Approximate Matching with Suffix Trees

This page introduces the k-difference problem and uses suffix trees to extend the simple O(nm) dynamic programming technique to achieve an O(kn) time and O(m) space bound.
For an introduction to suffix trees see Suffix Trees in Computational Biology or the collection of links therein.

For a detailed explanation of suffix trees and their uses, see *"Algorithms on Strings, Trees, and Sequences"* by Dan Gusfield. This book was the reference for most of the information on this page.

Outline

Introduction

Approximate matching is a common real world problem with application to a variety of problems including: signal recovery, DNA sequence matching, and

pattern matching in a corrupt text. Numerous approximate matching techniques are available, and they fall in two broad categories: indexed search algorithms and online algorithms. Navarro [NAV2001] sites several problems with indexed search techniques for approximate matching including: a prohibitive increase in space required for approximate search indices; text volatility preventing the cost of index creation from being amortized over multiple searches; and simple inadequacy--indexing methods often fail to produce adequate speedup as the cost of indexing must be averaged over a number of queries.

Online algorithms process the matches without preprocessing the text. These algorithms can be divided into four categories: algorithms based on Dynamic Programming matrices, Finite Automata, Filters, and Bit-Parallelism; there are of course hybrids within each category such as Bit-parallel techniques based on automata, or bit-parallel techniques based on DP matrices [NAV2001].

- Dynamic programming techniques are based on the DP relation commonly known (at least in the bioinformatics world) for the sequence alignment algorithms of Needleman-Wunsch and Smith-Waterman [MOU2001, GUS1997]. DP methods give acceptable theoretical time and space bounds of `O(km)` and `O(m)` respectively [GUS1997], and are easily adapted to related problems but they tend to be too slow in practise [NAV2001].
- Algorithms based on Automata are of interest because they produce the best worst case time O(n), which is the lower bound for the approximate matching problem. Unfortunately this is achieved at the expense of a space bound exponential in either `k` or `m` [NAV2001].
- Techniques based on bit parallelism seek to exploit the parallel nature of bit operations inside a computer word. Bit-parallelism is generally used to parallelize other techniques such as the work of creating a non-deterministic automata or filling a dynamic programming matrix. This technique generally works quite well for short patterns and achieves significant speedups over non-parallel implementations. The speedup is achieved by exploiting the computers ability to perform bitwise operation such as `or` and `and` in `O(1)` time for numbers less than the word length of the computer (i.e. 32 or 64 bits).
- Filtering algorithms seek to reduce the search space of the problem by removing regions which cannot contain the pattern. This is generally achieved by applying exact matching techniques to small pieces of the

problem. Once candidate regions have been selected a non-filtering algorithm is applied to the candidate regions to find the pattern.

The k-difference problem

An efficient solution to the k-difference problem is presented. The k-difference problem is defined as follows [NAV2001]:

> Given `T`, `P`, `k`, and d(,) return the set of all text positions `j` such that there exists `i` with d(P,T i..T j) ≤ k.
> Where:
>
> - `Σ` is a finite alphabet of size |∑| = σ
> - T ε ∑* is a text of length `n`.
> - P ε ∑* is a pattern of length `m`.
> - k ε R is the maximum number of errors allowed.
> - d : ∑* χ ∑* -> R is a distance function between two strings

That is, return the list of positions in T where P can be found with no more than k differences.

An edit distance error model will be assumed throughout: the edit distance is the the number of insertions, deletions, or substitutions required to make the pattern match the text. For instance the edit distance of "automata" and "automatic" is 2. Approximate matching techniques are generally only interesting for a small number of mismatches (`k`) because a pattern will match any text given a large enough k-value. For instance the edit distance of the two unrelated words "hot" and "cold" is 4 with an error ratio `α = k/m` of `4/3`. Generally an error ratio between `1/m` and `1/2` is considered reasonable [NAV2001].

The k-mismatch problem is a simpler problem allowing only matches and mismatches--no insertions or deletions--which can be solved in O(kn) time using suffix trees. The same problem can also be solved in O(nm) time using dynamic programming. Extending the dynamic programming approach using suffix trees achieves the same O(kn) bound as the k-mismatch problem, however the DP approach is also able to solve the more difficult k-difference problem in O(kn) time.

