Seminar on Advanced Algorithms and Data Structures

On the Exact Complexity of Polyomino Packing

Report from Börge Scheel

1 Introduction

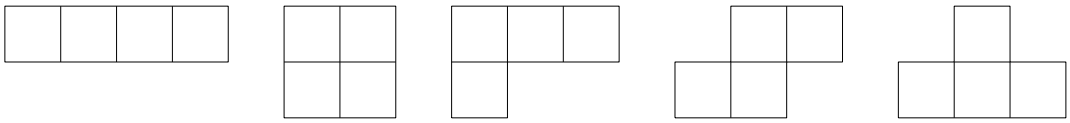
The goal of the report is to introduce the problem of polyomino packing and determine a tight bound (namely $2^{\Theta(n/\log n)}$, where n is the sum of the sizes of all polyominos that we get as input) on the runtime complexity under the assumption that the exponential time hypothesis holds. We will do this via reductions to other problems, which we already know. On the one hand, we will use a known algorithm for planar Subgraph Isomorphism, that has itself runtime $2^{O(n/\log n)}$ where n is the number of vertices in both input graphs, to construct an algorithm for polyomino packing. On the other hand, we will derive a lower bound on the runtime for the problem by reduction from 3-SAT. Note that this lower bound will only hold if the exponential time hypothesis holds.

2 Polyomino Packing

We will start by introducing the problem of polyomino packing and defining what a polyomino is:

Definition 1. *A **polyomino** is a geometric structure of 1×1-squares that are aligned edge to edge. It must be connected and has to have no holes. We define the size of a polyomino p ($|p|$) as the number of 1×1-squares it consists of.*

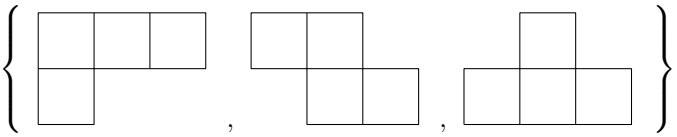
This rather formal definition will become clear if we take a look at examples of polyominoes. The following picture shows all polyominoes with size 4 (without identical ones with respect to rotation and reflection):

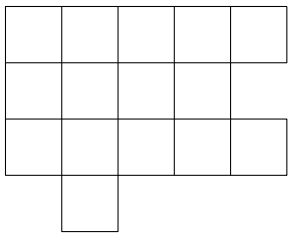


You probably know these polyominoes of size 4 also as Tetris-bricks. One could argue, that polyominoes are generalizations of Tetris-bricks and polyomino-packing is a generalization of Tetris. We now take a look at the formal definition:

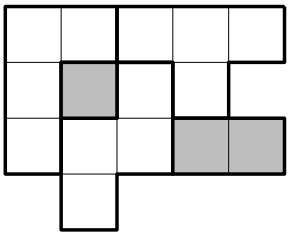
Definition 2. *For input (S, p_{target}), where S is a set of polyominoes of possibly different size and p_{target} is a polyomino, there exists a **polyomino packing** if and only if it is possible to position all polyominoes in S "on top of" p_{target}, such that each square of each polyomino in S lies exactly aligned on top of a square of the target polyomino. We don't allow that polyominoes in S overlap in the packing but we allow "holes" in the packing, i.e. it is possible that there are squares of the target polyomino, where no square of any polyomino in S is positioned on. It is allowed to rotate and/or flip the polyominoes. The problem of polyomino packing is to determine whether there exists such a packing or not. The input size n is defined as $n = |p_{target}| + \sum_{p \in S} |p|$.*

Again, the definition becomes more intuitive, if we take a look at an example.
The input is given as:

$S =$ 

$p_{target} =$ 

There exists a polyomino packing, namely:



Note that the "unused" squares of the target polyomino are marked grey and that the boundary lines of the polyominoes are drawn thicker.

