A Systolic Array Technique for Determining Common Approximate Substrings

to be presented as ISCAS2006

Kenneth B. Kent
Faculty of Computer Science
University of New Brunswick
Fredericton, New Brunswick, Canada
Email: ken@unb.ca

Jacqueline E. Rice
Dept. of Math & Computer Science
University of Lethbridge
Alberta, Canada
Email: j.rice@uleth.ca

***Abstract*— A new technique that makes use of a systolic array structure is proposed for solving the common approximate substring (CAS) problem. This approach extends the technique introduced in [1] from the computation of the edit-distance between two strings to the more encompassing CAS problem. The technique presented is validated and analyzed through simulation.**

I. Introduction

A known problem in bioinformatics is that of determining, within two or more strings of DNA, a *common approximate substring* (CAS) [2], [3]. Generally a search for a CAS is successful if a similar pattern of symbols is found within all of a given series of sequences, allowing a certain amount of error. Previous work [4] has investigated the use of field programmable gate array (FPGA) technology to combine the flexibility of software and the acceleration of hardware in finding a solution for the CAS problem.

A. Common Approximate Substring Matching

The problem we wish to solve is that of determining which, if any, substrings within a given set of strings are common to all of the strings in the set. The problem is made considerably more complex by allowing the incorporation of an error factor in the matching process. This is a technique used in DNA sequencing, where the discovery of sequence homology to a known protein or family of proteins may provide information about the function of a newly sequenced gene [5]. The discovery of homologous sequences and families begins with the search for common motifs [5], [6].

In searching for common motifs the goal is to find similar sequences of symbols, where the length of the sequence or motif is a predefined value (m). The search space is a given set of n DNA strings, or sequences, also of a defined length (l). Finally, the allowable number of errors in the match between a given motif and a substring within a DNA string is set at d. In this work we limit the definition of an "error" to that of a simple replacement of one symbol with another; shifts and/or gaps in the sequence are not permitted. Figure 1 illustrates how a motif of length 5 can be found within 4 DNA strings, allowing an error of 1.

TGACTCGACC
TACTGCCTCG
CTGGCTAATA
ATTCCTGACT

Fig. 1. An example of common approximate substrings of length 5 with an error of 1. Solution motifs for this example are: TGACT, TGCCT, TGGCT, and TGACT.

B. Previous Work

Previous work in this area includes [7], [8], [9], [5], [10] and [11]. Various approaches have been used, including dynamic programming [11], approximation of optimal alignments [5], the use of three-dimensional matrices [9] and the use of reconfigurable hardware [7], [8]. The work presented in this paper was motivated by [7] along with the desire to build a more efficient data structure specific to this problem.

II. Approach

The first step in processing the strings from the database is to preprocess the first search string. It is necessary to partition the string into $l - m + 1$ motifs, and then for each motif generate all possible motifs that are distance d errors from the generating motif. These are then stored as a forest of trees, where the forest represents all possible motifs for a given generator. Sharing of trees is possible in a restricted way. Entire paths from top node to leaf node can be shared, and identical upper portions of paths can be shared; however, unique paths must result in unique leaf nodes. An example is shown in Figure 2.

A. Nodes

Each node in a tree has processing and storage capabilities. The storage consists of the following:

- the character value of the node,

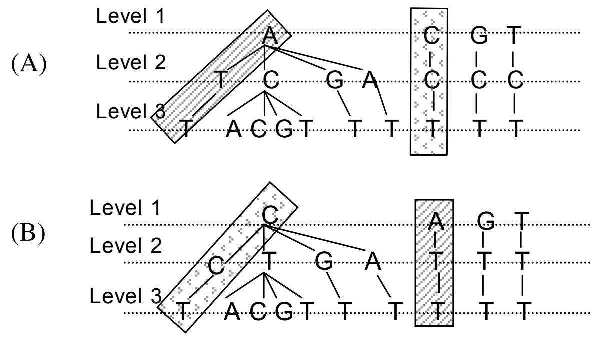

Fig. 2. An example illustrating how motifs ACT (A) and CTT (B) generate two distinct forests when $d = 1$. The shaded areas indicate where full trees, or complete branches of trees, can be shared.