To solve either problem in O(kn) time two techniques must be introduced: the Lowest Common Ancestor method and the Longest Common Extension method. These techniques will allow an efficient solution for exact matching with wilds cards, a solution to the k-mismatch problem, and be crucial to the O(kn) k-difference solution. Moving from the k-mismatch problem to the k-differences problem requires the introduction of dynamic programming to handle gaps. A dynamic programming solution to the k-mismatch problem is presented, followed by a hybrid suffix-tree/dynamic programming method, which directly results in a solution to the k-difference problem.

From this point forward the reference is Gusfield [GUS1997] (unless otherwise noted).

Lowest Common Ancestor Methods

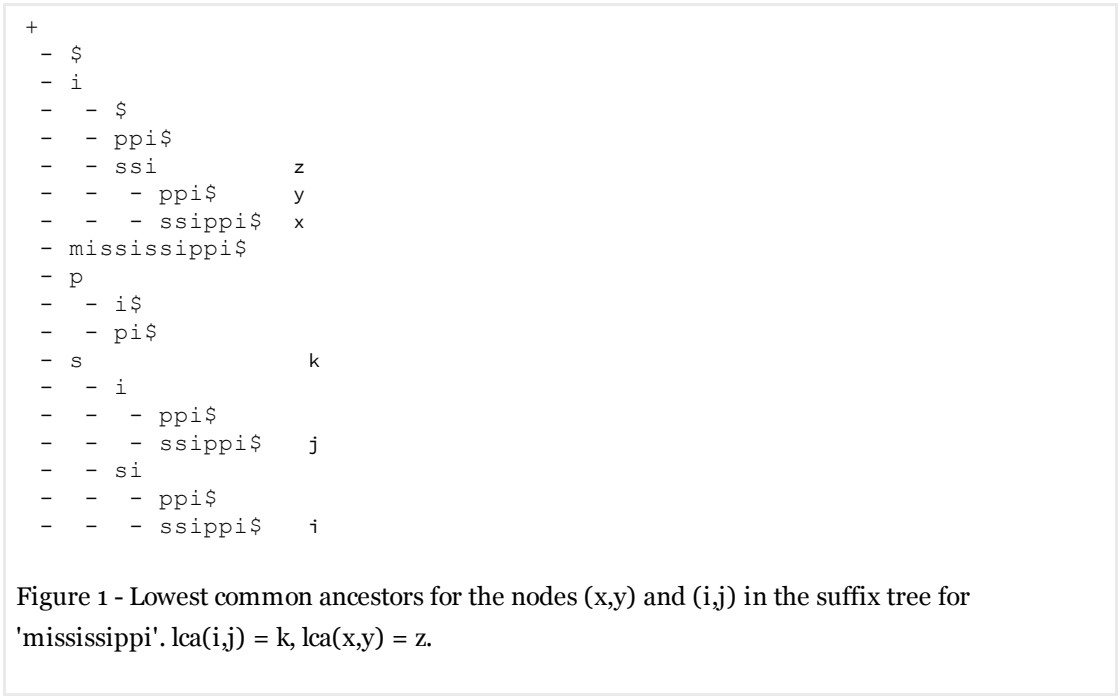


Figure 1 - Lowest common ancestors for the nodes (x,y) and (i,j) in the suffix tree for 'mississippi'. lca(i,j) = k, lca(x,y) = z.

The *lowest common ancestor (lca)* of two nodes `x` and `y` is the deepest node in the tree which is an ancestor of both `x` and `y` (Fig 1). This node can be easily located in `O(n)` time during a traversal of the tree, but more interesting is the fact that given an O(n) preprocessing phase, all subsequent lca queries can be accomplished in O(1) time.

Gusfield [GUS1997] provides an explanation of the Schieber-Vishkin LCA method. The explanation assumes a unit-cost RAM model in which numbers with up to O(log n) bits can have bitwise operations (AND, OR, XOR, shift left/right, locate least/most significant 1-bit) applied to them in constant time.

The complete binary tree

The LCA explanation is based on a complete binary tree model (Fig 2), where each node in the tree is labelled with it's *path number* -- the bits that describe the path from the root the node in the tree. A zero in the `ith` position indicates that the `ith` edge on the path goes left, while a 1 indicates that the `ith` edge on the path goes right. The path number is created by appending a 1 followed by 0's to pad the number out to (log p / log 2)+1 bits. As such the rightmost 1 in the path number marks the node.

