Morg

An exploration of 2D Search & Replace

Lulof Pirée (s3406512) Ananta Shahane (s2900718)

Leiden University June 10, 2022

Abstract

Search & Replace with regular expressions is a common feature in many text-editors. However, strings are 1D arrays of symbols, and regular expressions can be generalized to 2D and beyond; this may have applications in game design and Procedural Content Generation. This work is a first exploration into defining a 2D regular expression language, and proposes a first draft of a syntax, semantics and algorithms for implementing such a language. In many aspects, the semantic complexity of Search & Replace greatly increases when generalized to multiple dimensions.

Contents

1 Introduction

Many common text-editors (such as Vim, Emacs, Visual Studio Code, etc.) and programming languages (Python, JavaScript, etc.) offer a feature to search and replace patterns of characters in a text file or a string data-structure. The reader might be well acquainted with an example of an application offering such features. Some applications allow the search-patterns to be specified as regular expressions ("regex" for short), a compact notation for a language for strings. This allows more powerful expressions than providing a specific substring to find and replace.

Now, text files and strings are 1-dimensional arrays of symbols (note that line-breaks are just a special character such as `\n`). These symbols are usually from a character-alphabet such as ASCII or Unicode.

Many data-structures used computer programs, and in particular in games, can also be interpreted as n-dimensional arrays of symbols. Typical images are 2D arrays of pixel colours[1], simple 2D platformer games often employ tile-based obstacles worlds, RTS games may use a tiled world-map, and 3D voxel based games such as Minecraft are basically a 3D array of discrete symbols.

This project, with codename "Morg", is a first curious exploration in the possibilities of extending Search & Replace to multiple-dimensional arrays (in particular on 2D arrays). The Morg project explores how 2D search-pattern can be defined, how they could be semantically interpreted, identifies challenges and investigates algorithmic solutions for implementing such a Search & Replace tool in 2D.

1.1 Goals

There are many subtle complexities involved when generalizing regular expressions to 2D. For this reason, the Morg project mostly focuses on the very basic features of Search & Replace, which can be described as the following routines:

- Match: this routine finds the first substring in a given source-text that is described by a given regular expression.
- MatchAll: this routine finds *all* the substrings that match a regular expression.
- Replace: this subroutine uses Match to find a substring to replace a given replace-pattern.
- ReplaceAll: this routine finds *all* matching substrings and attempts to replace as many as possible.

The generalizations of these routines to 2D is the main interest of the Morg project.

[1] The set of possible colours per pixel is usually discrete. For example, 8-bit colours with 3 colour channels and 1 opacity channel use 32 bit per pixel. This means that colours are drawn from a finite alphabet of 2^{32} discrete colours.

1.2 Challenges and contributions

At the beginning of the project, it was yet unclear how difficult it is to design a regular-expression language for 2D symbol-arrays. This turned out to be rather involved, so it is beyond the scope of the project to present a complete framework. This might not even be desirable, as certain conventions need to be chosen — similar to the process of designing a programming language — which calls for an iterative process.

That being said, the project provides an exploratory look into the possibilities. In particular, this report offers:

- A syntax for the essential features of 2D regular expressions. These features allow already quite sophisticated patterns, expressing repetition and choices.
- Algorithms for most of these features. A subset of these algorithms has been proven to be correct, or was implemented in Python to demonstrate the feasibility empirically.
- A sketch of the software architecture for a a domain-independent 2D regex-engine.

1.3 Outline

This first section introduced the topic and the research goals. Section 2 briefly introduces notational conventions and basic definitions used in this report. This is followed by Section 3, which starts with the basics of Search & Replace regular-expressions in 1D. Section 4 builds further on these concepts and gives a high-level overview of how 2D regular expressions work. The 2D search-patterns are further explained in 5, which introduces the algorithm MatchAllGroup, and continues to discuss optimisations. Section 6 explains how the algorithm ReplaceAll could be implemented, and how this basic algorithm can be further optimised. A discussion of design decisions, limitations and future work can be found in Section 8. The discussion is followed by a conclusion in Section 9. Finally, the appendix (Section A) will argue that repeated application of Search & Replace is a Turing Universal computational mechanism.

2 Notations and Definitions

This section provides a brief reference for notations and definitions used throughout this report. Feel free to refer to it when needed.

This report makes heavy use of the notions of symbols and alphabets. Symbols are mutually distinct atomic entities, and alphabets are sets of symbols. For example, if it happens that we include the character `a` in the alphabet Σ, then $\texttt{a} \in \Sigma$. This special font for symbols is used consistently throughout this report.

Let Σ^* be the set of all finite strings that can be created by concatenating symbols from Σ. Then a string is a vector $\vec{\lambda} \in \Sigma^*$, that is, an ordered array of symbols from Σ. 2D arrays of symbols are simply matrices with entries from an alphabet Σ.

2.1 Notation Conventions

The following notations are used:

- $\vec{\lambda}[i]$ to denote the $i + 1^{th}$ character of $\vec{\lambda}$. So for example, $\vec{\lambda}[0] \in \Sigma$ would be its the first character.
- $\vec{\lambda}[i : j]$ to denote the substring of $\vec{\lambda}$ starting from its $i + 1^{th}$ character up to its and including its j^{th} character. (Note that $\vec{\lambda}[j]$ is not included, since we start counting from 0. $\vec{\lambda}[j - 1]$ is the j^{th} character of $\vec{\lambda}$.)

- $\left|\vec{\lambda}\right|$ denotes the amount of characters in $\vec{\lambda}$.
- $\vec{\lambda} :: \vec{\tau} \in \Sigma^*$ to denote the concatenation of two strings $\vec{\lambda}, \vec{\tau} \in \Sigma^*$. For a concrete example, we would have `ap` :: `ple` = `apple`.
- $\vec{\lambda}^n$ denotes the concatenation of n times $\vec{\lambda}$. For example, $\vec{\lambda}^3 \equiv \vec{\lambda} :: \vec{\lambda} :: \vec{\lambda}$.
- ε to denote the empty string. I.e., $\varepsilon \in \Sigma^*$ is the string of 0 characters.
- Σ^n with $n \in \mathbb{N}$ denotes the set of strings obtained by concatenating n symbols from Σ (repetition is allowed). Similarly, if L is a set of strings, then L^n denotes the set of strings formed by concatenating n strings from L.
- Σ^* (the Kleene star) denotes the set of strings that can be obtained by concatenating any number of characters from Σ. It includes concatenating 0 characters, so $\varepsilon \in \Sigma^*$ holds for all alphabets Σ. Also for a set of strings L, L^* is defined analogously, but then concatenations of entire strings. Again, $\varepsilon \in L^*$.
- $\mathbb{B} := \{\text{TRUE}, \text{FALSE}\}$, the set of Boolean values.

2.2 Basic Definitions

A few definitions are used throughout this report to refer to specific structures. These are the following:

- *Source-text*: the array to which a search (and replace) command is applied. For strings this is a vector in an alphabet Σ, for 2D applications it is a matrix instead.
- *Search-pattern*: a regular expression that is the description of the mini-language of substrings that should be found in a given source-text. It itself is a vector (1D case) or matrix (2D case) of symbols of a special alphabet denoted as Ξ.
- *Pattern-expression*: a semantic building block of a search-pattern. It may contain nested pattern-expressions. Each well-formed search-pattern describes at least one pattern-expression.
- *Replace-pattern*: the vector/matrix to insert in the place of substrings in a source-text that a search-pattern identified.
- *Rectangle*: an small data-structure $((r_1, c_1), (r_2, c_2)) \in \mathbb{N}^2 \times \mathbb{N}^2$, describing the left-upper $((r_1, c_1))$ and lower-right $((r_2, c_2))$ corner of a rectangle in a grid. Note that this is a sufficient description to identify a rectangular subset of grid coordinates. Both corner-coordinates are inclusive. If R is a given rectangle, then the attributes $R.r_1$, $R.c_1$, $R.r_2$ and $R.c_2$ give the values of the respective corner-coordinates.
- *Match*: a rectangle whose contained grid-coordinates describe the indices of a source-text that is a substring/submatrix described by a search-pattern.

3 Search & Replace in 1D

Before discussing 2D regular expression, we first explain Search & Replace for 1D strings. This is not only to make the reader acquainted with the used notations and terminology of this report, but more importantly because 2D generalizations follow quite naturally from this discussion.

3.1 Warming up: 1D Regular Expression

Search & Replace features in text editors, command-line tools and programming languages usually take a source-text, a search-pattern and a replace-pattern as input.

Let us begin with a simple example. In Vim, the command to replace all occurrences of `dragon` by `unicorn` would be:

```
:%s/dragon|chicken/unicorn
```
(1)

The search pattern is `dragon|chicken`, and the replacement pattern is `unicorn` (`:%s` encodes the command that we call the ReplaceAll). When applied to the source-text

```
Do you think dragons eat chickens?
```
, (2)

the search-pattern finds the matches `dragon` starting at the 14^{th} character, and `chicken` at the 26^{th} character. Both substrings are then removed and replaced with `unicorn`, yielding the output:

```
Do you think unicorns eat unicorns?
```
. (3)

Note that the length of the source-text has changed. (2) contains 34 characters, while (3) contains 35 characters. This is because the match `dragon` contains one character less than the replace-pattern `unicorn`.

3.1.1 A Formal Model

In order to discuss algorithms for Search & Replace, the above ideas need to be formalized further. So, let $\vec{t}$ denote the source-text in which to search and replace. If $\ell \in \mathbb{N}$ is the length of this text (*before* the replacement happens, which might change the length of the text), then we may write $\vec{t} \in \Sigma^\ell$ for some alphabet Σ. The replacement pattern is also a string $\vec{r} \in \Sigma^m$ of a fixed length $m \in \mathbb{N}$.

The search-pattern, however, is quite different: it uses characters not in Σ. In the above example (1), the pattern `dragon|chicken` contains a character `|` to denote choice — it is not meant to match a `|` in the source-text. For this reason, we define another alphabet for search patterns, named $\mathcal{R}_\Sigma$. If $\vec{s}$ is a search-pattern of length n that is defined for a source-text over Σ, then we denote this as $\vec{s} \in \mathcal{R}_\Sigma^n$.

Mini-Language The alphabet $\mathcal{R}_\Sigma$ is used to construct *pattern-expressions*. Each pattern-expression defines a "mini-language" over Σ. For example, `[abc]?` $\in \mathcal{R}_\Sigma^5$ expresses the mini-language $\{\varepsilon, \texttt{a}, \texttt{b}, \texttt{c}\} \subset \Sigma$. Another example is the wildcard `.`, whose corresponding mini-language is Σ. Combining the `.` with the Kleene-star, `.*`, gives the language Σ^*.

The language $\mathcal{R}_\Sigma$ also includes a pattern-expression for each symbol in Σ. For example, if `a` $\in \Sigma$, then there is a corresponding character `a` $\in \mathcal{R}_\Sigma$ (usually written as the same character, which seems most readable), whose mini-language is $\{\texttt{a}\} \subset \Sigma$.

Formally, the pattern-expressions are a language $\mathcal{L}_\Sigma \subseteq \mathcal{R}_\Sigma^*$ of strings from $\mathcal{R}_\Sigma$. The expressions in $\mathcal{L}_\Sigma$ are connected to their mini-languages by means of a mapping $f_s \colon \mathcal{L}_\Sigma \to 2^{\Sigma^*}$.

Nested Pattern-Expressions Pattern-expressions are often nested. In the example above, `[abc]?` contains already 3 layers of nested: the topmost is a quantifier `?`, which contains a range `[...]`. The range, in turn, contains three atomic pattern-expressions: `a`, `b` and `c`.

Matching a Search-Pattern It is now clear that search-patterns are a concatenation of pattern-expressions, and each pattern-expression expresses a mini-language over Σ. Now it remains to define how this can be used to search in a source-text $\vec{t} \in \Sigma^*$.

This is relatively intuitive to describe. Let $\rho_1\rho_2\ldots\rho_k \in \mathcal{L}_\Sigma$ be the pattern-expressions occurring (in the given order) in the search-pattern $\vec{s} \in \mathcal{R}^*_\Sigma$. A substring $\vec{\tau}$ of $\vec{t}$ *matches* $\vec{s}$ if we can substitute each ρ_i with a string from its corresponding the mini-language such that the resulting string is $\vec{\tau}$. Formally, this can be expressed as:

Definition 1 (matching of a search-pattern)**.** *A string* $\vec{\tau} \in \Sigma^*$ matches *a search-pattern* $\vec{s} \in \mathcal{R}^*_\Sigma$*., written as* $\vec{\tau} \sqsubseteq \vec{s}$*, if and only if*

$$\exists_{\rho_1,\ldots,\rho_k \in \mathcal{L}_\Sigma} \left[\vec{s} = \rho_1 :: \rho_2 :: ... :: \rho_k \ \wedge \ \exists_{\sigma_1 \in f_s(\rho_1),\ldots,\sigma_k \in f_s(\rho_k)} [\vec{\tau} = \sigma_1 :: ... :: \sigma_k]\right]. \tag{4}$$

Note that $\rho_1, \ldots, \rho_k$ are necessarily the $k \in \mathbb{N}$ top-level pattern-expressions in $\vec{s}$, which are not nested in another pattern-expression. However, they may contain nested expressions themselves.

As for a concrete example, take $\vec{tau} =$ `app` $\in \Sigma^* = \{$`a`, `b`, `c`, `p`$\}$ and $\vec{s} =$ `[abc]ap*` $\in \mathcal{R}^*_\Sigma$ we can say that $\vec{\tau} \sqsubseteq \vec{s}$, or equivalently,

`aap` $\sqsubseteq$ `[abc]ap*`.

This holds because the top-most pattern-expressions of $\vec{s}$ are `[abc]`, `a` and `p*`, and since:

$$\begin{aligned}
&\texttt{a} \in f_s(\texttt{[abc]}) = \{\texttt{a}, \texttt{b}, \texttt{c}\} \\
&\texttt{a} \in f_s(\texttt{a}) = \{\texttt{a}\} \\
&\texttt{p} \in f_s(\texttt{p*}) = \{\varepsilon, \texttt{p}, \texttt{pp}, \texttt{ppp}, \ldots\}.
\end{aligned}$$

3.1.2 Examples of Pattern-Expressions

To generalize search-patterns to 2D, it is useful to first consider which pattern-expressions are commonly used in 1D. Below follow some examples, which will be familiar to users of regexes in Vim, or Python, or Visual Studio Code, etc.:

- `.`: this expression intuitively means "any single character". In other words, we would simply have $f_s(\texttt{.}) = \Sigma$.
- `|`: intuitively means "or". In the definitions of this document, it would be defined as $f_s(\rho_1\texttt{|}\rho_2) := f_s(\rho_1) \cup f_s(\rho_k)$ for any $\rho_1, \rho_2 \in \mathcal{L}_\Sigma$.
- `[`$\rho_1 \ldots \rho_n$`]` where $\rho_1, \ldots, \rho_k \in \mathcal{L}_\Sigma$ (for some $n \in \mathbb{N}$). This is a short-hand for `(`ρ_1`|`$\ldots$`|`ρ_n`)`.
- `(` `)` are used as precedence delimiters for nested patterns. They are particularly useful in combinations with quantifiers such as `*`, `?` and `+`.
- ρ`*` means 0 or more repetitions of any string described by $\rho \in \mathcal{L}_\Sigma$. So formally this would be $f_2(\rho\texttt{*}) := f_s(\rho)^*$, where the last $*$ is the Kleene-star.
- ρ`?` matches either (1) a string described by the pattern-expression $\rho \in \mathcal{L}_\Sigma$, or (2) the empty string ε. So, formally: $f_2(\rho\texttt{?}) := f_s(\rho) \cup \{\varepsilon\}$, where the last ε is the empty string.
- ρ`+` means 1 or more repetitions from a string described by $\rho \in \mathcal{L}_\Sigma$. Formally:

$$f_2(\rho\texttt{+}) := \{\sigma_1 :: \sigma_2 \mid \sigma_1 \in f_s(\rho), \sigma_2 \in f_s(\rho\texttt{*})\}.$$

- ρ`{`a`,`b`}` with $a, b \in \mathbb{N}, a \leq b$ indicated either $a, a+1, \ldots, b-1$ or b repetitions from a string described by $\rho \in \mathcal{L}_\Sigma$. Formally:

$$f_2(\rho\texttt{+}) := \bigcup_{i=a}^{b} f_s(\rho)^i$$

- $\rho_1\rho_2$ denotes any string $\vec{\lambda_{1,2}} = \vec{\lambda}_1 :: \vec{\lambda}_2$ that is a concatenation of any string $\vec{\lambda}_1$ described by ρ_1 and any string $\vec{\lambda}_2$ described by ρ_2:

$$f_s(\rho_1\rho_2) = \{\vec{\lambda}_1 :: \vec{\lambda}_2 \mid \vec{\lambda}_1 \in f_s(\rho_1), \vec{\lambda}_2 \in f_s(\rho_2)\}.$$

- Each $\sigma \in \Sigma$ to denote the singleton-set. See below for an elaboration.

As mentioned earlier, $\mathcal{R}_\Sigma$ contains a symbol ρ corresponding to each symbol in $\sigma \in \Sigma$. These are used for form atomic pattern-expressions ("literals"), i.e. $\rho \in \mathcal{L}_\Sigma$. These literals match to exactly one string, namely σ. Usually, these patterns ρ are also denoted with exactly the same symbol as the single-character string $\sigma \in \Sigma$ they match (i.e., $\rho = \sigma$). For example, we would have `a` $\in \mathcal{L}_\Sigma$ when `a` $\in \Sigma$. More formally, this means that:

$$\{\sigma \mid \sigma \in \Sigma\} \subseteq \mathcal{L}_\Sigma, \tag{5}$$

$$\forall_\sigma \left[\sigma \in \Sigma \ : \ f_s(\sigma) = \{\sigma\}\right]. \tag{6}$$

Many real-world regex implementations offer more features, such as "look-ahead", "look-behind", capturing groups, etc. Also these can likely be generalized to 2D, but this is beyond the scope of this report.

3.2 The replacement pattern $\vec{r}$

The replacement pattern $\vec{r}$ has much simpler semantics than the search-pattern. In the simplest case, it is a string in Σ^*. The substrings of $\vec{\tau}$ that match $\vec{s}$ are then directly substituted by this replacement substring (possibly changing the length of the text vector $\vec{\tau}$).

3.2.1 Replace patterns with holes

One simple way that can greatly improve the utility of replacement patterns (albeit not very practical in text editors), is to allow "holes". Each "hole" in the replacement pattern would correspond to a character in $\vec{\tau}$ that is *not* replaced.

A basic implementation is to introduce a "hole" symbol, □, that may also appear in the replacement pattern. In this case, $\vec{r} \in (\Sigma \cup \{\square\})^*$. To illustrate the idea, assume that a substring of $\vec{\tau}$ starting at index i (say $\vec{\tau}[i:k]$) matched the search-pattern, and that $\vec{r}[j] = \square$. Before we replace $\vec{\tau}[i:k]$ in $\vec{\tau}$ with $\vec{r}$, we fill the hole by assigning:

$$\vec{r}[j] \leftarrow \vec{\tau}[i+j]. \tag{7}$$

Note that this idea requires further work in case it is to be used with replacement patterns that are longer than the matched substrings.

Some further extensions to the holes are, for example, quantifying the amount of original characters that are re-used, or be annotated with an offset from the index of the first matched character.

3.3 Algorithm for String Search & Replace

The notations and definitions described in this section can be combined into an algorithm for executing Search & Replace on strings.

To find a substring $\vec{\tau}$ of a source-text $\vec{t} \in \Sigma^*$ that matches a search-pattern $\vec{s} \in \mathcal{R}_\Sigma$, we can apply Definition 1 on all possible substrings of $\vec{t}$. This idea is expressed in pseudocode as GreedySeach (Algorithm 1). Note that this algorithm simply returns the first matching substring it found, hence the name "greedy" (different variations are, of course, possible here).

The complete procedure of searching and replacing a single match is given in GreedySearchAndReplace (Algorithm 2). This description includes the resolution of holes. Note that, for simplicity, it is again a greedy version, and it replaces only a single match. Replacing all matches is of course possible, but adds extra complexity (this will be discussed in detail for the 2D case).

Algorithm 1:
GreedySearch $(\vec{t}, \vec{s})$

Input: $\vec{t} \in \Sigma^*$: the source-text; A string of characters from some alphabet Σ in which to find a match.
$\vec{s} \in \mathcal{R}_\Sigma$: a search-pattern.

Output: $(i, j) \in \mathbb{N}^2$, $0 \leq i < j \leq |\vec{t}|$, two indices such that $\vec{t}[i:j] \sqsubseteq \vec{s}$.
Here i is the *smallest* index such that a match exist, and j is the *largest* possible end-index for the given i.
Return NIL if no match exists.

```
ℓ ← |t⃗| ;
for i = 0 to ℓ − 1 do
    for j = ℓ downto i + 1 do
        if t⃗[i : j] ⊑ s⃗ then
            return (i, j) ;
return NIL;
```

Algorithm 2:
GreedySearchAndReplace $(\vec{t}, \vec{s}, \vec{r})$

Input: $\vec{t} \in \Sigma^*$: a string of characters from some alphabet Σ in which to find a match.
$\vec{s} \in \mathcal{L}^*_\Sigma$: a search-pattern.
$\vec{r} \in (\Sigma \cup \{\square\})^*$ a replacement-pattern.

