Theoretical Computer Science ••• (••••) •••–•••

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

On the average-case complexity of pattern matching with wildcards

Carl Barton

Birkbeck, University of London, Malet St, Bloomsbury, London WC1E 7HX, United Kingdom of Great Britain and Northern Ireland

ARTICLE INFO

Article history:
Received 20 January 2022
Received in revised form 5 April 2022
Accepted 10 April 2022
Available online xxxx
Communicated by R. Giancarlo

Keywords:
Average case complexity
Pattern matching with wildcards
Stringology
Pattern matching
Pattern matching with don't care symbols

ABSTRACT

Pattern matching with wildcards is a string matching problem with the goal of finding all factors of a text t of length n that match a pattern x of length m, where wildcards (characters that match everything) may be present. In this paper we present a number of complexity results and fast average-case algorithms for pattern matching where wildcards are allowed in the pattern, however, the results are easily adapted to the case where wildcards are allowed in the text as well. We analyse the *average-case* complexity of these algorithms and derive non-trivial time bounds. These are the first results on the average-case complexity of pattern matching with wildcards which provide a provable separation in time complexity between exact pattern matching and pattern matching with wildcards. We introduce the *wc-period* of a string which is the period of the binary mask x_b where $x_b[i] = a$ *iff* $x[i] \neq \phi$ and b otherwise. We denote the length of the wc-period of a string x by $\text{WCP}(x)$. We show the following results for constant $0 < \epsilon < 1$ and a pattern x of length m and g wildcards with $\text{WCP}(x) = p$ the prefix of length p contains g_p wildcards:

- If $\lim_{m\to\infty} \frac{g_p}{p} = 0$ there is an optimal algorithm running in $\mathcal{O}(\frac{n \log_\sigma m}{m})$-time on average.
- If $\lim_{m\to\infty} \frac{g_p}{p} = 1 - \epsilon$ there is an algorithm running in $\mathcal{O}(\frac{n \log_\sigma m \log_2 p}{m})$-time on average.
- If $\lim_{m\to\infty} \frac{g}{m} = \lim_{m\to\infty} 1 - f(m) = 1$ any algorithm takes at least $\Omega(\frac{n \log_\sigma m}{f(m)})$-time on average.

1. Introduction

In this paper we consider the pattern matching with wildcards problem. Pattern matching with wildcards is a string matching problem where the alphabet consists of standard characters and a *wildcard* character, ϕ, which matches every character in the alphabet. Given a text t of length n and a pattern x of length $m < n$, the problem consists of finding all factors of the text that match the pattern. In this section we will first cover some real world applications of pattern matching with wildcards, followed by a survey of the algorithmic results on this and related problems.

E-mail address: c.barton@bbk.ac.uk.

https://doi.org/10.1016/j.tcs.2022.04.009
0304-3975/

Please cite this article as: C. Barton, On the average-case complexity of pattern matching with wildcards, Theoretical Computer Science, https://doi.org/10.1016/j.tcs.2022.04.009

1.1. Real word applications

Wildcards are commonly used to model uncertainly in a variety of real worlds applications. Wildcards naturally arise in problems in bioinformatics, mainly in the analysis of data from DNA sequencing experiments where errors or biological phenomena need to be modelled. For example, DNA methylation is a biological process by which methyl groups are added to DNA that has been shown to be an important factor in the regulation of gene expression. Bisulphite sequencing experiments are a common way to measure levels of DNA methylation via next generation sequencing techniques and one of the primary strategies for unbiased alignments of reads produced from these experiments involves aligning reads which contain wildcard characters [2,21,19]. In DNA methylation analysis, wildcard matching is used to allow Ts in reads to match either C or T in the genome. This is an example of using wildcard characters to model single nucleotide polymorphisms (SNPs). Re-sequencing methods are also affected by SNPs that occur between individual samples and these modifications can be explicitly modelled for further downstream analysis as a wildcard.

We can see that to capture this phenomenon of uncertainty, a useful framework is to model uncertain positions as wildcard symbols. The algorithmic challenge is then to solve the string matching problem where in the given text or pattern some positions may be uncertain. Analysing uncertain sequences is therefore more complicated than the traditional pattern matching problem.

1.2. Previous algorithmic work

An early result on pattern matching with wildcards was the fast Fourier transform based algorithm [8] running in $\mathcal{O}(n \log m \log \sigma)$-time for an alphabet of size σ. A subsequent approach presented in [18] works by breaking the pattern into smaller pieces without wildcard characters and matches these using an Aho-Corasick automaton in $\mathcal{O}(n^2)$-time. Much subsequent work focused on removing the dependency on the alphabet size, with randomized $\mathcal{O}(n \log n)$-time and $\mathcal{O}(n \log m)$-time solutions being proposed in [12] and [13] respectively. Deterministic $\mathcal{O}(n \log m)$-time solutions were proposed soon after, initially in [6] and then a simplified version in [3]. If we allow for k mismatches, and k is small, then there is an algorithm that runs in $\Theta(n(k + \log m \log k) \log n)$-time presented in [4]. The only known lower bound for the worst-case of this problem is due to [17] who showed that in the worst-case the problem is equivalent to computing boolean convolutions. In the streaming model the problem can be solved for d wildcards using an algorithm presented in [7] with $\mathcal{O}(d + \log m)$-time worst-case complexity per character.

