Practical Course Report

Computing Tangles Using a SAT Solver

Vasil Alistarov

Date of Submission: October 3, 2022
Advisors: Prof. Dr. Alexander Wolff
Johannes Zink, M. Sc.

Julius-Maximilians-Universität Würzburg
Lehrstuhl für Informatik I
Algorithmen und Komplexität

1 Introduction

The subject of this practical course is the ListFeasibility problem. In ListFeasibility we are given a set of n y-monotone curves called *wires* as well as a (multi)set of *swaps* between them; one must then test whether the given swaps (and only the given swaps) can indeed be applied (step-wise) onto the wires while swapping only adjacent wires in every step. The order of the wires on the layers created this way is known as a *tangle*.

Minimising the number of steps needed to apply all swaps was first done algorithmically by Olszewski et al. [OMK$^+$18]. However, the provided algorithm was not efficient; in fact, in 2019, Firman et al. [FKR$^+$19] showed that the optimisation version of the problem (designated as Tangle-Height Minimization) is NP-hard by reducing from 3-Partition. One year later, the same research group discussed [FFK$^+$23] the decision version of the problem – ListFeasibility – and concluded by reducing from Positive NAE 3-SAT Diff that it is NP-hard as well. Note that in Positive NAE 3-SAT Diff, one is given a boolean formula with three literals per clause such that those literals all represent different variables. One must then decide whether the formula can be satisfied while not assigning the same truth value to all literals in a clause.

In this Practical Course, we will analyse the connection between ListFeasibility and the standard Satisfiability problem (SAT) as well, however in the other direction: Instead of reducing from a SAT problem, we will design a formulation of ListFeasibility as a SAT instance. Note that our aim is indeed the classical SAT problem and not a version of it with tighter constraints. We will examine the total size of the formulation and prove its correctness. Furthermore, we will implement our model in the Scala programming language and apply the `Sat4j` solver on a set of ListFeasibility instances as a proof-of-concept. The reasoning behind the use of SAT is two-fold: on the one hand, it has been established as a good problem to model instances of other problem as, and on the other hand, we can use this for comparison with some already existing implementations based on other approaches.

1.1 Terms and Definitions

In this subsection, we define some terminology to use across our report. A large part of the notation is taken from [FKR$^+$19]. As already briefly mentioned above, the problem we are working on revolves around a set of y-monotone curves which we call *wires*. We enumerate the wires $1 \le i \le n$ and refer to them by their number, called *index*. The wires are initially in a specific *configuration* which is a permutation on the set $\{1, \dots, n\}$. They start in the configuration $(1, \dots, n)$ but later on they may be arranged in different configurations. For this, we define the set S_n as all different permutations of the set $\{1, \dots, n\}$. If in some configuration, wire i is to the left of wire j, we write $i < j$. Also, for any positive integer k, we define $[k] = \{1, \dots, k\}$.

Additionally, we are given a symmetric $n \times n$ matrix L such that all entries lie in $\mathbb{N}_0$ and such that it has a zero diagonal. This matrix is known as a *(swap) list of order n*. We denote the entries of the matrix as l_{ij} for $i, j \in [n], i \neq j$. We define $L(i)$ as the multiset of swaps of wire i, and also $L(i, j) = L(i) \cap L(j)$. Furthermore, the whole matrix can

be translated into an equivalent (multi)set of tuples $L' = \bigcup_{i,j \in [n]} L(i,j)$ where a single tuple is equivalent to a swap. A swap (i,j) creates a new configuration of the wires where the order of the wires i and j is changed compared to the previous configuration and the rest of the configuration remains unchanged. Applying a swap onto two wires in a configuration where they are not adjacent is forbidden. For instance, the swap $(1,2)$ is allowed in the configuration $(4,1,2,3)$ but not in $(4,1,3,2)$. We define m as the length of the list of swaps L as $\sum_{i<j} l_{ij}$.

We call a list of configurations (or *layers*) $\langle \pi_1, \pi_2, ...\pi_h \rangle$ a *tangle* T of height h if for each two adjacent configurations π_i, π_j we can create π_j from π_i by legally applying a swap. Also, we call the application of a swap between two layers a *step*. A tangle T *realises* a swap list L if and only if the configurations of T can be achieved by applying all swaps of L exactly once. Also, we call a list *feasible* if there exists a tangle that realises that list starting from the configuration $(1, \ldots, n)$.

The problem of minimising the number of steps needed to exhaust all swaps, when we can apply multiple swaps at once, is known as Tangle-Height Minimization as mentioned above. On the other hand, the problem of testing the feasibility of a given list is called ListFeasibility. This is the problem we are working on in this practical course. In particular, we design a formulation of ListFeasibility as a SAT problem.

The Boolean Satisfiability Problem, or SAT, is one of the oldest problems in theoretical computer science and also the first one that was proven to be NP-complete. In SAT, one is given a boolean formula built from a set of n boolean variables $x_i, 1 \leq i \leq n$, as well as the logical operators AND ($\wedge$), OR ($\vee$), NOT ($\neg$) and some number of parentheses. A formula F is said to be *satisfiable* exactly when there exists an assignment of the boolean values $true$ and $false$ to the variables such that the whole formula evaluates to $true$. SAT itself consists of deciding whether or not such an assignment exists. Oftentimes the formula is given with some particular structure. A (negated) variable or a disjunction of multiple (negated) variables is called a *clause*; the formula itself may for example be presented in *Conjunctive Normal Form* (CNF) – either as a single clause or as a conjunction of multiple clauses.

2 Formulation

When modeling a ListFeasibility instance as a SAT instance, we need to represent both the values (wires and swaps) and the rules (e.g., when can a swap be performed legally). We identify a ListFeasibility instance by its swap list L. At the end, we aim for a boolean formula F for which holds:

$$F \text{ satisfiable} \leftrightarrow L \text{ feasible}$$

In this section, we present a formulation that encodes an $n \times n$ swap list L with length m as a SAT instance. Note that the number of wires can be determined by simply taking the dimension size of the swap list; hence, it is irrelevant whether it is explicitly given or not. Moreover, from a valid SAT solution, one is even able to determine the exact order in which the swaps were applied. We present all clauses in an intuitive, comprehensible

way with respect to the reasoning behind them; however, we also describe them in CNF for convenience during the implementation phase. The latter representation is given in the Appendix, in Subsection 7. The transformation between the two representations is straightforward and therefore omitted.

Note also that our construction assumes there is exactly one swap between two layers, so it will not necessarily be minimal. This does not contradict our previous claims since if a list is feasible in m steps, then it is also feasible in total. The opposite direction holds as well: Suppose a feasible list needed more than m steps. This would imply through the pigeonhole principle that there exist some neighbouring layers between which no swap was applied; hence, one can skip those steps and receive a tangle in at most $m+1$ layers. Finally, we assume that in the beginning, the wires are ordered corresponding to the natural ordering of their indices.