There are multiple variations of the problem. We could restrict the positioning of the polyominoes to translation only (fixed polyomino packing) or only allow translation and rotation (one-sided polyomino packing). Furthermore, it is possible to require that the sum of all sizes of polyominoes in S is equal to the size of the target polyomino (also called exact polyomino packing). In the report, we will only look at the polyomino packing described in the definition (also called free polyomino packing). Note that the reductions and proofs that I will show in this report can be modified slightly, such that the bounds also hold for these variations.

3 Upper Bound

We will now construct an algorithm for polyomino packing which has runtime $2^{O(n/\log n)}$. This proves that the time complexity of polyomino packing is in $2^{O(n/\log n)}$. We will use a reduction to planar subgraph isomorphism, where it is known that there exists an algorithm (described in [2]) with runtime $2^{O(n/\log n)}$ where n is the number of vertices in both input graphs.

An algorithm for Subgraph isomorphism decides for two input graphs G and H whether H is isomorphic to a subgraph of G. This means intuitively, whether it is possible to remove edges and/or vertices from G to get a subgraph G_0 that „looks like H“, i.e. whether there's a bijection f between the vertices of G_0 and H such that $\{u, v\} \in E(G_0) \iff \{f(u), f(v)\} \in E(H)$ and $u \in V(G_0) \iff f(u) \in V(H)$.

A planar graph is a graph that can be embedded in the plane, i.e. it is possible to draw it on a paper without any edges intersecting. Planar subgraph isomorphism, therefore, is the subgraph isomorphism problem where G and H are restricted to be planar graphs.

3.1 Reduction

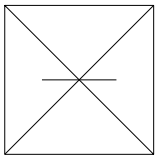
We will now show, how the reduction exactly works. Note that I modified the construction in the paper by choosing a slightly different structure. The following pseudocode is meant to give a high-level overview on how the reduction is structured. The exact construction will be described afterwards.

Algorithm 1: Polyomino packing (with planar subgraph-isomorphism)

Input : $(S = \{p_1, ..., p_k\}, p_{target})$
Output: true if there exists a polyomino packing, false otherwise

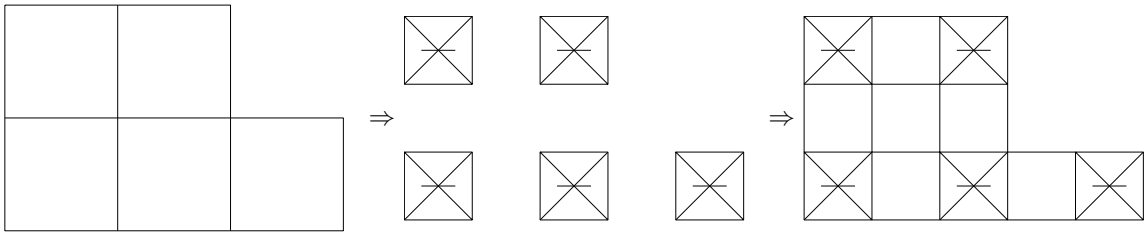
```
1 H ← (∅, ∅)
2 for i = 1, ..., k do
3 |   H ← H ∪ generateGraph(p_i)
4 end
5 G ← generateGraph(p_target)
6 return PlanarSubgraphIsomorphism(G, H)
```

We will now describe the *generateGraph* function for an arbitrary polyomino p. For each square in p, we insert the following structure:



Note that this graph is planar and that the vertex in the middle has degree 6.

