\ln M}$. We pretend that the set of primes is random and that we are to store them in a structure to support the query of whether given number is prime. Clearly, we could use some kind of compression (e.g. implicitly omit the even numbers or sieve more carefully), but for the purpose of this example we will not do so.

In the second example we consider Canadian Social Insurance Numbers (S.I.N.'s), allocated to each individual. Canada has approximately 28 million people and each person has a 9 digit Social Insurance Number. One may want to determine whether or not a given number is allocated. This query is in fact a membership query in the universe of size $M = 10^9$ with a subset of size $N = 28 \cdot 10^6$, since we ignore the check digit for a valid S.I.N.

Both examples deal with sparse sets and we can use the method of § 3.3.1 directly with using in buckets a perfect hashing function described in [48]. On the other hand, no special features of data are used which makes our space calculations slightly pessimistic.

Using an argument similar to that of Lemma 3.2, we observe that the worst case distribution occurs when all buckets are equally sparse, and therefore, we can assume that in each bucket there are $\frac{N}{p}$ elements.

Table 3.2 contains the sizes of data structures for both examples comparing a hash function, a bit map, and a tuned version of our structure (computed from eq. (3.10)) with the information-theoretic bound.

Example	M	N	B	ours	hash	bit map
primes	$1.0 \cdot 2^{32}$	$1.4 \cdot 2^{27}$	$1.6 \cdot 2^{29}$	$1.9 \cdot 2^{30}$	$1.4 \cdot 2^{32}$	$1.0 \cdot 2^{32}$
SINs	$1.9 \cdot 2^{29}$	$1.7 \cdot 2^{24}$	$1.1 \cdot 2^{27}$	$1.2 \cdot 2^{28}$	$1.6 \cdot 2^{29}$	$1.9 \cdot 2^{29}$

Table 3.2: *Space usage for sets of primes and SINs for various data structures.*

3.7 Discussion and Conclusions

In this chapter we have presented a solution to a static membership problem. Our initial solution answers queries in constant time and uses space within a small constant factor

of the minimum required by the information-theoretic lower bound. Subsequently, we improved the solution reducing the required amount of space to the information-theoretic lower bound plus a lower order term. We also addressed the dynamic problem and proposed a solution based on a standard doubling technique.

Data structures used in solutions consist of three major sub-structures which are used in different ranges depending on the relative sparseness of the set at hand, that is, depending on the ratio between the size of the set and the universe. When the set is relatively sparse we use a perfect hashing; when the set is relatively dense we use a bit map; and in the range between we use recursive splitting (indexing). The depth of the recursion is bounded by the use of *word-size truncation* and in our case it is 2.

The practicality of the data structure was addressed through a couple of examples. However, to make the structure more practical one would need to tune the parameters c, ϵ, λ, and α mentioned Table 3.1 and Algorithm 3.1. Moreover, for the practical purposes it is probably necessary to increase the depth of recursive splitting to cancel out the effect of a constant hidden in the order notation and, in particular, to decrease the size of the table of small ranges below the information-theoretic minimum defined by N and M at hand. For example, in the case of currently common 64 and 32 bit architectures (cf. [56]) the depths should be increased to 4 and 5 respectively.

There are many open problems. The most intriguing one is to decrease the second order term in the space complexity as there is still a substantial gap between our result, $B + O(\frac{B}{\lg^{(3)} M})$, and the information-theoretic minimum, B. But do we need a more powerful machine model to close this gap?

Chapter 4

Word-Size Parallelism

So we grew together,
Like to a double cherry,
seeming parted;
But yet a union in partition,
Two lovely berries moulded
on one stem:
So, with two seeming bodies,
but one heart;

William Shakespeare, A Midsummer Night's Dream

In this chapter we present a programming technique called *word-size parallelism* and use it to solve a number of examples (problems) whose solutions will prove useful later in a more general context. Word-size parallelism is a programming technique based on splitting of the register into a number of blocks on which parallel, non-interfering operations are performed. For example, a bitwise Boolean negation can be considered as a number of *parallel* single bit negations (cf. also SMID and MIMD architectures).

The chapter is organized as follows: after a brief introduction we define a linear register in § 4.2 and then use it in § 4.3 to present word-size parallelism. We continue by introducing rectangular and multidimensional registers in § 4.4 and § 4.5 respectively, and conclude by developing algorithms for search of extremal set bits in such registers.

4.1 Introduction

Though one can find individual examples of word-size parallelism in the literature (cf. [10, 11, 53]) the technique itself has never been formally exploited. This kind of parallelism is found in standard processor instruction sets where it is perfectly justifiable to consider any bitwise Boolean operation as a parallel operation performed on the individual bits of the register. Such parallelism has, with the latest 32 and 64 bit generation of RISC processors, even greater practical significance (cf. [56]).

The model we use in this chapter is the ERAM of Definition 2.4. As mentioned, in this model the memory register and communication channel are m bits wide, while arithmetic registers are $b = \Theta(m)$ bits. Further, the the active block of the model can perform its operations in unit-time (cf. uniform cost criterion in [5, p.12]). These operations are those found in standard processor instruction sets including integer multiplication and division, and bitwise Boolean operations (cf. MBRAM from § 2.3.1).

4.2 Linear Registers

In this chapter registers do not denote locations in memory, but entities on which operations are performed (arithmetic registers or accumulators). They are $b = \Theta(m)$ bits wide (cf. multiple precision operations),[1] where the constant is essentially the dimension of a register and thus small (cf. § 5).

[1] If the actual processor can not perform multiple precision operations, we assume that they are done in software at the cost of a constant factor delay.

Definition 4.1 *A **linear register** x_l consists of b bits enumerated from 0 to $b-1$ where x_l.b[0] is the least significant bit. The integer value stored in a register is*

$$\sum_{i=0}^{b-1} x_l.\mathtt{b[}i\mathtt{]} \cdot 2^i \ . \tag{4.1}$$

Obviously the value of a register with all bits set is

$$P = 2^b - 1 \ . \tag{4.2}$$

In the course of work we will represent linear registers graphically and thus we need to define their orientation. Usually pictures show the least significant bit of the register at its right, but for our purposes this proves to be inappropriate, since in § 5 we relate positions of a bit in a register to the position of a point in a coordinate system. Therefore we decided to *reverse* the orientation of the register and put the least significant bit `b[0]` at its left end, and the most significant bit $\mathtt{b[}b-1\mathtt{]}$ at its right end. Because of this decision also operations `ShiftLeft` and `ShiftRight` exchange their usual meaning and now become division and multiplication by a power of 2 respectively. Thus

$$x.\mathtt{b[}i\mathtt{]} \equiv (x \text{ div } 2^i) \wedge 1 = \mathtt{ShiftLeft(}x\mathtt{,}\ i\mathtt{)\ AND\ 1} \tag{4.3}$$

extracts the value of the ith bit from x. Using a similar approach we mask out all less significant (left) bits than $x.\mathtt{b[}i\mathtt{]}$ (i.e. all bits $x.\mathtt{b[}j\mathtt{]}$, where $j < i$) by

$$x\ \mathtt{AND\ ShiftRight(}P\mathtt{,}\ i\mathtt{)} \tag{4.4}$$

and eliminate all more significant (right) bits by

$$x\ \mathtt{AND\ ShiftLeft\ (}P\mathtt{,}\ b\mathtt{-1-}i\mathtt{)}\ . \tag{4.5}$$

We conclude this section defining two special linear registers:

Definition 4.2 *Let $0 < k < s$. Then a register x is (s,k)-**sparse** if all its set bits are among $x.\mathtt{b[}a + si + j\mathtt{]}$ (for some a), where $0 \leq j < k$ and $0 \leq i < \lceil \frac{b-k-a}{s} \rceil$.*

In simpler terms, an (s,k)-sparse register has all bits set to 0 with a possible exception of those bits which are in blocks of size k where starting bits of these blocks appear in an arithmetic progression with a difference s. The first block starts at a bit $\mathtt{b[}a\mathtt{]}$.

Definition 4.3 *A register x is s-**small** if all its set bits are among the least significant s bits $x.\mathtt{b[}i\mathtt{]}$ where $0 \leq i < s$.*

Note, that for a given b an s-small register is also a (b,s)-sparse register.

4.3 The Technique

We present *word-size parallelism* first through a generic form of a function in Algorithm 4.1 and later accompany it with a few examples. The generic form, also illustrated in Figure 4.1, consists of two steps: in the first step operations $\mathtt{F}_i$ are applied in parallel on parameters x[i] ($i = 1, \ldots l$) and in the second step the results of the first step, stored in vector `answer`, are combined by the function `C` into final result y. A parameter x[i] may contain $k \leq b$ bits and can be considered either a k-bit register or a b-bit register. However, in the latter case the remaining $b - k$ bits are irrelevant for our purposes. For example, let x[i] be a 1-bit register containing only a bit x[i]`.b[7]` (the value of x is x[i]`.b[7]` $\cdot 2^7$) then in its b-bit representation of other bits can have values either 1 or 0. Further:

```
GENERIC PROCEDURE Technique (x);
  FOR i:=1 TO l DO PARALLEL
    answer[i]:= Apply (F_i, x[i]);
  END;
  y:= C (answer);
  RETURN y;
END Technique;
```

Algorithm 4.1: *Generic form of word-size parallelism.*

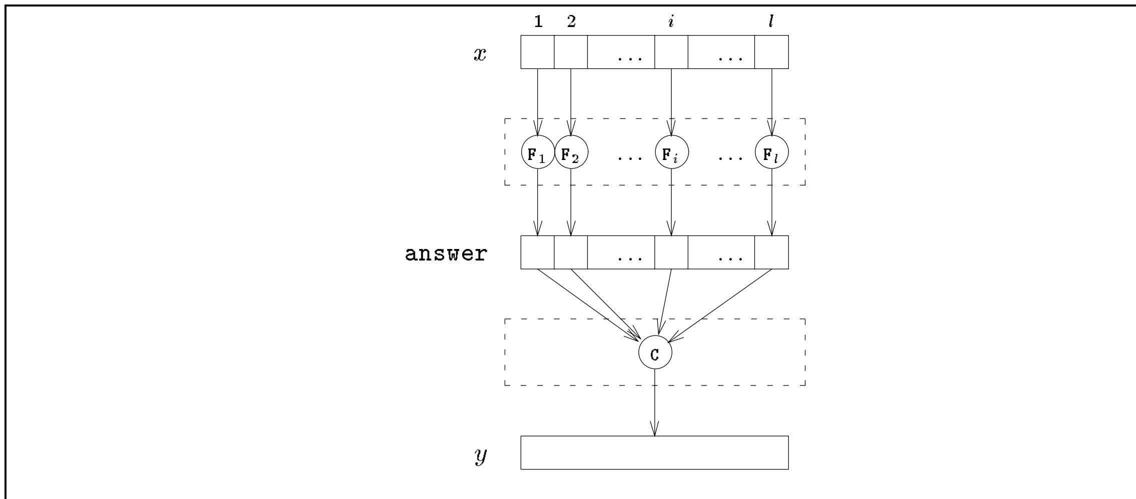

Figure 4.1: *Graphic illustration of word-size parallelism.*

Definition 4.4 *A register* `u` ***is in conflict*** *with a register* `v` *if there is a bit* `u.b[`k`]` $\neq$ `v.b[`k`]`.

If u is fewer than b bits wide, then it can conflict with v only on bits it contains, because, as mentioned above, values of other bits in a b-bit representation of u are irrelevant – they can be either 1 or 0 and thus always "match" bits of v.

The main idea of the technique is to replace the parallel loop of Algorithm 4.1 with a few (sequential) operations of our model's active block (processor). In other words, we substitute for a constant number of sequential operations l parallel operations and still have the same run time as the parallel version.

First, the combination function C can be an identity and hence omitted. Next, the operation $\mathtt{F}_i$ can take more than one parameter x[i]. Finally, individual operations need not be identical, though in this work we assume that they are (e.g. all $\mathtt{F}_i$ are subtractions, or all are multiplications, etc.). The last constraint simplifies the derivation of a proper sequential replacement:

Lemma 4.1 *If in Algorithm 4.1*

(i.) parameters are not in a conflict: x[i] does not conflict with any x[j] and for multiple parameters let this hold for each parameter separately,

(ii.) intermediate results are not in a conflict: `answer[`i`]` *does not conflict with any* `answer[`j`]`*, and*

(iii.) all $\mathtt{F}_i$ are identical: all $\mathtt{F}_i$ are F*,*

then the value of procedure `Technique (`x`)` *is*

$$\mathtt{C}\ (\ \mathtt{F}\ (\sum_{i=1}^{l} x[i]))\ . \qquad (4.6)$$

When operations $\mathtt{F}_i$ have multiple parameters, then F in eq. (4.6) has also multiple parameters each of which is a sum of corresponding parameters of $\mathtt{F}_i$.

Proof: First, since parameters x[i] are not in conflict, they can be added together. Next, since all $\mathtt{F}_i$ are identical they can be replaced by F and, consequently, F can be applied to the added parameters x. Finally, since results of $\mathtt{F}_i$ are not in conflict, neither are the results of F applied to the added parameters. $\mathcal{QED}$

In the rest of this section we apply the technique in a number of very simple practical examples. These examples are subsequently used in solutions of more involved problems.

EXAMPLE 4.1: Let x be a k-small register. We want to generate an (s,k)-sparse register y in which the individual blocks are copies of the k least significant bits in x. That is, given $s \cdot t = b$, we want $x.\mathrm{b}[i] = y.\mathrm{b}[i+sj]$ for $0 \le j < l$ and $0 \le i < k$. In the copying process, illustrated in Figure 4.2 and implemented in Algorithm 4.2, the final

combination step C is omitted because it is an identity. Further, the parameter x is the same for all i and it does not conflict with itself. Also parameters 2^{is} are not in conflict, nor are results of individual $\mathtt{F}_i$. Finally, since all $\mathtt{F}_i$ are identical (they are multiplications) by Lemma 4.1 the expression $x \cdot \sum_{j=0}^{l-1} 2^{js} = x \cdot \frac{2^{ls}-1}{2^s-1}$ or its equivalent

$$x \text{ * (ShiftRight(1, } l\text{*}s\text{) - 1) / (ShiftRight(1, } s\text{) - 1)} \tag{4.7}$$

replaces Algorithm 4.2. $\mathcal{EF}$

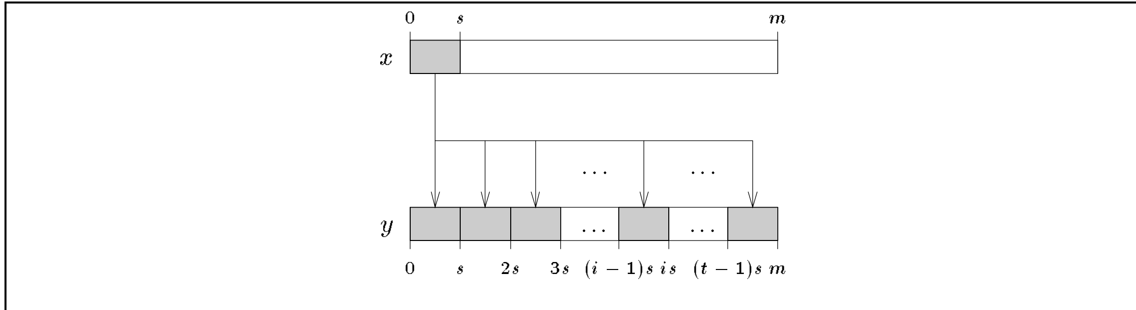

Figure 4.2: *Distribution of a block across a linear register.*

```
PROCEDURE Distribute1D (s, l, x);
  FOR i:=0 TO l-1 DO PARALLEL
    y:= x * 2^is
  END;
  RETURN y;
END Distribute1D;
```

Algorithm 4.2: *Distribution of a block of at most s bits l times across the register.*

Above we distributed bits across the register, while now we spread them.

EXAMPLE 4.2: Let x be an s-small register consisting of l blocks of k bits, i.e. $s = k \cdot l$. Spread these blocks across register y to get an $(s-k, k)$-sparse register, where the order of spread blocks is a reverse of the original one as shown in Figure 4.3. More precisely, since $s = k \cdot l$ we have $x.\mathtt{b}[(l-i-1) \cdot k + j] = y.\mathtt{b}[i \cdot s + j]$ where $0 \leq i < l$ and $0 \leq j < k$. Note that, since y is b bits wide, $s \cdot (l-1) + k \leq b$ or equivalently $s < \left\lfloor \frac{b-k}{l-1} \right\rfloor$. In fact, for $l = s$, that is when each block contains a single bit, this becomes $s < \left\lfloor \sqrt{b - 0.75} - 0.5 \right\rfloor$.

The parallel loop of Algorithm 4.3 is this time slightly more complicated: first we copy all s bits (cf. Algorithm 4.2) and then mask out the unwanted bits in each copy. Algorithm 4.3 also contains the final step, C, which shifts y for $s-k$ bits to the left.

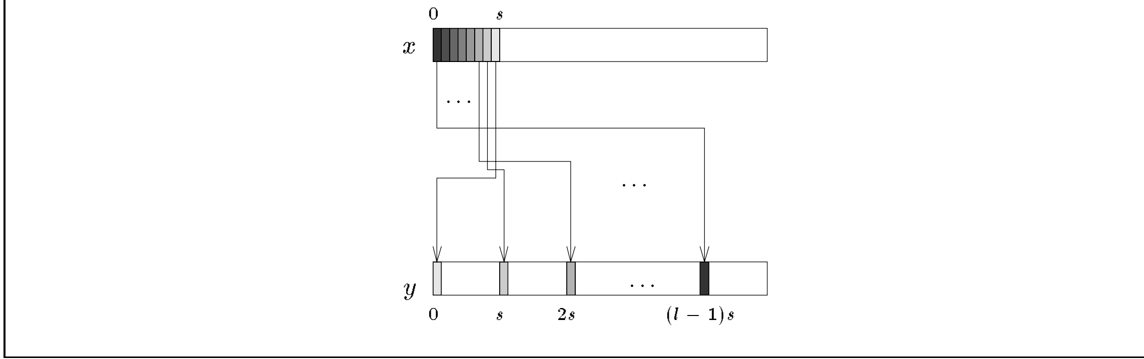

Figure 4.3: *Spreading of bits across a linear register.*

```
PROCEDURE Spread1D (x, s, k);
  l:= s / k;
  FOR i:=0 TO l-1 DO PARALLEL
    offset:= 2^(is);
    y:= x * offset;
    y:= y AND (2^((l-i-1)k) * (2^k-1) * offset);
  END;
  RETURN ShiftLeft (y, s-k);
END Spread1D;
```

Algorithm 4.3: *Spreading of a block of $k \leq s$ bits across the register in a reverse order creating an $(s-k,k)$-sparse register.*

Finally, since none of the parameters or the intermediate results are in conflict, and since all $\mathtt{F}_i$ are identical, we replace the parallel loop of Algorithm 4.3 by the expression

$$\left(x \cdot \sum_{i=0}^{l-1} 2^{is}\right) \wedge \left((2^k-1) \cdot 2^{(l-1)k} \cdot \sum_{i=0}^{l-1} 2^{(s-k)i}\right) = (x \cdot \frac{2^{ls}-1}{2^s-1}) \wedge ((2^k-1) \cdot \frac{2^{ls}-2^s}{2^s-2^k}) \quad , \quad (4.8)$$

which is under our model computable in a constant time. $\mathcal{EF}$

The last example in this section is the most elaborate and gives the feeling of the full power of the technique. It is, in a way, an inverse of Example 4.2 where we produced an $(s-k,k)$-sparse register from an s-small register by spreading its k-bit blocks. This time we take an (s,k)-sparse register and compress its k-bit blocks together into an s-small register, but without reversing their relative order.

EXAMPLE 4.3: Let x be an (s,k)-sparse register from which we produce an s-small register y by "squeezing" out all zero bits. That is, let $t \cdot k \leq s$ and, because x is b bits wide, $t \cdot s + k \leq b$, then $y.\mathtt{b}[i + j \cdot k] = x.\mathtt{b}[i + j \cdot s]$ for $0 \leq i < k$ and $0 \leq j < t$.

The compression is performed in two steps, shown in Figure 4.4, which coincide with two steps of our technique. First, all blocks are copied to contiguous positions by a parallel

loop of Algorithm 4.4. Then, the copied blocks are shifted left to the least significant position while unwanted bits are masked out.

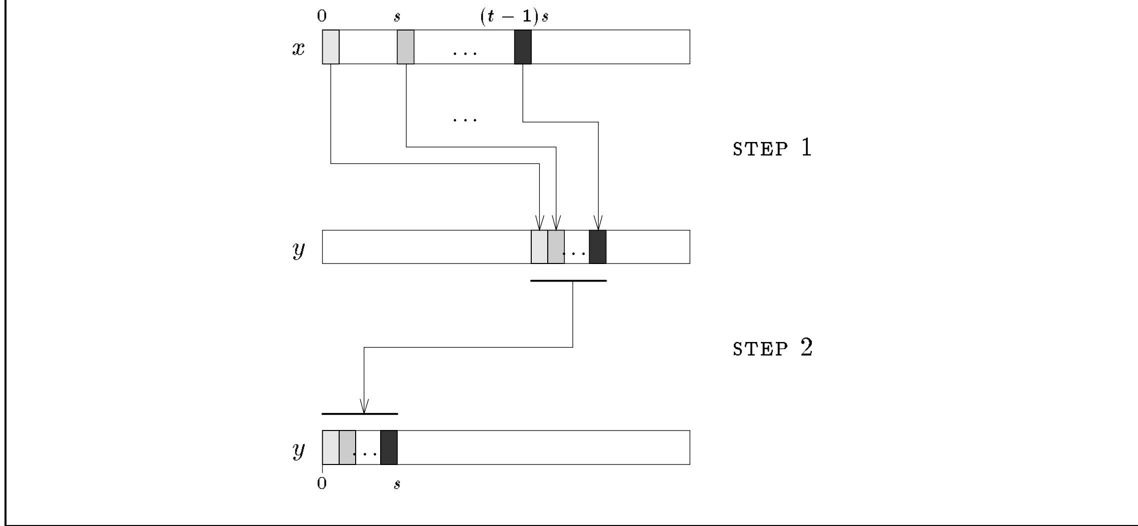

Figure 4.4: *Two step compression of an (s, k)-sparse register.*

```
PROCEDURE Compress (x, s, k, t);
  FOR l:= 0 TO t-1 DO PARALLEL
    y:= x * 2^((t-l-1) · s + (l+1) · k);
  END;
  RETURN ShiftLeft (y, (t-1)*s + k) AND (2^(tk)-1);
END Compress;
```

Algorithm 4.4: *Compression of an (s, k)-sparse register.*

To replace the parallel loop using Lemma 4.1, we have to satisfy the lemma's three conditions. Obviously, all $\mathtt{F}_i$ are identical and parameters are not in conflict, while we prove the same for intermediate results by a contradiction.

First, by Lemma 4.1, we rewrite the parallel loop into

$$
\begin{aligned}
y &= x \cdot \sum_{l=0}^{t-1} 2^{(t-l-1)\cdot s+(l+1)\cdot k} && (4.9)\\
&= \sum_{i=0}^{t-1}\sum_{j=0}^{k-1}\sum_{l=0}^{t-1} x.\mathtt{b}[i \cdot s + j] \cdot 2^{i\cdot s+j+(t-l-1)\cdot s+(l+1)\cdot k} && \text{by Definition 4.2}\\
&= 2^{(t-1)\cdot s+k} \cdot \sum_{i=0}^{t-1}\sum_{j=0}^{k-1}\sum_{l=0}^{t-1} x.\mathtt{b}[i \cdot s + j] \cdot 2^{(i-l)\cdot s+l\cdot k+j} \;.
\end{aligned}
$$

Next, assume that two intermediate results are in conflict at some bit, which means for two exponents $(i_1-l_1)\cdot s+l_1\cdot k+j_1=(i_2-l_2)\cdot s+l_2\cdot k+j_2$ the values of bits $x.\mathtt{b}[i_1\cdot s+j_1]$ and $x.\mathtt{b}[i_2\cdot s+j_2]$ are different. Since $0\leq j_1,j_2<k$, this is true iff $j_1=j_2$ which gives us $(i_1-l_1)\cdot s+l_1\cdot k=(i_2-l_2)\cdot s+l_2\cdot k$. Using the same reasoning again, though this time based on an assumption $t\cdot k\leq s$, we get $l_1=l_2$, and, finally, $i_1=i_2$. This makes bits $x.\mathtt{b}[i_1\cdot s+j_1]$ and $x.\mathtt{b}[i_2\cdot s+j_2]$ identical and thus they can not have different values. Therefore, expression (4.9) replaces the parallel loop of Algorithm 4.4. Further, it can be rewritten into $x\cdot\frac{2^{ts}-2^{tk}}{2^{s-k}-1}$ which is computable in a constant time under our model. $\mathcal{EF}$

If in Example 4.3 we set $k=1$ we get Fredman and Willard's Lemma 3 in [53].[2] Their lemma also inspired the name for our (s,k)-sparse registers: in the lemma they define a family of d-sparse registers which corresponds to our $(d,1)$-sparse registers. Example 4.3 can be also used to develop the field packing algorithm in [11]. Finally, combining Algorithm 4.3 and Algorithm 4.4 we get a constant time algorithm which reverses the relative order of k-bit blocks in a s-small register, where $s=l\cdot k\leq\left\lfloor\frac{b-k}{l-1}\right\rfloor$ (cf. [23]).

4.4 Rectangular Registers

In Chapter 5 we search for the closest neighbour in two dimensions and represent the universe by a bit map stored in a rectangular register. In this section we define these registers and their relation to linear registers. We also present some special instances of rectangular registers and how they are generated using word-size parallelism.

Definition 4.5 *A rectangular register x_r consists of r rows and c columns of bits, where $r\cdot c=b$. The bit $x_r.b[i,j]$, for $0\leq i<c$ and $0\leq j<r$, is positioned in the ith column of jth row, and the bit $x_r.b[0,0]$ is the least significant bit of x_r. The integer value stored in a rectangular register is*

$$\sum_{i=0}^{c-1}\sum_{j=0}^{r-1}x_r.b[i,j]2^{j\cdot c+i}\quad, \tag{4.10}$$

i.e. concatenate all rows and read as binary number.

In the notation used for rectangular registers, and later for multidimensional ones, we could adopt either *matrix* or *geometric* convention. For example, under matrix convention bit $\mathtt{b}[i,j]$ is on the ith row and the jth column, but under a geometric one on the ith column and the jth row. We chose to use a geometric convention which will prove more convenient in § 5.[3]

A special rectangular register is a *square register*:

[2] However, their proof did not use word-size parallelism.

[3] Note, indices are increasing from left to right and bottom to top.

Definition 4.6 *A* ***square register*** *is a rectangular register with the same number of rows and columns*

$$r = c = p = \sqrt{b} \ . \tag{4.11}$$

Rectangular and linear registers are only a helpful logical abstraction – a special interpretation of a number. Assuming the value of a number remains unchanged when we switch its interpretation from a linear x_l to a rectangular x_r, we get, from eq. (4.10) and eq. (4.1), the bijective mapping between bits of two interpretations

$$\begin{aligned} x_r.\mathtt{b}[i,j] &= x_l.\mathtt{b}[i + j \cdot c] \\ x_l.\mathtt{b}[k] &= x_r.\mathtt{b}[k \bmod c, k \operatorname{div} c] \ . \end{aligned} \tag{4.12}$$

Subsequently, we have a bijective mapping between two different rectangular registers x_1 ($r \times c$) and x_2 ($s \times d$)

$$x_1.\mathtt{b}[i,j] = x_2.\mathtt{b}[(i + j \cdot c) \bmod d, (i + j \cdot c) \operatorname{div} d] \ . \tag{4.13}$$

In our work we require masks to zero (by `AND` operations) bits in specific portions of other registers. Such rectangular registers are generalizations of s-small registers from Definition 4.3 and are shown in Figure 4.5.

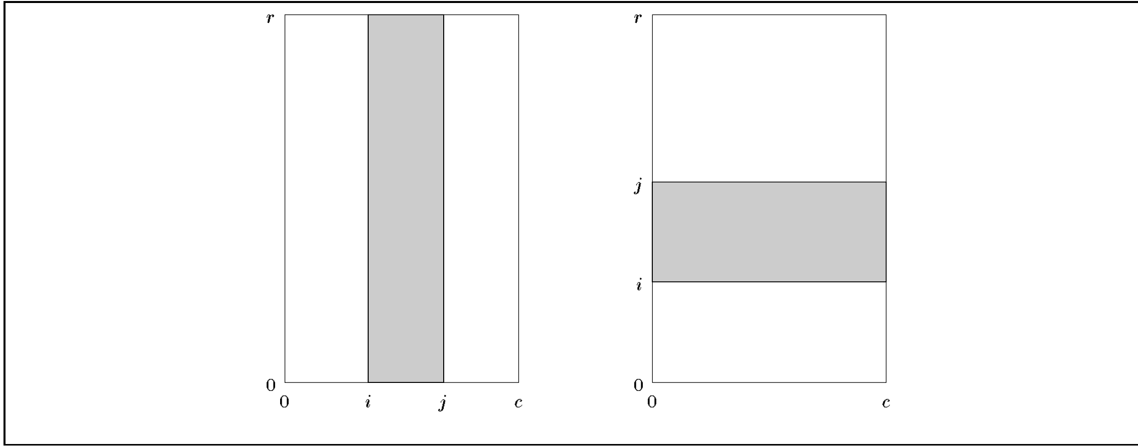

Figure 4.5: *Column- and row-stripe registers* $S^C_{i,j}$ *and* $S^R_{i,j}$.

Definition 4.7 *A* ***column-stripe register*** $S^C_{i,j}$ *has set bits only from column i to j*

$$S^C_{i,j}.b[k,l] = \begin{cases} 1 & \text{if } 0 \leq i \leq k < j \leq c \text{ and } 0 \leq l < r \\ 0 & \text{otherwise} \end{cases} \tag{4.14}$$

and the ***row-stripe register*** $S^R_{i,j}$ *only between rows i and j*

$$S^R_{i,j}.b[k,l] = \begin{cases} 1 & \text{if } 0 \leq k < c \text{ and } 0 \leq i \leq l < j \leq r \\ 0 & \text{otherwise} \ . \end{cases} \tag{4.15}$$

Using mapping in eq. (4.12) we observe that the column-stripe register $S^C_{i,j}$ is also $(c, j-i)$-sparse register from Definition 4.2 (the offset $a = i$).

Two special stripe registers are

$$S^R_{0,1} = \sum_{i=0}^{c-1} 2^i = 2^c - 1 = \texttt{ShiftRight(1, } c\texttt{) - 1} \tag{4.16}$$

with bits set only in the first row and

$$S^C_{0,1} = \sum_{i=0}^{r-1} 2^{ci} = \frac{2^{rc}-1}{2^c-1} = \frac{2^b-1}{2^c-1} = \frac{P}{S^R_{0,1}} \tag{4.17}$$

with bits set only in the first column.[4] These registers, both of which can be constructed in constant time, are used to generate all other stripe registers in constant time using word-size parallelism.

Example 4.4: Algorithm 4.5 generates $S^C_{i,j}$ from $S^C_{0,1}$ by copying $S^C_{0,1}$ to all columns from i to $j-1$ in parallel. Applying Lemma 4.1, Algorithm 4.5 is equivalent to the expression

$$S^C_{i,j} = S^C_{0,1} \cdot \sum_{l=i}^{j-1} 2^l = S^C_{0,1} \cdot (2^j - 2^i) \ . \tag{4.18}$$

```
PROCEDURE GenerateColumnStripe (i, j);
  FOR l:= i TO j − 1 DO PARALLEL
    y:= S^C_{0,1} * 2^l
  END;
  RETURN y;
END GenerateColumnStripe;
```

Algorithm 4.5: *Generation of $S^C_{i,j}$ from $S^C_{0,1}$.*

Similarly, Algorithm 4.6 generates $S^R_{i,j}$ by parallel copying of $S^R_{0,1}$. Again using Lemma 4.1, we replace Algorithm 4.6 by

$$S^R_{i,j} = S^R_{0,1} \cdot \sum_{l=i}^{j-1} 2^{cl} = S^R_{0,1} \cdot \frac{2^{cj} - 2^{ci}}{2^c - 1} = 2^{cj} - 2^{ci} \tag{4.19}$$

since, by eq. (4.16), $S^R_{0,1} = 2^c - 1$. $\mathcal{EF}$

The next registers we consider are generalization of $(s,1)$-sparse register:

[4] From eq. (4.17) we also get $P = S^R_{0,1} \cdot S^C_{0,1}$, where P is defined in eq. (4.2).

```
PROCEDURE GenerateRowStripe (i, j);
  FOR l:= i TO j - 1 DO PARALLEL
    y:= S^R_{0,1} * 2^{cl}
  END;
  RETURN y;
END GenerateRowStripe;
```

Algorithm 4.6: *Generation of* $S^R_{i,j}$ *from* $S^R_{0,1}$.

Definition 4.8 *A rectangular register* x *is an* ***s*-column (*s*-row) sparse** *if all its set bits are among* `b[a+si, j]` (`b[i, a+sj]`)*, where* $0 \leq i < \lceil \frac{c-1-a}{s} \rceil$ *and* $0 \leq j < r$ *(*$0 \leq i < c$ *and* $0 \leq j < \lceil \frac{r-1-a}{s} \rceil$*).*

In simpler terms, an s-column sparse rectangular register has set bits only in columns that are s bits apart. A similar characterization holds for row sparse registers.[5]

A number which is $S^C_{0,h}$ under an $r \times c$ rectangular interpretation ($c = h \cdot s$ for some integer s) becomes the s-row sparse register x_r under the $(s \cdot r) \times h$ interpretation. This follows by using eq. (4.13) and eq. (4.14), as for the non-zero bits we have the mapping $S^C_{0,h}$.b$[i, j] = x_r$.b$[i, j \cdot s]$.