Describing the configurations. To describe a configuration c and ultimately the solution, we create variables

$$x^r_{i,j} \qquad \forall r \in [m+1] \; \forall i,j \in [n] : \; i \neq j \, . \tag{1}$$

Having a variable $x^r_{i,j}$ set to *true* is interpreted as having wire i to the left of wire j in layer r. Since we assume that the swaps are applied one by one, there will be $m+1$ layers. To handle some edge cases and avoid illegal assignments, we implement the following clauses to model the rules of transitivity:

$$x^r_{i,j} \wedge x^r_{j,k} \Rightarrow x^r_{i,k} \qquad \forall r \in [m+1] \; \forall i,j,k \in [n] : \; i \neq j \neq k \tag{2}$$

and antisymmetry:

$$x^r_{i,j} \Leftrightarrow \neg x^r_{j,i} \qquad \forall r \in [m+1] \; \forall i,j \in [n] : \; i \neq j \tag{3}$$

Those two rules assert that no invalid ordering within a layer can take place. Moreover, as mentioned above, we must make sure that in layer 1 the natural ordering of the wires $(1 < 2 \cdots < n)$ holds.

$$x^1_{i,i+1} \qquad \forall i \in [m-1] \tag{4}$$

Setting the order of wire i and its successor $i+1$ only is sufficient for this since the transitivity rule will propagate the ordering to the rest of the wires. For instance, given $1 < 2$ and $2 < 3$, $1 < 3$ will follow.

Describing the swaps. The second crucial part of the problem definition are the swaps. In order to describe those, we introduce variables to indicate when they occur. Let L' be the list of swaps L represented not as a matrix, but rather as a (multi)set of tuples

as described in Subsection 1.1. This way we can easily iterate over the tuples. Then, we introduce the variables

$$y_s^r \qquad \forall s \in L' \; \forall r \in [m] \tag{5}$$

Setting a variable y_s^r with $s = (i,j)$ to *true* means that wires i and j are swapped in the step between layers r and $r+1$. As per the definition of our problem, we must now ensure by another set of clauses that (i) all swaps are taken exactly once, (ii) exactly one swap is used between every neighbouring pair of layers and (iii) applying a swap changes the value of the corresponding x variables after the swap has taken place. For (i), it is sufficient to first assert that we cannot take a swap twice – one cannot use the same swap more than once *without* using it twice.

$$\neg(y_s^r \wedge y_s^{r'}) \qquad \forall s \in L' \; \forall r, r' \in [m] : \; r < r' \tag{6}$$

Then, we also must take each swap at least once:

$$\bigvee_{r=1}^{m} y_s^r \qquad \forall s \in L' \tag{7}$$

Ensuring that each swap is taken at least once but less than twice is trivially equivalent to saying that it is applied exactly once. For (ii) we can simply add a clause asserting that some swap has been used in each step. Note that if each of the m swaps is taken exactly once and after each of the m layers (i.e., excluding the final layer) there is a swap, then per pigeonhole principle there is *exactly* one swap in every step.

$$\bigvee_{s \in L'} y_s^r \qquad \forall r \in [m] \tag{8}$$

Finally, to assert (iii), if in some step we swap the wires i and j, then their corresponding x-variables must have different values: if i was to the left of j before the swap, it will be to its right afterwards. For this, we introduce the following clauses:

$$y_s^r \Rightarrow (x_{i,j}^r \neq x_{i,j}^{r+1}) \qquad \forall r \in [m] \; \forall i,j \in [n] \; \forall s \in L(i,j) : \quad i \neq j \tag{9}$$

The opposite direction must hold as well. However, note that simply reversing the implication will be incorrect in the case of non-simple swap lists, that is, swap lists that contain any $l_{ij} > 1$. This is due to the fact that a $y_s^r, s = (i,j)$ can potentially be ambiguous if the wires i and j are swapped multiple times. Instead, in this case we must ensure that having them swapped between layers r and $r+1$ implies that *any* swap of those two wires has been used in that step:

$$(x_{i,j}^r \neq x_{i,j}^{r+1}) \Rightarrow \bigvee_{s \in L(i,j)} y_s^r \qquad \forall r \in [m] \; \forall i,j \in [n] : \quad i \neq j \tag{10}$$

Combining those rules together with the antisymmetry and the transitivity imposed on the x-variables is sufficient to ensure that we only transition between valid states, and that the transitions themselves are also performed correctly.

2.1 Size

Next, we show that the size of our formulation is polynomial in the number of wires and swaps. In order to do this we analyse the number of variables and clauses that are created in the process, i.e., in the equations (1) to (9). In particular, we will do so in terms of the number of wires n and the number of swaps m as those measures are independent from each other. As mentioned before, SAT problems are usually given in CNF notation; hence, we will consider the final CNF forms of those equations that is given in Appendix 7.

1. There are in total $(m+1) \cdot n \cdot (n-1)$ many $x_{i,j}^r$ variables. Asymptotically this is in $\Theta(m \cdot n^2)$.
2. To assert transitivity, we need $(m+1) \cdot n \cdot (n-1) \cdot (n-2)$ many clauses, so in total $\Theta(m \cdot n^3)$ many. Each clause has constant length of three.
3. For antisymmetry, we require $2 \cdot (m+1) \cdot n \cdot (n-1)$ clauses since the equivalence can be expressed through 2 implications. Each clause holds two variables. In total, this leads to $\Theta(m \cdot n^2)$ clauses.
4. Setting the initial permutation of the wires can be done in $m-1 \in \Theta(m)$ clauses consisting of a single variable.

For the swaps, we work with the (multi)set L'. Since L' is just another representation of L, it holds that the size of L' is also m.

5. The total number of y_s^r variables is $m \cdot |L'| = m^2 \in \Theta(m^2)$.
6. Asserting that every swap has been taken at most once requires $m \cdot \binom{m}{2} \in \Theta(m^3)$ many clauses with two variables each.
7. On the other hand, making sure that each swap has been used at least once takes m clauses with m many variables per clause which corresponds to a size in $\Theta(m^2)$.
8. The latter holds for Equation (8) as well.
9. We require $2 \cdot m \cdot m \in \Theta(m^2)$ clauses with constant length of three to assert that each swap has been applied. Note that the factor 2 arises from to the deconstruction of the inequivalence, similarly to Equation (3).
10. Finally, the number of clauses asserting that wires are swapped only if an appropriate swap has been applied is $m \cdot n \cdot (n-1) \in \Theta(mn^2)$ clauses with each having a length of at most $m+2$, which corresponds to a total size in $\Theta(m^2 n^2)$.

In total, this yields $m^2 + (m+1) \cdot n \cdot (n-1) \in \Theta(m^2 + mn^2)$ many variables. The number of clauses is in $\Theta(m^3 + mn^3)$ with each clause containing $\mathcal{O}(m)$ many variables. Thus any ListFeasibility instance can be formulated as a SAT one by using only a polynomial number of variables and clauses and the total size is in $\Theta(m^3 + m^2 n^2 + mn^3)$.

3 Implementation

We implement our ideas as a proof of concept in order to show their feasibility. For this, we write a script in the Scala programming language that is able to read swap lists as input, then converts the given swap list to a set of clauses in Conjunctive Normal Form and runs the result by a SAT solver. If the SAT instance is satisfiable, the corresponding tangle is created as a SVG file. The code is publicly available on GitHub.

