Reconstructing General Matching Graphs

Amihood Amir
Department of Computer Science, Bar Ilan University, Ramat Gan 52900, Israel
Georgia Tech, College of Computing, Atlanta, GA, USA

Michael Itzhaki
Department of Computer Science, Bar Ilan University, Ramat Gan 52900, Israel

Abstract

The classical pattern matching paradigm is that of seeking occurrences of one string in another, where both strings are drawn from an alphabet set Σ. Motivated by many applications, algorithms were developed for pattern matching where the matching relation is not necessarily the "=" relation. Examples are pattern matching with "don't cares", approximate matching, less-than matching, Cartesian-tree matching, order preserving matching, parameterized matching, degenerate matching, function matching, and more. Some of the matchings above allow for efficient pattern matching algorithms, while others do not.

Much work has not been done on categorization of the complexity of various string matching queries based on the *type* of matching. For example, when can exact matching be done fast? When can approximate matching be calculated fast? When can tandem or palindrome recognition be efficiently calculated?

This paper defines the *matching graph* of a given string under a matching relation. We show that the type of graph affects various string algorithms. The matching graph can also be a tool for lower bounds. We provide a lower bound for finding palindromes in a general degenerate graph. We also show some results in recognizing the minimum alphabet required for reconstructing a string that presents a given matching graph.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases Pattern Matching, Matching Graphs, Reconstruction, $\mathcal{NP}$-hardness

Digital Object Identifier 10.4230/LIPIcs.CPM.2024.2

Funding *Amihood Amir*: Partially supported by ISF grant 168/23 and BSF grant 2018-141.
Michael Itzhaki: Partially supported by ISF grant 168/23.

1 Introduction

In the classical pattern matching model, we seek occurrences of a string, or more generally, a set of strings, in a distinguished string. All strings are comprised of symbols from an alphabet set Σ. The basic problem in this paradigm is that of *standard string matching*, that is, the problem of finding all occurrences of a pattern string of length m in a text string of length n. This problem can be solved in $O(n+m)$ time-independent of the alphabet size $|\Sigma|$ [15, 27, 43].

While the exact matching paradigm, where a *match* means alphabet equality, is a common and important one, historically, many problems were identified where a *match* between symbols has a different meaning. The first such model was the *pattern matching with don't cares*, where a special symbol $\phi \notin \Sigma$ is added, where ϕ matches every symbol in Σ. As we will see, this changes the matching graph, and the matching relation is no longer transitive. Fischer and Paterson [28, 34] showed that convolutions can solve this problem efficiently. Convolutions have been useful in the case of *less-than matching* [8]. Here the alphabet is natural numbers, and a pattern letter p matches the text letter t if $p \leq t$. The new twist in this matching relation is that it is not symmetric. *Approximate*

matching seeks all pattern occurrences with errors. There are several algorithms for Hamming distance errors, generally convolutions-based [1, 13], and dynamic programming algorithms for Levenshtein edit distance [44]. In *Cartesian Tree matching*, two strings match if they have the same Cartesian tree [46], and in *order-preserving matching*, two strings match if the relative order of their elements is the same [30]. *Parameterized Matching* was introduced by Brenda Baker [20]. In this matching, a text and pattern match if there is a bijection that, when applied to the pattern alphabet, will match the text. In *function matching*, the function applied to the pattern alphabet is a general function [14]. In *degenerate string matching*, the alphabet consists of non-empty subsets of alphabet Σ [26, 31, 35].

The various matchings mentioned above led to many different algorithms. The work of [19] classifies regex-matching problems by their structure. However, except for the latter, a systemic work on categorizing different matching types has not been performed. In this paper, we propose a method of analyzing some matching relations, called the *matching graph*. We give an example where a lower bound can be achieved due to the matching graph and study some of the insights that the matching graph offers.

▶ **Definition 1.** *Let* $\mathcal{M} \subseteq \Sigma \times \Sigma$ *be a* matching relation *of elements in alphabet* Σ. *Let* $S = S[1], ..., S[n]$ *be a string over* Σ. *The* Matching graph of S *is the graph* $G = (V, E)$, *whose nodes are* $V = \{1, ..., n\}$ *and where there is an edge* $\overline{ij}$ *if* $\mathcal{M}(i, j)$.

▶ **Example 2.** For alphabet $\Sigma = \{a, b, c\}$, the matching graph of string $S = a, a, b, a, b, b, c, c, b$ is the graph consisting of the three cliques $\{1, 2, 4\}, \{3, 5, 6, 9\}$ and $\{7, 8\}$.

▶ **Example 3.** Let $\Sigma = \{a, b\}$, and let ϕ be the don't care symbol. Then the matching graph of $S = a, a, b, \phi, b, a, \phi, a$ can be seen in Fig. 1.

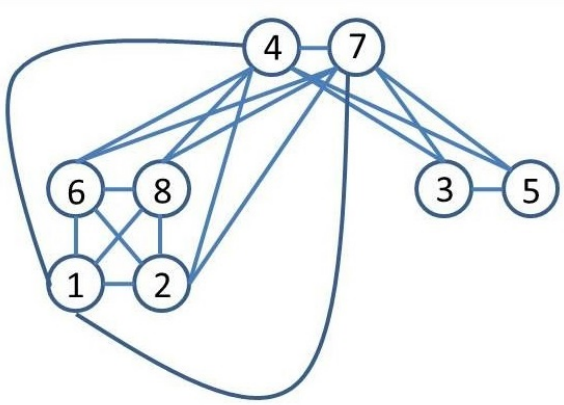


Figure 1 Matching Graph of a string with don't cares.

Note that the graph is always a set of cliques when the matching relation is transitive, as seen in Example 2. When the relation is symmetric, the graph is undirected; otherwise, it is not.

Our contribution

In this paper, we consider the matching relation in a generic string. We prove that any possible matching relation corresponds to a degenerate string, which implies lower bounds on most degenerate string matching algorithms. We also show that the number of characters required to reconstruct a degenerate string is tightly $O(n^2)$, as more characters introduce

no new information, and some matching relations cannot be reconstructed with less than $O(n^2)$ alphabet characters. We show that constructing a degenerate string from a matching relation with the smallest possible alphabet is NP-hard.