Next consider a number which is $S^R_{0,h}$ under an $r \times c$ rectangular interpretation. Under the linear interpretation it is a $(c \cdot h)$-small register. Now we apply Algorithm 4.3, with block size $k = c$, and convert the result back to an $r \times c$ interpretation. What we end up with is an $(s-1)$-row sparse register. In other words, application of Algorithm 4.3 on an $r \times c$ row-stripe register $S^R_{0,h}$, with $k = c$ and $s = c \cdot h$, produces an $(s-1)$-row sparse register with reversed relative order of rows. This observation extends to the reversal of blocks of rows.

Later we will need to generate an $(s-1)$-column spread register from $S^C_{0,s}$ by spreading its columns in reverse order (cf. Example 4.2 for linear registers).

EXAMPLE 4.5: We spread only individual columns (cf. Figure 4.6) though we could spread blocks of columns in the same way as we spread blocks of bits in a linear register. Let x be a column-stripe register $S^C_{0,s}$ with $s \leq \sqrt{c-1}$. We spread its left s columns across the register in a reverse order producing an $(s-1)$-row sparse register y.[6] Formally, x.b$[s-i-1, j] = y$.b$[j \cdot (s-1), i]$ for $0 \leq i < s$. The implementation in Algorithm 4.7 is similar to that of Algorithm 4.3, though this time we work with columns. Again, the three conditions of Lemma 4.1 are satisfied and thus we can replace the parallel loop in

[5] In a notation for sparse, small, and stripe registers (linear and rectangular) we leave out the offset a appearing in definitions, since it is possible to derive its value from the context – in most cases it is 0.

[6] Obviously, not all bits in first s columns of x need be set.

Algorithm 4.7 by the expression

$$\left(x \cdot \sum_{i=0}^{s-1} 2^{is}\right) \wedge \left(S^C_{0,1} \cdot \sum_{i=0}^{s-1} 2^{s-i-1+is}\right) = \left(x \cdot \sum_{i=0}^{s-1} 2^{is}\right) \wedge \left(S^C_{0,1} \cdot 2^{s-1} \cdot \sum_{i=0}^{s-1} 2^{(s-1)\cdot i}\right) ,$$

which simplifies into

$$\left(x \cdot \frac{2^{s^2}-1}{2^s-1}\right) \wedge \left(S^C_{0,1} \cdot \frac{2^{s^2-1}-2^{s-1}}{2^{s-1}-1}\right) . \tag{4.20}$$

$\mathcal{EF}$

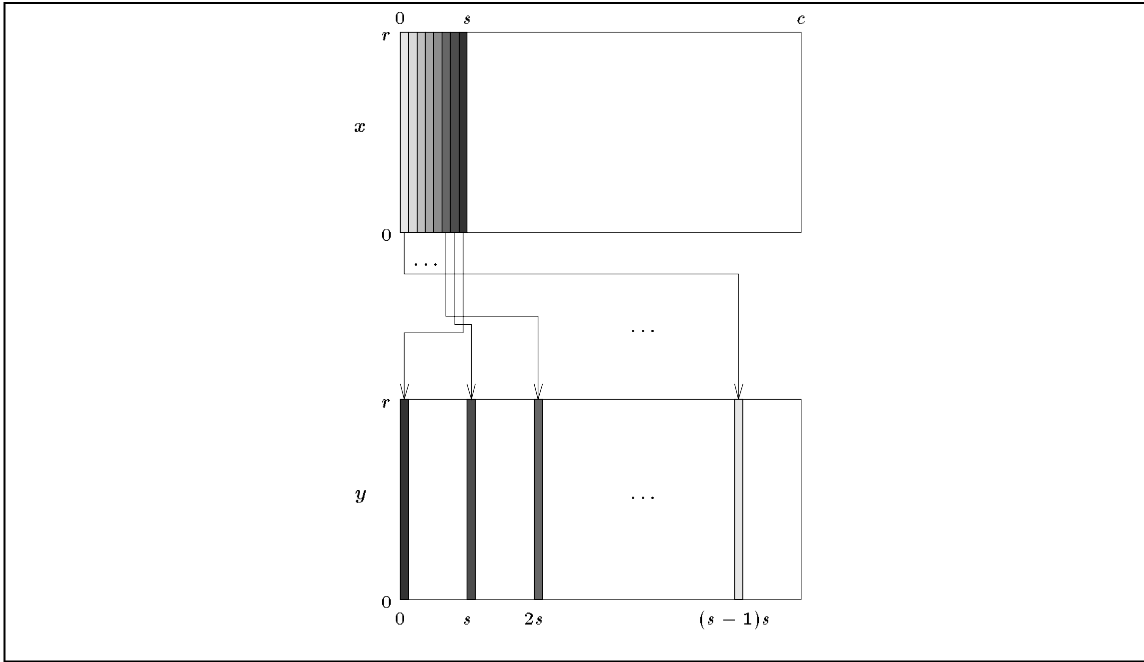

Figure 4.6: *Spreading of a stripe across a rectangular register.*

The last group of rectangular registers we consider are $P^{\diagup}_{c,\delta}$ and $P^{\diagdown}_{c,\delta}$, depicted in Figure 4.7. The registers are defined by two groups of parameters: the first specifies the direction of the slope, and the second, its size. More precisely, the second group defines r and c, the number of rows and columns in the register, and the height of a single block h. Assuming $r \equiv 0 \pmod h$ and since, by Definition 4.5, $b = r \cdot c$, we need only two of three values, in particular c and the slope

$$\delta = \frac{c}{h} . \tag{4.21}$$

Formally the registers are defined as

$$P^{\diagup}_{c,\delta}.\mathtt{b}[i,j] = \begin{cases} 1 & \text{if } i \geq \lfloor \delta \cdot (j \bmod h + 1) \rfloor - 1 = \left\lfloor \delta \cdot (j \bmod \frac{c}{\delta} + 1) \right\rfloor - 1 \\ 0 & \text{otherwise} \end{cases} \tag{4.22}$$

```
PROCEDURE Spread2D (x, s);
  FOR i:=0 TO s-1 DO PARALLEL
    offset:= 2^(is);
    y:= x * offset;
    y:= y AND (S^C_{0,1} * 2^(s-i-1) * offset);
  END;
  RETURN ShiftLeft (y, s-k);
END Spread2D;
```

Algorithm 4.7: *Spreading of columns across the register in a reverse order creating an s-column sparse register.*

and

$$P^{\diagdown}_{c,\delta}.\mathtt{b}[i,j] = \begin{cases} 1 & \text{if } i \geq c - \lfloor \delta \cdot (j \bmod h + 1) \rfloor - 1 = c - \lfloor \delta \cdot (j \bmod \frac{c}{\delta} + 1) \rfloor - 1 \\ 0 & \text{otherwise} \end{cases} \tag{4.23}$$

where in both equations $0 \leq i < c$ and $0 \leq j < r$. Note, $P = P^{\diagdown}_{b,b} = P^{\diagup}_{b,b}$, where P is defined in eq. (4.2). In addition to P we need two more special instances of $P^{\diagdown}_{c,\delta}$ and $P^{\diagup}_{c,\delta}$: in a square register let $\delta = 1$, then, by eq. (4.11) and eq. (4.21),

$$P^{\diagdown} = P^{\diagdown}_{p,1} \qquad \text{and} \qquad P^{\diagup} = P^{\diagup}_{p,1} \ . \tag{4.24}$$

The border between set and unset bits in these registers is a diagonal running from the top left corner, in $P^{\diagdown}$, and top right corner, in $P^{\diagup}$.

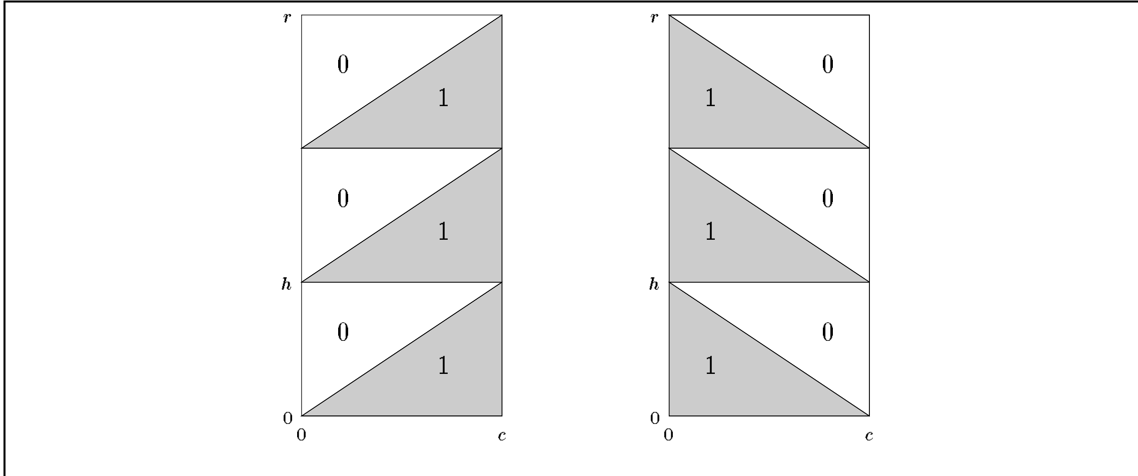

Figure 4.7: *Graphical representation of rectangular registers $P^{\diagup}_{c,\delta}$ and $P^{\diagdown}_{c,\delta}$ for $\delta = \frac{3c}{r} = \frac{3c^2}{b}$.*

Because of the bijective mapping between linear and rectangular interpretations of numbers, all operations on linear registers are directly applicable to the rectangular ones.

The most interesting are shifts, and it is not hard to see that a shift of a number's rectangular interpretation k rows down (up) is equivalent to shift of its linear counterpart $k \cdot r$ bits to the left (right).

Shifting by columns is slightly more complicated, and first we investigate the effect of a shift of linear interpretation x_l on a rectangular one x_r. Shifting x_l h bits to the right, sets x_l.b[k] to the value of x_l.b[$k-h$]. Further, let these bits correspond to x_r.b[i_1, j_1] and x_r.b[i_2, j_2] respectively. Now, if x_r.b[i_1, j_1] and x_r.b[i_2, j_2] are both in the same row, that is if $j_1 = j_2$, the shift looks like a shift inside a row. However, if they are in different rows, x_r.b[i_1, j_1] is "wrapped around" to the right end of a row. For example, in Figure 4.8 registers $P^{\searrow}_{c,\delta}$ and $P^{\nearrow}_{c,\delta}$ are shifted to the right for k bits and the wrapped around bits appear in a row above of the original one. Similar conclusions can be drawn for shift to the left illustrated in Figure 4.9. This brings us to the constant time shifting of rectangular registers by rows and columns:

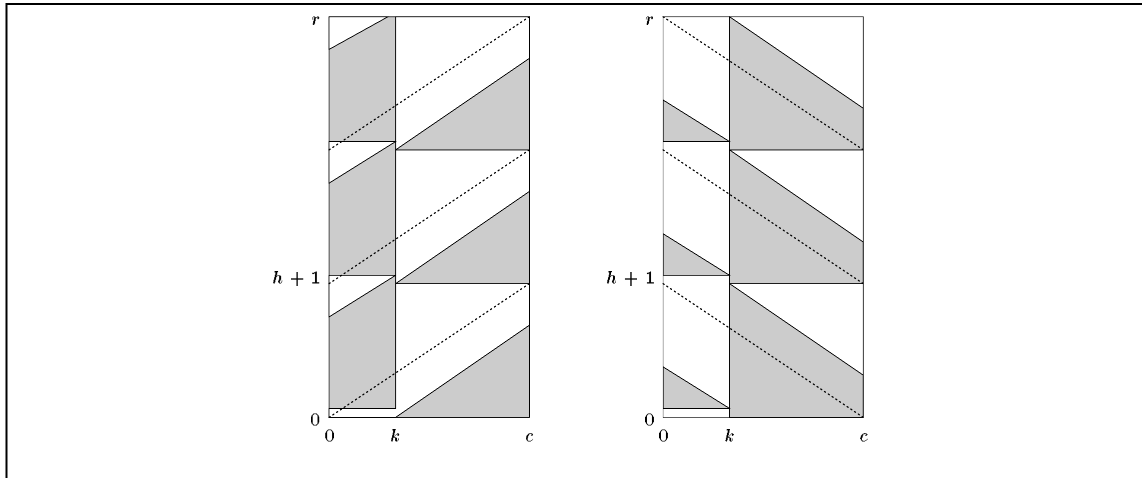

Figure 4.8: *Shifting of $P^{\nearrow}_{r,\delta}$ and $P^{\searrow}_{r,\delta}$ k bits to the right.*

Example 4.6: First, as mentioned, shifting k rows up (`ShiftUpRows`) and down (`ShiftDownRows`) is equivalent to shifting right and left $k \cdot c$ bits respectively.

Shifting k columns left and right is basically shifting left and right for k bits respectively, with additional elimination of unwanted portions of the result. For example, in the shift k columns right (cf. Figure 4.8) we want to eliminate left k columns which form $S^C_{0,k}$. Similarly, at the shift k columns left (cf. Figure 4.9) we have to deal with the right k columns – $S^C_{k,c-k}$. We can eliminate columns by setting them either to 0 or to 1 (see Figure 4.10). In Algorithm 4.8 we indicate this by the third parameter `shiftedIn`. There is similar algorithm for shifting to the left (`ShiftLeftColumns`) while all algorithms for shifting by rows or columns have the same parameters.

Finally, since column-stripe and row-stripe registers can be generated in constant time (see Example 4.4), all shifting algorithms run in constant time as well. $\mathcal{EF}$

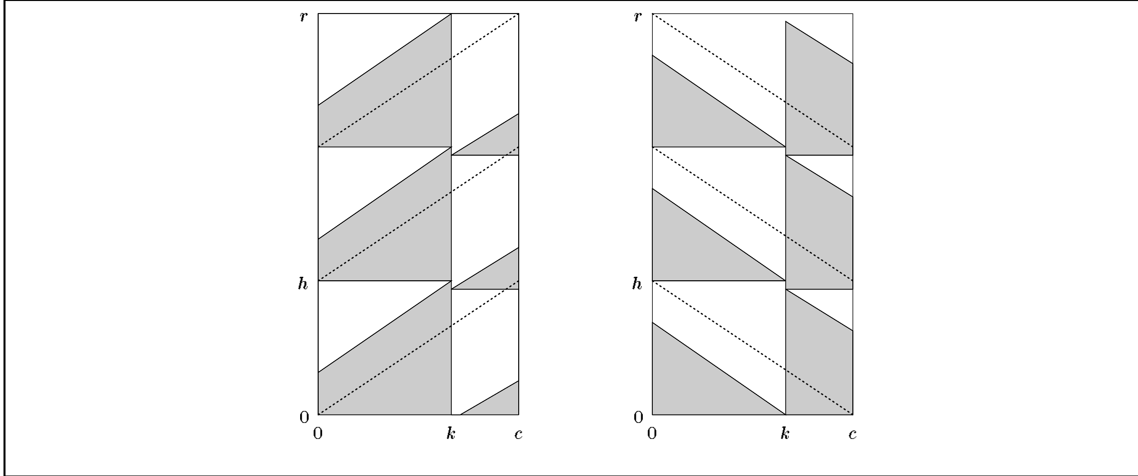

Figure 4.9: *Shifting of* $P^{\nearrow}_{r,\delta}$ *and* $P^{\searrow}_{r,\delta}$ k *bits to the left.*

```
PROCEDURE ShiftRightColumns (x, k, shiftedIn);
  x:= ShiftRight (x, k);
  IF shiftedIn THEN RETURN x OR S^C_{0,k};
  ELSE RETURN x AND S^C_{k,c-k};
  END;
END ShiftRightColumns;
```

Algorithm 4.8: *Shifting of the rectangular register* x k *columns to the right.*

4.5 Multidimensional Registers

The last interpretation of a number is an obvious generalization to d dimensions:

Definition 4.9 *The hyper-cuboidal register* x_c *has the following properties:*

(i.) it has dimensions $\{s_1, s_2, \ldots, s_d\}$ *where* $d = O(1)$ *is some predefined constant;*

(ii.) the total number of bits in a register is $b = \prod_{i=1}^{d} s_i$*;*

(iii.) the bit in position $(\pi_1, \pi_2, \ldots, \pi_d)$*, where* $0 \leq \pi_i < s_i$ *and* $0 < i \leq d$*, is denoted by* $x_c.b[\pi_1, \pi_2, \ldots, \pi_i, \ldots, \pi_d]$*;*

(iv.) bit $x_c.b[0, 0, \ldots, 0]$ *is the least significant bit;*

(v.) the integer value stored in the register is

$$\sum_{\pi_1=0}^{s_1-1} \sum_{\pi_2=0}^{s_2-1} \cdots \sum_{\pi_d=0}^{s_d-1} \left(x_c.b[\pi_1, \pi_2, \ldots, \pi_i, \ldots, \pi_d] \cdot 2^{\sum_{k=1}^{d} \pi_k \prod_{j=1}^{k-1} s_j} \right) \quad . \tag{4.25}$$

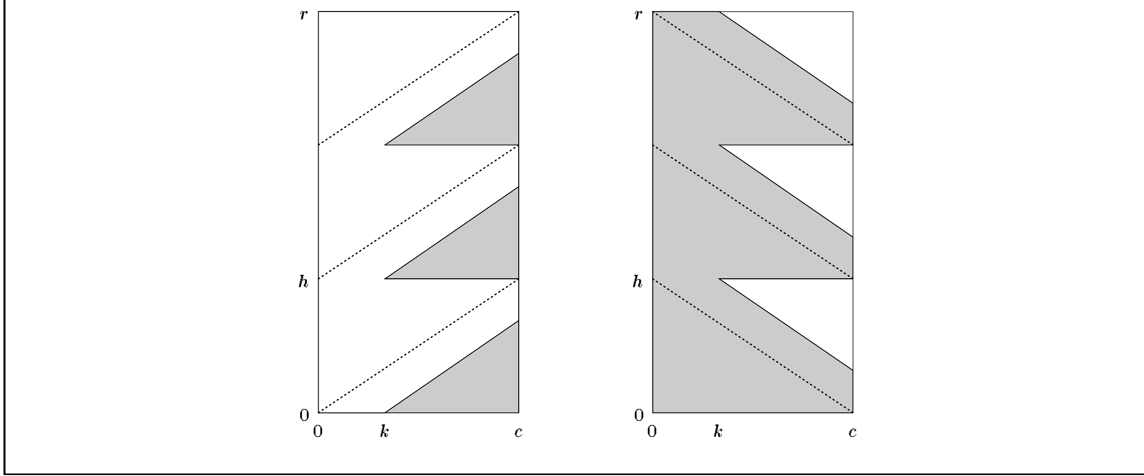

Figure 4.10: *Results of* `ShiftRightColumns(`$P^{\nearrow}_{r,\delta}$`, `k`, FALSE)` *and* `ShiftRightColumns(`$P^{\searrow}_{r,\delta}$`, `k`, TRUE)`.

Setting d to 1 or 2 in Definition 4.9 and eq. (4.25) gives corresponding definitions for linear or rectangular registers respectively. Similar observation can be made for every multidimensional entity in this section.

In § 4.4 we dealt with a general form of rectangular registers and similarly we could deal with hyper-cuboidal ones. However, we are interested only in its special form:

Definition 4.10 *A* ***hyper-cubic*** *register is a hyper-cuboidal register with the same number of bits in all dimensions* ($0 < i \leq d$)

$$s_i = p = \sqrt[d]{b} \tag{4.26}$$

As with as rectangular and linear registers, hyper-cubic registers are also just an interpretation of a number. The bijective mapping between different interpretations is based on the fact that change of an interpretation must not change the value of a number. Let x_c be a hyper-cubic, x_r a $p^{d-k} \times p^k$ rectangular, and x_l a linear interpretation of the same number. Then bits of x_c and x_l are related by

$$x_l.\mathtt{b}[\sum_{k=1}^{d} \pi_k \cdot p^{k-1}] = x_c.\mathtt{b}[\pi_1, \pi_2, \ldots, \pi_k, \ldots \pi_d] \ , \tag{4.27}$$

and, using eq. (4.12), bits of x_c and x_r by

$$\begin{aligned} x_r.\mathtt{b}[i,j] &= x_l.\mathtt{b}[i + j \cdot p^k] && \text{by eq. (4.12)} \\ &= x_l.\mathtt{b}[(\sum_{l=1}^{k} \pi_l \cdot p^{l-1}) + (\sum_{l=k+1}^{d} \pi_l \cdot p^{l-k-1}) \cdot p^k] \\ &= x_c.\mathtt{b}[\pi_1, \pi_2, \ldots, \pi_l, \ldots \pi_d] && \text{by eq. (4.27)} \end{aligned}$$

where

$$\begin{aligned} 0 \quad \leq \quad & i = \sum_{l=1}^{k} \pi_l \cdot p^{l-1} \quad < \quad p^k \\ 0 \quad \leq \quad & j = \sum_{l=k+1}^{d} \pi_l \cdot p^{l-k-1} \quad < \quad p^{d-k} \ . \end{aligned} \tag{4.28}$$

There are some useful mappings of special hyper-cubic registers to rectangular ones:

EXAMPLE 4.7: Consider a hyper-cubic register with bits set if $\pi_k = \pi$. Then, using eq. (4.28), indices of set bits in its $p^{d-k} \times p^k$ rectangular interpretation are $x_r.\mathtt{b}[i, j]$, where $\pi \cdot p^{k-1} \leq i < (\pi + 1) \cdot p^{k-1}$ for every row $0 \leq j < p^{d-k}$. Thus, by eq. (4.14), x_r is a column-stripe register $S^C_{\pi \cdot p', (\pi+1) \cdot p'}$ where $p' = p^{k-1}$ (see Figure 4.11). $\mathcal{EF}$

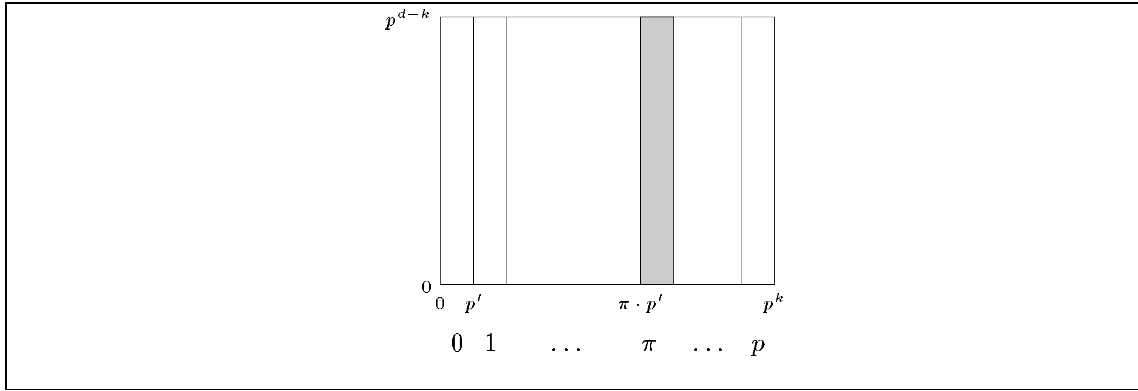

Figure 4.11: *Rectangular interpretation of a hyper-cubic register with bits set if $\pi_k = \pi$ is a column-stripe register.*

The last numbers we introduce have the following hyper-cubic interpretation

$$H_{k,l}.\mathtt{b}[\ldots, \pi_k, \ldots, \pi_l, \ldots] = \begin{cases} 1 & \text{if } \pi_k \geq \pi_l \\ 0 & \text{otherwise} \end{cases} \tag{4.29}$$

and

$$H_{k,-l}.\mathtt{b}[\ldots, \pi_k, \ldots, \pi_l, \ldots] = \begin{cases} 1 & \text{if } \pi_k \geq p - \pi_l \\ 0 & \text{otherwise} \ . \end{cases} \tag{4.30}$$

That is, $H_{k,l}$ has those bits set for which the kth index is smaller than the lth (for $k < l$). A similar characterization holds for $H_{k,-l}$. Figure 4.12 shows the relation between indices in both registers, while the following example investigates their rectangular interpretations:

EXAMPLE 4.8: Consider $p^{d-k} \times p^k$ rectangular interpretation x_r of $H_{k,l}$. First, by eq. (4.28) and eq. (4.29), the set bits of $x_r.\mathtt{b}[i, j]$ are

$$i = \sum_{g=1}^{k-1} \pi_g \cdot p^{g-1} + \pi_k \cdot p^{k-1} \geq \sum_{g=1}^{k-1} \pi_g \cdot p^{g-1} + \pi_l \cdot p^{k-1} \ . \tag{4.31}$$

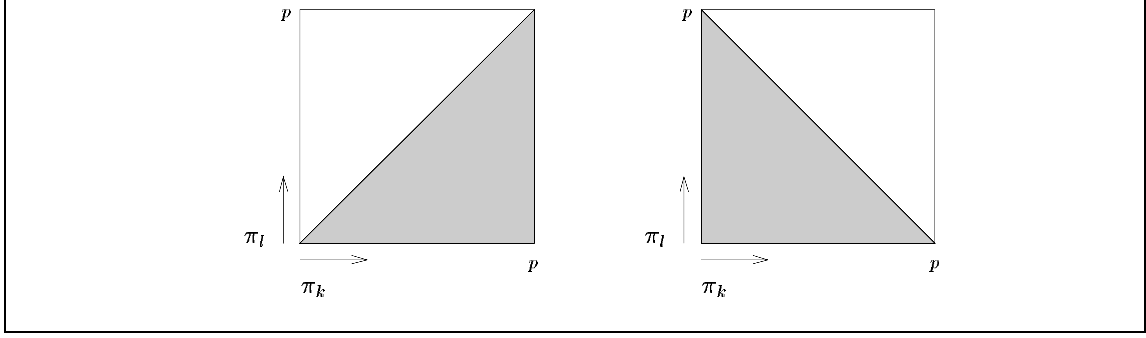

Figure 4.12: *Relation between k^{th} and l^{th} index of $H_{k,l}$ and $H_{k,-l}$.*

Further, if $h = p^{l-k}$ and $\delta = p^{2k-l}$, then by eq. (4.28) we get

$$\begin{aligned} j \bmod h &= \left(\sum_{g=k+1}^{l-1} \pi_g \cdot p^{g-k-1} + \pi_l \cdot p^{l-k-1} + \sum_{g=l+1}^{d} \pi_g \cdot p^{g-k-1} \right) \bmod h \\ &= \sum_{g=k+1}^{l-1} \pi_g \cdot p^{g-k-1} + (\pi_l \cdot p^{k-1}) \cdot p^{l-2k} \\ &= \sum_{g=k+1}^{l-1} \pi_g \cdot p^{g-k-1} + (\pi_l \cdot p^{k-1}) \cdot \delta^{-1} \ . \end{aligned}$$

Next, we rewrite $\sum_{g=k+1}^{l-1} \pi_g \cdot p^{g-k-1}$ to $\sum_{g=2k-l+1}^{k-1} \pi_g \cdot p^{g-2k+l-1}$ and using eq. (4.31) get

$$\begin{aligned} i &\geq \left\lfloor \delta(j \bmod h) + \sum_{g=1}^{k-1} \pi_g \cdot p^{g-1} - \sum_{g=2k-l+1}^{k-1} \pi_g \cdot p^{g-2k+l-1} \cdot \delta \right\rfloor \\ &\geq \left\lfloor \delta(j \bmod h) + \sum_{g=1}^{k-1} \pi_g \cdot p^{g-1} - \sum_{g=2k-l+1}^{k-1} \pi_g \cdot p^{g-1} \right\rfloor \\ &\geq \left\lfloor \delta(j \bmod h) + \sum_{g=1}^{2k-l} \pi_g \cdot p^{g-1} \right\rfloor \end{aligned}$$

where the sum is at most $p^{2k-l} - 1 = \delta - 1$ and thus $i \geq \lfloor \delta(j \bmod h + 1) \rfloor - 1$. This inequality defines $P^{\searrow}_{r,\delta}$ from eq. (4.22); that is, number $H_{k,l}$ is in $p^{d-k} \times p^k$ rectangular interpretation $P^{\searrow}_{r,\delta}$ where $\delta = p^{2k-l}$. Similarly, $H_{i,-j}$ is in a rectangular interpretation $P^{\nearrow}_{r,\delta}$ as defined in eq. (4.22). $\mathcal{EF}$

4.6 Extremal Bits in Linear Registers

In the rest of this chapter we are looking for the extremal set bits in registers. The first register we consider is a linear register and in particular we search for the left most (the least significant) set bit in an s-small linear register:

Lemma 4.2 *Let $s^2 + \lceil \lg s \rceil - 1 \leq b$. Then there is a constant time algorithm which finds the left most set bit in an s-small linear register using $O(m)$ bits of memory.*

Proof: Without loss of generality we assume the number in question $x \neq 0$ and $s^2 + \lceil \lg s \rceil - 1 = b$. The main idea of Algorithm 4.9 is to iterate i from 0 to $s-1$ and count the number of iterations in which no less significant bit than x.b[i] is set. Obviously, this count gives us the index of the least significant set bit in x.

```
PROCEDURE Lmb (x)
  x_d:= Distribute1D (s+1, s, x);
  FOR i:=0 TO s-1 DO PARALLEL
    offset:= 2^(i(s+1));
    x_s:= Negate ((x_d AND ((2^(i+1)-1)*offset)) + (2^s-1)*offset) AND (2^s*offset);
  END;
  x_s:= ShiftLeft (x_s, s);
  RETURN ShiftLeft (x_s * (2^(s^2+s)-1)/(2^(s+1)-1), s*s-1) AND (2^(s+1)-1)
END Lmb;
```

Algorithm 4.9: *Searching for the left most set bit in an s-small linear register.*

In detail, the algorithm works in three steps: first, a block of $s+1$ least significant bits of x is distributed across the register using Algorithm 4.2;[7] second, the most significant bit in the ith copy is set iff none of the less significant bit in the copy than the ith is set; and third, the number of set bits is counted. The last two step are implemented using word-size parallelism.

To describe the second step consider an expression

$$(x_i \wedge (2^{i+1} - 1)) + (2^s - 1) \tag{4.32}$$

where x_i is the ith copy of $s+1$ least significant bits in x. The first term of the expression masks out all but i left most bits, and the second term "slides" the remaining set bits to position b[$s+1$]. In other words, if none of the i left most bits was set then the most significant bit is 0 and otherwise it is 1. Obviously, if we negate expression (4.32) and mask it with 2^s we end up with the most significant bit set iff none of i left most bits in x_i

[7] Note the most significant bit of a block, and thus in each of its copies, is 0.

was set. This is also the expression used in each iteration of the second step – the parallel loop of Algorithm 4.9. The most significant bits of all copies form $(s+1,1)$-sparse linear register with an offset $a = s$ (see Definition 4.2). Finally, we shift the register for s bits left and get $x_s = \sum_{i=0}^{q-1} x_s.\mathtt{b}[i \cdot t]\, 2^{it}$, where $t = s+1$, and $q \cdot t \leq b$.

The last step of Algorithm 4.9 counts the number of set bits in x_s. Let us multiply[8] x_s by $\kappa = \sum_{j=0}^{q-1} 2^{jt} = \frac{2^{qt}-1}{2^t-1}$ which gives

$$x_s \cdot \kappa = \left(\sum_{i=0}^{q-1} x_s.\mathtt{b}[i \cdot t] \cdot 2^{it} \right) \cdot \left(\sum_{j=0}^{q-1} 2^{jt} \right) = \sum_{k=0}^{2(q-1)} \left(2^{kt} \cdot \sum_{\substack{0 \leq i,j < q \\ i+j=k}} x_s.\mathtt{b}[k \cdot t] \right) \quad . \tag{4.33}$$

Since $t = s+1$ and $q = s$ all internal sums are smaller than $q < 2^t$ and thus they are not in conflict (cf. Lemma 4.1). Moreover, the sum at $k = q-1$ is the sum of all sparse bits in x_s. From eq. (4.33) we also get the bound $(q-1) \cdot t + \lceil \lg q \rceil \leq b$, which, in our case, translates into $s^2 - 1 + \lceil \lg s \rceil \leq b$.

Finally, since all three conditions of Lemma 4.1 are satisfied we replace the parallel loop of Algorithm 4.9 by an expression.

$$\begin{aligned} x_s &= \overline{\left(\left(x_d \wedge \left(\sum_{i=0}^{s-1} (2^{i+1} - 1) \cdot 2^{i(s+1)} \right) \right) + \sum_{i=0}^{s-1} (2^s - 1) \cdot 2^{i(s+1)} \right)} \wedge \sum_{i=0}^{s-1} (2^s \cdot 2^{i(s+1)}) \\ &= \overline{\left(\left(x_d \wedge \left(\sum_{i=0}^{s-1} 2^{i(s+2)+1} - \sum_{i=0}^{s-1} 2^{i(s+1)} \right) \right) + \overline{2^s \cdot \sum_{i=0}^{s-1} 2^{i(s+1)}} \right)} \wedge \left(2^s \cdot \sum_{i=0}^{s-1} 2^{i(s+1)} \right) \\ &= \overline{((x_d \wedge (\zeta - \eta) + \overline{\sigma})} \wedge \sigma \end{aligned}$$

where

$$\zeta = \sum_{i=0}^{s-1} 2^{i(s+2)+1} = \frac{2^{(s+1)^2} - 2}{2^{s+2} - 1} \;, \qquad \eta = \sum_{i=0}^{s-1} 2^{i(s+1)} = \frac{2^{s^2+s} - 1}{2^{s+1} - 1} \;, \text{ and} \qquad \sigma = 2^s \cdot \eta \;.$$

This proves that Algorithm 4.9 runs in $O(1)$ time and $O(m)$ bits of memory. $\mathcal{QED}$

This leads us to the theorem:

Theorem 4.1 *There is a constant time algorithm which computes the index of the left most set bit in a linear register using $O(m)$ bits of memory.*

Proof: Algorithm 4.10 works in two phases: in the first phase it splits linear register x into t blocks of $s = \left\lceil \sqrt{b} + 1 \right\rceil$ bits each and finds the left most non-zero block; in the second phase it computes the index of the left most set bit in the previously found block. The combination of results of both phases gives the index of the left most set bit in x.

