Faster Alignment Calculations via Recursion and Culling

Daniel Speyer, dls2192

Abstract

Alignment is a way of comparing two sequences, usually of DNA. It computes a score based on how unlikely a series of insertions, deletions and changes are needed to transform one into the other. It is used throughout bioinformatics for many purposes. The standard way to compute it is a dynamic algorithm which computes the alignment of every pair of start-anchored subsequences. In this paper, I propose an alternative which avoids most of these calculations, and present data on its real-world speed. The worst-case asymptotic behavior remains quadratic, and the greater code complexity often results in a net-slowdown, but for the right problem it can be effective.

Best-Case Predictions

If we are given two sequences of known length, we can state the highest-possible alignment score without actually examining the sequences. If the sequences are the same length, the best possibility is that they are identical, in which case the alignment score is matchScore · length. If they are not the same length, then there must be sufficient insertions to make up the difference, but the rest could match, making the score (matchScore · shorterLengh) – (insertionPenalty · lengthDifference). There could be more insertions with deletions to cancel them out, but those would only decrease the score, and we are computing a maximum possibility,

Culling

The principle of possibility-culling is simple. If we are only interested in the best valid alignment of two sequences, and we already know one alignment is possible, we are not interested in possibilities that are worse than the one we have. This is useful because alignment scores are generally found using subsequences. If we know the score of one side of a split and we already have a possibility in mind for the whole sequence, we know how good the other side of the split needs to be in order to matter.

Putting it Together

Let us now consider an algorithm that finds an alignment score between two sequences that is better than a given target, or determines that no such alignment exists. We will keep track of both known-best alignments and known-to-be-worse-than alignments.

- If we already know the alignment, use that
- If we already know that the alignment is worse than our target, return failure
- If the best possible score for sequences of these lengths is worse than our target, return failure
- First try aligning the last characters to each-other, then try ending with an insertion on either side, For each case:
  - Calculate how good the subsequence match would need to be to meet our target
  - Recursively calculate that alignment
  - If it succeeds, use that as our new target
- If we had any successes, return and remember the best
- If not, remember that this alignment cannot be better than our target and return failure.

In practice, this means that we will only consider as many insertions as could be worth it. See figure 1 for a detailed example.