3.1 Input Reading

Our script is able to create a swap list from a given swap matrix. In particular, it is able to read a JSON file containing the swap matrix as an $n \times n$ array of integers. For this, we use uJson, an efficient JSON library for Scala by Li Haoyi[1]. It can be downloaded as part of the uPickle library from GitHub. From each JSON input, we create a `SwapList` instance that exposes the number of wires and swaps as well as a representation of the swap matrix as a sequence.

3.2 Conversion to DIMACS

In most modern SAT solvers, such as `minisat` and `Sat4j`, the input CNF formula is passed in the DIMACS format. Here, we will briefly explain the details of that format and also give a way to represent our variables in it.

Each DIMACS file begins, aside from comments (lines beginning with `c`), with a line specifying that the formula is in CNF, as well as the number of variables and clauses in it. Recall that a formula in CNF is essentially the conjunction of multiple clauses where each clause is the disjunction of multiple (negated) variables. In DIMACS, each variable is represented as a natural number; this number is positive in the cases where the variable occurs non-negated, and negative otherwise. Aside from the "header", every following line represents one single clause. A single clause is then given as one or many whitespace-separated (positive or negative) integers for the variables, as described above, and ending with 0. Consider the following example:

$$(x_1 \vee x_3 \vee \neg x_4) \wedge (x_4) \wedge (x_2 \vee \neg x_3)$$

In this case, we are given a total of 4 variables and 3 clauses. The corresponding representation in the DIMACS format will look like this:

```
c
c optional comments
c
p cnf 4 3
1  3 -4 0
4  0
2 -3  0
```

[1] See Li Haoyi's website.

We have already counted the number of variables in our formulation in Subsection 2.1. Now it is necessary to find an injective function mapping a variable to an integer (for the purposes of DIMACS) and vice versa. For this, we make two crucial observations:

The x variables can be uniquely identified through their three indices – i, j, r. In particular, recall that $i \in [n], j \in [n], r \in [m+1]$. Therefore the set $\{1, \ldots, (m+1) \cdot n^2\}$ is *sufficiently large* for representing this type of variables. Note that this size is not *necessary* as there are, for instance, no $x_{i,i}$ variables (a wire being to the left of itself). However, this offset of size $(m+1) \cdot n$ is negligible and can be ignored for implementation simplicity[2]; the integers corresponding to those variables will simply remain unassigned. This allows us to apply the following mapping function:

$$\langle i, j, r \rangle = i + (j-1) \cdot n + (r-1) \cdot n^2 \tag{11}$$

For visualisation purposes, the variables can be considered as points with specific coordinates in a three-dimensional space with two of the axes running up to n and the last one running up to $m + 1$. The given function is thus merely enumerating those points.

As mentioned earlier, we also need to be able to determine the unique variable that has been mapped to a given integer z. Let $\lfloor \ \rfloor$ denote the integer division, and mod – the modulo division operation. Then:

- $r = \lfloor z/n^2 \rfloor + 1$
- $j = \lfloor (z \mod n^2) \ / \ n \rfloor + 1$
- $i = (z \mod n^2) \mod n$.

The y variables are similar in that they also have a layer index r; however, unlike the x variables, in general they cannot be uniquely identified through the two wire indices in the swap s they represent. This is due to the fact that there may be multiple swaps between those two wires. Instead, an uncomplicated solution would be to consider the (multi)set L' generated from L not as a (multi)set, but as a *sequence* of length m. In that sequence, the swaps may for instance be ordered by the natural order of the wires they consist of. This would result in a sequence like $\{(1,2),(1,2),(1,4),(2,3),\ldots\}$. One can now identify without much effort each swap by its unique index $k \in [m]$ in that sequence. Similarly to the x variables, this allows for a (bijective) mapping to the set $\{1, \ldots, m^2\}$. However, due to the fact that the first $(m+1) \cdot n^2$ integers are already in use by the x variables, an appropriate offset for the representation of the y variables is necessary. This results in the following function:

$$\langle k, r \rangle = (m+1) \cdot n^2 + k + (r-1) \cdot m \tag{12}$$

[2]If we were to exclude the corresponding integers from the set, then the mapping function would be not only injective, but surjective as well.

Once again, the right summand can be visualised as a function enumerating points in a two-dimensional space, with both axes running up to m. The inverse function would then operate as follows on an integer z:

- $value = z - (m+1) \cdot n^2$
- $r = \lfloor value \ / \ m \rfloor + 1$
- $k = value \mod m$

Note that due to Scala, similarly to most modern programming languages, starting its indexing at 0, an offset of $(-)1$ is necessary for both the given formulas and their inversions. Similarly, the indices of the wires lie in the set $\{0, \dots, n-1\}$ and the indices of the layers – in $\{0, \dots, m\}$. Those technical details do not influence the general satisfiability of our model.

3.3 Computing Satisfiability

Once the ListFeasibility instance has been formulated as a SAT instance in a DIMACS file, the latter must be tested for satisfiability to check the feasibility of the original swap list. There are various pieces of software that can be used for this purpose, most notably `minisat` as a standalone solution, which is however rather outdated. Our final choice is `Sat4j`, an open-source Java library for solving satisfiability problems. Since it is written in Java, it can be seamlessly integrated into our Scala program for convenient usage. As a downside however, it is supposed to be slower than e.g. its equivalent in the C++ language, according to the authors. Nevertheless, any solver accepting DIMACS files as input can in theory be applied. As long as the SAT instance is satisfiable and `Sat4j` does not reach a timeout, it will output a list of positive or negative integers – the encoding of our variables – from which a specific tangle can be read.

3.4 Result Visualisation

Our program is also able to visualise a tangle realising any swap list that was found to be feasible. For this, the positive y variables are extracted from the output of `Sat4j` and mapped back to the swaps they represent. For the visualisation itself, we use the Python script `svgExporter` found on GitHub. The script utilises the `svgwrite` Python library and was created with the purpose of visualising templates of chaotic attractors. In the case of ListFeasibility, it can handle up to 20 wires and produces an SVG image file with the resulting tangle.

4 Relaxing Height Constraints and Height Minimization

The model we presented so far serves the purpose of solving ListFeasibility, i.e., the problem to decide whether or not a given swap list can be realised at all. It is based on the trivial observation that a list of length m is realisable *iff* there exists a tangle of height m

that realises it. We modelled this by asserting that between each pair of adjacent layers, exactly one swap is taken. It is however possible, by minor changes, to also apply our basic idea to solve the Tangle-Height Minimization problem.