Output: $\vec{\theta}$: $\vec{t}$ with the first (greedy) match with $\vec{s}$ replaced according to $\vec{r}$.

```
(i, j) ← GreedySearch(t⃗.s⃗) ;
if (i, j) = NIL then                          // No match was found; nothing to replace
    return t⃗ ;
ℓ ← |t⃗| ;
Let θ⃗ ← t⃗[0 : i] :: r⃗ :: t⃗[j :: ℓ];
/* Now it remains to resolve possible holes.                                    */
for k = i to i + |r⃗| do
    if θ⃗[k] = □ then
        k̂ ← min(k, ℓ − 1) ;                    // Avoid indexing beyond the end of t⃗
        θ⃗[k] ← t⃗[k̂] ;
return θ⃗ ;
```

4 Generalization to 2D Search & Replace

Section 3 provided a formal framework for reasoning about regular expressions in 1D strings. This section discusses how these definitions can be generalized to 2D. Recall that strings are 1-dimensional arrays of symbols, so this section focusses on 2-dimensional arrays of symbols. Two-dimensional source-texts arise naturally in many applications, such as pixel-map computer images, tile-maps for games, integer matrices, icons

in square GUI menus of smart-phones, etc. Two-dimensional regular expressions can be further generalized to N dimensions, basically by repeating the same steps taken to generalize from 1D to 2D. However, for the ease of interpretability, this report focusses only on the 2D case.

So, in the remainder of this section, the structure of the source-text to search and replace in will be a n_1-by-n_2 matrix $T \in \Sigma^{n_1 \times n_2}$. More generally, when n and m are unknown, we write $T \in \Sigma^{*\times *}$ to denote that T is a rectangular 2D matrix of symbols in Σ. Also replace-patterns and search-patterns become matrices. When allowing holes in replace-patterns, we can denote such a replace-pattern-matrix as $R \in (\Sigma \cup \{\square\})^{*\times *}$.

Search-patterns will be collections of pattern-expressions, with some indication of relative spatial alignment in 2D. We chose to implement search-patterns as matrices with symbols from a new alphabet, Ξ_Σ, which extends $\mathcal{R}_\Sigma$ with symbols to build 2D pattern-expressions. This 2D search-pattern language is further discussed in Section 5.

4.1 Intuition of 2D Regular Expressions

In 1D, a match is a substring of the source-text, but in 2D it is a 2-dimensional subset of the source-text. Since we defined 2D replace-patterns to be rectangular matrices, it fitting to require matches to be rectangular as well. It is technically possible to allow matches of other shapes, but these seem semantically confusing to human end-users. We will refer to non-rectangular matches as *ragged matches*. In the remainder of this report, a *match* is defined as a rectangle of indices in a 2D source-text (as in Section 2).

Replacement is straightforward when the replace-pattern has the same shape as the match: simply substitute the values indicated by the match with the replace-pattern, expect if the replace-pattern has a hole at the given index. This becomes more complicated when the replace-pattern and the match have different shapes, which is discussed in detail in Section 6.

4.1.1 Example

Before discussing the details of the 2D Search & Replace algorithms, it may be illustrative to show a concrete example. Consider the following source-text:

$$T = \begin{matrix} \texttt{akbbaaa} \\ \texttt{kpbcdbc} \\ \texttt{qweaakp} \end{matrix}$$

and the following search-pattern S:

$$S = \begin{matrix} \boxtimes & \frown & \frown & \frown & \frown & \frown & \frown & \frown & \boxtimes & \\ \texttt{(} & \texttt{[} & \texttt{a} & \texttt{,} & \texttt{b} & \texttt{]} & \rightarrow^* & \texttt{a} & \texttt{)} & \rightarrow^+ \\ \texttt{(} & \texttt{b} & \texttt{[} & \texttt{c} & \texttt{,} & \texttt{d} & \texttt{]} & \rightarrow^* & \texttt{)} & \\ \boxtimes & \smile & \smile & \smile & \smile & \smile & \smile & \smile & \boxtimes & \\ & \downarrow^? & & & & & & & & \end{matrix}$$

And replacement-pattern R:

$$\texttt{1111121112} \tag{8}$$

When matching T with S, we find the following match (taking the biggest possible match, highlighted in red and blue):

$$T = \begin{matrix} \texttt{akbba}\texttt{aa} \\ \texttt{kpbcd}\texttt{bc} \\ \texttt{qweaakp} \end{matrix}$$

Here the $\rightarrow^{+}$ evaluated to 2 repetitions of the outermost group (in (, $\frown$, $\smile$ and) parentheses) in the vertical direction, and the $\downarrow^{?}$ to 1 repetition in the downward direction. Furthermore, the $\rightarrow^{*}$ in both [a,b]$\rightarrow^{*}$ and [c,d]$\rightarrow^{*}$ evaluated to 2 repetitions in the first repetition of the outermost group (in red), and to 2 repetitions in the second repetition of the outermost group (in blue).

After replacing the biggest possible match with the replacement pattern R, we obtain the output:

$$\hat{T} = \begin{matrix} \texttt{ak11111} \\ \texttt{kp21112} \\ \texttt{qweaakp} \end{matrix}$$

5 2D Search-Patterns

This section discusses the syntax and semantics of 2D search-patterns, as well as algorithms for finding matches in 2D.

5.1 Search-pattern Semantics

One-dimensional search and replace-patterns consists of pattern-expressions concatenated along one axis (i.e., placed next to each other). For the two dimensional case, the symbols can be aligned with respect two each other according to two axes. The most practical way to implement this seems to be organizing pattern-expressions in a matrix. Let Λ_Σ denote the language of 2d pattern-expressions.

Like in the 1D case, the pattern-expressions in Λ_Σ will be encoded with symbols a special alphabet, which we call Ξ_Σ. However, the symbols of Ξ_Σ are not used to form strings, but pattern-expressions instead are 2D matrices of symbols from Ξ_Σ. Hence a 2D search-pattern is a matrix $S \in \Xi_\Sigma^{*\times *}$, which encodes a matrix of pattern-expressions (each pattern-expression is encoded as a sub-matrix). Note that there are again two levels of abstraction: a low-level matrix of symbols from Ξ_Σ, which is used to describe a high-level matrix with pattern-expressions from Λ_Σ.

It is possible that the sub-matrices describing pattern-expressions do not form a rectangular matrix when placed together. This is easily resolved by padding unused space with empty entries, denoted with $\boxtimes$. This symbol is not to be confused with $\square$, the "hole" in one-dimensional replace-patterns.

Each 2D pattern-expression in Λ_Σ again expresses a mini-language whose entries are 2D matrices in $\Sigma^{*\times *}$. Matching will again take place by substituting each pattern-expression by an entry from the corresponding language. However, there are many subtleties to this, which are further discussed in Section 5.3.

We introduce some further technical constraints to the matrix (or grid) S of any 2D search-pattern:

1. The top-left entry is non-empty (i.e., $S[0,0] \neq \boxtimes$).
2. There are no rows or columns of only empty entries.
3. There is exactly one connected component of non-empty entries (entries are considered neighbours when they have a Manhattan-distance of 1 in the matrix S).
4. Let $K \in \Lambda_\Sigma^{*\times *}$ be the high-level matrix of pattern-expressions encoded by S. Then K has no missing entries at any position $K[i,j]$ if there exists a position $K[i+n,j]$ or $K[i,j+n]$ for any $n \in \mathbb{N}$ that is not empty. Note that is no special entry in Λ_Σ for empty pattern-expressions. These are simply encoded by their absence in S (e.g. by padding with $\boxtimes$s, or when the neighbouring pattern-expressions' encodings fill the matrix already).

The first requirement gives us an anchor-point we need to run a matching algorithm. The second requirement simply avoids overly large matrix representations. The last requirement is needed to avoid ambiguities in

matching. Note that empty cells ⊠ do not specify anything about the pattern, they do not match "any symbol" like `.` does.

5.2 Search-Pattern Syntax

The main idea behind extending the syntax of search-patterns from 1D to 2D is to quantifiers to express repetition along either of the two axes. This not only means that Ξ_Σ will contain horizontal and vertical counterparts for `*`, `+` and `?`, but also that delimiters are needed in both directions.

5.2.1 The Alphabet Ξ_Σ

We define Ξ_Σ to contain the the following symbols:

- $\rightarrow^*$, $\rightarrow^+$, $\rightarrow^?$ and $\rightarrow$. These correspond to the familiar `*`, `+` and `?`. The $\rightarrow$ is used with numerical quantifiers `{ }`.
- $\downarrow^*$, $\downarrow^+$, $\downarrow^?$ and $\downarrow$. These are the vertical counterparts of the symbols from the above item.
- ⊠, the placeholder symbol for empty entries.
- `(`, `)`, $\frown$, $\smile$ as delimiters for each axis.
- `{`, `}`, `,`, and the numbers `0-9` to describe specific quantifiers (in the same way as for one-dimensional regular expressions). In addition, there are also vertical quantifier brackets $\begin{smallmatrix}\frown\\|\end{smallmatrix}$ and $\begin{smallmatrix}|\\\smile\end{smallmatrix}$. Vertical quantifiers use the same symbols `,` and `0-9` within their brackets; this should cause no ambiguity.
- `[`, `]`, `-` to describe multiple (but restricted) options for a character as usual. The vertical counterparts for the delimiters are $\begin{smallmatrix}\sqcap\\|\end{smallmatrix}$ and $\begin{smallmatrix}|\\\sqcup\end{smallmatrix}$.
- `|` and `_` to denote choice (set union) in the horizontal and vertical case, respectively.
- `.`, identical to the `.` in string regular expressions.

The Σ subscript of Ξ_Σ is used to imply that Ξ can also describe the literal characters from Σ. So Ξ_Σ also includes a of each symbol in Σ. If a symbol $\sigma \in \Sigma$ happens to conflict with one of the symbols in Ξ already defined above, we prefix it with a `\` (which counts as an additional symbol in Ξ): `\`σ. This is also commonly used in 1D regular expressions. Also this `\` has a vertical counterpart: $\begin{smallmatrix}/\\\sigma\end{smallmatrix}$.

5.2.2 Pattern-Expression Types

A valid search-pattern $S \in \Xi_\Sigma$ consists of literal pattern-expressions in Λ_Σ and empty-symbols ⊠. The following paragraphs describe the different pattern-expressions that make up Λ_Σ. This include atomic pattern-expressions describing individual characters, and compound pattern-expressions for repetition and choice that contain nested pattern-expressions.

Literal Pattern-Expressions Literal pattern-expressions are the simplest expressions. They map to a mini-language of strings of length 1. A literal expression L maps, in a context-free grammar-like choice, to:

$$L \rightarrow \texttt{.} \mid \texttt{\}\sigma \mid \begin{smallmatrix}/\\\sigma\end{smallmatrix} \mid \lambda \tag{9}$$

Where σ is any symbol satisfying $\sigma \in \Sigma \cap \Xi_\Sigma$, and similarly, λ is any symbol satisfying $\lambda \in \Sigma \setminus \Xi_\Sigma$.

It is useful to further distinguish between vertical and horizontal literals. Actually, this 2-directional duality will be used for almost all pattern-expressions, where the subscript h is used for the horizontal axis, and v for the vertical axis. For literals this gives:

$$L_h \to \texttt{.} \mid \texttt{\textbackslash}\sigma \mid \lambda \tag{10}$$

$$L_v \to \texttt{.} \mid \begin{matrix}\texttt{/}\\ \sigma\end{matrix} \mid \lambda \tag{11}$$

Quantifier pattern-expressions Quantifier expressions can also be defined in a context-free manner. There are two kinds of quantifiers, horizontal (Q_h) and vertical (Q_v), defined symmetrically as:

$$Q_h \to \texttt{→}^* \mid \texttt{→}^+ \mid \texttt{→}^? \mid \texttt{→\{}R_h\texttt{\}} \mid \boxtimes \tag{12}$$

$$R_h \to \hat{N}_h \mid \hat{N}_h\texttt{,}\hat{N}_h \tag{13}$$

$$\hat{N}_h \to \texttt{0}N_h \mid \texttt{1}N_h \mid \ldots \mid \texttt{9}N_h \tag{14}$$

$$N_h \to \varepsilon \mid \texttt{0}N_h \mid \texttt{1}N_h \mid \ldots \mid \texttt{9}N_h \tag{15}$$

And:

$$Q_v \to \begin{matrix}\texttt{↓}\\ \texttt{⤙}\\ R_v\\ \texttt{⤚}\end{matrix}^* \mid \begin{matrix}\texttt{↓}\\ \texttt{⤙}\\ R_v\\ \texttt{⤚}\end{matrix}^+ \mid \begin{matrix}\texttt{↓}\\ \texttt{⤙}\\ R_v\\ \texttt{⤚}\end{matrix}^? \mid \begin{matrix}\texttt{↓}\\ \texttt{⤙}\\ R_v\\ \texttt{⤚}\end{matrix} \mid \boxtimes \tag{16}$$

$$R_v \to \hat{N}_v \mid \begin{matrix}\hat{N}_v\\ \texttt{,}\\ \hat{N}_v\end{matrix} \tag{17}$$

$$\hat{N}_v \to \begin{matrix}\texttt{0}\\ N_v\end{matrix} \mid \begin{matrix}\texttt{1}\\ N_v\end{matrix} \mid \ldots \mid \begin{matrix}\texttt{9}\\ N_v\end{matrix} \tag{18}$$

$$N_v \to \varepsilon \mid \begin{matrix}\texttt{0}\\ N_v\end{matrix} \mid \begin{matrix}\texttt{1}\\ N_v\end{matrix} \mid \ldots \mid \begin{matrix}\texttt{9}\\ N_v\end{matrix} \tag{19}$$

$$\tag{20}$$

Compound Pattern-Expressions: Choices Compound pattern-expressions do contain other expression embedded within them, as a sort of container. However, as seen above, the sub-expressions can have different sizes and even be defined in different directions. This makes it non-trivial to define compound expressions rigorously without introducing confusingly many new definitions and notations [2]. The definitions below try to reach a happy balance by being sufficiently unambiguous, while not overly restricting or overcomplicating the language. That being said, there are probably still many improvements possible.

Some choice compound pattern-expressions, in particular the choice-patterns, can also be expressed in a

[2] E.g., a formal method that appears to work, would an extension to context-free grammars. This extension would add numerical variables as arguments to the grammar's variables, used to denote the size of the bounding box. The substitution rules could then generate an amount of literals and grammar-variables dynamically depending on the numerical variables. This could ensure that the expressions remain enclosed in a matrix, and that empty cells are filled with ⊠. However, this description would easily take several pages, without providing the reader much intuition about the underlying ideas.

context-free manner. So we define horizontal and vertical choice-patterns C_h and C_v recursively as follows:

$$C_h \to \texttt{[}\rho_1\rho_2\ldots\rho_k\texttt{]}Q_h \mid \texttt{[}D_h\texttt{]}Q_h \mid \rho_1\texttt{|}\rho_2\texttt{|}\ldots\texttt{|}\rho_k \tag{21}$$

$$D_h \to L_h\texttt{-}L_h \mid L_h\texttt{-}L_hD_h \tag{22}$$

$$C_v \to \begin{matrix}\texttt{⌐}\\ \lambda_1\\ \lambda_2\\ \ldots\\ \lambda_k\\ \texttt{⌙}\\ Q_h\end{matrix} \quad\Big|\quad \begin{matrix}\texttt{⌐}\\ D_v\\ \texttt{⌙}\\ Q_h\end{matrix} \quad\Big|\quad \begin{matrix}\lambda_1\\ \texttt{\_}\\ \lambda_2\\ \texttt{\_}\\ \ldots\\ \texttt{\_}\\ \lambda_k\end{matrix} \tag{23}$$

$$D_v \to \begin{matrix}L_v\\ \texttt{-}\\ L_v\end{matrix} \quad\Big|\quad \begin{matrix}L_v\\ \texttt{\_}\\ L_v\\ D_v\end{matrix} \tag{24}$$

Here, $\rho_1, \rho_2, \ldots, \rho_k$ are $k \in \mathbb{N}, k \geq 1$ horizontal (but nor rectangular, i.e., vertical size 1) or single-character pattern-expressions. Similarly, $\lambda_1, \lambda_2, \ldots, \lambda_k$ are k vertical pattern-expressions with horizontal size 1. The `-` indicates a range, and the literals surrounding it must be part of some well-known ordering, with the left/top literal preceding the right/bottom literal. Examples for ASCII strings are `[0-9]` to indicate a choice of all digits from 0 to 9, and `[a-zA-Z]` indicates a choice of any digit of the Roman alphabet, either capitalized or not.

Compound Pattern-Expressions: Groups Groups are probably one of the most powerful constructs. They act as `( )` does in the string-case: allow to quantify repetition over any sequence of contained pattern-expressions. In the 2D case, they can specify the repetition of rows, columns and even entire matrices.

There are three kind of groups: undirected groups (which enclose a matrix (or vector) of any shape), horizontal groups and vertical groups. The latter two groups are only introduced for convenience: both can always be expressed as an undirected group.

The main challenge with groups is that the contained pattern-expressions are allowed to have different shapes and sizes. This causes no theoretical problems: by padding the empty space with ⊠s, the groups would still be rectangular.

Undirected Groups An undirected group is a collection of patterns enclosed in parenthesis, of the form:

$$\begin{matrix}
\boxtimes & \frown & \cdots & \frown & \cdots & \frown & \boxtimes & \boxtimes \\
\texttt{(} & \rho_{1,1} & & \rho_{1,2} & \cdots & \rho_{1,k} & \texttt{)} & Q_h \\
\vdots & & \vdots & & \vdots & & \vdots & \boxtimes \\
\texttt{(} & \rho_{2,1} & \cdots & \rho_{2,k} & \cdots & \rho_{2,k} & \texttt{)} & \boxtimes \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \boxtimes \\
\texttt{(} & \rho_{m,1} & \cdots & \rho_{m,2} & \cdots & \rho_{m,k} & \texttt{)} & \boxtimes \\
\boxtimes & \smile & \cdots & \smile & \cdots & \smile & \boxtimes & \boxtimes \\
\boxtimes & Q_v & \boxtimes & \boxtimes & \boxtimes & \boxtimes & \boxtimes & \boxtimes
\end{matrix} \tag{25}$$

Here each $\rho_{i,j}$ for $0 < i \leq m$ and $0 < j \leq k$ is a pattern-expression, and the group contains $m \times k$ expression-patterns (which can be literals, choices, smaller groups, etc.). Some of these patterns are allowed to be empty, but only if either (or both) the other patterns in the same row or column are also empty. "Rows" and "columns" here refer to the numbering used for the contained expression-patterns; each row and column is

multiple rows/columns of symbols high/wide, respectively. The amount of characters used per row/column is the maximum height/width of any pattern-expression enclosed in it. The other pattern-expressions are padded with ⊠s (to their right and below).

The quantifiers Q_v and Q_h can be used to indicate repetition of the group in the vertical and horizontal direction respectively. Note that quantifiers are allowed to be empty (i.e., ⊠, see their definition, Eq. (15) and (20)), and that also repetition in both directions is allowed.

The relative positions of contained pattern-expression express the way they are concatenated. A target matrix $T \in \Sigma^{*\times *}$ matches a single repetition of a group if there exists strings in the mini-languages of the $\rho_{i,j}$'s such that T is the concatenation of these strings in the directions of the group.

For example, consider the following group:

$$\begin{array}{cccccccccc} \boxtimes & \frown & \frown & \frown & \frown & \frown & \frown & \frown & \boxtimes \\ (& [& a & , & b &] & \to^* & a &) \\ (& b & [& c & , & d &] & \to^* &) \\ \boxtimes & \smile & \smile & \smile & \smile & \smile & \smile & \smile & \boxtimes \end{array} \tag{26}$$

(perhaps easier to visualize as $\begin{bmatrix} [a,b]\to^* & a \\ b & [c,d]\to^* \end{bmatrix}$). This pattern would match $\begin{bmatrix} a & a \\ b & c \end{bmatrix}$, but also $\begin{bmatrix} b & b & a \\ b & d & d \end{bmatrix}$.

Single-Directional Groups Single-directional groups only contain horizontal brackets or only vertical brackets. They may only contain pattern-expressions in one direction, and may only use one quantifier in their direction. Furthermore, they may not contain or intersect groups in the other direction (when needed, such structures are probably modelled more clearly with undirected groups). The reason for this is simple: the amount of symbol-rows in horizontal groups and the amount of symbol-columns in vertical groups is fixed; the amount of symbols in the other axis can be stretched as required depending on the contained pattern-expressions.

These groups act as a shorthand when repetition in one direction is sufficient to express the desired search-pattern, but they clearly do not offer functional benefit over undirected-groups.

5.2.3 Combining Pattern-Expressions in one Search-Pattern

It remains to show how the highest non-nested pattern-expressions can actually be combined as a matrix S. This is straightforward: if there are multiple top-level (not-nested) pattern-expressions in a search-pattern S, then they are treated as one undirected group: without quantifiers or parenthesis, but with the same semantics, including the row/column structure described above (Section 5.2.2).

5.3 Search-Pattern Matching Algorithms

Recall from Definition 1 that in the 1D case, a search pattern $\vec{s}$ *matches* a substring $\vec{\tau}$ of a source-text $\vec{t}$ when each pattern-expression in $\vec{s}$ matches a piece (say a "sub-match") of $\vec{\tau}$. Formally we expressed these sub-matches as a string σ from the mini-language of the pattern-expression, but an equivalent perspective is to identifying a sub-match as the *indices* of the substring of $\vec{\tau}$ that is equal to σ. Seen this way, the match is successful when the concatenation of those sub-matches (in the order in which the expressions occur in $\vec{s}$) together form $\vec{\tau}$.

The same approach will be taken for 2D matches. We will continue from the perspective that pattern-expressions match to a subset of indices of the source-text (and multiple such subsets are possible for a given source-text). However, in 2D, matches are rectangles, and so are sub-matches of pattern-expressions. These sub-matches need to be concatenated along two axes. This introduces a new freedom, not seen in 1D: there are different ways in which this concatenation can be defined.

5.3.1 Ragged Matches

As already mentioned in Section 4.1, it is possible to allow "ragged-matches": simply allowing the sub-matches to be concatenated like a domino pieces in a domino game, with the only restriction that the resulting composite match must be connected. An example visualization is shown in Figure 1.

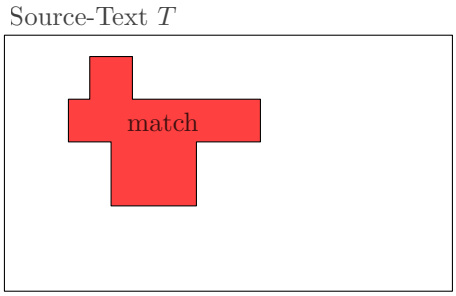


Figure 1: Visualization how the indices in a "ragged match" could be distributed over the source-text $T \in \Sigma^{*\times *}$.

