Searching the Referentially-compressed Genomes by Incomplete Patterns

Mohammad Nassef
Department of Computer Science
Faculty of Computers and Information
Cairo University, Egypt

Amr Badr
Department of Computer Science
Faculty of Computers and Information
Cairo University, Egypt

Ibrahim Farag
Department of Computer Science
Faculty of Computers and Information
Cairo University, Egypt

ABSTRACT

Genome banks contain precious biological information that is mostly not discovered yet. Biologists in turn are keen to precisely explore these banks in order to discover effective patterns (such as motifs and retro-transposons) that have a real impact on the function and evolution of living creatures. Because the modern genome sequencing technologies produce genomes in high throughputs, many techniques have emerged to store genomes in the lowest possible space. Reference-based Compression algorithms (RbCs) efficiently compress the sequenced genomes by mainly storing their differences with respect to a reference genome. Therefore, RbCs give very high compression ratios compared to the traditional compression algorithms. However, in order to search a compressed genome for specific patterns, it has to be totally decompressed, wasting both time and storage. This paper introduces searching for either exact or incomplete patterns inside the referentially compressed genomes without their complete decompression. The introduced search methodolgy is based on instantly searching subsequent sequences that are partially decompressed from the compressed genome. Moreover, the same search process is allowed to simultaneously search for multiple patterns, thus saving more resources. The experimental results showed noticeable performance gains compared to traditionally searching the same compressed genomes after their complete referential decompression.

General Terms:

Bioinformatics, Compression, Data Mining.

Keywords:

Reference-based Genome Compression; Partial Genome Decompression; Searching Compressed Genome; Incomplete Patterns; inCompressi.

1. INTRODUCTION

The last decade witnessed considerable research efforts that have been dedicated to study the genomic similarities and differences between individuals belonging to the same species. Examples of these projects are The HapMap Project [1], The 1000-Genome Project [2], and The Personal Genome Project (PGP) [3]. Consequently, massive genome resequencing techniques have been employed, resulting in very large genome banks. These banks are growing in a pace that outperforms the current storage technologies. Hence, the need to shrink the genomic sequences becomes urgent. The traditional compression algorithms cannot exploit the intensive similarities between the resequenced genomes, and so, its resulting compression ratios are not effective. Conversely, Reference-based Compression algorithms (RbCs) can effectively compress genomic sequences with very high compression ratios by basically storing the differences between the resequenced genomes.

To search a genome that is referentially compressed, biologists traditionally had to (1) completely decompress this genome into file(s), (2) load it from its file(s), and then (3) search it using some exhaustive search technique. For simplicity, this paper refers to these three steps as the *Traditional Search of Compressed Genome* procedure, (TSCG) for short. Unfortunately, TSCG takes considerable disk storage and runtime while building the original plain genome through a complete referential decompression process. Moreover, it consumes more memory to load and cache the decompressed genome. After that, it spends more runtime to search the decompressed genome.

Wandelt and Leser [4] presented the possibility of searching the referentially compressed genomes for strings (patterns). To prove its feasibility, the authors have illustrated their search algorithm over genomes that are referentially compressed using their own Reference-based Compression algorithm (RbC) [5]. Their algorithm depends on creating a repository that contains an index structure for the first genome, followed by the matches and differences of each successively compressed genome (with respect to the first genome as a reference). Because the big index structure of the first genome usually has a significant size, their actual compression gain takes effect after compressing approximately five to ten genomes, where just the matches and differences of every subsequent genome with respect to the first genome are stored. Eventually, the repository's overall size will be smaller than the total size of the original plain individual genomes. Their search algorithm is fast in searching the overall repository (all genomes) for a given pattern, however, it cannot search a predetermined genome. Moreover, removing an already compressed genome from the repository is not implemented yet. Furthermore, their algorithm does not support searching by incomplete patterns.

Chern et al. at Stanford University [6] developed an RbC that completely avoids the index structure overhead by exploiting the sliding window of the LZ77 algorithm [7]. For simplicity, this paper refers to their algorithm as "Stanford". Stanford introduces better compression ratios in many cases compared to the other RbCs such as GRS [8] and GReEn [9]. In addition, Stanford compresses every genome separately, and so, every compressed genome can be searched individually. Moreover, the compression gains of Stanford are directly noticeable for every compressed genome. For example, using the hg18 human genome as a reference, Stanford compresses the James Watson's (JW) human genome from 2,991 megabytes (MB) down to 6.99 MB. Similarly, storing 1,000 non-compressed human genomes costs 2,991,000 MB, whereas

Stanford compresses them into 9,974 MB with compression ratio exceeding 99% by keeping one of them as a plain reference genome (2,991 MB) and referentially compressing the other 999 genomes (6,983 MB) . In other words, Stanford can utilize the same plain space of one thousand genomes (2,991,000 MB) to store 427,469 referentially compressed genomes in addition to one non-compressed genome as a reference.

