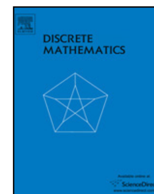
Discrete Mathematics 309 (2009) 4576–4583

Contents lists available at ScienceDirect

Discrete Mathematics

journal homepage: www.elsevier.com/locate/disc



Counting polycubes without the dimensionality curse☆

Gadi Aleksandrowicz, Gill Barequet *

Center for Graphics and Geometric Computing, Department of Computer Science, The Technion—Israel Institute of Technology, Haifa 32000, Israel

ARTICLE INFO

Article history:
Received 8 January 2008
Received in revised form 23 December 2008
Accepted 18 February 2009
Available online 17 March 2009

Keywords:
Enumeration
Polycubes
Lattice animals
Subgraph counting

ABSTRACT

d-dimensional polycubes are the generalization of planar polyominoes to higher dimensions. That is, a d-D polycube of size n is a connected set of n cells of a d-dimensional hypercubic lattice, where connectivity is through $(d-1)$-dimensional faces of the cells. Computing $A_d(n)$, the number of distinct d-dimensional polycubes of size n, is a long-standing elusive problem in discrete geometry. In a previous work we described the generalization from two to higher dimensions of a polyomino-counting algorithm of Redelmeier [D.H. Redelmeier, Counting polyominoes: Yet another attack, Discrete Math. 36 (1981) 191–203]. The main deficiency of the algorithm is that it keeps the entire set of cells that appear in any possible polycube in memory at all times. Thus, the amount of required memory grows exponentially with the dimension. In this paper we present an improved version of the same method, whose order of memory consumption is a (very low) *polynomial* in both n and d. We also describe how we parallelized the algorithm and ran it through the Internet on dozens of computers simultaneously.

© 2009 Elsevier B.V. All rights reserved.

1. Introduction

A d-dimensional polycube[1] of size (or order) n is a face-connected set of n cells on the hypercubic lattice $\mathbb{Z}^d$. *Fixed* polycubes are considered distinct if they differ in their shapes *or* orientations. The number of fixed d-dimensional polycubes of size n is denoted by $A_d(n)$. For example, Fig. 1(a,b) show the $A_3(2) = 3$ and $A_3(3) = 15$ dominoes and trominoes, respectively, in three dimensions. There are two main open problems related to polycubes: (i) The number of d-dimensional polycubes of size n, as a function of d and n; and (ii) For a fixed dimension d, the growth-rate limit of $A_d(n)$ (that is, $\lim_{n\to\infty} A_d(n+1)/A_d(n)$). Since no analytic formula for $A_d(n)$ is known, even for $d = 2$, a great portion of the research has so far focused on efficient algorithms for computing $A_2(n)$ for as high as possible values of n. In recent years, most of the effort has been focused on utilizing algebraic methods for enumerating exactly special classes of two-dimensional polyominoes (see, e.g., [4,6]).

Polyominoes and polycubes have triggered the imagination of not only mathematicians. Extensive studies of them can also be found in statistical-physics literature, where fixed polycubes are usually referred to as *strongly embedded lattice animals*. Animals play an important role in computing the mean cluster density in percolation processes, in particular those of fluid flow in random media [5], and in modeling the collapse of branched polymer molecules in dilute solution [22]. Polyominoes are also used by two-dimensional error-correcting codes [24].

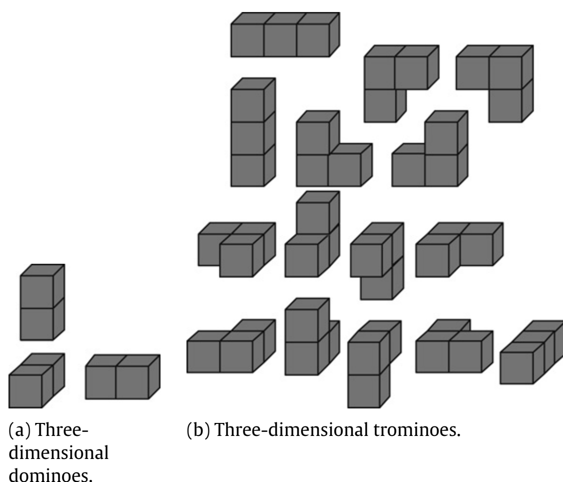
Redelmeier [23] introduced the first efficient algorithm for counting polyominoes (two-dimensional polycubes), in the sense that it generates all the polyominoes sequentially *without* repetitions. Thus, it wastes time neither on computing previously-generated polyominoes, nor on checking whether a generated polyomino is really new. The algorithm only

☆ Work on this paper has been supported in part by Jewish Communities of Germany Research Fund.

* Corresponding author. Tel.: +972 4 829 3219; fax: +972 4 829 5538.
E-mail addresses: gadial@cs.technion.ac.il (G. Aleksandrowicz), barequet@cs.technion.ac.il (G. Barequet).

[1] Term coined by Lunnon [14].

0012-365X/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.disc.2009.02.023



(a) Three-dimensional dominoes.

(b) Three-dimensional trominoes.

