Freie Universität Berlin

Bachelorarbeit am Institut für Informatik der Freien Universität Berlin

Visibility-Constrained Square Packing on Discrete Grids

Kilian Chung
Matrikelnummer: 5566775
kiliac03@zedat.fu-berlin.de

Betreuer/in:	Prof. Dr. László Kozma
Eingereicht bei:	Prof. Dr. László Kozma
Zweitgutachter/in:	Dr. rer. nat. Max Willert

Berlin, August 20, 2025

Abstract

Packing problems are a classical topic in combinatorics and theoretical computer science with many practical applications. The goal of these problems is to arrange a set of objects within a bounded space under certain constraints, commonly optimizing a metric, such as maximizing the number of objects placed. This thesis explores a geometric packing problem on a two-dimensional grid, where non-overlapping, axis-aligned squares must be placed such that each square "sees" exactly K other squares along horizontal or vertical lines.

To find configurations that maximize the number of placed squares, we adapt the classical backtracking framework and introduce two algorithmic methods for exact searches. Additionally, we implement a simulated annealing approach to produce approximate solutions, thereby reducing search times for larger grid sizes where exhaustive search becomes computationally infeasible. We also discuss how problem-specific properties, such as symmetry of solution grids, can accelerate searches and find lower bounds on the maximal number of placeable squares.

Our experiments show how both backtracking algorithms compare in terms of computation time and configurations visited. We were able to reproduce and, in some cases, improve solutions of previous work. Overall, this thesis contributes to a better understanding of the structure and complexity of visibility-constrained packing problems.

Contents

1 Introduction

Packing problems arise from various real-world contexts, from efficient space usage in logistics to layout design for chip components. And although there are many variations of the packing problem, the central challenge often remains the same: how to optimally fit a set of objects into a constrained space.

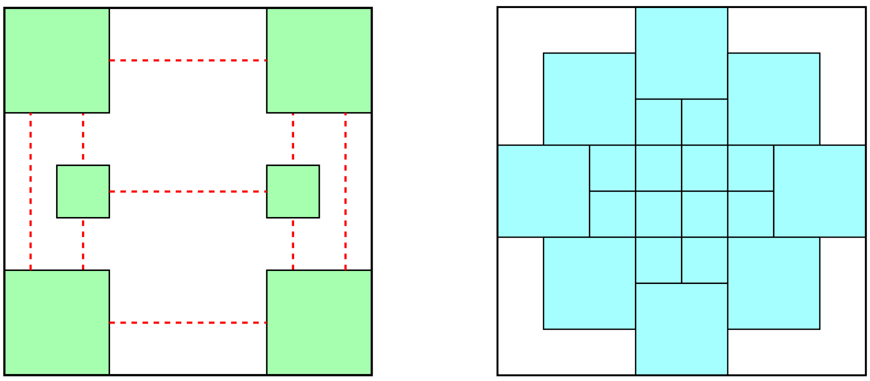
Beyond the practical applications, these kinds of problems are of particular interest in theoretical computer science as well due to their non-trivial nature. Many such problems are known to be NP-hard, and designing efficient approximation algorithms has become its own research area.

1.1 Problem Overview

In this thesis, we study a 2D geometric packing problem where we try to fit axis-aligned squares into a square grid. The side length for both the placed squares and the container must be integer, as well as the points of the squares' corners. Furthermore, placed squares may touch (edge-to-edge) but must not intersect with each other.

The property that makes the problem both non-trivial and interesting is the "visibility constraint". This constraint requires each square to "see" exactly K other squares. Squares can only see in straight, axis-parallel lines, and the visible connection between two squares may be blocked by another square in between. When two squares share an edge, they also see one another.

Fig. 1.1 shows two example grids that meet all requirements, with each square seeing exactly three (left) or four (right) other squares each.



(a) $K = 3$ (b) $K = 4$

Figure 1.1: Examples for valid configurations

Our goal is to compute the number $S_K(N)$, that is, the maximal number of squares placeable on an $N \times N$ grid, such that each square sees exactly K other squares. It turns out that the example shown in Fig. 1.1b is actually optimal in that sense for $N = 8$ and $K = 4$, hence $S_4(8) = 20$.

Despite its intuitive definition, this problem poses an interesting combinatorial challenge due to the immense search space and tight constraints. We suspect that deciding whether a partial configuration can be extended into a valid one is an NP-complete problem with respect to N, given the pair of parameters (N, K) and the initial partial configuration. The problem is clearly in NP with the extended, valid configuration serving as certificate: Since the number of squares is upper-bounded by N^2, the certificate can be represented in polynomial length with respect to N. For each square, the line of sight can be checked sequentially, counting the visible neighbors for each.

Although we suspect it to be NP-hard, we do not aim to provide a complexity analysis of the problem, but rather focus on practical implementations and their effectiveness for moderate problem sizes to find valid configurations.

1.2 Motivation

Despite a long history in research, packing problems still pose a combinatorial challenge, and finding efficient ways of solving such problems remains part of active research [7]. This particular problem is of interest not only because of its theoretical complexity and visually pleasing results, but also because it is fairly niche and not well-researched in the literature. Additionally, found solutions can be directly quantified by comparing them to known bounds. Furthermore, related problems have shown practical relevance in areas such as layout design for circuits and communication paths [5]. These aspects underline the value of researching the nature of this problem further.

1.3 Contributions of the Thesis

This thesis provides a case study for this specific optimization problem and explores techniques such as backtracking search and simulated annealing (SA). We introduce two algorithmic methods for computing optimal (exact) solutions (Section 4.2 and Section 4.3), prove their correctness, and compare their runtimes in experiments (Section 6.1). With an implementation of these algorithms, we were able to compute $S_K(N)$ and their solution grids for small values of N and K.

We could reproduce and thereby prove the tightness of known lower bounds on $S_3(5)$,$S_3(6)$,$S_3(7)$,$S_3(8)$,$S_4(8)$, and obtained even better, exact solutions for $S_5(11)$ and $S_5(12)$. Both $S_1(N)$ and $S_2(N)$ already have a known closed formula and a construction method to produce solution grids (see Section 6.2 and Appendix A.4). [13]

Moreover, we designed an SA framework in an effort to generate approximate solutions more efficiently than the exact searches and discuss its shortcomings and potential improvements (Section 4.4, Section 6.3). Finally, we explore problem-specific properties that may be harnessed to find lower bounds of $S_K(N)$ even faster (Section 6.5).

2 Fundamentals and Related Work

This exact problem has been proposed by Friedman as *Problem of the Month (September 2007)*. On his page, he summarized findings regarding the problem and encouraged readers to further investigate this problem. Section 6.2 compares our findings to the findings of previous work. Moreover, Friedman proposed two interesting variants:

1. Finding the *limiting density* on any finite or infinite grid, i.e., the ratio of occupied cells over all cells.
2. Allowing square placements on real-valued positions, not only integer grid positions.

Although these variants are interesting as well, this thesis focuses on the original problem formulation. [13]

2.1 Packing Problems

Packing problems are a class of optimization problems that involve arranging a set of items within a given container under specific constraints. These constraints typically require the items not to overlap, and a common objective is to maximize the utilization of available space. There are many types of packing problems, some more researched than others. While packing problems can also be defined for higher-dimensional settings, our focus is primarily on two-dimensional cases.

Classical 2D packing problems include *bin packing*, where the goal is to pack a set of items into as few fixed-sized containers (bins) as possible, and *strip packing*, where the container has a fixed width and the objective is to pack all items within the minimum height [11, 16]. Another well-known variant is the geometric *disk packing*, where the goal is to pack equal disks as densely as possible into a container, maximizing the disks' radius [2, 8]. In all of these variants, items must not overlap.

Many of these problems are computationally challenging due to the exponential growth of the search space. The respective decision problems—e.g., "Is it possible to pack this set of items into a container of height X?"—are often NP-complete, which implies that the corresponding optimization versions are unlikely to be solvable in polynomial time. As a result, heuristic or approximate methods are commonly employed to tackle larger problem instances. [11, 12, 16]

A key aspect of packing problems is the set of allowed transformations for the items. For most formulations studied in the literature, only translation is permitted, while rotation and scaling are disallowed. This constraint reflects many real-world applications, as noted by Lodi *et al.* [16], including the cutting of corrugated materials in industries such as woodworking, glass, and textiles, as well as newspaper paging layouts. Our problem, on the other hand, allows scaling.

The problem studied in this thesis sets itself apart from previously mentioned packing problems in several important aspects. Most notably, we do not have a fixed set of items we need to fit into the container. The objective of our problem is to maximize the number of squares we can fit, and because the squares are not fixed-sized, neither

the number nor the exact shape of our items is known in advance.
Another key difference is the visibility constraint. Whereas many traditional packing problems can be approached using local heuristics—considering only nearby items when evaluating a placement—our visibility constraint introduces global dependencies between squares. This makes standard greedy or local search-based methods inapplicable for this problem, calling for other approaches.

2.2 Backtracking Searches

A simple, yet powerful tool to exhaust a finite search space in a structured manner is the backtracking framework. It has been formalized numerous times, among others by Dijkstra [9] and Knuth [15].

Let us say, we seek all sequences $\langle x_1 x_2 \ldots x_n \rangle$ where some property $P_n(x_1, x_2, \ldots, x_n)$ holds. We also define so-called *cutoff* properties $P_l(x_1, \ldots, x_l)$ for $1 \leq l < n$, which determine whether the partial solution $\langle x_1 x_2 \ldots x_l \rangle$ can be extended into a valid solution. We require that

$$P_l(x_1, \ldots, x_l) \text{ is true whenever } P_{l+1}(x_1, \ldots, x_{l+1}) \text{ is true;} \tag{1}$$

$$P_l(x_1, \ldots, x_l) \text{ is easy to test, if } P_{l-1}(x_1, \ldots, x_{l-1}) \text{ holds.} \tag{2}$$

The key idea of backtracking is to incrementally build solutions by assigning values to the variables x_k for $k \to n$. Whenever P_l is false, we know that $\langle x_1 x_2 \ldots x_l \rangle$ cannot be extended into a valid solution. We abandon this candidate by backtracking, i.e. going to the last set variable (x_l) and trying a different value. If we run out of values to try, we backtrack even further by going back one more variable (to x_{l-1}) and trying a different value there, etc. Following this recipe, we are sure to visit all solutions $\langle x_1 x_2 \ldots x_n \rangle$. [15]

In his work, Knuth applied a basic backtracking algorithm to the n-queens problem, which may help to build intuition for the backtracking framework. The problem is defined as follows:

> How many ways are there to place n queen pieces on an $n \times n$ chess board, such that no queen could capture another in one move? In other words, no two queens are allowed on the same rank (row), file (column), or diagonal.

Clearly, each rank and file has to be occupied by exactly one queen. To avoid counting duplicate solutions (permutations), we say that the variable assignment $x_k \leftarrow c$ denotes the placement of a queen at row k and column c. Notice how no two queens can be on the same row due to the way we represent queen placements: x_k can only be one value at a time. The domain for each variable is $\{1, 2, \ldots, n\}$ and the cutoff property $P_l(x_1, \ldots, x_l)$ is the condition that no two queens are on the same column or diagonal: $x_j \neq x_k$ and $|x_k - x_j| \neq k - j$ for $1 \leq j < k \leq l$. With these definitions done, the backtracking algorithm can now be applied.

The behavior of the search is best visualized by its *search tree*. In this tree, each node represents a board state, the root being an empty board. Edges from parent to child denote the one queen placement required to transform the parent's board into the child's board. The backtracking search traverses these nodes in depth-first-search order. For example, the search tree of the 4-queen problem (Fig. 2.1) has 17 nodes:

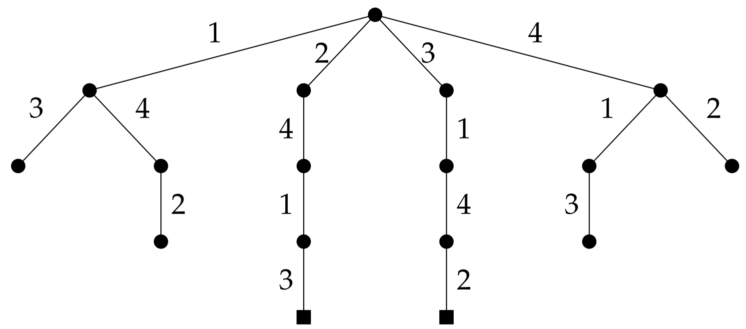


Figure 2.1: 4-queen search tree

As one can see, only two strands reach the desired depth of 4. Fig. 2.2 shows these two boards.

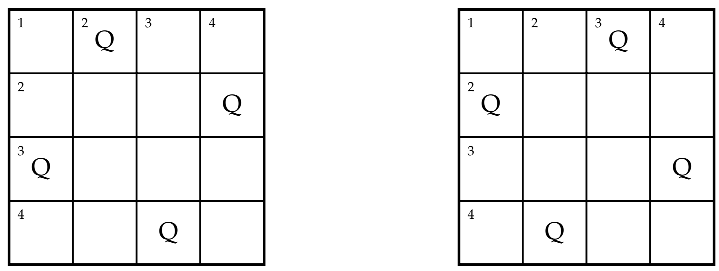


Figure 2.2: 4-queen solutions

Backtracking is a fundamental component of the exact algorithms described in Sections 4.2 and 4.3.

2.3 Meta-Heuristic Approach

While backtracking searches are exact, heuristic approaches trade exactness for efficiency. Whereas they may not always find an optimal solution, they often provide good enough solutions much faster, especially for large problem spaces.

Simulated Annealing (SA) is a heuristic method first introduced by Kirkpatrick *et al.* to give an approximation for the traveling salesman problem [14]. Since then, it has seen applications in a variety of application areas, as highlighted by the numerous examples compiled by Collins *et al.* [6].