A significant amount of work has focused on the indexing version of the problem with many different succinct and non-succinct indexes being proposed. An early index was proposed in [11] where an index supporting queries in $\mathcal{O}(m + \alpha)$-time was presented with wildcards permitted in the pattern. This approach is based on a similar approach to that used in [18]. A short coming of these approaches is that in the worst-case $\alpha = \mathcal{O}(n^2)$. In [5] an index was presented which for a text with k wildcards and an integer d, allows searching for any pattern with at most d wildcards. For a pattern containing $g \leq d$ wildcards, the matching takes $\mathcal{O}(m + 2^g \log^d n \log \log n + occ)$-time, when wildcards are restricted to either the pattern or the text the query takes $\mathcal{O}(m + 2^g \log \log n + occ)$-time or $\mathcal{O}(m + \log^k n \log \log n + occ)$-time respectively. A drawback of the index of [5] is that once the index has been built it can only be used to search for patterns with at most d wildcards. In [1] a linear space index was presented with query time $\mathcal{O}(m + \sigma^g \log \log n + occ)$-time and a linear query time index with space complexity $\mathcal{O}(\sigma^{d^2} n \log^d \log n)$-time. The results of [1] can be further improved using recent work on weighted ancestor queries [9] and these results were further improved in [15]. In the area of succinct indexes solutions have been presented in [20] with a space usage of $((2+o(1))n \log \sigma + \mathcal{O}(n) + \mathcal{O}(h \log n) + \mathcal{O}(j \log j))$-bits for a text containing h groups of j wildcards in total. The authors of [10] proposed a compressed index where wildcards can only occur in the text with space usage $(nH_y + o(n \log \sigma) + \mathcal{O}(h \log n))$-bits.[1] The first non-trivial $o(n \log n)$ bit indexes were presented in [16].

1.3. Details of this paper

In this paper we focus our attention on the average-case complexity of pattern matching with wildcards. We note that when discussing the average-case complexity of online string matching problems it is customary to make a distinction between time taken *preprocessing* the pattern and the *search time*. Optimal average-case complexity customarily refers to achieving the optimal search time, not necessarily considering the preprocessing time required to achieve it. In this paper we focus only on the average-case search time as we wish to explore the minimum number of characters that need to be read on average to search in a text for a pattern with wildcards. In this paper we consider the following problem, where $\Sigma' = \Sigma \cup \{\phi\}$, where Σ is a finite alphabet.

Problem 1 *(Wildcards in the pattern).* Given a text t of length n drawn from Σ, and a pattern x of length m drawn from Σ'. Find all factors of the text t that match the pattern x.

[1] H_y is the y-th-order empirical entropy ($y = o(\log_\sigma n)$) of the text.

Our Contribution: In this article, we present average-case complexity results for pattern matching with wildcards for different wildcard ratios $\frac{g_p}{p}$. We show the following results for constants $0 < \epsilon < 1$ and a pattern x of length m and g wildcards with $\text{WCP}(x) = p$ and the prefix of length p contains g_p wildcards:

- If $\lim_{m\to\infty} \frac{g_p}{p} = 0$ there is an optimal algorithm running in $\mathcal{O}(\frac{n \log_\sigma m}{m})$-time on average.
- If $\lim_{m\to\infty} \frac{g_p}{p} = 1 - \epsilon$ there is an algorithm running in $\mathcal{O}(\frac{n \log_\sigma m \log_2 p}{m})$-time on average.
- If $\lim_{m\to\infty} \frac{g}{m} = 1 - f(m) = 1$ any algorithm takes at least $\Omega(\frac{n \log_\sigma m}{f(m)})$-time on average.

2. Preliminaries

An *alphabet* Σ is a finite non-empty set, of size σ, whose elements are called *characters*. A *string* on an alphabet Σ is a finite, possibly empty, sequence of elements of Σ. The zero-character sequence is called the *empty string*, and is denoted by ε. The *length* of a string x is defined as the length of the sequence associated with the string x, and is denoted by $|x|$. All strings of length q are denoted by Σ^q and we refer to any $x \in \Sigma^q$ as a q-gram. We denote by $x[i]$, for all $0 \le i < |x|$, the character at index i of x. By $x[i\,..\,j]$ we denote the string $x[i] \cdots x[j]$ called a *factor* of x (if $i > j$, then the factor is the empty string ε). Each index i, for all $0 \le i < |x|$, is a position in x when $x \neq \varepsilon$. It follows that the i-th character of x is the character at position $i - 1$ in x, and that

$$x = x[0\,..\,|x| - 1].$$

Equivalently, a string x is a *factor* of a string y if there exist two strings u and v, such that $y = uxv$. Consider the strings x, y, u, and v, such that $y = uxv$. If $u = \varepsilon$, then x is a *prefix* of y. If $v = \varepsilon$, then x is a *suffix* of y. A *wildcard* character, denoted by ϕ, is a special character that does not belong to Σ, and matches with itself as well as with any character of Σ. We denote this new alphabet $\Sigma' = \Sigma \cup \{\phi\}$. Two characters a and b of alphabet Σ' are said to *correspond* (denoted by $a \approx^\phi b$) if they are identical or at least one of them is a wildcard character. Similarly two factors correspond if and only if the strings correspond in every position.

