EPSRC Vacation Bursary

A Practical Investigation Into Modern Pattern Matching Techniques

Ben Smithers

September 15, 2010

1 Introduction

Over recent years, there have been many theoretical advances in approximate pattern matching. The aim of this project has been to consider how these advances perform in practice, with the general aim of comparing the methods against a naïve approach in order to determine at what input sizes they become practical.

Approximate pattern matching considers searching areas of a text string for areas which are 'similar' to a given pattern. Because there are such a vast number of different measures of similarity and algorithms for solving these problems, it is clearly beyond the capabilities of one person or project to hope to consider all of them. Hence this project has also included the founding of an online Wiki, StringPedia (http://stringpedia.bsmithers.co.uk), in the hope that others will join this initiative helping to create both a library of code to solve these various problems and a body of knowledge into the practical aspects of the solutions.

The particular problems considered during this project were the exact matching with don't cares problem and the k-mismatches problem. The problems, their solutions and the outcome of the practical analysis will be discussed in sections 2 and 3. Section 4 details some of the work there was insufficient time to complete and immediate open questions generated from this work. In section 5, a brief overview of the library of code produced and what it allows is given whilst I conclude with my thoughts on the project in section 6.

2 Matching With Don't Cares

2.1 The Problem

The exact matching problem is well studied [6]. It considers finding substrings of a piece of text which exactly match the given pattern. Formally, it may be defined as: given a text t of length n and a pattern p of length m, determine if:

$$\exists j < n - m + 1 : p_i = t_{i+j} \ \forall \ 0 < i < m$$

Exact matching with don't cares considers the same problem with the variation that the text and pattern may contain an arbitrary number of wild card characters (denoted '?' in this document), each of which match any single character in the alphabet. In this way, the pattern approximately matches the text at a given location. Formally the problem is to determine if:

$$\exists j < n - m + 1 : p_i = t_{i+j} \text{ OR } p_i =? \text{ OR } t_{i+j} =? \ \forall \ 0 < i < m$$

2.2 Solutions

2.2.1 Naïve Method

The naïve method simply considers all $O(n)$ possible alignments of the pattern against the text, checking each until a mis-match is found. It has a time complexity of $O(nm)$.

2.2.2 FFT-Based Methods

In these algorithms, the text and pattern are considered to be strings of integer values (the ASCII value of each character, for example). After this integer conversion, it is easy to see that an exact match (with no wild cards) would occur between $p[0 \dots m-1]$ and $t[i \dots i+m-1]$ if and only if:

$$\sum_{j=0}^{m-1} (p_i - t_{i+j})^2 = 0$$

In order to accommodate wild-cards, this summation may be altered in order to 'mask' characters which are not wildcards. For example, if wildcards are encoded as a 0 during the integer conversion, then an approximate match would occur between $p[0 \dots m-1]$ and $t[i \dots i+m-1]$ if and only if [3]:

$$\sum_{j=0}^{m-1} p_j t_{i+j}(p_i - t_{i+j})^2 = 0$$

In this way, if either p_j or t_{i+j-1} are a wildcard, this element of the summation will be 0, indicating that the characters match. Similarly, if neither character are a wildcard, but the characters do match, then $(p_i - t_{i+j-1})^2 = 0$ thus this element of the summation is 0, indicating that the characters match. Otherwise, if the characters do not match and neither are wildcards, then this element of the summation will be strictly positive. Hence the text substring matches the pattern if and only if all elements of the summation are equal to 0 and thus sum to 0.

The utility of this summation becomes more obvious after expanding the products and including the results in an array S, such that $S_i = 0$ if and only if there is an approximate match between $p[0 \dots m-1]$ and $t[i \dots i+m-1]$. We then have:

$$S_i = \sum_{j=0}^{m-1} (p_j^3 t_{i+j} - 2p_j^2 t_{i+j-1}^2 + p_j t_{i+j}^3)$$

Thus we require the computation of three correlations each of which can be determined by computing a convolution after apply the Fast Fourier Transform (FFT). This takes $O(n \log n)$ time.

The time complexity can be further improved to $O(n \log m)$ time by applying a standard trick: the text is split into n/m substrings of size $2m$ each of which overlap the previous substring by m characters before performing the same computation on each of the substrings [3]. The time complexity is thus:

$$O((n/m)m \log m) = O(n \log m)$$

We note that this overlap is necessary, otherwise we would not find occurrences of the pattern which overlap two substrings of the text.

Whilst not reducing the theoretical time complexity, practical improvements can be made by reducing the number of required convolutions through the use of Monte Carlo methods. For example, in [12] Kalai shows how the use of slightly different masking and the mapping of inputs to random values can reduce the number of convolutions needed to 2 and if the case where there are wild cards only in the pattern then only one convolution is needed (similarly, the deterministic method only needs two convolutions in this special case as we no longer need to multiply by the 'mask' of the text)

2.2.3 Number Theoretic Methods

One potential problem with the use of the Fourier Transform is that it uses floating-point arithmetic. Hence floating point errors may be a problem. For example, a match starting at index i of the text occurs if $S_i = 0$. However, because floating-point arithmetic is used, we must actually report a match if $S_i < \alpha$, where α is some small constant. However, if the maximum possible floating point error that can be generated from the process is greater than α then a match could be missed.

The use of number theoretic methods addresses this issue by only using integer-based arithmetic. Thus this approach involves computing the same products and summations as in the FFT-based methods; the difference is that instead of using FFTs to compute correlations, integer multiplication algorithms are used. The run-time is thus dependent on the particular algorithm used. The library that was used in the implementation of this method selected the algorithm to use based on the inputs and their sizes, choosing from Karatsuba, Schönhage-Strassen and others [11].

2.3 Practical Issues In Implementation

A number of issues arose during implementation some of which either affected or provided variations on the implementation.

2.3.1 Getting The Best Out Of FFTW

The well-known library FFTW (Fastest Fourier Transform in the West) was used to perform optimized Fourier Transforms. One of the reasons that the library is so fast is because it 'plans' each transform – essentially working out how best to break the problem down into smaller tasks that it can solve very efficiently and setting appropriate 'twiddle factors', all the while taking into account the system architecture (FFTW calls its knowledge on how to perform transforms 'Wisdom'). However, in order to properly plan a particular transform, many sets of smaller problems are worked out and in fact benchmarked thus this is a time consuming process. FFTW does though provide a mechanism for planning transforms in advance, although the performance may not be quite as good as a transform planned and measured at run-time.

In addition, transforms is which the number of elements is a power of 2 may be computed more efficiently. Thus generating Wisdom for transforms of size 2^k for a range of k and then padding inputs with 0s so that the number of elements is a power of 2 was shown to be effective. In addition, FFTW's benchmarks show that, for transforms which are not powers of 2, the speed

the transform operates is quite varied.

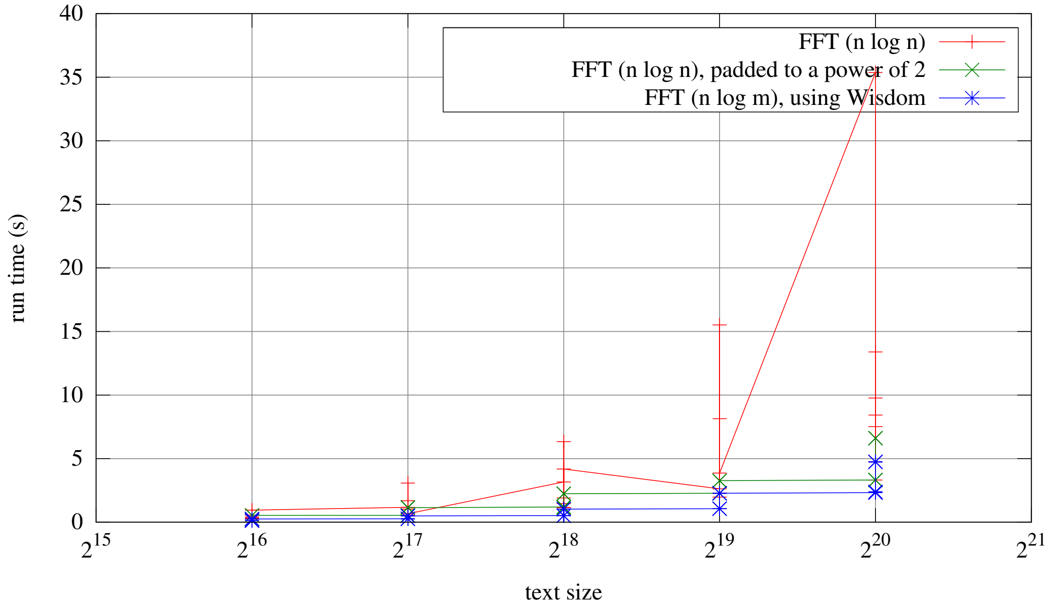

Figure 1: The effect of padding and Wisdom on the run-time of the $O(n \log m)$ method with text sizes that are a power of 2, or very close to a power of 2. The pattern size was fixed at 100.