Tangle-Height Minimization is an optimisation problem regarding the number of layers. Such problems can be solved e.g. by limiting the solution space and then performing binary search for the best solution with an algorithm for the decision version of the problem. In our case, no feasible swap list can require more than m steps to be realised. On the other hand it can also not admit a realisation within less than $\min_{i \in [n]}\{m_i \mid m_i = |L(i)|\}$ steps since a wire can only participate in at most one swap per step. So, suppose that checking whether a list is feasible within some fixed number of layers can be done in $\mathcal{O}(t_{\text{check}}(|L|, m, n))$ time. This yields the following algorithm running in $\mathcal{O}(t_{\text{check}}(|L|, m, n) \cdot \log m)$ time:

```
function minimiseTangleHeight(L, n, m)
    l ← min_{i∈[n]} {m_i | m_i = |L(i)|}
    r ← m
    minHeight ← −1
    // -1 indicates an unfeasible tangle
    repeat
        k ← ⌊(l+r)/2⌋
        currHeight ← CheckFeasibility(L, k)
        if currHeight == -1 then
            l ← k + 1
        else
            r ← k − 1
            minHeight ← currHeight
    until l > r
    return minHeight
```

Note that CheckFeasibility(L, k) returns -1 if L is not feasible in at most k steps; otherwise, it returns some number of steps in which L is feasible. This number can also be strictly less than k, i.e., the algorithm can achieve a greater reduction in a single iteration than expected.

For our idea to be applicable to such approaches, we must modify it to allow taking multiple swaps per step. In particular, this means removing the clauses produced by Equation 8. Doing so still means that each swap is taken exactly once, however there can be steps where multiple or no swaps are used. On the other hand, a new issue arises in the form of multiple non-disjunct swaps taking place in the same step (e.g., the swaps $(5, 6)$ and $(6, 7)$). We need a new set of clauses to assert that those events cannot occur. For instance, we could add the condition that if a swap $s = (i, j)$ is taken right after layer r then no other swaps containing wire i or wire j can be used. This does not apply only to swaps in the form (i, j) but also for any $(i, k), k \neq j$ or $(k, j), k \neq i$:

$$y_s^r \Rightarrow \neg \bigvee y_{s'}^r \qquad \forall r \in [m] \; \forall s = (i,j) \in L \; \forall s' \in L(i) \cup L(j) : \quad s \neq s' \tag{13}$$

Now, our SAT instance can be satisfiable and still contain some layers after which no swap happens. Simultaneously, the given swap list may be feasible in general, but the corresponding SAT instance will be unsatisfiable if not enough layers are provided.

The first case does in fact provide an opportunity for an alternative, iterative algorithm. If a swap list is deemed feasible with a realisation with height h, then one can check how many of those steps are empty and thus potentially determine a smaller height h'. Then, another check can be performed on height $h' - 1$ and so on. As soon as the list is not feasible in height $h' - 1$, the process terminates and outputs h' as a result. This algorithm runs in $\mathcal{O}(t_{\text{check}}(|L|, m, n) \cdot m)$ time. While the first algorithm is *asymptotically* faster, it is expected to perform logarithmically many iterations. On the other hand, depending on the SAT solver, it is possible that the second algorithm terminates after only two feasibility checks. This would be the case when the first check fits the swaps in a tangle of optimal height, i.e., h' is already minimal (even if h itself is potentially not). Then the second iteration would conclude that the swap list is unfeasible within a smaller height, i.e., h' is optimal.

5 Testing and Evaluation

We perform a set of tests to assert the applicability and measure the performance of our model. For this, we largely stick to the examples found in the GitHub repository of Kindermann mentioned above. While some of them originate from the GitLab repository of the tool "cate" presented in [OMK+18], the vast majority was created by Kindermann and the research group he was part of at that time.

Our test bench consists of a virtual machine (VM) running on Julia, the JMU's High Perfomance Computing Cluster (HPC). The VM has the following properties: 8 virtual CPUs (VCPUs), 64 GB of RAM and 40 GB of memory – more than enough for our experiments. The VM is based on an Ubuntu image, in particular a v18.04 one. For the sake of consistency, all of the experiments are run sequentially and not in parallel, so as to not induce any conflicts in the available (virtual) computational resources.

In order to evaluate the performance of our implementation in deeper detail, we conduct a set of experiments:

- an instance-wise comparison of runtimes for solving Tangle-Height Minimization,
- a computation of runtime variance for solving Tangle-Height Minimization, and
- a fine-grained computation of the solution of Tangle-Height Minimization for selected examples.

Note that the first test also serves as an assertion of the correctness of our model and implementation. For this, we also conduct this test on Kindermann's Python code and

Johannes Zink's Java code (both available in Kindermann's repository) and compare the final results in terms of tangle height. Note that Kindermann's code computes tangles using a dynamic program while Johannes Zink applies a branching algorithm for the purpose. All resulting measurements are given in milliseconds (ms). We now present our results from the given tests.

Special instances. In addition to the already mentioned examples, we consider a particular set of swap lists $L_i, i \in \mathbb{N}^+$ defined as follows. First, we create 2^i many wires indexed from 0 until $2^i - 1$. Next, we specify the swaps. Wires i and j swap 2 times if $i \mid j \neq j$, where $\mid$ represents the bitwise or-operation. Of course, having two wires swap is a symmetric relation, so there will also be two swaps if $i \mid j \neq i$. Otherwise, wires i and j do not swap at all.

The instance L_4 was of particular interest – while L_1 to L_3 were known to be feasible as proven by the dynamic program and the branching algorithm, L_4 was at the time of this practical course still unclear. One of our tasks was to test the conjecture that L_4 is in fact unfeasible. Indeed, our model was able to confirm that. On the Julia HPC, the computation took 518465 ms on average, or roughly 8.6 minutes, using the iterative approach. The binary search was able to confirm the result as well, however the computation was over four times slower: the mean computation time was 2178412 ms, or roughly 36.3 minutes.

5.1 Variance

It is clear that measuring the time needed to compute a tangle once does not provide a precise statement about the performance of our model. To get an idea about the possible deviation in the computations for our code, we compute the variance over a set of measurements on the same example. We select two examples: one that is not feasible within height m (and thus not feasible at all), and a feasible one where the model must first seek the minimal height of the tangle. We apply the binary search and the iterative approach on both examples and conduct $k = 100$ such runs to acquire a measurement sample. We use the *corrected sample variance*, given by the formula

$$\sigma^2 = \frac{\sum (x_i - \overline{x})^2}{k - 1}$$

where $\overline{x}$ is the mean of the sample set.

We select L_3 as the feasible example. Figures 1 and 2 display the data from the repetitions of the binary search and the iterative methods, respectively. Note that in both cases we have sorted the data entries in ascending order. The resulting variance for the binary search approach is 142.4084 while the iterative method yields a variance of 107.5144.