Let x be a non-empty string and y be a string. We say that there exists an *occurrence* of x in y or, more simply, that x *occurs in* y when x is a factor of y. Every occurrence of x can be characterised by a position in y. Thus we say that x occurs at the *starting position* i in y when $y[i\,..\,i + |x| - 1] = x$. Clearly, before any characters in the text have been read, each position in the start is a possible starting position of the pattern. For each potential starting position in the text, we refer to the possible occurrence as a *candidate*. Table 1 shows the possible 8 starting positions and the associated candidates for the pattern $ab\phi\phi b\phi\phi a$.

When we read a character from the text at position i, we call this an *access* at position i. When we access position i in the text, any candidates which have non-wildcard characters aligned at this position are said to have a *c-intersection*. Conversely if the candidate has a wildcard character aligned at the access it is said to have a *w-intersection*. For example, looking at Table 1 an access at position 7 would intersect the candidates at starting position 0, 3, 6 and 7 as the occurrence starting at those positions has a non-wildcard aligned at position 7. We refer to a sequence of accesses on the text as an *inspection scheme*, denoted $\mathcal{I} = (i_0, i_1, \ldots, i_{2m-2})$ where i_j is the index of the j-th position accessed.

A string y is a *period* of a string x if $x = y^k y'$ where $k \ge 1$ and y' is a prefix of y. Let x be a string of length m over Σ' and let x_b be a binary mask of length m such that $x_b[i] = a$ *iff* $x[i] = \phi$ and $x_b[i] = b$ *iff* $x[i] \neq \phi$. Then x has a *wc-period* of length p if x_b has a period of length p denoted $\text{WCP}(x) = p$. Let x^i denote the i-th rotation of x, where $x^0 = x$ and $x^i = x[i \ldots n-1]x[0 \ldots i-1]$ if $i \neq 0$. Let x_R denote the reverse of string x, $x_R = x[m-1]x[m-2] \ldots x[1]x[0]$. To be consistent with previous works consider the word RAM model of computation with word size $w = \Omega(\log n)$.

3. Fast average-case algorithm for wildcard matching

In this section we wish to show upper bounds for algorithms that either match the average-case complexity of pattern matching or are within a logarithmic factor of optimal.

The algorithms we design in this paper follow a common paradigm for average-case algorithm and consist of two stages: a *filtering* stage and a *verification* stage. In the filtering stage we create a sliding window of length $2m - 1$ and read a small number of characters to try to prove that no occurrences exist in the window. If we cannot show the pattern doesn't exist within the window, the verification algorithm is then run on that window of the text. The window is then shifted by m positions. The verification scheme used in this algorithm consists of naively checking all possible alignments of the pattern against the window. Each possible start position takes $\mathcal{O}(m)$-time and there are m possible start positions for a window of size $2m - 1$ so it takes $\mathcal{O}(m^2)$-time in total. The main focus of the section is therefore on the creation of an efficient filtering scheme.

For the rest of the article we assume that the text t is of length n and is random and uniformly drawn from Σ. It is known that the uniform random string model is not totally realistic for all applications as people don't tend to search

Table 1
Columns 0-7 are the possible starting positions and each row shows one candidate. Clearly each candidate has an associated possible starting position.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
a	b	ϕ	ϕ	b	ϕ	ϕ	a							
	a	b	ϕ	ϕ	b	ϕ	ϕ	a						
		a	b	ϕ	ϕ	b	ϕ	ϕ	a					
			a	b	ϕ	ϕ	b	ϕ	ϕ	a				
				a	b	ϕ	ϕ	b	ϕ	ϕ	a			
					a	b	ϕ	ϕ	b	ϕ	ϕ	a		
						a	b	ϕ	ϕ	b	ϕ	ϕ	a	
							a	b	ϕ	ϕ	b	ϕ	ϕ	a

in random texts. However, the random string model is a reasonable approximation when we consider those stretches of text that do not contain any occurrence of the pattern. Clearly in the situation of an occurrence every character must be inspected, as such we are mainly interested in minimising the characters read in those sections where there are no occurrences. In this situation the random string model is often a reasonable approximation. Additionally it allows us to gain some insight into the overall structure of the problem.

