AN EFFICIENT METHOD FOR COMPRESSING AND SEARCHING GENOMIC DATABASES

Jeffrey B. Wallace, Gregory L. Vert, Sara Nasser
Department of Computer Science and Engineering
University of Nevada, Reno
Reno, NV 89557
E-mail: jwallace@tmcc.edu, {gvert,sara}@cse.unr.edu

KEYWORDS
Genomic sequence, search, algorithm, compression, bioinformatics, database

ABSTRACT

Biological databases are growing significantly, as are the number of queries directed at them. In 2005, the genomic databases at the National Center for Biotechnology Information (NCBI) received about 50 million web hits per day, at peak rates of about 1,900 hits per second. As these databases become more popular, there is increased demand to make them faster and more efficient. In this paper, we propose a method for compressing and searching selected genome databases using techniques appropriate for computers of virtually any size. This search technique is expected to produce its best results with large search sequences against large DNA databases, and lends itself to parallel computation techniques with little communication overhead required. Because the compression algorithm uses a lossless binary encoding format, search results are exact – not approximate. Furthermore, searches take place on the compressed data, obviating the need for decompression prior to executing a search.

INTRODUCTION

Biological databases are growing significantly as organisms are being sequenced. These genomic databases help biologists understand the underlying structure of organisms and aid research in the area of genomic sciences. Publicly available databases can be used by biologists to compare organisms, find related species, etc.

To retrieve information, users run queries against one of these databases, such as a search for a nucleotide sequence from a nucleotide database. A typical search involves matching a query nucleotide sequence with all the sequences available in the database. If a database is large in size, several comparisons are required until a match is found. Efficiency is critical in such a database system. The user desires to find the exact or most similar result in the shortest amount of time. A database search can be viewed as finding the longest common subsequence(s) between the query sequence and a database sequence. A longest common subsequence indicates the similarity between the query and the database sequence.

Genomic sequences are fairly large in size ranging from several thousand to million character long sequences. Searching against a large database can be time consuming therefore there is a need to make database access faster and better.

A typical search may involve comparing a string of 200K characters against a database that may contain millions of sequences of similar or larger sizes. A user who may be trying to search against a large database may require several minutes to get the reply. Added to that a PC has limited resources and searching against a large database can be quite time consuming. Querying a database involves several factors such as the speed of the tool, accuracy of the match, etc. An ideal tool for such querying should satisfy these requisites.

Since large databases have huge memory requirements, a method to compress data can be beneficial. Compression of data can greatly reduce the processing time. For example, if the data is compressed five-fold, then a 200,000 character sequence now becomes a 40,000 character sequence. The same query search now involves comparison with smaller sequences making the search faster.

DNA data is sensitive to changes, such as replacements, insertions, deletions. A compression technique that permits full recovery of the genome sequences is required. For example, consider two sequences:

Sequence 1:

ACTTACGTATCGCC<u>C</u>CC

Sequence 2:

ACTTACGTATCGCC<u>A</u>CC

Sequence 1 and 2 are similar in that there is only one character difference between them. An ideal compression technique should maintain the similarity relationship between the two sequences after compression. In the above case the compressed data should also have a distance of 1.

The research in this paper focuses on providing fast search, fast retrieval speed from disk, efficient memory utilization, and easily parallelizable implementation for even faster searches and/or distributed deployment. A

Proceedings 21st European Conference on Modelling and Simulation
Ivan Zelinka, Zuzana Oplatková, Alessandra Orsoni ©ECMS 2007
ISBN 978-0-9553018-2-7 / ISBN 978-0-9553018-3-4 (CD)

lossless compression technique is proposed that allows full recovery of data. The next section presents the background, followed by the proposed algorithm in the following section. In the following section we analyze the technique and conclude with future work in the last section.

BACKGROUND

In this section we discuss some of the existing tools for sequence alignment. We also discuss some of the earlier methods propose to improve sequence alignment.

Sequence alignment is an important field in bioinformatics. Sequence alignment provides method to compare new sequences with previously completed or assemble sequences. The completed sequences are stored in databases.

