A Practical Index for Genome Searching

Heikki Hyyrö[1]* and Gonzalo Navarro[2]**

[1] Dept. of Comp. and Inf. Sciences, Univ. of Tampere, Finland. helmu@cs.uta.fi
[2] Dept. of Comp. Science, Univ. of Chile. gnavarro@dcc.uchile.cl

Abstract. Current search tools for computational biology trade efficiency for precision, losing many relevant matches. We push in the direction of obtaining maximum efficiency from an indexing scheme that does not lose any relevant match. We show that it is feasible to search the human genome efficiently on an average desktop computer.

1 Introduction

Approximate string matching [5] is a recurrent problem in many branches of computer science, with important applications to computational biology. Efficiency is crucial to handle the large databases that are emerging, so indexes are built on the text to speed up queries later [12, 8]. Although there exist several indexed search tools like BLAST and FASTA, these usually trade time for precision, losing many relevant answers [12]. In this paper we aim at building a fast index that does not lose any answer. We combine and optimize the best existing previous lossless approaches [3, 7] and focus on the simplified case of DNA search using Levenshtein distance. This case is important in the current stage of analyzing gene functionality once the genome projects are completing their first task of obtaining the DNA sequences. In particular, approximate searching in genomes is necessary to identify homologous regions, which is fundamental to predict evolutionary history, biochemical function, and chemical structure [12].

Our main result is a practical product that can be used to search the human genome on an average desktop computer. Unique features of our index are: optimized selection of pattern pieces, bidirectional text verification, and optimized piece neighborhood generation. Our tools can be generalized to more complex problem such as weighted edit distances.

2 Indexed Approximate String Matching

The problem we focus on is: Given a long text $T_{1...n}$, and a (comparatively) short pattern $P_{1...m}$, both sequences over alphabet Σ of size σ, retrieve all substrings of T ("occurrences") whose *edit distance* to P is at most k. The edit distance, $ed(A, B)$, is the minimum number of "errors" (character insertions, deletions and substitutions) needed to convert one string into the other. So we permit an "error level" of $\alpha = k/m$ in the occurrences of P.

* Supported by the Academy of Finland and Tampere Graduate School in Information Science and Engineering. ** Partially supported by Fondecyt Project 1-020831.

The most successful approach to indexed approximate string matching [8] is called *intermediate partitioning* [3, 7]. It reduces the approximate search of P to approximate search of substrings of P. Their main principle is that, if P matches a substring of T, j disjoint substrings are taken from P, then at least one of these appears in the occurrence with at most $\lfloor k/j \rfloor$ errors. These indexes split P into j pieces, search the index for each piece allowing $\lfloor k/j \rfloor$ errors, and finally check whether the piece occurrences can be extended to occurrences of P. The index is designed for exact searching of pieces, so approximate searching is handled by generating the "d-neighborhood" of each piece S, $U_d(S) = \{S' \in \Sigma^*, ed(S, S') \leq d\}$, and searching the index for each $S' \in U_d(S)$.

In [3] all the text q-grams (substrings of length q), where $q = \lceil \log_\sigma n \rceil$, are stored together with their text positions. Then the pattern is recursively split into 2 or 3 pieces at each level (dividing also the number of errors permitted), until the final pieces are short enough to be searchable with the index (Fig. 1). The paper is not very explicit on how the partitioning is exactly done.

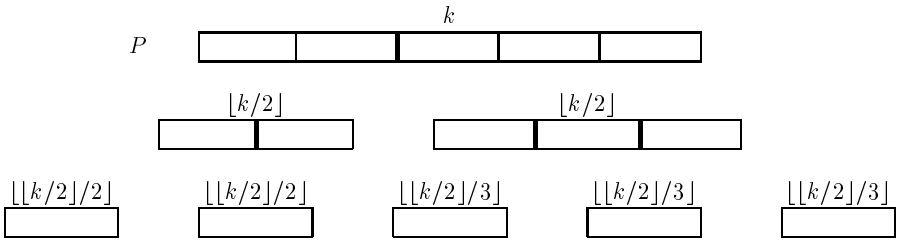

Fig. 1. The pattern is recursively split into smaller and smaller pieces, also dividing the number of errors. Above each piece we show the number of errors we permit for it.