SA works by emulating the physical process of a solid cooling down until eventually reaching a "frozen" structure. This frozen structure is a minimum energy configuration, which can be described as the optimum of some (energy) cost function. To be specific, SA operates with these basic elements:

1. A finite set S, all possible system states/configurations.

2. A cost function f defined on S.

3. For each $i \in S$, a set $\mathcal{N}(i) \subset S - \{i\}$, called the neighbor set of i.

4. A cooling schedule function T, where $T(t)$ is called the *temperature* at time t. This function is nonincreasing.

5. An initial state $x_0 \in S$.

We progress from the state at time t to the next one ($x_t \to x_{t+1}$) by selecting a random neighbor $j \in \mathcal{N}(x_t)$. We then determine x_{t+1} with following formula:

$$
\begin{aligned}
&\text{If } f(j) \leq f(x_t), \text{ then } x_{t+1} = j \\
&\text{If } f(j) > f(x_t), \text{ then} \\
&\qquad x_{t+1} = j \text{ with probability } \exp[-(f(j) - f(x_t))/T(t)] \\
&\qquad x_{t+1} = x_t \text{ otherwise}
\end{aligned}
$$

Progressing with slightly worse solutions—with mentioned probability $\exp[-(f(j) - f(x_t))/T(t)]$—allows SA to escape local cost minima in the solution space. We define a threshold for the temperature as stop condition for the algorithm. In Section 4.4, we go over the details of applying SA to our problem and how we define the cost and neighborhood functions. [3]

We chose this meta-heuristic approach because an in-depth analysis by Bertsimas and Tsitsiklis concluded that SA is simple to implement, generally applicable to a wide range of optimization problems, and is proven to produce good solutions in practice, even if the underlying structure of the problem is not well understood. [3]

Dowsland has already successfully applied SA to a specific form of packing problem [10]. The problem they describe consists of packing axis-aligned rectangles at integer positions into a rectangular container. While searching, they allow the elements to overlap and treat this overlap as part of the cost function. They achieved impressive results with this approach, but hinted that tabu search may prove to be a better type of approach for these kinds of problems. The problem they studied differs in two key points from ours:

1. The number and shapes of elements are fixed, in contrast to our problem.

2. The heavy visibility constraint in our problem.

Aarts and Lenstra introduced the notion of a *multilevel* or *combined* approach. In Section 4.4, we adopt this idea and use an exact search as cost function for configurations that SA operates on. [1]

3 Terminology and Notation

Besides the problem definition in Section 1.1, we introduce the following supplementary terminology.

Intuitively, we say two squares A and B *see each other* in the grid if there exists a straight horizontal or vertical line intersecting both A and B while not passing through any other square. Each square we may place can be described by a tuple (x, y), denoting the position of its top-left corner, and a side length l. All values x, y, l must be integers and $0 \leq x, y < N$ and $1 \leq l \leq N$.

We define $S_K(N)$ to be the maximal number of squares that can be placed on a square grid container of side length N, s.t. each square sees exactly K others. The core objective of this thesis is to study approaches to exactly determine or approximate $S_K(N)$ efficiently.

The following additional terms will be used throughout this thesis:

- A square is a *visible neighbor* to another square if they see each other.
- A square is *satisfied* if it sees exactly K other squares.
- The *visibility constraint* holding for a square is equivalent to it being satisfied.
- We define a *placement* to be the tuple (x, y, l) where x, y are the 0-based coordinates of a square, and l is its side length. It is sometimes used interchangeably with the term *square*.
- We call a *construction* a sequence of unique placements, denoting the process of placing squares iteratively.
- The result of a construction will be a *configuration*, that is, the grid with the placed squares inside, without placement order information.
- Furthermore, we call two constructions *equivalent* if they are permutations of one another, resulting in the same configuration.
- We call a *configuration valid* if the visibility constraint holds for all placed squares.
- A *construction* is *valid* if its resulting configuration is valid.
- Solutions to $S_K(N)$ are configurations with $S_K(N)$ squares on an $N \times N$-grid.

We use subscripts $\mathcal{C}_d$ to denote the dth placement of a construction $\mathcal{C}$.

To maintain conformity within this thesis, visualized grid coordinates go left-to-right and top-to-bottom. Fig. 3.1 shows the (x, y)-coordinates on a grid of side length 4, alongside the highlighted square $(x = 1, y = 2, l = 1)$. We may also use the notion of the *positional index* of a placement. This index is equivalent to $y \cdot N + x$.

(0,0)	(1,0)	(2,0)	(3,0)
(0,1)	(1,1)	(2,1)	(3,1)
(0,2)	(1,2)	(2,2)	(3,2)
(0,3)	(1,3)	(2,3)	(3,3)

Figure 3.1: $(1, 2, 1)$ on a 4×4 grid

4 Algorithmic Methodology

In this thesis, we examine two different *exact* algorithms for finding $S_K(N)$:

1. A simple brute force backtracking search (Section 4.2);

2. A more sophisticated backtracking search, which cuts off the search tree branch if newly placed squares block the visual connection between two other squares (Section 4.3).

Furthermore, a meta-heuristic *approximate* search was explored:

3. A Simulated Annealing approach that mutates a starting configuration iteratively in order to obtain a partial configuration of a valid solution. It uses exact searches to evaluate how "good" the partial configurations are (Section 4.4).

4.1 Preliminaries for Backtracking Searches

The behavior of backtracking searches can be nicely visualized by a tree, the so-called *search tree* [15]. In this tree, each node represents a state of the system, i.e., the grid, while the edges represent the operation required to transform a parent node into its child. In our case, this operation is the placement of a single square. The backtracking searches traverse this tree in depth-first-search (DFS) order, starting at the root, which is an empty grid. These trees will become very large for greater values of N and take a lot of time to traverse. Cutting off as many branches as possible on each level is thus essential for an efficient search. Fig. 4.1 shows an example search tree, traversed by our second exact search algorithm (`BT_2`). It finds the optimal solution at the 4th node.

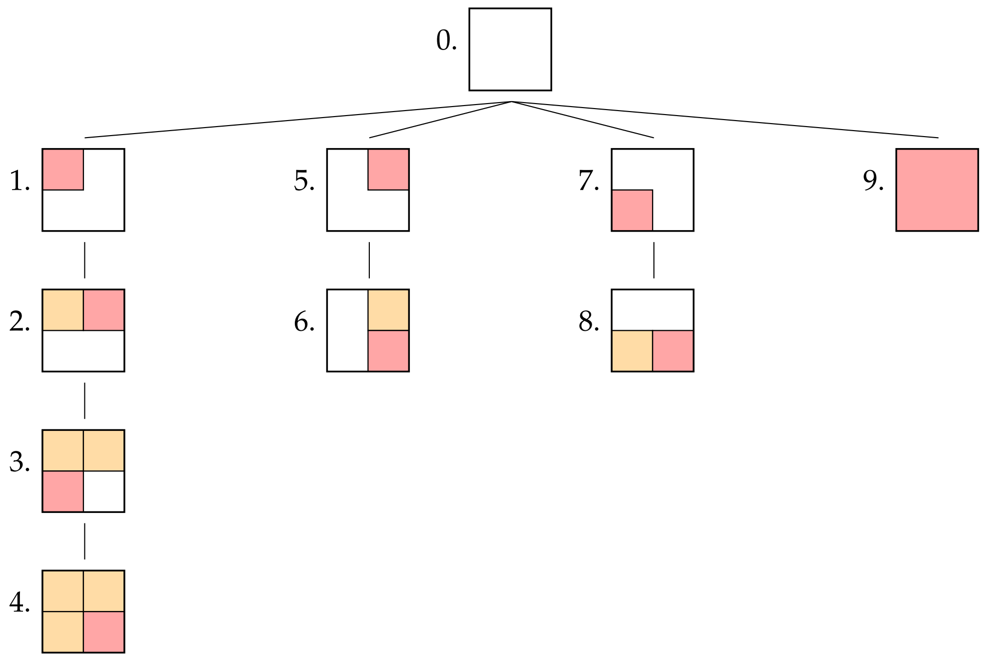


Figure 4.1: Search tree traversed by `BT_2` for $N = K = 2$

A recursive approach fits the depth-first traversal of the tree quite naturally, as backtracking simply involves removing the placed square and returning from the recursion to the previous level.

Before exploring the search algorithms, we first have to establish the interface for them to use. As a basic framework, we define a `grid` class, which holds the placed squares in a stack-like (last-in-first-out) structure. This kind of structure works hand in hand with a backtracking search. The `grid` structure has the following properties:

- `.can_place(x, y, l)`: Returns whether a square at position (x, y) and side length l fits on the grid without intersecting with another square.
- `.valid()`: Returns whether all placed squares satisfy the visibility constraint.
- `.push(x, y, l)`: Places a square in the grid at (x, y) and side length l.
- `.pop()`: Removes the last placed square from the grid.
- `.placed`: The number of placed squares.
- `.sq[]`: The stack of placed squares.

A more in-depth exploration of this structure is presented in Section 5.2.

The search functions take in a grid instance with side length N, which is initialized to be empty for now, and return $S_K(N)$.

4.2 BT_1 Algorithm

For our first approach, we use a simple recursive backtracking search. We generate all possible configurations and check after each placement whether the visibility constraint holds. We then return the maximal number of placements we made while not hurting the visibility constraint.

For that, we first define a recursive function `BT_1*`. In each call to `BT_1*`, we place a square, call itself recursively, then unplace the square we placed earlier. We do that for all available square positions and for all possible square side lengths $1 \leq l \leq N$.

The function ought to keep track of and also return the maximal number of squares (`max_squares`) placeable in the given grid (`grid`) such that the visibility constraint holds. Listing 4.1 shows a pseudo-implementation.

Listing 4.1: Backtracking Search Pseudo Implementation Version 1*

```
procedure BT_1*(grid):
    max_squares := 0
    if grid.valid():
        max_squares = grid.placed

    for l := 1,..,N:
        for x := 0,..,N-1:
            for y := 0,..,N-1:
                if not grid.can_place(x, y, l): continue

                grid.push(x, y, l)
                max_squares = max(max_squares, BT_1*(grid))
                grid.pop()

    return max_squares
```

This procedure employs the backtracking strategy described by Knuth (*basic backtrack*), applied to our problem [15]. Here, the cutoff property is given by the `grid.can_place` check.

It examines all possible positions and side lengths for each placed square and is certain to simulate all possible constructions, thus reaching all configurations and finding $S_K(N)$ correctly. This shows that this search algorithm is *exact* and always yields the optimal solution for $S_K(N)$.

The downside to this algorithm is its poor runtime if implemented exactly like this. The search generates the same configuration for each permutation of placed squares. We can easily see that by the diagrams in Fig. 4.2, showing different constructions, squares numbered by the order they were placed in. For example, when searching for $S_2(2)$, all of these and many more equivalent constructions will be examined.

To prevent the search from generating these equivalent constructions, an ordering must be employed. Therefore, we define the following constraint for the search: A square placement p_s is only allowed if its *rank* is strictly greater than the *rank* of the last square's placement p_{s-1}. The exact implementation of the *rank function* does not

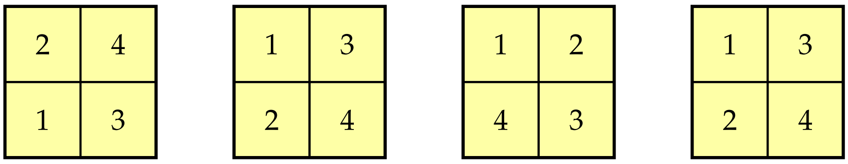


Figure 4.2: Visualization of equivalent constructions

matter per se as long as it maps a unique value to each placement. In other words, we define a total order on all grid placements. This simple constraint eliminates all duplicate equivalent constructions in the search tree.

Lemma 4.1. *`BT_1*` does not revisit the same construction: Each possible construction is visited at most once by `BT_1*`. This also implies that equivalent configurations in the search tree must stem from different constructions.*

Proof. Each possible square placement is considered only once per `BT_1*` call. Before the algorithm revisits the same depth d, the construction up to this point must have changed. We can argue inductively that `BT_1*` at depth $(d-1)$, popps and places a different square than before, resulting in a different construction. Therefore, the construction up to depth $(d-1)$ must be different before choosing the same square placement again at depth d. □

Lemma 4.2. *The rank constraint eliminates equivalent constructions from the search tree.*

Proof. Let us say we reached two equal configurations by two different constructions, $\mathcal{A}$ and $\mathcal{B}$. Since both $\mathcal{A}$ and $\mathcal{B}$ generate the same configuration, the placements contained in $\mathcal{A}$ must be exactly the same as the ones in $\mathcal{B}$. Furthermore, let r be the rank function and $R_a = r(\mathcal{A}_1), \ldots, r(\mathcal{A}_d)$ and $R_b = r(\mathcal{B}_1), \ldots, r(\mathcal{B}_d)$ denote the sequence of ranks of both constructions, respectively. Because of the rank constraint, we know $(R_a)_k < (R_a)_n$ for $1 \leq k < n \leq d$. Analogously, we know this to be true for R_b.

We also know from Lemma 4.1 that $(R_a)_k \neq (R_b)_k$ for at least one $1 \leq k \leq d$. WLOG let k be the first value where R_a and R_b differ and let $(R_a)_k < (R_b)_k$. Since a construction cannot contain a placement twice, we know $(R_a)_k \notin \{(R_a)_1, \ldots, (R_a)_{k-1}\}$ and thus also $(R_a)_k \notin \{(R_b)_1, \ldots, (R_b)_{k-1}\}$. But $(R_a)_k$ must be contained within $R_b \implies (R_a)_k \in \{(R_b)_{k+1}, \ldots (R_b)_d\}$. This would mean $(R_b)_k > (R_a)_k = (R_b)_{k+n}$ for a certain $1 \leq n \leq (d-k)$, which contradicts our rank constraint on $\mathcal{B}$.

This contradiction implies that it is impossible to reach multiple equivalent constructions with the rank constraint in place. □

Lemma 4.3. *The rank function may ignore the side length l of a given placement.*

Proof. It is sufficient for the rank function to only consider the position of a placement. A configuration has an injective mapping from square positions to their square's side lengths, because two squares cannot be placed at the same position. Two equivalent constructions will thus always have the same position-to-side-length-mapping.

In other words, the defining difference between equivalent constructions lies in the positional values of their squares. □

A simple rank function could look like:

```
procedure rank(x, y):
    return (y * N) + x
```

When indexing the grid positions left-to-right, top-to-bottom, this rank function reduces to the index of the position. We can use this in our BT algorithm to make the rank function implicit by only considering position indices greater than the one from the last recursion level. To retrieve the (x, y) coordinates, we can use $(idx \mod N)$ and $(\lfloor \frac{idx}{N} \rfloor)$, respectively. For the initial call to this recursive procedure, we declare a wrapper function. Putting everything together, we get the final version of the first algorithm (Listing 4.2):

Listing 4.2: Backtracking Search Pseudo Implementation Version 1

```
1  procedure BT_1_rec(grid, last_index):
2      max_squares := 0
3      if grid.satisfied:
4          max_squares = grid.placed
5
6      for l := 1,..,N:
7          for pos := last_index+1,..,(N*N)-1:
8              if not grid.can_place(pos % N, pos / N, l):
9                  continue
10
11             grid.push(pos % N, pos / N, l)
12             max_squares = max(max_squares, BT_1_rec(grid, pos))
13             grid.pop()
14
15     return max_squares
16
17 procedure BT_1(grid):
18     return BT_1_rec(grid, -1)
```

Proposition 4.4. *BT_1 visits all configurations.*

Proof. We start by declaring that each configuration has a unique *canonical* construction. In this construction, the squares are placed in an order that complies with the provided rank function r. For this *canonical* construction $\mathcal{C}$ of size d, the following condition must hold: $r(\mathcal{C}_1) < \ldots < r(\mathcal{C}_d)$.