In the rest of this section we will determine an upper bound for the number of character inspections required for one window on the text. Our results are primarily based on reductions to different variants of the set cover problem. The general set cover problem is known to be NP-hard, however we make use of known approximation algorithms which can be computed very efficiently.

First we note a structural property of searching within a window of size $2m-1$ that is important for our later analysis. For any position $0 \leq j < m-1$ in the sliding window, by accessing positions j and $j+m$ we have exactly one c-intersection or one w-intersection per candidate (See Table 1 positions 0 and 8 for an example) and for position $m-1$ every candidate has one c-intersection or one w-intersection with one access. Combining this we get the fact below:

Fact 1. *For a window of length $2m-1$, with at most two accesses, we can ensure every candidate in the window will either have 1 c-intersection or 1 w-intersection.*

We also note that by looking at Table 1 and focusing on a specific column we can define a string for each column. For example, position 1 gives a and position 8 gives $a\phi\phi b\phi\phi b$. By concatenating the string from column j with the string from column $j+m$ we always get x_R or a rotation of x_R. For position 7 we get x_R for position 0 and 8 we get $aa\phi\phi b\phi\phi b$ which is x_R^{m-1} and it can easily be seen that every rotation of x_R can be formed this way.

Based on this observation we define the following family of sets for $0 \leq j < m$ that we will use throughout the rest of the paper.

$$i \in S_j \text{ if and only if } x_R^{m-j-1}[i] \neq \phi$$

Each set contains the possible start positions of candidates which have non wildcard characters aligned at the accessed positions; in other words the set consists of possible start positions of all candidates with a c-intersection at these accesses. By Fact 1 each set corresponds to one or two accesses.

Now we define a combinatorial property that is similar to the concept of a period called a *wc-period*. This property gives a simple way to reduce the computational overhead for certain strings with wildcards.

Definition 2. Let x be a pattern of length m over Σ' and let x_b be a binary mask of length m such that $x_b[i] = a$ *iff* $x[i] = \phi$ and $x_b[i] = b$ otherwise. Then x has a *wc-period* of length p if and only if x_b has a period of length p.

Based on this we get the following Lemma.

Lemma 3. *Let x be a pattern of length m over Σ' with wc-period p. Then any inspection scheme of x c-intersecting the candidates at possible starting positions $\{0, 1, \ldots, p-1\}$ k times, is also an inspection scheme c-intersecting the candidates at possible starting positions $\{0, 1, \ldots, m-1\}$ k times.*

Proof. For $i \in \{0, 1, \ldots, p-1\}$, if $i \in S_j$ then by the definition of periodicity $i + cp \in S_j$ for any integer c such that $0 < i + cp < m$. □

Due to the above we can focus on finding c-intersections in the candidates whose potential starting positions are contained within the shortest wc-period of the string. Each time a c-intersection occurs, there is a chance that the two non-wildcard characters do not match, this would invalidate the candidate as a possible occurrence of the pattern. Our goal in the remainder of this section is to determine how many c-intersections each candidate needs to guarantee that the probability of triggering a verification is low. The idea is to make the expected verification time per window negligible. One way

to do this is to ensure that all candidates have at least $3\log_\sigma m$ c-intersections, this causes the probability that an individual candidate is not invalidated by a mismatch to be at most $\frac{1}{\sigma}^{3\log_\sigma m} = 1/m^3$. As there are $\mathcal{O}(m)$ candidates this means that the probability that at least one has not been invalidated is at most $1/m^2$. We pick $1/m^2$ as this means that the expected verification time for each window is constant as $\mathcal{O}(m^2) \times 1/m^2 = \mathcal{O}(1)$.

In the next section we will focus on how to design an inspection scheme that guarantees each candidate has at least $3\log_\sigma m$ c-intersections. The difficulty in designing an inspection scheme is that we must explicitly consider how each individual candidate is affected by an access in the presence of wildcards. Clearly a w-intersection does not help to identify or rule out possible occurrences, so this must be taken into account in the analysis. Due to this we model the inspection scheme as a number of set covers as each set cover guarantees at least 1 c-intersection per candidate. We therefore derive our upper bound via reduction to the set cover problem and will use some known approximation results for the set cover problem to derive our result.

3.1. Sparse wildcard matching

First we consider the case of sparse wildcard matching. We consider the problem sparse when $\lim_{m\to\infty} \frac{g_p}{p} = 1 - \epsilon$. To design an inspection scheme for this case we use a reduction to the ε-dense set cover problem. Below we formally state the problem along with a known approximation results.

Problem 2. Given $0 < \varepsilon < 0$, a set of elements $\mathsf{U} = \{0, 1, 2, \ldots, r-1\}$, a family $\mathbb{S}$ of ℓ sets such that every element of U occurs in εr sets and the union of $\mathbb{S}$ equals U. Find the minimum number of sets from $\mathbb{S}$ such that their union is U.

Lemma 4 *([14]). There exists an approximation algorithm for the ε-dense set cover problem with output size* $\log_{\frac{1}{1-\varepsilon}} r$.

