# Squaring the square – New methods for determining the number of perfect square packings

H. Langenau

Preprint 2017-7

Preprintreihe der Fakultät für Mathematik
ISSN 1614-8835

# Squaring the square

**New methods for determining the number of perfect square packings**

Holger Langenau[*]

January 5, 2018

Given a square with integer side length $n$, we ask for the number of different ways to divide it into sub-squares, considering only the list of parts. We enumerate all possible lists and check whether a placement with those squares is possible. In order to do this, we propose a new algorithm for creating perfect square packings.

*Keywords: Square, Packing, Partition, Backtracking*
*Mathematics Subject Classification (2000):* Primary 52C15;
Secondary 05B40, 90C35

## 1 Introduction and basic terminology

A perfect partition of an $n \times n$ square is a collection of squares with integer side lengths that can be arranged in such a way that they fill the square and leave no space unoccupied. We ask how many perfect partitions exist for a given number $n$, i.e., we are only interested if for a given $n$-tuple $(p_n, \dots, p_1)$, representing a set of $p_1$ $1 \times 1$ squares, $p_2$ $2 \times 2$ squares, and so on, fulfilling $\sum_{k=1}^{n} p_k k^2$, there exists a placement in such a way that no two squares overlap and the large square is filled completely, not counting multiple arrangements for the same partition. These partition numbers make up the sequence A034295 in OEIS [7], and were known up to $n = 16$, namely, 1, 2, 3, 7, 11, 31, 57, 148, 312, 754, 1559, 3844, 7893, 17766, 37935, 83667. Figure 1 shows all possible valid partitions for $n = 4$. This can easily be verified by hand.

While we can easily enumerate all partitions of $n^2$ into squares, the main problem lies in determining whether a valid placement of those squares can be found. This problem is known to be NP-complete, and has been studied for quite a while, see, e.g., [5, 6, 4, 1, 2]. A good algorithm to generate such a placement, if one exists, is the one given by Hougardy [5], which works also for rectangles and non-perfect packings. Here, we provide a new algorithm, that, in most of the cases, works a lot faster than the one given by Hougardy,

[*]Fakultät für Mathematik, TU Chemnitz, 09107 Chemnitz, Germany
e-mail: holger.langenau@mathematik.tu-chemnitz.de

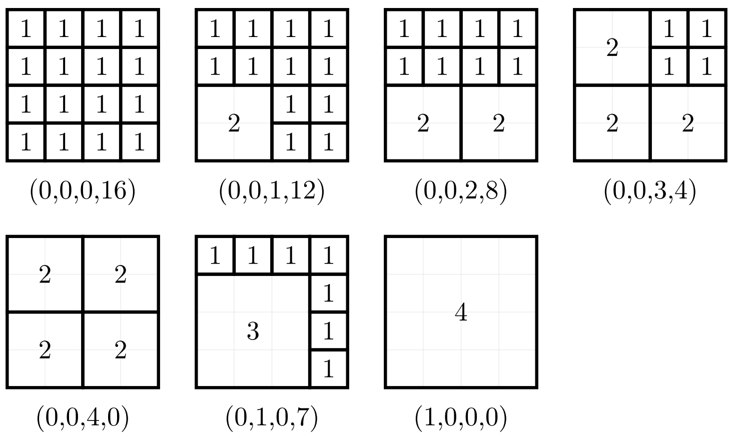

Figure 1: Illustration of the possible partitions of a $4 \times 4$ square. The only feasible partition missing in this list is $(0, 1, 1, 3)$, but there is no placement with those squares, since placing a $2 \times 2$ square next to a $3 \times 3$ square would require at least a square of size $5 \times 5$. Thus, this partition is invalid.

Table 1: List of new terms for OEIS sequence A034295

| $n$ | $a(n)$ | $n$ | $a(n)$ |
|---|---|---|---|
| 17 | 170165 | 21 | 3154006 |
| 18 | 369698 | 22 | 6424822 |
| 19 | 743543 | 23 | 12629174 |
| 20 | 1566258 | 24 | 25652807 |

at least for square packings. Employing this algorithm, we were able to identify more terms of the sequence, which are given in Table 1.

We want to fix some terminology which we will need later on. A *(feasible) partition* of an $n \times n$ square is a set of squares of side length $k \times k$, $1 \leq k \leq n$ in such a way that the total area of those squares equals $n^2$, the area of the partitioned square. We can represent a partition as an $n$-tuple $(p_n, \dots, p_1)$, of non-negative integers $p_k$, where each $p_k$ stands for the number of squares of size $k$ in this partition. Therefore, we denote by

$$\mathcal{F}_n := \Big\{(p_n, \dots, p_1) \in \mathbb{N}_0^n : \sum_{k=1}^{n} p_k k^2 = n^2\Big\} \tag{1}$$

the set of all feasible partitions. A partition is called *valid*, if there exists a *placement* $\{(x_i, y_i, s_i)\}_{i=1}^N$, $N = \sum_{k=1}^n p_k$, with (see [5])

$$\begin{gathered} 0 \leq x_i \leq n - s_i,\ 1 \leq i \leq N, \\ 0 \leq y_i \leq n - s_i,\ 1 \leq i \leq N, \\ x_i + s_i \leq x_j \vee x_j + s_j \leq x_i \vee y_i + s_i \leq y_j \vee y_j + s_j \leq y_i,\ 1 \leq i < j \leq N, \\ |\{i \in \{1, \dots, N\} : s_i = k\}| = p_k,\ 1 \leq k \leq n. \end{gathered}$$

