Faster Compression of Patterns to Rectangle Rule Lists

Ian Albuquerque Raymundo Da Silva * Gruia Calinescu † Nathan De Graaf ‡

August 5, 2019

Abstract

Access Control Lists (ACLs) are an essential security component in network routers. ACLs can be geometrically modeled as a two-dimensional black and white grid; our interest is in the most efficient way to represent such a grid. The more general problem is that of Rectangle Rule Lists (RRLs) minimization, which is finding the least number of rectangles needed to generate a given pattern. The scope of this paper focuses on a restricted version of RRLs minimization in which only rectangles that span the length or width of the grid are considered. Applegate et al.'s paper "Compressing Rectilinear Pictures and Minimizing Access Control Lists" gives an algorithm for finding an optimal solutions for strip-rule RRLs minimization in $O(n^3)$ time, where n is the total number of rows and columns in the grid. Following the structure of Applegate et al.'s algorithm, we simplify the solution, remove redundancies in data structures, and exploit overlapping sub-problems in order to achieve an optimal solution for strip-rule RRLs minimization in $O(n^2 \log n)$ time.

1 Introduction

We consider the following problem of generating patterns by drawing rectangles. A target pattern is given as a grid of black and white cells. We begin with a white grid and place solid black or white rectangles, each rectangle covering any previously placed rectangles. Placing arbitrary sized rectangles makes the problem NP-hard, so we consider only placing rectangles that extend either the full height or width of the grid. Our goal is to find the smallest number of rectangles required to create the target pattern.

1.1 Problem definition

Define a pattern P to be an n_R by n_C grid of black and white squares. Let $n = n_R + n_C$. See Fig. 1 for an example of a pattern. A rectangle strip-rule on P is either a black or white rectangle that extends from one side of the pattern to

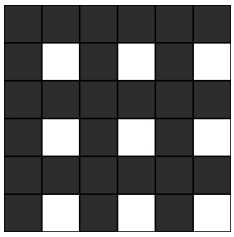

Figure 1: An example of a black and white pattern

the opposite side. Precisely, a rectangle in P is either a set of contiguous rows of P or a set of contiguous columns of P.

A rectangle strip-rule list (a RSRL) that generates P is an ordered list of rectangle strip-rules, that when applied in order to a blank (white) grid the size of P creates the target pattern. See Fig. 2 for an example of a pattern generated by a RSRL.

*Pontifical Catholic University of Rio de Janeiro, Brazil. ian.albuquerque.silva@gmail.com
†Illinois Institute of Technology, Chicago, Illinois, USA. calinescu@iit.edu
‡Iowa State University, Ames, Iowa, USA. nathandegraaf@gmail.com

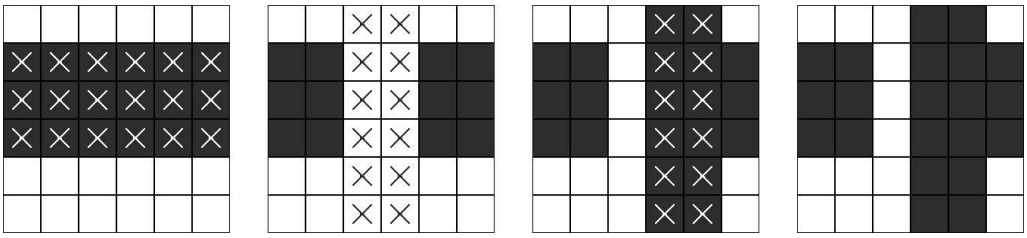

Figure 2: A pattern generated by a RSRL of 3 elements.

We say a pattern P is a strip-rule pattern if there is a RSRL that generates P. Note that not every pattern P is a strip-rule pattern. See Fig. 3 for an example of a pattern that is not a strip-rule pattern.

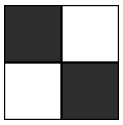

Figure 3: The 2×2 checkerboard: an example of a pattern that is not a strip-rule pattern.

An RSRL is considered optimal if it has the minimum number of rectangle strip-rules of any RSRL that generates P.

1.2 Problem background

1.2.1 Rectilinear pictures and access control lists.

The method of stacking rectangles to create patterns has applications in both graphics and network routers. A common method for drawing graphics is to allow the user to repeatedly apply a rectangle tool to a blank canvas (see Xfig or PowerPoint). Each rectangle is of a solid color and covers everything in a defined rectangular region. This sequence of drawn rectangles can be represented by a rectangle rule list (RRL). The problem of finding the minimum length RRL needed to create a given pattern is a generalization of the RSRL minimization problem that we explore.

Alternatively, instead of being given an n_R by n_C grid as input, one can start with the numbers n_R and n_C, and a list of m rules. The goal is to find a shortest list that gives the same pattern as the input list. As shown in an extended version of Applegate et al. [1] available on some of the paper authors' homepages, it is possible to construct the n_R by n_C grid in time $O(n_R \cdot n_C + m^2)$.

One important application of RRLs is in access routers. An internet service provider might use access control lists (ACLs) on network router line cards in order to choose whether to forward or drop a packet based on the sending or receiving IP address. This decision could be answered by checking a two-dimensional Boolean array based on the sender and receiver IP addresses. This problem can be translated into a restricted version of the RRL minimization problem (as explained below). An in-depth discussion of this translation and its properties is provided in detail in Applegate et al.'s paper [1]. Briefly, the idea in ACL minimization is to construct a given grid of size 2^w by 2^w, indexed by binary strings from 0^w to 1^w. Rectangles allowed on this grid are defined by a pair of binary strings (y,z) and cover any squares whose indices have y and z as a prefix respectively.

Unfortunately, the general problem of finding minimum length RRLs has been shown to be NP-hard by Applegate et al. [1]. Instead, we work on a restricted version of the problem in which any rectangle rules applied must extend either the height or width of the original pattern and only black or white rectangles are allowed. This 2-color strip-rule problem was originally posed in [1], where an optimal solution is given in $O(n^3)$-time. Applegate et al. [1] obtains an optimal solution to the strip-rule version of the ACL problem with a similar (but more complicated) $O(wn^3)$-time algorithm. A 1.5 ratio approximation algorithm is given for the problem in $O(n^2)$-time. While there exist numerous results related to various other restricted problems regarding RRLs, we know of no other work related to this 2-color strip-rule problem.

1.2.2 Related work.

ACLs can be used in firewalls [10]. Kang et al. [10] introduces axioms with the goal of creating and analyzing algorithms for optimizing the rewriting of rules in Software Defined Networks. Liu et al. [17] considers classifiers in dimensions higher than two, which reduce to either ACL or RRL lists in dimension two. They propose and experimentally evaluate heuristics without performance guarantees, as well as relate these classifiers to the Firewall Decision Diagrams of Gouda and Liu [8]. See also [3]. Pao and Lu [23], and Comerford et al. [4] also consider higher dimensional classification based on rectangle rules. Kang et al. [9] and Zhang et al. [31] introduce more general rule-minimization problems. Sun and Kim [27] uses rule minimization as a start for a solution to an extended problem. Efficiently removing redundant rules has been proposed by Liu and Gouda in [15] and [16], by Liu et al. in [14], and by Sun and Kim [28].

Lam. et al. [12] consider the problem of changing the rules such that the chance of getting a "hit" in the first N rules (point in one of the last N rectangle, in our terminology) is maximized. They propose a heuristic and present simulation results. Demianiuk et al. [6] allow approximate packet classifiers, where in order to limit the length of the ACL one does not have to obtain exactly the given pattern.

Sanger et al. [26] discusses a methodology (and algorithms) of checking if two rule sets are equivalent, which in our terminology, means that they generate the same pattern. Architectures for packet classification that use modifications of ACLs have been proposed by [18], [20], and [25]. Compacting ACLs on multiple switches has been studied by [24], [19], [30], and [13].

Giroire et al. [7] study a problem related to RRL optimization in which the rectangles are either a column, a row, or a cell, and there may be more than two colors. This paper proves this problem to be NP-hard, answering an open question of [29], and provides approximation algorithms.

The complexity of finding minimum length ACLs in two dimensions is still unknown, but with an arbitrary number of dimensions, this problem is NP-hard according to Kang et al. [11]. Applegate et al. [1] gives a $O(\min(m^{1/3}, OPT^{1/2}))$-approximation algorithm for finding minimum length RRLs, where m is the length of the input RRL. This is still the best published ratio.

Daly et al. [5] provides heuristics for higher-dimensional ACL and RRL minimization (reference [5] calls the ACLs minimization prefix-ACL, while range-ACL is their terminology for finding minimum length RRLs), on the way improving the approximation ratios provided by Applegate et al. [1] for ACLs minimization. The approximation algorithms of [5] and [1] use as a subroutine the strip-rule version that we study.

1.3 Our results

Using the structure defined in Applegate et al.'s [1] $O(n^3)$ exact algorithm for 2-color RSRL minimization, we give an improved $O(n^2 \log n)$ exact algorithm for the RSRL minimization problem. Given the similar structures of the RRL and ACL optimal solutions given by [1], we expect our solution can be extended to improve the running time of an exact algorithm for the strip-rule ACL problem from $O(wn^3)$ to $O(wn^2 \log n)$. As strip-rule ACLs occur in a high percentage of ACL minimization cases [1], this is one reason to study RSRL minimization. Another two reasons are: we believe RSRL minimization is a natural problem, and RSRL minimization is used in the approximation algorithms for RRL minimization.

Our result is obtained by digging deeper into the structure of the dynamic programming of Applegate et al. [1] combined with the use of geometric data structures to speed up the process. We use fast two-dimensional orthogonal range queries, using existing data structures (a time bound of $O(\log n)$ time per query being textbook material [2]).

1.3.1 Flowchart of the paper.

We start the next section with an algorithm, the Pick-Up-Sticks algorithm, which finds a RSRL for the given pattern if one RSRL exists. In fact these "sticks" are equivalent in some sense with RSRL and to minimize the RSRL one can instead minimize the number of sticks that are picked up. A fair number of properties are used to get the equivalence; these are from [1]. If [1] omitted proofs for lack of space. we provide our own proofs. We also introduce some new properties which we use in the algorithm.