Let us now look at `BT_1_rec` right after placing $\mathcal{C}_{d-1}$. We will call `BT_1_rec` recursively (Listing 4.2, line 12). This next call will then iteratively try placing the next square in all allowed positions that comply with r, inevitably also placing $\mathcal{C}_d$. And since also the first call to `BT_1_rec` inevitably places $\mathcal{C}_1$ at some point, we can argue per induction that the canonical construction $\mathcal{C}$ will be visited by this procedure. □

Corollary 4.5. *BT_1 is exact and always finds the optimal solution $S_K(N)$.*

4.3 BT_2 Algorithm

The second algorithm, too, is a backtracking search. On a high level, this algorithm satisfies one placed square, call it *root*, at a time by placing other squares in its line of sight until *root* sees exactly K other squares. We then forbid newly placed squares to intersect with the line of sight of *root*. This constraint allows us to reduce the number of nodes searched and cut down big parts of the search tree. We will discuss the intuition behind this approach first, before moving to the details of its realization. The general idea described in Section 4.3.1 is applicable to a range of optimization problems for which (1) there are constraints on elements that may be satisfied iteratively and (2) satisfying the constraint for some element may introduce new elements.

`BT_2` heavily relies on the use of bitboards. Modeling a search problem to use bitboards is a general method to improve performance. Segundo *et al.* define bitboard operations, also used in our approach. [17]

4.3.1 Intuition

For this algorithm, we define two intertwined procedures, namely `BT_2_inner` and `BT_2_outer`. The task of `BT_2_inner` is to satisfy one square already placed. To be precise, it loops over all possibilities of satisfying *root* and calls itself recursively. At this point, the need for a data structure arises that keeps track of already satisfied squares and chooses which one to satisfy next. Conveniently, there exists a neat and implicit approach to this, adhering to the first-in-first-out principle and eliminating the need for an explicit data structure. It works by keeping track of the depth d, i.e., the number of times the procedure was called. The *root* inside `BT_2_inner` is always chosen to be the dth square on the grid's stack.

Fig. 4.3 is a visualization of the grid's square stack over time, building an intuition for this approach. Here, each cell represents a square placement on the grid.

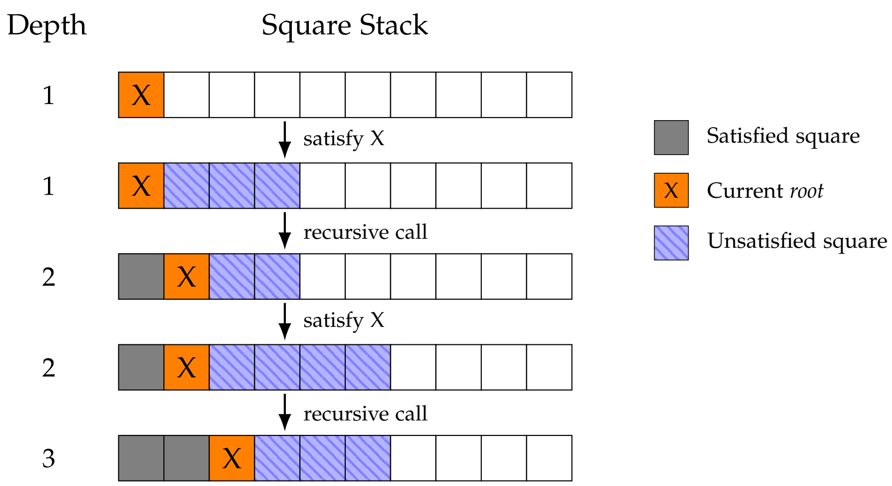


Figure 4.3: Visualization of the square stack over time

Picking the *root* this way works hand-in-hand with backtracking. Whenever we are at a certain depth and want to backtrack, we need to remove all newly placed squares and return. Since they were the last placed squares, they must be on top of the stack, thus easily poppable. Fig. 4.4 shows the grid's square stack while backtracking.

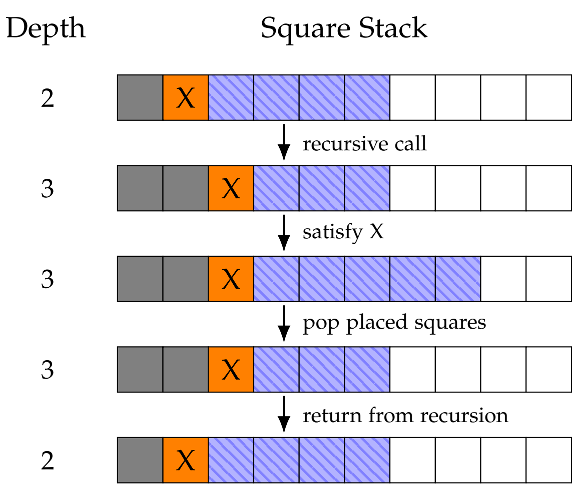


Figure 4.4: Square stack while backtracking

The last notable case is when encountering a valid configuration. We know we hit a valid configuration if there are no unsatisfied squares left. In other words, all placed squares must have been satisfied. This is the case when the current depth d is one greater than the number of placed squares. It is because of this revelation that the algorithm does not require the `grid.satisfied` property to detect valid configurations. Fig. 4.5 shows the process of finding a valid configuration. We reach depth 10, whilst only having placed 9 squares. Notice how satisfying X does not always involve placing new squares. In this case, X is already satisfied by previously placed squares.

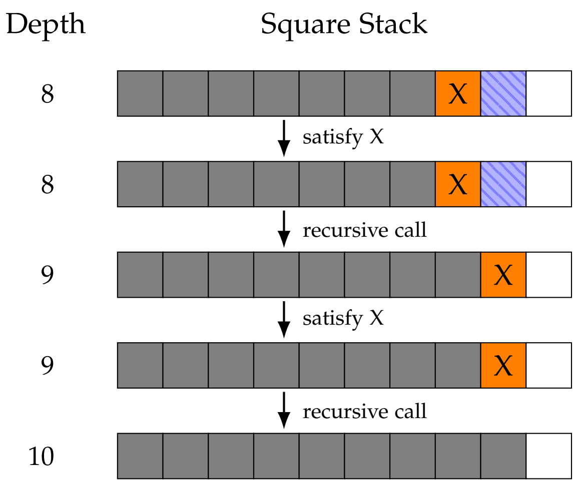


Figure 4.5: Square stack when reaching a valid configuration

The `BT_2_inner` procedure is an integral part of our `BT_2` algorithm, but is not sufficient to find $S_K(N)$ on its own. This becomes apparent when starting with any arbitrary square and tasking it to find $S_1(3)$. For any starting square, satisfying it means placing a second square, such that they see each other. This results in the second square also being satisfied and thus forming a valid configuration (Fig. 4.6). `BT_2_inner` cannot search beyond a valid configuration, therefore never finding $S_1(3) = 4$ shown in Fig. 4.7.

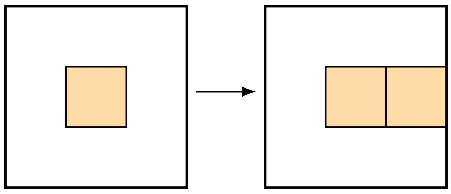


Figure 4.6: `BT_2_inner` not finding $S_1(3)$

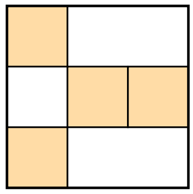


Figure 4.7: $S_1(3) = 4$

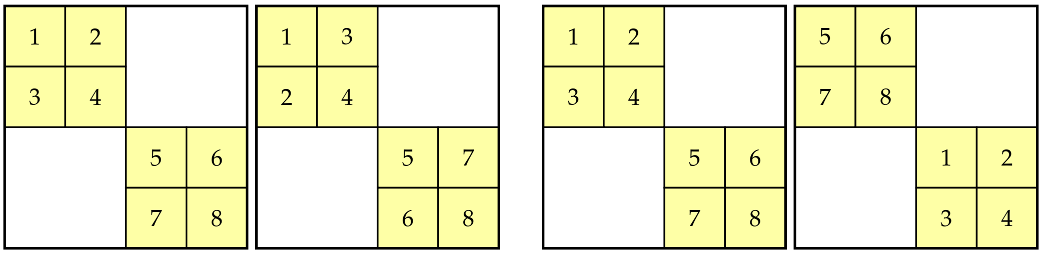
The root of this problem lies within the nature of the solution's *visibility graph*. In the *visibility graph* of a configuration, the vertices represent the square placements. An edge between two vertices v_a and v_b exists if and only if their corresponding squares a and b see each other. `BT_2_inner` only places new squares to satisfy already placed squares, which implies that all reached configurations will have a visibility graph consisting of a single, connected component. One can quickly verify that the solution for $S_1(3)$ shown in Fig. 4.7 has a disconnected visibility graph, consisting of two disjoint components. We call *components of a configuration/construction* the components of its corresponding visibility graph.

To address this issue and also find configurations with disconnected visibility graphs, we introduce the `BT_2_outer` procedure. This procedure takes a valid configuration, places a new square at a position that no other square sees, and calls `BT_2_inner` again. In other words, `BT_2_outer` starts a new component, whilst `BT_2_inner` grows it until it is satisfied. We call the squares placed by `BT_2_outer` *initial squares* since they start a new component within the visibility graph. Whenever `BT_2_inner` reaches a valid configuration, i.e., no more squares to satisfy, it calls `BT_2_outer`. This also takes care of the first square we have to place: For an empty grid, the configuration is valid, causing `BT_2_inner` to call `BT_2_outer`, which then places the first square. The wrapper procedure `BT_2` simply calls `BT_2_inner` with an initial depth of 1. Implementing `BT_2_outer` s.t. it loops over all possible square placements ensures that all valid configurations are reached. See Appendix A.1 for how this approach may find a solution for $S_1(3)$.

4.3.2 Detailed Analysis

Opposite to the first search algorithm, this one does not aim to reach every configuration. Instead, we claim that it reaches every valid configuration. To reiterate, `BT_2_outer` loops over all square placements that do not see any squares placed prior, and calls `BT_2_inner` for each placement. `BT_2_inner` picks a *root* square based on the depth d, then loops over all sequences of placements that satisfy *root*, s.t. all newly placed squares see the *root* and do not see any previously satisfied square, i.e. previous *root*s. `BT_2` is analogous to an initial call to `BT_2_inner` with a depth of 1.

`BT_2` eventually reaches every valid configuration, which we will prove at the end of this section. But like with `BT_1`, we have the problem of reaching the same configuration with different equivalent construction permutations. Fig. 4.8 shows the two kinds of construction permutations that produce the same valid configurations. Like in Fig. 4.2, the numbers inside the squares represent the order in which they were placed. These two kinds of permutations stem from the very nature of the twofold approach using `BT_2_inner` and `BT_2_outer`. First, inner permutations like the ones shown in Fig. 4.8a stem from `BT_2_inner` visiting the same configurations twice by satisfying the initial squares (1 and 5) the same way in two different orders. Outer permutations, shown in Fig. 4.8b, arise from `BT_2_outer` choosing the same initial squares in a different order.



(a) Inner permutations (b) Outer permutations

Figure 4.8: `BT_2` equivalent constructions for $S_2(4)$

To prune these equivalent constructions from the search tree, a similar approach as for `BT_1` may be employed. We define a *canonical* construction for every valid configuration and only search these constructions. Like before, we can enforce only visiting canonical constructions by imposing the *rank constraint*: A square placement is only allowed if its *rank* is strictly greater than the previous placement. Since we have two types of permutations on different levels, inner and outer, we may employ different, decoupled rank functions for each. The rank function for inner permutations takes square placements within a single call to `BT_2_inner` as inputs. Meanwhile, the outer rank function operates on disjoint components of configurations, on the level of `BT_2_outer`. They ought to map a unique value to each input. We furthermore require the initial square for each component to be unique. This may be enforced by keeping track of the initial square's position of the current component. Then, while satisfying the component, no square may be placed that, e.g., has a lower positional index.

Lemma 4.6. *The rank constraint realized with both the inner and outer rank function prevents the search from reaching equivalent constructions.*

Proof. Let r_{inner}, r_{outer} denote the inner or outer rank function, respectively, and let $\mathcal{A}$, $\mathcal{B}$ be two different but equivalent constructions of size d visited by `BT_2`. WLOG, let the kth placement be the first placement where $\mathcal{A}$ and $\mathcal{B}$ differ. The following cases may occur:

1. $\mathcal{A}_k$ and $\mathcal{B}_k$ are in the same component and initial squares.
2. $\mathcal{A}_k$ and $\mathcal{B}_k$ are in the same component, different "parent" placements.
3. $\mathcal{A}_k$ and $\mathcal{B}_k$ are in the same component, same "parent" placements.

4. They are in different components.

In any case, following reasoning holds:

$$\begin{aligned}
&\mathcal{A}_k \notin \{\mathcal{A}_1, \ldots, \mathcal{A}_{k-1}\} \\
\implies &\mathcal{A}_k \notin \{\mathcal{B}_1, \ldots, \mathcal{B}_{k-1}\} \\
\implies &\mathcal{A}_k \in \{\mathcal{B}_{k+1}, \ldots, \mathcal{B}_d\}
\end{aligned}$$

Analogously, we can argue that $\mathcal{B}_k \in \{\mathcal{A}_{k+1}, \ldots, \mathcal{A}_d\}$.

Claim 1. $\mathcal{A}_k \in \{\mathcal{B}_{k+1}, \ldots, \mathcal{B}_d\}, \mathcal{B}_k \in \{\mathcal{A}_{k+1}, \ldots, \mathcal{A}_d\}$

In case 1, both $\mathcal{A}_k$ and $\mathcal{B}_k$ are in the same component c, and WLOG $\mathcal{A}_k$ is an initial square, i.e., the first square of its component. Following this, $\mathcal{B}_k$ must be initial square for the same component:

$$\begin{aligned}
&\mathcal{A}_k \text{ is initial square to component } c \\
\implies &\{\mathcal{A}_1, \ldots, \mathcal{A}_{k-1}\} \text{ does not contain placements from } c \\
\implies &\{\mathcal{B}_1, \ldots, \mathcal{B}_{k-1}\} \text{ does not contain placements from } c \\
\implies &\mathcal{B}_k \text{ is initial square to component } c
\end{aligned}$$

But as we already stated, the initial square to a component is unique; hence, this case cannot occur.