2 Preliminaries

We begin with basic definitions and notation, generally following [29].

Let $S = S[1]S[2]\dots S[n]$ be a *string* of length $|S| = n$ over an ordered alphabet Σ. By ε we denote the empty string. For two positions i and j on S, we denote by $S[i..j] = S[i]S[i+1]..S[j]$ the *factor* (sometimes called *substring*) of S that begins at position i and ends at position j (it equals ε if $j < i$). A *prefix* of S is a factor that begins at position 1 ($S[1..j]$), and a *suffix* is a factor that ends at position n ($S[i..n]$). We say that S^R is the reversal of S, which is $S[n]S[n-1]...S[1]$.

▶ **Definition 4** (Index Matching Function). *Let S be a string of length $|S| = n$. The $\mathcal{M}_S : [n] \times [n] \to \{0, 1\}$ is the matching function of string S and $\mathcal{M}_S(i, j) = 1$ iff $S[i] = S[j]$. The matching function will be denoted as $\mathcal{M}$ if S is clear from the context.*

The matching function of a standard string (as defined above) is transitive, reflexive, and symmetric.

▶ **Definition 5** (Palindrome). *A palindrome is a string S that equals its reversal S^R. Using definition Definition 4, a palindrome is a string that $\forall i, \mathcal{M}(i, n-i) = 1$. A palindrome factor is a string factor $P = S[i..j]$ such that P is a palindrome. A maximal palindromic factor is a palindromic factor $P = S[i..j]$ such that $S[i-1..j+1]$ is either not defined or not a palindrome.*

▶ **Definition 6** (Don't care). *The special character "don't care", denoted as ϕ is a character that matches any other character, including itself. A string S having a don't care at index m satisfies $\forall i < |S|, \mathcal{M}_S(m, i) = \mathcal{M}_S(i, m) = 1$.*

This paper addresses an interesting matching relation - equality in *degenerate strings*.

▶ **Definition 7** (Degenerate string). *Let Σ be an alphabet. S is called a degenerate string, if $S \in \{P(\Sigma)/\phi\}^*$, where $P(\Sigma)$ is the power set of Σ.*

The length *of the string, n, is the number of characters (sets) within that string. The* size *of the string, N, is $\sum_{i=1}^{n} |S[i]|$. We call the sets in S* terminals *and the characters in Σ,* elements. *The empty set can not be a terminal.*

▶ **Example 8.** Let $\Sigma = [5] = \{1, 2, 3, 4, 5\}$, and let $S_1 = \{1, 4\}\{1, 5\}\{4\}\{1, 2, 3\}$, and $S_2 = \{1\}\{2\}\{5\}\{5\}$. The lengths of S_1, S_2, denoted respectively by n_1, n_2 are both 4. However, the sizes, N_1, N_2 (resp.) are different, where $N_1 = 8$ and $N_2 = 4$.

▶ **Definition 9** (Primitive terminal). *Let c be a terminal of a degenerate string. We say that c is* primitive *if $|c| = 1$.*

▶ **Definition 10** (Terminals equality). *Let c_1, c_2 be two terminals of a degenerate string. We say that c_1* matches *c_2 if $c_1 \cap c_2 \neq \emptyset$. Throughout the paper, we denote terminals equality between $S[i]$ and $S[j]$ as $S[i] = S[j]$.*

▶ **Observation 11.** *Degenerate string matching where the only non-primitive terminal is Σ is equivalent to string matching with don't cares.*

▶ **Definition 12.** *Let $G = (V, E)$ be an undirected graph, meaning that $\forall i, j$ s.t. $(i, j) \in E \rightarrow (j, i) \in E$. An* induced subgraph, or simply a *subgraph $G' = (V', E')$ of G is formed from a subset of the vertices of the original graph, and all of the edges that connect vertices in the subset. Formally, $V' \subseteq V$, $\forall i, j$, $i \in V' \wedge j \in V'$ if and only if $(i, j) \in E'$.*
A clique is a subgraph $G' = (V', E')$ such that $\forall i \neq j \in V'$, $(i, j) \in E'$.

▶ **Definition 13.** *The complete graph K_n is a clique with n vertices. A bipartite graph $G = (V_1 \cup V_2, E)$ is a graph such that all vertices in E connect a vertex in V_1 with a vertex from V_2, formally, $\forall (i, j) \in E$, either $i \in V_1, j \in V_2$ or $i \in V_2, j \in V_1$. The complete bipartite graph $K_{n,m}$ is a bipartite graph $G = (V_1 \cup V_2, E)$ such that $|V_1| = n, |V_2| = m$ and E has all possible edges under the bipartite restriction. Bipartite graphs have no odd-length cycles, and therefore K_3 is not a subgraph of $K_{n,m}$, for any n and m.*

▶ **Definition 14** (Edge Clique Cover)**.** *Let $G = (V, E)$ be a graph. An* edge clique cover of G is a set of subgraphs of G, $\{(V_1, E_1), (V_2, E_2), ..., (V_m, E_m)\}$ such that $E = \bigcup_{i=1}^{m} E_i$. The *edge clique cover number is the size of the smallest possible set that covers G. Deciding if a graph can be covered with less than k cliques is NP-hard, and also hard to estimate.*

3 Matching function in Pattern Matching

The standard definition of a string defines a string as an ordered array of characters, i.e., $S \in \Sigma^*$. However, the alphabet is not crucial for most string algorithms and can be replaced by the numbers $1, 2, ..., |\Sigma|$. This possible replacement is because most algorithms only consider whether two characters are equal. This would not be true for algorithms considering a more complicated relation between the characters, for example, DNA algorithms that can predict a particular illness from a specific DNA subsequence or algorithms concerning the value of the characters, for example, ordered matching or Cartesian tree matching.

We call algorithms that only concern characters equality *alphabet comparison* algorithms. We may perceive the input to such algorithms as *matching oracle* $\mathcal{M}$ rather than a string S.