Fig. 1. Fixed three-dimensional dominoes and trominoes.

has to count the number of generated polyominoes. Since the algorithm generates each polyomino in constant time, its total running time is $O(A_2(n))$.[2] Redelmeier implemented his algorithm in Algol W (and for efficiency also in the PDP assembly language). The program required about 10 months of CPU time on a PDP-11/70 to compute the number of all fixed polyominoes of up to order 24.

It is known [13] that the limit $\lambda_2 := \lim_{n\to\infty} \sqrt[n]{A_2(n)}$ exists. A similar method shows that the limit $\lambda_d := \lim_{n\to\infty} \sqrt[n]{A_d(n)}$ exists for any fixed value of $d > 2$. Only less than a decade ago it was proven by Madras [17] that the limit $\lim_{n\to\infty}(A_d(n+1)/A_d(n))$ also exists and that it is equal to λ_d, for any fixed value of $d \geq 2$. None of the values of the constants λ_d (for $d \geq 2$) is to-date known. The constant λ_2 is estimated to be around 4.06 [9]. It is proven in [3] that the asymptote of λ_d is roughly $2ed$, where e is the natural base of logarithms.

The best currently-known algorithm (in terms of running time) for counting two-dimensional fixed polyominoes is that of Jensen [11]. This is a so-called transfer-matrix algorithm, which does not generate all the polyominoes. Instead, it generates classes of polyominoes with identical "boundaries", while being able to compute efficiently the number of polyominoes in each such class. Jensen was able to parallelize his algorithm and to compute $A_2(n)$ up to $n = 56$ [12].

Here is a brief summary of attempts to count three-dimensional fixed polycubes:

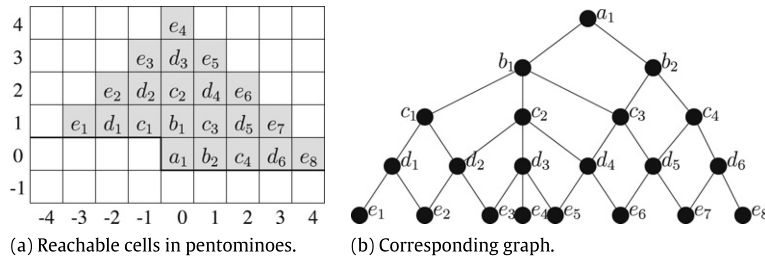
- Lunnon [15] analyzed in 1972 three-dimensional polycubes by considering symmetry groups, and computed (manually!) $A_3(n)$ up to $n = 6$. Values of $A_3(n)$ up to $n = 12$ can be derived from a subsequent work of Lunnon [16] in 1975 (see below).
- Sykes et al. [25] used in 1976 a method proposed by Martin [19] in order to derive and analyze series expansions on a three-dimensional lattice, but did not compute new values of $A_3(n)$.
- Gaunt, Sykes, and Ruskin [9] listed $A_3(n)$ up to $n = 13$ (with a slight error in $A_3(13)$).
- Gong [10] computed, in a series of attempts (in 1992, 1997, and 2004) $A_3(n)$ up to $n = 9$, 15, and 16, respectively.
- However, Flammenkamp [7] computed $A_3(17)$ already in 1999.
- Nevertheless, the correct values of $A_3(n)$ up to $n = 17$ could already be derived from data provided by Madras et al. in 1990 [18].[3]
- The authors of this paper computed $A_3(18)$ in 2006 [1].

In higher dimensions, the first work on counting polycubes that we are aware of is that of Lunnon [16], in which he counted polycubes that could fit into restricted boxes. (In fact, Lunnon counted *proper* polycubes, that is, polycubes that cannot be embedded in lower-dimensional lattices, but the numbers of all polycubes can easily be deduced from the numbers of proper polycubes.) Lunnon computed values up to $A_4(11)$, $A_5(9)$, and $A_6(8)$ (with slight errors in $A_6(7)$ and $A_6(8)$). Gaunt, Sykes, and Ruskin [9] provided values up to $A_4(11)$, $A_5(10)$, $A_6(9)$, and $A_7(9)$. Gaunt [8] provided values up to $A_8(9)$ and $A_9(9)$. In [1] we generalized Redelmeier's algorithm in a rather naive way, and computed values up to $A_4(15)$, $A_5(13)$, $A_6(9)$, $A_7(6)$, $A_8(5)$, and $A_9(4)$.

Redelmeier's original algorithm [23] counts (two-dimensional) polyominoes. Although it is not presented in this way, it is based on counting connected subgraphs in the underlying graph of the two-dimensional orthogonal lattice, that contain one

2 This is true under the regular assumption that operations on integer numbers can be done in constant time. This work is targeted, however, at situations where the handled numbers are large enough to make this issue significant, so that each operation requires time that is linear in the number of bits required to represent the respective number. In this perspective, just counting the polyominoes really takes $O(A_2(n) \log A_2(n)) = O(A_2(n)n)$ time.

3 To do this, one needs to consider all the coefficients $c_{n,e,u}$ of terms of the form $x^n b^e \lambda^u$ in [18, App. B, pp. 5346–5349], ignore all coefficients with $u > 0$, and sum up the rest. That is, $A_3(n) = \sum_e c_{n,e,0}$. Note that the symbol λ used here is not the asymptotic growth rate of polyominoes.



