JOURNAL OF COMPUTATIONAL BIOLOGY
Volume 17, Number 10, 2010
© Mary Ann Liebert, Inc.
Pp. 1279–1293
DOI: 10.1089/cmb.2010.0005

Error Tolerant Indexing and Alignment of Short Reads with Covering Template Families

*ELDAR GILADI,[1] *JOHN HEALY,[1] GENE MYERS,[2] CHRIS HART,[3] PHILIPP KAPRANOV,[1] DORON LIPSON,[1] STEVE ROELS,[1] EDWARD THAYER,[4] and STAN LETOVSKY[1]

ABSTRACT

The rapid adoption of high-throughput next generation sequence data in biological research is presenting a major challenge for sequence alignment tools—specifically, the efficient alignment of vast amounts of short reads to large references in the presence of differences arising from sequencing errors and biological sequence variations. To address this challenge, we developed a short read aligner for high-throughput sequencer data that is tolerant of errors or mutations of all types—namely, substitutions, deletions, and insertions. The aligner utilizes a multi-stage approach in which template-based indexing is used to identify candidate regions for alignment with dynamic programming. A template is a pair of gapped seeds, with one used with the read and one used with the reference. In this article, we focus on the development of template families that yield error-tolerant indexing up to a given error-budget. A general algorithm for finding those families is presented, and a recursive construction that creates families with higher error tolerance from ones with a lower error tolerance is developed.

Key words: filtration, gapped q-gram, gapped seed, multiple spaced seeds, sequence similarity, string matching.

1. INTRODUCTION

THE NEW GENERATION OF HIGH-THROUGHPUT SEQUENCERS—including Helicos™' HeliScope single molecule sequencing platform (Harris et al., 2008; Lipson et al., 2009), Illumina's Genome Analyzer, and Life Technologies' SoLID—are significantly accelerating the investigation of many biological questions. Applications that have already benefited from these technologies include expression profiling (Lipson et al., 2009), the study of genetic variations between individuals (Wang et al., 2008), and the study of DNA protein interactions (Johnson et al., 2007). In addition, these technologies enable massive re-sequencing endeavors such as the 1000-Genomes project (Kaiser, 2008).

[1]Helicos BioSciences Corporation, Cambridge, Massachusetts.
[2]Janelia Farm Research Campus, Ashburn, Virginia.
[3]Division of Natural Sciences, New College of Florida, Sarasota, Florida.
[4]Microsoft Health Solutions, Seattle, Washington.
*These two authors contributed equally.

For re-sequencing and counting applications, the first step in the analysis is the correct placement of the DNA sequence reads on a reference genome or transcriptome. This is a challenging step due to the need to provide sensitivity in the presence of errors and sequence variations of all types (insertion, deletions, and substitutions) in reads as short as 25 nucleotides, and the need to rapidly align a very large number of reads, up to 10^9 per run for a HeliScope. Furthermore, the different short read technologies have very different error profiles, with substitutions being most common in the Illumina system, while deletions and insertions are the dominant error type in single molecule sequencing. Hence, in order to achieve an accurate, cost-effective and timely analysis, the aligner should be capable of aligning massive amounts of short reads with errors or sequence variations of all types to large genomic references.

In this article, we present a high-throughput alignment tool for effectively aligning variable length short reads that contain mutations or errors of all types. Our aligner has three stages: a novel error-tolerant indexing stage for seeding alignments, an edit distance (Myers, 1999) based filter for filtering out non-promising candidate locations, and finally a full dynamic programming alignment. Many aligners use a similar multi-stage strategy, including BLAST, BLASTZ, and BLAT (Altschul et al., 1997; Kent, 2002; Schwartz et al., 2003); and in the short read arena, SHRIMP and Mosaik (Rumble et al., 2009; Stromberg, 2009).

This article focuses on the seeding stage in which the portions of the reference that are most similar to a read are identified. Seeding alignments is generally achieved by finding small patterns of nucleotides, called ''seeds,'' shared exactly by the read and the reference. This process is fast and can be accelerated using a lookup table. Seeding is generally followed by an expensive alignment step, so reducing the portion of the reference considered in the second stage greatly reduces the overall computational cost. In *lossless* seeding, there is a guarantee that all regions of the reference within a certain error budget from a fixed size window of the query are found. In the *lossy* case, many but not all such regions are found.

In standard seeding with *k*-mers, the patterns are contiguous stretches of *k* nucleotides called ''words'' or ''*k*-mers.'' This approach is subject to the following tradeoff: short *k*-mers yield good sensitivity at the expense of lower specificity (more false positive locations) and longer computation time. Long *k*-mers increase specificity but reduce sensitivity. An increase in specificity with only a small reduction in sensitivity can also be achieved by requiring that each read have multiple short *k*-mer matches in the same reference locus. Aligners that use this approach include BLAST (Altschul et al., 1997), BLAT (Kent, 2002), and Mosaik (Stromberg, 2009) for short reads. Another extension to the contiguous seed model is found in BLAT (Kent, 2002), where each seed is allowed to contain a mismatch in at most one location.

Substantial increases in sensitivity with the same specificity can be achieved by using gapped words or gapped-seeds. Here, the common pattern between a read and a reference consists of a subset of nucleotides in a word, with mismatches allowed in the intermediate positions. For example, if only the first, fifth, and tenth nucleotide from a 10-mer are used, those nucleotides are concatenated into a 3-mer word, which is then compared to a reference's set of words generated by the same pattern. This approach has been widely studied in both the lossy (Buhler, 2001; Buhler et al., 2003; Burkhardt and Kärkkäinen, 2002; Califano and Rigoutsos, 1993; Choi and Zhang, 2004; Keich et al., 2004; Kucherov et al., 2004; Li et al., 2004; Ma et al., 2002; Noé and Kucherov, 2004; Sun and Buhler, 2004; Xu et al., 2004) and the lossless case (Burkhardt and Kärkkäinen, 2003; Kucherov et al., 2005; Pevzner and Waterman, 1995). Aligners that use gapped-seeds include FLASH (Califano and Rigoutsos, 1993), patternHunter (Ma et al., 2002), and blastZ (Schwartz et al., 2003), and the short read aligners include SHRIMP (Rumble et al., 2009) and MAQ (Li et al., 2008). MAQ use pairs of gapped seeds.

PatternHunter (Ma et al., 2002) uses a single seed with optimal sensitivity. The optimization is for a fixed specificity and a uniform probability model for substitutions. Algorithmic solutions to the problem of finding an optimal seed are presented in Choi and Zhang (2004) and Keich et al. (2004).

Multiple gapped seeds improve the sensitivity-specificity tradeoff even further. FLASH (Califano and Rigoutsos, 1993) uses multiple randomly chosen gapped seeds, and a similar approach was taken in LHS-ALL_PAIRS (Buhler, 2001). Algorithms for designing sensitive sets of gapped seeds are presented in Buhler et al. (2003), Li et al. (2004), Sun and Buhler (2004), and Xu et al. (2004). Computing the seeds assuming a Markov alignment model was considered in Buhler et al. (2003) and Sun and Buhler (2004). Extensions that enable protein similarity searches are presented in Brejova et al. (2003) and Brown (2004).