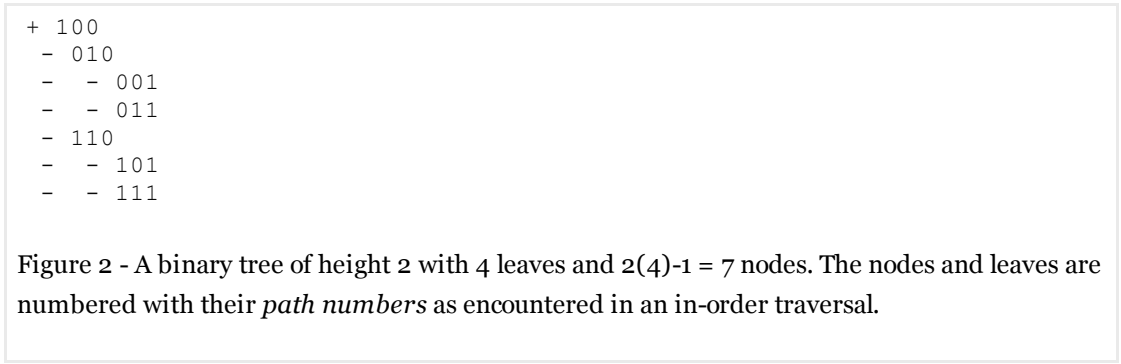


Figure 2 - A binary tree of height 2 with 4 leaves and 2(4)-1 = 7 nodes. The nodes and leaves are numbered with their *path numbers* as encountered in an in-order traversal.

The LCA of two nodes `i` and `j` can be found by taking the XOR of two nodes. The two paths will be the same to the left most 1 in the result. Recall that XOR produces a 1 only if the two bits differ, and that the nodes are labelled based on an in-order assignment of path bits. So up to the point where the two paths differ (the first 1 in the XOR result) the paths are the same. Thus if the first 1 occurs at position `k`, then the first k-1 edges are common to the two nodes.

For instance given i = 001 and j = 011 XOR (i,j) = 010. Thus k = 2 as `k` is the position of the leftmost `1 bit` in the result, and the first k-1 edge of `i` and `j` is common to each node. The first bit of either `i` or `j` is `0` (and must be the same for each node), so adding a 1 padded by 0's to this number we get an lca node number of `010`. Similarly the lca of i = 101 and j = 111 is XOR(i,j) = 010, bit 1 of i or j = 1, and padded with 10 gives a lca node number of `110`.

It is possible to achieve O(1) lca queries based on a mapping from the binary tree to a general tree given an O(n) preprocessing phase and O(n) space after preprocessing. Let's examine the case where we explicitly map from the binary tree to a general tree.

```
+ 1            (0001)
 - 2           (0010)
 -  - 3        (0011)
 -  - 4        (0100)
 - 5           (0101)
 -  - 6        (0110)
 -  - 7        (0111)
 -  -  - 8     (1000)
 -  -  - 9     (1001)
 -  -  - 10    (1010)
```

Figure 3 - general tree with nodes number in a depth-first pre-order traversal. The node numbers are given in binary next to each node.

Mapping the binary tree to a general tree

In a linear time traversal number each node in the tree with it's depth-first pre-order number (Fig 3).

For any node *j* in the general tree, h(j) denotes the position from the right of the least-significant 1-bit in j. i.e. h(8)=4 since 8 in binary is *1000* and h(5)=1 as 5 in binary is *101*.

In the complete binary tree the *height* of a node *k* is the number of nodes on the path from it to a leaf and the height of a leaf is 1. For any node *k* in the binary tree, h(k) equals the height of the node in the complete binary tree. For example, node 4 (binary 100) is at height 3 (Fig 2).

For a node *v* in the general tree, I(v) returns a node *w* in the general tree such that h(w) is a maximum over all nodes in the subtree of *v* (including v). For instance in figure 3 I(1), I(5), I(7) and I(8) are all 8, I(2) and I(4) are both 4, and I(v) = v for v = 3,6,9,10.

If *v* is an ancestor of *w* then h(I(v)) ≥ h(I(w)) because the h(I(v)) value never decreases along any upward path in the general tree. Furthermore for every I(v) there is exactly one w in v's subtree (including v) such that h(w) is a maximum.

The mapping of the general tree to the binary tree aims to preserve enough of the ancestry relations in the general tree to allow lca relations in the binary tree to determine lca queries in the general tree. The function I(v) is central to this operation.

First partition the general tree into classes whose members have the same$I$. G= {1,2,3,4,5,6,7,8,9,10} (Fig 3) can be partitioned into 6 runs {{1,5,7,8},{2,4},{3},

{6},{9},{10}}. A *run* is a maximal subset of nodes in the general tree that all have the same *I* value.

