# Context-Aware Seeds for Read Mapping

**Hongyi Xin**
Computer Science Department, School of Computer Science,
Carnegie Mellon University, Pittsburgh, PA, USA

**Mingfu Shao**
Department of Computer Science and Engineering,
The Pennsylvania State University, University Park, PA, USA

**Carl Kingsford**[1]
Computational Biology Department, School of Computer Science,
Carnegie Mellon University, Pittsburgh, PA, USA

## Abstract

**Motivation:** Most modern seed-and-extend NGS read mappers employ a seeding scheme that requires extracting $t$ non-overlapping seeds in each read in order to find all valid mappings under an edit distance threshold of $t$. As $t$ grows (such as in long reads with high error rate), this seeding scheme forces mappers to use more and shorter seeds, which increases the seed hits (seed frequencies) and therefore reduces the efficiency of mappers.
**Results:** We propose a novel seeding framework, context-aware seeds (CAS). CAS guarantees finding all valid mapping but uses fewer (and longer) seeds, which reduces seed frequencies and increases efficiency of mappers. CAS achieves this improvement by attaching a confidence radius to each seed in the reference. We prove that all valid mappings can be found if the sum of confidence radii of seeds are greater than $t$. CAS generalizes the existing pigeonhole-principle-based seeding scheme in which this confidence radius is implicitly always 1. Moreover, we design an efficient algorithm that constructs the confidence radius database in linear time. We experiment CAS with *E. coli* genome and show that CAS reduces seed frequencies by up to 20.3% when compared with the state-of-the-art pigeonhole-principle-based seeding algorithm, the Optimal Seed Solver.
**Availability:** `https://github.com/Kingsford-Group/CAS_code`

**2012 ACM Subject Classification** Applied computing → Bioinformatics

**Keywords and phrases** Read Mapping, Seed and Extend, Edit Distance, Suffix Trie

**Digital Object Identifier** 10.4230/LIPIcs.WABI.2019.15

**Supplement Material** `https://github.com/Kingsford-Group/CAS_code`

**Funding** This research is funded in part by the Gordon and Betty Moore Foundation's Data-Driven Discovery Initiative through grant GBMF4554 to CK, by the U.S. National Science Foundation (CCF-1319998) and by the U.S. National Institutes of Health (R01GM122935). This work was partially funded by the Shurl and Kay Curci Foundation. This project is funded, in part, by a grant (4100070287) from the Pennsylvania Department of Health. The department specifically disclaims responsibility for any analyses, interpretations, or conclusions.

## 1 Introduction

Read mapping is used ubiquitously in bioinformatics. Commonly, it is defined as follows:

▶ **Problem 1** (Read Mapping). *Given read $R$ and reference $T$ (usually with $|T| \gg |R|$), an edit distance measurement $D(\cdot,\cdot)$, and an error tolerance threshold $t$, we say a substring of $T$ at location $[l_1, l_2]$, i.e., $T[l_1, l_2]$, is a valid mapping of $R$ if we have $D(R, T[l_1, l_2]) < t$.*

[1] corresponding author, email: `carlk@cs.cmu.edu`


To efficiently map reads, modern mappers usually employ the *seed-and-extend* mapping strategy [8, 9, 1, 14]: a mapper extracts a substring of $R$ as a *seed*, $s$; iterates through all seed locations of $s$ in $T$; at each seed location, performs sequence alignment of $R$ against the surrounding text in $T$; reports alignments that have edit distances below $t$ as valid mappings.

For mappers that use non-overlapping seeds, the number of seeds to extract from a read $R$ is governed by the pigeonhole principle: to find all valid mappings of $R$, the mapper must divide $R$ into at least $t$ non-overlapping seeds. Otherwise, the mapper will not be able to consistently find all valid mappings of $R$ in $T$. As $t$ grows, the length of seeds is reduced. Using short seeds significantly increases the workload of a mapper [6, 11]. Shorter seeds appear more frequently in $T$, hence increasing the number of alignments while mapping a read. To improve the performance of mappers, it is desirable to use fewer non-overlapping seeds under a fixed $t$, which lets a mapper not only use fewer seeds, but also use longer seeds.

In this paper, we focus on improving seed-and-extend mappers that use non-overlapping seeds. We propose a novel seeding scheme, called *context-aware* seeds (CAS). CAS enables a mapper to use fewer than $t$ seeds without missing any valid mappings. CAS attaches each seed $s$ with a *confidence radius* score, $c_s$, with $c_s \geq 1$. Let $S$ be a set of non-overlapping seeds from $R$. CAS ensures that as long as $\sum_{s \in S} c_s \geq t$, then $S$ is sufficient to find all valid mappings of $R$ under an error tolerance threshold of $t$. When $S$ includes any seed $s$ with $c_s > 1$, then $|S| < t$ and all valid mappings are secured with fewer-than-$t$ seeds ($|S|$ denotes the number of seeds in $S$). In the worst case where $c_s = 1$ for all $s \in S$, CAS degenerates into the case governed by pigeonhole principle with $|S| = t$.

Figure 1 compares CAS and the pigeonhole-principle-based seeds. Assume that we have verified that the two CAS seeds `AACC` and `TTGG` have confidence radii of $c_s = 2$. Therefore CAS can be guaranteed to find all valid mappings with just these two seeds, as $\sum_s c_s = 4 \geq t$. Using the pigeonhole principle, however, a mapper needs to select $t = 4$ non-overlapping seeds. It forces the mapper to pick short and repetitive seeds, making the mapper perform more local alignments.

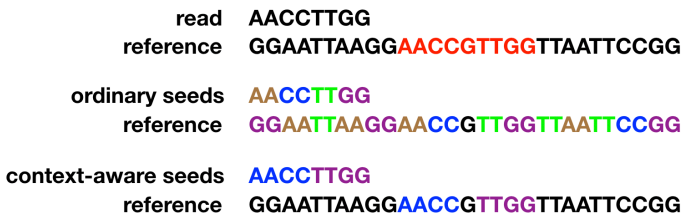


**Figure 1** Illustration of CAS. The upper part shows a read and a reference. Suppose that $t = 4$, i.e., we want to find all alignments of the read in the reference with fewer than 4 edits. There is only one such locally optimal alignment (marked as red). The middle part shows the seed extraction result with the pigeonhole principle, which splits the read into $t = 4$ seeds. This gives many seed locations and thus many alignments. With CAS (in the lower part), we can split the read into 2 long seeds while still guarantee to find all valid mappings. The two long seeds together have a total seed frequency of 2, drastically reducing the number of alignments.