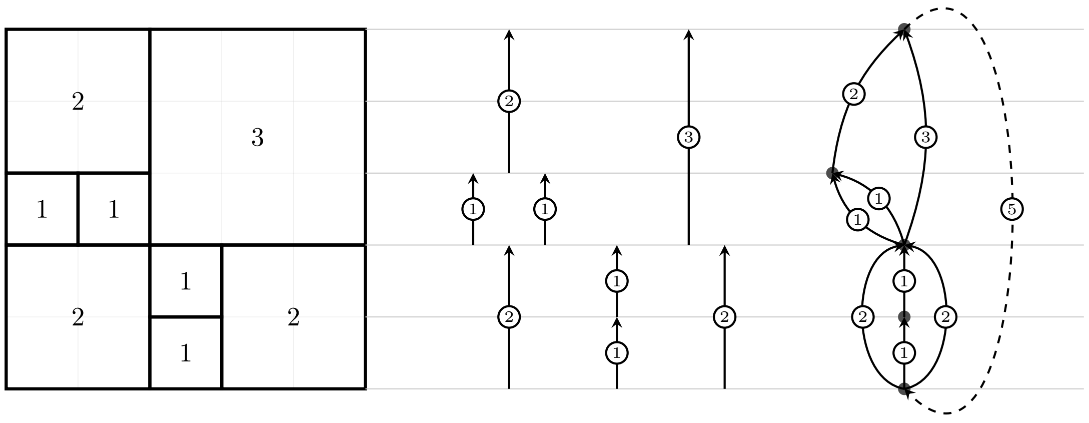

Figure 2: Translating a partition to a network flow. The squares are replaced by arcs with a weight and length matching the square size and, subsequently, each level is reduced to a single node. The resulting graph is a network and each node satisfies Kirchhoff's rule.

That is, no two squares overlap and all together fit inside the larger square. Given (1), this is a perfect placement. The set of all valid partitions is denoted by $\mathcal{P}_n \subseteq \mathcal{F}_n$. Therefore, we are interested in the number $|\mathcal{P}_n|$, the cardinality of $\mathcal{P}_n$.

## 2 Checking whether a given partition is valid

The central problem of the question stated above is to check whether a given partition is valid, i.e., if there exists a corresponding placement. The algorithms we found so far all try to create the placement immediately. Here, we present an algorithm which is a two-stage process. First, we check if some necessary condition can be fulfilled. If this is the case, we can use the information gathered during this check to determine a placement, if one exists. This condition has also been considered by Brooks, Smith, Stone, and Tutte [3]. They observed that a valid placement can be translated to a graph representing a network. All horizontal borders between two squares become the nodes which are connected by weighted edges, corresponding to the squares between those borders. If we consider the weights as electric current, the nodes all fullfil Kirchhoff's rule. An example is given in Figure 2.

Since we do not know the placement beforehand, we now have to find an arrangement of the edges in such a way that we obtain a valid network. Moreover, we do not know whether we need multiple nodes on each level. Therefore, we consider each level as a single node.

The algorithm works as follows. We start with $n + 1$ nodes $0, 1, \ldots, n$, and set their distance to be $d(i, j) = |j - i|$. Let node 0 be the source of a network, with initial charge $n$. For each square of size $s$ in the partition that is going to be checked, we assign a so-called ruler. This is an edge-to-be of given length $s$ whose position has to be determined later on. The length of the ruler also represents the capacity in the flow that we will generate.

In the node with positive charge that has the smallest index, we try to place a matching ruler, i.e., it first has to satisfy the condition that the sum of this index and the length of the ruler are not larger than $n$, and second that Kirchhoff's rule must not be violated, i.e., the capacity of the ruler must not exceed the charge of this node. Due to our model, we assume that the capacity is always fully used. If in the current step no ruler can be placed in this way, we backtrace and try with other rulers.

When all rulers are placed, the sink $n$ has charge $n$ and all other nodes have charge 0. We then check whether there is a placement of squares that corresponds to the generated flow graph. If this is the case, we are done and have found a valid placement. Otherwise, we backtrace again and try to find other valid flow graphs. If no such flow graph could be found or for none of the ones we found there was a valid placement, there is no such placement and the algorithm terminates with a negative answer.

