Sparse Matrix Compression of Matrices with Double Logarithmic Widths

Dominik Köppl [*] Vincent Limouzy [†] Andrea Marino [‡] Giulia Punzi [§]
Jannik Olblich [¶] Takeaki Uno [‖]

Abstract

The sparse matrix compression problem asks for a one-dimensional representation of a binary $n \times \ell$ matrix, formed by an integer array of row indices and a shift function for each row, such that access to the matrix can be done in constant time by consulting the representation. It has been shown that the decision problem for finding an integer array of length $\ell + k$ or restricting the shift function up to values of k is NP-complete, and that common greedy algorithms exhibit an approximation ratio of $\Theta(\sqrt{\ell + k})$. In that light, we propose a dynamic programming algorithm solving the problem in polynomial time for matrices of width $\ell \in \mathcal{O}(\lg \lg n)$.

1 Introduction

Binary matrices have ubiquitous applications in computer science, such as in databases, data mining, and machine learning. For instance, a binary matrix can represent an incidence matrix of a bipartite graph or a transition matrix of a finite automaton. In practice, these matrices are often sparse, which means that most of the entries are zero. To save space and time, it is desirable to compress these matrices. One way to compress a sparse matrix is to represent it as a one-dimensional array of row indices and a shift function for each row. Such a representation allows for constant-time access to the matrix, and has already been proposed in the 1970s. It was also studied in the context of compilers and databases in the 1980s [1]. The problem of finding such a representation is known as the *sparse matrix compression problem*, which we denote by minMaxShift. The original version of the decision problem of minMaxShift is defined as follows:

> minMaxShift, [11, Chapter A4.2, Problem SR13]
> **Input:** An $n \times \ell$ binary matrix $A[1..n][1..\ell]$ with n rows and ℓ columns and entries $A[i][j] \in \{0, 1\}$ for all $i \in [1..n]$, $j \in [1..\ell]$, and an integer $k \in [0..\ell \cdot (n-1)]$.
> **Task:** Decide whether there exists an integer array $C[1..\ell+k]$ with $C[i] \in [0..n]$ for every $i \in [1..\ell+k]$, and a function $s : [1..n] \to [0..k]$ such that $A[i][j] = 1 \Leftrightarrow C[s(i) + j] = i$ for all $i \in [1..n]$ and $j \in [1..\ell]$.
> **Note:** Assume $A[0][j] = 0$ for all j to allow setting $C[i] = 0$ for some i, modeling that this entry is unassigned.

In what follows, we call C the *placement* and s the *shift function* of the representation of A asked by minMaxShift.

[*]University of Yamanashi
[†]University Clermont Auvergne
[‡]University of Florence
[§]University of Pisa
[¶]University of Ulm
[‖]National Institute of Informatics

Example 1.1. Consider the MINMAXSHIFT problem with $k = 2$ for the following matrix A (of size 3×6) defined as follows (first zero row omitted):

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Intuitively, we wish to find a way to shift each row in a way such that no column contains two '1's, and use this information to compress the matrix into a one-dimensional vector of row indices of length $6 + 2 = 8$.

For instance, the shift function $s(1) = 0, s(2) = 2, s(3) = 1$ allows no column collisions for the '1's. Such shift can be represented as follows:

$$\begin{array}{cccccccc} 0 & 1 & 0 & 0 & 1 & 0 & & \\ \rightarrow & \rightarrow & 0 & 1 & 0 & 1 & 0 & 0 \\ \rightarrow & 0 & 1 & 0 & 0 & 0 & 1 & \end{array}$$

From this, we can obtain the placement $C = [0, 1, 3, 2, 1, 2, 3, 0]$ by setting $C[j]$ to be equal to the only row index which has a 1 in column j after the shift function is applied. Since such C is of the desired length, the problem has a positive answer.

Solving this problem is not an easy task. For instance, a brute-force algorithm checks for all possible shifts whether we can linearize the shifted matrix A with C. However, the number of possible configurations for the shift function s is $(k + 1)^n$, which is prohibitive even for a maximum shift k of 1.