We establish the theoretical foundation of CAS and demonstrate that with CAS future mappers can map reads more efficiently using fewer, longer and less frequent seeds without losing valid mappings. We also propose a suffix-trie-based CAS database construction algorithm that builds a CAS database from $T$ in linear time, based on which we design a greedy CAS seeding algorithm that extracts CAS from reads. We test the greedy CAS seeding algorithm against a state-of-the-art pigeonhole-principle-based seeding algorithm, Optimal Seed Solver (OSS), on an *E. coli* dataset.

## 2 Context-Aware Seeds

CAS reduces seed usage in read mapping by introducing a novel metric for seeds in $T$, the confidence radius. A seed $s$ in $T$ has a confidence radius $c_s$ if $c_s$ is a smallest value (a lower bound), such that all substrings in $T$ whose edit distance is smaller than $c_s$ must occur in $T$ within a small window where $s$ occurs. The window equals to extending $s$ by $c_s - 1$ letter(s) at both ends. For example, under $t = 2$, seed `AACC` in $T$ from Figure 1 has a confidence radius of 2. Any substring in $T$ whose edit distance to `AACC` equals 1 (e.g., `AAC`, `ACC`, `GAACC`, `AACCG`) locates within the 1-letter extended window of `AACC` (`GAACCG`). The confidence radius of each possible seed in $T$ can be computed by profiling $T$. CAS guarantees that all valid mappings of a read $R$ can be located, as long as the seeds $s$ extracted from $R$ collectively have a confidence radius of $\sum_s c_s > t$. Below, we give the formal definition of CAS and prove the correctness of CAS.

Let $s$ be a string in $T$ and $[l_1, l_2]$ be a pair of locations. We say string $T[l_1, l_2]$ is in the *vicinity* of $s$ under an integer $c$, if $\exists[l_{s1}, l_{s2}]$, where $l_1 - c < l_{s1} < l_{s2} < l_2 + c$ and $T[l_{s1}, l_{s2}] = s$. Furthermore, let seed $s$ be a substring of $R$ at $[l_{r1}, l_{r2}]$ ($s = R[l_{r1}, l_{r2}]$) and let $T[l_1, l_2]$ be a valid mapping of $R$. We say $T[l_1, l_2]$ is in the *vicinity of $s$ with regard to $R$* under $c$, if string $T[l_1 + l_{r1}, l_1 + l_{r\_2}]$ is in the vicinity of $s$ under $c$. If a valid mapping $T[l_1, l_2]$ is in the vicinity of $s$ with respect to $R$ under $t$, then $T[l_1, l_2]$ can be discovered by locally aligning $R$ against the surrounding text in $T$ at each seed location of $s$.

The pigeonhole principle states that by dividing $R$ into a set of $t$ non-overlapping seeds, denoted by $S$, then $\forall[l_1, l_2]$ where $T[l_1, l_2]$ is a valid mapping of $R$, there must be $s \in S$ where $T[l_1, l_2]$ is in the vicinity of $s$ with regard to $R$.

CAS seeks to retain the seed vicinity guarantee of the pigeonhole principle, where all valid mappings of a read $R$ are in the vicinity of its seeds with regard to $R$ under $t$, with **fewer than $t$ seeds**. Given two substrings $s$ and $s'$ of $T$ and a edit-distance threshold $t$, we say $s'$ is a neighbor of $s$ if $D(s, s') < t$. Assume that $s'$ is a neighbor of $s$ under $t$, CAS defines $s'$ as a *trivial neighbor* of $s$, if and only if $\forall[l_1, l_2]$ where $T[l_1, l_2] = s'$, $T[l_1, l_2]$ is in the vicinity of $s$ under $D(s, s')$. Otherwise CAS defines $s'$ as a *nontrivial neighbor* of $s$. Finally, CAS defines *the confidence radius* $c_s$ of $s$ as the minimum of 1) $t$ and 2) the minimum edit-distance between $s$ and all nontrivial neighbors of $s$. Since a seed is trivial to itself and is at least 1-edit-distance away from any other string, we have $t \geq c_s \geq 1$ for any seed $s$.

We now give the central theorem of CAS, the theoretical foundation that enables seed-and-extend mappers to find all valid mappings using fewer than $t$ seeds.

▶ **Theorem 1.** *Let $S$ be a set of non-overlapping seeds of a read $R$, if $\sum_{s \in S} c_s \geq t$, then $\forall[l_1, l_2]$ where $D(R, T[l_1, l_2]) < t$, $\exists s \in S$ where $T[l_1, l_2]$ is in the vicinity of $s$ with regard to $R$ under $t$.*

**Proof.** Assume that $T[l_1', l_2']$ is a valid mapping of $R$, where $D(R, T[l_1', l_2']) < t$. Further assume that $T[l_1', l_2']$ is not in the vicinity, with regard to $R$ under $t$, of any $s \in S$. In the minimum-edit-distance alignment between $R$ and $T[l_1', l_2']$, assume that the non-overlapping seeds $s_1$, $s_2$, …, $s_n$ of $R$ are aligned to the non-overlapping segments $s_{T1}$, $s_{T2}$, …, $s_{Tn}$ of $T[l_1', l_2']$, with $n = |S|$. Since $T[l_1', l_2']$ is not in the vicinity, with regard to $R$ under $t$, of any $s \in S$; and also because $c_{si} \leq t$ for all $i$; there does not exist $i$ where $s_{Ti}$ is in the vicinity of $s_i$, under $c_{si}$. Therefore, $s_{Ti}$ is a nontrivial neighbor of $s_i$ for all $i \in [1, n]$. Because $c_{si}$ is the minimum edit-distance between $s_i$ and any of its nontrivial neighbors, we have $D(R, T[l_1', l_2']) \geq \sum_i D(s_i, s_{Ti}) \geq \sum_s c_s \geq t$. $D(R, T[l_1', l_2']) \geq t$ contradicts the assumption that $T[l_1', l_2']$ is a valid mapping of $R$. Therefore such $T[l_1', l_2']$ does not exist. ◀

# 3 Construction of Confidence Radius Database

The confidence radius $c_s$ of each seed $s$ is stored in a table, called *the confidence radius database*. The confidence radius database only needs to be constructed once offline for a reference $T$.