Finding the best sticks to pick up, and in which order, is reduced by [1] to finding a shortest path in a directed acyclic graph, the "segment graph" described in the second subsection of the next section. [1] proves that only $O(n^2)$ vertices of the segment graph need to be visited to find a shortest path. These vertices are generated from

previous vertices. The graph distance is computed as part of the vertex generation. Our contribution consists of reducing the time for generating a new vertex to $O(\log n)$.

Section 3 describes our algorithm: we use the same segment graph but group the nodes in "S-groups" and then process all the nodes in a group together. The complexity of the algorithm is analyzed in Section 4. We conclude in Section 5.

2 Preliminaries

We begin by exploring the strategies and tools for finding an optimal RSRL. The Pick-Up-Sticks algorithm will detail the basic structure used to find an RSRL, but not necessarily an optimal RSRL. Then we will explain the concept of equivalence classes, which can be grouped into "segments". All of these concepts will be put together in our algorithm for finding an optimal solution.

2.1 The Maximal Pick-Up-Sticks (MPUS) algorithm

As proposed by Applegate et al.'s paper, we can build an algorithm for finding whether a pattern is a strip-rule pattern and, if so, finding an RSRL that generates it.

2.1.1 The Pick-Up-Sticks (PUS) algorithm.

The Pick-Up-Sticks algorithm of Applegate et al. [1] is an algorithm for generating a RSRL of a pattern P if such a list exists. The idea is to pick up monochromatic rows and columns in order to build the RSRL backwards. Every time we pick a row or column, we color it gray.

Let P be a black and white pattern. We define a column or row in P as being *pseudo-monochromatic* if it is composed of only gray and white cells or of only black and gray cells. Note that monochromatic columns and rows are also pseudo-monochromatic. Moreover, we use *pseudo-black* or *pseudo-white* for a row or column whose cells are only black and gray, or only white and gray respectively. The Pick-Up-Sticks algorithm builds an RSRL as follows:

While there are still black cells, choose a pseudo-monochromatic column or row and color all its cells gray. After a row or column is colored gray, add a rectangle that corresponds to covering that row or column with whatever non-gray color was left in that row or column to the beginning of our RSRL. Note that, in our problem definition, we begin applying RSRLs with a white grid, so we can stop picking up sticks when no black cells exist. If, on the other hand, we modify the problem to begin with a grid of some fourth color, then we would only stop picking up sticks when all cells are gray. There may be a difference of one rule between the optimum solutions to these two problems (white initial grid or some fourth color initial grid) with the same target pattern. For the sake of symmetry, from now on we use this *modified version* of the problem. The method works with minor modifications for the original version.

The algorithm may not succeed, as it may not find a pseudo-monochromatic column or row. If the algorithm does end in an all-gray grid, then the algorithm has "picked up" the rectangles of the RSRL in reverse order. Fig. 4 gives a representation of one possible execution of the Pick-Up-Sticks algorithm and Fig. 5 shows the generation of the original pattern from the resultant RSRL.

The Pick-Up-Sticks algorithm is guaranteed to generate an RSRL if one exists (see below), but it is not necessarily an optimal RSRL of minimum length. At any stage of the algorithm there could be more than one option on which row or column to pick up. Different choices can lead to different sizes of RSRLs. In the next sections we will discuss on how Applegate et al.'s paper narrows down the number of choices for each stage in order to find an optimal RSRL.

Theorem 1 (part of Theorem 3.1 of Applegate et al. [1]) *A black and white pattern P is a strip-rule pattern if and only if the Pick-Up-Sticks algorithm results in a grid with all of its cells gray. In that case, the reverse list of picked up rows and columns is an RSRL for P.*

Since Theorem 3.1 of [1] is not proven in their conference version, we include a proof for the sake of completeness.

Proof. Let us separate the proof into the two following parts:

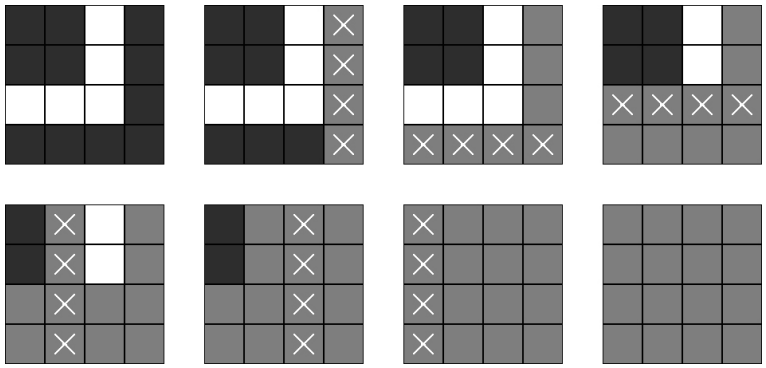

Figure 4: The execution of the Pick-Up-Sticks algorithm on a strip-rule pattern.

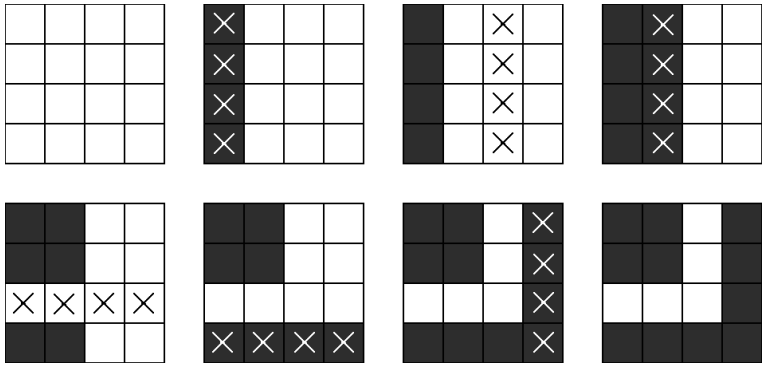

Figure 5: The RSRL generated by the execution of the PUS algorithm on the pattern of the Fig. 4.

- (P is a strip-rule pattern) $\rightarrow$ (The Pick-Up-Sticks algorithm results in an all-gray grid)

  Suppose P is a pattern that can be generated by an RSRL $S = (s_1, s_2, \cdots, s_m)$ of size m. Suppose we have picked up some sticks. Let s_i is the highest-index rule from S that still has a row/column c such that c is not all-gray. If there is no such s_i, then we must have picked up everything. For every integer j, $i < j \leq m$, all the rows/columns of s_j must have already been entirely colored gray. That means that all cells of c that belong to a rule s_j, with $i < j \leq m$, are gray. The cells of c that are not gray must have the same color as the rule s_i, as they will not be overwritten by any rule s_j with $i < j \leq m$. Hence, c must be pseudo-monochromatic and then the Pick-Up-Sticks may proceed. As a result, the algorithm never gets stuck until the all-gray grid is reached.

- (The Pick-Up-Sticks algorithm results in an all-gray grid) $\rightarrow$ (P is a strip-rule pattern)

  The reverse list of the picked-up rows and columns is a RSRL since on every iteration of the algorithm we only pick up strips of P. If the Pick-Up-Sticks algorithm results in an all-gray grid, every row and column of P has been picked up by the end of the algorithm. Hence, the reversed list of picked up rows and columns is an RSRL that generates P, making P a strip-rule pattern.

This ends our proof. ■

The same proof works (the number of rules may differ by one) if the Pick-Up-Sticks algorithm results in a grid with none of its cells black.

2.1.2 Improving by picking maximal sticks.

One important observation by Applegate et al.'s paper on the Pick-Up-Sticks (PUS) algorithm is that there can be no harm in always using maximal strip-rules. A maximal strip-rule is a strip-rule that picks up a maximal contiguous sequences of pseudo-monochromatic rows or columns. Following Applegate et al. [1], we call such a contiguous sequence a *block*.

This is possible because replacing a non-maximal strip-rule by the maximal one that contains it does not adversely affect the pseudo-monochromaticity of any later rule. Hence, we may always use the Maximal Pick-Up-Sticks (MPUS) algorithm instead of the Pick-Up-Sticks (PUS) algorithm. In this new algorithm, once we choose the pseudo-monochromatic column (or row) to pick, we pick the maximal contiguous set of pseudo-monochromatic columns (or rows, respectively) that contains it.

2.2 Equivalence classes

We now introduce the concept of equivalence classes of rows, as used by Applegate et al.'s paper. This definition will allow us to give structure on the order in which columns and rows should be picked up during an execution of MPUS in order to find an optimal RSRL.

2.2.1 Definition and properties.

Given a pattern P, we define two rows or columns as being in the same equivalence class if and only if they are both of the same type (row or column) and their cells have the exact same colors in P, in the same order; that is, they are identical.

We will denote a column equivalence class in a pattern P as being C_x, where x is the number of black cells in the original pattern. Analogously, we will denote R_y as being the row class with y black cells in the original pattern. See Fig. 6 for an example of a pattern and its equivalence classes. It follows from the monotonicity property (Theorem 2 below) that all the columns in C_x are identical in all positions, for all x, and the same property holds for all the rows of R_y, for all y. See figures 7 and 8 for intuition on understanding the monotonicity property.

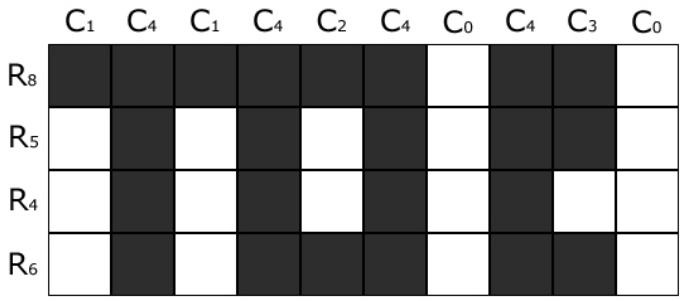

Figure 6: Equivalence classes of a black and white pattern P.

Theorem 2 (part of Theorem 3.1 of Applegate et al. [1]) *(Monotonicity Property) Let P be a black and white pattern. P is a strip-rule pattern if and only if the two equivalent properties hold:*

(i) For any color $c \in \{black, white\}$, the rows of P are hierarchical: given any two rows of P, the set of columns where c is present in the first row either contains or is contained in the set of columns where c is present in the second.

(ii) The same property of item (i) holds with the roles of row and column switched.

This theorem will allow us to order the columns and rows of a strip-rule pattern P in a non-decreasing sequence with respect to the number of black cells. Once the i-th black cell is achieved in the sequence, that i-th cell will remain black until the end of the sequence.