Assume that a bottom-level piece P^i is to be searched with d_i errors. Its occurrences are found by generating its *condensed d_i-neighborhood* $UC_{d_i}(P^i)$: $A \in UC_d(B)$ iff $A \in U_d(B)$ and $A' \notin U_d(B)$ for any A' prefix of A. Any occurrence of P^i with d_i errors errors must have a prefix in $UC_{d_i}(P^i)$. Then, all these occurrences are located fast by searching the q-gram index for each string in $UC_{d_i}(P^i)$. These occurrences are then extended by going up the splitting hierarchy in stepwise manner. Each step consists of merging pieces back together and checking, with dynamic programming, whether the merged piece occurs in the text with its permitted error threshold. This recursive process is continued until either some internal node cannot be found, or we find the whole pattern.

In [7], a suffix array [2] is used instead of a q-gram index, so it can choose the partition according to optimization goals rather than guided by the constraint on the final piece lengths. They show that the optimum is $j = O(m/\log_\sigma n)$. Other differences are that text verification uses an efficient bit-parallel algorithm instead of dynamic programming, and that hierarchical verification is not used.

3 Our Proposal

The design of our index is based on the following four assumptions: (1) The indexed text is a DNA sequence. (2) The whole text is available in primary

memory. (3) The index has to work efficiently on secondary memory. (4) The error level α is typically < 0.25.

The first assumption means that the alphabet size is small, $\sigma = 4$, so we can store each nucleotide in 2 bits and hence store the text in $n/4$ bytes. This permits storing the human genome in about 750 MB, which makes the second assumption more realistic in the case of the human genome. This assumption is important when evaluating the cost of accessing the text at piece verification time. The third assumption arises when one considers that the most efficient indexes take a significant amount of space, and it might not be realistic to assume that also the index will fit in main memory. Thus the index should have a suitable structure for secondary memory. The fourth assumption is based on the search parameters used in real computational biology applications. It is also very convenient because no index works well for higher α values if $\sigma = 4$.

Like [3], we use a q-gram index, d-neighborhood generation and hierarchical verification. However, we take some elements of [7] such as optimizing pattern partitioning and piece verification. We also consider secondary memory issues.

3.1 Index Structure

Our q-gram index is almost identical to that of Myers. Each q-gram is coded as a base-4 number (e.g., "agct" $\rightarrow 0321_4$). The index has two tables, the header table and the occurrence location table. The header table H_q contains, for each q-gram, the start position of the interval in the location table L_q, which holds in ascending order all the locations of the q-gram in the text. The location table L_q holds the intervals of locations consecutively in increasing order numerical representation. Hence, the occurrences of the q-gram with numerical value x are located in $L_q[H_q[x] \ldots H_q[x+1]-1]$.

The value of q affects the length of the pattern pieces that can be efficiently retrieved with the index. Having a large q is only a problem if the size of table H_q, $O(\sigma^q)$, becomes an issue. This is because a q-gram index can be used also in finding shorter substrings. The locations of the $(q-c)$-gram with numerical representation x are those in the interval $L_q[H_q[x\sigma^c] \ldots H_q[(x+1)\sigma^c]-1]$. This corresponds to all q-grams having the given $(q-c)$-gram as a prefix. On the other hand, a small q may significantly degrade the performance.

Using Myers' setting $q = \lceil \log_\sigma n \rceil$ would result in the value $q = 16$ when indexing the human genome. This would result in a huge header table. Even though the index can be in secondary memory, we prefer to keep the header table in main memory (see Sec. 3.5). Hence we have opted to use $q = 12$, which results in a header table of 67 MB. With the 3 billion nucleotides human genome, the location table is roughly 12 GB, since we use 32-bit integers for all entries. It is straightforward to build this index in $O(n + \sigma^q)$ time and space.

3.2 Optimizing the Intermediate Partitioning

We employ a hierarchical partitioning that differs from [3] in that it is done bottom-up. We first determine the pieces and then build up the hierarchy. The top-down partitioning (Fig. 1) has less control over which are the final pieces.