---

[8] For sum manipulation formulae see e.g. [58, ch.2].

```
PROCEDURE LMB (x)
  s:= Floor (sqrt (x) - 1); t:= Ceiling (m/s);
                                    (* -- PHASE 1 -- *)
  FOR i:=0 TO t-1 DO PARALLEL                              (* Representatives *)
    offset:= 2^(is);
    x_r:= (x OR ((x AND ((2^(s-1)-1)*offset)) + (2^(s-1)-1)*offset)) AND (2^(s-1)*offset);
  END;
  x_r:= ShiftLeft (x_r, s-1);
  y:= Compress (x_r, s, 1, t);
  k_r:= Lmb (y);                                      (* the index of the block *)
  k_r:= k_r * s;                                      (* which starts at the bit *)
                                    (* -- PHASE 2 -- *)
  x:= ShiftLeft (x, k_r) AND (2^s-1);
  k_b:= Lmb (x);
  RETURN (k_r + k_b);
END LMB;
```

Algorithm 4.10: *Searching for the left most set bit in the linear register x.*

In the first phase each block x_i first "elects" its representative: block's most significant bit x_i.b[s] is set iff at least one bit in the block is set. The election is equivalent to test if x_i.b[s] $= 1$ or if some less significant bit is 1 which is used in

$$\left(x_i \wedge 2^{s-1}\right) \vee \left(\left((x_i \wedge (2^{s-1}-1)) + (2^{s-1}-1)\right) \wedge 2^{s-1}\right) \tag{4.34}$$

where the second term employs the same "sliding" of set bits to the most significant position as it was used in eq. (4.32). Further, eq. (4.34) simplifies into

$$\left(x_i \vee \left((x_i \wedge (2^{s-1}-1)) + (2^{s-1}-1)\right)\right) \wedge 2^{s-1} \quad , \tag{4.35}$$

which is used in the parallel loop of Algorithm 4.10 – another application of word-size parallelism. The loop produces an $(s,1)$-sparse register of representatives that are shifted to the left, to the least significant position, and compressed (using Algorithm 4.4) into a t-small linear register y.[9] The phase concludes by computing the left most set bit in y (using Algorithm 4.9) which also corresponds to the left most non-zero block.

The found block is in the second phase shifted to the least significant position which permits re-application of Algorithm 4.9 on it. Finally, the indices from both phases are combined into the final result.

Next, parameters of the parallel loop in Algorithm 4.10 are not in conflict and all parallel operations are the same. Further, by considering separately the most significant

[9] Note that $t = \lceil \frac{b}{s} \rceil < s = \left\lceil \sqrt{b} + 1 \right\rceil$ and hence we can use `Lmb` and `Compress`.

bit of each block we built a "fire-wall" bit between blocks and hence also intermediate results are not in a conflict. Therefore, using Lemma 4.1, we can replace the loop by the expression

$$\begin{aligned} x_r &= \left(x \vee \left(\left(x \wedge \overline{((2^{s-1}-1)\sum_{i=0}^{t-1} 2^{is})}\right) + \overline{(2^{s-1}-1)\sum_{i=0}^{t-1} 2^{is}}\right)\right) \wedge \left(2^{s-1}\sum_{i=0}^{t-1} 2^{is}\right) \\ &= \left(x \vee \left(\left(x \wedge \overline{(2^{s-1}\sum_{i=0}^{t-1} 2^{is})}\right) + \overline{(2^{s-1}\sum_{i=0}^{t-1} 2^{is})}\right)\right) \wedge \left(2^{s-1}\sum_{i=0}^{t-1} 2^{is}\right) \end{aligned} \tag{4.36}$$

which simplifies into $(x \vee ((x \wedge \overline{\sigma}) + \overline{\sigma})) \wedge \sigma$ where $\sigma = 2^{s-1}\sum_{i=0}^{t-1} 2^{is} = \frac{2^{st+s-1}-2^{s-1}}{2^s-1}$. $\mathcal{QED}$

Unfolding function calls in Algorithm 4.10 and optimizing code a bit, we get less than 30 instructions (excluding assignments), none of which is branching (cf. [22]). This makes Algorithm 4.10 especially suitable for modern pipelined computer architectures. Finally, all algorithms in this section assume non-zero parameters, and to make them robust, a proper test has to be added.

It is not hard to verify that by omitting negation in a parallel loop of Algorithm 4.9 we get function `Rmb` which finds the right most set bit of s-small linear register. Furthermore, by replacing calls of `Lmb` by calls of `Rmb` in Algorithm 4.10 we get function `RMB` which computes the right most set bit in a linear register[10]. This brings us to the theorem:

Theorem 4.2 *There is a constant time algorithm which computes the index of the right most set bit in a linear register using $O(m)$ bits of memory.*

Proof: See discussion above or refer to the computation of $\lfloor \lg x \rfloor$ in [53].[11] $\mathcal{QED}$

4.7 Extremal Bits in Rectangular Registers

In the previous section we searched for extremal bits in linear registers and here we do the same in rectangular ones. In rectangular registers we have four extremal set bits: the left most, the bottom most, the right most and the top most. They are not defined unambiguously, because there can be more than one set bit in the same row or column. However, in this work any of these ambiguous bits is acceptable. Therefore, and because of bijective mapping in eq. (4.12), function `LMB` from Algorithm 4.10 is used to find the bottom most set bit. Similarly, procedure `RMB` is used to find the top most set bit. Thus:

[10] The computation of the most significant set bit in x is equivalent to $\lfloor \lg x \rfloor$.

[11] Indeed, our algorithms were inspired by work of Fredman and Willard, though, because of the word-size parallelism, we could develop and verify them in a more straightforward manner.

Corollary 4.1 *The indices of the bottom most and the top most set bits in a rectangular register are computed in constant time using $O(m)$ bits of memory.*

Let `BMBofRectReg` (x, r, c) (`TMBofRectReg` (x, r, c)) denote the function that computes the index of the bottommost (topmost) set bits in $r \times c$ rectangular register x.

To find the left most and the right most set bits it is sufficient to find the extremal non-zero columns and then choose any set bit in them – in particular the top or the bottom most. We describe in detail only the algorithm for search of the left most non-zero column, while the search for the right most is just sketched.

Now, if we count the number of bits in each column, then the left most non-zero count represents the left most non-zero column. Unfortunately, we can count bits only in columns of s-column sparse rectangular registers:

Lemma 4.3 *Let x be an s-column sparse rectangular register ($\lceil \lg r \rceil \leq s \leq c$) and let*

$$y = ((x \cdot S_{0,1}^C) \text{ div } 2^{c(r-1)}) \wedge S_{0,1}^R \quad . \tag{4.37}$$

Then the value stored in bits $y.b[is, 0] \ldots y.b[(is + s - 1), 0]$, is the sum of bits in the $(is)^{th}$ column of x. That is $\sum_{k=0}^{s-1} y.b[is + k, 0] \cdot 2^k = \sum_{l=0}^{c-1} x.b[is, l]$.

Proof: Algorithm 4.11 first applies eq. (4.33) in parallel for all non-zero columns x_i. Note, a single column is a $(c, 1)$-sparse linear register. Now, since $u = \lfloor \frac{c}{s} \rfloor$, and $t = c$, q and κ in eq. (4.33) became r and $S_{0,1}^C$ respectively. Further, using mapping from eq. (4.12), we observe that eq. (4.33) leaves the counting result for a^{th} column in the ath column of the $(r - 1)^{\text{st}}$ row for any $0 \leq a < c - \lceil \lg r \rceil$ (cf. Definition 4.2). Therefore, we must shift results down for $r - 1$ rows and mask out the unwanted rows (see Example 4.6).

```
PROCEDURE CountBits (x, s);
  FOR l:=0 TO u-1 DO PARALLEL
    y:= x_i * S^C_{0,1};
  END;
  RETURN ShiftDownRows (y, r-1, FALSE)
END CountBits;
```

Algorithm 4.11: *Counting the number of set bits in individual columns of the s-column sparse rectangular register x.*

Next, all operations in the parallel loop are identical, and parameters are not in conflict. To prove that partial results are not in conflict either we observe that in the proof of Lemma 4.2 internal sums of eq. (4.33) are all smaller than t, which in our case

is r. Now, since $s \geq \lceil \lg r \rceil$ no two internal sums inside an individual column nor between two columns are in conflict. Finally, by Lemma 4.1, the parallel loop of Algorithm 4.11 is replaced by $x \cdot S_{0,1}^{C}$ and subsequently the whole algorithm by eq. (4.37). $\mathcal{QED}$

Now it is easy to find the extremal non-zero column of s-column sparse register:

Lemma 4.4 *Let x be an s-column sparse rectangular register, where $\lceil \lg r \rceil \leq s \leq \lceil \sqrt{c} + 1 \rceil$. Then there is a constant time algorithm which finds the left (right) most non-zero column in x using $O(m)$ bits of memory.*

Proof: Algorithm 4.12 adds up each column and then finds the left most non-zero sum. Replacing the call of function `LMB` by a call of `RMB` (right most set bit) we get procedure `Rmc` which finds the right most non-zero column. The lemma follows immediately from Lemma 4.3, and Theorem 4.1. $\mathcal{QED}$

```
PROCEDURE Lmc (x, s);
  x_c:= CountBits (x, s);
  x_r:= LMB (x_c);
  RETURN x_r DIV s;
END Lmc;
```

Algorithm 4.12: *Searching for the left most non-empty column in the s-column sparse rectangular register x.*

It remains to show how to find the left most non-zero column in an arbitrary rectangular register:

Lemma 4.5 *There is a constant time algorithm which computes the index of the left most non-empty column in a rectangular register x using $O(m)$ bits of memory.*

Proof: Algorithm 4.13 works in similar two phases as Algorithm 4.10: in the first phase it splits the register into column stripes $s = \lceil \sqrt{c} + 1 \rceil$ bits wide and finds the left most non-zero stripe, which, in the second phase, searches for the left most non-zero column. Combination of results of both phases gives the index of the searched column.

First, each stripe elects its representatives (cf. Theorem 4.1) using eq. (4.35). The election is done in parallel for all stripes and for all rows, where the double parallel loop can be replaced by a single one. The representatives are than shifted to the left-most column creating an s-column sparse rectangular register x_r. The left most non-zero column in x_r, which defines the left most non-zero stripe, is then found using Algorithm 4.12.

In the second phase, the found stripe is shifted to the left most column and then spread across the register using Algorithm 4.7 creating again an s-column sparse rectangular register. However, this time the columns are in a reverse order which is taken into

```
PROCEDURE LMC (x, r, c);
                              (* -- PHASE 1 -- *)
  s:= Ceiling (Sqrt (r) + 1); t:= Ceiling (r/s);
  FOR j:=0 TO r-1 DO PARALLEL                          (* Representatives *)
    FOR i:=0 TO t-1 DO PARALLEL
      offset:= 2^(is+jc);
      x_r:= (x OR ((x AND ((2^(s-1)-1)*offset)) + (2^(s-1)-1)*offset)) AND
           (2^(s-1)*offset);
    END;
  END;
  x_r:= ShiftLeftColumns (x_r, s-1, FALSE);
  stripe:= Lmc (x_r, s)*s;               (* start of the left most non-zero stripe *)
                              (* -- PHASE 2 -- *)
  x:= ShiftLeftColumns (x, stripe, FALSE);
  x:= Spread2D (x, s);
  inStripe:= s - Rmc (x, s);
  RETURN (stripe + inStripe)
END LMC;
```

Algorithm 4.13: *Searching for the left most non-empty column in the $r \times c$ rectangular register x.*

account by applying the function `Rmc`, which returns the right most non-zero column. The combination of both phases' results gives the searched column.

Finally, since all three conditions of Lemma 4.1 are satisfied we can replace parallel loops in Algorithm 4.13 by the expression

$$\left(x\vee\left(\left(x\wedge((2^{s-1}-1)\cdot\sum_{i=0}^{t-1}\sum_{j=0}^{r-1}2^{is+jc})\right)+(2^{s-1}-1)\cdot\sum_{i=0}^{t-1}\sum_{j=0}^{r-1}2^{is+jc}\right)\right)\wedge\left(2^{s-1}\cdot\sum_{i=0}^{t-1}\sum_{j=0}^{r-1}2^{is+jc}\right)$$

which, using $S_{0,1}^{C}$ from eq. (4.14) and $\sigma = \sum_{i=0}^{t-1} 2^{is} = \frac{2^{st}-1}{2^s-1}$ simplifies into

$$\left(x \vee \left(\left(x \wedge \overline{(2^{s-1} \cdot S_{0,1}^{C} \cdot \sigma)}\right) + \overline{(2^{s-1} \cdot S_{0,1}^{C} \cdot \sigma)}\right)\right) \wedge \left(2^{s-1} \cdot S_{0,1}^{C} \cdot \sigma\right) \quad . \tag{4.38}$$

$\mathcal{QED}$

Note, that for $r = 1$ eq. (4.38) becomes eq. (4.36) as expected. Further, it is not hard to see if in Algorithm 4.13 we swap calls of functions `Lmc` by `Rmc` we get procedure `RMC` which finds the right most non-zero column in x. This brings us to the final theorem:

Theorem 4.3 *Let x be a rectangular register, then there are constant time algorithms which find the extremal set bits in x using $O(m)$ bits of memory.*

Proof: To find the top most and the bottom most set bit see Corollary 4.1. Further, Algorithm 4.14 finds the left most set bit of a rectangular register, while the implementation of a similar function to find the right most set bit, RMBofRectReg, is similar. $\mathcal{QED}$

```
PROCEDURE LMBofRectReg (x, r, c);
  i:= LMC (x, r, c);                                  (* first find the column *)
  x:= ShiftLeftColumns (x, row, FALSE);                 (* shift it to the left *)
  x:= Compress (x, r, 1, c);                            (* compress the rest *)
  j:= LMB (x);                                          (* and get the row *)
  RETURN [i, j]
END LMBofRectReg;
```

Algorithm 4.14: *Searching for the left most set bit in the $r \times c$ rectangular register x.*

4.8 Extremal Bits in Multidimensional Registers

In this section we are interested in hyper-cubic registers only. The searching algorithms for extremal set bits in them heavily depends on a mapping between multidimensional and rectangular registers in eq. (4.28). As in two dimensions, many set bits may have the same k^{th} dimension index, and we accept any such bit as a feasible solution.

Theorem 4.4 *Let x be a hyper-cubic register. Then there is a constant time algorithm which finds the extremal set bits in the kth dimension of x using $O(m)$ bits of space.*

Proof: In general there are two extremal set bits in the kth dimension: the left most and the right most – the one with the smallest and with the largest k^{th} index respectively. In Example 4.7 we saw that hyper-cubic registers with set π^{th} bit in the kth dimension are under rectangular interpretation column-stripe registers. Thus, the search for the left most set bit in the kth dimension of x is equivalent to the search for the left most set bit in its $p^{d-k} \times p^k$ rectangular interpretation as shown in Algorithm 4.15. The implementation of RMBofCubic is almost identical. $\mathcal{QED}$

```
PROCEDURE LMBofCubic (k, x);
  RETURN LMBofRectReg (x, p^(d-k), p^k);
END LMBofCubic;
```

Algorithm 4.15: *Searching for the left most set bit in the kth dimension of the hyper-cubic register x.*

4.9 Conclusions

This chapter used the programming technique of *word-size parallelism* in a number of examples. The main advantage of word-size parallelism is that it permits a formal development of fast sequential algorithms using parallelism inherently available in processor's instruction set. It is not surprising that the operations from which the technique benefited most are integer multiplication and division, and bitwise Boolean operations. The intuition behind this is that multiplication and division permit rapid information spreading across a register.

We also formally introduced linear, rectangular and hyper-cubic registers, revealing them as different interpretations of one another. Finally, using word-size parallelism we developed constant time algorithms for search of extremal set bits in the various registers. These algorithms will be used in the following chapters.

Chapter 5

The Closest Neighbour in Constant Time

Μὴ μοῦ τοὺς κύκλους τάραττε

Ἀρχιμήδης

Don't touch my circles!

Archimedes

Chapter 3 dealt with the problem of a simple membership over a finite universe. In this chapter we extend the operations to include finding the closest value in the given set to a query element.

The chapter consists of three major parts. First we define the problem with some additional notation and review the literature. The bulk of the chapter deals with solutions to the problem in one, two and d dimensions. The final section includes some conclusions and a short discussion.

5.1 Introduction

Given a set of points, a query point and a distance metric, the closest neighbour problem is that of determining the point of the set whose distance from the query point is minimal. Note, that if the query point is a member of the given set then it will be the solution. Furthermore, if two or more elements are of equal distance from the query point we choose one of them arbitrarily. Most of our attention will be restricted to the L_∞ and L_1 norms, though we will keep as much of the discussion as possible independent of the norm.

The closest neighbour problem arises in many other areas such as modeling of robot arm movements and integrated circuits layouts (cf. [96]). In computational geometry the problem is usually solved using Voronoi diagrams. Furthermore, the problem can be generalized by considering the points as multidimensional records in which individual fields are drawn from an ordered domain (cf. [81]).

In § 5.2.1 we give a general overview of the problem, but the principal version we address in this chapter is a static neighbourhood problem in a bounded universe on a d-dimensional grid (d is fixed) under the norm L_∞. The solution to the problem is presented as a combination of two straightforward approaches: under the first, the universe is represented by a bit map; and under the second, each point of the universe "knows" who is its closest neighbour – it has a pointer to the closest point. Under the model we use (cf. Definition 2.4) these approaches use M^d and $M^d \cdot \lg M$ bits of space respectively. The advantage of the first approach is that it minimizes space required if the measure is based solely on the size of the universe. The second guarantees constant response time. Using word-size parallelism introduced in § 4 and some geometric properties of the norm L_∞, we are able to combine both approaches into a constant time solution using $M^d + o(M^d)$ bits of space.

Our general approach is to divide the universe into regions we call *tiles*. Each tile contains a number of universe points equal to the number of bits necessary to write down the coordinates of an arbitrary point in the universe. If any of the points in a tile are present, then we simply store a bit map representation of a tile; and if a tile is empty we store a *candidate* value. This value is the closest element in the entire set to the middle of the tile. Note that because of a choice of size of a tile, both options require the same

amount of space. The method, however, does not seem to apply to the norm L_2 because candidate values there do not restrict the searching space sufficiently.

In this chapter we show that the closest neighbour to any query point, in a d-dimensional space under the norm L_∞, can be determined by inspecting $O(d^2 \cdot 2^d) = O(1)$ tiles. On the other hand, word-size parallelism facilitates finding the closest element in a tile represented by a bit map.

5.2 Definitions and Background

In general we deal with the set of points in d-dimensional space where d is a predefined constant. The points are defined by d orthogonal coordinates

$$T = (x_1, x_2, \ldots, x_d) \tag{5.1}$$

where each individual coordinate is chosen from a bounded universe of size M as defined in Definition 2.5.

Given two points $T_1 = (x_{1,1}, x_{1,2}, \ldots, x_{1,d})$ and $T_2 = (x_{2,1}, x_{2,2}, \ldots, x_{2,d})$, there are a number of different measures of distance between them. However, the family of distance functions we use has a general form (cf. [96, p.222])

$$\delta_f(T_1, T_2) = (\sum_{i=1}^{d} \left|(x_{1,i} - x_{2,i})^f\right|)^{\frac{1}{f}} \tag{5.2}$$

for a real parameter $1 \leq f \leq \infty$. The distance function $\delta_f(\cdot)$ also defines L_f, the *norm of the space*. The family of the distance functions defined this way satisfy the *triangle inequality*,

$$\delta_f(T_1, T_2) + \delta_f(T_2, T_3) \geq \delta_f(T_1, T_3) \ . \tag{5.3}$$

Although in eq. (5.2) $1 \leq f \leq \infty$, we will focus on $f = \infty$ which, as the limit as $f \to \infty$, defines the distance function

$$\delta_\infty(T_1, T_2) = \max_{0<i\leq d} |x_{1,i} - x_{2,i}| \ . \tag{5.4}$$

When d is 1 or 2, things are a bit easier. First, when $d = 1$ eq. (5.2) becomes $|x_{1,1} - x_{2,1}|$ for any f. Secondly, Lee and Wong ([75]) proved that in two dimensional space ($d = 2$), a search for the closest neighbour under L_∞ is computationally equivalent to a search under L_1.

In this chapter we use the ERAM machine model from Definition 2.4. The instruction set of this model includes integer multiplication and division, and bitwise Boolean operations, while the width of a memory register and transportation channel is m bits

($m = \lg M$). In accordance with Definition 2.5 we assume that one memory register is large enough to store one coordinate of a point. Further, d *memory registers* are grouped together and they can represent either one point in the universe or a $b = d \cdot m$ point small universe called a *tile*. Note that b is also the width of ERAM *arithmetic registers* and, therefore, the active block of ERAM, its processor, can handle one tile at a time – reading $b = d \cdot m$ bits takes time d but arithmetic on them only takes unit time.

Based on eq. (5.2) we formally define the problem:

Definition 5.1 *Let $\mathcal{N}$ be a subset of points from the universe $\mathcal{M} = [0 \ldots M]^d$. The* ***static closest neighbour*** *problem is to represent these points in a data structure so that given a query point, $T \in \mathcal{M}$, the closest member of $\mathcal{N}$ under the norm L_f can be found efficiently.*

Note that if the query point is in the set, then it is its own closest neighbour. Furthermore, if there are several points of minimal distance to the query point, any of them is a satisfactory answer. The dynamic version of a problem is addressed in § 6.

All solutions presented in this chapter consist of two parts: first we explain how to search for the closest neighbour in a small, b-point universe, and second how to search in a big M^d-point universe ($M^d = 2^b$).

Throughout the section we assume that all divisions which define the size of a problem at hand do not produce a reminder. It can be verified, that by dropping this assumption, all algorithms and data structures remain correct, though the third order terms of the space bounds may be changed.

5.2.1 Literature Background

Finding the closest element in a set to a query element is an important problem arising in many subfields of computer science, including computational geometry, pattern recognition, VLSI design, data compression and learning theory (cf. [34, 81, 96]). We will highlight some of the key ideas that have been applied to various versions of the problem and relate them to the approach taken in our solutions.

As noted in previous section, there are several versions of the problem. Clearly, the number of dimensions, d, and the distance norm, typically L_2, L_1 or L_∞, impact the appropriate choice of methods.

First we consider a continuous searching space (domain), and in it, the basic static version of the problem and its deterministic solutions. In one dimensional space, where all norms are equivalent, there is a simple logarithmic lower bound under the comparison based model[1] which is matched by a binary search algorithm.

[1] A comparison based model is essentially a pointer machine model (cf. [101]).

In two dimensions and under the Euclidean norm, L_2, the problem is also known as a *post-office* problem ([70]) and is related to the *point-location* problem (cf. [40]). The most common approach to solve it is to use Voronoi diagrams (cf. [40, 96, 117]) which gives logarithmic running time (see also [28]). However, Chang and Wu in [26] went outside the comparison based model. Using hashing they achieved constant running time at the cost of using, in the worst case, $O(N^2+M)$ words. All Voronoi diagram based approaches have similar bounds also under the norms L_1 and L_∞, although the diagrams have different shapes ([75]). Finally, most logarithmic solutions generalize to higher dimensions at the expense of using $O(N^{2^{d+1}})$ words ([117]).

Going to probabilistic solutions, we observe that they range from expected logarithmic time under the comparison based model (cf. [89, 103]), to expected constant time under the random access machine model which includes integer division and multiplication (cf. [20]). All mentioned solutions use $O(N)$ words.

The next distinction is between static and dynamic versions of the problem. Unlike the static problem, there is no known efficient deterministic solution to the dynamic version. There are, however, poly-logarithmic expected time algorithms to maintain Voronoi diagrams under the comparison based model ([31]) and constant time probabilistic solutions under the random access machine model with integer division and multiplication (cf. [46, 97]). Both of these use $O(N)$ words.

This distinction can be extended even further. One version is to consider the problem of being given the set and a single query point (or perhaps a few points) and being asked for the closest set member to the query point (cf. [35]). Another form of the problem is not to allow the preprocessing otherwise inherently present in all above mentioned static solutions. In this case, we are given the set of points and when the query comes, only the part of a data structure, relevant to the query point, is constructed ([4]) – lazy construction. The first queries take $\Theta(N)$ time, but the subsequent ones might take less time until, eventually (when the complete structure is constructed), they are answered in a logarithmic time. Another variation restricts search to the closest neighbour inside a specific angle (cf. [59, 111]). This problem arises in geographic applications ([27]).

A generalization of the problem is to find the k closest neighbours where k is a predefined constant. Chazelle et al. in [30] use filtering search ([29]) to present a logarithmic time solution which they claim is extensible to higher dimensions. The main idea of filtering search is a two phase approach: first restrict the searching space to a small enough area so that the second phase finds the solution efficiently. The two phases in our solutions, though substantially different, have similar roles.

A further generalization is to search for the kth closest, or k closest neighbours, where k is not fixed in advance (cf. [36]). Arya et al. in [13] present a logarithmic approximate solution to the later problem for any number of dimensions and any fixed norm. Their approach is to recursively split space into smaller, d-dimensional boxes, with a limited ratio between the longest and the shortest side. Splitting is a very common technique in

neighbourhood related problems (among others also [33, 109, 118, 119]) and we apply it in our solutions as well. However, the split in our case is controlled by the size of the universe and not by the size of the set.

The final distinction we consider is between a continuous and discrete domain upon which, in combination with a bounded universe, we concentrate in this work. The bounded discrete universe permits completely different data structures. For example, Karlsson in [66], and Karlsson, Munro and Robertson in [67] adapt the van Emde Boas et al. one-dimensional stratified trees ([45]) to two dimensions. Thus they achieve, for a static problem in a universe of size $M \times M$ points and under the norms L_1 and L_∞, a worst case running time of $O(\log^{(2)} M)$ using $O(N)$ words.

On the other hand, the pattern recognition approach for the same problem usually requires $O(N)$ searching time and uses a plain $M \times M$ bit map though there are cases which use M^2 words rather than bits (cf. [106]). Because of the nature of digitized images on large point sets the $\Theta(M^2)$ bits of space is usually the best one can achieve. The universe in our solutions is essentially represented by a bit map as well. To search for the closest neighbour in a vicinity of a query point, we use word-size parallel algorithms. When the neighbour is farther away, we use additional information stored in the structure.

Finally, the only known lower bound under the cell probe model on the space required to support constant time search in a bounded discrete universe is the trivial one, $\left\lceil \lg \binom{M}{N} \right\rceil$ (see also eq. (3.1)).

5.3 One Dimension

First we study search for the closest neighbour in one dimension. The family of distance functions from eq. (5.2) simplify to

$$\delta(T_1, T_2) = |x_1 - x_2| \tag{5.5}$$

where x_1 and x_2 are coordinates of the respective points.

We present two algorithms: the first assumes the size of the universe is at most m consecutive points, and the second one deals with up to $M = 2^m$ points. We start with the smaller universe:

Theorem 5.1 *Let the size of the universe be m points and let $\mathcal{N}$ be the subset of that universe. Then there is an algorithm which finds the closest neighbour in $\mathcal{N}$ to a query point in constant time using m bits of memory for the data structure and $O(m)$ bits for internal constants that do not depend on $\mathcal{N}$.*

Proof: We represent the set as a simple bit map, `domain`, over the m points of the universe. The presence of the point $U = (u)$ is indicated by bit u being set to 1, i.e. `domain.b[`u`]=1`.

To find the right neighbour of the point $T = (x)$ we simply mask out all bits up to (but not including) `domain.b[x]` and then find the left most bit which is set. The left neighbour is found in a similar manner. Algorithm 5.1[2], gives pseudo-code for this procedure. $\mathcal{QED}$

```
PROCEDURE SUneighbour (domain, T);
                                                          (* Right neighbour:  *)
  right:= ShiftRight (P, x);   (* P = 1*, this zeros the left most x bits (cf. eq. (4.4)), *)
  N→:= LMB (domain AND right);             (* left most bit by Algorithm 4.10.  *)
  left:= ShiftLeft (P, m-1-x);             (* Similarly, get the left neighbour. *)
  N←:= RMB (domain AND left);
  RETURN Closest (∞, 1, T, N←, N→)  (* Choose closer of two values, see eq. (5.6).  *)
END SUneighbour;
```

Algorithm 5.1: *Searching for the closest neighbour of point $T = (x)$ in an m-point universe* `domain`.

For the sake of notational simplification, Algorithm 5.1 uses a polymorphic procedure

$$\texttt{Closest}\ (f,\ d,\ T,\ T_1,\ \ldots,\ T_i,\ \ldots) \tag{5.6}$$

that returns the closest d-dimensional point T_i to the point T, under the metric $\delta_f(\cdot)$. We assume that the number of parameter points T_i is not fixed, but it is small (i.e. less than 5). We now extend Theorem 5.1 to a search over a large universe:

Theorem 5.2 *Let the size of the universe be at most $M = 2^m$ points and let $\mathcal{N}$ be a subset of that universe. Then there is an algorithm which finds the closest neighbour in $\mathcal{N}$ to a query point in constant time using $M + \frac{M}{\lg M} + O(\log M)$ bits of memory.*

Proof: We split the universe into $\frac{M}{m}$ m-point tiles, where a point $T = (x)$ lies on tile $x \operatorname{div} m$. Associated with each tile is a bit `B[i]`, which indicates whether the tile is nonempty. Nonempty tiles are then represented by bit maps `S[i]`, and the space of empty tiles is used to indicate the closest neighbour to the middle of the tile.[3] Thus, the point $T = (x)$ is mapped to a bit

$$\texttt{S}[x \operatorname{div} m]\texttt{.b}[x \bmod m] \tag{5.7}$$

and the space used for the data structure is $(m+1) \cdot \frac{M}{m} = M + \frac{M}{\lg M}$ bits.

To find the closest neighbour of T we examine the register `S[i]` (cf. eq. (5.7)), corresponding to the tile containing T, and the registers `S[i-1]` and `S[i+1]` corresponding

[2]The name `SUneighbour` stands for "*S*mall *U*niverse *neighbour*".

[3]Ties are broken arbitrarily.

to the tiles on either side.[4] From each register we determine the *candidate point* for the closest neighbour to T. If the tile is empty, the candidate is the point closest to the middle of a tile, otherwise it is the closest point to T in the tile. The solution to the closest neighbour query is the closest of these three, not necessarily distinct, candidate points. The correctness of this approach is immediate.

Algorithm 5.2 gives pseudo-code for finding the candidate points and Algorithm 5.3 gives the final solution although ignoring the trivial complication of T falling on an extremal tile (cf. footnote 4).

```
PROCEDURE SUneighbour (domain, bitMap, origin, T);
  IF bitMap THEN T:= SubCoordinates (T, origin);
    right:= GenerateMask (T);                              (* generate masks ...   *)
    left:= Negate (right);
    N→:= LMB (domain AND right);               (* ... for search in both directions *)
    N←:= RMB (domain AND left);
    RETURN AddCoordinates (origin, Closest (∞, 1, T, N←, N→))
  ELSE RETURN domain
  END
END SUneighbour;
```

Algorithm 5.2: *Generalized searching for the closest neighbour of the point T in a small universe.*

```
PROCEDURE Neighbour (T)
  i:= T DIV m;                                          (* First tile from eq. (5.7), *)
  origin:= i*m;
                                                        (* and local coordinates.  *)
  N0:= SUneighbour (S[i], B[i], origin, T);             (* Then search tile i ...  *)
  N1:= SUneighbour (S[i+1], B[i+1], origin+m, T);       (* ... and both ...        *)
  N2:= SUneighbour (S[i-1], B[i-1], origin-m, T);       (* ... neighbours.         *)
  RETURN Closest (∞, 1, T, N0, N2, N1);        (* Finally, select the closest point. *)
END Neighbour;
```

Algorithm 5.3: *Searching for the closest neighbour of the query point T in one dimension.*

Algorithm 5.2 actually extends Algorithm 5.1 and hence permits a more uniform treatment in higher dimensions. First, it establishes a *local coordinate system* with origin at the left most point of the tile. The location of the local origin in global coordinates is given by `origin`. The query point T is still given in the global coordinates, but it is translated into local ones if the tile is not empty and we search it. The translation between coordinate systems is done using procedures `SubCoordinates` and `AddCoordinates`.

[4] If $i = 0$ ($i = \frac{M}{m} - 1$) there is no left (right) tile, though.

Next, the query point need not lie in the tile. This affects mask generation and is taken into account by `GenerateMask` (otherwise based on eq. (4.4)). Also, it is not hard to see that the left mask can be generated by a negation of the right mask instead of using eq. (4.5). Finally, the parameter `bitMap` specifies whether the value of `domain` is a bit map or a pointer – i.e. whether the tile is non-empty. $\mathcal{QED}$

Algorithm 5.3 is easily modified to deal with a universe of size $S \leq m \cdot 2^m = M \lg M$ points. Reference to the closest neighbour of a centre of an empty tile is simply replaced by a reference to the tile containing that neighbour. The exact point is found using Algorithm 5.2 on that tile. Thus:

Corollary 5.1 *Let the size of a universe be S points, where $0 < S \leq m \cdot 2^m$ and let $\mathcal{N}$ be a subset of that universe. Then there is an algorithm which finds the closest neighbour to the query point in $\mathcal{N}$ in constant time using $S + \frac{S}{m} + O(m)$ bits of memory.*

Finally, note that this technique does not necessarily find the right (left) neighbour of T. However, by extending the size of a tile to $2m$ points, we can explicitly store the left and the right neighbours of the middle of an empty tile. Hence, this more general neighbour search problem, of finding either the left or right neighbour of a query point, can also be solved with essentially the same time and space bounds.

5.4 Two Dimensions

In this section we extend the results of the previous section to two dimensions. In one dimension, the distances between points (see eq. (5.5)) were the same for all norms. This is not true in higher dimensions and leads to our consideration of only L_∞ and L_1. Although the mapping

$$x' = \frac{y + x}{2} \qquad y' = \frac{y - x}{2} \tag{5.8}$$

of the point (x, y) under L_∞ into the point (x', y') under L_1 preserves the closest neighbourhood property ([75]), we present solutions under each of the norms separately to maintain the space bound of $M^2 + o(M^2)$ bits for the complete structure.