The I values can be assigned for each node in a linear time, bottom up traversal as follows:

> For every leaf v set I(v)=v. For every internal node v set I(v) = v if h(v) is greater than h(v') for every child v' of v. Otherwise, I(v) is set to the I(v') value of the child v' whose h(I(v')) value is the maximum over all children of v.

This results in each run forming an upward path on nodes in the general tree because the h(I()) values never decrease. So for any node v, I(v) refers to the deepest node in the run which contains node v.

This allows the *tree map* to be defined as the mapping of nodes from the general tree to the complete binary tree such that every node v of the general tree maps to the node I(v) of the complete binary tree.

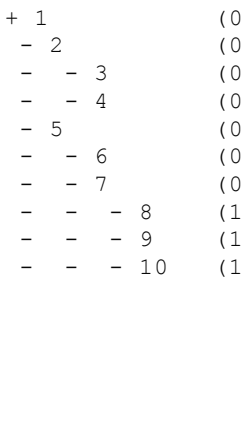


Figure 4a - the general tree with nodes numbered in a depth-first pre-order traversal. The node numbers are given in binary next to each node. I(v) values are indicated following the binary representation.

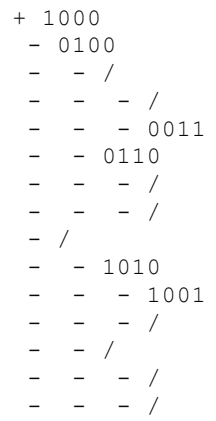


Figure 4b - Each node v in the general tree has been mapped to it's I(v) value in the binary tree. Nodes labelled with a / are not involved in the mapping.

The O(n) preprocessing algorithm

1. Do a depth first traversal of the general tree to assign depth-first search numbers to the nodes. For each node set a pointer to its parent.
2. Using the linear time bottom up algorithm described earlier, compute I(v) for each node. For each k such that I(v) = k for some v, set L(k) to point to the head of the run containing k.
3. - The head of a run is identified when computing I values. v is identified as the head of its run if the I value of v's parent is not I(v).
   - After this step, the head of a run containing an arbitrary node v can be located in constant time. First compute I(v) then look up the value L(I(v)).
4. Given a complete binary tree with node-depth ceiling(log n)-1, map each node v in the general tree to I(v) in the binary tree (Fig 4).
5. For each node v in the general tree create on O(log n) bit number A v. Bit A v(i) is set to 1 if and only if node v has an ancestor in the general tree that maps to height i in the binary tree. i.e. iff v has an ancestor u such that h(I(u)) = i. {Need to track information about the node in the binary tree v mapped to. This step is performed in a linear traversal of the tree after performing steps 1-3.}

Given the above preprocessing step on the general tree, it is possible to guarantee that if z is an ancestor of x in the general tree, then I(z) is an

ancestor of I(x) in the binary tree. That is, if z is an ancestor of x in the general tree, then either z and x are in the same run in the general tree, or I(z) is a proper ancestor of I(x) in the binary tree. This guarantee allows lca queries to be answered in constant time.

Answering an lca query in constant time

Let x an y be two nodes in the general tree and z be their lowest common ancestor. If we know h(I(z)) we can find z in constant time.

Let b be the lowest common ancestor of I(x) and I(y) in the binary tree. Let h(b)=i and j be the smallest position ≥ i such that both A x, and A y have 1-bits in position j. Then node I(z) is at height j in the binary tree, or h(I(z)) = j.

1. Find the lowest common ancestor b in the binary tree of nodes I(x) and I(y).
2. Find the smallest position j ≥ h(b) such that both numbers A x and A y have 1-bits in position j. This gives j = h(I(z)).
3. Find node x′, the closest node to x on the same run as z:

4. 1. Find the position l of the right-most 1 bit in A x
   2. If l = j, then set x′ = x {x and z are on the same run in the general graph} and go to step 4.
   3. Find the position k of the left-most 1-bit in A x that is to the right of position j. Form the number consisting of the bits of I(x) to the left of the position k, followed by a 1-bit in position k, followed by all zeros. {That number will be I(w)}
      Look up node L(I(w)), which must be node w. Set node x′ to be the parent of node w in the general tree.