In case 2, both $\mathcal{A}_k$ and $\mathcal{B}_k$ must have been placed to satisfy another already placed square. Let us call these "parent" placements $\mathcal{A}_x$ and $\mathcal{B}_y$, respectively, and WLOG let $x < y$. `BT_2` satisfies one square at a time and never places new squares in their line of sight further down the search tree. In $\mathcal{B}$, square $\mathcal{B}_y$ being satisfied by $\mathcal{B}_k$ implies that $\mathcal{B}_x$ is already satisfied and its line of sight will not be altered anymore. This stands in direct contradiction to the fact that placement $\mathcal{A}_k$ in $\{\mathcal{B}_{k+1}, \ldots, \mathcal{B}_d\}$ (Claim 1) will be in line of sight of $\mathcal{B}_x$.

In case 3, the placements $\mathcal{A}_k$ and $\mathcal{B}_k$ are both in line of sight with their common parent placement. Claim 1 leads us to $r_{inner}(\mathcal{A}_k) < r_{inner}(\mathcal{B}_k)$ and $r_{inner}(\mathcal{B}_k) < r_{inner}(\mathcal{A}_k)$ since both constructions have to obey the inner rank constraint. This leaves us with a contradiction, rendering case 3 impossible.

For case 4, $\mathcal{A}_k$ and $\mathcal{B}_k$ are in different components, whilst the $k-1$ placements before them are exactly the same. Since $\mathcal{A}$ and $\mathcal{B}$ are equivalent, they must be comprised of the same components. Using `BT_2`, components are consecutive subsequences of the construction sequence. This is because the search only grows the "current" component (`BT_2_inner`) or starts a new one (`BT_2_outer`); It never revisits already satisfied components. Let us denote $comp(p)$ the function that returns the entire component of a placement p as an abstract object. Claim 1 implies that $comp(\mathcal{B}_k)$ is entirely contained within $\{\mathcal{A}_{k+1}, \ldots, \mathcal{A}_d\}$ and vice versa. Because $\mathcal{A}$ and $\mathcal{B}$ comply with the outer rank constraint, we thus know $r_{outer}(comp(\mathcal{A}_k)) < r_{outer}(comp(\mathcal{B}_k))$, but also $r_{outer}(comp(\mathcal{B}_k)) < r_{outer}(comp(\mathcal{A}_k))$. Again, this results in a contradiction, also rendering this case impossible.

Since all the cases are impossible, the initial assumption that there exist two different but equivalent constructions reached by `BT_2`, must be false. □

The outer rank function may be implemented to simply return the positional index of its unique initial square placement, since no two disjoint components can coexist with the same initial square.

Proposition 4.7. *BT_2 visits all valid configurations.*

Proof. In this proof, we only concern ourselves with the last placed component. Since a valid configuration without its last component is still a valid configuration, we can argue by induction that it will also be reached.

Let r_{inner} and r_{outer} denote the inner and outer rank function, respectively. Each valid configuration has one unique *canonical construction*, that is, the construction where the components appear in an order according to r_{outer} and the placements are ordered in a way that complies with r_{inner} within each component.

When arguing that canonical construction $\mathcal{C}$ will be reached, we take a look at $\mathcal{C}_k$, which resides in the component of greatest r_{outer} value, i.e., the last placed component. Two cases can be differentiated for $\mathcal{C}_k$:

1. $\mathcal{C}_k$ is the initial placement for its component.
2. $\mathcal{C}_k$ can be seen by $\mathcal{C}_p$ (with $p < k$, s.t. p is minimal), which is part of the same component.

For case 1, we know that after `BT_2_inner` finishes satisfying the component of $\mathcal{C}_{k-1}$, it calls `BT_2_outer` to loop over all possible square placements that do not see any other squares and comply with the outer rank constraint. Since the component of $\mathcal{C}_k$ has the greatest r_{outer} value, $\mathcal{C}_k$ will be placed by definition of `BT_2_outer`.

In case 2, $\mathcal{C}_k$ belongs to a subsequence $\mathcal{C}[i,j]$ of placements made by one call to `BT_2_inner` in order to satisfy $\mathcal{C}_p$. Since we defined our canonical construction to adhere to the inner rank constraint for each placement, `BT_2_inner` will eventually also place $\mathcal{C}[i,j]$ while satisfying $\mathcal{C}_p$.

With case 1 as base case and case 2 as inductive step, we arrive at our complete induction. It proves that all canonical constructions for valid configurations will be reached by `BT_2`. □

Corollary 4.8. *BT_2 is exact and always finds the optimal solution* $S_K(N)$.

4.3.3 Implementation Outline

To reiterate, bitboards are containers that store a truth value for each cell of the grid. The interface of this data structure is defined in Section 5.3. Two important bitboards will be passed through the calls of `BT_2_inner` and `BT_2_outer`: `occ` (occupancy) and `obs` (obstacles). `occ` indicates which cells of the grid are forbidden for new placements. A cell may be forbidden because it is either already occupied by a square placement or it is in the line of sight of an already satisfied square. `obs` indicates which cells are occupied by square placements. Hence, `obs` is a subset of `occ`. Furthermore, we keep track of the *depth*, used to determine the *root* in `BT_2_inner`, and the positional index of the initial square to the current component (`min_pos`).

In the pseudo implementation of `BT_2_outer` (Listing 4.3), following helper functions were used:

Name	Description
`bb_sq(pos, l)`	Creates a bitboard with cells that indicate the placement of side length `l` at position (index) `pos`.
`next_gen(l, templ, occ)`	Returns a "generator" bitboard; Each set bit represents a possible placement with side length `l`. Each placement in the resulting generator has to have at least one common cell with `templ`, that is the "generator template", and no common cells with `occ`.

As outer rank function, the positional index of the initial placement is used (Listing 4.3, line 10).

Listing 4.3: BT_2_outer Pseudo Implementation

```
1  procedure BT_2_outer(grid, occ: bb, obs: bb, min_pos, depth):
2    max_squares := grid.placed
3
4    for l := 1..N:
5      bb gen := next_gen(l, o̅c̅c̅, occ)
6      if gen.empty(): break
7
8      while not gen.empty():
9        pos := gen.pop_lsb()
10       if pos < min_pos: continue
11
12       grid.push(pos % N, pos / N, l)
13       occ = occ ⊕ bb_sq(pos, l)
14       obs = obs ⊕ bb_sq(pos, l)
15
16       max_squares = max(max_squares, BT_2_inner(occ, obs, pos,
            depth))
17
18       grid.pop()
19       occ = occ ⊕ bb_sq(pos, l)
20       obs = obs ⊕ bb_sq(pos, l)
21
22   return max_squares
```

This implementation loops over all possible square placements that do not see any previously placed squares (line 8). It then places the square in both grid and bitboards, and finally calls `BT_2_inner` (line 16). After that, it removes the square placement again, ready to try the next one. Like with `BT_1`, we have to keep track of the maximal number of squares placeable in the given grid (`max_squares`). Since we call

`BT_2_outer` only when a valid configuration is reached, we initialize `max_squares` to the number of squares currently placed (line 2).

The pseudo-implementation for `BT_2_inner` can be found in Appendix A.2. Inside this procedure, we loop over all possible sequences of placements seen by the *root* that satisfy the root. For that, we perform an iterative backtracking search, where `d` denotes the number of placements carried out within this `BT_2_inner` call (Listing A.1, loop at line 30). Each level has the following variables to keep track of its state:

Name	Description
`ls[p]`	The side length of the `d`th placement.
`occs[p]`	Occupancy bitboard after `d` placements.
`obss[p]`	Obstacle bitboard.
`gen[p]`	Generator bitboard (positions yet to try).
`templ[p]`	Template bitboard, used to create generator bitboards.
`hitss[p]`	Bitboard signifying which visible neighbors of `root` from before this `BT_2_inner` call have been blocked by `d`th placement.
`ranks[p]`	Holds the *rank* from the inner rank constraint of `d`th placement.

To know when the root is satisfied, we need to count how many visible neighbors it has. `d` already tells us how many new squares we placed in the root's vision. In addition, we need to keep track of the visible neighbors root had before this call to `BT_2_inner`. We therefore use a `counter` that keeps track of them and registers when a newly placed square completely blocks the visible connection to one. After each newly placed square, the `counter` is consulted to check if the root sees exactly K other squares. Whenever that is the case, we call the procedure recursively (line 54).

After confirming that a placement is (1) valid, (2) has a greater positional index than the initial square of this component, and (3) complies with the inner rank constraint, we proceed by placing it and preparing the bitboards for the next level (lines 44-50).

4.4 Simulated Annealing

For greater values of N, performing an exact search for $S_K(N)$ becomes computationally infeasible. We therefore also ran experiments with a "Simulated Annealing" (SA) approach. As a framework, we used the general approach for threshold algorithms mentioned by Aarts and Lenstra [1] (Listing 4.4). In this algorithm, i and j represent states of the system (in our case, configurations), and $\mathcal{N}(i)$ denotes the set of neighbor states of i.

We substituted the threshold check (Listing 4.4, line 9) with the SA state update procedure stated by Bertsimas and Tsitsiklis [3]:

$$
\begin{aligned}
&\text{If } f(j) \leq f(i), \text{ then } i = j \\
&\text{If } f(j) > f(i), \text{ then} \\
&\qquad i = j \text{ with probability } \exp[-(f(j) - f(i))/T(t)]
\end{aligned}
$$

where $T(t)$ is the temperature at timestamp t and f is some real-valued cost function for given state. As mentioned in Section 2.3, we employ the negative of an exact

Listing 4.4: Pseudocode of a class of threshold algorithms

```
procedure THRESHOLD_ALGORITHM;
begin
  INITIALIZE(i_start);
  i := i_start;
  k := 0;

  repeat
    GENERATE(j from N(i));
    if f(j) - f(i) < t_k then i := j;
    k := k+1;
  until STOP;
end;
```

search (`BT_2`) as cost function f to evaluate given state.

For SA to function properly, we also need an encoding for grid configurations and a definition of the neighborhood function. We therefore devised the notion of a square *group*, which consists of one *root* square and K other squares in its vision. This way, the exact search does not have to satisfy the roots anymore, and it also reduces the possible placement by the line of sight of roots. SA operates on configurations that consist of either one or more such square groups. Now, we can define the neighborhood for such a configuration to be the result of one *mutation* to a root/group. Mutations involve:

- In-/Decrease root square side length.
- Move an entire group north, south, east, west, or diagonally.
- Keep root square, while replacing all other squares in the group.
- Relocate root and replace all squares in the group.
- Delete an entire group.
- Create a new group.

When satisfying the root of a group, placements closer to the root are favored by decreasing the placement chance as the distance to the root gets larger. This ought to emulate the observation that in solution grids, visible neighbors of squares are often close in proximity, and also help maximize the number of placed squares.

Before running the SA algorithm, an initial configuration is generated. This initial configuration comprises $N^2/(4 \cdot (K+1) \cdot \texttt{L\_MIN}^2)$ groups, which is a reasonable trade-off between being too densely populated and slow exact search runtimes. The temperature function employed is $T(t) = \alpha^t$ for some constant $\alpha \in (0,1)$. Our stop condition is some arbitrary threshold on the temperature: $T_{min} \approx 10^{-12}$. Section 6.3 comprises the results of this approach.

5 Implementation Details

`BT_1` and `BT_2` differ slightly in the requirements for their data structures. Since only `BT_1` uses `grid.satisfied`, we will have two versions of the `square` and `grid` structure. `BT_2` uses the plain versions whilst `BT_1` requires the extended versions for better performance. On the other hand, `BT_2` uses the `bitboard` and `counter` structures for even better performance.

5.1 The `square` structure

The square structure at its core should simply be able to represent a square placement. So, it should at least contain a position (x, y) and the side length l.

For a faster check later on (Section 5.2, second improvement), whether the grid is satisfied, we may also store the number of visible neighbors inside each square. And to efficiently keep this number up to date, we introduce four additional counters, one for each side. This allows us to quickly recount the visible neighbors on one side of the square.

Following are UML diagrams for both versions:

square
+ (x,y): Position. + `l`: Side length.

square (extended)
+ (x,y): Position. + `l`: Side length. + `n`: Number of visible neighbors. + `n_no`, `n_ea`, `n_so`, `n_we`: Number of visible neighbors on respective side (north, east, south, west).

5.2 The `grid` class

At its core, the `grid` class is a last-in-first-out (LIFO) stack of `squares`. Hence, the base implementation consists of a pre-allocated array, accompanied by a counter. Additionally, it has a map that stores one square pointer for each *grid cell*. A *grid cell* is an indexable entry of a pointer map. Empty cells within the grid are represented by a special *empty* value. Thereby, our simple `grid` class is complete:

grid
+ `sq[MAX_SQ]`: Stack of squares. + `placed`: Size of stack. + `pmap[N*N]`: Pointer map to square at position $(y \cdot N) + x$.
+ `push(x, y, l)`: Places square. + `pop()`: Removes last placed square.

The implementation of `push` for the simple `grid` version involves growing the square stack by storing the given position and side length on top of the stack, then updating `pmap` to point to the new square at cells occupied by that square. `pop` reverts the `push` operation by setting the occupied cells in `pmap` to the special *empty* value and decreasing the `placed` counter by one.

`BT_1` requires `grid` to additionally support the following functionalities:

1. `grid.valid()`, which checks whether the current configuration is valid and

2. `grid.can_place(x, y, l)`, which checks whether a square of side length l and position (x, y) still fits on the grid.

`grid.can_place(x, y, l)` is implemented by iterating over all cells in `pmap` that would be occupied by placement (x, y, l) and returning whether all cells are empty.

The implementation of `grid.valid()`, on the other hand, poses a nontrivial design problem. The naive implementation loops over all placed squares and checks if they are satisfied. It does so by walking the `pmap` from all cells on the square's edge in each direction until it hits another square and counting the unique visible neighbors. It is quite obvious that this approach does not perform well when it comes to efficiency and becomes a critical bottleneck since `grid.valid()` is an integral part of the `BT_1` algorithm. We therefore propose two variants of the naive approach.

Our first variant, "Directional Maps" of this approach introduces additional pointer maps, one for each direction—north, east, south, and west. For each cell, they hold the pointer to the next visible square in the given direction. This way, we don't have to loop over map entries to find the next visible neighbor and can simply read it from the respective map. To keep these maps up-to-date, they have to be modified for each `push` or `pop` call. But since each call to `grid.valid()` queries all squares for their satisfaction; the instant lookup times outweigh the cost of maintaining these additional maps.