Computing $c_s$ of seed $s$ involves finding the minimum edit distance to its nontrivial neighbors. Below we propose an algorithm that constructs the confidence radius database in $O(|\Sigma|^2 \cdot M)$ time, where $\Sigma$ is the alphabet set of $T$ and $M$ is the total number of neighbors of all strings in $T$ (up to length $P$ and under the edit distance threshold $t$).

The confidence radius database is constructed in two steps: first, we construct a neighbor database, which stores all neighbors of all seeds (up to length $P$) under the edit distance threshold $t$; then, we find the confidence radius of each from its neighbors. We prove that both steps can be done in $O(|\Sigma|^2 \cdot M)$ time.

## 3.1 Construction of the Neighbor Database

To find all neighbors of all substrings in $T$ (up to a maximum length $P$), we first build a $P$-level suffix trie of $T$, then find all neighbors of each seed in $Trie$ by systematically traversing the suffix trie in a top-down manner. Formally, let $Trie = (V, E)$ be a suffix trie of $T$ of a maximum depth of $P + t$. Let $r \in V$ be the root of $Trie$. Each node represents a substring in $T$, i.e., the string obtained by concatenating the letters on edges along the path from $r$ to $v$. We denote the edit distance between these two substrings corresponding to $u$ and $v$ as $D(u, v)$. We aim to solve the following problem:

▶ **Problem 2.** *Given a suffix trie $Trie = (V, E)$ and an integer $t$, to compute all pairs of nodes $u, v \in V$ such that $D(u, v) \leq t$.*

For any $v \in V$, $p(u)$ denotes the parent node of $v$ in $Trie$. $\sigma(p(v), v)$ denotes the letter on the edge between $v$ and $p(v)$, i.e., $(p(v), v) \in E$. We have the following lemmas.

▶ **Lemma 2.** *Let $u, v \in V$. Then $D(u, v) \leq t$ only if $D(p(u), p(v)) \leq t$.*

**Proof.** Proved in Landau and Vishkin [7] by enumerating and validating all possible scenarios. ◀

▶ **Lemma 3.** *Let $u, v \in V$. We have*

$$D(u, v) = \min \begin{cases} D(p(u), p(v)) + \delta_{uv} \\ D(p(u), v) + 1 \\ D(u, p(v)) + 1 \end{cases}$$

*where $\delta_{uv} = 1$ if $\sigma(p(u), u) \neq \sigma(p(v), v)$ and $\delta_{uv} = 0$ if $\sigma(p(u), u) = \sigma(p(v), v)$.*

**Proof.** This follows the dynamic programming algorithm for the edit distance problem. ◀

Lemma 2 shows that nodes are neighbors only if their parents are neighbors. Hence the neighbors of a child node must be the children of the neighbors of its parent node. Lemma 3 further shows that the edit distance between two children nodes can be computed in constant time, given the edit distances between one child and the parent node of the other child, as well as the edit distance between the two parent nodes.

We construct the neighbor database by traversing $Trie$ as follows: First, assign each node in $V$ an integral *rank* from $\{1, 2, \cdots, |V|\}$ following a top-down, left-to-right order. The root $r$ of $Trie$ has rank of 1, and then the children of children of $r$ have ranks of $2, 3, \cdots$, from the

leftmost child to the rightmost child, and so on. Nodes that are deeper in $Trie$ rank higher. Among nodes of the same depth, children of a higher ranking parent node rank higher. A breadth-first-search traversal of $Trie$ ranks all nodes.

For any $v \in V$, we define $X_v := \{u \in V \mid D(u, v) \leq t\}$ as the set of neighbors of $v$, including $v$ itself, and define $Y_v := \{D(u, v) \mid u \in X_v\}$ as the accompanying edit-distance set of $X_v$. For every neighbor node $u$ in $X_v$, $Y_v$ provides the edit distance between $u$ and $v$. We compute $X_v$ and $Y_v$ for each node $v \in V$ from low ranking nodes to high ranking nodes. Both $X_v$ and $Y_v$ are implemented as arrays.

The algorithm for constructing the neighbor database is summarized in Algorithm 1. We iterate through all nodes by rank from low to high. For each node $v \in V$, we iterate through all children of $v$. For each children node $v'$ of $v$, we compute $X_{v'}$ and $Y_{v'}$ of $v'$ based on the previously computed $X_v$ and $Y_v$ of $v$. Figure 2 illustrates the process of validating a candidate neighbor $u'$ of another node $v'$, based on the information of its parent node $v$ and the neighbor $u$ of $v$, where $u$ is also the parent of $u'$ (lines 4–17 in Algorithm 1). We prove that this algorithm maintains the following three invariants:

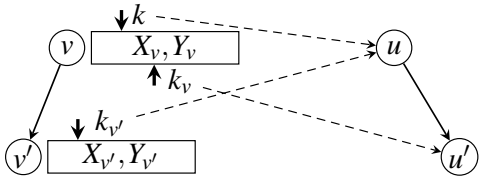


**Figure 2** Illustration of processing a single node $v$ (i.e., lines 4–17 of Algorithm 1).

**Algorithm 1** Linear Time Algorithm for Problem 2.

**Input:** Suffix trie $Trie = (V, E)$ and the edit-distance threshold $t$
**Output:** $X_v$ and $Y_v$ for each $v \in V$

0. Initialize $X_r$ and $Y_r$ for root $r \in V$.
1. **FOR** each node $v \in V$ in ascending order:
2. Initialize pointer $k_v = 0$ for arrays $X_v$ and $Y_v$.
3. Initialize arrays $X_{v'}$ and $Y_{v'}$ for each child $v'$ of $v$ as empty arrays.
4. Initialize pointer $k_{v'} = -1$ for arrays $X_{v'}$ and $Y_{v'}$ for each child $v'$ of $v$.
5. **FOR** $k = 0 \rightarrow |X_v|$:
6. **LET** $u := X_v[k]$.
7. **FOR** each child $u'$ of $u$:
8. **FOR** each child $v'$ of $v$:
9. **LET** $\delta = 1$ if $\sigma(u, u') \neq \sigma(v, v')$ and $\delta = 0$ if $\sigma(u, u') = \sigma(v, v')$. Compute $D_1 = Y_v[k] + \delta$, i.e., $D_1 = D(v, u) + \delta$.
10. Increase $k_v$ until $X_v[k_v] \geq u'$. **IF** we have $X_v[k_v] = u'$, i.e., $u' \in X_v$, **THEN** compute $D_2 = Y_v[k_v] + 1$, i.e., $D_2 = D(v, u') + 1$; otherwise set $D_2 = \infty$.
11. Increase $k_{v'}$ until $X_{v'}[k_{v'}] \geq u$. **IF** we have $X_{v'}[k_{v'}] = u$, i.e., $u \in X_{v'}$, **THEN** compute $D_3 = Y_{v'}[k_{v'}] + 1$, i.e., $D_3 = D(v', u) + 1$; otherwise set $D_3 = \infty$.
12. Compute $D(v', u') = \min\{D_1, D_2, D_3\}$. **IF** $D(v', u') < t$, **THEN** add $u'$ to the end of $X_{v'}$ and add $D(u', v')$ to the end of $Y_{v'}$.
13. **END FOR**
14. **END FOR**
15. **END FOR**
16. **END FOR**

