Counting Polyominoes in Two and Three Dimensions

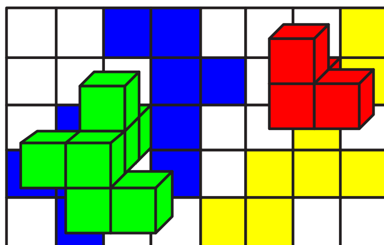
Micha Moffie

Counting Polyominoes in Two and Three Dimensions

Research Thesis

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE IN COMPUTER SCIENCE

Micha Moffie



SUBMITTED TO THE SENATE OF THE TECHNION - ISRAEL INSTITUTE OF TECHNOLOGY HAIFA

Kislev, 5763 Haifa December 2003

Dr. Gill Barequet supervised this thesis under the auspices of the computer science department

Acknowledgment

THE GENEROUS FINANCIAL HELP OF THE TECHNION IS GRATEFULLY ACKNOWLEDGED.

Contents

Abstract **5**

1 Introduction **6**

1.1 Previous Work 8

1.2 Redelmeier's Subgraph-Counting Algorithm 10

1.3 Jensen's Transfer-Matrix Algorithm 11

1.4 Organization of the Thesis 14

2 Complexity of the Transfer-Matrix Algorithm **15**

2.1 The Number of Different Signatures 15

2.1.1 Introduction 15

2.1.2 The Theorem 16

2.2 Computational Complexity 20

3 Attempts to Improve the Transfer-Matrix Algorithm **23**

3.1 Doubling the Bounding Box 23

3.1.1 Combining Signatures 24

3.1.2 Optimization 26

3.2 Multi-Level Combination of Bounding Boxes 28

3.3 Advantages and Disadvantages 31

4 Extending Jensen's Algorithm to Three Dimensions **33**

4.1 Boundary Encoding 33

4.2 Updating Rules 34

4.3 Optimizations . . . 36
4.4 Results . . . 37

5 Extending Redelmeier's Algorithm to Three Dimensions 38
5.1 Counting Polycubes . . . 38
5.2 Implementation Issues . . . 38
5.2.1 Graph Representation . . . 38
5.2.2 Recursion . . . 40
5.2.3 Large Numbers . . . 40
5.2.4 Warm Restart . . . 41
5.3 Results . . . 41

6 Conclusion 42

Acknowledgment 44

Bibliography 45

Appendix A: C Source Code of the Three-Dimensional Version of Redelmeier's Algorithm 47

List of Figures

1.1 Fixed two-dimensional dominoes, trominoes, and tetrominoes 6

1.2 Fixed three-dimensional dominoes and trominoes 7

1.3 Polyominoes as subgraphs of a specific base graph 11

1.4 Redelmeier's algorithm [Rede81, p. 196] 11

1.5 A sample intermediate polyomino, its right boundary, and its signature (example taken from Figure 1 of [Je01]) 13

2.1 The leftmost signature cell connected to the Wth signature cell (when the latter is occupied) 17

2.2 All possible signatures for $1 \leq W \leq 4$ 18

2.3 (Continuation of Figure 2.2) All possible signatures for $W = 5$ 19

2.4 Mapping a signature to a path 20

2.5 String realization 20

3.1 Combining two configurations into one 25

3.2 Combining two configurations with an overlapping column 27

3.3 Combining two extended configuration into one 29

3.4 Combining two extended configurations into an illegal configuration 30

4.1 A polycube and its corresponding boundary 35

4.2 A polycube after adding one site (note the kink in the boundary) 35

4.3 A polycube after adding two sites (note the kink in the boundary) 36

5.1 Valid cubes in a three dimensional lattice 39

5.2 The underlying graph in three dimensions 40

List of Tables

1.1 Numbers of two-dimensional fixed polyominoes 9
1.2 Numbers of three-dimensional fixed polyominoes 10

3.1 Numbers of original and extended signatures 32

4.1 Effects of the optimizations on the running time 37

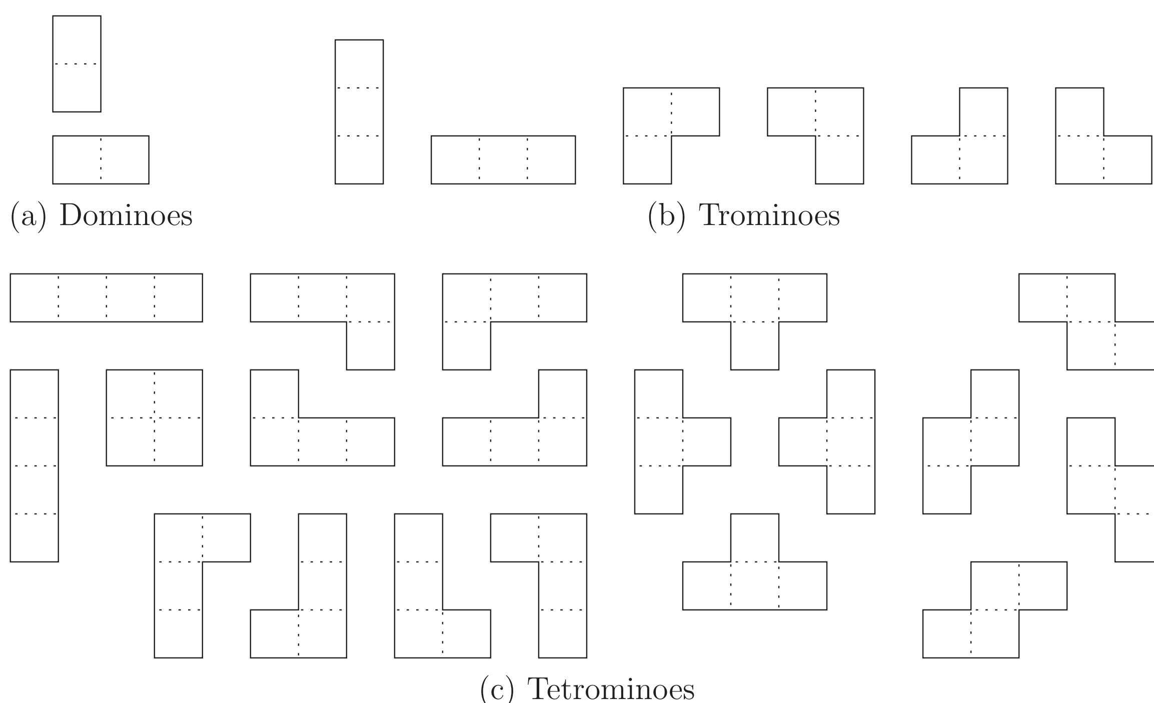
Abstract

A *polyomino*, also known in the literature as an *animal*, of order n is an edge-connected set of n squares on a regular square lattice. Recently, I. Jensen [Je01] published a novel transfer-matrix algorithm for computing the number of two-dimensional polyominoes in a rectangular lattice. We provide a rigorous computation that roughly confirms Jensen's estimation of the computational complexity of his algorithm. This is done by analyzing the number of some class of strings, which play a significant role in the algorithm. It turns out that this number is closely related to Motzkin numbers [Mo48]. We also present two extensions of Jensen's algorithm, designed to reuse previously-computed results. We then generalize the algorithm to three dimensions. In addition, we describe an efficient implementation of Redelmeier's serial algorithm [Rede81] for counting three dimensional polyominoes (termed polycubes). We confirm Lunnon's results [Lu71] for polycubes, and unpublished results found on the Internet (up to size 17). We also provide the number of polycubes of size 18, which, to the best of our knowledge, has never been published in the literature.

Chapter 1

Introduction

A polyomino of size n is an edge-connected set of n squares on a regular square lattice. *Fixed* polyominoes are considered distinct if they have different shapes *or* orientations. The symbol $A(n)$ in the literature usually denotes the number of fixed polyominoes of size n. In this thesis we use the notation of $A_2(n)$ to denote the *two* dimensions. Figure 1.1(a) shows the only two fixed *dominoes* (adjacent pairs of squares). Similarly, Figures 1.1(b) and 1.1(c) show the 6 (resp., 19) fixed *trominoes* (resp., *tetrominoes*)—polyominoes of size 3 and 4, respectively. Thus, $A_2(2) = 2$, $A_2(3) = 6$, $A_2(4) = 19$, and so on.

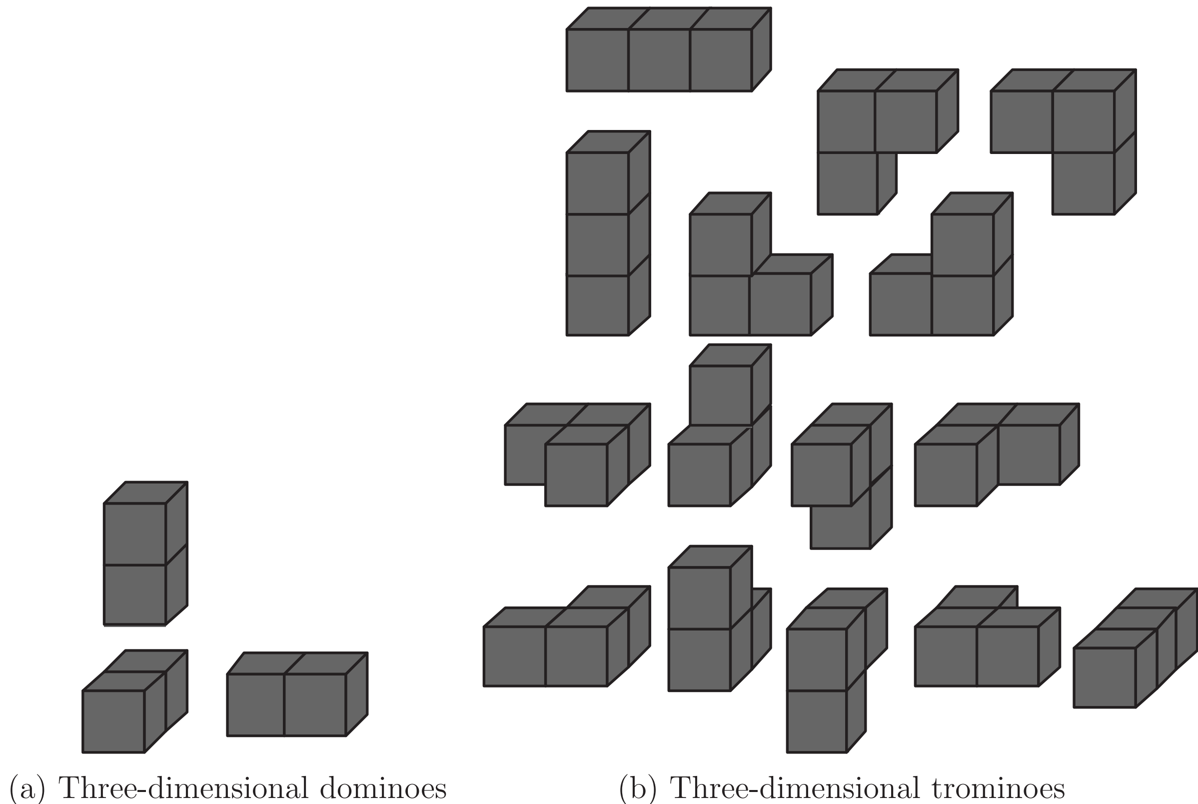


(a) Dominoes (b) Trominoes

(c) Tetrominoes

Figure 1.1: Fixed two-dimensional dominoes, trominoes, and tetrominoes

Three-dimensional polyominoes are called polycubes [Lu71]; a polycube of size n is a face-connected set of n cubes in a Euclidean three-dimentional space. We denote by $A_3(n)$ the number of distinct fixed polycubes of size n. Figure 1.2(a) shows the three polycubes of size 2. Similarly, Figure 1.2(b) shows the 15 polycubes of size 3. Therefore, $A_3(2) = 3$, $A_3(3) = 15$, and so on.



(a) Three-dimensional dominoes (b) Three-dimensional trominoes

Figure 1.2: Fixed three-dimensional dominoes and trominoes

Polyominoes and polycubes have triggered the imagination of not only mathematicians. The number of fixed polycubes is also related to investigating the properties of liquid flow through grained material [BH57], such as water flowing through coffee grains. Statistical physicians refer to polyominoes as *lattice animals*, whose number is relevant to computing the mean cluster density in percolation processes.

To this day there is no known analytic formula for $A_2(n)$ (or $A_3(n)$). The only known methods for computing $A_2(n)$ or $A_3(n)$ are based on explicitly or implicitly enumerating all the polyominoes or polycubes.

1.1 Previous Work

The following is a brief overview of the developments in counting fixed polyominoes:

- Read [Rea62] derived in 1962 generating functions for calculating the number of fixed polyominoes. These functions become intractable very fast and were used for computing $A_2(n)$ for only $n = 1, \ldots, 10$ (with an error in $A_2(10)$).
- Parkin et al. [PLP67] computed in 1967 the number of polyominoes of up to size 15 (with a slight error in $A_2(15)$) on a CDC 6600 computer.
- In 1971 Lunnon [Lu71] computed the values of $A_2(n)$ up to $n = 18$ (with a slight error in $A_2(17)$). His program generated polyominoes that could fit into a restricted rectangle. Since his algorithm generated the same polyominoes more than once, the program spent a considerable amount of time on checking polyominoes for repetitions. The program ran for about 175 hours (a little more than a week) on the Chilton Atlas I.
- An algorithm of Martin [Ma74] and Redner [Redn82] that computes polyominoes of a given size and perimeter was used in 1976 by Sykes and Glen [SG76] to enumerate $A_2(n)$ up to $n = 19$.
- In 1981 Redelmeier [Rede81] introduced a new enumeration algorithm. His new method, which was actually a procedure for subgraph counting, did not reproduce any of the previously-generated polyominoes. Thus he did not need to keep in memory all the already generated polyominoes, nor did he need to check if a newly generated polyomino was already counted. He implemented his method efficiently, and computed $A_2(n)$ up to $n = 24$. His program required about 10 months of CPU time on a PDP-11/70. Mertens and Lautenbacher [ML92] later devised a parallel version of Redelmeier's algorithm and used it for computing the number of polyominoes (of up to some size) on a triangular lattice.
- In 1995 Conway [Co95] introduced a transfer-matrix algorithm, subsequenetly used by Conway and Guttmann [CG95] for computing $A_2(25)$.
- Oliveira and Silva used (in an unpublished work) a parallel version of Redelmeier's algorithm to count polyominoes of up to size 28.
- Jensen [Je01] significantly improved the algorithm of Conway and Guttmann, and computed $A_2(n)$ up to $n = 46$. Soon Knuth [Kn01] applied (in an unpublished work) a few local optimizations to Jensen's algorithm and was able to compute $A_2(47)$. In a further computation Jensen [Je01a] claimed to also obtain $A_2(48)$.

Table 1.1 tabulates all the known values of $A_2(n)$.

n	$A_2(n)$	Ref.	n	$A_2(n)$	Ref.
1	1		25	20,457,802,016,011	[CG95]
2	2		26	79,992,676,367,108	
3	6		27	313,224,032,098,244	
4	19		28	1,228,088,671,826,973	
5	63		29	4,820,975,409,710,116	
6	216		30	18,946,775,782,611,174	
7	760		31	74,541,651,404,935,148	
8	2,725		32	293,560,133,910,477,776	
9	9,910		33	1,157,186,142,148,293,638	
10	36,446	[Rea62]	34	4,565,553,929,115,769,162	
11	135,268		35	18,027,932,215,016,128,134	
12	505,861		36	71,242,712,815,411,950,635	
13	1,903,890		37	281,746,550,485,032,531,911	
14	7,204,874		38	1,115,021,869,572,604,692,100	
15	27,394,666	[PLP67]	39	4,415,695,134,978,868,448,596	
16	104,592,937		40	17,498,111,172,838,312,982,542	
17	400,795,844		41	69,381,900,728,932,743,048,483	
18	1,540,820,542	[Lu71]	42	275,265,412,856,343,074,274,146	
19	5,940,738,676	[SG76]	43	1,092,687,308,874,612,006,972,082	
20	22,964,779,660		44	4,339,784,013,643,393,384,603,906	
21	88,983,512,783		45	17,244,800,728,846,724,289,191,074	
22	345,532,572,678		46	68,557,762,666,345,165,410,168,738	[Je01]
23	1,344,372,335,524		47	272,680,844,424,943,840,614,538,634	[Kn01]
24	5,239,988,770,268	[Rede81]	48	1,085,035,285,182,087,705,685,323,738	[Je01a]

Table 1.1: Numbers of two-dimensional fixed polyominoes

We are aware of only two attempts to count fixed polycubes:

- In 1975 Lunnon [Lu75] extended his algorithm [Lu71] to compute multi-dimensional polyominoes. He computed $A_3(n)$ up to $n = 12$.
- Sykes et al. [SGG76] used the method proposed by Martin [Ma74] in order to derive and analyse series expansions on a three-dimensional lattice.