The next subsection gives some geometric background. This background is essentially norm independent, but most efficiently applied when considering the L_∞ and L_1 norms.

5.4.1 Circles and Some More Circles

First we extend our one dimensional tiles to the more natural two dimensional space (cf. cells in [17, 18, 19, 20]). We require tiles to have the following properties: they have to tile plane in a regular pattern such that from the coordinates of a point we can

efficiently compute the tile in which it lies; and if we put a circle under the relevant norm with diameter m anywhere on a plane, it must lie in $O(1)$ tiles. Obviously there are many different tilings which satisfy above conditions, but for the purpose of simplicity of explanation we choose to define:

Definition 5.2 *A* ***tiling polygon (tile)*** *is a regular polygon, which tiles the plane (see Figure 5.1). The tiles sharing a common edge with a given tile are its* ***direct neighbours****.*

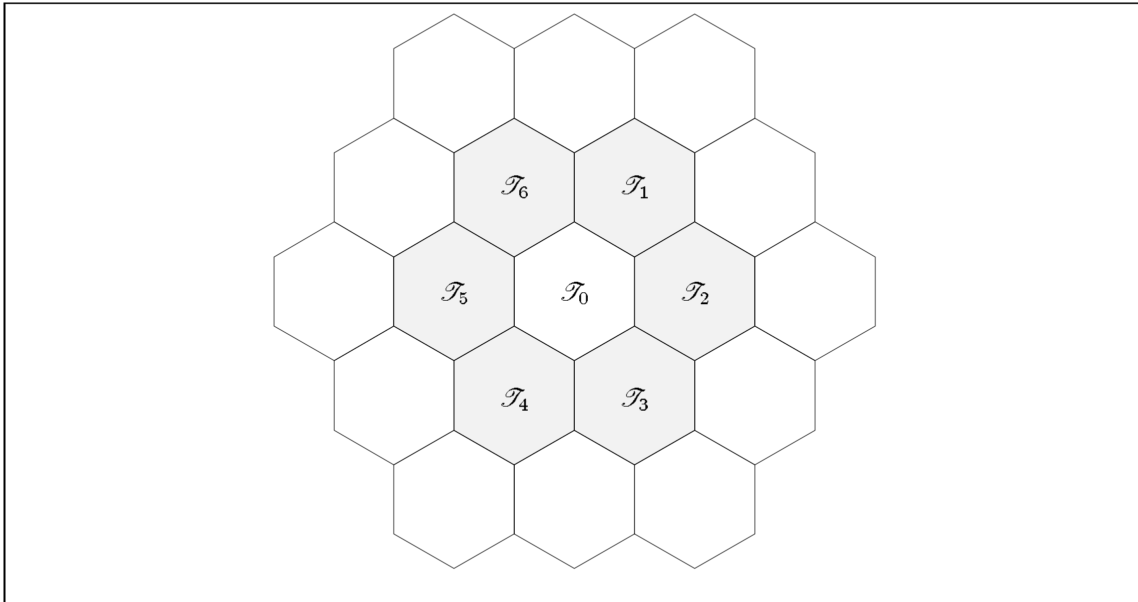

Figure 5.1: *Regular hexagons as tiling polygons.*

For convenience we number direct neighbours of a tile in a clockwise manner. Hence, in Figure 5.1, the direct neighbours of $\mathscr{T}_0$ are $\mathscr{T}_1$ through $\mathscr{T}_6$.

There are only 3 regular polygons which tile the plane: triangles, squares, and hexagons. In this section we use all of them to illustrate geometrical entities and their properties. However, later, in § 5.4.2 and in § 5.4.3, we will further restrict our attention to squares.

The circles are sets of points that are equidistant from some central point under the norm that is being used. Therefore:

Definition 5.3 *Let C_x be the middle of a tiling polygon $\mathscr{T}_x$,*[5] *then $\mathscr{C}_x$, the* ***empty circle*** *of $\mathscr{T}_x$, is the largest circle with centre C_x whose* ***interior*** *is empty. Thus, if N_x is the closest neighbour of C_x, N_x lies on the circumference of $\mathscr{C}_x$.*

[5] C_x need not be a point in the discrete domain.

This notion is illustrated in Figure 5.2. Here we consider the tiling triangle $\mathscr{T}_0$, with centre C_0, and its direct neighbours $\mathscr{T}_1$, $\mathscr{T}_2$ and $\mathscr{T}_3$. The closest neighbours of the middles of tiles $\mathscr{T}_1$, $\mathscr{T}_2$ and $\mathscr{T}_3$ are (respectively) N_1, N_2 and N_3. These infer the empty circles $\mathscr{C}_1$, $\mathscr{C}_2$ and $\mathscr{C}_3$ indicated in white.

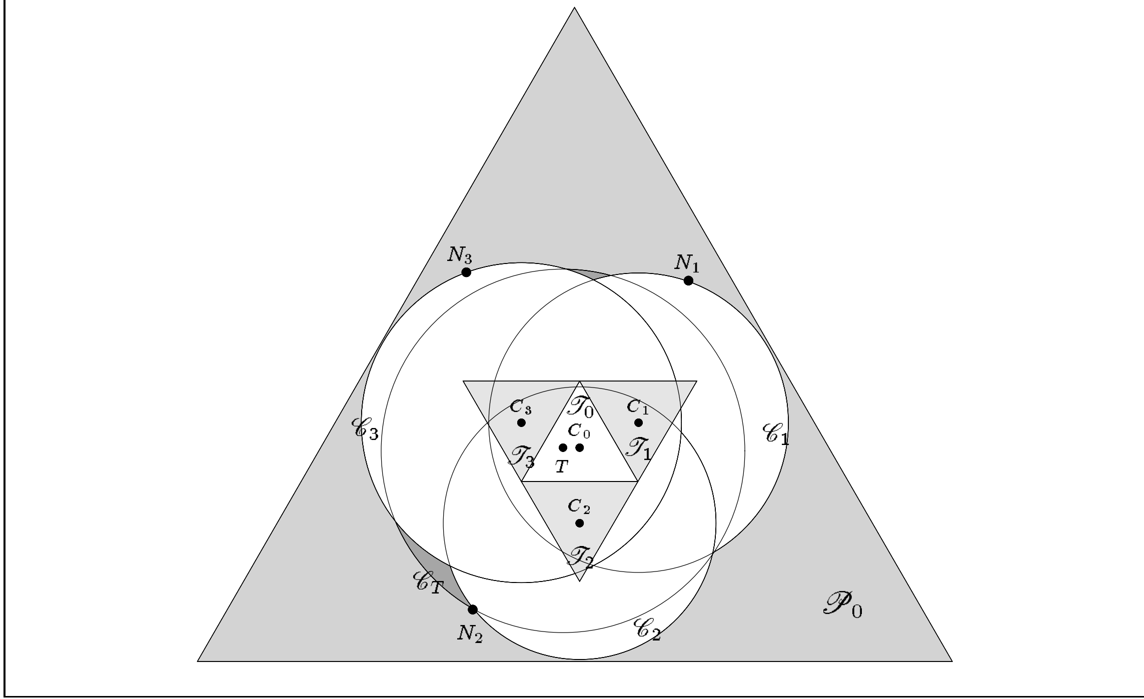

Figure 5.2: *Query point T, empty circles, a circle of candidates, and an enclosing polygon in two dimensions.*

Based on the definition of tiling polygons and empty circles we define an *enclosing polygon*:

Definition 5.4 *Let $\mathscr{T}_0$ be a polygon in a regular tiling of degree k and let $\{\mathscr{C}_i\}$ be the empty circles of its direct neighbours (respectively). Then $\mathscr{P}_0$, the **enclosing polygon** of $\mathscr{T}_0$, is the smallest polygon that has sides parallel to $\mathscr{T}_0$ and includes all empty circles (see the big shaded triangle in Figure 5.2).*

A particularly interesting part of the plane is the area which is inside the enclosing polygon, but outside the empty circles. In order to properly identify this area we define first a *wedge*:

Definition 5.5 *Let $\mathscr{T}_0$ be a tiling polygon, and $\mathscr{P}_0$ its enclosing polygon as above. Let $\mathscr{T}_i$ and $\mathscr{T}_j$ (where $j = (i \bmod k) + 1$) be direct neighbours of $\mathscr{T}_0$. Further, draw lines from C_0 through C_i, and from C_0 through C_j. Then the quadrilateral defined by these two lines and sides of the enclosing polygon is called a **wedge** of the enclosing polygon.*

In Figure 5.3 we have tiling triangles, and the wedge associated with C_3 and C_1 has a heavier line around it. Obviously, if we draw lines from C_0 through middles of all direct neighbours, we split the enclosing polygon into the same number of wedges as is the number of corners, that is the degree, of a tiling polygon.

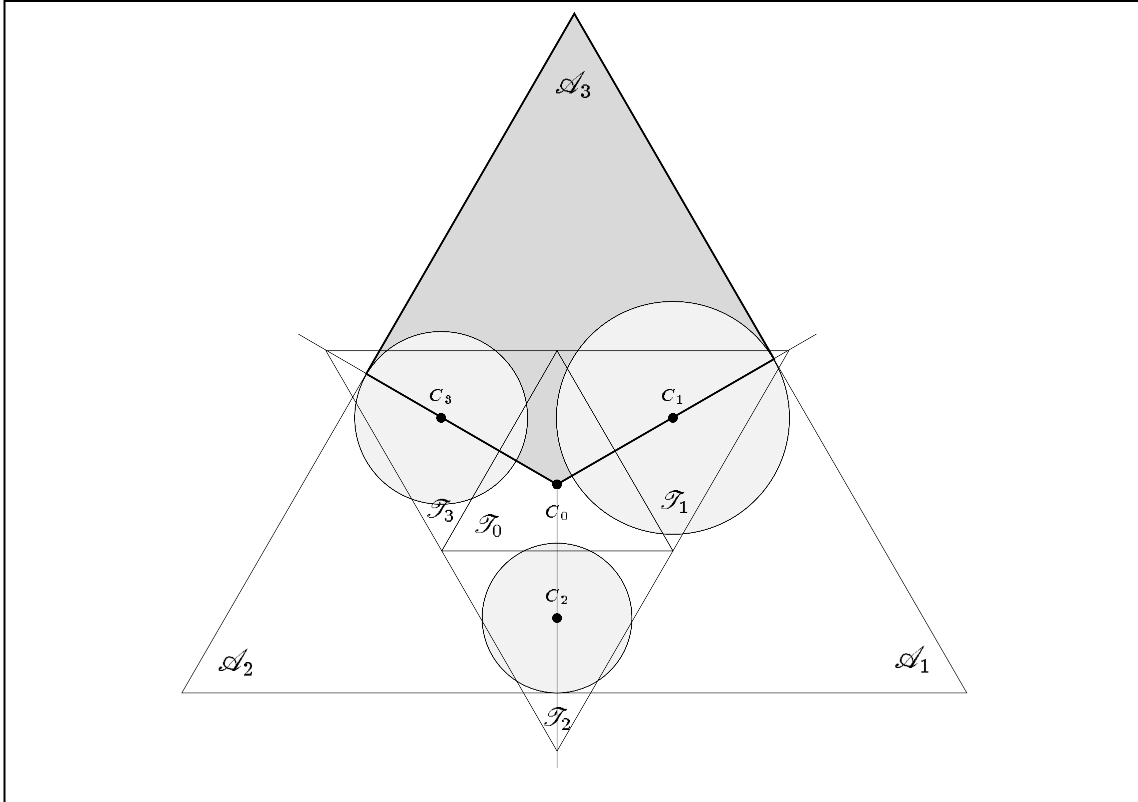

Figure 5.3: *Wedge and a corner area of a polygon as defined by empty circles $\mathscr{C}_1$ and $\mathscr{C}_3$.*

Inside the wedge we define a *corner area*:

Definition 5.6 *Consider the wedge defined by direct neighbours $\mathscr{T}_i$ and $\mathscr{T}_j$ as above. Then the area that lies inside the wedge and outside empty circles of all direct neighbours is called a* ***corner area*** *$\mathscr{A}_i$.*

In Figure 5.3 the dark shaded area is the corner area, $\mathscr{A}_3$. Since the number of corner areas is at most the number of wedges, which is itself at most k, it follows that:

Lemma 5.1 *If the tiling polygon has degree k, then there are at most k corner areas.*

The next term used in our discussion is the *circle of candidates*:

Definition 5.7 *Let the point T lie on the tiling polygon $\mathscr{T}_0$ of degree k, and let C_i $(0 < i \leq k)$ be middle points of respective direct neighbours with their closest neighbours N_i. Further, among all points N_i, let N_x be the closest point to T. Then the circle $\mathscr{C}_T$ with centre at T and N_x on its circumference is called the* ***circle of candidates****.*

As an illustration, see the small parts of the dark shaded circle of candidates in Figure 5.2. Observe there are at most k such regions.

Based on Definition 5.7 and Definition 5.3, we can restrict the location of the closest neighbour of a given point:

Lemma 5.2 *Let $\mathscr{T}_0$ be a tile of degree k and let T be a point inside it. Then the closest neighbour of T lies on the circumference or inside the circle of candidates $\mathscr{C}_T$, and outside the interior of the empty circles $\mathscr{C}_i$, where $0 < i \leq k$ (see dark shaded areas in Figure 5.2).*

Proof: By definition, the point N_x is on the circumference of the circle of candidates $\mathscr{C}_T$, and therefore the closest neighbour of T is either N_x itself or some other point, which is closer than N_x. However, all such points lie inside the circle of candidates. On the other hand, from Definition 5.3 we know that there is no point inside an empty circle. $\mathcal{QED}$

Lemma 5.2 concludes our brief geometrical excursion and hints at the idea behind our algorithm: compute empty circles of direct neighbours, compute the circle of candidates and search its intersection with the union of complements of empty circles. Later we will show that under the norms L_1 and L_∞ the intersection lies inside corner areas, and that the corner areas are small enough that we can perform an exhaustive search on them.

5.4.2 L_∞, or How Circles Became Squares

We explore the L_∞ norm where distance is defined by eq. (5.4), using $d = 2$. Under this norm, "circles" have a square shape, nevertheless, the results proven in § 5.4.1 still hold.

The solution for L_∞ is presented in two steps. First we show how to answer queries in constant time on a single tile. This result is later used for the general solution on a larger universe.

The Small Universe

The small universe is a square containing order m points. We represent it by a square register described in Definition 4.6. As in § 5.3, we map the point $T = (x_1, x_2)$ to a bit $\mathtt{b}[x_1, x_2]$ of the register and denote the point's presence or absence by setting the bit to 1 or 0 respectively.

The main parameter used in the definition of a square register is b, the number of bits in a register. In our case this is also the size of the small universe. Since in the big universe we require $2m$ bits to denote one point, it is convenient to have tiles of the same size, i.e. $2m$ points. Hence, we work with square registers of size

$$b = 2 \cdot m \tag{5.9}$$

bits. This sets the number of rows and columns in the register to $p = \sqrt{b} = \sqrt{2 \cdot m}$ (cf. eq. (4.11)).

The search algorithm is based on the idea of a search inside several (in this case 4) *search regions* (see Figure 5.4):

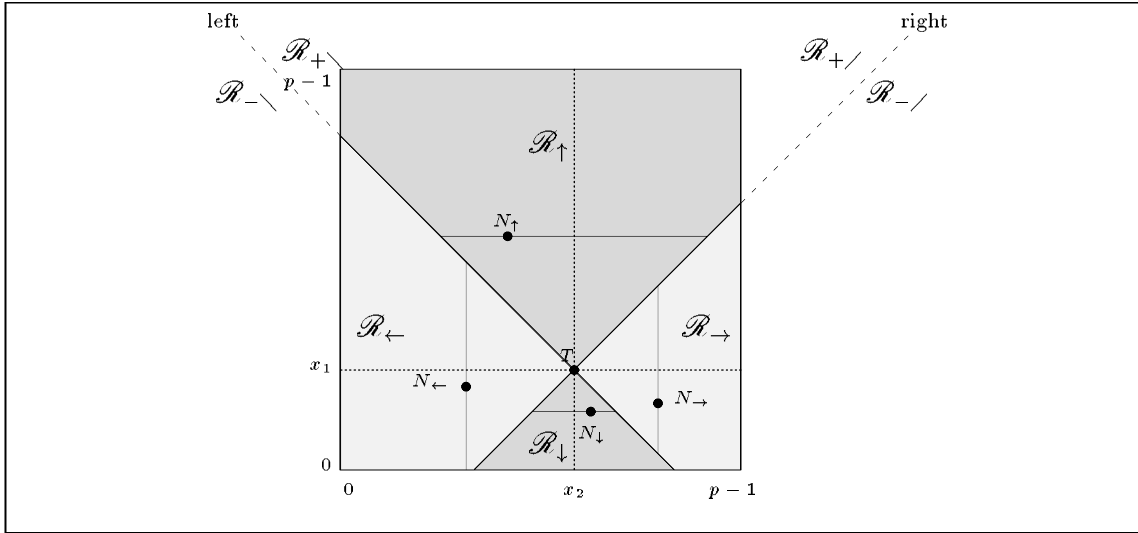

Figure 5.4: *Four search regions in a plane.*

Definition 5.8 *Let $T = (x_1, x_2)$ be a query point, at which a* ***left border line*** *and a* ***right border line****, with slopes $+45°$ and $-45°$ respectively, cross. These lines divide the plane into four* ***search regions*** *$\mathscr{R}_{\uparrow}$, $\mathscr{R}_{\rightarrow}$, $\mathscr{R}_{\downarrow}$, and $\mathscr{R}_{\leftarrow}$.*

In order to search one region at a time, we eliminate points from other regions. This requires that we generate proper masks:

Lemma 5.3 *Let the left and right border lines cross at point $T = (x_1, x_2)$. Then we can generate masks for all four search regions in constant time using $O(m)$ bits of space.*

Proof: Each border line splits the plane into a positive and a negative half-plane, where the negative half-plane lies below the respective border line. In Figure 5.4 the half-planes are denoted by $\mathscr{R}_{+\diagdown}$, $\mathscr{R}_{-\diagdown}$, $\mathscr{R}_{+\diagup}$ and $\mathscr{R}_{-\diagup}$.

Let us assume that we have masks available for half-planes $\mathscr{R}_{-\diagdown}$ and $\mathscr{R}_{-\diagup}$; then masks for the search regions can be computed using formulae

$$\begin{aligned}
\mathscr{R}_{\downarrow} &= \mathscr{R}_{-\diagdown} \wedge \mathscr{R}_{-\diagup} \\
\mathscr{R}_{\uparrow} &= \overline{\mathscr{R}_{-\diagdown} \vee \mathscr{R}_{-\diagup}} \\
\mathscr{R}_{\leftarrow} &= \overline{\mathscr{R}_{-\diagup} \vee \mathscr{R}_{\uparrow}} \\
\mathscr{R}_{\rightarrow} &= \overline{\mathscr{R}_{-\diagdown} \vee \mathscr{R}_{\uparrow}} \ .
\end{aligned} \tag{5.10}$$

It remains to generate masks $\mathscr{R}_{-\diagdown}$ and $\mathscr{R}_{-\diagup}$ efficiently. First observe these masks are, perhaps shifted, numbers $P^{\diagdown}$ and $P^{\diagup}$ respectively (see eq. (4.24)). More precisely, the right border line has equation $y = x - \Delta$, where $\Delta = x_1 - x_2$. Thus, to get $\mathscr{R}_{-\diagup}$ we have to shift $P^{\diagup}$ right for Δ columns, if $\Delta > 0$, and left for $-\Delta$ columns, if $\Delta < 0$. Similarly we get mask $\mathscr{R}_{-\diagdown}$. At this point we only assume that there exists procedure `GenerateMask` which implements the described mask generation, while its detailed description will be given later in a more general context (see Algorithm 5.11). Obviously, `GenerateMask` runs in constant time, and, thus, masks for all search regions can be generated in constant time using eq. (5.10). $\mathcal{QED}$

Using Lemma 5.3 we can easily prove:

Theorem 5.3 *Let the universe be a set of $b = 2m$ discrete points on a square grid and let $\mathcal{N}$ be a subset of that universe. Then there is an algorithm that finds the closest neighbour to a query point in $\mathcal{N}$ under the norm L_∞ in constant time using b bits for a data structure and $O(m)$ bits for internal constants.*

Proof: As a data structure representing the set we use the obvious bit map stored in a b-bit square register `domain`. The search algorithm divides the plane into four search regions from Definition 5.8. It then determines the closest point to the query point T in each region. Because of the norm we are using (cf. eq. (5.4)) this amounts, for $\mathscr{R}_{\downarrow}$ and $\mathscr{R}_{\uparrow}$, to finding the point in the row closest to T and, for $\mathscr{R}_{\leftarrow}$ and $\mathscr{R}_{\rightarrow}$, to finding the point in the closest column (see Figure 5.4). Since the universe is represented by a square register `domain`, we can employ extremal set bits searching algorithms from § 4.7 (see Theorem 4.3). This leads us to the full algorithm as represented at the schematic level in Algorithm 5.4. $\mathcal{QED}$

Another approach to finding the extremal set bits in a register is to employ table lookup. Clearly a table of all possible register values would have M^2 entries, too many for our purposes. However, if we divide a register into $\frac{1}{\epsilon}$ disjoint pieces, and build a table of all possible values any of these pieces can have, then the space required will be $O(M^{1-\epsilon} \log^{(2)} M)$ bits. Searches still require only a constant time.

The Big Universe

Algorithm 5.4 allows us to search quickly for the closest neighbour under the L_∞ norm in a square universe of a size up to that of an arithmetic register. Using ideas similar to

```
PROCEDURE SUneighbour (domain, bitMap, origin, T);
  IF bitMap THEN T:= SubCoordinates (T, origin);
    𝓡_\ := GenerateMask (T, '\');               (* First, masks for both half-planes *)
    𝓡_/ := GenerateMask (T, '/');                      (* using Algorithm 5.11, *)
    𝓡↓:= 𝓡_\ AND 𝓡_/;                (* and then masks for all search regions ... *)
    𝓡↑:= Negate (𝓡_\ OR 𝓡_/);                         (* ... using eq. (5.10). *)
    𝓡← := Negate (𝓡_/ OR 𝓡↑);  𝓡→ := Negate (𝓡_\ OR 𝓡↑);
                                            (* Next, search for points in search regions *)
    N↓:= TMBofRectReg (domain AND 𝓡↓, p, p);
    N← := RMBofRectReg (domain AND 𝓡←, p, p);
    N→ := LMBofRectReg (domain AND 𝓡→, p, p);
    N↑:= BMBofRectReg (domain AND 𝓡↑, p, p);
                                   (* and, finally, the closest among the found points. *)
    RETURN AddCoordinates (origin, Closest (∞, 2, T, N↓, N←, N↑, N→))
  ELSE RETURN domain
  END;
END SUneighbour;
```

Algorithm 5.4: *General version of a search for the closest neighbour of T in a small universe under the norm L_∞.*

those of § 5.3, we extend this result to the universe exponential in the register size, that is to $M \times M$ points where $M = 2^m$.

Most of the discussion in § 5.4.1 was based on a notion of a tiling polygon. The tiling polygons we use here are squares of size $p \times p = b$ points ($b = 2m$, cf. eq. (5.9)). The sides of the tile are parallel to the axes of the coordinate system. This also implies the orientation and shape of an enclosing polygon $\mathscr{P}_0$: $\mathscr{P}_0$ is a rectangle with sides parallel to the axes of the coordinate system (cf. Definition 5.4).

A "circle", the locus of all points equidistant from a given point, is, under the L_∞ norm, in fact a square with sides parallel to coordinate axes. Hence the empty circles and the circle of candidates are indeed squares. Moreover, circles, tiles, and enclosing polygons all have parallel sides.

The remaining entities of interest are corner areas which, by Definition 5.6, consist of the area inside an enclosing polygon and outside empty circles. Under the L_∞ norm, the corner areas have the following important property:

Lemma 5.4 *Let $\mathscr{T}_0$ be some tiling square. Then there are at most four corner areas associated with $\mathscr{T}_0$, each of which lies in at most six tiles.*

The most important consequence of Lemma 5.4 is that corner areas can be exhaustively searched in constant time using Algorithm 5.4.

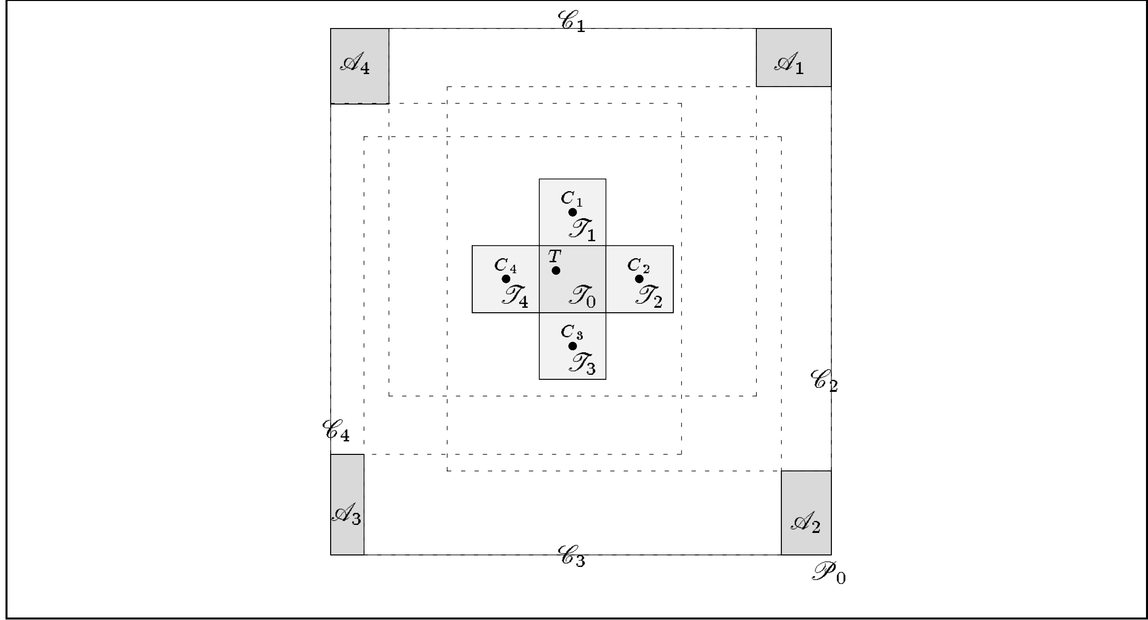

Figure 5.5: *Query point T on a tile $\mathscr{T}_0$ with its direct neighbour tiles $\mathscr{T}_1$, $\mathscr{T}_2$, $\mathscr{T}_3$ and $\mathscr{T}_4$, corresponding empty circles $\mathscr{C}_1$, $\mathscr{C}_2$, $\mathscr{C}_3$ and $\mathscr{C}_4$, an enclosing rectangle $\mathscr{P}_0$, and corner areas $\mathscr{A}_1$, $\mathscr{A}_2$, $\mathscr{A}_3$ and $\mathscr{A}_4$.*

Proof: By Lemma 5.1 there are at most four corner areas (cf. Figure 5.5). Without loss of generality we confine our attention to $\mathscr{A}_1$ (see Figure 5.6). Let a_1 and a_2 be the radii of $\mathscr{C}_1$ and $\mathscr{C}_2$ respectively. First, assume that either a_1 or a_2 is at least p and let the sides of $\mathscr{A}_1$ be of lengths u and v (see the left diagram in Figure 5.6). Then, since the distance from C_2 to the top side of $\mathscr{P}_0$ is $v + a_2 = p + a_1$ and since the distance from C_1 to the right side of $\mathscr{P}_0$ is $u + a_1 = p + a_2$, $u + v = 2p$ and, consequently, $0 \leq u, v \leq 2p$. Furthermore, the area of $\mathscr{A}_1$ is $u \cdot v \leq p^2 = b$. Thus, $\mathscr{A}_1$ lies on at most 6 tiling squares.

On the other hand, if $a_1, a_2 \leq p$, then $\mathscr{A}_1$ lies on both of the tiles that are adjacent to $\mathscr{T}_1$ and $\mathscr{T}_2$ (the right diagram in Figure 5.6). It is not hard to verify that $\mathscr{A}_1$ lies on at most three other tiles and that this occurs when $p \geq a_1, a_2 \geq \frac{p}{2}$. $\mathcal{QED}$

The next property relates the circle of candidates and the enclosing polygon:

Lemma 5.5 *Under the norm L_∞, when tiles are squares aligned with coordinate axes, the circle of candidates lies inside the enclosing polygon.*

Proof: Let the middle of tile $\mathscr{T}_0$ be point $C_0 = (0, 0)$ and let $T = (x_T, y_T)$ be a query point where $-\frac{p}{2} < x_T, y_T < \frac{p}{2}$. Furthermore, let the radius of the circle of candidates be $r_T = \min_{1 \leq i \leq 4} \delta_\infty(T, N_i)$, where N_i are closest neighbours of middles of direct neighbours. Thus, we have to show, for all points U on the circumference of the enclosing polygon $\mathscr{P}_0$, that $\delta_\infty(T, U) \geq r_T$.

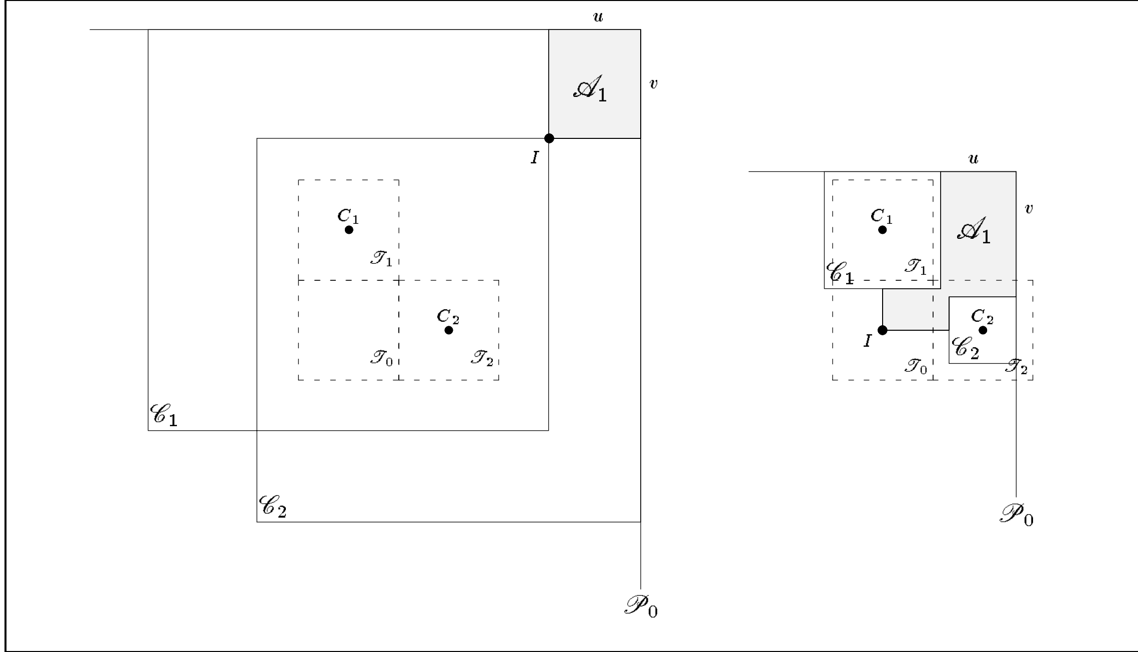

Figure 5.6: *Corner area $\mathscr{A}_1$ limited by a distance between centres of empty circles $\mathscr{C}_1$ and $\mathscr{C}_2$.*

Without loss of generality, we may assume that the closest point to T on the circumference of $\mathscr{P}_0$ is $W = (x_W, y_W)$ where $y_W = y_T$. Simultaneously, by the definition of the enclosing polygon (Definition 5.4) x_W is either $-\delta_\infty(C_4, N_4) - p$ or $\delta_\infty(C_2, N_2) + p$. Further, again without loss of generality we may assume $x_W = -\delta_\infty(C_4, N_4) - p$ and thus

$$\delta_\infty(T, W) = |x_T - p - \delta_\infty(C_4, N_4)| = |x_T - p| + \delta_\infty(C_4, N_4) \ . \tag{5.11}$$

However, since tile $\mathscr{T}_4$ is immediately to the left of $\mathscr{T}_0$, $C_4 = (-p, 0)$ and hence $\delta_\infty(T, C_4) = |x_T - p|$. Therefore, using eq. (5.11) and a triangle inequality (5.3), we get

$$\delta_\infty(T, W) = \delta_\infty(T, C_4) + \delta_\infty(C_4, N_4) \geq \delta_\infty(T, N_4) \geq \min_{1 \leq i \leq 4} \delta_\infty(T, N_i) = r_T \ .$$

$\mathcal{QED}$

In the rest of this section we prove:

Theorem 5.4 *Let the universe be a set of $M \times M$ discrete points on a square grid and let $\mathcal{N}$ be a subset of that universe. Then there is an algorithm which finds the closest neighbour in $\mathcal{N}$ to a query point under the norm L_∞ in constant time using $M^2 + \frac{M^2}{2 \lg M} + O(\log M)$ bits of memory.*

Proof: We tile universe with an $\frac{M}{p} \times \frac{M}{p}$ array of b-point square tiles with origin in the bottom left. Therefore point $T = (x, y)$ falls on a tile, `tile`, with coordinates

$$\texttt{tile[1]} = (x \operatorname{div} p) \qquad \text{and} \qquad \texttt{tile[2]} = (y \operatorname{div} p) \ . \tag{5.12}$$

As in Algorithm 5.3, each tile is associated with a bit `B[tile]`. This bit indicates whether the tile is nonempty. If the tile is nonempty it is represented by a bit map stored in `S[tile]`, and if it is empty `S[tile]` stores the coordinates of the closest neighbour to the centre of the tile. Each of the arrays `S[.]` and `B[.]` has $(\frac{M}{p})^2$ entries, and thus the whole data structure occupies $(\frac{M}{p})^2 \cdot b + (\frac{M}{p})^2 = M^2 + \frac{M^2}{2 \lg M}$ bits of space ($b = p^2$, cf. eq. (5.9)).