	-	a	a	a	c	c	a	t	t	t	g	a	a	t	g	g	a	t	g	t	c
-	**0**	**-2**	-4	-6	-8	-10	-12	-14	-16	-18	-20	-22	-24	-26	-28	-30	-32	-34	-36	-38	-40
a	-2	1≤**1**	**-1**	-3	-5	-7	-9	-11	-13	-15	-17	-19	-21	-23	-25	-27	-29	-31	-33	-35	-37
t	-4	-1	0≤**0**	**-2**	-4	-6	-8	-8	-10	-12	-14	-16	-18	-20	-22	-24	-26	-28	-30	-32	-34
g	-6	-3	-2	-1≤**-1**	**-3**	-5≤**-2**	-7	-9	-9	-11	-11	-13	-15	-17	-19	-21	-23	-25	-27	-29	-31
g	-8	-5	-4	-3≤**0**	-2≤**-2**	**-4**	-6≤**-1**	-8	-10	-10	-10	-12	-14	-16	-16	-18	-20	-22	-24	-26	-28
a	-10	-7	-4	-3	-4≤**1**	-3≤**-1**	**-3**	-5≤**-2**	-7≤**-1**	-9	-11	-9	-11	-13	-15	-17	-17	-19	-21	-23	-25
t	-12	-9	-6	-5	-4	-5≤**2**	-4≤**0**	**-2**	-4≤**-1**	-6≤**0**	-8	-10	-10	-10	-12	-14	-16	-16	-18	-20	-22
g	-14	-11	-8	-7	-6	-5	-6≤**1**	-4≤**-1**	**-3**	-5≤**0**	-5≤**1**	-7	-9	-11	-9	-11	-13	-15	-15	-17	-19
t	-16	-13	-10	-9	-8	-7	-6	-5≤**2**	-3≤**0**	**-2**	-4≤**-1**	-6≤**2**	-8	-8	-10	-10	-12	-12	-14	-14	-16
c	-18	-15	-12	-11	-8	-7	-8	-7≤**3**	-5≤**1**	-4≤**-1**	**-3**	-5≤**0**	-7≤**3**	-9	-9	-11	-11	-13	-13	-15	-13
a	-20	-17	-14	-11	-10	-9	-6	-8	-7≤**2**	-6≤**0**	-5≤**-2**	**-2**	-4≤**1**	-6≤**3**	-8	-10	-10	-12	-14	-14	-15
a	-22	-19	-16	-13	-12	-11	-8	-7	-9	-8≤**3**	-7≤**1**	-4≤**-1**	**-1**	-3≤**2**	-5≤**2**	-7≤**3**	-9	-11	-13	-15	-15
t	-24	-21	-18	-15	-14	-13	-10	-7	-6	-8	-9≤**2**	-6≤**0**	-3≤**-2**	**0**	-2≤**1**	-4≤**1**	-6≤**2**	-8	-10	-12	-14
c	-26	-23	-20	-17	-14	-13	-12	-9	-8	-7	-9≤**3**	-8≤**1**	-5≤**-1**	**-2**	**-1**	-3≤**0**	-5≤**0**	-7≤**1**	-9	-11	-11
c	-28	-25	-22	-19	-16	-13	-14	-11	-10	-9	-8	-10≤**3**	-7≤**1**	-4≤**-1**	**-3**	**-2**	-4≤**-1**	-6≤**-1**	-8≤**2**	-10≤**3**	-10
g	-30	-27	-24	-21	-18	-15	-14	-13	-12	-11	-8	-9	-9≤**4**	-6≤**2**	-3≤**0**	**-2**	**-3**	-5≤**-1**	-5≤**0**	-7≤**1**	-9≤**4**
a	-32	-29	-26	-23	-20	-17	-14	-15	-14	-13	-10	-7	-8	-8≤**3**	-5≤**1**	-4≤**-1**	**-1**	**-3**	-5≤**-2**	-6≤**-1**	-8≤**2**
c	-34	-31	-28	-25	-22	-19	-16	-15	-16	-15	-12	-9	-8	-9≤**4**	-7≤**2**	-6≤**0**	-3≤**-2**	-2≤**-2**	**-4**	-6≤**-1**	-5≤**0**
t	-36	-33	-30	-27	-24	-21	-18	-15	-14	-15	-14	-11	-10	-7	-9≤**5**	-8≤**3**	-5≤**1**	-2≤**-1**	-3≤**-3**	**-3**	-5≤**-2**
t	-38	-35	-32	-29	-26	-23	-20	-17	-14	-13	-15	-13	-12	-9	-8	-10≤**6**	-7≤**4**	-4≤**2**	-3≤**0**	-2≤**-2**	**-4**

Figure 1: An alignment matrix of two short sequences using this method. The algorithm only computes the numbers in bold, skipping everything else. Matches are worth 1, mismatches -1 and insertions -2.

Performance

The performance of this algorithm depends on the data being compared. If the sequences match perfectly, only n cases must be considered, whereas if they are completely different, almost $n^2/2$ must be.

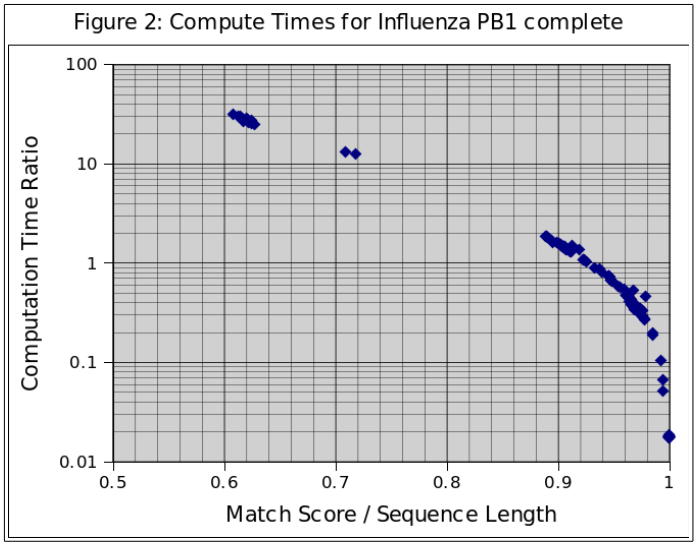
In order to collect practical data, I downloaded 20 versions of the Influenza Polymerase Basic 1 gene from NCBI. The choice of gene was largely arbitrary. PB1 is conveniently sized (between 2250 and 2300 base pairs) and there are many sequenced copies of it.