- a bit vector whose length is equal to the level on which the node resides, and
- a current piece of data. This may be a character or a numeric value as alternated in the input stream. The numeric value will never exceed 2 times the number of levels in the trees.

The character value of the node is required for performing comparisons with the characters that are streamed into the nodes. The bit vector is required to record the number of errors encountered at that node when performing comparisons; however, the memory of the bit vector (*i.e.* the number of errors to be remembered) is limited by the level in the tree at which the node resides. Root, or top nodes are at level 1 while the leaf or final nodes are at level m. The final piece of data is either a character for comparison to the node's own character or a numeric value that is used for summing the errors that have been encountered on the motifs' travel down through the levels of the tree.

Each node must be able to process either characters passed into it or numbers. If a character is passed to a node then the node must first perform a comparison of the given character to the node's own character. If their values match then a value of 0 is shifted into the right end of the node's bit vector. If their values do not match then a value of 1 is shifted into the bit vector. An example of this is shown in Figure 3 (A). If a number is passed to a node then the node adds to it the value of the leftmost bit in the node's bit vector. An example of this is shown in Figure 3 (B).

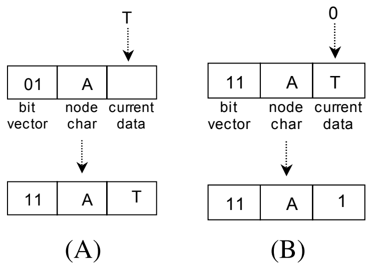

Fig. 3. (A) An example of how a level 2 node processes a character when the character does not match, and (B) an example of how the node processes a number.

Most of the nodes in the system follow the requirements as given above. However below the leaf nodes, at level $m + 1$, there must also be a special type of node called an *exit node*. There is a one-to-one correspondence between each leaf node (at level m) and each exit node. Exit nodes collect and compare sums passed out of the above leaf nodes to d, and record which strings have found the path represented by that leaf node to be a potential CAS solution. If a sum value s is less than or equal to d then we record in the exit node the number of the string currently being processed in a bit vector. The reason for this is that any leaf nodes that result in any sums of d or less are satisfiable CAS solutions for the given string. Leaf nodes that result in potential CAS solutions for every string are *verified* CAS solutions for all strings. Once all strings have been processed through the systolic system we can determine which leaves are terminators for verified CAS solutions by checking which exit nodes have recorded a '1' bit for all input strings. Figure 4 illustrates a path with its exit node and the data currently stored in it.

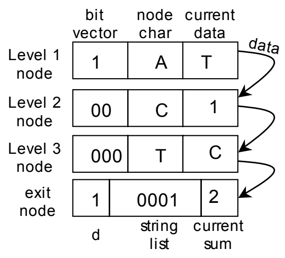

Fig. 4. An example of a path through a tree with its exit node recording the value of d, the current sum, and the 4 input strings for which this path has resulted in a potential CAS solution (currently just string 1).

For example, if we begin with the length 3 motif ACT and allow 1 error then the clump of trees generated consists of 3 levels, with 10 leaf nodes as shown in Figure 2 (A). Figure 5 shows nodes on one path leading to a leaf node; this path represents the generating motif ACT. If we now begin processing an example string TCT then Figures 6 to 10 illustrate how the characters, alternating with digits, are propagated through the system. This example assumes there is only a total of 4 input strings.

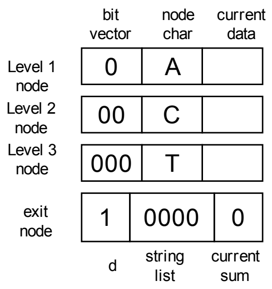

Fig. 5. The data initially stored in the tree nodes of a path representing motif ACT.