According to Lemma 5.2, to find the closest neighbour of T we search that part of the interior of the circle of candidates $\mathscr{C}_T$ which is outside the empty circles $\mathscr{C}_1$, $\mathscr{C}_2$, $\mathscr{C}_3$, and $\mathscr{C}_4$. By Lemma 5.5, $\mathscr{C}_T$ lies inside the enclosing polygon, and thus it is sufficient to search the corner areas. Furthermore, according to Lemma 5.4, each corner area overlaps at most six tiles. Since by Theorem 5.4 each tile can be searched in constant time, the closest neighbour can be found in constant time.

In the rest of the proof we describe the searching algorithm in a greater detail. First, we introduce Algorithm 5.5, which exhaustively searches the interior of $\mathscr{C}_T$ in one corner area. The circle of candidates, $\mathscr{C}_T$, is defined by the query point T and its closest neighbour found so far, N_T.[6] Further, the corner area is defined by the tile `tile`, and the direction, Δ, into which the area "extends" from `tile`. The direction Δ also determines the order in which the corner area is searched. Finally, to test when the search in the area reaches the border of $\mathscr{C}_T$ the function `Outside` is used. Its implementation is left out, but the function obviously runs in constant time and $O(m)$ bits of space.

```
PROCEDURE SearchCorner (tile, Δ, T, N_T);
  columnStart:= tile[1];                          (* Starting tile in each row.   *)
  WHILE (NOT Outside (T, N_T, tile)) DO              (* Then, search by rows *)
    WHILE (NOT Outside (T, N_T, tile)) DO         (* and in each row by columns:  *)
      origin:= (tile[1]*p, tile[2]*p);            (* origin of a tile by eq. (5.12), *)
      N_T:= Closest (∞, 2, T,                    (* search it using Algorithm 5.4, and *)
                N_T, SUneighbour (S[tile], B[tile], origin, T));
      tile[1]:= tile[1] + Δ[1]                     (* go to the next column.   *)
    END;
    tile:= (columnStart, tile[2]+Δ[2]);
  END;
  RETURN N_T;
END SearchCorner;
```

Algorithm 5.5: *Searching for the closest neighbour of T in a corner area starting from* `tile` *under the norm L_∞.*

For search of an individual tile, Algorithm 5.4 is used. If the tile to be searched is empty, the algorithm returns the closest neighbour of the centre of such a tile instead

[6] This circle of candidates is not necessarily the same as in Definition 5.7, because it might shrink due to updates of N_T. However, this does not affect the correctness of Lemma 5.2.

of the closest neighbour of the query point in the tile. However, it is easy to see that this difference has no influence on the correctness of Algorithm 5.5. Finally, the most "expensive" are searches of non-empty tiles (see again Algorithm 5.4). By a proper ordering of how tiles are being searched, we can reduce the number of such searches to at most *three* per corner area. Thus, in total at most 12 non-empty tiles are searched.

Finally, Algorithm 5.6 presents a pseudo-code of the complete searching procedure though ignoring the trivial complication if T falls on an extremal tile. Note, if direct neighbours of $\mathscr{T}_0$ are not empty Algorithm 5.6 does not exactly construct their respective empty circles, but this has no influence on the correctness or running time of the algorithm. Algorithm 5.6 uses function `Intersect` to compute the tile, `tile`, on which lies I, the "intersection" of the empty circles $\mathscr{C}_i$ and $\mathscr{C}_j$ (cf. Figure 5.6). The first two pairs of parameters of `Intersect` define $\mathscr{C}_i$ and $\mathscr{C}_j$ (tile on which lies the centre of the respective circle and the point on the circle's circumference), while the last parameter determines which of two possible intersections of the circles is needed. Although the implementation of `Intersect` is left out, it obviously runs in constant time and $O(m)$ bits. $\mathcal{QED}$

```
PROCEDURE Neighbour (T);
  T_0:= ((T[1] DIV p), (T[2] DIV p));             (* First, using eq. (5.12) get tile T_0  *)
  T_1:= (T_0[1], T_0[2]+1); T_2:= (T_0[1]+1, T_0[2]);   (* and its direct neighbours.   *)
  T_3:= (T_0[1], T_0[2]-1); T_4:= (T_0[1]-1, T_0[2]);
                         (* Next, circles are implicitly defined by the closest neighbours: *)
  N:= (∞, ∞);                                 (* the initial circle of candidates C_T; *)
  FOR i:= 1 TO degree DO origin:= (T_i[1]*p, T_i[2]*p);                        (* all *)
    N_i:= SUneighbour (S[T_i], B[T_i], origin, T);            (* empty circles C_i, and *)
    N:= Closest (∞, 2, T, N, N_i);                                    (* updated C_T.  *)
  END;
                                                       (* Finally, search corner areas:  *)
  Δ_1:= (-1, +1); Δ_2:= (+1, +1);                          (* directions in which ...   *)
  Δ_3:= (+1, -1); Δ_4:= (-1, -1);                          (* ... corner areas extend; *)
  FOR i:= 1 TO degree DO j:= (i MOD degree) + 1;           (* search corner area A_i,  *)
    tile:= Intersect (T_i, N_i, T_j, N_j, i);              (* which starts on tile.    *)
    N:= Closest (∞, 2, T, N, SearchCorner (tile, Δ_i, T, N));
  END;
  RETURN N
END Neighbour;
```

Algorithm 5.6: *Searching for the closest neighbour of the query point T under the norm L_∞.*

A couple of trivial improvements further reduce the running time of Algorithm 5.6. First, in the initial approximation of the circle of candidates, $\mathscr{C}_T$, we can use information about tile $\mathscr{T}_0$ stored in S[$\mathscr{T}_0$], instead of putting the point N to the infinity (i.e. N:= `SUneighbour (S[`$\mathscr{T}_0$`], B[`$\mathscr{T}_0$`], origin, `T`)`). Secondly, if any empty circle has a diameter smaller than p, we search tile $\mathscr{T}_0$ several times (cf. the right diagram in Figure 5.6) and this can be avoided.

5.4.3 L_1, or How Circles Became Diamonds

Under the norm L_1, the distance between points $T_1 = (x_1, y_1)$ and $T_2 = (x_2, y_2)$ is defined as

$$\delta_1(T_1, T_2) = |x_1 - x_2| + |y_1 - y_2| \ . \tag{5.13}$$

As before, we first describe a solution for a small universe and then extend it to the more general case.

As previously noted (see eq. (5.8)) there is a direct mapping between the norms L_1 and L_∞, which preserves the closest neighbourhood property. However, this mapping expands the domain, and so we opt to build our solution from scratch to achieve the space bound $M^2 + o(M^2)$ bits.

The Small Universe

The solution for a small universe is presented in two parts: the representation of the universe, and the searching algorithm. Under the L_∞ norm, circles have a square shape which made the square tiling of the plane very convenient. Under the L_1 norm we have a similar, though not quite as convenient situation in that circles are now *diamonds* (squares rotated 45° from the axes). This is also the shape of our small, b-point ($b = 2m$, cf. eq. (5.9)) universe (see the diamond shaped area in Figure 5.7).

The universe lies in a plane with the *original coordinate system*, whose axes, x and y, run on the bottom and the left sides respectively. As before, inside the small universe we have a *local coordinate system* which consists of rows and columns (r and c respectively). The local system is rotated clockwise 45° and shifted p points right, where $p = \frac{\sqrt{2b}}{2} = \sqrt{m}$ (see Figure 5.7). This puts the point (x, y) of the original systems at the point (c, r) in the local system where

$$\begin{aligned} r &= (x - (p-1) + y) \bmod 2p &&= (x + y + (p+1)) \bmod 2p \\ c &= (-(x - (p-1)) + y) \bmod 2p &&= (-x + y + (p-1)) \bmod 2p \ . \end{aligned} \tag{5.14}$$

In the mapping we omit a $\frac{\sqrt{2}}{2}$ factor (cf. [25, pp. 227–228] and [75]), but introduce an additional "mod" operation, the reason for which will become evident later. Note that adjacent points in Figure 5.7 differ in both, c and r, coordinates and that the coordinates along the columns (rows) of local system go up by 2's.

The range of mapping (5.14) is $4p^2 = 2b$, since $0 \leq r < 2p$ and $0 \leq c < 2p$. Thus, half of points, call them *empty points*, in the local system are not images of any point from the original system. Therefore, we do not need to define the inverse mapping for these points either. Finally, observing that empty points are in odd columns of even rows and in even columns of odd rows, and using eq. (5.14) we get the mapping

$$x = \frac{c - r}{2} + (p - 1) \qquad \text{and} \qquad y = \frac{r + c}{2} \tag{5.15}$$

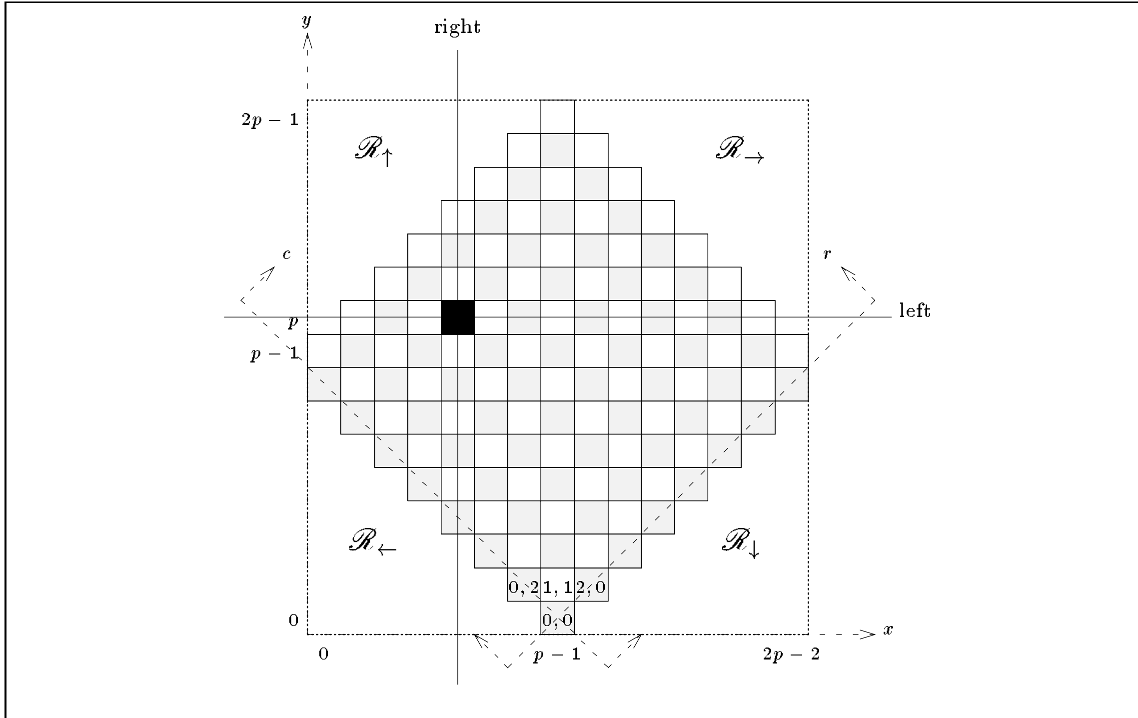

Figure 5.7: *Small universe under the norm L_1.*

of the point (c, r) in the local system to the point (x, y) in the original system.[7]

As before, we represent a small universe by a bit map:

Lemma 5.6 *Let the small universe lie in original coordinate system as above. Then there is a b-bit bit map representation of the small universe in which the mapping between the bits of the bit map and the points in the universe is computable in constant time in each direction.*

Proof: The mapping between the bit map and the original coordinate system is done via the local coordinate system, which is, however, never materialized. Therefore we do not need to store empty points. This halves the amount of necessary space.[8] The bit map is stored in a pair of m-bit square registers `domain[.]` where the even rows are stored in the first register and the odd ones in the second one (leaving out the empty points). In detail, the point (c, r) of the local coordinate system is represented by a bit

$$\texttt{domain[}r\ \texttt{MOD 2].b[}c\ \texttt{DIV 2, }r\ \texttt{DIV 2]}\ , \tag{5.16}$$

[7] Omission of factor $\frac{\sqrt{2}}{2}$ in eq. (5.14) not only introduces empty points, but also puts non-empty points on a discrete grid.

[8] If we had used the mapping in eq. (5.8) we would have to store empty points as well.

while the bit `domain[k].b[i,j]` represents a point

$$(2 \cdot i + k, \quad 2 \cdot j + k) \ . \tag{5.17}$$

Finally, since both these mappings and the mappings between the local and original coordinate systems (see eq. (5.14) and eq. (5.15)) can be computed in constant time, the mappings between the bit map representation and the original coordinate system can be computed in constant time as well. $\mathcal{QED}$

Figure 5.8 illustrates the mapping between the local coordinate system and bit map representation stored in registers `domain[0]` and `domain[1]` (the left and the right diagram respectively[9]). The inner numbers in each diagram indicate local coordinates and the outer ones bit indices in registers. Further, the query point, the black square in Figure 5.7, is mapped in a bit only in one register (the black square in the left diagram and the black dot in the right one).

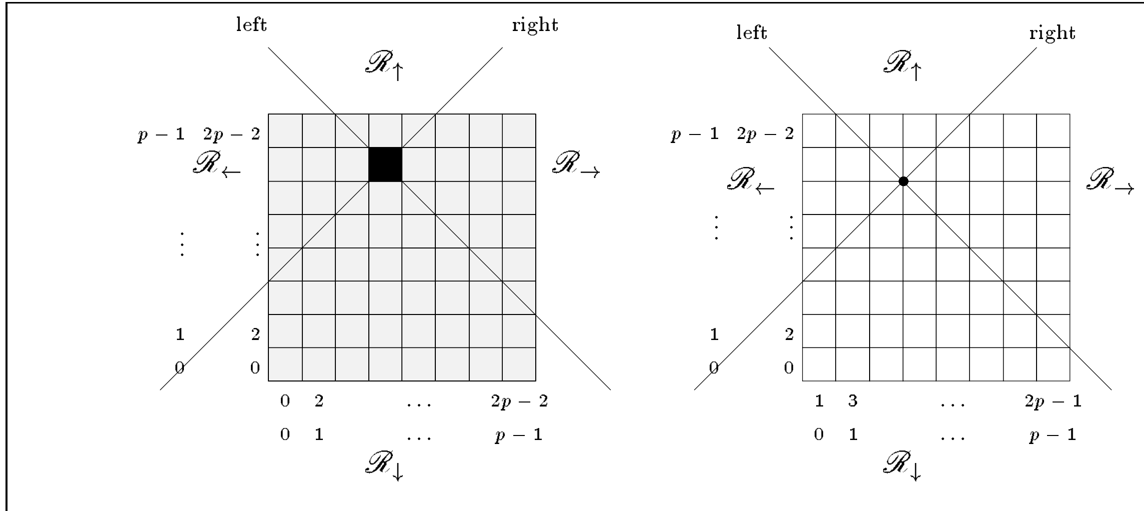

Figure 5.8: *Mapping of points from the local coordinate system into bits of a register pair.*

The following lemma relates the search for the closest neighbour in the original system to one in the local coordinate system:

Lemma 5.7 *The circle under the norm L_1 in the original coordinate system maps bijectively to the circle under the norm L_∞ in the local coordinate system.*

Proof: Trivially from eq. (5.15) and eq. (5.14). $\mathcal{QED}$

Lemma 5.7 essentially says that the search for the closest neighbour in the original coordinate system under the L_1 norm is equivalent to the search for the closest neighbour

[9] The left diagram is intentionally shaded to show a clear connection with shaded points in Figure 5.7.

in the local coordinate system under the L_∞ norm. Thus, as under L_∞, we define left and right border lines (cf. Figure 5.4), which, in turn, split the universe into four search regions $\mathscr{R}_\uparrow$, $\mathscr{R}_\rightarrow$, $\mathscr{R}_\downarrow$, $\mathscr{R}_\leftarrow$ (see Figure 5.8 and Figure 5.7). Now, to find the closest neighbour, we search under the norm L_∞ each of this regions separately.

Theorem 5.5 *Let the universe be a set of $b = 2m$ discrete points on a diamond shaped grid (see Figure 5.7), and let $\mathcal{N}$ be a subset of that universe. Then there is an algorithm which finds the closest neighbour in $\mathcal{N}$ to a query point under the norm L_1 in constant time using b bits for the data structure and $O(m)$ bits for internal constants.*

Proof: We represent $\mathcal{N}$ by a b-bit bit map stored in m-bit registers `domain[0]` and `domain[1]` (see Lemma 5.6).

Next, by Lemma 5.7 the search for the closest neighbour under the norm L_1 in original coordinates is equivalent to the search under the norm L_∞ in local coordinates. Thus, to find the closest neighbour we search each register `domain[.]` separately using Algorithm 5.7 – a version of Algorithm 5.4. In the algorithm the indices of the bit, `bit`, whose closest set bit we are looking for, are not restricted to integers. However, the sums and differences of the indices have to be integers to make the generated masks correct. For example, in the left and right diagrams of Figure 5.8 the indices have values $(3, p-2)$ and $(2.5, p-2.5)$ respectively. Because of correctness of Algorithm 5.4, the whole algorithm is correct.

```
PROCEDURE SearchRegister (register, bit);
  𝓡_\ := GenerateMask (bit, '\');                        (* First, masks for both half-planes, *)
  𝓡_/ := GenerateMask (bit, '/');
  𝓡↓ := 𝓡_\ AND 𝓡_/;  𝓡↑ := Negate (𝓡_\ OR 𝓡_/);     (* and for search regions.   *)
  𝓡← := Negate (𝓡_/ OR 𝓡↑);  𝓡→ := Negate (𝓡_\ OR 𝓡↑);
  N↓ := TMBofRectReg (register AND 𝓡↓, √m, √m);         (* Next, search for points ...   *)
  N← := RMBofRectReg (register AND 𝓡←, √m, √m);            (* ...in search regions,   *)
  N→ := LMBofRectReg (register AND 𝓡→, √m, √m);
  N↑ := BMBofRectReg (register AND 𝓡↑, √m, √m);
  RETURN Closest (∞, 2, bit, N↓, N←, N↑, N→)             (* and return the closest.   *)
END SearchRegister;
```

Algorithm 5.7: *Searching for the closest set bit to the* `bit` *in the square* `register` *under the norm L_∞.*

Finally, Algorithm 5.8 gives a pseudo-code of complete searching algorithm. It obviously runs in constant time and uses a b-bit bit map as a data structure and additional $O(m)$ bits for internal constants. $\mathcal{QED}$

We left out the implementation of functions for mapping from the original to the local coordinate systems, `Orig2Local`, and from indices of bits in bit maps back to the original system, `BMindex2Orig`. They obviously run in constant time by Lemma 5.6.

```
PROCEDURE SUneighbour (domain, bitMap, origin, T);
  IF bitMap THEN T:= SubCoordinates (T, origin);     (* In domain is a bit map, so *)
    local:= Orig2Local (T);             (* switch to local coordinates by eq. (5.14).  *)
                                                     (* Then search in even rows:  *)
    indices:= (local[1]/2, local[2]/2);  (* get indices of searched bit by eq. (5.16), *)
    N_E:= BMindex2Orig (SearchRegister (domain[0], indices));      (* and search; *)
                                                     (* and similarly in odd rows.  *)
    indices:= AddCoordinates (indices, (-1/2, -1/2));
    N_O:= BMindex2Orig (SearchRegister (domain[1], indices));
                                      (* Finally, return the closer of two found points.  *)
    RETURN AddCoordinates (origin, Closest (1, 2, T, N_E, N_O)
  ELSE RETURN domain
  END;
END SUneighbour;
```

Algorithm 5.8: *Searching for the closest neighbour of the query point T in a small domain under the norm L_1.*

The Big Universe

The big universe consists of $M \times M$ points where $M = 2^m$. We tile it with b-point diamond tiles (squares rotated 45°). The searching algorithm takes the same approach as the one for the norm L_∞: first, construct empty circles of direct neighbours and circle of candidates; next, compute corner areas; and, finally, exhaustively search that part of the corner areas which is inside the circle of candidates. Obviously, search inside a tile is done using Algorithm 5.8.

Before going into the details of the searching algorithm, we address the tiling and in particular the mapping from the point $T(x, y)$ into a data structure. As shown in Figure 5.9, the tiles are positioned in *even* and *odd* rows. Each set of rows is numbered separately by a pair of indices (i, j) starting at the bottom left corner. A tile has its own original and local coordinate systems, and an *origin*. The *origin* of a tile is the point in which we have to put point $(0, 0)$ of the original coordinate system to bring the point $(0, 0)$ of the local coordinate system to a point $(p-1, 0)$ in the original system. For example, in Figure 5.7 the point $(0, 0)$ is itself the origin of a tile; while in Figure 5.9 the point in the bottom left corner of the shaded square (near the "3") is the origin of the tile marked 0. This convention permits us to search individual tiles using results from the previous section in combination with the following lemma:

Lemma 5.8 *Given the coordinates (x, y) of a point T, we can in constant time determine the tile on which it lies and coordinates of this tile's origin.*

Proof: We split the universe into $2p$-point stripes, which form a $\frac{M}{2p} \times \frac{M}{2p}$ mesh of squares (see the shaded square in Figure 5.9). Each square consists of five regions also

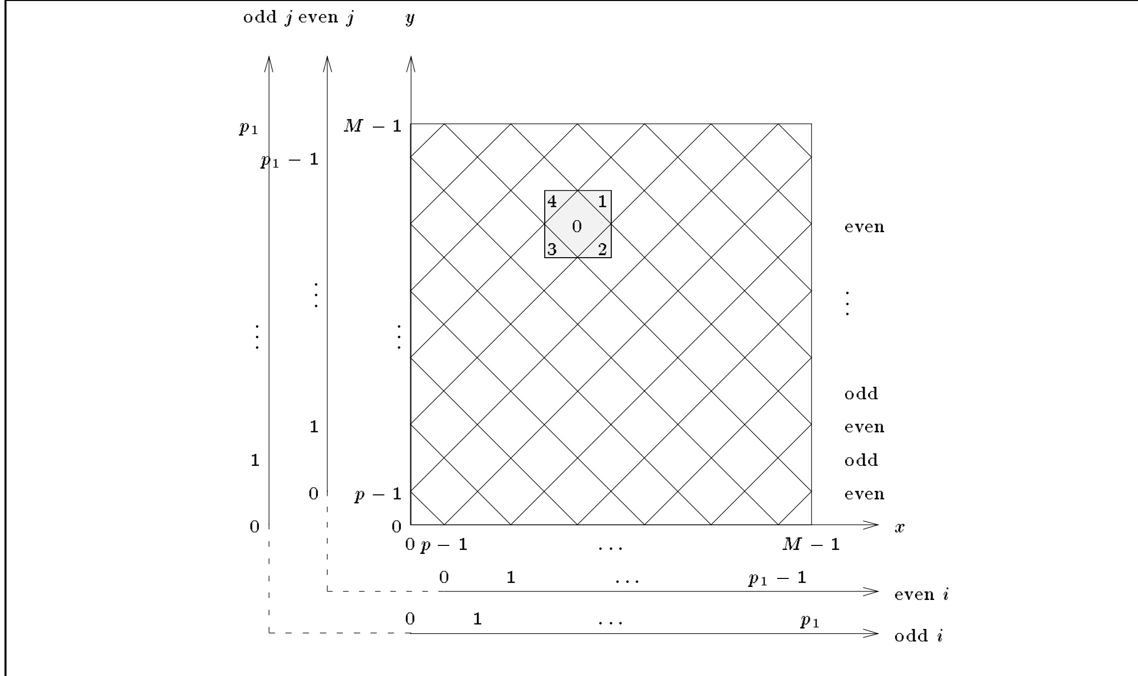

Figure 5.9: *Big universe tiled by diamonds.*

shown in Figure 5.9. The region 0 is an even row tile with the origin in the bottom left corner of the square. The other regions are parts of odd row tiles which have their origins properly shifted. Further details are presented in Algorithm 5.9, which computes the tile and the coordinates of its origin through a simple four-case analysis. Obviously, the algorithm runs in constant time and uses $O(m)$ bits of space for constants. $\mathcal{QED}$

Under the L_1 norm, circles are diamonds and have, as under the L_∞ norm, sides parallel to the tiles. This property limits the size of corner areas, and, hence, permits us to perform an exhaustive search (cf. Lemma 5.4):

Lemma 5.9 *Let $\mathscr{T}_0$ be some tiling diamond. Then there are at most four corner areas associated with $\mathscr{T}_0$ and each of them lies on at most six tiles.*

In the proof we refer to a general arrangement of objects shown in Figure 5.10.
Proof: By Lemma 5.1 there are four corner areas and to limit the size of each of them we consider two general cases shown in Figure 5.11 (cf. Figure 5.6). The lemma easily follows for the right case using similar reasoning as in the proof of Figure 5.6.

For the left case we use Pythagoras' theorem and the definition of tiling to get the distance between C_4 and C_1 (centres of tiling diamonds $\mathscr{T}_4$ and $\mathscr{T}_1$ respectively) $2p = \frac{u}{\sqrt{2}} + \frac{v}{\sqrt{2}}$. Therefore, $0 \leq u, v \leq 2\sqrt{2} \cdot p < 3p$. Furthermore, the area of $\mathscr{A}_4$ is $u \cdot v = u \cdot (2\sqrt{2} \cdot p - u)$, which has a maximum at $2p^2 = b$. Thus, $\mathscr{A}_4$ lies on at most 6 tiles. $\mathcal{QED}$

```
PROCEDURE PointToTile (T);
  tile:= (T[1] DIV (2*p), T[2] DIV (2*p));
  row:= 0;                          (* Assume T lies in region 0 – an even row tile.  *)
  origin[1]:= tile[1] * (2*p); origin[2]:= tile[2] * (2*p);
  offset:= SubCoordinates (T, origin); (* Position of the point in the shaded square, *)
  local:= Orig2Local (offset);          (* and in local coordinates by eq. (5.14).    *)
                                   (* Finally, check if T is not in region 0 in Figure 5.9: *)
  IF local[1] ≥ 2*p THEN row:= 1;                                      (* region 1 *)
    INC (tile[1]); INC (tile[2]); origin:= AddCoordinates (origin, (p, p));
  ELSIF local[2] < 0 THEN row:= 1;                                     (* region 2 *)
    INC (tile[1]); origin:= AddCoordinates (origin, (p, -p));
  ELSIF local[1] < 0 THEN row:= 1;                                     (* region 3 *)
    origin:= AddCoordinates (origin, (-p, -p));
  ELSIF local[2] ≥ 2*p THEN row:= 1;                                   (* region 4 *)
    INC (tile[2]); origin:= AddCoordinates (origin, (-p, p));
  END;
  RETURN (row, tile, origin);
END PointToTile;
```

Algorithm 5.9: *Computation of the tile on which the point T lies and of the origin of that tile.*

As under L_∞ (cf Lemma 5.5) we have:

Lemma 5.10 *Under the norm L_1, when tiles are diamonds, the circle of candidates lies inside the enclosing polygon.*

Proof: This lemma is an immediate consequence of Lemma 5.5 and a neighbourhood preserving mapping in eq. (5.8). $\mathcal{QED}$

This brings us to the theorem:

Theorem 5.6 *Let the universe be a set of $M \times M$ discrete points on a square grid and let $\mathcal{N}$ be a subset of that universe. Then there is an algorithm which finds the closest neighbour in $\mathcal{N}$ to a query point under the norm L_1 in constant time using*

$$M^2 + \frac{M^2}{2m} + M \cdot \frac{2m+1}{\sqrt{m}} + O(m) \tag{5.18}$$

bits of memory where $M = 2^m$.

Proof: The universe is tiled as explained in Lemma 5.8. As before, if a given tile is non-empty we indicate this by setting the corresponding bit in B and then store the bit map representation of the tile in the corresponding entry of S. Otherwise, the tile is empty and we store in S the position of the closest neighbour to the centre of this

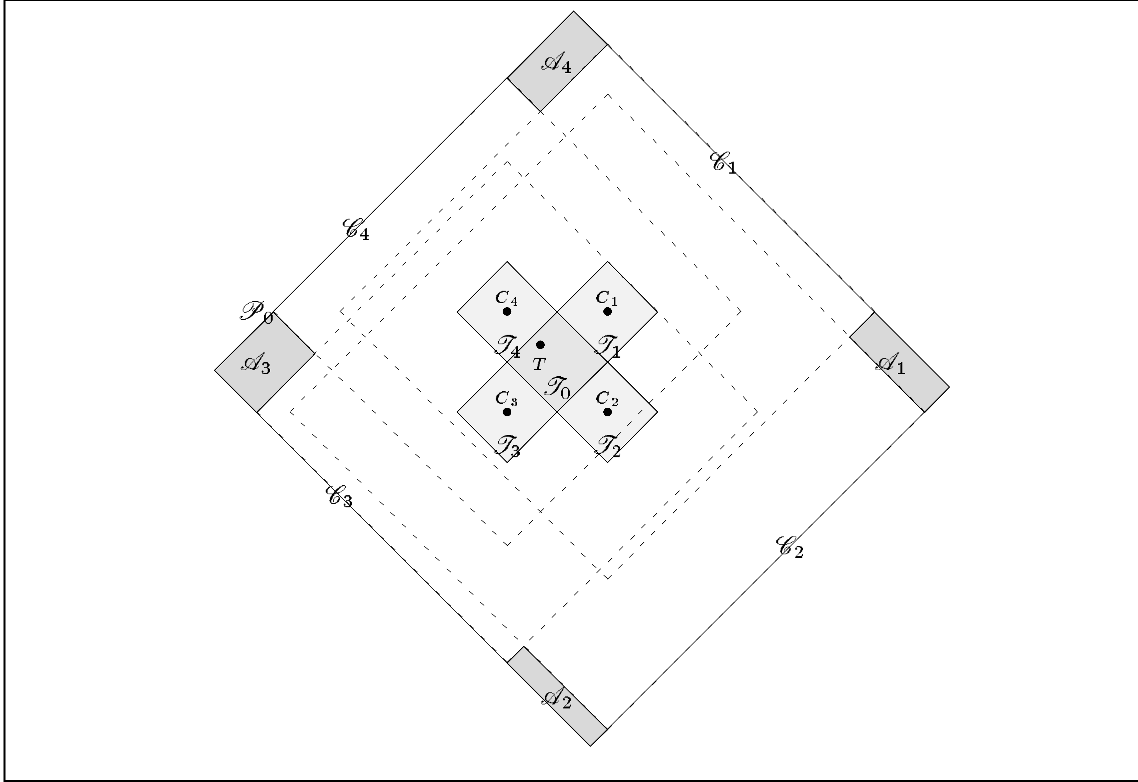

Figure 5.10: *Placement of corner areas under the norm L_1.*

tile. Both arrays, `B` and `S`, have three indices. For example, entries `B[0,` i`,` j`]` and `S[0,` i`,` j`]` correspond to the even row tile with indices i and j (cf. Figure 5.9).

The number of even-row tiles is $\frac{M}{2p} \times \frac{M}{2p}$, and the number of odd-row tiles $(\frac{M}{2p}+1) \times (\frac{M}{2p}+1)$ (cf. Figure 5.9). Therefore, the number of all tiles, and thus the number of entries in arrays `S` and `B`, is $2 \cdot (\frac{M}{2p})^2 + 2 \cdot \frac{M}{2p} + 1$ which puts the size of a data structure at

$$M^2 + \frac{M^2}{2m} + M \cdot \frac{2m+1}{\sqrt{m}} + 2m + 1 \qquad (5.19)$$

bits.

Since Algorithm 5.10 searches for the closest neighbour exactly in the same way as does Algorithm 5.6 under the norm L_∞, we omit further details. We also assume the existence of functions `Intersect` and `SearchCorner` with similar roles to their counterparts in Algorithm 5.6.

Finally, the space used by Algorithm 5.10 is bounded by eq. (5.19) for the data structure and $O(m)$ bits for internal constants, as indicated in eq. (5.18). $\mathcal{QED}$

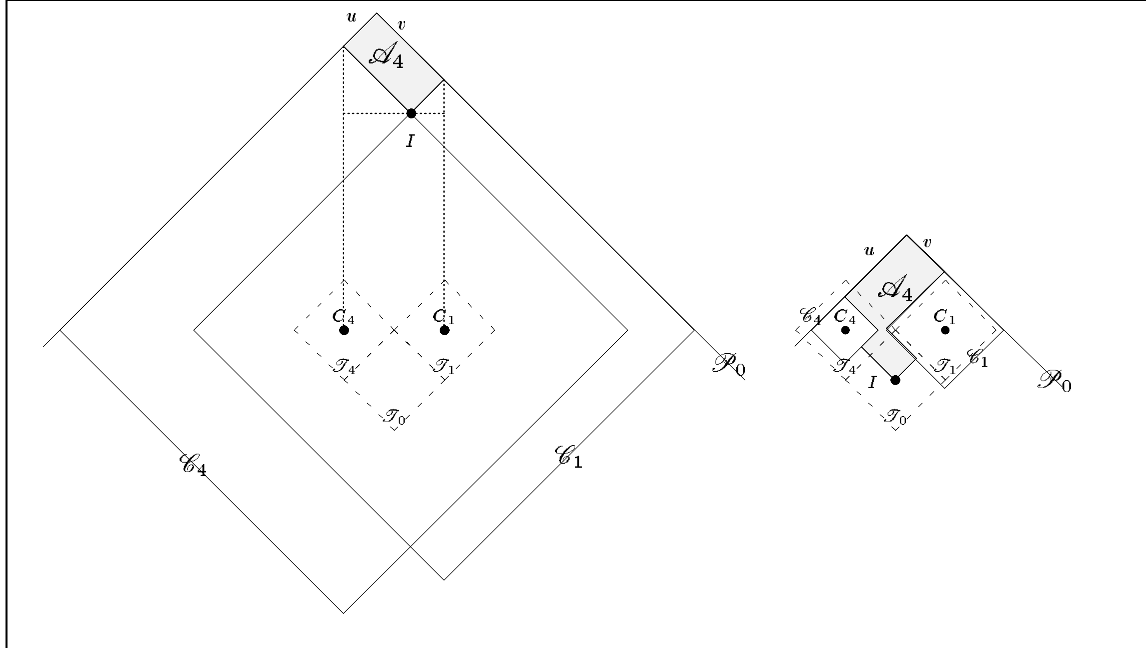

Figure 5.11: *Corner area $\mathscr{A}_4$ limited by the distance between centres of empty circles $\mathscr{C}_4$ and $\mathscr{C}_1$.*

The tiling presented in Lemma 5.8 leaves the left most and the bottom most tiles inherently broken (not whole). Hence, not all bits of bit map representations of these tiles are used. One can fix this by combining appropriate bit map entries of S. In particular, if the number of points in the right most and the top most "half-tiles" of odd rows (see Figure 5.9) is at most $\frac{b}{2}$ we can reduce, by mere combination of bit maps, the size of a data structure to

$$M^2 + \frac{M^2}{2 \lg M} + O(\log M) \tag{5.20}$$

bits, which is the same bound as under the norm L_∞ mentioned in Theorem 5.4.