Previous partitioning methods have assigned $d_i = \lfloor k/j \rfloor$ errors to each piece when the pattern P was partitioned into j pieces $P^1, \ldots, P^j$. However, in [8] a more accurate rule was proposed. If a string A contains no pattern piece P^i with d_i errors, then $ed(A,P) \geq \sum_{i=1}^{j}(d_i+1) = \sum_{i=1}^{j} d_i + j$, as each piece P^i needs at least $d_i + 1$ errors to match. So we must have $\sum_{i=1}^{j} d_i + j \geq k+1$ to ensure that no approximate occurrence of P is missed, which can be rephrased as the condition $\sum_{i=1}^{j} d_i \geq (k+1) - j$. Naturally the best choice is to allow the fewest possible errors, and thus we use the strict requirement $\sum_{i=1}^{j} d_i = k - j + 1$.

Since we have a q-gram index, we partition the pattern into pieces of length at most q. We also fix an upper bound d_M on the d_i values (see later).

We have tested two partitioning methods. A simple scheme, similar in nature to previous methods, is to partition the pattern into $j = \lceil k/d_M \rceil$ pieces, the minimum yielding $d_i \leq d_M$. Then, the pattern is split into j pieces of lengths $\lfloor m/j \rfloor$ or $\lceil m/j \rceil$, pruning pieces that are longer than q. To enforce the strict error limit $\sum_{i=1}^{j} d_i = k - j + 1$, we set $d_i = \lfloor k/j \rfloor$ for (m modulo j) $+ 1$ pieces (giving preference to the longest pieces), and $d_i = \lfloor k/j \rfloor - 1$ for the rest.

The second, more sophisticated, approach is to precompute and store for each r-gram x, $r \in 1 \ldots q$, and for each $d \in 0 \ldots \min(d_M, \lceil 0.25 \times r \rceil - 1)$, the number of text occurrences of all the r-grams in the d-neighborhood of x. This value, $C_{x,d}$, is used to find the optimal splitting. Let us define $M_{i,t}$ as the minimum number of text positions to verify in order to search for $P_{i \ldots m}$ with t errors. Then the following recurrence holds:

$$M_{i,t} = 0, \text{ if } t < 0; \qquad M_{i,t} = \infty, \text{ if } i > m \wedge t \geq 0;$$
$$M_{i,t} = \min(M_{i+1,t}, \min_{d \in 0 \ldots \min(t, d_M), r \in 1 \ldots q}(C_{P_{i \ldots i+r-1}, d} + M_{i+r, t-d-1})), \text{ otherwise.}$$

so the minimum possible verification cost is $M_{1,k}$, and we can easily retrieve from M the optimal partitioning reaching it. Once the values $C_{x,d}$ are precomputed (at indexing time), the above algorithm adds $O(qmk^2)$ to the search time, which is rather modest compared to the work it saves.

Precomputing $C_{x,d}$ is not prohibitively slow. What is more relevant is the amount of memory necessary to store $C_{x,d}$. Since the information for $d = 0$ has to be kept anyway (because it is the length of the list of occurrences of x, and it is known also for every $r \leq q$), the price is $d_M - 1$ more numbers for each different r-gram. A way to alleviate this is to use fewer bits than necessary and reduce the precision of the numbers stored, since even an approximation of the true values will be enough to choose an almost optimal strategy.

We form a hierarchy on the pattern pieces similar to that of Myers (Fig. 1). However, as we begin by optimizing the pieces at the lowest level, we form the hierarchy in bottom-up order.

Let j_h be the number of pieces and $P^{i,h}$ the ith piece at the hth level of the hierarchy. Also let $d_{i,h}$ be the number of errors associated to piece $P^{i,h}$. The top level corresponds to the whole P at the root, so $j_1 = 1$, $P^{1,1} = P$ and $d_{1,1} = k$. Assume that our optimized splitting leads to an ℓth level partitioning with j_ℓ pieces $P^{1,\ell}, \ldots, P^{j_\ell,\ell}$. In general the $(h-1)$th level is formed by pairing together

two adjacent pieces from the hth level, $P^{i,h-1} = P^{2i-1,h}P^{2i,h}$. If j_h is odd, the last piece will be added to the last pair, $P^{j_{h-1},h-1} = P^{2j_{h-1}-1,h}P^{2j_{h-1},h}P^{2j_{h-1}+1,h}$. We will always have $j_{h-1} = \lfloor j_h/2 \rfloor$. This is continued until we reach level 1.