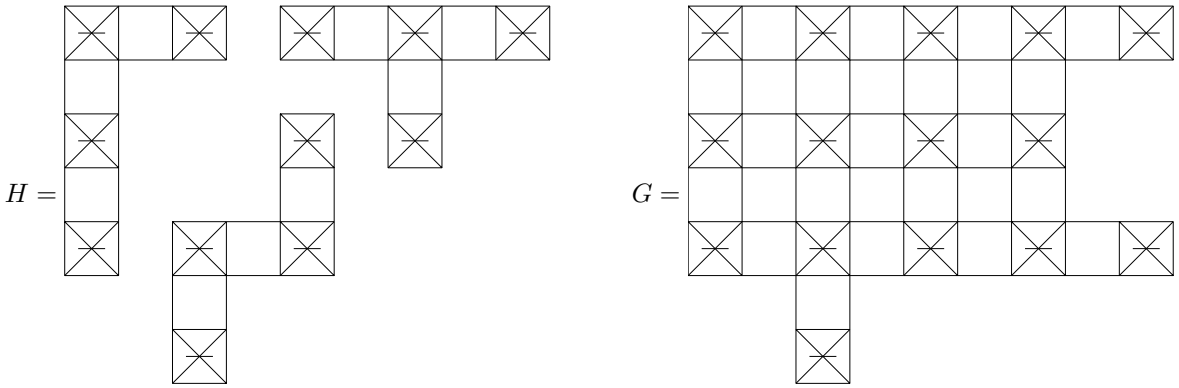
Next, we will connect the structures of squares, that are aligned edge to edge at the corners of the structures that correspond to the corners of the mutual edge. To visualize this, we will look at an example:



Note that the resulting graph is also planar and connected because the polyomino is connected itself. Furthermore, only the vertices in the middle of the structures have degree > 5.

We now combine the graphs that we generated for the polyominoes in S and name it H. H now consists of $|S|$ components that are planar. Therefore H is planar. We name the graph generated for the target polyomino G. We now call the algorithm for planar subgraph isomorphism and return the result, as described in Algorithm 1.

We will again look at an example. The input-set is the same as in the example in the introduction. Note that I draw the graphs in such a way, that the subgraph isomorphism and the original shapes of the polyominoes are clearly visible.



3.2 Runtime

We now want to analyze the runtime of the algorithm: For a polyomino p, $generateGraph(p)$ and the union can be executed in $O(|p|)$. Therefore lines 1 to 5 can be executed in $O(n)$. The number of vertices in G and H is linear in n. Therefore the time required for line 6 is in $2^{O(n/\log n)}$. Altogether the runtime of the algorithm is $2^{O(n/\log n)}$.

3.3 Correctness

Now we have to prove that the algorithm is correct, i.e

$$\exists \text{ polyomino packing for input } (S, p_{target}) \iff PlanarSubgraphIsomorphism(G, H) = true$$

($\Rightarrow$) Assume there is a polyomino packing for (S, p_{target}). We now consider the graph G generated by our algorithm from the target polyomino and remove all vertices of the structures that correspond to a square of the target polyomino that is not used by one of the polyominoes in S. Afterwards, we remove the connections between structures if the squares corresponding to the structures are used by different polyominoes. We name this graph G_0. We analyze, what is left if G_0: For each square of each polyomino in S there is a structure in the G_0. Furthermore, between the structures of the same polyomino, there are still the connections and structures of different polyominoes are not connected. It is now easy to see that G_0 is isomorphic to H generated by our algorithm. Therefore there exists a subgraph isomorphism and our algorithm returns true.

($\Leftarrow$) Assume, there is a subgraph isomorphism between G and H, which were generated by our algorithm. We note that in a subgraph the degree of a particular vertex is smaller or equal to the degree in the original graph. Because only the vertices in the middle of each structure have degree > 5, we know that each middle vertex in H has to map to a middle vertex in G. We see, that because the structure is the same for all squares both in G and H, each structure in H has to map to a structure in G. If structures were connected in H (i.e. the squares were neighbors in a polyomino), they also have to be connected in G. We can now identify a place were we can place the squares of the polyominoes: We place it on the corresponding square in the target polyomino where the corresponding structure of that square maps to. Because of the connections between structures, the squares of a polyomino are placed in the correct way (i.e. it stays the same polyomino). This is a valid placement because two structures from different squares of polyominoes in S cannot map to the same structure in the target polyomino.