Algorithm 2.1 gives a recursive implementation of this procedure. Here, $p = (p_n, \ldots, p_1)$ denotes a partition, $k$ is the size of the square which will be placed, $E$ is the set of edges, and $f$ is the flow state. A flow state is an $(n+1)$-tuple $(c_0, c_1, \ldots, c_n)$ with $\sum_{i=0}^{n} c_i = n$, where $c_i$ denotes the excess charge in node $i$. We call the entry $c_n$ the terminal charge. We note that the we use the term partition here although it is just a collection of squares. The map $R_k$ is given by

$$R_k : \mathbb{N}_0^n \to \mathbb{N}_0^n, \quad (p_n, \ldots, p_{k+1}, p_k, p_{k-1}, \ldots, p_1) \mapsto (p_n, \ldots, p_{k+1}, p_k - 1, p_{k-1}, \ldots, p_1),$$

i.e., it reduces the partition by one square of size $k$. The function $F_k$ is defined as

$$\begin{aligned} F_k : \mathbb{N}_0^{n+1} \to \mathbb{N}_0^{n+1}, &(f_0, \ldots, f_{l-1}, f_l, f_{l+1}, \ldots, f_{l+k-1}, f_{l+k}, f_{l+k+1}, \ldots, f_n) \\ &\mapsto (f_0, \ldots, f_{l-1}, f_l - k, f_{l+1}, \ldots, f_{l+k-1}, f_{l+k} + k, f_{l+k+1}, \ldots, f_n), \end{aligned}$$

where $l = \operatorname{argmin}_{0 \le j \le n}\{f_j > 0\}$. Effectively, this moves a charge of $k$ from node $l$ to node $l + k$.

The starting parameters of this algorithm are the partition which we want to check as $p$, a flow state $f$ of $(n, 0, \ldots, 0)$, i.e., the full charge is in the bottom node, an empty set of edges $E$ and each $k \in \{1, \ldots, n\}$ for which $p_k > 0$, until we get a positive result.

The subroutine CheckNetwork checks whether for the given edges a horizontal placement of squares exist. This will be described later.

**Algorithm 2.1** Determining a flow.

```
1: function GenerateFlow($p, f, k, E$)
2:     $l \leftarrow \text{argmin}_{0\le j\le n}\{f_j > 0\}$        ▷ Determine leftmost set index in flow
3:     if $f_l < k$ or $l + k > n$ then        ▷ Check whether square fits in this flow
4:         return false
5:     end if
6:     $p' \leftarrow R_k p$        ▷ Take square of size $k$ from partition
7:     $f' \leftarrow F_k(f)$        ▷ Advance flow
8:     $E' \leftarrow E \cup \{(l, l+k)\}$        ▷ Add edge to set of edges
9:     if $p' = (0, \dots, 0)$ then        ▷ All squares have been used
10:        if CheckNetwork($E'$) then        ▷ Try to place the squares
11:            return true
12:        else
13:            return false
14:        end if
15:    end if
16:    for all $j \in \{1 \le j \le n : p'_j > 0\}$ do
17:        if GenerateFlow($p', f', j, E'$) then
18:            return true
19:        end if
20:    end for
21:    return false
22: end function
```

Although we gave a recursive definition here, we note that in the actual implementation a stack is used. Since the stack depth coincides with the numbers of edges already placed, we know the maximal size beforehand, knowing the total number of squares in the partition.

Starting with the largest fitting square on the current lowest level, we try to advance the flow. If this is not possible, we backtrace and continue with the next smaller square. Doing so, we would check every permutation of the squares. In the following, we will define a number of pruning rules used in this backtracking algorithm.

## Pruning Rule I: Limit flow level count

The number of non-zero non-terminal charges, the flow levels, must not exceed the number of squares left to place, i.e., given a partition $p = (p_n, \dots, p_1)$ and a flow state $f = (c_0, \dots, c_n)$, the following must hold

$$|\{i \in \{0, \dots, n-1\} : c_i > 0\}| \le \sum_{j=1}^{n} p_j.$$

By inserting an edge in the network, we reduce the number of squares left by one. Furthermore, we can reduce at most one charge to zero. In the end the number of flow levels, has to be zero since the whole charge has been transferred to the terminal charge. If there are more flow levels than squares left, it is not possible anymore to reduce all to zero.

This check can be done in constant time. On each placement, we just need an update that is scale invariant.

## Pruning Rule II: Failed horizontal placement

After finding a valid flow, we check whether the squares can be arranged horizontally in such a way that we get a valid placement. Anticipating the method for doing this, this is done from the bottom up. If such a placement is not possible, there is a highest level smaller than $n$ up to which this was possible. Since we change the flow only in the lowest positive charge, all edges placed higher than this highest level can be dropped. The problem occurred earlier. Therefore, we can backtrace to a state with this level.

Although this effectively solves another NP-hard problem (it is just a reduced form of the original problem), this requires almost no additional computation, since the check has to be done anyway.