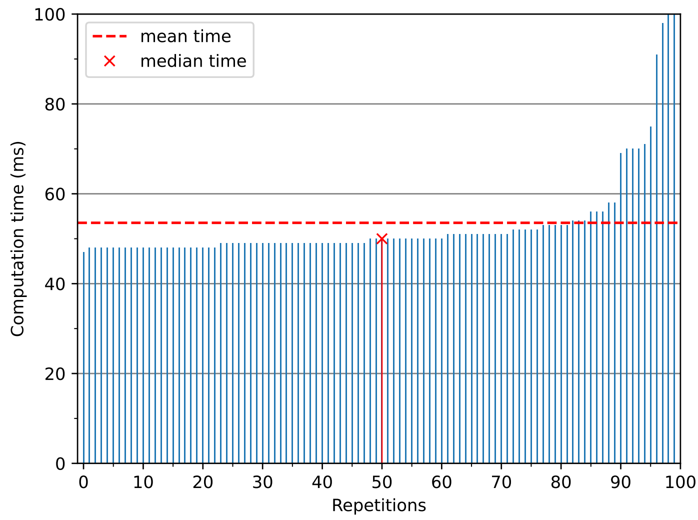

Fig. 1: Minimising the height of L_3 with the binary search method. Data is sorted in ascending order.

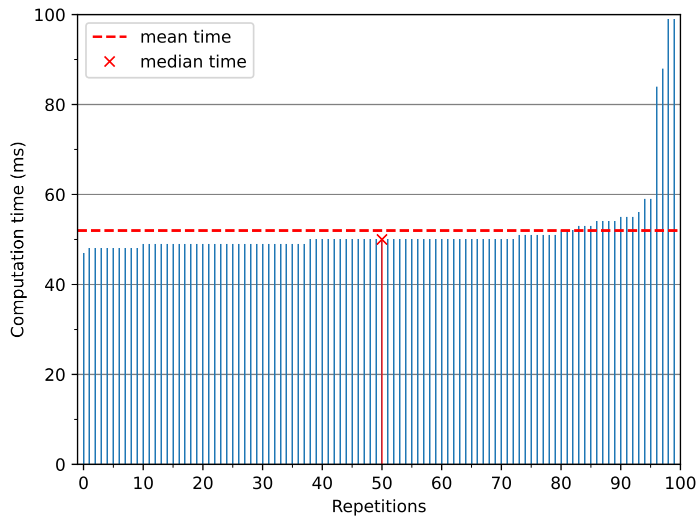

Fig. 3: Minimising the height of instance 28-1 with the binary search method. Data is sorted in ascending order.

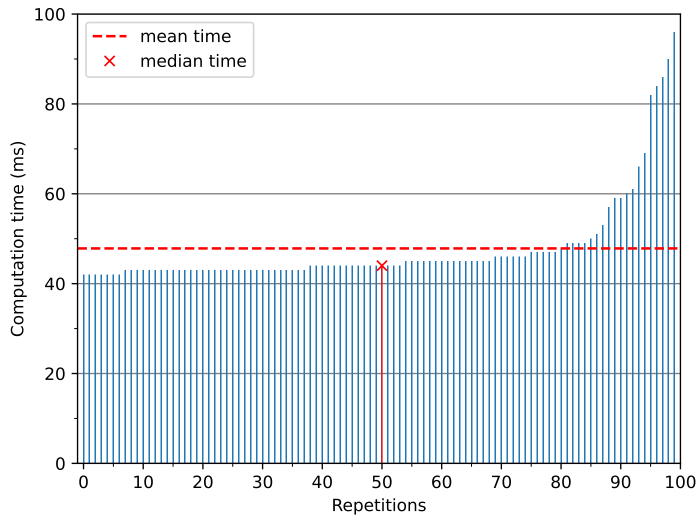

Fig. 2: Minimising the height of L_3 with the iterative method. Data is sorted in ascending order.

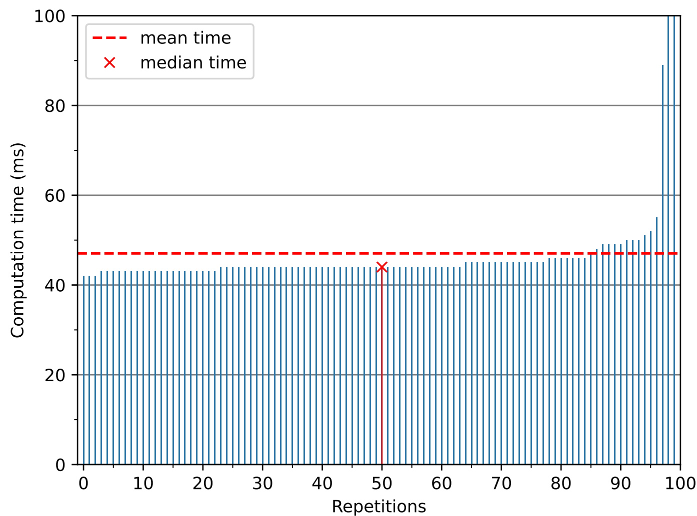

Fig. 4: Minimising the height of instance 28-1 with the iterative method. Data is sorted in ascending order.

As an unfeasible example we choose *extra_ 5x5_ 28-1* from Kindermann's repository. Similarly to L_3, the results are displayed on Figures 3 and 4. The variances here are 73.6595 and 199.9184 for the binary search and the iterative method, respectively.

In both cases, the binary search approach needs slightly longer than the iterative approach. This confirms our original assumption that while the binary search is asymptoti-

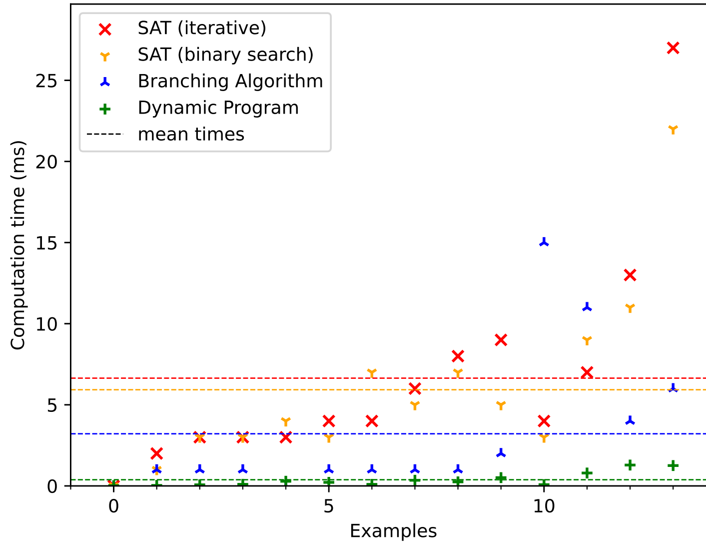

Fig. 5: Computation times of the 5×5 examples for each program.

cally faster, the iterative approach potentially needs less runs. A potential explanation is that our SAT solver of choice, `Sat4j`, is as an implementation of `minisat` very effective. We assume that the computation time spikes are due to the SAT solver taking suboptimal paths. Indeed, since SAT solvers are usually non-deterministic, it is less likely that the multiple repeating values are a product of e.g. branch prediction, but have emerged naturally. As for the larger variance values, those are also clearly results of the singular spikes in the computation time in some of the runs.

5.2 Runtime comparison

Next, we took all examples from the repository of Olszewski et al. and applied the four approaches (by us: two SAT-based methods, the dynamic program by Kindermann and a branching algorithm by Zink) available to us in order to make a comparison of the actual runtimes. The results are presented in Figures 5, 6 and 7. Note that the data sets are sorted in ascending order by the average of the four runtimes. Additionally in each set, the mean time for each approach is displayed by a dashed line. Due to the large difference in computation times between the instances, Fig. 6 and 7 use a logarithmic y-scale for the computation time.