Table 1.2 tabulates all the known values of $A_3(n)$.

Other related problems studied in the literature are the number of convex (resp., general) polyominoes of a given perimeter [KR74, DV84, Ki88] (resp., [Me90, ML92]), column-convex polyominoes [De88], polyominoes on other lattices (e.g., triangular) [RW81, ML92], polyominoes with holes [GJ$^+$00], etc.

It is known that $A_2(n)$ is exponential in n. Klarner [Kl67] showed that $A_2(n) \sim C\lambda^n n^\theta$ (for some constants $C > 0$ and $\theta \approx -1$), so that the limit $\lambda = \lim_{n\to\infty}(A(n+1)/A(n))$

n	$A_3(n)$	Ref.	n	$A_3(n)$	Ref.
1	1		10	8,294,738	
2	3		11	60,494,549	
3	15		12	446,205,905	[Lu75]
4	86		13	3,322,769,321	
5	534		14	24,946,773,111	
6	3,481		15	188,625,900,446	
7	23,502		16	1,435,074,454,755	
8	162,913		17	10,977,812,452,428	[IntSeq]
9	1,152,870		18	84,384,157,287,999	This thesis

Table 1.2: Numbers of three-dimensional fixed polyominoes

exists. Golomb [Go65] gave λ its well-known name *Klarner's constant*, out of which not even a single significant digit is known for sure. There have been several attempts to lower and upper bound λ, as well as to estimate it, based on knowing $A_2(n)$ up to certain values of n. The constant λ is estimated to be around 4.06 [CG95].

1.2 Redelmeier's Subgraph-Counting Algorithm

In this section we briefly describe Redelmeier's algorithm for counting polyominoes. The reader is referred to the original paper [Rede81] for the full details.

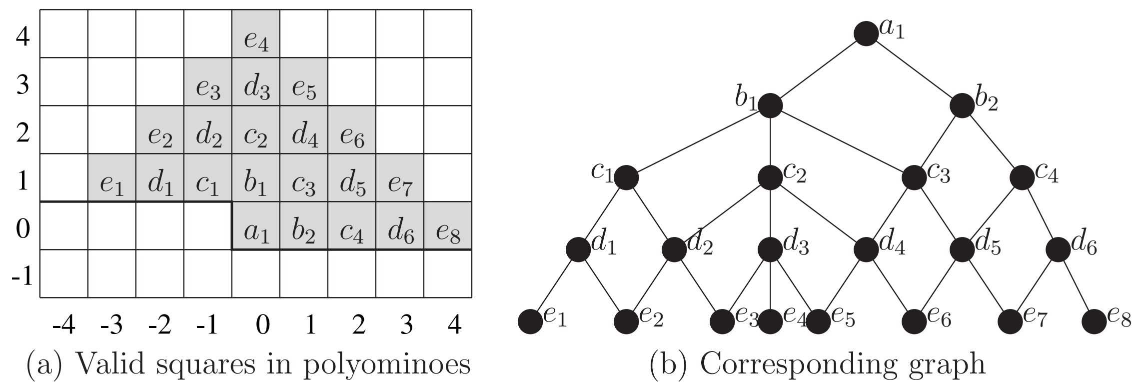
Redelmeier's algorithm is a procedure for connected-subgraph counting, where the underlying graph is induced by the square lattice. Since translated copies of a fixed polyomino are considered identical, one must decide upon a canonical form. Redelmeier's choice was to fix the leftmost square of the bottom row of a polyomino at the origin, that is, at the square (0,0). (Note that coordinates are associated with squares and not with their corners.) Thus we need to count the number of edge-connected sets of squares (that contain the origin) in

$$\{(x, y) \mid (y > 0) \text{ or } (y = 0 \text{ and } x \geq 0)\}.$$

The squares in this set are located above the thick line in Figure 1.3(a).

The shaded area in Figure 1.3(a) consists of the possible locations of squares of polyominoes of order 5. Counting these sets of squares amounts to counting all the connected subgraphs of the graph shown in Figure 1.3(b), that contain the vertex a_1.

The algorithm [Rede81] is shown in Figure 1.4. This sequential subgraph-counting algorithm can be applied to any graph, and it has the property that it never produces the same subgraph twice.



(a) Valid squares in polyominoes (b) Corresponding graph

Figure 1.3: Polyominoes as subgraphs of a specific base graph

```
Initialize the parent to be the empty polyomino, and the
untried set to contain only the origin.

  1. Remove an arbitrary element from the untried set.

  2. Place a cell at this point.

  3. Count this new polyomino.

  4. If the size is less than n:

     (a) Add new neighbors to the untried set.
     (b) Call this algorithm recursively with the new parent
         being the current polyomino, and the new untried
         set being a copy of the current one.
     (c) Remove the new neighbors from the untried set.

  5. Remove newest cell.
```

Figure 1.4: Redelmeier's algorithm [Rede81, p. 196]

1.3 Jensen's Transfer-Matrix Algorithm

In this section we briefly describe Jensen's algorithm for counting polyominoes. The reader is referred to the original paper [Je01] for the full details.

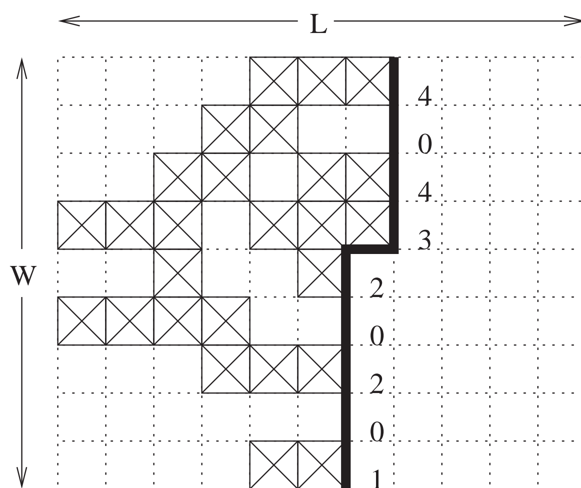
The algorithm counts separately all the polyominoes of size n bounded by boxes whose dimensions are W (its width, y-span) and L (its length, x-span). In each iteration of the algorithm different values of W and L are considered, for all possible values of W and L. (Due to symmetry only values of $W \leq L$ need to be considered.)

For specific values of W and L, the strategy is as follows. The polyominoes are built from left to right, and in each column from top to bottom. Instead of keeping track of all polyominoes, the procedure keeps records of the numbers of polyominoes with identical right boundaries. Hence, the right boundaries of the (yet incomplete) polyominoes are encoded by signatures as is described below. Expanding a polyomino is done in the current column, cell by cell, from top to bottom. The new cell is either occupied (i.e., belongs to the new polyomino) or empty (i.e., does not belong to it). Thus the right boundaries of the (yet incomplete) polyominoes have a "kink" at the currently considered cell. By "expanding" we mean updating both the signatures (possibly creating new signatures) and their respective numbers of polyominoes. For implementation purpose the numbers are maintained as polynomials in the form of generating functions: The terms of the polynomial $P(t) = \sum_i c_i t^i$ mean that c_i distinct (possibly incomplete) polyominoes of size i correspond to some signature.

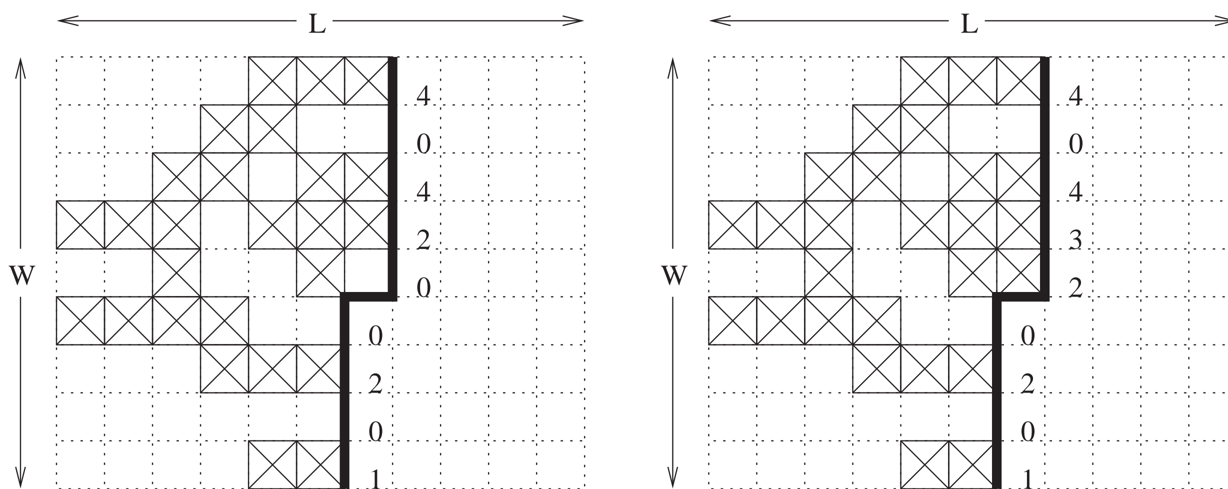
The right boundaries of (yet incomplete) polyominoes are encoded by signatures that contain five symbols: the digits 0–4. The symbol '0' stands for an empty cell. The symbol '1' stands for an occupied cell that is not connected by other cells to any other boundary cells. The other three symbols represent cells which are connected to other boundary cells. In case there are several boundary cells that are connected, either along the boundary or by cells to the left of it, the lowest cell is encoded by the symbol '2', the highest cell by the symbol '4', and all the other cells (if any) in that group by the symbol '3'.

Figure 1.5 (similar to Figure 1 of [Je01]) shows the signature of some (yet incomplete) polyomino, before and after cell expansion. Note that this polyomino is indeed incomplete because it has three disconnected components. Yet the algorithm needs to consider such intermediate configurations since the yet-to-be-added cells on the right may connect disconnected components and yield a legal polyomino. Only at the termination of the iteration does the algorithm need to check whether a signature corresponds to legal polyominoes. Otherwise the corresponding number of polyominoes (of that signature) is ignored and not counted. Also, note in the figure that the lowest '2' and the highest '4' in the signature encode boundary cells which are connected by cells to the left of the boundary.

There are several implementation details that are omitted here. One such detail is the initialization of an iteration, that is, building the set of signatures that correspond to (yet incomplete) polyominoes spanning only one column. Another detail is the exhaustive set of polyomino-expansion rules. In an expansion step the boundary kink is lowered by one by considering the kinked cell and either making it occupied or leaving it empty. The symbols encoding the cells adjacent to the kink, together with the fact of whether or not the new cell is occupied, determine how to update the signature. Most combinations require only local updates, whereas a few combinations require scanning the entire signature for updating. In addition, the signature needs to be expanded by two bits that indicate whether (yet incomplete) polyominoes touch or do not touch the top and bottom boundaries of the bounding rectangle. Some optional optimizations are also possible.



(a) Before cell expansion



(b.1) Kink cell empty (b.2) Kink cell occupied

(b) After cell expansion

Figure 1.5: A sample intermediate polyomino, its right boundary, and its signature (example taken from Figure 1 of [Je01])

Since only polyominoes of size n are sought, the signatures of intermediate polyominoes exceeding this size may be discarded. In addition, local rules can be applied to prune out intermediate configurations that cannot be completed into valid polyominoes. For example, it would save time to discard signatures of intermediate polyominoes that have more than one connected component, and whose union into one component will require a total of more than n cells. The same applies for signatures of intermediate polyominoes whose yet-to-be-constructed connections to all of the top, bottom, and right borders will require more than n cells in total.

At the termination of each iteration, the procedure discards all signatures that correspond to illegal polyominoes, such as disconnected polyominoes (as mentioned above), or polyominoes that do not touch all the boundaries of the bounding rectangle. Then all the numbers of polyominoes associated with legal polyominoes are summed up to yield the number of legal polyominoes bounded by some $W \times L$ rectangle. By iterating over all possible values of W and L and summing up, the algorithm computes the total number of polyominoes of size n. (Naturally, for one specific value of W, the iterations of the inner loop on values of L may be used to save execution time.)

1.4 Organization of the Thesis

This thesis is organized as follows. In Chapter 2 we analyze the running-time complexity of Jensen's algorithm. First we obtain the number of some class of strings, which play a significant role in the algorithm. It turns out that this number is closely related to Motzkin numbers [Mo48]. Jensen did not provide a rigorous analysis of the running time of his algorithm. Instead, he empirically estimated [Je01, §2.1.3] that it is proportional to $(\sqrt{2})^n$ by observing a graph that plots in a logarithmic scale the number of computed "signatures" (Section 1.3) as a function of n, the size of the sought polyominoes. As Jensen observed, this is indeed the dominant factor in the running time, but it is actually proportional to $(\sqrt{3})^n$ (multiplied by some polynomial in n). The full analysis shows that the running time of the algorithm is $O(n^{5/2}(\sqrt{3})^n)$.

In Chapter 3 we extend Jensen's [Je01] transfer-matrix algorithm. First we introduce a method by which we compute the number of polyominoes within a box that is twice the length of a given box. This is achieved by using the same data structure used by Jensen. We then go one step further and compute the same quantity *recursively*. To this aim we extend Jensen's data structure.

In Chapter 4 we describe how we use ideas from the extended data structure to develop a transfer-matrix algorithm for counting polycubes. In addition, we describe in Chapter 5 an efficient implementation of Redelmeier's serial algorithm [Rede81] for counting polycubes, and show results of up to order 18.

Chapter 2

Complexity of the Transfer-Matrix Algorithm

2.1 The Number of Different Signatures

In this section we analyze the number of signature strings that play the most significant role in Jensen's transfer-matrix algorithm for counting polyominoes. It turns out that this number is closely related to Motzkin numbers.

2.1.1 Introduction

We begin by investigating the number of strings of length W that are defined as follows. Consider a square lattice of width W and *unrestricted* height h, in which some of the $W \cdot h$ cells are occupied and the other cells are empty. Identify the connected components of the occupied cells, where connectivity is through edges only. Assign a symbol to each connected component and a special symbol '-' to empty cells, and regard the lowest row of the lattice as the "signature" string representing the configuration of occupied cells. The completely empty string (containing only '-'s) is not allowed. Obviously, a signature may represent many different configurations. Moreover, the number of distinct symbols needed to specify all signatures is $\lceil W/2 \rceil + 1$, since there may be at most $\lceil W/2 \rceil$ connected components. Figure 2.4(a) shows a lattice configuration and its representing signature string. The question is, then, what is $J(W)$, the number of different signature strings (as a function of W, up to renaming of the non-'-' symbols).

The signature strings defined above play an important role in an efficient transfer-matrix algorithm presented by Jensen [Je01] (see Section 1.3). This algorithm was the first method for counting fixed polyominoes without actually generating them. The algorithm uses a complex set of rules for expanding the polyominoes cell by cell, from left to right

in columns, and in each column, from top to bottom. The main idea is to keep only the right *boundaries* of all the configurations of cells, where each boundary is associated with the number of all configurations that have that boundary. Jensen's notion of a (vertical) boundary is equivalent to our (horizontal) signature string. Naturally, during the course of the algorithm, the intermediate partially-built polyominoes are not necessarily valid, i.e., they consist of more than one connected component. For this purpose the boundary, that is, the signature, encodes the connected components. At the termination of the algorithm only signatures that correspond to legal polyominoes are considered and their associated numbers are summed up. Thus, most of the polyominoes are not generated explicitly; large sets of polyominoes are traced by representative signatures. In addition, the completely empty boundary is not allowed, since in the corresponding configurations occupied cells to the left of the boundary column cannot be connected to columns to the right of the boundary.

The running time of Jensen's algorithm has two factors: a polynomial in W, and $J(W)$, the number of distinct signatures. Jensen did not provide an analysis of the running time of his algorithm. Instead, he showed a graph that plots $J(W)$ for $1 \leq W \leq 23$. The graph was drawn in a half-logarithmic scale, and the plotted points seemed visually to be more-or-less located around some line. From the slope of this imaginary line, Jensen estimated that $J(W)$ was proportional to 2^W. For counting fixed polyominoes of size n Jensen's algorithm needed to consider signatures whose length was at most $n/2$. Therefore, Jensen concluded that the running time of his algorithm was proportional to $(\sqrt{2})^n$ times some polynomial factor. In this chapter we show that $J(W)$ actually equals $M(W+1)-1$, where $M(W)$ is the Wth Motzkin number [Mo48]. Since the Wth Motzkin number is proportional to 3^W (neglecting some polynomial factor), the running time of Jensen's algorithm is actually proportional to $(\sqrt{3})^n$ times some polynomial factor.

2.1.2 The Theorem

We now prove the relation between the number of signature strings and Motzkin numbers.

