Algorithm Details

Peter Dolan

2018-10-26

Let Γ denote the alphabet out of which the strings to be analyzed can be constructed. Without loss of generality Γ = {C,G,T,A}– any modifications for different alphabets being minor. A unique *termination* character, here denoted 'X', is appended to each string and the results are concatenated before analysis. Let this string be denoted `Str1`. (See Algorithm 1a). In a genome reference an 'X' would be used to separate contigs, and in a transcriptome reference to separate the transcripts.

The data structure is a |Γ|-ary tree of stacks with children labelled by the entries of Γ. In our example it is a 4-ary tree for which each node has an 'A'-child, a 'C'-child, a 'G'-child, 'and a 'T'-child. (see Figure 1) The tree is built using depth-first recursion and only the nodes on the path from the root to the current node are retained in memory. The nodes contain stacks of locations within `Str1` that correspond to the path from root to node (see figure below).

Except for temporary processing– the stacks of internal nodes are empty. At each step in the creation-process, a leaf node produces up to four children labeled, A, C, G, or T. The locations contained in the parent node are popped from the stack and the value at that location is used to determine the stack to which that location will subsequently be pushed. Prior to being pushed from the parent-stack onto the child-stack the value of a location is incremented by 1. (See Algorithm 1b)

%For example, if the string to be analyzed started with 'ATAGCG' and an interior node contained 3,4, and 5 (zero-based indexing corresponding to characters 'G', 'C', and 'G') then during processing, the 3 would be popped, the 'G' at location 3 would determine that the destination stack belongs to the 'G'-child and a 4 (three plus one) would be pushed onto the 'G'-child's stack. The next location to be processed, a 4, would be popped and determine that a 5 should be pushed on to the 'C'-child's stack, etc. This process is continued until all non-empty stacks contain only one entry. Encountering an 'X', stops the process regardless of the number of entries on a stack (this requires a bit of care to deal with gracefully in some applications). Using depth first recursion the tree is produced by analyzing the nodes depth-first. Depth information can be passed as an argument to the recursively called function which allows a modified form of the algorithm to be stopped at a particular depth (guaranteeing a linear run-time).

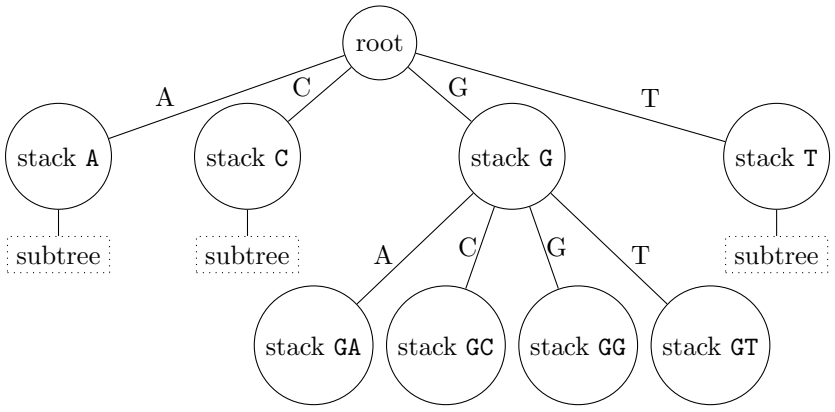

Figure 1: Tree of Stacks

Algorithm 1 Overview

(a) **Setup**

```
1: Append a stop-symbol to every string to be analyzed and concatenate
2: Str1 ← cocnatenated string
3: n ← Str1.length
4: Create root node and root node stack
5: Push 0:(n-1) to the root node's stack
6: Push -1 to the root node's stack
7: ProcessNode(root,0)
```

(b) **Process Node**

```
1: function ProcessNode(node, depth)
2:     while stack not empty do
3:         l ← pop(node)
4:         if Str1[l] =='X' then
5:             Record depth in array dtou and return
6:         push l + 1 on child-stack indicated by Str1[l]
7:     [Apply any novel processing]
8:     ProcessNode(child A)
9:     ProcessNode(child C)
10:    ProcessNode(child G)
11:    ProcessNode(child T)
```

The path from root to leaf determines the substring associated to the node's stack, but this information does not need to be stored since the depth of the node can be used with any location in that node's stack to determine the substring of associated to the path. This is not used in calculating the distance to uniqueness metric, but can be useful for novel processing needs.

