Simon Fraser University

Department of Computer Science

Bioinformatics Algorithms - Assignment 1

Name: Niloufar Saeidi

ID: 301590708

4.1

We can iterate through all of the elements of X in a way that each time we take the ith element and calculate its distance with the other elements. To avoid calculating a distance multiple times, at each iteration, i iterates through all elements of X while j iterates through all the elements before the one that i is currently pointing at:

Algorithm 1 finding ΔX given X

```
1: for i in range(0, len(X)) do
2:     for j in range (0, i) do
3:         add the absolute value of $X[i] - X[j]$ to $\Delta X$
4:     end for
5: end for
```

4.2

a) $L = \{1, 1, 1, 2, 2, 3, 3, 3, 4, 4, 5, 5, 6, 6, 6, 9, 9, 10, 11, 12, 15\}$
L has 21 elements which means $\frac{n(n-1)}{2} = 21$ so $n = 7$ (ΔX has 7 elements.)
We first add 0 and max(L) to ΔX:
b)$\Delta X = \{0, 15\}$
Then we delete 15 from L: $L = \{1, 1, 1, 2, 2, 3, 3, 3, 4, 4, 5, 5, 6, 6, 6, 9, 9, 10, 11, 12\}$
Now the maximum element of L is 12. The distances of 12 and elements of L are 12 and 3, both of which exist in ΔX. So we add 12 to ΔX and delete 12 and 3 from L:
c) $\Delta X = \{0, 12, 15\}$
$L = \{1, 1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6, 6, 9, 9, 10, 11\}$
Now δ is 11. The distances are 11, 1, 4, all of which are in L. So:
d) $\Delta X = \{0, 11, 12, 15\}$
$L = \{1, 1, 2, 2, 3, 3, 4, 5, 5, 6, 6, 6, 9, 9, 10\}$
Now δ is 10. The distances are 10, 1, 2, 5, all of which are in L. So:
e) $\Delta X = \{0, 10, 11, 12, 15\}$
$L = \{1, 2, 3, 3, 4, 5, 6, 6, 6, 9, 9\}$
Now δ is 9. The distances are 9, 1, 2, 3, 6, all of which are in L. So:
f) $\Delta X = \{0, 9, 10, 11, 12, 15\}$
$L = \{3, 4, 5, 6, 6, 9\}$
Now δ is 9. The distances are 9, 0, 1, 2, 3, 6, while 0 is not in L. So we can not make this choice. instead we can go with 15-9=6. id $\delta = 6$, the distances are 6, 3, 4, 5, 6, 9, all of which are in L. So:

g) $\Delta X = \{0, 9, 6, 10, 11, 12, 15\}$

$L = \{\}$

This leads us to a solution $\Delta X = \{0, 9, 6, 10, 11, 12, 15\}$ which has 7 elements.

To find the other solutions, we have to backtrack. In the last step, our only choice was 6, so we cannot modify that. we go one other step back, and change our choice from 9 to 6. So we have

$\Delta X = \{0, 10, 11, 12, 15\}$

$L = \{1, 2, 3, 3, 4, 5, 6, 6, 6, 9, 9\}$

and we want to choose 6. The distances are 6, 4, 5, 6, 9, all of which are in L. Therefore,

$\Delta X = \{0, 6, 10, 11, 12, 15\}$

$L = \{1, 2, 3, 3, 6, 9\}$

. If we choose 9, the distances will be 9, 3, 1, 2, 3, 6, all of which are in L. So we have another solution:

$\Delta X = \{0, 6, 9, 10, 11, 12, 15\}$

$L = \{\}$

Which is the same as the previous one.

We backtrack to find another solution once gain. We have had all the possible choices in the levels g and f and e. Let us make a different choice in level d. if we choose 5 instead of 10 in this level, the distances will be 5, 6, 7, 10. We do not have 7 in L. so we go back to c). If we choose 4, the distances will be 4, 8, 11 while there is no 8 in L. In level b, we can choose 3 instead of 12. The distances will therefore be 3 and 12, both of which are in L. So:

$\Delta X = \{0, 3, 15\}$

$L = \{1, 1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6, 6, 9, 9, 10, 11\}$.