1. For any node $v \in V$, array $X_v$ is always sorted according to their ranks, i.e., nodes that are added to $X_v$ are always in ascending order *w.r.t.* their ranks.
2. Right before processing node $v$ (i.e., before line 4 of Algorithm 1), $X_v$ and $Y_v$ are already computed and sorted *w.r.t.* their ranks.
3. Right after processing node $v$ (i.e., after line 17 of Algorithm 1), $X_{v'}$ and $Y_{v'}$ are computed and sorted *w.r.t.* their ranks for each child $v'$ of $v$.

The initialization step Algorithm 1 (line 2) computes $X_r$ and $Y_r$ for root node $r$. Its neighbors include all nodes whose depth in $Trie$ is no greater than $t$. The edit distance between $r$ to a neighbor node $u$ is simply the depth of $u$ minus 1 (we assume that root $r$ is at depth 1). Root $r$ is also in $X_r$ with $D(r, r) = 0$. Clearly, the first and the second invariant hold for root $r$.

In the main loop (lines 3–18), for a node $v \in V$, Algorithm 1 iterates through all of its children. For a child $v'$ of $v$, lines 4–17 compute $X_{v'}$ and $Y_{v'}$ of $v'$. Line 4–6 initialize the pointers that will be used to fetch the edit distances $D(v, u')$ and $D(v', u)$, which are stored in $Y_v$ and the partially computed $Y_{v'}$, respectively. Because $u$ ranks higher than $u'$, by the time of computing $D(v', u')$, $D(v', u)$ is already computed and stored in $X_{v'}$. $D(v', u')$ is then computed according to Lemma 3. Specifically, pointer $k$ tracks the position of $u$ in array $X_v$ (i.e., the index of $u$ in array $X_v$); pointer $k_v$ tracks the position of $u'$ in array $X_v$; and pointer $k_{v'}$ tracks the position of $u$ in array $X_{v'}$. Line 11 computes $D_1 := D(v, u) + \delta$, in which $D(v, u)$ is fetched from $Y_v$ indexed by $k$. Line 12 computes $D_2 := D(v, u') + 1$, in which $D(v, u')$ is fetched from $Y_v$ indexed by $k_v$. Line 13 computes $D_3 := D(v', u) + 1$, in which $D(v', u)$ is fetched from $Y_{v'}$ indexed by $k_{v'}$. Line 14 computes $D(v', u') := \min\{D_1, D_2, D_3\}$; adds $u'$ to $X_{v'}$ and adds $D(v', u')$ to $Y_{v'}$ if $D(v', u') < t$.

Algorithm 1 maintains the first invariant. For each child $v'$ of $v$, assuming $X_v$ is sorted, then neighbors are also added to $X_{v'}$ in a sorted manner, as Algorithm 1 iterates through neighbors ordered by $X_v$. Since $X_r$ is sorted for root $r$, given the inductive nature of Algorithm 1, we conclude that $X_v$ must be sorted for any $v \in Trie$.

Algorithm 1 maintains the third invariant. According to Lemma 2, a node $u' \in X_{v'}$ requires $u \in X_v$ for their parents $u$ and $v$. Any node $\bar{u}'$ whose parent $\bar{u} \notin X_v$ results in $\bar{u}' \notin X_{v'}$. Algorithm 1 iterates through all $u$ in $X_v$. Therefore, after line 17, all neighbors of child $v'$ must have been found, assuming the second invariant holds. The second invariant holds because all neighbors of $r$ are correctly defined during initialization. As the algorithm propagates, because of the inductive nature of Algorithm 1, the second invariant holds.

Let $M$ denote the member size of set $\{(u, v) \mid D(u, v) \leq t\}$. The complexity of Algorithm 1 is $O(|\Sigma|^2 \cdot M)$.

▶ **Theorem 4.** *Algorithm 1 computes $X_v$ and $Y_v$ for each $v \in V$ in $O(|\Sigma|^2 \cdot M)$ time.*

**Proof.** For each $v \in V$, lines 4–17 compute $X_{v'}$ and $Y_{v'}$ for each child $v'$ of $v$ in $O(|X_v| \cdot |\Sigma|^2 + \sum_{v':p(v')=v} |X_{v'}|)$ time. Since pointers of $k_v$ and $k_{v'}$ can only move forward, lines 12–13 cost $|X_v| + \sum_{v':p(v')=v} |X_{v'}|$ operations. Operations in lines 11–14 cost constant time. Hence, lines 7–17 cost $O(|X_v| \cdot |\Sigma|^2)$ operations, as the number of children of each node is bounded by $|\Sigma|$. The overall run time of Algorithm 1 is thus bounded by $\sum_{v \in V} O(|X_v| \cdot |\Sigma|^2 + \sum_{v':p(v')=v} |X_{v'}|) = O(|\Sigma|^2 \cdot M)$. ◀

With $|\Sigma|$ being a small constant (for example $\Sigma = \{A, C, G, T\}$ for DNA analysis), Algorithm 1 finds all $M$ neighbor pairs in $Trie$ in $O(M)$ time.

## 3.2 Computing the Confidence Radius Among Nontrivial Neighbors

The neighbor database stores both the trivial and nontrivial neighbors of each seed. However, CAS only requires the minimum edit distance to the nontrivial neighbors of each seed. In order to derive the confidence radius of each seed, we propose an augmentation to Algorithm 1, such that it computes the minimum edit distance to nontrivial neighbors while constructing the neighbor database. We prove that the augmentation does not increase the time complexity of Algorithm 1.