With the exception of the work of Burkhardt and Kärkkäinen (2002), prior approaches are useful only for finding regions of un-gapped similarity. Specifically, the gapped seed approach does not generate a match when insertions or deletions occur inside a seed, even in positions where a mismatch is allowed.

The seeding we develop here is both *lossless* and capable of finding *gapped similarities*. This is achieved by using an extension to the multiple gapped seed approach that we call *covering template families*. Here, a *template* is a pair of gapped seeds: one associated with the read and the other with the reference. A template generates a match if the read seed from the query matches a reference seed at some reference position. The gapped seeds discussed in Burkhardt and Kärkkäinen (2002) are equivalent to templates for the one gap case.

Covering template families guarantee that, for a given error budget, at least one template in the family generates a match between a fixed size sequence and another at *edit distance* no greater than the error budget. The edit distance is the minimal number of insertions, deletions, or substitutions between two sequences. This approach has "perfect sensitivity" (sensitivity is equal to 1) within the error budget, without sacrificing specificity. Covering sets of gapped seeds for *un-gapped similarity* were computed in Xu et al. (2004).

Here we present several approaches for developing covering template families. First a greedy exhaustive algorithm is presented. Then we present a recursive construction that creates families with higher error tolerance from ones with lower error tolerance.

The article is organized as follows. In Section 2.1, we introduce definitions of key concepts. In Section 2.2, we describe the process of indexing with covering template families. Section 2.3 describes the exhaustive greedy algorithm for finding covering template families, and indicates some properties of those suited to common usage. In Section 2.3, we present an explicit construction that leverages a family with a certain error budget to create a family with a higher error budget. Section 3 includes various results.

2. METHODS

2.1. Definitions

In this section, we introduce several definitions, with the ultimate goal of introducing the concept of *covering template families*.

2.1.1. Gapped words. A *word S* of size N is a contiguous stretch of N nucleotides, N is the *word size*. A *key* of weight w is a vector of w distinct non-negative integers sorted in increasing order Keys are used to define a subsequence pattern within an arbitrary sequence. The *key size* is one plus the maximum integer in the key. Note that *0* based indexing is used throughout the article.

We denote the length of a sequence S by $L(S)$, and given sequences S_1 and S_2, we denote their concatenation by $S_1 + S_2$.

Let $K = (k_0, \ldots, k_{w-1})$ be a key of weight w. We denote by $K' = m + K$ the key $K' = (k_0 + m, \ldots, k_{w-1} + m)$.

For a sequence $S = (S_0, \ldots, S_{M-1})$ and a key K of weight w and size k such that $L(S) \geq k$ we denote by $S[K]$ the sequence obtained by concatenating the w nucleotides $S_{K[0]}, \ldots, S_{K[w-1]}$. We call it the *gapped word* corresponding to S and K.

2.1.2. Templates and template families. In order to develop an indexing scheme that is tolerant of both substitutions and indels, gapped words are not sufficient. What is required is the *template* construct, which is essentially a pair of keys: one used with the query and one with the reference.

Definition: Given two keys K_1 and K_2 of same weight w, the pair $T = (K_1, K_2)$ is called a *template*. K_1 is called the *reference key* and denoted by *T.referenceKey*, while K_2 is called the *query key* and is denoted by *T.queryKey*.

Definition: A collection of M templates $\boldsymbol{F} = \{T_1, \ldots, T_M\}$ of same weight w, maximum reference key size N and maximum read key size f is an (N, w, f) *template family*.

Definition: An *e-error instance E* derived from a word S is the sequence of nucleotides obtained by introducing e errors (insertions deletions or substitutions) into S.

Definition: Let E be an e-error instance derived from a word S. We say that a template T *induces a match* between E and S provided $E[T.queryKey] = S[T.referenceKey]$. Here, T has reference key size no greater than $L(S)$ and read key size no greater than $L(E)$.

Definition: An (N,w,f) template family $\boldsymbol{F} = \{T_1, \ldots, T_M\}$ is an (N, w, f, e) *covering template family* provided that for any n-error instance E with $n \leq e$ derived from a word S of length N there is $T \in \boldsymbol{F}$ that induces a match between E and S.

2.2. Indexing with covering template families

In this section, we motivate the use of covering template families for error tolerant indexing, and explain the actual algorithm for indexing with them. We also highlight computational complexity issues and tradeoffs associated with the use of template families.

2.2.1. Template indexing algorithm. Consider a read D sequenced from a known reference R in the presence of sequencing errors and/or mutations. We assume that the prefix E of D was derived from a word S of size N in R by introducing e errors and that $L(E) = f \leq N + e$. The goal of the indexing algorithm is to discover the match between D and R in order to seed an alignment between them.

Any (N, w, f, e) covering template family $\boldsymbol{F}$ has a template $T^* \in \boldsymbol{F}$ for which $E[T^*.queryKey] = S[T^*.referenceKey]$. If S begins at position m of R, then

$$D[T^*.queryKey] = R[m + T^*.referenceKey] \tag{1}$$

This match between a small section of the read and a small section of the reference is found despite the presence of e errors or mutations in that section.

The template T^* and m required to find the match are unknown. Therefore, equation (1) should be tested for all members of $\boldsymbol{F}$ and all possible values of m. In addition, to increase the likelihood that a match be found, $T^*.queryKey$ can be applied in multiple positions of the query. This yields the basic template indexing algorithm of Figure 1.

The computational complexity of the indexing algorithm of Figure 1 is $O(|\boldsymbol{F}|(1 + \lfloor (L(D) - f)/\Delta \rfloor)\,|R|)$, where each term results from a loop in the algorithm. This suggests features of $\boldsymbol{F}$ and implementation details that will reduce the computational complexity and actual run time.

The loop in line A yields the $|\boldsymbol{F}|$ term and it reveals that the smallest possible covering template family for a given error tolerance e, is desirable.

The loop in line C yields the $|R|$ term and can be avoided by using an index lookup. Specifically, an index of all w-mers $R[m + T.referenceKey]$, $0 \leq m \leq L(R) - N$, is computed for each $T \in \boldsymbol{F}$. If indexes are precomputed the computational complexity becomes $O(|\boldsymbol{F}|(1 + (L(D) - f)/\Delta))$. Fewer indexes are required if multiple templates share the same reference key, another desirable feature of $\boldsymbol{F}$.

The sensitivity of the algorithm is increased at the expense of run-time when a large number of positions in the read are probed. The step-size Δ on line B controls this tradeoff.

Note that conventional indexing with k-mers is a special case of the algorithm of Figure 1 with $\boldsymbol{F} = \{T : T.queryKey = T.referenceKey = (0, \ldots, k-1)\}$.