Since Theorem 3.1 of [1] is not proven in their conference version, we include a proof for the sake of completeness.

Proof.

Let us begin our proof by verifying that items (i) and (ii) of the Monotonicity Property are equivalent.

I.) $(i) \leftrightarrow (ii)$

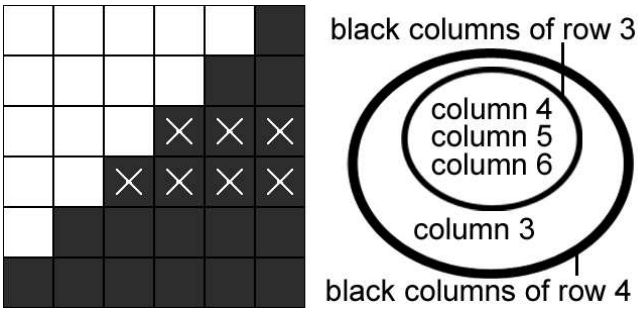

Figure 7: A pattern that follows the Monotonicity Property

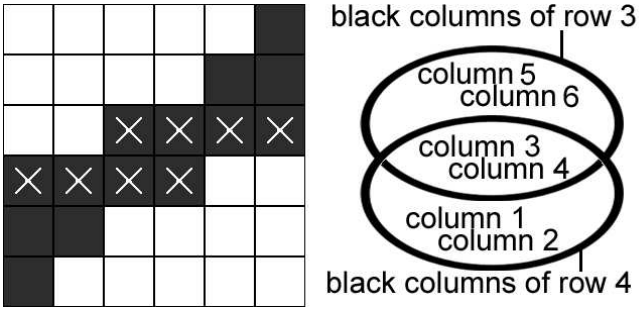

Figure 8: A pattern that does not follow the Monotonicity Property

Let us start by proving that $\neg(i) \rightarrow \neg(ii)$. Suppose (i) does not hold. Then, given $c \in \{black, white\}$, there are two rows A and B , $A \neq B$, of P that do no follow the property mentioned above. Without loss of generality, let's say $c = black$. Since one set is not contained into each other, it must be the case where both sets contains a element that the other set does not have.

This implies that both of the following are true at the same time:

(a) There is a column that has a black cell on row A and a white cell on row B.

(b) There is a column that has a black cell on row B and a white cell on row A.

For reference, let's call this case a checkerboard case; Fig. 9 provides an example.

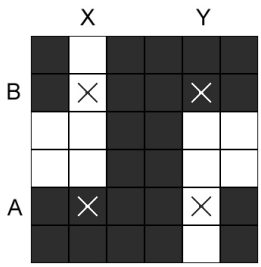

Figure 9: Example of a checkerboard case, where (i) does not hold.

Let X and Y be one of those columns described in (a) and (b) respectively. The set of rows where black is present in X has A as one of its rows and the set of rows where black is present in Y has B as one of its rows. Hence, those sets are not contained into each other. As a result, (ii) does not hold. Thus it must be true that (ii) implies (i). $((ii) \rightarrow (i))$

The same argument can be made with the roles of rows and columns switched, leading to a conclusion that (i) implies (ii). $((i) \rightarrow (ii))$ Then it must be true that (i) and (ii) are equivalent.

Let us continue with the equivalence of P being a strip-rule pattern and the Monotonicity property. Let us divide the proof into two steps:

- (P is a strip-rule pattern) $\rightarrow$ (Monotonicity Property)

  We will prove that $\neg$(Monotonicity Property) $\rightarrow \neg$(P is a strip-rule pattern). Suppose the Monotonicity property does not hold true. By contradiction, suppose that P is a strip-rule pattern and let S be a strip-rule RSRL of size m that generates P such that $S = (s_1, s_2, \cdots, s_m)$.

  Since the Monotonicity property does not hold true, the same argument as before for the checkerboard case can be made. Let s_i be the last rule of S that includes one of the rows or columns of the checkerboard case. Since s_i is the last rule that they appear, no other rule after s_i could have changed the intersection of the rows A, B and columns X, Y of the checkerboard case. Then, because of s_i, either A, B, X or Y must have same color cells in the checkerboard. However, if that happens, the checkerboard case does not hold anymore - a contradiction. Hence, P is not a strip-rule pattern and (P is a strip-rule pattern) $\rightarrow$ (Monotonicity Property) must hold true.

- (Monotonicity Property) $\rightarrow$ (P is a strip-rule pattern)

  Suppose that the Monotonicity property holds. By contradiction, suppose that P is not a strip-rule pattern. By Theorem 1, the MPUS algorithm will not stop with a all-gray grid. Then it must have been stuck at a grid with no pseudo-monochromatic row or column.

  Let A be the row with the largest number of cells colored black. Since this row is not monochromatic, there must be a column Y which intersection with A is a white cell. Y must also not be pseudo-monochromatic. So there must a row B that intersects Y with a black cell. Since B has a black cell in a column that A does not have, than it must be true that the black columns of A are a subset of the black columns of B because of the Monotonicity property. Note also that those sets must not be equal.

  Hence, the number of black cells in B is higher than A, a contradiction. As a conclusion, (Monotonicity Property) $\rightarrow$ (P is a strip-rule pattern) must hold, which concludes our proof.

■

In conclusion, we have shown that patterns generated by RSRLs and patterns that satisfy monotonicity are equivalent. This allows us to label the row and column classes by the number of black cells they contain. For example, every row that has exactly 4 black cells will be identical and can be grouped into the class named R_4. Our algorithm uses this principle to organize and determine the order of classes to be removed. The fact that RSRL patterns have all the properties of monotonicity will be helpful in a number of following proofs.

Note that every row and column belongs to exactly one equivalence class. Also, if two columns belong to an equivalence class in the beginning of the MPUS algorithm, then they will remain of the same class until one of them is picked up by the algorithm, and at that moment, the other column is also available to be picked up. This is since picking up other columns does not change these two columns at all, while picking up rows modify these two columns in exactly the same way. The same property holds for two rows that belong to an equivalence class.

Moreover, if two columns start in different equivalence classes and they are not pseudo-monochromatic during the execution of MPUS, then they have different numbers on non-gray cells. Indeed, if the two columns differ in a row, this row cannot be picked up by MPUS before at least one of the column is picked up, as the row would not be pseudo-monochromatic. The same property holds for two rows that do not belong to the same equivalence class.

We also define an equivalence class as being *active* during the execution of the MPUS algorithm if some member of that class is pseudo-monochromatic but not all gray. As argued above, all the rows/columns of a class that have not been picked up yet are identical. Therefore, we have classes that are active for white (its the rows/columns are pseudo-white) and classes that are active for black (its the rows/columns are pseudo-black) and

We will use the next proposition.

Proposition 3 (Applegate et al. [1], Observation 6. on page 1070) *During the execution of the MPUS algorithm on a black and white strip-rule pattern, there are always exactly two active classes at any given time. The two active classes are either a row and a column class of the same color or both classes of the same kind (rows or columns), being one of each color.*

The intuition on why this happens is that in order for a new class to become active, all the members of an old active class need to be picked up. Since Observation 6 of [1] is not proven in their conference version, we include a proof for the sake of completeness.

Proof. Let A be a row with the largest number of white cells and B be a row with the largest number of black cells. Let X be a column with the largest number of white cells and Y be a column with the largest number of black cells. Note that the intersections of A and B with X and Y cannot be gray as neither of these rows and columns have been picked up.

If the intersection of A and X is black, then A does not have and white cells. Indeed, if X' is a column that intersects A in a white cell, than, by the monotonicity property, X' starts with more white cells than X. As argued above, X' still has more white cells than X, contradicting the choice of X. With A not having white cells, it follows that no white cells exist in the current pattern. Then A and B have the same number of black cells, since we cannot have a column X' with the intersection of X' and A gray and the intersection of X' and B black (X' was not picked up earlier, so the intersection of X' and A cannot be gray). Thus all rows are in the same equivalence class, and all are available for pickup. The symmetric argument with the roles of rows and column reversed shows that X and Y and all the other non-gray columns are also in the same equivalence class, and all pseudo-monochromatic. Thus there are exactly two active classes, one a row class and one a column class, and their members are all pseudo-black.

A symmetric argument, with black and white reversed, holds when the intersection of B and Y is white. In this case, the two active classes are one a row class and one a column class, and their members are all pseudo-white.

Now assume we are in the case where the intersection of A and X is white and the intersection of B and Y is black. Consider the subcase where the intersection of A and Y is white and the intersection of B and X is black. We claim that the classes of A and B are active, one with pseudo-white rows and the other with pseudo-black rows. Indeed, if there is a column Y' whose intersection with A is black, then Y' ends up with more black cells than Y, contradicting the choice of Y. Thus A is pseudo-white. Similarly, if there is a column X' whose intersection with B is white, then X' ends up with more white cells than X, contradicting the choice of X. Thus B is pseudo-black. We continue to argue that no other equivalence classes can be active. First, let A' be a row that was not picked up and is in neither the class of A nor in the class of B. Then A' must have a black cell, in a column where A has a white cell. Also, A' must have a white cell, in a column where B has a black cell. A' is not pseudo-monochromatic. Second, let X' be any column that was not picked up. X' has a white cell at its intersection with row A (A being pseudo-white) and a black cell at its intersection with row B (B being pseudo-black), and thus X' is not pseudo-monochromatic. In conclusion, in this subcase there are exactly two active classes, and one has pseudo-white rows and the other pseudo-black rows.

We continue the case where the intersection of A and X is white and the intersection of B and Y is black. The subcase where the intersection of A and Y is black and the intersection of B and X is white is symmetric with the subcase above, with columns instead of rows. In this subcase there are exactly two active classes, and one has pseudo-white columns (including X) and the other pseudo-black columns (including Y) .