## Pruning Rule III: Lowest level must not be to high

All squares have to be placeable. If there is a square left whose size is larger than the distance of the lowest level (the smallest index of a positive charge), it is not possible anymore to place this square, and we have to backtrace. This can be improved a little bit. Let $K$ be the size of the largest square, $L$ the lowest level, i.e.,

$$K := \underset{1\le k\le n}{\operatorname{argmax}}\{p_k > 0\}, \quad L := \underset{0\le l\le n-1}{\operatorname{argmin}}\{c_l > 0\}.$$

The condition above then reads that we backtrace if $d := n - L < K$. Now, if $d = K$ all squares of this size have to be placed on this level. Therefore, the total charge on this level has to be enough for all, in other words

$$K \cdot p_K \le c_L$$

must hold. Since we keep track of the largest square and the lowest level at all times, no search is necessary and we can perform this in constant time.

## Pruning Rule IV: Flow defect

Fix some square size $u$. Next, replace all larger squares by the appropriate amount of squares of size $u$ that can be packed into them. This is the number of squares that have to be placed. If an estimate on the number of squares of size $u$ that can be packed is smaller then we have to backtrace.

This requires a more detailed explanation. For this we go back to the placement of squares rather than edges in a graph. For each flow state the best configuration to allow for the most squares of one size to be placed is a stairway. That is, we have plateaus

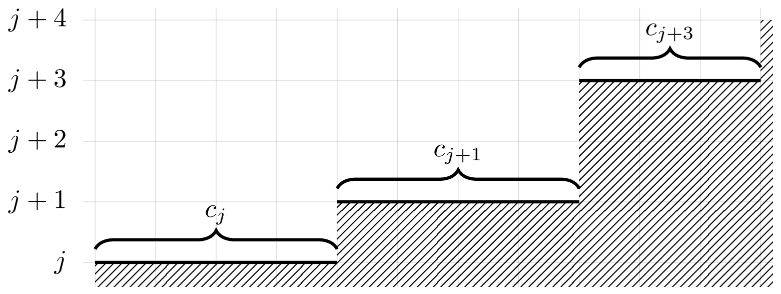

Figure 3: Most favorable layout of a flow state.

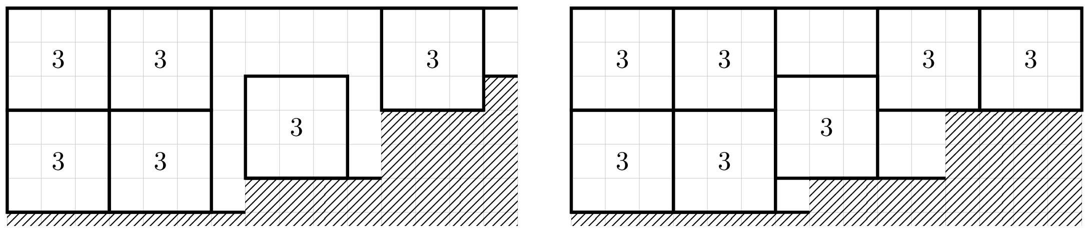

Figure 4: Example for different flow defects. In the left picture we have $D_3(f) = 2$ and in the right $D_3(f) = 3$. A maximal placement of squares of size 3 is shown. In the right picture we can place one square more, due to the overhang that is now possible.

on each level with a width reflecting the charge. Figure 3 gives an idea of the situation. The flow defect for a unit square size $u$ of a flow state $f = (c_0, \dots, c_n)$ is given by

$$D_u(f) \coloneqq \sum_{j=0}^{n-u} (c_j \bmod u).$$

Note that charges to close to the top are not accounted for. This is due to the fact that no square of size $u$ can be placed higher than the level $n - u$. The flow defect is a measure of how good squares of size $u$ can be packed on such a stairway. If $D_u(f) < u$ we could pack towers of such squares on each plateau, independently of all the others, due to the fact that a tower that is one part over the lower plateau and one part on the higher plateau can only be as high as the distance of the higher plateau to the top. It would therefore be more effective to fully pack the tower on the lower plateau. Because of the condition the individual defects cannot add up. This simplyfies the calculation a lot. Figure 4 illustrates the problems that occur if the flow defect is not strictly smaller than the unit square size.

We estimate the maximal number of squares of size $u$ that can be packed for a given flow state $f$ by

$$P_u(f) \coloneqq \sum_{l=0}^{n-u} \left\lfloor \frac{c_l}{u} \right\rfloor \cdot \left\lfloor \frac{n-l}{u} \right\rfloor .$$

The number of squares that have to be placed, in terms of unit squares of size $u$ for a given partition $p$ is

$$T_u(p) := \sum_{m=1}^{n} p_m \cdot \left\lfloor \frac{m}{u} \right\rfloor^2 .$$

Now, if $T_u(p)$ exceeds $P_u(f)$ for any $u \geq 2$, we have to backtrack. However, the calculation, even when utilizing an update, is quite expensive. In our current implementation only $u = 2$ is used. This also improves the calculation since we do not have to use division but a bit shift. Additonally taking higher unit square sizes into account seemed just to make things worse in terms of the overall computation time.

In the case that $D_2(f) = 2$, we still can apply the above rule, with one correction. In this case, there are exactly two levels with a defect of 1. Then we can make a overhang from the higher one of those two (and possibly shift all others, resulting in overhang on the lower levels), and add the correction term to $P_u(f)$. Thus, this rule becomes less restrictive and we can apply it more often. Using an update approach, this can be tested in constant time.