4 Lower Bound

The Lower Bound for polyomino packing is shown by reduction from the 3-SAT-problem. We briefly revisit the 3-SAT problem:

Definition 3. *A **literal** is either a variable (positive form) or a negation of a variable (negative form). A **clause** is a disjunction of literals. A formula is in **3-CNF** if and only if it's a conjunction of clauses, which each consist of at most 3 literals. A formula is **satisfiable** if and only if there exists an assignment of variables to true or false, such that the formula is true. The problem of **3-SAT** is to determine whether a 3-CNF formula is satisfiable.*

The shown proof only works if the exponential time hypothesis holds. We revisit the exponential time hypothesis:

Definition 4. *The **Exponential Time Hypothesis** (short: **ETH**) states that 3-SAT cannot be solved in truly subexponential time (i.e. with runtime in $2^{o(n)}$) in the worst case. If the ETH holds, this directly implies $NP \neq P$, because we know that 3-SAT is in NP.*

In the reduction, we construct an algorithm for 3-SAT with help of a (theoretical) algorithm, that solves Polyomino packing. We show, that if the runtime complexity of polyomino packing would not be in $2^{\Omega(n/\log n)}$, we could solve 3-SAT in truly subexponential time what would contradict the ETH. Therefore if the ETH holds, Polyomino packing is in $2^{\Omega(n/\log n)}$.

4.1 Reduction

Again we will look at the pseudocode, that is meant to give you a high-level overview over the reduction. The exact definitions of the functions will be given afterwards.

Algorithm 2: 3-SAT (with Polyomino packing)

Input : n-variable 3-CNF formula Φ with m clauses
Output: true if Φ is satisfiable, false otherwise

```
1  Φ_1, ..., Φ_l ← sparsification(Φ)
2  for i = 1, ..., l do
3  |  Φ'_i ← reduceOccurences(Φ_i)
4  |  S_i ← generatePolyominoes(Φ'_i)
5  |  p_target ← generateTargetPolyomino(Φ'_i)
6  |  if PolyominoPacking(S_i, p_target) then
7  |  |  return true
8  |  end
9  end
10 return false
```

The sparsification of Φ is described in [3] and [4]. The algorithm uses backtracking to transform for any fixed $\epsilon > 0$ an arbitrary n-variable k-CNF formula into an equivalent disjunction of $2^{\epsilon n}$ n-variable k-CNF formulas that each have $O(n)$ clauses ($\Phi \equiv \Phi_1 \vee ... \vee \Phi_l$). We want to have this condition because we want to have an upper bound on the size of the polyominoes that we will construct. Because it is a disjunction, Φ is satisfied if and only if at least one of the $\Phi_1, ..., \Phi_l$ is satisfiable. Therefore we can just look at a single Φ_i to proof the correctness.

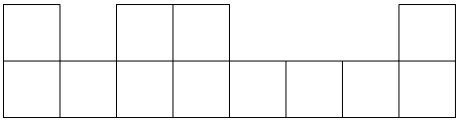
We will first look at the reduceOccurences function: This is a standard derivation that has the goal is to restrict how often each variable occurs in Φ'_i. We do this by replacing each variable x_j that occurs $k > 3$ times by one of k new variables $x_j^1, ..., x_j^k$ and add the clauses $(\neg x_j^1 \vee x_j^2) \wedge (\neg x_j^2 \vee x_j^3) \wedge ... \wedge (\neg x_j^{k-1} \vee x_j^k) \wedge (\neg x_j^k \vee x_j^1)$. The added clauses ensure that all $x_j^1, ..., x_j^k$ have the same value in a satisfying assignment. Furthermore, if some x_j occurs only positive or only negative, we remove the clauses, that contain x_j. If x_j occurs both positive and negative in the same clause, we remove this clause. It is obviously the case, that Φ'_i is satisfiable if and only if Φ_i is satisfiable. This step only increases (or decreases) the total number of variables and clauses linearly and has only linear runtime. Note that Φ'_i is a $n' \in O(n)$-variable 3-CNF formula with $m' \in O(n)$ clauses in with each variable occurs at most 3 times and at least once positive and once negative. Therefore each variable occurs at most twice positive and at most twice negative in Φ'_i. We will need this condition in the following construction.