2.2.2. Sensitivity and specificity. It is useful to compare the sensitivity and specificity of indexing with k-mers to that of indexing with an (N,w,f,e) covering template family $\boldsymbol{F}$. To that end, we introduce two functions $k(\boldsymbol{F})$ and $K(\boldsymbol{F})$. The former is the largest k-mer size that yields the same error budget guarantee as $\boldsymbol{F}$ in k-mer indexing, while the later is the k-mer size that yields the same specificity as $\boldsymbol{F}$. We also introduce the set:

```
Foreach T ∈ F {                                  // Line A
  For (n=0; n < L(D)-f; n=n+Δ) {                 // Line B
    For (m=0; m < L(R)-N; m++){                  // Line C
      If (D[n+T.readKey] = R[m+T.referenceKey]) {
        Match found between D and R at (m ,n)
      }
    }
  }
}
```

FIG. 1. Basic template indexing algorithm.

$$\boldsymbol{I} = \{E | E \text{ is an } e\text{-error instance obtained from } S, L(S) = N\}.$$

The use of $\boldsymbol{F}$ in Figure 1 guarantees a match between S and any $E \in \boldsymbol{I}$. Then $k(\boldsymbol{F})$ is then the minimum over $E \in \boldsymbol{I}$ of the longest error free subsequence of E. The minimum occurs when E contains only substitutions and the length of E's error free segments are essentially equal

$$k(\boldsymbol{F}) = \lceil (N - e)/(e + 1) \rceil. \tag{2}$$

Moreover, a match is guaranteed when all of E's positions are queried, i.e., $\Delta = 1$ in line B of Figure 1.

To determine $K(\boldsymbol{F})$, we calculate the expected number[1] of random/false positive matches in the algorithm of Figure 1 on a reference of size M for both k-mer indexing and template indexing. We assume here that both methods probe the same number of positions in the read and find

$$E(\text{Number of random matches with } k\text{-mer indexing}) \approx M \times \lfloor 1 + (L(D) - f)/\Delta \rfloor / 4^k, \tag{3}$$

$$E(\text{Number of random matches when indexing with } \boldsymbol{F}) \approx M \times |\boldsymbol{F}| \times \lfloor 1 + (L(D) - f)/\Delta \rfloor / 4^w. \tag{4}$$

The second term in the numerator on the right of (3) also found in (4) is the number of positions indexed in the read, and $1/4^k$ is the probability of a word matching at a given location in the reference. In equation (4) $|\boldsymbol{F}|$ is the number of templates, and $1/4^w$ is the probability of a query gapped words matching at a reference position.

Equating the right sides of equations (3) and (4), and solving for k yields the k-mer index size $K(\boldsymbol{F})$ that generates an equal number of random/false positive matches as indexing with $\boldsymbol{F}$.

$$K(\boldsymbol{F}) = w - \log_4 |F|. \tag{5}$$

In Sections 3.1.1 and 3.1.2, $k(\boldsymbol{F})$ and $K(\boldsymbol{F})$ are computed and compared for several template families. We find that for a given error budget guarantee the covering template family approach is more efficient than k-mer indexing because it yields a substantially smaller number of false positive matches. We recall that matches seed costly dynamic programming alignments and minimizing them reduces the overall cost of any alignment algorithm.

2.3. Greedy algorithm for finding covering template families

We now present a greedy algorithm for finding small covering template families. At each iteration of the algorithm, a new template is added to that family. This template covers as many yet uncovered error instances as possible. The algorithm generates an (N,w,f,e) covering template family provided that $N - e \geq w$ and $N \leq f \leq N + e$. We then prove that the algorithm terminates with a correct solution.

2.3.1. Algorithm description. The algorithm is presented in Figure 2. and we refer to it as the greedy covering template family algorithm. It requires two inputs: the set of error instances and the set of query keys which we now describe.

Input 1: Generate symbolically the set $\boldsymbol{I}$ of all e error instances of size f from the symbolic word $W = (0, \ldots, N-1)$. Specifically, for each set of e errors an error instance is generated as follows. The character I is inserted at each insertion position of W, the character S replaces each substitution's position, and each deletion removes its respective position from W in an error instance. The resulting sequence is then padded with the symbol N or truncated to length f. We refer to this set as the *error instance set*.

Input 2: Generate all query keys, i.e., all vectors of w integers chosen without replacement from the integers $\{0, \ldots, f-1\}$ and sorted in increasing order. This set of vectors is called the *key set*.

We illustrate these sets for the $(N, w, f, e) = (8, 6, 8, 1)$ family. Table 1 presents the set of error instances, while Table 2 presents the key set

The description of the greedy algorithm requires the following definition.

Definition: Given a key K and an error instance $E \in \boldsymbol{I}$ we say that $E[K]$ *is valid* if it does not contain any of the symbols in {S, I, N}, then K is a valid key for E.

---

[1]Reference is assumed to be generated by an iid model, with equal probability for each nucleotide.

```
Input: (N,w,f,e) with N-e ≥ w and N ≤ f ≤ N+e
Generate errorInstanceSet Input 1 above
Generate keyset Input 2 above

templateFamily = ∅
while(errorInstanceSet ≠ ∅) {                                          // Line A
  choose R ∈ errorInstanceSet
  bestCount = 0
  bestKey = UNASSIGNED
  bestErrorInstanceSubset = ∅
  foreach (K ∈ keyset) {                                               // Line B
    if (R[K] is valid) {                                               // Line C
      count = 1
      errorInstanceSubset = {R}

      // Count for how many other error instances S, S[K]
      // is valid and S[K] == R[K]

      foreach (S ∈ errorInstanceSet, S ≠ R) {                          // Line D
        if (S[K] == R[K]) {                                            // Line E
          count++
          errorInstanceSubset = errorInstanceSubset ∪ {S}
        }
      }
      if (count > bestCount) {                                         // Line F
        bestKey = K
        bestCount = count
        bestErrorInstanceSubset = errorInstanceSubset
      }
    }
  }                                                                    // Line G

  // Add the new template to the set of templates

  errorInstanceSet = errorInstanceSet - bestErrorInstanceSubset        // Line H
  newTemplate.queryKey = bestKey                                       // Line I
  newTemplate.referenceKey = R[bestKey]                                // Line J
  templateFamily = templateFamily ∪ {newTemplate}
}
```

FIG. 2. Covering template family algorithm.

Definition: A position j in $E \in \boldsymbol{I}$ is a *valid position* $E[j] \notin \{\mathrm{S}, \mathrm{I}, \mathrm{N}\}$.

2.3.2. Proof of correct termination. We now prove in three steps that the algorithm generates a covering template family $\boldsymbol{F}$. First, we show that it always terminates in a bounded number of iterations, then that $\boldsymbol{F}$ yields a match between E and W for all $E \in \boldsymbol{I}$, and finally that $\boldsymbol{F}$ yields a match for all error instances E with $n \leq e$ errors.

Lemma 1: For each $E \in \boldsymbol{I}$, there are at least $N-e$ valid positions in the N symbol prefix.

Proof. The proof is by induction on e $0 \leq e \leq N-w$. If $e = 0$, all N prefix positions are valid. Suppose that the lemma is true for $e < N-w$ errors. Consider an instance E with $e+1$ errors, and let E' be derived from W by introducing the first e errors. By the induction hypothesis, the N symbol prefix of E' has at least $N-e$ valid positions.