The second variant, "Square Counters" involves keeping track of the number of satisfied squares and which squares are satisfied. We therefore use the extended version of the `square` structure introduced in Section 5.1. Each square additionally stores the number of its visible neighbors and a counter for visible neighbors in each direction—north, east, south, and west. This allows us to efficiently maintain the counter of satisfied squares when updating the grid. With this setup, `grid.valid()` becomes a simple comparison between the number of placed squares (`placed`) and the number of satisfied squares (`satisfied`). It returns `true` only in the case that both counters are equal.

How these two variants compare to the naive version is discussed in Section 6.4. This finalizes our extended `grid` class:

grid (extended)
– `sq[MAX_SQ]`: Stack of squares. + `placed`: Size of stack. – `satisfied`: Number of satisfied squares. – `pmap[N*N]`: Pointer map to square at position $(y \cdot N) + x$. – `pmap_no[N*N]`, `pmap_ea[N*N]`, `pmap_so[N*N]`, `pmap_we[N*N]`: Directional pointer maps.
+ `push(x, y, l)`: Places square. + `pop()`: Removes last placed square. + `can_place(x, y, l)`: Checks whether square fits. + `valid()`: Yields `satisfied==placed`.

5.3 The `bitboard` class

A bitboard can be seen as a set of bitwise truth values indicating whether a value is in the set or not. It is implemented either by a single or multiple integer words, depending on how many values are needed. Our implementation uses 64-bit words.

The following bitwise operations are supported:

bitboard
– s[BB_WORDS]: integer words, bits hold truth values.
+ `empty()`: Returns `true` if entire bitboard is 0, `false` otherwise. + `set(i)`: Sets bit at index `i` to 1. + $A \cup B$: Union, 1-bits set in A or B. + $A \cap B$: Intersection, 1-bits set in both, A and B. + $A \oplus B$: XOR, 1-bits set in exclusively one, A or B. + $\overline{A}$: Inverse, flips 1-bits to 0, and vice versa. + `pop_lsb()`: Returns lowest-indexed 1-bit, whilst flipping it to 0. + `shift_no()`, `shift_ea()`, `shift_so()`, `shift_we()`: Shifts all bits once in corresponding direction, discarding overflowing bits.

The bits represent the existence or absence of an object on our $N \times N$ grid by a 1- or 0-bit at the corresponding index, respectively. To reiterate, we index our grid left-to-right, top-to-bottom. The indices signify the order of bits, e.g., an index of 0 is the least significant bit (lsb). Fig. 5.1 shows the bitboard representation for a grid with side length 8, where the x coordinates go along the horizontal and y coordinates along the vertical axis.

The number of words in each bitboard, `BB_WORDS`, is constant and chosen depending on N and the number of bits in a word, in our case 64. It is set to the value `BB_WORDS` $:= \lceil N^2/64 \rceil$.

In the case $N = 8$, the entire grid can be perfectly represented by a single 64-bit integer. For cases $N < 8$, the last, unnecessary bits will be masked away after operations that may set these bits. For cases $N > 8$, we need to stitch together our grid using multiple words. We therefore use an approach mentioned by Browne, where the

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

Figure 5.1: Bitorder for a bitboard of side length 8

words follow each other immediately [4]. As an example, Fig. 5.2 shows the layout for a board with $N = 14$, using 4 words.

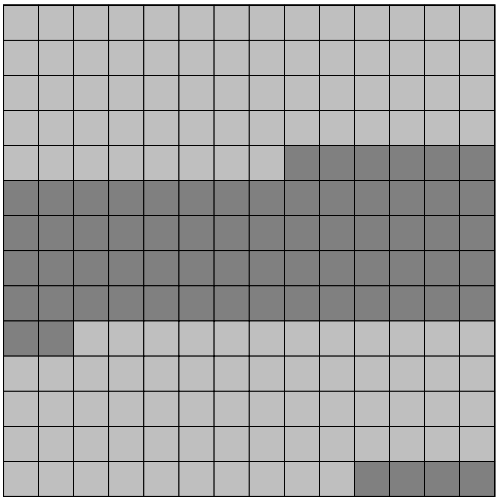


Figure 5.2: Multi-word bitboard ($N = 14$)

When using multiple words, the bitwise operations are simply applied to all words in the bitboard. For the shift operations, one needs to be especially careful to copy values over from one word to another, on the literal edge cases. Here, the order in which the words are bit-shifted is important to think about in order not to overwrite values still to be shifted. To shift east or west, we shift the words by one, right or left, respectively. Shifting north or south requires shifting all words by N to the left or right.

Because these operations are based on bitwise integer operations, they are fast to execute in practice.

5.4 The `counter` class

The `counter` keeps track of squares in the vision of a `root`, excluding newly placed squares by `BT_2_inner`. It is responsible for recognizing when a newly placed square blocks the vision of a former visible neighbor. It may be implemented by storing one integer counter for each placement other than `root`. This integer counter keeps track of how many "visibility lines" connect each square to the root. Hence, when it reaches zero, `total` must be decreased by one; When it becomes one (from zero), `total` must be incremented by one. Fig. 5.3 illustrates an example to build intuition for these integer counters, which are represented by the numbers inside the squares.

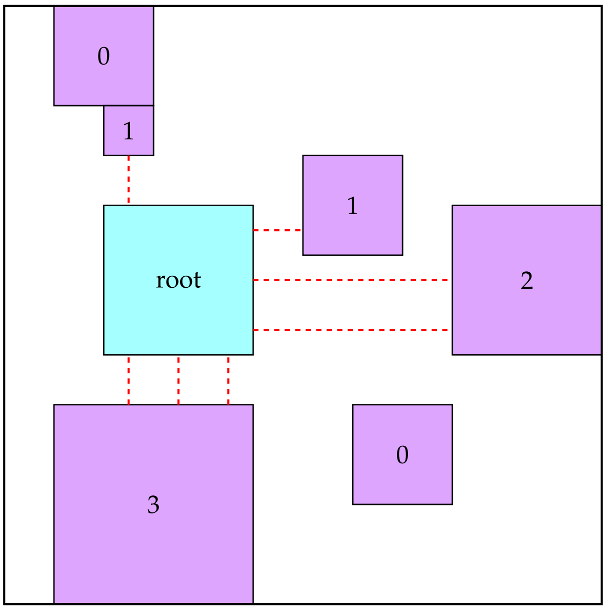


Figure 5.3: Number of visibility lines to the root

This class has the following interface:

counter
+ `total`: The number of distinct placements still in the vision of `root`. + `smaller`: The number of squares placed **before** `root` in the current construction. – `visibles`: Internal bitboard keeping track of visible neighbors' cells that are still visible.
+ `raise(grid, hits)`: Raises the count for squares at cells indicated by the bitboard `hits`. + `drop(grid, root, p)`: Calculates previously visible cells blocked by placement `p`. It returns the bitboard of affected cells.

Calling `drop` potentially removes set bits from `visibles`, whilst `raise` puts them back. Hence, calling `counter.raise` with a bitboard acquired by a `counter.drop` call, results in the same state as before.

6 Experimental Findings

All experimental benchmarks were conducted on an Intel® Xeon® E5-2680 v4 (2.40 GHz). The search program is written in the C programming language to avoid performance bottlenecks by the language. The parameters N and K are specified during compilation time to enable effective optimizations by the compiler, GCC 12.2 with -O3 -flto flags. Moreover, the entire program requires <1MiB of memory throughout execution, which ensures good cache performance.

The experiments we ran are categorized into the following parts:

- A comparison between `BT_1` and `BT_2` (Section 6.1)
- Solutions we were able to compute by exact search (Section 6.2)
- The effectiveness of our meta-heuristic approach (Section 6.3)
- Effectiveness of the improvements to the `grid` structure discussed in Section 5.2 (Section 6.4)
- Experiments on ideas to further enhance the search (Section 6.5)

6.1 Comparison of Exact Searches

When comparing the exact search algorithms proposed, there are two metrics of significance: NPS (nodes per second) and the total number of nodes visited. We may define a node to be a recursive call to either `BT_1_rec`, `BT_2_inner`, or `BT_2_outer`. But these procedures behave differently due to their distinguished tasks. To illustrate, `BT_2_inner` may perform multiple placements in one call in order to satisfy the current *root* placement in all possible ways, while `BT_1_rec` gets one call for every placement made. In order to compare both algorithms in an unbiased way, we use the metrics of "placements per second" and the total number of placements.

Fig. 6.1 shows a comparison of both approaches with respect to their respective placements per second in millions (MPPS), averaged over small values of N (2-6). Since the PPS is not directly influenced by N, averaging over N does not falsify the results and takes multiple runs into account, thereby stabilizing the measurements.

We could only collect sensible data for $K \in \{1, 2, 3, 4\}$ in a reasonable amount of time. Since the measurements suggest a constant number of placements per time unit for each method, we deem it to be unfit as the deciding factor for which method performs better. Next, we take a look at the total number of placements made per search.

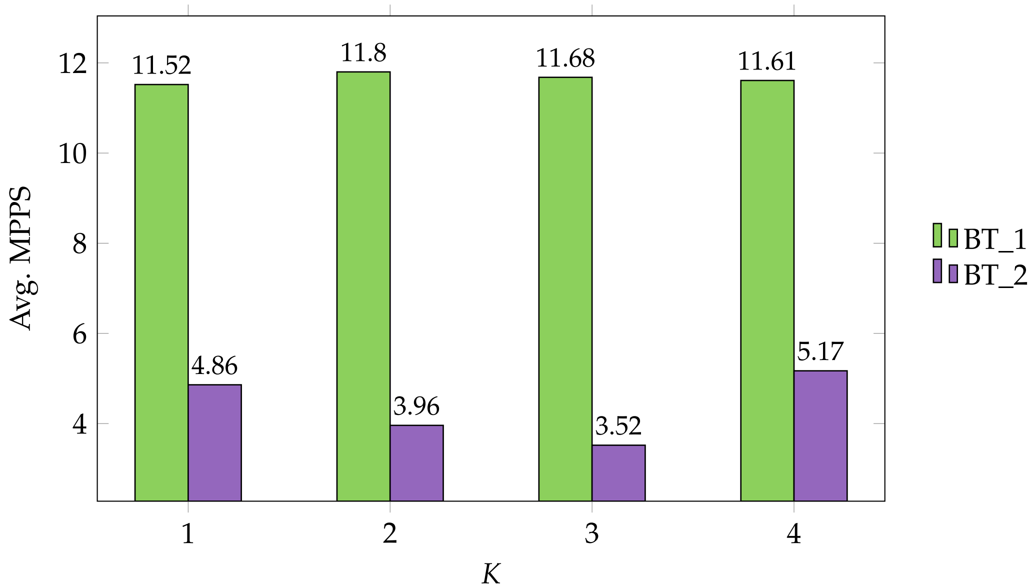


Figure 6.1: Placements per second (millions) by `BT_1` and `BT_2`

Fig. 6.2 shows the placements made by `BT_1` over the course of each search. Since the searches generate the exact same sequence of placements for each K, the graphs look the same for all K. The apparent improvement for $K > 4$ stems from the `L_MIN` optimization discussed in Section 6.5.1 and will further reduce placements as K surpasses multiples of 4.

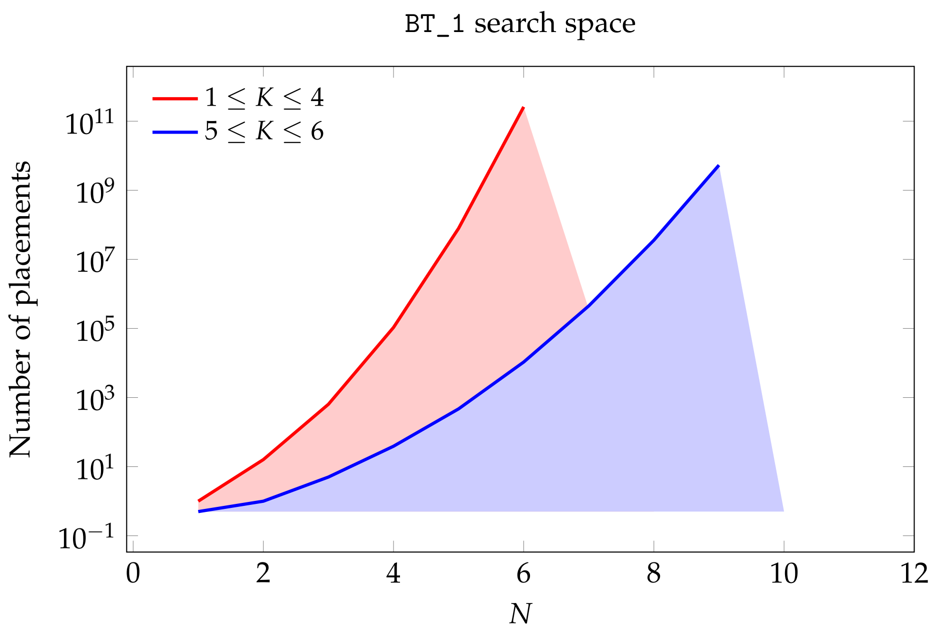


Figure 6.2: Placements made by `BT_1`

Fig. 6.3 compares the placements made by `BT_2` to the previously mentioned `BT_1` values. As one can see, `BT_2` makes remarkably fewer (orders of magnitude) placements than its BT_1 counterparts for the same values of K. This allowed us to compute the solutions for larger values of N. The trends suggest that the difference between them only grows more apparent for even greater N.

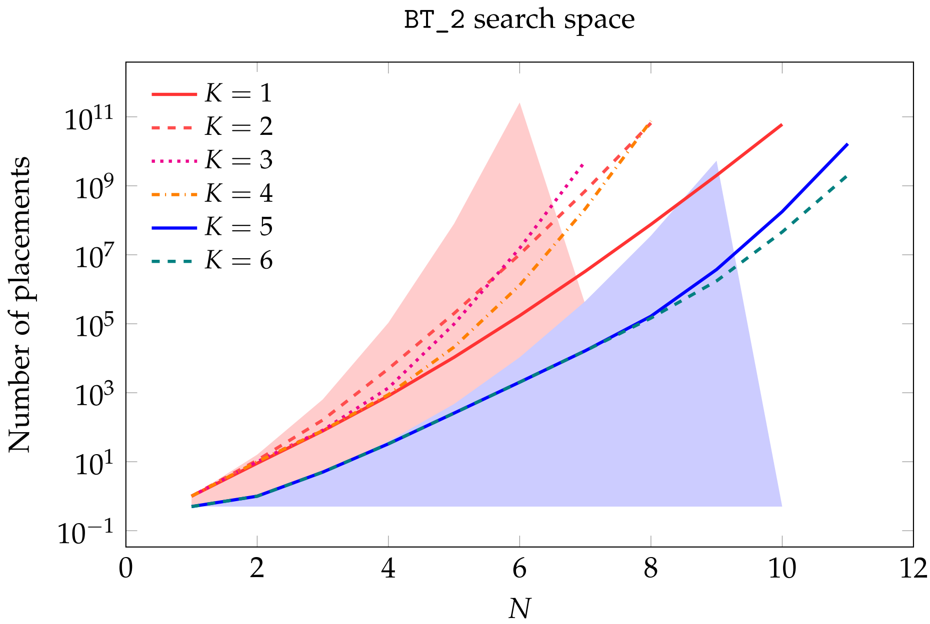


Figure 6.3: Placements made by BT_2

Conclusively, this comparison shows that BT_2 outperforms the other algorithm and is always preferred over BT_1 for an exact search.