We continue the case where the intersection of A and X is white and the intersection of B and Y is black. Consider the subcase subcase where the intersection of A and Y is black and the intersection of B and X is also black. We claim that the classes of B and Y are both active, both with pseudo-black rows/columns. Indeed, if there is a row A' whose intersection with Y is white, then A' ends up with more white cells than A, contradicting the choice of A. Thus Y is pseudo-black. Similarly, if there is a column X' whose intersection with B is white, then X' ends up with more white cells than X, contradicting the choice of X. Thus B is pseudo-black. We continue to argue that no other equivalence classes can be active. First, let A' be a row that was not picked up and is not in the class of B (A' may be A). A' intersection with Y cannot be white, as otherwise A' ends up with more white cells than A, contradicting the choice of A. Thus A' has a black cell. As A' is in not in the same class as B, it must have a white cell. Thus A' is not pseudo-monochromatic. Second, let X' be a column that was not picked up and is not in the class of Y (X' may be A). we can use a symmetric argument to obtain that X' has a black cell at the intersection with B and a white cell elsewhere, and therefore X' is not pseudo-monochromatic. Thus there are exactly two active classes, one a row class and one a column class, and their members are all pseudo-black.

We continue the case where the intersection of A and X is white and the intersection of B and Y is black. Consider the subcase subcase where the intersection of A and Y is white and the intersection of B and X is also white. By a symmetric argument with the previous subcase (X and Y, A and B, and white and black are interchanged) we obtained that the only active classes are those of A and X, both with pseudo-white rows/columns. This is the last case. ■

2.2.2 Ordering of equivalence classes.

As we pick up all the members of a column class, we change the rows of the pattern. Since the other columns remain intact, the new class to become active must be a row class. By using the same argument for row classes, we see that whenever we pick up all the members of a class, the next class to become active is of the opposite type. This is formally stated below.

Proposition 4 (Implicit in Applegate et al. [1]) *While executing MPUS for a pattern of two colors, if one picks up all columns of an active class and does not make all the grid gray, then a row class for the opposite color becomes active. If one picks up all rows of an active class and does not make all the grid gray, then a column class for the opposite color becomes active.*

Proof. From Proposition 3, we know that there are always two active classes at a time. Picking up a (monochromatic) column active class will not change the monochromatic property of any other column class. Hence, the class that will become monochromatic needs to be a row class. The same thing holds with the roles of rows and columns switched.

Also, the new active class must be of the opposite color because if it was not, the class becoming active would need to be already active before, since the color being removed is the same as the other cells of that class, which means that it would need to be monochromatic already. ■

Let N_C be the number of column equivalence classes, and N_R be the number of row equivalence classes. We can order the equivalence classes of both rows and columns in a given pattern P by the number of black cells on it. In that ordering, we can see from the proposition below that all the rows that two consecutive column classes differ form exactly an equivalence class. This holds analogously for columns.

Proposition 5 (Implicit in Applegate et al. [1]) *Let i be such that $1 \leq i < N_C$ and j be such that $1 \leq j < N_R$. Let x_i be the number of black cells in each column from the i^{th} column equivalence class and let y_j be the number of black cells in each row from the j^{th} row equivalence class.*

a) Let S_i be the set of rows that have white cells on the intersections of the columns of C_{x_i} and black cells in the intersection of the columns of $C_{x_{i+1}}$. Then S_i is a row class.

b) Let S_j be the set of columns that have white cells on the intersections of the rows of R_{y_j} and black cells in the intersection of the columns of $R_{y_{j+1}}$. Then S_j is a column class.

Proof. [Proof of item (a)] Let i be such that $1 \leq i < N_C$ and S_i be the set of rows described in the statement of the proposition. Let $\alpha, \beta \in S$. By contradiction, suppose α and β differ from each other. Let γ be one of the columns that they differ and $k \in \mathbb{N}^*$ be such that C_{x_k} is the class of γ. Without loss of generality, assume C_{x_k} has a black cell on the row α and a white cell on the row β.

Because of monotonicity, the fact that C_{x_k} has a black cell on α and C_{x_i} has a white cell on α implies that $x_k > x_i$ must hold true. At the same time, the fact that C_{x_k} has a white cell on β and $C_{x_{i+1}}$ has a black cell on β implies that $x_k < x_{i+1}$ must hold true. Hence, $x_i < x_k < x_{i+1}$. However, $x_i < x_k < x_{i+1}$ implies that $i < k < i+1$, which is a contradiction because $k \in \mathbb{N}^*$.

Hence, α and β do not differ from each other, which implies that all the rows of S_i are in the same row class. And if we take another row from the same class, we can see it belongs to S_i. Thus S_i is a row class.

The proof of item (b) follows analogously with the roles of rows and columns switched. ■

Proposition 5 can be used to prove:

Proposition 6 *The number of row classes and the number of column classes differ by at most one; that is $|N_R - N_C| \leq 1$.*

Proof. Let $A = max(N_R, N_C)$ and $B = min(N_R, N_C)$. Note that $|N_R - N_C| = A - B$. If $N_R = N_C$, we have that $|N_R - N_C| = |0| = 0 \leq 1$.

Now, suppose $N_R \neq N_C$. Assume $A = N_C$. It follows that $B = N_R$.

Let i_1 be such that $1 \leq i_1 < N_C$. Let S_{i_1} be the set of rows that have white cells on the intersections of the columns of $C_{x_{i_1}}$ and black cells on intersection of the columns of $C_{x_{i_1+1}}$. According to Proposition 5 we have that S_{i_1} is a row class.

Let i_2 be such that $1 \leq i_2 < N_C$ and $i_2 \neq i_1$. Let S_{i_2} be the set of rows that have white cells on the intersections of the columns of $C_{x_{i_2}}$ and black cells on intersection of the columns of $C_{x_{i_2+1}}$. According to Proposition 5(a) we have that S_{i_2} is a row class.

If $i_2 < i_1$, we have that $x_{i_2+1} \leq x_{i_1}$, which means that the rows of S_{i_2} must have a black cell on its intersections with $C_{x_{i_1}}$ due to monotonicity. This implies that $S_{i_2} \neq S_{i_1}$ because they differ on $C_{x_{i_1}}$.

If $i_2 > i_1$, we have that $x_{i_2} \geq x_{i_1+1}$, which means that the rows of S_{i_2} must have a white cell on its intersections with $C_{x_{i_1+1}}$ due to monotonicity. This implies that $S_{i_2} \neq S_{i_1}$ because they differ on $C_{x_{i_1+1}}$.

Thus, for every i such that $1 \leq i < N_C$ we have a different S_i that is a row class. From that, we have that $N_R \geq N_C - 1$, since $N_C - 1$ is the number of different possibilities for i. Hence, $N_C - N_R = A - B \leq 1$.

Now assume $A = N_R$. It follows that $B = N_C$. Using the same reasoning with Proposition 5(b), we have that $N_R \geq N_C - 1$. Hence, $N_R - N_C = A - B \leq 1$.

Therefore, it is true for any case that $|N_R - N_C| \leq 1$. ■

Using that information, we can set up the following ordering for the equivalence classes:

1. Start with the column class with only white cells. If this class does not exist, then start with the row class with only black cells (the existence of this row class follows immediately from the monotonicity property – Theorem 2).

2. Alternate columns and rows, putting the columns in ascending order of black cells and the rows in descending order of black cells.

Let $C_{x_0}, C_{x_1}, \ldots, C_{x_{(N_C-1)}}, C_{x_{N_C}}$ be the ascending ordering of column classes by the number of black cells. Let $R_{y_0}, R_{y_1}, \ldots, R_{y_{(N_R-1)}}, R_{y_{N_R}}$ be the ascending ordering of row classes by the number of black cells. Using the rules above, we should get an ordering such as $C_{x_0}, R_{y_{N_R}}, C_{x_1}, R_{y_{(N_R-1)}}, \ldots, R_{y_1}, C_{x_{(N_C-1)}}, R_{y_0}, C_{x_{N_C}}$ (alternating row and column classes). This array may start and/or end with row equivalence classes, instead of columns as above (four possibilities in total; for example, the array can be $C_{x_0}, R_{y_{N_R}}, C_{x_1}, R_{y_{(N_R-1)}}, \ldots, R_{y_1}, C_{x_{(N_C)}}, R_{y_0}$). Let $N = N_C + N_R$.

Proposition 7 *There is an array E_1, E_2, …, E_N of equivalence classes that respect the following property: Given a and b such that $1 \leq a < b \leq N$, if E_a and E_b are the active classes at some moment during the execution of the MPUS algorithm, then for every c such that $1 \leq c < a$ or $b < c \leq N$, E_c has already been picked up. Also, if $b = a + 1$, then either $E_a \in R$ and $E_b \in C$ or $E_a \in C$ and $E_b \in R$. Moreover, none of the columns/rows of a class E_c for $a < c < b$ are pseudo-monochromatic.*

Proof. We know from Proposition 3 that there are always exactly two active classes during an execution of PUS. We combine this idea with the concept of monotonicity in order to create an organization on the possible orders the classes are picked up in. For the original pattern, let A and B be the two original active classes. If A, B are both column classes, we must have that one is white and the other is black. Assume without loss of generality that A is white and B is black. Then we have that $A = C_{x_1} = C_0$ and $B = C_{x_{N_C}} = C_{n_R}$. To build the array, we will assign $E_1 = A = C_{x_1}$ and $E_N = B = C_{x_{N_C}}$ since A and B need to be on both ends of the array because both are active even if no other classes have been picked. If we pick up C_{x_1}, a row class will need to become active. Also, this class will need to be the row class with the largest number of black cells due to monotonicity and Proposition 5. Hence, we should assign $E_2 = R_{y_{N_R}}$. Following the same reasoning, the next member will need to be the column class with the lowest number of black cells after C_{x_1}. Hence, we should assign $E_3 = C_{x_2}$. Analogously, if we pick up $C_{x_{N_C}}$, the row class with the least amount of black cells will become available, this being R_{y_1}. Hence, we should assign $E_{N-1} = R_{y_1}$, $E_{N-2} = C_{x_{N_C-1}}$ and so on.

Therefore we can build this array assigning $E_{2i-1} = C_{x_i}$ and $E_{2j} = R_{y_{N_R-j+1}}$ for every $1 \leq i \leq N_C$ and $1 \leq j \leq N_R$ with $E_N = C_{x_{N_C}}$. Given a and b such that $1 \leq a < b \leq N$, we have that if E_a and E_b are the active classes, every class that comes before E_a or after E_b must have already been picked up. Also, note that two consecutive elements of the array are not of the same type of equivalence classes (row/column). Note also that that $E_{2N_C-1} = C_{x_{N_C}} = E_N$, which means that $N = N_R + N_C = 2N_C - 1$ and $N_R + 1 = N_C$, which agrees with Proposition 6.