The optimization version of MINMAXSHIFT is to find the smallest k such that there exists a solution. We can also understand this problem as finding the smallest upper bound on the maximum shift.

However, the decision problem is already NP-complete, as shown by Even et al. [10]. They showed that the problem is NP-complete by reducing the 3-COLORING problem to MINMAXSHIFT. The 3-COLORING problem is to decide whether a given graph can be colored with three colors so that no two adjacent vertices have the same color.

However, here we study a modification of MINMAXSHIFT, which we justify by the fact that the original problem does not take into account that the leftmost columns of all rows could be empty (as in the matrix of Example 1.1). So, we want to consider the length of the placement $(b_1, \ldots, b_{\ell+k})$ after trimming its empty borders. It is therefore no longer the case that minimizing the length of the placement and minimizing the maximum length of a shift are equivalent. For instance, a matrix consisting of only the empty row would give a sequence of length ℓ. We name our variant MINLENGTH, which also models the problem as a combinatorial puzzle of 1-dimensional polyominos with gaps, which we call *tiles* in the following. The problem is formally defined as follows.

MINLENGTH

Input: A set of n binary strings $S_1, \ldots, S_n$ of length ℓ such that $S_i \in \{0, i\}^\ell$ and an integer $k \in [0..\ell n]$.

Task: Decide whether there exists a placement $S[1..k] \in [0..n]^k$ such that S_i has a match in S, where 0 always matches (making it possible to match longer strings with shorter ones).

An instance of MINLENGTH with $S_i[1] = S_i[\ell] = i$ is equivalent to MINMAXSHIFT if we increment k by ℓ. Vice versa, an instance of MINMAXSHIFT is equivalent to MINLENGTH when we

1. trim all columns by their prefixes and suffixes of '0's,
2. decrement k by the length of the longest column after trimming, and
3. map each '1' to its corresponding row index.

By the problem statements, given a matrix A

with minimal-length parameter k such that MINLENGTH of this instance (A, k) is true, then MINMAXSHIFT is true for the same parameters (A, k), where k now represents the allowed maximum shift.

Example 1.2. For the matrix A of Example 1.1, we have a corresponding instance of MINLENGTH given by $S_1 = 1001, S_2 = 202$, and $S_3 = 30003$. For $k = 6$, a possible placement is $S = 132123$.

Next, consider the matrix B having two rows $B[1] = 10^\ell$ and $B[2] = 1^\ell 0$. We can solve MINMAXSHIFT with the optimal maximum shift of 1 by shifting the second row by 1. With the same strategy, we can solve MINLENGTH optimally with a minimum length of $\ell + 1$.

To observe that the trimming is the crucial operation that makes both problems inequivalent, we study the matrix C has two rows $C[1] = 0^\ell 1$ and $C[2] = 10^\ell$. On the one hand, since no column has more than one '1', MINMAXSHIFT can be solved without shifting, i.e., the maximum shift is zero. On the other hand, we obtain a placement of minimal length two for MINLENGTH by shifting the second row by $\ell - 1$ positions. Without trimming, the best minimal length is $\ell + 1$ achieved by the same solution as for MINMAXSHIFT.

Lemma 1.3. A placement of MINLENGTH without holes is optimal, but not every problem instance admits a placement without holes.

2 Related Work

Ziegler [14] was the first who mentioned MINMAXSHIFT. He gave a heuristic that tries to fit the next row at the leftmost possible available position. He also augmented his heuristic with a strategy that rearranges the order of the rows by prioritizing the row with the largest number of '1's. This strategy is known as Ziegler's algorithm. Despite the fact that Ziegler's algorithm is only a heuristic, it is often used in practice due to its good performance. For instance, Sadayappan and Visvanathan [13] similarly studied practical aspects of Ziegler's algorithm, and Aho et al. [1, Section 3.9.8] recommend using Ziegler's algorithm to represent the state transition of a deterministic finite automaton (DFA) in a compressed form. However, the approximation ratio of Ziegler's algorithm has not been studied yet.