▶ **Definition 15.** *A* matching oracle $\mathcal{M}$ *is a function $\mathcal{M} : [n] \times [n] \rightarrow \{0, 1\}$, where $\forall i, j$ $M(i, j) = 1$ iff $S[i] = S[j]$.*

3.1 Matching Oracle Properties

While many algorithms claim to be comparison-based, most make additional assumptions about the matching function. The most common assumption is for the the function $\mathcal{M}(i, j)$ to define an equivalence relation, i.e.:

1. $M(i, j) = M(j, i)$ (symmetric)
2. $M(i, i) = 1$ (reflexive)
3. $M(i, j) = 1 \wedge M(j, k) = 1 \rightarrow M(i, k) = 1$ (transitive)

These assumptions work very well for standard equality. However, the last few decades have prompted the evolution of pattern matching from a combinatorial solution of the exact string matching problem to an area concerned with approximate matching of various relationships motivated by computational molecular biology, computer vision, and complex searches in digitized and distributed multimedia libraries [16, 32].

3.1.1 Parameterized strings

An important type of non-exact matching is the *parameterized matching* problem, which was introduced by Baker [21, 22]. Her main motivation lay in software maintenance, where program fragments are to be considered "identical" even if variable names are different. Therefore, strings under this model are comprised of symbols from two disjoint sets Σ and Π containing *fixed symbols* and *parameter symbols* respectively. In this paradigm, one seeks *parameterized occurrences*, i.e., exact occurrences up to renaming the pattern string parameter symbols in the respective text location. This renaming is a bijection $b : \Pi \to \Pi$. An optimal algorithm for exact parameterized matching appeared in [9]. It uses the KMP automaton for a linear-time solution over fixed finite alphabet Σ. Approximate parameterized matching was investigated in [17, 21, 37]. Idury and Schäffer [40] considered multiple matching of parameterized patterns.

Parameterized matching has proven useful in other contexts as well. An interesting problem is searching for images (e.g. [7, 18, 47]). Assume, for example, that we are seeking a given icon in any possible color map. If the colors were fixed, then this is an exact two-dimensional pattern matching [6]. However, if the color map is different , the exact matching algorithm will not find the pattern. A parameterized two-dimensional search is precisely the algorithm needed. If, in addition, one is also willing to lose resolution, then a two-dimensional function matching search should be used, where the renaming function is not necessarily a bijection [5, 14]. Another degenerate parameterized condition appears in DNA matching. Because of the base pair bonding, exchanging A with T and C with G, in both text and pattern, produces a match [38].

As defined, Parameterized matching is not an alphabet comparison matching. However, it has been shown to be equivalent to exact matching on a *prev* array, which is a transitive alphabet comparison matching. A *prev* is an array defined on a string S, where $A[i] = \max_{j<i}\{j \mid S[j] = S[i]\}$, or 0 if no such index exists [21].

3.1.2 Don't cares

Pattern Matching with don't cares is indeed an alphabet comparison matching. It can be defined via a matching oracle, where the don't care symbol ϕ satisfies $\forall i, j$ s.t. $S[i] = \phi$, $M(i, j) = 1$. However, this relation is not an equivalence relation. $\mathcal{M}$ is not transitive.

▶ **Example 16.** Let $S = a\phi b$. $M(1, 2) = 1, M(2, 3) = 1$ but $M(1, 3) = 0$.

However, the matching function is still quite structured even if the transitivity property is omitted. For example, let i, j, k, w be distinct indices such that $M(i, j) = M(j, k) = M(k, w) = 1$. In the don't care settings, we know that $M(i, k) = 1$ or $M(j, w) = 1$. This is true, because if $M(i, k) = 0$, then both $S[i], S[k] \neq \phi$, $S[i] \neq S[k]$, which means that $S[j] = \phi$ and therefore $M(j, w) = 1$. Pattern matching with don't cares has efficient solutions using convolutions [28, 34].

3.1.3 Less-than Matching

The matching with don't cares is an example of a non-transitive alphabet comparison matching relation. We do not know of any matching relation that is not reflexive.

Some non-symmetric alphabet comparison matching relations have been researched. Subset matching [12] and less-than matching [8]. The less-than matching problem is:

Input: Text string $T = T[1], ..., T[n]$ and pattern string $P = P[1], ..., P[m]$ where $T[i], P[i] \in N$ (the set of natural numbers).

Output: All locations i in T where $T[i + k] \geq P[k], \quad k = 1, ..., m$.

In words, every matched element of the pattern is not greater than the corresponding text element. If the text and pattern are drawn schematically, we are interested in all positions where the pattern lies below the text.

It turns out that convolutions could be used for efficient matching less-than matching. However, by giving up the symmetry requirement, we may ambiguate some basic strings' constructs - such as periods and palindromes. Therefore, in the remainder of this paper, we will only consider symmetric matching functions.

In the next section, we consider the *degenerate string matching problem*, which generalizes the don't care matching problem.

4 Degenerate string detection

Generalized degenerate strings and elastic degenerate strings are motivated by problems in Computational Biology. Much work has been done on efficient algorithms for matching as well as lower bounds [2–4, 23–26, 31, 35, 41]. In this section we will focus on detecting a *degenerate string* from a matching function. We will show that every symmetric matching graph has a corresponding degenerate string. We will also show that reconstructing a degenerate string from a matching function over a minimal alphabet is $\mathcal{NP}$-hard, and we will eventually show that for certain matching graphs, the degenerate string alphabet Σ is of quadratic order. The following two theorems will be proven in this section.

▶ **Definition 17.** *A degenerate string S is said to* reconstruct a matching graph G if the matching graph of S equals to G [1].

▶ **Theorem 18.** *Every symmetric matching graph has a corresponding degenerate string.*

▶ **Theorem 19.** *Recovering a degenerate string over minimal alphabet from a symmetric matching graph is $\mathcal{NP}$-hard.*

4.1 Matching graph representation

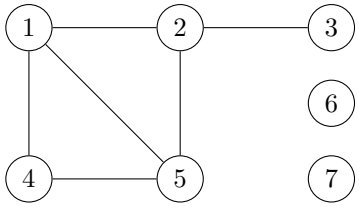
As we have seen, the matching function can be represented as a graph. One of the standard representations of a dense graph is by storing an adjacency matrix. Storing the list of neighbors for each vertices is more efficient if the graph is sparse. As we consider a symmetric function, the graph is undirected.

Because each symmetric matching function is equivalent to an undirected graph, we consider the problem of reconstructing a degenerate string from an arbitrary undirected graph. We begin by showing the existence of such a reconstruction. We later prove that finding a minimal alphabet is a hard problem.