Genome analysis algorithms usually start their work by a preparatory step to load a part of or all the non-compressed genomes from disk into memory. In addition to the cost of the original disk storage, this preparatory step costs both runtime and memory consumption overhead. Compressing genomes using any RbC reduces their disk storage overhead, however, this advantage will be instantly lost when an analytical algorithm asks for a complete genome decompression in order to start its usual work. This resource optimization problem motivated us to think in a way to pick sequences from a referentially compressed genome without its complete decompression. Overcoming this problem can effectively reduce one or all of the disk storage, memory consumption, and runtime overheads. In [10], an algorithm, namely inCompressi, is applied to efficiently eliminate all these overheads by performing partial decompressions of genomes that are referentially compressed using the Stanford algorithm. In addition, inCompressi is exploited to perform a complete genome decompression that is more efficient than the original Stanford's decomresssion. Basically, inCompressi makes a complete genome decompression through performing successive partial decompressions of the compressed genome. Thus, it reduces the memory consumption and the runtime overhead resulting from managing that memory. Moreover, the same article presented inCompressi-Blocks that works exactly like inCompressi, but by performing partial decompressions in relatively short fixed-length blocks. Therefore, it takes lower runtime and memory consumption. This paper uses the term ”inCompressi” to simply refer to the inCompressi-Blocks algorithm.

This article introduces an enhancement to inCompressi in order to support searching by either exact or incomplete patterns inside the referentially compressed genomes without their complete decompression. The enhancement introduced to inCompressi depends on searching the blocks resulting from its subsequent partial decompressions. So, inCompressi consumes memory that is proportional to an individual fixed-length block, and with disk storage just for the compressed genomes in addition to a plain reference genome.

The following subsection mentions the possible alphabetic characters representing the genomic sequences. The next two subsections give a brief overview of Stanford and inCompressi. After that, Section 2 gives more implementation details about how the pattern search is implemented in inCompressi. The experimental results are discussed in Section 3. Section 4 concludes the paper and highlights interesting future work topics.

1.1 Alphabetical Representation of the Sequenced Genomes

Genomic sequences basically consists of four chemically different base pairs (Adenine, Cytosine, Guanine, and Thymine). These base pairs are respectively represented by the alphabetical characters A, C, G, and T. Because the modern genome sequencing techniques are not exact, some base pairs of the sequenced genome may have multiple values. According to the level of uncertainty, the suspecious base pair is represented by a different character, namely an ambiguity code. For example, character N represents a base pair that is completely not recognized, while character R represents a base pair that can be A or G, and character Y represents base pair that can be C or T. The complete list of the fifteen possible ambiguity codes can be found in [11]. By considering the four main alphabetical characters, a genomic sequence with two characters can have 4^2 different possible values: AA, AC, AG, AT, CA, CC, CG, CT, GA, GC, GG, GT, TA, TC, TG, or TT. However, when considering all the fifteen ambiguity codes, a two-characters sequence can have one of 15^2 possible values. Generally speaking, according to the exactness of the genomic sequence, a sequence with n characters can be a pattern from 4^n or 15^n possible patterns.

1.2 Stanford Overview

Let X be the target chromosome to be referentially compressed with respect to a reference chromosome Y. Stanford performs compression by encoding the longest matches (H_s) between X and Y, and similarly encoding the differences resulted from merging H_s whenever possible. Differences include substitutions (S_s), insertions (I_s), and deletions (D_s) to be applied to H_s to recover X from Y. Two sequences X_i^j and Y_a^b form the longest match if they have the same characters, and $X_{j+1} \neq Y_{b+1}$. RC(X,Y) denotes Stanford's Referential Compression of X with respect to Y:

$$Encoded(H_s, S_s, I_s, D_s) = RC(X, Y) \tag{1}$$

Conversely, X is reproduced by performing Stanford's complete Referential Decompression (Eqn 2). Stanford performs decompression by firstly decoding all the matching instructions (H_s) and all the differences (S_s, I_s, D_s). It then starts building X by copying the matches H_s from Y, and applying all differences in their reversed order: (D_s, then I_s, then S_s). Obviously, that traditional genome decompression results in huge runtime and storage overhead, especially if a biologist is just interested in specific sequences of the compressed genome.

$$X = RD(Y, Encoded(H_s, S_s, I_s, D_s)) \tag{2}$$

For more details about the exact functionality of the Stanford's compression and decompression, please refer to [6], or alternatively refer to subsection 1.1 to 1.3 in [10] for more illustrative examples.

1.3 inCompressi Overview

inCompressi refers to picking partial sequences from compressed genomes or chromosomes without their complete decompression. Moreover, inCompressi can also decompress an entire chromosome via successive partial decompressions. The key strength behind inCompressi is summarized in how it can efficiently re-adjust the offset of the needed sequence to determine its location inside the reference chromosome without the real execution of differences preceding this sequence inside the target chromosome. In turn, inCompressi adjusts the offsets of the differences falling inside the picked sequence before applying them. Moreover, inCompressi can efficiently handle cases where the needed sequence spans multiple match instructions.

Common to Stanford decompression, inCompressi loads and decodes the match instructions and differences of the overall compressed genome. Unlike Stanford decompression, inCompressi can determine the exact matching instruction(s) that contain the queried sequence. It can then pick that sequence from the reference chromosome, and then applies all the differences falling inside this sequence to become an identical part of the target genome. The key strength behind inCompressi is summarized in how it can efficiently re-adjust the offset of the needed sequence to determine its location inside the reference chromosome without the real execution of differences preceding this sequence. Consequently, inCompressi also adjusts the offsets of the differences falling inside the picked sequence before applying them.