The number of errors for piece $P^{i,h-1}$ is found by locally enforcing the rule $\sum_{i=1}^{j} d_i = k-j+1$. For the piece $P^{i,h-1}$, this means $d_{2i-1,h}+d_{2i,h} = d_{i,h-1}-2+1$, which defines $d_{i,h-1}$. If piece $P^{i,h-1}$ is formed by joining three pieces, then we have $d_{i,h-1} = d_{2i-1,h} + d_{2i,h} + d_{2i+1,h} + 2$. Although the lowest level pieces may not cover P, upper level pieces are stretched to cover P. This reduces the probability of finding them in the text.

3.3 Generating d-neighborhoods

We also use a different way of generating d-neighborhoods. Given a string A, instead of computing Myers' condensed d-neighborhood $UC_d(A)$, we compute a "length-q artificial prefix-stripped" d-neighborhood $UP_d(A)$. This is done by collecting all different strings that result from applying d errors into A in all possible combinations, with the following restrictions: (1) Errors are applied only within the window of the first q characters. (2) A character is only substituted by a different character. (3) No characters are inserted before or after the first or the last character. (4) The string is aligned to the left of the length-q window. That is, characters to the right of a deletion/insertion are moved one position to the left/right. (5) A character introduced by an insertion or substitution is not further deleted or substituted.

In practice we have noted that $UP_d(A)$ is often slightly smaller than $UC_d(A)$. For example, if $A =$ "atcg" and $d = 1$, the strings "aatcg", "tatcg", "catcg" and "gatcg" belong to $UC_d(A)$, but of these only "aatcg" belongs to $UP_d(A)$. But there are also strings in $UP_d(A)$ and not in $UC_d(A)$. For example if $B =$ "attaa" and $d = 2$, then "ataaa" is in $UP_d(A)$ but not in $UC_d(A)$, as its prefix "ataa" is in $UC_d(A)$. However, also Myers' index will fetch q-grams with prefix "ataaa" if $q \geq 5$.

The set $UP_d(A) \subseteq U_d(A)$ can be built in $O((3q\sigma)^d)$ time [11]. In our experiments with $d \leq 2$, our d-neighborhood generation was twice as fast as Myers'.

3.4 Fast Verification

In [3] they used dynamic programming approximate string matching algorithm in the stepwise merging/checking process. They also grouped into a single interval piece occurrences that were close to each other, so as to process the whole interval in a single pass and avoid checking the same text multiple times. In [7] they used a faster bit-parallel algorithm, but a more crude approach: they searched the text between the positions $j-m-k \ldots j+m+k$ whenever a piece occurrence ended at text position j. They also merged checking of adjacent occurrences.

We check each piece occurrence separately on the bottom-level of the hierarchy. We use a bit-parallel algorithm for computing edit distance [1] instead of approximate string matching. This method [6] was much faster than previous ones (Sec. 4). On the upper levels we use interval merging and a bit-parallel approximate string matching algorithm [4].

The bottom-level verification works as follows. Let $P^i = P_{i...i+b}$ be a pattern piece, and let $A \in UP_d(P^i)$ occur starting from T_j. Also let substring $P^f = P_{i-u...i+v}$ be the "parent" of P^i in the hierarchy, so P^f contains P^i. Initially we set $d = d_f + 1$, where d_f is the number of errors for P^f. Value d will be the number of errors in the best match for P^f found so far. If P^i is not the rightmost piece in P^f, then $ed(T_{j...j+a}, P_{i...i+v})$ is computed for $a = 0, 1, 2, \ldots$ until either $ed(T_{j...j+a}, P_{i...i+c}) \geq d$ for all $c \in [1 \ldots v]$, or we obtain $ed(T_{j...j+a}, P_{i...i+v}) = 0$. Whenever a value $ed(T_{j...j+a}, P_{i...i+v}) = d - 1$ is found, we set $d = d - 1$. This forward edit distance computation will process at most $v + d_f + 2$ characters, as after that the first stopping condition must be true. If $d = d_f + 1$ after stopping, we know that P^f does not occur. If $d \leq d_f$, we start computing the edit distance $ed(T_{j-a...j-1}, P_{i-u...i-1})$ for $a = 1, 2, \ldots$ similarly as above, starting with $d = d_f - d + 1$ and this time stopping as soon as $ed(T_{j-a...j-1}, P_{i-u...i-1}) < d$, since then we have found an occurrence of P^f with at most d_f errors.