(a) Reachable cells in pentominoes. (b) Corresponding graph.

Fig. 2. Pentominoes as subgraphs of some underlying graph.

particular node. However, as already observed by Mertens [20], Redelmeier's algorithm does not depend on any particular property of the graph. In the generalization of this algorithm to higher dimensions [1], we first computed the respective lattice graphs, clipped to the size of the sought-after polycubes, and then applied the same subgraph-counting algorithm. The main drawback of this approach in higher dimensions is that it keeps in memory at all times the entire set of "reachable" cubes—that is, all cubes that are part of some polycube. Since the number of such cells is roughly $(2n)^d$, the algorithm becomes useless for relatively small values of d. For example, we were able in [1] to compute $A_7(n)$ up to only $n = 6$. In the current paper we significantly improve the algorithm by not keeping this set of reachable cells in memory *at all*. Instead, we maintain only the "current" polycube and a set of its immediate neighboring cells, entirely omitting the lattice graph. The latter is computed locally "on demand" in the course of the algorithm. Hence, we need to store in memory only $O(nd)$ cells. Furthermore, we parallelized the new version of the algorithm.

In a sense, our algorithm bears some resemblance with the general *reverse search* technique of Avis and Fukuda [2], which allows us to search combinatorial configurations with low time and space resources. In particular, our improved version of Redelmeier's algorithm is similar to the application of the reverse search to the enumeration of connected subgraphs [2, Section 3.4]. The main difference is that we use heavily the self-repeating structure of the square lattice. A minor difference is that we enumerate only subgraphs that contain one specific vertex; however, this only limits the search.

It is worth noting that in two dimensions, in terms of running time, Redelmeier's subgraph-counting method is inferior to Jensen's transfer-matrix method. However, the latter cannot be adapted easily to higher dimensions since it is not known how to encode efficiently the polycubes' boundaries in more than two dimensions.

In the next sections we describe Redelmeier's original algorithm, its generalization to higher dimensions, our new ingredient that eliminates the need to hold the entire underlying graph in memory, and how to parallelize the algorithm. This enabled us to compute $A_d(n)$ for values of d far beyond any previous attempt. With contemporary computing resources, the bottleneck of the newest version of the algorithm (in higher dimensions) is again its running time and not its consumed amount of memory. This bottleneck is partially remedied by our ability to run the algorithm simultaneously on many computers over the Internet.

2. The original algorithm

In this section we briefly describe Redelmeier's algorithm for counting two-dimensional polyominoes. The reader is referred to the original paper [23] 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, he needed 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 Fig. 2(a). The shaded area in this figure consists of all the *reachable* cells (possible locations of cells) of pentominoes (polyominoes of order 5). Counting pentominoes amounts to counting all the connected subgraphs of the graph shown in Fig. 2(b), that contain the vertex a_1. The algorithm [23] is shown in Fig. 3. Step 4(a) deserves some attention. By "new neighbors" we mean only neighbors of the new cell c that was chosen in Step 2, which *were not neighbors* of any cells of the polyomino prior to adding c to the polyomino. This ensures that we will not count the same polyomino more than once. We elaborate more on this in Section 3.2.

This sequential subgraph-counting algorithm can be applied to any graph (which, indeed, is exactly what we do in the next sections), and it has the property that it never produces the same subgraph twice.

3. Polycubes in higher dimensions

In this section we describe the extension of Redelmeier's algorithm to counting polycubes in orthogonal lattices in higher dimensions.

```
Initialize the parent to be the empty polyomino, and the untried set
to contain only the origin.  The following steps are repeated until
the untried set is exhausted.

 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.
```

Fig. 3. Redelmeier's algorithm (taken verbatim from [23, p. 196]).

3.1. *Naive generalization*

In [1] we generalized Redelmeier's algorithm to higher dimensions in a rather simple way. We reorganized the algorithm in two stages. In the first stage we computed the cell-adjacency graph and marked the canonical cell. In the second (main) step of the algorithm we applied the original procedure, which was nothing else but counting the connected subgraphs that contained the canonical cell. As noted above, the subgraph-counting method does not depend in any way on the structure of the graph. Thus, in order to generalize the algorithm to counting d-dimensional polycubes, we only needed to rewrite the first stage so that it would compute the respective neighborhood graph in the appropriate dimension (and up to the size of the sought-after polycubes). Then, we invoked the same subgraph-counting procedure.

Let us briefly review the computation of the lattice graph in d dimensions. Denote the coordinates of a cell as a vector $x = (x_1, x_2, \ldots, x_d)$, and regard it as a number in some base $t \in \mathbb{N}$ with d digits. We mapped the lattice cell x to the integer number $\Gamma(x) = \sum_{k=1}^{d} x_k t^{k-1}$. The number t was chosen large enough so that no two cells were mapped to the same number. The minimum possible choice of t was obviously the size of the range of coordinates attainable by reachable cells of the polycubes, that is, $t = 2n - 2$, where n is the polycube size. A clear benefit of this representation was the ability to compute neighboring cells efficiently: The images under $\Gamma(\cdot)$ of the immediate neighbors of a cell x along the kth direction were $\Gamma(x) \pm t^{k-1}$.