The reader can find more details about the inCompressi algorithm in subsection 2.1 in [10]. Also, (Figure 1) contains an illustrative diagram about how inCompressi operates. Using the reference genome, inCompressi decompresses fixed-length block(s) from the compressed genome with fast response and low memory consumption compared to the Stanford's complete decompression. These blocks could further be saved to rebuild the

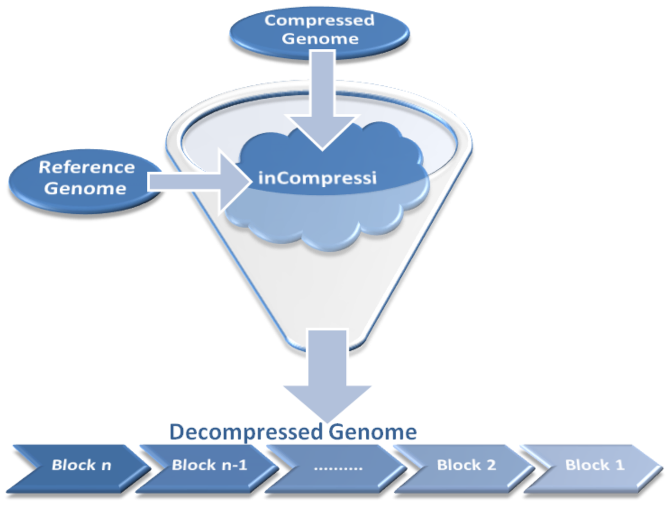

Fig. 1: *inCompressi's block decompression of a referentially compressed genome. Using the reference genome, inCompressi starts building fixed-length blocks of the compressed genome. Every established block could further be processed by any genome analysis algorithm.*

original genome, or alternatively, could be searched for specific patterns.

2. METHODS

This section introduces the search techniques added to inCompressi. Inherently, inCompressi is able to partially decompress individual successive blocks of the referentially compressed genome to pick either a small sequence, an entire chromosome, or the entire genome. So, whatever inCompressi decompresses, it can be efficiently searched by instantly searching its decompressed blocks one by one.

2.1 Searching for Exact Patterns

The inCompressi's search depends on performing subsequent partial decompressions in fixed-length blocks, and instantly searching every block using any exhaustive search technique for possible occurrences of the given pattern. Because the given pattern may be scattered over two blocks (starts at the end of the current block, and ends at the beginning of the next block), inCompressi searches every subsequent block with the tail of its previous block concatenated. For example, given a pattern P=”ACGT” to search for in a referentially compressed genome, inCompressi performs subsequent partial decompressions that traverses and decompresses the overall genome block by block. Because an occurrence of P may be scattered between two blocks (for example, a block that ends with ”ACG”, while its successive block starts with ”T”), inCompressi searches for P in every successive block, but after appending the last three characters of its preceding block to its beginning.

2.2 Searching by Incomplete Patterns

Biologists sometimes become rationally interested in searching genomes by incomplete patterns [11]. An incomplete pattern contains wildcard character(s) that can be replaced by any of the genomic base pairs (such as A, C, G, T ... etc). For example, a pattern ”CT - AG” has one wildcard character, thus, its exact value can be ”CT<u>A</u>AG”, ”CT<u>C</u>AG”, ”CT<u>G</u>AG”, or ”CT<u>T</u>AG”.

From one hand, for a pattern like ”CTAG - - - - - TCT”, biologists might be interested in exploring all the existing sub-patterns of the unknown five characters in a given sequence. In addition, they might be interested in knowing what are the most dominant characters in these sub-patterns, which in turn form a dominant exact pattern (Consensus pattern). On the other hand, biologists might be interested in knowing the occurrences of the overall pattern regardless of the value of the five characters in middle.

Therefore, in addition to searching for exact patterns inside referentially compressed genomes, inCompressi allow searching by incomplete patterns that are involved with successive wildcard characters. This possibility is not supported neither by the exhaustive search algorithms nor by the index-based search technique provided in [4]. (Listing 1) shows how searching by incomplete patterns is implemented in an efficient way so that it has a modest effect on either the memory consumption or the runtime. More specific, if inCompressi is searching by the incomplete pattern P = ”CTAG - - - - - TCT”, then it divides P into two exact sub-patterns: P1 = ”CTAG” and P2 = ”TCT”. inCompressi can then exhaustively search for the exact sub-pattern P1 in every decompressed block, and, for every occurrence of P1, inCompressi skips five characters (corresponding to the wildcard characters), then checks for the existence of P2.

Listing 1: *Searching a referentially compressed genome for incomplete patterns using inCompressi*

```
inCompressi_Incomplete_Search(Pattern[0..m-1],
Compressed_Genome)
{
 1:    hitsList = []
 2:    count = 0
 3:    dashStart = Pattern.Find('-')
 4:    dashCount = Pattern.Count('-')
 5:    dashEnd = dashStart+dashCount
 6:    Ptrn1 = Pattern[0..dashStart]
 7:    Ptrn2 = Pattern[dashEnd..m-1]
 8:    prevBlk = ” ”
 9:    foreach decompressed block Blk[0..t-1] in
Compressed_Genome:
10:         newBlk = prevBlk[t-(m-1)..t-1] + Blk
11:         indx1 = find next Ptrn1 in newBlk
12:         while Ptrn1 is found in newBlk:
13:              indx2 = indx1 + dashEnd
14:              if (newBlk[indx2..indx2+len(Ptrn2)] equals Ptrn2):
15:                   append indx1 to hitsList
16:                   count = count + 1
17:              indx1 = find next Ptrn1 in newBlk
18:         prevBlk = Blk
19:    return (hitsList,count)
}
```

Although searching by incomplete patterns is not supported by the exhaustive search algorithms, it is possible to exploit them to find the same incomplete pattern P by firstly searching for all occurrences of P1, then searching for all occurrences of P2, and then intersecting the two lists of occurrences. This intersection only keeps every occurrence of P1 that is followed by an occurrence of P2 with any five characters in between. (Listing 2) shows how searching by incomplete patterns is traditionally implemented. It is clear that this procedure carries much more overhead compared to either the traditional exact pattern search or the inCompressi's search.