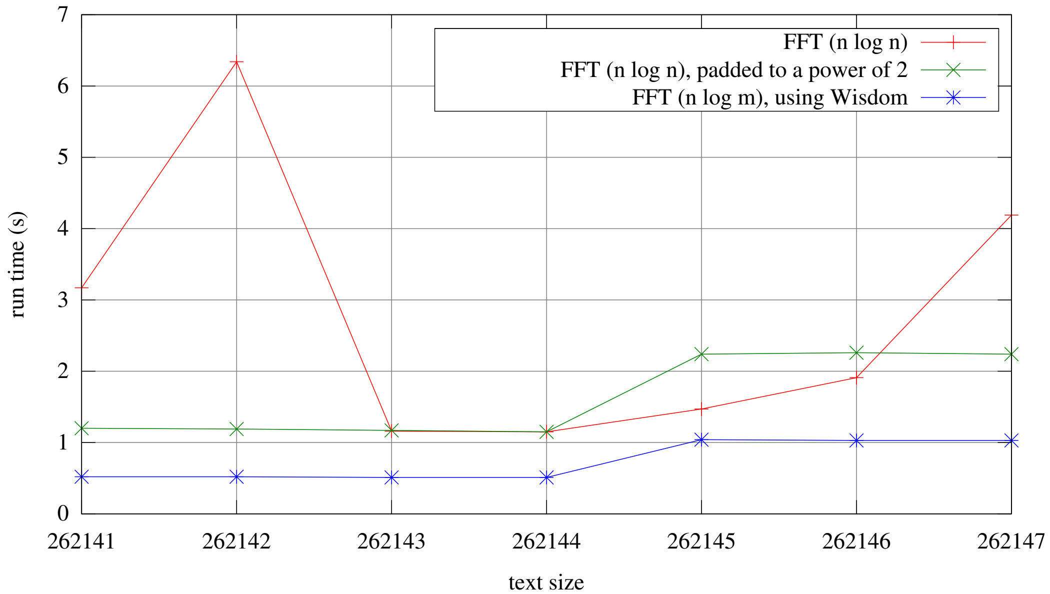

Figure 2: Figure 1, zoomed in at $2^{18} = 262144$

Figures 1 and 2 demonstrate a number of points. Firstly, as was mentioned above, the run-time of the transform is very erratic when the size is not a power of 2 – changing the number of elements by just one can increase the run-time by as much as a factor of 5! Second, we can see that in the case where the transform size is just over a power of 2, then padding can result in a slightly slower run-time than the unpadded case. However, if the pre-generated Wisdom is allowed, then these are always faster than the unpadded case, even when the transform is padded by a large amount. Lastly, and perhaps expectedly, the run-times of the two padded methods show a kind-of stepping behavior – as the size of the text increases past each power of two, the size of the transform is also increased to the next power of 2, hence we see a step which roughly doubles the run-time.

In addition to using Wisdom to improve performance, FFTW supports a wide variety of transforms. All transforms were originally 'Real-to-Complex' and 'Complex-to-Real', which allows exploitation of certain factors because all input is real. FFTW claims this is roughly twice as fast as the standard DFT. Some further improvements were made through the use of 'Real-to-Half-Complex' and 'Half-Complex-to-Real' transforms, which were found to generally increase performance by a small factor. These transforms exploit the symmetry in the output of DFTs where the input is real, thus reducing the number of required multiplications to perform the convolution. Performance benefits did vary, though this has never been found to have a negative impact. Differences in how optimal the generated plans actually were may well be the case of this variation, though this is only speculation.

2.3.2 Run-Time Quirks Of The $O(n \log m)$ FFT-based Method

Early on during the investigation, it was noted that the performance of the $O(n \log m)$ methods (in which the text is split into substrings of size $2m$) was very erratic, especially for very small transform sizes. As figure 3 demonstrates, the run-times of both $O(n \log n)$ methods (with and without Wisdom) are unaffected by the pattern size, whilst all three $O(n \log m)$ methods (with Wisdom, with padding and without either) show erratic run times as m increases, before becoming steady as m becomes larger than 1000-2000.

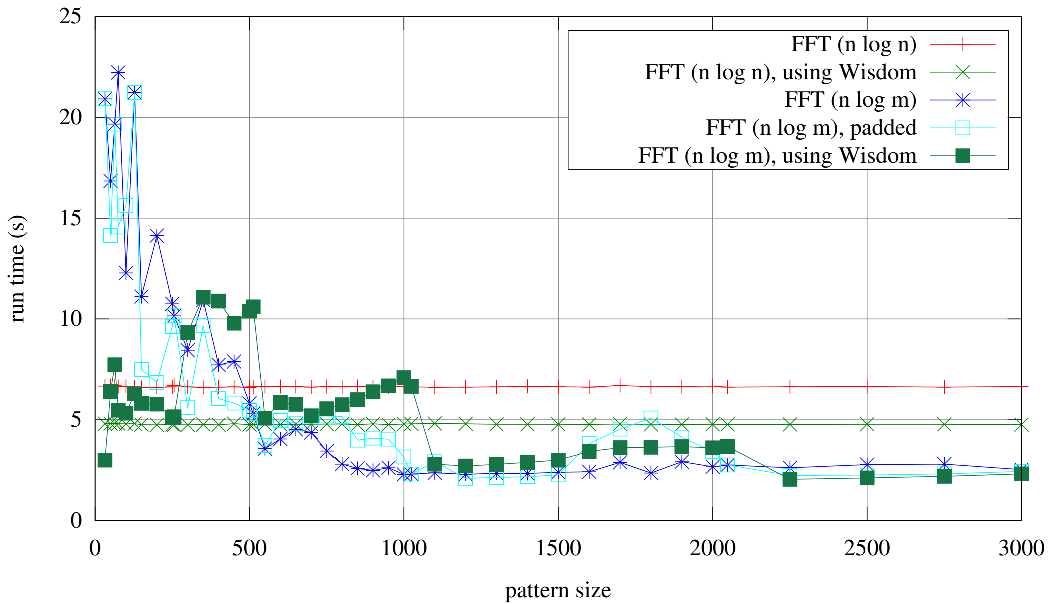

Figure 3: Comparison of FFT-based methods showing erratic behavior with small pattern sizes. The text size was fixed at $2^{21} = 2,097,152$

Clearly the unpadded case could be attributable to the previously noted behavior of FFTW with non-powers of 2, but this does not explain the padded cases. However, it was eventually found that the likely cause of this was another quirk of FFTW (and Fourier Transform algorithms in general) which appear to operate at a higher speed for medium sizes transforms when the transform sizes are a power of 2. Figure 4 shows the speed achieved by various FFT algorithms in FFTW's benchmarks for transforms that are a power of 2 on one particular architecture (further examples can be found here: http://www.fftw.org/speed/).