Unfortunately, finding the optimal solution is generally hard. Even et al. [10] gave an NP-hard proof based on 3-coloring, which found an entry in the textbook of Garey and Johnson [11, Chapter A4.2, Problem SR13]. They showed that the problem is NP-complete even if the maximum needed shift is at most two. MINMAXSHIFT has been adapted to Bloom filters [4, 5, 9], and has also been studied under the name COMPRESSED TRANSITION MATRIX [7, Sect. 4.4.1.3] problem. Finally, Bannai et al. [2] showed that MINMAXSHIFT is NP-hard even when the width ℓ of the matrix is $\ell \in \Omega(\lg n)$. Their hardness proof can be applied directly to MINLENGTH, proving that MINLENGTH is NP-hard even when all tiles have a width of $\ell \in \Omega(\lg n)$.

Chang and Buehrer [3] considered a different variant in which cyclic rotations of a matrix row are allowed. However, this work is only practical and does not provide any theoretical guarantees. Another variation is to restrict rows to have no holes and to have not only one placement, but a fixed number of placements of the same length, which reduces to a rectangle packing problem [8, Section 2]. Finally, Manea et al. [12] studied a variant of embedding subsequences with gap constraints.

3 Dynamic Programming Algorithms

The main ingredient for our dynamic programming algorithms is a pattern matching algorithm that respects wildcards, which we use in the following lemma for finding all positions at which we can merge a tile with a placement.

Lemma 3.1. Given a placement P of length n and a tile T of length m, we can find in $\mathcal{O}(n \lg m)$ time all positions of P at which we can insert T.

Proof. We make use of the algorithm of Clifford and Clifford [6] for pattern matching on partial words. With partial word we mean a string that contains the wildcard symbol 0 that can match any character. The task there is to return all text positions at which a pattern occurrence starts, while allowing both pattern and text to contain single character wildcards. Clifford and Clifford show that this is possible in $\mathcal{O}(n \lg m)$ time.

By exchanging '1's with a letter unique to each (not necessarily different) tile, we can directly apply this algorithm to find all positions at which we can insert T into P. □

3.1 One Distinct Tile with Logarithmic Length

Assume that all tiles are the same. Then we can devise an DP-algorithm that solves minLength in time polynomial in n if the tile length ℓ is logarithmic in n. If we consider all different placements we can produce with i tiles, for increasing $i \in [1..n]$, a placement of i tiles can have a length of at most $i\ell$. The number of placements we thus can form can be exponential ($\leq 2^{i\ell}$), which is prohibitive. An insight is that we only need to track the last ℓ bits of each placement for a fixed i, if we prolong a placement always to its right. This is without loss of generality by symmetry. So, given a fixed placement P, we try to obtain all possible different ℓ-length suffixes by combining P with a newly chosen tile. Among all the placements with i tiles having the same ℓ-length suffix, we only keep the shortest one. Thus, for each i, we produce $O(2^\ell)$ placements from the placements of $i-1$ (or the empty string if $i = 1$). For $i = n$, we report the shortest placement among all produced ones.

Algorithmic Details In concrete terms, the DP table is a table $X[0..n][0..2^\ell - 1]$. For an integer $i \in [1..n]$ and an ℓ-length bit vector Y, X obeys the invariant that $X[i][Y] = \lambda > 0$ if and only if there is a placement of length λ using i tiles having an ℓ-length suffix equal to Y, where λ is chosen to be minimal. We start with $X[0][0] = 0$, and fill $X[i][\cdot]$ as follows: For each defined value $X[i-1][Y]$, compute all possible ℓ-length suffixes $Z = Y \bowtie T$, where T is the input tile. Here, $\bowtie$ denotes the merging operation of two ℓ-length bit vectors, which is defined if and only if there is no position $j \in [1..\ell]$ such that both $Y[j]$ and $T[j]$ are set to '1'. Formally, for two strings A and B, let $\ell \in [0..|A|]$ such that for all $i \in [\ell+1..|A|]$, $A[i] = 0$ or $B[i-\ell] = 0$ holds. Written the other way around, there is no $i \in [\ell+1..|A|]$ such that both $A[i]$ and $B[i-\ell]$ are non-wildcard symbols. For each such ℓ define $S_\ell[1..\max(|A|, |B|+\ell)]$ such that