If we choose 11 now, the distances will be 11, 8, 4 while we do not have 8 in L. So we have to choose 15-11=4. Then the distances will be 4, 1, 11, all of which are in L. So

$\Delta X = \{0, 3, 4, 15\}$

$L = \{1, 1, 2, 2, 3, 3, 4, 5, 5, 6, 6, 6, 9, 9, 10\}$.

If we choose 10, the distances will be 10, 7, 6, 5 but there is no 7 in L. So we have to choose 15-10 = 5. The distances will then be 5, 2, 1, 10, all of which are in L. So

$\Delta X = \{0, 3, 4, 5, 15\}$

$L = \{1, 2, 3, 3, 4, 5, 6, 6, 6, 9, 9\}$.

If we choose 9, the distances will be 9, 6, 5, 4, 6, all of which are in L(we might backtrack here later.) So

$\Delta X = \{0, 3, 4, 5, 9, 15\}$

$L = \{1, 2, 3, 3, 6, 9\}$.

We can't choose 9 because there is no 0 in L. if we choose 15-9 = 6, the distances will be 6,

3, 2, 1, 3, 9, providing a solution:
$\Delta X = \{0, 3, 4, 5, 6, 9, 15\}$
$L = \{\}$.
We back track to see if there exists another solution. we go two levels up to
$\Delta X = \{0, 3, 4, 5, 15\}$
$L = \{1, 2, 3, 3, 4, 5, 6, 6, 6, 9, 9\}$.
and choose 6 instaed of 9. This way we end up with the same previous solution. SO, THERE ARE ONLY TWO DISTINCT SOLUTIONS: $\Delta X = \{0, 9, 6, 10, 11, 12, 15\}$ and $\Delta X = \{0, 3, 4, 5, 6, 9, 15\}$.

4.4

We can use a backtracking algorithm that makes subsets and adds each element to it one by one, and deletes and replaces the elements every time the subset reaches the length m. For example, if our multiset is $\{1, 2, 2, 3\}$ and $m = 3$, then it goes like this: $\{\}$, $\{1\}$, $\{1, 2\}$, $\{1, 2, 2\}$, $\{1, 2\}$, $\{1, 2, 3\}$, $\{1\}$, $\{\}$, $\{2\}$, $\{2, 2\}$, $\{2, 2, 3\}$. I added an if statement to eliminate the repetitive subsets. The algorithm complexity is $O(n * m$ but it may be possible to do it faster if we find a smarter way other than eliminating repetitive subsets after iterating through all of them.

Algorithm 2 All m-subsets of the multiset

```
subsets = []
procedure ALG($n, m, i, X, tempX, tempidx$)
    if tempidx == m then
        for j in range(m) do
            if $data[j] \notin subsets$ then
                add data[j] to subsets
            end if
            return
        end for
    end if
    if $i \geq n$ then
        return
    end if
    $tempX[tempidx] \leftarrow X[i]$
    ALG($n, m, i+1, X, tempX, tempidx+1$)
    ALG($n, m, i+1, X, tempX, tempidx$)
end procedure
```

4.5

$U \oplus V = \{u + v : u \in U, v \in V\}$ and $U \ominus V = \{u - v : u \in U, v \in V\}$. Let's assume $a, b \in U$ and $c, d \in V$. Now according to the definition, $a + c, b + d, a + d, b + c \in U \oplus V$ and $a - c, b - d \in U \ominus V$. Therefore, $\Delta(U \oplus V) = \{a+c-b-d, c-d, a-b, b-a, d-c, a+d-b-c\}$. With doing some rearranging, we can find out that $\Delta(U \oplus V)$ is the same as $\Delta(U \ominus V)$. In the same way, we can start from $\Delta(U \ominus V)$ and see that we can get to $\Delta(U \oplus V)$. This proves the bijection between the two sets to be true.