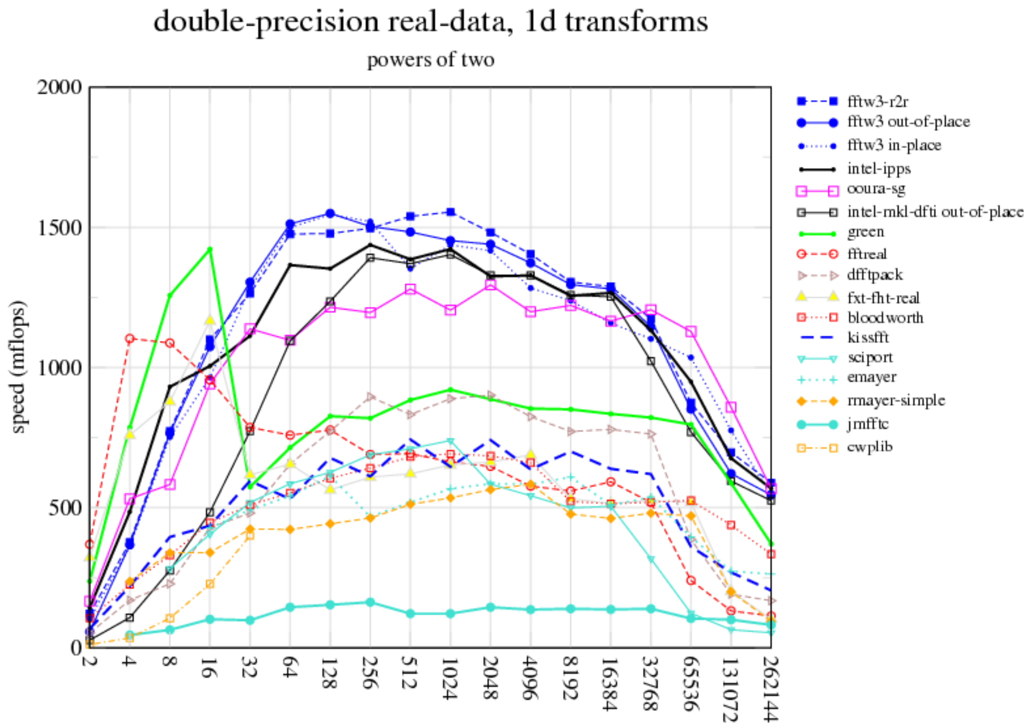

Figure 4: Graph from FFTW's benchmarks taken from: http://www.fftw.org/speed/PentiumM-1.6GHz-gcc/

Given that there are a huge number of transforms being performed when the text is large and the pattern is short, even small variations in the speed of the transforms could have large effects on the overall run-time. Thus a minimum size for the substring of $2^{11} = 2048$ was imposed i.e. no substring of the text was shorter than 2048 (assuming the text was sufficiently long) and the pattern was padded further if necessary.

2.4 Run-Time Analysis

2.4.1 Methodology

- Computer Specifications
  - 2x Intel Xeon 1700Mhz Processors
  - 2GBs of memory
  - Running 32 bit Ubuntu 9.10 (Karmic Koala)

- Compiler: gcc 4.4.1
- Flags: -Wall -pedantic -std=c99 -O2

The text was generated as $a^{n-1}b$ and the pattern as $a^{m-1}b$, ensuring worst-case performance. Note that although neither text nor pattern contain wild-cards in this case, the presence (or lack of) wildcards should have no effect on run times, but leaving them out does allow a comparison against methods where we assume there are no wild cards in the text. Each test was run 7 times and an average taken.

2.4.2 When Is Naïve Really Naïve?

Figure 5 demonstrates run-times of various FFT-based methods, the FLINT library for number theory and the naïve approach for small pattern sizes, showing the points at which the naïve method becomes slower than others. The main findings are:

- The best $O(n \log n)$ approach in this case is the real-to-real transform using Wisdom, which runs faster than the naïve method for $m \gtrapprox 300$
- The best $O(n \log m)$ approach in this case is the real-to-real transform with a forced minimum sub-string size ad Wisdom, which runs faster than the naïve method for $m \gtrapprox 80$
- If we consider the variation where we only allow wild-cards in the pattern and not the text these bounds are further lowered to 200 and 50 respectively.
- The worst FFT-based approach by far is in fact the only FFT-based method which did not use Wisdom. Given a dedicated searching application, however, it seems reasonable that time to generate Wisdom before-hand would be available.
- The FLINT library doesn't outperform the naïve method until $m \gtrapprox 800$

These observations seem to hold within reasonable bounds for other text sizes. Graphs of different text sizes can be found at StringPedia: `http://stringpedia.bsmithers.co.uk/index.php?title=Exact_Pattern_Matching_With_Don%27t_Cares#List_Of_Graphs`

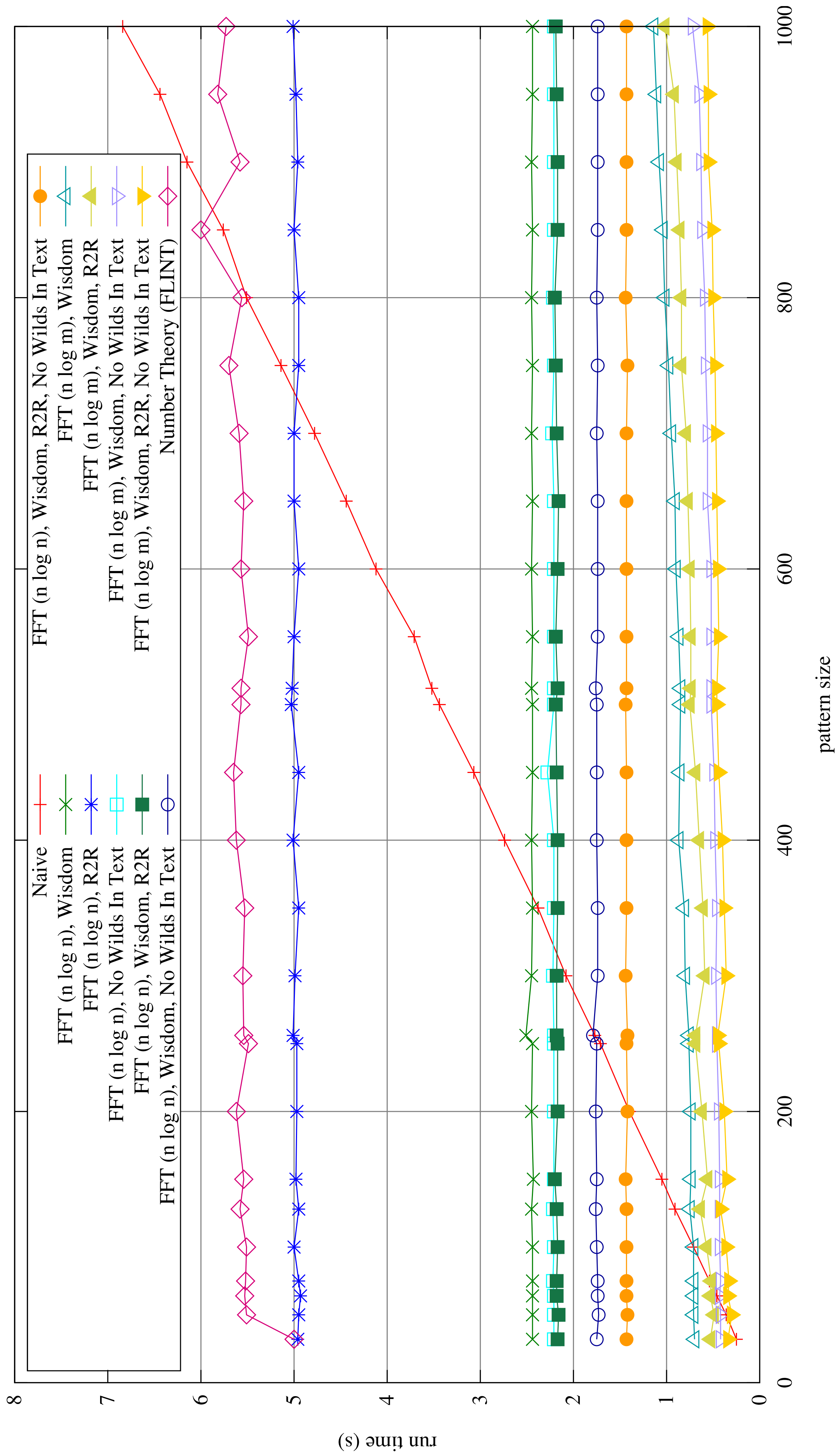

Figure 5: Graph showing the comparison of a number of methods vs the naïve approach.