The algorithm is as follows, assuming n strings of length l, and that we are searching for CAS motifs of length m with d

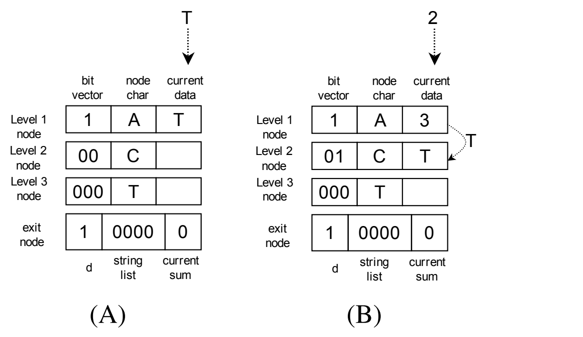

Fig. 6. (A) The first character to compare with, T, is passed into the top node. The characters do not match so a 1 is shifted into the bit vector. (B) A 2 is next passed in; T moves to the next node and the value in the top bit vector is added to the 2 passed in. A 2 is passed in because we do not yet have a valid substring; until the length of the substring is $>= m$ the sum values passed in begin at value $d+1$.

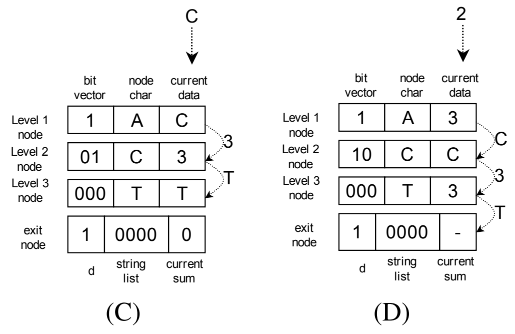

Fig. 7. (C) The next character to compare with, C, is passed into the top node. The currently stored 3 is passed to level 2, and the T from level 2 goes to level 3; this is a match so a 0 goes into the level 3 bit vector. (D) A 2 is next passed in, shuffling each of the pieces of data down to the next levels.

permitted errors.

1) Preprocessing Step:

```
with first DNA string
for $i = 0$ to $l - m$
  for motif consisting of characters $i$ to $i + m - 1$
    generate all possible motifs at distance $d$
    build tree consisting of those motifs
reduce nodes by merging trees with identical roots
```

2) Processing Step:

```
for $j = 2$ to $n$
  for $k = 0$ to $l$
    (1) input character $k$ from string $j$
          to the top node(s) of the tree array
        pass currently stored data to
         next node(s) down
    (2) if tickcount $\leq m$
            set $x = d + 1$
        else set $x = 0$
        input $x$ to the top node(s) of the tree array
        pass currently stored character to
         next node(s) down
  for $k = l$ to $l + m$
    input - to top nodes
      in order to propagate final
       sums to exit nodes
determine which exit nodes are verified
  CAS solutions
```

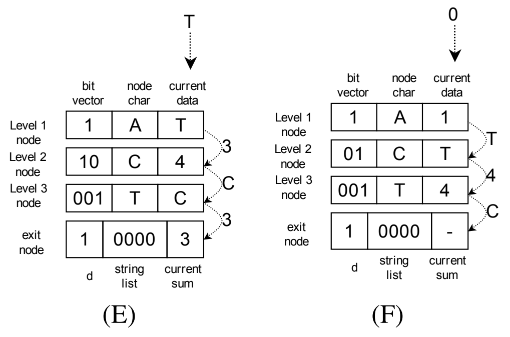

Fig. 8. (E) The final character T is passed in, causing the 3, C, and 3 values to move to the next levels down. The middle 3 becomes a 4 as we add the leftmost bit ofof the bitvector at level 2. (F) A 0 is passed in to the top node.