Theorem 1 *$J(W) = M(W+1) - 1$.*

Proof: Let $J^*(W) = J(W)+1$, that is, the number of all legal signatures plus the unique empty signature (-,-,-,...,-) that is illegal in the context of polyominoes. We will now evaluate $J^*(W)$. The number of signature strings whose last character is '-' is simply $J^*(W-1)$. Otherwise (when the last character is not '-'), the corresponding last cell of the signature may be connected (through the boundary and/or the area to the left of the boundary, which in our notation is the area on top of the signature) to other signature cells. Let i be the lowest index (counting from left to right) of the signature cell connected to the Wth cell (obviously $1 \leq i \leq W$). For $i \geq 2$, the $(i-1)$st cell must be empty, and the number of distinct signatures is the product of the number of subsignatures of the

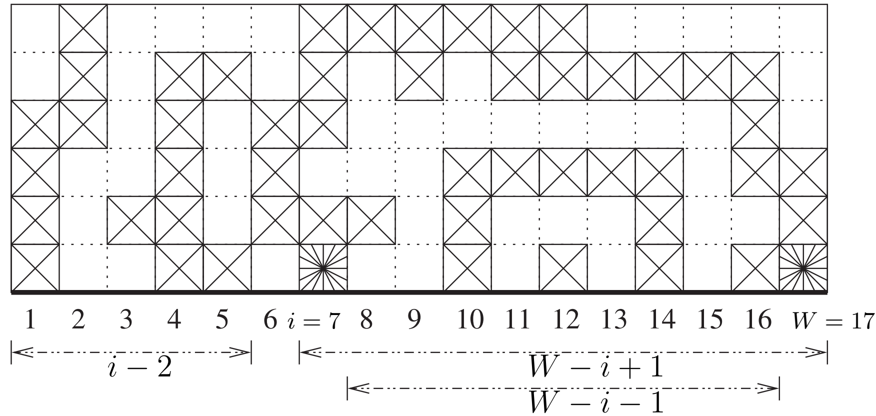


Figure 2.1: The leftmost signature cell connected to the Wth signature cell (when the latter is occupied)

leftmost $i-2$ cells and the number of subsignatures of the rightmost $W-i+1$ cells (of which the leftmost and rightmost are occupied by definition, so there is freedom only in the middle $W-i-1$ cells). Figure 2.1 shows an example in which $W = 17$ and $i = 7$. (Recall that cell connectivity is through edges only!)

Thus

$$J^*(W) = J^*(W-1) + \sum_{i=1}^{W} (J^*(i-2) \cdot J^*(W-i-1)),$$

with the convention $J^*(-1) = J^*(0) = 1$. It is easily seen that this is exactly the recurrence of the Motzkin series (with a shift of one index):

$$M(k) = M(k-1) + \sum_{i=0}^{k-2} (M(i) \cdot M(k-i-2))$$

for $k \geq 2$ and $M(0) = M(1) = 1$. It follows immediately that $J(W) = J^*(W) - 1 = M(W+1) - 1$. Indeed, $\{J(W)\}_{W=1}^{\infty} = \{1, 3, 8, 20, 50, 126, \ldots\}$ while the first few Motzkin numbers are $\{M(k)\}_{k=0}^{\infty} = \{1, 1, 2, 4, 9, 21, 51, 127, \ldots\}$. □

Figures 2.2 and 2.3 show all the different possible boundaries and their signatures for $1 \leq W \leq 5$.

It is easy to see that $M(k) < 3^k$ since $M(k)$ is also the number of paths from $(0, 0)$ to $(k, 0)$ in a $k \times k$ grid, that use only the steps $(1, 1)$, $(1, 0)$, and $(1, -1)$, and do not go under $y = 0$. On the other hand, it is well-known that $M(k) > 3^{(1-o(1))k}$. In fact, $M(k)$ is asymptotically 3^k divided by some polynomial factor in k.

An alternative proof for Theorem 1 is by showing a bijection between signature strings and Motzkin paths. Consider the edges between boundary cells. Edges between occupied cells of the same connected component (or between empty cells) are mapped to the

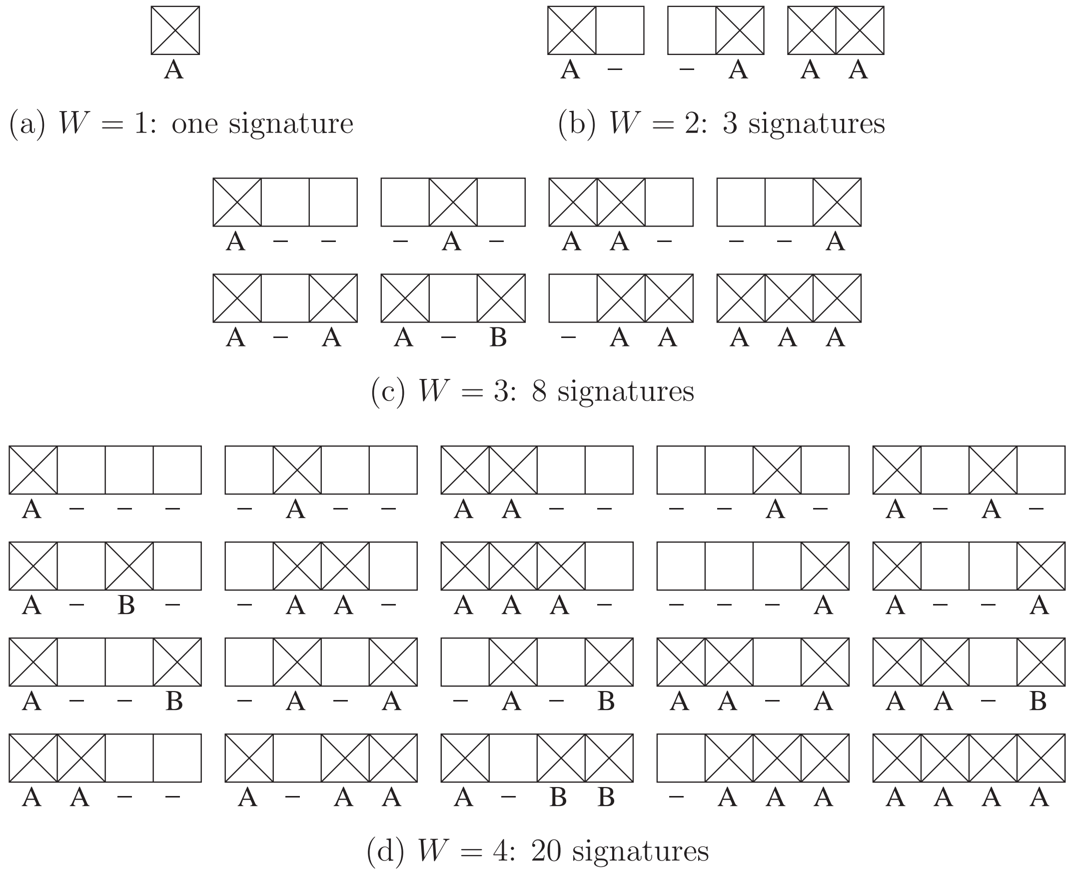


Figure 2.2: All possible signatures for $1 \leq W \leq 4$

step $(1, 0)$. For a block of consecutive occupied cells of the same connected component, distinguish between four cases:

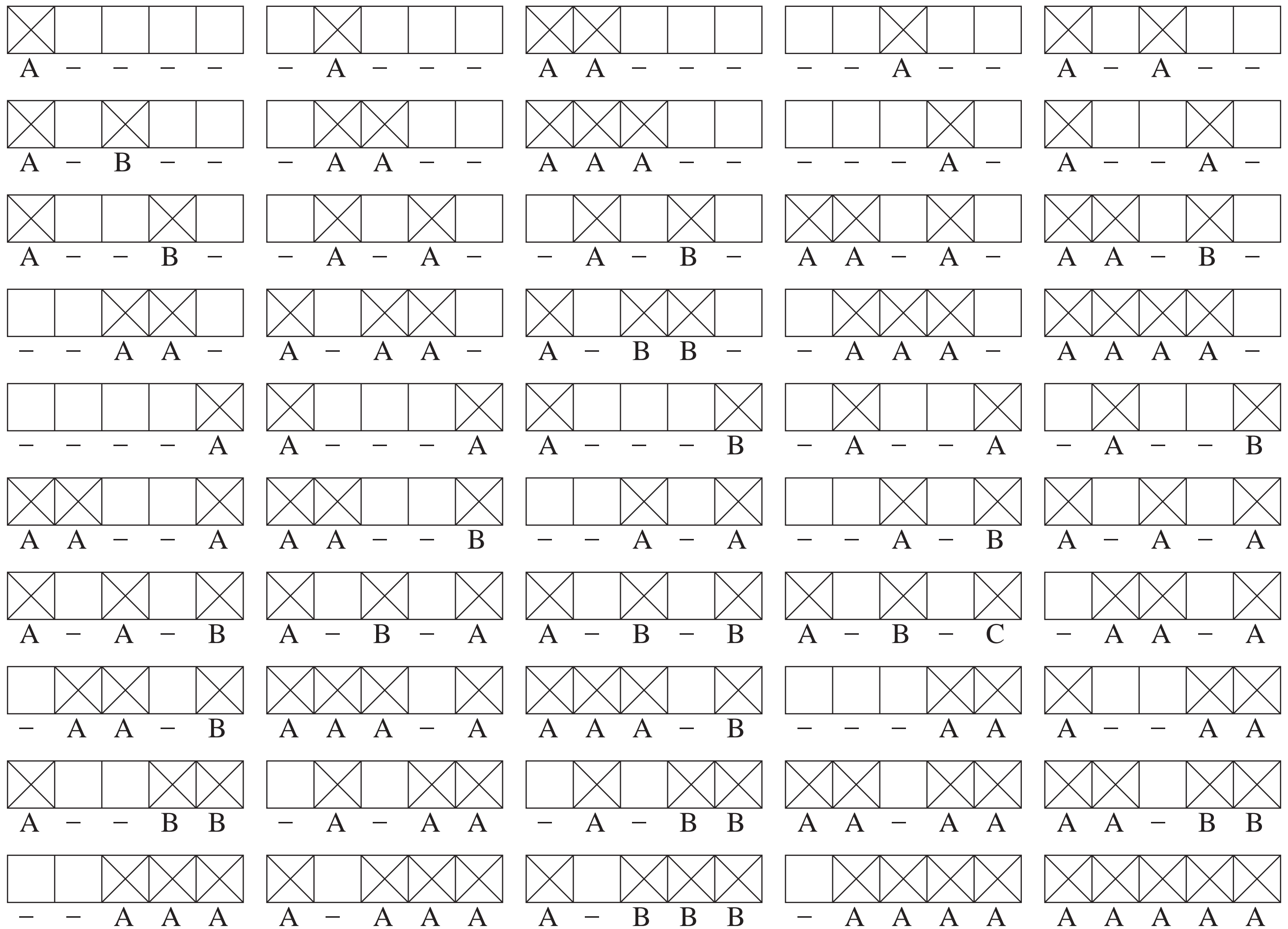
1. If there exists only one block (of the same component), then its left (resp., right) bounding edge is mapped to the step $(1, 1)$ (resp., $(1, -1)$);

Otherwise (in case there are at least two blocks):

2. For the first block both bounding edges are mapped to the step $(1, 1)$;
3. For an intermediate block (if any), the left (resp., right) bounding edge is mapped to the step $(1, -1)$ (resp., $(1, 1)$);
4. For the last block both bounding edges are mapped to the step $(1, -1)$.

Figure 2.4 shows an example of this bijection.

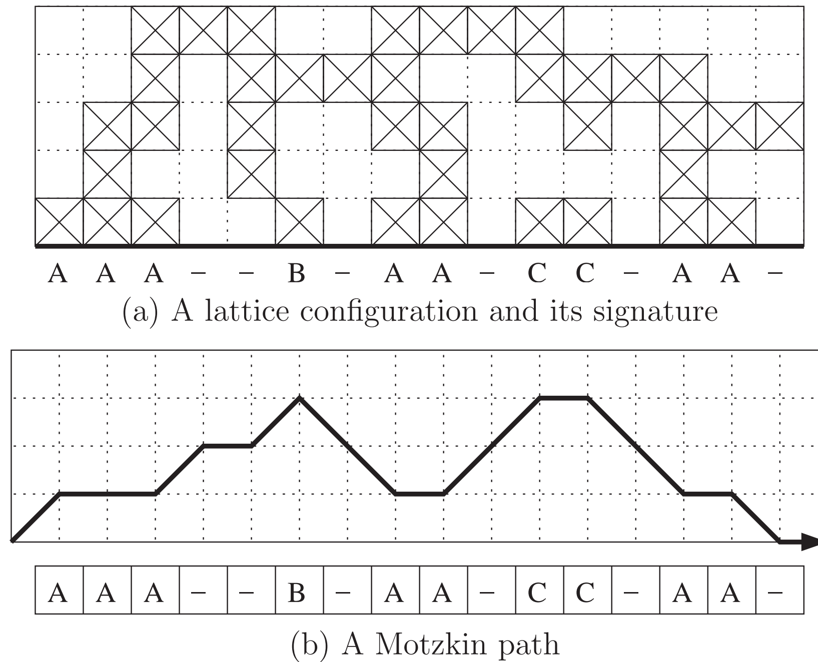
It is rather easy to prove that this is a bijection. By definition different strings correspond to different paths, and vice versa. We only need to verify that every signature



(e) $W = 5$: 50 signatures

Figure 2.3: (Continuation of Figure 2.2) All possible signatures for $W = 5$

string is mapped to a valid path, and that every path can be realized by some configuration and its respective string. To this aim we note that the order of start and end of blocks (of the same connected component) along the signature must be a legal parentheses sequence. Otherwise the borders of the connected components would intersect. On one hand, this guarantees, following directly from the definition of the bijection, that a signature is always mapped to a path that does not go below the x-axis. On the other hand, it ensures that every path is realizable. Every path is uniquely mapped into a string (by applying the bijection rules backward), and since it is a legal parentheses sequence, a cell configuration represented by this signature can easily be constructed, e.g., by building rectilinear "bridges" connecting between blocks of the same symbol, supported by "legs" standing on top of the left cell of each block, and high enough to allow lower bridges found between legs supporting the same bridge. Figure 2.5 shows the realization of the string shown in Figure 2.4.



(a) A lattice configuration and its signature

(b) A Motzkin path

Figure 2.4: Mapping a signature to a path

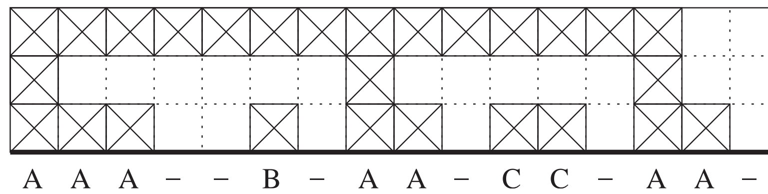


Figure 2.5: String realization

Finally we explain the index shift between the number of signature strings and Motzkin numbers. That is, why $J(W) = M(W\mathbf{+1}) - 1$. This is simply because a signature of length k is mapped to a path of length $k+1$, while the $(k+1)$st Motzkin number is the number of paths of length $k+1$. The missing string is the empty string (-,-,-,...,-), that corresponds to the straight x-parallel path. The reader can also observe the resemblance between signature strings to drawing chords in an outer planar graph, which is yet another way to describe Motzkin numbers.

2.2 Computational Complexity

In this section we closely follow the notation of [Je01]. Denote by n the size of the polyominoes whose number we intend to compute. Overall, we have k iterations of Jensen's algorithm that differ in the dimensions of the bounding box of the polyominoes counted in each iteration. Denote by W (resp., L) the width (resp., length), that is, the y- and

x-span, respectively, of a bounding box in one iteration of Jensen's algorithm. In each such iteration W is at most $W_{\max}$ (the maximum width needed to be considered), and $L = 2W_{\max} + 1 - W$, thus $L_{\max}$ (the maximum length) is at most $2W_{\max}$. In each iteration of the algorithm we expand the so-far traversed lattice by $W \cdot L$ cells. Each such expansion consists of setting one cell to either an occupied or empty state. This involves the consideration of all possible signatures σ for the expansion. Denote the total number of possible signatures by N_{Conf}. Each individual cell-expansion step requires the following operations:

- Fetching a signature from the database of signatures;
- Computing a new signature that is a function of the old signature and determining whether the new cell is occupied or not;
- Updating the polynomial that counts the number of possible distinct configuration for the specific signature; and
- Searching for the new signature in the database. If it did not previously exist, insert it and its polynomial. Otherwise, replace the existing polynomial by the sum of the existing and the new polynomial.