If A, B are both row classes, we may build this array analogously assigning $E_{2j-1} = R_{y_j}$ and $E_{2i} = C_{x_{N_C-i+1}}$ for every $1 \leq i \leq N_C$ and $1 \leq j \leq N_R$ and $E_N = R_{y_{N_R}}$. For this case, $N_C + 1 = N_R$.

For A and B being of different types, let's assume the color of the active classes, which is the same, is white. We can build this array assigning $E_{2i-1} = C_{x_i}$ and $E_{2j} = R_{y_{N_R-j+1}}$ for every $1 \leq i \leq N_C$ and $1 \leq j \leq N_R$ with $E_N = R_{y_1}$. For this case, $E_{2N_R} = R_{y_1} = E_N$ and $N = N_R + N_C = 2N_R$ and $N_R = N_C$, which agrees with Proposition 6. If the color is black, we can build this array assigning $E_{2j-1} = R_{y_j}$ and $E_{2i} = C_{x_{N_C-i+1}}$ for every $1 \leq i \leq N_C$ and $1 \leq j \leq N_R$

with $E_N = C_{x_{N_C}}$. For this case, $E_{2N_C} = C_{x_{N_C}} = E_N$ and $N = N_R + N_C = 2N_C$ and $N_R = N_C$. By construction it is easy to see that a column/row of a class E_c must not be monochromatic while the two active classes are E_a and E_b for $a < c < b$. From our construction and definitions, the only way for a class to become active, is to have either the class immediately to its left or right in the hierarchical array already removed. This therefore cannot be true while $a < c < b$ since we know the only classes removed are those that come before a or come after b. ■

We will call this ordering the *hierarchical array* of the pattern P. The hierarchical array is a useful extension of Observation 5 on page 1070 of Applegate et al. [1].

We will also refer to this ordering of equivalence classes as $E_1, E_2, \ldots, E_N$. That is, if the hierarchical array starts with a column, then $E_1 = C_{x_0}$, $E_2 = R_{y_{N_R}}$, $E_3 = C_{x_1}$, and so on, and if the hierarchical array starts with a row, then $E_1 = R_{y_{N_R}}$, $E_2 = C_{x_0}$, $E_3 = R_{y_{(N_R-1)}}$, and so on. Note that N_C, the number of column classes, is at most n_C, the number of columns, and the same property holds for rows.

2.2.3 Embedded rows and columns.

We can start improving the MPUS algorithm by introducing the concept described by Applegate et al.'s paper as embedded rows and columns. This will allow us to pick up more rows and columns using the same number of rectangles.

First, we will define the concept of an embedded column or row at a given stage of the MPUS algorithm. Given a point during the execution of the MPUS algorithm, let b be a column of an active equivalence class B. Let a_1 be the first column to the left of b that does not belong to B and has not yet been picked up. Similarly, let a_2 be the first column to the right of b that does not belong to B and has not yet been picked up. We say that b is *embedded* in equivalence class A if both a_1 and a_2 belong to that class A. The definition is analogous for rows. Note that columns can be embedded only in column classes and rows only in row classes. See Fig. 10 for an example.

Figure 10: Example of a pattern P where one member of C_0, the second column, is embedded in C_2.

If during the execution of the MPUS algorithm we have the option to pick up two column-blocks that embed a set of columns of another active class, then we can pick up the two column-blocks that embed the third one with two rectangles and then the embedded one, totaling three rectangles. However, it is never worse to pick the embedded block first with one rectangle, and then pick the two blocks that were previously embedding it with only one rectangle, thus using a total of two rectangles for the same set of columns. As an example, in Fig. 10, one cannot benefit by picking up the first and the third column, when one can pick up first the second column, followed by picking up the block of the first three columns.

2.3 Segments

During MPUS, there are always two active classes. A "segment" succinctly represents the blocks that are available for pick up while a certain pair of classes are active. These segments can be translated into nodes of a graph (the *segment graph* that is discussed in detail later), that can be used for finding an optimal solution, as described below. Note that our representation of segments is a slightly condensed version of the one described in the original Applegate et al.'s paper [1]. The difference is that the original definition is a five tuple with two redundant elements.

2.3.1 Definition and properties.

Whenever a new class becomes active during the execution of the MPUS algorithm on a pattern P, we will define a *segment* S as being a tuple of three elements (E_x, E_y, U), where E_x and E_y are the active equivalence classes and U is the subset of members (rows or columns) left unpicked of E_y ($U \subseteq E_y$).

Proposition 8 *Let E_1, E_2, …, E_N be the hierarchical array of a pattern. Given a and b such that $1 \leq a < b \leq N$ it is possible to know what colors correspond to each active class of $\{E_a, E_b\}$.*

Proof. Let a and b be such that $1 \leq a < b \leq N$. There are four possible cases for the hierarchical array (it can start with a row class or a column class, and it can end with a row class or a column class). In all cases, the number of black cells in columns is increasing and the number of black cells in rows is decreasing.

When we pick from either sides, the color alternates for each item picked up as well as its type. Hence, the color of E_a will always match the color of E_1 if they have the same type and differ if they are from different types. The same will happen with E_b and E_N. So in all cases, if E_a is a column class, then E_a's columns are pseudo-white and if E_a is a row class, then E_a's rows are pseudo-black. For E_b, it is reversed: if E_b is a column class, then E_b's columns are pseudo-black and if E_b is a row class, then E_b's rows are pseudo-white. ■

It is a consequence of this proposition the fact that having a segment $S = (E_{k_1}, c_{k_1}, E_{k_2}, c_{k_2}, U)$, $U \subset E_{k_2}$, is redundant. We may start referring to a segment as only $S = (E_{k_1}, E_{k_2}, U)$, $U \subset E_{k_2}$.

We can represent the sequence of actions of the MPUS execution as a sequence of segments, as described in this paragraph. Given the pick-up order obtained through the execution of the MPUS, let's create a segment (E_0, E_N, E_N) for the starting state of pattern P and a segment (E_x, E_y, U), $U \subseteq E_y$, for every time a new class E_x becomes active; we say that the RSRL *passes through* the segment whenever such a segment becomes active during the MPUS execution corresponding to the RSRL.

Since E_x is the new class, all of the members of E_x are still in the pattern. E_y, in the other hand, is a class that was already active. It may have some of its members already picked up. We store that information in U by maintaining the members that have not been picked up yet. Note also that any classes that are between E_x and E_y in the hierarchical array of P have not become active yet, resulting in the fact that all of their members are still there. At the same time, members that come before and after the interval delimited by E_x and E_y have all been picked up. Hence, for each one of those segments that are created whenever a class becomes active it is possible to reconstruct the pattern at that moment of the execution of the MPUS algorithm.

2.3.2 Segment branching.

Applegate et al.'s paper [1] shows that if we are looking for an optimal solution, then we can narrow down the number of segments significantly by reducing the number of options into which a segment can "branch" to two. By branching we mean, given a segment, find the other nodes in the segment graph that can be reached by traversing one edge.

Suppose that a minimum-length RSRL for a strip-rule pattern P passes through the segment $S = (E_x, E_y, U)$ and that to get in MPUS to the next segment, if it exists, we pick up all members of E_x (possibly picking up some members of $U \subseteq E_y$) while at least one member of U remains active.

- If $U \subseteq E_y$ has a member that is not embedded in E_x, then there exists a minimum-length RSRL that passes through the segment $S = (E_x, E_y, U)$ and that to get to the next segment you first pick up (in maximal blocks) all the members of $U \subseteq E_y$ that are embedded in E_x, followed by picking up (in maximal blocks) all members of E_x.
- If all members of $U \subseteq E_y$ are embedded in E_x, then there exists another minimum-length RSRL that passes through the segment $S = (E_x, E_y, U)$ and that to get to the next segment you pick up all the members of $U \subseteq E_y$ (thus, finishing picking up every member of E_y in the pattern). Note that in this case, one gets to the next segment by picking up all the members of $U \subseteq E_y$ before picking up all the members of E_x; in other words E_y is picked first.

The property above follows Proposition 9 below. The same property holds with the roles of E_x and U switched. In this case, we assume the next segment from S, if it exists, is originally reached by picking up all members of $U \subseteq E_y$.

Intuitively, this property happens because we can always pick embedded members at no extra cost, if one is to pick all the blocks of the embedding class. Note that, if all members of $U \subseteq E_y$ are embedded in E_x, we completely pick up E_y before E_x and we can apply the property with roles of E_x and U switched, resulting in all the members of E_x embedded in E_y being picked up first, in maximal blocks, followed by picking up in maximal blocks all the members of U.

Proposition 9 (statement 2(a) of Lemma 4.2 of Applegate et al. [1]) *Suppose we have two sets U_x and U_y of active blocks of the two active classes E_x and E_y. Suppose there is an optimal solution that completely picks up E_x before completely picking up E_y. Then there also exists an optimal solution that picks up all members of U_y embedded in E_x before completely picking up E_x.*

Since Lemma 4.2 of [1] is not proven in their conference version, we include a proof for the sake of completeness.

Proof. This follows closely from the definition of embedded members, which can be effectively picked up at zero net cost along with the blocks embedding them. This is easy to observe in an example: consider the leftmost three columns of Fig. 10. Here we see a member of C_0 (the white column) embedded in two blocks of C_2. We could pick up the embedding blocks using two rules, or we could first pick up the embedded block followed by picking up both embedding blocks using only one wider rule. The overall result is still two rules which pick up all members of the embedding class. This is, without loss of generality, either a helpful or irrelevant action. So in our solution we will always choose to remove the embedded blocks before the embedding blocks. ■

Moreover, suppose we have two sets U_x and U_y of active blocks of the two active classes E_x and E_y. To finish picking up blocks (except at the very end), we must reach the situation that either E_x or E_y are not active anymore; say E_x becomes not active, while E_y is still active. Picking up any block from the class E_y that is not embedded in E_x can be done, without loss of optimality, later. So we can assume that only the members of U_y that are embedded in E_x are picked up while E_x is active.

Note also that for every segment we can pick the latest class that became active or the oldest, a total of two options. We can then say that every segment will branch into two other segments. In both cases, we can determine the next segment reached by eliminating every embedded member of the class not being picked up, followed by eliminating every member of the class being picked up. The only exception to this occurs when all members of U_x are embedded in E_y or all members of U_y are embedded in E_x. In the former exception case, based on the discussion above, we can assume an optimal solution picks up all in maximal sticks U_x, and in the latter case we can assume an optimal solution picks up all in maximal sticks U_y. In both exception cases, the original segment gives rise to only one segment, instead of branching into two segments.