Listing 2: *An algorithm for traditionally searching a decompressed genome for occurrences of an incomplete pattern.*

```
Incomplete_Pattern_Search(Pattern[0..m-1], Genome[0..n-1])
{
 1:    hitsList = []
 2:    cntOfPtrn = 0
 3:    dashStart = Pattern.Find('-')
 4:    dashCount = Pattern.Count('-')
 5:    dashEnd = dashStart+dashCount
 6:    P1 = Pattern[0..dashStart]
 7:    P2 = Pattern[dashEnd..m-1]
 8:    (HitsOfP1, cntOfP1) = Exhaustive_Search(P1, Genome)
 9:    (HitsOfP2, cntOfP2) = Exhaustive_Search(P2, Genome)
10:    i = j = 0
11:    while(i < cntOfP1 AND j < cntOfP2):
12:        if(HitsOfP1[i] + dashEnd == HitsOfP2[j]):
13:            hitsList[cntOfPattern] = HitsOfP1[i]
14:            cntOfPattern = cntOfPattern + 1
15:            i = i + 1
16:            j = j + 1
17:        else:
18:        if(HitsOfP1[i] + dashEnd > HitsOfP2[j]):
19:            j = j + 1
20:        else:
21:            i = i + 1
22:    return (hitsList,cntOfPattern)
}
```

2.3 Simultaneous Search by Multiple Patterns

Assuming that there is enough space to completely decompress a genome, it can then traditionally be decompressed only once, but exhaustively searched multiple times for many patterns. Conversely, inCompressi's search costs performing subsequent block decompressions of the overall genome for each given pattern. Consequently, searching the same compressed genome for three patterns redundantly triples the decompression cost. In order to avoid this redundant decompression overhead, inCompressi allows simultaneous search by a pool of patterns during one genome decompression. Thus, for each block partially decompressed from the given genome, inCompressi instantly searches this block for occurrences of all the given patterns. Eventually, the total runtime of inCompressi's search is significantly reduced, because the overall genome is decompressed once, but searched multiple times for more than one pattern. In other words, the more the number of patterns to simultaneously search for, the higher the reduction in runtime.

3. RESULTS AND DISCUSSION

3.1 The Experimental Environment

The experiments of this article are performed using the same two datasets used in [10]. The first dataset contains the Arabidopsis thaliana *TAIR8* (TAIR8 Website) and *TAIR9* (TAIR9 Website) genomes with approximate size of 120 MB for each. The second dataset consists of the *hg18* (HG18 Website) and the *yh* (YH Website) human genomes with approximate size of 3,000 MB for each. The TAIR experiments were performed on a machine with an Intel Core2 Duo CPU @ 2.10 GHz with 3.77 gigabytes (GB) of RAM. Because the human genomes are much longer than the TAIR genomes, the human genome experiments were executed on a machine with an Intel Core i5 CPU @ 2.50 GHz with 3.89 GB of RAM. Both machines run on LinuxMint-14 OS, and only one core is used by the implementated algorithm.

This section analyzes the performance of inCompressi while searching the referentially compressed TAIR9 and yh genomes through successive block decompressions. The experimental results in [10] showed that selecting a specific block length depends on how fast the architecture that inCompressi is running on, and in turn, how well inCompressi will perform. For example, the best block length for the "2.10 GHz" machine was 100,000 characters, whereas, the best block length for the "2.50 GHz" machine was 1,000,000 characters. Therefore, the search experiments mentioned by this article use these same block lengths.

3.2 Implementation and Validation

In general, a good implementation of an exact algorithm has to give correct and identical results regardless of the programming language used to implement it. Because Stanford and inCompressi are complicated enough, they were implemented in Python. Implementing an algorithm in Python, as an interpreter-based language, is relatively simpler but slower than implementing the same algorithm using compiler-based languages (like C/C++). More details about the implementation of inCompressi can be found at this link: `http://mnassef.comuf.com/implementations/inCompressi.php`.

From one hand, in order to validate the search results of inCompressi, the compressed genome had to be traditionally searched after completely decompressing it (TSCG procedure). On the other hand, to measure the performance gains of inCompressi compared to TSCG, searching the overall genome by TSCG has to be done using the same exhaustive search algorithm used by inCompressi to exhaustively search the individually decompressed blocks. This urged us to implement TSCG in Python as well. It is worthy to note that Python searches text exhaustively using an efficient combination [12] between both Boyer-Moore [13] and Horspool [14] fast text search algorithms.

3.3 Searching for Exact Patterns using inCompressi

This subsection introduces an example of searching the comperssed TAIR9 genome for an exact pattern using inCompressi. The search results in (Table 1) show the number of occurrences (hits) as well as the offsets of pattern "AAACCCGGGTTT" at every chromosome of the referentially compressed TAIR9 genome.