By the definition of the sets $S_0, \ldots, S_{p-1}$ each candidate occurs in $p - g_p$ sets, so the problem can be seen as an ε-dense set cover problem when $p - g_p \geq \varepsilon p$. This condition is true when there exists some $0 < \epsilon < 1$:

$$\frac{g_p}{p} \leq 1 - \varepsilon$$

For our purpose we use the slightly stronger condition $\lim_{m\to\infty} \frac{g_p}{p} = 1 - \epsilon$. We wish to study an iterated version of the set cover problem for our purposes. We wish to apply the approximation algorithm for ε-dense set cover problem, remove the set cover and apply the algorithm again until we achieve at least $3\log_\sigma m$ c-intersections per candidate. We now define a family of functions, which all take the pattern x as argument, which allow us to analyse the iterated version of the ε-dense set cover problem applied on S. The goal is to define a sequence of functions that bound the maximum output size of the iterated ε-dense set cover problem after i applications. The functions used in our analysis are stated below:

- $S^i(x)$ is the set S after i set covers have been removed,
- $\mathcal{D}^i(x)$ is the density of S^i,
- $\mathbb{O}^i(x)$ is the output size of i-th set cover,
- $\mathcal{G}^i(x)$ is the size of the inspection scheme required to guarantee at least i c-intersections per candidate.

Our proof will proceed as follows

1. We first show that $\mathcal{D}^0(x)$ is asymptotically the same as $\mathcal{D}^i(x)$ when i is not too large,
2. Based on this we will bound the size of $\mathbb{O}^0, \mathbb{O}^1, \ldots, \mathbb{O}^i$ and therefore $\mathcal{G}^0, \mathcal{G}^1, \ldots, \mathcal{G}^i$ and derive the required result.

By Lemma 4 the size of the output from the set cover is

$$\log_{\frac{1}{1-\varepsilon}} r = \frac{\log_2 r}{\log_2 \frac{1}{1-\varepsilon}}$$

We note that $\mathbb{O}^0 = 0$ and by substituting $\epsilon = \frac{m-g}{m}$ we get that $\mathbb{O}^1$ would be of size.

$$\mathbb{O}^1 = \frac{\log_2 p}{\log_2 \frac{1}{1-\mathcal{D}^0}}$$

Inspecting the above definition of O^1 and Lemma 4 we can see that the $\log_2 p$ factor comes from the underlying universe size. Each time that we run the algorithm we wish to c-intersect all p candidates, so the size of the universe is unchanging and this factor remains unchanged after multiple applications of the algorithm, it is only the denominator that will change. From this we can get the following definition of $\mathbb{O}^i$.

$$\mathbb{O}^i = \frac{\log_2 p}{\log_2 \frac{1}{1-\mathcal{D}^{i-1}}}$$

After each application of the set cover algorithm we remove the set cover, reducing the total number of sets available and the number of sets a candidate occurs in. Due to this we must ensure that after repeated applications the resulting set is still dense enough. We are now in a position to define a density function which gives the density of the remaining sets after i applications of the ε-dense set cover problem.

$$\mathcal{D}^0(x) = \frac{p - g_p}{p}$$
$$\mathcal{D}^i(x) = \frac{p - g_p - \alpha}{p - \alpha}, \text{ where}$$
$$\alpha = \sum_{j=0}^{i} \mathbb{O}^j$$

With the above definitions in place we can see by inspection, or by exhaustive computation, that for $i = c = \mathcal{O}(1)$ it holds that:

$$\mathcal{O}(\mathbb{O}^1) = \mathcal{O}(\mathbb{O}^2) = \ldots = \mathcal{O}(\mathbb{O}^c) = \mathcal{O}(\log p)$$

By substituting those values into the definition of $\mathcal{D}^i$ we can also see that:

$$\lim_{m\to\infty} \mathcal{D}^0 = \lim_{m\to\infty} \mathcal{D}^1 = \ldots = \lim_{m\to\infty} \mathcal{D}^c$$

The number of characters inspected to guarantee $i > 0 = \mathcal{O}(1)$ c-intersections per candidate is then given by:

$$\mathcal{G}^i(x) = \underbrace{\mathcal{O}(\log_2 p) + \mathcal{O}(\log_2 p) + \ldots + \mathcal{O}(\log_2 p)}_{\text{c times}}$$