A clear trend is that in the vast majority of the examples, our two approaches performed similarly well compared to each other – presumably due to efficient decisions by the SAT solver leading to swift height minimisation. Even more notably, both of our approaches exhibit a significantly slower growth in comparison to the branching algorithm and the dynamic program. We now add some of the *extra_ 5x5* examples, namely from 10-0 to 30-0. Unlike the previous examples however, not all of the *extra*-s have a

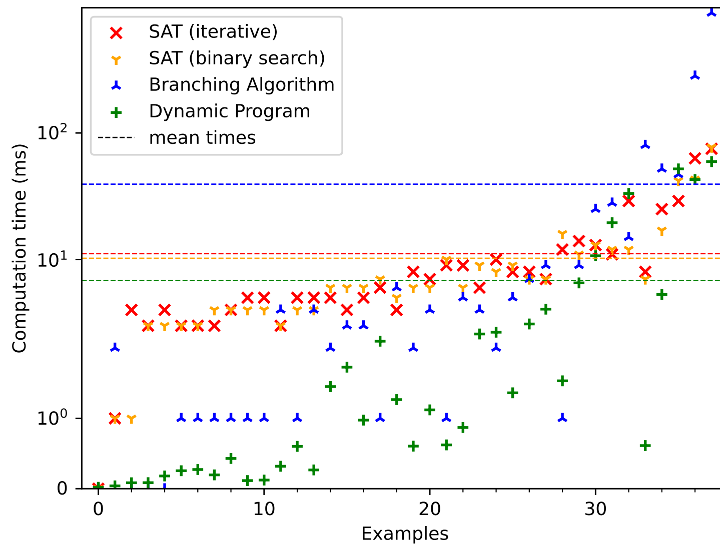

Fig. 6: Computation times of the 6×6 examples for each program.

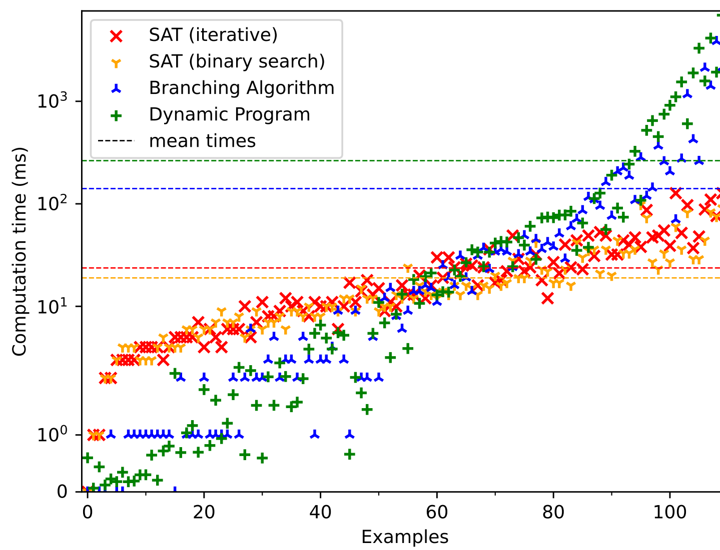

Fig. 7: Computation times of the 7×7 examples for each program.

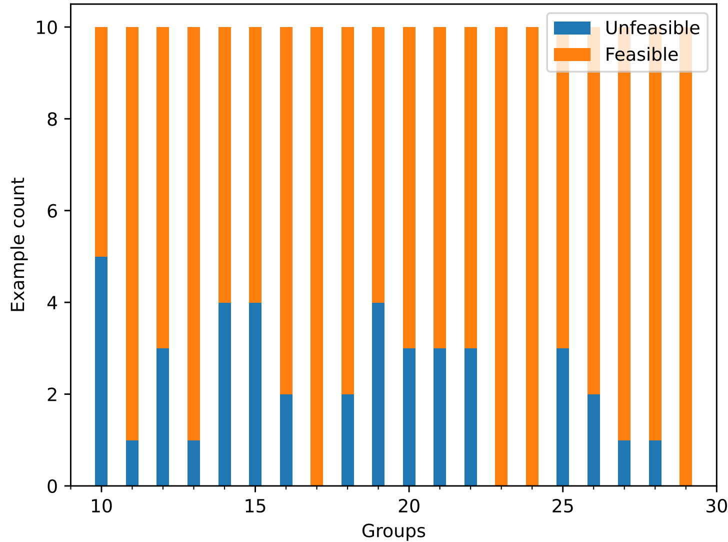

Fig. 8: Distribution of feasible and unfeasible examples in the *extra*_5×5 set.

corresponding tangle. For this reason we separate the data in two subgroups: feasible and unfeasible, the distribution of the two groups can be seen in Figure 8. Those groups' times are visualised using a logarithmic y-scale in the Figures 9 and 10. Note that the number of swaps per instance for each group is encoded in the group name. This means that e.g. group 30 contains examples with exactly 30 swaps.

In the case of the feasible examples, we see a continuation of many trends from the previous instances: the computation times of the two SAT-based approaches are very similar and growing with a gradual slope, unlike the more steep one of the branching algorithm. However, unlike e.g. the 7×7 instances, the growth of the dynamic program is also gradual and the average computation time is the lowest of the four. We are presented with a different situation in the case of the 32 unfeasible examples. There is a clear difference between the iterative SAT approach and the binary search. This is due to the fact that while the former only needs to test one maximal height (m), the binary search still has to go through $\mathcal{O}(\log n)$ many steps. A rather remarkable case is presented by the branching algorithm which performs best on almost all examples. The dynamic program, however, exhibits a steeper growth than the other two approaches, similarly to its performance on the 7×7 examples.

For better comparison, we also present the average runtimes of the four approaches in dedicated overviews. Figures 11 and 12, which use a logarithmic y-scale for the times as well, give better insights in the height minimisation of the *extra* instances. In the repository of Kindermann, those are named depending on the number of swaps they have – for instance, *extra*_*5x5*_*28-1* refers to the second example (since indexing begins at 0) in a group of 10, every one of which has 28 swaps. We apply this grouping to the data we

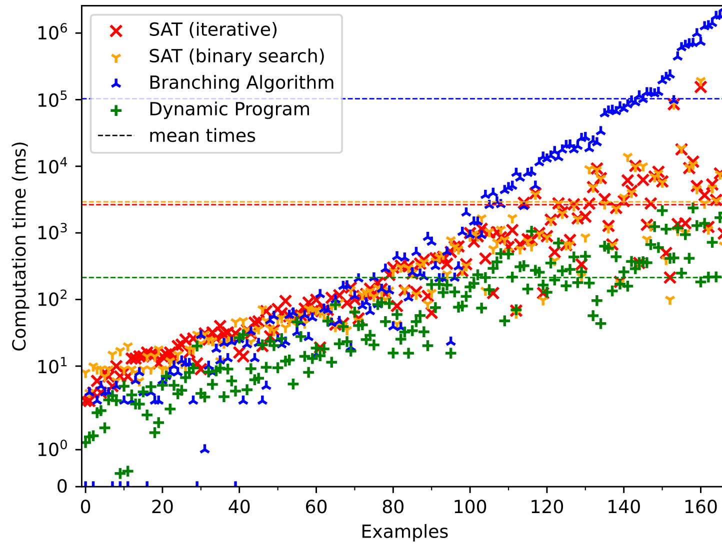

Fig. 9: Computation times of the feasible $extra_5 \times 5$ examples for each program.