We denote by h the time needed for random access into the signatures database, by s the time for computing a new signature, and by u the time needed for updating a polynomial. Clearly, the over-all running time of the algorithm is

$$O(k \cdot W_{\max} \cdot L_{\max} \cdot N_{\text{Conf}} \cdot (h + s + u)).$$

Obviously, $W_{\max} = \lceil n/2 \rceil$ and $L_{\max} = n$. Therefore, $W_{\max}, L_{\max} = O(n)$. In the original description of the algorithm, $k = W_{\max} \cdot L_{\max} = O(n^2)$. However, instead of two nested loops for W and L, one can have only one loop on W with only $L = L_{\max}$. After every processed column L, one can check all the configurations. The polynomials representing the respective numbers of polyominoes of all the valid configurations (those with one connected component spanning the entire $W \times L$ box) will update the number of animals of the appropriate sizes. Thus we can improve k by a factor of n and have $k = O(n)$.

If we implement the signatures database as a perfect hashing table (with the signature as a key), we can expect each access to the database to depend (in the average case) only linearly on the size of the signature, independently of the number of signatures stored in the database. That is, $h = O(W_{\max}) = O(n)$.

It follows directly from the description of the algorithm that $s = O(W_{\max}) = O(n)$. Indeed, most of the signature updates are local, and are reflected by $O(1)$ operations in the vicinity of the expanded cell. Only a few update rules require the traversal of the entire signature (whose complexity is $O(n)$) and a few global updates of it. A typical

such update is required when the expanded cell touches the top boundary cell of some connected component, and we either need to find the second-to-top, or even the bottom boundary cell of the same component. For this purpose we traverse the signature top to bottom and count top and bottom cells of connected components until they balance appropriately, providing enough information for identifying the sought boundary cell. The running time of this traversal is linear in n.[1]

The polynomial operations required by Jensen's algorithm are the addition of two polynomials $P_1(t), P_2(t)$ of maximum degree n and multiplying a polynomial $P(t)$ of maximum degree n by t. (The latter operation implements the addition of one occupied cell.) Clearly each such operation requires $O(n)$ time, and since each cell-expansion operation requires a constant number of such polynomial operation, we have $u = O(n)$.

Most of our effort is, therefore, targeted at providing an upper bound on N_{Conf}. We use here the notation $N_{\mathrm{Conf}}(W)$ to denote the number of signatures of length W. We ignore the two bits of the signature that indicate whether the respective set of polyominoes touch or do not touch the top and bottom boundaries of the bounding box, because these bits add only a constant factor of 4 on the number of different signatures.

In Secion 2.1.2 we showed that the number of signatures without a kink is exactly $M(W + 1) - 1$, where $M(k) = \Theta(3^k/k^{3/2})$ is the kth Motzkin's number. Now consider boundaries with a kink. At most, this applies a constant factor on the number of signatures. To see this, regard the length-W signatures as a subset of all the signatures of length (W + 1) obtained by adding the kink cell to the boundary as an empty cell, and then dropping from the signature the symbol '0' from the position of the kink. By dropping these kink symbols different signatures may be identified, but this only helps. Thus we conclude that $N_{\mathrm{Conf}}(W) = O(3^W/W^{3/2}) = O((\sqrt{3})^n/n^{3/2})$.[2]

Summing everything up, we obtain $O(n^4 M(n/2)) = O(n^{5/2}(\sqrt{3})^n)$ as an asymptotic bound on the running time of Jensen's algorithm. As mentioned earlier, Jensen's algorithm also prunes out signatures of boundaries that can never close into a legal polyomino because, for example, the number of cells remaining to be occupied is insufficient for the polyomino to touch both upper and lower boundaries of the bounding box, or because there are not enough cells to connect all occupied cells into one connected component. Obviously, most of the signatures are pruned out when W and L are close to $n/2$. This is probably the reason for the difference between the $(\sqrt{3})^n$ term in our bound and the $(\sqrt{2})^n$-like behavior observed empirically by Jensen.

[1] We believe that with a little effort, and with applying only a constant multiplier on the amount of memory needed to store a signature, we can show that $s = o(n)$ by using an appropriate disjoint-sets data structure. However, this improvement is asymptotically negligible since s is dominated by h and u.

[2] We believe that it is easy to obtain a sharper upper bound on the number of signatures with kinks, but since $M(k) = \Theta(M(k+1))$ (as $\lim_{k\to\infty} M(k+1)/M(k) = 3$) we have $N_{\mathrm{Conf}}(W) = \Theta(M(W+2)) = \Theta(M(W))$, and the exact constant of proportionality is of no interest.

Chapter 3

Attempts to Improve the Transfer-Matrix Algorithm

In this chapter we introduce two extensions of Jensen's algorithm. Both extensions use previously-computed information. Jensen's algorithm counts separately all the polyominoes of size n bounded by boxes whose dimensions are W (its width, y-span) and L (its length, x-span). In each iteration of the algorithm different values of W and L are considered, for all possible values of W and L. Our extensions aim to reduce the number of iterations of the algorithm.

In particular, when considering a box whose dimensions are $W \times L$, the first extension uses the information computed while considering a $W \times L/2$ box to directly compute information required for computing all polyominoes of size n bounded by a $W \times L$ box. The second extension goes even further. We obtain more information during an iteration of the algorithm. This information can then be used in a *recursive* manner to compute information needed for computing all polyominoes of size n bounded by twice as long a box. For example, consider the information computed while considering a $W \times L$ box. We can use it to directly compute information needed for computing all polyominoes of size n bounded by a $W \times 2L$ box. Next, we can use the latter information to directly compute the information needed for computing all polyominoes of size n bounded by a $W \times 4L$ box, and so on.

3.1 Doubling the Bounding Box

As explained in Section 1.3, the algorithm counts separately all the polyominoes bounded by boxes whose dimensions are W (y-span) and L (x-span). Each such box is represented by a set of signatures and their respective polynomials (exactly as in Jensen's algoritm). Given two such sets (with the same width), we present a method for computing the number of polyominoes bounded by a larger box of the same width and whose length is

the sum of lengths of the two boxes. Thus, when the input boxes are identical in both length and width, the resulting box is of the same width and twice the length.

First, we consider all signature pairs, one signature representing the right boundary of the left box, and the other signature representing the left boundary of the right box. We then examine each such pair separately, as described in Section 3.1.1, and consider if the partially-built polyominoes with their respective boundaries can be *combined* into valid polyominoes. Each pair that passes this test is then processed to compute the number of polyominoes in the new combined bounding box. we repeat this procedure for all pairs of signatures, summing up and obtaining the total number of polyominoes contained in the combined bounding box.

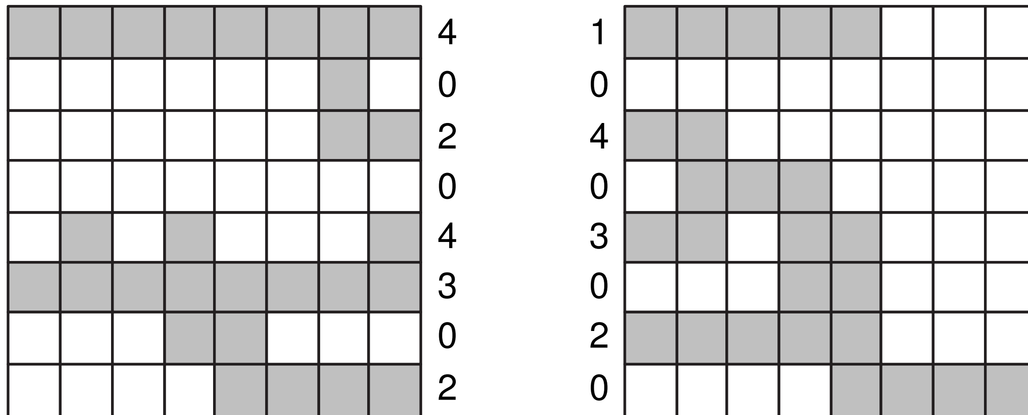
Figure 3.1 shows an example of a pair of signatures that can be combined to form polyominoes contained in the combined box. Note that the sets of the right (resp., left) boundaries of the polyominoes and the left (resp., right) boundaries are identical. Therefore, we compute the set of signatures corresponding to only one such box, and simply consider all pairs of signatures.

Although this method allows us to compute the number of polyominoes bounded by the combined box, we cannot repeat this step recursively because we "lose" the signatures during the concatenation of the boxes. This happens as a result of the input to this step which maintains the signature of only *one* boundary of each of the bounding boxes. When the boundaries are connected facing each other (see Figures 3.1), no boundary of the combined box is attributed by a signature. In Section 3.2 we suggest a remedy for this. Meanwhile, we proceed with describing how exactly we combine signatures.

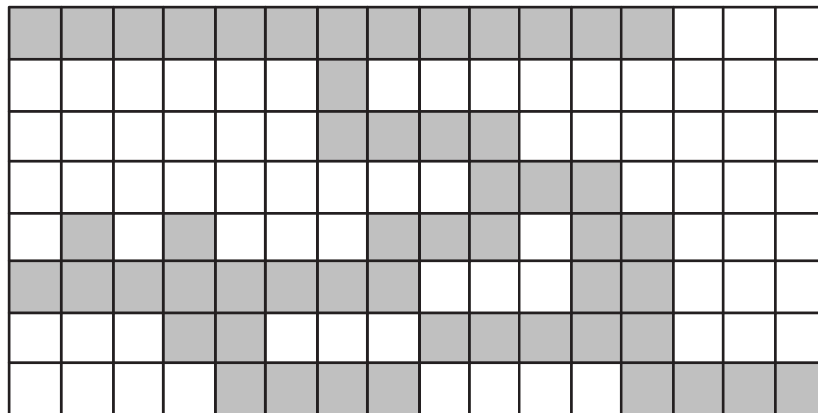
3.1.1 Combining Signatures

Each signature characterizes a set of configurations contained in a bounding box, with a unique boundary (the right boundary of the left configuration and the left boundary of the right configuration, as shown in Figure 3.1(a)). Each configuration is a collection of occupied sites that may or may not be connected. The respective polynomial of a signature represents the number of different configurations that match the boundary signature.

Refer again to Figure 3.1(a) that shows two boxes with sample configurations. The combined configurations may have different signatures. The signature of the left configuration encodes its right boundary, whereas the signature of the right configuration encodes its left boundary. By joining the two configurations we can create a new combined configuration with the cumulative length. Figure 3.1(b) shows the combined configuration.



(a) two configurations



(b) new combined configuration

Figure 3.1: Combining two configurations into one

In the example shown in Figure 3.1 the new configuration is a legal polyomino. In case the new configuration is not a legal polyomino, it is discarded. The new configuration is legal only if it meets two conditions: all occupied sites are connected, and all box edges are touched. If both conditions hold, the new configuration consists of a legal polyomino.

The encoding of a signature is sufficient to determine whether the entire respective set of configurations (represented by that signature) is legal. This is easy to do since the signature's encoding holds both the connectivity of all connected components that touch the boundary, and the edges of the box that are touched. Note that any intermediate configuration with a disconnected component not touching the boundary is immediately discarded since this component will never be connected to other components. This is done already in the original algorithm.

We argue that we have enough information about the connectivity in each box, and about touched box edges, to be able to determine whether the combined signatures and their respective sets of configurations are legal. In the combined configurations, the orig-

inal boundaries lie next to each other (see Figure 3.1(b)) and new connections between components may be made. We need only examine whether the sites of the combined configurations form one component. The connectivity of the combined configurations is resolved using a simple DFS-like (Depth First Search) step. Finding out whether all the edges of the combined box are touched is trivial. The left and right edges are always touched since the left and right edges of both configurations are always touched (otherwise we would have created a disconnected configuration). Therefore, in the combined box both the left and right edges are also touched. If, in addition, both the top and bottom edges have been touched in either of the original boxes, then all edges in the new box are touched.

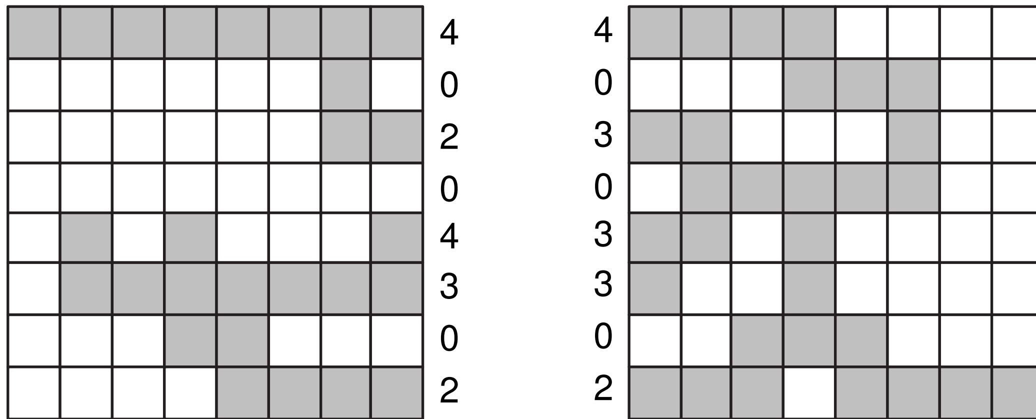
When the combination of two signatures corresponds to a legal set of configurations (and hence a legal set of polyominoes), we also need to compute the respective new polynomial that counts the number of legal polyominoes. Since each of the original signatures had a respective polynomial representing the number of polyominoes (of different sizes) with the same signature, the product of these two polynomials is the appropriate representation of the number of the combined polyominoes and their sizes.

3.1.2 Optimization

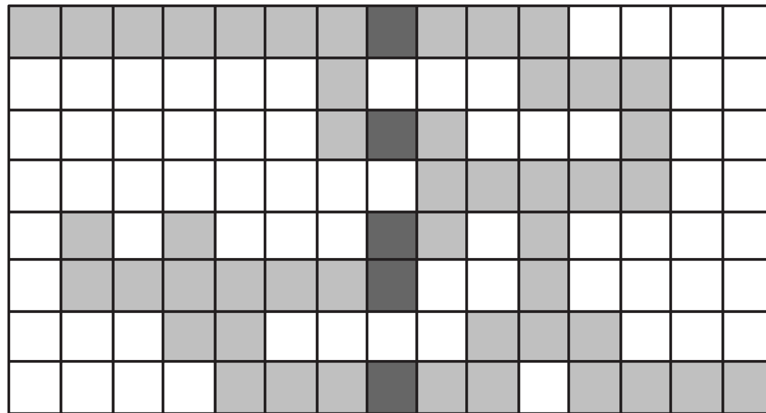
Instead of considering all pairs of signatures from both boxes, whose number is (see chapter 2.2) $O((N_{\text{Conf}}(W))^2) = O((3^W)^2) = O(9^W)$, we may consider signatures with the same occupied sites, that is, combine two bounding boxes with one overlapping column. This allows us to consider fewer combinations (as we show below), for the modest price of having a combined box whose length is smaller by 1. Figure 3.2(a) shows two configurations in two boxes, in which the same sites are occupied in the right (resp., left) boundary of the left (resp., right) box. (Obviously, the two respective signatures need not be identical.) By joining the configurations in a one-column overlap manner, we can create the new configuration shown in Figure 3.2(b). The rest of the algorithm is unchanged.

We provide here only a rough estimate of the amount of work performed by this optimization, since it was still inferior to the original implementation of Jensen's algorithm (which considered $O(3^W)$ signatures of length W per expansion step).

The number of different boundaries (sequences of sites, regardless of their respective signatures) is obviously 2^W (each site can either be occupied or empty). However, each boundary can correspond to multiple signatures. We therefore attempt to bound this multiplicity.



(a) Two signatures (before combination)



(b) After combination

Figure 3.2: Combining two configurations with an overlapping column

The worst-case scenario is when all the $O(3^W)$ (or rather, $O(3^W) - 2^W + 1 = O(3^W)$) signatures correspond to one specific boundary, allowing all the other boundaries only one possible signature. (This is, of course, not a practical situation, but we give this computation to show the perspective of this issue.) In such a theoretical case the number of combinations that we will consider is $(2^W - 1) \cdot 1 + 1 \cdot O((3^W)^2) = O(9^W)$.

It is easily seen that the best case would be if the signatures were distributed evenly between the different boundaries. In such a case the average number of signatures per boundary would be $O(3^W)/2^W = O(1.5^W)$. In such a case the number of considered combinations would be $2^W \cdot O((1.5^W)^2) = O(4.5^W)$.

In practice, the signatures are not distributed evenly between boundaries. The more "blocks" of consecutive occupied sites the boundary has, the more signatures to which it can correspond (since then the number of possible connections between these blocks is larger). A boundary of length W can have at most $\lceil W/2 \rceil$ "blocks," since each block (except the first and last) is preceded and followed by at least one empty site. The

number of occupied sites in such boundaries is equal to the number of blocks, and each such site can be encoded by any of the symbols $\{1, 2, 3, 4\}$. Thus, the maximum number of signatures is $4^{W/2} = 2^W$. (An alternative explanation is through Catalan numbers: Determining the signatures that correspond to $W/2$ blocks can be carried out by setting legal parentheses sequences of length at most W. The number of distinct parentheses sequences of length W (which outnumber all shorter sequences) is the $W/2$-th Catalan number, which is $O(4^{W/2}) = O(2^W)$.) Therefore, the maximum number of considered combinations is $2^W \cdot (2^W)^2 = 8^W$.