Now that we have seen how these formulas behave for any constant value, we wish to show that this result also holds for some non constant values of i. More specifically we wish to show that the above results still holds for $i = \mathcal{O}(\log_\sigma m)$ so that we can bound the size of the inspection scheme that c-intersects every candidate $3\log_\sigma m$ times. Looking first at $\mathcal{D}^i$ for $i = \mathcal{O}(\log_\sigma m)$ we can see that removing $\mathcal{O}(\log_\sigma m)$ set covers of size $\mathcal{O}(\log p)$ has no significant impact on the density and the limit of $\mathcal{D}^i$ is the same as $\mathcal{D}^0$ when $i = \mathcal{O}(\log_\sigma m)$. We also note that the size of $\mathbb{O}^i$ will only change once enough sets are removed to alter the limit of $\mathcal{D}^i$. So for $i = \mathcal{O}(\log_\sigma m)$ we have that:

$$\alpha = \sum_{j=0}^{i} \mathbb{O}^j = \mathcal{O}(\log_\sigma m)\mathcal{O}(\log p)$$
$$\mathcal{D}^i(x) = \frac{p - g_p - \mathcal{O}(\log_\sigma m)\mathcal{O}(\log p)}{p - \mathcal{O}(\log_\sigma m)\mathcal{O}(\log p)}$$

As the limits of $\mathcal{D}^i(x)$ remain unaffected, the output size of $\mathbb{O}^i(x)$ too remains asymptotically the same, that is $\mathbb{O}^i(x) = \mathcal{O}(\log p)$ when $i = \mathcal{O}(\log p)$.

It remains to ensure we have enough characters to read in the window to make this analysis valid. This is true when the below holds, which can always be made true for sufficiently large m.

$$g_p + \mathbb{O}^0 + \mathbb{O}^1 \ldots + \mathbb{O}^{\mathcal{O}(\log p)-1} + \mathbb{O}^{\mathcal{O}(\log p)} < \epsilon p$$

which is always true for sufficiently large m. Finally we get that for sufficiently large p and $i = \mathcal{O}(\log p)$:

$$\mathcal{G}^i(x) = \mathcal{O}(\log_\sigma m \log_2 p)$$

Recall that we wish to intersect each candidate at least $3\log_\sigma m$ times so for $i = 3\log_\sigma m + 1$ we get the size of the inspection scheme as $\mathcal{G}^{3\log_\sigma m+1}(x) = \mathcal{O}(\log_\sigma m \log_2 p)$ for sufficiently large p. This upper bound is for a block of the text of size $2m-1$ and assumes all character inspections can be considered as independent tests which may not be the case depending on how far the sliding windows shifts over the text after a window is invalidated.

To convert this to a general bound for a text of size n consider the text partitioned into blocks of size $2m-1$ that overlap by m characters. After inspecting $\mathcal{O}(\log_\sigma m \log_2 p)$ characters we expect that we can discard the entire block and can shift by m characters. It may be the case that we have already read some of the characters we need to inspect in the new window, but as the analysis above holds for $i = \mathcal{O}(\log_\sigma m)$ we could simply compute an inspection scheme that c-intersects each candidate $6\log_\sigma m$ times to ensure enough sets for the new window to have an independent inspection scheme that c-intersects every candidate at least $3\log_\sigma m$ times. Combining all of this we get the next result.

Theorem 5. *For any constant $0 < \epsilon < 1$ there exists an algorithm for Problem 1 with average-case search time $\mathcal{O}(\frac{n \log_\sigma m \log_2 p}{m})$ when $\lim_{m\to\infty} \frac{g_p}{p} = 1 - \epsilon$.*

3.2. Super sparse wildcard matching

We now consider the case of supersparse wildcard matching. We consider the problem super sparse when $\lim_{m\to\infty} \frac{g_p}{p} = 0$. We recall the definition of the δ-superdense set cover problem along with a known approximation results.

Problem 3. Given constants $0 < \delta < 1$ and $\gamma > 0$, a set of elements $\mathsf{U} = \{0, 1, 2, \ldots, r-1\}$, a family S of ℓ sets such that every element of U occurs in $r - \gamma r^\delta$ sets and the union of S equals U. Find the minimum number of sets from S such that their union is U.

Lemma 6 *([14]). There exists an algorithm for the δ-superdense set cover problem with output size $\frac{2}{1-\delta} \log_\ell r$.*

By the definition of the sets $S_1, \ldots, S_p$ each candidate occurs in $p - g_p$ sets, we have $\ell = p$ sets and $r = p$. So when $p - g_p > p - \gamma p^\delta$ we can consider finding an inspection scheme is an instance of Problem 3 and this is true for the following condition on $\frac{g_p}{p}$ for any $0 < \delta < 1$ and $\gamma < 0$:

$$\frac{g_p}{p} < \gamma p^{\delta - 1}$$

Or equivalently that $\lim_{m\to\infty} \frac{g_p}{p} = 0$. With the values of ℓ and r given above, Lemma 6 shows that the output of the approximation is of constant size. We will apply the approximation algorithm for the δ-superdense set cover problem, remove the set cover and repeat the application until we achieve $3 \log_\sigma m$ c-intersections per candidate. Repeated application of the algorithm and an almost identical argument to that used for the sparse wildcard matching problem allows the construction of an inspection scheme of size $\mathcal{O}(\log_\sigma m)$.