$$S_\ell[i] = \begin{cases} A[i] & \text{if } A[i] \neq 0 \wedge i \in [1..|A|], \\ B[i-\ell] & \text{if } B[i-\ell] \neq 0 \wedge i \in [\ell+1..\ell+|B|], \\ 0 & \text{else,} \end{cases}$$

for every $i \in [1..\max(|A|, |B|+\ell)]$. Then $A \bowtie B := \{S_\ell\}_\ell$ is the set of all such constructed S_ℓ. By Lemma 3.1, we can compute $A \bowtie B$ in $\mathcal{O}(|A| \lg |B|)$ time.

Example 3.2. For $A = \texttt{aa0a0a}$ and $B = \texttt{b0b}$ (we use letters instead of 1 for clarity) we have $A \bowtie B = \{\texttt{aababa}, \texttt{aa0abab}, \texttt{aa0a0ab0b}\}$.

Given d is the number of new positions added to the placement by merging Y and T, if $X[i][Z]$ is undefined or $X[i][Y] + d < X[i-1][Z]$, we set $X[i][Z] \leftarrow X[i-1][Y] + d$. After filling $X[i][\cdot]$ we recurse for increasing i until $i = n$. Finally, in $X[n][\cdot]$, we return the minimum defined value in $X[n][\cdot]$ as the length of the optimal placement using all n tiles. The steps are summarized in pseudocode in Algorithm 1.

Correctness The correctness of the algorithm follows by induction on i. For $i = 0$, the only possible placement is the empty placement of length 0 with an ℓ-length suffix of 0. We here interpret the suffix as the non-trimmed placement prior to removing all surrounding positions with '0's. Suppose that the invariant holds for $i-1$. Then, for each defined $X[i-1][Y]$, we compute all possible ℓ-length suffixes Z by merging Y with the tile P. Now suppose that there is actually a placement P of i tiles with a suffix of Y using less than $X[i][Y]$ positions. Since it is possible to extract the placement of the rightmost tile T in P, we can remove T from P and obtain a placement of $i-1$ tiles with a suffix of some Y' such that merging Y' with T gives Y. However, by the induction hypothesis, we have $X[i-1][Y'] \leq |P| - d$, where d is the number of new positions added by merging Y' and T, a contradiction. Informally, we cannot omit a possible solution by only tracking the ℓ-length suffixes of placements since otherwise we would have already considered a prefix of that placement when processing less tiles.

Complexity We compute Z from Y and the tile P by leveraging Lemma 3.1 to find all positions at which we can merge the tile with the placement represented by Y. Since Y has a length of at most ℓ, we need $O(2^\ell \cdot \ell \log \ell)$ time for each i, thus $O(n\ell 2^\ell \log \ell)$ total time. As a side note, it suffices to keep only the latest computed row $X[i][\cdot]$ in memory, so the space is $O(2^\ell)$. We conclude that MINLENGTH is in polynomial time solvable if all tiles are non-distinct and $\ell = O(\lg n)$.

Theorem 3.3. MINLENGTH with only one distinct tile of length ℓ is solvable in $O(n^2 2^\ell \ell \log \ell)$ time. In particular, it is fix-parameter tractable in ℓ and polynomial-time solvable if $\ell = O(\lg n)$.

3.2 General DP-algorithm

The main obstacle in generalizing Thm. 3.3 to more distinct tiles is that the order of the chosen tiles now becomes important. More precisely, we group tiles by their binary representation in *tile-kinds*. Hence, two equal tiles have the same tile-kind. If there are κ tile-kinds, i.e., κ distinct tiles, each with cardinality $c_i \geq 1$ for $i \in [1..\kappa]$, then the number of ways to sort all n tiles is $\frac{n!}{c_1! c_2! \cdots c_\kappa!}$, which is exponential in n for general κ.