If the reverse complement of a genome sequence is added to the set of strings to be analyzed then the algorithm can detect repetition in both the forward and the reverse complement direction.

First Refinement

Using an actual tree of stacks requires dynamic memory allocation and the memory requirements become nondeterministic. The call-back stack of the recursive function call can be used to encode the tree portion of the tree of stacks described above. The stacks may all be efficiently stored in a single array of fixed size as described below. In this way the memory usage of everything except the run-time call-back stack can be determined before run-time. There is a run-time expense that is incurred for this deterministic approach and if the amount of repetition in the sequence being analyzed is high the call-stack can still run out of memory.

Instead of actual stacks, an array of `longints` becomes a collection of "virtual stacks" denoted in the psuedo-code as `I1`. The index of an entry in array `I1` corresponds to a location in the sequence being analyzed (the current depth of the recursion is added to the index to determined the base-pair location). The value of an entry in `I1` is another index of `I1` that corresponds in the same way to the next base-pair position stored in that "virtual stack". An index value of -1 represents the bottom of the stack. The array `I1` is initialized as the same length of the sequence and contains the values 1 to $n-1$ and a final -1. At this point the `I1` is encoding a single stack containing all the locations. The function loops through all the entries in its stack, stopping when it encounters the value -1. (see Algorithm 2). Upon entry into `recurseTree` The variable `Ip1` holds the top of the current stack during the `while`-loop it continues to represent the top of the stack.

In other words, when the recursion is at $depth = m$ Then $I1[Ip1]$ contains the location of the **next** sub-string identical to $Str1[Ip1 : (Ip1 + m)]$.

Algorithm 2 Recursion structure

```
1: function RECURSETREE(Ip1, depth)
2:     Set children stack index-pointers to -1: A1 = C1 = G1 = T1 = −1
3:     if Ip1 == −1 then return false
4:     if I1[Ip1] < 0 then
5:         dtou[Ip1] ← depth
6:         return false
7:     i ← Ip1
8:     while i ≠ −1 do
9:         i ← Ip1
10:        next ← I1[i]
11:        switch S1[i + depth] do
12:            case 'A'
13:                I1[i] ← A1
14:                A1 ← i
15:            case 'C'
16:                I1[i] ← C1
17:                C1 ← i
18:            case 'G'
19:                I1[i] ← G1
20:                G1 ← i
21:            case 'T'
22:                I1[i] ← T1
23:                T1 ← i
24:            case 'N' or 'X'
25:                I1[i] ← −1
26:                dtou[i] ← depth
27:        i ← next
28:    RecurseTree(A1,depth+1)
29:    RecurseTree(C1,depth+1)
30:    RecurseTree(G1,depth+1)
31:    RecurseTree(T1,depth+1)
```

Second Refinement

By adding a `stackDepth` variable to the procedure the program can swich to a more efficient `while`-loop when only two elements are contained in its stack– this removes the overhead of the long-tail recursion that would otherwise result for highly repetitive regions that are only duplicated once. It does, however, introduce added overhead, both in terms of run-time and memory in the `RecurseTree` function. The run-time penalty is more than paid for by the improvements when the stack-size is down to two, but the extra memory requirements accrued by the variables that hold the stack sizes limit the recursion depth– this can still be an issue when there are more than two copies of a highly repetitive region. Depth-limiting the recursion fixes this problem at the cost of putting an upper-limit on the reported length of a repetitive region.

Algorithm analysis

The run-time is determined by the amount of repetitiveness in the sequence. Each character in the sequence is accessed as many times as the length of the shortest unique subsequence containing it. For example, the complete genome of E. coli K-12 has 97 percent of its unique subsequences of length 19 or less and only 2.8 percent (details of the calculation can be found in the `dtou` package's introductory vignette). This is close to an optimal situation and the run-time for analysis on this 4.64 million base sequence was approximately

3.4 second on an i7-3770 CPU @ 3.40GHz. In contrast the collection of contigs available for the first human chromosome, contain alternate assemblies that are almost completely identical to other regions in the collection. Running the software on that data required depth-limiting to 5,000 to ensure the callback stack didn't over overflow and still took about 38 minutes despite only being 51.4 times larger.

Technically the depth-limited version of the algorithm is linear, being $\mathcal{O}(\text{depth limit} * n)$ but the constants are large enough that the run-time is not particularly satisfying if the repetition level is high.