Note that there could be a segment that does not have any next segment. In this case, the segment is a segment at the very end of MPUS execution. If we pick up any of either class, then we will obtain an all-gray grid and the algorithm will be over. Also, note that for this to happen we must have that E_x and E_y are next to each other in the hierarchy array, and picking-up one of them will also result in picking-up the other one.

2.3.3 Segment graph.

We build a "segment graph" using the segments as nodes, as described by the following structure proposed by Applegate et al. [1]:

- Start with the segment (E_0, E_N, E_N), that represents the grid as it is in the very beginning of the MPUS algorithm. This will be the starting node.
- For every node (segment) in the graph, generate the two segments (possibly one, as described in the previous subsection) that it branches into and add a directed edge from it to the new generated nodes. There will be one segment for each of the two options of picking up the newest or the oldest of the two classes in the segment. In both cases, we can determine the next segment reached by eliminating every embedded member of the class not being picked up, followed by eliminating every member of the class being picked up.
- Define the cost for each edge as the number of rules in the MPUS algorithm used to go from one segment to another.
- Create an artificial node corresponds to an all-gray grid. This will be the end node of the graph. Every time a segment has its two classes adjacent to each other in the hierarchy array, create an edge from it to this all-gray grid node. This edge should have cost one (since we solve the modified version); if we were to solve the original version, then this edge should have cost zero if the members of the classes of the segment were white and one otherwise.

Because of what was discussed in Subsection 2.3.2, at least one of the paths from the starting node to the end node of the graph will be a minimum-length RSRL. Note that having (E_N, E_0, E_0) as the starting node will lead to a graph that represents the same possible steps for the MPUS because it also represents the same starting pattern. Note also that some nodes may be reached by more than one node. Every node will only branch into nodes that correspond to patterns with a smaller number of total of columns and rows. This implies that there are no cycles in this graph.

We say that one segment *reaches* another if there is a path in this graph from that segment to the other one. The distance from one segment to another is the sum of the costs of the edges that compose the path between them, if such a path exists. One very important property of this graph is that if two segments (E_x, E_y, U) and (E_x, E_y, U') are reachable from the starting node of the graph, then either $U \subseteq U'$ or $U' \subseteq U$.

Lemma 10 (Containment Lemma – Lemma 4.3 of Applegate et al. [1]) *Let S and S' be two segments on the same equivalence classes (in the same order). Then either $U \subseteq U'$ or $U' \subseteq U$.*

Intuitively, this follows from the fact that we have a strict ordering on the equivalence classes that could embed columns in U. See Applegate et al.'s paper [1] for a rigorous inductive proof of this lemma. We will see (and it was proven in [1]) that this implies that the number of nodes in this graph is $O(n^2)$, where n is the number of rows and columns in the original pattern.

3 Algorithm

The flow of our algorithm closely follows that of the $O(n^3)$ algorithm given by Applegate et al. We similarly create a segment graph of the reachable states in a MPUS of the pattern. However, we create this graph faster by grouping similar segments into S-groups, which can be processed together.

3.1 Setup

We begin by reading the input pattern into an equivalence class list and creating the first node in our segment graph.

3.1.1 List of equivalence classes.

We are given as input an $n_R \times n_C$ grid of black and white cells. Assign each of the rows an index from 1 to n_R from top to bottom, similarly each column, an index from 1 to n_C from left to right. Group rows and columns into equivalence classes and order the classes by number of black cells as previously described in the Subsection 2.2. Precisely, construct the hierarchical array:

$$C_{x_0}, R_{y_{N_R}}, C_{x_1}, R_{y_{(N_R-1)}}, \ldots, R_{y_1}, C_{x_{(N_C-1)}}, R_{y_0}, C_{x_{N_C}}$$

(this array may start and/or end with row equivalence classes, instead of columns as above).

We will need to determine from a range of column indexes if there are columns also in a range of equivalence classes, and the similar question with rows instead of columns. A data structure such as an orthogonal range tree [2] accomplishes these queries in $O(\log n)$ time with $O(n \log n)$ space and setup time. We call this data data structure *column range tree*, and we also keep a similar *row range tree*.

3.1.2 S-groups.

Define an S-group (E_1, E_2) (for equivalence classes E_1 and E_2, where the order of E_1 and E_2 in the tuple above matters) to be a collection of all segments $\{C_1, C_2, U'\}$ such that $C_1 = E_1$ and $C_2 = E_2$. To completely represent each of these segments, an S-group (E_1, E_2) maintains a list of segments S and a master list U. This list U will be used to keep track of all the individual lists of all the segments of this S-group (the list U' of each segment), as described below. This allows each segment to be stored in constant space, while the S-group gets stored in $O(|E_1| + |E_2|)$ space.

The list U contains the index of every member of E_2 ordered such that for every segment $\{E_1, E_2, U'\}$ in the S-group, the first $|U'|$ members of U comprise the same set as U'. Such an ordering is guaranteed to exist by the Containment Lemma 10. We generate this ordering by noting that indexes appearing in more segments appear earlier in the list U.

The list S of segments maintained by the S-group, keeps these segments ordered by the size of their set U'. Each segment s_i will have three values, $u(s_i)$, $d(s_i)$, and $p(s_i)$. Set $u(s_i) = |U'|$. The field d gives the minimum number of sticks required to reach the state of segment starting from the original pattern. We want to maintain the shortest path to each node, so we use the value p to store a reference to the segment right before this one on this path.

3.1.3 Origin S-group.

We begin with the S-group (E_1, E_2) with $U = \tilde{E_2}$ and S comprised of a single segment $(u, d, p) = (|\tilde{E_2}|, 0, null)$, where $\tilde{E}$ represents the set of all the members of equivalence class E. The classes E_1 and E_2 are the first and last members of the hierarchical array. We will explore outwards from this S-group in a breadth-first fashion, enumerating all reachable segments. The graph on the S-groups is guaranteed to be acyclic because the number of picked up classes is strictly increasing along each edge. This graph with S-groups implicitly stores all the useful information of the segment graph.

3.2 Finding the next S-group

Suppose we have some S-group (E_1, E_2) with associated lists U and S. Without loss of generality we will assume E_1 to be a column class. We wish to generate all S-groups reachable from this S-group. The two S-groups immediately reachable from this S-group correspond to completely removing either class E_1 or E_2.

To determine the cost of removing a class E_i from a segment s_j, we count how many members of E_{3-i} can be picked up for free, as described in detail later. If two members of E_i may be picked up with the same stick assuming we had E_1 and E_2 as active classes and all the members of E_{3-i} have been picked up, then we will call the two members *contiguous*. It follows from Proposition 7 that a pair of columns is contiguous in an S-group (E_1, E_2) if every column between them is either already picked up or of type E_1 or E_2.

Before the class E_i can be removed from a segment (E_1, E_2, U') (all the members of the class being picked up), we pick up every embedded member of E_{3-i}. To determine the cost of moving to the next segment, we must count how many of the embedded columns are also contiguous. Then any embedded columns must be removed from either the set U' (if E_1 is removed) or E_1 (if U' is removed) in the next segment and, by doing this for all segments of an S-group, in the next S-group. In fact, we process one S-group at a time, and obtain segments of another at most two S-groups, as explained below.

Because E_1 and E_2 are not symmetric (E_2 comes with its set U), we will describe the process of removing classes E_1 and E_2 separately.

3.2.1 Remove E_1.

Let E_3 be the equivalence class adjacent to E_1 in the equivalence class list, which has not yet been removed. The S-group corresponding to the result of removing E_1 will be (E_3, E_2) with lists $\hat{U}$ and $\hat{S}$ (with their associated $u(.)$, $d(.)$, and $p(.)$ fields).

Count Contiguous Consecutive Pairs of Columns of E_1. To determine the distances to the segments of the next S-group, we must count the number of contiguous E_1 ranges.

To do so, we must determine the range of equivalence classes which have not yet been picked up. Let E_1 be the column class C_a. If E_2 is also a column class, then call it C_b. If E_2 is a row class, then C_b is the column class adjacent to E_2 in the equivalence class list that has already been picked up. We assume $a < b$, with the other case being symmetric. By Proposition 7, a column class C_i other than C_a has not been picked up if and only if $i \in (a, b)$.

For each pair of adjacent members of E_1 we check to see if they are contiguous. Precisely, for each member $e_i \in E_1$, we query our column range tree for the existence of members in the range $[a+1, b-1]$ with index in the range $[e_i, e_{i+1}]$, where e_{i+1} is the column of E_i that has the smallest index among those with index higher than e_i (e_{i+1} is the next column of E_i after e_i). Let c be the number of pairs of consecutive members of E_1 which are also contiguous. The value $(|E_1| - c)$ corresponds to the number of sticks required to completely remove E_1 after any embedded columns of E_2 are picked up.

Get Embedded E_2 Blocks. If E_2 is also a column class, then we must count and remove any members of E_2 embedded in E_1. Similar to the way we checked that columns of E_1 were contiguous, we will check if adjacent columns of E_1 are both contiguous and contain columns of E_2. In order to accurately keep track of the cost to remove these embedded columns we must give each embedded column a tag based on which two columns surrounded the embedded column. Two members of E_2 can be picked up with the same stick if and only if they have the same tag.

When an embedded column is found, tag it with the index of the E_1 column to its left. Then add this column to a list, B, sorted by the column's index. After we have found all of the embedded columns, we are ready to generate $\hat{U}$ and $\hat{S}$ (with their associated $u(.)$, $d(.)$, and $p(.)$ fields).