Within the neighbor array $X_v$ of a node $v$, let the sub-array $X_v^0$ store all trivial neighbors and $X_v^1$ store all nontrivial neighbors, where $X_v = X_v^0 \cup X_v^1$. By definition, the confidence radius of $v$ is computed as $c_v := \min_{u \in X_v^1} D(u, v)$. To compute $c_v$, instead of finding all nontrivial neighbors, $X_v^1$, we compute a subset $X_v^2 \subset X_v^1$, where $\min_{u \in X_v^2} D(u, v) = \min_{u \in X_v^1} D(u, v)$.

Let $u$ be a neighbor of $v$; we say $u$ is an *immediate* neighbor of $v$ if $u$ is a substring, or a superstring, or an overlapping string of $v$; otherwise we say $u$ is a *non-immediate* neighbor of $v$ (see Figure 3 for examples). Immediate neighbors are not necessarily trivial neighbors. If $u$ is a trivial neighbor of $v$, by definition, then $u$ must be an immediate neighbor of $v$. However, the opposite is not necessarily true, i.e., $u$ could be a substring of $v$ (an immediate neighbor) yet $u$ is nontrivial to $v$. Substring $u$ may appear at more locations in $T$ than $v$ does. It is easier to determine whether $u$ is an immediate neighbor to $v$ than whether $u$ is a trivial neighbor to $v$.

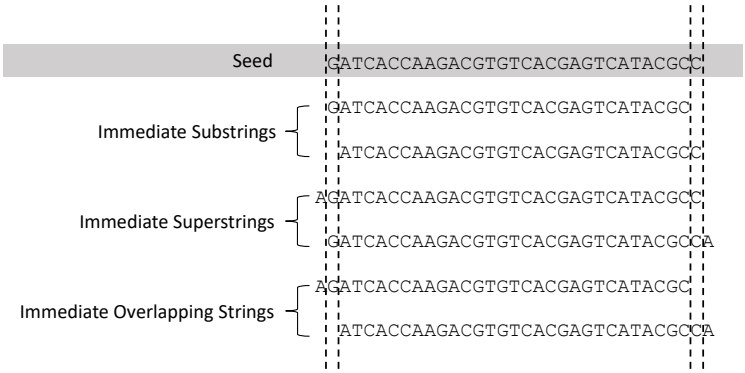


**Figure 3** Examples of trivial neighbors of a seed, including substrings, superstrings, and overlapping strings of this seed.

Let $X_v^2$ be the set of non-immediate neighbors of a node $v$. The minimum edit distance from $v$ to nontrivial neighbors of $v$ equals to the minimum edit distance between $v$ to neighbors in $X_v^2$. We prove this in Theorem 7. To prove Theorem 7, we first prepare the following two lemmas.

▶ **Lemma 5.** *If $u$ is a superstring of $v$, then $u$ is a trivial neighbor of $v$.*

**Proof.** Since $u$ is a superstring of $v$, for any location $[l_1, l_2]$ of $u$, $\exists [l_1, l_2]$ where $T[l_1, l_2] = v$ and $l_1 - D(u, v) \leq l_1 < l_2 \leq l_2 + D(u, v)$. By definition, $u$ is a trivial neighbor of $v$. ◀

▶ **Lemma 6.** *If $u$ is a substring or an overlapping string of $v$ and $u$ is a nontrivial neighbor of $v$, then $\exists w \in Trie$, where $w$ is neither an immediate neighbor nor a trivial neighbor of $v$, with $|w| = |v|$ and $D(v, w) \leq D(v, u)$.*

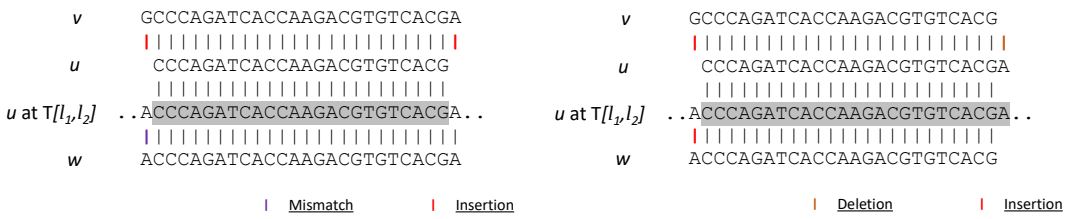


**Figure 4** Illustration of Lemma 6. The figure to the left shows an example where $u$ is a substring of $v$, while the figure to the right shows an example where $u$ is an overlapping string of $v$. Notice that in both figures, $w$ is always optimally aligned to $v$.

**Proof.** Since $u$ is a nontrivial neighbor of $v$, $\exists[l_1, l_2]$, where $T[l_1, l_2] = u$ but $T[l_1, l_2]$ is not in the $D(v, u)$-edit vicinity of $v$. We extract a substring $w$ within $T[l_1 - D(u, v), l_2 + D(u, v)]$, where $w$ locally and optimally aligns to $v$ in $T[l_1 - D(u, v), l_2 + D(u, v)]$, with $|w| = |v|$, as shown in Figure 4. Then $w$ must be a nontrivial neighbor of $v$ since $T[l_1, l_2]$ is not in the $D(u, v)$-edit vicinity of $v$. Because $w$ is optimally aligned to $v$ within $[l_1 - D(u, v), l_2 + D(u, v)]$, we have $D(w, v) \leq D(u, v)$. ◀

By combining Lemmas 5 and 6 we prove the following theorem.

▶ **Theorem 7.** *$c_v = \min_{u \in X_v^2} D(u, v)$, where $X_v^2$ is the set of non-immediate neighbors of $v$.*

**Proof.** Lemmas 5 and 6 state that for any nontrivial immediate neighbor $u$ of seed $v$, there must exist a nontrivial and non-immediate neighbor $w$ of $v$ where $D(w, v) \leq D(u, v)$. Therefore, by definition, we have $c_v = \min_{u \in X_v^2} D(u, v)$. ◀

We find the immediate neighbors, $X_v^3$, of each node $v \in Trie$, by checking if a neighbor $u \in X_v$ is a immediate substring, superstring or overlapping string of $v$. We associate with each node $v$ a new vector $Z_v := \{F(v, u) \mid u \in X_v\}$, where $F(v, u)$ stores the information of whether $u \in X_v^3$. With $X_v^2 = X_v \setminus X_v^3$, the updated workflow is illustrated in Figure 5.