Table 1. : *Searching the referentially compressed TAIR9 genome for pattern "AAACCCGGGTTT" using inCompressi. The 1st column refers to the searched chromosome. The 2nd and 3rd columns refer respectively to the number of hits as well as their offsets inside every chromosome.*

Chromosome	*# of Hits*	*Offsets of Hits*
1	3	1,061,791 - 11,468,046 - 20,675,696
2	4	6,442,547 - 7,739,463 - 9,915,426 - 15,193,105
3	1	17,548,995
4	1	15,444,888
5	5	55,415 - 6,951,615 - 9,403,854 - 17,806,902 - 18,684,836
Total Hits	14	

3.4 Searching by Incomplete Patterns using inCompressi

Using inCompressi, the compressed TAIR9 genome is searched by three incomplete patterns: "AACCGGTT - - - AACCGGTT", "AACCGGTT - - - - - AACCGGTT", and "AACCGGTT - - - - - - - - - - AACCGGTT". All these three patterns have the same start and end subpattern "AACCGGTT", however, they have different number of wildcards. Each of these patterns is found exactly once throughout the TAIR9 genome. The 1st and 2nd patterns are found respectively with exact values "AACCGGTT**TTG**AACCGGTT" and "AACCGGTT**TTTTA**AACCGGTT" at offsets 15,042,150 and 12,042,845 in the TAIR9's 2nd chromosome. The 3rd pattern is found with exact value "AACCGGTT**TCTCCTCAAT**AACCGGTT" at offset 21,618,420 in the TAIR9's 5th chromosome.

(Table 2) shows the number of hits, their offsets, and their exact values of searching by pattern ”TGCGA - - - ACGCT” at every chromosome of the referentially compressed TAIR9 genome. It is clear from the last column in this table that the inCompressi’s incomplete-pattern search provides biologists with a smooth investigation of the pattern they would be exploring, especially the derivation of consensus patterns. For example, the wildcards of the later pattern has a dominant value of ”TGA”, because ’T’ is found 8 times as the first wildcard, ’G’ is repeated 6 times as the middle wildcard, and ’A’ exists 9 times as the last wildcard. Hence, the consensus pattern of the incomplete pattern ”TGCGA - - - ACGCT” is ”TGCGA**TGA**ACGCT”.

Table 2. : *Searching the referentially compressed TAIR9 genome by the incomplete pattern ”TGCGA - - - ACGCT” using inCompressi. The* 1^{st} *column refers to the searched chromosome. The next columns refer respectively to the number of hits, their offsets, and their exact values inside every chromosome.*

Chromosome	*# of Hits*	*Hit Offset*	*Hit Value*
1	2	17,531,396	TGCGA**TGA**ACGCT
		29,972,472	TGCGA**GGA**ACGCT
2	2	8,406,311	TGCGA**GCA**ACGCT
		9,116,714	TGCGA**TGC**ACGCT
3	1	10,735,231	TGCGA**GGA**ACGCT
4	2	2,574,160	TGCGA**ATC**ACGCT
		11,956,657	TGCGA**TCG**ACGCT
5	8	1,961,616	TGCGA**TTA**ACGCT
		1,970,334	TGCGA**TTA**ACGCT
		10,169,322	TGCGA**TCA**ACGCT
		13,335,109	TGCGA**TGA**ACGCT
		15,068,650	TGCGA**AGG**ACGCT
		15,708,290	TGCGA**GAT**ACGCT
		17,422,227	TGCGA**AAA**ACGCT
		24,440,861	TGCGA**TCC**ACGCT
Total Hits	15	**Consensus:**	TGCGA**TGA**ACGCT

In addition to searching the overall genome, inCompressi allows searching by incomplete patterns inside specific chromosomes. For instance, searching the TAIR9’s 1^{st} chromosome for the pattern ”TTTTT - - - - - - - - - - AAAAA” resulted in 1,416 different hits. The first hit is ”TTTTT**AGTTGTGGCG**AAAAA”, whereas the last hit is ”TTTTT**GACACACCAC**AAAAA”.

3.5 inCompressi versus TSCG

Recall that the total disk storage used by inCompressi is only for storing the compressed genomes in addition to the reference genome, whereas the disk storage needed to perform TSCG is for every distinct non-compressed genome to be searched. So, for searching 1,000 human genomes, inCompressi can manipulate these genomes while they are compressed in apporximate total space of 9,993 MB (3,000 MB for a non-compressed reference genome, and 6,993 MB for the 999 compressed genomes), whereas, for TSCG to traditionally search the same genomes, they need to be stored without compression in 3,000,000 MB approximately. Moreover, for a fair comparison between inCompressi and TSCG, it is assumed that these genomes are already compressed using some reference genome, and that, for TSCG to traditionally search them, it has to wait for the complete decompression of every genome in order to be searchable.

(Table 3) and (Table 4) respectively shows the experimental results performed on the TAIR9 and yh human genomes. In both tables, column (a) shows the runtime of TSCG while traditionally searching the plain genome starting by its complete referential decompression, whereas column (b) shows inCompressi’s runtime while directly searching the same compressed genome without its prior complete decompression. Every pattern in those tables has two rows showing the runtime and memory used to search for this pattern. Column (a) of both tables is divided into four sub-columns. The first three sub-columns have measures of the memory consumption and runtime of the stages needed to traditionally search a compressed genome using TSCG. The first stage is the complete genome decompression using inCompressi. The second stage measure the average resources needed to load the decompressed genome into flat contiguous streams, while the final stage measures the resources consumed to traditionally search the entire cached genome. Finally, the forth sub-column accumulates the runtime and memory consumed by the preceding three stages.