While technically possible, it does not seem to be a good design philosophy to allow ragged matches. For an end-user, it would make it more difficult to predict how their search-pattern will match with a source-text. An even bugger problem is that one needs to find intuitive semantics for substituting a ragged match with a rectangular replace-pattern.

Furthermore, the pattern-expression `.` (especially in combination with quantifiers such as `?` and `*`) is already well suited to produce practically the same match as a ragged match, since it can match the remaining characters around the ragged pattern to form a rectangular match. This approach is also more explicit, and avoids having two mechanism for achieving the same effect (one mechanism would be ragged matches, the other would be to use matches padded with `.`s). When forbidding ragged matches, both matches and replacement-patterns are rectangles, which makes substitution semantics easier (although still more complicated than in 1D, as the shape of the source-text may still need to change).

There are counterarguments to this philosophy. First of all, it is based on human intuition, while 2D Search & Replace might also be useful as a subroutine for fully automated (perhaps even learned) algorithms, such as for PCG creation. Such algorithms do not require semantics to be intuitive for humans. A second argument is that padding with `.`s may require search-patterns to contain more boilerplate to obtain the same effect as a simpler search-pattern when using ragged matches.

In the remaining of this report, ragged matches will be forbidden; a match is defined to be a rectangle, and so are sub-matches of pattern-expressions.

5.3.2 Required Routines

It is now clear that all pattern-expressions must match to a rectangular match. This also means that we need to define an algorithm for each type of pattern-expression. In particular, if T is the source text in $\Sigma^{*\times *}$, S is our search-pattern in $\Xi_\Sigma^{*\times *}$ and P is pattern-expression in Λ_Σ or a group of pattern-expressions in $\Lambda_\Sigma^{*\times *}$, then we seek to define the following routines:

- MatchAllGroup(`T,P, order`) with $P \in \Lambda_\Sigma^{*\times *}$ being a group or a top-level search-pattern. This attempts to match all the sub-pattern-expressions, and align them in the row/column structure as explained in Section 5.2.2. The `order` indicates the order in which the sub-pattern-expressions should be matched (e.g. the order of visited indices of sub-patterns in P could be $(0,0),(1,0),(0,1),(2,0),(1,1),(0,2),\dots$).

- MatchAllLiteral(T, P) with P containing only a single literal. This simply checks if T contains the given literal, and returns the indices where it occurs in T.
- MatchAllRange(T, P). Here $P \in \Lambda_\Sigma$ is a range `[...]`, such as `[a-zA-Z]`. It recursively calls `MatchLiteral` for all the possible literals in the given range. The same mechanism can be used for `.` and $\begin{bmatrix} \square \\ \vdots \\ \square \end{bmatrix}$ ranges.
- MatchAllQuantifier(T, P). Here P contains a quantifier and a nested pattern-expression (a range or a group) that is being quantified. It can try to recursively match the quantified expression for any allowed amount of repetitions.

5.3.3 Matching Matrices of Pattern-Expressions

With only rectangular sub-matches, matching a search-pattern $S \in \Xi_\Sigma^{*\times *}$ with a source-text $T \in \Sigma^{*\times *}$ is relatively straightforward. Let $K \in \Lambda_\Sigma^{*\times *}$ be the high-level matrix of pattern-expressions that S encodes (K is the "compiled" search-pattern). For each pattern-expression $\rho \in \Lambda_\Sigma$ that occurs in K, we find a corresponding sub-match in T. If we can do this in such a way for all pattern-expressions, that the sub-matches can be combined into one larger rectangle $M \in 2^{\mathbb{N}\times\mathbb{N}}$, then M is a match of T to S. Of course, this combination of sub-matches must correspond to the relative alignment of the search-patterns in K. We chose to use the upper-left corner of a sub-match as the anchor point; the sub-match M_ρ of a pattern-expression ρ must be located in M directly below the first non-empty sub-match of ρ's first non-empty up neighbour's sub-match, and similarly it must be located directly left of the first non-empty left-neighbour of ρ. When no such neighbour exists in a particular direction, While this may seem very complex in words, it is quite intuitive when visualized, as seen in Figure 2.

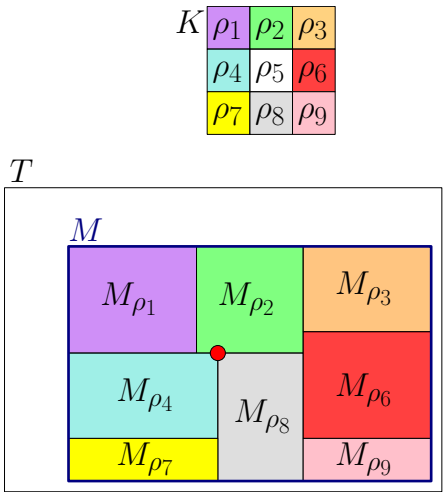


Figure 2: Visualization how sub-matches are combined in one large match M (which is a rectangular subset of indices of a source-text T). For each pattern-expression ρ_i in the abstract pattern-expression matrix $K\Lambda_\Sigma^{*\times *}$, M_{ρ_i} corresponds to the sub-match of ρ. The red dot gives the anchor point for M_{ρ_8}. Note that M_{ρ_5} is empty, which is allowed: hence M_{ρ_8}'s up-neighbour is M_{ρ_2}, and M_{ρ_8}'s left-neighbour is M_{ρ_4}.

It is important to realize that it cannot be deduced from the alignment of sub-patterns in K whether the corresponding sub-matches will be neighbours. Some sub-matches might be empty or extend so far in a certain

direction that they become neighbours of sub-matches of pattern-expressions in a different row/column of K.

5.3.4 Algorithm MatchAllGroup

The discussion above of compiled search-pattern matching is exactly the same for matching a group pattern-expression, which, like $K \in \Lambda_\Sigma^{*\times *}$, is a matrix of sub-expressions. In fact, we define the compiled search-patten K to be a group pattern-expression itself.

With this realization in mind, we can now formalize the matching algorithm for groups as MatchAllGroup in Algorithm 3. It uses a recursive subroutine MatchAllGroupRec, for which a basic implementation named MatchAllGroupRecBasic is shown in Algorithm 4 (a more advanced implementation will be covered in Section 5.4). Note that MatchAllGroup finds *all* possible rectangular (non-ragged) matches. In a backtracking matter, it traverses the sub-patterns of the outermost group of the compiled search-pattern K, and stores the resulting rectangular sub-match in a table called `rects` (only if compatible with the rectangles in already there). Note that multiple matches may exist for each sub-pattern: they are all tried with Depth-First Search (DFS) (or, in other words, every combination in the Cartesian product of the sub-matches is tried).

MatchAllGroup takes an argument `order`, which defines the order in which the entries of $K \in \Lambda_\Sigma^{*\times *}$ are being matched. The order should ensure that all sub-matches have been computed before computing the anchor-point of the next sub-match (which is the upper-left corner, which is discussed in more detail in Section 5.4.1). It suffices to simply iterate over all rows, and for each row iterate over all columns: a simple nested for-loop. This variant is called DefaultOrder, and is given in Algorithm 5. A call `order`$[i]$ gives the i^{th} sub-pattern of K, according to the given order. If K has no such element (i.e., when i is too large), then it returns nil. The order may not seem relevant when the full Cartesian product of all possible sub-matches per sub-pattern of K is explored, but this changes when optimizations are added to MatchAllGroup, which is done in Section 5.4.

MakeRect combines a matrix of rectangles in one large rectangle, and returns nil if this is not possible. Implementation-time optimizations of MatchAllGroupRec should ensure that the output nil never occurs here, since this would imply wasted computations that could have been pruned earlier. The implementation of this algorithm is further discussed in Section 5.5.

The expression PE refers to a pattern-expression in Λ_Σ. The notation PE.MatchAll(T, PE) is a shorthand for the matching subroutine corresponding to PE, which can be MatchAllGroup, MatchAllLiteral or MatchAllQuantifier.

Algorithm 3:
MatchAllGroup $(T, K, \texttt{order})$

`Input:`

- $T \in \Sigma^{*\times *}$: source text to find a match in.
- $K \in \Lambda_\Sigma^{*\times *}$: compiled search-pattern whose top-level pattern-expression is a group.
- `order` : $\mathbb{N} \times \mathbb{N} \to (\mathbb{N} \times \mathbb{N}) \cup \{\text{nil}\}$: mapping giving the next index of the sub-pattern in K to match, given the previously matched subpattern (returns nil when the last index is given as input).

`Output:` `matches` $\in 2^{(\mathbb{N}\times\mathbb{N})^2}$ set of rectangles of indices of T describing the matching subarrays of T, or $\emptyset$ when no match was found.

```
return MatchAllGroupRecBasic(T, K, order, 0, ∅) ;
```

Algorithm 4:
MatchAllGroupRecBasic $(T, K, \texttt{order}, i, \texttt{rects})$

`Input:`

- $T \in \Sigma^{*\times *}$: source text to find a match in.
- $K \in \Lambda_\Sigma^{*\times *}$: compiled search-pattern whose top-level pattern-expression is a group.
- `order` : $\mathbb{N} \times \mathbb{N} \to (\mathbb{N} \times \mathbb{N}) \cup \{\text{NIL}\}$: mapping giving the next index of the sub-pattern in$K$ to match, given the previously matched subpattern (returns NIL when the last index is given as input).
- $i \in \mathbb{N}$: index of next top-level pattern-expression in K to match, i.e., the $(i+1)^{th}$ sub-pattern in K according to `order`.
- `rects`: a set of rectangles $((x_1, y_1), (x_2, y_2)) \in (\mathbb{N} \times \mathbb{N})^2$. This should be the empty set in the initial call.

`Output:` `matches` $\in 2^{(\mathbb{N}\times\mathbb{N})^2}$ set of rectangles of indices of T describing the matching subarrays of T, or $\emptyset$ when no match was found.

```
1  matches ← ∅;
   /* Pick the next not-yet-matched sub-Pattern-Expression (PE),          */
   /* and try to match it with the unmatched space in T.                  */
2  PE ← order(K, i) ;
3  if PE = NIL then
4  |  return {MakeRect(rects)};
   /* Match the sub-pattern-expression PE, using the correct MatchAll call (PE can be
      any type of pattern-expression).                                    */
5  PE_matches ← PE.MatchAll(T, PE) ;
6  if PE_matches = ∅ then
7  |  return ∅ ;
   /* Implementation note: many iterations of the following loop can be saved, when
      known beforehand that rects ∪ {m} can never be combined in one large rectangle.
      */
8  for match m ∈ PE_matches do
9  |  r ← MatchAllGroupRecBasic(T, K, order, i + 1, rects ∪ {m}) ;
   |  /* It is possible that r = ∅, but this does not lead to problems.  */
10 |  matches ← matches ∪ r ;
11 return matches ;
```

Algorithm 5:
DefaultOrder (K, i)

`Input:` $K \in \Lambda_{\Sigma}^{n \times m}$: compiled search-pattern whose top-level pattern-expression is a group.
`order` : $\mathbb{N} \times \mathbb{N} \to (\mathbb{N} \times \mathbb{N}) \cup \{\text{NIL}\}$: mapping giving the next index of the sub-pattern in$K$ to match, given the previously matched subpattern (returns NIL when the last index is given as input). $i \in \mathbb{N}$: (row, column) index (in K) of the sub-pattern to return.

`Output:` $\text{PE} \in \Lambda_{\Sigma}$: the $(i+1)^{th}$ pattern-expression of K when iterating over all rows, and for each row over all columns, or NIL when K has no $(i+1)^{th}$ entry.

```
1 c ← i  mod (m) ;
2 r ← ⌊i⌋n ;
3 if r̂ ≥ n then
4 |  return NIL ;
5 return K[r, c] ;
```

Lemma 1. *Given some implementation of MakeRect, then for a given non-empty group $K \in \Lambda_{\Sigma}^{*\times *}$ and a source text $T \in \Sigma^{*\times *}$, then any existing combination of sub-matches of the pattern-expressions in K that make MakeRect return a non-NIL value, will be found by MatchAllGroup (Algorithm 3), when using MatchAllGroupRecBasic (Algorithm 4) as implementation for MakeAllGroupRec and DefaultOrder (Algorithm 5) as* `order` *parameter.*

Proof. The correctness of brute-force DFS may, at first sight, appear straightforward. However, there is a caveat: the pattern-expressions in K may themselves be groups. Hence we will proceed by induction on the depth of group pattern-expression nestings $n \in N$. To be clear, for $n = 0$, neither K nor any of its pattern-expressions (nor their nested pattern-expressions, nor theirs, etc.) contains a group pattern-expression. For $n = 1$, either one of K's contained pattern-expressions is a group, or one of their nested pattern-expressions, or one of theirs, etc.

Base case: $n = 0$
Let y be the amount of sub-patterns in K (i.e., `order.next`$(K, y-1) \neq$ NIL but `order.next`$(K, y) =$ NIL). We now prove the following claim:

Claim 1.1. *For all $i \in \mathbb{N}$ with $0 \leq i \leq y$, and a set of rectangles* `rects`*,*
MatchAllGroupRecBasic$(T, S,$ `order`*$, i,$* `rects`*) returns the set of of rectangles:*

$$\left\{\mathcal{R} \ : \ \mathcal{R} = \textit{MakeRect}\left(\texttt{rects} \cup \{\sigma_i, \ldots, \sigma_{y-1}\}\right), \forall_{j=i}^{y-1}[\sigma_j \in PE.\textit{MatchAll}(T, \texttt{PE}),\ \texttt{PE} = \texttt{order}(K, j)]\right\} \tag{27}$$

The idea behind this claim is that the i^{th} sub-pattern of K, and sub-patterns with a higher index, will try to complement `rects`, with every possible combination of their sub-matches, and returns all the attempts deemed successful by MakeRect.

Proof of Claim 1.1. Proceed with induction on $i \in \mathbb{N}$ with $0 \leq i < y$, in inverse order:

Base case: $i = y$
In this case, the expression $\{\sigma_i, \ldots, \sigma_{y-1}\}$ of Eq. 27 is the empty set. So MatchAllGroupRecBasic is only expected to return {MakeRect(`rects`). Now since $i > y - 1$, `order`$(K, i) =$ NIL, so Line 3 will return exactly this set.

Inductive step: $0 \leq i < y$
Induction Hypothesis: the claim holds for all $j \in \mathbb{N}$ such that $i < j \leq y$. Now we need to show that the claim

also holds for i. We have $0 \leq i \leq y-1$, so PE $\leftarrow$ $\texttt{order}(K, i)$ (Line 2) returns a well-defined sub-pattern. Then in Line 5, we obtain all possible matches for this sub-pattern (stored in `PE_matches`).

Observe that the for-loop of Line 8 computes:

$$\texttt{matches} = \bigcup_{m \in \texttt{PE\_matches}} \textsc{MatchAllGroupRecBasic}(T, S, \texttt{order}, i+1, \texttt{rects} \cup \{m\}) \tag{28}$$

But by the Induction Hypothesis, this is equivalent to:

$$\bigcup_{m \in \texttt{PE\_matches}} \Big\{ \mathcal{R} \;:\; \mathcal{R} = \textsc{MakeRect}\left(\texttt{rects} \cup \{m\} \cup \{\sigma_{i+1}, \ldots, \sigma_{y-1}\}\right), \\ \forall_{j=i}^{y-1}[\sigma_j \in \text{PE.}\textsc{MatchAll}(T, \texttt{PE}),\ \texttt{PE} = \texttt{order}(K, j)] \Big\} \tag{29}$$

Now rewrite $m = \sigma_i$, and migrate the $\bigcup_{m \in \texttt{PE\_matches}}$ into the semantics of the set-builder notation, and we obtain:

$$\Big\{ \mathcal{R} \;:\; \mathcal{R} = \textsc{MakeRect}\left(\texttt{rects} \cup \{\sigma_i, \ldots, \sigma_{y-1}\}\right), \\ \forall_{j=i}^{y-1}[\sigma_j \in \text{PE.}\textsc{MatchAll}(T, \texttt{PE}),\ \texttt{PE} = \texttt{order}(K, j)] \Big\}, \tag{30}$$

which is indeed Eq. 27 of the claim. Finally, we observe that `matches` is also returned, as expected. □

Now observe that MatchAllGroup initially calls MatchAllGroupRecBasic with $i = 0$ and $\texttt{rects} = \emptyset$, so by Claim 1.1, MatchAllGroupRecBasic will return the rectangles that are the output of MakeRect for any accepted combination of sub-matches of all the patterns in K.

Inductive case: $n > 0$

Induction Hypothesis: for all $0 \leq \hat{n} < n$, *MatchGroupAll will return the union of* MakeRect(`rects`) *for all combinations of sub-matches* `rects` *of the sub-patterns in* K, *if the depth of nestings of group-patterns in* K *is at most* $n-1$. Note that any group-pattern nested in K, either directly or indirectly, is equivalent to a search-pattern whose maximum group-pattern nesting depth is $n-1$. So by the IH, any such group-pattern PE will execute PE.MatchAll correctly. Hence for K with nesting depth n we can treat nested group-patterns as any other pattern-expression. The remainder of the argument is the same as for $n = 0$.

Conclusion

We have shown that the lemma holds for the nesting-depth of $n = 0$, and for any depth n when it holds for all maximum depths $n-1$. So by induction, it holds for all nesting depths. This concludes the proof. □

5.4 Fail-Fast Optimization

The algorithm MatchAllGroup (Algorithm 6) as presented earlier is rather inefficient. It computes the full Cartesian product over the sets of sub-matches of all sub-patterns in the given group K. This will likely result in a great deal of unnecessary work. For example, after picking a sub-match for the first two sub-patterns, it is already clear that they will never form a rectangle together when they overlap. Yet MatchAllGroup will continue to try all possible combinations of the sub-matches of the remaining sub-patterns; these are all wasted computations! Note that overlap is not necessarily the only issue. The shape of the sub-matches found so far may become so ragged that there is no way the remaining matches can be picked to complete a rectangle. Or the sub-matches may not be aligned in the way that the corresponding sub-patterns are aligned in K.

This subsection will first formalize the notion of a *match* in more detail, especially the semantic constraints. Then it will present two optimisations: the first is to stop early when a sub-pattern's upper-left corner is not

in the required position, and the second restricts the search space of sub-patterns. It concludes with improved versions of MatchAllGroup and MatchAllRec.

5.4.1 Formalizing a Match

In order to discuss the optimizations for the matching algorithm, it is useful to repeat the definition of a (sub-)match, and to add a few new definitions.

A match $M_\rho \in (\mathbb{N}\times\mathbb{N})^{n\times m}$ is an n by m matrix, whose elements are indices $(i,j) \in \mathbb{N}\times\mathbb{N}$ of some source-text $T \in \Sigma^{*\times *}$. Recall that a pattern expression $\rho \in \Lambda_\Sigma$ matches T in a match M_ρ if the indices of T described by M_ρ are in the mini-language described by ρ. A *sub-match* is a match belonging to a pattern-expression $\rho \in \Lambda_\Sigma$ that is nested in another pattern-expression. Such nested expressions are also called *sub-patterns*.

Matchings as Mapping In a more abstract sense, we can see a match M_ρ as a mapping of a pattern-expression to a subset of T. The purpose of the algorithm MatchAllGroup(T, K, ORDER) is then to find all possible mappings M of sub-patterns in the $h \times w$ matrix K to sub-matches (each describing a subset of T), such that a collection of constraints hold.

When representing M as a matrix, it is notationally convenient to map the *indices* of M to the matches: technically $M\colon (\mathbb{N}\times\mathbb{N}) \to 2^{\mathbb{N}\times\mathbb{N}}$. However, one can clearly interpret it as $M\colon K \to 2^{\mathbb{N}\times\mathbb{N}}$, i.e., a mapping of the sub-patterns to their sub-matches.

The required constraints of M are:

1. The union of the matches covers all indices of T:

$$\bigcup_{i=0}^{h-1}\bigcup j = 0^{w-1}M[i,j] = \{(r,c) \mid 0 \le r < n \wedge 0 \le c < m\} \tag{31}$$

2. There is no overlap between any pair of matches:

$$\forall_{(i,j),(\hat{i},\hat{j})}\left[0 \le i,\hat{i} < h,\ 0 \le j,\hat{j} < w, (i,j) \ne (\hat{i},\hat{j})\ :\ M[i,j] \cap M[\hat{i},\hat{j}] = \emptyset\right] \tag{32}$$

3. The alignment of left-upper corners of sub-matches correspond to the respective positions of sub-patterns in K (see Figure 2):

$$\mathrm{UL}\left(M[i,j]\right).r = \begin{cases} 1 + \max_{r\in\mathbb{N}}\left\{(r,c) \in M[\hat{i},j] \mid c \in \mathbb{N}\right\} & \text{if } i > 0 \\ & M[\hat{i},j] := \max_{k\ge 1, k\in\mathbb{N}}\left\{M[i-k,j] \mid M[i-k,j] \ne \emptyset\right\} \\ 0 & \text{if } i = 0 \text{ or } M[\hat{i},j] \text{ does not exist} \end{cases} \tag{33}$$

$$\mathrm{UL}\left(M[i,j]\right).c = \begin{cases} 1 + \max_{c\in\mathbb{N}}\left\{(r,c) \in M[i,\hat{j}] \mid r \in \mathbb{N}\right\} & \text{if } j > 0 \\ & M[i,\hat{j}] := \max_{k\ge 1, k\in\mathbb{N}}\left\{M[i,j-k] \mid M[i,j-k] \ne \emptyset\right\} \\ 0 & \text{if } j = 0 \text{ or } M[i,\hat{j}] \text{ does not exist} \end{cases} \tag{34}$$

Eq. (33) and (33) seem rather complicated, but the idea is straightforward. Let $\rho_{i,j}$ be the $(i,j)^{\text{th}}$ element of K. Then Eq. (33) specifies that the left-upper row coordinate of $\rho_{i,j}$ is 1 cell below its upper-neighbour (the lowest cell of the upper-neighbour). In a symmetrical way, Eq. (34) specifies that the left-upper column coordinate of $\rho_{i,j}$ is 1 cell to the left of the left-neighbour's leftmost cell. The complexities arise from the

special cases: the upper or left neighbour of $\rho_{i,j}$ might map to an empty match. The solution is to use the first sub-pattern above or to the left of $\rho_{i,j}$ that does not map to an empty match. This are $M[\hat{i}, j]$ and $M[i, \hat{j}]$ respectively. Finally, there is the case that $\rho_{i,j}$ is in the first row or the first column of K (or that *all* sub-patterns above or to the left of $\rho_{i,j}$ map to empty matches). In this case we simply set the corresponding index of the left-upper corner to 0. So, for example, it follows that $\mathrm{UL}(M[0,0]) = (0,0)$ always holds.