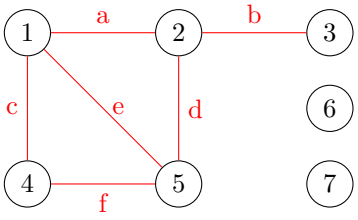
4.2 Reconstructing a degenerate string from an undirected graph

There has been much work recently on reverse engineering data structures [10, 11, 33, 36, 39, 42, 45]. Reverse engineering determines whether a given input is a valid instance of a particular data structure. We also refer to this as *reconstructing* the data structure. As we have seen, reconstructing the string from a matching graph for simple equality matching is simple. Each clique gives the indices of a unique symbol. Any graph that is not a collection of disjointed cliques is an illegal data structure. The matching graph of a degenerate string has the particular property that *every* undirected graph is legal. We now describe how, given an undirected graph G, we reconstruct a degenerate string whose matching graph is G.

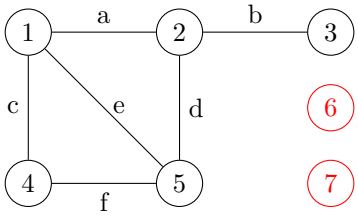
[1] Not isomorphic, as nodes have significance



(a) Matching graph, S is initialized $S = \{\}\{\}\{\}\{\}\{\}\{\}\{\}$.



(b) All edges are labeled with a character $S = \{a, c, e\}\{a, b, d\}\{b\}\{c, f\}\{d, e, f\}\{\}\{\}$.



(c) Empty sets are reconstructed $S = \{a, c, e\}\{a, b, d\}\{b\}\{c, f\}\{d, e, f\}\{g\}\{h\}$.

Figure 2 Reconstructing a degenerate string from the matching graph using Algorithm 1. We reconstruct with characters and not numbers to avoid confusion between edges and nodes.

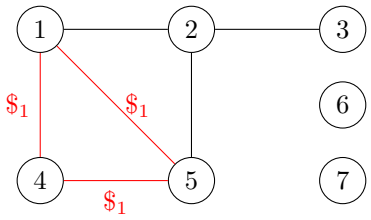
Let $G = (V, E)$ be an undirected graph, where $V = [n] = \{1, 2, ..., n\}$ represent the indices of the degenerate string, and an edge (i, j) exists if and only if the reconstructed string matches between indices i and j, denoted as $S[i] = S[j]$.

Algorithm 1 Reconstruct a degenerate string from an undirected graph.

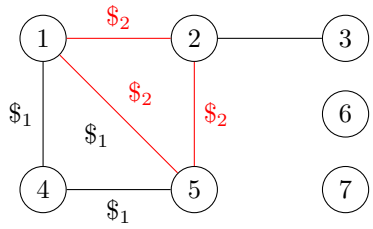
Data: Undirected graph $G = (V, E)$
Result: Degenerate string S
1 $S \leftarrow \{\{\}, \{\}, ...\{\}\}$ `// Initialize the output degenerate string with` $|V|$ `empty sets`
2 $c \leftarrow 1$ **for** $e = (i, j) \in E$ **do**
3 $S[i].add(c)$
4 $S[j].add(c)$
5 $c \leftarrow c + 1$
6 **for** *each empty s set in* S **do**
7 $s.add(c)$
8 $c \leftarrow c + 1$

The second `For` loop is necessary for the following reason. At the end of the first `For` loop, we may have empty sets for any node that does not match any other node, and empty sets are not allowed in degenerate strings. Thus, we finish the algorithm by adding a new character for every such node, which is the only symbol in that set and does not occur anywhere else. An example can be found at Figure 2.

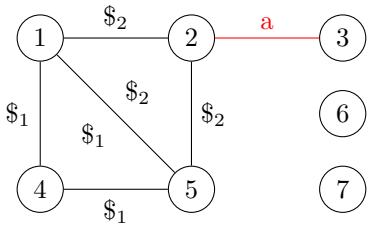
▶ **Lemma 20.** *Algorithm 1 reconstructs a degenerate string S with the same matching graph as the input $G = (V, E)$.*



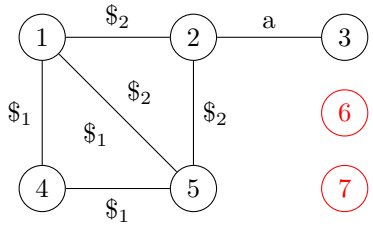
(a) First clique is colored
$S = \{\$_1\}\{\}\{\}\{\$_1\}\{\$_1\}\{\}\{\}$.



(b) Second clique is colored
$S = \{\$_1, \$_2\}\{\$_2\}\{\}\{\$_1\}\{\$_1, \$_2\}\{\}\{\}$.



(c) Third trivial clique is colored with a regular label, to distinguish it from non trivial cliques.
$S = \{\$_1, \$_2\}\{\$_2, a\}\{a\}\{\$_1\}\{\$_1, \$_2\}\{\}\{\}$.



(d) Empty sets are reconstructed
$S = \{\$_1, \$_2\}\{\$_2, a\}\{a\}\{\$_1\}\{\$_1, \$_2\}\{b\}\{c\}$.

Figure 3 Reconstructing a degenerate string from the matching graph of Figure 2, using cliques.

Proof. We show that $S[i] = S[j]$ iff $(i, j) \in E$.

We first prove that for every edge (i, j), the terminals $S[i]$ and $S[j]$ match. If (i, j) is an edge, then $S[i]$ and $S[j]$ both have the same character c, and therefore $S[i] \cap S[j] \neq \phi$, hence $S[i]$ matches $S[j]$.

We now prove that for every pair of indices i, j where $S[i]$ matches $S[j]$, the edge (i, j) exists. If $S[i]$ matches $S[j]$, it means that $S[i] \cap S[j] \neq \phi$. Let $c \in S[i] \cap S[j]$. c was either added to the algorithm in the first *For loop or the second. It is clear that if c was added in the first* For loop, then (i, j) is an edge, but in the second loop, all the characters added are unique, so it is impossible that both $S[i]$ and $S[j]$ have the same character that was added in that loop. ◀

The above reconstruction algorithm is simple and linear on the input size but does not produce a degenerate string with a minimal alphabet. Some graphs can be reconstructed to a degenerate string over an alphabet of constant size, while the algorithm will produce a quadratic size.