Originally, the canonical cell was fixed as $(0, \ldots, 0)$. Except in the dth direction, the original range of reachable coordinates was $[-(n-2), \ldots, n-1]$, so it was shifted by $n-2$ in order to have only nonzero coordinates. In the dth direction the range of reachable coordinates is $[0, \ldots, n-1]$, and so no shift was needed. Thus, the canonical cell was shifted to $\overline{0} = (n-2, \ldots, n-2, 0)$. It is easy to verify that a cell $x = (x_1, \ldots, x_d)$ is reachable if and only if $x \geq \overline{0}$ lexicographically, and reachable cells are in the d-dimensional box defined by $(0, \ldots, 0)$ and $x_M = (2n-3, \ldots, 2n-3, n-1)$. The function $\Gamma(\cdot)$ maps reachable cells to numbers in the range 0 to $M = \Gamma(x_M) = \sum_{k=1}^{d-1}(2n-3)(2n-2)^{k-1} + (n-1)(2n-2)^{d-1} = (2n-2)^{d-1}n - 1$.[4] Fig. 4 shows the algorithm for building the d-dimensional graph. The cell-neighborhood relations are kept in the array `neighbors` $[x][y]$, where x is the identity number of the current cell and y is a serial number in the range $(0, \ldots, 2d-1)$ (all possible directions in d dimensions). The subgraph-counting step starts from the cell with identity $\overline{0}$.

The cell-adjacency graph contains, then, at most $(2n-2)^{d-1}n$ nodes. Actually, only roughly one half of these nodes, that is, about $2^{d-2}(n-1)^{d-1}n$, are really reachable (see Fig. 2(a) for an illustration). In two dimensions, this number (n^2) is negligible for polyominoes whose counting is time-wise possible. For example, Redelmeier counted polyominoes of up to size 24. However, in higher dimensions the "dimensionality curse" appears. The size of the graph, and not the number of distinct polycubes, becomes the algorithm's bottleneck. For example, in order to count polycubes of size 7 in 7 dimensions, we need to maintain a graph with about 10^7 nodes. Recall that each node has $2d$ neighbors—14, in the last example. This means that even for modest values of d and n the algorithm would need hundreds of Megabytes of memory, let alone for higher values, for which computing $A_d(n)$ becomes infeasible.

3.2. *Eliminating the computation of the graph*

The main goal of this work is to eliminate the need to hold the entire graph in memory throughout the computation. Clearly, the distribution of cells of different statuses is extremely uneven: At most n cells are (at a time) in the current polycube, at most $2nd$ cells are (at a time) in the untried set (since each of the n used cells has at most $2d$ free neighbors), and all the other cells are free (unoccupied). Since the lattice graph, in any dimension, has a well-defined structure, it is not

---

[4] In fact, x_M itself is not reachable. The reachable cell with the largest image under Γ is $y = (n-2, \ldots, n-2, n-1)$ (the top cell of a "stick" aligned with the dth direction). We have $\Gamma(y) = (t^{d+1} - 2t^{d-1} - t + 2)/(2(t-1))$, where $t = 2n-2$.

```
ALGORITHM GraphOrthodD(int n, int d)
  begin
    1. o := Γ((n − 2, ..., n − 2, 0)); M := (2n − 2)^{d−1} n − 1;
    2. for i = o, ..., M do
      2.1 b := 1; counter := 0;
      2.2 for j = 1, ..., d do
        2.2.1 if i + b ≥ o then do
          2.2.1.1 neighbors[i][counter] := i + b;
          2.2.1.2 counter := counter+1;
        end if
        2.2.2 if i − b ≥ o then do
          2.2.2.1 neighbors[i][counter] := i − b;
          2.2.2.2 counter := counter+1;
        end if
        2.2.3 b := b(2n − 2);
      end for
      2.3 neighbors_num[i] := counter;
    end for
  end GraphOrthodD
```

Fig. 4. Computing the graph for a d-dimensional orthogonal lattice.

necessary to create it in advance. Instead, one could create the graph locally "on demand". Given a cell, one could compute its neighbors on-line and always in the same order. In principle, instead of maintaining a huge vertex set (lattice cells) of the graph, and keeping the status of each cell (occupied, in the untried set, or free), we could keep only the first two small sets of cells, and deduce that a cell is free from the fact that it does not belong to any of these two sets.

The major potential obstacle is that, in principle, we need to distinguish between free cells that, in the course of the algorithm, were already part of some counted polycubes, and are thus "done with", and cells that were not yet part of any polycube. It turns out that this differentiation is redundant, and we can manage without keeping this information for each free cell.

To show this, let us formally prove the correctness of the algorithm.[5] We need to prove that even though the graph contains plenty of cycles, no subgraph is obtained twice with different orders of nodes. It will then become clear that no information about the nodes of the underlying graph should be maintained in the course of the algorithm. From the definition of the algorithm, it is clear that every polycube will be counted at least once. This is because a polycube is a *connected* set of cells that includes the origin, and so any exhaustive procedure of adding cells by connectivity will reach any polycube. The more delicate issue is to prove that every polycube is counted only once.