TABLE 1. ERROR INSTANCES FOR (8,6,8,1)

I 0 1 2 3 4 5 6	0 1 2 3 I 4 5 6
1 2 3 4 5 6 7 N	0 1 2 3 5 6 7 N
S 1 2 3 4 5 6 7	0 1 2 3 S 5 6 7
0 I 1 2 3 4 5 6	0 1 2 3 4 I 5 6
0 2 3 4 5 6 7 N	0 1 2 3 4 6 7 N
0 S 2 3 4 5 6 7	0 1 2 3 4 S 6 7
0 1 I 2 3 4 5 6	0 1 2 3 4 5 I 6
0 1 3 4 5 6 7 N	0 1 2 3 4 5 7 N
0 1 S 3 4 5 6 7	0 1 2 3 4 5 S 7
0 1 2 I 3 4 5 6	0 1 2 3 4 5 6 I
0 1 2 4 5 6 7 N	0 1 2 3 4 5 6 N
0 1 2 S 4 5 6 7	0 1 2 3 4 5 6 S

TABLE 2. KEY SET FOR THE (8, 6, 8, 1) FAMILY

2 3 4 5 6 7	0 1 3 5 6 7	0 3 4 5 6 7	0 1 2 4 5 6
1 3 4 5 6 7	0 1 3 4 6 7	0 2 4 5 6 7	0 1 2 3 6 7
1 2 4 5 6 7	0 1 3 4 5 7	0 2 3 5 6 7	0 1 2 3 5 7
1 2 3 5 6 7	0 1 3 4 5 6	0 2 3 4 6 7	0 1 2 3 5 6
1 2 3 4 6 7	0 1 2 5 6 7	0 2 3 4 5 7	0 1 2 3 4 7
1 2 3 4 5 7	0 1 2 4 6 7	0 2 3 4 5 6	0 1 2 3 4 6
1 2 3 4 5 6	0 1 2 4 5 7	0 1 4 5 6 7	0 1 2 3 4 5

Introducing the $e+1$st error into E' followed by a truncation or padding with the symbol N yields E. If this error is an insertion, the positions that follow are shifted to the right and only the last position in the prefix of E' is no longer in that of E. Hence, the latter has at least $(N-e)-1=N-(e+1)$ valid positions. The case of substitution and deletion are proved in a similar fashion. ■

The next lemma shows that the greedy algorithm always terminates in a bounded number of steps.

Lemma 2: At each iteration of the block beginning line A, the size of errorInstanceSet decreases by at least one.

Proof: In view of lemma 1, the error instance R chosen after line A has at least $N-e \geq w$ valid positions in its N symbol prefix, so there is at least one valid key K for R. When K is a valid key for R, the condition on line C holds and errorInstanceSubset is initialized to contain R. Then, in the block of line D, errorInstanceSubset can only increase in size. In the block of line F, bestErrorInstanceSubset is either initialized or possibly updated to the current errorInstanceSubset, so it has at least one member, and at line H, errorInstanceSet is reduced by at least one member. It follows that the algorithm will complete in at most $|\boldsymbol{I}|$ iterations. ■

The next lemma shows that the template family generated by the algorithm yields a match between all $E \in \boldsymbol{I}$ and W.

Lemma 3: Let $\boldsymbol{F}$ be the set of templates generated by the greedy covering template family algorithm. Then for any $E \in \boldsymbol{I}$ there is a $T^* \in \boldsymbol{F}$ that generates a match between W and E.

Proof. Consider the iteration of line A at which E was removed from errorInstanceSet, then $E \in$ bestErrorInstanceSubset on line H. Let T^* be the template defined on lines I and J at this iteration. Then in view of line C and line E $E[T^*.queryKey]$ is valid and line J implies $E[T^*.queryKey] = T^*.referenceKey$. Now, since $W=(0, \ldots, N\text{ - }1)$, $W[T^*.referenceKey] = T^*.referenceKey$ hence $W[T^*.referenceKey] = E[T^*.queryKey]$ ■

Error instances of $\boldsymbol{I}$ have exactly e errors while the definition of covering template family requires that all error instances with $n \leq e$ have a match. The following remark demonstrates that this property holds for $\boldsymbol{F}$.

Remark 1: Let E be an error instance generated by introducing $n < e$ errors into W, and let $\boldsymbol{F}$ be the template family generated by the greedy covering template family algorithm. Then there is a $T^* \in \boldsymbol{F}$ that yields a match between W and E.

Proof. We first substitute e-n nucleotides of E by the symbol S to obtain E'. By lemma 3, there is a $T^* \in \boldsymbol{F}$ for which $E'[T^*.queryKey] = W[T^*.referenceKey]$. Since $E'[T^*.queryKey]$ is valid, $T^*.queryKey$ contains none of the artificial substitution's positions. Hence $E'[T^*.queryKey] = E[T^*.queryKey] \Rightarrow W[T^*.referenceKey] = E[T^*.queryKey]$ ■

We summarize the analysis of this section with the following theorem.

Theorem 1: The greedy covering template family algorithm always terminates in a bounded number of steps and finds a covering template family.

We note that the speed of the greedy algorithm depends on the parameters of the family, as the latter affect the size of the error instance set, and the key set. We also note that the covering template family algorithm can be used with error instances involving either errors of all types, or errors of just one type (e.g., just deletions). In fact, any combination of error types is possible.

2.4. Modular template family construction

We now construct a family tolerant to $k+1$ errors using a family tolerant of k errors. Specifically, a $(N+w/2,\ w,\ N+w/2+k+1,\ k+1)$ covering template family $\boldsymbol{F}'$ is constructed using templates from a $(N, w, N+k, k)$ covering template family $\boldsymbol{F}$ and a few additional templates. The families generated here require much fewer indexes as compared to the ones generated by the greedy covering template family algorithm.

2.4.1. Preliminary results. We begin by compiling preliminary definitions and results.

Definition: A (K, D) *symmetric key,* is the key $(0, \ldots, K-1, K+D, \ldots, 2K+D-1)$.

A symmetric key is used to extract from a sequence two words of length K separated by a gap of length D.

Definition: A *symmetric template family STF(K, D, k)* $\boldsymbol{F}$ is the collection of $2k+1$ templates such that for each $T \in \boldsymbol{F}$: *T.referenceKey* is the *(K,D)* symmetric key and *T.queryKey* $\in \{(K, D \pm j)$ symmetric key $\mid j = 0, \ldots, k\}$.

Lemma 1: Consider word S and an error instance E obtained by introducing n errors $\{e_1, \ldots, e_n\}$ into S, where e_i is either an insertion, a deletion or a substitution, then

$$L(E) = L(S) + \Phi(e_1, \ldots, e_n),$$

$$\Phi(e_1, \ldots, e_n) = \sum_{i=1}^{n} \phi(e_i),$$

$\phi(e_i) = 1$ for an insertion, $\phi(e_i) = -1$ for a deletion, and $\phi(e_i) = 0$ for a substitution.