▶ **Example 21.** Consider the degenerate string $\{a\}^n$, of length n and of size $N = n$. The corresponding matching graph G is a clique of size n. However, after running Algorithm 1 on a clique of size n, the resulting degenerate string will be $\{\{1, 2, ..., n\}, \{1, n + 2, ...\}, \{2, n + 2, 2n + 3, ...\}, ...\}$, a string of length n but of size $N = n^2$.

Is there an efficient algorithm to reconstruct $G = (V, E)$ using a minimal alphabet? The answer is probably no, as we show that this problem is an $\mathcal{NP}$-hard problem.

▶ **Lemma 22.** *Let $G = (V, E)$ be a matching graph, and let $G' = (V', E')$ be an arbitrary sub-clique of G. Applying Algorithm 1 on $G'' = (V, E/E')$ and then adding a new character \$ for all vertices in V' is a valid degenerate string reconstruction.*

Proof. We will use Lemma 20 again. We need to prove that $\forall i,j$, $S[i] = S[j]$ iff $(i,j) \in E$. Lemma 20 shows that for every $(i,j) \in E/E'$, $S[i] = S[j]$. Also, for every $(i,j) \in E'$, we have $S[i] = S[j]$, as we required all of the characters participating in the clique to have a unique new character $\$_i$. We handle the other side similarly. We have a common character for every i,j where $S[i] = S[j]$. If the character is some $\$_i$, there must be an edge between $S[i]$ and $S[j]$, as they participate in the same clique. Otherwise, the terms proof in Lemma 20 holds, which completes the proof. ◀

▶ **Lemma 23.** *Let $G = (V,E)$ be a matching graph, and let $G' = (V',E')$ be a subgraph of G which is not a clique. If the reconstruction algorithm assigns the same character to all indices in V', then the resulting degenerate string does not have G as a matching graph.*

Proof. Let $G = (V,E)$, $G' = (V',E')$ be a graph and a subgraph as defined in the lemma. Let i,j be vertices such that $i,j \in V'$ but $(i,j) \notin E'$. Such a pair must exist, as a subgraph with only one vertex must be a clique, and a subgraph with more than one edge where all distinct vertices are connected is a clique.

If the algorithm assigns all indices i,j in the subgraph G' with the same character c, then $c \in S[i] \cap S[j]$, which means that $S[i] = S[j]$, but $(i,j) \notin E$. ◀

▶ **Observation 24.** *Lemma 22 can be applied iteratively to different cliques of G.*

An example of Algorithm 1 with clique coloring can be found at Figure 3.

▶ **Observation 25.** *Algorithm 1 applies Lemma 22 iteratively to all cliques of size 2, i.e., cliques having exactly two nodes and one edge.*

▶ **Observation 26.** *Let $G = (V,E)$ be a matching graph, and let $e \in E$. Every algorithm reconstructing a degenerate string from G will output a different string to $G = (V,E)$ and $G' = (V, E/\{e\})$.*

▶ **Lemma 27.** *Given a graph $G = (V,E)$ and a degenerate string S that reconstructs it, the string S defines an Edge-Clique-Cover for G.*

Proof. Let Σ be the alphabet of S. For every character $\sigma \in \Sigma$, all string-indices $i_1, i_2, ..., i_k$ whose terminals $S[i_j]$ contain σ are connected in the matching graph G (by the definition of reconstruction) and must form a clique (Lemma 23). Also, every edge $(i,j) \in E$ corresponds to at least one character in Σ (Observation 26), and therefore every character in Σ corresponds to a clique in G, where the vertices are all terminal indices containing σ [2]. ◀

▶ **Observation 28.** *Reconstructing a degenerate string from a matching graph with an alphabet of size k finds a Clique-Edge-Cover of size k to the matching graph, which is $\mathcal{NP}$-hard.*

Degenerate string equivalence

As seen at Observation 28, reconstructing a degenerate string from a matching function over a minimal alphabet is hard. However, reconstructing a degenerate string without limiting the resulting alphabet size is easy. We consider two different degenerate strings that have the same matching relation as *self-equivalent*. As shown in Example 21, every degenerate string can be rewritten as an equivalent string with at most $O(n^2)$ characters and a maximal terminal size of $n-1$.

[2] Some cliques can be sub-cliques of other cliques.

4.3 Constructing the matching function

We have defined the matching function and matching graph and will use it to prove some lower bounds. Before we proceed, we discuss the complexity of constructing the matching graph. We show that it is at least as hard as boolean matrix multiplication.

▶ **Lemma 29.** *Let S be a degenerate string of length n over an ordered alphabet $\Sigma = \{1, 2, ..., k\}$. Let $d(S[i])$ be the* indicator *of $S[i]$, i.e., a binary vector $w = d(S[i])$ where $w[i] = 1$ iff $i \in S[i]$, and let D be a matrix*

$$D = \begin{bmatrix} \mathbf{d}(S[1]) \\ \mathbf{d}(S[2]) \\ \vdots \\ \mathbf{d}(S[n]) \end{bmatrix}$$

The matching graph of S is $G = (\{1, 2, ..., n\}, E)$, where $E = \{(i, j) \mid (D \times D^T)_{i,j} = 1\}$.

Proof. The vertices of the matching graph are always defined as $[n]$. An edge (i, j) exists if and only if $S[i] = S[j]$. The element $(D \times D^T)_{i,j}$ equals to $\mathbf{d}(S[i]) \cdot \mathbf{d}(S[j])$, and the boolean inner product of binary vectors $\mathbf{v}, \mathbf{w}$ equals one if the vectors are orthogonal, and in our construction it means that $S[i] = S[j]$. ◀

▶ **Lemma 30.** *If finding the matching graph of a degenerate string S of length $2n$ and size $O(n^2)$ can be performed in time $f(n)$, then Boolean Matrix Multiplication can be computed in time $O(f(n))$.*

Proof. Let us denote by $G = (V, E)$ the matching graph constructed from S.

Let A, B be boolean matrices of size $n \times n$. We want to compute $C = A \times B$ in time $f(n)$.