Build the Next S-group. Begin with $\hat{U}$ as an empty list. We also need a set T to keep track of which tags have been accounted for. Iterate through U, searching for each member m of U in the set of embedded columns, B. If $m \notin B$, then append m to the end of $\hat{U}$. Otherwise add m's tag value to the set T if it is not already there. Once we have checked all of the members U' in a segment s_i (U' being the first $u(s_i)$ elements of U) , add a new segment $\hat{s}_i$ to $\hat{S}$ with the following values:

$$u(\hat{s}_i) = |\hat{U}|,$$
$$d(\hat{s}_i) = d(s_i) + |T| + |E_1| - c,$$
$$p(\hat{s}_i) = s_i$$

(note that $|\hat{U}|$ and $|T|$ are computed for the sets $\hat{U}$ and T exactly when having finished processing the last element of U' while iterating through U; $\hat{U}$ and T can change later on). If $\hat{S}$ already contains a segment $\hat{s}_j$ with $u(\hat{s}_j) = u(\hat{s}_i)$, then keep only the segment with shorter distance d in the set $\hat{S}$, and remove the other one. As an aside, one can see that if we do not remove duplicates, if two segments from the same S-group s_i and s_j have $u(s_i) > u(s_j)$, then $u(\hat{s}_i) \geq u(\hat{s}_j)$, which is used in the proof of the Containment Lemma 10 of [1] (precisely, Case 2a).

3.2.2 Remove E_2.

Let E_3 be the equivalence class adjacent to E_2 in the equivalence class list, which has not yet been removed. The S-group corresponding to the result of removing E_2 will be (E_3, E_1) with lists $\hat{U}$ (whose elements are members of the class E_1) and $\hat{S}$.

Count Contiguous Consecutive Pairs of Rows/Columns of E_2. To count contiguous ranges of E_2, we must count the contiguous ranges within each segment separately. We use a counter c, initially set to 0. Fortunately, if a pair of rows/columns is contiguous in one segment then it is also contiguous in all larger (with bigger value of u) segments. For each index $u_j \in U$, insert u_j into a sorted list of indexes, I. Get the predecessor and successor of u_j in I and check if these ranges from u_j to its neighbors are contiguous. If either of these ranges exists and is contiguous (which we check with column range trees as in the previous subsection), then this column can be picked up for free. If not, we increment our cost counter c. Once we have iterated over the first $u(s_i)$ blocks, we save the state of our counter in a value $c_i = c$. This value corresponds to the cost to completely pick up the rows/columns in the segment s_i after any embedded members of E_1 have been picked up. Proceed (with c possibly increasing) until we finish the list U.

Get Embedded E_1 Blocks. If E_2 is also a column class, then we must count and remove the embedded columns of E_1. Each segment could have a unique number of embedded columns, where the larger the segment, the more columns of E_1 can be embedded and the smaller the resulting segment will be in the next S-group. As an aside, this is an argument used in the proof of the Containment Lemma 10 of [1] (precisely, Case 2b).

To generate the ordering of $\hat{U}$ and the values in $\hat{S}$ and we must carefully count the embedded columns.

We iterate through the columns of U, keeping track of which columns of E_1 are embedded and will get picked up. To help us with this task, we start with a sorted list V of the indexes of the columns of E_1, an empty list for $\hat{U}$, and a counter d, initialized to $d = 0$, to measure the number of required sticks. Also start with B, a set of columns of E_1, initialized as the empty set.

Similar to the way we counted contiguous blocks, for each column $u_i \in U$ we insert u_i into a sorted list of indexes I. Get the predecessor and successor of u_i in I, called u_j and u_k respectively, if they exist (in which case $j < i$ and $k < i$). Use range search to check if u_j and u_i are contiguous, and if u_i and u_k are contiguous; if a pair does not exist, then treat it as not being contiguous. If neither of these two pairs is contiguous, then do nothing. If exactly one of these pairs is contiguous, then use binary search in V to obtain the set of columns of E_1 embedded between the pair, add this set to B, and increment d. If both of these pairs are contiguous, then use binary search to determine if there are elements of $\tilde{E_1}$ (defined earlier as all the columns of class E_1) between u_j and u_i and between u_i and u_k; in which case we increment d. After we have iterated through $u(s_i)$ columns of U, it is time to add to $\hat{U}$ and create a new segment $\hat{s}_i \in \hat{S}$ with the following values:

$$u(\hat{s}_i) = |V| - |B|,$$
$$d(\hat{s}_i) = d(s_i) + d + c_i,$$
$$p(\hat{s}_i) = s_i.$$

Remove all of the members of B from V, then add all of these members to the beginning of $\hat{U}$. Reset B to be the empty set. Continue iterating through U.

Once we have finished iterating through U, add the remaining elements in V to the beginning of $\hat{U}$. Notice that now we have that $u(s_i)$ is a decreasing as a function of i; we therefore invert the $u(.)$ list and the associated $d(.)$ and $p(.)$ fields.

Build the Next S-group. If E_2 is a row class, then $\hat{U} = \tilde{E_1}$ and $\hat{S}$ contains one segment $\hat{s}$. To find $\hat{s}$, we iterate over each segment $s_i \in S$ looking for the segment s_i with minimum value $d(s_i) + c_i$. $\hat{s}$ has the following values:

$$u(\hat{s}) = |\tilde{E_1}|,$$

$$d(\hat{s}) = d(s_i) + c_i,$$

$$p(\hat{s}) = s_i.$$

If E_2 is a column class, $\hat{U}$ and $\hat{S}$ were created while we found embedded columns.

3.3 Merging Identical S-groups

Once we have generated a new S-group (E_i, E_j) we add it to a two-dimensional table, where the row is determined by E_i's index in the original list of equivalence classes, and the column similarly determined from E_j's index.

If an S-group (E_i, E_j) already exists, then we must merge the two S-groups. Given two (E_i, E_j) S-groups, $G_1 = \{U_1, S_1\}$ and $G_2 = \{U_2, S_2\}$, we will compute a new S-group, $G_3 = \{U_3, S_3\}$, that encompasses both of these S-groups which will then get stored in our table. We iterate through both G_1 and G_2 concurrently in order to create G_3, as described below. We start with U_3 and S_3 being empty.

Merge S_1 and S_2, maintaining the ordering based on u_1 and u_2. Iterate through each s_k in this merged list. Let S_q be the list which contains s_k ($q = 1$ or $q = 2$). Remove elements from the start of U_q, adding them to the end of U_3 until $|U_3| = u(s_k)$. (Do not add an element to U_3 if it is already in U_3.) This works because of the Containment Lemma 10. (As an aside, the proof in Applegate et al. [1] of the Containment Lemma, precisely subcase 2c, relies on proving the fact that all the values of $u(s)$ with $s \in S_1$ are at most the minimum of the values of $u(s)$ with $s \in S_2$, or vice versa. So this merge procedure could be simplified) Add s_k to S_3. If G_1 and G_2 both have segments such that $u(s_a) = u(s_b)$, then choose the segment of smaller distance (field $d(.)$) to add to S_3.

This merge can be done in $O(|E_2| \log n)$ as one can use binary search trees to check which elements are in U_3.

3.4 Finding an optimum RSRL

While we build the graph on S-groups, we keep track of the segment which is a valid endpoint of the smallest distance. We define a *valid endpoint* to be a segment which is completely gray (this is since we solve the modified version of the problem; for the original problem, we would have a segment which is completely white and gray). A segment (E_1, E_2) is an endpoint if $E_1 = E_2$. (In the original version, we also have the case where E_1 and E_2 are adjacent in the equivalence class list and the cell where E_1 and E_2 intersect is white in the original pattern.) Then once the graph on S-groups is finished, we build a path to the valid endpoint using the values stored in p. This path will give an optimal order to remove equivalence classes, which can then be translated into an optimal list of rectangle strip-rules.

This algorithms returns an optimum solution, as follows from all the discussion above.

4 Time and Space Complexity

In this section we will show the time and space complexity of the previously defined algorithm is $O(nN \log n)$ and $O(nN)$ respectively – where N is defined earlier as the total number of equivalence classes and n is defined earlier as the total number of rows and columns in the original pattern. The ideas of our proof are partially taken from a more complete version of Applegate et al. [1], which showed the number of reachable segments to be in $O(n^2)$. Indeed, our contribution is a faster way of processing a segment, cutting down this processing time down from $O(n)$ to $O(\log n)$.

Following Applegate et al. [1], we will show that if the complexity for an S-group (E_a, E_b) is in $O(|E_a| + |E_b|)$, then the overall complexity of all S-groups is $O(nN)$. For each S-group, add the first term of its complexity, $O(|E_a|)$, to one two-dimensional array, and its second, $O(|E_b|)$, to a second array – each at location (E_a, E_b).

$$\begin{bmatrix} E_1 & E_1 & \dots & E_1 \\ E_2 & E_2 & \dots & E_2 \\ \vdots & \vdots & \ddots & \vdots \\ E_N & E_N & \dots & E_N \end{bmatrix} \begin{bmatrix} E_1 & E_2 & \dots & E_N \\ E_1 & E_2 & \dots & E_N \\ \vdots & \vdots & \ddots & \vdots \\ E_1 & E_2 & \dots & E_N \end{bmatrix}$$

By noting that the sum of all members of all equivalence classes equals the total number of rows and columns, we get $\sum_{i=0}^{N} |E_i| = n$. The sum of the columns in the first matrix and the sum of the rows in the second matrix both equal n. The sum of all terms in both matrices is $2nN$, so the complexity of all S-groups is in $O(nN)$.

The space complexity of the algorithm is determined by the sizes of the lists storing the S-groups. Each S-group maintains a list U, which holds members of E_b, and is therefore in $O(|E_b|)$. The list S contains all segments. Each segment is guaranteed to have a unique size u, and the values of u are positive values at most $|E_b|$. The complexity of each segment is constant, so we again have $O(|E_b|)$. As we have previously shown, since each segment is in $O(|E_a| + |E_b|)$, the overall space complexity is $O(nN)$.

Each operation we perform on an S-group (E_a, E_b) happens in time either $O(|E_a| \log n)$ or $O(|E_b| \log n)$, as indeed every "check" from the algorithm's description takes time $O(\log n)$, after $O(n \log n)$ initialization of the range search data structure or the ordered list represented by a balanced binary tree. So the runtime for processing the equivalence class list is $O(nN \log n)$.

Since reading the input takes $O(n^2)$ time and N is in $O(n)$ (this follows immediately from Theorem 2), we relax our bounds and say our space complexity is $O(n^2)$ and the overall runtime is $O(n^2 \log n)$.

5 Conclusions

As noted in Norige et al. [21, 22], these solutions do not generalize to dimensions higher than two. This is the most interesting open question, in our opinion.