Computation of $F(v, u)$ can be piggybacked on top of computing $D(v, u)$ in Algorithm 1. Given $u$ and $v$, $F(v, u)$ stores whether $v$ and $u$ possess any of the below *immediate conditions*: (1) $v$ is a prefix of $u$. (2) $v$ is a suffix of $u$. (3) $u$ is a prefix of $v$. (4) $u$ is a suffix of $v$. (5) $v$ is neither a prefix nor a suffix but a substring of $u$. (6) $u$ is neither a prefix nor a suffix but a substring of $v$. (7) A prefix of $v$ is a suffix of $u$. (8) A suffix of $v$ is a prefix of $u$.

From above immediate conditions, we deduce the immediate relationship between $v$ and $u$. With conditions 1–6, we can infer the superstring-substring relationship. With Condition 7–8, we can infer the overlapping relationship. If $v$ and $u$ qualifies none of the above immediate conditions, then they must be non-immediate neighbors.

For simplicity, we initialize each node as satisfying immediate conditions 1, 2, 3 and 4 to itself. We initialize the root node $r$ as a prefix to any of its neighbors; and any neighbors of $r$ as a suffix to $r$. Finally, $r$ is not an overlapping string or a substring of any neighbor.

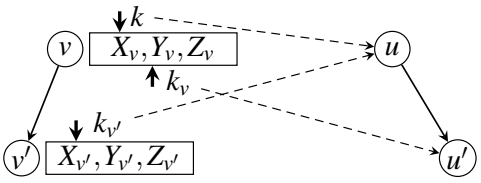


**Figure 5** Illustration of adding $Z_v := \{F(u, v) \mid u \in X_v\}$ to each node.

$F(u, v)$ can be computed in constant time if $F(p(v), p(u))$, $F(p(v), u)$ and $F(v, p(u))$ are known. For example, in Figure 5, $F(v', u')$ satisfies condition 1, only if (a) $v' = u'$ or (b) $F(v', u)$ satisfies condition 1. $F(v', u')$ satisfies condition 2, only if (a) $u' = v'$ or (b) $F(v, u)$ satisfies condition 2 and $\sigma(u, u') = \sigma(v, v')$. Conditions 3 and 4 are mirror cases of conditions 1 and 2, respectively with $v$ and $u$, $v'$ and $u'$ trading places. $F(v', u')$ satisfies condition 5, only if (a) $F(v', u)$ satisfies condition 5 or (b) $F(v', u)$ satisfies condition 2, while $v' \neq u'$ and $v$ is not root. Condition 6 is a mirror case of condition 5. $F(v', u')$ satisfies condition 7 only if (a) $F(v, u')$ satisfies condition 7 or (b) $F(v, u')$ satisfies condition 2, while $v \neq u'$ and $v$ is not root. Condition 8 is a mirror case of condition 7.

The computation of $F(\cdot, \cdot)$ is piggybacked on top of the computation of $D(\cdot, \cdot)$, as both methods use dynamic programming. Both methods require piror knowledge between the child-parent and parent-parent node pairs; and from prior results, both methods compute the new result of the child-child node pair in constant time. As a result, piggybacking the computation of immediateness does not increase the complexity of Algorithm 1.

Finally, the confidence radius of node $v$ equals $\min D(v, u)$ where $u \in X_v^2$, where $F(v, u)$ does not satisfy any of the immediate conditions. The confidence radius of a node can be found by simply scanning its neighbor array, which finishes in linear time. The overall complexity of constructing the confidence radius database is still $O(|\Sigma|^2 \cdot M)$.

The confidence radius database is stored in a $|T|$-by-$P$ table, where $P$ is user-provided. The $[x, y]$ entry of the table stores the $c_s$ of seed $T[x, x + y]$. In practice, $|T| \gg P$ and when needed, we can condense the confidence radius database into bit-vectors to reduce the table size. If necessary, when $|T|$ is large, we can sub-sample seeds only at fixed-length intervals to further reduce the storage footprint.

## 4 A Seeding Scheme with Context-Aware Seeds

While the major goal of this paper is to establish the theoretical framework of CAS, to test the effectiveness of CAS, we propose a greedy seed selection method, referred to as greedy CAS seeding. Greedy CAS seeding selects consecutive Maximum Exact Matching substrings (MEMs, which are seeds that cannot be further extended without bumping into errors) from a read as seeds. At the end of each MEM, greedy CAS seeding heuristically skips the next two base pairs, in an effort to skip potential errors. Greedy CAS seeding sorts seeds by their frequency from low to high, into $S_{raw}$. Then selects the minimum number of seeds $S$ from $S_{raw}$ in sequential order such that $\sum_{s \in S} c_s \geq t$. In the rare cases where there is insufficient number of CAS seeds such that $\nexists S$ with $\sum_{s \in S} c_s \geq t$, greedy CAS seeding reverts back to using the pigeonhole principle, by dividing the read into $t$ non-overlapping seeds.

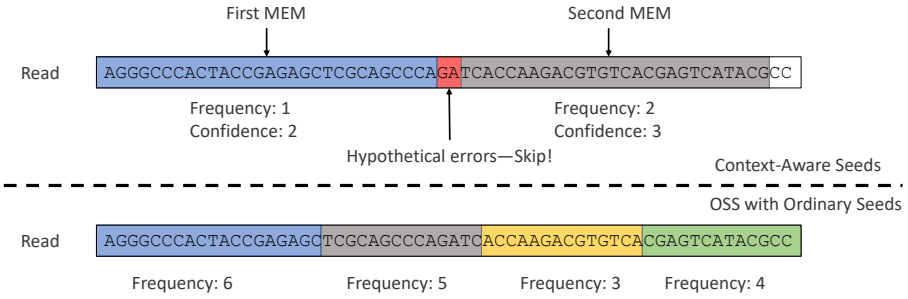


**Figure 6** An example of drawing context-aware seeds from a read.

Figure 6 compares the seed extraction results of greedy CAS seeding against the state-of-the-art, pigeonhole-principle-based seeding method, the Optimal Seed Solver (OSS) [15]. OSS has been previously shown that it generates the least frequent seeds, when compared to other pigeonhole-principle-based seeding methods, such as flexible-placement k-mers or spaced seeds. Figure 6 demonstrates both seeding methods in action under $t = 4$. Greedy CAS seeding is shown in the upper half while OSS is shown in the lower half. Compared to OSS, which uses a total of $t = 4$ seeds, greedy CAS seeding uses only two seeds. As a result, greedy CAS seeding can afford longer and less frequent seeds.