4.2 Construction

The main idea of the construction is to look at Φ'_i not as a conjunction of clauses, but from the perspective of an assignment. We will construct for each variable x_j a variable-setting polyomino that must be placed in a packing into the so-called formula-encoding polyomino of x_j. This will correspond to assigning x_j to true or false. We also want to have a clause-checking polyomino for each clause. Each assignment of a variable can lead to clauses that are true. The idea is that the placement of the variable-setting polyomino of x_j blocks certain places where clause-checking polyominoes could go and therefore only allow the placement of the clause-checking polyominoes which correspond to clauses, which are true due to the chosen assignment of x_j, in the formula-encoding polyomino of x_j. Therefore we need one polyomino per variable and one polyomino per clause and the target polyomino that will consist of one polyomino per variable.

Note that I modified the construction and the proof from the paper.

We will use a special type of polyominoes that we can describe as a bitstring: Given a bitstring of length h, its corresponding polyomino has two rows: The top one has a square on the j-th position if and only if the bitstring has a 1 in the j-th position. The bottom row consists of h consecutive squares. Therefore for every bitstring, the corresponding structure is a valid polyomino and its size is smaller or equal to 2h. Note that this translation is injective. We will look at an example:

10110001 ⇒ 

We concatenate two polyominoes corresponding to bitstrings b_1 and b_2 by taking the polyomino corresponding to the concatenation of the two bitstrings b_1 and b_2. We say that a polyomino A fits into a polyomino B if and only if $polyominoPacking(\{A\}, B) = true$.

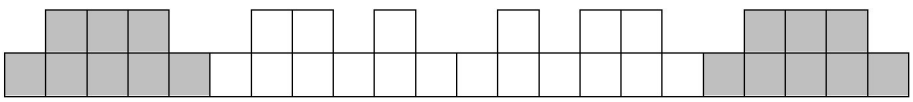
We want to construct the polyominoes in such a way that we can abstract it as a concatenation of building blocks, which are defined by bitstrings. Furthermore, we want to restrict the placement of the polyominoes in the packing to the placement of building blocks into building blocks. Therefore we want to have the following properties:

- The building blocks should all have the same "width"
  → The bitstrings of the building blocks should have the same length.
- We don't want to allow that the building blocks can overlap with multiple building blocks of the target polyomino
  → special structure at the start and at the end of each building block
- We want that each clause/variable is identified uniquely (with rotation and flipping) by one building block polyomino
  → Use unique and palindromic bitstrings

We label the variables and clauses uniquely with the numbers 1 to $(n' + m')$ and associate a bitstring with each of these identifiers. To ensure that building blocks cannot overlap with multiple building blocks of the target polyomino, we choose as a identifying feature of the special structure 3 consecutive ones. Therefore we have to ensure that there are 3 consecutive ones in concatenations of building blocks only at the positions of the special structures. To prevent that the concatenation of a special structure with an arbitrary bitstring leads to more than 3 consecutive ones at the boundary, we choose as special structure the bitstring 01110. To ensure that there are not 3 consecutive ones in the bitstrings corresponding to the identifiers, we restrict the maximum number of consecutive ones to 2. We do this by taking the $\lceil \log(n' + m') \rceil$-bit binary representation of that number (unique and same length), replacing every 0 by 01 and every 1 by 10 (thus ensuring that the bitstring contains at most 2 consecutive ones and same length) and then appending a reversed copy of the bitstring to itself (making it palindromic). Afterwards we add the special structure 01110 at the start and at the end. The bitstring of the building block has now length $10 + 4\lceil \log(n' + m') \rceil$. Now, the only places where 3 consecutive ones are in the concatenation of bitstrings of building blocks are in the special structures at the start and at the end of each building block.