Genomic database servers process tens of thousands of queries a day. GenBank is one such database maintained by National Center for Biotechnology Information (NCBI). GenBank has over 55 million sequence entries from at least 200,000 different organisms (GenBank 2005). GenBank's search tool is known as BLAST. In 2005, NCBI received about 50 million web hits per day, at peak rates of about 1,900 hits per second, and about 400,000 BLAST searches per day from about 2.5 million users (Astell 2005).

BLAST is a de facto standard tool used for measuring similarity between sequences (Altschul 1990, 1997). BLAST is popular for its efficiency, and has undergone several updates for efficiency and speed. Mega BLAST is a greedy search method that works on BLAST data for DNA sequence alignment search and is known to be faster than BLAST. Mega BLAST uses a greedy algorithm for nucleotide sequence alignment search (Zhang 2000). This program is optimized for aligning sequences that differ slightly and can align much longer sequences than BLAST. Mega BLAST can only work with DNA sequences.

The Institute for Genomic Research (TIGR) is another center for deciphering and analyzing genomes. TIGR's Genome Project contains a collection of curated databases containing DNA and protein sequence, gene expression, cellular role, protein family, and taxonomic data for microbes, plants and humans (IGR 2007).

BLAST is a tool that does an exhaustive search. FASTA is another tool that does an exhaustive search. An exhaustive search is costly in terms of speed. A query string does not have to be compared to the entire database. Heuristics can be added to a search process to make it faster and accurate. Keyword based searches are one such example of non-exhaustive search. Keyword-based search has been popularized by Internet search engines and is not generally provided by traditional databases (Agrawal 2002). An example of keyword-based search over structured databases is EKSO (Su 2005). EKSO indexes interconnected textual content in relational databases, providing intuitive and highly efficient keyword search capabilities over this content. It trades storage space and offline indexing time to significantly reduce query time computation.

There are several other search tools such as Flash, SST and CAFÉ (Cao 2005, Baxevanis 2005). CAFÉ is based on an indexed scheme with appropriate data structures and has shown a faster query search than exhaustive searching. A comparison of BLAST, FASTA, and CAFÉ has been studied (Williams 2002). An indexing technique for answering approximate keyword search queries was developed by Fei and Mefford (Shi 2005). This technique has two principal components – a data structure called V-tree and its partition methods for clustering words in the vocabulary into subgroups. It stores the words in the vocabulary into a V-tree based on its partition methods. The V-tree data structure can reduce the number of distance computations needed to answer the query.

Even though there has been much research for making for making query searches faster, there has been less research in terms of reducing storage requirements for both the database and the user. The algorithm proposed in the following section can perform searches on compressed data in a fast and efficient manner.

THE PROPOSED METHOD

This section describes the proposed compression and searching algorithms, including sequence encoding, data structures, and query processing. Examples are provided in each of these areas.

Sequence Encoding and Compression

Genomic sequences are commonly stored as strings comprised of the four DNA bases: C, G, A, and T. Generally, these are stored as ASCII characters, where each symbol requires a single byte of storage.

However, only two bits are required to adequately express these four symbols. Thus, 'C' can be replaced with the bits 00, 'G' with 01, 'A' with 10, and 'T' with 11. This simple binary encoding of base strings allows four DNA bases to be represented within one byte. Thus, the sequence AGGT can be represented in binary form as 10 01 01 11, which is readily converted to its decimal equivalent of 151.

For the purposes of the proposed search algorithm, however, it is desirable to represent octets of DNA bases as a single unit. Since each base requires two bits, all possible octets can be represented as 16-bit integers ranging from 0 (all Cs) to 65535 (all Ts) as shown in Figure 1. Since each integer in this range uniquely

encodes a string of eight nucleotides (an octet), these integers will serve as hash values in the data structure at runtime. In the meantime, these compressed values are either stored sequentially on disk or transmitted sequentially over a network during a file read operation.