Greedy CAS seeding has a maximum complexity of $O(|R| + |S| \log(|S|))$ ($|R|$ denotes the length of $R$ while $|S|$ denotes the cardinality of set $S$). We use Burrows-Wheeler Transformation (BWT) array to index seeds. With BWT array, it takes $O(|s|)$ operations to access the seed database for seed $s$ and locate all seed locations of $s$. Given that $\sum_{s \in S} |s| \leq |R|$, and $|S| \leq t \ll |R|$, we conclude that the maximum complexity of greedy CAS seeding is $O(|R| + t \log(t))$.

## 5 Experiments

We benchmark greedy CAS seeding against OSS on the *E. coli* genome. We benchmark both seeding schemes on a 22-million, 100-bp *E. coli* read set from EMBL-EBI, ERX008638-1. We build a confidence radius database for *E. coli* genome with a maximum edit distance threshold $t = 5$ and a max seed length $P = 60$. We measure the effectiveness of both approaches by comparing the average total seed frequency of selected seeds under different edit distance thresholds $t = \{1, 2, 3, 4, 5\}$. The average total frequency is the sum of seed frequencies extracted from each read, averaged over all reads in the read set.

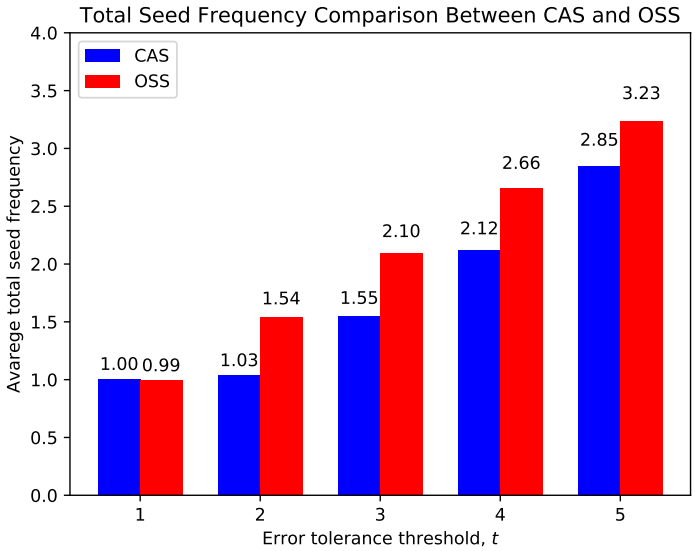


**Figure 7** Comparison between CAS and OSS in terms of total seed frequency, with various edit distance thresholds $t$.

Figure 7 shows the average total seed frequency comparison between the two approaches. OSS has slightly smaller total seed frequency (averaged over all reads) under $t = 1$, but it quickly increases, exceeding CAS at $t > 1$. OSS out performs CAS under $t = 1$ because greedy CAS seeding extracts seeds sequentially; while OSS scans through all possible MEM placements in a read and picks the least frequent placement. When $t$ gets larger, OSS is

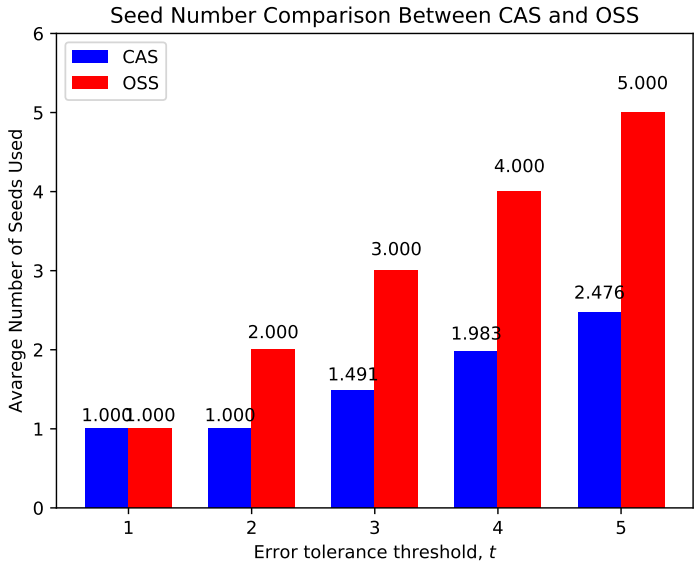


**Figure 8** Comparison between CAS and OSS in terms of average number of seeds used, with various edit distance thresholds $t$.

pressured to use more seeds, which leads to using shorter and more frequent seeds. To the contrary, greedy CAS seeding often uses fewer than $t$ seeds, as shown in Figure 8, which let it use longer and less frequent seeds. At $t = 4$, greedy CAS seeding out performs OSS by 20.3%.

CAS is expected to perform better on larger genomes. The *E. coli* genome is a small genome, which has only around 4.6 million base pairs. In comparison, the human genome has more than 3 billion base pairs. For small genomes, seeds becomes less frequent by nature. Therefore short seeds becomes acceptable as they are not as frequent as they are in larger genomes. We therefore expect CAS to perform better in larger genomes. However, due to practical (not theoretical) limitations in scaling up the construction of the confidence radius database on larger genomes (further elaborated in the Discussion section), we only demonstrate CAS on the *E. coli* genome.

While the focus of this paper is to establish the theoretical foundation of CAS, instead of providing a complete read mapping solution, it is worth mentioning that greedy CAS seeding (only the seeding mechanism) is more practical than OSS. OSS requires scanning through all substrings of $R$, which has a total size of $O(|R|^2)$, for seed frequencies. Combined with BWT, it takes at least $O(|R|^2)$ operations to collect all seed frequencies with OSS. Greedy CAS seeding, to the contrary, finishes in $O(|R| + t\log(t))$ time with $t \ll |R|$.