Theorem 1. *Redelmeier's algorithm counts every connected subgraph (polycube) that contains the origin exactly once.*

Proof. Denote a polycube of size n as an ordered set $(a_1, \ldots, a_n)$, where the cells a_i are ordered according to their arrival at the polycube in the course of the algorithm. (Obviously, $a_1 = \underline{0}$ in all polycubes.) Assume for contradiction that the same polycube P is generated twice, i.e., there are two ordered sets $A = (a_1, \ldots, a_n)$ and $B = (b_1, \ldots, b_n)$, generated by the algorithm, which differ only in their permutations of cells. Let A be the set that was generated first, and k (for $2 \le k \le n$) be the smallest index for which $a_i \ne b_i$. (It is clear from the definition of the algorithm that the same ordered set cannot be generated twice.)

Let us focus on the recursion subtree whose root is the polycube $P' = (a_1, \ldots, a_{k-1})$. We can assume by induction (on the level of recursion) that P' is generated only once in the course of the algorithm. First, a_k is moved from the untried set to the polycube. Then, in a lower subtree of the recursion, *all* the sets whose prefix is $(a_1, \ldots, a_{k-1}, a_k)$ are explored. Following this, a_k is removed from the polycube. Possibly, other cells are moved (one at a time) from the untried set to the polycube, and subtrees of the recursion are performed. Finally, b_k is moved from the untried set to the polycube, forming the set $(a_1, \ldots, a_{k-1}, b_k)$, and the recursion goes down again. Our goal is to prove that in this subtree of the recursion, the cell a_k cannot be added to the polycube, and, hence, P cannot be generated again.

To this aim we need a few observations:

1. When P' is generated the first time, both a_k and b_k are in the untried set. This is because a recursion subtree fully exhausts all the polycubes whose prefix is the one at the top of that subtree. Thus, A and B must be generated at the same subtree, and for this to happen, the untried set must contain both a_k and b_k when P' is generated.
2. Executing a subtree of the recursion does not change the untried set at the top of the subtree; down the recursion, cells are added and removed to the untried set in a symmetric way.[6]

[5] Redelmeier [23, p. 195] only gave a brief description of the course of the algorithm, from which the main point of the current paper could not be inferred. The reader is also referred to the discussion in [20, Section 2].

[6] In his actual implementation of the algorithm, Redelmeier [23, p. 197] took advantage of this fact to avoid the creation of copies of the untried set in the recursive calls; instead, he used only one expanding and shrinking stack for maintaining all versions of this set.

3. By definition, a cell is in the untried set if it is an immediate neighbor of the current polycube. A cell is added to this set only when it becomes a new neighbor; later in the course of the algorithm, the cell cannot remain in the untried set if it is no longer a neighbor of the polycube. This follows from the order of the algorithm: once a cell enters the untried set, this set is fully exhausted before the recursion goes up and shrinks the polycube at the top of this subtree.

From these observations we deduce that both a_k and b_k are neighbors of P'. (Otherwise, they could not be added to the untried set when P' was first created; in addition, since P' is created only once, a_k and b_k are added to the untried set at the same time.) After the termination of the recursion subtree, at the top of which $(a_1, \ldots, a_{k-1}, a_k)$ is, a_k is removed from the polycube (but not returned to the untried set), and later b_k is added to the polyomino. At this time (and down this subtree), a_k *cannot* be added to the polycube, because it cannot be added to the untried set. This follows from the fact that a_k cannot become a *new* neighbor of the polycube, since it is already a neighbor of it. (Recall the definition of "new neighbors.")

This contradicts the assumption that $(b_1, \ldots, b_n)$ contains a_k as a cell, and the claim follows. □

The above proof implies that after a_k is removed from the current polycube, it will never again be added to the untried set (and hence neither to the polycube) as long as the recursion continues with P' as its root. However, when the recursion is unfolded, and the current polycube is shrunk so that it is a subset of P' plus some other cells, it is certainly possible that a_k will be added again to the untried set and later to the polycube. Nevertheless, no future polycube will have P' as a subset.

The above proof also implies that we do not need to store any information about the free cells. This is because in order to avoid counting the same polyomino twice, all that is required is the current polyomino and its neighboring cells. To maintain this information we only need to be able to explore the graph locally "on demand," and for each neighbor c' of a cell c which is moved from the untried set to the polycube, determine whether or not it is now a *new* neighbor. The first task is easy to accomplish, since the structure of the orthogonal lattice is well defined, and so is the definition of the mapping $\Gamma(\cdot)$. The second task is also feasible: Such a cell c' is a new neighbor if and only if its only neighboring cell in the current polycube is c. This is easy to determine using the polycube's set of cells.