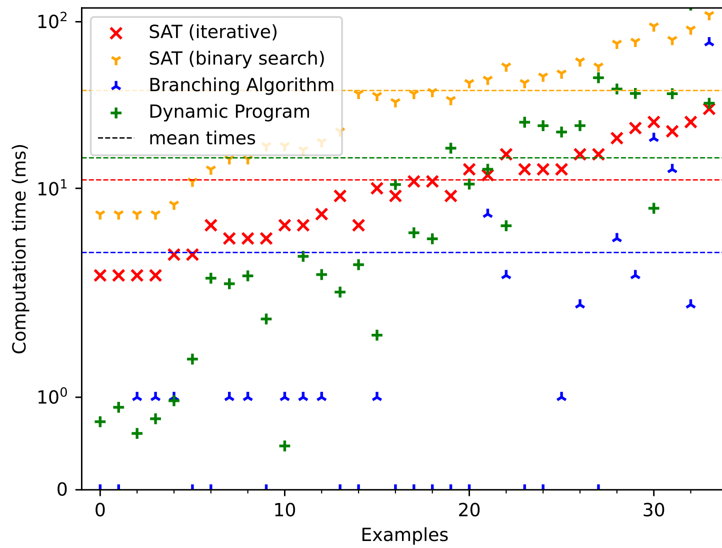

Fig. 10: Computation times of the unfeasible $extra_5 \times 5$ examples for each program.

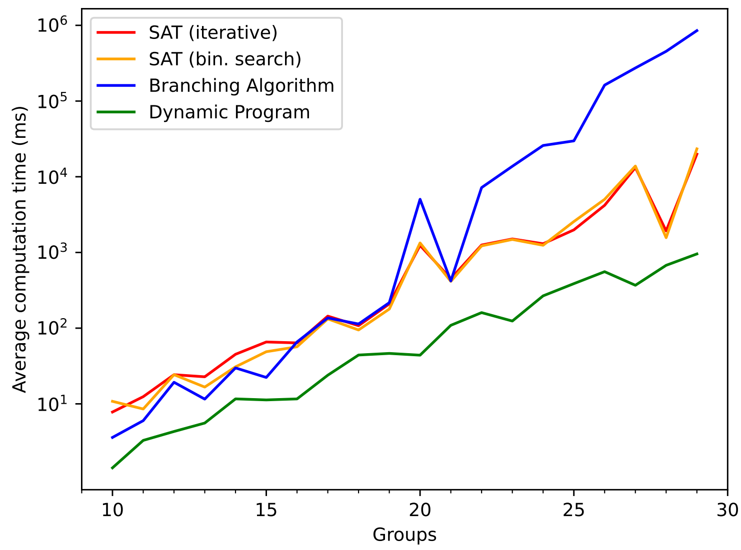

Fig. 11: Computation times of the feasible *extra*_5×5 examples, grouped by number of swaps.

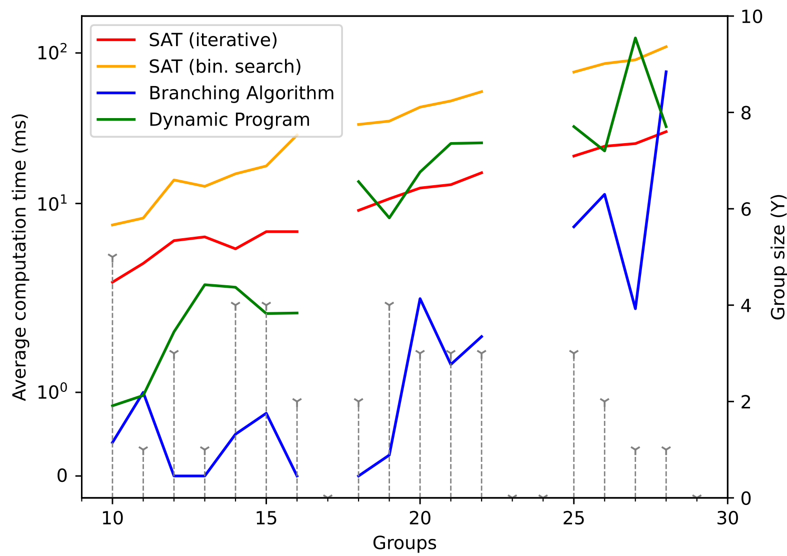

Fig. 12: Computation times of the unfeasible *extra*_5×5 examples, grouped by number of swaps.

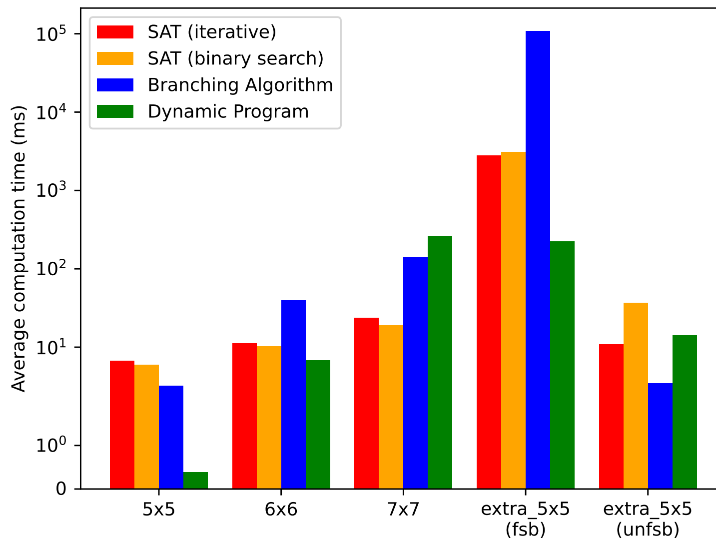

Fig. 13: Average times of the four approaches on all example groups.

collected and take the average[3] of each group, resulting in the above-mentioned Figures. Note that the curves on Figure 12 are not connected since there are groups without unfeasible examples. The number of unfeasible instances for each group is displayed with gray dashed lines and an Y-marker, relating to the right y-axis. As expected, all four methods' runtimes, seen on the logarithmic y-axis to the left, are proportional to the number of swaps and while they do not exhibit monotonic growth, in general larger instances need longer to be solved. In the case of the unfeasible examples, it is noteworthy that the dynamic program and the branching algorithm exhibit a very inconsistent and random-looking behaviour; however, it is difficult to speculate on why this could be the case.

