



Fast subsequence search using Incomplete Search Trees

Viacheslav Fofanov, Yuriy Fofanov, B. Montgomery Pettitt,

Department of Computer Science, University of Houston, Houston, Texas 77204

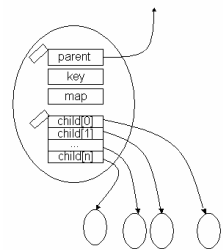


Abstract

New developments in applied genomics, such as, microarrays, PCR, antisense, etc., require searching for relatively short (12 – 60 bases) subsequences unique to particular genes or even whole genomes depending on research goals. The search for this type of unique subsequences employs very large number of runs of a string matching process. Best string matching algorithms, such as the Knuth-Morris-Pratt algorithm and the Boyer-Moore algorithm, are of order $O(m)$, where m is the size of the searched sequence, which makes the whole procedure of searching for unique subsequences hopelessly time-consuming. The ideas of precomputing and use of special data-structures, such as hash tables, used in BLAST and FASTA applications, are employed in order to speed up this process. Fastest possible search for appearance of subsequence of size n in a sequence of size m ($n \ll m$) can be provided by using a search tree corresponding to sequence m . However this approach is extremely memory intensive.

We propose a special type of data structure containing simultaneously the original sequence and the Incomplete Search Tree, nodes of which correspond only to non-unique subsequences contained in the original sequence. The string matching speed of this data structure corresponds to that of the search tree, but the memory cost is drastically reduced. We successfully expanded this data structure to contain multiple sequences simultaneously and introduced several simple operations analogous to set operations, such as union, intersection, etc. We were able to implement this data structure and create Incomplete Search Trees for several microbial genomes.

Data structure description



```
static const int childrenNo = 4;
class Node {
    int key;
    int map;
    Node* parent;
    Node* child [childrenNo];
};
```

int key - number of non-unique subsequences that terminate with this node;

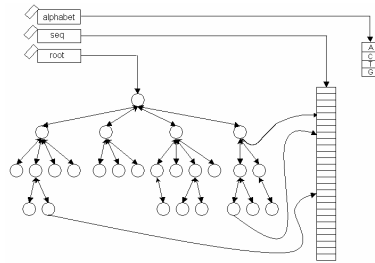
*Node * child* - array of up to 32 pointers to child nodes, the index of the array corresponds to a particular element within the legal for the tree alphabet (e.g. A is *child[0]*, C is *child[1]*), and etc.

Important: When a sequence terminating in a node is unique, the data type of the appropriate element in the child array is changed to integer and the starting location of the unique subsequence within the original sequence is stored in this location!!!

The general idea is to save on memory space by using the array of pointers to children to contain both, pointers to child nodes OR addresses of the unique subsequence. These references to the original sequence are the main feature that reduces the memory requirements of the "complete" search tree to manageable size.

int map - a 32-bit array that specifies whether specific location of the child array points to another node or to a location within the original sequence. For example, the value of map ...000000101 (5), signifies that child 0 and 2 point to nodes in the tree, if the value of *child[3]* is not NULL, and map shows that it does not point to a node, then it must contain the location of a unique

Description of Incomplete Tree class



```
class NodeTree {
public:
    Node * root;
    char * seq;
    int seqSize;
    int depth;
    int n_nodes;
private:
    char * alphabet;
    int aSize;
};
```

*char * alphabet* - pointer to heap-dynamically allocated array that contains the alphabet (legal characters/members) used in creation/upkeep of the incomplete tree data structure.

*char * seq* - pointer to heap-dynamically allocated array containing the original sequence in the internal format (0,1,2... instead of actual characters), transformed according to the alphabet.

*Node * root* - pointer to heap-dynamically allocated "root" node. This is a virtual node, in a sense that it does not have any "meaning" beyond that of starting node of the tree and holding it together.