3.3. Counting proper polycubes

As reported in Section 5, our counts of $A_6(7)$ and $A_6(8)$ did not match those of Lunnon [16]. As mentioned in the introduction, Lunnon actually computed $\mathrm{DX}(n, d)$, the number of *proper* polycubes of size n in d dimensions. (Proper d-dimensional polycubes span all the d dimensions, and, thus, cannot be embedded in $d - 1$ dimensions.) According to Lunnon's formula, $A_d(n) = \sum_{i \le d} \binom{d}{i} \mathrm{DX}(n, i)$. In order to trace the source of the discrepancies, we modified our program to also count proper polycubes.

To this aim, we should have maintained a "dimension status" that should have counted, for each dimension $1 \le i \le d$, how many cells of the current polycube have the ith coordinate different than $\overline{0}$, the cell at the origin. Each addition or deletion of a cell to/from the polycube should have been accompanied by updating the appropriate counter in the dimension-status record. To achieve this, we needed an efficient implementation of $\Gamma^{-1}(\cdot)$, a function that maps numbers (cell ids) back to their original source cells with d coordinates.

However, for efficiency of the program, we wanted to avoid altogether the use of such a function $\Gamma^{-1}(\cdot)$. When we add a cell c to the polycube, instead of increasing by 1 the counters of all its dimensions relative to the origin (which are many, and not simple to identify), we increase only the counter of its dimension relative to the cell c' due to which it was previously put in the untried set (which is unique, and is known while c is added). This specific counter is decreased by 1 when cell c is removed from the polycube. During the entire period in which c belongs to the polycube, this dimension of the polycube is used. Thus, although the counter of dimension i does not really count the number of cells that occupy the ith dimension, it is easy to see that it is still positive whenever there is any cell that does so. Obviously, the cell at the origin does not occupy any dimension. In conclusion, a polycube is proper if and only if all the d counters in the dimension-status record are greater than zero.

3.4. Parallelization

Since now the bottleneck of the algorithm is the running time, we also parallelized the new version of Redelmeier's algorithm. Mertens and Lautenbacher [21] did this for the two-dimensional version of the problem, while we do this for any dimension. Since, as is already observed above, the execution of any subtree of the recursion does not change the contents of the data structures at the top of the subtree, the computation of the subtree can be done independently and in parallel, while the main algorithm skips the subtree.

In practice, the main procedure recurses only until some level $k < n$ (for some fixed dimension d). Regardless of the value of n, this induces $A_d(k)$ subtrees, which can be assigned independently to the processors we have at hand. The results for all levels $k < m \le n$, and, in particular, $m = n$, are collected from all the processors, and their summation yields the values of $A_d(m)$. This can be done for counting both proper and all (proper and improper) polycubes.

Here is a brief description of our parallel implementation. The main server runs on one computer and is continuously waiting for messages sent by Internet clients through a dedicated port. When a new client introduces itself to the server, the latter responds with a request to compute a specified range of the $A_d(k)$ subtrees mentioned above. The client then invokes

a polycube-counting program, which performs the entire computation up to level k, and recurses to the subtrees only in the specified range. By the end of the computation, the program reports the results to the client, which, in turn, forwards the results to the server. The server keeps track of all tasks (ranges of subtrees) assigned to the clients, being able to reassign a task to another client if no results were reported by the original client to which the task was assigned. The programs for the Internet server and client were written in Ruby, while versions of the polycube-counting program were written in C and C++. The client can be run in either a Windows or Linux environment.

4. Complexity analysis

For the analysis we assume that $n > d$, otherwise the polycubes cannot really span all the d dimensions. In d dimensions, there are $\Theta((2n)^d)$ distinct reachable cells, so the amount of memory (and time) needed to store (and process) the identity of a single cell is $\Theta(d \log n)$. (Note that in other studies this factor was considered as $\Theta(1)$ or $\Theta(d)$.)

We maintain two data structures: The set of currently-occupied cells (the current polycube) and the set of untried cells (neighbors of the polycube). The size of the current polyomino is at most n, while the size of the untried set is at most $2dn - 2(n-1) = 2((d-1)n+1)$. That is, the amount of cells stored by the algorithm is $\Theta(dn)$. For this we need $\Theta(d^2 n \log n)$ space. For a fixed value of d, this is $\Theta(n \log n)$.

Since the new version of the algorithm does not compute the lattice graph, the size of the latter does not directly affect the algorithm's running time. The important factor is the number of operations (additions and deletions of cells) on the untried set. This number is proportional to $A_d(n)$, the number of counted polycubes. (More precisely, the number of these operations is proportional to the total number of d-dimensional polycubes of *all* sizes up to n, but the polycubes of size n outnumber all the smaller polycubes.)

We maintain the current polycube as a balanced binary tree (sorted by $\Gamma(\cdot)$ values), and since it always contains up to n cells, each search, addition, or deletion requires $O(\log n)$ steps (where each step requires $O(d \log n)$ time, as explained above). The untried set should be a queue-like structure which supports two types of operations: (1) Adding new cells at the rear and removing cells (to be added to the current polycube) from the front. (2) Searching for a node, so that a cell will not be added twice to the set. An appropriate tree-like structure (or two trees with cross references) can do the job. Since the size of the untried set is $O(dn)$, each operation on the set requires $O(\log d + \log n) = O(\log n)$ steps.