(Table 3) lists and (Figure 2) depicts the results of multiple search experiments over the TAIR9 genome that has been originally compressed using the TAIR8 genome as a reference. Searching the compressed TAIR9 genome for any of the five exact patterns below using TSCG took 5.44 sec on average (after adding the decompression and caching runtimes). Alternatively, inCompressi took 1.92 sec on average to directly search the TAIR9 compressed genome for the same exact patterns. On the other hand, while searching for the five incomplete patterns below, TSCG took 8.60 sec on average, whereas inCompressi took around 2.26 sec. So, inCompressi outperforms TSCG while searching for either exact or incomplete patterns. Regarding the memory consumption, TSCG consumes more than 70 MB, whereas inCompressi consumes only 8 MB in normal cases. Patterns that are very common (such as ”AAAA” and ”TTT - - - - - - - - - - - - - - - - TTT”) exceptionally consume more memory, because the more the occurrences of a pattern, the larger the list of hits to be maintained. However, inCompressi still consumes memory that is noticeably lower than TSCG.

(Table 4) lists and (Figure 3) depicts the results for the same experiments of (Table 3), but this time on the yh human genome that has been compressed with respect to the hg18 human genome. It is clear from (Table 4) that inCompressi is still highly compititor to TSCG. Compared to TSCG, inCompressi’s search still has lower runtime and memory consumption, and zero disk storage. Although TSCG takes 463 sec on average to search for an exact pattern, inCompressi takes only 406 sec. Searching by incomplete patterns costs 505 sec on average, however, inCompressi takes around 402 sec on average. Regarding memory consumption, TSCG consumes at least more than one GB of memory. In exceptional cases while searching for very common patterns it consumes around two GB of memory. Alternatively, inCompressi consumes around 556 MB of memory, and in exceptional cases it still consumes memory that is significantly lower than TSCG. So, at the worst case, if inCompressi consumes runtime and memory as equal as TSCG, inCompressi consumes no disk space.

It is worthy to note that the time taken to search for a pattern differs from one pattern to another according to how many partial or identical occurrences of that pattern exist in the given genome. In other words, the higher the similarity of the pattern to sequences of the given genome, the more runtime to be spent by the search algorithm to determine whether these sequence are identical to the given pattern. This explains why patterns such as ”AAAA” takes more search time than searching for pattern ”ACGTACGT”.

3.6 Simultaneous Search by Multiple Patterns

By searching a compressed genome for one pattern, inCompressi is highly competitor to TSCG with respect to both memory consumption and runtime. However, when it comes to handling simultaneous pattern search, TSCG can search a given genome with multiple patterns after performing only one complete decompression for that genome. To keep compitition with TSCG,

Table 3. : *The runtime and storage needed to search the TAIR9 genome using (a) TSCG and (b) inCompressi. Every pattern has two rows, the* 1st *row shows the average runtime in milliseconds (ms), while the* 2nd *row shows the memory consumption in MB. Notice that the TAIR9's complete decompression time (2.25 sec on average) is added to the runtime of TSCG, because both TSCG and inCompressi start working on a compressed TAIR9 genome. The runtime of inCompressi represents the overall search time inside the referentially compressed genome.*

Pattern			**(a)** TSCG				**(b)** inCompressi
Index	Value	Hits	Decompression	Caching	Searching	**Total**	
1	**TTTTTCCCCC**	260			245 ms 5 MB	4,795 ms 70 MB	1,263 ms 8 MB
2	**ACGTACGT**	1,128			543 ms 5 MB	5,093 ms 70 MB	1,576 ms 8 MB
3	**AACCGGTT**	2,708			1,079 5 MB	5,629 ms 70 MB	2,120 ms 8 MB
4	**AAACCCGGGTTT**	14			1,112 5 MB	5,662 ms 70 MB	2,133 ms 8 MB
5	**AAAA**	2,024,672	2,250 ms	2,300 ms	1,465 24 MB	6,015 ms 89 MB	2,488 ms 70 MB
6	**ACGT - - - ACGT**	665	36 MB	29 MB	1,738 9 MB	6,288 ms 74 MB	1,765 ms 8 MB
7	**ACGT - - - TGCA**	985			2,000 10 MB	6,550 ms 75 MB	1,764 ms 8 MB
8	**TATAT - - - - - CGCGC**	12			1,250 8 MB	5,800 ms 73 MB	1,745 ms 8 MB
9	**TTTTT - - - - - AAAAA**	7,741			2,557 18 MB	7,107 ms 83 MB	1,841 ms 8 MB
10	**TTT - - - - - - - - - - - - - - - - TTT**	317,053			12,669 107 MB	17,249 ms 172 MB	4,311 ms 18 MB