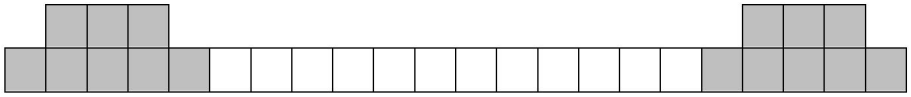
We take a look at an example ($\lceil \log(n' + m') \rceil = 3$):

	identifier 3
⇒ (binary representation)	0 1 1
⇒ (0→ 01, 1→ 10)	011010
⇒ (palindromic)	011010010110
⇒ (add special structure)	0111001101001011001110
⇒ (transform to polyomino)	



Note: In the examples, the special structures are always gray for illustration purposes.

We define the so-called wildcard polyomino as the polyomino corresponding to the concatenation of 01110, the bitstring with $4\lceil \log(n'+m') \rceil$ zeros and 01110. It thus has the same length as all other building blocks and is different from all other building blocks. Note that the special structure only occurs at the start and at the end. For the same size as in the last example, the wildcard polyomino looks the following way:



By construction, no polyomino corresponding to a clause/variable fits into a polyomino corresponding to another clause/variable. The wildcard polyomino fits into all polyominoes corresponding to a clause/variable. Only the wildcard polyomino fits into the wildcard polyomino.

We now generate the three polyomino-types described at the beginning. This corresponds to line 4 and 5 in the algorithm.

To give an visualization we will simultaneously look at the following example:

$$\Phi'_i \equiv (x_1 \vee x_2) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$$

We label the variables/clauses in the following way:
$x_1 \to 1$, $x_2 \to 2$, $(x_1 \vee x_2) \to 3$, $(\neg x_1 \vee x_2) \to 4$, $(\neg x_1 \vee \neg x_2) \to 5$.

Let X1, X2, C3, C4, C5 denote the corresponding building blocks (polyominoes).
Let WP denote the wildcard polyomino.

- **clause-checking polyominoes**: For each clause, we take the building block of that clause. We already know that this building block only fits into itself.
  Therefore in our example, our clause-checking polyominoes are C3, C4 and C5.
- **formula-encoding polyominoes**: For each variable x_j we construct the formula-encoding polyomino by concatenation of 7 polyominoes: We start with the building block corresponding to x_j. Next, for each time (at most two) x_j occurs positively in a clause, we take a building block corresponding to that clause. If x_j occurs only once in positive form, then we take (for padding) a copy of the wildcard polyomino, because we don't want that a clause-checking polyomino is placed there. Then, we take another copy of the building block for x_j. Next, we take the polyominoes corresponding to clauses in which x_j occurs negative. Again, we add the wildcard polyomino if x_j only occurs negated once. Finally, we take another copy of the building block corresponding to x_j.
  Therefore in our example, our formula-encoding polyominoes are X1 C3 WP X1 C4 C5 X1 and X2 C3 C4 X2 C5 WP X2.
- **variable-setting polyomino**: We want that the variable-setting polyomino for x_j can only be placed into exactly two positions in the formula-encoding polyomino of x_j and either blocks all the building blocks in the formula-encoding polyomino of x_j of clauses where x_j occurs positive or all where x_j occurs negated. For each variable x_j the variable-setting polyomino is formed by concatenating in the following order: The polyomino for the variable, 2 copies of the wildcard polyomino and another copy of the polyomino for x_j.
  In our example, our variable-setting polyominoes are X1 WP WP X1 and X2 WP WP X2.
  Note that X1 WP WP X1 only fits into the following two positions in the formula-encoding polyomino of x_1:

p_{target}:	X1	C3	WP	X1	C4	C5	X1
placement:	X1	WP	WP	X1			

  (this corresponds to setting $x_1 = false$ and blocks the building blocks of the clauses, where x_1 occurs positive) or

p_{target}:	X1	C3	WP	X1	C4	C5	X1
placement:				X1	WP	WP	X1

  (this corresponds to setting $x_1 = true$ and blocks the building blocks of the clauses, where x_1 occurs negative). The same property holds for X2 WP WP X2.