6.2 Solutions Found

Applying BT_2 to a set of small parameters of N and K, we can find the results in Tab. 6.1 in a reasonable amount of time. The computation times and solution grids for each of these results can be found in Tab. 6.2 and Appendix A.3, respectively. Because we proved that the algorithm is exact, the found values as well are guaranteed to be exact.

$K \backslash N$	1	2	3	4	5	6	7	8	9	10	11
1	0	2	4	4	6	8	8	10	12	12	14
2	0	4	6	8	10	12	14	16	N/A	N/A	N/A
3	0	0	0	0	6	12	16	20	N/A	N/A	N/A
4	0	0	0	0	0	0	0	20	N/A	N/A	N/A
5	0	0	0	0	0	0	0	0	0	0	14

Table 6.1: $S_K(N)$ for small values of N and K

$K \backslash N$	1	2	3	4	5	6	7	8	9	10	11
1	<1ms	<1ms	<1ms	<1ms	3ms	48ms	1s	12s	9min	4.8h	186h
2	<1ms	<1ms	<1ms	1ms	62ms	2.5s	2.4min	3.7h	N/A	N/A	N/A
3	<1ms	<1ms	<1ms	<1ms	18ms	2.8s	17min	202h	N/A	N/A	N/A
4	<1ms	<1ms	<1ms	<1ms	4ms	325ms	33s	3.2h	N/A	N/A	N/A
5	<1ms	<1ms	<1ms	<1ms	<1ms	1ms	5ms	29ms	1.3s	73s	2.1h

Table 6.2: Computation times for $S_K(N)$

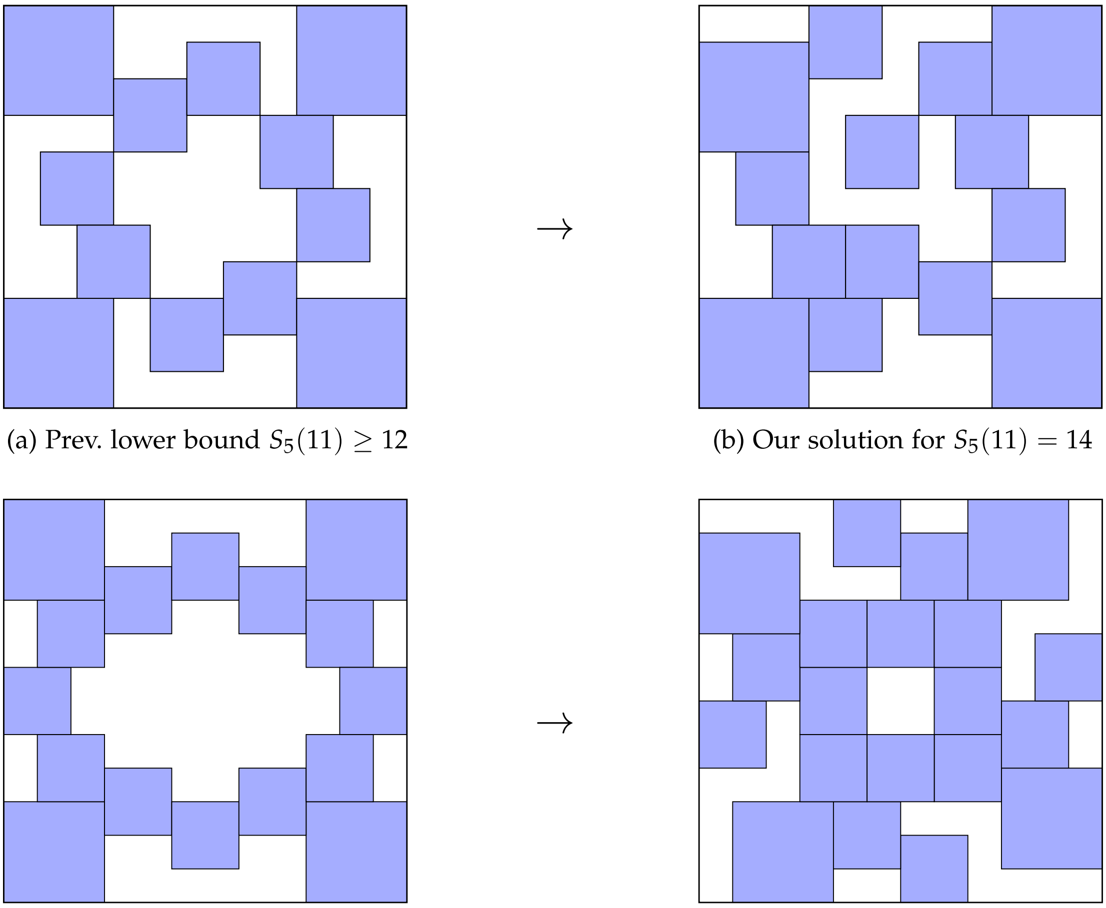
Previous findings compiled by Friedman included the following bounds:

- $S_0(N) = N$
- $S_1(N) = 2(N - 1 - \lfloor \frac{N-1}{3} \rfloor)$ *
- $S_3(5) = 6, \quad S_3(6) \geq 12, \quad S_3(7) \geq 16, \quad S_3(20) \geq 20$ which we could all confirm to be tight.
- $S_4(2N) = 4N^2 - 12N + 4$
- $S_5(11) \geq 12$, which we could prove to be 14.
- $S_5(12) \geq 16$, which we could prove to be 20.

Solutions for $K \in \{0, 1, 2, 4\}$ can be easily created using a construction recipe (see Appendix A.4). Further lower bounds were provided, which we could neither improve nor prove to be tight. [13]

Overall, we could reproduce and confirm the findings of previous work. Further, we did improve the known bounds by Morandi and guarantee exactness for $S_5(11)$ (Fig. 6.4b) and $S_5(12)$ (Fig. 6.4d). [13]

* The mentioned formula $\lfloor 4N/3 \rfloor$ by Friedman was corrected.



(a) Prev. lower bound $S_5(11) \geq 12$

(b) Our solution for $S_5(11) = 14$

(c) Prev. lower bound $S_5(12) \geq 16$

(d) Our solution for $S_5(12) = 20$

Figure 6.4: Improved exact bounds compared to previous results

6.3 Meta-Heuristic Performance

After numerous experiments ($K \geq 5, N \geq 11$) with the SA approach as declared in Section 4.4, no valid configurations (apart from the trivial empty grid) have been found. An analysis of the search space was conducted, using `BT_2`. Fig. 6.5 shows the ratio of valid configurations over the total number of configurations visited during a search. For reasonable values of K (≥ 3), one can observe a drastic decrease in that ratio as N increases, meaning valid configurations become more sparse. Hence, for greater values of N, a random initial configuration is highly unlikely to be a subset of a valid configuration. This poses a serious issue, since the SA in our approach is solely guided by the maximal number of squares placeable with this initial configuration, such that the visibility constraint holds for every square. If this metric is 0 for almost every initial configuration SA tries, it becomes practically impossible for it to improve towards a valid configuration.

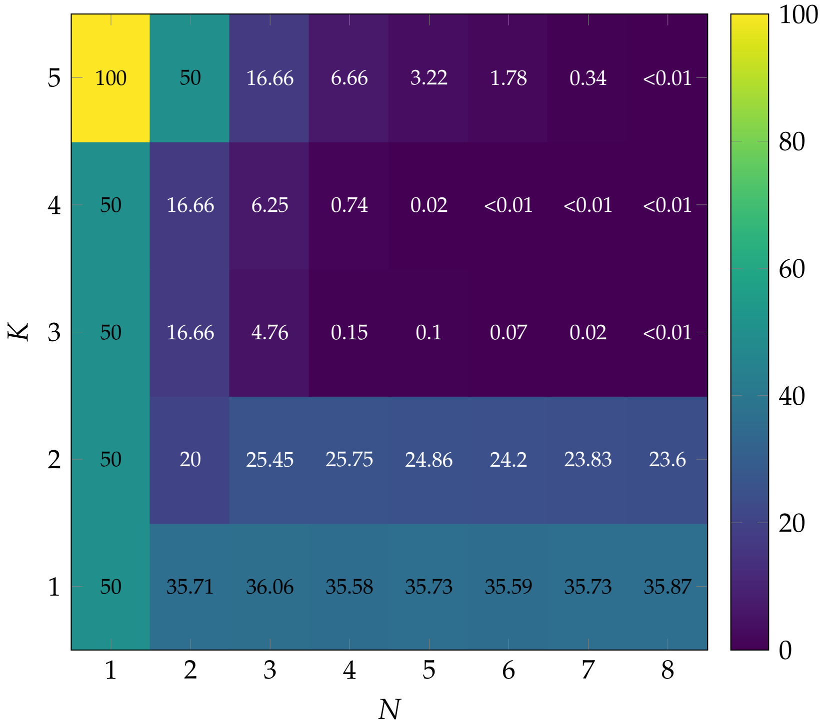


Figure 6.5: Valid configurations during search (% of visited configurations)

Following modification to the cost function SA uses may greatly improve its ability to find valid configurations: Allow placements to violate the visibility constraint, while incorporating minimizing constraint violations into the cost function, similar to Ref. [10]. We propose a cost function of following structure:

$$cost = -\max_{c \supseteq \mathcal{C}_i} \left[\alpha \, |c| - \beta \sum_{p \in c} \left(N(p) - K \right)^2 \right] \tag{6.1}$$

where $\mathcal{C}_i$ is the initial configuration produced by SA and c is any configuration considered by the exact search with that initial configuration. $N(p)$ denotes the number of visible neighbors for a placement p. This cost function rewards maximizing the number of placed squares ($|c|$), while punishing visibility constraint violations (the sum). α and β are constant coefficients to weigh the respective parts of the cost. Again, we negate our objective function to comply with the convention that a lower *cost* value is better. On a final note, for this cost function, `BT_1` may be a better fit since it also considers invalid configurations, whereas `BT_2` only visits valid configurations.

During experiments, SA was also observed to frequently undo mutations during the next iteration and revisit initial configurations. This flaw could be mitigated using some kind of history to avoid duplicate computation, e.g., tabu search. These findings align with those reported by Ref. [10].

6.4 `grid` Optimizations

In Section 5.2, we presented two variants of the naive `grid` implementation. Fig. 6.6 shows a comparison of efficiency for both. In this experiment, we timed the speed of all combinations of variants, with parameters $K = 2, N \in [1, 50]$. The metric used, MNPS, denotes the number of nodes visited per second in millions. In this case, a node denotes one call to the `BT_1_rec` procedure introduced in Section 4.2. An increase in NPS correlates with a more efficient implementation and is thus directly tied to the conceptual improvement by the variation proposed.

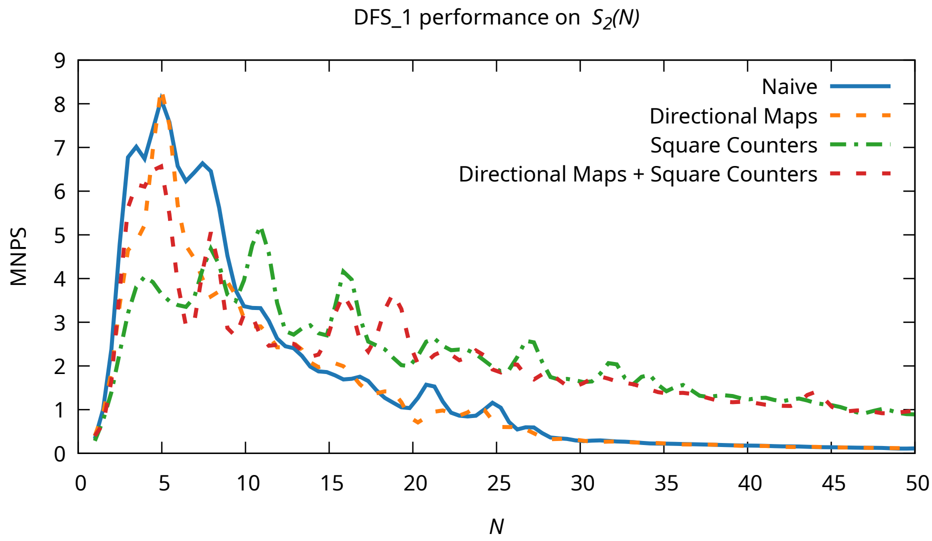


Figure 6.6: (`BT_1`) Nodes per second, comparing `grid.valid()` implementations

As one can see, the second variant's ("Square Counters") speed-up is quite impressive ($\sim 10\times$ naive approach's NPS) but only noticeable for higher values of N. The first variant ("Directional Maps") further speeds up the "Square Counters" approach for smaller N.

Measurements for other values of K showed the same results, although less extreme. Because capturing data for searches with such high N is computationally infeasible, the data was captured over 60,000,000 nodes on a random initial grid for each N.

In conclusion, the naive implementation remains the most efficient for small values of N (< 10). After that point, the variant "Square Counters" overtakes the naive version in terms of NPS and speeds up the search significantly.

6.5 Additional Insights

This section comprises further ideas regarding possible speed-ups to the search and observations about the nature of this packing problem. Therefore, these concepts are not to be generalized and may not be directly transferable to other problems.

6.5.1 Minimal side length and Upper bound on $S_K(N)$

One can observe that any square placed in a valid $S_5(N)$ configuration has a side length of at least 2. That is because a square with side length 1 can see at most 4 other squares, one in each direction. This observation implies that we can impose a minimum `L_MIN`=$\lceil K/4 \rceil$ on the side length of placed squares. We may thereby prune all placements with side lengths smaller than `L_MIN`.

This observation is also the key idea for the upper bound $S_K(N) \leq \lfloor N/\lceil K/4 \rceil \rfloor^2$ proposed by Bevan [13]. This bound allows us to backtrack early, as soon as we place too many squares, cutting off unnecessary nodes in the search tree.

These search speed-ups are employed in both proposed exact algorithms.