The point at which the FFT-based methods (and, indeed, the Number Theoretic approach, thought this was not explored) can be further lowered using Monte Carlo Methods, as outlined in section 2.2.2. Figure 6 shows the results:

- For wildcards in both text and pattern, the point at which the naïve method becomes slower is reduced from around 80 to about 45
- If wildcards are only in the pattern, it is reduced from around 50 to about 25

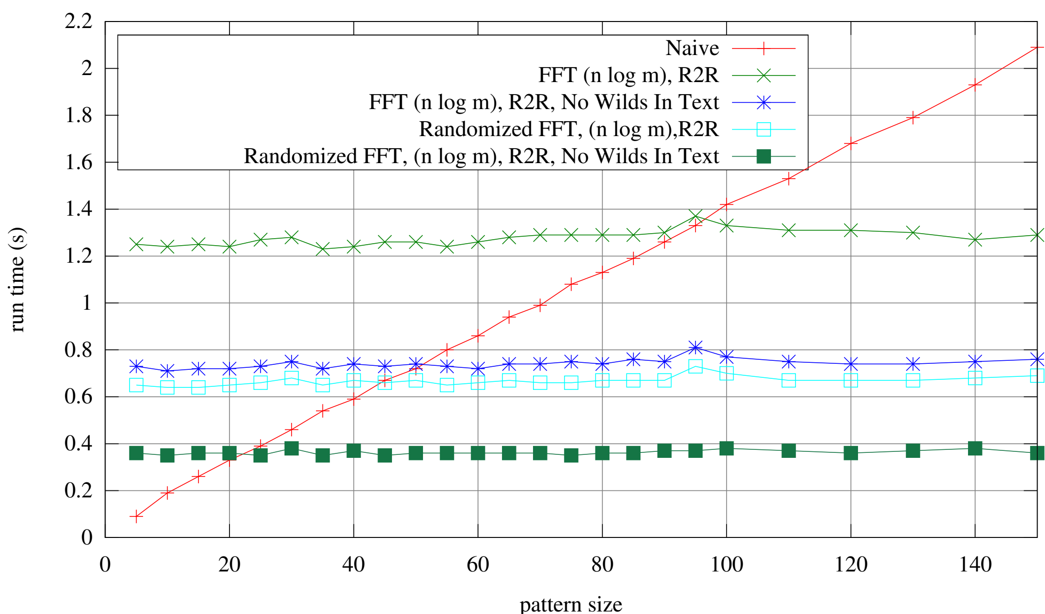

Figure 6: Graph showing the run-times of randomized algorithms versus the naïve method. Two deterministic methods are also included for comparison

Finally, it may be worth noting that, for a fixed text size, the run-time of the naïve approach is linear in the pattern size only if $m \ll n$. This is because the asymptotic notation hides some of the detail – there are in fact $n - m + 1$ possible alignments of the pattern with the text, each of which requires m comparisons in the worst case, resulting in a worst-case of $nm - m^2 + m$ comparisons. In practice, this means the run-time of the naïve method beings to plateau and will eventually start to reduce. Consider, for example, the case where $m = n$: there is only 1 alignments and hence only m comparisons are needed in the worst case. However, for most applications the pattern is likely to be much shorter than the text, so this caveat is of little interest.

2.4.3 Performance Of Number Theoretic Approach

As was shown in the previous section, the number theoretic library performed worse than any of the FFT-based methods for small pattern sizes. Figure 7 shows that this trend continues for larger pattern sizes. As it shows, the performance is much worse than the FFT-based methods and also does vary to some extent; probably due to changes in the algorithm selection. Whether or not this performance difference shown is due to the libraries themselves (i.e. perhaps another number theoretic library exists that performs better) or due to the differences in inherent differences in the methodology it is hard to say, however.

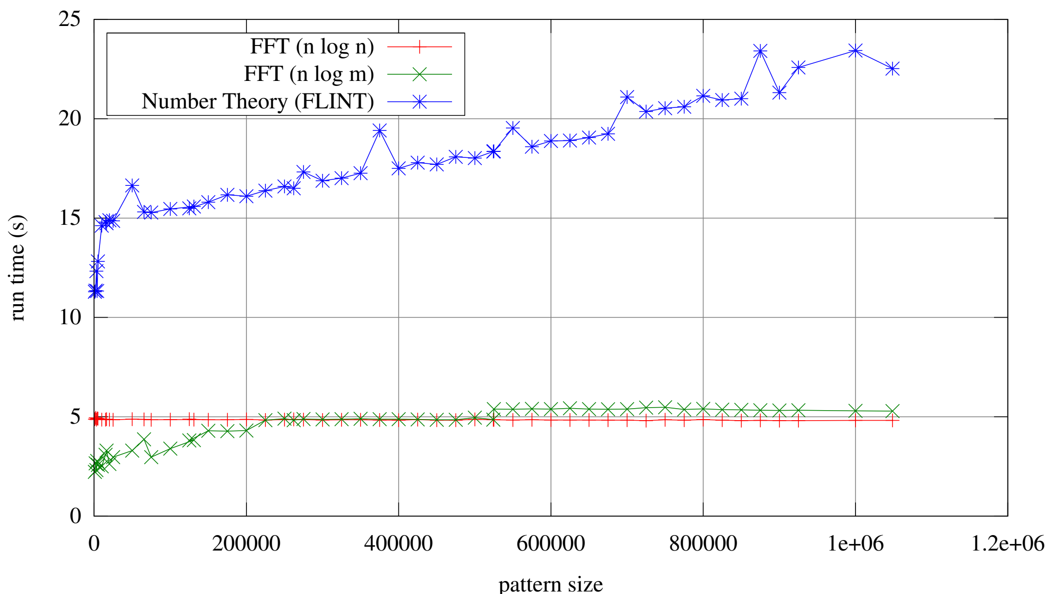

Figure 7: Graph showing the run-time of the number theory library, compared to two FFT-based methods (both using Wisdom) for larger pattern sizes.

2.4.4 $O(n \log n)$ vs $O(n \log m)$

Figure 8 shows run-times for a variety of FFT-based methods. It includes 4 variations: Standard, R2R (using the 'Real-to-Half-Complex' and 'Half-Complex-to-Real' transforms), the special case of no wilds in the text, and a combination of R2R and no wilds in the text. Each of these are shown in $O(n \log n)$ and $O(n \log m)$ algorithms.

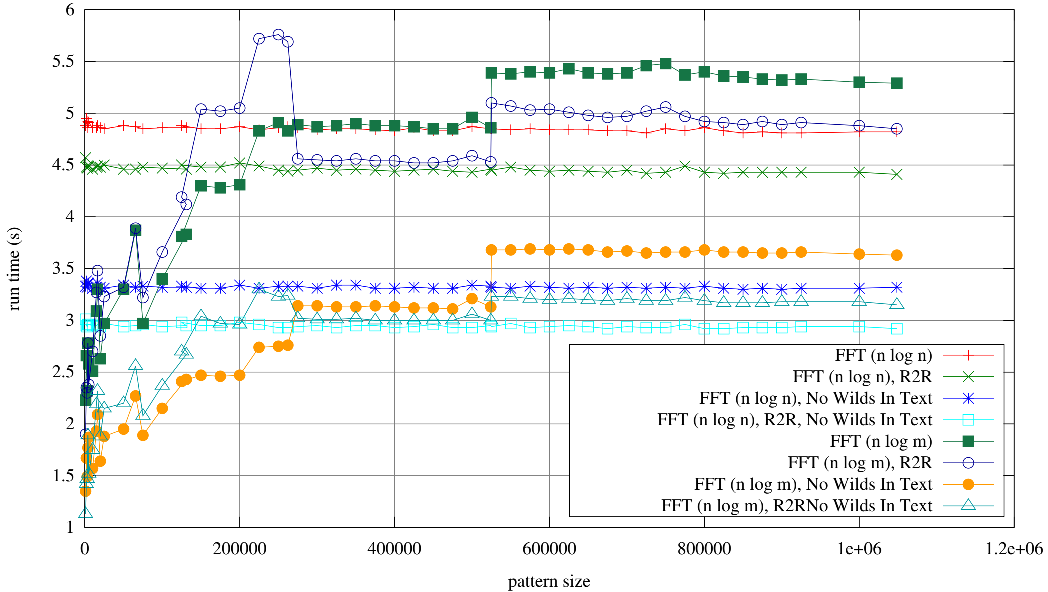

Figure 8: Graph showing the run-time of various FFT methods for larger pattern sizes. All use Wisdom, and all $O(n \log m)$ methods use a minimum sized transform.