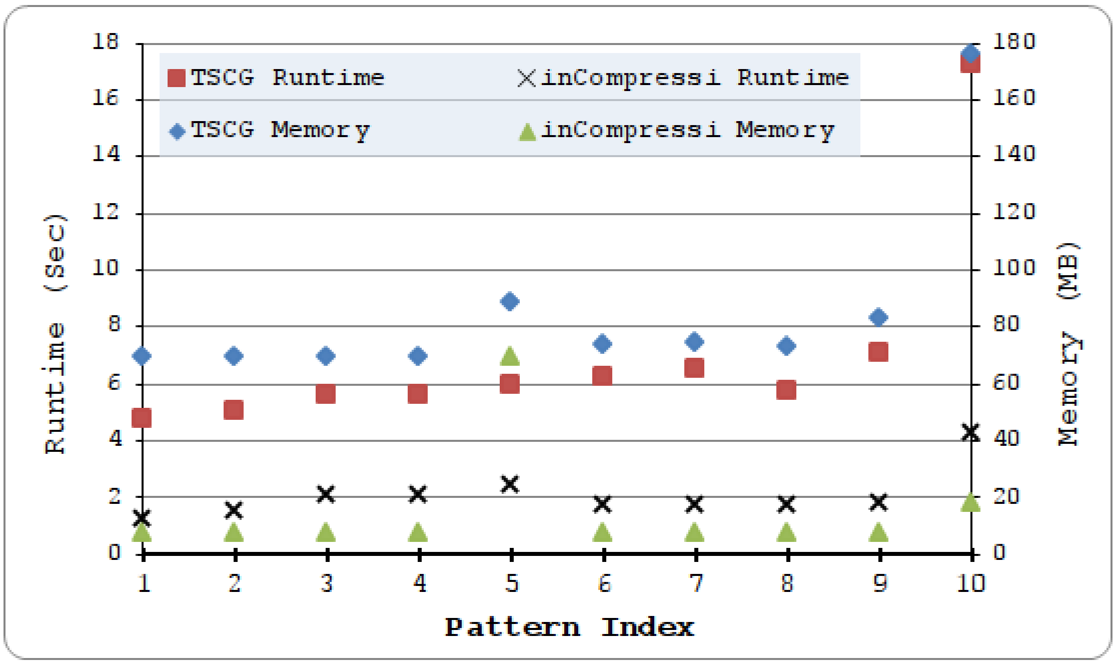

Fig. 2: *The runtime and storage taken by TSCG and inCompressi while searching the compressed TAIR9 genome for the ten patterns listed in (Table 3).*

Table 4. : *The runtime and storage needed to search the yh human genome using (a) TSCG and (b) inCompressi. Every pattern has two rows, the* 1st *row shows the average runtime in ms, while the* 2nd *row shows the memory consumption in MB. Notice that the yh's complete decompression time (400 sec on average) is added to the runtime of TSCG, because both TSCG and inCompressi start working on a compressed yh genome. The runtime of inCompressi represents the overall search time inside the referentially compressed genome.*

Pattern			**(a)** TSCG				**(b)** inCompressi
Index	Value	Hits	Decompression	Caching	Searching	**Total**	
1	**TTTTTCCCCC**	15,831			4 sec 5 MB	454 sec 1,034 MB	404 sec 556 MB
2	**ACGTACGT**	1,855			9 sec 5 MB	459 sec 1,034 MB	396 sec 556 MB
3	**AACCGGTT**	3,575			16 sec 5 MB	466 sec 1,034 MB	411 sec 556 MB
4	**AAACCCGGGTTT**	39			16 sec 5 MB	466 sec 1,034 MB	417 sec 556 MB
5	**AAAA**	44,053,189	400 sec 794 MB	50 sec 235 MB	19 sec 1171 MB	469 sec 2,200 MB	404 sec 1,898 MB
6	**ACGT - - - ACGT**	1,850			22 sec 18 MB	472 sec 1,047 MB	397 sec 556 MB
7	**ACGT - - - TGCA**	14,042			31 sec 77 MB	481 sec 1,106 MB	389 sec 556 MB
8	**TATAT - - - - - CGCGC**	84			18 sec 34 MB	468 sec 1,063 MB	393 sec 556 MB
9	**TTTTT - - - - - AAAAA**	89,701			39 sec 157 MB	489 sec 1,186 MB	403 sec 556 MB
10	**TTT - - - - - - - - - - - - - - - - TTT**	6,596,959			168 sec 856 MB	618 sec 1,855 MB	427 sec 745 MB

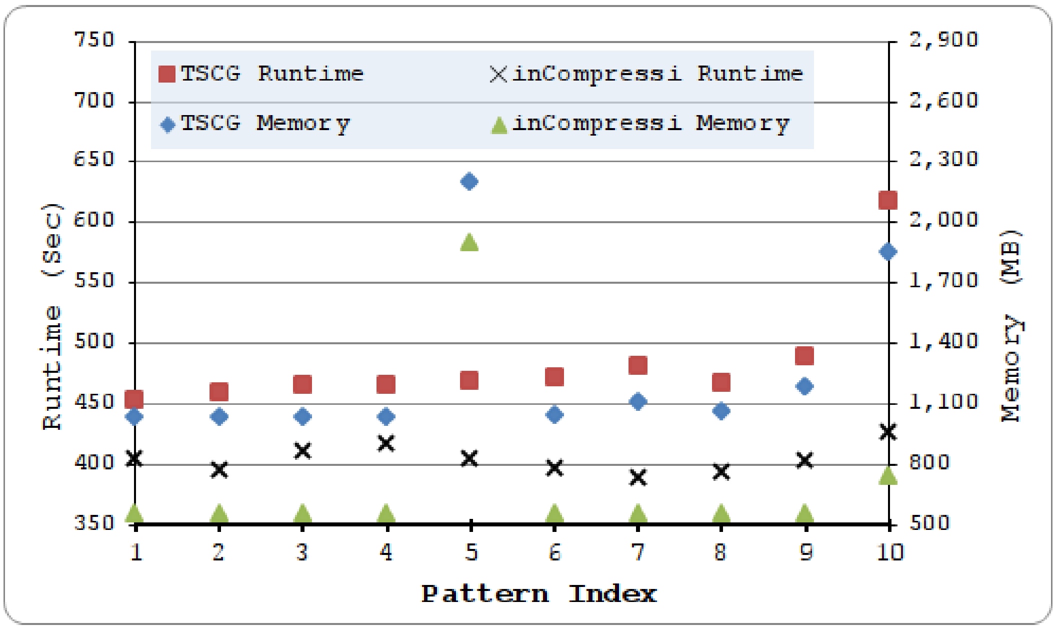

Fig. 3: *The runtime and storage taken by TSCG and inCompressi to search the compressed yh genome for the ten patterns listed in (Table 4).*