As described in algorithm 2, we define the target Polyomino as the concatenation of all formula-encoding polyominoes and the input set S as the set of all formula-encoding polyominoes and clause-checking polyominoes.
Therefore in our example, the target polyomino has the following form:

X1 C3 WP X1 C4 C5 X1 X2 C3 C4 X2 C5 WP X2

Our input set S is {X1 WP WP X1, X2 WP WP X2, C3, C4, C5}.

We now construct the following packing:

p_{target}:	X1	C3	WP	X1	C4	C5	X1	X2	C3	C4	X2	C5	WP	X2
packing:	X1	WP	WP	X1	C4	C5			C3		X2	WP	WP	X2

The placement of the variable-setting polyominoes correspond to $f(x_1) = false$ and $f(x_2) = true$.

A detailed visualization can be found in the appendix at the end of the report.

4.3 Correctness

We now have to prove the correctness. It is sufficient to prove that (for all i and Φ):

$$\Phi_i' \text{ is satisfiable } \iff PolyominoPacking(S, p_{target}) = true$$

($\Rightarrow$). Assume Φ_i' is satisfiable. Then there is a satisfying assignment f that allocates each variable x_j to a boolean value $f(x_j)$. If $f(x_j) = true$, we place the variable-setting polyomino for x_j aligned to the right side of the formula-encoding polyomino for x_j. Otherwise, we place it aligned to the left side of the formula-encoding polyomino for x_j. This is possible because the wildcard polyomino fits into every other building block and the building block of x_j fits into itself. The variable-setting polyominos trivially don't overlap.
Because f is a satisfying assignment, each clause c_j must be true. Therefore there is at least one literal in c_j that is true. Let this literal belong to some x_k. If it is $\neg x_k$, the building block in the fifth or sixth position in the formula encoding polyomino for x_k has to be (by construction) the building block corresponding to c_j. We place the clause checking polyomino for c_j there. This is possible because if $\neg x_k$ is true, $f(x_k) = false$ and therefore it doesn't overlap with the placement of the variable-setting polyomino of x_k. If the literal occurs in positive form we place it on the second or third position, which is possible with the same arguments. The clause-checking polyominoes don't overlap because there is exactly one per clause and each of them only fits into itself.

($\Leftarrow$). Assume there is a polyomino packing. We first observe, that because of the special structure 01110 a placed building block cannot overlap with multiple building blocks of the target polyomino. We also observe, that the variable setting polyomino for x_j only can be placed at two different locations: Either aligned to the left or the right side of the formula-encoding polyomino for x_j. In the first case, we set the allocation f of x_j to false, otherwise to true. We now show that f is a satisfying assignment, i.e. that each clause c_j must be true. We now observe, that the clause-checking polyomino only fits into itself. Therefore there must be a x_k, such that the clause-checking polyomino of c_j is placed into the formula-encoding polyomino of x_k. Because it cannot overlap with the variable-setting polyomino of x_k we know that if $f(x_k) = true$ x_k occurs positive in c_j respectively if $f(x_k) = false$ x_k occurs negative in c_j. Therefore each clause c_j is true under f, what implies that Φ_i' is satisfiable.

4.4 Runtime

We now take a look at the runtime of our algorithm:

By [3] and [4] we know that for any fixed $\epsilon > 0$ there is a sparsification algorithm such that $l \leq 2^{\epsilon n}$ and that Line 1 can be executed in $O(n^t \cdot 2^{\epsilon n})$ for some constant t. The papers [3] and [4] also states (because we can choose $\epsilon > 0$ arbitrarily small), that if we can solve the 3-SAT problem for a O(n)-variable 3-CNF formula with O(n) clauses (i.e. for Φ_i) in subexponential time, we also can solve 3-SAT in subexponential time (using the sparsification algorithm for a small enough ϵ and the for-loop). Therefore we only have to show that the runtime of lines 3 to 8 is subexponential.