Proof. The proof is by induction on n. If $n = 0$ $\Phi = 0$ and $E = S$ therefore $L(E) = L(S)$. Assume the lemma holds for up to n errors. Consider the set of $n+1$ errors $\{e_1, \ldots, e_{n+1}\}$. Let E' be obtained by introducing e_1 into S, then

$$L(E') = L(S) + \Phi(e_1), \tag{6}$$

since a substitution does not change the length of a sequence while an insertion or a deletion respectively increases or decreases the length by one. Introducing $\{e_2, \ldots, e_{n+1}\}$ into E' yields E and by the induction hypothesis:

$$L(E) = L(E') = + \Phi(e_2, \ldots, e_{n+1}). \tag{7}$$

Combining (6), (7), and the definition of Φ proves the lemma ■

Lemma 2: Consider a word S of size $N = 2K + D$ and its subsequences: S_1 its first K nucleotides, S_2 the next D nucleotides and S_3 the last K nucleotides. Let $\boldsymbol{F} = STF(K,D,k)$, $k \leq D$. Then for any error instance E obtained from S by introducing n errors, $n \leq k$, into S_2 there is a $T \in \boldsymbol{F}$ that yields a match between E and S.

Proof. Let S_2' be derived from S_2 by introducing n errors $\{e_1, \ldots, e_n\}$. Then $E = S_1 + S_2' + S_3$ and by lemma 2 $L(S_2') = D + \Phi(e_1, \ldots, e_n)$ where $-n \leq \Phi(e_1, \ldots, e_n) \leq n$. From the definition of a symmetric template family, there is a $T^* \in \boldsymbol{F}$ for which *T*.queryKey* is the $(K, D + \Phi(e_1, \ldots, e_n))$ symmetric key, and therefore $E[T^*.\text{queryKey}] = S_1 + S_3$ then $S[T^*.referenceKey] = E[T^*.queryKey]$ ■

2.4.2. The modular construction. To construct $\boldsymbol{F}'$, we consider a sequence S with $L(S) = N + w/2$ and the set

$$\boldsymbol{I} = \{E | E \text{ is an } l \text{ error instance of } S, l \leq k+1\}.$$

We determine $\boldsymbol{F}'$ by constructing a set of templates that yields a match between S and any $E \in \boldsymbol{I}$. To that end we consider the subsequences of S: S_1 the first $w/2$ nucleotides, S_2 the next $N-w/2$ and S_3 the last $w/2$. Any $E \in \boldsymbol{I}$ is partitioned accordingly into S_1', S_2' and S_3' with S_i' obtained by introducing the corresponding part of the l errors into S_i. Based on the distribution of the errors in S_i', $i = 1, 2, 3$, we partition $\boldsymbol{I}$ into groups:

Group 1: $S_1' + S_2'$ is an n error instance of $S_1 + S_2$ with $0 \leq n \leq k$ errors. This group includes instances E with no more than k errors. Since $L(S_1 + S_2) = N$ a member of $\boldsymbol{F}$ yields a match between $S_1 + S_2$ and $S_1' + S_2'$ and therefore between S and E.

Group 2: E has $k+1$ errors and $S_2' + S_3'$ is an n error instance of $S_2 + S_3$ with $0 \leq n \leq k$ errors. Since $L(S_2 + S_3) = N$ there is a $T^* \in \boldsymbol{F}$ that yields a match between $S_2 + S_3$ and $S_2' + S_3'$ hence

$$S[w/2 + T^*.referenceKey] = E[L(S_1') + T^*.queryKey]. \tag{8}$$

The number of errors in S_1' is $k+1-n$ and $L(S_1' \geq 0$ so using lemma 1

$$\max(0, w/2 - k - 1 + n) \leq L(S_1') \leq w/2 + k + 1 - n. \tag{9}$$

Moreover, lemma 1 implies $L(S_2' + S_3') \leq N + n$ so that size($T^*.queryKey$) $\leq N + n$ and

$$L(S_1') + \text{ size}(T^*.queryKey) \leq N + w/2 + k + 1.$$

It follows that

$$\max(0, w/2 - k - 1) \leq L(S_1') \leq N + w/2 + k + 1 \text{ - size}(T^*.queryKey). \tag{10}$$

This implies that there is a $T_s \in Shift(\boldsymbol{F})$ that yields a match between S and E where:

$$Shift(\boldsymbol{F}) = \{T_s | \ T_s = (w/2 + T.referenceKey, \ l + T.queryKey), T \in \boldsymbol{F}$$
$$\max(0, w/2 - k - 1) \leq l \leq N + w/2 + k + 1 - \text{size}(T.queryKey)\}$$

Group 3: S_2' is a $k+1$ error instance of S_2. This is the configuration of lemma 2 with $K = w/2$ and $D = N - w/2$. Hence, members of $STF(w/2, N-w/2, k+1)$ yield a match for all error instances in this group.

We summarize this analysis in the following theorem.

Theorem 2: Let $\boldsymbol{F}$ be an $(N, w, N+k, k)$ covering template family and

$$\boldsymbol{F}' = \boldsymbol{F} \ \cup \ Shift(\boldsymbol{F}) \ \cup \ STF(w/2, N - w/2, k+1). \tag{11}$$

Then $\boldsymbol{F}'$ is an $(N + w/2, \ w, \ N + w/2 + k + 1, \ k+1)$ covering template family.

In section 2.4.3 we show that indexes of $\boldsymbol{F}'$ are those of $\boldsymbol{F}$ plus the single index of $STF(w/2, N-w/2, k+1)$.

2.4.3. The modular recursion. The construction of Section 2.4.2 can be applied recursively to obtain a hierarchy of covering template families $\boldsymbol{F_k}$ with parameters (N_k, w, f_k, k). Here, we study properties of these families. We study the recursion in which $\boldsymbol{F_1}$ is given.

Lemma 3: The local error rate k/N_k is monotonically increasing with k and

$$\lim_{k \to \infty} k/N_k = 2/w. \tag{13}$$

Proof. $N_k = N_{k-1} + w/2$ so that

$$N_k = (k-1)w/2 + N_1. \tag{14}$$

Dividing k by N_k and taking the limit with respect to k yields (13). To show monotonic behavior, we take the derivative with respect to k of k/N_k to obtain

$$d/dk(k/N_k) = (N_1 - w/2)/((k-1)w/2 + N_1)^2. \quad (15)$$

Both numerator and denominator in (15) are positive. ■

The derivative in (15) converges to zero as $k \to \infty$, indicating that only a few applications of the recursion are useful in practice.

We now determine the number of indexes and templates required to search with $\boldsymbol{F_k}$ when an index lookup is used to replace the loop of Line C in the template indexing algorithm of Figure 1.

Definition: Let T be a template then $T_s = (m + T.referenceKey,\ n + T.queryKey)$ $m,\ n \geq 0$ is a *shifted version* of T.

Lemma 4: Let $Base(\boldsymbol{F}_k) = \bigcup_{j=1}^{k-1} STF(w/2, N_j - w/2, j+1) \cup \boldsymbol{F}_1$ then

(a) Any template of $\boldsymbol{F_k}$ belongs to $Base(\boldsymbol{F_k})$ or is a shifted version of a template in $Base(\boldsymbol{F_k})$.
(b) The number of templates in $Base(\boldsymbol{F_k})$ is $O(k^2)$.

Proof. We show a) by induction on k using (11). We show b) using ■

$$|Base(\boldsymbol{F}_k)| = |\boldsymbol{F}_1| + \sum_{j=1}^{k-1} (2(j+1)+1) = O(K^2). \quad (16)$$

In (16) $2(j+1)+1$ is the number of templates in $STF(w/2,\ N_j\text{-}w/2,\ j+1)$ ■

Lemma 5: Let $T_s \in \boldsymbol{F}_k$ be a shifted version of $T \in \boldsymbol{F}_k$.