inCompressi allows simultaneous searching of the compressed genome for multiple patterns. For instance, by searching the TAIR9 genome simultaneously for all the ten patterns in (Table 3), inCompressi took around 12 sec to find all their occurrences instead of searching for them one by one (with multiple redundant decompressions) in more than 21 sec. Conversely, and assuming the need for only one complete decompression, TSCG took around 30 sec to traditionally decompress and cache the TAIR9 genome once, and then search it for the same patterns one by one.

By simultaneously searching the yh human genome for the 10 patterns in (Table 4), inCompressi took 553 sec instead of searching for the same patterns one by one with total runtime 4,041 sec. Considering one genome decompression (400 sec) and one caching process (50 sec), TSCG took the total runtime of 792 sec. Moreover, inCompressi consumed zero disk storage and less memory compared to TSCG.

4. CONCLUSION AND FUTURE WORK

This article proved how it is worthy to search the referentially compressed genomes without their complete decompression. The existing inCompressi algorithm is enhanced to search the individual blocks it generates by performing partial decompressions over genomes referentially compressed by the Stanford algorithm. Thus, inCompressi eliminates the extra disk storage that would be needed to traditionally store genomes after their complete decompression. Moreover, inCompressi consumes memory that is proportional to the length of individually decompressed blocks rather than an entire decompressed genome. Furthermore, compared to the traditional search procedures, inCompressi took lower runtime to search the referentially decompressed genome, especially by incomplete pattern. In addition, it needs no extra memory or disk storage as would be needed by the traditional search procedures to store a totally decompressed genome before searching it. The inCompressi's search experiments over versions of the TAIR genome and the human genome showed how inCompressi is promising. In addition, inCompressi's runtime could be further enhanced by utilizing the multicore technologies of the nowadays machines. Finally, we believe that implementing both Stanford and inCompressi with compiler-based languages could have further performance gains compared to their Python's implementations.

5. REFERENCES

[1] R.A. Gibbs, J.W. Belmont, P. Hardenbol, T.D. Willis, F. Yu, et al. The international hapmap project. *Nature*, 426(6968):789–796, 2003.

[2] N. Siva. 1000 genomes project. *Nature biotechnology*, 26(3):256–256, 2008.

[3] G.M. Church. The personal genome project. *Molecular Systems Biology*, 1(1), 2005.

[4] S. Wandelt and U. Leser. String searching in referentially compressed genomes. In *KDIR*, pages 95–102, 2012.

[5] S. Wandelt, U. Leser, et al. Adaptive efficient compression of genomes. *Algorithms for Molecular Biology*, 7:30, 2012.

[6] B.G. Chern, I. Ochoa, A. Manolakos, A. No, K. Venkat, and T. Weissman. Reference based genome compression. In *Information Theory Workshop (ITW), 2012 IEEE*, pages 427–431. IEEE, 2012.

[7] A.D. Wyner and J. Ziv. The sliding-window lempel-ziv algorithm is asymptotically optimal. *Proceedings of the IEEE*, 82(6):872–877, 1994.

[8] C. Wang and D. Zhang. A novel compression tool for efficient storage of genome resequencing data. *Nucleic Acids Research*, 39(7):e45–e45, 2011.

[9] A.J. Pinho, D. Pratas, and S.P. Garcia. Green: a tool for efficient compression of genome resequencing data. *Nucleic Acids Research*, 40(4):e27–e27, 2012.

[10] M. Nassef, A. Badr, and I. Farag. An algorithm for browsing the referentially-compressed genomes. *International Journal of Computer Applications*, 86(8):1–10, January 2014. Published by Foundation of Computer Science, New York, USA.

[11] A. Cornish-Bowden. Nomenclature for incompletely specified bases in nucleic acid sequences: recommendations 1984. *Nucleic acids research*, 13(9):3021, 1985.

[12] Python's fast search implementation using a mix between boyer-moore and horspool algorithms. `http://svn.python.org/view/python/trunk/Objects/stringlib/fastsearch.h?revision=68811&view=markup`.

[13] R.S. Boyer and J.S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[14] R.N. Horspool. Practical fast searching in strings. *Software: Practice and Experience*, 10(6):501–506, 1980.

Websites

TAIR8 Website: `ftp://ftp.arabidopsis.org/home/tair/Sequences/whole_chromosomes/OLD/` [Accessed: 5 December 2012]

TAIR9 Website: `ftp://ftp.arabidounsrtpsis.org/home/tair/Sequences/whole_chromosomes/OLD/TAIR9_chromosome_file/` [Accessed: 5 December 2012]

HG18 Website: `http://hgdownload.cse.ucsc.edu/goldenPath/hg18/bigZips/chromFa.zip` [Accessed: 6 Jan 2013]

YH Website: `ftp://public.genomics.org.cn/BGI/yanhuang/fa/` [Accessed: 7 Jan 2013]