5. Find node y′, the closest node to y on the same run as z using the approach described in step 3.
6. If x′ < y′ then set z to x′ else set z to y′

It should be noted that the binary tree used throughout this explanation is necessary only for the explanation. Only step one of the constant time algorithm relied on the binary tree to find b relative to I(x) and I(z), for use in step 2 to find h(b). Instead h(b) can be found by finding the rightmost common 1-bit of I(x) and I(z), which means that b is unnecessary.

The Longest Common Extension

Given two strings `S1` and `S2` and a *long* list of index pairs (i,j), the problem is to find the length of the longest substring of `S1` starting at `i`--that is the longest prefix of suffix `S1[i]`--matching the longest prefix of suffix `S2[j]` for each pair (i,j).

This question can be answered in O(n) time using the LCA method described above.

- Create the GST corresponding to S1 and S2.
- Process the GST so that it can do constant time LCA queries; this requires time O(n).
- Find the lca of the two leaves corresponding to the suffixes `S1[i]` and `S2[j]`. The string depth of this node gives the longest common extension for the pair (i,j); this requires constant time per query.

Exact matching with wild cards

The above longest common extension method allows all matches of `P` in `T` given `k` wild cards throughout the strings (a wild card is allowed to match a single character) to be found in O(km) time, where m ≥ n is the length of `T` and `n` is the length of `P`.

The algorithm works by first aligning the left end of `P` with a position in `T` and then advancing left through the two strings performing longest common extension queries. When a mismatch occurs the algorithm checks that it occurred at a wild card, if so it performs another longest common extension query, if not the next position in T is examined. A match is found when the lce queries extend the length of `P`.

Given lce(P[j], T[i]) = l -- the longest common extension of suffix P[j] and T[i].

- for i = 1; i > m - n + 1; i++

- 1. j = 1, i′ = 1
  2. l = lce(P[j], T[i′]
  3. if l+j = n + 1, P occurs in T starting at i; continue next i;
  4. if P[l+j] is a wildcard, or T[i′+l] is a wildcard, j=j+l+1, i′=i′+l+1; Go to 2

5. else P does not occur in T starting at i; continue next i;

The outer loop repeats O(n) times, while the inner loop repeats at most k times -- assuming each wildcard is examined. Thus the overall timing is O(kn) after an O(n+m) preprocessing step to setup the tree for lca and lce queries.

k-mismatch problem

A k-mismatch is a match of pattern `P` in a text `T` with at most `k` mismatches. That is, a region |P| characters long in `T` which matches |P|-k characters in P.

The k-mismatch problem is to find all k-mismatches of `P` in `T`.

Suffix Tree Solution

The Suffix Tree approach to the k-mismatch problem is essentially the same as the approach taken when matching with wild-cards; though in this case `k` is the number of allowable mismatches rather than the number of wildcards. The algorithm proceeds by executing up to `k` constant-time lce queries for each position `i` in the text. If the the lce queries span the end of `P`, then `P` occurs starting at position `i` of the text.

- for i = 1; i > m-n+1; i++

- 1. j = 1, i′ = i, count = 0
  2. l = lce (P[j], T[i′])
  3. if j+l = n+1, P occurs in T at i with only count mismatches; continue next i;
  4. if count ≤ k, k++, j = j+;+1, i′=i′+l+1, go to 2
  5. if count = k+1, then a k mismatch of P does not occur at i; continue next i;

An alternate approach, when both k and the alphabet size are relatively small is to generate all permutations of P with up to k changes and then search for each of these permutations in the tree. This would run in time O(P′ + m), where P′ is the combined length of the permissible permutations of P.

Dynamic Programming solution

At this point a dynamic programming algorithm for the k-mismatch problem is presented because dynamic programming is required to achieve the O(kn)

solution to the k-difference inexact matching problem. Dynamic programming is needed to handle gaps in the pattern and text, something that is not easily accomplished using suffix trees. Thus an O(mn) dynamic programming algorithm is presented to solve the k-mismatch problem. This algorithm is then extended using suffix trees and longest common extension queries to produce an O(kn) algorithm, then further extended to provide the O(kn) solution to the k-difference inexact matching problem.

Dynamic programming techniques are based on a recurrence relation that uses prior knowledge to compute new values. Past values are stored in a data structure that facilitates efficient lookups. For example the global alignment algorithm of Needleman-Wunsch fills in a matrix, where each new entry depends on one of three previous entries (Fig 5).