Empty Matches To avoid confusion, it is important to realize that matches are allowed to be empty. For example, in a string, `.*` would match the empty string. Similarly in 2D, some quantified pattern-expressions such as `([abc])`$\rightarrow^*$ would also match an empty sub-matrix. An empty match simply has the value $\emptyset$. In the context of algorithms, it is possible that a value for a match has not yet been computed. To avoid confusion with a computed empty match, the value NIL will be used for not-yet-computed matches.

The possibility of empty matches complicate the design of algorithms. For example, the sub-match $M_{\rho_{i,j}}$ of some $\rho_{i,j} \in K \in \Lambda_\Sigma^{*\times *}$ may not be neighbouring (in terms of indices of T that they cover) the sub-match $M_{\rho_{i-1,j}}$ nor $M_{\rho_{i,j-1}}$ if these are empty. It is possible that the left neighbour of $M_{\rho_{i,j}}$ is $M_{\rho_{i-99,j-99}}$! This becomes especially tricky when computing the desired location for the Upper-Left corner of a match, or when restricting the sub-pattern to a subset of T to find a match in.

5.4.2 Impossible matches

Recall that a recursive call of MatchAllGroupRec receives a set of already-matches sub-matches as its `rects` argument, and the index i a sub-pattern $\rho = \texttt{order}(K, i) \in \Lambda_\Sigma$ to find all sub-matches for. The first optimization we propose tries to identify some of the scenarios where the upper-left corner of ρ's bounding-box is *ill-defined*. If ρ's upper-left corner is ill-defined, then MatchAllGroupRec can immediately return NIL – saving the effort of finding all possible combinations of sub-match for the remaining sub-patterns in K (that come after i).

We write $\mathrm{UL}(H)$ denote the left-upper index of a rectangular match H. So, for example, when $H = \{(2,2),(2,3),(3,2),(3,3)\}$, then $\mathrm{UL}(H) = (2,2)$. Similarly, $\mathrm{LR}(H)$ denotes the index of the lower-right corner point (that is $(3,3)$ in the example of H). Furthermore, let $\vec{p}.r$ and $\vec{p}.c$ to denote the row and column index value of an index $\vec{p} \in \mathbb{N} \times \mathbb{N}$.

We can now define *ill-defined* upper-left corners as follows:

Definition 2 (Ill-defined upper-left corner of a sub-match)**.** *Let $M_{\rho_{i,j}}$ be any sub-match for a sub-pattern $\rho_{i,j} = K[i,j] \in \Lambda_\Sigma$, and let* `rects` *be a set of earlier computed matches, which includes a sub-match for at least every $K[i',j']$ with $(0 \le i' < i \wedge 0 \le j' \le j)$ or $(0 \le i' \le i \wedge 0 \le j' < j)$. The upper-left corner $UL(M[i,j]) = (r,c)$ is* ill-defined *when at least one of the following holds:*

1. *The index $UL(M[i,j]) = (r,c)$ is contained in a rectangle in* `rects`*.*
2. *There exist an index (r',c') in T with $r' < r \wedge c' \le c$ or $r' \le r \wedge c' < c$ that will not be covered by any rectangle in $(\texttt{rects} \cup ((r,c),(r,c))$. That is, no valid sub-match of ρ with upper-left corner (r,c) can be added to* `rects` *in order for a rectangle in* `rects` *to contain this entry (r',c').*

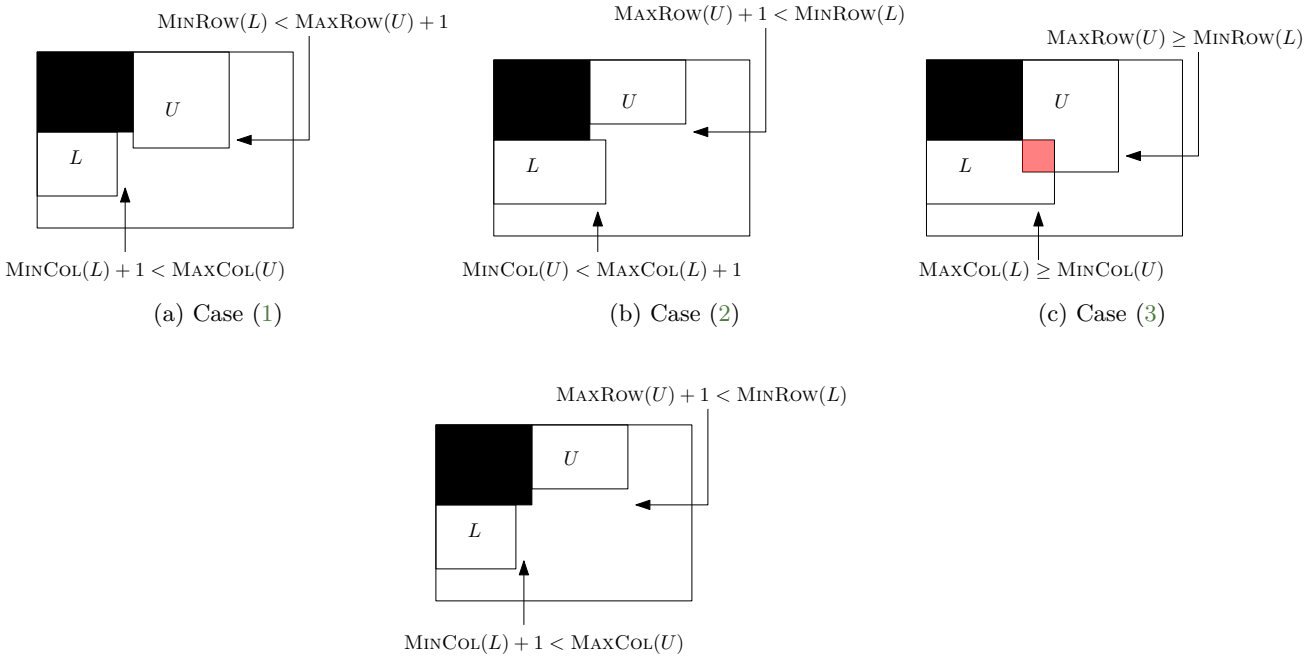
Note that, generally, all possible sub-matches of a sub-pattern must have the same upper-left corner. An exception is the first sub-pattern that does not map to an empty sub-match (this will be $K[0,0]$ in most cases); this sub-pattern can set the upper-left corner of the composite match (the rectangle that is the collection of all the sub-matches).

Ill-defined upper-left corners occur only under certain conditions. Assume we wish to check for the sub-pattern $\rho_{i,j} = K[i,j]$ whether $\mathrm{UL}(M[i,j])$ is well-defined. For convenience, let $\textsc{MaxRow}(H)$ and $\textsc{MinRow}(H)$ denote the minimum and maximum row indices included in a match H, and similarly let $\textsc{MaxCol}(H)$ and

$\textsc{MinCol}(H)$ denote the minimum and maximum column indices. Furthermore, let the first non-empty up neighbour of $\rho_{i,j}$ be $U = \max_{k\geq 1, k\in\mathbb{N}} \{M[i-k,j] \mid M[i-k,j] \neq \emptyset\}$, and let the first non-empty left neighbour be $L = \max_{k\geq 1, k\in\mathbb{N}} \{M[i,j-k] \mid M[i,j-k] \neq \emptyset\}$. We assume that both $\mathrm{UL}(U)$ and $\mathrm{UL}(L)$ satisfy Eq. (33) and (33) (and hence $\rho_{i,j}$ is assumed not to be in the first row nor in the first column of M).

Now, the following are clearly sufficient conditions for $\mathrm{UL}(M[i,j])$ to be ill-defined:

1. $\textsc{MinRow}(L) < \textsc{MaxRow}(U) + 1 \ \wedge \ \textsc{MinCol}(L) + 1 < \textsc{MaxCol}(U)$
   (A gap on the left, see Figure 3a)
2. $\textsc{MaxRow}(U) + 1 < \textsc{MinRow}(L) \ \wedge \ \textsc{MinCol}(U) < \textsc{MaxCol}(L) + 1$
   (A gap above, see Figure 3b)
3. $\textsc{MaxRow}(U) > \textsc{MinRow}(L) \ \wedge \ \textsc{MaxCol}(L) > \textsc{MinCol}(U)$
   (L and U overlap, see Figure 3c)
4. $\textsc{MaxRow}(U) + 1 < \textsc{MinRow}(L) \ \wedge \ \textsc{MaxCol}(L) + 1 < \textsc{MinCol}(U)$
   (The corner is would overlap with other matches than L and U, see Figure 3d)



(a) Case (1) (b) Case (2) (c) Case (3)

(d) Case (4)

Figure 3: Four sufficient conditions for $\mathrm{UL}(M[i,j])$ to be ill-defined, where L and U are the matches corresponding to the first Left and Upper neighbour of $K[i,j]$ that does not map to an empty match. Matches U and L are projected on the matrix T. The black area represents "older" matches that were needed to define $\mathrm{UL}(U)$ and $\mathrm{UL}(L)$.

Now define the following conditions:

(OK1) $\textsc{MinRow}(L) = \textsc{MaxRow}(U) + 1$

(OK2) $\textsc{MinCol}(U) = \textsc{MaxCol}(L) + 1$

At least one of these conditions must hold for $\mathrm{UL}(M[i,j])$ to be well-defined:

Lemma 2. *If $\neg(OK1) \wedge \neg(OK2)$ then $UL(M[i,j])$ is ill-defined.*

Proof. Assume $\neg$(OK1) $\wedge$ $\neg$(OK2). Both are equality conditions, so the negation of either condition could imply either a "<" or a ">" condition. We now distinguish the four possible cases created this way.

Case 1: both smaller Then we have:

$$\begin{aligned} &\textsc{MinRow}(L) < \textsc{MaxRow}(U) + 1 \ \wedge \ \textsc{MinCol}(U) < \textsc{MaxCol}(L) + 1 \\ \Rightarrow\ &\textsc{MaxRow}(U) \geq \textsc{MinRow}(L) \ \wedge \ \textsc{MaxCol}(L) \geq \textsc{MinCol}(U) \end{aligned}$$

Hence this implies impossibility case (3).

Case 2: first condition smaller, second greater That is:

$$\begin{aligned} &\textsc{MinRow}(L) < \textsc{MaxRow}(U) + 1 \ \wedge \ \textsc{MinCol}(U) > \textsc{MaxCol}(L) + 1 \\ \Leftrightarrow &\textsc{MinRow}(L) < \textsc{MaxRow}(U) + 1 \ \wedge \ \textsc{MaxCol}(L) + 1 < \textsc{MinCol}(U) \end{aligned}$$

which is equivalent to impossibility case (1).

Case 3: first condition greater, second smaller That is:

$$\begin{aligned} &\textsc{MinRow}(L) > \textsc{MaxRow}(U) + 1 \ \wedge \ \textsc{MinCol}(U) < \textsc{MaxCol}(L) + 1 \\ \Leftrightarrow &\textsc{MaxRow}(U) + 1 < \textsc{MinRow}(L) \ \wedge \ \textsc{MinCol}(U) < \textsc{MaxCol}(L) + 1 \end{aligned}$$

which is also equivalent to an impossibility case, namely (2).

Case 4: both greater That is:

$$\begin{aligned} &\textsc{MinRow}(L) > \textsc{MaxRow}(U) + 1 \ \wedge \ \textsc{MinCol}(U) > \textsc{MaxCol}(L) + 1 \\ \Leftrightarrow &\textsc{MaxRow}(U) + 1 < \textsc{MinRow}(L) \ \wedge \ \textsc{MaxCol}(L) + 1 < \textsc{MinCol}(U) \end{aligned}$$

which is equivalent to the last remaining impossibility case (4). □

5.4.3 Restricting the Search Space for Sub-Matches

The second optimization we propose is to restrict the subset of the indices of T where a sub-pattern may try to find a sub-match. We call a such subset of indices a *bounding box*. This optimization will already trim the search of overlapping sub-matches, and some of the matches that do not correspond to the alignment in K.

This optimization is implemented in the subroutine GetBoundingBox (Algorithm 6), and MatchAllGroupRecOptimized (Algorithm 8) shows where it is called. The routine GetBoundingBox finds the sub-array of T in which a sub-pattern of K is allowed to search for a sub-match. Only this subset of T is passed to the MatchAll method of the sub-pattern when MatchAllGroupRecOptimized queries the set of sub-matches. Note that GetBoundingBox uses both the already-matched rectangles in `rects` to avoid overlap, and the alignment of the pattern-expressions in K, to determine the bounding box.

Figure 4 gives an example of a bounding box in practice.

When queried to find the bounding box of a sub-pattern $\rho_{i,j} \in \Lambda_\Sigma$, given previous matches M, GetBoundingBox (Algorithm 6) proceeds as follows (assuming both preconditions hold). It first finds the indices

(r_{ul}, c_{ul}) at which the upper-left corner of any non-empty sub-match of $\rho_{i,j}$ would be located (according to Eq. (33) and Eq. (34)). Then it linearly tries cells to directly the left of this upper-left corner (i.e. on row r_{ul}, but all columns left of c_{ul}), until it finds a cell that is either already occupied, or the end of the source-text has already been reached (i.e. c_{max}). Here "occupied" (the function OCCUPIED in pseudocode) simply means that a cell is not contained in a rectangle in M. The intuition behind this is that no overlap may occur, which clearly leads to an impossible compound match. The last valid column visited is saved as the index for the lower-right corner.

This process is then repeated in the downward direction: all rows r below r_{ul} are tried until the last row of the source-text (max) is reached, or until $r+1$ is occupied. The final value of r is then used as the lower-right corner.

In an earlier draft, we had GETBOUNDINGBOX also stop traversing left if it had reached a column $c > c_{ul}$ such that $(r_{ul} - 1, c)$ was not occupied (and a symmetrical concept for the down direction). However, this is not correct: a sub-pattern at a column $c' > c_{ul}$ may still fill a "gap" that is left above $M[i, j]$. An example of this is visualized in Figure 5. Of course, the same reasoning holds for the downward case.

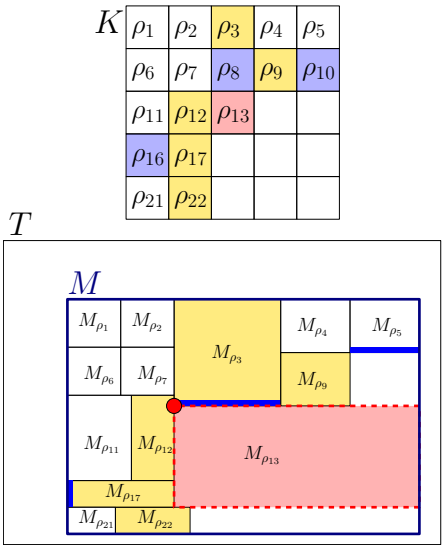


Figure 4: Example of the shape of a bounding box. The upper figure shows the relative alignment of sub-patterns in a group matrix $K \in \Lambda_{\Sigma}^{5\times 5}$. All included symbols refer to sub-patterns for which already a sub-match has been chosen (in the current combination of sub-matches), except ρ_{13} whose bounding box is of interest. The lower figure shows the corresponding source-text $T \in \Sigma^{*\times *}$, with the previously matched sub-matches drawn in. The resulting bounding box for $M_{\rho_{13}}$ (ρ_{13}'s sub-match) is given as the red dotted rectangle. Yellow sub-matches indicate neighbours of $M_{\rho_{13}}$, and blue sub-matches indicate empty matches. The red dot is the location of $\mathrm{UL}(M_{\rho_{13}})$. The large rectangle M is the largest composite match that is still possible at this point.

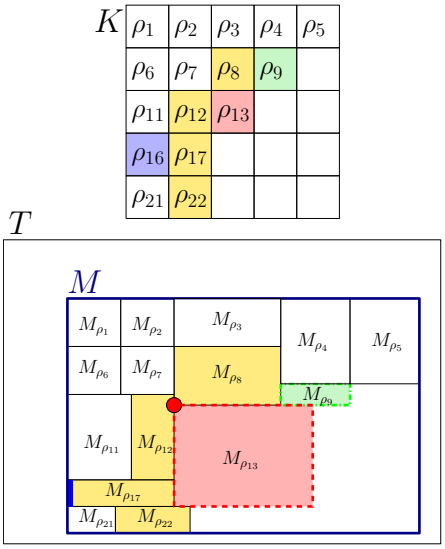


Figure 5: Counterexample to false intuition that bounding boxes should not allow a sub-match to leave "gaps" above or to the left of it. The usage of names and colours is the same as in Figure 4. Assume that ρ_9 and ρ_{13} have not yet been matched, but all other named sub-patterns of the group K have. Then the red rectangle shows the bounding box for ρ_{13}. It leaves a gap to the upper-right, but the definition of the upper-left corner (Eq. (33) and Eq. (34)) still allow ρ_9 to fill this gap later.

Algorithm 6:
GetBoundingBox $(M, r_{max}, c_{max}, i, j)$

Input:

- M: Mapping of indices to sub-matches. In particular, M is a $h \times w$ matrix of previously matches rectangles $((r_1, c_1), (r_2, c_2))$ and NIL elements. $M \in \left((\mathbb{N} \times \mathbb{N})^2 \cup \{\text{NIL}\}\right)^{h \times w}$, $h, w \in \mathbb{N}$.
- $r_{max}, c_{max} \in \mathbb{N}$: maximum row and column index of any possible rectangle, i.e., the size of the grid.
- $i, j \in \mathbb{N}$: indices of the element of M to compute the bounding box of.
- **Precondition 1:** $M[i', j'] = \text{NIL}$ for all $(i', j') \in [0, h-1] \times [0, w-1]$ such that $i' \geq i \wedge j' \geq j$.
- **Precondition 2:** $M[i', j'] \neq \text{NIL}$ for all $(i', j') \in [0, h-1] \times [0, w-1]$ such that $0 \leq i' < i \wedge 0 \leq j' \leq j$ or $0 \leq i' \leq i \wedge 0 \leq j' < j$.

Output: A rectangle $((r_1, c_1), (r_2, c_2)) \in (\mathbb{N} \times \mathbb{N})^2$ that gives the bounding box of $M[i, j]$ in the grid, or NIL if there exists no valid bounding box for $M[i, j]$.

```
if x = 0 ∧ y = 0 then
    /* The entire grid is still vacant.                                    */
    return ((0, 0), (r_max, y_max))
Let (r_ul, c_ul) be the upper-left corner for any sub-match at (i, j) according to Eq. (33) and Eq. (34) ;
if (r_ul, c_ul) is ill-defined according to Lemma 2 then
    return NIL ;
r ← r_ul ;
while r + 1 ≤ r_max ∧ ¬Occupied(M, r + 1, c_ul) ∧ (c_ul = 0 ∨ Occupied(M, r + 1, c_ul − 1)) do
    r ← r + 1 ;
while c + 1 ≤ c_max ∧ ¬Occupied(M, r_ul, c + 1) ∧ (r_ul = 0 ∨ Occupied(M, r_ul − 1, c + 1) do
    c ← c + 1 ;
return ((r_ul, c_ul), (r, c)) ;
```

5.4.4 Correctness of GetBoundingBox

The purpose of GetBoundingBox is to find a subset of the source-text in which a sub-pattern can search for a sub-match, in a way that it will not cause conflicts with other sub-matches. Obviously, overlapping sub-matches cause conflicts when combining the sub-matches of the different sub-patterns into one large rectangle. So for this reason, we want GetBoundingBox to return bounding boxes that do not overlap with already determined sub-matches of other sub-patterns. The following lemma shows that this behaviour is actually achieved, at least under certain pre-conditions:

Lemma 3. *Consider an input $M \in ((\mathbb{N} \times \mathbb{N})^2 \cup \{\text{NIL}\})^{n \times m}$ $(n, m \in \mathbb{N})$ of previously fixed sub-matches, belonging to a source-text $T \in \Sigma^{r_{max}+1 \times c_{max}+1}$, and two indices $i, j \in [0, n] - 1 \times [0, m-1]$ such that $M[i, j] =$ NIL, for which it holds that:*

- *It satisfies Precondition 1 and 2 of GetBoundingBox (Algorithm 6).*
- *All non-NIL sub-match in M satisfy Eq. (33) and (34) (i.e., that have their upper-left corner in the expected position), and their upper-left corners are well-defined.*
- *The non-NIL sub-matches in M do not overlap.*

Then GetBoundingBox$(M, r_{max}, c_{max}, i, j)$ *(Algorithm 6) will return a rectangle that does not overlap with a sub-match in* M*.*

Proof. Assume for contradiction that the above conditions are met,
but that GetBoundingBox$(M, r_{max}, c_{max}, i, j)$ still outputs a rectangle $Z = ((r_{ul}, c_{ul}), (r_{lr}, c_{lr}))$ that overlaps with other rectangles in M.
Let $X = ((r_1, c_1), (r_2, c_2))$ be the rectangle in M with the minimum row-coordinate in its upper-left cell of all the rectangles of M that overlap with Z.

Algorithm GetBoundingBox explicitly checks for overlap with Z in the row r_{ul} for columns in $[c_{ul}, c_{lr}]$ (see Lines 7-8), and in the column c_{ul} for rows in $[r_{ul}, r_{lr}]$ (Lines 9-10). Hence X cannot overlap with Z in row r_{ul} or column c_{ul}.