Theorem 7. *There exists an algorithm for Problem 1 with average-case search time $\mathcal{O}(\frac{n \log_\sigma m}{m})$ when $\lim_{m\to\infty} \frac{g_p}{p} = 0$.*

4. A general lower bound

In this section we will consider the lower bound for the average-case complexity of pattern matching with wildcards. First we start with a simple result for pattern matching algorithms that inspect the positions of a sliding window on the text in the same order regardless of the pattern being searched for. This is a ubiquitous property in pattern matching algorithms and in the study of average-case complexity it is known that the order in which the characters of the text are inspected can affect the time complexity of the problem. In [22] it was shown that having a predetermined inspection scheme negatively effects the average-case runtime of exact pattern matching algorithms for $m < n < 2m$ when compared with a dynamic inspection scheme. For pattern matching with wildcards we show that this effect is more pronounced.

We refer to the sequence of probing positions on the text as the *inspection scheme*, denoted $\mathcal{I} = (i_0, i_1, \ldots, i_{2m-2})$ where i_j is the index of the j-th position inspected. If the inspection scheme is the same for any pattern then we call these algorithms *static*, all other algorithms are *dynamic*. We show that for any static inspection scheme there exists patterns that performs badly in the best case. Considering an inspection scheme $\mathcal{I} = (i_0, i_1, \ldots, i_{2m-2})$ there exists a pattern of length m with wildcards in the first g positions of I. Therefore we get the following simple lower bound for static algorithms which resembles the lower bound for approximate matching.

Theorem 8. *Static algorithms solving Problem 1 have a lower bound of $\Omega(\frac{n(g+\log m)}{m})$.*

Now we consider dynamic algorithms and derive an average-case lower bound for any algorithm solving Problem 1 with an arbitrary pattern and any value for g that is significantly lower than the above bound. We have shown that in this problem the order in which the characters are inspected becomes important. Some inspection schemes do not have much effect on the expected number of candidates still remaining. It can be the case that inspections that, should wildcards not be present, would lead to a window being invalidated may give very little information when wildcards exist.

We consider the following simplification of the problem for the lower bound. The text is partitioned into non-overlapping windows of size $2m - 1$, and we only report matches that occur entirely within one block. An optimistic assumption as this excludes those matches which overlap two blocks. In the following section we will determine a lower bound for the number of character inspections required for *one* window. The lower bound for the general problem can then be derived from this.

Given an access to position z in window b we can only invalidate candidate c if there exists some y such that $x[y] \neq b[z]$ and $c + y = z$. For all c-intersections at access i_j, there is a probability of at most $1/\sigma$ that the candidate will not be invalidated. For those candidates where this access intersects a wildcard there is probability 1 it will not be invalidated. We make the following assumptions in our analysis.

- Any access intersects all $m - g$ candidates.
- Intersections are distributed uniformly across all candidates.

The effect of this is that $m - g$ candidates have a chance of being ruled out at every block access. After k block accesses in this model we have made $(m-g)k$ intersections and we assume that these are distributed uniformly across all m candidates. This is optimistic as this means that we always intersect candidates with the highest probability of occurrence, something which may not actually be possible. The expected number of candidate not ruled out is then be evaluated as follows:

$$\sum_{i=0}^{m-1} \frac{1}{\sigma^{\frac{(m-g)k}{m}}} = \frac{m}{\sigma^{\frac{(m-g)k}{m}}}$$

For each candidate we need to either rule it out as a possible starting position or declare a match. So the optimal is to determine when we would expect to have ruled out every candidate position or read $2m - 1$ characters. We minimise the following so that we expect to have less than one candidate left or until we have read all $2m - 1$ positions.

$$\frac{m}{\sigma^{\frac{(m-g)k}{m}}} < 1$$

Rearranging this we get the following:

$$\log_\sigma m < \frac{(m-g)k}{m}$$

$$\frac{m \log_\sigma m}{m-g} < k$$

Now we know that the lower bound for each block is $\Omega(\frac{m \log_\sigma m}{m-g})$ and there are $n/2m$ blocks and the result below follows:

Theorem 9. *The average-case lower bound for wildcard matching with wildcards only in the pattern is* $\Omega(\frac{n \log_\sigma m}{m-g})$.

This result suggests that pattern matching with wildcards is on average asymptotically harder than exact string matching, in particular when $g = m - f(m)$ and $f(m) = o(m)$. It is simple to design patterns that require the lower bound on average.[2] This lower bound matches exact string matching except for those extreme values discussed above, we note the following Fact below and achieve our final result.

Fact 10. *If* $\lim_{m\to\infty} \frac{g}{m} = \lim_{m\to\infty} 1 - f(m) = 1$ *any algorithm takes at least* $\Omega(\frac{n \log_\sigma m}{f(m)})$*-time on average.*

5. Conclusions

In this paper we have investigated the average-case complexity of pattern matching with wildcards. The question of a tight bound on the search complexity of pattern matching with wildcards remains open. We have shown an algorithm which has optimal average-case search time when there are few wildcards in the pattern and within a logarithmic factor up until $\frac{g_p}{p} \leq 1 - \epsilon$. We showed a lower bound for pattern matching with wildcards shows a provable separation in time complexity between wildcard matching and exact matching for extreme values of g.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

[2] A pattern such as $a^{m-g}\phi^g$ achieves this.