However, as we will see, it suffices to track only with a Parikh vector of the used tiles instead of their order. Let $\vec{p}_0 = (0, 0, \ldots, 0)$ be the empty Parikh vector of length κ and $\vec{p}_n = (c_1, c_2, \ldots, c_\kappa)$ be the Parikh vector of all input tiles. That is because, as before, we always prolong a partial solution to the right by adding a new tile of a given tile-kind at the ℓ-length suffix. By tracking partial solutions for each Parikh vector individually, we change in our table $X[i][\cdot]$ the integer i with a Parikh vector $\vec{p}$, and induce the table by filling $X[\vec{p}][\cdot]$ from all solutions $X[\vec{p'}][\cdot]$ such that $\vec{p}$ can be obtained by adding one tile of a given tile-kind to $\vec{p'}$. In particular, $|\vec{p'}| = |\vec{p}| - 1$. By doing so, X holds the following invariant, which can be proven by induction on $|\vec{p}|$. Given $X[\vec{p}][Y] = \lambda > 0$, there

Algorithm 1 DP-algorithm for MINLENGTH with one distinct tile.

```
1: Function minLengthDP(n, P)
2: Input: n tiles P of length at most ℓ
3: initialize X[0][0] ← 0
4: for i from 1 to n do
5:     for each Y with X[i − 1][Y] defined do
6:         for each distinct ℓ-length suffixes among the strings in Y ⋈ P do
7:             let d be the number of new positions added by merging Y and P
8:             if X[i][Z] is undefined or X[i − 1][Y] + d < X[i][Z] then
9:                 X[i][Z] ← X[i − 1][Y] + d
10: return the minimum defined value in X[n][·]
```

is a placement of length λ using the multi-subset of tiles represented by $\vec{p}$ having an ℓ-length suffix equal to Y, where λ is chosen to be minimal. This invariant leads to the correctness of the algorithm, which outputs the minimum defined value in $X[\vec{p}_n][\cdot]$ as the length of the optimal placement using all n tiles. We summarize the steps in pseudocode in Algorithm 2.

Complexity There are n tiles, and each tile can be represented by a binary number with 2^ℓ bits. Thus, the number of different tile-kinds κ is at most 2^ℓ. A Parikh vector thus has length at most 2^ℓ. Since a number in a Parikh vector has the domain $[1..n]$, the count of all Parikh vectors is upper bounded by n^{2^ℓ}. For each Parikh vector $\vec{p}$, we compute $O(2^\ell)$ suffixes Z from each defined $X[\vec{p'}][Y]$ using Lemma 3.1. Thus, the total time is $O(n^\kappa \ell n 2^\ell n) \subset O(n^{2^\ell} \ell n 2^\ell n)$.

Theorem 3.4. MINLENGTH with n tiles of length ℓ is solvable in $O(n^{2^\ell} \ell n 2^\ell n)$ time. In particular, it is polynomial-time solvable if $\ell = O(\lg \lg n)$, or if $\ell = O(\lg n)$ and the number of distinct tiles is $O(\lg n)$.

Implementation Details Instead of creating the whole DP-matrix X as a two-dimensional array, we can implement X by hash tables H_i representing $X[\vec{p}][\cdot]$ for all $\vec{p}$ with $|\vec{p}| = i$. In fact, we only need to keep two hash tables H_i and H_{i+1} in memory at the same time. Computing H_{i+1} from H_i can be done as with X by scanning all entries in H_i.

4 Experiments

The point of freedom in Ziegler's algorithm lies in the order in which tiles are placed. While Ziegler suggests sorting tiles by decreasing frequencies of ones, we here experiment with different sorting strategies to see their effect on the final placement length. We used the following sorting strategies:

- `none`: no sorting
- `incFreq`: sort by increasing number of ones
- `decFreq`: sort by decreasing number of ones
- `incDens`: sort by increasing density $\frac{\text{number of ones}}{\text{length}}$
- `decDens`: sort by decreasing density

Additionally, we implemented a strategy, named `random`, that calls Ziegler's algorithm 10 times with

Algorithm 2 DP-algorithm for MINLENGTH with distinct tiles.