1 2 3 4
CCGGAATT GACTTCAG TCAGACTT CTCGAAGG

GATTACAA GACTTCAG GACTTCAG
5 6 7

Octet (seq #)	Decimal
CCGGAATT (1)	1455
GACTTCAG (2,6,7)	25545
TCAGACTT (3)	51599
CTCGAAGG (4)	12709
GATTACAA (5)	28554

Figure 1: Octet Encoding

Data Structure

Since DNA search and target strings are typically very large, simple M x N string comparisons take too much time to be considered a viable solution – a more sophisticated data structure is required in order to achieve reasonable performance in large searches. The applicability of a data structure is largely determined by the context of the problem – there is no such thing as the "perfect" data structure. In the context of genomic search, tree structures suffer because the data has little order other than sequence, so complete searches require either complete tree traversal or extra links within the tree structure. Arrays suffer from the same problem. Various array indexing schemes offer speed improvements, but at the expense of memory. The technique used for video compression suffers because the distances to reference words are typically longer than the two bits necessary to store the base being encoded (1).

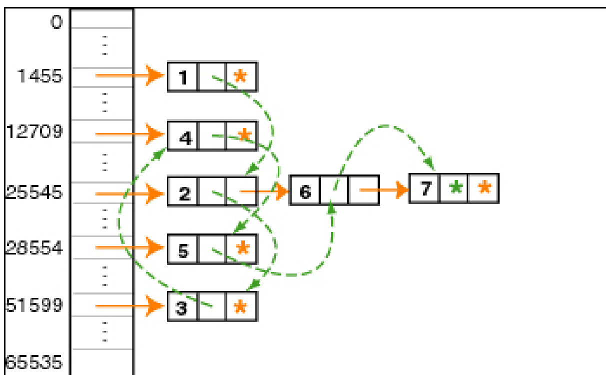

Figure 2: Data Structure

The proposed data structure consists of a doubly-chained hash table, where one chain of pointers (solid lines in Figure 2) implements a linked-list of all octets that hash to the same integer value. A second linked list (shown with dashed lines) is used to traverse the octets in sequential order. In addition to preserving the integrity of the data, this structure ensures that only the portions of the targets sequence that match the search string will be searched.

This data structure is easily serialized. Each octet is encoded to binary (interpreted as decimal) form as described above and stored sequentially on disk – which offers the best I/O performance at runtime.

As each octet (16-bit integer) is read, a new node is created and inserted at the end of the linked list for the octet's hash value (solid lines), which ensures that each hash value's linked list is stored in sorted order. The node also contains sequence position information in order to report where a match begins, as well as a second linked list (dashed lines) that allows the data structure to be efficiently traversed sequentially. Both of these linked lists are traversed during a search operation.

```
startNode = hashTable( decimal value of first
octet of search string )

FOR EACH of the possible ((# bases mod 8) + 1)
search windows, DO

  searchNode = first complete octet of the
  windowed search string

  WHILE startNode != null, DO
    currNode = startNode

    WHILE currNode.hashValue ==
    searchNode.hashValue

      currNode = currNode.next
      searchNode = searchNode.next

      //end of target without match?
      IF currNode == null AND searchNode != null
         BREAK loop

      ELSE IF searchNode == null OR
      searchNode is a partial octet
        IF searchNode.partialBits !=
        currNode.partialBits
           BREAK loop

        ELSE report MATCH at
        startNode.sequencePos*8 + windowNum

    startNode = startNode.next
```

Figure 3: Algorithm

Algorithm

Sequence searching takes place on compressed sequence octets stored in the data structure described above. In order to make comparisons to this data structure, the search string also needs to be encoded as binary octets using the same compression technique. Because of the iterative windowing required by this algorithm, search strings need to contain at least fifteen bases.

Descriptions and examples of each of the algorithm functions are covered below. The algorithm shown in Figure 3 assumes that the target sequence has been encoded and stored in the data structure described above. It also assumes that each base search sequence has been encoded as a two-bit binary value, and that the search sequence contains at least 15 bases. In the following algorithm, the variable currNode follows the sequence-ordered linked list (the dashed list in Figure 2). The variable startNode follows the hash-ordered linked list (the solid-line list).