It also follows, in particular, that:

$$\mathrm{UL}(X).r \neq r_{ul} \ \vee \ \mathrm{UL}(X).c \neq c_{ul} \tag{35}$$

We proceed with a case distinction on Eq. (35):

Case 1: $\mathbf{UL}(X).r < r_{ul} \ \vee \ \mathbf{UL}(X).c < c_{ul}$
Then, in order for X to overlap with Z, it must contain the point (r_{ul}, c_{ul}), which contradicts our observation that X contains no cells in row r_{ul} (nor in column c_{ul}). See Figure 6.

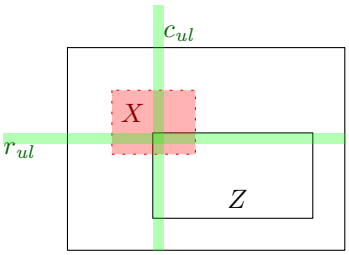


Figure 6: First case of the case-distinction in the proof of Lemma 3: $\mathrm{UL}(X).r < r_{ul} \ \vee \ \mathrm{UL}(X).c < c_{ul}$. So X must overlap with Z in either row r_{ul} or column c_{ul} (in green).

Case 2: $\mathbf{UL}(X).r > r_{ul} \ \wedge \ \mathbf{UL}(X).c > c_{ul}$
Suppose that neither $\mathrm{UL}(X).r > r_{lr}$ nor $\mathrm{UL}(X).c > c_{lr}$. That is, X is entirely contained in Z. Then X must have an ill-defined upper-left corner: it has no up-neighbour rectangle from M (by the way we picked X from the set of all rectangles overlapping with Z), nor can we have $\mathrm{UL}(X) = 0$ because $\mathrm{UL}(X).r > r_{ul} \geq 0$.
See also Figure 7.

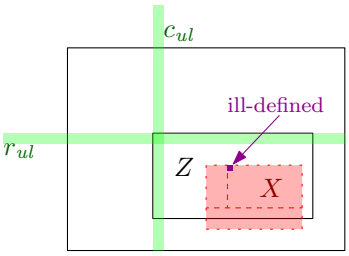
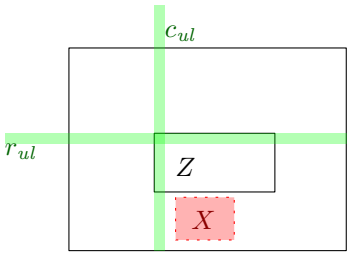
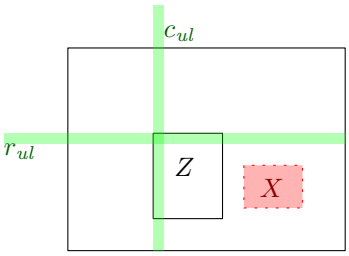


Figure 7: Second case of the case-distinction in the proof of Lemma 3: X is in Z, that is, $\mathrm{UL}(X).r > r_{ul} \wedge \mathrm{UL}(X).c > c_{ul}$ but neither $\mathrm{UL}(X).r > r_{lr}$ nor $\mathrm{UL}(X).c > c_{lr}$. Then X does have an ill-defined upper-left corner, at it must have at least one vacant cell above $\mathrm{UL}(X)$. Note that the case distinction does not assert that X is the only overlapping match (hence others are drawn as well), but it is among the ones with the least row-index for the upper-left corner.

So either $\mathrm{UL}(X).r > r_{lr}$ or $\mathrm{UL}(X).c > c_{lr}$. If $\mathrm{UL}(X).r > r_{lr}$, then X is entirely below Z, and can therefore not overlap with Z (Figure 8a). Similarly, if $\mathrm{UL}(X).c > c_{lr}$, then X is entirely right of Z and can also not overlap with Z (Figure 8b).



(a) Case $\mathrm{UL}(X).r > r_{lr}$: X is below Z.



(b) Case $\mathrm{UL}(X).c > c_{lr}$: X is right of Z.

Figure 8: Remaining sub-cases for case 2 of the case-distinction in the proof of Lemma 3: either $\mathrm{UL}(X).r > r_{lr}$ or $\mathrm{UL}(X).c > c_{lr}$.

Case 3: $\mathbf{UL}(X).r > r_{ul} \wedge \mathbf{UL}(X).c \leq c_{ul}$
Then X must overlap with Z in column c_{ul} if X is to overlap with any point with Z (Figure 9).

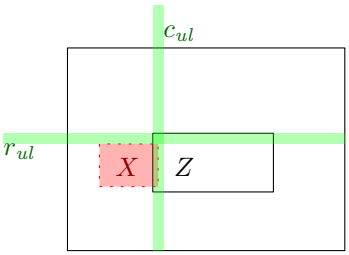


Figure 9: Third case of the case-distinction in the proof of Lemma 3: $\mathrm{UL}(X).r > r_{ul} \wedge \mathrm{UL}(X).c \leq c_{ul}$ So X must overlap in the column c_{ul} (in green).

Case 4: $\mathbf{UL}(X).r \leq r_{ul} \wedge \mathbf{UL}(X).c > c_{ul}$
This case is symmetrical to case 3, but with rows and columns mirrored: now X must overlap with Z in row r_{ul} for X to overlap with any point with Z (Figure 10).

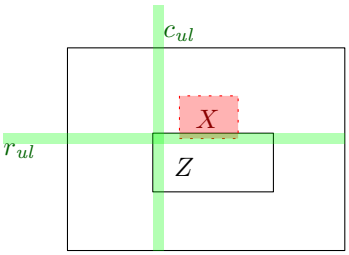


Figure 10: Fourth and final case of the case-distinction in the proof of Lemma 3: $\mathrm{UL}(X).r \leq r_{ul} \wedge \mathrm{UL}(X).c > c_{ul}$ So X must overlap in the row r_{ul} (in green).

These cases cover all possible consequences of Eq. (35), but all lead to a contradiction. So no such rectangle X can exist in M, and it follows that Z cannot overlap with any rectangle in M. □

5.4.5 Placing the first match of GetBoundingBox

There is one final but important aspect of GetBoundingBox (Algorithm 6. The constants on the upper-left corner (Eq. (33) and Eq. (34)) assign the coordinate 0 to sub-matches without upper or left neighbour. However, we do allow the compound match to have its upper-left corner in any point in a given source text $T \in \Sigma^{**}$. This is easily fixed by adapting MatchAllGroup: it should make an initial call to MatchAllGroupRec for *every* possible upper-left corner of the compound match, and simply gather all the different results. This feature is demonstrated in pseudocode in MatchAllGroupOptimised (Algorithm 7).

5.4.6 Alternative MatchAllGroup

Algorithm MatchAllGroupOptimised (Algorithm 7) and MatchAllGroupRecOptimised (Algorithm 8) implement the two optimizations in pseudocode. Note that these optimisations use a *matrix* $M \in ((\mathbb{N}\times\mathbb{N})^2 \cup \{\text{NIL}\})^{**}$ of previously-matched sub-matches, whereas the non-optimised MatchAllGroup (Algorithm 3) and MatchAllGroupRecBasic (Algorithm 4) used a *set* RECTS of sub-matches.

Both the checking for ill-defined upper-left corners (the first optimisation, Lemma 2) and the computation of the bounding boxes (the second optimisation) are mostly implemented in GetBoundingBox, whose pseudocode was presented earlier in Algorithm 6. However, bounding-boxes only restrict where a sub-pattern may search, they do not force the sub-pattern to actually contain the upper-left corner of the bounding box. For this reason, MatchAllGroupRecOptimised still needs to filter away the sub-matches that violate their upper-left corner constraints (Line 15).

As a final note, it seems possible that no bounding box is available for the current sub-pattern ρ, but that a composite match can still be formed. For example, the current sub-pattern might be the last sub-pattern that still needs a sub-match, and the previous sub-matches already form a rectangle (according to MakeRect). This is not problematic if ρ allows empty matches, which is quite possible when ρ contains a quantifier such as $\rightarrow^*$, $\rightarrow^?$, $\downarrow^*$ or $\downarrow^?$. See Line 9 of MatchAllGroupRecOptimised.

5.5 MakeRect Subroutine

The MakeRect algorithm was defined in §?? as *an algorithm that takes a set of sub-matches, and returns the large rectangle that encloses these sub-matches*. Recall from §5.4.1 that we represent this set of sub-matches corresponding to a matrix of pattern-expression $K \in \Lambda_\Sigma^{w\times h}$ as matrix $M \in ((\mathbb{N}\times\mathbb{N})^2 \cup \{\text{NIL}\})^{w\times h}$ of the same size as K, and that M's entries are entire rectangles representing sub-matches.

Algorithm 7:
MatchAllGroupOptimised $(T, K, \texttt{order})$

`Input:`

- $T \in \Sigma^{n_1 \times n_2}$: source text to find a match in.
- $K \in \Lambda_\Sigma^{h \times w}$: compiled search-pattern whose top-level pattern-expression is a group.
- $\texttt{order} : \mathbb{N} \times \mathbb{N} \to (\mathbb{N} \times \mathbb{N}) \cup \{\text{NIL}\}$: mapping giving the next index of the sub-pattern in K to match, given the previously matched subpattern (returns NIL when the last index is given as input).

`Output:` $\texttt{matches} \in 2^{(\mathbb{N}\times\mathbb{N})^2}$ set of rectangles of indices of T describing the matching subarrays of T, or $\emptyset$ when no match was found.

Let $n_1 \times n_2 \in \mathbb{N} \times \mathbb{N}$ be the size of T (width and height) ;
Let $h \times w \in \mathbb{N} \times \mathbb{N}$ be the size of K ;
$\texttt{matches} \leftarrow \emptyset$;
$M \leftarrow [\text{NIL}]^{h \times w}$ // A $w \times h$ matrix with NIL entries.
;
/* Try all possible top-left corners for the matching a rectangle. */
for $x = 0$ to $n_1 - 1$ do
    for $y = 0$ to $n_2 - 1$ do
        $new \leftarrow$ MatchAllGroupRecOptimised$(T[x : n_1, y : n_2], K, \texttt{order}, 0, M)$;
        /* Map the sub-match in of the sub-matrix $T[x : n_1, y : n_2]$ back to indices in the global T. */
        $new \leftarrow \{((x_1 - x, y_1 - y), (x_2 - x, y_2 - y)) \mid ((x_1, y_1), (x_2, y_2)) \in \texttt{matches}\}$;
        $\texttt{matches} \leftarrow \texttt{matches} \cup new$;
return matches ;

Algorithm 8:
MatchAllGroupRecOptimised $(T, K, \texttt{order}, i, M)$

Input:

- $T \in \Sigma^{n_1 \times n_2}$: source text to find a match in.
- $K \in \Lambda_\Sigma^{h \times w}$: compiled search-pattern whose top-level pattern-expression is a group.
- $\texttt{order} : \mathbb{N} \times \mathbb{N} \to (\mathbb{N} \times \mathbb{N}) \cup \{\text{NIL}\}$: mapping giving the next index of the sub-pattern in S to match, given the previously matched subpattern (returns NIL when the last index is given as input).
- $i \in \mathbb{N}$: index of next top-level pattern-expression in K to match, i.e., the $(i+1)^{th}$ sub-pattern in K according to `order`.
- M: $((\mathbb{N} \times \mathbb{N}) \cup \{\text{NIL}\})^{h \times w}$: a matrix of the same shape as K, mapping K's sub-patterns to sub-matches. The entries of M are $((x_1, y_1), (x_2, y_2)) \in (\mathbb{N} \times \mathbb{N})^2$ rectangles, or the placeholder value NIL. In the initial call, all entries should be NIL.

Output: `matches` $\in 2^{(\mathbb{N}\times\mathbb{N})^2}$ set of rectangles of indices of T describing the matches of K with T, or $\emptyset$ when no match was found.

```
1  matches ← ∅;
   /* Pick the next not-yet-matched sub-Pattern-Expression (PE),          */
   /* and try to match it with the unmatched space in T.                  */
2  PE ← order(K, i) ;
3  Let (i1, i2) be the index of PE in K ;
   // In implementation, order should return this tuple.
4  if PE = NIL then
5  |  return MakeRect(M) ;
   /* Find subset of indices of T in which PE is allowed to find a match. */
6  Let n1 × n2 ∈ ℕ × ℕ be the size of T (width and height) ;
7  ((xs, ys), (xe, ye)) ← GetBoundingBox(K, i, n1 − 1, n2 − 1, M);
8  if ((xs, ys), (xe, ye)) = NIL then
9  |  if PE allows an empty match then          // PE can, e.g., be →* or ↓?.
10 |  |  M[i1, i2] ← ∅ ;
11 |  else
12 |  |  return ∅ ;
13 else
   |  /* Match the sub-pattern-expression PE, using the correct MatchAll call (PE can
   |     be any type of pattern-expression).                               */
14 |  PE_matches ← PE.MatchAll(T[xs : xe, ys : ye], PE) ;
   |  /* That a sub-match is inside a bounding box does not imply that it contains the
   |     upper-left corner of the bounding box.                            */
15 |  Discard all sub-matches from PE_matches with ill-defined upper-left corners, according to
   |   Lemma 2 ;
16 |  for match m ∈ PE_matches do
17 |  |  M[i1, i2] ← m ;
18 |  |  r ← MatchAllGroupRec(T, K, order, i + 1, M);
   |  |  /* It is possible that r = ∅, but this does not lead to problems. */
19 |  |  matches ← matches ∪ r ;
20 return matches ;
```

Computing the enclosing rectangle is not difficult (see Algorithm 9), but checking if the smaller rectangles do not overlap and fit exactly in the larger rectangle is more complicated. Especially since some of the smaller rectangles are allowed to be empty, and since their alignment in the larger rectangle must correspond to their relative alignment in the matrix. This last constraint simply means that if a rectangle has a larger index in a given axis as some other rectangle, then in the larger rectangle it will have greater coordinates in that axis as well (for each axis for which this might be relevant).

The remaining of this section will first present a simple algorithm GetContainingRect for computing the large enclosing rectangle. Thereafter it will present a brute-force implementation for MatchRect, named MakeRectSetOps: this algorithm also checks if the matches indeed fill the rectangle computed by GetContainingRect correctly.

5.6 Algorithm GetContainingRect

Computing the containing rectangle is simple: simply find the minimum row and column indices of all the matches, and also the maximum row and column indices. Clearly, these four values give the smallest rectangle enclosing all the matches. This approach is described in pseudocode as GetContainingRect in Algorithm 9. It is important to recall that matches store absolute coordinates, i.e., indices of a source matrix $T \in \Sigma^{n_1 \times n_2}$. So a match typically does not have $(0, 0)$ as it upper-left corner point. In fact, it is quite possible none of the matches contains this point, as the larger enclosing rectangle does not need to cover all indices of T (it is a match itself, the match of the group containing the expressions in G).

To avoid any confusion, a rectangle R is said to *enclose* a smaller rectangle r if all points of $\mathbb{N} \times \mathbb{N}$ described by r are in the set of points described by R.

Algorithm 9:
GetContainingRect (M, h, w)

`Input:` $M : [0, h-1] \times [0, w-1] \to 2^{\mathbb{N}\times\mathbb{N}}$: a matrix mapping of indices in of a $h \times w$ matrix to matches. A match is a rectangle $((r_1, c_1), (r_2, c_2))$ identified by its upper-left and lower-right corner-points.

`Output:` A rectangle $((r_{min}, c_{min}), (r_{max}, c_{max})) \in 2^{\mathbb{N}\times\mathbb{N}}$ describing the smallest rectangle containing all the matches described by M.

```
1  r_min ← ∞ ;
2  c_min ← ∞ ;
3  r_max ← 0 ;
4  c_max ← 0 ;
5  for i = 0 to h − 1 do
6  |  for j = 0 to w − 1 do
7  |  |  ((r_1, c_1), (r_2, c_2)) ← M[i, j] ;
8  |  |  r_min ← min(r_1, r_min) ;
9  |  |  c_min ← min(c_1, c_min) ;
10 |  |  r_max ← max(r_2, r_max) ;
11 |  |  c_max ← max(c_2, c_max) ;
12 return ((r_min, c_min), (r_max, c_max))
```

The running-time of GetContainingRect (Algorithm 9) is clearly $\mathcal{O}(h \cdot w)$. Now to show the correctness of the algorithm, it is required to show that (1) the output encloses all the matches in M, and (2) there is no smaller rectangle that does enclose all the matches. This is done in Lemma 4 and 5 below.

Lemma 4. *Let $M : [0, h-1] \times [0, w-1] \to 2^{\mathbb{N}\times\mathbb{N}}$ be an $h \times w$ matrix of rectangular matches. Then $((r_{min}, c_{min}), (r_{max}, c_{max})) :=$ GetContainingRect(M, h, w) encloses all matches described by M (i.e., all points in $\mathbb{N} \times \mathbb{N}$ described by $\bigcup_{i,j \in [0,h-1]\times[0,w-1]} M[i, j]$).*

Proof. Assume for contradiction that there exist some $(i, j) \in [0, h-1] \times [0, w-1]$ such that $M[i, j]$ is not fully enclosed in $R = ((r_{min}, c_{min}), (r_{max}, c_{max})) := \text{GetContainingRect}(M, h, w)$. Let (k, ℓ) be one of the points described by $M[i, j]$ that is not in the rectangle described by R. Then a least one of the following cases must hold:

- $k > \text{MaxRow}(R)$. But then GetContainingRect would have assigned $r_{max} \leftarrow k$ in Line 10.
- $k < \text{MaxRow}(R)$. In this case GetContainingRect would have assigned $r_{min} \leftarrow k$ in Line 8.
- $\ell > \text{MaxCol}(R)$. Now, GetContainingRect would have assigned $c_{max} \leftarrow \ell$ in Line 11.
- $\ell > \text{MaxCol}(R)$. Here GetContainingRect would have assigned $c_{min} \leftarrow \ell$ in Line 9.

None of the cases is possible, so no such point (k, ℓ) exists. But then all points in the rectangle $M[i, j]$ are in the rectangle R. □

Lemma 5. *Let $M : [0, h-1] \times [0, w-1] \to 2^{\mathbb{N} \times \mathbb{N}}$ be an $h \times w$ matrix of rectangular matches. Then no rectangle that encloses all matches described by M (i.e., $\bigcup_{i,j \in [0,h-1] \times [0,w-1]} M[i, j]$) is smaller than the rectangle $((r_{min}, c_{min}), (r_{max}, c_{max})) := \text{GetContainingRect}(M, h, w)$.*

Proof. Let $R := ((r_{min}, c_{min}), (r_{max}, c_{max})) := \text{GetContainingRect}(M, h, w)$. By Lemma 4, R contains all points in all matches in the rectangles in image of M. Now assume that there exist a rectangle $Q = ((r'_1, c'_1), (r'_2, c'_2))$ that also contains these points, but that is smaller than R. Then one of the following cases must hold:

- $r'_1 > r_1$. But then, by Line 8 there exists a $(i, j) \in [0, h-1] \times [0, w-1]$ such that $\text{UL}(M[i, j]).r < r'_1$. This implies that Q does not enclose $\text{UL}(M[i, j])$, so Q would not contain all points described by M.
- $r'_2 < r_2$. Then, using Line 10, there exist a $(i, j) \in [0, h-1] \times [0, w-1]$ such that $\text{LR}(M[i, j]).r > r'_2$, hence the point $\text{LR}(M[i, j])$ would not be in Q.
- $c'_1 > c_1$. Again, now using Line 9, there would exist a point $(i, j) \in [0, h-1] \times [0, w-1]$ such that $\text{LR}(M[i, j]).c < c'_1$, so Q would not enclose $\text{LR}(M[i, j])$.
- $c'_2 < c_2$. The idea is yet the same: by Line 11, there would exist a $(i, j) \in [0, h-1] \times [0, w-1]$ such that $\text{LR}(M[i, j]).c > c'_2$, so Q would lack the required point $\text{LR}(M[i, j])$.

All cases lead to a contradiction, so no such rectangle Q exists. Then it follows that R must be the smallest rectangle that encloses all points described by any rectangle in the image of M. □

5.7 Algorithm MakeRectSetOps

Our implementation of MakeRect uses, as its name implies, set operations. In particular, it first explicitly computes the set A all points in the enclosing rectangle R (which is given by GetContainingRect). Then it iterates over the matches, and simply "crosses off" each point contained in a match. If a point is crossed off twice, or a point remains in A that is not crossed off, then it returns nil. Otherwise it returns R, which it had already computed anyway.
This is described in more detail as MakeRectSetOps in Algorithm 10.

Algorithm 10:
MakeRectSetOps (M, h, w)

Input: $M : [0, h-1] \times [0, w-1] \to 2^{\mathbb{N}\times\mathbb{N}}$: a matrix mapping of indices in of a $h \times w$ matrix to matches. A match is a rectangle $((r_1, c_1), (r_2, c_2))$ identified by its upper-left and lower-right corner-points.

Output: A rectangle $((r_{min}, c_{min}), (r_{max}, c_{max})) \in 2^{\mathbb{N}\times\mathbb{N}}$ describing the smallest rectangle containing all the matches described by M, or NIL if no such rectangle exist that contains each point in the union of the points in the matches in M or if matches overlap.

```
1  R = ((r_min, c_min), (r_max, c_max)) ← GetContainingRect(M, h, w) ;
2  A ← {(r, c) : r_min ≤ r ≤ r_max ∧ c_min ≤ c ≤ c_max ∧ (r, c) ∈ ℕ²} ;
3  for i = 0 to h − 1 do
4  |  for j = 0 to w − 1 do
5  |  |  ((r_1, c_1), (r_2, c_2)) ← M[i, j] ;
6  |  |  B ← {(r, c) : r_1 ≤ r ≤ r_2 ∧ c_1 ≤ c ≤ c_2 ∧ (r, c) ∈ ℕ²} ;
   |  |  /* If two matches m_1, m_2 overlap, then m_1 would remove the point from A
   |  |     before the following check is performed on m_2:                    */
7  |  |  if |A ∩ B| ≠ |B| then
8  |  |  |  return NIL ;
   |  |  /* Each point of R is allowed to be in only one match.                */
9  |  |  A ← A \ B ;
   /* If some point in R was not contained in any match, M fails to cover R.   */
10 if A ≠ ∅ then
11 |  return NIL;
12 else
13 |  return ((r_min, c_min), (r_max, c_max)) ;
```

The correctness of MakeRectSetOps (Algorithm 10) is straightforward to check. The running time of is algorithm is more interesting: it is bounded by $\mathcal{O}(h \cdot w \cdot n_r \cdot n_c)$. For a brute-force algorithm, this is not particularly bad. The following lemma justifies this claimed bound:

Lemma 6. *The running time complexity of* MakeRectSetOps(M, h, w) *is in* $\mathcal{O}(h \cdot w \cdot n_r \cdot n_c)$*, where* $n_r \cdot n_c$ *is the size of the smallest rectangle containing all points in any match in* M *(i.e., the rectangle returned by* GetContainingRect*).*

Proof. We analyse MakeRectSetOps (Algorithm 10) line by line.