Because this algorithm can be extremely slow in the worst case, I dropped all pairs of sequences with match scores (still using +1, -1, -2) less than half their average length. Of the 190 possible pairs, this left 119 (including 20 pairs-with-self.

For each pair, I ran both the standard algorithm and my algorithm three times and took the mean, then divided the means. The code is single-threaded and ran on a multicore system with no other cpu-intensive tasks, so the data should be fairly reliable.

As expected, the resulting times were almost monotonic. For sequence pairs with match ratios over 0.93, this algorithm outperformed the traditional, while for those below it did not. Excluding the extremely close matches (those with ratios over .99), the relationship looks exponential. Using simple linear regression, we find that $\ln(\text{timeRatio}) \approx -12.04 \cdot \text{matchRatio} + 10.97$ with $R^2 = 0.96$.

The results are graphed in figure 2, and included in full in the supplemental data file.



Next Steps

It is likely that more small optimizations could be done in the implementation. In particular, recursion is rarely the most efficient. Some speedup could probably be achieved using a manual stack. Also, the stl maps used to hold the data may not be ideal.

Some deeper optimizations may also be possible. Intuition suggests that there are cases in which we can use existing knowledge to precompute a lower upper-bound on a subalignment. Also, there is benefit to trying to more likely subalignment first, and the current heuristic of "aligns, insertion in s1, insertion in s2" cries out for improvement.

In any case, some form of early-abort for poorly matched sequences is clearly needed. This algorithm is unlikely to ever match brute dynamic programming for uncorrelated sequences. In order to get results as quickly as possible, we want to stop trying this algorithm and switch to the other quickly when encountering such a pair.

Finally, there has been considerable work on using subsequence hashing to optimize alignment calculation. Some of those algorithms are nondeterministic, and some call out to dynamic programming to fill in cases they don't handle. Since that approach is completely unlike this one, it is likely that they could be combined to good effect.