5.4.4 L_2, or the Trouble with Curved Circles

The basic technique for the large universe, used in § 5.4.2 and § 5.4.3, is to construct empty circles of direct neighbours, construct the circle of candidates, and exhaustively search the area inside the circle of candidates and outside empty circles. By Lemma 5.4 and Lemma 5.9, under L_∞ and L_1 respectively, these areas are not only small, but they also intersect only a few tiles.

Unfortunately the same is not true under the norm L_2. Consider the case illustrated in Figure 5.12, where query point T is approximately in the middle of the universe. The closest neighbours of C_1 and C_2 (N_1 and N_2 respectively) are in opposite directions from

```
PROCEDURE Neighbour (T);
  (row, 𝒯0, origin0):= PointToTile (T);                  (* First, tile 𝒯0 and its origin, *)
  𝒯1:= (𝒯0[1]+1-row, 𝒯0[2]+1-row));                    (* then direct neighbours and ...  *)
  origin1:= AddCoordinates (origin0, (p, p));                        (* ... their origins.  *)
  𝒯2:= (𝒯0[1]+1-row, 𝒯0[2]-row));  origin2:= AddCoordinates (origin0, (p, -p));
  𝒯3:= (𝒯0[1]-row, 𝒯0[2]-row);  origin3:= AddCoordinates (origin0, (-p, -p));
  𝒯4:= (𝒯0[1]-row, 𝒯0[2]+1-row)  origin4:= AddCoordinates (origin0, (-p, p));
                                   (* Next, closest points implicitly define circles:  *)
  N:= (∞, ∞);                                  (* the initial circle of candidates 𝒞T; *)
  otherRow:= (1-row);                          (* direct neighbours are in "other" row *)
  FOR i:= 1 TO degree DO                                (* construct empty circle 𝒞i  *)
    offset:= SubCoordinates (T, origini);
    Ni:= SUneighbour (S[otherRow, 𝒯i], B[otherRow, 𝒯i], origini, offset);
    N:= Closest (1, 2, T, N, Ni);                                  (* and update 𝒞T.  *)
  END;
                                                   (* Finally, search corner areas.  *)
  Δ1:= (+1, +1); Δ2:= (+1, -1); Δ3:= (-1, -1); Δ4:= (-1, +1);
  FOR i:= 1 TO degree DO j:= (i MOD degree) + 1;
    (row, tile):= Intersect (𝒯i, Ni, 𝒯j, Nj, i);
    N:= Closest (1, 2, T, N, SearchCorner (row, tile, Δi, T, N));
  END;
  RETURN N
END Neighbour;
```

Algorithm 5.10: *Searching for the closest neighbour to the query point T under the norm L_1.*

T, and $\delta_2(T, C_1) \approx \delta_2(T, C_2)$ and $\delta_2(T, N_1) \approx \delta_2(T, N_2)$. In this case the corner area, $\mathscr{A}_1$ (the right dark shaded area in Figure 5.12) has area $\Theta(\sqrt{b} \cdot M)$ and lies on $\Theta(\frac{M}{\sqrt{b}})$ tiles.

One might hope that adding more empty circles would decrease the number of tiles which need be searched. Assume that the centre of the next empty circle, $\mathscr{C}_i$, is at point C_i. Now draw a line from query point T through C_i and put the closest neighbour of C_i, N_i, at the point where this line crosses the circumference of $\mathscr{C}_T$. It is not hard to verify that this is a possible situation. Obviously, we decreased the area of $\mathscr{A}_1$, but we did not decrease the number of tiles on which it lies. The same step can be repeated arbitrary many times.

The reason the technique does not seem to work is the difference in curvatures of the circle of candidates, $\mathscr{C}_T$, and the empty circles, $\mathscr{C}_i$. In other words, in the worst case circumferences of $\mathscr{C}_T$ and $\mathscr{C}_i$ have in common only one point and thus the empty circle does not substantially decrease the "length" of the corner area. On the other hand, under the norms L_1 or L_∞ one common point implies a long common section of both circumferences (curvatures of $\mathscr{C}_T$ and $\mathscr{C}_i$ are the same – circumferences are straight lines) and the length of the corner area is decreased to a few tiles only. The same reasoning we have applied to L_2 can be extended to other norms, L_f, where $1 < f < \infty$. However,

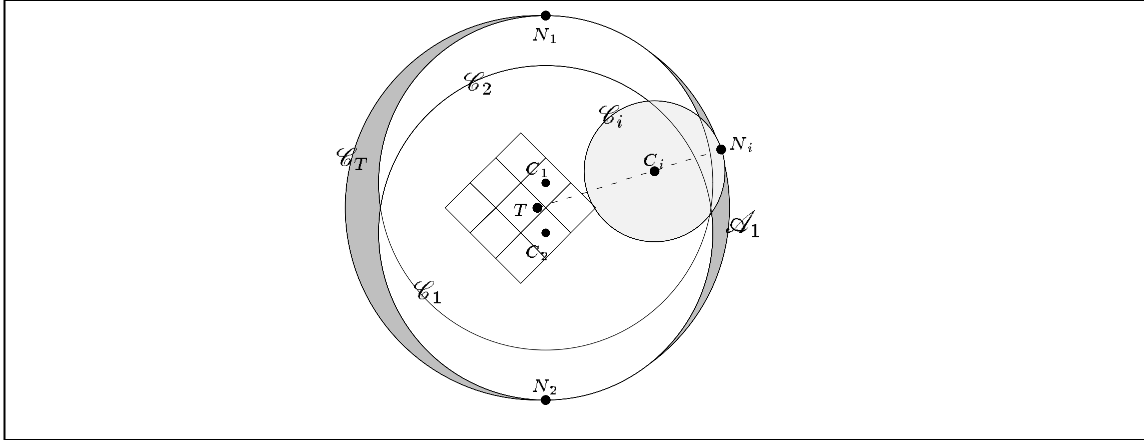

Figure 5.12: *Circles under L_2.*

there remains the open question of how many tiles the discrete points of a corner area can intersect. We conjecture that this number is also too large. It is possible to stop adding new empty circles after a constant number of steps and get an *approximate* solution. It remains open as to how good an approximation this gives.

5.4.5 Conclusion

In § 5.4.2 and § 5.4.3 we presented constant time solutions for the static closest neighbour problem in two dimensions under the norms L_∞ and L_1 respectively. Both solutions assumed a universe of size $M \times M$ points. We can easily adapt these solutions to other rectangular universes: we just have to change the tiling and, subsequently, the enumeration of arrays `S[.]` and `B[.]`. Next, if we store in the array `S[.]` either the bit map or a pointer to the tile, instead of to the point, we can cope with even a larger universe (cf. Corollary 5.1):

Corollary 5.2 *Let the universe be a set $S_1 \times S_2$ discrete points on a grid, where $0 < W = S_1 \cdot S_2 \leq 2m \cdot M^2 = m \cdot 2^{2m+1}$, and let $\mathcal{N}$ be a subset of that universe. Then there are algorithms which find the closest neighbour to a query point in $\mathcal{N}$ under either norm L_∞ or L_1, in constant time using*

$$W + \frac{W}{2m} + O((S_1 + S_2) \cdot \sqrt{m}) \tag{5.21}$$

bits of space.

The third order term in eq. (5.21) comes from eq. (5.18). As mentioned, under certain circumstances (see eq. (5.20)) this term can be eliminated and the total space bound reduced to

$$W + \frac{W}{d \cdot m} + O(m) \tag{5.22}$$

bits. In eq. (5.22), d is the dimension of the space and by setting it to 1 (that is by setting $S_2 = 1$) we get a one-dimensional solution from Corollary 5.1.

5.5 Many Dimensions

In this section we extend our solution of the closest neighbour problem under the norm L_∞ to an arbitrary, though constant, number of dimensions, d. First we generalize geometrical entities defined in § 5.4.1, and then describe the data structure and the searching algorithm.

5.5.1 Introduction

In higher dimensions we use similar entities as were defined in § 5.4.1. In fact, most of the entities keep the same names though prefixed by "hyper-". The prefix is, however, omitted when this does not introduce ambiguity.

We could define tiles as generally as in two dimensions (cf. Definition 5.2), but for the purpose of this work we define only one kind of them:

Definition 5.9 *A* ***hyper-tile****, is a d-dimensional hyper-cube with facets parallel to coordinate axes. The tiles sharing a common facet with a given tile are its* ***direct neighbours****.*

The direct neighbours of tile $\mathscr{T}_0$ in i^{th} dimension are $\mathscr{T}_i$ and $\mathscr{T}_{d+i}$.

A *hyper-sphere*, the generalization of a circle, is a set of equidistant points from some central point, which gives (cf. Definition 5.3):

Definition 5.10 *Let C_X be a middle point of a hyper-tile $\mathscr{T}_X$, then the* ***empty hyper-sphere*** *of $\mathscr{T}_X$, $\mathscr{C}_X$, is the largest hyper-sphere with centre C_X and empty interior. Thus, the closest neighbour of C_X, N_X, lies on the surface of $\mathscr{C}_X$.*

Since the only norm we use in this section is L_∞, hyper-spheres are hyper-cubes.

From a hyper-tile, its direct neighbours and direct neighbours' empty hyper-spheres, we get a definition of an *enclosing hyper-cuboid* (cf. Definition 5.4):

Definition 5.11 *Let $\mathscr{T}_0$ be a tiling hyper-cube and let $\{\mathscr{C}_i\}$ be empty spheres of respective direct neighbours. Then the* ***enclosing hyper-cuboid*** *of $\mathscr{T}_0$, $\mathscr{P}_0$, is the smallest hyper-cuboid that has facets parallel with $\mathscr{T}_0$ and includes all empty spheres $\mathscr{C}_i$.*

Inside the enclosing hyper-cuboid we have hyper-wedges (cf. Definition 5.5):

Definition 5.12 *Let $\mathscr{T}_0$ be a tiling hyper-cube, and C_0 and $\mathscr{P}_0$ its centre and enclosing hyper-cuboid respectively. Next, make d hyper-planes parallel to coordinate axes and intersecting in C_0. Then each volume enclosed by all hyper-planes and $\mathscr{P}_0$ which contains a vertex of $\mathscr{P}_0$ is a* ***hyper-wedge****.*

It is not hard to see that Definition 5.12 in two dimensions becomes Definition 5.5, because there the centre of a direct neighbour always lies on the line which is orthogonal on a side of an enclosing polygon and goes through the middle point C_0 (cf. Figure 5.3).

Corner area (cf. Definition 5.6) and circle of candidates (cf. Definition 5.7) are the last two geometrical entities generalized to d dimensions:

Definition 5.13 *Consider the hyper-wedge defined by the corner point V_i of $\mathscr{P}_0$. Then the volume that lies inside this hyper-wedge and outside the empty hyper-spheres of all direct neighbours is called the* ***corner hyper-area*** *$\mathscr{A}_i$.*

Definition 5.14 *Let the point T lie in the tiling hyper-cube $\mathscr{T}_0$ and let C_i be centres of its direct neighbours with respective closest neighbours N_i. Further, among N_i let N_X be the closest to T. Then the hyper-sphere $\mathscr{C}_T$ with a centre in T and N_X on its surface is a* ***hyper-sphere of candidates****.*

Both lemmata from § 5.4.1 easily generalize to d dimensions:

Lemma 5.11 *There are at most 2^d corner areas.*

Proof: Trivially from Definition 5.13 since hyper-cube has 2^d vertices. $\mathcal{QED}$

Lemma 5.12 *Let the point T lie inside hyper-tile $\mathscr{T}_0$. Then the closest neighbour of T lies on a surface or inside the sphere of candidates $\mathscr{C}_T$, and outside the interior of the empty spheres of $\mathscr{T}_0$'s direct neighbours.*

Proof: Following the same reasoning as in Lemma 5.2. $\mathcal{QED}$

This concludes the geometrical part of section and we proceed with description of the data structure and the searching algorithm.

5.5.2 Search for the Closest Neighbour

As mentioned we deal only with the norm L_∞ (cf. eq. (5.4)). As before, we first describe how to search individual tile and then the whole universe. Both solutions are generalization of respective solutions for lower dimensions under the norm L_∞.

The Small Universe

The small universe is a d-dimensional hyper-cube of $p^d = b$ points and is represented by a bit map stored in a hyper-cubic register x_c (see Definition 4.10). The presence or absence of a point $T = (x_1, x_2, \ldots, x_d)$ is denoted by setting of bit $x_c.\mathtt{b}[x_1, x_2, \ldots, x_d]$ to 1 or 0 respectively.

The searching algorithm is based on the same idea as it was used in Algorithm 5.4: split the universe into regions and then search each of them. Therefore we generalize the notion of search regions:

Definition 5.15 *Let $T = (t_1, t_2, \ldots, t_d)$ be a query point in a d-dimensional universe. Then hyper-pyramids, a pair for each dimension $0 < i \leq d$,*

$$\begin{array}{rcl} \mathscr{R}_{i\rightarrow} & = & \{X = (x_1, x_2, \ldots, x_d) \mid \forall j : (0 < j \leq d) \wedge (x_i - t_i \geq |x_j - t_j|)\} \\ \mathscr{R}_{\leftarrow i} & = & \{X(x_1, x_2, \ldots, x_d) \mid \forall j : (0 < j \leq d) \wedge (x_i - t_i \leq |x_j - t_j|)\} \end{array} \tag{5.23}$$

are called ***search regions****.*

In simpler terms region (hyper-pyramid) $\mathscr{R}_{i\rightarrow}$, which lies in the intersection of half-spaces $x_i - t_i \geq x_j - t_j$ and $x_i - t_i \geq -(x_j - t_j)$ (where $1 \leq j \leq d$), has its top at the point T and extends "in the ith dimension to the right". Similarly, the region $\mathscr{R}_{\leftarrow i}$, the intersection of complements of the above half-spaces, extends "in the ith dimension to the left". For example, in two dimensions $\mathscr{R}_{1\rightarrow}$ is $\mathscr{R}_{\rightarrow}$, $\mathscr{R}_{\leftarrow 1}$ is $\mathscr{R}_{\leftarrow}$, $\mathscr{R}_{2\rightarrow}$ is $\mathscr{R}_{\uparrow}$, and $\mathscr{R}_{\leftarrow 2}$ is $\mathscr{R}_{\downarrow}$ (see Definition 5.8 and Figure 5.4).

Because the search is performed in each region separately, we have to eliminate points from all other regions. This is done using proper masks, which we can generate efficiently:

Lemma 5.13 *Let the hyper-planes defining the search regions intersect at the point T. Then we can generate masks for all search regions in $O(d^2) = O(1)$ time using $O(d^2 m) = O(m)$ bits of space.*

Proof: By eq. (5.23) each search region is defined as an intersection of $2d$ half-spaces,[10] and the intersection, when we deal with bit masks, translates into a conjunction of corresponding bit masks. Assuming that we have masks for individual half-spaces ($\mathscr{H}_{k,l}$ and $\mathscr{H}_{k,-l}$ for $x_k - t_k \geq x_l - t_l$ and $x_k - t_k \geq -(x_l - t_l)$ respectively) we can generate masks for individual search regions by

$$\begin{array}{rcl} \mathscr{R}_{k\rightarrow} & = & \bigwedge_{l=1}^{d} (\mathscr{H}_{k,l} \wedge \mathscr{H}_{k,-l}) \\ \mathscr{R}_{\leftarrow k} & = & \bigwedge_{l=1}^{d} (\overline{\mathscr{H}_{k,-l}} \wedge \overline{\mathscr{H}_{k,-l}}) \ . \end{array} \tag{5.24}$$

[10] Half-spaces $x_i \geq x_i$ and $x_i \geq -x_i$ are whole universes and we would not need to consider them at all.

in $O(d)$ time. Finally, since there are $2d$ search regions, we can generate masks for all of them in $O(d^2) = O(1)$ time.

The numbers $H_{k,l}$ and $H_{k,-l}$, defined in eq. (4.29) and eq. (4.30) respectively, are masks for special case of half-spaces which intersect in points $x_k = x_l = \frac{p}{2}$ (see also Figure 4.12). Further, in Example 4.8 we have shown that these numbers are in rectangular interpretation $P^{\diagup}_{r,\delta}$ and $P^{\diagdown}_{r,\delta}$ respectively (see eq. (4.23), eq. (4.22), and Figure 4.7). Finally, it is easy to see that using shift operation from Example 4.6 we can generate from these special masks all $\mathscr{H}_{k,l}$ and $\mathscr{H}_{k-l}$ (see Figure 4.10). This brings us to Algorithm 5.11 which generates $\mathscr{H}_{k,\pm l}$ which intersect at `point`. Note the only restriction on the coordinates of `point` is that their difference and sum have to be integer values. Obviously Algorithm 5.11 uses $O(m)$ bits of space and runs in a constant time. $\mathcal{QED}$

```
PROCEDURE GenerateMask (point, k, l, slope);
  IF k = l THEN RETURN P END;                        (* All bits set by eq. (4.2).  *)
  negate:= k > l;                            (* The order of dimensions is reversed.  *)
  IF negate THEN tmp:= k; k:= l; l:= tmp END;
  r:= p^k; δ:= p^(2k-l);              (* Which rectangular register we are dealing with.  *)
  IF slope = '/' THEN                                        (* The left border plane *)
    maskSeed:= P^/_(r,δ); Δ:= point[k] - point[l]                  (* by eq. (4.23), or *)
  ELSE                                                            (* the right one *)
    maskSeed:= P^\_(r,δ); Δ:= p^(d-k) - point[k] - point[l]        (* by eq. (4.22).  *)
  END;
  IF negate THEN maskSeed:= Negate (maskSeed) END;
  IF Δ = 0 THEN RETURN maskSeed                                (* Finally, shifting.  *)
  ELSIF Δ > 0 THEN RETURN ShiftRightColumns (maskSeed, Δ, NOT negate)
  ELSE RETURN ShiftLeftColumns (maskSeed, -Δ, negate)
  END
END GenerateMask;
```

Algorithm 5.11: *Generation of mask* $\mathscr{H}_{k,l}$ *or* $\mathscr{H}_{k,-l}$.

By setting $k = 1$, $l = 2$, and $d = 2$ in Algorithm 5.11 we get a two-dimensional instance of the function.

Finally, we prove:

Theorem 5.7 *Let the universe be a set of* $b = d \cdot m$ *discrete points on a hyper-cubic grid and let* $\mathcal{N}$ *be a subset of that universe. Then there is algorithm which finds the closest neighbour in* $\mathcal{N}$ *under the norm* L_∞ *in constant time using* b *bits for a data structure and* $O(m)$ *bits for internal constants.*

Proof: The data structure we use is an obvious bit map stored in a b-bit hyper-cubic register. Further, Algorithm 5.12 is a mere generalization of two-dimensional Algo-

rithm 5.4 under the norm L_∞. For search of extremal set bits it uses functions `LMBofCubic` and `RMBofCubic` presented in Algorithm 4.15. $\mathcal{QED}$

```
PROCEDURE SUneighbour (domain, bitMap, origin, T);
  IF bitMap THEN T:= SubCoordinates (T, origin);      (* domain stores bit map:  *)
    FOR i:= 1 TO d DO                                 (* First, generate masks for ...  *)
      FOR j:= 1 TO d DO                                        (* ... all half-spaces.  *)
        H_{i,j}:= GenerateMask (T, i, j, '/');
        H_{i,-j}:= GenerateMask (T, i, j, '\');
      END
    END;
    FOR i:= 1 TO d DO                                 (* Next, generate masks for ...  *)
      R_{i→}:= H_{i,1} AND H_{i,-1};                 (* ... search regions by eq. (5.24).  *)
      R_{←i}:= Negate (H_{i,1}) AND Negate (H_{i,-1});
      FOR j:= 2 TO d DO
        R_{i→}:= R_{i→} AND H_{i,j} AND H_{i,-j};
        R_{←i}:= R_{←i} AND Negate (H_{i,j}) AND Negate (H_{i,-j});
      END;
    END;
    N:= ∞;                                            (* The closest point is far away.  *)
    FOR i:= 1 to d DO                                 (* Finally, search all regions:  *)
      N_←:= RMBofCubic (i, domain AND R_{←i});                (* first left direction, *)
      N_→:= LMBofCubic (i, domain AND R_{i→});                (* then the right one, *)
      N:= Closest (∞, d, T, N, N_←, N_→);            (* and choose the closest point.  *)
    END;
    RETURN AddCoordinates (origin, N)
  ELSE RETURN domain                                  (* The domain is a pointer.  *)
  END;
END SUneighbour;
```

Algorithm 5.12: *Searching for the closest neighbour in a small d-dimensional universe under the norm* L_∞.

Similarly as before, the solution can be adapted from hyper-cubic universe to a more general hyper-cuboid universe (cf. Definition 4.9).

The Big Universe

The big universe has a hyper-cubic shape, and contains M^d points where $M = 2^m$. We "tile" it with hyper-cubic tiles (see Definition 5.9) of size $b = d \cdot m$ points (the length of a side is $p = \sqrt[d]{b} = \sqrt[d]{dm}$). Note that the size of a tile is again chosen so that its bit map representation takes as much space as the specification of an arbitrary point in the universe. Further, we generalize Lemma 5.4:

Lemma 5.14 *Let $\mathscr{T}_0$ be some some tiling hyper-cube. Then there are at most 2^d corner areas each of which lies on at most 3^d tiles.*

Proof: By Lemma 5.11 there are 2^d corner areas. Let us consider a corner area $\mathscr{A}_s$. Similarly as before, we have two cases (cf. Figure 5.6) and we restrict our attention only to the more general one, where $\mathscr{T}_0$ is not part of $\mathscr{A}_s$.

If we project $\mathscr{A}_s$ on any pair of coordinates (i, j) we get a two dimensional corner area which sides have lengths l_i and l_j. It is easy to see (cf. Lemma 5.4) that the lengths are bounded by $0 \leq l_i, l_j \leq 2p$ and thus the projected corner area lies on at most $3^2 = 9$ tiles. Using the same argument over all coordinates we get the bound of 3^d tiles.[11] $\mathcal{QED}$

Next we generalize Lemma 5.5:

Lemma 5.15 *Under the norm L_∞ and hyper-cubic tiling as explained above, the sphere of candidates lies inside the enclosing body.*

Proof: Let us assume that this is not true. Now, consider any pair of coordinates (i, j) and projections of a sphere of candidates and an enclosing body onto these coordinates. The bodies map into a circle of candidates and an enclosing polygon respectively. Because of our initial assumption there must exist a pair of coordinates (i, j) projection onto which maps a circle of candidates partially outside the enclosing polygon. However, this contradicts Lemma 5.5. $\mathcal{QED}$

With machinery built so far we can prove the final theorem:

Theorem 5.8 *Let the universe be a set of M^d d-dimensional discrete points and let $\mathcal{N}$ be a subset of that universe. Then there is an algorithm which finds the closest neighbour of a query point in $\mathcal{N}$ under the norm L_∞ in $O(d^2 \cdot 2^d \cdot 3^d) = O(1)$ time and using $M^d + \frac{M^d}{d \lg M} + O(\log M)$ bits of memory.*

Proof: Again, after tiling the universe, we store in the array `S[.]` the bit map representations of non-empty tiles and coordinates of the closest neighbour of centres of empty tiles. As before, to distinguish what is stored in the array `S[.]` we use the array of bits, `B[.]`. The mapping between a point and a bit map representation is straightforward

$$T = (x_1, x_2, \ldots, x_d) \Rightarrow \mathtt{S}[y_1,\ y_2,\ \ldots,\ y_d].\mathtt{b}[z_1, z_2, \ldots, z_d] \tag{5.25}$$

where $y_i = x_i$ div p and $z_i = x_i \bmod p$. Further, since there are $\frac{M}{p}$ tiles in each dimension, the space occupied by arrays `S[.]` and `B[.]` is

$$\frac{M^d}{p^d} \cdot (b+1) = M^d + \frac{M^d}{d \lg M} \tag{5.26}$$

[11] Obviously, this bound can be improved (cf. Lemma 5.4).

bits.

Since the implementation of a search in Algorithm 5.13 is almost identical to the search in two dimensions (cf. Algorithm 5.6), we skip a more detailed explanation. However, we observe, that the running time of Algorithm 5.13 is dominated by search in corner areas. By Lemma 5.14 there are 2^d corner areas and each lies on at most 3^d tiles. Further, by Theorem 5.7 each tile is searched in $O(d^2)$ time. Therefore, the total running time of Algorithm 5.13 is $O(d^2 \cdot 2^d \cdot 3^d) = O(1)$ time. $\mathcal{QED}$

```
PROCEDURE Neighbour (T);
  FOR i:= 1 TO d DO 𝒯0[i]:= T[i] DIV p END;          (* First, tile 𝒯0 from eq. (5.25) *)
  FOR i:= 1 TO d DO                                   (* and its direct neighbours.   *)
    𝒯i:= 𝒯0; INC (𝒯i[i]); 𝒯d+i:= 𝒯0; DEC (𝒯d+i[i])
  END;
                        (* Next, spheres are implicitly defined by the closest neighbours: *)
  N:= ∞;                                    (* the initial sphere of candidates 𝒞T; *)
  FOR i:= 1 TO 2*d DO FOR j:= 1 TO d DO origin[j]:= 𝒯i[j]*p END;              (* all *)
    offset:= SubCoordinates (T, origin);                      (* empty spheres 𝒞i, *)
    Ni:= SUneighbour (S[𝒯i], B[𝒯i], origin, offset);
    N:= Closest (∞, d, T, N, Ni)                                (* and update 𝒞T.   *)
  END;
                                                     (* Finally, search corner areas: *)
  FOR i:= 1 TO 2^d DO                         (* directions in which is expanding 𝒜i *)
    Δ[i]:= CreateDirectionVector (i);             (* are computed in O(d) time; *)
  END;
  FOR i:= 1 TO 2^d DO                                        (* search corner area 𝒜i *)
    tile:= Intersect (N, Δ[i]);
    N:= Closest (∞, d, T, N, SearchCorner (tile, Δ[i]), T, N)
  END;
  RETURN N
END Neighbour;
```

Algorithm 5.13: *Searching for the closest neighbour of the query point T in the d-dimensional universe under the norm L_∞.*

By a proper order of tile-searching in corner areas we can reduce the number of non-empty tiles being searched to $O(d^2 \cdot 2^d)$ and thus the time to $O(d^4 \cdot 2^d)$. (cf. explanation following Algorithm 5.5).

At the very end, after the same deliberation as for Corollary 5.1 and Corollary 5.2, we generalize these corollaries to d dimensions:

Corollary 5.3 *Let the universe be a set of $\{S_1, S_2, \ldots S_d\}$ discrete d-dimensional points, where $0 < W = \prod_{i=1}^{d} S_i \leq dm \cdot 2^{dm+1}$, and let $\mathcal{N}$ be a subset of that universe. Then there is an algorithm which finds the closest neighbour of a query point in $\mathcal{N}$ under the norm L_∞ in constant time using $W + \frac{W}{d \lg M} + O(\log M)$ bits of memory.*

5.6 Conclusions and Discussion

This chapter addressed the problem of a static closest neighbour problem. We presented solutions which in constant time find the closest point to the query point under the norm L_∞ using $W + \frac{W}{d \cdot m} + O(m)$ bits of space, where W is the number of points in a d-dimensional universe, and m the size of a memory register. If we use a table lookup method to find the extremal set bits, the third order term becomes $O(M^{1-\epsilon} \cdot \log^{(2)} M)$ bits due to the table size. In some applications, the points may have names. These can be stored in a perfect hash table, using their coordinates as the key (cf. [47, 48, 51]).

Because of properties of a distance function in one dimension, the same solution applies to all norms. In two dimensions we developed a separate solution for the norm L_1 although Lee and Wong showed that the norms L_1 and L_∞ are computationally equivalent. The reason for a separate solution was to keep the bound on the use of space at $W + o(W)$ bits.

The extension of our approach to other norms (e.g. Euclidean, L_2, norm) to more than one dimension does not seem to work. The main reason is that we can not bound the number of tiles on which individual corner area lies, and therefore, we can not perform in constant time an exhaustive search inside corner areas. Consequently, it remains an interesting open question whether there exists a constant time algorithm which finds the closest neighbour under the norms L_f, where $1 < f < \infty$, using $W + o(W)$ bits of space (W is the size of universe). We conjecture that there is no such algorithm. In fact, we do not even know if there is such an algorithm for $d > 2$ and the norm L_1.

The second order term in the amount of space of our solutions is $\frac{W}{d \cdot m}$. But, can we further reduce it? This question introduces another interesting open problem: a lower bound on the amount of space that would still permit constant response time. For a simple membership problem and a cell probe model we have a lower bound $\left\lceil \lg \binom{M}{N} \right\rceil$ from eq. (3.1) and, to our knowledge, this is also the best lower bound for the closest neighbour problem. We think, though, that the real lower bound is higher and depends on a sparseness of a set (see eq. (3.3)). Note, that when the relative sparseness is constant, our solutions come within a constant factor of a lower bound for a membership problem.

The construction of the complete structure takes $2d$ sweeps of the universe (two per dimension) and, since the time of one sweep is proportional to the number of tiles, the time $O(\frac{M^d}{\log M})$. In each sweep we find, for the center of each empty tile, the closest neighbour in one of the centre's $2d$ hyper-pyramids (see Definition 5.15). These hyper-pyramids now extends over the complete universe and are not restricted to a single tile.

Chapter 6

One-Dimensional Dynamic Neighbour Problem in Constant Time

Tous pour un, un pour tous!

Alexandre Dumas, Les Trois Mousquetaires

All for one, one for all!

Alexandre Dumas, Three Musketeers

The natural next step from a static closest neighbour problem addressed in § 5 is to make it dynamic, that is to permit insertions and deletions. This appears to be a difficult problem even in the bounded universe in which we are operating. In one dimension the problem was solved in $\Theta(\lg^{(2)} M)$ time by van Emde Boas et al. ([43, 45]). This solution was shown to be optimal under the comparison based model ([83]). We extend the model to RAMBO (see § 2.6), which permits an individual bit to appear in several memory registers, and provide a constant time solution under this model.

After a short introduction and motivation, we define a few additional terms, show how a one-dimensional dynamic closest neighbour problem is related to some other common problems, and briefly browse through the literature. In the following section, § 6.3, we unwrap the van Emde Boas et al. recursive data structure (cf. [43, 45]) and use this unwrapped structure in § 6.4 for the final solution of the problem. The chapter concludes with a short discussion and some open problems.

6.1 Introduction and Motivation

There are two orthogonal approaches to the closest neighbour problem. If each element in the universe knows its closest neighbour, constant query response time is easy, but updates are time consuming. This approach may be very attractive when there are only a few updates, and is clearly helpful if there are none (cf. § 5). The second approach is to note the presence or absence of a value in a small number of places (a bit map is the extreme), but this makes queries very costly.

Fredman in [50] goes through the same arguments in maintaining the partial sums of an array. As a tradeoff between the approaches he introduces a tree-like recursive structure which does not store a complete answer to a query in one spot, but distributes it over the structure. This way he avoids the worst case scenario of either approach. Our solution to the dynamic closest neighbour problem follows similar lines, as the RAMBO facilitates more efficient handling of the distributed answers.

6.2 Related Problems and Literature

We study a one-dimensional dynamic closest neighbour problem:

Definition 6.1 *Let $\mathcal{N}$ be a subset of elements of the universe $\mathcal{M} = \{0 \ldots M-1\}$. Then the* ***one-dimensional dynamic closest neighbour*** *(neighbourhood) problem is to support the following operations efficiently (cf. [67]):*

- `Insert` (e) *which inserts element e into the subset $\mathcal{N}$.*

- `Delete` (e) *which deletes element* e *from the subset* $\mathcal{N}$.
- `Left` (e) (`Right` (e)) *which returns the largest (smallest) element of* $\mathcal{N}$ *smaller (larger) than* e. *If such an element does not exist* $-\infty$ $(+\infty)$ *is returned.*

To be able to talk also about both neighbours of the largest and the smallest element in $\mathcal{N}$, we augment $\mathcal{M}$ by $-\infty$ and $+\infty$. Consequently the right (left) neighbour of $-\infty$ $(+\infty)$ is the smallest (largest) element in $\mathcal{N}$.