Let S be a degenerate string of length $2n$ over numbers alphabet $[n] = \{1, 2, ..., n\}$. We rewrite:

$$A = \begin{bmatrix} \mathbf{v}_1 \\ \mathbf{v}_2 \\ \vdots \\ \mathbf{v}_n \end{bmatrix}, B = \begin{bmatrix} \mathbf{u}_1, \mathbf{u}_n, \ldots, \mathbf{u}_n \end{bmatrix}, C = A \times B = \begin{bmatrix} \mathbf{v}_1 \cdot \mathbf{u}_1, \mathbf{v}_1 \cdot \mathbf{u}_2, \ldots \mathbf{v}_1 \cdot \mathbf{u}_n \\ \mathbf{v}_2 \cdot \mathbf{u}_1, \mathbf{v}_2 \cdot \mathbf{u}_2, \ldots \mathbf{v}_2 \cdot \mathbf{u}_n \\ \vdots \\ \mathbf{v}_n \cdot \mathbf{u}_1, \mathbf{v}_n \cdot \mathbf{u}_2, \ldots \mathbf{v}_n \cdot \mathbf{u}_n \end{bmatrix}$$

We choose the terminals of S to be the following:

$$S[i] = \begin{cases} \mathbf{v}_i, & \text{if } i \leq n \\ \mathbf{u}_{i-n}, & \text{otherwise.} \end{cases}$$

Regarding only the elements of v, u and not their vector type. [3]

Let D be the matrix defined in Lemma 29. The edges of the matching graph of S are described by $D \times D^T$. Moreover, $(D \times D^T)_{i,n+j} = \mathbf{v}_i \cdot \mathbf{u}_j$, and therefore $C[i][j] = 1$ iff $(i, j + n) \in E$, hence, completing the proof. ◀

[3] In the definition of $S[i]$ the elements of A are row vectors and the elements of B are column vectors. However, in our definition of degenerate strings, row and column vectors have exactly the same meaning, therefore the direction can be altogether ignored.

5 Palindromes and degenerate strings

In the previous section, we discussed the degenerate string matching problem and showed that in a degenerate string S over an arbitrary alphabet, there are no restrictions on the edges of the matching graph G, whereas, in a regular string, there is a very rigid structure to the graph.

In this subsection, we show how the matching graph can be used to prove unconditional lower bounds for finding maximal palindromes in a degenerate string.

▶ **Lemma 31.** *Let S be an arbitrary degenerate string. A comparison-based algorithm A cannot find all longest palindromes of S using less than $O(n^2)$ time.*

Proof. Let us assume that we have a comparison-based algorithm A that can find all longest palindromes in a degenerate string S using less than $O(n^2)$ time.

Let S be a degenerate string of length n such that S has palindromes of length exactly $n/4$ in all centers that fit such a long palindrome. Also, let us assume that no other palindromes exist within S. Such a construction is achievable from Theorem 18. There are $O(n)$ palindromes of size $O(n)$, so comparing all indices within maximal palindromes takes $O(n^2)$ work. However, the algorithm does not perform $O(n^2)$ work, so there is a comparison $S[i], S[j]$ that lays within a maximal palindrome that is not checked, so a similar string $\tilde{S}$ where $\tilde{S}[i] \neq \tilde{S}[j]$, and otherwise is identical to S. Running A on $\tilde{S}$ will result in the same maximal palindromes array, but one of its palindromes is shorter. ◀

The above lemma, in effect, means that all edges in the matching graph must be examined to find the maximum palindrome. The reason is that there are no conditions on the edges of the graph, so one may not infer an edge by knowing other edges.

A conditional lower bound for finding all maximal palindromes from a degenerate string was given by [3]. Recall:

▶ **Theorem 32.** *Given a degenerate string of length $4n$ over an alphabet of size $\sigma = \omega(logn)$, all maximal GD palindromes cannot be computed in $O(n^{2-\epsilon} \cdot \sigma^O(1))$ time, for any $\epsilon > 0$, unless the Strong Exponential Time Hypothesis fails.*

The difference between M. Alzamel et al. theorem and ours, is that theirs shows a conditional lower bound on SETH, given a degenerate string. At the same time, we give an unconditional lower bound given a general matching graph. Of course, our proof relies on the fact that any general graph is a matching graph of some degenerate string. Our construction requires a quadratic size alphabet. For fixed-sized finite alphabets, the situation may be different. We are aware that, given a fixed finite alphabet, it is not hard to find algorithms that run in time $\tilde{O}(n^2)$ and find all maximal palindromes [4]. However, in that later case, the input size is not quadratic in n, but rather linear. The question is whether our lower bound applies in this case, i.e. can general graphs be matching graphs of degenerate strings over finite alphabets?

▶ **Observation 33.** *Given a matching graph $G = (V, E)$ of any degenerate string, all maximal palindromes can be found in time $O(n^2)$ by checking maximal palindrome around all possible centers.*

We show in Lemma 31 that $O(n^2)$ work is always required in the general case, and in Theorem 32 that $O(n^2)$ work is required under the SETH assumption. Given the matching graph, we also see in Observation 33 that exactly $O(n^2)$ is an upper bound. Therefore,

building the matching graph is at least as hard as finding all maximal palindromes on the general case and at least as hard as finding all maximal palindromes in degenerate strings with an alphabet of size $\omega(logn)$ under the SETH assumption.

We show that a quadratic number of characters is necessary to reconstruct a general matching graph.

▶ **Lemma 34.** *There exist matching graphs $G = (V, E)$ that cannot be reconstructed with less than an alphabet Σ of size less than $O(|V|^2)$.*

Proof. Consider the complete bipartite graph $K_{n,n}$. This graph is triangle-free and has a quadratic number of edges. As every character in the reconstructed degenerate string corresponds to a clique, and every clique has exactly one edge, there must be a quadratic number of cliques in the clique cover of the graph, hence a quadratic number of characters in any degenerate string S reconstructing G. ◀

The conclusion from all the above is that we have an unconditional lower bound for finding maximal palindromes in general graphs. The bound is the number of edges in the matching graph, $O(n^2)$. We also know that general matching graphs imply degenerate strings over alphabets of size $O(n^2)$. It may look like we have a tight algorithm, but this is not the case. Our algorithm has two stages:

1. Construct the matching graph $G = (V, E)$ from the degenerate string.
2. Use the matching graph to find all palindromes in time $O(|E|)$.