Function minLengthDPGeneral(n tiles of length at most ℓ)
Compute $\vec{p}_n$ from the input tiles, let $T_1, T_2, \ldots, T_\kappa$ be the different tile-kinds
initialize $X[\vec{p}_0][0] \leftarrow 0$
for each Parikh vector $\vec{p}$ with $X[\vec{p}][Y]$ defined and $|\vec{p}|$ from 0 to $n-1$ **do**
    for each $i \in [1..\kappa]$ with $\vec{p}[i] < \vec{p_n}[i]$ **do**
        let $\vec{p'}$ be $\vec{p}$ with $\vec{p'}[i] = \vec{p}[i] + 1$
        for each distinct ℓ-length suffixes among the strings in $Y \bowtie P$ **do**
            let d be the number of new positions added by merging Y and P
            if $X[\vec{p'}][Z]$ is undefined or $X[\vec{p}][Y] + d < X[\vec{p'}][Z]$ **then**
                $X[\vec{p'}][Z] \leftarrow X[\vec{p}][Y] + d$
return the minimum defined value in $X[\vec{p}_n][\cdot]$

random shufflings of the input tiles and returns the shortest result.

Unfortunately, except `random`, all strategies performed bad on the following instance using only two tile types X and Y. In detail, we select two integer parameters c and g, and define $X = (10^g)^c1$ and $Y = (10^{g-1})^c10^c1$. Both tiles have length $c(g+1)+1$ and contain exactly $c+1$ ones. We give n tiles in the order $X, Y, X, Y, \ldots$ as input to our heuristics. By construction, all defined sorting strategies behave the same, as can be observed in the left plot of Fig. 1. Only `random` is able to find a better placement. To explain this phenomenon, we note that this chosen instance is favorable to random shuffling since the optimal solution would pick an order that groups X's and Y's individually.

In a second example, we reveal a bad behavior of `random`. The idea is to introduce more freedom in the tile selection by adding a third tile Z defined as $Z = 10^{c(g-2)}1(0^{2g})^c1$. This tile has the same number of ones and the same length as X and Y. If we additionally define filler tiles such that the minimal solution has no holes, we obtain the results shown in the right plot of Fig. 1. There, `random` performs increasingly worse than all other strategies for increasing c.

Acknowledgements

This work was supported by the Research Institute for Mathematical Sciences, an International Joint Usage/Research Center located in Kyoto University, and a research stay thanks to Bézout Labex. We thank Jesper Jansson, Hideo Bannai, and the participants at LA Symposium 2024 Winter for constructive discussion.

The research was supported by ROIS NII Open Collaborative Research 2026 Grant Number 252M-23667, and JSPS KAKENHI Grant Numbers JP23H04378 and JP25K21150.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley series in computer science / World student series edition. Addison-Wesley, 1986.

[2] Hideo Bannai, Keisuke Goto, Shunsuke Kanda, and Dominik Köppl. NP-completeness for the space-optimality of double-array tries. *arXiv CoRR*, abs/2403.04951, 2024. doi: 10.48550/ARXIV.2403.04951.