The first thing to notice is that all versions of the $O(n \log m)$ algorithm are susceptible to variations in run-time, even though they all have substrings padded to a power of 2. The reason for this seems to be explained by the fact if a given m is 'far' from a power of 2 it must then be rounded considerably, whereas if m is 'close' to a power of 2, it is only rounded a small amount. Because the overlap between text substrings is maintained at m, this results in less relative overlap in the former case and hence less overlapping substrings are required in total to cover all of the text. As m grows, these variations become less and instead we observe the perhaps expected behavior of run-time being unaffected by changes in m between each power of 2.

Next we consider the point at which the $O(n \log m)$ algorithms become slower than their equivalent $O(n \log n)$ counterparts. In each of the 4 variations outlined above, the cross-over point is when $m \approx n/4$.

Finally, although not shown in this graph, we should note that if m increases further past $n/2$, we would see that the run-time of the $O(n \log m)$ algorithm becomes significantly worse than that of the $O(n \log n)$ algorithm. This is because the text substrings which are of length $2m$ are now greater than n, and so we are now computing transforms larger than n, whilst the $O(n \log n)$ only computes transforms of size n.

3 K-Mismatches

3.1 The Problem

K-Mismatches considers a different kind of approximate matching. We are once again given a text t of length n and a pattern p of length m. But rather then giving explicit locations of characters which are allowed to match anything (don't care characters) it specifies a parameter k and says that a match occurs between the pattern and a substring of the text if all of the characters match, except up to k of them. Further, it is required that we report the number of mismatching characters only if it $\leq k$. Otherwise, we say a match does not occur. In other words, for each alignment of the pattern against the text, we must report the Hamming Distance if it is $\leq k$ or that no match has occurred otherwise.

As an example, we consider text 'abacaa' and pattern 'acab'. Figure 9 shows the hamming distance and required output for the algorithm for each alignment of the text against the pattern.

```
Text:                         abacaa          abacaa           abacaa
Pattern:                      acab             acab              acab

Hamming Distance:             2               4                1
Required Output:              2               No Match         1
```

Figure 9: An example of the required output for the k-mismatches problem with text 'abacaa', pattern 'acab' and $k = 2$

In addition, we may also consider the special case where $k = m$ i.e. there is no bound on the hamming distance, so we report it at every alignment.

3.2 Solutions

3.2.1 Naïve Method

Like the naïve solution to the matching with don't cares problem, this method follows straight from the description of the problem: we consider all $O(n)$ alignments of the text and pattern and count all of the mismatches at each alignment, stopping if we find more than k of them. The time complexity is $O(nm)$.

3.2.2 Abrahamson's/Kosaraju's Method

This method given independently by Arahamson and Kosaraju is independent of the value of k [1, 4, 13]. It reports the Hamming Distance at all alignments. There are two key ideas. The first is that the number of mismatches is intrinsically linked to the number of *matches* – if we count matches, we can determine the number of mismatches by subtracting this number from m. The second idea is that we consider characters occurring 'frequently' and those occurring 'infrequently' in the pattern in different ways.

Algorithm 1 - Frequent Characters: as we have seen in section 2.2.2, for a text t of length n and a pattern p of length m, we can compute $\sum_{j=0}^{m-1} p_i t_{i+j}$ for all $0 < i < n - m + 1$ in $O(n \log m)$ time using Fast Fourier Transforms (FFTs). Now, for each frequently occurring character, we can create a 'mask' of the text and pattern (we'll call these t' and p' respectively). For example, $t'_i = 1$ if t_i is the frequent character under consideration and 0 otherwise. The pattern mask, p', is created in the same way. The creation of these masks takes $O(n)$ time. Thus the calculation of $\sum_{j=0}^{m-1} p'_i t'_{i+j}$ for all i yields the number of times this frequent character matches in each alignment. If we repeat this for each frequently occurring character and add the results ($O(n)$ time), we now have the total number of matches involving frequent characters and this takes time equal to $O(n \log m)$ times the number of characters which are frequent.

Algorithm 2 - Infrequent Characters: if a character doesn't occur frequently, then simply counting these occurrences is actually quite efficient. This step requires the creation of an auxiliary array C in which to store the counts. The idea then is that for each infrequent character occurring at index i of the text we determine the each location j of this character in the pattern. We then add one to C_{i-j}. How long does this take? Clearly if we have to check every index of the pattern then it will take $O(nm)$ time.

But we can do better than that. We start by sorting the pattern in $O(m \log m)$ time (in fact, we must actually sort a data structure which also maintains the original location of each character in the pattern so that we can update the correct index of our auxiliary array, C). In addition, we maintain a second structure in which we insert the index of the first of each character within our sorted array. If we are dealing with ASCII characters, a simple array will probably suffice – for multi-byte characters we could employ hashing to reduce memory and time overhead. Thus, for each infrequent character occurring at index i of the text, we can now lookup the starting point in our sorted array (constant time) and we can then add to our auxiliary

array for each occurrence of this infrequent character. This step then takes $O(n)$ times the maximum number of occurrences of a character in the pattern that is still 'infrequent'.

Putting It Together: We are then left with deciding how often a character needs to occur within the pattern to be frequent. If we say that a character occurring at least b times is frequent, then the maximum number of different characters that can be frequent is m/b. Thus the run-time of the first algorithm is $O((m/b)n \log m)$, whilst the run-time of algorithm 2 is simply $O(nb)$. The overall algorithm then runs in $O((m/b)n \log m) + O(nb)$ time. The best time complexity is thus achieved by setting b to minimize this value. This gives $b = \sqrt{m \log m}$ to give an overall time complexity of $O(n\sqrt{m \log m})$.

The final detail then is to work out, for each character in the text, if it is frequent or infrequent. Fortunately, this can be incorporated into the process of creating our table that looks up the indexes of characters in our sorted pattern – we must loop though our sorted pattern in order to generate this anyway so we simply count the occurrences of each character and then store a flag indicating the type of character along with the index of it's location in the sorted pattern.

3.2.3 Amir et al's Method

Amir, Lewenstein, and Porat gave an algorithm which does only report the Hamming Distance if it is $\leq k$ and runs in $O(n\sqrt{k \log k})$ time [2]. The following is a simplified version of the algorithm, which has a time complexity of $O(n\sqrt{k} \log m)$ as presented in [4].

The algorithm is an extension of the Abrahamson/Kosaraju approach outlined above. We start by defining a character to be frequent if it occurs at least $\sqrt{k}$ times in the pattern. We then separate into two cases.

Case 1 - Less than $2\sqrt{k}$ frequent characters: In this case, we simply apply the algorithm of Abrahamson/Kosaraju. How long does this take? Setting $b = \sqrt{k}$ in $O((m/b)n \log m) + O(nb)$ as above would yield $O((m/\sqrt{k})n \log m) + O(n\sqrt{k})$, however a neater bound can be found: since there are at most $2\sqrt{k}$ frequent characters, the total run-time of Algorithm 1 is $O(2\sqrt{k} \cdot n \log m) = O(n\sqrt{k} \log m)$. The total run-time for this case is then $O(n\sqrt{k} \log m + n\sqrt{k})$.

Case 2 - At least $2\sqrt{k}$ frequent characters: With this number of frequent symbols, running Algorithm 1 on all of them becomes too expensive.

Instead, we add a filtering stage in order to reduce the number of possible locations at which a k-mismatch can occur. We do this in a similar way to the 'counting' seen in Algorithm 2 above – indeed, the operation is almost exactly the same: we create an auxiliary array of counts, C. Then, for each frequent character occurring at index i of the text we determine each location j of this character in the pattern. We then add one to C_{i-j}. However, we only consider the first $2\sqrt{k}$ different frequent characters (i.e. if the character in the text is frequent, but not one of the first $2\sqrt{k}$ symbols that are frequent we do nothing) and further only add to the auxiliary array for the first $\sqrt{k}$ locations of each frequent character in the pattern. In practice, we can easily modify the data structures (the sorted pattern coupled with each character's original location and the lookup of character type and starting position in the sorted array) we used in Algorithm 2 (recall that we use these data structures in order to work out which characters are frequent and hence how many are frequent, so they will exist regardless of which case we are using). to suit our purpose: we can traverse the sorted pattern and simply modify the character type of the first $2\sqrt{k}$ frequent characters to flag them in some other way.