6.5.2 Early Backtrack

The search can be sped up by backtracking as soon as a placement causes any square to see more than K other squares. This approach cuts off large parts of the search tree. Although all experiment runs returned with the correct result, some valid constructions may not be reached with this approach if they fulfill the following properties:

1. Some placement $\mathcal{C}_t$ in the construction $\mathcal{C}$ leads to a square seeing more than K other squares.
2. A later placement $\mathcal{C}_d$ for a $d > t$, “blocks” the visible connection between multiple squares, such that the visibility constraint again holds for every square.

Fig. 6.7 shows such a construction for $S_3(8)$. This valid configuration would not have been reached by “early backtrack” search if X was placed before X′. Before placing X′, X already sees 4 other squares, which is more than $K = 3$, but X′ blocks visible neighbors of X, resulting in a valid configuration. Whether or not a configuration is

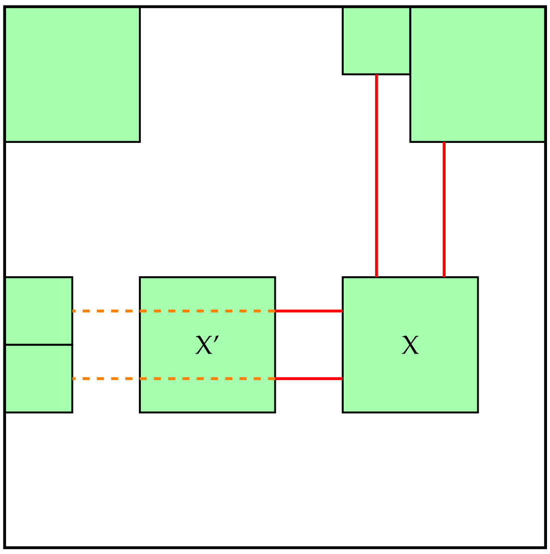


Figure 6.7: Early Backtrack cutoff miss

reachable also depends on the order in which the squares are placed, i.e. the rank functions introduced in Section 4.2 or Section 4.3.2. Proving whether there exists a solution for every $S_K(N)$ that would be found while employing early backtrack is beyond the scope of this thesis. Empirically, this approach shows a speed-up of around $2\times$ in experiments, making it a useful tool for finding lower bounds on $S_K(N)$.

6.5.3 Symmetry Property

When looking at solutions compiled by Friedman, one may assume that for every $S_K(N)$ there exists a solution that is reflective, diagonal, or point symmetric [13]. If this property were true, one could search all valid configurations significantly faster by only considering half of the board. The other half would be implied by the respective symmetry. But searching for symmetric solutions for $S_5(11)$ with one of our exact searches returns no solutions, proving that this property does not hold for all $S_K(N)$.

Although this property doesn't hold for all $S_K(N)$, the sketched approach could still be used to speed up the search and find lower bounds on $S_K(N)$.

7 Conclusion

In this thesis, we tackled a specific packing problem on a discrete grid, where each placed square has to "see" exactly K other squares. We were interested in the metric $S_K(N)$, the maximal number of such squares placeable on an $N \times N$ grid, along with respective solution grids.

With this objective in mind, we explored two exact backtracking algorithms (`BT_1` and `BT_2`) to traverse the search space. Our findings include showing that existing lower bounds for $S_3(5)$, $S_3(6)$, $S_3(7)$, $S_3(8)$, $S_4(8)$ are tight, and improving the bounds for $S_5(11)$ and $S_5(12)$. A comparison between the two backtracking algorithms shows a vast improvement from `BT_1` to `BT_2` (Section 6.1). This demonstrates how eliminating branches on each level of the search tree can speed up the backtracking search significantly, in our case even by orders of magnitude. However, the exponential growth in the number of visited configurations renders exact searches impractical when aiming to compute solutions for larger problem sizes. Our benchmarks suggest that simply increasing computational power or parallelizing the computation will not suffice to solve instances for larger values of N. Therefore, new algorithmic approaches are needed for further progress, be it exact or approximate.

In an effort to reduce computation times, we also applied simulated annealing to the problem. Although it proved ineffective in its current form, the suggested variation of this approach (Section 6.3) may be more promising for future research. Other techniques that remain to be explored include the methods discussed in Section 6.5, leveraging problem-specific properties, such as symmetries of solutions, etc. These techniques are significantly faster than our exact searches but may not always find optimal solutions. Additionally, a more effective implementation of heuristic search algorithms—such as SA with the proposed cost function adjustments—may yield even better results.

We learned that the simulated annealing framework requires careful design of both the cost function and the neighborhood function. A cost function that is too coarse-grained cannot pick up on meaningful differences between configurations, making it difficult for the algorithm to converge towards an optimal solution.

Overall, this thesis contributes to a deeper understanding of this geometric packing problem, presents improvements to known bounds, and introduces methods that may also be applicable to other combinatorial tasks of similar structure. Future work may extend these ideas to related problems, such as maximizing occupied area, or explore refined heuristics and hybrid algorithms to overcome the challenges our approaches faced.

References

[1] E. H. L. Aarts and J. K. Lenstra. *Local search in combinatorial optimization*. en. 30. Wiley, 1997, pp. 91–120. ISBN: 9780471948223.

[2] B. Addis, M. Locatelli, and F. Schoen. "Disk Packing in a Square: A New Global Optimization Approach". en. In: *INFORMS Journal on Computing* 20 (4 Nov. 2008), pp. 516–524. DOI: 10.1287/ijoc.1080.0263. URL: https://doi.org/10.1287/ijoc.1080.0263.

[3] Dimitris Bertsimas and John Tsitsiklis. "Simulated Annealing". In: *Statistical Science* 8 (1 Feb. 1993). DOI: 10.1214/ss/1177011077. URL: https://doi.org/10.1214/ss/1177011077.

[4] Cameron Browne. "Bitboard Methods for Games". In: *ICGA Journal* 37 (2 June 2014), pp. 67–84. DOI: 10.3233/icg-2014-37202. URL: https://doi.org/10.3233/icg-2014-37202.

[5] Katrin Casel et al. "Combinatorial Properties and Recognition of Unit Square Visibility Graphs". en. In: *Discrete & Computational Geometry* 69 (4 June 2023), pp. 937–980. DOI: 10.1007/s00454-022-00414-8. URL: https://doi.org/10.1007/s00454-022-00414-8.

[6] N.E. Collins, R.W. Eglese, and B.L. Golden. "Simulated Annealing - An Annotated Bibliography". en. In: *American Journal of Mathematical and Management Sciences* 8 (3-4 Feb. 1988), pp. 209–307. DOI: 10.1080/01966324.1988.10737242. URL: https://doi.org/10.1080/01966324.1988.10737242.

[7] Hallard T Croft, Kenneth Falconer, and Richard K Guy. *Unsolved problems in geometry: unsolved problems in intuitive mathematics*. Vol. 2. Springer Science \& Business Media, 2012.

[8] T. Csendes et al. *Packing equal circles in a square II.-New results for up to 100 circles using the TAMSASS-PECS algorithm*. en. Springer, 2001, pp. 207–224. ISBN: 9781402000096.

[9] Ole-Johan Dahl, Edsger Wybe Dijkstra, and Charles Antony Richard Hoare. *Structured programming*. en. Academic Press, 1972. ISBN: 9780122005503.

[10] Kathryn A. Dowsland. "Some experiments with simulated annealing techniques for packing problems". en. In: *European Journal of Operational Research* 68 (3 Aug. 1993), pp. 389–399. DOI: 10.1016/0377-2217(93)90195-s. URL: https://doi.org/10.1016/0377-2217(93)90195-s.

[11] Kathryn A. Dowsland and William B. Dowsland. "Packing problems". en. In: *European Journal of Operational Research* 56 (1 Jan. 1992), pp. 2–14. DOI: 10.1016/0377-2217(92)90288-k. URL: https://doi.org/10.1016/0377-2217(92)90288-k.

[12] Robert J. Fowler, Michael S. Paterson, and Steven L. Tanimoto. "Optimal packing and covering in the plane are NP-complete". en. In: *Information Processing Letters* 12 (3 June 1981), pp. 133–137. DOI: 10.1016/0020-0190(81)90111-3. URL: https://doi.org/10.1016/0020-0190(81)90111-3.

[13] Erich Friedman. "Problem of the Month (September 2007)". en. In: (2007). URL: https://erich-friedman.github.io/mathmagic/0907.html.

[14] Scott Kirkpatrick, C Daniel Gelatt Jr, and Mario P Vecchi. "Optimization by simulated annealing". In: *science* 220.4598 (1983), pp. 671–680.

[15] Donald Knuth. *Art of Computer Programming, Volume 4B, Fascicle 5: The - Mathematical Preliminaries Redux; Backtracking; Dancing Links*. en. Addison Wesley, 2019. ISBN: 0134671791.

[16] Andrea Lodi, Silvano Martello, and Michele Monaci. "Two-dimensional packing problems: A survey". en. In: *European Journal of Operational Research* 141 (2 Sept. 2002), pp. 241–252. DOI: 10.1016/s0377-2217(02)00123-6. URL: https://doi.org/10.1016/s0377-2217(02)00123-6.

[17] Pablo Segundo et al. "Efficient Search Using Bitboard Models". In: *2006 18th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'06)* (Arlington, VA, USA). IEEE, Nov. 2006, pp. 132–138. DOI: 10.1109/ictai.2006.53. URL: https://doi.org/10.1109/ictai.2006.53.

A Appendix

A.1 BT_2 finds $S_1(3)$

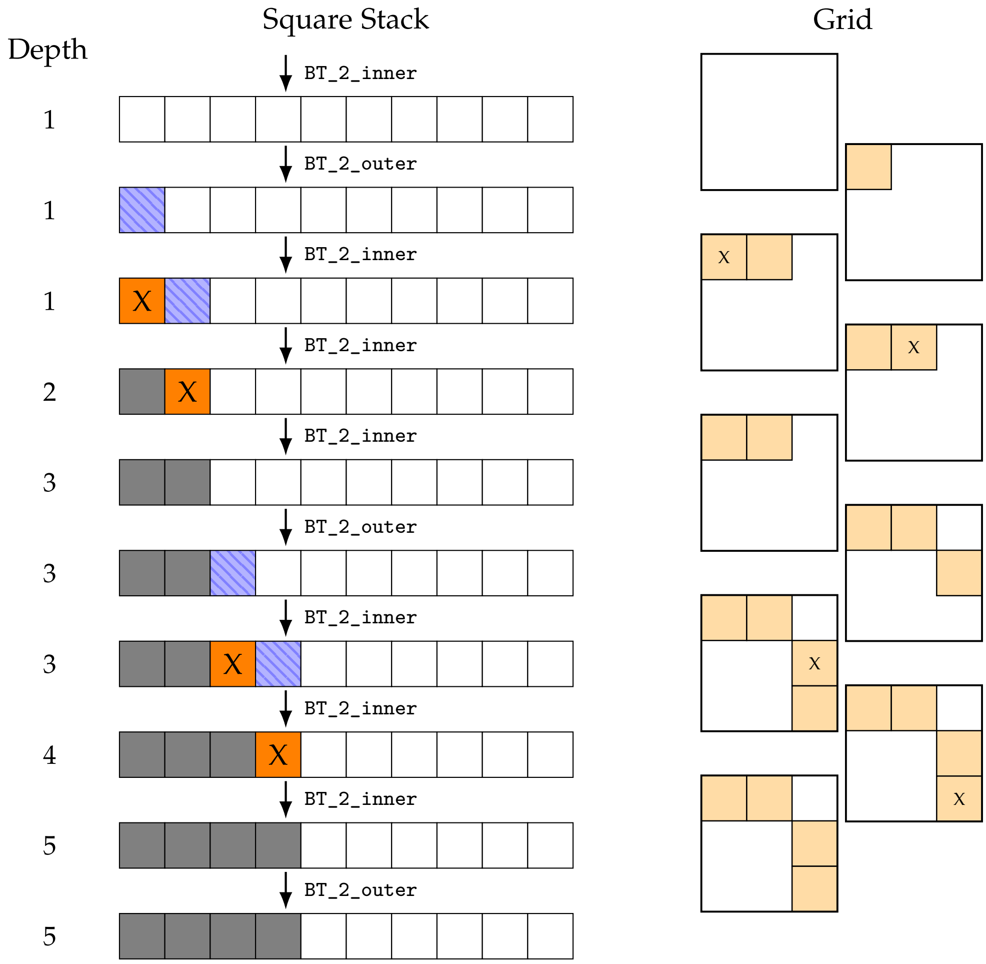


Figure A.1: Interplay between BT_2_inner and BT_2_outer for $S_1(3)$

Fig. A.1 shows the square stack, as well as the grid configuration for each call to BT_2_inner or BT_2_outer. The last call to BT_2_outer registers the found solution and will return 4 eventually.

A.2 `BT_2_inner` Pseudo Implementation

Following additional helper functions are used in the implementation:

- `cast_rays(p, occ, obs)`: Walks the grid from the square defined by placement `p` in all four directions, until it hits another square. It returns both the available cells seen by the root (excluding `occ` cells) and the first occupied cells in its path.
- `count_neighbors(grid, hits: bb)`: Returns a `counter` object that has already registered the cells indicated by `hits`.
- `pos_rank(root, pos)`: The inner rank function, dependent on root's placement.
- `next_occ(root, p, root_rays, prev_occ)`: Returns an altered version of the `prev_occ` bitboard: It adds the cells of placement `p` and the connecting visible line to `root` to it.
- `next_templ(root, p, occ, prev_templ)`: Removes the "shadow" cast by `root` on `p` from `prev_templ` and returns resulting bitboard.