## 6 Discussion

Although Algorithm 1 finishes in $O(|\Sigma|^2 \cdot M)$ time, in practice, $M$ could be on the scale of trillions or more, for large and complex genomes. This is because for large genomes, the suffix trie is close to full in the first ten to twenty levels, where almost every permutation of letters exists. Nodes in these levels have large numbers of neighbors: the number of neighbors of a node $v$, equals to the number of unique strings formed by editing the string of $v$ with up to $t$ edits. After each edit, the resulting string is guaranteed to appear in $Trie$. This is further amplified by the exponentially-growing number of nodes in each level. In human genomes, there are more than one billion unique 15-base-pair suffixes. This means that for

human genomes, under $t = 4$, there could be more than 1 trillion total neighbors just for 15-base-pair suffixes. Maintaining metadata at such scale vastly exceeds the capacity of our currently available computational power. From our experiment, it takes around 300 CPU hours to compute the confidence radius database for the *E. coli* genome under $t = 5$ and $P = 60$ on a multi-cpu, mechanical hard drive system. However, it is worth noting that as a theoretical study, the database construction program is not fully optimized for speed and is currently I/O-bound due to frequently reading and writing neighbor information into neighbor arrays of nodes in $Trie$.

While there are many nodes (long suffixes) with fewer neighbors, given that Algorithm 1 traverses $Trie$ in a top-down manner, it is unavoidable to track the massive number of neighbors for short suffixes. This is an interesting algorithmic problem for future work.

CAS may be applied to situations other than NGS read mapping. For example, the idea of context-aware seeds may improve long-read mapping. Long reads suffer from high error rates [13, 3, 4]. Finding error-free seeds for long reads is very challenging [5]. CAS can serve as a metric measuring the likelihood of seeds having errors: if there exists a seed, $s$, with high confidence radius, it is highly likely that $s$ is free of errors. The likelihood of obtaining a reference-matching seed through many accidental errors is small.

Finally, CAS can be applied to develop probes for DNA and RNA identification. When designing probe sequences, it is important to make certain that the target sequence is unique in the genome [12, 2, 10]. It prevents probes from accidentally annealing to a similar sequences. CAS checks the existence of similar sequences by consulting the confidence radius database.

## 7 Conclusion

In this work, we proposed a new seeding framework, context-aware seeds (CAS). CAS extends the pigeonhole principle and guarantees finding all valid mappings with fewer seeds. CAS associates each seed $s$ with a confidence radius $c_s$, defined as a lower bound of edit distances towards nontrivial neighbors of $s$. We proved that the CAS can find all valid mappings of any read $R$, as long as its seeds $s$ satisfy $\sum c_s \geq t$.

We proposed a linear-time algorithm for constructing the confidence radius database. It computes the confidence radii of seeds by traversing the suffix trie of a reference. We experimented CAS on *E. coli* genome and compared it against the state-of-the-art pigeonhole-principle-based seeding scheme, OSS, and showed that CAS outperforms OSS by reducing the sum of seed frequencies by up to 20.3%.

This paper focuses on the theoretical aspects of CAS, especially how it extends the pigeonhole principle into using fewer seeds. Composing a practical solution of Algorithm 1 on larger genomes is an interesting-yet-separate problem for future work.

**Financial disclosure.** C. K. is co-founder of Ocean Genomics, Inc.

## References

1 Alexander Dobin, Carrie A Davis, Felix Schlesinger, Jorg Drenkow, Chris Zaleski, Sonali Jha, Philippe Batut, Mark Chaisson, and Thomas R Gingeras. STAR: ultrafast universal RNA-seq aligner. *Bioinformatics*, 29(1):15–21, 2013.

2 Eric Dugat-Bony, Eric Peyretaillade, Nicolas Parisot, Corinne Biderre-Petit, Faouzi Jaziri, David Hill, Sébastien Rimour, and Pierre Peyret. Detecting unknown sequences with DNA microarrays: explorative probe design strategies. *Environmental Microbiology*, 14(2):356–371, 2012.

**3** Ehsan Haghshenas, Faraz Hach, S Cenk Sahinalp, and Cedric Chauve. Colormap: correcting long reads by mapping short reads. *Bioinformatics*, 32(17):i545–i551, 2016.

**4** Ehsan Haghshenas, S Cenk Sahinalp, and Faraz Hach. lordFAST: sensitive and fast alignment search tool for long noisy read sequencing data. *Bioinformatics*, 35(1):20–27, 2018.

**5** Chirag Jain, Alexander Dilthey, Sergey Koren, Srinivas Aluru, and Adam M Phillippy. A fast approximate algorithm for mapping long reads to large reference databases. In *International Conference on Research in Computational Molecular Biology*, pages 66–81. Springer, 2017.

**6** Szymon M Kiełbasa, Raymond Wan, Kengo Sato, Paul Horton, and Martin C Frith. Adaptive seeds tame genomic sequence comparison. *Genome Research*, 21(3):487–493, 2011.

**7** Gad M Landau and Uzi Vishkin. Fast parallel and serial approximate string matching. *Journal of Algorithms*, 10(2):157–169, 1989.

**8** Ben Langmead and Steven L Salzberg. Fast gapped-read alignment with Bowtie 2. *Nature Methods*, 9(4):357, 2012.

**9** Heng Li. Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM. *arXiv*, 2013. `arXiv:1303.3997`.

**10** Qingge Li, Guoyan Luan, Qiuping Guo, and Jixuan Liang. A new class of homogeneous nucleic acid probes based on specific displacement hybridization. *Nucleic Acids Research*, 30(2):e5–e5, 2002.

**11** Ngoc Hieu Tran and Xin Chen. AMAS: optimizing the partition and filtration of adaptive seeds to speed up read mapping. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 13(4):623–633, 2016.

**12** Juexiao Sherry Wang and David Yu Zhang. Simulation-guided DNA probe design for consistently ultraspecific hybridization. *Nature Chemistry*, 7(7):545, 2015.

**13** Jason L Weirather, Mariateresa de Cesare, Yunhao Wang, Paolo Piazza, Vittorio Sebastiano, Xiu-Jie Wang, David Buck, and Kin Fai Au. Comprehensive comparison of Pacific Biosciences and Oxford Nanopore Technologies and their applications to transcriptome analysis. *F1000Research*, 6, 2017.

**14** Hongyi Xin, Donghyuk Lee, Farhad Hormozdiari, Samihan Yedkar, Onur Mutlu, and Can Alkan. Accelerating read mapping with FastHASH. *BMC Genomics*, 14(1):S13, 2013.

**15** Hongyi Xin, Sunny Nahar, Richard Zhu, John Emmons, Gennady Pekhimenko, Carl Kingsford, Can Alkan, and Onur Mutlu. Optimal seed solver: optimizing seed selection in read mapping. *Bioinformatics*, 32(11):1632–1642, 2015.