3.2 Multi-Level Combination of Bounding Boxes

The main drawback of the extension described in the previous section is the fact that it cannot be repeated in a divide-and-conquer manner. Although the number of polyominoes bounded by the combined box is computed, a new set of signatures cannot be computed. This prevents us from applying the method recursively. The reason for this is that the original signatures hold data only about occupied sites in *one* (right or left) boundary of each of the bounding boxes.

To enable repeated joining of the information computed while considering different bounding boxes, we will need to keep information about both vertical boundaries of each bounding box. In addition, we will need to keep some information about the connectivity of all connected components lying between both boundaries. This will allow us to compute the boundary signatures of the combined bounding box, and then the new connectivity information between the new boundaries.

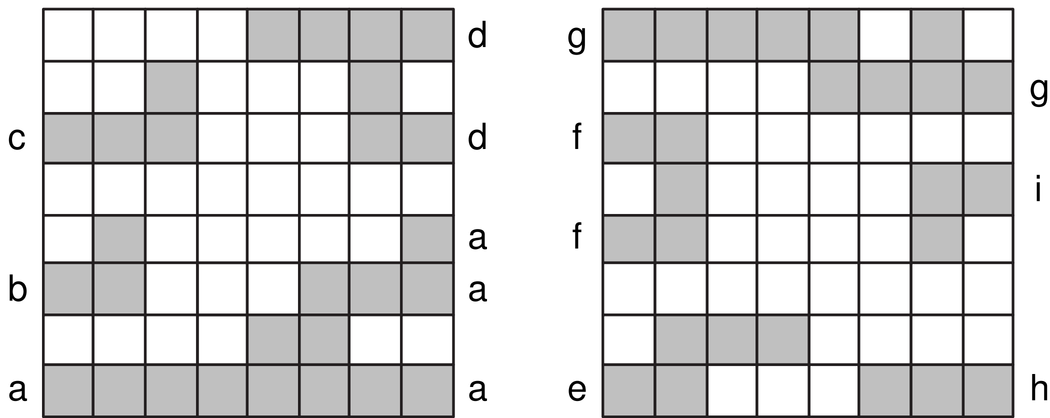
We extended Jensen's encoding by a less-efficient boundary encoding which holds more information. We assign an ID number to each occupied site in both left and right boundaries. The ID numbers are allocated according to the connected components, one ID per component. Thus, occupied sites in both boundaries that belong to the same connected component are assigned the same ID. We call our new data structure an extended signature.

The box-combination algorithm is very similar to the algorithm used in the first extension. We consider all pairs of extended signatures representing the left (resp., right) boundary of the right (resp., left) box. We examine each such pair as described next and determine whether the corresponding configurations can be combined. Each pair that passes this test is used to create two new extended signatures, describing the new left and right boundaries of the combined bounding box.

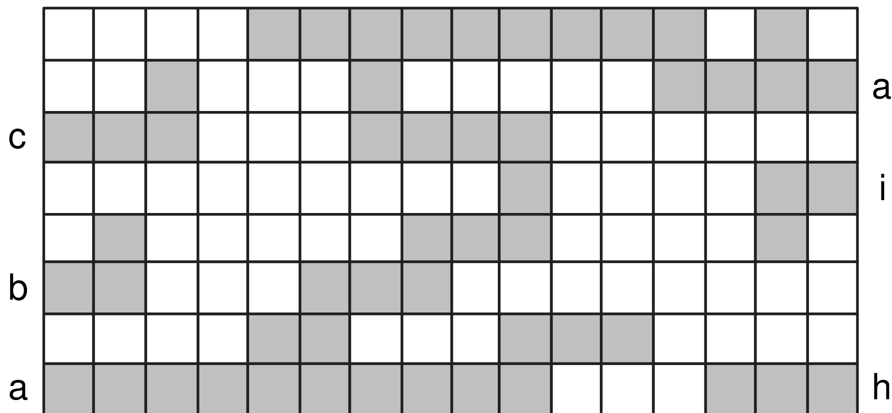
Each pair of extended signatures is joined in a fashion very similar to that of combining the regular signatures, presented in Section 3.1.1. The main difference is that the combined box is not necessarily the target box. Therefore, we have to allow the new box to contain illegal (partially built) polyominoes. This is so that the polyomino may

become legal through a further combining step. When the final box is created, only legal polyominoes are counted. Recall that legal polyominoes are those that consist of only one connected component and touch all four edges of the box. Figure 3.3(a) depicts the combination of two extended signatures of two boxes into one box. (The IDs are shown as letters just to distinguish the extended signatures from the regular signatures).

The encoding of the extended signature is used both to determine whether the new configuration is legal and to compute the new left and right boundary signatures. Determening if the set of configurations is legal is done similarly to the procedure applied for regular signatures. The additional information is used to compute the new left and right boundaries. To do this we observe that the occupied sites of the left (resp., right) boundary of the combined box are exactly the same as those of the left (resp., right) boundary of the original left (resp., right) box. The difference in the signatures is only due to the connections established between those sites. By employing a simple DFS procedure on



(a) Two extended signatures before making the connection



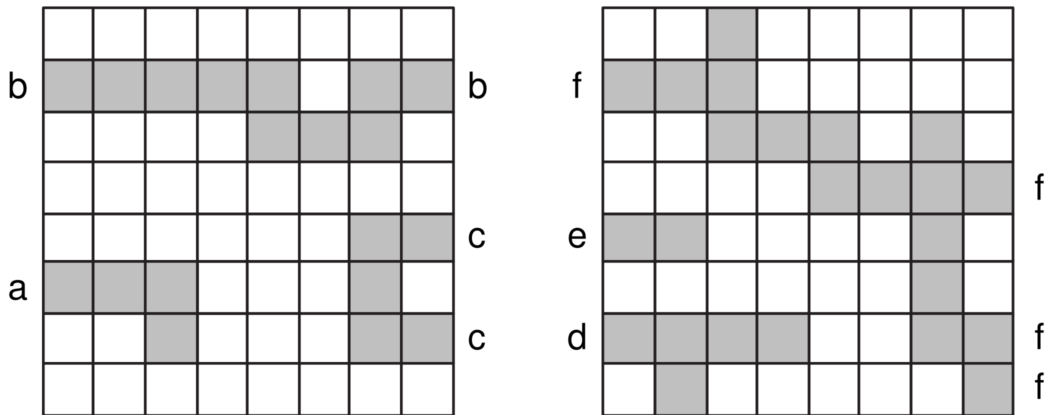
(b) After making the connection

Figure 3.3: Combining two extended configuration into one

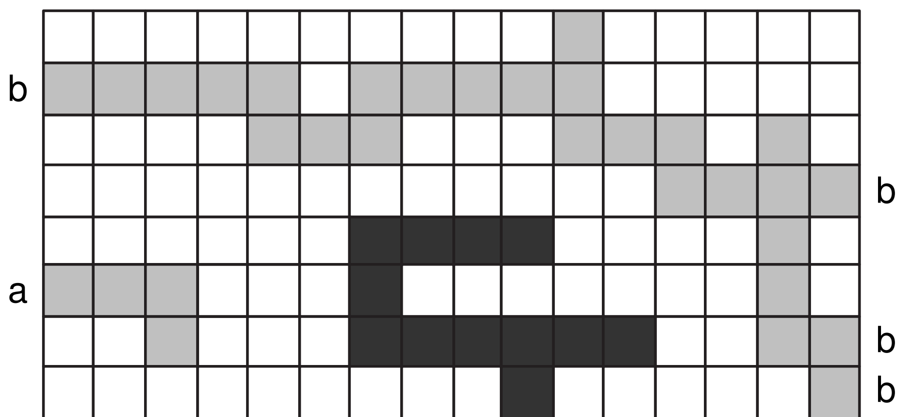
the joined boundaries (the right boundary of the left box and the left boundary of the right box), we compute the connected components of the combined configuration and assign each such component a new unique ID. The new IDs are then propagated to the left and right boundaries of the combined box. Figure 3.3(b) shows the result of combining extended signatures. The result is also an extended signature. Note the new encoding (with IDs) given for each site in both boundaries.

Any component not touching any vertical boundary will never be connected to the other components. This causes the combined configuration to be illegal; thus, it is discarded immediately. In Figure 3.4 we show two legal configurations being joined into an illegal configuration. The created disconnected component is shown in darker gray.

The optimizations of the previous extension can also be implemented. For the more involved recursive combining.



(a) Two signatures before making the connection



(b) After making the connection, an illegal combined configuration

Figure 3.4: Combining two extended configurations into an illegal configuration

3.3 Advantages and Disadvantages

The main drawback of both extensions is the fact that they do not address the bottleneck of the transfer-matrix algorithm, which is its memory consumption. A huge amount of memory is used for storing the signatures created during the iterations of the algorithm. The required amount of memory depends linearly on the number of signatures $N_{\text{Conf}}(W)$.

The first extension uses the same amount of memory as the original Jensen's algorithm (up to minor implementation details). In our experiments we did not encounter any benefit of using this extension. In the second extension, not only does the size of each extended signature increase, but the number of extended signatures is of much higher order than that of the original signatures (although some configurations are found illegal and hence discarded). Therefore, by using this extension we run out of memory faster than when we run Jensen's original algorithm.

The second extension might prove useful in the future, when more memory is available. In such a case, larger bounding boxes will probably be considered. For some of those bounding boxes, namely bounding boxes that are short and long ($W \ll L$), we expect the extension to be useful. In these cases we will be able to compute the L-long bounding box in $\log_2(L)$ complex steps instead of $W \cdot L$ original steps.

In Table 3.1 we plot the number of signatures and extended signatures computed in different bounding boxes. These signatures and extended signatures were created for polyominoes of up to size 50.

Width of bounding box	Length of bounding box	Number of original signatures	Number of extended signatures
3	1	7	7
	2	15	37
	4	15	101
	8	15	101
	16	15	101
4	1	15	15
	2	39	175
	4	39	815
	8	39	823
	16	39	823
5	1	31	31
	2	97	781
	4	98	6,165
	8	98	6,544
	16	98	6,544
6	1	63	63
	2	237	3,367
	4	246	44,881
	8	246	51,644
	16	246	51,644

Table 3.1: Numbers of original and extended signatures

Chapter 4

Extending Jensen's Algorithm to Three Dimensions

In this chapter we extend Jensen's algorithm to counting three-dimensional polycubes. Recall that Jensen's algorithm counts separately all the polyominoes of size n bounded by boxes whose dimensions are W (its width, y-span) and L (its length, x-span). In each iteration of the algorithm different values of W and L are considered, for all possible values of W and L. When counting polycubes our algorithm will count separately all the polycubes of size n bounded by a box whose dimensions are W (its width, y-span), L (its length, x-span) and D (its depth, z-span). In each iteration specific values of W, L and D will be considered, for all possible values. There are two main differences between the two-dimensional and three-dimensional versions of the algorithm. The first difference relates to the boundary and its encoding, and the second difference lies in the updating rules employed in each iteration of the transfer-matrix algorithm.

4.1 Boundary Encoding

Recall that the boundary used in the two-dimensional transfer-matrix algorithm was a vertical column made of squares (with a "kink" created in the course of the algorithm). It consisted of a series of sites, from the top to the bottom of the bounding rectangle, and could be regarded as a line with or without a kink. In the three-dimensional version of the algorithm, the boundary is a rectangle embedded in three dimensions (with a more complicated "kink"). That is, the boundary is a face of a box (a slice). The boundary is thus a two-dimensional set of sites, with or without a kink. Figure 4.1 shows the boundary of a box without a kink, whereas Figures 4.2 and 4.3 show boundaries with kinks (the sites of the boundaries are shown in green and yellow/lighter gray).

To represent a boundary in the two-dimensional version of the transfer-matrix algorithm, we only need to encode a linear list of sites. This could be implemented by a

simple array. In order to represent a boundary in the three-dimensional version of the transfer-matrix algorithm, we will need to encode a two-dimensional layer of sites. This is easy to achieve by using a two-dimensional array.

In the two-dimensional version of the algorithm, the boundary sites and their interconnections were encoded very efficiently by only five symbols, namely, $\{0, 1, 2, 3, 4\}$. The encoding was very efficient because it uses the trait that different connected components (of the partially-built polyomino) cannot be interleaved along the boundary. For example, if sites at positions 1 and 5 along the boundary are connected (that is, belong to the same component), sites at positions 3 and 7 cannot be connected and belong to another component. The linearity of the boundary thus allows an efficient legal-parentheses-sequence-like encoding. However, in three dimensions, the occupied sites of the two-dimensional boundary can be connected in much more complicated manner. This prohibits the efficient encoding available in one lower dimension. Therefore, we extended Jensen's algorithm by a less-efficient boundary representation, which holds the connectivity information for the occupied boundary sites by using explicit component IDs. As in Section 3.2, we attribute the ID number of its conected component to each occupied site. This had a profound theoretical impact on the size of each boundary signature: instead of 3 bits that sufficed in two dimensions to encode any one of the five symbols, we needed many more bits to encode specific component IDs. However, in practice we did not need more bits. Polycubes of orders of at most 24 may have at most 16 connected components, since a new occupied site can connect no more than three previously disconnected components (see also below). Since we did not even approach this order of polycubes, we needed only 4 bits (per site) to encode the boundary. An addition of a fifth bit would suffice for many more orders of polycubes.

In addition to the encoding of the connected components of the occupied sites in the boundary, the signature needs to encode which borders of the bounding box the partially-built polycubes touch. Recall that in two dimensions the signature included two bits that indicated whether the top and bottom of the bounding rectangle were touched. In three dimensions we need four bits to indicate whether or not the four sides of the box (orthogonal to the boundary) are touched.

Figures 4.1, 4.2, and 4.2 show polycubes built inside three-dimensional bounding boxes. The left sides of the figures show the polycubes, while the rights sides plot the corresponding boundaries and the IDs allocated to the occupied sites.

4.2 Updating Rules

Similarly to the two-dimensional transfer-matrix algorithm, we define expansion rules that update the boundary encoding according to whether or not the newly-considered site is empty or occupied, and according to whether or not its neighbors are occupied, and if so, according to the identities of their connected components. The updating rules in the

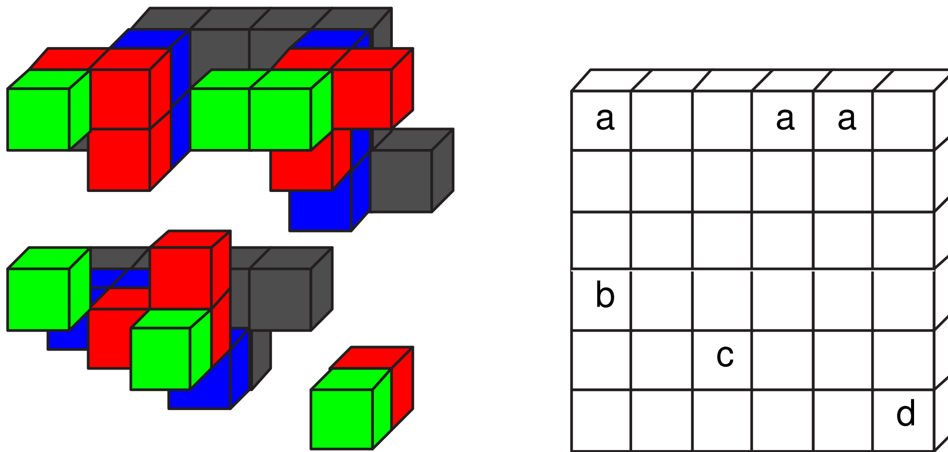


Figure 4.1: A polycube and its corresponding boundary

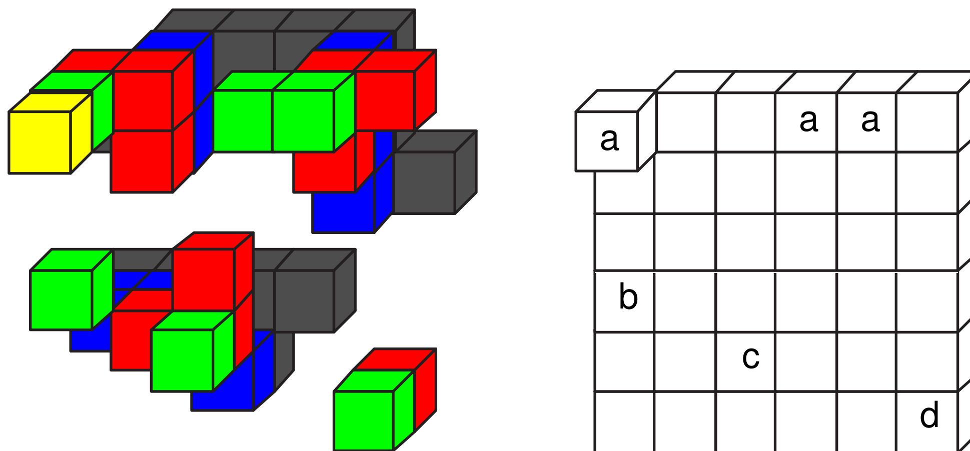


Figure 4.2: A polycube after adding one site (note the kink in the boundary)

three-dimensional version of the algorithm have the same semantics as those of the two-dimensional version, except that the former version has a few more cases to handle. We also need to consider the different encoding scheme in three dimensions.