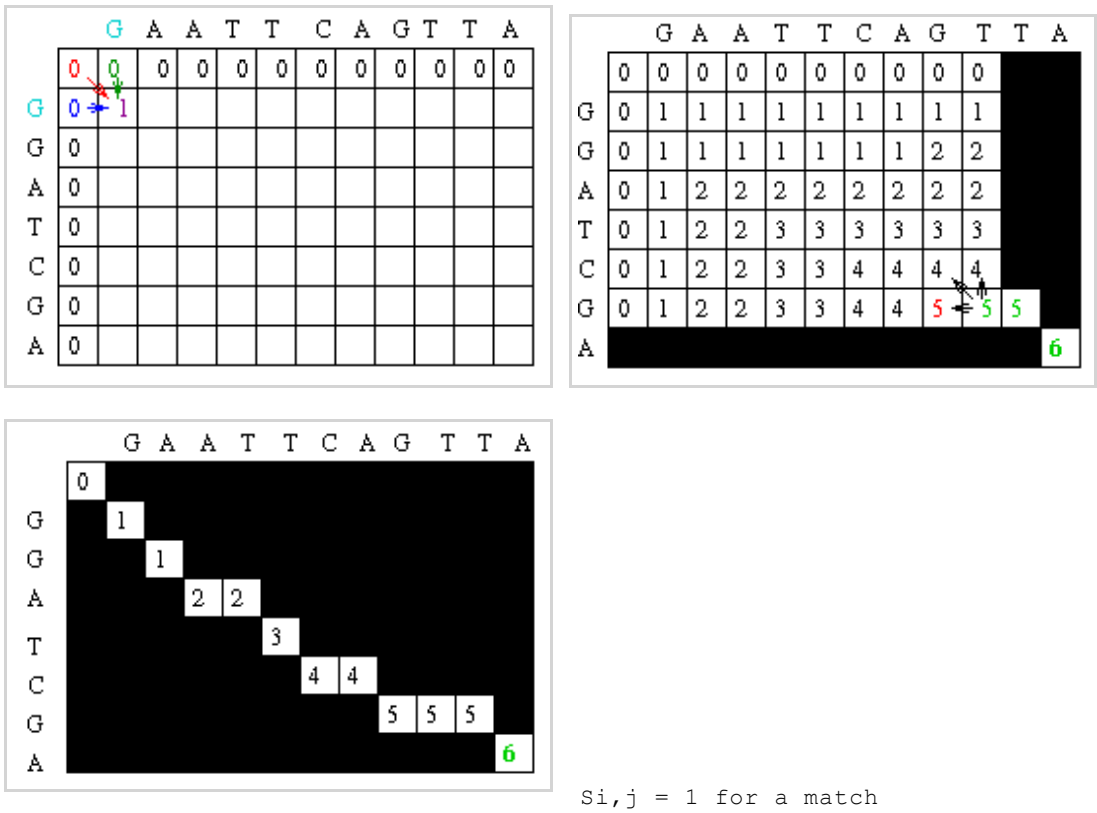
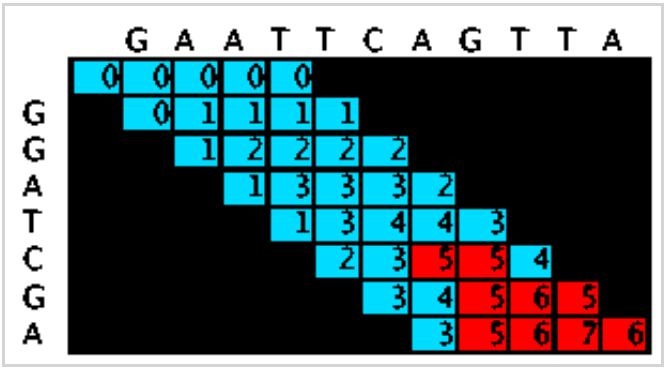


Figure 5. *Top* (left to right): 1. The initialized table. Row one and column one are filled with 0 as this example is using no gap open or gap extension penalty (w=0). Cells are filled from top to bottom (or left to right) with each new entry depending on one of three previous values--as defined by the dynamic programming relation. 2. The filled table. From here a traceback is performed stepping back to the cell with the highest value from which the current value was derived. *Middle* (left to right) 1. The complete traceback showing the optimal alignment of these two sequences. 2. The scores used in the relation. *Middle* 1. The dynamic programming relation for global alignment with the Needleman - Wunsch algorithm. Full example and original images [Rouchka]

The O(mn) global alignment algorithm can be easily modified to solve the k-mismatch problem. Recall that the k-mismatch problem does not allow gaps in either string. This means that only the first term in the dynamic programming relation is relevant: Mi,j = Mi-1, j-1 + Si,j. As the pattern is allowed to start anywhere in the text--gaps at the beginning of the text are not penalized--the top row is initialized with 0. As gaps at the beginning of the pattern are not allowed, any value below the initial diagonal need not be computed. Similarly values beyond the terminal diagonal need not be computed because the pattern cannot be contained in the remaining portion of the text.