When a cell c is moved from the untried set to the polycube, its $2d$ neighbors are computed, and each such neighbor c' is tested to see if it is a *new* neighbor. Recall that c' is a new neighbor if it is free and among all its own $2d$ neighbors, *only* c belongs to the current polycube. Hence, each such test can be done in $O(d \log n)$ steps by using the polycube structure, for a total of $O(d^2 \log n)$ steps for all the neighbors of c. This can be done more efficiently if we keep a secondary set of cells that were removed from the polycube but are still its neighbors, thus, they cannot become "new neighbors" as long as they maintain this property. The size of this set is comparable to that of the untried set, that is, $\Theta(dn)$. For each cell c in this secondary set we maintain the count of its neighbors that belong to the polycube, cells due to which the cell c cannot become a new neighbor. When a cell is removed from the polycube, not only it is put in the secondary set and its count is initialized appropriately, but also the "neighbor counts" of all its neighbors (which are in this set) are decreased by 1. When the neighbor count of a cell reaches 0, the cell is removed from this secondary set, thus, it may again become a new neighbor. With this set we can test in only $O(\log n)$ steps whether or not a cell c' is a new neighbor of the polycube. If it is not new, then it is not added to the untried set. (A byproduct of this is that the data structure implementing the untried set does not have to support the search operation, since we will never attempt to add to it a cell which is already there.)

Let us now sum up the amount of work (in steps) performed in each operation. Recall that a single operation on any of the lists requires $O(\log n)$ steps. Removing a cell c from the polycube takes $O(\log n)$ steps. We also add c to the secondary set in $O(\log n)$ steps, and initialize its count in $O(d \log n)$ steps by checking which of its $2d$ neighbors belong to the polycube. In addition, we need to update the counts of the $O(d)$ neighbors of c in the secondary list. For each such neighbor we compute its id in $O(1)$ steps, search for it in the list in $O(\log n)$ steps, update its count (if it is in the list) in $O(1)$ steps, and remove it from the list (if necessary) in $O(\log n)$ steps. This amounts to $O(\log n)$ steps per neighbor, for a total of $O(d \log n)$ steps for all neighbors of c.

Moving a cell c from the untried set to the polycube takes $O(\log n)$ steps. In addition, we need to compute which cells, neighboring to c, become new neighbors of the polycube. For each such neighbor we compute its id in $O(1)$ steps, and search for it both in the polycube and in the secondary list in $O(\log n)$ steps. Then, we either add it to the untried set (if it is neither in the polycube nor in the secondary set) in $O(\log n)$ steps, or update its count (if it is only in the secondary set) in $O(1)$ steps. This amounts to $O(\log n)$ steps per neighbor, for a total of $O(d \log n)$ steps for all neighbors of c.

Overall, each basic operation of the algorithm (adding or removing a cell) requires $O(d \log n)$ steps. Since the algorithm performs $A_d(n)$ operations, and each step requires $O(d \log n)$ time, the total running time is $O(A_d(n) d^2 \log^2 n)$. In fact, we implemented the three sets as one hash table. Practically, this reduces only the polynomial term, while the major factor, $A_d(n)$, is exponential in n, where the base of the exponent is an unknown constant that depends solely on d (e.g., $\sim$4.06 for $d = 2$).

In the variant of the program that identifies proper polycubes we need to invest $O(\log d)$ time per generated polycube to maintain the dimension-status record (by accessing the count of the appropriate dimension), and $O(d \log d)$ time for checking whether or not a polycube is proper (by processing the counts of all the d dimensions). This does not change asymptotically the running time of the algorithm.

Table 1
Numbers of fixed d-dimensional polycubes (new values in boldface).

n	$A_6(n)$	$A_7(n)$	$A_8(n)$
1	1	1	1
2	6	7	8
3	66	91	120
4	901	1,484	2,276
5	13,881	27,468	49,204
6	231,008	551,313	1,156,688
7	4,057,660	11,710,328	28,831,384
8	74,174,927	259,379,101	750,455,268
9	1,398,295,989	5,933,702,467	
10	**27,012,396,022**	**139,272,913,892**	

5. Results

We implemented the algorithm in C and C++, and ran the serial program locally in an MS Windows environment on an IBM X500 with four 2.4 GHz XEON processors and 3.5 GB of RAM. The parallel version ran over the Internet on dozens of computers simultaneously, offering a wide range of processor frequency and available amount of memory.

Table 1 shows the values of $A_d(n)$ obtained in 6, 7, and 8 dimensions. New values, which are tabulated for the first time, are shown in bold font. Values of $A_8(n)$ confirm previously-published values which relied on unproven formulae. The nonparallel version of the program computed the reported values of $A_6(\cdot)$ and $A_8(\cdot)$ in about 26 days and a little more than a week, respectively. The values of $A_7(\cdot)$ were computed by the parallel program in about a week, using a dozen computers for gathering a total of 91 days of CPU. All the old values in the table agree with previous publications, including ours [1], except for the values of $A_6(7)$ and $A_6(8)$ computed by Lunnon [16]. From his calculations of proper polyominoes, one can infer the values 4038,205 and 71,976,512, respectively. However, the now-known [3] explicit formulae $\mathrm{DX}(n, n-1) = 2^n n^{n-3}$ and $\mathrm{DX}(n, n-2) = 2^{n-3} n^{n-5}(n-2)(2n^2 - 6n + 9)$ confirm our counts.