Appendix 1: The Code

```
// Compile with g++ --std=c++11
// Run with one argument, a glob at which to find the genes

#include<fstream>
#include<iostream>
#include<limits>
#include<map>
#include<vector>
#include<glob.h>
#include<sys/time.h>

using namespace std;

// Throughout this code, the idea of exploring the subsequence alignment matrix is used.
// Therefore we can talk about subsequences to be aligned as "coordinates".
typedef pair<int,int> coords;
coords operator+(coords a, coords b) {
  return make_pair(a.first+b.first, a.second+b.second);
}
ostream & operator<<(ostream& os, coords c) {
  return os << c.first << "," << c.second;
}

vector<coords> direc = {{-1,-1},{-1,0},{0,-1}};

// neginf needs to be lower than anything we expect to see, but not so low that we get integer underflow
const int neginf = numeric_limits<int>::min() / 2;

// The alignment scores
const int MATCH=1;
const int MISMATCH=-1;
const int INSERTION=-2;

// This is a class just so that its state does not need to be passed in every recursive call
class recursiveCullingAligner {
  map<coords, int> knownexact;
  map<coords, int> knownleq;
  string s1;
  string s2;

  // Returns the alignment, or neginf if it cannot find one
  int align(coords c, int target) {
    int origtarget = target;
    if (c.first<0 || c.second<0) {
      return neginf;
    }
    auto knownit = knownexact.find(c);
    if (knownit!=knownexact.end()) {
```

```
      return knownit->second;
    }
    knownit = knownleq.find(c);
    if (knownit!=knownleq.end() && knownit->second<=target) {
      return neginf;
    }
    int opt = MATCH*min(c.first, c.second) + INSERTION*abs(c.first-c.second);
    if (opt<target) {
      return neginf;
    }
    int ismatch = (s1[c.first-1]==s2[c.second-1]) ? MATCH : MISMATCH;
    int best = neginf;
    for (int i=0; i<3; i++) {
      int d = i ? INSERTION : ismatch;
      int pos = align(c+direc[i], target-d);
      if (pos>neginf) {
       int v = pos + d;
       if (v>target) {
         target=v;
       }
       if (v>best) {
         best=v;
       }
      }
    }
    if (best>origtarget) {
      knownexact[c] = best;
      return best;
    } else {
      knownleq[c]=origtarget;
      return neginf;
    }
  }

public:
  recursiveCullingAligner(string _s1, string _s2) : s1(_s1), s2(_s2) {
    knownexact[{0,0}]=0;
  }

  int calculateScore() {
    return align({s1.length(), s2.length()}, neginf);
  }
};

// The standard dynamic programming approach
int plainalign(string a, string b) {
  vector< vector<int> > m;
  int al = a.length();
  int bl = b.length();
  m.resize(al+1);
  for (auto& i : m) {
    i.resize(bl+1);
  }
  for (int j=0; j<=bl; j++) {
    m[0][j] = INSERTION*j;
  }
  for (int i=1; i<=al; i++) {
    m[i][0] = INSERTION*i;
    for (int j=1; j<=bl; j++) {
      int best = m[i-1][j]+INSERTION;
      int maybebest = m[i][j-1]+INSERTION;
      if (maybebest > best) best = maybebest;
      maybebest = m[i-1][j-1]+(a[i-1]==b[j-1] ? MATCH : MISMATCH);
      if (maybebest > best) best = maybebest;
      m[i][j]=best;
    }
  }
  return m[al][bl];
}

int operator-(timeval t1, timeval t2) {
  return 1000000*(t1.tv_sec-t2.tv_sec)+t1.tv_usec-t2.tv_usec;
}
```

```
// Compare the two algorythms for a given string pair
void compare(string s1, string s2, int reps) {
  float plainTime=0, recTime=0;
  int plainVal, recVal;
  timeval t1,t2;
  float length = (s1.length()+s2.length()) / 2.0;
  for (int i=0; i<reps; i++) {
    // Test normal algorithm
    gettimeofday(&t1,NULL);
    plainVal=plainalign(s1,s2);
    gettimeofday(&t2,NULL);
    plainTime+=static_cast<float>(t2-t1)/reps;
    if (plainVal < length/2) {
      // These are so bad we do not want to wait for recursiveCulling
      return;
    }
    // Test recursive culling
    gettimeofday(&t1,NULL);
    recVal=recursiveCullingAligner(s1,s2).calculateScore();
    gettimeofday(&t2,NULL);
    recTime+=static_cast<float>(t2-t1)/reps;
    // Make sure the answers match
    if (recVal!=plainVal) {
      cout << "ERROR: different values!" << endl;
      cout << "s1=" << s1 << endl;
      cout << "s2=" << s2 << endl;
      cout << "recVal=" << recVal << " plainVal=" << plainVal;
      exit(1);
    }
  }
  cout << length << ", " << plainTime/1000 << ", " << recTime/1000 << ", " << recVal << endl;
}


main(int argc, char** argv) {
  if (argc!=2) {
    cout << "Must pass exactly one argument, a glob where the genes can be found.\n"
         << "Remember not to let the shell expand it." << endl;
    exit(1);
  }
  glob_t glob_result;
  glob(argv[1],GLOB_TILDE,NULL,&glob_result);
  vector<string> strings(glob_result.gl_pathc);
  for(int i=0; i<glob_result.gl_pathc; ++i){
    ifstream ifs(glob_result.gl_pathv[i]);
    ifs >> strings[i];
    ifs.close();
  }
  cout << "length, plain, rec, match" << endl;
  for (int i=0; i<strings.size(); i++) {
    for (int j=0; j<=i; j++) {
      compare(strings[i],strings[j],3);
    }
  }
}
```

Appendix 2: The Genes

The genes used in this were AB539737.1 AF250477.1 AF398871.1 AY582046.1 AY582047.1 AY582048.1 AY582049.1 AY582050.1 AY582051.1 AY582052.1 AY582053.1 AY582054.1 AY582055.1 AY582056.1 AY582057.1 AY582058.1 DQ415294.1 DQ415295.1 DQ415296.1 JX473008.1

They were the 20 returned by

http://www.ncbi.nlm.nih.gov/nuccore/?term=influenza+pb1+complete

then downloaded using

http://www.ncbi.nlm.nih.gov/sviewer/viewer.fcgi?val=${i}&db=nuccore&dopt=genbank&retmode=text

and cleaned using

```
for i in *; do
  cat $i  |
    perl -e 'while(<>){/ORIGIN/ and last;}while(<>){print;}' |
     tr -c -d actg > gene_$i;
done
```

This technique has little to recommend it, but it does work, and it is unlikely that it introduced any relevant biases.