## Pruning Rule V: Symmetry breaking

This is the only time we use an heuristic approach. This idea comes from [5]. Since a valid placement is still valid if we flip top and bottom, we may assume that one prior picked square is placed in the upper half. Our tests showed the best results for the last of the third smallest square. This still exhibits a potential for improvement. However, this check can also be done in constant time.

## Checking the horizontal placement

After we have determined a valid flow, we are left with a sequence of squares. On each level we now have only a fraction of the squares which we have to place. Again, we start with the largest squares before possibly backtracing. By placing the squares, we generate a skyline. This is organized as a number of bars that grow.

The placement of each square lifts a part of skyline. A bar is the combination of all parts of the skyline that are on the same level and connected. If both neighbor bars are higher, we call this a valley. Figure 5 illustrates this.

Since we began to generate the flow from the bottom up, the squares are placed on the lowest level, too. Therefore, in each update, we are placing the squares always in a valley. In order to efficiently handle the updates, the bars are stored in a mixed data structure of a heap and a list. The heap is used to determine the next valley and the list preserves the neighbor relation between the bars. The access of the heap is at most logarithmic, and the access of the list can be done in constant time, since we only need the information of the left and right neighbor.

Due to the placement in valleys, a variation of the pruning rule from [5] can be used, namely the Valley Area Check. If the total area of unplaced squares from a higher level that are not larger than the width of the valley is smaller than the valley area, we cannot fill the valley with those squares and therefore have to backtrack.

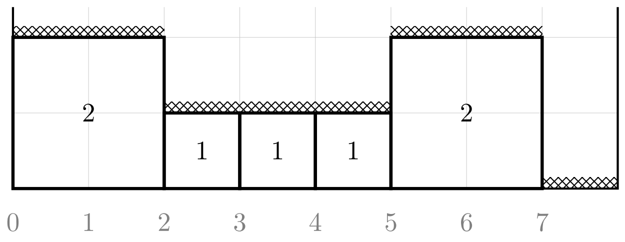

Figure 5: Example for a skyline containing four bars. The first starting at position 0, with a width of 2 and a height of 2, the second from 2 width a width of 3 and a height of 1, and so on. There are two valleys in this picture, one between positions 2 and 5 with a depth of 1 and one starting from position 7 with a depth of 2.

If this check succeeds, the partition is valid. If it does not, we return the highest level we reached back to the flow algorithm.

## 3 Preliminary checks

There are some partitions that are "obviously" invalid or valid, but do take a very long time to calculate or are so simple that we do not want to use the algorithm described above. There are some preliminary checks that we run once before starting the algorithm.

### S1: Two largest squares do not fit

If the sum of the sizes of the two largest squares exceeds $n$, this partition is invalid. We already did encounter one such example in the introduction: $(0, 1, 1, 3)$.

### S2: Number of second largest squares must fit in maximal configuration

If the sizes of the two largest squares are $k_1$ and $k_2$, and if $k_1 > k_2$ as well as $k_1 + k_2 = n$, the amount of squares of size $k_2$ must not be larger than $2\lfloor k_1/k_2 \rfloor + 1$. The largest square can only be on one border. The maximal configuration would be to pack this into one corner and pack the squares of size $k_2$ to the two free sides and one in the opposite corner. Figure 6 gives an example for such a maximal configuration.

### S3: Amount of virtual squares must not be to large

This check makes up a little bit for the amount of calculations necessary in Pruning Rule IV in our flow algorithm. There we retreated to only apply the rule for a unit square size of 2. Here, we try all sizes, since we only have to do it once and not billions of times. Using the same notation as we used there, we get simpler expressions for them, since we do not have to take the flow or the defect into account anymore (we only have the initial

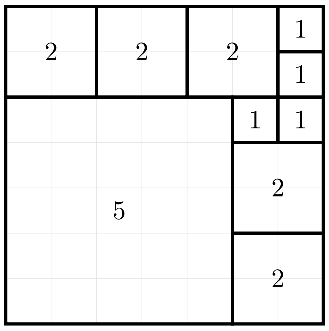

Figure 6: Maximal configuration for a $7 \times 7$ square and largest squares of size $k_1 = 5$, $k_2 = 2$.

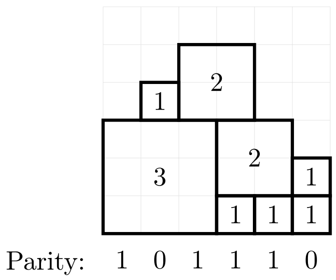

Figure 7: Example for the parity change by placing squares.

flow). That being said, we have for all $u = 2, 3, \ldots, n$

$$P_u((n, 0, \ldots, 0)) = \left\lfloor \frac{n}{u} \right\rfloor^2,$$

$$T_u(p) = \sum_{j=2}^{n-u} p_j \cdot \left\lfloor \frac{j}{u} \right\rfloor^2.$$

And again, if $T_u(p) > P_u((n, 0, \ldots, 0))$ for any such $u$, the partition is invalid.