Indeed, one may construct the matching graph in linear time when the alphabet is finite, but then we are not sure that the matching graph is general, and therefore, the lower bound on finding the palindrome does not apply. Consider the following example:

▶ **Example 35.** Let S be a degenerate string over binary alphabet $\{a, b\}$. Every string element is either $\{a\}$, $\{b\}$ or $\{a, b\}$. Since $\{a, b\}$ matches both $\{a\}$ and $\{b\}$, the problem of finding palindromes in string S is equivalent to the problem of finding palindromes in a regular string over binary alphabets with *don't cares*. As was seen in Example 3, the matching graph in this case is well structured. Hence, there may not be a need to traverse all edges. We also know that pattern matching with don't cares has efficient solutions using convolutions. Accordingly, it may be the case that finding all palindromes in regular strings over binary alphabets with don't care has more efficient solutions than the quadratic.

In the case of alphabets of size $O(n^2)$ (quadratic alphabets), the lower bound applies, and we have an $O(n^2)$ time algorithm for finding palindromes that matches the lower bound, but that algorithm assumes a given matching graph. We have shown a conditional lower bound for constructing the matching graph of a degenerate string over a quadratic alphabet as bounded by the complexity of Boolean matrix multiplication, so our algorithm's time is now dependent on the time to construct the matching graph.

6 Conclusion and Open Problems

We have shown a simple data structure, the *Matching Graph*, that gives information on the matching relation of a pattern matching problem. We can infer from the graph whether a relation is transitive or symmetric. We also show that the graph may be useful for finding lower bounds, as in finding palindromes in degenerate strings.

Some very interesting open problems remain. An important one is finding an optimal algorithm for constructing the matching graph of degenerate strings. Such an algorithm will immediately imply an optimal algorithm for finding all palindromes in a degenerate string. This problem is especially relevant for small alphabets ($O(\log n)$), where no lower bounds are known.

Another intriguing problem is finding optimal algorithms for finding palindromes in degenerate strings over a fixed finite alphabet. A notorious example is finding all palindromes in a string over a binary alphabet, with *don't cares*.

Finally, given a degenerate string over a very large alphabet ($\Omega(n^2)$), we know that there is an equivalent degenerate string over an $O(n^2)$-size alphabet. We have shown that finding an equivalent degenerate string with the minimal alphabet is $\mathcal{NP}$-hard. However, it is easy to construct an equivalent degenerate string over a $O(n^2)$-size alphabet in time $N + n^2|\Sigma|$. Can it be done faster?

References

1 K. Abrahamson. Generalized string matching. *SIAM J. Comp.*, 16(6):1039–1051, 1987.

2 M. Alzamel, L. A. K. Ayad, G. Bernardini, R. Grossi, C. S. Iliopoulos, N. Pisanti, S. P. Pissis, and G. Rosone. Degenerate string comparison and applications. In *Proc. 18th International Workshop on Algorithms in Bioinformatics (WABI)*, volume 113 of *LIPIcs*, pages 21:1–21:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPICS.WABI.2018.21.

3 M. Alzamel, L. A. K. Ayad, G. Bernardini, R. Grossi, C. S. Iliopoulos, N. Pisanti, S. P. Pissis, and G. Rosone. Comparing degenerate strings. *Fundam. Informaticae*, 175(1-4):41–58, 2020. doi:10.3233/FI-2020-1947.

4 M. Alzamel, C. Hampson, C. S. Iliopoulos, Z. Lim, S. P. Pissis, D. Vlachakis, and S. Watts. Maximal degenerate palindromes with gaps and mismatches. *Theor. Comput. Sci.*, 978:114182, 2023. doi:10.1016/J.TCS.2023.114182.

5 A. Amir, A. Aumann, M. Lewenstein, and E. Porat. Function matching. *SIAM Journal on Computing*, 35(5):1007–1022, 2006.

6 A. Amir, G. Benson, and M. Farach. An alphabet independent approach to two dimensional pattern matching. *SIAM J. Comp.*, 23(2):313–323, 1994.

7 A. Amir, K. W. Church, and E. Dar. Separable attributes: a technique for solving the submatrices character count problem. In *Proc. 13th ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 400–401, 2002.

8 A. Amir and M. Farach. Efficient 2-dimensional approximate matching of half-rectangular figures. *Information and Computation*, 118(1):1–11, April 1995.

9 A. Amir, M. Farach, and S. Muthukrishnan. Alphabet dependence in parameterized matching. *Information Processing Letters*, 49:111–115, 1994.

10 A. Amir, E. Kondratovsky, G. M. Landau, S. Marcus, and D. Sokol. Reconstructing parameterized strings from parameterized suffix and LCP arrays. *Theor. Comput. Sci.*, 981:114230, 2024. doi:10.1016/J.TCS.2023.114230.

11 A. Amir, E. Kondratovsky, and A. Levy. On suffix tree detection. In *Proc. 30th Int. Symp. on String Processing and Information Retrieval (SPIRE)*, volume 14240 of *Lecture Notes in Computer Science*, pages 14–27. Springer, 2023. doi:10.1007/978-3-031-43980-3_2.

12 A. Amir, M. Lewenstein, and E. Porat. Approximate subset matching with "don't care"s. In *Proc. 12th ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 305–306, 2001.

13 A. Amir, M. Lewenstein, and E. Porat. Faster algorithms for string matching with k mismatches. *J. Algorithms*, 50(2):257–275, 2004.

14 A. Amir and I. Nor. Generalized function matching. *J. of Discrete Algorithms*, 5(3):514–523, 2007.

15 O. Amir, A. Amir, D. Sarne, and A. Fraenkel. On the practical power of automata in pattern matching. *SN Computer Science*, 2024. to appear.

16 A. Apostolico and Z. Galil (editors). *Pattern Matching Algorithms*. Oxford University Press, 1997.

17 A. Apostolico, M. Lewenstein, and P. Erdös. Parameterized matching with mismatches. *Journal of Discrete Algorithms*, 5(1):135–140, 2007.

18 G.P. Babu, B.M. Mehtre, and M.S. Kankanhalli. Color indexing for efficient image retrieval. *Multimedia Tools and Applications*, 1(4):327–348, November 1995.