Line 1 and 2 visit each element of the $h \times w$ matrix M once, so take $\mathcal{O}(h \times w)$ time.

The double loop starting in Line 3 runs $h \times w$ times. Line 5 is a constant time look-up operation. Line 6 takes at most $\mathcal{O}(n_r \cdot n_c)$ time, since the match is contained in the $n_r \times n_c$ rectangle R. Set union of A and B (Line 7) can be done in $\mathcal{O}(\min(|A|, |B|))$ time. Since both $|A|$ and $|B|$ are bounded by $\mathcal{O}(n_r \cdot n_c)$, this operation is in $\mathcal{O}(n_r \cdot n_c)$ (see [6]). Removing the elements from B from A (Line 9) can also be done in $\mathcal{O}(|B|) \subseteq \mathcal{O}(n_r \cdot n_c)$ time (also here see [6]). Hence the loop is bounded by $\mathcal{O}(h \cdot w \cdot n_r \cdot n_c)$ time.

The remaining operations only take constant time, so it is clear that the loop starting in Line 3 with a bound of $\mathcal{O}(h \cdot w \cdot n_r \cdot n_c)$ is the bottleneck. □

It would seem possible to construct an algorithm that does not iterate over the $n_r \times n_c$ indices of the large enclosing rectangle R, but directly use the $h \times w$ high-level $((r_1, c_1), (r_2, c_2))$ entries in M. This might be a significant speed-up in case $h << n_r$ and/or $w << c_c$. Unfortunately, approaches to find such an algorithm failed.

5.7.1 Further Improvements Appear Difficult

At first sight, it may seem sufficient to iterate over the rectangles in the rows of M and verify that they succeed each other without overlap or gap, top to bottom for every column. And then repeat this left-to-right for the columns of M, for every row. If all rectangles had, say, the same width, then this approach would have worked and would be computationally efficient. Unfortunately, the rectangles can have very different shapes, and no clear row-column structure might be preserved in the large enclosing rectangle – even when their upper-left corners correspond to the grid-layout in M.

One reason such row-column structure disappears, is that sub-matches might be empty, and that another row's or column's entry can be used to fill up the gap in the row/column under consideration. Figure 11 illustrates this with a concrete example. It is not trivial to find which entry of M is used to fill the gap, unless one iterates over large subsets of grid-cells, which would bring the $n_r \times n_c$ term back into the runtime complexity.

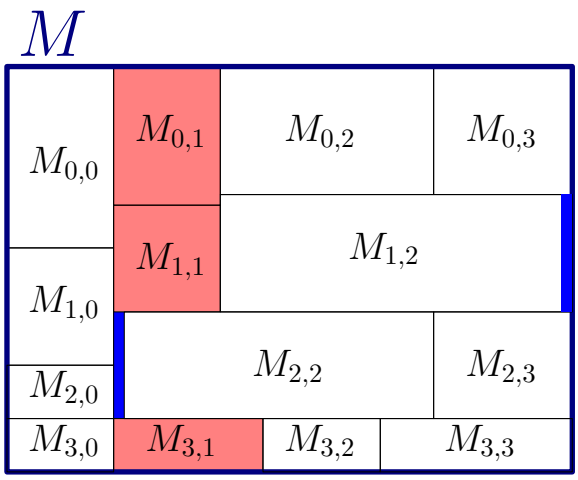


Figure 11: Example showing that the composite match belonging to the matrix of sub-matches M of a matrix of sub-patterns $K \in \Lambda_{\Sigma}^{w \times h}$ may not preserve a clear row-column structure, even when the sub-matches follow the alignment in rows and columns of the sub-patterns in K. In particular, column 1 (in red) contains an empty sub-match ($M_{1,2}$), which is filled by a rectangle from column 2 ($M_{2,2}$). Blue sub-matches indicate empty sub-matches.

6 2D Replacement

The sections above covered the semantics of searching and matching in 2D. However, this is only one half of the story of Search & Replace: the replacement still needs to be specified. Unfortunately, doing so in 2D gives rise to two new challenges:

1. The replace-pattern might have a different shape than the match. In 1D, one simply changes the length of the output text with respect to the input length. However, is not trivial to generalize to 2D, and we present only one out of multiple possible resize-semantics (see Section 6.1).
2. When replacing multiple occurrences ("ReplaceAll"), the resizing adds a new difficulty: after applying the first match, the positions of the other matches might have changed. It would seem simple to solve: compute greedily only the first match, then replace it, and repeat until no matches are available. Unfortunately, this quickly resolves in an endless loop when the search-pattern matches the replacement-pattern! This issue is further discussed in Section 6.2 below.

This section will first describe the concept of replacement with resizing the source-text (starting in Section 6.1). A simple but flawed algorithm for replacement follows (in Section 6.3). Finally it will present an improved algorithm for resizing (see Section 6.4).

6.1 Resizing in 2D

Replacing a match in a source-text $T \in \Sigma^{*\times *}$ with a replace-pattern $R \in (\Sigma \cup \{\square\})^{*\times *}$ is straightforward when R has the same shape as the match. However, it is possible that R is smaller or larger than the rectangle defined by the match: in this case, the edited text (which we will call $\Theta \in \Sigma^{*\times *}$) has a different shape than the input source text T.

This behaviour is not unlike in string Search & Replace. For example, assume input string $\vec{t} :=$ `Hello World!` and one replaces the search $\vec{s} := o(r)?$ with replacement $\vec{r} :=$ `XXX`, then the result will be `HellXXX WXXXld!`. This has a different length than the input.

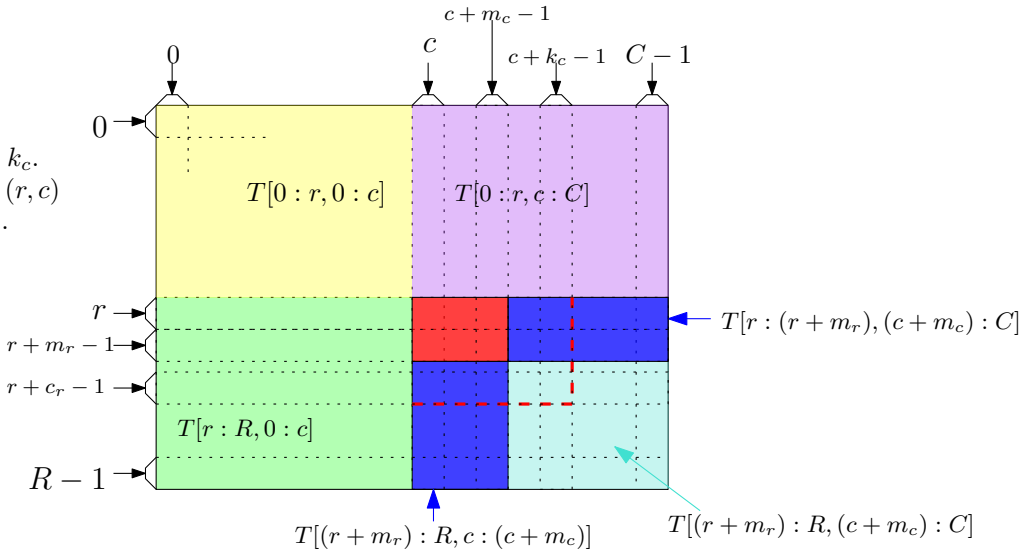
Unfortunately, in 2D the resizing becomes more complicated. One cannot simply make one row of source-text $T \in \Sigma^{n_1 \times n_2}$ longer than the other rows — all rows need to be moved, since the output must be matrix. Furthermore, this may cause empty spaces to occur. These can be filled with a special symbol, for which $\square$ would seem most appropriate.

6.1.1 Growing in 2D

The first case to consider is when the replacement-pattern is larger than the match: in this case, the output-text should be larger than the input source text (the input should "grow"). Figure 12 shows the chosen specification how a replacement algorithm should resize the source-text (when necessary). Old values are positioned in the resized output to preserve relative positions where possible — old values (not part of the match) that are aligned on the same row or the same column in T are still aligned in Θ on one axis.

Input source-text T:
R rows, C columns.
Indexing starts at 0.

Replacement pattern: $k_r \times k_c$.
Match of size $m_r \times m_c$ at (r, c)
to $(r + m_r - 1, c + m_c - 1)$.



Replaced text Θ:
$k_r - 1$ new rows,
$k_c - 1$ new columns.
White space is
padded with holes.

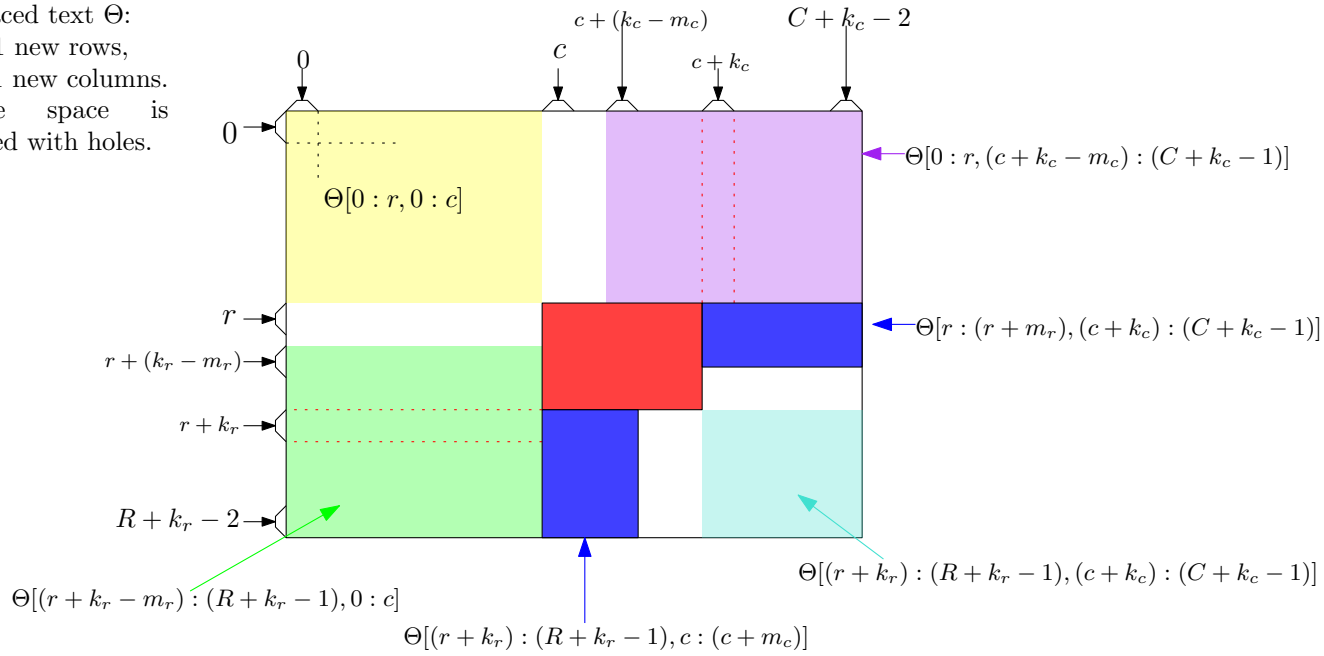


Figure 12: Generalized semantics of resizing. The upper figure shows the input text $T \in \Sigma^{*\times *}$, and the bottom the result $\Theta \in \Sigma^{*\times *}$. A search with a match of size $m_r \times m_c$, positioned at (r, c) in T (red rectangle in the upper image), is replaced by a $k_r \times k_c$ sized replace-pattern. Assuming $k_r > m_r$ and $k_c > m_c$, Θ may have a greater size than T. The image shows how old values of T that are not part of the match are positioned in Θ.

It must be stressed that the specification of resizing used in this report (Figure 12) is only one convention

out of many different possibilities. It seems to be an empirical question which resize semantics are the most practical approach. In fact, it is quite possible that for some application domains, growing the output is not practical altogether. For example, the dimensions of the output may be strictly fixed when generating textures. One could instead trim the replace-pattern to fit the match, or override source-text symbols outside of the match,

6.1.2 Shrinking in 2D

The second scenario is when the replacement-pattern is smaller than the match. Again, in one-dimensional strings the solution is obvious: make the string shorter. E.g. take source-text $\vec{t} :=$ `Hello world!`, search-pattern $\vec{s} :=$ `o` and the empty replace-pattern $\vec{r} := \varepsilon$. Then the expected output is simply `Hell wrld!`, which is shorter than the input.

As usual, things are more complicated in 2D. The replacement might be smaller than the match, but when located (for example) in the middle of the source-text, we cannot simply resize the output: non-matching values around the match would be discarded as well. Figure 13 shows an example.

One solution is to is pad the replacement pattern with additional □s such that it fits the match. Another approach is to first replace all the not-replaced entries of T, that are inside the match, with □s. This latter approach allows, in some cases, to resize the output: when it has rows and/or columns with only □s, these can safely be removed.

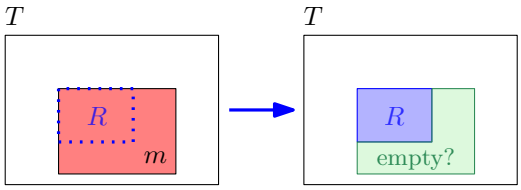


Figure 13: Simple visualization of replacement in a source-text $T \in \Sigma^{*\times *}$ where the match m is larger than the replace-pattern $R \in (\Sigma \cup \{\square\})^{*\times *}$. The left image shows the match m in red as located in T, and the blue dotted rectangle indicates the shape of R. The right image is the result of the replacement. The shape of T has not changed, but the inserted replace-pattern R does not cover the entire match. The remaining values are indicated in green (the polygon labelled "*empty?*"), and can either be left unchanged or substituted with □s.

6.1.3 Mixed cases

Finally, it is possible that the replacement-pattern is larger than the match in one dimension, but smaller in the other (say y). Without loss of generality, assume that the replacement-pattern R has more rows that the match, but fewer columns. This raises no new problems: combining the ideas above, we append columns of □s to R, and resize the edited source-text's amount of rows according to Figure 12.

6.2 Replacement of multiple matches

When one invokes ReplaceAll, one expects all non-overlapping matches to be replaced by the replacement-pattern, and no further changes. In particular, if the replacements would generate new matches, these are ignored.

For a concrete 1D example, consider the input string

$$\vec{t} := \texttt{This is a string!}.$$

Now when applying the Search & Replace $\vec{s}$:= `i`, $\vec{r}$:= `$i$`, one would expect the output:

`Th$i$s $i$s a str$i$ng!`.

Note a few important details:

- The inserted `i`s are not being replaced. If they were, infinitely many `$`s would be added.
- The indexes of the old matches shift as soon as one is applied, since the matches are 1 character, but the replacement-pattern spans 3 characters.

The last point shows that the algorithm is not to compute the positions of all matches first, and then replace them one by one. A better formulation instead would be the following:

1. The algorithm finds the leftmost match.
2. The algorithm substitutes this match with `$i$`.
3. This is repeated for the remainder of $\vec{t}$ that is to the right of the last inserted `$i$`.

This last algorithm can be generalized to 2D, although *to the right* must be replaced with *to the right or below*.

6.3 A basic Replacement Algorithm

The ideas above can be combined in a simple algorithm for performing 2D replacement. It is presented as ReplaceAll in Algorithm 11, with subroutine ReplaceAt in Algorithm 12.

The algorithm ReplaceAll (Algorithm 11) replaces matches one-by-one. Because the other matches may change after replacing the first match, it recomputes the set of matches after each replacement (Line 11). To avoid recursively searching and replacing in previous replacements (which may cause an endless loop), it "bounds off" the region in which replacement is still allowed. That is, it only allows replacements of matches whose row and column indices are both greater than any previously replaced cell (implemented by filtering the set of matches, see Line 12). The maximum row and column index of the replaced cells are the variables r and c, and there are updated after every replacement (Lines 9 and 10). The subroutine ReplaceAt (Algorithm 12) simply replaces a single match m in a source text T with a replacement pattern R, and resizes the output according to Figure 12 when needed.

Note that these algorithm have an additional argument, `no_resize`, that can be used to forbid resizing. When given, the replacement-pattern will simply be trimmed or padded with □s to fit the match exactly. This might be necessary in applications where the source-text has a very strict shape.

Another small detail is that ReplaceAll can introduce □s in the output (ReplaceAt explicitly allows □s in its input source-text T for compatibility). It is left to the user to handle such holes, or to design the Search & Replace such that they never occur. A simple solution (depending on the application) would simply be to declare □ ∈ Σ. One can, for example, use □ to represent empty space in a tile-map, or transparent pixels in a pixel-array.

6.3.1 Limitations of the basic ReplaceAll Algorithm

The algorithm ReplaceAll suffers from many flaws. The first is its inefficiently: it calls the computationally expensive function MatchAllGroup after every replacement, even when the replacement did not affect the correctness of previously computed matches. This occurs when no growth occurred, or if the other matches were only transposed.

Algorithm 11:
ReplaceAll $(T, S, R, \texttt{no\_resize})$

Input: $T \in \Sigma^{*\times *}$: source text to find a match in.
$S \in \Xi_{\Sigma}^{*\times *}$: search-pattern whose top-level pattern in an (implicit) group.
$R \in (\Sigma \cup \{\square\})^{*\times *}$: replacement-pattern.
$\texttt{no\_resize} \in \mathbb{B}$: flag that forbid resizing of the input. That is, if $\texttt{no\_resize} = \textsc{True}$, then the size of the output $\hat{T}$ is the size of T.
`order`: order for pattern-matching, see Algorithm 3.

Output: $\hat{T} \in (\Sigma \cup \{\square\})^{*\times *}$: copy of T, possibly resized, with a set of *non-conflicting* matches of S replaced by R.

$r \leftarrow 0$;
$c \leftarrow 0$;
$\hat{T} \leftarrow T$;
`matches` $\leftarrow$ MatchAllGroup$(\hat{T}, S, \texttt{order})$;
`/* Returns a set of rectangles identified by the upper-left and lower-right corners:` $((r_1, c_1), (r_2, c_2))$ `*/`
`matches` $\leftarrow \{m \in \texttt{matches} \mid m.r_1 > r \wedge m.c_1 > c\}$;
`while` $|\textit{matches}| > 0$ `do`
  $m \leftarrow \underset{m \in \texttt{matches}}{\arg\min} \{\min(m.r_1 - r, m.c_1 - c)\}$;
  $\hat{T}, (r', c') \leftarrow$ ReplaceAt$(\hat{T}, R, m, \texttt{no_resize})$;
  $r \leftarrow \max(r, r')$;
  $c \leftarrow \max(c, c')$;
  `matches` $\leftarrow$ MatchAllGroup$(\hat{T}, S, \texttt{order})$;
  `matches` $\leftarrow \{m \in \texttt{matches} \mid m.r_1 > r \wedge m.c_1 > c\}$;
`return` $\hat{T}$;

Algorithm 12:
ReplaceAt $(T, R, m,$ `no_resize`$)$

`Input:` $T \in (\Sigma \cup \{\square\})^{*\times *}$: source text in which to replace match m with R.
$R \in (\Sigma \cup \{\square\})^{*\times *}$: replacement-pattern.
$m = ((r_1, c_1), (r_2, c_2)) \in (\mathbb{N} \times \mathbb{N})^2$: rectangle describing a match.
`no_resize` $\in \mathbb{B}$: flag that forbid resizing of the input. That is, if `no_resize` $=$ True, then the size of the output $\hat{T}$ is the size of T.

`Output:` $\hat{T} \in (\Sigma \cup \{\square\})^{*\times *}$: copy of T, possibly resized, with values described by the match m substituted with R.
$r', c' \in \mathbb{N}$: coordinates of the lower-right corner of the replacement in $\hat{T}$.

$\hat{R} \leftarrow R$;
`for` *index* (i, j) *in* $\hat{R}$ `do`
  `if` $\hat{R}[i, j] = \square \wedge (i + m.r_1, j + m.c_1)$ *is in* T `then`
    $\hat{R}[i, j] \leftarrow T[i + m.r_1, j + m.c_1]$;
$\hat{T} \leftarrow T$;
`if` ***no_resize*** `then`
  `if` $\hat{R}$ *has more rows/columns than* m `then`
    Remove rows/columns of $\hat{R}$ (highest indices first) until it is no more larger than m ;
  `if` $\hat{R}$ *has less rows/columns than* m `then`
    Append rows/columns of $\square$ to $\hat{R}$ until $\hat{R}$ is no more smaller than m ;
`else`
  Reshape $\hat{T}$ according to Figure 12 ;
$\hat{T}[m.r_1 : (m.r_2 + 1), m.c_1 : (m.c_2 + 1)] \leftarrow \hat{R}$;
`if` ***no_resize*** `then`
  Discard any rows/columns of $\hat{T}$ that only consist of $\square$s ;
`return` $\hat{T}$, $(m.r_2, m.c_2)$

A second shortcoming is that the bounding mechanism is too crude. It can forbid matches that would still be safe to replace, but fall inside the rectangular bounds dictated by r and c (see the pseudocode Algorithm 11). Figure 14 visualizes an example of this phenomenon.