Adding an empty site to the boundary may result in losing the connection between a component to the boundary. In such a case the new configuration becomes illegal so that it is discarded from the signatures database (similarly to the operation taken in two dimensions). Otherwise, the boundary is updated and stored again in the database.

Adding an occupied site may result in one of the following effects on the connected components touching the boundary: (a) Creating a new component; (b) Adding a new site to an existing component; and (c) Adding a new site that unites two or three components. (In the two-dimensional version of the algorithm, a new site can unite only two components. In three dimensions a new occupied site cannot unite more than three

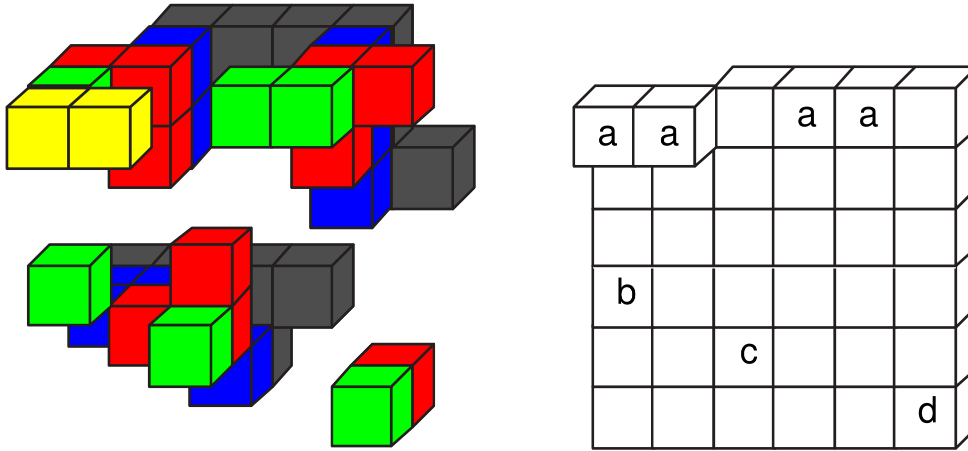


Figure 4.3: A polycube after adding two sites (note the kink in the boundary)

components due to the order of processing the boundary; three of the six faces of the new site cannot yet touch any other occupied site.) In all cases, the addition of a new site reflects the new connections (if they are made) between previously-existing components. If such unifications occur, components are renamed accordingly. In addition, the algorithm also updates the bits that indicate whether or not the four faces of the bounding box (orthogonal to the boundary face) are touched.

4.3 Optimizations

We have implemented severl optimizations whose aim was to reduce the number of signatures during the course of the algorithm (although we did not expect them to have any impact on the asymptotic running). The three optimizations that we implemented are similar in nature to those of [Je01]:

1. Remove signatures that represent polycubes whose size is larger than that of the target polycube order. (That is, discard polycubes whose number of connected components plus the minimum number of additional occupied sites needed to unite them into one component is larger than the target order.)
2. Consider the current depth for computing the size of polycubes. (That is, discard polycubes whose size plus the minimum number of additional sites needed to make the polycube span the entire depth of the bounding box is larger than the target order.)
3. Similarly, consider the distance of the partially-built polycube from the four sides of the bounding box orthogonal to the boundary face. (That is, discard polycubes

whose size plus the minimum number of additional sites needed to make the polycube touch all the four sides is larger than the target order.)

Obviously, combinations of these rules may also be used for discarding intermediate configurations. Table 4.1 shows the effect of the three optimizations described above on the running time of the algorithm. In this tabulation we used the target size of 8. We interpret the "negative" improvement in the depth-only experiment as evidence that the overhead of running this text was higher than the amount of time saved by this improvement.

Used optimizations			Time (sec.)	Improvement
Oversize	Depth	Touching sides		
No	No	No	115.890	
Yes	No	No	77.015	33.5%
No	Yes	No	117.468	-1.7%
No	No	Yes	75.234	35.1%
Yes	Yes	Yes	74.750	35.5%

Table 4.1: Effects of the optimizations on the running time

4.4 Results

We implemented the algorithm in C++ and ran the program on an IBM ThinkPad with one 1900MHz Pentium4 processor and 384 Megabytes of memory. After 701 seconds the program reported the value of $A_3(11)$. The values that we found of $A_3(n)$ (for $1 \leq n \leq 11$) agreed with the values published in the past. We ran out of memory before the computation of $A_3(12)$ was completed. Knuth, in [Kn01] states he needed about 850 Megabytes of RAM and about 10 Gigabytes of disk for computing $A_2(47)$. We did not use more than 384 Megabytes of memory (our RAM).

Chapter 5

Extending Redelmeier's Algorithm to Three Dimensions

5.1 Counting Polycubes

Recall that Redelmeier's algorithm is a procedure for counting some class of connected subgraphs in a graph, where the underlying graph is induced by the square lattice. In order to extend the algorithm to three dimensions, we need only modify the underlying graph, so that it will represent a three-dimensional cubic lattice. Then we must decide upon a canonical form of a polycube. We fix the origin at the leftmost cube in the "closest" row in the bottom layer. This way polycubes are built only at

$$\{(x, y, z) \mid (z > 0) \text{ or } ((z = 0) \text{ and } (y > 0)) \text{ or } ((z = 0) \text{ and } (y = 0) \text{ and } x \geq 0)\}.$$

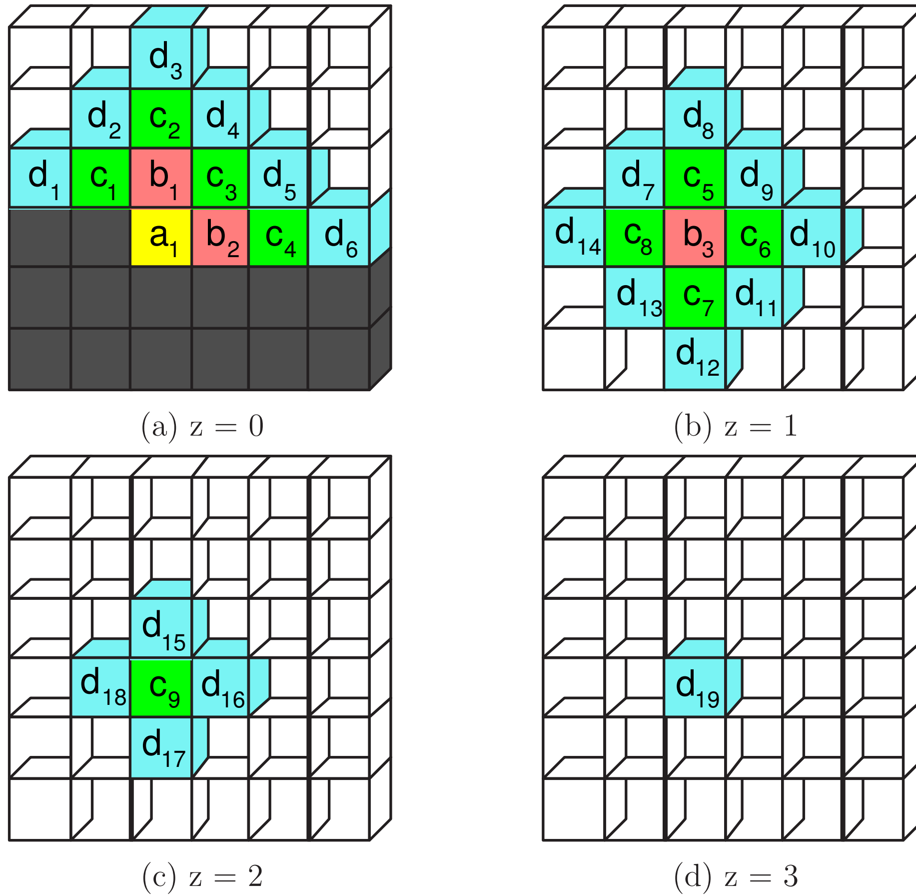
The origin cube is shown in yellow in Figure 5.1(a). The colored areas in the Figure 5.1 are the possible locations of cubes of polycubes of order 4. The corresponding graph in which the polycubes are counted is drawn in Figure 5.2.

Our implementation of the algorithm is generic and can be easily modified to every dimension. In addition, we can also use it for different kinds of grids, for example a triangular lattice and a hexagonal lattice.

5.2 Implementation Issues

5.2.1 Graph Representation

Our internal representation of the graph is a set of 'extended vertices', that is, a set of vertex structures, where each structure contains the unique ID number of a vertex (a natural number) and a small set of ID numbers of its neighbors in the graph. Since the



(a) z = 0 (b) z = 1

(c) z = 2 (d) z = 3

Figure 5.1: Valid cubes in a three dimensional lattice

graph is embedded in the cube lattice, the cardinality of every vertex in the graph is at most six (see Figures 5.2 and 5.1). The ID numbers are computed in a preprocessing step, in which every cube in the lattice, which corresponds to a vertex in the graph, is mapped to a natural number. We chose natural numbers as vertex IDs in order to maintain a simple array of vertex structures that speeds up the search algorithm.

As in [Rede81], we maintained all the versions of the set of untried cells in a single linked list, where the distinction between the different versions was achieved by using "headers" of the list which actually pointed to intermediate nodes in the list. In practice, the linked list was implemented as a simple array: the value stored in the ith position of the array was the ID of the cell pointed at by the ith cell.

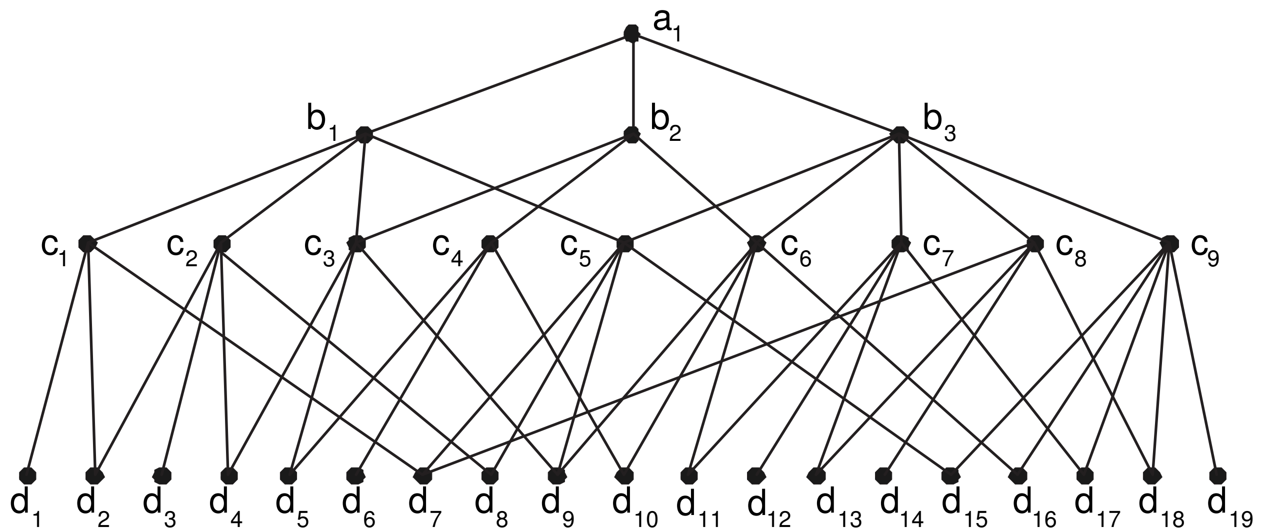


Figure 5.2: The underlying graph in three dimensions

5.2.2 Recursion

The polyomino-counting procedure calls itself recursively for each counted polyomino of every order. Therefore, the number of recursive calls in the run of our program reported here was $\Sigma_{n=1}^{18} A_3(n) \approx 9.6 \cdot 10^{13}$. In order to save this huge amount of function-call overhead, we implemented the recursion ourselves. Our implementation of the algorithm requires only three variables (ID numbers) to be kept in a context switch. Thus, we maintained a recursion stack in a short array, where every member was a triple of numbers. The maximum depth of the stack was obviously 18. Finally, a recursive call was simulated by a `push` operation (copying three IDs into the next available stack member, and advancing the stack pointer) and a jump to the beginning of the function, while recursive backtracking was simulated by a `pop` operation (fetching three IDs from the stack and updating the stack pointer) and a jump to immediately after the jump command described above. Our experimentation (with computing polycubes up to order 18) shows that this mechanism saved about 5% of the running time of the program.

5.2.3 Large Numbers

The number of polycubes of high orders exceeds `UINT_MAX` (the maximum possible value of a C `unsigned int` variable on a 32-bit word machine), which is $2^{32} - 1 = 4,294,967,295$. Therefore, we implemented large numbers "manually," i.e., we maintained each such number as a quadruple of 32-bit numbers. The only operations we had to do with these numbers were initialization (to zero) and increment by one. In the latter operation we simply checked the "lowest" 32-bit number for overflow, in which case we reset it to zero

and incremented the "higher" 32-bit number by one. We performed a similar operation on the second and third 32-bit numbers. In fact, we set the overflow to occur when a 32-bit word exceeded one billion (10^9) instead of $2^{32} - 1 \approx 4.3 \cdot 10^9$ so as to facilitate the printing of the compound number in a decimal form (each 32-bit part could then be printed separately).

5.2.4 Warm Restart

Since the program ran for 51.37 days, crashes or hang-ups of the running machine were inevitable or at least should have been considered. (We experienced at least 2 crashes.) We implemented a simple mechanism in which the entire memory of the program (about 733 Kilobytes) was saved in a file at every billionth (10^9th) recursive call to the function that counts a new polycube and generates all the new candidates. This occurred on the average every six or seven minutes. The dump-file mechanism enabled a warm restart of the program: it was possible to rerun the program starting with loading the contents of a specified file into the program's memory.

5.3 Results

We implemented the algorithm in C (the source code is given in the Appendix) and ran the program on an IBM workstation with one 1700MHz Pentium4 processor and 1024 Megabytes of memory. After 51.37 days the program reported the values given in Table 1.2. The values of $A_3(n)$ for all $1 \leq n \leq 17$ agree with previous publications. To the best of our knowledge, this is the first tabulation of $A_3(18)$ in the literature.

Chapter 6

Conclusion

In this thesis we presented our research on several aspects of counting polyominoes and polycubes. Our work included showing an upper bound on the running time of Jensen's algorithm (for computing polyominoes), and introducing a few extensions for Jensen's algorithm. In addition, we extended both algorithms to enable counting of three-dimensional polycubes, verified the already-known first 17 values of the series A_3, and found one new value, namely, $A_3(18)$.

We established a relation between a certain class of strings and Motzkin's numbers. This allowed us to analyze accurately Jensen's algorithm for counting fixed polyominoes, setting the bound $O(n^{5/2}(\sqrt{3})^n)$ on the algorithm's running time. Except for the polynomial factor, the asymptotic exponential bound is tight.

We provided two extensions for Jensen's algorithm for counting polyominoes. Both aim to reduce the number of iterations of the algorithm by using previously-computed information. The main drawback of both extensions is the fact that they do not solve the main bottleneck of the transfer-matrix algorithm, which is its memory consumption. In the future, when more memory is available, so that larger bounding boxes may be considered, these extensions might prove useful for computing polyominoes in some of the bounding boxes, namely, bounding boxes that are short and long ($W \ll L$).

We also extended both Jensen's and Redelmeier's algorithms to enable counting of three-dimensional polycubes. For Jensen's algorithm we implemented a scheme for encoding boundaries of three-dimensional polycubes, and derived suitable updating rules. We also presented an efficient implementation of Redelmeier's algorithm (which actually fits any dimension), and used it to find the number of fixed polycubes of order 18. To the best of our knowledge, this number has never been published in the literature.

In the future we intend to further enhance our extensions for Jensen's algorithm by encoding all the boundaries of a bounding box, to be used as a tile joining rectangles in all directions. We also plan to investigate more efficient encoding schemes for three-dimensional boundaries, and also to try to compute the numbers of fixed polyominoes

of much higher dimensions ($d > 3$). In addition, we would like to implement parallel versions of both Jensen's and Redelmeier's algorithms, possibly combining them in an efficient way.