Fully Aligned Octets – A Simple Example

Figure 4 builds on the previous figures, and shows a simple, near-best case scenario. In this case, the search sequence matches the 6th and 7th octet of the target sequence.

The search begins by accessing the linked-list addressed by the hash value of the search sequence's first octet. In this case, the first element of the hash value's linked list is octet #2 in the sample sequence. Indeed, the first octet of the search sequence matches octet #2 of the sample sequence, so the (dashed-line) linked list is followed to find the octet #3 in the sample sequence. In this case, the hash value of octet #3 of the sample sequence (51599) does not match the hash value of the second octet of the search sequence (25545), so this once-promising search is abandoned, and the (solid-line) linked list is followed to the next octet that matches the first octet of the search sequence – in this case, octet #6.

The next possible match begins at octet #6 in the sample sequence. Following the (dashed) linked list to octet #7 reveals that it matches the second octet in the search sequence. Since there are no more terms in the search sequence, there is a match beginning at octet #6 in the sample sequence. If there were more octets in the (solid-line) linked list, further matches could be discovered by repeating the process until the end of the (solid-line) list is reached.

Partial Search Strings

In the previous example, the lengths of the search strings were multiples of eight, so they aligned nicely with octet boundaries. More often than not, this perfect alignment is not the case in actual searches, so the algorithm requires a further refinement in order deal with non-aligned search strings. Instead of the last octet of the search sequence being compared to an entire octet in the target sequence, partial octets are compared using an AND operation as shown in Figure 5. As will be seen in the next section, this partial alignment technique may also be used on the first octet of the search string.