3.5 Secondary Memory Issues

We discuss now how to handle indexes that do not fit in main memory. The biggest disadvantage of secondary memory is slow seek time. That is, although data transfer times are acceptable, performance worsens significantly if the data is not read from a moderate number of continuous locations. When using our q-gram index, queries will typically access more or less scattered positions of table L_q. When d-neighborhood generation is used, the number of q-gram lists fetched, and hence seek operations over L_d, grows exponentially with d. To limit this effect, we use bound d_M, the maximum d value. Based on practical experience we have chosen limit $d_M = 1$ in secondary memory scenarios. We also store the header table H_q in main memory to avoid an extra seek operation per q-gram. Hence the need to use a moderate q so that H_q fits in main memory.

The effects of secondary memory can also be considered when choosing the partitioning. We can weight the value $C_{x,d}$ of the occurrence table (Sec. 3.2) with an estimated cost for querying the q-gram index with the strings in $UP_d(x)$. If $C^w_{x,d}$ is the weighted cost for substring x and d errors, we use the formula

$$\begin{aligned} C^w_{x,d} &= C_{x,d} \times (\textit{verification-cost} + \textit{disk-transfer-cost}) \\ &+ \textit{d-neighborhood-size}(x,d) \times \textit{disk-seek-cost} \end{aligned}$$

normalized to the form $C_{x,d} + c \times \textit{d-neighborhood-size}(x,d)$. The weight value c depends on the actual type of memory used in storing the index, and thus it should be based on empirical tests.

4 Test Results

As the test results in [7] found the index of Myers to be the best method in the case of DNA, we have compared our performance against that index. The implementation of Myers' index, from the original author, is only a limited prototype constrained to pattern lengths of the form $q \times 2^x$ and $q = \lceil \log_\sigma n \rceil$.