Acknowledgment

We are grateful to Günter Rote and Stefan Felsner for suggesting the relation between signature strings and Motzkin numbers, and to Noga Alon for providing important insights on these numbers.

Bibliography

[BH57] S.R. Broadbent and J.M. Hammersley, Percolation processes: I. Crystals and mazes, *Proc. Cambridge Philos. Soc.*, 53 (1957), 629–641.

[Co95] A.R. Conway, Enumerating 2D percolation series by the finite-lattice method: Theory *J. Physics, A: Mathematical and General*, 28 (1995), 335–349.

[CG95] A.R. Conway and A.J. Guttmann, On two-dimensional percolation, *J. Physics, A: Mathematical and General*, 28 (1995) 891–904.

[De88] M.P. Delest, Generating functions for column-convex polyominoes, *J. of Combinatorial Theory, Ser. A*, 48 (1988), 12–31.

[DV84] M.-P. Delest and G. Viennot, Algebraic languages and polyominoes enumeration, *Theoretical Computer Science*, 34 (1984), 169–206.

[Ed61] M. Eden, A two-dimensional growth process, *Proc. 4th Berkeley Symp. on Mathematics, Statistics, and Probability*, Vol. IV, Univ. of California Press, Berkeley, CA, 1961.

[Go65] S.W. Golomb, Polyominoes, 2nd ed., Princeton Univ. Press, 1994.

[Gu82] A.J. Guttmann, On the number of lattice animals embeddable in the square lattice, *J. Phys. A*, 15 (1982), 1987–1990.

[IntSeq] The On-Line Encyclopedia of Integer Sequences, `http://www.research.att.com/~njas/sequences/index.html` .

[GJ+00] A.J. Guttmann, I. Jensen, L.H. Wong, and I.G. Enting, Punctured polygons and polyominoes on the square lattice, *J. Phys. A: Math. Gen.*, 33 (2000), 1735–1764.

[Je01] I. Jensen, Enumerations of lattice animals and trees, *J. of Statistical Physics*, 102 (2001), 865–881.

[Je01a] D.E. Knuth, WWW homepage: `http://sunburn.stanford.edu/~knuth/programs/jensen.txt` . (personal communication with I. Jensen).

[Ki88] D. Kim, The number of convex polyominoes with given perimeter, *Discrete Mathematics*, 1988, 47–51.

[Kl67] D.A. Klarner, Cell growth problems, *Canadian J. of Mathematics*, 19 (1967), 851–863.

[Kn01] D.E. Knuth, WWW homepage: `http://sunburn.stanford.edu/~knuth/programs.html#polyominoes` .

[KR65] D.A. Klarner and R.L. Rivest, Some results concerning polyominoes, *Fibonacci Quarterly*, 3 (1965), 9–20.

[KR73] D.A. Klarner and R.L. Rivest, A procedure for improving the upper bound for the number of n-ominoes, *Canadian J. of Mathematics*, 25 (1973), 585–602.

[KR74] D.A. Klarner and R.L. Rivest, Asymptotic bounds for the number of convex n-ominoes, *Discrete Mathematics*, 8 (1974), 31–40.

[Lu71] W.F. Lunnon, Counting polyominoes, in: *Computers in Number Theory* (A.O.L. Atkin and B.J. Birch, eds.), Academic Press, London, 1971, 347–372.

[Lu75] W.F. Lunnon, Counting multidimensional polyominoes, *The Computer Journal*, 18 (1975), 366–367.

[Ma74] J.L. Martin, Computer techniques for evaluating lattice constants, *Phase Transitions and Critical Phenomena*, vol. 3 (C. Domb and M.S. Green, eds.), Academic Press, London, 97–112, 1974.

[Me90] S. Mertens, Lattice animals: A fast enumeration algorithm and new perimeter polynomials, J. of Statistical Physics, 58 (1990), 1095–1108.

[ML92] S. Mertens and M.E. Lautenbacher, Counting lattice animals: A parallel attack, J. of Statistical Physics, 66 (1992), 669–678.

[Mo48] T. Motzkin, Relations between hypersurface cross ratios, and a combinatorial formula for partitions of a polygon, for permanent preponderance, and for non-associative products, *Bull. American Mathematical Society*, 54 (1948), 352–360.

[PLP67] T.R. Parkin, L.J. Lander, and D.R. Parkin, Polyomino enumeration results, *SIAM Fall Meeting*, Santa Barbara, CA, 1967.

[Rea62] R.C. Read, Contributions to the cell growth problem, *Canadian J. of Mathematics*, 14 (1962), 1–20.

[Rede81] D.H. Redelmeier, Counting Polyominoes: Yet another attack, *Discrete Mathematics*, 36 (1981), 191–203.

[Redn82] S. Redner, A Fortran program for cluster enumeration, *J. of Statistical Physics*, 29 (1982), 309-315.

[RW81] B.M.I. Rands and D.J.A. Welsh, Animals, trees, and renewal sequences, *IMA J. of Applied Mathematics*, 27 (1981), 1–17; corrigendum 28 (1982), 107.

[SGG76] M.F. Sykes, D.S. Gaunt, and M. Glen, Percolation processes in three dimensions, *J. Phys. A: Math. Gen.*, Vol. 9, No. 10, (1976), 1705–1712.

[SG76] M.F. Sykes amd M. Glen. Percolation processes in two dimensions I. Low-density series expansions, *J. Phys. A: Math. Gen.*, 9 (1976), 87–95.

Appendix A: C Source Code of the Three-Dimensional Version of Redelmeier's Algorithm

This appendix contains the listing of our C program for three-dimensional polyominoes by using Redelmeier's algorithm.

```
#include <time.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
void bzero (void *mem, int n);

#define MAX_SIZE  18

// East / West
#define X_SIZE    (2 * MAX_SIZE - 2)
// North / South
#define Y_SIZE    (2 * MAX_SIZE - 2)

#define FULL_SIZE (X_SIZE * Y_SIZE * MAX_SIZE)
// Index of 0, in lattice
#define _0_INDEX  (MAX_SIZE - 2)

// Idea: count up to 1G (1,000,000,000) and then increase the next word
// start with [0][size], then [1][size] and so on ...
#define NUM_COUNTERS      4
int big_int[NUM_COUNTERS][1 + MAX_SIZE];

#define SET_BOARD(_pos, _value)    board [_pos] = (_value)
#define CHECK_BOARD(_pos, _value)  board [_pos] == (_value)
#define GET_SUCC(_pos1, _pos2)     _pos2 = untried [_pos1]
#define SET_SUCC(_pos1, _pos2)     untried [_pos1] = _pos2

/* Note that the location of neighbors is important:
   the valid neighbors must be in the first positions.
   Look at the while(hdr) (step 4) */
```

```
/* 3D */
typedef enum {
      north, east, up, west, south, down
} dir_t;

typedef struct {
      int n_neis;
      int neis [6]; // (Y)north, south, (X)east, west, (Z)up, down
} neighbors_t;

typedef enum {
      FREE, BORDER, OCCUPIED, REACHABLE
} cell_t;


cell_t    board    [FULL_SIZE];
int       untried  [FULL_SIZE];
neighbors_t all_neis [FULL_SIZE];

int size = 0;
int n_calls = 0;
int n_saves = 0;

void fixed (
      int hdr
);

#define MAX_STACK 1000
int stack [3*MAX_STACK];
int p_stack = 0;
#define PUSH(_a, _b, _c) { \
      stack [p_stack] = _a; \
      stack [p_stack + 1] = _b; \
      stack [p_stack + 2] = _c; \
      p_stack += 3; \
}
#define POP(_a, _b, _c) { \
      _c = stack [p_stack - 1]; \
      _b = stack [p_stack - 2]; \
      _a = stack [p_stack - 3]; \
      p_stack -= 3; \
}

clock_t clk0;
double ptime;
```

```
int main (
      int   argc,
      char* argv []
) {
      int i, j, p, hdr;
      int x, y, z;
      char fn [64];
      FILE* fp;

      /* [Warm restart] */
      // -f<file>
      if (argc >= 2) {
            if (strncmp (argv [1], "-f", 2) == 0) {
                  FILE* fp = fopen (argv [1] + 2, "rb");
                  if (fp == NULL) {
                        printf ("fixed: can't open file %s\n", argv [1] + 2);
                        exit (0);
                  }
                  fread (& i, sizeof (int), 1, fp);
                  if (i != MAX_SIZE) {
                        printf ("ERROR MAX_SIZE:
                              expected=%d found=%d\n",MAX_SIZE, i);
                        exit(0);
                  }
                  fread (& size, sizeof (int), 1, fp);
                  fread (& n_saves, sizeof (int), 1, fp);
                  fread (big_int, sizeof (int), (NUM_COUNTERS)*(1 + MAX_SIZE), fp);
                  fread (board, sizeof (cell_t), FULL_SIZE, fp);
                  fread (untried, sizeof (int), FULL_SIZE, fp);
                  fread (all_neis, sizeof (neighbors_t), FULL_SIZE, fp);
                  fread (& hdr, sizeof (int), 1, fp);
                  fread (& p_stack, sizeof (int), 1, fp);
                  fread (stack, sizeof (int), p_stack, fp);
                  fread (& ptime, sizeof (double), 1, fp);
                  printf ("red: MS=%d sz=%d ns=%d hdr=%d pstk=%d ptime=%f\n",
                          i,size,n_saves,hdr,p_stack,ptime);
                  fclose (fp);
                  printf ("fixed: loaded file %s, MAX_SIZE = %d, n_saves = %d\n",
                          argv [1] + 2, MAX_SIZE, n_saves);
                  clk0 = clock ();
                  goto WARM_RESTART;
            } else {
                  printf ("fixed: unknown option %s\n", argv [1]);
                  exit (0);
            }
      }
      /* [Init] */
      ptime = 0.0;
      clk0 = clock ();
      bzero (big_int, sizeof (big_int));
      bzero (untried, sizeof (untried));
```

```
bzero (all_neis, sizeof (all_neis));

/* Preparing neighbors lists for 3D lattice */
for (z = 0; z < MAX_SIZE; z ++) {// Z
      for (y = 0; y < Y_SIZE; y ++) { // Y
            for (x = 0; x < X_SIZE ; x ++) { // X
                  if (  (z == 0 && y <  _0_INDEX ) ||
                        (z == 0 && y == _0_INDEX && x < _0_INDEX) )
                        continue;

                  p = z * Y_SIZE * X_SIZE + y * X_SIZE + x ;

                  all_neis [p].n_neis = 3;
                  all_neis [p].neis [east] = p + 1;         // East
                  all_neis [p].neis [north] = p + X_SIZE;    // North
                  all_neis [p].neis [up] = p + Y_SIZE * X_SIZE;    // Up

                  // West
                  if (   ( z == 0 && y == _0_INDEX && x > _0_INDEX ) ||
                         ( z == 0 && y > _0_INDEX ) ||
                         ( z > 0 )      )
                  {
                        all_neis [p].n_neis ++;
                        all_neis [p].neis [west] = p - 1;
                  }

                  // South
                  if (   ( z == 0 && y == _0_INDEX + 1 && x >= _0_INDEX ) ||
                         ( z == 0 && y >  _0_INDEX + 1 ) ||
                         ( z > 0)  )
                  {
                        all_neis [p].n_neis ++;
                        all_neis [p].neis [south] = p - X_SIZE;
                  }

                  // Down
                  if (   ( z > 1 ) || ( z == 1 && y >  _0_INDEX ) ||
                         ( z == 1 && y == _0_INDEX && x >= _0_INDEX ) )
                  {
                        all_neis [p].n_neis ++;
                        all_neis [p].neis [down] = p - Y_SIZE * X_SIZE;
                  }
            } // X
      } // Y
} // Z


/* The algorithm is started with the parent being the empty polyomino, */
bzero (board, sizeof (board));
/* and the untried set containing only the origin. */
hdr = _0_INDEX + _0_INDEX * X_SIZE ;
```

```
    WARM_RESTART:
    fixed (hdr);

    /* [Print Number of Polycubes] */
    sprintf (fn, "n-fixed-%d-final-3D.txt", MAX_SIZE);
    fp = fopen (fn, "wb");
    if (fp) {
        for (p = 1; p <= MAX_SIZE; p ++)
        {
            fprintf (fp, "Fixed (%d)\t= %d,%d,%d,%d\n",
                     p,
                     big_int[3][p],
                     big_int[2][p],
                     big_int[1][p],
                     big_int[0][p]); // NUM_COUNTERS fix num of %d
        }
        ptime += (double) (clock () - clk0) / CLOCKS_PER_SEC;
        fprintf (fp,
                 "\nTime: %5.2f seconds, %5.2f minutes, \
                  %5.2f hours, %5.2f days\n",
                 ptime, ptime / 60.0, ptime / 3600.0, ptime / 86400.0);
        fclose (fp);
    }
}

void fixed (
    int hdr
) {

    /* [Local variables] */
    register int cur, save_hdr, n, i;
    int* p;
    int m_c;

    RESTART:

    n_calls ++;
    if (n_calls == 1000000000) {
        char fn [64];
        FILE* fp;

        n_saves ++;
        sprintf (fn, "n-fixed-%d-%d-3D.dat", MAX_SIZE, n_saves);
        fp = fopen (fn, "wb");
        if (fp) {
            int ms;
            ptime += (double) (clock () - clk0) / CLOCKS_PER_SEC;
            ms = MAX_SIZE;
            fwrite (& ms, sizeof (int), 1, fp);
            fwrite (& size, sizeof (int), 1, fp);
```

```
            fwrite (& n_saves, sizeof (int), 1, fp);
            fwrite (big_int, sizeof (int), (NUM_COUNTERS)*(1 + MAX_SIZE), fp);
            fwrite (board, sizeof (cell_t), FULL_SIZE, fp);
            fwrite (untried, sizeof (int), FULL_SIZE, fp);
            fwrite (all_neis, sizeof (neighbors_t), FULL_SIZE, fp);
            fwrite (& hdr, sizeof (int), 1, fp);
            fwrite (& p_stack, sizeof (int), 1, fp);
            fwrite (stack, sizeof (int), p_stack, fp);
            fwrite (& ptime, sizeof (double), 1, fp);
            printf ("wrt: MS=%d sz=%d ns=%d hdr=%d pstk=%d ptime=%f\n",
                    i,size,n_saves,hdr,p_stack,ptime);
            fclose (fp);
            printf ("fixed: saved file %s, MAX_SIZE = %d, n_saves = %d\n",
                    fn, MAX_SIZE, n_saves);
            n_calls = 0;
            clk0 = clock ();
      }
}

/* The following steps are repeated until the untried set is exhausted.*/
while (hdr) {

      /* 1. Remove an arbitrary element from the untried set. */
      cur = hdr;
      GET_SUCC (cur, hdr);

      /* 2. Place a cell at this point. */
      SET_BOARD (cur, OCCUPIED);

      /* 3. Count this new polyomino. */
      for (m_c = 0; m_c < NUM_COUNTERS; m_c ++) {

            big_int[m_c][ size ] ++;

            if ( big_int[m_c][ size ] == 1000000000 ) {
                  big_int[m_c][ size ] = 0;
                  // and go to next loop with m_c ++
            }
            else
                  break; // don't go...
      }

      /* 4. If the size is less than P: */
      if (size < MAX_SIZE) {

            /* (a) Add new neighbors to the untried set. */
            save_hdr = hdr;
            /* Note the following assumption:
               All valid neighbors should be in the first places, i.e.,
               2 valid neighbors should be in [0],[1]
               3 valid neighbors should be in [0],[1],[2] and so on.
```

```
                Note that we do not check for each of the 6 neighbors but
                the exact number of neighbors by using p ( p++, .... )
              */
              for (i = 0, p = all_neis [cur].neis;
                                   i < all_neis [cur].n_neis; i ++) {
                    n = * p ++;
                    if ( n!=0 && CHECK_BOARD (n, FREE)) {
                          SET_SUCC (n, hdr);
                          hdr = n;
                          SET_BOARD (n, REACHABLE);
                    }
              }

              /* (b) Call this algorithm recursively with the  */
              /*     new parent being the current polyomino,  */
              /*     and the new untried set being a copy */
              /*     of the current one. */
              PUSH (hdr, save_hdr, cur);
              goto RESTART;
              RESUME:

              /* (c) Remove the new neighbors from the untried set. */
              while (hdr != save_hdr) {
                    SET_BOARD (hdr, FREE);
                    GET_SUCC (hdr, hdr);
              }
        }

        /* 5. Remove newest cell - set site to forbidden */
        /* keeping it occupied (not setting it to DONE) is sufficient */
        /* SET_BOARD (cur, DONE); */
        size --;
  }

  if (p_stack) {
        POP (hdr, save_hdr, cur);
        goto RESUME;
  }
}
```

ספירת polyominoes בשניים ושלושה מימדים

מיכה מופי

ספירת polyominoes בשניים ושלושה מימדים

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר
מגיסטר למדעים במדעי המחשב

מיכה מופי

הוגש לסנט הטכניון – מכון טכנולוגי לישראל
כסליו תשס"ג   חיפה   דצמבר 2003

המחקר נעשה בהנחיית ד"ר גיל ברקת בפקולטה למדעי המחשב.

