Algorithms and Data Structures for Biology
18 March 2019 — Lab Session

Ugo Dal Lago Thomas Leventis

1 A Solution to Last Week's Assignment

A solution to last week's assignment, seen as an algorithm, is the following one:

Require: s is a string, p is a pattern, d is a natural number
Ensure: P is the set of starting positions in s of all substrings at distance at most d from p.
  $L \leftarrow \{p\}$
  for $i \leftarrow 1$ **to** d **do**
    for all $r \in L$ **do**
      $L \leftarrow L \cup \{q \mid q \text{ is at distance 1 from } r\}$
    end for
  end for
  $P \leftarrow \emptyset$
  for all $r \in L$ **do**
    for $i \leftarrow 1$ **to** $|s| - |r|$ **do**
      $u \leftarrow$ the string of length $|r|$ starting at position i in s
      if $u = r$ **then**
        $P \leftarrow P \cup \{i\}$.
      end if
    end for
  end for

Implementing it as a Python algorithm requires a bit of care, and is bound to produce a program which is structurally much more complicated than the pseudocode algorithm.

First of all let us define the list of characters we consider:

```
Sigma = ['A','T','C','G']
```

In the pseudocode we use the "set of all q at distance 1 from r", but this operation does not correspond to a simple Python code. We define a function which, given a pattern q returns the set L of all strings which are at distance 1 from the argument string q:

```
def Close(q):
        L = []
        for i in range(len(q)):
                L.append(q[:i] + q[i+1:])
                for c in Sigma:
                        L.append(q[:i] + c + q[i:])
                        if c != q[i]:
                                L.append(q[:i] + c + q[i+1:])
        for c in Sigma:
                L.append(q + c)
        return L
```

This function is overly simple and could easily be improved, for instance by checking if a string is already in L before appending it. But it is correct in the following sense: it returns L such that every string at distance 1 from q is in L, and conversely every string in L is at distance 1 from q.

Next we can use a loop to find all strings at distance 1 from a *list* l of strings.

```
def CloseList(l):
        L = []
        for q in l:
                L = L + Close(q)
        return L
```

By iterating this last function d times we can find all strings at distance at most d from a list of strings l.

```
def Approx(l,d):
        if d == 0:
                return l
        else:
                L = Approx(l,d-1)
                return L + CloseList(L)
```

The whole algorithm can then be seen as the following function.

```
def ApproximateStringMatching(s,p,d):
        P = []
        for r in Approx([p],d):
                P = P + StringMatching(s,r)
        return P
```

where `StringMatching` takes care of computing the ordinary pattern matching as follows:

```
def LocalMatch(s,r,i):
        match = True
        for j in range(len(r)):
                if s[i+j] != r[j]:
                        match = False
        return match

def StringMatching(s,r):
        pos = []
        if len(r) == 0:
                for i in range(len(s)):
                        pos.append(i)
        else:
                for i in range(len(s)-len(r)+1):
                        if LocalMatch(s,r,i):
                                pos.append(i)
        return pos
```

2 Checking Correctness

Checking if a program always computes the correct result is far from simple. The safest way is to prove it mathematically, but this is also the most complex and difficult way to go. In practice (and when some failures are acceptable) an efficient way to test a program is to compare its results on a selection of arguments for which the results are known.

The file `lab2_test.ty` contains instructions to test a function `ApproximateStringMatching(s,p,d)`.

3 Evaluating Efficiency

If you followed the suggested algorithm and tried to applied your program to a distance 4, you probably noticed that the computation takes quite a bit of time.

To know more precisely what your program is doing, you can use the `cProfile` module. Add the line:

```
import cProfile
```

at the beginning of your code, then use the method `cProfile.run` to get detailed information on the computations performed by a function call. For instance try calling:

```
cProfile.run('ApproximateStringMatching(s,p,d)')
```

with your ASM function and some string `s`, pattern `p` and distance `d`.

In general, can you approximately quantify the number of operations performed by your function, depending on the lengths of s and p and the distance d? The suggested algorithm is divided in two parts: first we compute the list L of all strings at distance at most d from p (and s is not used at all), then we look for elements of L in s (and we do not use p or d anymore). Therefore we can reason on these two parts independently. How many strings can there be at distance d from a pattern p? How many times do we need to read each character in s to solve the Approximate String Matching problem?

4 Improving Implementation

Checking if a string s is in a list of strings L can simply be performed by the code `s in L`, but for long lists this is not very efficient. Try to find an algorithm which checks if s is in a set L *while reading each character* $s[i]$ *of* s *only once*. You may want to change the way you describe "sets of strings" to achieve this. Feel free to use whatever structure comes to your mind.

Then use this algorithm to simplify your Approximate String Matching program. How much of a difference does this make?

Do you have any idea to improve your program's efficiency even further?