We present our final overview in Figure 13 – it displays the average times of the four approaches, as already seen on Figures 5, 6, 7, 9 and 10, next to each other for the purpose of comparison. While all four methods handle unfeasible instances surprisingly fast, it is clear how making the instances generally more difficult to solve by adding more swaps does result in increased average computation times.

5.3 Detailed height minimisation

Our experiments are concluded by a step-by-step height minimisation run using our iterative and our binary search approach. For this, we selected two *extra* instances: 30-3 and 31-5. In particular, we recorded what height the method was attempting to achieve

[3] Note that for the average, we divide by the number of feasible or unfeasible examples from that group respectively, and not simply by the total number of examples in the group.

in each step, what height it actually achieved, and how much time it needed to compute the result. Table 1 presents this data for the iterative and the binary search method, respectively, applied on 30-3. Note how the SAT solver easily deduces when the target height is sufficient if presented with enough "operating space": it makes jumps of size 5 and 3 in the beginning of the iterative approach for ca. 100 milliseconds. However, the tighter the bound gets, the more the computation time increases: The actual minimal height (which can not be reduced further) of 17 takes almost 2000 milliseconds for both approaches, and attempting to create a tangle with sub-minimal size results in a huge computation time spike of ca. 8 minutes. A similar behaviour is observed for example 31-5, presented on Table 2. However, here the spikes when trying to create a non-existent tangle are even larger.

Target height (iter.)	Achieved height (iter.)	Time (iter.) (ms)	Target height (bin.)	Achieved height (bin.)	Time (bin.) (ms.)
30	25	112	22	21	58
24	21	122	18	18	72
20	19	50	16	-	498315
18	18	83	17	17	1811
17	17	1960			
16	-	470050			

Tab. 1: Height minimisation times for instance 30-3 with optimal height of 17.

Target height (iter.)	Achieved height (iter.)	Time (iter.) (ms)	Target height (bin.)	Achieved height (bin.)	Time (bin.) (ms.)
31	25	60	24	21	373
24	21	354	20	20	199
20	20	194	18	-	2955719
19	-	5681968	19	-	5621725

Tab. 2: Height minimisation times for instance 31-5 with optimal height of 20.

6 Conclusion and Future Work.

In this practical course, we developed a SAT-based approach for solving the problems ListFeasibility and TangleHeightMinimisation. In particular, both the number of variables and clauses they create are polynomial in the number of wires and swaps of the given ListFeasibility instance. We implemented our approaches in the Scala programming language using the SAT solver `Sat4j`, which is based on `minisat` but is implemented in Java. Finally, we used the Julia HPC to conduct various experiments on a large example database using not only our approaches, but also a dynamic program created by Philipp

Kindermann and a branching algorithm created by Johannes Zink. We showed that in general, both the iterative and the binary search approach of our SAT formulation exhibit a gradual slope regarding the increase of the computation time in relation to the size of the instance that is being tested. We also showed that our approaches can in some cases outperform the other two.

All of this establishes a good base for further research. For instance, it would be interesting to check how other SAT solvers perform in comparison to `Sat4j` as a good SAT solver is crucial for our models' performance. It would also be interesting to see how the four height minimisation methods perform compared to each other when implemented in the same programming language – while a Scala implementation can be expected to be marginally slower than its Java counterpart (since Scala must load its own libraries in addition to those of Java, resulting in a larger windup time), it is also possible to optimise it using Scala's functional features. Similarly, the use of an optimised SAT solver written natively in Scala would be interesting to see. Finally, one could also investigate how well the approaches scale explicitly in regard of the number of swaps of the test instance, or the number of wires.

7 Appendix

Transformation of the given rules to CNF.

Here, we give the equations that were presented in Section 2 in Conjunctive Normal Form (CNF) for the sake of completeness.

Equation 2:

$$x^r_{i,j} \wedge x^r_{j,k} \Rightarrow x^r_{i,k}$$
$$\leftrightarrow$$
$$(\neg x^r_{i,j} \vee \neg x^r_{j,k} \vee x^r_{i,k})$$

Equation 3:

$$x^r_{i,j} \Leftrightarrow \neg x^r_{j,i}$$
$$\leftrightarrow$$
$$(x^r_{i,j} \vee x^r_{j,i}) \wedge (\neg x^r_{i,j} \vee \neg x^r_{j,i})$$

Equation 6:

$$\neg(y^r_s \wedge y^{r'}_s)$$
$$\leftrightarrow$$
$$(\neg y^r_s \vee \neg y^{r'}_s)$$

Equation 9:

$$y^r_s \Rightarrow (x^r_{i,j} \neq x^{r+1}_{i,j})$$
$$\leftrightarrow$$
$$(\neg y^r_s \vee \neg x^r_{i,j} \vee \neg x^{r+1}_{i,j}) \wedge (\neg y^r_s \vee x^r_{i,j} \vee x^{r+1}_{i,j})$$

Equation 10:

$$(x^r_{i,j} \neq x^{r+1}_{i,j}) \Rightarrow \bigvee_s y^r_s$$

$$\leftrightarrow$$

$$\left(x_{i,j}^r \vee \neg x_{i,j}^{r+1} \vee \bigvee_s y_s^r\right) \wedge \left(\neg x_{i,j}^r \vee x_{i,j}^{r+1} \vee \bigvee_s y_s^r\right)$$

Equation 13:

$$y_s^r \Rightarrow \neg \bigvee_{s'} y_{s'}^r$$

$$\leftrightarrow$$

$$\bigwedge_{s'} (\neg y_s^r \vee \neg y_{s'}^r)$$

References

[FFK+23] Oksana Firman, Stefan Felsner, Philipp Kindermann, Alexander Ravsky, Alexander Wolff, and Johannes Zink: The complexity of finding tangles. In Leszek Gąsieniec and Peter Gurský (editors): *Proc. 49th Int. Conf. Current Trends Theory & Practice Comput. Sci. (SOFSEM'23)*, Lecture Notes in Computer Science. Springer, 2023. `https://arxiv.org/abs/2002.12251`, to appear.

[FKR+19] Oksana Firman, Philipp Kindermann, Alexander Ravsky, Alexander Wolff, and Johannes Zink: Computing height-optimal tangles faster. In Daniel Archambault and Csaba D. Tóth (editors): *Proc. 27th Int. Symp. Graph Drawing & Network Vis. (GD'19)*, volume 11904 of *Lecture Notes in Computer Science*, pages 203–215. Springer, 2019. `https://doi.org/10.1007/978-3-030-35802-0_16`.

[OMK+18] Maya Olszewski, Jeff Meder, Emmanuel Kieffer, Raphaël Bleuse, Martin Rosalie, Grégoire Danoy, and Pascal Bouvry: Visualizing the template of a chaotic attractor. In Therese C. Biedl and Andreas Kerren (editors): *Proc. 26th Int. Symp. Graph Drawing & Network Vis. (GD'18)*, volume 11282 of *Lecture Notes in Computer Science*, pages 106–119. Springer, 2018. `https://doi.org/10.1007/978-3-030-04414-5_8`.