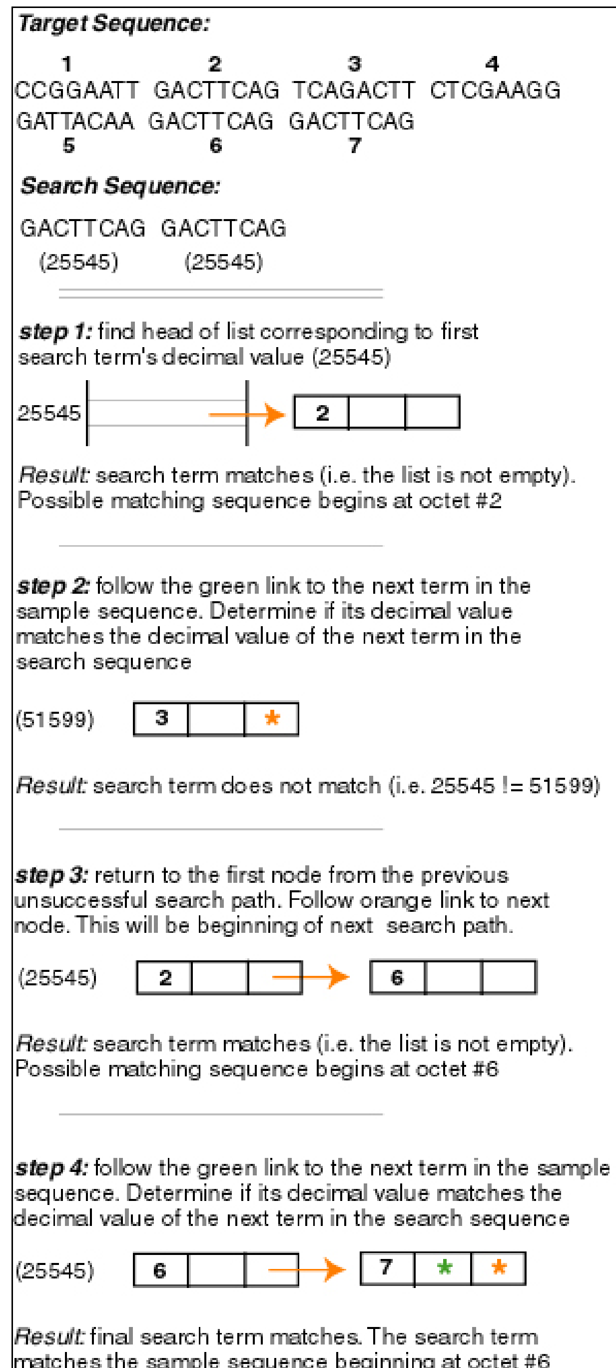

Figure 4: Searching the Data Structure

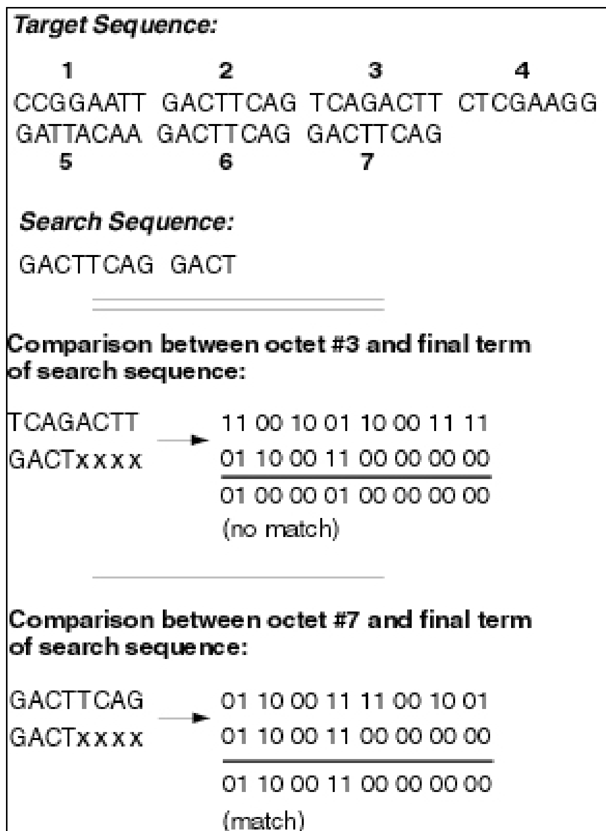

Figure 5: Partial Octet Matching

Search Sequence Windowing

In both of the examples described above, the matching sequences occurred on octet boundaries. Again, this is not usually the case in actual searches. In most cases (or when searching for all matches in a target string), the search algorithm uses a windowing technique where the search sequence is shifted one base (2 bits) to the right for each of the eight possible positions in the octet. Note that only the search sequence is shifted – the data structure for the target sequence remains unchanged. Although this requires eight separate searches of the target sequence, these basic comparison operations are sufficiently fast in modern processors.

An example of the windowing algorithm is shown in Figure 6. At each iteration, the octet contained in the box would serve as the first octet to be matched. As before, the hash value is simply a decimal interpretation of the binary base encoding.

For each window iteration, the algorithm begins by matching all complete octets as illustrated previously in Figure 4. As usual, if any mismatches occur along the way, the search moves to the next target sequence matching the hash value of the first octet of the windowed search string. If all complete octets match, the partial octet at the end of the search sequence (if it exists) is matched against the target sequence as illustrated previously in Figure 5. If the sequences continue to match, the partial octet at the beginning of the search sequence (if it exists) is finally matched against the target sequence. Note that this data structure does not include backward chaining – which would require unacceptable memory overhead because it would require extra pointers for every octet, even though only one backward pointer would be used for a given search string. Instead, the partial sequence preceding the first complete octet is retrieved from disk, which is a relatively fast direct access file I/O operation since the sequence position is known.