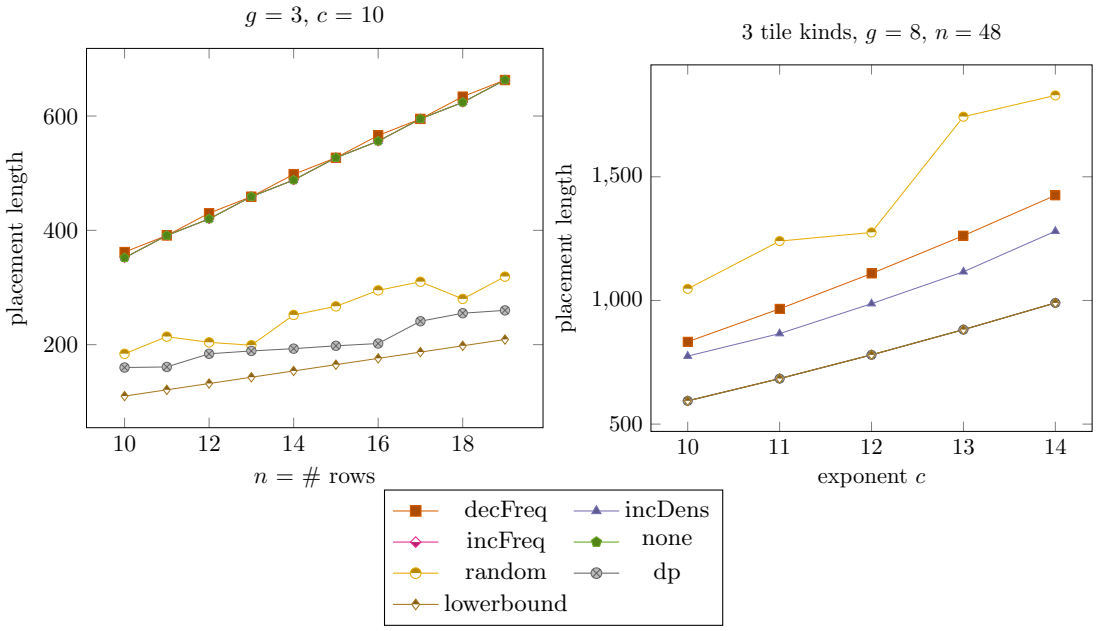

Figure 1: Placement lengths for different heuristics studied in Sect. 4. Left: Input of the form $X, Y, X, Y, \ldots$ with two tile types. Right: Input of the form $X, Y, Z, X, Y, Z, \ldots$ with three tile types, and additional helper tiles to fill all gaps in the shortest solution. All tiles X, Y and Z have the same number of ones.

[3] Chin-Chen Chang and Daniel J. Buehrer. An improvement to Ziegler's sparse matrix compression algorithm. *J. Syst. Softw.*, 35(1):67–71, 1996. doi: 10.1016/0164-1212(95)00086-0.

[4] Chin-Chen Chang and Tzong-Chen Wu. A letter-oriented perfect hashing scheme based upon sparse table compression. *Softw. Pract. Exp.*, 21(1):35–49, 1991. doi: 10.1002/SPE.4380210104.

[5] Chin-Chen Chang, Huey-Cheue Kowng, and Tzong-Chen Wu. A refinement of a compression-oriented addressing scheme. *BIT Numerical Mathematics*, 33(4):529–535, 1993. doi: 10.1007/BF01990533.

[6] Peter Clifford and Raphaël Clifford. Simple deterministic wildcard matching. *Inf. Process. Lett.*, 101(2):53–54, 2007. doi: 10.1016/j.ipl.2006.08.002.

[7] Jan Daciuk, Jakub Piskorski, and Strahil Ristov. Natural language dictionaries implemented as finite automata. In Carlos Martín-Vide, editor, *Scientific Applications Of Language Methods*, volume 2, chapter 4. World Scientific, 2010.

[8] Erik D. Demaine and Martin L. Demaine. Jigsaw puzzles, edge matching, and polyomino packing: Connections and complexity. *Graphs Comb.*, 23(Supplement-1):195–208, 2007. doi: 10.1007/S00373-007-0713-4.

[9] Peter C. Dillinger, Lorenz Hübschle-Schneider, Peter Sanders, and Stefan Walzer. Fast succinct retrieval and approximate membership

using ribbon. In *Proc. SEA*, volume 233 of *LIPIcs*, pages 4:1–4:20, 2022. doi: 10.4230/LIPICS.SEA.2022.4.

[10] S. Even, D.I. Lichtenstein, and Y. Shiloah. Remarks on Ziegler's method for matrix compression. unpublished, 1977.

[11] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. A Series of books in the mathematical sciences. Bell Laboratories, 1979.

[12] Florin Manea, Jonas Richardsen, and Markus L. Schmid. Subsequences with generalised gap constraints: Upper and lower complexity bounds. In *Proc. CPM*, volume 296 of *LIPIcs*, pages 22:1–22:17, 2024. doi: 10.4230/LIPICS.CPM.2024.22.

[13] P. Sadayappan and V. Visvanathan. Efficient sparse matrix factorization for circuit simulation on vector supercomputers. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 8 (12):1276–1285, 1989. doi: 10.1109/43.44508.

[14] S. F. Ziegler. Small faster table driven parser. Technical report, Madison Academic Computing Center, University of Wisconsin, 1977. unpublished.