6.2.1 Related Problems

The dynamic closest neighbour problem is related to a number of problems including the *union-split-find* problem and the *priority queue* problem. In the union-split-find, or interval sets, problem ([55]) we have a set of contiguous and pairwise disjoint interval sets taken from a bounded universe. The interval $\mathcal{I}$ is identified by its minimum element, $\min(\mathcal{I})$. The following operations are to be supported (cf. [84]):

- `Find` (e) which returns $\min(\mathcal{I})$, where $e \in \mathcal{I}$,
- `Split` (e) which splits the interval $\mathcal{I}$ $(e \in \mathcal{I})$ into consecutive intervals $\mathcal{I}_1$ and $\mathcal{I}_2$ such that $\min(\mathcal{I}_1) = \min(\mathcal{I})$ and $\min(\mathcal{I}_2) = e$, and
- `Merge` $(\mathcal{I})$ which puts elements from intervals $\mathcal{I}$ $(e \in \mathcal{I})$ and $\mathcal{I}_1$ $((e-1) \in \mathcal{I}_1)$ into the interval $\mathcal{I}_2$. Obviously, $\min(\mathcal{I}_2) = \min(\mathcal{I}_1)$.

Since an interval is identified by its smallest element, we can replace the set of intervals with the set of these smallest elements. This makes union-split-find and neighbourhood problems equivalent as shown in Table 6.1.

neighbourhood	$\Leftrightarrow$	interval sets
`Insert` (e)	$\Leftrightarrow$	`Split` (e)
`Delete` $(\mathcal{I})$	$\Leftrightarrow$	`Merge` $(\mathcal{I})$
`Left` (e)	$\Leftrightarrow$	`Find` (e)

Table 6.1: *A mapping between the neighbourhood and interval sets problems.*

On the other hand, in an extended version[1] of the priority queue problem we have a set of elements, $\mathcal{N}$, from a bounded universe and the following operations (cf. [112]):

- `Insert` (e) which inserts an element e into $\mathcal{N}$.

[1] The basic version of the problem includes only operations `Insert` and `DeleteMin`.

- `DeleteMin` which deletes the smallest element from $\mathcal{N}$ and returns it,
- `ChangePriority` $(e,\ \Delta)$ which changes the value of the element e in $\mathcal{N}$ for Δ,
- `Delete` (e) which deletes an element e from $\mathcal{N}$, and
- `MinPriority` which returns the smallest element in $\mathcal{N}$.

As presented in Table 6.2 priority queue operations can be efficiently emulated by the operations of the neighbourhood problem. Thus, the priority queue problem is no harder than the neighbourhood problem. However, it is not known to be easier. In summary, a solution of a neighbourhood problem gives us also solutions for the union-split-find and the priority queue problems with the same time and space bounds.

priority queue	$\Rightarrow$	neighbourhood
`Insert` (e)	$\Rightarrow$	`Insert` (e)
`DeleteMin`	$\Rightarrow$	`tmp:= Right` $(-\infty)$`; Delete (tmp);` `return tmp`
`ChangePriority` $(e,\ \Delta)$	$\Rightarrow$	`Delete` (e)`; Insert` $(e+\Delta)$
`Delete` (e)	$\Rightarrow$	`Delete` (e)
`MinPriority`	$\Rightarrow$	`Right` $(-\infty)$

Table 6.2: *A mapping of the priority queue problem onto the neighbourhood problem.*

6.2.2 And the Last Stroll Through the Literature

Under the pointer machine model (cf. [101]) there are two versions of algorithms for the union-split-find problem: *separable* – in which data structures contain no path between subgraphs representing different sets, and *non-separable*. For separable algorithms there is an obvious logarithmic upper bound achieved by any balanced tree (cf. [2, 3, 34, 74, 112, 114]) which matches the amortized lower bound of Mehlhorn, Näher, and Alt in [83]. In the same papers they also prove an amortized lower bound of $\Omega(\log^{(2)} M)$ for non-separable algorithms. This lower bound is matched by the stratified trees of van Emde Boas et al. ([43, 45, 57]), which were improved by Johnson ([65]) to use less space. Our solution is also based on the van Emde Boas et al. work. La Poutreé in [93] proves that the sequence of $M-1$ splits and f finds takes at least $\Omega(M+f\cdot\alpha(f,M))$ operations under either the separable or non-separable model. This is matched by a number of algorithms (cf. [5, 94, 104]).

Under the more powerful cell-probe model, and restricting space to $\Theta(N)$ words, there is a lower bound of $\Omega(\log^{(2)} M)$ on a static version of the problem due to Ajtai ([7]). This bound is matched in [113] by Willard who combines stratified trees with the perfect hashing of Fredman, Komlós, and Szemerédi ([51]). This solution can be upgraded

to the same amortized expected bound for a dynamic version of the problem using the Dietzfelbinger et al. ([38]) randomized hashing scheme (cf. [82]). Employing [37] or [39] gives an even better result. Finally, under the same model in [86] Miltersen proved a $\Omega(\sqrt{\log^{(2)} M})$ lower bound on the `Find` operation for a split-find problem (without merge).

Assuming words are large enough to store the complete data structure (a trie in their case), Ajtai, Fredman, and Komlós in [8] describe a constant time solution to the problem. Although such an approach is in general unrealistic, it becomes manageable if the size of the data structure is proportional to the word size. Thus, Maggs and Rauch in [79] combine the Ajtai et al. structure for small chunks of a problem with stratified trees on a large scale to obtain a solution using $O(N + M^{\epsilon})$ bits, and performing p operations in worst case $O(p \log^{(2)} M)$ and expected $O(p)$ time. Their approach is similar to the word-size truncated recursion technique used in § 3.

For the priority queue problem there is a straightforward logarithmic amortized lower bound under pointer and random access machines models. It is derived from the lower bound for sorting and matched by any balanced tree structure (cf. [2, 3, 34, 74, 112, 114]). On the other hand, to our knowledge there is no known lower bound under the cell probe model. However, there is a recent $O(N \log^{(2)} N)$ sorting algorithm ([11]) under the ERAM model (see Definition 2.4) which may hint of a double-logarithmic amortized lower bound – the same as for the neighbourhood problem. Finally, all algorithms for the neighbourhood problem are directly applicable to the priority queue problem with the same time and space bounds using the reductions in Table 6.2.

6.3 Stratified Trees and Binary Tries

Stratified trees were originally defined as a recursive data structure ([43, 45]). However, "unwrapping" the recursive definition reveals a complete binary tree (trie) with the elements of $\mathcal{M}$ at the leaves of the tree and those leaves representing the elements of $\mathcal{N}$ joined in a doubly linked list (cf. Figure 6.1). We tag each that is a root of a subtree containing an element of $\mathcal{N}$ (including leaves in $\mathcal{N}$). Furthermore, each tagged internal node has a pointer to the largest and smallest elements of $\mathcal{N}$ in its subtree.

To find either neighbour of any $e \in \mathcal{M}$ it is sufficient to find the other as the desired value can be reached following pointers at the leaves. An approach to locating one of e's neighbours is to find the lowest tagged node on the path from the leaf e to the root. A linear scan does this in $O(\log M)$ time. Van Emde Boas at al. ([43, 45]) realized that it can be done using binary search. This changes the role of tags and some details of the representation, but more importantly, it leads to an algorithm with $O(\log^{(2)} M)$ worst case running time. On the other hand, the RAMBO machine model (see Definition 2.7) permits us to use word-size parallelism to perform this search in constant time. Thus, the

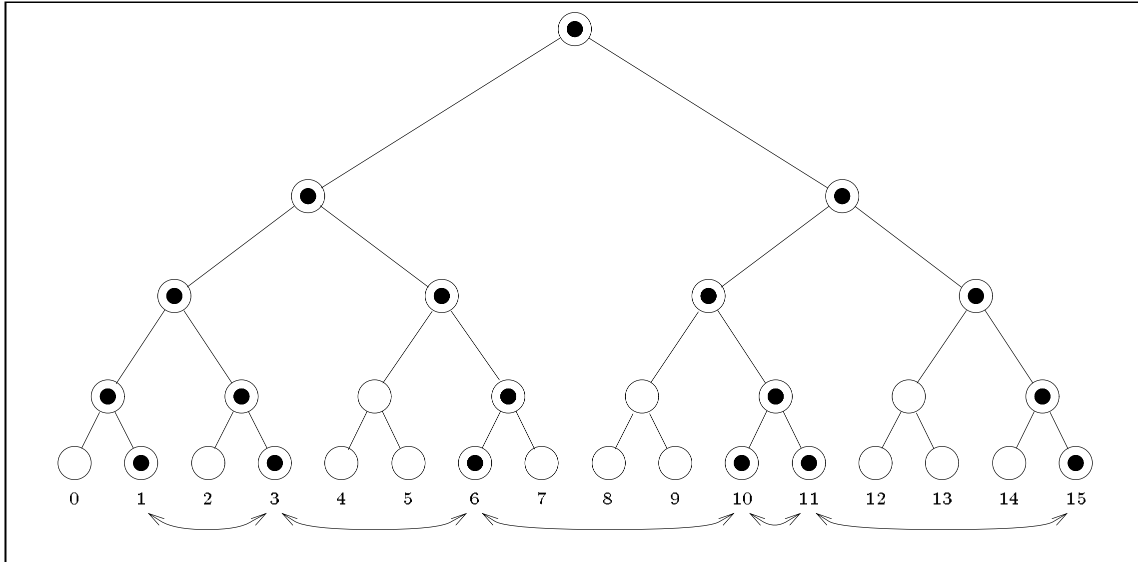

Figure 6.1: *Underlying complete binary tree (trie) of the stratified tree for a subset of the universe of size 16.*

discussion in the remainder of § 6.3 tacitly assumes the ability to quickly find the lowest tagged node above a given point and concentrates on minimizing the changes necessary in performing updates in the structure.

6.3.1 Tagging Trees

Summarizing the description above we define:

Definition 6.2 *A* ***simple tagged tree*** *is a complete binary tree with elements of $\mathcal{M}$ at its leaves and*

(i.) each node that is the root of a subtree containing an element of $\mathcal{N}$ is tagged,

(ii.) each internal tagged node has a pointer to the largest and smallest elements of $\mathcal{N}$ in its subtree, and

(iii.) elements of $\mathcal{N}$ are connected in a doubly linked list.

The neighbours of an element in a simple tagged tree are easily found in constant time once the lowest tagged ancestor of the query point is found. The difficulty is that a single element of $\mathcal{N}$ may be referred to by up to $\lg M$ internal nodes. This would appear to impose a $\Omega(\lg M)$ bound on any update algorithm. A simple modification of the structure to remove a multiple reference problem begins with the idea of a *splitting node* (see boxed nodes in Figure 6.2):

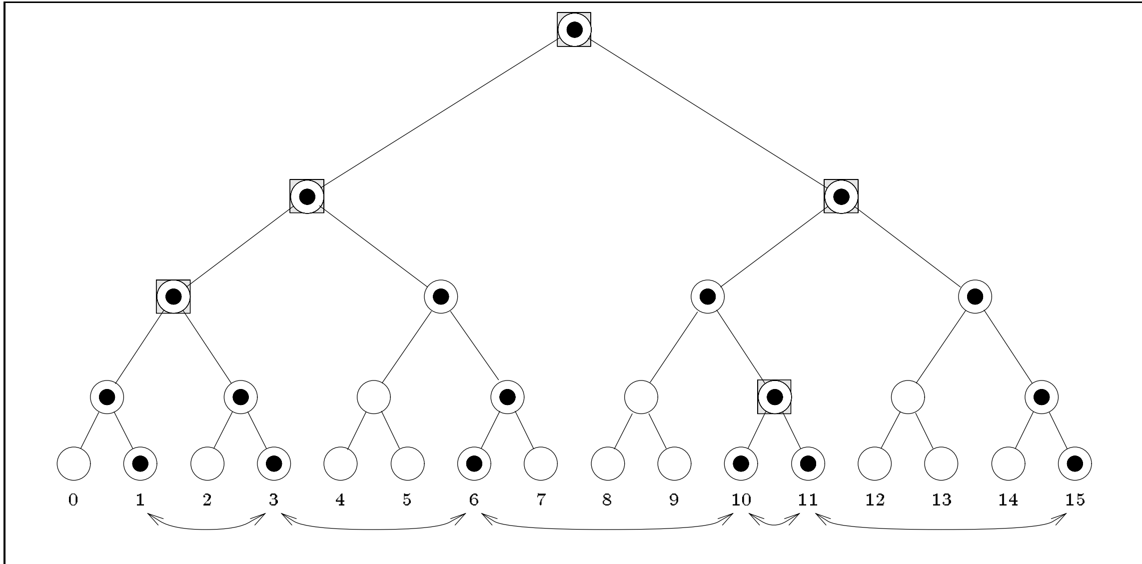

Figure 6.2: *A tree with boxed splitting nodes.*

Definition 6.3 *An internal node is a* ***splitting node*** *if there is at least one element of $\mathcal{N}$ in each of its subtrees.*

We now maintain tags and pointers only at splitting nodes. This does not quite solve the problem as a single element of $\mathcal{N}$ may still be the smallest (largest) in up to $\lg M$ subtrees (see Figure 6.3). The final twist is to maintain references to the leftmost (smallest) element in the right subtree and the rightmost (largest) element in the left subtree. That is to the "inside" rather than "outside" descendents. Thus, we have:

Definition 6.4 *A* ***split tagged tree*** *is a complete binary tree on $\mathcal{M}$ in which:*

(i.) a splitting node of the tree has a tag and pointers to the largest element in its left subtree and the smallest element in its right subtree,

(ii.) each leaf representing an element from $\mathcal{N}$ is tagged, and

(iii.) the elements of $\mathcal{N}$ are connected in a doubly linked list.

We now show that a split tagged tree supports both constant time updates and constant time queries.

6.3.2 Where to Look for Neighbours

To simplify further discussion we introduce a few terms:

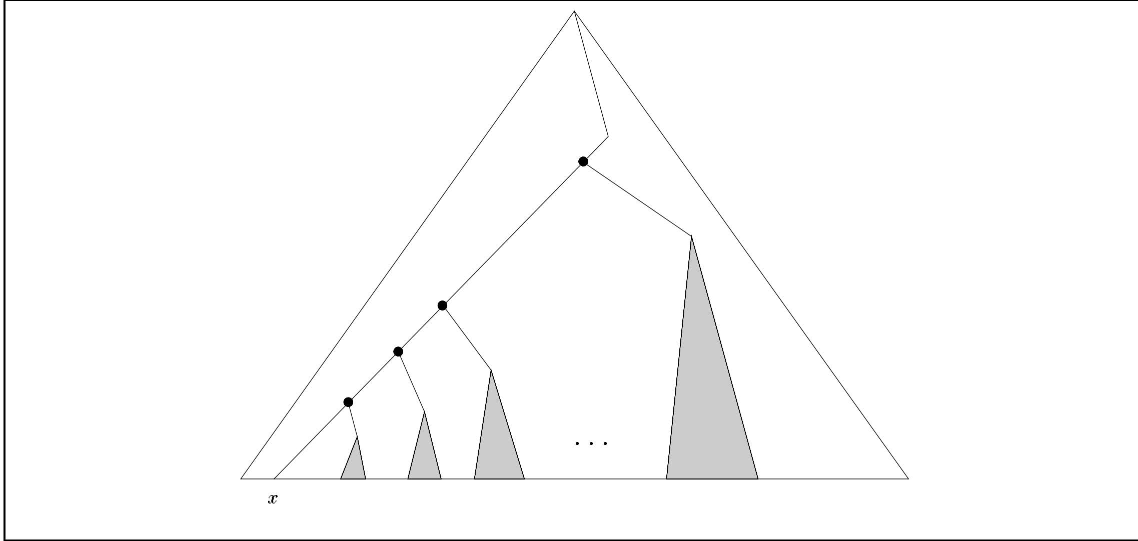

Figure 6.3: *Element x as the smallest in many subtrees.*

Definition 6.5 *The **lowest common node** (or lowest common ancestor) of two leaves is the root of the smallest subtree containing both leaves.*

Definition 6.6 *The node n is a **left (right) splitting node** of e if e is a leaf in the left (right) subtree of n. The first left (right) splitting node on a path from e to the root is the **lowest left (right) splitting node** of e.*

To permit constant time updates only a few splitting nodes may have references to a given element in $\mathcal{N}$. In fact, there are at most *two* such nodes:

Lemma 6.1 *Consider a split tagged tree, a tagged leaf e, and all left (right) subtrees at splitting nodes of the tree. Then e is the largest (smallest) element in the left (right) subtree of e's lowest left (right) splitting node, and in no other subtree rooted at a splitting node.*

Proof: We prove only half of the lemma since the other half is symmetrical. First, if e is the smallest element in $\mathcal{N}$, then it has no left splitting node. Next, if n is e's right splitting node, then e is larger than any element in the left subtree of n. Since, by Definition 6.6, the lowest left splitting node, n_l is the first left splitting node on the path from e to the root, all other elements in the left subtree of n_l are smaller than e. Finally, assume e is also the largest element in the left subtree of the left splitting node n_i. Since n_l is e's left splitting node there is an element, f, in the right subtree of n_l and $e < f$. By Definition 6.6 n_i is above n_l and thus e and f are both in the left subtree of n_i. However, this contradicts the assumption that e is the largest element in the left subtree of n_i. $\mathcal{QED}$

Finally, the following lemma indicates how to answer queries quickly:

Lemma 6.2 *Let n_l and n_r be e's lowest left and right splitting nodes respectively, and let $x, y \in \mathcal{N}$ be the neighbours of e where $x < e < y$. Then, if $e \notin \mathcal{N}$ either the pointers at n_l or at n_r point to x and y; and if $e \in \mathcal{N}$ then the pointers at n_r refer to x and e, and those at n_l refer to e and y.*

Proof: If $e \in \mathcal{N}$, this lemma follows from Lemma 6.1. If $e \notin \mathcal{N}$ Figure 6.4 presents two of four possible situations (the other two are symmetrical). Let n_c be the lowest common node of x and y. By definition, it is also a splitting node, and moreover, it is a splitting node on a path from e to the root. Since x and y are e's neighbours, they are each other's neighbours in $\mathcal{N}$ and therefore, x is the largest element in n_c's left subtree and y the smallest element in n_c's right subtree. Consequently, the pointers at n_c point to x and y, e's neighbours. A contradiction argument similar to that of the proof of Lemma 6.1 shows that n_c is either n_l or n_r. $\mathcal{QED}$

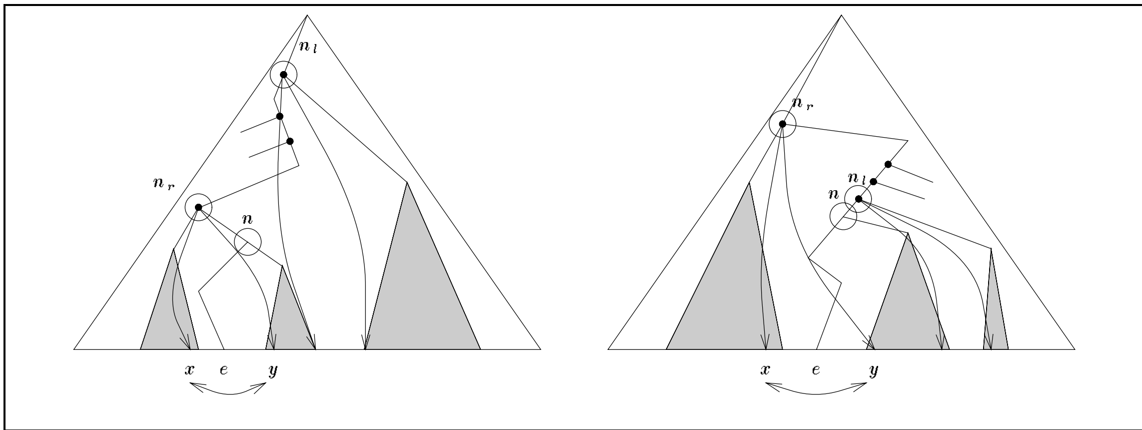

Figure 6.4: *Pointers to both neighbours of element $e \notin \mathcal{N}$ are either at the lowest left (n_l) or right (n_r) splitting node.*

6.4 The Solution

To solve the dynamic one-dimensional closest neighbour problem we first describe the data structure and how it is stored in memory, and then give the algorithms.

6.4.1 The Data Structure

Our data structure is a split tagged tree, from Definition 6.4, with the nodes represented in an array in standard heap order, i.e. the root is n_1, n_2 and n_3 are its left and right child respectively. In general, node n_i has left and right children n_{2i} and n_{2i+1}.

To store this data structure, we divide the passive block (memory) into an overlapped (see RAMBO model in § 2.6) and a non-overlapped (conventional) part. The graph describing the overlapped part is a complete binary tree of height m (it is a level shorter than the split tagged tree) where the registers are represented by paths from leaves to the root (cf. Figure 6.5). Thus, the leaves are not shared and appear in the least significant position of individual registers, while the root, which is shared by all registers, appears in the most significant position of registers.[2] The bits of the overlapped memory, $\mathcal{B}_i$ where $1 \leq i < 2^m = M$, are enumerated in standard heap order as well. Hence,

$$\texttt{reg}[i]\texttt{.b}[j] = \mathcal{B}_k \qquad \text{where} \qquad k = (i \text{ div } 2^j) + 2^{m-j-1} \tag{6.1}$$

since $k = (i + 2^{m-1}) \text{ div } 2^j = (i \text{ div } 2^j) + 2^{m-j-1}$. This brings us to a formal description of how the split tagged tree is stored in the memory:

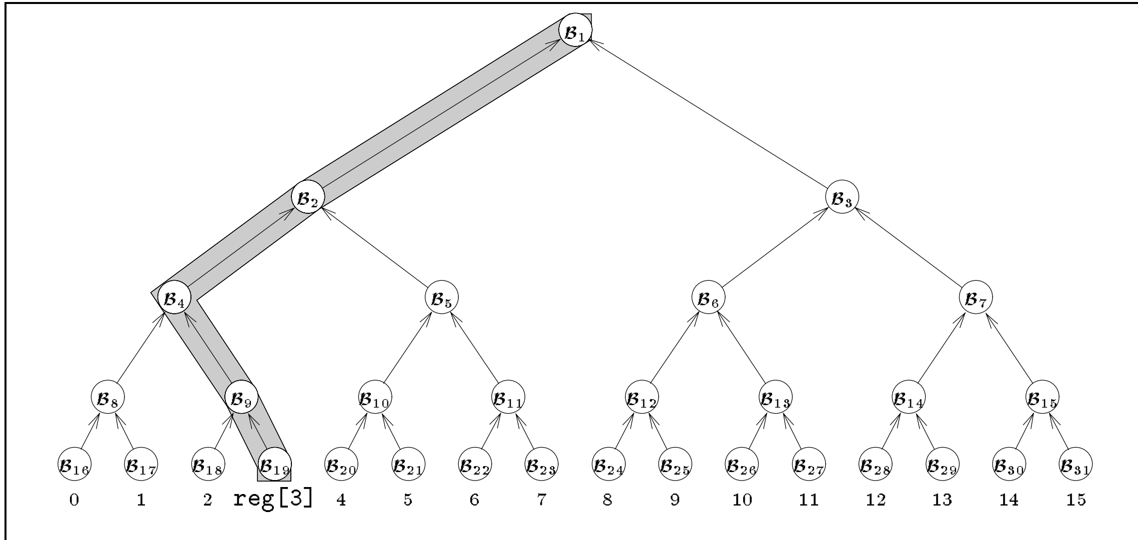

Figure 6.5: *Overlapped memory, modeled as a complete binary tree of height $m = 5$, with marked* `reg[3]`.

Definition 6.7 *The* ***memory representation*** *of a split tagged tree from Definition 6.4 in Algorithm 6.1 consists of four variables residing in two parts of memory:*

`reg`**:** *overlapped registers storing internal nodes of the tree (see eq. (6.1)). Bit $\mathcal{B}_i$ is set iff node n_i is tagged.*

`elt`**:** *leaves of the tree where bit* `elt[`e`]` *is set iff leaf e is tagged.*

[2] It will be seen later that we could not choose the root to appear as the least significant bit of registers.

internal: *internal node pointers to the largest element in the left subtree and to the smallest element in the right subtree of a given node. Pointers* `internal[i]` *correspond to node* n_i.

leaf: *ordered doubly linked list of elements of* $\mathcal{N}$.

```
CONST M= 2^m;                                                    (* size of universe 𝓜 *)
VAR                                                          (* BINARY TREE OF TAGS: *)
  reg: ARRAY [0..M/2-1] OF WORD IN Overlapped;                  (* internal nodes and *)
  elt: ARRAY [0..M-1] OF BOOLEAN IN Conventional;                          (* leaves. *)
                                                                       (* POINTERS: *)
  internal: ARRAY [1..M-1] OF RECORD                        (* at internal nodes point to *)
              left,                         (* the largest element in the left subtree and *)
              right: WORD;            (* the smallest element in the right subtree, and *)
            END IN Conventional;
  leaf: ARRAY [0..M-1] OF RECORD                                            (* at leaves *)
            prev, next: WORD;          (* connect elements of 𝓝 in a doubly linked list. *)
          END IN Conventional;
```

Algorithm 6.1: *Memory representation of a split tagged tree (data structure) used for the dynamic neighbourhood problem.*

The size of this data structure is dominated by the arrays of pointers and remains $\Theta(M \log M)$ bits as in the original van Emde Boas solution. We will reduce it later.

6.4.2 Finding Internal Nodes

To find the neighbours of e, by Lemma 6.2 we need only find e's lowest left and right splitting nodes. By Definition 6.7 the register `reg[`e `DIV 2]` represents the path from the leaf e to the root and the set bits in the register correspond to splitting nodes. Therefore, we must separate those bits representing left internal nodes from those representing right internal nodes, and find the least significant set bit among each.

To separate the bits consider a path from the leaf e to the root and the ith level node n_k on this path. It is not hard to see that if $e.\mathtt{b}[i] = 0$, where $k = (e \operatorname{div} 2^i) + 2^{m-i-1}$ (cf. eq. (6.1)), then element e is in the left subtree of n_k and otherwise it is in the right subtree of n_k. In other words, the element e itself is a mask that can be used to separate bits representing left and right internal splitting nodes, and hence expressions

$$\mathtt{reg}[e\ \mathtt{DIV}\ 2] \wedge e \qquad \text{and} \qquad \mathtt{reg}[e\ \mathtt{DIV}\ 2] \wedge \overline{e} \qquad (6.2)$$

extract bits representing e's right and left splitting nodes respectively. We can use such simple expressions only because we put the root of the tree in the most significant position of the overlapped registers `reg[.]` (cf. footnote 2). Now it is easy to prove:

Lemma 6.3 *Given an element $e \in \mathcal{M}$ we can compute its lowest left and right splitting nodes in constant time.*

Proof: After separating bits by eq. (6.2) we compute the least significant (the left most in Theorem 4.1) set bit by Algorithm 4.10 (for details see Algorithm 6.2). $\mathcal{QED}$

```
PROCEDURE LowestSplittingNodes (e);
  tmp:= e DIV 2; path:= reg[tmp];              (* The path from e to the root, *)
  j:= LMB (path AND Negate (e));   (* the least significant set bit by by eq. (6.2), *)
  n_l:= (tmp DIV 2^j) + 2^(m-1-j);     (* and the corresponding node by eq. (6.1). *)
  j:= LMB (path AND e);              (* Similarly, the lowest right splitting node. *)
  n_r:= (tmp DIV 2^j) + 2^(m-1-j);
  RETURN (n_l, n_r)
END LowestSplittingNodes;
```

Algorithm 6.2: *The lowest left and right splitting nodes of e.*

We also require the lowest common node of elements e and f:

Lemma 6.4 *The lowest common node of two elements can be computed in constant time.*

Proof: The lowest common node of elements e and f is the last common node in the paths from the root to e and f. Therefore, the most significant set bit of exclusive or of e and f corresponds to the node immediately below the lowest common node. Algorithm 6.3 presents the details of the calculation. $\mathcal{QED}$

```
PROCEDURE LowestCommonNode (e, f);
  j:= LMB (e XOR f) + 1;                 (* The appearance of bit in a register, *)
  n_j:= (e DIV 2^(j+1)) + 2^(m-1-j);     (* and corresponding node by eq. (6.1). *)
  RETURN (n_j, j)
END LowestCommonNode;
```

Algorithm 6.3: *The lowest common node of e and f, and an appearance of a corresponding bit in overlapped registers.*

6.4.3 The Algorithms

In this section we finally implement the operations required in the dynamic closest neighbour problem. We use the data structure of Definition 6.7. All implementations find both closest neighbours of $e \in \mathcal{M}$ in $\mathcal{N}$. From the preceding discussion and Algorithm 6.4, we easily see by Algorithm 6.5:

```
PROCEDURE Neighbours (e);
  IF elt[e] THEN                          (* If e ∈ N we can use a double linked list.  *)
    RETURN leaf[e]
  ELSE (n_l, n_r):= LowestSplittingNodes (e);        (* Otherwise pointers at one *)
    IF Between (internal[n_l], e) THEN              (* of the lowest splitting nodes *)
      RETURN internal[n_l]                           (* point to both neighbours.   *)
    ELSE RETURN internal[n_r] END
  END
END Neighbours;
```

Algorithm 6.4: *The neighbours of* $e \in \mathcal{M}$ *in* $\mathcal{N}$.

```
PROCEDURE Left (e);
  (left, right):= Neighbours (e);
  RETURN left;
END Left;

PROCEDURE Right (e);
  (left, right):= Neighbours (e);
  RETURN right;
END Right;
```

Algorithm 6.5: *Searching for the left and the right neighbour of* e *in* $\mathcal{N}$.

Lemma 6.5 *The left and the right neighbours of* e *in* $\mathcal{N}$ *can be found in constant time.*

Figure 6.6 is helpful in describing the insert and delete operations. It shows the effect of inserting e into the corresponding diagrams of Figure 6.4. Now, to insert:

Lemma 6.6 *Updates necessary for an insertion can be performed in constant time.*

Proof: Let x and y be the left and right neighbours of e ($e \notin \mathcal{N}$), which is to be inserted. We prove that Algorithm 6.6 properly inserts e into $\mathcal{N}$ maintaining the split tagged tree of Definition 6.4. First, the algorithm tags the proper leaf and inserts it in a doubly linked list, so the second and third parts of Definition 6.4 are satisfied.

By a simple counting argument it is easy to see that if we insert one element, exactly one internal node in a split tagged tree becomes a splitting node. Without loss of generality assume n_r is the lowest right splitting node of e, n is the lowest common node of e and y, and n is lower than the lowest common node of x and e. To prove that Algorithm 6.6 also maintains the first part of Definition 6.4 we have to show that after the insertion n becomes a splitting node (it gets tagged) and that the pointers at e's lowest left and right splitting nodes are properly updated.

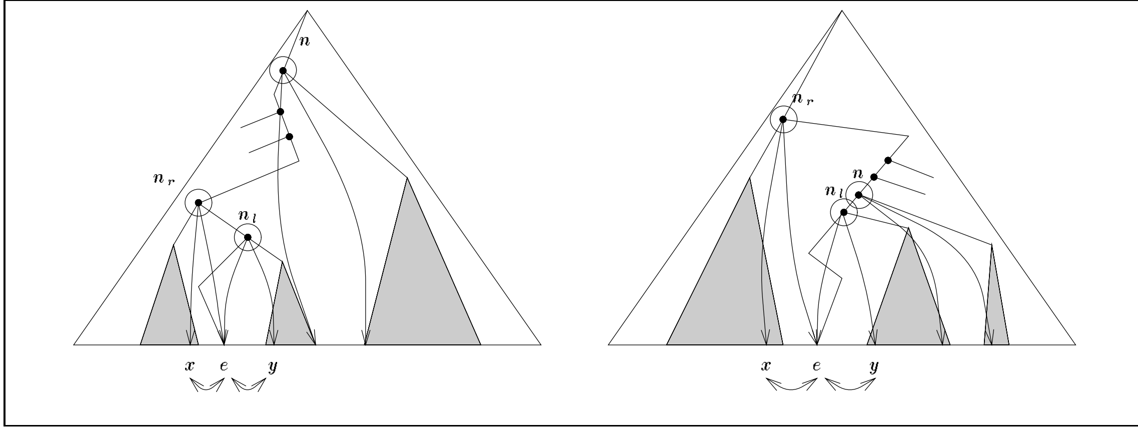

Figure 6.6: *The situation after an insertion of e in $\mathcal{N}$.*

As n is the lowest common node of e and y, these elements are in its left and right subtrees respectively (cf. left diagram in Figure 6.4). Hence n is a splitting node after the insertion. If n were a splitting node before the insertion, there would be at least one element, z, in the left subtree of n. Hence, either $x < z < e$ or $e < z < y$ which contradicts the initial assumption about e's neighbours. Moreover, after the insertion e is the only element in the left subtree of n, and so n is also e's lowest left splitting node, while n_r remains its lowest right splitting node. Algorithm 6.6 properly updates pointers at both nodes, while Lemma 6.2 guarantees that no other pointers need be changed. Constant run time follows from Algorithm 6.3 and Algorithm 6.4. $\mathcal{QED}$

Finally, we deal with a deletion:

Lemma 6.7 *An element can be deleted from $\mathcal{N}$ in constant time.*

Proof: Algorithm 6.7 takes constant time. Assuming $e \in \mathcal{N}$, its correctness follows from Lemma 6.2 by similar reasoning to the proof of Lemma 6.6. $\mathcal{QED}$

We conclude the section by:

Theorem 6.1 *Under an implicit RAMBO model (see Definition 2.7) the one-dimensional dynamic closest neighbour problem on a universe of size M can be solved in constant time and $O(M \log M)$ bits of space.*

Proof: The space bound follows from the data structure described in Definition 6.7, and the time bound from Lemma 6.5, Lemma 6.6, and Lemma 6.7. $\mathcal{QED}$

6.4.4 Final Improvements and Simplifications

The doubly linked list `leaf` is used only to find neighbours when $e \in \mathcal{N}$. In this case, by Lemma 6.2, the pointers to e's neighbours are located in its lowest splitting nodes, and

```
PROCEDURE Insert (e);
  (n_l, n_r):= LowestSplittingNodes (e);
                                                        (* UPDATE LEAVES:  *)
  (x, y):= Neighbours (e);                        (* Find both neighbours, *)
  elt[e]:= TRUE;                                   (* tag a proper leaf and *)
  leaf[x].next:= e; leaf[y].prev:= e;     (* insert e in a doubly linked list. *)
                                                       (* UPDATE THE TREE:  *)
  (n_x, j_x):= LowestCommonNode (x, e);        (* First, lowest common nodes *)
  (n_y, j_y):= LowestCommonNode (e, y);          (* with both neighbours *)
  IF n_x > n_y THEN                          (* and the lower of them will be *)
    n:= n_x; bit:= j_x
  ELSE n:= n_y; bit:= j_y
  END;                                               (* a new splitting node; *)
  reg[e DIV 2].b[bit]:= TRUE;                          (* so, it gets a tag.  *)
  IF e.b[bit] = TRUE THEN      (* Finally, get e's new lowest splitting nodes and *)
    newRight:= n; newLeft:= n_l
  ELSE newRight:= n_r; newLeft:= n END;
  internal[newRight].left:= x; internal[newRight].right:= e;       (* update *)
  internal[newLeft].left:=  e; internal[newLeft].right:=  y;  (* their pointers. *)
END Insert;
```

Algorithm 6.6: *Insertion of e into $\mathcal{N}$.*

so, by Lemma 6.3, they can be computed in constant time and we can dispense with the doubly linked list.