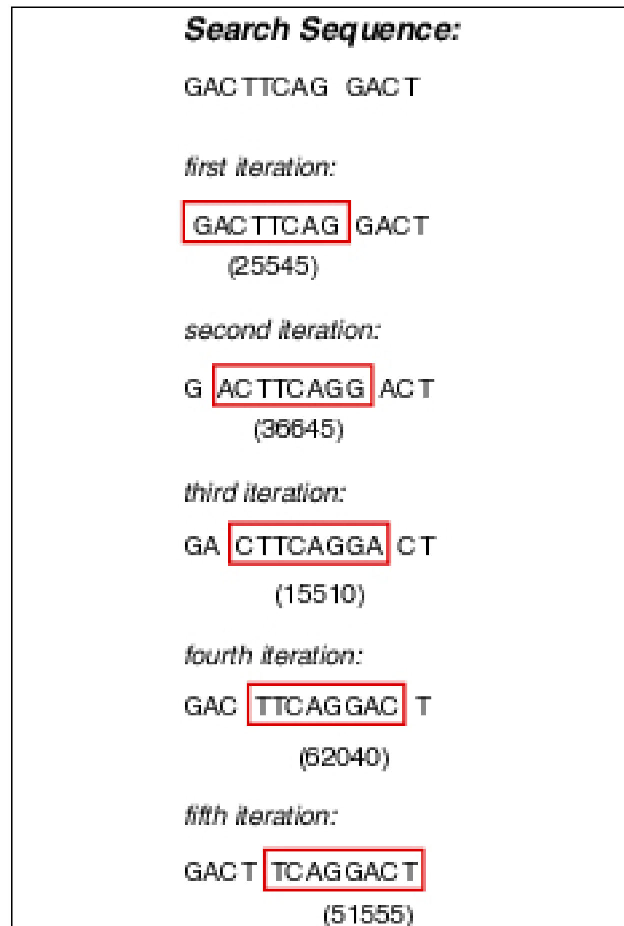

Figure 6: Search Windowing

ANALYSIS

A software prototype that implements this method has been developed. Preliminary testing on a dataset containing 5.6 million base pairs appears promising:

a) The simple compression algorithm achieves a constant 4:1 compression ratio on standard ascii-coded FASTA databases.

b) Base octets appear to be reasonably well distributed, resulting in similar-sized linked lists for each possible hash value. Thus, in a target sequence with 3 billion base pairs, the average linked list for

a given hash value (the solid-line list) would be expected to contain approximately 5700 nodes (3B $\div 2^{16} \div 8$).

c) Searches for strings containing up to 5000 base pairs are executing in under 100 ms on an (old) 2GHz AMD 64 with 1 MB of RAM.

CONCLUSIONS AND FUTURE WORK

The expected benefits of this approach are four fold:

a) Fast search. Only the portion(s) of the target sequence that match the search string will be searched. Non-matching areas are ignored.

b) Fast I/O. Compressed data are saved sequentially, resulting in fast retrieval speed from disk.

c) Efficient memory utilization. Compressed data does not need to be decompressed. The algorithm works directly on compressed data.

d) Easily parallelizable for even faster searches and/or distributed deployment. The algorithm is embarrassingly parallel and involves little communication overhead.

As described above, this data structure relies heavily on memory address pointers. Since these pointers cannot be meaningfully stored on disk (when the data is reloaded, it will most likely load into different memory addresses), search operations will be most efficient on machines with enough memory to store the entire compressed target sequence.

The memory required to store a target sequence is expected to be directly proportional to the number of bases in the sequence and the word size of the computer used. With a 64-bit processor, each octet is expected to require 20 bytes (32 bits for a sequence number and two 64-bit addresses). With a 32-bit processor, each octet is expected to require 12 bytes. Thus, the data structure required to hold a target sequence with 3 billion base pairs on a 64-bit processor is expected to require approximately 7.5 GB of memory. Although this is a significant amount of memory, it is within the realm of a workstation-sized computer.

Preliminary testing of this approach appears promising. Over the next couple of months, the following extensions to this project are anticipated:

a) Finish programming the prototype search tool.

b) Measure search speed of various-sized search strings on a broad range of actual genomic datasets, and compare these results against those using tools such as BLAST, FASTA, and CAFÉ.

c) Compare memory and storage efficiency against existing tools such as BLAST, FASTA, and CAFÉ.

d) Parallelize the algorithm so it can take advantage of multiple processors and large shared memory clusters. This is likely to lead to significant performance gains.

e) Explore methods for implementing wildcard (similarity) matching in order to be able to search strings "similar" to a search string.