(a) The index for T can be used for T_s.
(b) If $T_s.queryKey = T.queryKey$ then T_s is redundant.

Proof. Since $T_s.referenceKey = T.referenceKey + m$ any query in T's index will generate the hits obtained by querying T_s's index and potentially m additional hits that can be ignored. This demonstrates a). If $T_s.queryKey = T.queryKey$ query results of T_s are included in those of T, and this shows b). ■

Lemma 6: The number of indexes required by the k'th family is O(k).

Proof: Lemma 4 a) and Lemma 5 a) imply that the indexes of $Base(\boldsymbol{F_k})$ are those of $\boldsymbol{F_k}$. The family $STF(w/2, N_j - w/2, j+1)$ has a unique reference key and therefore a unique index, so the number of indexes in $Base(\boldsymbol{F_k})$ is bounded by $(k-1) + |\boldsymbol{F_1}|$. ■

Lemma 7: The number of non-redundant templates in $\boldsymbol{F_k}$ is O(k^3).

Proof. For each $T \in Base(\boldsymbol{F}_k)$ and $0 \leq n \leq f_k$ we define the sets: ■

$$[T,n] = \{T_s | T_s \in \boldsymbol{F}_k, T_s = (T.referenceKey + m,\ T.queryKey + n)\},$$
$$M(T,n) = \{m | \exists\ T_s \in [T,n], T_s = (T.referenceKey + m, T.queryKey + n)\}.$$

Note that for some n $[T,n]$ may be empty and that

$$\boldsymbol{F}_k = \bigcup_{T \in Base(F_k)} \bigcup_{n=0}^{f_k} [T,n]. \quad (17)$$

Moreover, if $m^* = \min M(T, n)$ and $T^* = (T.referenceKey + m^*, T.queryKey + n)$ Lemma 5 b) implies that all templates in $[T,n]$ are redundant except T^*. In conjunction with (17) this yields the bound

TABLE 3. TEMPLATE FAMILY PROPERTIES

Family	*Number of templates*	*Local % error e/N*	*k(**F**)*	*K(**F**)*
(18,16,18,1)	26	5.55%	9	13.65
(20,16,20,1)	14	5.00%	10	14.1
(20,16,20,2)	329	10%	6	11.8
(25,16,25,2)	86	8%	8	12.8
(26,16,26,2)	77	7.70%	8	12.8
(29,16,29,2)	51	6.90%	9	13.16

$|Base(\boldsymbol{F_k})|\times(f_k+1)$ on the number of non-redundant templates. Now (14) implies $f_k=(k-1)\ w/2+N_1+k$, which when combined with (16) proves the lemma. ■

3. RESULTS

3.1. Families generated by greedy algorithm

3.1.1. General properties. We now present properties of template families generated by the greedy covering template family algorithm. Table 3 presents the number of templates, $k(\boldsymbol{F})$ and $K(\boldsymbol{F})$ of equation (2) and (5), and the ratio *e/N* in % for several template families $\boldsymbol{F}$.

The larger the value of $K(\boldsymbol{F})-k(\boldsymbol{F})$ the more efficient the template family in achieving the error budget guarantee as compared to *k*-mer indexing, because *k*-mer indexing will generate $4^{(K(\boldsymbol{F})-k(\boldsymbol{F}))}$ times more random spurious alignments than template indexing, in order to achieve the same error budget guarantee.

3.1.2. Comparison with k-mer indexing. In this section, we verify computationally the results of Sections 2.2.2 and 3.1.1 for the (26,16,26,2) template family $\boldsymbol{F}$. Specifically, a set of 100000 randomly chosen two-error reads of length 26 from human chromosome 1 are aligned to chromosome 1 with both $\boldsymbol{F}$, and *k*-mer indexing where *k* is varied in the range $k(\boldsymbol{F}), \ldots, K(\boldsymbol{F})$.

When a query word matches more than $M=10000$ locations in the reference, it will be ignored. Table 4 presents the percentage of reads with at least one match to their location of origin (% Found), the total number of candidate locations generated in each alignment run (Candidate positions), and the number of candidate locations normalized by the number generated by $\boldsymbol{F}$.

The fraction of reads that match their location of origin when seeding with $\boldsymbol{F}$ and with $k=k(\boldsymbol{F})$ is, respectively, 99.84% and 98.9%. Neither seeding strategies achieve their theoretical guarantees because query words with a number of matches greater than $M=10000$ are ignored. Indexing with $k(\boldsymbol{F})$ yields 54 times more candidate locations than with $\boldsymbol{F}$. Indexing with $k=K(\boldsymbol{F})=13$, yields a slightly greater number of candidate positions than $\boldsymbol{F}$ but finds the correct location for only 81% of the reads. This highlights the superiority of template family indexing over *k*-mer indexing in both sensitivity and cost.

3.1.3. Template family structure. In this section, we study the template structure of a frequently used family generated by the greedy covering template family algorithm. This structure yields a compression scheme for pre-computed indexes in one of the implementations of the indexing algorithm which is described in Section 3.3.2.

TABLE 4. COMPARISON WITH K-MER INDEXING, M = 10000

Indexing type	*% Found*	*Candidate positions*	*Relative number of positions*
(26,16,26,2)	99.84	57,324,646	1
k = 9	98.90	3,135,083,623	54.69
k = 10	98.10	974,329,226	17.00
k = 11	95.32	328,012,196	5.72
k = 12	90.66	132,708,178	2.32
k = 13	83.99	67,536,030	1.18
k = 14	74.85	41,268,785	0.72

TABLE 5. MODULAR FAMILY'S PARAMETERS

k	N	*f*	*k/N in %*	*Num indexes*
1	18	19	5.56	9
2	26	28	7.69	10
3	34	37	8.82	11
4	42	46	9.52	12
5	50	55	10	13

The Appendix presents the templates of the (*18*,*16*,*18*,*1*) family. Tables 6–9 correspond to the substitution only, deletion only, insertions only and all error types, respectively.

The nine reference keys K_i, $i = 0, 1, \ldots, 8$, in Tables 6 and 7 are identical and consist of the integers $0, \ldots, 17$ with the exception of the gap $[2i, 2i + 1]$. Read keys have an identical structure to the reference keys, but have a gap of size two and one for the substitution and deletion case, respectively. Read keys with a gap size of three yield templates of read key size 19 that cover all insertion cases.

The last template in Tables 6 and 7 are identical so the total number in all three tables is *26*, and is equal to the one in Table 9. We also note that there is some redundancy in the templates at the edge of the word. For example, the first templates in Tables 6 and 7 are equivalent, because the reference is indexed at every position.

3.2. Modular families

Here, we present the parameters of the families generated by the modular recursion of Section 2.4.3, where $\boldsymbol{F_1}$ is the $(18, 16, 19, 1)$ family of Section 3.1.3. We use the formulas derived in Section 2.4.3 to obtain Table 5.

We see that a relatively high error tolerance can be achieved with a modest number of indexes.