After this 'counting' step, we can make the following two observations:

1. The total number of additions made to the auxiliary array C can be no more than $n\sqrt{k}$, since at each of the n indexes of the text we match at most $\sqrt{k}$ occurrences of a frequent character

2. For any given alignment i, the maximum number of matches found (i.e. C_i) is $2\sqrt{k} \cdot \sqrt{k}$, since we are considering a maximum of $\sqrt{k}$ positions of exactly $2\sqrt{k}$ symbols

From the second observation it follows that if a k-mismatch occurs starting at a given index, the count at that index must be at least $2k - k = k$, if not, we have already found more than k mismatches. Since there are a total of $n\sqrt{k}$ additions made to the auxiliary array, then a maximum of $n\sqrt{k}/k$ locations can have a count of at least k. Hence the filtering has now found $O(n\sqrt{k}/k)$ locations at which a k-mismatch can possibly occur.

The final step is to verify these potential locations. This requires the creation of the generalized suffix tree of t and p which is then processed to allow linear-time Lowest Common Ancestor (LCA) queries. An LCA query on a suffix tree is equivalent to finding the Longest Common Extension (LCE) of two substrings [10]. The LCE of two strings is the length of the longest match between them, starting at the beginning of each string. This method of computation allows LCEs to be computed in constant time after linear pre-processing (needed to create the Suffix Tree which supports constant

time LCA queries). Hence if we have found a potential k-mismatch starting at index i of the text, we perform an LCE query between $t[i \cdots i+m]$ and $p[0...m]$, the result of which we'll call l. If $l = m$, we have found a k-mismatch with a Hamming Distance of 0, otherwise we know the Hamming Distance is at least one, since t_{i+l+1} and $p_l + 1$ did not match and so we find LCE($t[i+l+1 \cdots i+m]$ and $p[l+1...m]$). We repeat this until we either find $k+1$ mismatches or we reach the end of the pattern. In the latter case, we've found a k-mismatch which we can then report.

Since the LCE queries take constant time, this verification stage take $O(k)$ time per location. Hence the running time of case 2 is $O(n\sqrt{k}/k \cdot k) = O(n\sqrt{k})$. This means that the running time of the whole algorithm is the maximum of the two cases, which is case 1: $O(n\sqrt{k}\log m)$.

3.3 Avoiding The Use Of A Generalized Suffix Tree

A suffix tree is something of a heavy-weight data structure. While it can be build in linear time and does support constant time LCA/LCE queries, there are significant constant factors hidden. Further, the suffix tree is quite memory hungry. One potential solution is to avoid building a generalized suffix tree. Instead, we can build a suffix tree only of the pattern. In [10] Gusfield describes how most of the functionality of a generalized suffix tree can be achieved though the use of the suffix tree of the pattern and matching statistics. In this project, a different approach was taken, using the work in [5]. Here, the motivation was driven by the use of an online model of computation: the pattern is given in advance, but the text arrives one character at a time. Clearly a suffix-tree of the text is impractical. The method first creates a suffix-tree of the pattern and then creates a representation of the text (a so called *p-representation*) which can be updated as new characters arrive. For completeness, a p-representation is a set of linked *p-regions*. A p-region is a set of 3 integers: i, j and l which together represent a text substring starting at index i of length l, which is equal to a substring of the pattern starting at index j. By building up these p-regions, we represent the whole text in a p-representation. The key results of interest here are:

- A p-representation can be built in $O(n)$ time (in fact, the p-representation can be created with constant time updates for each character of the text that arrives, but this is unnecessary here)
- Once built, the p-representation allows us to compute LCE queries between the pattern and text by instead computing up to 3 pattern-pattern LCE queries (i.e. LCE queries between two substrings of the pattern) which only require the suffix tree of the pattern.

Unfortunately, there was little time for testing how the creation of a suffix tree for the pattern and a p-representation compared to the creation of a generalized suffix tree of the pattern and text. The testing that was performed was very inconclusive and so this is an area where further work is required.

3.4 Run-Time Analysis

3.4.1 Methodology

All tests were run on the same platform as outlined in section 2.4.1 with the exception of some compiler flags (e.g. -DNDEBUG). However, the features of the algorithms mean that constructing inputs is somewhat more difficult. Generally, the aim has been to test worst-case behavior, but this is actually quite difficult to achieve for the k-mismatch problem. The worst-case for the naïve method occurs when the majority of the text and pattern are a single character, with different characters occurring at the end of each string. However, if inputs are constructed (as was the case for matching with don't cares) in this manner, this doesn't provide a fair test because the other algorithms are dependent on the alphabet size and the frequency of occurrences of characters in the pattern. This dependence on the features of the input lends itself more to testing with a corpus, however this does mean that worst-case performance is not guaranteed (or, indeed, very likely at all). Finally, it should be noted that this problem doesn't apply for finding the hamming distance at all locations as such: the naïve method will always have to do the same amount of work for a fixed n and m, regardless of the alphabet size and makeup of the inputs. On the other hand, actual worst case performance of the Abrahamson/Kosaraju method isn't necessarily achieved.

3.4.2 Finding The Hamming Distance At All Alignments

We start by comparing the naïve and Abrahamson/Kosaraju methods. Figure 10 shows a run-time comparison for small pattern sizes, using a corpus of DNA data taken from the Pizza&Chili website [7]. The alphabet size is hence 4, whilst the text size is $4,345,138$.

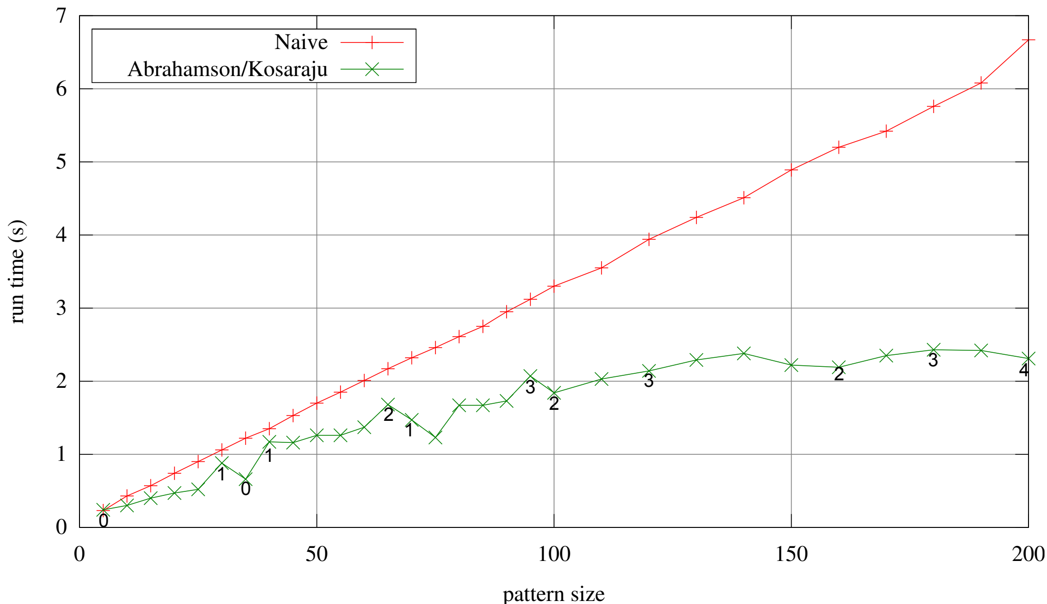

Figure 10: Graph showing the run-time of the two methods for finding the hamming distance at all alignments. The number of different characters matched using Algorithm 1 (with FFTs) is shown when it changes along the graph. The text is from a DNA corpus, with an alphabet size of 4, and has a length of $4,345,138$ characters

There are a number of interesting points. Firstly, in this case the naïve method only outperforms the more complex approach when the pattern size is 5 and even then only by 0.01s. One of the reasons for this is that when the pattern size is small and the overhead of performing FFTs is comparatively large, the likelihood of a character occurring enough times to be frequent in a randomly generated pattern is much lower than that at a large pattern size. This is demonstrated in figure 11 which shows how the the number of occurrences needed to be frequent changes with m (i.e. $\sqrt{m \log m}$) and as a percentage of m: $((\sqrt{m \log m})/m) \cdot 100$. This is reflected in the run-time graph in figure 10, which shows when the number of characters matched with FFTs changes along the graph. We notice that at $m = 200$, all 4 of the alphabet characters are being counted with FFTs. This is unsurprising: as figure 11 shows, a character needs to occur in roughly 20% of the pattern with $m = 200$ and so with $|\Sigma| = 4$, this is quite likely.