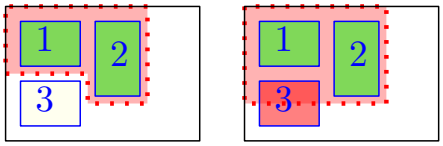


Figure 14: Example showing how a rectangular visited-region can result in the wrong results. The left image shows that, if matches 1 and 2 have been replaced (in green), and the visited-area is not rectangular, then match 3 can still be — correctly — replaced. However, when using rectangular visited-areas, as in the right image, then match 3 will not be replaced. Also note that, in this particular example, both approach would be correct if match 3 were replaced before match 2.

6.4 ReplaceAll Optimised

The basic version of ReplaceAll, Algorithm 11, calls MatchAllGroup (Algorithm 3) for *every* replacement. This is not strictly necessary, and since the MatchAllGroup algorithm is computationally expensive, it seems a worthwhile point to improve.

There were three key observations that played a part in designing the desired optimisation, which are:

1. The not-yet-replaced matches' indices shift if the replacement of a first match resulting in a growth of the source-text. But as discussed below, these shifts can be computed deterministically.
2. Apart from shifting, it is also possible that rows or columns of □s are inserted in some of the not-yet-replaced matches. This "splits the match in two halves", which we will call a *disturbance*. A match overlapping with another match is also said to be disturbed when the other is replaced.
3. The amount of potential disturbances caused by replacing a certain match (given a set of matches) can be computed *before* substituting the replace-pattern. This allows one to choose a strategic order to minimize the damage done.

These three points are discussed in more detail below.

6.4.1 Shifting of Remaining Matches on Growth

Growth of the source-text occurs a match $m = ((r_1, c_1), (r_2, c_2))$ of size $h_m \times w_m$ is replaced with a replace-pattern $R \in (\Sigma \cup \{□\})^{h_r \times w_r}$ such that $w_r > w_m$ or $h_m > h_r$ (see Section 6.1.1). The algorithm ReplaceAll obtains a set of matches from GroupMatchAll, and replaces them one-by-one; so it is possible that the remaining matches (other than m) are affected by the growth, while they still need to be replaced. in the following, we will assume that both $w_r > w_m$ and $h_m > h_r$, but it should be clear how the reasoning generalizes to cases where growth occurs in only one direction.

Let $T \in \Sigma^{**}$ be the source-text that is about to grow, and let the increase in dimensions be $\Delta x = w_r - w_m$ and $\Delta y = h_r - h_m$. The shift of location of any element $T[h_0, w_0]$ in T, due to the growth, can be tracked as follows:

- $\mathbf{w_0 < c_1}$ **and** $\mathbf{h_0 < r_1}$: The element remains in the same place (w_0, h_0).
- $\mathbf{w_0 > c_2}$ **and** $\mathbf{h_0 > r_2}$: The element shifts both left and right, moving to index $(w_0 + \Delta x, h_0 + \Delta y)$.

- $\mathbf{w_0 \geq c_1}$ **and** $\mathbf{h_0 \leq r_2}$: The element shifts right, moving to the new index $(w_0 + \Delta x, h_0)$.
- $\mathbf{w_0 \leq c_2}$ **and** $\mathbf{h_0 \geq r_1}$: The element shifts upward, and its coordinates after replacement will be $(w_0, h_0 + \Delta y)$.

These cases are visualized in the Figure 15. The algorithm ShiftMatches (Algorithm 13) describes the above index-shift computation as a pseudocode routine, for an entire set of matches.

Algorithm 13: ShiftMatches $(m, \texttt{matches}, \Delta r, \Delta h)$

`Input:`

- $m = ((r_1, c_1), (r_2, c_2)) \in (\mathbb{N} \times \mathbb{N})^2$: match whose indices in a source-text are to be replaced by a bigger replace-pattern.
- `matches` $\in 2^{(\mathbb{N}\times\mathbb{N})^2}$: set of matches whose new position needs to be computed if the source-text's elements described by m are replaced with growth. It is assumed none are rendered invalid by inserting a replacement with growth in the source-text match m.
- $\Delta r \in \mathbb{N}$: amount of rows by which the to-be-inserted replace-pattern is greater than m.
- $\Delta h \in \mathbb{N}$: amount of columns by which the to-be-inserted replace-pattern is greater than m.

`Output:` `out` $\in 2^{(\mathbb{N}\times\mathbb{N})^2}$: same matches as `matches`, but shifted in position such that they refer to the same elements of the source-text after replacing m.

```
1  out ← ∅;
2  for m̂ ∈ matches do
3      if m̂.c2 < mc1 and m̂.r2 < mr1 then
           /* Match m̂ is in the top left corner, and not shifted.                    */
4          out ← out ∪ {m̂}
5      else if m̂.c1 > mc2 and m̂.r1 > mr2 then
           /* Match m̂ is in the bottom right corner; it shifts to the left and right. */
6          m̂.c1 ← m̂.c1 + Δc
7          m̂.c2 ← m̂.c2 + Δc
8          m̂.r1 ← m̂.r1 + Δr
9          m̂.r2 ← m̂.r2 + Δr
10         out ← out ∪ {m̂}
11     else if m̂.c1 ≥ mc1 and m̂.r1 ≥ mr2 then
           /* Match m̂ is in the top right corner; it shifts only right.              */
12         m̂.c1 ← m̂.c1 + Δc
13         m̂.c2 ← m̂.c2 + Δc
14         out ← out ∪ {m̂}
15     else if m̂.c1 ≤ mc2 and m̂.r1 ≥ mr1 then
           /* Match m̂ is in the bottom left corner; it shifts only down.             */
16         m̂.r1 ← m̂.r1 + Δr
17         m̂.r2 ← m̂.r2 + Δr
18         out ← out ∪ {m̂}
19 return out ;
```

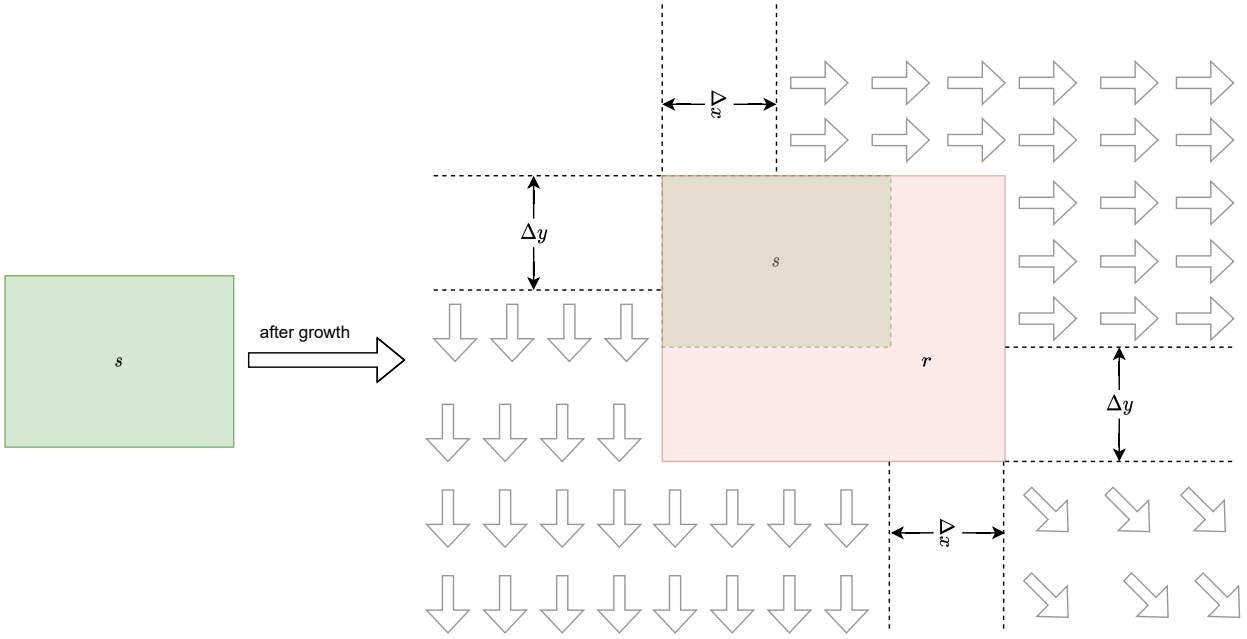


Figure 15: Visualization how indices of points in a source-text shift when the match s is being replaced by the replace-pattern r, causing a growth of Δx in the horizontal direction direction, and Δy in the vertical direction.

6.4.2 Disturbance of Remaining Matches

As explained above ReplaceAll replaces a first match $m = ((r_1, c_1), (r_2, c_2))$, then the remaining matches may not only shift, but may also become *disturbed* (see Figure 16 for an illustration). Disturbed matches cannot, in general, not be replaced any more.

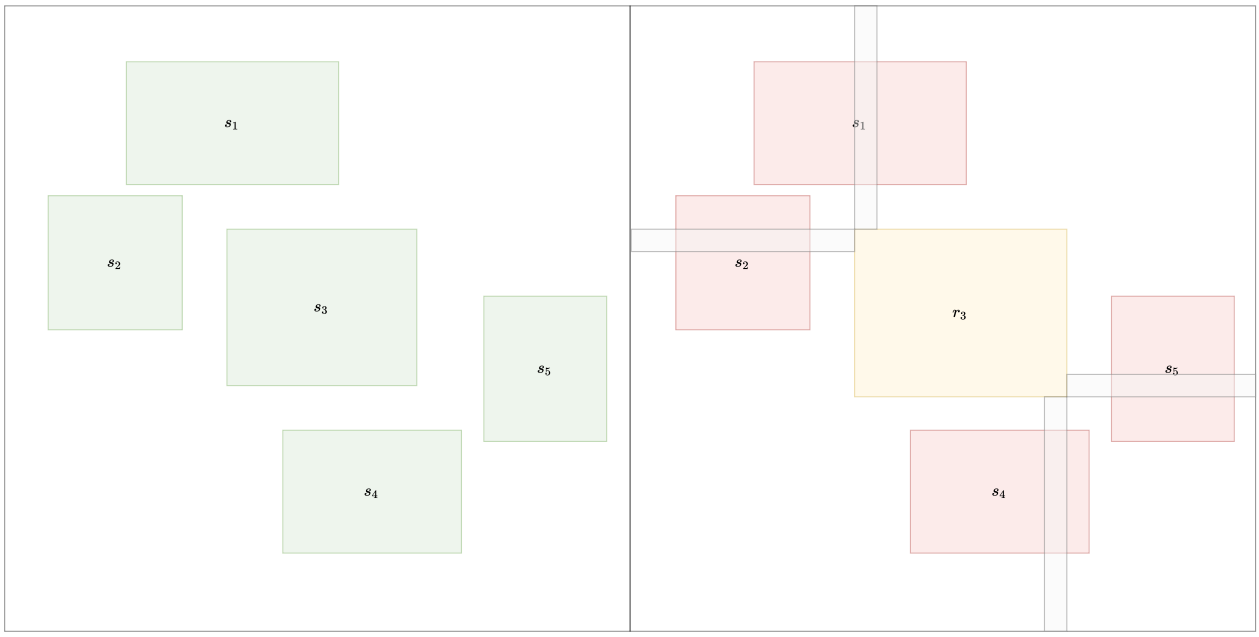


Figure 16: The scenario where there are five matches $s_1, \ldots, s_5$ to be replaced by ReplaceAll (left image). If replace-pattern r_3 were to be substituted in s_3 causing a growth in both directions, then s_1, s_2, s_4 and s_5 would be disturbed in inserted □s. Note that the values of the source-text of these remaining matches are not replaced, the □s are inserted in-between.

Like for the index-shifts, also disturbances can be computed, again as a case distinction. Let $m = ((r_1, c_1), (r_2, c_2))$ be the match that is being replaced by a larger replace-pattern. A remaining match $m' = ((r'_1, c'_1), (r'_2, c'_2))$ becomes disturbed if at least one of the following cases holds:

$$m' \text{ is above } m: \quad c'_1 < c_1 \wedge c'_2 > c_1 \wedge r'_2 < r_1 \tag{36}$$

$$m' \text{ is below } m: \quad c_1 < c_2 \wedge c'_2 > c_2 \wedge r'_1 > r_2 \tag{37}$$

$$m' \text{ is left of } m: \quad r'_1 < r_1 \wedge r'_2 > r_1 \wedge c'_2 < c_1 \tag{38}$$

$$m' \text{ is right of } m: \quad r'_1 < r_2 \wedge r'_2 > r_2 \wedge c'_1 > c_2 \tag{39}$$

$$m' \text{ and } m \text{ overlap}: \quad ((c_1 \leq c'_1 \leq c_2) \vee (c_1 \leq c'_2 \leq c_2)) \wedge ((r_1 \leq r'_1 \leq r_2) \vee (r_1 \leq r'_2 \leq r_2)) \tag{40}$$

Note that the above/below/left/right cases may also hold when m' actually overlaps with m. In case that m' overlaps with m (the last case), it will not be split by inserted □s, but it may be altered by the values directly substituted in m'. Depending on the replace-pattern, in many situations this would render m' invalid after replacing m. However, this is not the case if all the replaced elements, whose index is contained in m', do not change their value. Also, these elements may may be changed in such a way that the search-pattern would still find the match m.

Based these observations, it is possible to design an algorithm that computes the set of matches disturbed by replacing a given match m. The pseudocode is straightforward, and can be found as FindDisturbances in Algorithm 14.

Algorithm 14: FindDisturbances (`matches`, R)

`Input:`

- *matches* $\in 2^{(\mathbb{N}\times\mathbb{N})^2}$ set of all matches $((r_1, c_1), (r_2, c_2))$ returned by MatchAllGroup (Algorithm 3).
- $R \in (\Sigma \cup \{\square\})^{*\times *}$ is the replacement pattern to be inserted in the matches.

`Output:` D `matches` $\to 2^{\texttt{matches}}$: mapping of each match $m \in$ `matches` to the set of matches are disturbed when m is replaced first in the source-text.

```
Let D(m) ← ∅ for each m ∈ matches ;
for m ∈ matches do
    for m′ ∈ matches \ {m} do
        if replacing m with R disturbs m′ (at least one of Eq. (36)-(40) holds) then
            D(m) ← D(m) ∪ {m′} ;
return D
```

6.4.3 Improved `ReplaceAll`

Using FindDisturbances (Algorithm 14), ReplaceAll can be improved to make informed choices which matches to improve first. Ideally, all initial non-overlapping matches are replaced, since this corresponds best to the 1D string case. However, this is often not possible in 2D; the best one can do is to maximize the amount of initially-present matches that are successfully replaced. We propose two strategies:

- The cheapest is to replace the matches in ascending order in the amount of matches they disturb, and skipping disturbed matches. This is a fast but suboptimal heuristic. It is given as pseudocode ReplaceAllOptimised in Algorithm 15.
- One can simulate a complete recursive search to find the order that maximizes the amount of matches that are disturbed. Note that, by the asymmetry of the growth semantics (see Figure 12, the shifted columns below the match are different from the shifted columns above it, and idem for the rows in the other direction), the disturbance mapping may change after each replacement. The resulting computations of the entire search would form a large search-tree, whose depth and branching factor are proportional to the amount of matches. Also FindDisturbances needs to be called for every node in the tree, so this seems computationally inefficient.

Algorithm 15: ReplaceAllOptimised (T, S, R)

Input:

- $S \in \Xi_\Sigma^{*\times *}$: a search-pattern whose top-level pattern in an (implicit) group.
- $R \in (\Sigma \cup \{\square\})^{h_r \times w_r}$: a replace-pattern of size $h_r \times w_r$.
- $T \in \Sigma^{*\times *}$: the source-text in which as many initially present matches of S are to be replaced by R.

Output: $\hat{T} \in (\Sigma \cup \{\square\})^{*\times *}$: copy of T, possibly resized, with a set of *non-conflicting* matches of S replaced by R.

$\hat{t} \leftarrow T$;
matches $\leftarrow$ MatchAllGroup(T, S) ;
$D \leftarrow FindDisturbances(matches, R)$;
/* Sort in ascending order to reduce the number of matches that become invalid. */
Sort matches in ascending order, by amount of disturbed matches $|D(m)|$ each match m causes ;
/* Set for the matches that have already been replaced. */
completed $\leftarrow \emptyset$;
for $m \in$ *matches (in sorted order)* do
  $\hat{T}, (i, j) \leftarrow$ ReplaceAt$(T, R, m,$ False$)$;
  completed $\leftarrow$ completed $\cup \{m\}$;
  matches $\leftarrow$ matches $\setminus \{m\}$;
  matches $\leftarrow$ matches $\setminus D(m)$;
  $\Delta r \leftarrow \min(0, (m.r_2 - m.r_1) - h_r)$;
  $\Delta c \leftarrow \min(0, (m.c_2 - m.c_1) - w_r)$;
  matches $\leftarrow$ ShiftMatches$(m,$ matches$, \Delta c, \Delta r)$;
  $D \leftarrow FindDisturbances(matches, R)$;
  Resort matches, again according to $|D(\hat{m})|$ for each $\hat{m} \in$ matches ;
return $\hat{T}$;

7 Software Design

This section will briefly discuss the design ideas behind the implementation of the "Morg" 2D regex engine. The main topics are architectural choices; it was beyond the scope of this project to implement all required modules (although some parts have been implemented in Python).

7.1 Pattern-Expressions as a Class Hierarchy

The search-pattern syntax discussed in Section 5, with alphabet Ξ_Σ, provides a language for expressing abstract pattern-expressions in Λ_Σ. These pattern-expressions are abstract objects that encode a certain mini-language, and often make use of embedded pattern-expressions. Furthermore, they show similar behaviour (e.g., they offer a MatchAll method). For these reasons, it seemed most convenient to design pattern-expressions as a class hierarchy, and require all classes to implement the same interface named `PatternExpression`. Search-patterns can then be translated from a matrix in Ξ_Σ to an instance of a `PatternExpression` (which is a group pattern, containing the other sub-patterns embedded in it), a process we call *compilation*. Figure 17 shows a sketch of this class hierarchy as a UML class diagram.

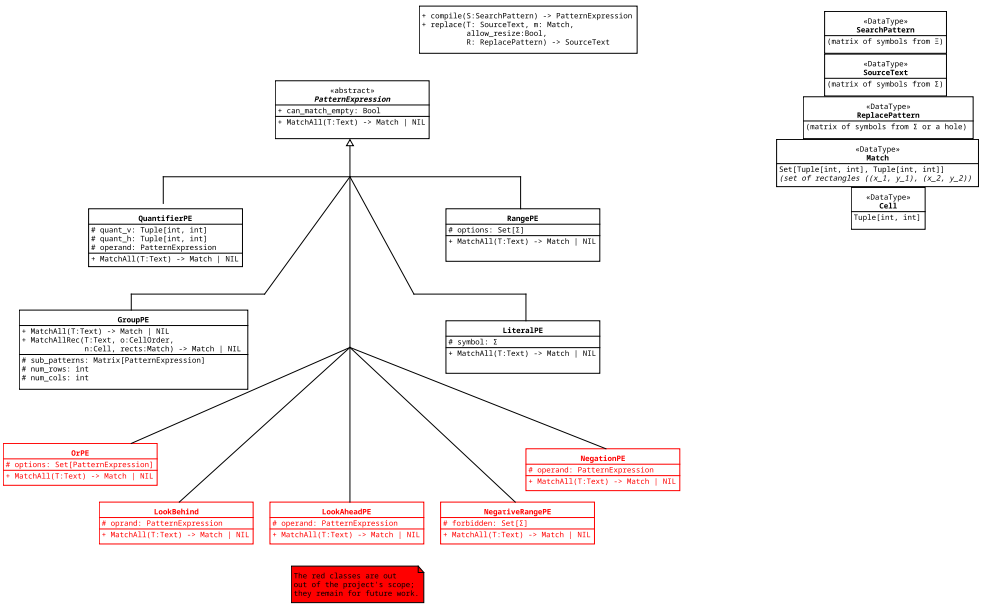


Figure 17: Sketch of the class hierarchy designed for the "Morg" framework, as a UML class diagram. There is a `PatternExpression` subclass for each type of pattern-expression as described in Section 5.2.2. Also pattern-expressions that remain for future work are shown in red.

7.2 Software Architecture

The "Morg"-framework is designed as a *general-purpose* 2D Search & Replace engine [3]. It is meant to be compatible with any desired end-application, whether this be a tiled game-level of a platformer, a texture of pixels, or a world-map for a strategy game. This means that the framework does not need to be familiar with the semantics of of the chosen symbol alphabet Σ (whose symbols could represent, for example, building blocks of levels, pixel-colours, map elements, etc.). Instead, Morg uses arrays of integers as source-texts and replace-texts ($\Sigma = \mathbb{N}$), which are more efficient in terms of computer resources than strings or abstract objects. This requires a simple bijective mapping between application-specific symbols ($\hat{\Sigma}$) and integers (Σ), but this is usually trivial to implement. We reserve the number 0 for the hole "□".

An advantage of this decoupling is that one can specify $\hat{\Sigma}$'s entries as human-readable strings, and that they can be compressed to small binary numbers (in most cases a byte per symbol would probably allow sufficiently many distinct symbols), requiring much less memory and being faster to operate on.

Figure 18 sketches the organization of the software. The "Application" can be any grid-structured data on which generalized Search & Replace is to be applied. This can a matrix of pixels (a digital image), a tile-map of a platformer game, a world-map of a strategy game, a 3D map of blocks for games such as Minecraft, etc. Each application might have a different internal representation for its grid. However, it seems safe to assume that they can always be converted back and forth to an N-dimensional array of strings in $\hat{\Sigma}$, denoted as `Array[string]` in the figure. The same holds for the replace-pattern

Furthermore, we assume that the search-pattern is also given as an array of strings (if the user specifies them as a line-separated text file, e.g., using ASCII encoding, then a small encoding convention of Ξ_Σ into ASCII is needed. This convention needs to be coded into the compiler. Note that it is also possible to design a GUI

[3] Our implementation, and therefore also this document, is restricted to 2D arrays, but the reader can hopefully see how the concepts can easily be generalized.

in which all elements of Ξ_Σ can directly be accessed without encoding to ASCII first.).