Lines 3 to 5 can be done in time $O(n)$. The input size k to the polyomino packing algorithm can be bound by $O(n) \cdot (maxsize(buildingblock))$. Because the number of clauses and the number of variables is in $O(n)$, the size of a building block is in $O(\log n)$ and therefore k is in $O(n \log n)$.

Let $g(k)$ denote the runtime of the polyomino packing algorithm. Then the runtime of lines 3 to 8 is $O(n) + g(n \log n)$.

Let's assume g is in $2^{o(k/\log k)}$ for input size k. Then we also know that the runtime of the polyomino packing algorithm is in $2^{o(n \log n/\log(n \log n))} \subseteq 2^{o(n \log n/\log n)} \subseteq 2^{o(n)}$. That means that Line 6 also has subexponential runtime.

Putting all together we get that the runtime of lines 3 to 8 is $O(n) + 2^{o(n)} \subseteq 2^{o(n)}$. Therefore we could solve 3-SAT in subexponential time if polyomino packing would be in $2^{o(k/\log k)}$ for input size k. If we assume that the exponential time hypothesis holds, this is a contradiction, what implies that polyomino packing for input size k is in $2^{\Omega(k/\log k)}$ if the ETH holds.

4.5 Lower bounds in restricted cases

In the reduction our input size k for the polyomino packing algorithm is equal to $n \log n$, where n is the number of variables in Φ. To distinguish the input sizes I will always use k and n in that way.

We observe that the target polyomino in our construction always fits into a $2 \times O(n \log n) = 2 \times O(k)$ (n formula encoding polyominoes with size $O(\log n)$) rectangular polyomino and that each polyomino in S always fits into a $2 \times O(n \log n) = 2 \times O(k)$ rectangular polyomino. That means if the ETH holds even this subproblem is in $2^{\Omega(k/logk)}$.

We also could modify our algorithm in a way, that the target polyomino is a $3 \times O(n \log n) = 3 \times O(k)$ rectangular, by adding the polyominos corresponding to the negated bitstrings of the formula-encoding polyomino to the input set S and modifying the special structure slightly. So even this special case is in $2^{\Omega(k/logk)}$.

The paper also states, that there are subproblems, that can be done in strongly subexponential time, even if the ETH holds. For example, if the target polyomino is a $2 \times O(k)$ rectangle, we can find an algorithm with subexponential runtime $2^{O(k^{3/4} \log k)} \subseteq 2^{o(k)}$. Or if the polyominoes in S and the target polyomino are all rectangular, we can find an algorithm with subexponential runtime $2^{O(\sqrt{k} \log k)} \subseteq 2^{o(k)}$.

5 References

[1] H. L. Bodlaender and T. C. van der Zanden, "On the exact complexity of polyomino packing," 2018.

[2] H. L. Bodlaender, J. Nederlof, and T. C. van der Zanden, "Subexponential time algorithms for embedding h-minor free graphs," pp. 9:1–9:14. [Online]. Available: https://pure.tue.nl/ws/portalfiles/portal/53480205/subex.pdf

[3] R. P. Russell Impagliazzo and F. Zane, "Which problems have strongly exponential complexity?" p. 63:512–530, 2001. [Online]. Available: https://cseweb.ucsd.edu/~russell/ipz.pdf

[4] D. Scheder, "Sparsification and eth." [Online]. Available: http://users-cs.au.dk/dscheder/SAT2012/sparsification.pdf

6 Appendix

Visualization of the example on page 6. The target polyomino is always in the top row and the packed polyominos are in the bottom row. The target polyomino is one connected structure but for illustration purposes drawn as multiple segments.