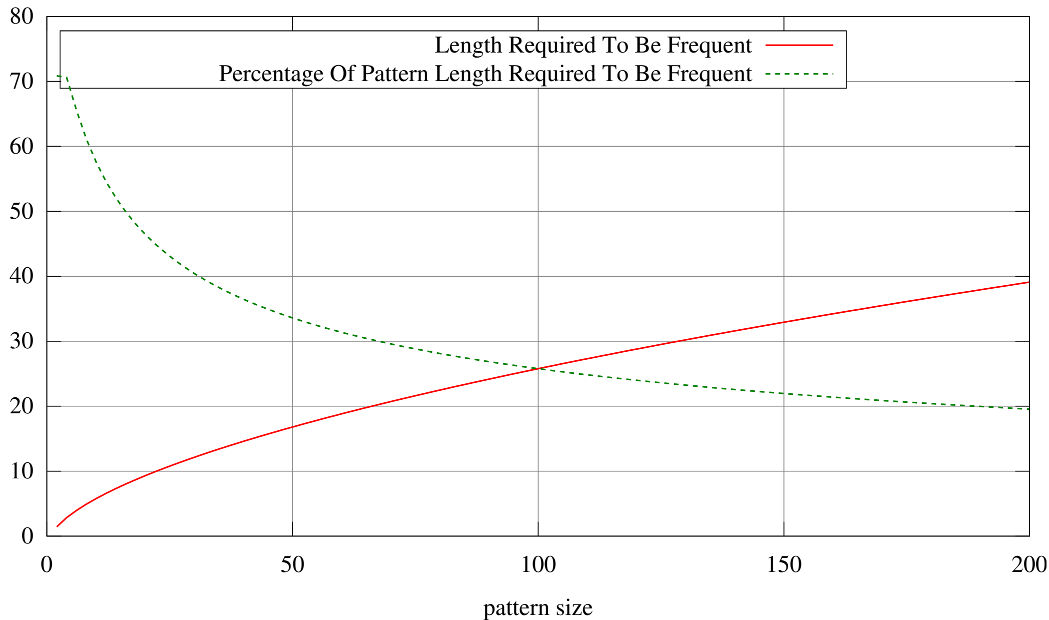

Figure 11: Graph showing how often a character needs to occur in order to be frequent in the Abrahamson/Kosaraju method. The requirements as shown as a number and as a percentage of the length of the pattern

However, this is not the whole story. For example, we can see variations in run-time even when the number of characters matched with FFTs remains the same. There are two reasons for this: firstly, regardless of how often a frequent character occurs, the same amount of work is needed. Thus, a character which occurs only just enough times to be considered frequent will have more relative overhead than a character that occurs far more times. Secondly, and perhaps far more importantly, ignoring the overhead of determining which characters are frequent, Algorithm 2 only counts matches; *it does not have to do any work for mismatches.* Thus, runtime variations as the makeup of the pattern varies are to be expected and, more importantly, this offers the explanation of why this method does so well compared to the naïve approach: the naïve method method always has to perform $m \cdot (n - m + 1)$ comparisons between the text and pattern i.e. it does the same work regardless of whether or not the character matches or mismatches. This is demonstrated further in figure 12, which compares the methods using input taken from an English text sourced from Project Gutenberg. As the alphabet size is much larger, less matches are likely to occur. Further, with a larger alphabet size a character is less likely to occur the required number of times to be considered frequent. These two facts increase the use of Algorithm 1 (matching with counting) and the speed of Algorithm 1.

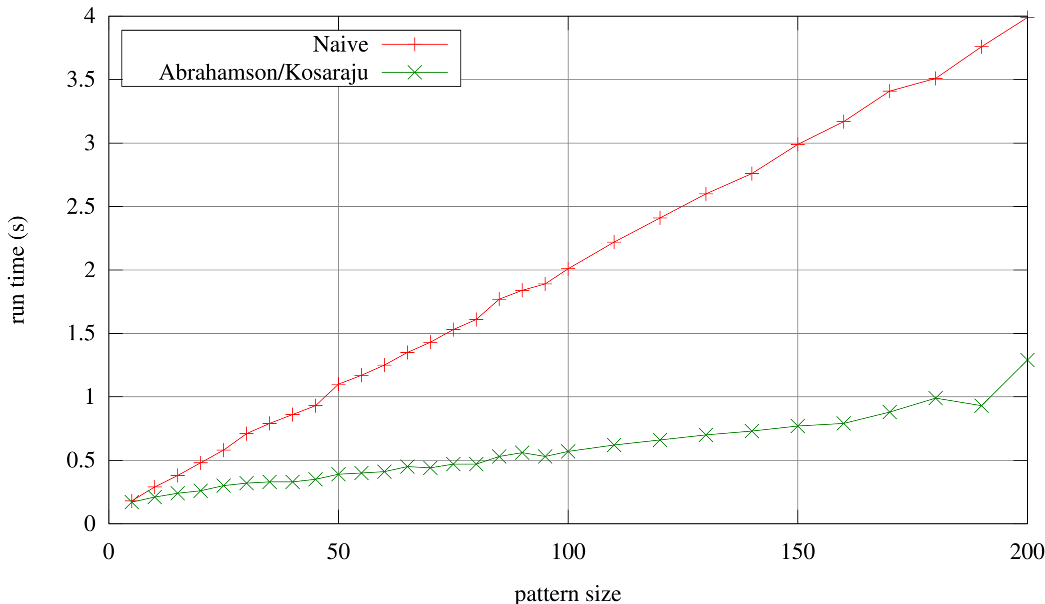

Figure 12: Graph showing the run-time of the two methods for finding the hamming distance at all alignments. The text is from a English text with a relatively large alphabet size and has a length of $4,345,138$ characters

Finally, we may observe an interesting implication. With only small changes to the implementation, the Abrahamson/Kosaraju method can handle don't care characters. Since finding the hamming distance at all locations also tells us if there is an exact match at any given location, this method may actually be useful for solving the exact matching with don't cares problem for small pattern sizes. There are two caveats to this: firstly, the methodology in testing solutions to these two problems has been a little different: if this method was used with the inputs given to the matching with don't cares algorithms, character 'a' would occur frequently and character 'b' infrequently, which may affect how the method performs. On the other hand, beating the naïve algorithm with real-world data for pretty much all pattern sizes is definitely a good thing! Secondly, we have noted that one of the reasons this method is fast is because it only counts matches; introducing (large numbers of) wildcards would increase the number of matches and hence the amount of work needed.

3.4.3 K-Mismatches For Small K

Unfortunately, the results comparing Amir et al's algorithm with the naïve approach have not been so successful. Part of the reason for was mentioned in the methodology above (Section 3.4.1): it was not possible to construct

worst-case performance test cases which also fairly asses running of Amir's algorithm.

Secondly, the implementation was hampered by problems with Suffix Tree libraries. Initially, a fairly simple library by Tsadok and Yona [14] was used, however it did not support LCA queries. A different library called SDSL from the University of Ulm was then used, as it did support LCA queries [9]. One of the focuses of SDSL is compression, so it is probably to be expected that it may not be as fast as other implementations. After removing as much of the compression as possible, an updated library from the author and some testing with compiler flags, SDSL took around twice as long to build a suffix tree as the former library. This was a manageable difference, however the real performance hit seems to be in traversal of the suffix tree, which is required to generate the p-represenations [5]. In this case, SDSL was 5-10x slower than the previous library. A compromise could be made by creating suffix trees with both libraries: one with SDSL used to perform LCA queries and a second with Tsadko and Yona's library to create the p-representations.