19 Arturs Backurs and Piotr Indyk. Which regular expression patterns are hard to match? In *IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS 2016*, pages 457–466, December 2016. doi:10.1109/FOCS.2016.56.

20 B. S. Baker. A theory of parameterized pattern matching: algorithms and applications. In *Proc. 25th Annual ACM Symposium on the Theory of Computation*, pages 71–80, 1993.

21 B. S. Baker. Parameterized pattern matching: Algorithms and applications. *Journal of Computer and System Sciences*, 52(1):28–42, 1996.

22 B. S. Baker. Parameterized duplication in strings: Algorithms and an application to software maintenance. *SIAM Journal on Computing*, 26(5):1343–1362, 1997.

23 G. Bernardini, E. Gabory, S. P. Pissis, L. Stougie, M. Sweering, and V. Zuba. Elastic-degenerate string matching with 1 error. In *Proc. 15th Latin American symposium on Theoretical Informatics (LATIN)*, volume 13568 of *Lecture Notes in Computer Science*, pages 20–37. Springer, 2022. doi:10.1007/978-3-031-20624-5_2.

24 G. Bernardini, P. Gawrychowski, N. Pisanti, S. P. Pissis, and G. Rosone. Even faster elastic-degenerate string matching via fast matrix multiplication. In *Proc. 46th International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 132 of *LIPIcs*, pages 21:1–21:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICS.ICALP.2019.21.

25 G. Bernardini, P. Gawrychowski, N. Pisanti, S. P. Pissis, and G. Rosone. Elastic-degenerate string matching via fast matrix multiplication. *SIAM J. Comput.*, 51(3):549–576, 2022. doi:10.1137/20M1368033.

26 G. Bernardini, N. Pisanti, S. P. Pissis, and G. Rosone. Approximate pattern matching on elastic-degenerate text. *Theor. Comput. Sci.*, 812:109–122, 2020. doi:10.1016/J.TCS.2019.08.012.

27 R.S. Boyer and J.S. Moore. A fast string searching algorithm. *Comm. ACM*, 20:762–772, 1977.

28 P. Clifford and R. Clifford. Simple deterministic wildcard matching. *Information Processing Letters*, 101(2):53–54, 2007.

29 M. Crochemore, C. Hancart, and T. Lecroq. *Algorithms on Strings*. Cambridge University Press, 2007.

30 M. Crochemore, C. S. Iliopoulos, T. Kociumaka, M. Kubica, A. Langiu, S. P. Pissis, J. Radoszewski, W. Rytter, and T. Walen. Order-preserving indexing. *Theor. Comput. Sci.*, 638:122–135, 2016.

31 M. Crochemore, C. S. Iliopoulos, R. Kundu, M. Mohamed, and F. Vayani. Linear algorithm for conservative degenerate pattern matching. *Eng. Appl. Artif. Intell.*, 51:109–114, 2016. doi:10.1016/J.ENGAPPAI.2016.01.009.

32 M. Crochemore and W. Rytter. *Text Algorithms*. Oxford University Press, 1994.

33 J.P. Duval, T. Lecroq, and A. Lefebvre. Efficient validation and construction of border arrays and validation of string matching automata. *RAIRO Theor. Informatics Appl.*, 43(2):281–297, 2009.

34 M.J. Fischer and M.S. Paterson. String matching and other products. *Complexity of Computation, R.M. Karp (editor), SIAM-AMS Proceedings*, 7:113–125, 1974.

35 E. Gabory, N. M. Mwaniki, N. Pisanti, S. P. Pissis, J. Radoszewski, M. Sweering, and W. Zuba. Comparing elastic-degenerate strings: Algorithms, lower bounds, and applications. In *34th Symp. on Combinatorial Pattern Matching, CPM*, volume 259 of *LIPIcs*, pages 11:1–11:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICS.CPM.2023.11.

36 P. Gawrychowski, A. Jez, and L. Jez. Validating the knuth-morris-pratt failure function, fast and online. *Theory Comput. Syst.*, 54(2):337–372, 2014.

37 C. Hazay, M. Lewenstein, and D. Sokol. Approximate parameterized matching. In *Proc. 12th Annual European Symposium on Algorithms (ESA 2004)*, pages 414–425, 2004.

38 J. Holub, W. F. Smyth, and S. Wang. Fast pattern-matching on indeterminate strings. *J. Discrete Algorithms*, 6(1):37–50, 2008.

39 T. I, S. Inenaga, H. Bannai, and M. Takeda. Verifying and enumerating parameterized border arrays. *Theor. Comput. Sci.*, 412(50):6959–6981, 2011.

40 R.M. Idury and A.A Schäffer. Multiple matching of parameterized patterns. In *Proc. 5th Combinatorial Pattern Matching (CPM)*, volume 807 of *LNCS*, pages 226–239. Springer-Verlag, 1994.

41 C. S. Iliopoulos, R. Kundu, and S. P. Pissis. Efficient pattern matching in elastic-degenerate strings. *Inf. Comput.*, 279:104616, 2021. doi:10.1016/J.IC.2020.104616.

42 J. Kärkkäinen, M. Piatkowski, and S. J. Puglisi. String inference from longest-common-prefix array. In *Proc. 44th Intl. Coll. on Automata, Languages, and Programming, ICALP*, volume 80 of *LIPIcs*, pages 62:1–62:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

43 D.E. Knuth, J.H. Morris, and V.R. Pratt. Fast pattern matching in strings. *SIAM J. Comp.*, 6:323–350, 1977.

44 G.M. Landau and U. Vishkin. Efficient string matching in the presence of errors. *Proc. 26th IEEE FOCS*, pages 126–126, 1985.

45 Y. Nakashima, T. Okabe, T. I, S. Inenaga, H. Bannai, and M. Takeda. Inferring strings from lyndon factorization. *Theor. Comput. Sci.*, 689:147–156, 2017.

46 S.G. Park, M. Bataa, A. Amir, G.M. Landau, and K. Park. Finding patterns and periods in cartesian tree matching. *Theoretical Computer Sciencr*, 845:181–197, 2020.

47 M. Swain and D. Ballard. Color indexing. *International Journal of Computer Vision*, 7(1):11–32, 1991.