## S4: Parity check

The last of the preliminary checks is a little more complicated. Imagine a strip of length $n$ filled with zeros. We say that placing a square of size $k$ changes the parity on a strip of length $k$ exactly $k$ times, i.e., every 0 becomes a 1 and vice versa. Figure 7 illustrates this.

Assume for the moment that $n$ is an even number. Then, after placing all squares, the parity has to be zero in every column. Since placing an odd square changes the parity, we have to reverse this effect. This can be done by dividing the set of odd squares into two sets with the same sum of sizes of the odd squares, i.e., we have to find to partitions

$p^{(1)}$ and $p^{(2)}$ with $p^{(1)} + p^{(2)} = p$ and

$$\sum_{k=0}^{\lfloor (n-1)/2 \rfloor} (2k+1) p^{(1)}_{2k+1} = \sum_{k=0}^{\lfloor (n-1)/2 \rfloor} (2k+1) p^{(2)}_{2k+1}.$$

In the case of odd $n$, the two sums should differ by exactly $n$. This problem is known as the bin packing problem. Since this is itself in NP, we confine ourselves to two different odd sizes only. If there is only one type of odd sized squares, everything is fine and we start the algorithm. Considering only two different sizes, we can use a naive algorithm which runs in linear time with respect to the total number of odd squares.

This approach proved rather effective in our tests. For example, the $22 \times 22$ partition $(0, \dots, 0, 2, 0, 7, 15, 14, 1)$ visited over $2 \cdot 10^{11}$ states while searching for a valid flow although no such flow exists. Using this check, we could rule out a lot of invalid partitions.

## 4 Finding all valid partitions

So far, we have only checked single partitions. However, we are interested in finding all valid partitions for a given number $n$. Therefore, we have to be able to generate all feasible partitions. However, this is quite easy.

Starting with a valid partition, we combine $k^2$ $1 \times 1$ squares to one $k \times k$ square, $k$ from $\{2, \dots, \lfloor \sqrt{a_1} \rfloor\}$, and check whether the new partition is valid. If this is the case, we repeat the process. After no more valid partitions can be generated, we are done. To do this more systematically, we first define a map $T_k$ by

$$\begin{aligned} &T_k : \mathcal{F}_n^{(k)} := \{(p_n, \dots, p_1) \in \mathcal{F}_n : p_1 \geq k^2\} \to \mathcal{F}_n, \\ &\quad (p_n, \dots, p_{k+1}, p_k, p_{k-1}, \dots, p_2, p_1) \mapsto (p_n, \dots, p_{k+1}, p_k + 1, p_{k-1}, \dots, p_2, p_1 - k^2). \end{aligned} \tag{2}$$

This map does exactly what we desribed above. For a given $k$, if there are enough unit squares, add a square of size $k \times k$ and take $k^2$ unit squares out. The overall sum stays the same and if we started with a feasible partition, the new partition remains feasible. Algorithm 4.1 describes how to generate the set $\mathcal{P}_n$. If we do not test whether $T_k p$ is valid in line 4, and thus do not terminate the generation of feasible partitions, the generated set will be the full $\mathcal{F}_n$.

**Algorithm 4.1** Generate the list of all valid partitions.

```
1: $\mathcal{P}_n \leftarrow \{(0, \dots, 0, n^2)\}$
2: for $k = n$ to 2 do
3:     for all $p \in \mathcal{P}_n$ do
4:         while $p \in \mathcal{F}_n^{(k)}$ and $T_k p$ is valid do
5:             $\mathcal{P}_n \leftarrow \mathcal{P}_n \cup \{T_k p\}$
6:             $p \leftarrow T_k p$
7:         end while
8:     end for
9: end for
```

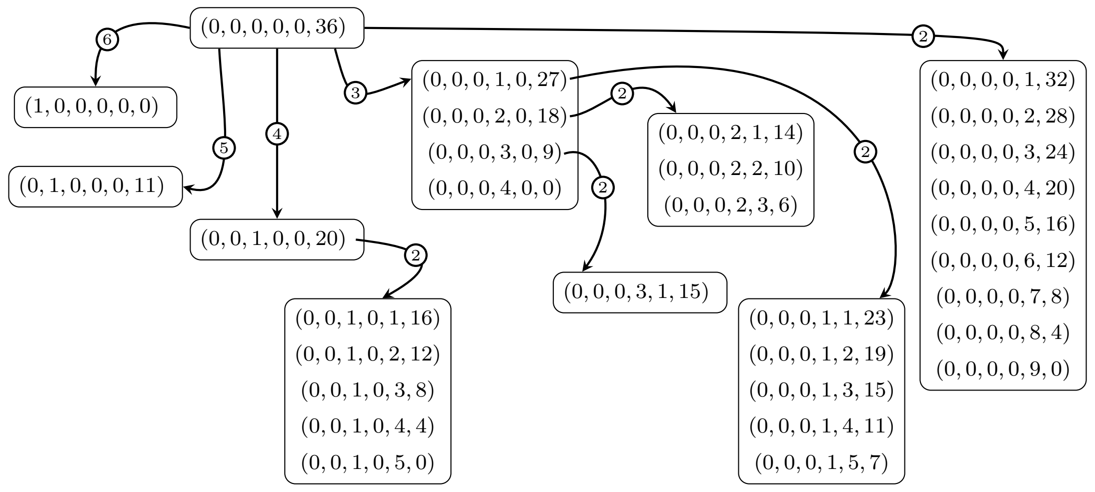

Figure 8: All valid partitions for $n = 6$, organized in a tree-like structure. The numbers on the arrows indicate the chain size.