Even with this approach, a comparison between the naïve method could only be made by *excluding the build time*. Results achieved here were still quite limited. Perhaps the most interesting is that reducing the number of frequent characters needed for case 2 to operate would be profitable: it was possible to extract two different patterns of the same size from an English text and using the same text and same value for k, one would not quite have enough frequent characters to fall into case 2 and would run more than 10x slower than the input that had enough frequent characters to use case 2! The likely culprit for this difference is the fact that a large number of matches with FFTs were performed, resulting in high overhead. It should be possible to adjust this threshold to $\alpha \cdot 2\sqrt{k}$ for some $\alpha < 1$ without affecting the time complexity. However, there was not enough time to investigate this. Generally, for larger values of k it was found that if case 2 was used, this was faster than the naïve approach but if case 1 was used, it was much slower than the naïve method.

Secondly, it was found that for small values of k, the run-times of the naïve method and Amir's algorithm were fairly equal, even though for small k, the significantly faster case 2 was almost always used and build time was excluded. Part of the reason for this is that a fairly large number of LCE queries are needed – each LCE query between text and pattern requires up to 3 pattern-pattern LCE queries and we must perform $O(k)$ text-pattern LCE queries. In addition, the constant problems hampered testing such that very little was done with larger inputs – all testing was performed on pattern sizes less than $20,000$. The biggest contributing factor perhaps though was that the naïve method was operating no-where near its worst-case performance;

a way to test both algorithms fairly is really needed.

4 Further Work and Open Questions

During this 10 week project, a number of questions have arisen that there has not been time to answer. The following is a list of further work that I would do if there were more time, in no particular order.

Floating Point Inaccuracies In The FFT

One of the reasons for using number theoretic algorithms in solving the matching with don't cares problem is that the FFT uses floating-point arithmetic and so may suffer from numerical accuracy. It would be nice to try and find any practical implications this has for solving the problem. FFTW's accuracy benchmarks suggest an imprecision which is generally less than 1×10^{-15} for double-precision real data in a 1D transform [8]. This would suggest that accuracy is unlikely to be a problem, even if we are performing hundreds of thousands of transforms (consider the $O(n \log m)$ algorithm with a text size in the millions and a short pattern size – a lot of transforms are performed) and it would certainly require that the error is generally either positive or negative as otherwise errors wouldn't accumulate. However, this observation does ignore the multiplication step required in order to perform a convolution – any errors here could be amplified so it is still not entirely clear how the errors would affect this particular problem.

Does The Abrahamson/Kosaraju Algorithm Have A Useful Application For Matching With Don't Cares?

As was mentioned above, it is possible to modify the algorithm for finding the Hamming Distance at all locations to solve the matching with don't cares problem. Since the Abrahamson/Kosaraju method outperformed the naïve approach for very small pattern sizes, it may be useful for this problem as well. There are some caveats discussed at the end of section 3.4.2 however.

Memory Usage

No attempt to study the memory usage has been made for any of these algorithms and this is clearly an important practical consideration. Generally the absolute ordering of algorithms in terms of memory usage is probably clear. For example, the naïve method in both algorithms requires no additional memory and the $O(n \log m)$ algorithm for solving matching with don't

cares is obviously going to use less than $O(n \log n)$ approach. On the other hand, quantifying the amount of memory used would be a good addition to the project.

False Positives In Monte-Carlo Methods

The randomized algorithm given for solving the matching with don't cares problem when there are no wild cards in the text is not a straightforward implementation of Kalai's method [12], although it does use a broadly similar idea. However, there is a difference in the way the inputs are mapped to random numbers and how the random numbers are exactly used. As an example, Kalai's method does not include an explicit mapping of characters in the alphabet to random values and so allows the possibility of two different characters mapping to the same value. This is not possible in the other method. It would be interesting to investigate both the run-time of these two implementations and the rate of false positives produced. Anecdotally, Kalai's method seemed to produce more false-positives; a much more thorough exploration of this is needed to give any actual results.

More Complete Results For K-Mismatches

It seems likely that the algorithms for solving k-mismatches with small k are impractical compared to the naïve approach, especially since build time of the required data structures had to be excluded to allow any kind of realistic comparison. However, results have been limited and even if this is the case there are still some interesting questions to be answered. For example, it is clear that as k tends to m, the naïve method will become worse, so it would be useful to determine at what point this is.

5 Code Library

One of the aims of this project was to create libraries of code to solve these pattern matching problems. The current release of the library can be found at `www.bsmithers.co.uk/libstringpedia-0.2.tar.gz`. At present, it supports all of the matching with don't care implementations discussed in this report, but only the naive and Abrahamson/Kosaraju methods for solving the k-mismatches problem. It was felt that Amir's method isn't sufficiently well developed to be released at this stage.

The library provides two ways of interfacing with the code: either through a command line 'harness' (which takes input from a text file and produces timing information that can be written to a data file.) or by including the

stringpedia.h header file and linking against libstringpedia.a in your own program. Further documentation and examples can be found by downloading the above file.

There are a number of improvements to the library that would be desirable to make but there has been insufficient time time to complete at this stage, mostly because time is needed in order to understand the relevant tools.

- Use the GNU Build Tools system (e.g. Automake and configure) – currently the install does not provide a mechanism for setting the install location and the option to compile without FLINT is a little messy. A configure script would solve these problems.
- Use Doxygen or similar for improved documentation.
- Include Amir's method is not included.
- Expose the full set of options to the command line harness. Most are available, but not all.
- Provide support in the linked library for loading from files and giving timings.
- Further testing. Code has currently only been tested on Ubuntu 9.04 and 9.10, with a recent version of gcc.

Aside from matching with don't cares and k-mismatches, the library also provides two other features:

- It automates the generation of Wisdom. See generatewisdom.sh or the readme for more details.
- It includes a program for extracting a text and pattern from a given source to be used as an input to the 'harnesses'. It allows text and pattern lengths to be specified and strips out newlines and other special characters.

6 Final Remarks

Overall this project has been an enjoyable experience. My initial aim was for it to help me decide whether or no I wished to persue a PhD after I graduate. Wether or not I'm closer to that decision I'm not sure, but I do feel I have a better understanding of what it may be like. On the otherhand, I don't think

I've entirely experienced the extremes of PhD study: going weeks without really achieving anything or the elation of discovering something new, but I have seen this in a limited sense.

I hope that I can find time in the future to make the improvements to the library, investigate some of the further work and keep StringPedia updated.

Finally, I would like to thank Raphael Clifford for agreeing to supervise this project and also Ben Sach and Markus Jalsenius for all their help over the summer.

References

[1] K. Abrahamson. Generalized string matching, SIAM journal on Computing, 16(6):1039–1051, 1987.

[2] A. Amir, M. Lewenstein, and E. Porat. Faster algorithms for string matching with k mis-matches, Journal of Algorithms, 50(2):257–275, 2004.

[3] P. Clifford, R. Clifford. Simple Deterministic Wildcard Matching, Information Processing Letters 101 (2007) 53-54

[4] R. Clifford. Advanced Algorithms Lecture Notes, Universty of Bristol, http://www.cs.bris.ac.uk/Teaching/Resources/COMSM1402/lect11-2008.pdf

[5] R. Clifford and B. Sach. Psuedo-realtime Pattern Matching: Closing the Gap, CPM 2010, LNCS 6129, pp 101-111

[6] T. H. Cormack, C. E. Leisersen, R. L. Rivest and C. Stein. Introduction to Algorithms, MIT press, 3rd edition

[7] P. Ferragina and G. Navarro. Universities of Pisa and Chile, http://pizzachili.dcc.uchile.cl/

[8] M. Frigo and S. G. Johnson. (FFTW) Fastest Fourier Transform in the West, Massachusetts Institute of Technology, http://www.fftw.org/accuracy/index.html

[9] S. Gog. Succint Data Structure Library, Ulm University , Germany http://www.uni-ulm.de/in/theo/research/sdsl

[10] D. Gusfield. Algorithms on Strings, Trees and Sequences, Cambridge University Press, 1997, Section 2 pp 87-207

[11] W. B. Hart. Fast Library for Number Theory: an introduction, http://sage.math.washington.edu/home/wbhart/flint-extended-abstract.pdf

[12] A. Kalai. Efficient Pattern-Matching With Don't Cares, Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 655-656

[13] S. R. Kosaraju. Efficient string matching, Manuscript, 1987.

[14] D. Tsadok and S. Yona. University of Haifa, Israel, http://mila.cs.technion.ac.il/ yona/suffix_tree/