Listing A.1: BT_2_inner Pseudo Implementation

```
procedure BT_2_inner(grid, occ: bb, obs: bb, min_pos, depth):
  if depth == grid.placed:
    return BT_2_outer(grid, occ, obs, min_pos, depth)

  root := grid.sq[depth]
  root_rays, root_hits := cast_rays(root, occ, obs)
  counter := count_neighbors(g, root_hits)

  already_seen := counter.smaller
  if already_seen > K: return 0
  to_be_placed := K - already_seen
  ls[to_be_placed]
  ranks[to_be_placed]
  bb gen[to_be_placed], hitss[to_be_placed], templ[to_be_placed + 1], occs[to_be_placed + 1], obss[to_be_placed + 1]
  ls[0]     = L_MIN
  ranks[0]  = 0
  templ[0]  = root_rays
  occs[0]   = occ
  obss[0]   = obs
  gen[0]    = next_gen(ls[0], templ[0], occs[0])

  max_squares := 0
  d           := 0

  if counter.total == K:
    max_squares := BT_2_inner(grid, occ ∪ root_rays, obs, min_pos, depth + 1)

  if to_be_placed == 0: return max_squares

  while true:
```

```
31      if gen[d].empty():
32        if ls[d] == N:
33          if d == 0:
34            break
35
36          grid.pop()
37          d -= 1
38          counter.raise(grid, hitss[d])
39        else:
40          ls[d] += 1
41          gen[d] = next_gen(ls[d], templ[d], occs[d])
42
43      pos := gen[d].pop_lsb()
44      if pos == BB_NONE || pos < min_pos || pos_rank(root, pos) <
            ranks[d]: continue
45      grid.push(pos % N, pos / N, ls[d])
46      p := (pos, ls[d])
47      occs[d + 1]  = next_occ(root, p, root_rays, occs[d])
48      obss[d + 1]  = obss[d] ∪ bb_sq(pos, ls[d])
49      templ[d + 1] = next_templ(root, p, occs[d + 1], templ[d])
50      hitss[d]     = counter.drop(grid, root, p)
51      d += 1
52
53      if counter.total + d == K:
54        max_squares = max(max_squares, BT_2_inner(grid, occs[d]
             ∪ templ[d], obss[d], min_pos, depth + 1))
55
56      if d >= to_be_placed:
57        grid.pop()
58        d -= 1
59        counter.raise(grid, hitss[d])
60      else:
61        ranks[d] = pos_rank(root, pos)
62        ls[d] = L_MIN
63        gen[d] = next_gen(ls[d], templ[d], occs[d])
64
65    return max_squares
```

By leveraging the fact that we cannot block vision from `root` to already satisfied squares, we do not have to place up to K new squares. Instead, we query `counter` to determine how many such fixed visible neighbors (line 9) and define the value `to_be_placed`, which is the maximum number of newly placed squares at a time.

`next_occ` prevents future placements from being in the line of sight of already placed squares, even if they do not entirely block the vision to that square. As a result, we must impose following constraint on the rank function `pos_rank`: Placements that are closer to the root must be chosen before other placements in their respective direction. This may seem arbitrary, but the construction pictured in Fig. A.2 may never occur without this constraint. If $r_{inner} = y \cdot N + x$, square A would be placed first. Since B is in the line of sight from the root to A, `next_occ` would forbid placing B after placing A. And since $r_{inner}(B) > r_{inner}(A)$, they also cannot be placed the other way around.

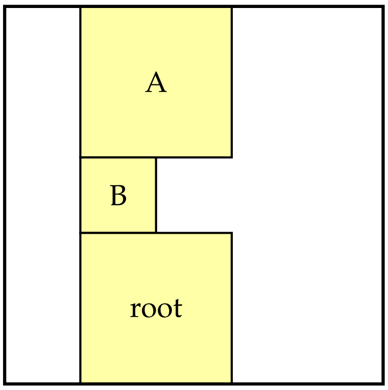
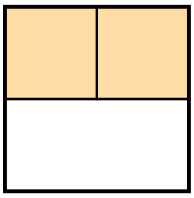


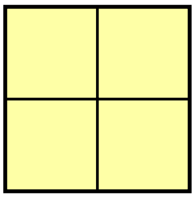
Figure A.2: Impossible construction with $r_{inner} = y \cdot N + x$

A.3 Solution grids for small K and N

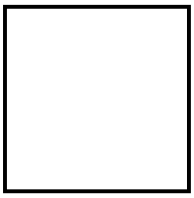
The following solution grids for $S_K(N)$, for $1 \leq K \leq 5$ and $2 \leq N \leq 11$ were found using the BT_2 algorithm.



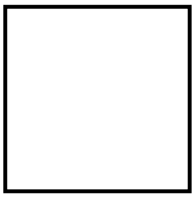
$S_1(2) = 2$



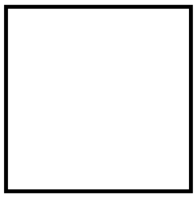
$S_2(2) = 4$



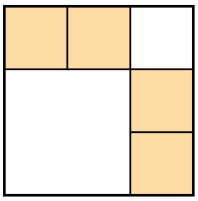
$S_3(2) = 0$



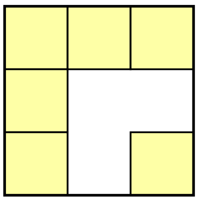
$S_4(2) = 0$



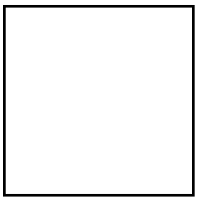
$S_5(2) = 0$



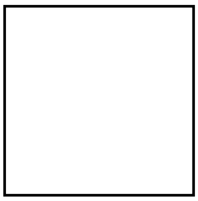
$S_1(3) = 4$



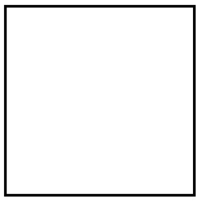
$S_2(3) = 6$



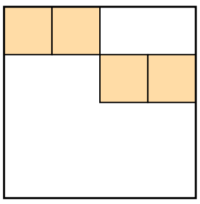
$S_3(3) = 0$



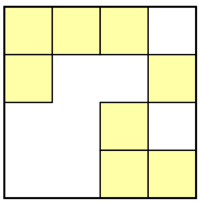
$S_4(3) = 0$



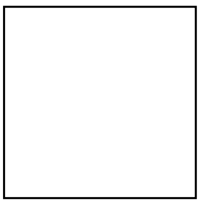
$S_5(3) = 0$



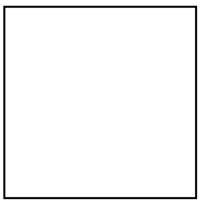
$S_1(4) = 4$



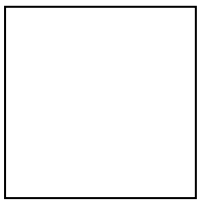
$S_2(4) = 8$



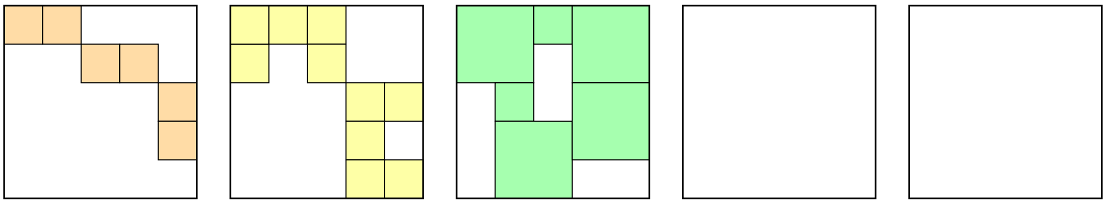
$S_3(4) = 0$



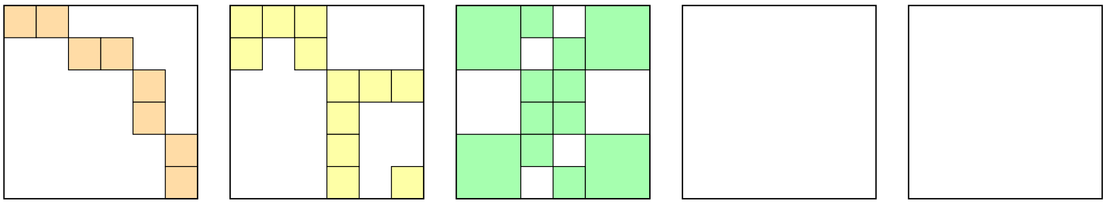
$S_4(4) = 0$



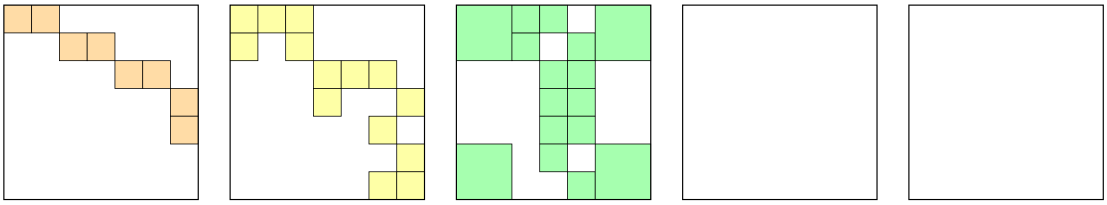
$S_5(4) = 0$



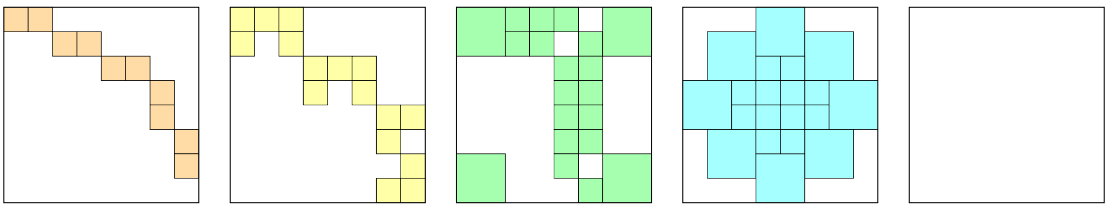
$S_1(5) = 6$ $S_2(5) = 10$ $S_3(5) = 6$ $S_4(5) = 0$ $S_5(5) = 0$



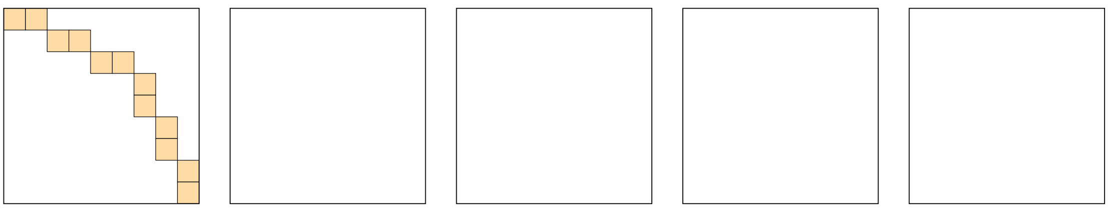
$S_1(6) = 8$ $S_2(6) = 12$ $S_3(6) = 12$ $S_4(6) = 0$ $S_5(6) = 0$



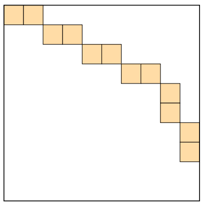
$S_1(7) = 8$ $S_2(7) = 14$ $S_3(7) = 16$ $S_4(7) = 0$ $S_5(7) = 0$



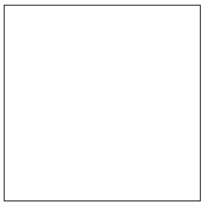
$S_1(8) = 10$ $S_2(8) = 16$ $S_3(8) = 20$ $S_4(8) = 20$ $S_5(8) = 0$



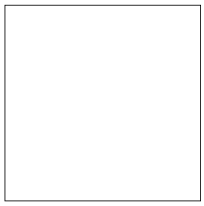
$S_1(9) = 12$ $S_2(9)$ N/A $S_3(9)$ N/A $S_4(9)$ N/A $S_5(9) = 0$



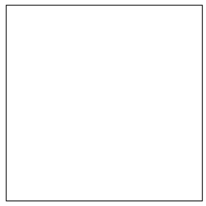
$S_1(10) = 12$



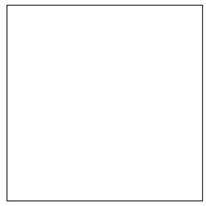
$S_2(10)$ N/A



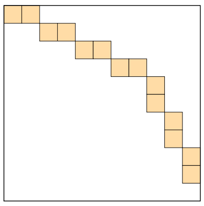
$S_3(10)$ N/A



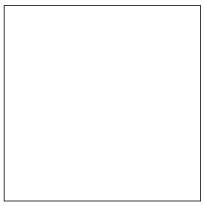
$S_4(10)$ N/A



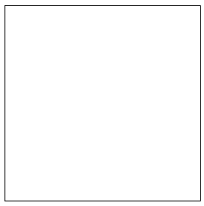
$S_5(10) = 0$



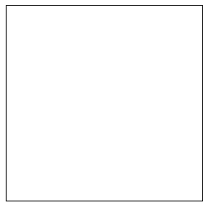
$S_1(11) = 14$



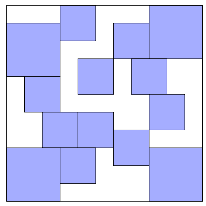
$S_2(11)$ N/A



$S_3(11)$ N/A



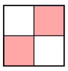
$S_4(11)$ N/A



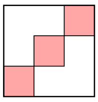
$S_5(11) = 14$

A.4 Construction Recipes

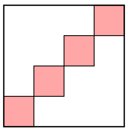
For $K \in \{0, 1, 2, 4\}$, solution grids can be easily obtained by continuing following solution sequences until the desired grid side length is reached. These solution grids were compiled by Friedman [13]:



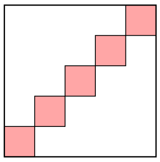
$K = 0, N = 2$



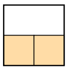
$K = 0, N = 3$



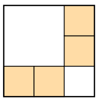
$K = 0, N = 4$



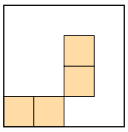
$K = 0, N = 5$



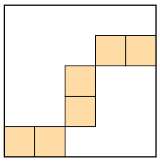
$K = 1, N = 2$



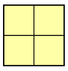
$K = 1, N = 3$



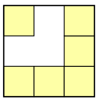
$K = 1, N = 4$



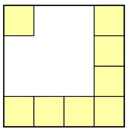
$K = 1, N = 5$



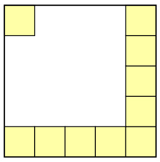
$K = 2, N = 2$



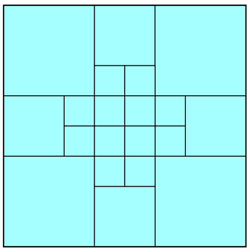
$K = 2, N = 3$



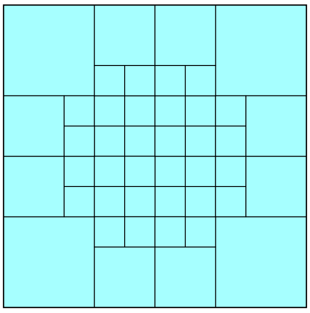
$K = 2, N = 4$



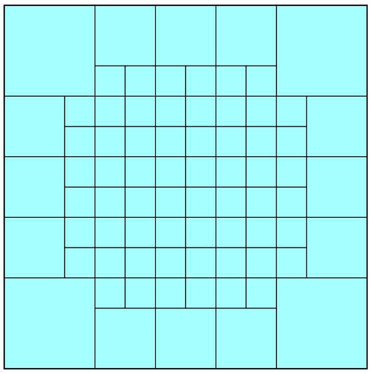
$K = 2, N = 5$



$K = 4, N = 8$



$K = 4, N = 10$



$K = 4, N = 12$