Furthermore, if matches are assigned a weight of 0, and mismatches a weight of 1, then no value need be computed once the previous $M_{i-1,j-1}$ score exceeds k (Fig 6). After filling the matrix, all k-mismatches can be located by scanning the final row of the matrix and selecting entries with a value $\leq k$.




```
w = 0

Mi,j
= Mi-1, j-1 + Si,j

Si,j = 0 for a match
Si,j = 1 for a mismatch
```

Figure 6. The modified dynamic programming relation and matrix to find k-matches. Regions of black are never computed; For k = 3 regions of blue will be computed, while regions of red need not be computed.

Dynamic Programming Suffix Tree Hybrid

In order to reduce the time required by this algorithm certain entries in the matrix must be ignored; the desired O(kn) time algorithm permits only k entries per diagonal. Notice that the values in position M[i,j] of the matrix increase only if there was a mismatch between the two characters T[i] and P[j]. Thus the only entries that need to be updated are those M[i,j] where a mismatch occurs. Identifying M[i,j] where mismatches occurs is easily accomplished.

Recall that longest common extension queries for two suffixes starting at `i` and `j`, S[i], T[j] can be computed in constant time. Observe that the lce takes you to the first mismatch in the diagonal starting at M[i,j]. Thus at most `k` lce queries need be performed per diagonal; one starting from the end of each previous lce. After each lce(i,j) = l query the value in M[i+l,j+l] is incremented. A k-mismatch occurs if the lce queries span the length of `P` with no more than k-mismatches (as in the suffix tree solution to the k-mismatch problem).

The k-Difference Algorithm

The k-difference algorithm is a (conceptually) simple extension of the dynamic programming suffix tree hybrid (refer to Gusfield for the full details).

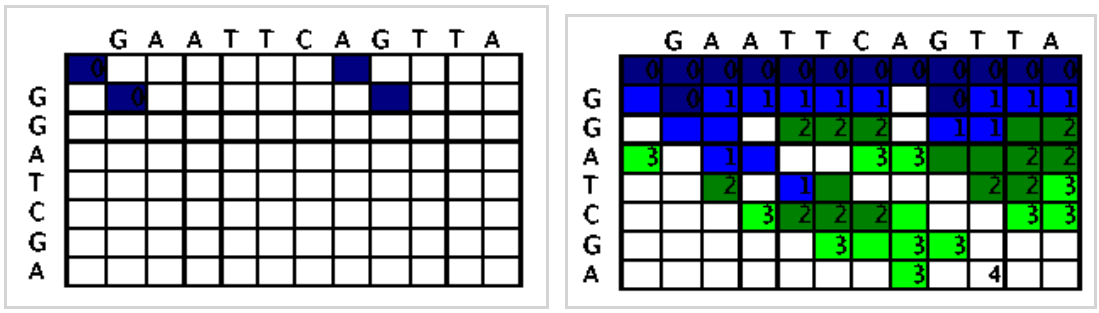


Figure 7. *left* The dynamic programming matrix after calculating 0-paths (blue boxes). *right* The dynamic programming matrix after calculating 1-paths (light blue boxes), 2-paths (dark green boxes), and 3-paths (light green boxes). Two possible k-matches for k≤4 are presented: GAATTCA and GGATCGA with 3 mismatches, and GAATTCAGT and GGAT_C_GT with 2 gaps and 1 mismatch

The algorithm begins by performing all lce queries involving 0 mismatches to find the so called 0-paths (Fig 7). These consist of all longest common extensions starting in row zero of the matrix. It then performs all lce queries containing 1 mismatch; finding the 1-paths. These consist of the diagonal starting at M[1,0], as well as from the lowest point achieved by the 0 mismatch extension with lce (i-1,j-1), lce (i, j-1), or lce (i-1, j). These three values take the place of the score calculated in the global alignment DP algorithm. This process is repeated k times for each of the n strings giving the desired O(kn) time bound.