It should be noted that, since we only work on $p_1$, we never generate a partition twice. Since for a fixed $k$ there is no interaction between the partitions, the loop in line 3 of the algorithm is very well suited for parallelization, where every processor runs the inner loop with a given unique partition. For fixed $k$ and $p$ we generate a chain of feasible partitions. Due to the solution richness of the problem, it is relatively easy to check whether a partition is valid in contrast to proving that it is invalid. The main reason for this is that for a valid partition there are usually many placements possible. One can get them for example by rotation or mirroring, just to name a few approaches. To prove that no placement can be found, we have to walk the whole tree, except for the parts that get excluded by the pruning rules. However, the moment we found an invalid partition in such a chain, we know that we found all valid partitions for this $k$ and $p$.

The chains provide one more benefit. Since the partitions in such a chain do not differ that much, we can use the search tree from the previous valid partition up to that point where the algorithm would have acted differently. We can just use the same stack again.

Another method for finding valid partitions without even running an explicit check are implicitly valid partitions. For example, if a partition with one $4 \times 4$ square is valid, also partitions where we replace this square by another $3 \times 3$ square or up to four $2 \times 2$ squares are valid, after filling up with the appropriate amount of unit squares. This is most easily realized if we save the valid partitions in a tree-like structure. Each node contains the partitions of a chain. Figure 8 shows the tree of all valid partitions for $n = 6$. Not only helps this structure with finding implicitly valid partitions, but it is also very space efficient, since we only have to store some meta information like the chain length and size and not any explicit partition. Also, we can start checking chains with some offset. There are more applications, which we will see later.

While inserting a partition in the tree, we also add all partitions with the above transformation. With even less effort, all partitions lying on a path are added. Consider for example the partition $(0,0,0,2,2,10)$. If we add this to the tree the partitions $(0,0,0,2,1,14)$, $(0,0,0,2,0,18)$, and $(0,0,0,1,0,27)$ are added just because they are on the path to the position where we want to insert it. By replacing the $3 \times 3$ squares one after the other by $2 \times 2$ squares, we also obtain $(0,0,0,1,3,15)$, $(0,0,0,1,2,19)$, $(0,0,0,1,1,23)$, $(0,0,0,0,4,20)$, $(0,0,0,0,3,24)$, and $(0,0,0,0,1,32)$, partly also because of the path.

Now that we already store the results in a tree, it can be accessed quite quickly. Therefore, we can employ the results of smaller instances during the generation of the partitions. For example, the $(n-1) \times (n-1)$ square in the partition $(0,1,0,\dots,0,2n-1)$ can be substituted by all its valid partitions. This alone accounts for almost half of the valid partitions of the $n \times n$ square. To incorporate this, we split the enumeration into two phases.

First, we only generate partitions with square sizes not smaller than $\lceil (n+1)/2 \rceil$. For these sizes we know that there can be only one. This can be seen as finding all possible borders around a large square. After this, we substitute the largest square by all corresponding partitions. If $n$ is an even number, we add the partition with four $n/2 \times n/2$ squares and apply the above procedure for each of those squares.

In the last phase, we generate all partitions with chain sizes from $\lfloor (n-1)/2 \rfloor$ to 2. The trees are then inserted for each square that is newly generated and thus matches the chain size.

The tree-like structure comes in handy when substituting the larger squares. Due to the method of implicitly valid partitions, it suffices to only add the last partition of each chain.

## 5 Parallelization

As mentioned above, the whole process of finding new partitons is very suitable for parallelization. Each base partition together with a given chain size can be run on its own processor. However, during the calculation it could happen that some partitions were checked that would be implicitly valid by another new partition. But since the valid partitions are the more simple problem, we can live with this.

A bigger problem are the invalid partitions. If we find an invalid partition that has, e.g., six $2 \times 2$ squares, we immediately know that the same partition with four of the $2 \times 2$ squares replaced by one $4 \times 4$ square is invalid. More general, all partitions that could be derived from this partition in much the same way as we generate the partitions, but with a larger unit square size, are invalid. In a single thread, checks for those partitions can be avoided quite easily. We just have to mark this in the result tree. However, with parallelization, it could and will happen that multiple partitions in this group are checked. At the moment, we give a signal if we found an invalid partition, and all threads check whether the current partition is a derivate of this invalid partition. If this is the case, we terminate the calculation. This leaves room for improvement, since most of the time it requires a large amount of computation power to prove that one partition is invalid. Even if we can cancel some partitions, it is still a little waste on resources when

| $n$ | Partitions | Running time [s] |
|---|---:|---:|
| 16 | 83667 | 2.7 |
| 17 | 170165 | 4.8 |
| 18 | 369698 | 31.1 |
| 19 | 743543 | 432.6 |
| 20 | 1566258 | 396.4 |
| 21 | 3154006 | 2801.9 |

Table 2: Running times for some larger $n$.