3.3. Implementation details

3.3.1. Indexing with merge sort. Indexing is implemented as a merge-sort process. Specifically, all gapped words are encoded as integers, by translating each *w*-mer as a base *4* number with $\{A, C, G, T\}$ respectively mapped to $\{0, 1, 2, 3\}$. Translation can be done either left to right or right to left, respectively called *right significant* and *left significant*.

Matches are found by sorting reference and query gapped words, and merging the resulting sorted lists. Only a portion of the sorted index needs to be in memory at any point in time.

3.3.2. Database compression. It is desirable to pre-compute the indexes in order to reduce computation time. A substantial compression of the index data-base is achieved for certain families by leveraging the structure of the reference keys.

We illustrate this for the $(18, 16, 19, 1)$ family. The *9* reference keys of Table 6 consist of two groups. The first five share their last *8* positions, while the last four share their first *8*. A single database for the first five reference keys, and last 4 keys is generated by encoding the full *18* nucleotide word in a right and left significant mode, respectively. The resulting lists are then sorted to create two databases.

TABLE 6. (18,16,18,1) SUBSTITUTION ONLY

Reference keys	*Read keys*
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17	2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
0 1 4 5 6 7 8 9 10 11 12 13 14 15 16 17	0 1 4 5 6 7 8 9 10 11 12 13 14 15 16 17
0 1 2 3 6 7 8 9 10 11 12 13 14 15 16 17	0 1 2 3 6 7 8 9 10 11 12 13 14 15 16 17
0 1 2 3 4 5 8 9 10 11 12 13 14 15 16 17	0 1 2 3 4 5 8 9 10 11 12 13 14 15 16 17
0 1 2 3 4 5 6 7 10 11 12 13 14 15 16 17	0 1 2 3 4 5 6 7 10 11 12 13 14 15 16 17
0 1 2 3 4 5 6 7 8 9 12 13 14 15 16 17	0 1 2 3 4 5 6 7 8 9 12 13 14 15 16 17
0 1 2 3 4 5 6 7 8 9 10 11 14 15 16 17	0 1 2 3 4 5 6 7 8 9 10 11 14 15 16 17
0 1 2 3 4 5 6 7 8 9 10 11 12 13 16 17	0 1 2 3 4 5 6 7 8 9 10 11 12 13 16 17
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Table 7. (18,16,18,1) Deletion Only

Reference keys	*Read keys*
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
0 1 4 5 6 7 8 9 10 11 12 13 14 15 16 17	0 1 3 4 5 6 7 8 9 10 11 12 13 14 15 16
0 1 2 3 6 7 8 9 10 11 12 13 14 15 16 17	0 1 2 3 5 6 7 8 9 10 11 12 13 14 15 16
0 1 2 3 4 5 8 9 10 11 12 13 14 15 16 17	0 1 2 3 4 5 7 8 9 10 11 12 13 14 15 16
0 1 2 3 4 5 6 7 10 11 12 13 14 15 16 17	0 1 2 3 4 5 6 7 9 10 11 12 13 14 15 16
0 1 2 3 4 5 6 7 8 9 12 13 14 15 16 17	0 1 2 3 4 5 6 7 8 9 11 12 13 14 15 16
0 1 2 3 4 5 6 7 8 9 10 11 14 15 16 17	0 1 2 3 4 5 6 7 8 9 10 11 13 14 15 16
0 1 2 3 4 5 6 7 8 9 10 11 12 13 16 17	0 1 2 3 4 5 6 7 8 9 10 11 12 13 15 16
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Table 8. (18,16,18,1) Insertion Only

Reference keys	*Read keys*
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16	2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
0 1 3 4 5 6 7 8 9 10 11 12 13 14 15 16	0 1 4 5 6 7 8 9 10 11 12 13 14 15 16 17
0 1 2 3 5 6 7 8 9 10 11 12 13 14 15 16	0 1 2 3 6 7 8 9 10 11 12 13 14 15 16 17
0 1 2 3 4 5 7 8 9 10 11 12 13 14 15 16	0 1 2 3 4 5 8 9 10 11 12 13 14 15 16 17
0 1 2 3 4 5 6 7 9 10 11 12 13 14 15 16	0 1 2 3 4 5 6 7 10 11 12 13 14 15 16 17
0 1 2 3 4 5 6 7 8 9 11 12 13 14 15 16	0 1 2 3 4 5 6 7 8 9 12 13 14 15 16 17
0 1 2 3 4 5 6 7 8 9 10 11 13 14 15 16	0 1 2 3 4 5 6 7 8 9 10 11 14 15 16 17
0 1 2 3 4 5 6 7 8 9 10 11 12 13 15 16	0 1 2 3 4 5 6 7 8 9 10 11 12 13 16 17
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Table 9. (18,16,18,1) All Error Types

Reference keys	*Read keys*
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16	2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
0 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17	0 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
0 1 3 4 5 6 7 8 9 10 11 12 13 14 15 16	0 1 4 5 6 7 8 9 10 11 12 13 14 15 16 17
0 1 4 5 6 7 8 9 10 11 12 13 14 15 16 17	0 1 3 4 5 6 7 8 9 10 11 12 13 14 15 16
0 1 2 5 6 7 8 9 10 11 12 13 14 15 16 17	0 1 2 5 6 7 8 9 10 11 12 13 14 15 16 17
0 1 2 3 5 6 7 8 9 10 11 12 13 14 15 16	0 1 2 3 6 7 8 9 10 11 12 13 14 15 16 17
0 1 2 3 6 7 8 9 10 11 12 13 14 15 16 17	0 1 2 3 5 6 7 8 9 10 11 12 13 14 15 16
0 1 2 3 4 7 8 9 10 11 12 13 14 15 16 17	0 1 2 3 4 7 8 9 10 11 12 13 14 15 16 17
0 1 2 3 4 5 7 8 9 10 11 12 13 14 15 16	0 1 2 3 4 5 8 9 10 11 12 13 14 15 16 17
0 1 2 3 4 5 8 9 10 11 12 13 14 15 16 17	0 1 2 3 4 5 7 8 9 10 11 12 13 14 15 16
0 1 2 3 4 5 6 9 10 11 12 13 14 15 16 17	0 1 2 3 4 5 6 9 10 11 12 13 14 15 16 17
0 1 2 3 4 5 6 7 9 10 11 12 13 14 15 16	0 1 2 3 4 5 6 7 10 11 12 13 14 15 16 17
0 1 2 3 4 5 6 7 10 11 12 13 14 15 16 17	0 1 2 3 4 5 6 7 9 10 11 12 13 14 15 16
0 1 2 3 4 5 6 7 8 11 12 13 14 15 16 17	0 1 2 3 4 5 6 7 8 11 12 13 14 15 16 17
0 1 2 3 4 5 6 7 8 9 11 12 13 14 15 16	0 1 2 3 4 5 6 7 8 9 12 13 14 15 16 17
0 1 2 3 4 5 6 7 8 9 12 13 14 15 16 17	0 1 2 3 4 5 6 7 8 9 11 12 13 14 15 16
0 1 2 3 4 5 6 7 8 9 10 13 14 15 16 17	0 1 2 3 4 5 6 7 8 9 10 13 14 15 16 17
0 1 2 3 4 5 6 7 8 9 10 11 13 14 15 16	0 1 2 3 4 5 6 7 8 9 10 11 14 15 16 17
0 1 2 3 4 5 6 7 8 9 10 11 14 15 16 17	0 1 2 3 4 5 6 7 8 9 10 11 13 14 15 16
0 1 2 3 4 5 6 7 8 9 10 11 12 15 16 17	0 1 2 3 4 5 6 7 8 9 10 11 12 15 16 17
0 1 2 3 4 5 6 7 8 9 10 11 12 13 15 16	0 1 2 3 4 5 6 7 8 9 10 11 12 13 16 17
0 1 2 3 4 5 6 7 8 9 10 11 12 13 16 17	0 1 2 3 4 5 6 7 8 9 10 11 12 13 15 16
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 17	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 17
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