4.12

The first algorithm that comes to mind is a brute force algorithm. It works by comparing each element of T and S and once there is a mismatch, even if some previous characters in the strings have matched, we start comparing the next element of T with the first element of s all over again. So in the worst case, if length of the string T is n and length of the string s is m, the complexity of this algorithm will be O(m*n). This is the pseudocode:

Algorithm 3 Find the first exact occurrence

```
procedure BruteForce(T, s)
    $i \leftarrow 0$
    $j \leftarrow 0$
    while $i \neq len(T)$ do
        if T[i]==s[j] then
            j = j+1
            i = i+1
        else
            j=0
            i = i+1
        end if
        if j == len(s) then
            return i - len(s)
        end if
    end while
    return -1
end procedure
```

We can reduce the time complexity of this algorithm by adding a policy to avoid going back to the next character in T whenever there is a mismatch. Instead of movig on T for only one character, we can check if there is any prefix in the last matched part that is also a suffix in the last matched part. If there is, we can start comparing the two strings from the character following the suffix. Here's the pseudocode for this new approach:

Algorithm 4 Find the first exact occurrence 2

```
procedure ImprovedAlg(T, s)
    $i \leftarrow 0$
    $j \leftarrow 0$
    while $i \neq len(T)$ do
        if T[i]==s[j] then
            j = j+1
            i = i+1
        else if there is a suffix in s[0:j-1] that is also its prefix then
            $str \leftarrow$ the suffix
            $eop \leftarrow$ the index of the ending character of the suffix in s
            $j \leftarrow eop + 1$
        else
            j=0
            i = i+1
        end if
        if j == len(s) then
            return i - len(s)
        end if
    end while
    return -1
end procedure
```

4.13

Consider $len(s) = m$. If $d_H(s, s') = k$, then at least $m - k$ characters should match in s and s'. I divide the problem to two cases, where in one case the first character of s and s' matches and in the other one it does not. For the first case, I add the variable $flag$ to handle the situation. For the second case, the problem will be a bit more complicated. What comes to my mind is to scan T twice. Once from the left, and once from the right, to handle the second case. the pseudocode will look like this:

Algorithm 5 Find the first mutated occurrence

```
procedure BruteForce_maxK(T, s, k)
    $i \leftarrow 0$
    $j \leftarrow 0$
    $flag \leftarrow 0$
    while $i \neq len(T)$ do
        if T[i]==s[j] then
            j = j+1
            i = i+1
        else
            if flag > k then
                j=0
                i = i+1
                flag = 0
            else
                flag = flag + 1
                j = j+1
                i = i+1
            end if
        end if
        if j == len(s) then
            return i - len(s)
        end if
    end while
    $i \leftarrow len(T) - 1$
    $j \leftarrow len(s) - 1$
    while $i \neq -1$ do
        if T[i]==s[j] then
            j = j-1
            i = i-1
        else
            if flag > k then
                j=len(s)-1
                i = i-1
                flag = 0
            else
                flag = flag + 1
                j = j-1
                i = i-1
            end if
        end if
        if j == -1 then
            len(T) - (return i - len(s))
        end if
    end while
    return -1
end procedure
```

The complexity of this algorithm will be $2 * O(m * n) = O(m * n)$

4.15

I believe that the algorithm that this is a cousin of is SIMPLEMOTIFSEARCH. I will also explain how I got to the conclusion that SIMPLEMOTIFSEARCH is the cousin algorithm and not the other ones. So BRUTEFORCEMOTIFSEARCH and BRUTEFORCEMOTIFSEARCHAGAIN are both computationally more expensive because they blindly go over all the possible motifs. Also, BRANCHANDBOUNDMOTIFSEARCH uses a smart way of eliminating the search space by guessing whether it is possible or not to find a motif in a special branch, which ANOTHERMOTIFSEARCH is obviously not using. So I conclude that SIMPLEMOTIFSEARCH is the most similar to ANOTHERMOTIFSEARCH.

ANOTHERMOTIFSEARCH starts by setting all the starting positions to one and uses recursion to find the best motifs. In the recursive function, the current sequence is given as the input to keep track of the sequences that the algorithm goes through. s is the current motif and t is the total number of sequences. In each recursion, it sets the best score to zero at first and uses a for loop to iterate through the sequences and updates the bestmotif and the bestScore whenever it finds a motif with a higher score than the previous best score found.

In SIMPLEMOTIFSEARCH, the algorithm start is the same as ANOTHERMOTIFSEARCH. However, instead of recursion, it uses a while loop which might make it more efficient. To go through the motifs without using recursion, the algorithm uses the function NEXTVERTEX which jumps from each sibling leaf to the other and once it visited all the leaf children of a node, it jumps to another node and visits the other node's children(which are leaves). The updating process is also the same as in ANOTHERMOTIFSEARCH. Also, both algorithms use the same Score function to evaluate the motifs.