אני מודה לטכניון על התמיכה הכספית הנדיבה בהשתלמותי

אני רוצה להודות לד"ר גיל ברקת ולהביע את הערכתי הרבה על ההנחיה, העזרה, והסבלנות הרבה במהלך המחקר.
ברצוני אף להודות למר דני ברונשטיין על העזרה הרבה אשר הגיש לי.

תוכן הענינים

הקדמה.......... 5
1. מבוא 6
1.1 עבודות קודמות 8
1.2 האלגוריתם של Redelmeier ספירה של תתי גרפים קשירים 10
1.3 האלגוריתם של Jensen ואלגוריתם ה-transfer matrix 11
1.4 ארגון התיזה 14
2. סיבוכיות אלגוריתם ה-transfer matrix 15
2.1. מספר החתימות 15
2.1.1 מבוא 15
2.1.2 המשפט 16
2.2 סיבוכיות זמן 20
3. ספירת polyominoes דו מימדיים 23
3.1 הכפלת המלבן החוסם 23
3.1.1 איחוד חתימות 24
3.1.2 אופטימיזציה 26
3.2 הכפלת המלבן חוסם באופן רקורסיבי 28
3.3 יתרונות והסרונות 31
4. הרחבת האלגוריתם של Jensen לשלושה מימדים 33
4.1 גבול transfer matrix תלת מימדי 33
4.2 חוקי עדכון 34
4.3 אופטימיזציה 36
4.4 תוצאות 37
5. הרחבת האלגוריתם של Redelmeier לשלושה מימדים 38
5.1 ספירת polyocubes תלת מימדיים 38
5.2 נושאים הקשורים למימוש 38
5.2.1 יצוג בגרף 38
5.2.2 רקורסיה 40
5.2.3 מספרים גדולים 40
5.2.4 אתחול חם 41
5.3 תוצאות 41
6. מסקנות 42
תודות 44
רשימת מקורות 45
נספח א: קוד מקור ב C של האלגוריתם של Redelmeier בשלושה מימדים 47

רשימת איורים

1.1 polyominoes קבועים דו מימדים 6
1.2 polyominoes קבועים תלת מימדים 7
1.3 גרף המייצג אזור בגריד בו ניתן לבנות polyominoes 11
1.4 האלגוריתם של Redelmeier 11
1.5 polyomino במהלך בניה, גבול, וחתימה 13
2.1 התא הימני ביותר בחתימה המחובר לתא w בחתימה 17
2.2 כל החתימות האפשריות עבור $1 \le w \le 4$ 18
2.3 (המשך של 2.2) כל החתימות האפשריות עבור $w = 5$ 19
2.4 מיפוי חתימה למסלול Motzkin 20
2.5 מימוש של מחרוזת 20
3.1 איחוד שתי קונפיגורציות 25
3.2 איחוד שתי קונפיגורציות עם עמודה חופפת 27
3.3 איחוד שתי קונפיגורציות מורחבות 29
3.4 איחוד שתי קונפיגורציות מורחבות לאחת שאינה חוקית 30
4.1 polyomino תלת מימדי והגבול המתאים 35
4.2 polyomino תלת מימדי והגבול המתאים לאחר צעד בודד 35
4.3 polyomino תלת מימדי והגבול המתאים לאחר שני צעדים 36
5.1 אזור חוקי בגריד תלת מימדי 39
5.2 הגרף המתאים 40

רשימת טבלאות

1.1 מספר ה-polyominoes הקבועים הדו מימדים 9
1.2 מספר ה-polyominoes הקבועים התלת מימדים 10
3.1 השוואת מספר החתימות למספר החתימות המורחבות 32
4.1 השפעת אופטימיזציות על זמן הריצה 37

תקציר

פוליומינו (polyomino) בגודל n הוא קבוצה בת n ריבועים (תאים) על גריד ריבועי רגיל, המחוברים ביניהם בצלעות. פוליומינו "קבועים" (fixed polyominoes) (להלן - polyominoes) נחשבים שונים אם צורתם שונה או אם הם בעלי צורה זהה אך הם מסובבים אחד ביחס לאחר. הסימון המקובל בספרות למספר ה-polyominoes בגודל n הוא $A(n)$. אנו נסמן ב-$A_2(n)$ את מספר ה-polyominoes הדו-מימדים בגודל n. ניתן לראות באיור 1.1 polyominoes קבועים שונים בגדלים 2, 3, ו-4. ניתן לראות כי $A_2(3) = 6$, $A_2(2) = 2$, וכן $A_2(4) = 19$. קיימים גם polyominoes תלת מימדים (הנקראים polycubes), המוגדרים באופן דומה: polycube בגודל n הוא קבוצה בת n קוביות על גריד תלת מימדי רגיל, המחוברת ביניהן בפאות. את מספר ה-polycubes בגודל n נסמן ב-$A_3(n)$. ניתן לראות באיור 1.2 שלושה ו-15 polycubes קבועים תלת מימדים שונים בגדלים 2 ו-3, בהתאמה. חוקרים בתחום הפיזיקה הסטטיסטית נעזרים במספר ה-polyominoes וה-polycubes כדי לחקור בעיות הקשורות בזרימה של נוזל דרך תווך מגורען. לדוגמא, זרימה של מים דרך פולי קפה.

ידוע כי $A_2(n)$ הוא אקספוננציאלי ב-n, עם מכפיל פולינומיאלי המוערך ע"י Cn^{-1}, כאשר C הוא קבוע כלשהו. במילים אחרות, הגבול $\lim_{n\to\infty}(A_2(n+1)/A_2(n))$ שואף לקבוע. נכון להיום, לא ידועות נוסחאות אנליטיות המביעות את $A_2(n)$ ואת $A_3(n)$ כפונקציה של n. האופן היחיד הידוע כיום לחישוב ערכים של $A_2(n)$ ו-$A_3(n)$ הוא על ידי הרצת תוכנית מחשב, המחשבת את הערכים ע"י יצור (ישיר או עקיף) של כל ה-polyominoes/polycubes וספירתם. ניתן לראות בטבלאות 1.1 ו-1.2 את מספרי ה-polyominoes וה-polycubes בגדלים הידועים כיום.

עבודה זו מתבססת על שתי עבודות קודמות בתחום זה. העבודה הראשונה פורסמה בשנת 1981 על ידי Redelmeier [Re81]. עבודה זו היתה הראשונה אשר הציגה אלגוריתם לספירת polyominoes ע"י בניית כל polyomino פעם אחת בלבד. (להבדיל מאלגוריתמים קודמים, אשר בנו כל polyomino מספר פעמים, והקדישו את רוב זמנם לאיתור חזרות.) כלומר, למעט הבנייה המפורשת של כל ה-polyominoes, לא מבצע אלגוריתם זה כל עבודה מיותרת. לפיכך, סיבוכיות זמן הריצה של האלגוריתם זהה למספר ה-polyominoes (תחת ההנחה שכל polyomino נוצר ונספר בזמן קבוע). האלגוריתם של Redelmeier מבוסס על ספירה של תתי גרפים קשירים בגרף, כאשר הגרף מייצג את האזור בגריד בו ניתן לבנות polyominoes. מאחר שיש לספור את כל ההזזות של polyomino על הגריד רק פעם אחת, יש להגדיר צורה קנונית של polyominoes. Redelmeier בחר לקבע את הריבוע השמאלי ביותר בשורה התחתונה ב-polyomino בראשית הצירים, כלומר ב-(0,0). יש, לכן, להגביל את החיפוש בגריד באזור הבא:

$$\{ (x,y) \mid (y>0) \text{ or } ((y=0) \text{ and } (x\geq 0)) \}.$$

ניתן לראות באיור 1.3 את האזור ה"חוקי" בגריד וכן את הגרף המתאים לספירת polyominoes בגודל 5. ספירת ה-polyominoes באזור החוקי (המכילים את (0,0)) זהה לספירת תתי גרפים קשירים המכילים את שורש הגרף.

עבודה נוספת העומדת בבסיס עבודה זו פורסמה בשנת 2001 על ידי Jensen [Je01]. בעבודה זו מתואר אלגוריתם לספירת polyominoes, המבוסס על סכימה הנקראת transfer matrix. האלגוריתם של Jensen סופר בנפרד את כל ה-polyominoes בגודל n, החסומים על ידי מלבן בגודל $W\times L$, כאשר W הוא גובה המלבן ו-L הוא רוחבו (בצירים y ו-x, בהתאמה). בכל איטרציה של האלגוריתם משתמשים בערכים שונים עבור W ו-L, עבור כל הערכים האפשריים. כדי למנוע מצב בו polyominoes זהים נספרים בשני מלבנים שונים, שומרים תמיד על התנאי הקובע כי כל polyomino נוגע בהכרח בכל צלעות המלבן החוסם.

עבור כל זוג ערכים של W ו-L, נספרים ה-polyominoes ע"י אלגוריתם ה-transfer matrix. האלגוריתם בונה את ה-polyominoes משמאל לימין, ובכל עמודה מלמעלה למטה. חשיבות מכרעת לעובדה כי האלגוריתם אינו סופר את כל ה-polyominoes, אלא עוקב אחר מספר ה-polyominoes בעלי גבול (תאים בגריד, תפוסים או ריקים) ימני זהה. לשם כך, מקודדים הגבולות הימניים של ה-polyominoes (תוך כדי בנייתם) באופן יעיל בעזרת חתימות (signatures). בניית ה-polyominoes נעשית על ידי הוספת תאים בעמודה הנוכחית (מימין לגבול הימני) מלמעלה למטה. כל תא חדש יכול להתווסף בשני אופנים: מלא (תפוס, חלק מה-polyominoes) או ריק (אינו חלק מהם). מאופן הוספת התאים נובע שבגבול הימני ישנה "ברך" בדיוק בתא המוסף; ראה איור 1.5. הוספת תא נעשית למעשה על ידי עידכון קבוצת החתימות ומספר ה-polyominoes הצמוד לכל חתימה. כל עידכון (של תא מלא או ריק) עשוי לא רק לשנות את מספר ה-polyominoes של חתימה מסוימת, אלא עשוי להוסיף חתימות חדשות ואף למחוק חתימות קיימות (כאשר נוצרים polyominoes לא חוקיים אשר אינם יכולים להפוך לחוקיים בעתיד). כיוון שקבוצת החתימות עשויה להיות גדולה מאוד (והיא אכן כך), מקודדות החתימות ביעילות על ידי חמישה סמלים. ניתן למצוא בפרק 1.3 את פירוט הסמלים ואת משמעותם. חשוב לציין כי הקידוד קובע באופן חד משמעי את החלקים הקשירים (ב-polyominoes הנבנים) הנוגעים בגבול הימני, ובאילו מצלעות המלבן החוסם הם נוגעים.

Jensen אינו מספק ב-[Je01] ניתוח של סיבוכיות זמן הריצה של האלגוריתם, אלא מסתפק בהערכה ויזואלית של גרף, המבטא את מספר החתימות כפונקציה של W עבור ערכי W עליהם הריץ את האלגוריתם. בפרק 2 אנו מציגים ניתוח סיבוכיות-זמן מדויק של האלגוריתם של Jensen. החסם המוצג הוא מדוייק (עד כדי הכופל הפולינומיאלי). לצורך ניתוח הסיבוכיות אנו מתמקדים בגודל קבוצת החתימות, כיוון שחישובה בכל צעד הוא חלק הארי של זמן הריצה. אנו מציגים מיפוי אחד-לאחד בין החתימות השונות לבין קבוצת מחרוזות מסויימת, המורכבות מאוצר תווים מסויים (והתו רווח), כך שכל בלוק תווים רצופים מורכב מאותו התו, וסדרת הבלוקים מהווה הכללה של סדרת סוגריים חוקיים. מסתבר כי קיים קשר בין מחרוזות אלו לבין מסלולי Motzkin. אנו נעזרים בכך שמספר מסלולי Motzkin ידוע על מנת לחסום את גודל קבוצת החתימות.

בפרק 3 אנו מראים שתי הרחבות לאלגוריתם של Jensen. שתי ההרחבות מנצלות אינפורמציה שחושבה באיטרציות קודמות. בהרחבה הראשונה אנו משתמשים בתוצאות החישוב של ה-polyominoes החסומים במלבן בגודל W×L על מנת לחשב בצעד יחיד (אך יקר) את מספר ה-polyominoes במלבן (רחב כפליים) בגודל W×2L. בנוסף, אנו מציגים דרך המאפשרת להוזיל משמעותית את עלות החישוב של הצעד היחיד, במחיר זניח של הקטנת המלבן החוסם ל-W×(2L-1).

ההרחבה השניה מאפשרת לחזור על ההרחבה הראשונה שוב ושוב. הרחבה זו מאפשרת לחשב באופן רקורסיבי את מספר ה-polyominoes (ואת חתימותיהם) החסומים במלבן בעל רוחב כפול 2L, בהינתן קבוצת החתימות ומספר ה-polyominoes הצמוד לכל חתימה עבור מלבן ברוחב L. כדי לאפשר זאת היה עלינו להרחיב את מושג החתימה. החתימה המורחבת כללה תיאור של החלקים הקשירים הנוגעים בשני הגבולות האנכיים (בעוד שהחתימה הרגילה מתארת את הגבול הימני בלבד). השימוש בחתימות המורחבות הביא לעליה ניכרת במספר החתימות במהלך הריצה, והגדיל באופן משמעותי את צריכת הזיכרון, דבר אשר הפך לצוואר הבקבוק של האלגוריתם.

אנו מציגים שתי הרחבות נוספות, האחת לאלגוריתם של Redelmeier והשנייה לאלגוריתם של Jensen, לספירת polycubes. הרחבת האלגוריתם של Redelmeier אינה כרוכה בשינויים עקרוניים. כיוון שאלגוריתם זה מבוסס על ספירה של תתי גרפים קשירים בגרף, כל שהיה עלינו לעשות הוא לשנות את הגרף כך שייצג את האזור בגריד קובייתי תלת מימדי, בו ניתן לבנות polycubes. ראשית, היה עלינו להגדיר צורה קנונית עבור polycubes. קיבענו את ראשית הצירים (0,0,0) בקוביה השמאלית ביותר בשורה הקרובה אלינו ביותר בשכבה התחתונה של ה-polycube. ניתן לראות באיורים 5.1 ו-5.2 את האזור החוקי בגריד ואת הגרף המתאים. למותר לציין, כי הגרף המתקבל רחב הרבה יותר ובעל דרגת קשירות גדולה יותר מן הגרף המתאים ל-polyominoes.

להתאמת האלגוריתם של Jensen נדרשנו לבצע שינויים רבים יותר. בראש ובראשונה היה עלינו לספור בנפרד את כל ה-polycubes בגודל n החסומים על ידי תיבה בגודל W×L×K, כאשר W הוא רוחב התיבה, L הוא אורכה, ו-K הוא עומקה. כמו כן, נדרשנו להתאים לשלושה מימדים את סדר הוספת התאים באלגוריתם ה-transfer matrix. נאלצנו לבצע שינוי משמעותי בקידוד החתימות (בגלל העדר לינאריות הגבול), אך לא במידע המיוצג בהן. מקידוד החתימות התלת מימדיות ניתן היה לדעת מהם החלקים הקשירים (ב-polycubes הנבנים) הנוגעים בגבול (המהווה "פאה שבורה"), ובאילו מפאות התיבה החוסמת נוגעים ה-polycubes.

לסיכום, אנו מנתחים בתיזה זו את סיבוכיות זמן הריצה של האלגוריתם של Jensen (בדו מימד) ומראים חסם עליון, בו החלק האקספוננציאלי מדוייק. כמו כן, אנו מציגים שתי הרחבות לאלגוריתם של Jensen, אך לא מצליחים להראות תוצאות חדשות כתוצאה משימוש יתר בזכרון. הרחבת האלגוריתם של Jensen לספירת polycubes נתקלה בקשיים דומים בצריכת זיכרון, בגין המספר העצום של חתימות. לעומת זאת, איפשרה הרחבת האלגוריתם של Redelmeier לשלושה מימדים למצוא ערך חדש בסדרת מספרי ה-polycubes: $A_3(18) = 84{,}384{,}157{,}287{,}999$. למיטב ידיעתנו, ערך זה לא פורסם קודם לכן בספרות.