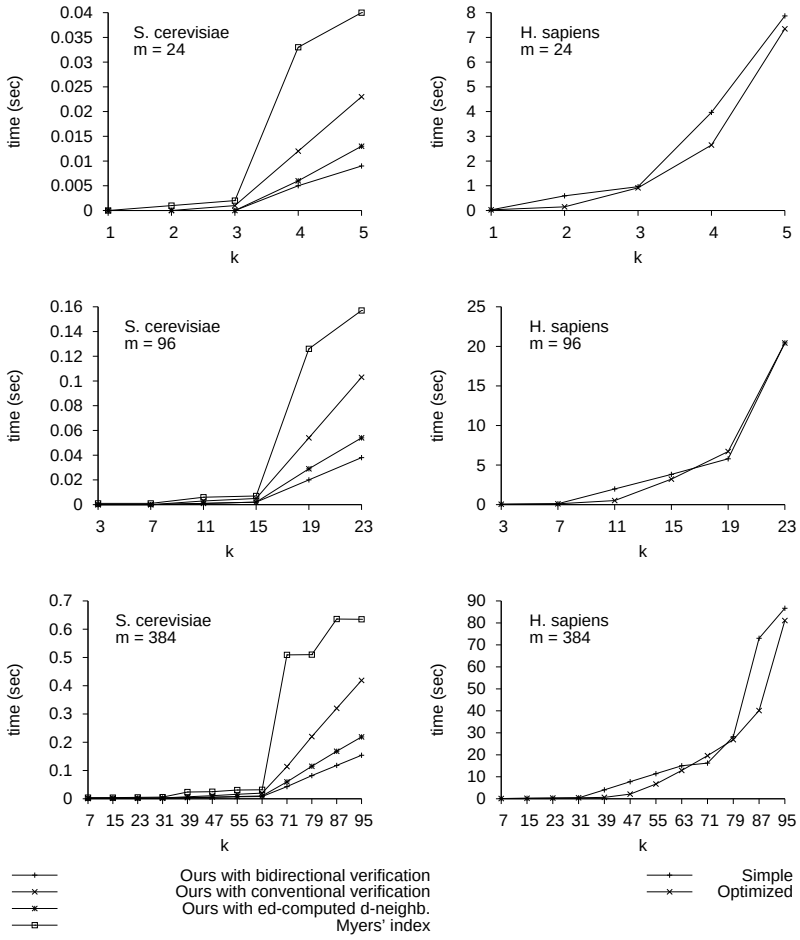

Fig. 2. On the left, Myers' index versus three variants of our index, in main memory, searching the $\approx$ 10 MB genome of *S. cerevisiae* (baker's yeast) [9]. Our variants use simple partitioning (Sec. 3.2) and $d_M = 2$. The first method uses bidirectional verification and the second conventional interval-merging combined with approximate string matching. Both of these use the d-neighborhood generation method of Sec. 3.3. The third method uses bidirectional verification combined with a d-neighborhood generation method closer to Myers' (backtracking with edit distance computation over the trie of all strings). We run on a P3 600 Mhz with 256 MB RAM and Linux OS, and compile with GCC 3.2.1 using full optimization. On the right, simple versus optimized partitioning for our index (Sec. 3.2). We use the best combination of verification/d-neighborhood generation from the tests on the left. Now the index is on disk, we use $d_M = 1$ and encode the text using 2 bits per nucleotide. The text is the Aug 8th 2001 draft of the human genome [10], of about 2.85 billion nucleotides. We run on an AMD Athlon XP 1.33 Ghz with 1 GB RAM, 40 GB IBM Deskstar 60GXP hard disk and Windows 2000 OS, and compile using Microsoft Visual C++ 6.0 with full optimization.

Fig. 2 (left) shows the results when searching the small *S. cervisiae* genome, where the index fits in main memory. We test three variants of our index, among which the clear winner is bidirectional verification of bottom-level pieces combined with our d-neighborhood generation. This is 2 to 12 (typically above 4) times faster than Myers' index. In many cases a large part of our advantage is explained by the strict rule $\sum_{i=1}^{j} d_i = k - j + 1$. This is more clear in the plots when k goes above $m/6$: at this point the index of Myers sets $d_i = 2$ for all the pieces, whereas our index increases the number of errors in a more steady manner. The difference between the search mechanisms themselves is seen when $k = m/6 - 1$ or $k = m/4 - 1$, as at these points both indexes set $d_i = 1$ or $d_i = 2$, respectively, for all the pieces. In these cases our fastest version is always roughly 4 times faster than Myers' index.

Our best combination from the above test was used for searching the human genome, where the index is on disk. We compared simple and optimized partitioning. As shown in Fig. 2 (right), in most cases using optimized partitioning had a non-negative gain, in the range 0-300%. There were also some cases where the effect was negative, but they were most probably due to the still immature calibration of our cost function. We also made a quick test to compare our disk-based index with the sequential bit-parallel approximate string matching algorithm of Myers [4]. For example in the case $m = 384$ and $k = 95$ our index was still about 6 times faster.

References

1. H. Hyyrö. A bit-vector algorithm for computing Levenshtein and Damerau edit distances. *Nordic Journal of Computing*, 10:1–11, 2003.
2. U. Manber and E. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, pages 935–948, 1993.
3. E. Myers. A sublinear algorithm for approximate keyword searching. *Algorithmica*, 12(4/5):345–374, Oct/Nov 1994.
4. G. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic progamming. *Journal of the ACM*, 46(3):395–415, 1999.
5. G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.
6. G. Navarro. NR-grep: a fast and flexible pattern matching tool. *Software Practice and Experience*, 31:1265–1312, 2001.
7. G. Navarro and R. Baeza-Yates. A hybrid indexing method for approximate string matching. *Journal of Discrete Algorithms (JDA)*, 1(1):205–239, 2000.
8. G. Navarro, R. Baeza-Yates, E. Sutinen, and J. Tarhio. Indexing methods for approximate string matching. *IEEE Data Engineering Bulletin*, 24(4):19–27, 2001.
9. National center for biotechnology information. `http://www.ncbi.nlm.nih.gov/`.
10. Ucsc human genome project working draft. `http://genome.cse.ucsc.edu/`.
11. Esko Ukkonen. Finding approximate patterns in strings. *J. of Algorithms*, 6:132–137, 1985.
12. H.E. Williams and J. Zobel. Indexing and retrieval for genomic databases. *IEEE Trans. on Knowledge and Data Engineering*, 14(1):63–78, 2002.