When processing a specific reference key bit-wise operations are used to extract the encoding for that key from that of the word. This results in a partially sorted index, which is fully sorted by the most significant *16* bit radix. This index is loaded into memory in small sections partitioned along the significant radix boundaries. Each section can be fully sorted with a small amount of additional computation.

3.3.3. Software. The helisphere distribution contains several programs associated with the aligner:

- IndexDP: very flexible version of the aligner, designed for references of size up to 100MB.
- Templates: program for generating template families used by indexDP. It is based on the greedy template family algorithm.
- IndexDPgenomic: designed for large references it has a small memory footprint. It makes use of the pre-computed compressed database discussed in section 3.3.2.
- PreprocessDB: creates compressed index database for indexDPgenomic.'

4. DISCUSSION

We have developed several approaches for generating covering template families and have demonstrated that they provide a very efficient form of indexing. Specifically, for a given error budget, they yield a match with substantially fewer spurious false positive matches than *k*-mer indexing. This new form of indexing enabled us to create a short read aligner that is tolerant of errors of all type.

5. APPENDIX

Here we present the templates associated with the (18,16,18,1) family, generated by the algorithm of Section 2.3. Tables 6–8 correspond to the case where the algorithm was run on each error type separately. Table 9 corresponds to the case where all error types where considered simultaneously.

DISCLOSURE STATEMENT

All of the authors of this manuscript (except G.M.) are or have been employees of Helicos BioSciences. G.M. was on the scientific advisory board of Helicos BioSciences Corporation.

REFERENCES

Altschul, S., Madden, T., Schäffer, A., et al. 1997. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Res.* 25, 3389–3402.

Brejova, B., Brown, D., and Vinar, T. 2003. Vector seeds: an extension to spaced seeds allows substantial improvements in sensitivity and specificity. *Proc. 3rd Int. WABI* 39–54.

Brown, D.G. 2004. Multiple vector seeds for protein alignment. *Proc. 4th Int. WABI* 170–181.

Buhler, J. 2001. Efficient large-scale sequence comparison by locality-sensitive hashing. *Bioinformatics* 17, 419–428.

Buhler, J., Keich, U., and Sun, Y. 2003. Designing seeds for similarity search in genomic DNA. *Proc. RECOMB 2003* 67–75.

Burkhardt, S., and Kärkkäinen, J. 2003. Better filtering with gapped q-grams. *Fund. Inform.* 56, 51–70.

Burkhardt, S., and Kärkkäinen, J. 2002. One-gapped q-gram filters for Levenshtein distance. *Proc. 13th Annu. Symp. Combin. Patt. Match*.

Califano, A., and Rigoutsos, I. 1993. Flash: A **F**ast **L**ook-up **A**lgorithm for **S**tring **H**omology. *1st Int. Conf. Intell. Syst. Mol. Biol.* 56–64.

Choi, K., and Zhang, L. 2004. Sensitivity analysis and efficient method for identifying optimal spaced seeds. *J. Comput. Syst. Sci.* 68, 22–40.

Harris, T.D., Buzby, P.R., Babcock, H., et al. 2008. Single-molecule DNA sequencing of a viral genome. *Science* 320, 106–109.

Johnson, D.S., Mortazavi, A., Myers, R.M., et al. 2007. Genome-wide mapping of in vivo protein–DNA interactions. *Science* 316, 1497–1502.

Kaiser, J. 2008. DNA sequencing. A plan to capture human diversity in 1000 genomes. *Science* 319.
Keich, U., Li, M., Ma, B., et al. 2004. On spaced seeds for similarity search. *Discrete Appl. Math* 138, 253–263.
Kent, J.W. 2002. BLAT-The BLAST-like alignment tool. *Genome Res.* 12, 656–664.
Kucherov, G., Noé, L., and Roytberg, M. 2005. Multiseed lossless filtration. *IEEE/ACM Trans. Comput. Biol. Bioinform.* 2, 51–61.
Kucherov, G., Noé, L., and Ponty, Y. 2004. Estimating seed sensitivity on homogeneous alignments. *IEEE 4th Symp. BIBE 2004*.
Li, H., Ruan, J., and Durbin, R. 2008. Mapping short DNA sequencing reads and calling variants using mapping quality scores. *Genome Res.* 18, 1851–1858.
Li, M., Ma, B., Kisman, D., et al. 2004. PatternHunter II: highly sensitive and fast homology search. *J. Bioinform. Comput. Biol.* 2, 417–440.
Lipson, D., Raz, T., Kieu, A., et al. 2009. Quantification of the yeast transcriptome by single-molecule sequencing. *Nat. Biotechnol.* 27, 652–658.
Ma, B., Tromp, J., and Li, M. 2002. PatternHunter: faster and more sensitive homology search. *Bioinformatics* 18, 440–445.
Mosaik. http://bioinformatics.bc.edu/marthlab/mosaik
Myers, G. 1999. A fast bit-vector algorithm for approximate string matching based on dynamic progamming. *J. ACM* 46, 395–415.
Noé, L., and Kucherov, G. 2004. Improved hit criteria for DNA local alignment. *BMC Bioinform.* 5.
Pevzner, P., and Waterman, M. 1995. Multiple filtration and approximate pattern matching. *Algorithmica* 13, 135–154.
Rumble, S.M., Lacroute, P., Dalca, A.V., et al. 2009. SHRiMP: accurate mapping of short color-space reads. *PLOS Comput. Biol.* 5.
Schwartz, S., Kent, J., Smit, A., et al. 2003. Human–mouse alignments with BLASTZ. *Genome Res.* 13, 103–107.
Stromberg, M. 2009. Mosaik assembler.
Sun, Y., and Buhler, J. 2004. Designing multiple simultaneous seeds for DNA similarity search. *Proc. RECOMB 2004* 76–84.
Wang, J., Wang, W., Li, R., et al. 2008. The diploid genome sequence of an Asian individual. *Nature* 456, 60–65.
Xu, J., Brown, D., Li, M., et al. 2004. Optimizing multiple spaced seeds for homology search. *15th Symp. Combin. Patt. Match.* 47–58.

Address correspondence to:
Dr. Eldar Giladi
Helicos BioSciences Corporation
One Kendall Square, Suite 7301
Cambridge, MA 02139

E-mail: egiladi@helicosbio.com