A special feature of this data structure is that we use the original sequence as a part of our structure and a special type of tree for fast navigation of the original sequence. To save on memory, we keep in the tree only the subsequences that are not unique in the original sequence (i.e. repeat 2+ times). Improvement over the traditional "complete" search tree is shown in the statistics section and is quite extreme.

This approach allows us not to spend memory on unique subsequences (which is the general flaw of traditional tree structures) and thus is the main feature that enables us to store the tree in memory and why it is called an "incomplete" tree.

The tree growing algorithm

- Setup the alphabet
- Setup the supporting boolean array. This is used in optimization, since only one unique tree sequence can start from any particular position in the original sequence
- Grow another level of leaves. The root node is considered level 0 and is by this point created.
 - Traverse the already existing tree using entries in the original sequence to decide on which branches to take (originally we start from the first element of the original sequence, and update the starting position by 1 each time through the loop). Keep going until reaching $n-1$ level, where n is the level you are building.
 - If in the course of traversing the tree there is a pass to a location in the original sequence – the subsequence is unique and there is nothing left to do \Rightarrow go to step 3.5
 - If the node on the next level already exists, then it is non-unique \Rightarrow so just update the key
 - If the subsequence is unique, record its starting location within the original sequence instead of the pointer to the next node. Change the value in the same position in the supporting boolean array to true.
 - If the subsequence (next node) is to be repeated (there is already a pass to the original sequence) \Rightarrow create a new node and connect it to the existing tree.
 - Update the starting location of the subsequence being placed into the tree and go back to step 3.1
- Repeat step 3 until the entire supporting boolean array is filled with true values.

This algorithm is of $O(N)$, where N is the number of elements in the original sequence, the number of levels is constant, and is relatively small (less than 4000 for most microbial genomes).

The is_in () algorithm

- Perform a controlled traversal of the tree, using the elements of the target subsequence to choose a direction at each level.
 - If at any point it is not possible to continue in the direction specified, meaning the target subsequence is not present, return *false* and exit the function
 - If the entire target subsequence has been matched without accessing the original sequence – it was not unique \Rightarrow set an appropriate flag to indicate non-uniqueness and return true.
 - If in the course of matching the target subsequence there is a pass to the original sequence, continue matching elements within the original sequence until first mismatch or complete success.

This algorithm is of $O(m)$, where m is the size of the target subsequence.

Statistics

Species	ID	Size (bp)	Max Tree Depth	Number of nodes	Tree growing time (sec ⁽¹⁾)
Agrobacterium tumefaciens strain C58 circular chromosome	NC_003082	2841581	442	2452936	30
Bacillus subtilis	NC_000964	4214814	2959	12259518	3979
Clostridium perfringens	NC_003366	3031430	1970	12944285	1579
Mycobacterium tuberculosis H37Rv	NC_000962	4411529	1699	14568591	1564
Mycoplasma genitalium	NC_000208	580074	245	772273	8
Pasteurella multocida	NC_002663	2257487	3578	15297560	3578

⁽¹⁾ All computations were performed with Dell Workstation 2GHz processor and 4GB RAM

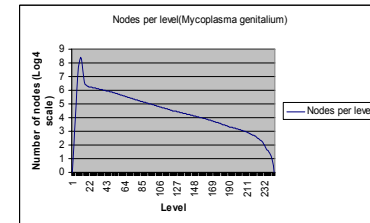


Fig 1. Visualization of the number of nodes created at each level of the incomplete tree for a specific genome - Mycoplasma genitalium

Conclusion

- The proposed data structure allows organizing massive and fast searches, necessary in design of microarray and PCR primers probe design.
- The search speed in the proposed structure is practically equivalent to that of n -ary search tree without the corresponding memory requirements, which allows using PCs to solve these types of problems.

References

Southern, E.M. 2001. DNA microarrays. History and overview. Methods Mol. Biol. 170: 1-15.

Mir, K.U. and E.M. Southern. 2000. Sequence variation in genes and genomic DNA: methods for large-scale analysis. Annu. Rev. Genomics Hum. Genet. 1: 329-360.