The procedure for search-patterns is a bit different. The strings need to be converted to numbers, using the same map as used for the source-text and the replace-pattern. However, strings representing symbols from Ξ_Σ should not be substituted, but interpreted ("compiled") to the correct data-structure: a `PatternExpression` object. A single search-pattern will result in one top-level `PatternExpression` object representing a group pattern-expression; this object will contain nested instances of `PatternExpression` objects.

This top-level `PatternExpression` object can then be given the numerical representations T of the source-text and R of the replace-pattern as inputs to apply the Search & Replace.

Finally, one obtains an output, an edited copy of T (a numerical array), to which the reverse string-to-integer mapping can be applied. This can then be passed back into the end-application.

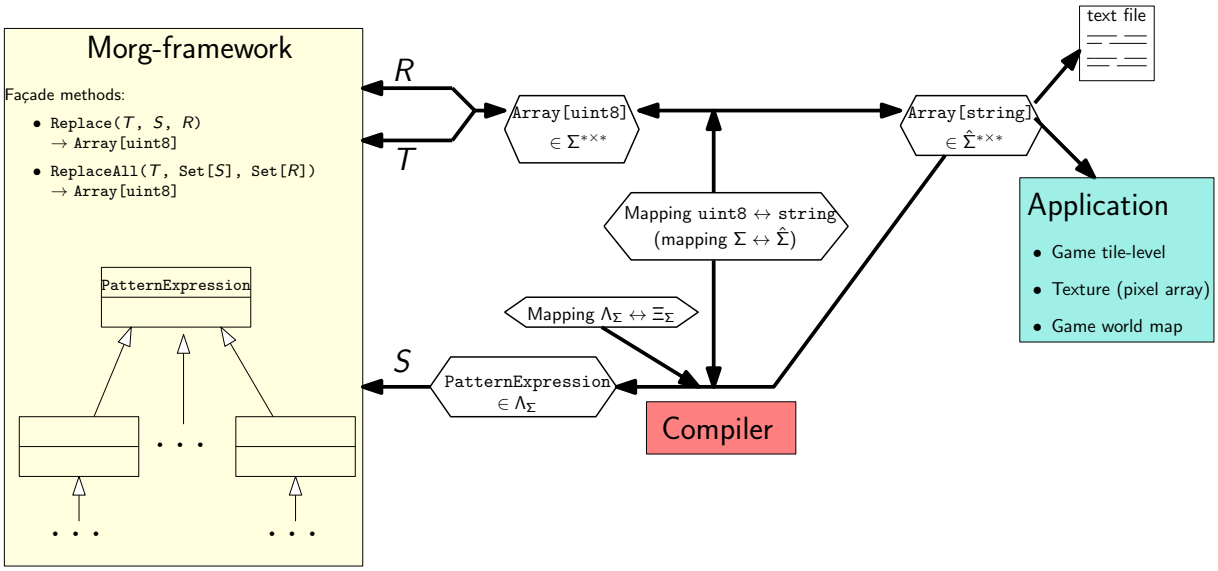


Figure 18: High-level organization of applying Search & Replace via the Morg-framework to an application. Thick arrows indicate conversions between data-representations. Source-texts are called "T", replace-patterns "R", and search-patterns "S". We assume that application-specific source-texts can be represented as strings (i.e., matrices in $\hat{\Sigma}^{*\times *}$). Matrices in $\hat{\Sigma}^{*\times *}$ are converted to matrices of integers before the Morg framework can work on them. Search-patterns need to be compiled from a text representation to a functional `PatternExpression` instance, as done by the "compiler". Note that Ξ_Σ depends on Σ, so the mapping $\Sigma \leftrightarrow \hat{\Sigma}$ is needed for the search-pattern compilation as well.

8 Discussion

This project was a first exploration into the possibilities of extending Search & Replace with regular expressions to multiple-dimensional inputs. As such, the current state is still far from a fully-fledged framework. However, this report identified many technical challenges in the design and implementation of a 2D regex

language, as well as many points where non-obvious design decisions need to be made. A careful reader may also observe a conflict of interests: on the one hand, the language should allow powerful expressions and be computationally efficient (which is particularly useful for automated use), but on the other hand, we also sought to make the language intuitive to use manual use. Design choices can focus on one of these two interests, or (as in this report), seek a trade-off.

The remaining of this section will explain which design decisions had to be made and the limitations of the work so far. Furthermore, also suggestions future work will be highlighted: some of which as necessary for a minimal functional framework, others are ideas for potential applications or extensions.

8.1 Design Decisions

During the formulation of the 2D regex language, several points where design decisions are needed were identified. The most important design choices are:

- The semantics of combining a collection of sub-matches $M \in ((\mathbb{N} \times \mathbb{N})^2 \cup \{\text{NIL}\})^{*\times *}$ of a group of pattern-expressions $K \in \Lambda_\Sigma^{*\times *}$. This report forbade ragged matches (Section 5.3.1), and anchored each sub-match $M[i,j]$ via its upper-left corner to a position in the compound match that corresponds to the relative alignment in K (Section 5.4.1).
- How to replace a matched sub-matrix of a source-text T with a replacement-pattern R that is *bigger* than the match (Section 6.1.1). This report chose to grow the dimensions of the source-text T, by shifting indices according to Figure 12 and adding □s into the new elements of T that are not part of R (as positioned in the output). The indices can also be shifted in different directions, but one may also choose not to resize the output in the first place. In that case, the decision remains whether the replacement-pattern must be trimmed, or whether it is allowed to override entries of T outside the match.
- How to replace a matched sub-matrix in a source-text T with a replacement-pattern R that is *smaller* than the match (Section 6.1.2). One can choose not to change the entries of T that are in the matched sub-matrix but outside the region where R is substituted in. Alternatively, these entries can be set to a special symbol such as □. With this last option, one may further wish to remove rows and/or columns of only □s from T, achieving a shrinking that is also observed in the 1D string case.
- The algorithm ReplaceAll may find a set of matches that cannot all be replaced without interference (e.g., replacing the first match may render the other invalid, which is especially possible if growth is allowed). In such cases, one needs to choose an order in which the matches are being replaced. One may greedily replace matches one-by-one, from left-to-right, top-to-bottom, but it is also possible to compute the order that maximized the amount of successful replacements. Finally, it is also possible to allow re-computation of the matches after every replacement (although care must be taken to avoid endless recursions). See Section 6.3.1 and Section 6.4.

There appears no theoretical optimum for any of these design choices. Rather, it seems an empirical question which choices are most practical.

Definition of Upper-Left Corner The definition of the upper-left corner (Eq. (33) and Eq. (34)) is not sufficiently general: in some cases, upper-left corners will be *ill-defined* while this is not necessary. For example, if we want the upper-left corner for some rectangle $M[i,j]$ in a matrix of rectangles $M \in ((\mathbb{N}\times\mathbb{N})\cup\{\text{NIL}\})^{*\times *}$, then $\text{UL}\,(M[i,j])$ would be ill-defined if $M[i-1,j] = \text{NIL}$ while $M[i-1,j+1]$ is not. According to Definition 2, $\text{UL}\,(M[i,j])$ is ill-defined as $\text{UL}\,(M[i,j])\,.r$ would overlap with $M[i-1,j+1]$. However, intuitively we would simply take $M[i-1,j+1]$ as up-neighbour of $M[i,j]$ instead of $M[i-1,j]$. See Figure 19 for a concrete visualization of this example (with $i=2$ and $j=1$).

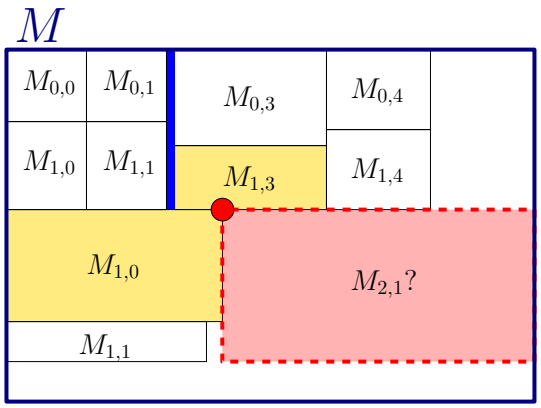


Figure 19: Example in which the provided definition of an upper-left corner (Eq. 33 and Eq. (34)) is too restricted. M is the composite match of sub-matches, corresponding to a matrix of pattern-expressions (not shown). The intuitively expected bounding box of $M[2,1]$ (here denoted as $M_{2,1}$) is drawn in red, with the upper-left corner as a red dot. $M_{0,2}$ and $M_{1,2}$ are empty matches. Intuitively, the neighbours of $M_{2,1}$ would be $M_{2,0}$ and $M_{1,3}$ (in yellow). However, Eq. 33 does not allow up-neighbours that have a different column index as $M_{2,1}$.

Other limitations Besides the limitations mentioned above, there are also the following:

- The provided group-matching algorithms are computationally inefficient. Both the basic version (Algorithm 3) as well as the optimised version (Algorithm 7) seem to have a significant computational complexity. For example, for an $n_1 \times n_2$-sized source-text $T \in \Sigma^{n_1 \times n_2}$, a pattern-expression may match *every* sub-matrix in T. As for another example, group pattern-expressions effectively compute the Cartesian product of the sub-matches of their sub-patterns, which may also quickly suffer from a combinatorial explosion. The optimisations in MatchAllGroupOptimised do not seem to significantly reduce the asymptotic runtime complexity (although both a formal analysis and empirical experiments of the optimisations remain open for future work).
- This report offers no literature study, due to time constraints.
- Algorithm MakeRectSetOps (Algorithm 10) has a poor runtime-complexity of $\mathcal{O}(h \cdot w \cdot n_r \cdot n_c)$ time. However, it still seems possible to avoid the $n_r \cdot n_c$ term with a more advanced algorithm.
- The order of replacements in the optimised version of ReplaceAll (ReplaceAllOptimised Algorithm 15) is a heuristic. One can find examples that this order does not always result in the maximum amount of replacements. A full recursive search (e.g., Depth-First Search or breadth-First Search) is needed to find the order that guarantees that the maximum amount of replacements is achieved.

8.2 Future Work

The following open tasks appear necessary for a practical and complete 2D regex engine:

- The design of MatchAll algorithms for quantifier pattern-expressions and choice pattern-expressions.
- Extending the language Λ_Σ with patterns for other pattern-expressions often found in string regular expression engined, such as negated groups, lookahead/look-behind groups, capturing groups (i.e., allowing part of the match to be programmatically inserted into the replace-pattern), etc. The implementation class diagram (Figure 17) already hinted at such extensions.

- Empirical assessment of the user-friendliness of the language definitions and semantics.
- Empirical assessment of the running time of the proposed algorithms. The asymptotic runtime complexity appears poor, but whether this is also an issue in practice remains an open question.

Aside from the necessary tasks, we also provide ideas for further investigations:

- Algorithm GetBoundingBox (Algorithm 6) is used to restrict the maximum size of a sub-match ahead of computing it, optimising the amount of computation done by MatchAllGroupRecOptimised (Algorithm 8). However, in some cases there appears to exist a minimum size of a sub-match as well, for the compound match to turn out as a well-formed rectangle. A formal investigation may reveal further optimisations to MatchAllGroup.
- Empirical experimentation using 2D Search & Replace for PGC creation. One could, for example, start by performing several rounds of Search & Replace on abstract symbols to lay out a general sketch (e.g., a coarse game level lay-out), followed by replacement with concrete symbols to add detail (e.g. the specific game tiles chosen).
- Learning a mapping from problem formulation to Search & Replace patterns. Perhaps it is possible to train a generative neural network that outputs Search & Replace patterns for PGC. In this case, one does not train a black-box PGC algorithm, but the algorithm that outputs (better interpretable) PGC algorithms.
- Stochastic elements in replace-patterns. This may be useful to add variety to Search & Replace for PGC applications.

9 Conclusion

This report explores the possibilities and challenges of generalizing Search & Replace with regular expressions to multiple-dimensional source-texts, especially in 2D. This exploration starts with a formal model for a subset of commonly found features in ordinary (1D) string Search & Replace engines. This model formulates source-texts and replace-patterns as a vectors in a symbol alphabet Σ, a search-patterns as vectors in another alphabet $\mathcal{R}_\Sigma$. A search-pattern $\vec{s}$ uses the symbols in $\mathcal{R}_\Sigma$ to encode a sequence $f_s(\vec{s}) \in \mathcal{L}_\Sigma$ of pattern-expressions, each of which encode a mini-language. A search-pattern $\vec{s}$ is said to match a sub-sequence $\vec{\tau}$of the source text $\vec{t}$ if there exist a vector in the concatenation of elements in the mini-languages of $\vec{s}$ that is equal to $\vec{\tau}$.

This model is then generalized to 2D by using matrices instead of vectors, and defining horizontal and vertical counter-parts for each pattern-expression in $\mathcal{L}_\Sigma$ (which results in a new language Λ_Σ). Finding matches no longer means concatenating vectors of symbols, but collecting small rectangles, as a puzzle, into a larger rectangle.

Both the task of finding matches and applying replacements become more involved than in 1D, as the addition of the additional axis allows for corner cases that are not possible in 1D. This report presents the recursive algorithm MatchAllGroup for finding all matches. It further suggests several optimizations for MatchAllGroup indented to reduce the computational cost of the algorithm. Similarly, the inefficient algorithm ReplaceAll is specified for applying both search and replace. Also this algorithm is further optimised to ReplaceAllOptimised.

The report also briefly discusses practical ideas for organizing the software implementation of a 2D regex engine.

Finally, this report discussed the design decisions that had to be made, the limitations of the current work, and outlines suggestions for future work.

Acknowledgements The majority of the figures in this report were created using IPE [1]. Features of the 1D Search & Replace engines discussed in this report were influenced by various applications used by the authors, especially Python [3] and Visual Studio Code [5].

References

[1] Otfried Cheong. *IPE*. Retrieved 5 June 2022. URL: http://ipe.otfried.org.

[2] Bruno Durand and Zsuzsanna Róka. "The Game of Life: Universality Revisted".
In: M. Delorme and J. Mazoyer. *Cellular Automata A Parallel Model*.
Springer Since+Business Media Dordrecht, 1999.

[3] *re - Regular expression operations*. Retrieved 9 March 2022. Python Software Foundation. 2022.
URL: https://docs.python.org/3/library/re.html.

[4] Michael Sipser. *Introduction to the Theory of Computation*. 3rd International Edition.
Cengage Learning, 2013. ISBN: 1-113-18781-1.

[5] *Visual Studio Code Tips and Tricks - Search and modify*. Retrieved 5 June 2022. Microsoft. 2022.
URL: https://code.visualstudio.com/docs/getstarted/tips-and-tricks#_search-and-modify.

[6] *wiki.python.org - TimeComplexity*. Accessed 15 May 2022. Python Software Foundation.
URL: https://wiki.python.org/moin/TimeComplexity.

A Turing Universality

This appendix briefly argues how 2D Search & Replace is a very general computational method: it is able to simulate a Turning machine. The argumentation is only on a high level, some technical details are left to the intuition of the reader.

Of course, a single replacement is not sufficient. We consider *multiple repeated* Search & Replace. This is defined as follows:

- *Multiple* Search & Replace concerts the use of a finite amount of pairs of a search- and replace-patterns $(S_1, R_1), \ldots, (S_z, R_z) \in \Xi_\Sigma^{*\times *} \times (\Sigma \cup \{\square\})^{*\times *}$ $(z \in \mathbb{N})$ on a single source-text $T \in \Sigma^{*\times *}$. In particular, for every pair, all matches of every search-pattern are computed, and all are concurrently replaced with the associated replace-pattern. This behaviour is undefined if any pair of matches overlap.
- *Repeated* Search & Replace involves applying a normal ReplaceAll with a search-pattern and a replace-pattern on a source-text as usual, obtaining output $\hat{T} \in \Sigma^{*\times *}$, but repeating the same ReplaceAll on this output text $\hat{T}$. This is repeated until the search-pattern finds no more matches.

When combined, we use multiple search-replace-pattern pairs on the same source-text, and repeat this on the output source-text, until none of the search-patterns finds a match.

Lemma 7. *The application of multiple repeated 2D Search & Replace on some (possibly infinite) source-text $T \in \Sigma^{*\times *}$, is Turing-Universal.*

Proof. Let $M = (Q, A, \Gamma, \delta, q_0, q_a, q_r)$ be a specific universal Turing machine (following the notation in [4][Definition 3.3, p168]), where:

- Q is M's finite set of states.
- A is M's input-alphabet.

- $\Gamma \supseteq A$ is M's tape-alphabet.
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\texttt{L}, \texttt{R}\}$ is M's transition function ($\texttt{L}$ and $\texttt{R}$ stand for a left and right head movement, respectively).
- $q_0, q_a, q_r \in Q$ are M's initial, accepting and rejecting state respectively.

We need to show that there exist a repeated multiple Search & Replace query that can simulate M. Hence, we need to specify:

- A finite integer $z \in \mathbb{N}$.
- A set of z search-patterns $\mathcal{S} = \{S_1, \ldots, S_z\}$ s.t. each $S_i \in \Xi_\Sigma^{*\times *}$.
- A set of z compiled replace-patterns $\mathcal{R} = \{R_1, \ldots, R_z\}$ s.t. $R_i \in (\Sigma \cup \{\square\})^{*\times *}$.
- A symbol alphabet Σ.

Three things need to be shown (following the example of [2]) for the simulation:

- An **encoding** of a Turing machine's tape, state and head-position into a source-text $T \in \Sigma^{*\times *}$.
- A **halting condition** of the repeated multiple Search & Replace that occurs if and only if the Turing machine halts on the input tape.
- A **decoding** of the resulting source-text $\hat{T}$ to a Turning machine's tape and state.

Encoding and Decoding the Tape Encoding and decoding the Turing machine's tape and state is straightforward. Let $\perp$ denote the empty-tape cell. Then we choose the source-text alphabet $\Sigma = A \cup Q$. We choose $T_0 \in \Sigma^{2\times\infty}$, and we set $T_0[0, 1:]$ to be the initial tape. Furthermore, we set $T_0[1, 1] = q_0$, and all elements of $T_0[1, 2:]$ to $\perp$. Finally, the first column, $T_0[:, 0]$ is set to $\begin{matrix}\perp\\\perp\end{matrix}$, which will serve as a padding for search-patterns.

To decode the resulting text and t steps of the Turing machine, we apply t repetitions of multiple Search & Replace, obtaining the output source-text $T_t \in \Sigma^{2\times\infty}$ (we will ensure it will not grow or shrink). We will design the Search & Replace such that $T_t[1, :]$ has a single entry that is not $\perp$. Let the index of this entry be $i + 1 \in \mathbb{N}$. Decoding is then simple:

- The output tape is given by $T_t[0, 1:]$.
- The state of the Turing machine $T_t[1, i]$.
- The position of the head is i (assuming we count tape cells starting form 0).

The Halting Condition The simulation halts when the state q_a or q_e appears in T_t, where T_t is the obtained source-text after t repetitions of multiple Search & Replace. If q_a occurred, then this is decoded as acceptance of the input tape by M, and similarly q_e is decoded as a rejection. We will ensure that no search-pattern matches after such a halt occurred. Of course, a halt should occur after t timesteps if and only if M halted after t timesteps on the corresponding tape.

The Search- and Replace-Patterns We simulate the Turning machine directly. $T_t[0, 1:]$ will correspond to the tape after t timesteps (of either t steps of the Turning machine or t repetitions of the multiple Search & Replace, they remain synchronized), and $T_t[1, 1:]$ will encode M's state and head position after t steps (by maintaining the invariant that $T_t[1, i+1] = q$ if M's head after t timesteps is at index t, in state $q \in Q$). This is done relatively straightforward: a search-pattern is created for every input pair (q, σ) of the transition-function δ, and the corresponding replace-pattern executes the symbol-writing on the tape by writing the symbol in $T_t[0, 1:]$, and executing the (state, head)-transition on $T[1, 1:]$. This means that we set $z = |\Gamma| \cdot |Q \setminus \{q_a, q_e\}|$, and that set of search- and replace-patterns required is then:

$$\left\{ (S, R) \ : \ q \in Q \setminus \{q_a, q_e\}, \sigma \in \Sigma, S = \begin{bmatrix} \cdot & \sigma & \cdot \\ \bot & q & \bot \end{bmatrix}, R = \left\{ \begin{array}{ll} \begin{bmatrix} \square & \sigma' & \square \\ q' & \bot & \square \end{bmatrix} & \text{if } \delta(q, \sigma) = (q', \sigma', \texttt{L}) \\ \\ \begin{bmatrix} \square & \sigma' & \square \\ \square & \bot & q' \end{bmatrix} & \text{if } \delta(q, \sigma) = (q', \sigma', \texttt{R}) \end{array} \right\} \right\}. \tag{41}$$

The correctness of the invariants can be shown inductively, but seems sufficiently straightforward that it is left to the intuition to the reader[4].

□

[4]This proof implicitly assumes that the Turing machine M never chooses to move left (L) while the head is already on the leftmost cell. For Turning machines this is typically defined to not result in any movement. The critical reader may have observed that such a move would result in a state written to $T_t[1, 0]$, which none of the provided search-patterns recognizes. However, the solution is straightforward: one can introduce a new symbol ⊛ and set $T_0[1, 0] = ⊛$. Then a special search-replace-pattern pair is introduced for each combination of a symbol $\sigma \in \Sigma$ and a state $q \in Q \setminus \{q_a, q_e\}$, defined as $S = \begin{bmatrix} \cdot & \sigma & \cdot \\ ⊛ & q & \bot \end{bmatrix}$ and $R = \left\{ \begin{array}{ll} \begin{bmatrix} \square & \sigma' & \square \\ ⊛ & q' & \square \end{bmatrix} & \text{if } \delta(q, \sigma) = (q', \sigma', \texttt{L}) \\ \\ \begin{bmatrix} \square & \sigma' & \square \\ ⊛ & \bot & q' \end{bmatrix} & \text{if } \delta(q, \sigma) = (q', \sigma', \texttt{R}) \end{array} \right.$. That is, R does, like the Turning machine, skip the left move to avoid running off the tape's left end.