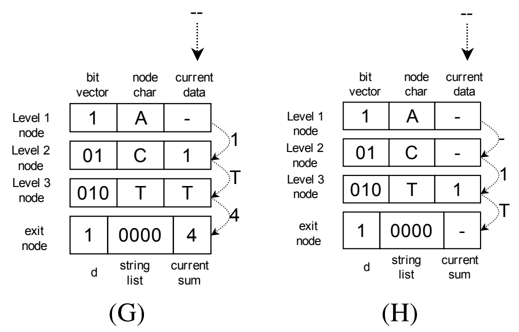

Fig. 9. In both (G) and (H) a - is passed in, in order to propagate the sum value for the entire motif down to the exit node.

3) Node Functionality - Regular Nodes:

```
if character data passed in
    if character matches node value
     shift 0 into right end of
      that node's bit vector
    otherwise
     shift 1 into right end of
      that node's bit vector
if numeric data passed in
    take the number passed in and add to it the
     leftmost bit of that node's bit vector
```

4) Node Functionality - Exit Nodes:

```
if numeric data passed in
    if value is $\leq d$
```

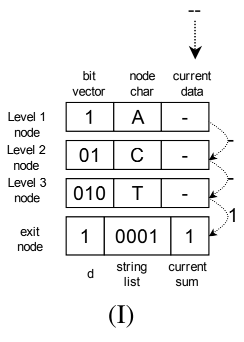

Fig. 10. (I) The final data is passed in, and the sum for the motif TCT reaches the exit node. The sum is equal to d indicating that TCT is a potential CAS solution.

```
    set bit $j$ representing string $j$ in the
      node's bit vector to '1'
  if bit vector is all '1's then output '1'
```

III. Discussion

There can be at most m levels in all trees, and at most

$$\sum_{i=0}^{d} 3^i \cdot \binom{m}{i}.$$

leaf nodes in each forest. This is the exact number of motifs generated for each group of m characters. For example, for $m = 10$ and $d = 2$ this results in 436 possible motifs.

Regardless of the number of leaves, the processing of each subsequent string is performed in constant time, requiring $2l + m$ steps for each string. This allows each character to move through the m levels of the forest as well as the intermediate numeric values for summing the errors encountered.

The big issue for any implementation of this technique is the memory limitations. Each regular node requires

- a maximum of m bits to store a bit vector,
- 2 bits for storing its character data, and
- memory for storing the data to be passed into the node.

If we allow 8 bits for the passed in data then the maximum sum value is 255, which should be far greater than will ever be requried (such a large value would mean that the motif lengths are 255, which would likely far exceed the memory capacity of any type of implementation). Exit nodes require

- a bit vector of length l to store the string list,
- memory for storing d and
- memory to store the data passed in as above.

It is likely that 4 bits would be sufficient to store d, as that would allow a maximum value of 15.

This is clearly a design intended for hardware implementation, as each node will in effect behave as a separate processor as the data is passed down through the structure on each clock pulse. The most important aspects to such a design are the node structure, as a great number of nodes are required, and the implementation of sharing of nodes. Shared paths must be identified in the preprocessing step, which would take place in software. Once the hardware design is in place, however, it would be possible to compare to any number and any size of DNA strings desired. An area requiring some thought is that of careful selection of the first DNA string, the string used to generate the systolic array structure.

Initial implementation work targeting a Spartan 2E 200 FPGA found that a processor node implementation requires 8 CLBs and operates at 166.639 MHz. An exit node implementation requires 130 CLBs with a clock rate of 57.991 MHz. A implementation of the forest illustrated in Figure 2 (A) (21 processing nodes and 10 exit nodes) resulted in 1452 CLBs operating at 57.991 MHz. However, since the exit nodes are only required to process every second input, the overall clock speed can be increased through simple clock management. A revised implementation of Figure 2 (A) using a clock divider resulted in 1472 CLBs with a clock speed of 93.032 MHz.