6. Conclusion

In this paper we presented an efficient implementation of Redelmeier's algorithm for counting fixed high-dimensional polycubes. We used our program to compute new terms of the series $A_6(n)$ and $A_7(n)$. The main contributions of this work is eliminating the need to store the entire lattice graph (in which polycubes are sought), and parallelizing the algorithm. The parallel version was actually implemented and run over the Internet. We plan to continue running it and compute even more yet unknown values of $A_d(n)$.

References

[1] G. Aleksandrowicz, G. Barequet, Counting d-dimensional polycubes and nonrectangular planar polyominoes, in: Proc. 12th Ann. Int. Computing and Combinatorics Conf., Taipei, Taiwan, in: Lecture Notes in Computer Science, vol. 4112, Springer-Verlag, 2006, pp. 418–427, full version to appear in Int. J. Comput. Geom. Appl.
[2] D. Avis, K. Fukuda, Reverse search for enumeration, Discrete Appl. Math. 65 (1996) 21–46.
[3] R. Barequet, G. Barequet, G. Rote, Formulae and growth rates of high-dimensional polycubes, Combinatorica (in press).
[4] M. Bousquet-Mélou, A. Rechnitzer, Lattice animals and heaps of dimers, Discrete Math. 258 (2002) 235–274.
[5] S.R. Broadbent, J.M. Hammersley, Percolation processes: I. Crystals and mazes, Proc. Camb. Philos. Soc. 53 (1957) 629–641.
[6] A. Del Lungo, E. Duchi, A. Frosini, S. Rinaldi, On the generation and enumeration of some classes of convex polyominoes, Electron. J. Combin. 11 (2004) pp. 46.
[7] The on-line encyclopedia of integer sequences. http://www.research.att.com/njas/sequences.
[8] D.S. Gaunt, The critical dimension for lattice animals, J. Phys. A: Math. Gen. 13 (1980) L97–L101.
[9] D.S. Gaunt, M.F. Sykes, H. Ruskin, Percolation processes in d-dimensions, J. Phys. A: Math. Gen. 9 (1976) 1899–1911.
[10] K. Gong, web page. http://kevingong.com/Polyominoes/Enumeration.html.
[11] I. Jensen, Enumerations of lattice animals and trees, J. Statist. Phys. 102 (2001) 865–881.
[12] I. Jensen, Counting polyominoes: A parallel implementation for cluster computing, in: Proc. Int. Conf. on Computational Science, Part III, Melbourne, Australia and St. Petersburg, Russia, in: Lecture Notes in Computer Science, vol. 2659, Springer, 2003, pp. 203–212.
[13] D.A. Klarner, Cell growth problems, Canad. J. Math. 19 (1967) 851–863.
[14] W.F. Lunnon, Counting polyominoes, in: A.O.L. Atkin, B.J. Birch (Eds.), Computers in Number Theory, Academic Press, London, 1971, pp. 347–372.
[15] W.F. Lunnon, Symmetry of cubical and general polyominoes, in: R.C. Read (Ed.), Graph Theory and Computing, Academic Press, New York, 1972, pp. 101–108.
[16] W.F. Lunnon, Counting multidimensional polyominoes, Comput. J. 18 (1975) 366–367.
[17] N. Madras, A pattern theorem for lattice clusters, Ann. Combin. 3 (1999) 357–384.
[18] N. Madras, C.E. Soteros, S.G. Whittington, J.L. Martin, M.F. Skeys, S. Flesia, D.S. Gaunt, The free energy of a collapsing branched polymer, J. Phys. A: Math. Gen. 23 (1990) 5327–5350.
[19] J.L. Martin, Computer techniques for evaluating lattice constants, in: C. Domb, M.S. Green (Eds.), in: Phase Transitions and Critical Phenomena, vol. 3, Academic Press, London, 1974, pp. 97–112.
[20] S. Mertens, Lattice animals: A fast enumeration algorithm and new perimeter polynomials, J. Statist. Phys. 58 (1990) 1095–1108.
[21] S. Mertens, M.E. Lautenbacher, Counting lattice animals: A parallel attack, J. Statist. Phys. 66 (1992) 669–678.
[22] P.J. Peard, D.S. Gaunt, $1/d$-expansions for the free energy of lattice animal models of a self-interacting branched polymer, J. Phys. A: Math. Gen. 28 (1995) 6109–6124.
[23] D.H. Redelmeier, Counting polyominoes: Yet another attack, Discrete Math. 36 (1981) 191–203.
[24] M. Schwartz, T. Etzion, Two-dimensional cluster-correcting codes, IEEE Trans. Inform. Theory 51 (2005) 2121–2132.
[25] M.F. Sykes, D.S. Gaunt, M. Glen, Percolation processes in three dimensions, J. Phys. A: Math. Gen. 10 (1976) 1705–1712.