References

[1] Philip Bille, Inge Li Gørtz, Hjalte Wedel Vildhøj, Søren Vind, String indexing for patterns with wildcards, in: Fedor V. Fomin, Petteri Kaski (Eds.), Algorithm Theory, in: Lecture Notes in Computer Science, vol. 7357, 2012, pp. 283–294.
[2] Pao-Yang Chen, Shawn Cokus, Matteo Pellegrini, BS Seeker: precise mapping for bisulfite sequencing, BMC Bioinform. 11 (2010) 203.
[3] Peter Clifford, Raphaël Clifford, Simple deterministic wildcard matching, Inf. Process. Lett. 101 (2) (January 2007) 53–54.
[4] Raphaël Clifford, Klim Efremenko, Ely Porat, Amir Rothschild, Pattern matching with don't cares and few errors, J. Comput. Syst. Sci. 76 (2) (2010) 115–124.
[5] Richard Cole, Lee-Ad Gottlieb, Moshe Lewenstein, Dictionary matching and indexing with errors and don't cares, in: Proc. of the 36th Annual ACM STOC, ACM, 2004, pp. 91–100.
[6] Richard Cole, Ramesh Hariharan, Verifying candidate matches in sparse and wildcard matching, in: Proc. of the 34th Annual ACM STOC, 2002, pp. 592–601.
[7] Funda Ergün, Elena Grigorescu, Erfan Sadeqi Azer, Samson Zhou, Periodicity in data streams with wildcards, in: Fedor V. Fomin, Vladimir V. Podolskii (Eds.), Computer Science – Theory and Applications, Springer International Publishing, Cham, 2018, pp. 90–105.
[8] Michael J. Fischer, Michael S. Paterson, String-matching and other products, Technical report, Cambridge, MA, USA, 1974.
[9] Paweł Gawrychowski, Moshe Lewenstein, Patrick K. Nicholson, Weighted ancestors in suffix trees, in: Andreas S. Schulz, Dorothea Wagner (Eds.), Algorithms - ESA 2014, in: LNCS, vol. 8737, 2014, pp. 455–466.
[10] Wing-Kai Hon, Tsung-Han Ku, Rahul Shah, Sharma V. Thankachan, Jeffrey Scott Vitter, Compressed text indexing with wildcards, in: Roberto Grossi, Fabrizio Sebastiani, Fabrizio Silvestri (Eds.), SPIRE, in: LNCS, vol. 7024, 2011, pp. 267–277.
[11] S. Costas Iliopoulos, M. Sohel Rahman, Pattern matching algorithms with don't cares, in: 33rd International Conference on Current Trends in Theory and Practice of Computer Science, SOFSEM (2), 2007, pp. 116–126.
[12] Piotr Indyk, Faster algorithms for string matching problems: matching the convolution bound, in: Proc. of the 39th Symposium on Foundations of Computer Science, 1998, pp. 166–173.
[13] Adam Kalai, Efficient pattern-matching with don't cares, in: Proc. of the 13th Annual ACM-SIAM SODA, 2002, pp. 655–656.
[14] Marek Karpinski, Alexander Zelikovsky, Approximating dense cases of covering problems, Technical report, 1996.
[15] Moshe Lewenstein, J. Ian Munro, Venkatesh Raman, Sharma V. Thankachan, Less space: indexing for queries with wildcards, Theor. Comput. Sci. 557 (2014) 120–127.
[16] Moshe Lewenstein, Yakov Nekrich, Jeffrey Scott Vitter, Space-efficient string indexing for wildcard pattern matching, CoRR, arXiv:1401.0625 [abs], 2014.
[17] S. Muthukrishnan, Krishna Palem, Non-standard stringology: algorithms and complexity, in: Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing, STOC '94, ACM, New York, NY, USA, 1994, pp. 770–779.
[18] Ron Y. Pinter, Efficient string matching with don't-care patterns, in: Alberto Apostolico, Zvi Galil (Eds.), Combinatorial Algorithms on Words, in: NATO ASI Series, vol. 12, 1985, pp. 11–29.
[19] Andrew Smith, Zhenyu Xuan, Michael Zhang, Using quality scores and longer reads improves accuracy of solexa read mapping, BMC Bioinform. 9 (2008) 128.
[20] Chris Thachuk, Succincter text indexing with wildcards, in: Raffaele Giancarlo, Giovanni Manzini (Eds.), CPM, in: LNCS, vol. 6661, 2011, pp. 27–40.
[21] Yuanxin Xi, Wei Li, BSMAP: whole genome bisulfite sequence mapping program, BMC Bioinform. 10 (2009) 232.
[22] Andrew Chi-Chih Yao, The complexity of pattern matching for a random string, SIAM J. Comput. 8 (3) (1979) 368–387.