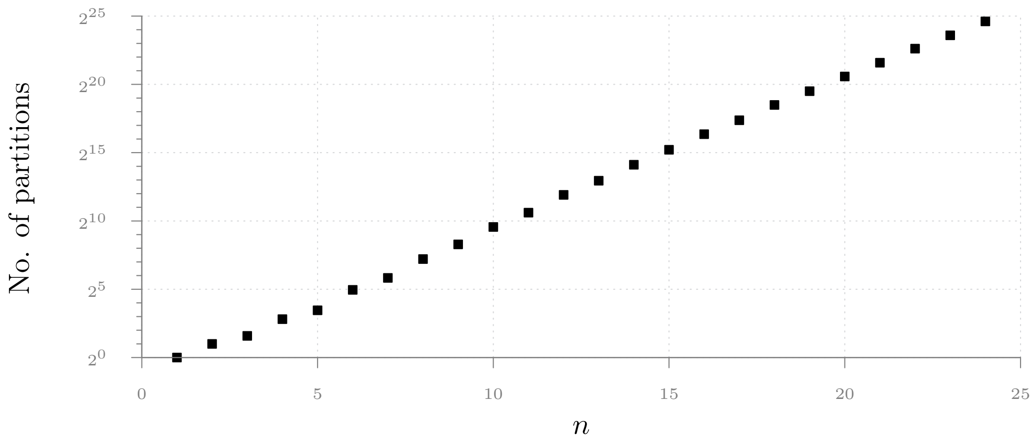

Figure 9: Number of valid partitions.

they all take a similar amount time.

Moreover, when running multiple threads we lose some determinism. This does not matter for the final results but makes it harder to reliable measure the number of, e.g., visited nodes or totally checked partitions.

## 6 Results and running times

Using the methods described above, we were able to push the limit of the known partition numbers a lot further. While these were previously known up to $n = 16$, we now gave the results up to $n = 24$ with reasonable effort. Table 2 lists some of the results together with the running time. The calculation was done one an Intel Core i5-3470 with four cores running at 3.2 GHz. In the experiments we ran four threads. The larger numbers were calculated on another machine, running at least 24 threads. For $n = 24$ this took about 18 hours.

With the new results, we now can anticipate a clear trend for the number of valid partitions. Giving asymptotic formulas for this can be part of future research.

## 7 Conclusion and outlook

In this paper we presented a new algorithm to check whether a perfect partition of a square can be achieved by a given set of squares. We also had a look on the bigger picture when we started not to consider the partitions separately anymore, but also relations between them. Even more, we incorporated results from smaller instances to quickly find a large number of valid partitions for the larger square.

There are still some open problems. First, we still use a rather naive algorithm for the bin packing in the preliminary check. We restricted ourselves to only two different sizes of odd sized squares. Taking more sizes into account requires a more sophisticated approach.

With each new $n$ we treat, there emerge more complicated partitions. The study of such partitions led in the past to the parity check. Here we noticed a clear pattern in the long running invalid partitions. Finding what such partitions have in common is significant for further lowering the computation time.

## Acknowledgments

I thank Christoph Helmberg and his team for the valuable feedback.

## References

[1] C. J. Bouwkamp. On the dissection of rectangles into squares. I. *Nederl. Akad. Wetensch., Proc.*, 49:1176–1188 = Indagationes Math. 8, 724–736 (1946), 1946.

[2] C. J. Bouwkamp. On the dissection of rectangles into squares. II, III. *Nederl. Akad. Wetensch., Proc.*, 50:58–71, 72–78 = Indagationes Math. 9, 43–56, 57–63 (1947), 1947.

[3] R. L. Brooks, C. A. B. Smith, A. H. Stone, and W. T. Tutte. The dissection of rectangles into squares. *Duke Math. J.*, 7:312–340, 1940.

[4] Adrianus Johannes Wilhelmus Duijvestijn. *Electronic computation of squared rectangles.* Thesis, Technische Wetenschap aan de Technische Hogeschool te Eindhoven, Eindhoven, 1962.

[5] Stefan Hougardy. A scale invariant exact algorithm for dense rectangle packing problems. `http://www.or.uni-bonn.de/~hougardy/paper/PerfectRectanglePacking.pdf`, 2011. Accessed: 2018-01-04.

[6] Joseph Y.-T. Leung, Tommy W. Tam, C. S. Wong, Gilbert H. Young, and Francis Y. L. Chin. Packing squares into a square. *J. Parallel Distrib. Comput.*, 10(3):271–275, 1990.

[7] The on-line encyclopedia of integer sequences, sequence A034395. `http://oeis.org/A034295`.