REFERENCES

Agrawal, Sanjay, Surajit Chaudhuri, Gautam Das, "DBXplorer: A System for Keyword-Based Search over Relational Databases," *18th International Conference on Data Engineering* (ICDE'02), 2002.

Altschul S., W. Gish, W. Miller, E. Myers, and D. Lipman, "Basic Local Alignment Search Tool," J. *Molecular Biology*, vol. 215, pp. 403-410, 1990

Altschul, SF, TL Madden, AA Schaffer, J Zhang, Z Zhang, W Miller, DJ Lipman, (1997) "Gapped BLAST and PSI-BLAST: a new generation of protein database search programs", *Nucleic Acids Res*, . 25, 3389–3402

Astell, James (2005), *"Databases of Discovery",* ACM Queue vol. 3, no. 3 - April 2005

Baxevanis, Andreas D. (Editor), B. F. Francis Ouellette, *"Bioinformatics: A Practical Guide to the Analysis of Genes and Proteins"*, New Jersey: Wiley 2005

Cao, Xia, Beng Chin Ooi, Tung, A.K.H., Hwee Hwa Pang, Kian-Lee Tan, "DSIM: A Distance-Based Indexing Method for Genomic Sequences", *IEEE Symposium on Bioinformatics and Bioengineering*, 2005. BIBE 2005, Publication Date: 19-21 Oct. 2005, (pp): 97- 104

GenBank (2005), http://www.nlm.nih.gov/news/press_releases/dna_rna_100_gig.html, date accessed Feb 2007.

The Institute for Genomic Research, http://www.tigr.org/db.shtml, data accessed March 2007.

Shi, Fei, Mefford, C., "A new indexing method for approximate search in text databases"**,** *The Fifth International Conference on Computer and Information Technology*, 2005. CIT 2005, Sept. 2005 pp: 70- 76, 2005

Su, Qi, Jennifer Widom, **"**Indexing Relational Database Content Offline for Efficient Keyword-Based Search**"** *Proceedings of the 9th International Database Engineering & Application Symposium* (IDEAS'05), Vol. 00, pp: 297 – 306, 2005

Williams, H. and J. Zobel. Indexing and Retrieval for Genomic Databases. *IEEE Transactions on Knowledge and Data Engineering*, 2002.

Zhang, Zheng, Scott Schwartz, Lukas Wagner, and Webb Miller (2000), "A greedy algorithm for aligning DNA sequences", *J Compuational Biol* 2000; 7(1-2):203-14.

AUTHOR BIOGRAPHIES

JEFF WALLACE was born on a farm in the upper peninsula of Michigan. He received his BS in Computer Science from the University of Michigan in 1982, his MBA from Santa Clara University in 1988 and his MFA in film with a specialization in Special Effects from the University of Southern California in 1996. He is currently a graduate student at the University of Nevada and is a tenured faculty member at Truckee Meadows Community College in the Department of Computer Office Technologies. His

research interests are in Bioinformatics and Artificial Intelligence. His email is jwallace@tmcc.edu

GREGORY VERT was born in Fairfield California in 1956. He received his BS in Geography with a specialization in GIS from the University of Washington in 1985, his MS in Information Systems Management, Seattle Pacific University, in Seattle, Washington in 1988, and his PhD in Computer Science from the University of Idaho in Moscow Idaho in 2000. He has been an Assistant Professor at the University of Nevada, Reno in the Computer Science and Engineering Department since 2002. His research is in the areas of GIS, Computer Security, Fuzzy System, Database and Bioinformatics. His email address is gvert@cse.unr.edu

SARA NASSER was born in Hyderabad, India. She received her BE in Computer Science and Engineering from MJCET, Osmania University, and her MS in Computer Science from the University of Nevada in 2003. She is currently a PhD student in Computer Science and Engineering and is expecting to graduate in 2008. Her research is in the area of Bioinformatics and Fuzzy Systems. Her e-mail address is sara@cse.unr.edu