IV. Conclusion and Future Work

This paper presents a design for a systolic type of structure intended for use in determining common approximate substrings amongst many DNA strings in a search set. This design is intended to use software for the preprocessing step, which will then generate a hardware description for implementation in a reconfigurable device. The implementation phase of this work is not yet completed, although preliminary results for each type of node are reported in Section III. The authors have also been able to simulate an implementation for the forest of nodes shown in Figure 2 (A), illustrating the feasibility of the design. Work in this area is continuing, and will ultimately result in a complete implementation which will be compared to previous work such as [7]. Results obtained thus far by this solution is positive.

There are some important differences between this and previous work. Yamaguchi [8] et al. suggest the use of a reconfigurable device in their solution; however they require two reconfiguration phases in the solution for the edit-distance calculation. Other work such as [7] and [9] require either processing in multi directions or a back-tracking phase. In our solution data progresses in one direction through the structure, thus giving us a fixed and constant time for determining CAS solutions.

This work was motivated by work presented in [7] and continuing work in [1]. Finding an optimal structure in which to represent a problem can often be the key to finding a good solution, and the combination of concepts from each of these works has resulted in such a solution for the CAS problem.

References

[1] K. B. Kent, R. B. Proudfoot, and Y. Zhao, "Parameter-Specific FPGA Implementation of Edit-Distance Calculation," 2006, submitted to the International Symposium on Circuits and Systems (ISCAS).

[2] A. D. Smith, "Common Approximate Substrings," Ph.D. dissertation, Faculty of Computer Science, University of New Brunswick, October 2003.

[3] P. A. Evans, A. D. Smith, and H. T. Wareham, "On the Complexity of Finding Common Approximate Substrings," *Theoretical Computer Science*, pp. 407–430, 2003.

[4] K. B. Kent, J. E. Rice, S. V. Schaick, and P. A. Evans, "Hardware-Based Implementation of the Common Approximate Substring Algorithm," in *Proceedings of the Euromicro Symposium on Digital System Design: Architectures, Methods and Tools (DSD)*, 2005, pp. 314–320.

[5] S. F. Altschul, W. Gish, W. Miller, E. W. Meyers, and D. J. Lipman, "Basic Local Alignment Search Tool," *Jornal of Molecular Biology*, pp. 403–410, Oct. 1990.

[6] P. Pevzner. and S.-H. Sze, "Combinatorial Approaches to Finding Subtle Signals in DNA Sequences," in *Proceedings of the Eighth International Conference on Intelligent Systems for Molecular Biology (ISMB)*, 2000, pp. 269–278.

[7] K. B. Kent, J. E. Rice, S. V. Schaick, and P. A. Evans, "Hardware-Based Implementation of the Common Approximate Substring Algorithm," in *Proceedings of the Euromicro Symposium on Digital System Design: Architectures, Methods and Tools (DSD)*, 2005, pp. 314–320.

[8] Y. Yamguchi, Y. Miyajima, T. Maruyama, and A. Konagaya, "High Speed Homology Search Using Run-Time Reconfiguration," in *Proceedings of Field-Programmable Logic and Applications (FPL) 2002*. Springer Lecture Notes in Computer Science (LNCS), 2002, pp. 281–291.

[9] H. Lee and F. Ercal, "RMESH Algorithms For Parallel String Matching," in *Proceedings of the 3rd International Symposium on Parallel Architec tures, Algorithms and Networks (I-SPAN'97)*, 1997, pp. 223–226.

[10] W. R. Pearson, “Flexible Sequence Similarity Searching With the FASTA3 Program Package,” *Methods in Molecular Biology*, pp. 269–278, 2000.

[11] G. Myers, “A Fast Bit-Vector Algorithm for Approximate String Matching Based on Dynamic Programming,” in *Proceedings of the 9th Combinatorial Pattern Matching Conference, Spring-Verlag LNCS Series #1448*, 1998, pp. 1–13.