The complete algorithm as detailed in Gusfield is presented here:

```
begin
d:= 0
for i := 0 to m do
  begin

  For i = 0 to m do
  find the longest common extension between P[1..n] and T[i..m]
.
  This specifies the end column of the farthest-reaching d-path
 on diagonal i.



  for d = 0 to k do
  begin
```

```
    for i = -n to m do
    begin
      using the farthest reaching (d-1) paths on diagonals i, i
-1, i+1, find
      the end, on diagonal i, of paths R1, R2,  and
      R3.  The farthest-reaching of these three paths is the
      farthest reaching d-path on diagonal i;
    end;
  end;

  Any path that reaches row n in column c defines an inexact ma
tch of P in
  T that ends at character c of T and contains at most k differ
ences.
```

A more motivating example

The example presented in Figure 7 fails to provide proper motivation for the k-difference algorithm, as the best match (k=3) identified using the k-difference algorithm is no better than the best match identified with the k-mismatch algorithm. A more interesting example is presented here using the strings "ACTGAACATG" and "TGACATG".

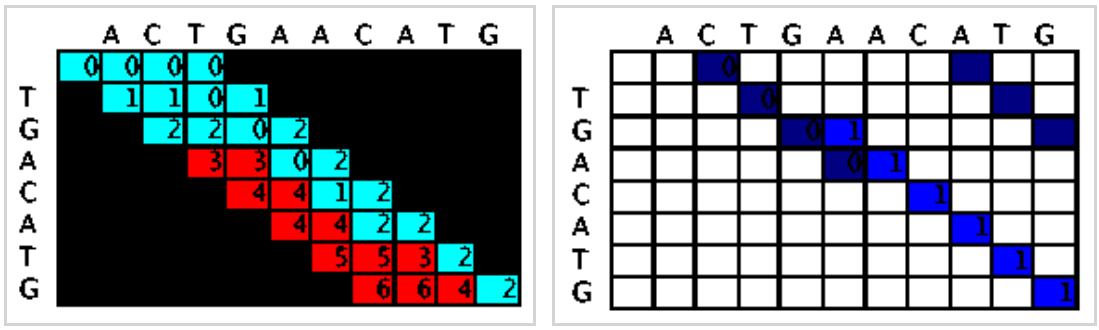


Figure 8. *Left* The k-mismatch table. *Right* The k-difference table showing 0-bands (dark blue) and 1-bands (light blue). Notice that the best k-mismatch has 2 mismatches, while the best k-difference has only a single gap.

```
  ACTGAACATG  ACTGAACATG
    TG ACATG    TGA CATG
```

Figure 9. The two optimal matches found by the k-difference algorithm. Note that there are two matches due to the overlap seen in the middle of the DP table in Figure 8 (*right*).

In this example the more powerful k-differences algorithm is able to find a better match than the k-mismatch algorithm because it can shift the last half of the pattern 1 space (insert a gap in the pattern) to find the optimal match (1 differences), while the k-mismatch algorithm is forced to pair mismatching characters and finds a lesser match (2 mismatches).

Conclusions

The speed and versatility of suffix trees was presented through the solutions to the Exact Matching with Wildcards problem and the k-mismatch problem. These solutions were enabled through the use of the Lowest Common Ancestor method and the Longest Common Extension method. These two techniques were crucial for this problem can be applied to many string problems.

Dynamic programming provides a powerful and flexible method for comparing pairs of sequences, but the O(nm) algorithms are too slow to be practical. On the other hand suffix trees provide efficient string operations but are not well suited to handling gaps in either sequence. A hybrid of these two methods was able to exploit the efficiency of the suffix tree and the the power of dynamic programming to provide an acceptable bound on the k-difference problem.

References

- [GUS1997] Dan Gusfield. *Algorithms on Strings, Trees and Sequences.* Cambridge Univ Pr. Jan 15, 1997.
- [MOU2001] David W. Mount. *Bioinformatics Sequence and Genome Analysis*. Cold Spring Harbour Laboratory Press. 2001.
- [NAV2001] Gonzalo Navarro. *A Guided Tour of Approximate String Matching*. ACM Computing Surveys, No. 1, March 2001, pp.31-88.
- [ROUCHKA] Dynamic Programming. http://www.sbc.su.se/~per/molbioinfo2001/dynprog/dynamic.html.

Original URL:
http://homepage.usask.ca/~ctl271/810/approximate_matching.shtml