The bit array `elt` stores tags at leaves of a binary tree. A leaf has a tag iff the corresponding element is in $\mathcal{N}$. However, by Lemma 6.2, $e \in \mathcal{N}$ iff one of the pointers at its lowest splitting node refers back to e. By Lemma 6.3 the lowest splitting node can be computed in constant time, so the array `elt` is redundant.

Next we reexamine `internal`, the array of m-bit pointers at the internal nodes. These references are to descendents of the nodes in question. Hence the pointers need only indicate the path from the internal node itself to the relevant leaf. If the internal node is i levels above the leaves this takes i bits. Moreover, it takes only $i-1$ bits, since the value of the ith bit of the pointer to the largest (smallest) element in the left (right) subtree is 0 (1). To summarize, from the data structure in Definition 6.7 we are left with overlapped registers `reg`, for a tree of tags, and an array of variable length pointers `internal`. The size of such a data structure is expressed by the recurrence (cf. [58, p. 323])

$$s(l) = \begin{cases} 0, & \text{if } i = 0 \\ 2 \cdot s(i-1) + (1 + 2 \cdot (i-1)), & \text{otherwise} \end{cases} \tag{6.3}$$

with a closed form

$$s(m) = 3 \cdot 2^m - 2m - 3 = 3M - 2m - 3 = 3M + O(m) \ . \tag{6.4}$$

```
PROCEDURE Delete (e);
  (n_l, n_r):= LowestSplittingNodes (e);
                                                    (* UPDATE LEAVES:              *)
  (x, y):= Neighbours (e);                          (* Find both neighbours,       *)
  leaf[x].next:= y; leaf[y].prev:= x;               (* delete e from a double linked and *)
  elt[e]:= FALSE;                                   (* take a tag off a proper leaf. *)
                                                    (* UPDATE THE TREE:            *)
  IF n_r > n_l THEN                                 (* The lowest splitting nodes  *)
    (tmp, bit):= LowestCommonNode (x, e); higher:= n_l
  ELSE (tmp, bit):= LowestCommonNode (e, y); higher:= n_r
  END;                                              (* are treated differently:    *)
  reg[e DIV 2].b[bit]:= FALSE;                      (* the lower is no longer a splitting node and *)
  internal[higher].left:= x;                        (* the higher gets pointers    *)
  internal[higher].right:= y;                       (* to both neighbours.         *)
END Delete;
```

Algorithm 6.7: *Deletion of e from $\mathcal{N}$.*

The simplification of the data structure not only improves the space bound, but also the running time of the algorithms, as they need not maintain leaves, `elt`, and a doubly linked list `leaf`, at these leaves.

A further saving in space is achieved by splitting the universe into buckets of m consecutive elements and representing each bucket by a bit map. The insertion and deletion in a bit map are trivial, and constant time neighbour searches follow from Theorem 5.1. Treating buckets as elements of a smaller, $\frac{M}{m}$-element universe (a bucket-element is present iff at least one of its elements is present), we build the split tagged tree structure as described above. The bit maps require M bits, and the tree on the top, by eq. (6.4), only $\frac{3M}{m} + O(m)$ bits. The second application of this bucketing trick reduces the space requirement to $M + \frac{M}{m} + \frac{3M}{m^2} + O(m)$ bits and so:

Theorem 6.2 *Under the implicit RAMBO model the one-dimensional dynamic closest neighbour problem on the universe of size M can be supported in constant time using space $M + \frac{M}{\lg M} + o(\frac{M}{\lg M})$ bits.*

For $N = \Theta(M)$ this matches within a constant factor the lower bound $\Omega(M)$ for the membership problem (cf. eq. (3.1)).

6.5 Conclusion, Discussion and Open Questions

This chapter presented a constant time, $M + o(M)$ bit solution to a generalization of the priority queue, the dynamic one-dimensional closest neighbour problem on $[0 \ldots M-1]$.

When about half of elements are present, our solution matches the lower bound of $M - o(M)$ bits. The solution, which was primarily influenced by the stratified trees of van Emde Boas et al. ([43, 45]), uses the RAMBO machine model and word-size parallelism to manipulate the data structure efficiently.

An interesting challenge is to use an implicit RAMBO to solve the dynamic neighbourhood problem in two dimensions. We suspect that a constant time solution may be best possible under the L_∞ norm and, perhaps, under L_1.

Finally, the only known lower bounds are those under the cell probe model and for restricted versions of the problem (cf. [52, 86]). There is, however, nothing known about the tradeoff between the space and time bounds. In particular, what is the minimal size of the structure that would still permit constant time operations under the implicit RAMBO or even ERAM? We believe that this depends on the size of $\mathcal{N}$, and on M.

Chapter 7

Summary and Conclusions

Da stech ich nun, ich armer
Tor!
Und bin so klug als wie
zuvor;
...
Und sehe, daß wir nichts
wissen könen!
Das will mir schier das
Herz verbrennen.

Johann Wolfgang von Goethe, Faust

Poor fool, with all this
sweated lore,
I stand no wiser than I was
before.
...
And round we go, on
crooked ways or straight,
and all I know that
ignorance is our fate,
and this I hate.

Johann Wolfgang von Goethe, Faust

The model used throughout the thesis is an extended random access machine model, ERAM, whose instruction set is augmented to include integer division and multiplication, and bitwise Boolean operations. On the other hand, the width of a memory register and the width of the communication channel between an active block (processor) and a passive block (memory) is bounded in the model by m bits. The bound on the width of a memory register also limits the number of different objects we can deal with in a single step to $M = 2^m$. These objects form a bounded universe, $\mathcal{M}$ ($|\mathcal{M}| = M$), of all possible objects.

In the thesis we further developed three efficient methods of bit management:

- table lookup for small ranges (domains),
- word-size parallelism, and
- various organizations of the passive block.

Table lookup for small ranges is useful when we have a data structure with many instances of the same sub-structure. In this case we keep only one instance of the sub-structure and replace the others by pointers to the kept one. This results in better space bounds.

Word-size parallelism is a programming technique which uses the parallelism inherently present in a processor's instruction set. It permits us to replace a number of non-conflicting parallel operations with a constant number of sequential operations. The technique improves time bounds.

The organization of bits in the passive block was the last technique introduced. The RAMBO approach views bits as individual entities mapped into registers. Individual bits may appear in several registers simultaneously. This multiple appearance drastically enhances information dissemination, and thus, improves time and space bounds.

These methods were employed in solving the following two problems:

- membership problem in a bounded universe, and
- closest neighbour problem in a d-dimensional bounded universe.

The solution to the static membership problem permits constant time queries and uses information-theoretic necessary bound, B, on the number of bits necessary, plus some lower order terms. In the solution of the dynamic version of the problem, the same space bound is kept, but the time bound degrades to a bit more than average constant time with a high probability. The final solution of the dynamic problem takes constant worst case time, with a high probability, but uses $O(B)$ bits. In all solutions the universe is recursively split into smaller pieces until they are small enough to be described succinctly. However, there are situations when this recursive split would require non-constant depth. In these cases we observe that as the depth of recursion increases, not all pieces can be

different. Therefore, we terminate the recursion using a method of table lookup for small ranges. The depth at which we stop the recursion depends on the size of the pieces, which in turn depends on the register (word) size, and so we call the technique word-size truncated recursion.

The closest neighbour problem depends heavily on the number of dimensions, d, and on the norm, L_f. When $d = 1$, all norms, L_f are equivalent and we present a constant time, $M + \frac{M}{m} + O(m)$ bit solution. For the two dimensional case under norms L_1 and L_∞, and for the d-dimensional case ($d > 2$) under norm L_∞ we again achieve a constant time and near optimal space solution. All these solutions work in two phases: first, they restrict the area in which the candidates for the closest neighbour can occur; and then they exhaustively search this area. Since the area to be searched exhaustively is proportional to the word size, the search takes constant time using word-size parallelism.

The solution to the dynamic closest neighbour problem for a one dimensional universe uses as a data structure a complete binary tree augmented with pointers at internal nodes. The complete structure uses only $M + o(M)$ bits. The binary tree is stored in a specially organized passive block that permits us, with the help of word-size parallelism, to find in constant time the lowest common ancestor of the query element and one of its neighbours. The node found and pointers associated with it lead us in a single step to both neighbours of the query element.

The general topic addressed in this thesis is the efficient handling of bits on a small scale (in registers) in a way that leads to improved algorithms on a larger scale. However, there do remain a number of unanswered questions that we have already highlighted in the conclusions of individual chapters. These intriguing questions are paths to further research which, in turn, will pose even more interesting questions.

Our knowledge is like an interior of a balloon and our ignorance is like a surface of balloon: more knowledge we have, the more gets balloon inflated; but more the balloon is inflated the larger is its surface and larger is our ignorance.

Bibliography

[1] Associative processing – remembrance of things past. *IEEE Computer*, 27(11), November 1994.

[2] G.M. Adel'son-Vel'sky and E.M. Landis. An algorithm for the organization of information. *Soviet Mathematics - Doklady*, 3:1259–1263, 1962. (English translation.).

[3] Г.М. Адельсон-Вельскиы and Е.М. Ландис. Один алгоритм организации информации. *Доклады Академии наук СССР*, 146(2):263–266, 1962.

[4] A. Aggarwal and P. Raghavan. Deferred data structure for the nearest neighbor problem. *Information Processing Letters*, 40(3):119–122, November 8th 1991.

[5] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, 1974.

[6] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, Reading, Massachusetts, 1987.

[7] M. Ajtai. A lower bound for finding predecessors in Yao's cell probe model. *Combinatorica*, 8(3):235–247, 1988.

[8] M. Ajtai, M. Fredman, and J. Komlós. Hash functions for priority queues. In *15th ACM Symposium on Theory of Computing*, pages 299–303, New York, 1983.

[9] S.G. Akl. Memory access in models of parallel computation: From folklore to synergy and beyond. In *Proceedings 2nd Workshop on Algorithms and Data Structures*, volume 519 of *Lecture Notes in Computer Science*, pages 92–104. Springer-Verlag, 1991.

[10] S. Albers and T. Hagerup. Improved parallel integer sorting without concurrent writting. In *3rd ACM-SIAM Symposium on Discrete Algorithms*, pages 463–472, Orlando, Florida, 1992.

[11] A. Andersson, T. Hagerup, S. Nilsson, and R. Raman. Sorting in linear time? In *27th ACM Symposium on Theory of Computing*, pages 427–436, Las Vegas, Nevada, 1995.

[12] A. Andersson and T.W.H. Lai. Fast updating of well-balanced trees. In *Proceedings 2nd Scandinavian Workshop on Algorithm Theory*, volume 447 of *Lecture Notes in Computer Science*, pages 111–121. Springer-Verlag, 1990.

[13] S. Arya, D.M. Mount, N.S. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching. In *5th ACM-SIAM Symposium on Discrete Algorithms*, pages 573–582, 1994.

[14] A.M. Ben-Amram and Z. Galil. On pointers versus addresses. *Journal of the ACM*, 39(3):617–648, July 1992.

[15] A.M. Ben-Amram and Z. Galil. When can we sort in $o(n \log n)$ time? In *34th IEEE Symposium on Foundations of Computer Science*, pages 538–546, Palo Alto, California, 1993.

[16] A.M. Ben-Amram and Z. Galil. On the power of the shift operation. *Information and computation*, 117(1):19–36, 1994.

[17] J.L. Bentley and J.H. Friedman. Data structures for range searching. *ACM Computing Surveys*, 11(4):397–409, 1979.

[18] J.L. Bentley and H.A. Maurer. Efficient worst-case data structures for range searching. *Acta Informatica*, 13:155–168, 1980.

[19] J.L. Bentley, D.F. Stanat, and E.H. Williams Jr. The complexity of finding fixed-radius near neighbors. *Information Processing Letters*, 6(6):209–212, 1977.

[20] J.L. Bentley, B.W. Weide, and A.C. Yao. Optimal expected-time algorithms for closest-point problems. *ACM Transactions on Mathematical Software*, 6(4):563–580, December 1980.

[21] J. Berstel and L. Boasson. Context-free languages. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 2, pages 59 – 102. Elsevier, Amsterdam, Holland, 1990.

[22] A. Brodnik. Computation of the least significant set bit. In *Proceedings Electrotechnical and Computer Science Conference*, volume B, pages 7–10, Portorož, Slovenia, 1993.

[23] A. Brodnik. Word-size parallelism. In *Proceedings 1994 TRIO/ITRC Research Retreat*, Kingston, Ontario, Canada, 1994.

[24] A. Brodnik and J.I. Munro. Membership in a constant time and a minimum space. In *Proceedings 2nd European Symposium on Algorithms*, volume 855 of *Lecture Notes in Computer Science*, pages 72–81. Springer-Verlag, 1994.

[25] J.N. Bronštejn and K.A. Semendjajev. *Matematični priročnik.* Tehniška založba Slovenije, Ljubljana, Slovenia, 1975. (In Slovene. Translated from Russian by Albin Žabkar.).

[26] C.-C. Chang and T.-C. Wu. A hashing-oriented nearest neighbor searching scheme. *Pattern Recognition Letters*, 14(8):625–630, August 1993.

[27] M.S. Chang and N.-F. Huang C.-Y. Tang. An optimal algorithm for constructing oriented Voronoi diagrams and geographic neighborhood graphs. *Information Processing Letters*, 35(5):255–260, August 28th 1990.

[28] B. Chazelle. An improved algorithm for the fixed-radius neighbor problem. *Information Processing Letters*, 16(4):193–198, May 13th 1983.

[29] B. Chazelle. Filtering search: A new approach to query-answering. *SIAM Journal on Computing*, 15(3):703–724, 1986.

[30] B. Chazelle, R. Cole, F.P. Preparata, and C. Yap. New upper bounds for neighbor searching. *Information and Control*, 68(1–3):105–124, 1986.

[31] Y.-J. Chiang and R. Tamassia. Dynamic algorithms in computational geometry. *Proceedings of the IEEE*, 80(9):1412–1434, September 1992.

[32] Y. Choueka, A.S. Fraenkel, S.T. Klein, and E. Segal. Improved hierarchical bit-vector compression in document retrieval systems. In *9th International ACMSIGIR Conference on Research and Development in Information Retrieval*, pages 88–96, Pisa, Italy, 1986.

[33] K.L. Clarkson. Fast algorithms for the all nearest neighbors problem. In *24th IEEE Symposium on Foundations of Computer Science*, pages 226–232, Tucson, Arizona, 1983.

[34] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, Massachusetts, 1990.

[35] W. Cunto and J.I. Munro. The closest neighbor problem. In *Proceedings 22nd Annual Allerton Conference on Communication, Control, and Computing*, pages 510–515, Urbana-Champaign, Illinois, 1984.

[36] F. Dehne. An $O(N^4)$ algorithm to construct all Voronoi diagrams for k nearest neighbor searching. In J. Diaz, editor, *Proceedings 10th International Colloquium on Automata, Languages and Programming*, volume 154 of *Lecture Notes in Computer Science*, pages 160–172. Springer-Verlag, 1983.

[37] M. Dietzfelbinger, J. Gil, Y. Matias, and N. Pippenger. Polynomial hash functions are reliable. In *Proceedings 19th International Colloquium on Automata, Languages and Programming*, volume 623 of *Lecture Notes in Computer Science*, pages 235–246. Springer-Verlag, 1992.

[38] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R.E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing*, 23(4):738–761, August 1994.

[39] M. Dietzfelbinger and F. Meyer auf der Heide. A new universal class of hash functions and dynamic hashing in real time. In *Proceedings 17th International Colloquium on Automata, Languages and Programming*, volume 443 of *Lecture Notes in Computer Science*, pages 6–19. Springer-Verlag, 1990.

[40] H. Edelsbrunner. *Algorithms in Combinatorial Geometry.* EATCS Monographs in Theoretical Computer Science. Springer-Verlag, Berlin, 1987.

[41] P. Elias. Efficient storage retrieval by content and address of static files. *Journal of the ACM*, 21(2):246–260, April 1974.

[42] P. Elias and R.A. Flower. The complexity of some simple retrieval problems. *Journal of the ACM*, 22(3):367–379, July 1975.

[43] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, 6(3):80 – 82, June 1977.

[44] P. van Emde Boas. Machine models and simulations. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity, chapter 1, pages 1 – 66. Elsevier, Amsterdam, Holland, 1990.

[45] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10(1):99–127, 1977.

[46] A. Faragó, T. Linder, and G. Lugosi. Nearest neighbor search and classification in $O(1)$ time. *Problems of Control and Information Theory*, 20:383 – 395, 1991.

[47] A. Fiat and M. Naor. Implicit $O(1)$ probe search. *SIAM Journal on Computing*, 22(1):1 – 10, 1993.

[48] A. Fiat, M. Naor, J.P. Schmidt, and A. Siegel. Nonoblivious hashing. *Journal of the ACM*, 39(4):764–782, October 1992.

[49] F. Fich and P.B. Miltersen. Tables should be sorted (on random access machines). In *Proceedings 4th Workshop on Algorithms and Data Structures*, volume 955 of *Lecture Notes in Computer Science*, pages 482–493. Springer-Verlag, 1995.

[50] M.L. Fredman. The complexity of maintaining and array and computing its partial sums. *Journal of the ACM*, 29(1):250–260, January 1982.

[51] M.L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM*, 31(3):538–544, July 1984.

[52] M.L. Fredman and M.E. Saks. The cell probe complexity of dynamic data structures. In *21st ACM Symposium on Theory of Computing*, pages 345–354, Seattle, Washington, 1989.

[53] M.L. Fredman and D.E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47:424–436, 1993.

[54] H.N. Gabow and R.E. Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences*, 30:209–221, 1985.

[55] Z. Galil and G.F. Italiano. Data structures and algorithms for disjoint set union problems. *ACM Computing Surveys*, 23(3):319–344, September 1991.

[56] L. Geppert. Not your father's CPU. *IEEE Spectrum*, 30(12):20–23, December 1993.

[57] G.H. Gonnet and R. Baeza-Yates. *Handbook of Algorithms and Data Structures.* International Computer Science Series. Addison-Wesley, Reading, Massachusetts, second edition, 1991.

[58] R.L. Graham, D.E. Knuth, and O. Patashnik. *Concrete Mathematics.* Addison-Wesley Publishing Co., Reading, Massachusetts, 1989.

[59] L.J. Guibas and J. Stolfi. On computing all north-east nearest neighbors in the L_1 metric. *Information Processing Letters*, 17(4):219–223, November 8th 1983.

[60] T. Hagerup, K. Mehlhorn, and J.I. Munro. Optimal algorithms for generating discrete random variables with changing distributions. In *Proceedings 20th International Colloquium on Automata, Languages and Programming*, volume 700 of *Lecture Notes in Computer Science*, pages 253 – 264. Springer-Verlag, 1993.

[61] J. Hartmanis and J. Simon. On the power of multiplication in random access machines. In *15th IEEE Symposium on Foundations of Computer Science*, pages 13 – 23, 1974.

[62] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, Massachusetts, 1979.

[63] G. Jacobson. Space-efficient static trees and graphs. In *30th IEEE Symposium on Foundations of Computer Science*, pages 549–554, 1989.

[64] G. Jacobson. *Succint Static Data Structures.* PhD thesis, Carbegie Mellon, 1989. (Also published as technical report CMU-CS-89-112.).

[65] D.B. Johnson. A priority queue in which initialization and queue operations take $O(\log\log D)$ time. *Mathematical Systems Theory*, 15:295–309, 1982.

[66] R.G. Karlsson. *Algorithms in a Restricted Universe.* PhD thesis, University of Waterloo, Waterloo, Ontario, Canada, 1984. (Also published as technical report CS-84-50.).

[67] R.G. Karlsson, J.I. Munro, and E.L. Robertson. The nearest neighbor problem on bounded domains. In W. Brauer, editor, *Proceedings 12th International Colloquium on Automata, Languages and Programming*, volume 194 of *Lecture Notes in Computer Science*, pages 318–327. Springer-Verlag, 1985.

[68] R.M. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity, chapter 17, pages 869 – 941. Elsevier, Amsterdam, Holland, 1990.

[69] D.E. Knuth. *The Art of Computer Programming: Fundamental Algorithms*, volume 1. Addison-Wesley, Reading, Massachusetts, 2nd edition, 1973.

[70] D.E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley, Reading, Massachusetts, 1973.

[71] А.Н. Колмогоров and В.А. Успенскиы. К определению алгоритма. *Успехи математическых наук*, 13(4(82)):3–28, 1958.

[72] A.N. Kolmogorov and V.A. Uspensky. On the definition of an algorithm. *AMS Translations, Series 2*, 29:217–245, 1963. (English translation.).

[73] A. Krikelis and C.C. Weems. Associative processing and processors. *IEEE Computer*, 27(11):12–17, November 1994.

[74] T.W.H. Lai. *Efficient Maintenance of Binary Search Trees.* PhD thesis, University of Waterloo, Waterloo, Ontario, Canada, 1990. (Also published as technical report CS-90-42.).

[75] D.T. Lee and C.K. Wong. Voronoi diagrams in L_1 (L_∞) metrics with 2-storage applications. *SIAM Journal on Computing*, 9(1):200–211, February 1980.

[76] M. Li and P.M.B. Vitányi. Kolmogorov complexity and its applications. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity, chapter 4, pages 187 – 254. Elsevier, Amsterdam, Holland, 1990.

[77] M. Li and P.M.B. Vitányi. Philosophical issues in Kolmogorov complexity. In *Proceedings 19th International Colloquium on Automata, Languages and Programming*, volume 623 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 1992.

[78] M. Li and P.M.B. Vitányi. *An Introduction to Kolmogorov Complexity and its Applications.* Texts and Monographs in Computer Science Series. Springer-Verlag, Berlin, 1993.

[79] B.M. Maggs and M. Rauch. An algorithm for finding predecessors in integer sets. In *Proceedings 3rd Workshop on Algorithms and Data Structures*, volume 709 of *Lecture Notes in Computer Science*, pages 483–493. Springer-Verlag, 1993.

[80] H.G. Mairson. The program complexity of searching a table. In *24th IEEE Symposium on Foundations of Computer Science*, pages 40–47, Tucson, Arizona, 1983.

[81] K. Mehlhorn. *Data Structures and Algorithms: Multi-dimensional Searching and Computational Geometry*, volume 3. Springer-Verlag, Berlin, 1984.

[82] K. Mehlhorn and S. Näher. Bounded ordered dictionaries in $O(\log \log N)$ time and $O(n)$ space. *Information Processing Letters*, 35:183 – 189, August 1990.

[83] K. Mehlhorn, S. Näher, and H. Alt. A lower bound on the complexity of the union-split-find problem. *SIAM Journal on Computing*, 17(1):1093–1102, December 1988.

[84] K. Mehlhorn and A. Tsakalidis. Data structures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity, chapter 6, pages 301 – 342. Elsevier, Amsterdam, Holland, 1990.

[85] P.B. Miltersen. The bit probe complexity measure revisited. In *Proceedings 10th Symposium on Theoretical Aspects of Computer Science*, volume 665 of *Lecture Notes in Computer Science*, pages 662–671. Springer-Verlag, 1993.

[86] P.B. Miltersen. Lower bounds for union-split-find related problems on random access machines. In *26th ACM Symposium on Theory of Computing*, pages 625–634, Montréal, Québec, Canada, 1994.

[87] M.L. Minsky and S.A. Papert. *Perceptrons.* The MIT Press, Cambridge, Massachusetts, expanded edition, 1988.

[88] D.S. Mitrinović. *Analytic Inequalities*, volume 165 of *Die Grundlehren der mathematischen Wissenschaften.* Springer-Verlag, Berlin, 1970.

[89] O.J. Murphy and S.M. Selkow. The efficiency of using k-d trees for finding nearest neighbors in discrete space. *Information Processing Letters*, 23(4):215–218, November 8th 1986.

[90] D. Perrin. Finite automata. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 1, pages 1 – 57. Elsevier, Amsterdam, Holland, 1990.

[91] J. Potter, J. Baker, S. Scott, A. Bansal, C. Leangsuksun, and C. Asthagiri. ASC: An associative-computing paradigm. *IEEE Computer*, 27(11):19–25, November 1994.

[92] J.L. Potter. *Associative Computing: A Programming Paradigm for Massively Parallel Computers*. Frontiers of Computer Science. Plenum Press, New York, 1992.

[93] J.A. La Poutré. Lower bounds for the union-find and the split-find problem on pointer machines (extended abstract). In *22nd ACM Symposium on Theory of Computing*, pages 34–44, Baltimore, Maryland, 1990.

[94] J.A. La Poutré. New techniques for the union-find problem. In *1st ACM-SIAM Symposium on Discrete Algorithms*, pages 54–63, Philadelphia, Pennsylvania, 1990.

[95] V.R. Pratt and L.J. Stockmeyer. A characterization of the power of vector machines. *Journal of Computer and System Sciences*, 12(2):198–221, 1976.

[96] F.P. Preparata and M.I. Shamos. *Computational Geometry*. Texts and Monographs in Computer Science. Springer-Verlag, Berlin, 2nd edition, 1985.

[97] V. Ramasubramanian and K.K. Paliwal. An efficient approximation-elimination algorithm for fast nearest-neighbour search based on a spherical distance coordinate formulation. *Pattern Recognition Letters*, 13(7):471–480, July 1992.

[98] A. Salomaa. Formal languages and power series. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 3, pages 103 – 132. Elsevier, Amsterdam, Holland, 1990.

[99] J.P. Schmidt and A. Siegel. The spatial complexity of oblivious k-probe hash functions. *SIAM Journal on Computing*, 19(5):775–786, October 1990.

[100] A. Schönhage. On the power of random access machines. In *Proceedings 6th International Colloquium on Automata, Languages and Programming*, volume 71 of *Lecture Notes in Computer Science*, pages 520–529. Springer-Verlag, 1979.

[101] A. Schönhage. Storage modifications machines. *SIAM Journal on Computing*, 9(3):490–508, August 1980.

[102] G. Shannon, 1993. (Private communcation.).

[103] R.F. Sproull. Refinements to nearest-neighbor searching in k-dimensional trees. *Algorithmica*, 6:579 – 589, 1991.

[104] R.E. Tarjan and J. van Leeuwen. Worst-case analysis of set union algorithm. *Journal of the ACM*, 31(2):245–281, April 1984.

[105] R.E. Tarjan and A.C. Yao. Storing a sparse table. *Communications of the ACM*, 22(11):606–611, November 1979.

[106] J.-I. Toriwaki and S. Yokoi. Voronoi and related neighbours on digitized two-dimensional space with applications to texture analysis. In G.T. Toussaint, editor, *Computational Morphology: A Computational Geometric Approach to the Analysis of Form*, volume 6 of *Machine intelligence and pattern recognition*, pages 207–228, Amsterdam, Netherlands, 1988. North-Holland.

[107] A.M. Turing. On computable numbers, with an application to the Etscheidungsproblem. *Proceedings of London Mathematical Society*, 42:230–265, 1936.

[108] J.D. Ullman. *Principles of Database Systems.* Computer Science Press, Rockville, Maryland, USA, 2nd edition, 1982.

[109] P.M. Vaidya. An $O(n \log n)$ algorithm for the all-nearest-neighbors problem. *Discrete & Computational Geometry*, 4(2):101–115, 1989.

[110] L.G. Valiant. General purpose parallel architectures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity, chapter 18, pages 943 – 971. Elsevier, Amsterdam, Holland, 1990.

[111] Y.C. Wee, S. Chaiken, and D.E. Willard. On the angle restricted nearest neighbor problem. *Information Processing Letters*, 34(2):71–76, March 16th 1990.

[112] M.A. Weiss. *Data Structures and Algorithm Analysis.* The Benjamin/Cummings Publishing Company, Redwood City, California, 1992.

[113] D.E. Willard. Log-logarithmic worst-case range queries are possible in space $\theta(n)$. *Information Processing Letters*, 17:81–84, August 1983.

[114] N. Wirth. *Algorithms + Data Structures = Programs.* Prentice-Hall, Englewood Clifs, N.J., 1976.

[115] D. Wood. *Theory of Computation.* John Wiley & Sons, New York, 1987.

[116] A.C.-C. Yao. Should tables be sorted? *Journal of the ACM*, 28(3):614–628, July 1981.

[117] F.F. Yao. Computational geometry. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity, chapter 7, pages 343 – 389. Elsevier, Amsterdam, Holland, 1990.

[118] G. Yuval. Finding near neighbors in k-dimensional space. *Information Processing Letters*, 3(4):113–114, March 1975.

[119] G. Yuval. Finding nearest neighbors. *Information Processing Letters*, 5(3):63–65, August 1976.

A

Glossary of Definitions

A

appearance set (bit), 15

B

bounded universe, 14

C

circle
- empty circle, 101
- of candidates, 104

closest neighbour problem
- dynamic, 145
- static, 91

corner area
- multi dimensional, 133
- two dimensional, 104

D

data structure, 156

E

enclosing
- body, 132
- polygon, 102

I

inverse Ackermann function, 7

iterated logarithm, 7

M

membership problem
- dynamic, 22
- static, 22

model of computation
- extended random access machine, ERAM, 12
- random access machine with bytes overlapping, RAMBO, 17

N

node
- lowest common, 153
- splitting, 152
  - left, 153
  - lowest left, 153
  - lowest right, 153
  - right, 153

O

order notation, 6

R

register
- hyper-cubic, 71
- hyper-cuboidal, 70
- linear, 51
  - (s, k)-sparse, 52
  - s-small, 52
- rectangular, 60
  - s-column sparse, 63
  - s-row sparse, 63
  - column-stripe $S^C_{i,j}$, 62
  - row-stripe $S^R_{i,j}$, 62
- square, 61

S

search region
- multi dimensional, 134
- two dimensional, 106

sphere
- empty sphere, 132

of candidates, 133

T

tile
tiling body, 132
tiling polygon, 100

W

wedge
multi dimensional, 132
two dimensional, 103

B

Notation

In the thesis we use a consistent notation which is briefly described in this list. Each notation is accompanied by a short explanation and the page of its detail definition or of its more illustrative usage.

Notation	*Explanation*	*Page*
$\approx$	approximation with error in lower order terms	23
$\alpha(m,n)$	inverse Ackermann function	7
$\log_\lambda^{(i)} x$	iterated general logarithm	7
$O, o, \Omega, \omega, \Theta$	order notation	6
`Member`	name of a function in an algorithm	33
`elt`	name of a variable in an algorithm	33
`PROCEDURE`	reserved word in an algorithm	33
m	size of a memory register or the width of a transportation channel	9
d	dimensionality of a space or arithmetic register	90
b	size of a tile, small universe, or arithmetic register at hand: $d \cdot m$	51
p	number of bits in a register or points in a tile in one dimension: $\sqrt[d]{b}$	71
$\mathtt{reg.b}[j]$	j^{th} bit in register `reg`	15
$x_l.\mathtt{b}[i]$	i^{th} bit in linear register x_l	51
$x_r.\mathtt{b}[i,j]$	bit in i^{th} column and j^{th} row of rectangular register x_r	60
$x_c.\mathtt{b}[\pi_1, \pi_2, \ldots, \pi_d]$	bit in multidimensional register x_c	70
$\mathcal{B}$	appearance-set of a bit – in fact it is a bit	15
$S_{i,j}^C$	column-stripe register	62

$S^R_{i,j}$	row-stripe register	62
$P^X_{r,\delta}$	rectangular register with set bits in triangles (X is $\diagup$ or $\diagdown$)	66
P	special form of $P^X_{r,\delta}$ with value $2^b - 1$	51
P^X	special rectangular form of $P^X_{r,\delta}$	67
$\mathcal{M}$	universe set	14
M	size of a universe set: $\lvert\mathcal{M}\rvert = 2^m$	14
$\mathcal{N}$	subset of $\mathcal{M}$: $\mathcal{N} \subseteq \mathcal{M}$	22
$\overline{\mathcal{N}}$	subset of all elements in universe $\mathcal{M}$ and not in $\mathcal{N}$: $\mathcal{M} - \mathcal{N}$	22
N	size of the subset $\mathcal{N}$: $\lvert\mathcal{N}\rvert$	22
r	sparseness of the subset $\mathcal{N}$: $\frac{M}{N}$	23
$\delta_f(\cdot)$	distance function	90
L_f	norm of space with a distance function δ_f	90
T	query point; from Slovene for point: "*točka [totʃka]*"	90
C_x	centre point of a tile $\mathscr{T}_x$	101
N_x	closest neighbour of point C_x	101
$\mathscr{T}_i$	tile $\mathscr{T}_0$ and its direct neighbours $\mathscr{T}_1, \mathscr{T}_2, \ldots$	100
$\mathscr{C}_X$	circle with a centre in point C_X or point X	101
$\mathscr{P}_x$	enclosing polygon with respect to tile $\mathscr{T}_x$	102
$\mathscr{A}$	corner area	104
$\mathscr{R}$	search region or mask associated with it	107