We believe that range trees can be replaced by ad-hoc methods to obtain a $O(n^2)$ algorithm for exact RSRL strip-rule minimization. The savings of $O(\log n)$ in running time comes at the expense of a more complicated algorithm which we decided not to present.

Acknowledgments.

Ian's research was done while at Illinois Institute of Technology, and supported by the Brazil Scientific Mobility Program. Gruia's and Nathan's work was done while at Illinois Institute of Technology, and supported by the National Science Foundation under awards NSF-1461260 (REU).

References

[1] Applegate, D., Călinescu, G., Johnson, D.S., Karloff, H.J., Ligett, K., Wang, J.: Compressing rectilinear pictures and minimizing access control lists. In: Bansal, N., Pruhs, K., Stein, C. (eds.) Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2007, New Orleans, Louisiana, USA, January 7–9, 2007. pp. 1066–1075. SIAM (2007)

[2] de Berg, M., van Kreveld, M., Overmars, M., Schwarzkopf, O.: Computational Geometry (second edition). Springer-Verlag (2000)

[3] Cheng, Y., Wang, W., Wang, J., Wang, H.: FPC: A new approach to firewall policies compression. Tsinghua Science and Technology **24**, 65–76 (02 2019). https://doi.org/10.26599/TST.2018.9010003

[4] Comerford, P., Davies, J.N., Grout, V.: Reducing packet delay through filter merging. In: Proceedings of the 9th International Conference on Utility and Cloud Computing. pp. 358–363. UCC '16, ACM, New York, NY, USA (2016). https://doi.org/10.1145/2996890.3007854, http://doi.acm.org/10.1145/2996890.3007854

[5] Daly, J., Liu, A.X., Torng, E.: A difference resolution approach to compressing access control lists. IEEE/ACM Trans. Netw. **24**(1), 610–623 (Feb 2016). https://doi.org/10.1109/TNET.2015.2397393, http://dx.doi.org/10.1109/TNET.2015.2397393

[6] Demianiuk, V., Kogan, K., Nikolenko, S.: Approximate classifiers with controlled accuracy. In: Proceedings of IEEE INFOCOM 2019 - IEEE Conference on Computer Communications (2019). https://doi.org/10.1109/INFOCOM.2019.8737476

[7] Giroire, F., Havet, F., Moulierac, J.: On the complexity of compressing two dimensional routing tables with order. Algorithmica **80**(1), 209–233 (Jan 2018). https://doi.org/10.1007/s00453-016-0243-7, https://doi.org/10.1007/s00453-016-0243-7

[8] Gouda, M.G., Liu, A.X.: Structured firewall design. Comput. Netw. **51**(4), 1106–1120 (Mar 2007). https://doi.org/10.1016/j.comnet.2006.06.015, http://dx.doi.org/10.1016/j.comnet.2006.06.015

[9] Kang, N., Liu, Z., Rexford, J., Walker, D.: Optimizing the "One Big Switch" abstraction in software-defined networks. In: Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies. pp. 13–24. CoNEXT '13, ACM, New York, NY, USA (2013). https://doi.org/10.1145/2535372.2535373, http://doi.acm.org/10.1145/2535372.2535373

[10] Kang, N., Reich, J., Rexford, J., Walker, D.: Policy transformation in software defined networks. In: Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication. pp. 309–310. SIGCOMM '12, ACM, New York, NY, USA (2012). https://doi.org/10.1145/2342356.2342424, http://doi.acm.org/10.1145/2342356.2342424

[11] Kogan, K., Nikolenko, S., Culhane, W., Eugster, P., Ruan, E.: Towards efficient implementation of packet classifiers in SDN/OpenFlow. In: Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking. pp. 153–154. HotSDN '13, ACM, New York, NY, USA (2013). https://doi.org/10.1145/2491185.2491219, http://doi.acm.org/10.1145/2491185.2491219

[12] Lam, H.Y., Wang, D.J., Chao, H.J.: A traffic-aware top-N firewall ruleset approximation algorithm. In: Proceedings of the 6th ACM/IEEE Symposium on Architectures for Networking and Communications Systems. pp. 9:1–9:2. ANCS '10, ACM, New York, NY, USA (2010). https://doi.org/10.1145/1872007.1872019, http://doi.acm.org/10.1145/1872007.1872019

[13] Li, W., Qin, Z., Li, K., Yin, H., Ou, L.: A novel approach to rule placement in software-defined networks based on OPTree. IEEE Access **PP**, 1–1 (01 2019). https://doi.org/10.1109/ACCESS.2018.2889194

[14] Liu, A.X., Meiners, C.R., Zhou, Y.: All-match based complete redundancy removal for packet classifiers in TCAMs. In: IEEE INFOCOM 2008 - The 27th Conference on Computer Communications (2008). https://doi.org/10.1109/INFOCOM.2008.31

[15] Liu, A.X., Gouda, M.G.: Complete redundancy detection in firewalls. In: Proceedings of the 19th Annual IFIP WG 11.3 Working Conference on Data and Applications Security. pp. 193–206. DBSec'05, Springer-Verlag, Berlin, Heidelberg (2005). https://doi.org/10.1007/11535706_15, http://dx.doi.org/10.1007/11535706_15

[16] Liu, A.X., Gouda, M.G.: Complete redundancy removal for packet classifiers in TCAMs. IEEE Trans. Parallel Distrib. Syst. **21**(4), 424–437 (Apr 2010). https://doi.org/10.1109/TPDS.2008.216, http://dx.doi.org/10.1109/TPDS.2008.216

[17] Liu, A.X., Meiners, C.R., Torng, E.: TCAM razor: A systematic approach towards minimizing packet classifiers in TCAMs. IEEE/ACM Trans. Netw. **18**(2), 490–500 (Apr 2010). https://doi.org/10.1109/TNET.2009.2030188, http://dx.doi.org/10.1109/TNET.2009.2030188

[18] Liu, A.X., Meiners, C.R., Torng, E.: Packet classification using binary content addressable memory. IEEE/ACM Trans. Netw. **24**(3), 1295–1307 (Jun 2016). https://doi.org/10.1109/TNET.2016.2533613, https://doi.org/10.1109/TNET.2016.2533613

[19] Maity, I., Mondal, A., Misra, S., Mandal, C.: Tensor-based rule-space management system in SDN. IEEE Systems Journal **PP**, 1–8 (11 2018). https://doi.org/10.1109/JSYST.2018.2879321

[20] Meiners, C.R., Liu, A.X., Torng, E.: Bit weaving: A non-prefix approach to compressing packet classifiers in TCAMs. IEEE/ACM Trans. Netw. **20**(2), 488–500 (Apr 2012). https://doi.org/10.1109/TNET.2011.2165323, http://dx.doi.org/10.1109/TNET.2011.2165323

[21] Norige, E., Liu, A.X., Torng, E.: A ternary unification framework for optimizing TCAM-based packet classification systems. In: Proceedings of the Ninth ACM/IEEE Symposium on Architectures for Networking and Communications Systems. pp. 95–104. ANCS '13, IEEE Press, Piscataway, NJ, USA (2013)

[22] Norige, E., Liu, A.X., Torng, E.: A ternary unification framework for optimizing TCAM-based packet classification systems. IEEE/ACM Trans. Netw. **26**(2), 657–670 (Apr 2018). https://doi.org/10.1109/TNET.2018.2809583, https://doi.org/10.1109/TNET.2018.2809583

[23] Pao, D., Lu, Z.: A multi-pipeline architecture for high-speed packet classification. Comput. Commun. **54**(C), 84–96 (Dec 2014). https://doi.org/10.1016/j.comcom.2014.08.004, http://dx.doi.org/10.1016/j.comcom.2014.08.004

[24] Rifai, M., Huin, N., Caillouet, C., Giroire, F., Moulierac, J., Pacheco, D.L., Urvoy-Keller, G.: Minnie: An SDN world with few compressed forwarding rules. Computer Networks **121**, 185 – 207 (2017). https://doi.org/https://doi.org/10.1016/j.comnet.2017.04.026, http://www.sciencedirect.com/science/article/pii/S1389128617301433

[25] Rottenstreich, O., Keslassy, I., Hassidim, A., Kaplan, H., Porat, E.: Optimal in/out TCAM encodings of ranges. IEEE/ACM Trans. Netw. **24**(1), 555–568 (Feb 2016). https://doi.org/10.1109/TNET.2014.2382031, http://dx.doi.org/10.1109/TNET.2014.2382031

[26] Sanger, R., Luckie, M., Nelson, R.: Identifying equivalent SDN forwarding behaviour. In: Proceedings of the 2019 ACM Symposium on SDN Research. pp. 127–139. SOSR '19, ACM, New York, NY, USA (2019). https://doi.org/10.1145/3314148.3314347, http://doi.acm.org/10.1145/3314148.3314347

[27] Sun, Y., Kim, M.S.: Bidirectional range extension for TCAM-based packet classification. In: Proceedings of the 9th IFIP TC 6 International Conference on Networking. pp. 351–361. NETWORKING'10, Springer-Verlag, Berlin, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12963-6_28, http://dx.doi.org/10.1007/978-3-642-12963-6_28

[28] Sun, Y., Kim, M.S.: Tree-based minimization of TCAM entries for packet classification. In: Proceedings of the 7th IEEE Conference on Consumer Communications and Networking Conference. pp. 827–831. CCNC'10, IEEE Press, Piscataway, NJ, USA (2010)

[29] Suri, S., Sandholm, T., Warkhede, P.R.: Compressing two-dimensional routing tables. Algorithmica **35**(4), 287–300 (2003). https://doi.org/10.1007/s00453-002-1000-7, https://doi.org/10.1007/s00453-002-1000-7

[30] Zhang, X., Yu, S., Zhang, J., Xu, Z.: Forwarding rule multiplexing for scalable SDN-based internet of things. IEEE Internet of Things Journal **6**(2), 3373–3385 (2019). https://doi.org/10.1109/JIOT.2018.2882855, https://doi.org/10.1109/JIOT.2018.2882855

[31] Zhang, Y., Natarajan, S., Huang, X., Beheshti, N., Manghirmalani, R.: A compressive method for maintaining forwarding states in SDN controller. In: Proceedings of the Third Workshop on Hot Topics in Software Defined Networking. pp. 139–144. HotSDN '14, ACM, New York, NY, USA (2014). https://doi.org/10.1145/2620728.2620759, http://doi.acm.org/10.1145/2620728.2620759