Finding patterns of non-continuous characters with a given gap region from DNA sequences

PhD Annual Talk

Speaker: Minghua ZHANG

July 23, 2003

Content

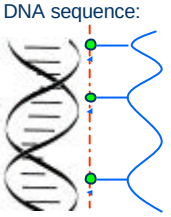
- Motivation
- Problem definition
- Algorithms
- Performance
- Future work
- Conclusion

Background

- The development of bioinformatics provides us with a lot of data.
  - DNA sequences, protein sequences
- The data contains lots of information.
- One way to discover the information is to use the data mining technology.

Motivation

- A DNA sequence:



- The subsequences of nucleotides in the similar orientation may carry useful information.
- Find such subsequences appear often enough.
- Property: Gaps between 2 consecutive nucleotides are not fixed, but within a small region. E.g: [10,11].

Problem Definition

- Σ: the alphabet of the characters
  - E.g: for DNA sequences, Σ={A,C,G,T}.
- Gap:
  - A sequence of wild-cards (".")
  - Length: Number of wild-cards in the gap.
    - x(n) = a gap of length n.
    - x(n,m) = a gap whose length is in [n, m].

Problem Definition (cont'd)

- Pattern:
  - A sequence of characters in Σ, and gaps;
    - Gaps cannot be at the beginning or end of the sequence.
    - E.g: Ax(1)G is a pattern; AGx(1) is not.
  - Length
    - No. of characters in a pattern P.
    - E.g: |Ax(1)G|=2.
  - Subpattern
    - A substring of pattern P, which itself is also a pattern.

Problem Definition (cont'd)

- Match
  - Given a sequence s, a pattern P
  - If we can find an occurrence of P in s, we say s matches P.
  - E.g: s=ACGGACT, P=Cx(2)A, then s matches P at offset 2.

7

Problem Definition (cont'd)

- Support
  - Given a sequence database D={s_1, s_2, … , s_n}, a pattern P
  - sup(P) = No. of sequences in D that match P.
- Frequent
  - If support (P) > = K

8

Problem Definition (cont'd)

- Problem:
  - Given D, K, N, M
  - Find all frequent patterns of form
    - $c_1x(N,M)c_2x(N,M)…c_{l-1}x(N,M)c_l$
  - c_i in Σ, l is any integer.
  - $c_1c_2…c_{l-1}c_l$

9

Property

- If p is frequent, all its subpatterns are frequent.
- However, its subsequences are not necessarily be frequent.
- E.g:
  - if ACGT is frequent,
  - then ACG is frequent,
  - but AGT may be infrequent.

10

Related Works

- Works on finding other types of patterns from biology sequences
  - TEIRESIAS
    - Find patterns composed of characters and wild-cards, but *not* flexible gaps.
      - Only x(n) can appear in the pattern, not x(n,m).
    - A requirement on the ratio of characters w.r.t wild-cards.
    - Roughly a depth-first search method.

11

Related Works (cont'd)

- Pratt:
  - Finding patterns with flexible gaps
  - other restrictions, e.g:
    - The longest length of a pattern
    - The maximal length of a gap, or m in x(n, m)
    - The maximal gap region size, or m-n+1 in x(n, m)
  - By way of graph
    - Scan database and builds a graph according to some regulations
    - Traverses the graph to get patterns

12

Related Works (cont'd)

- Mining frequent patterns from transactional databases (MFP):
  - Difference 1
    - Here: a sequence of characters
    - MFP: a sequence of itemsets
  - Difference 2
    - Here: the gap size between 2 consecutive characters are in a given region [N, M]
    - MFP: order only, no gap size requirement

13

Algorithms

- Algorithm A
  - C_i ={candidate patterns of length i}
  - L_i ={frequent patterns of length i}

```
i=1;
C_i = {i | i is in Σ};
While ( | C_i| >0)
{
    scan database to get L_i from C_i;
    C_{i+1} = Gen( L_i);
    i++;
 }
```

14

Algorithm A (cont'd)

- Count support
  - Given s and P, check whether s matches P.
  - Symbols:
    - P[i]: the i-th character in pattern P
    - S[i]: the i-th character in sequence s
  - Begins with going forward:
    - 1. Find p[1] in s, record the position as off[1]
    - 2. Find the first appearance of p[2] in s with offset no less than off[1]+N, recorded by off[2].
    - 3. If off[2]-off[1] <= M, continue with p[3], etc.
    - 4. Otherwise, going back

15

Algorithm A (cont'd)

- Going back:
  - If off[i]-off[i-1] > M, find a new appearance of P[i-1] in s with offset no less than off[i]-M.
  - If the new value of off[i-1] still meets off[i-1]-off[i-2] <= M, return to the going forward phase;
  - Otherwise, find new positions for P[i-2].
- If we can not find an appearance of a character,
  - s does not matches P.
- If all characters in P are processed successfully
  - s matches P.

16

Algorithm A (cont'd)

- Candidate generation
  - For every pair of patterns p_1 and p_2 in L_i
  - If suffix of p_1 = prefix p_2, then a candidate is got.
    - E.g: p_1= CACG, p_2= ACGT, c=CACGT
- Here we should do subpattern comparison for $|L_i|^2$ times.
  - not efficient

17

Algorithm A (cont'd)

- Observation:
  - When a candidate is generated, we know its prefix and suffix.
    - Keep them for later use.
  - Store L_i as a union of some subsets.
    - All patterns in a subset have the same prefix or suffix.
    - S_{p_u} : a subset, all patterns in it has the same prefix u
    - S_{s_v} : a subset, all patterns in it has the same suffix v
  - After scan the database, if a candidate is found to be frequent, insert it into the right subset S_{p_u} and S_{s_v}.

18

Algorithm A (cont'd)

- When generating C_{i+1} from L_i
  - For each subset S_{s_u}, check if there is a subset S_{p_u}
    - Yes: generate candidates for every pair of patterns p_1 (in S_{s_u}) and p_2 (in S_{p_u}).
  - the generation condition is checked UV times
    - U = the number of subsets S_{p_u}
    - V = the number of subsets S_{s_v}
    - UV < $|L_i|^2$

19

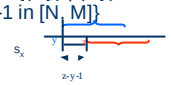
Algorithm B

- Terminologies:
  - id-list of a pattern
    - Given a sequence database D={s_1, s_2, … , s_n}, a pattern P
    - List(P) = {(x,y) | s_x matches P and in the match p[1]=s_x[y]}
  - List (P) -> sup(P)

20

Algorithm B (cont'd)

- id-list calculation
  - Given a pattern R, its prefix P, suffix Q
    - E.g: R=ACGT, P=ACG, Q=CGT
  - List(R) = {(x,y) | (x,y) in List(P) & ∃(x, z) in List(Q) s.t. z-y-1 in [N, M]}



21

Algorithm B (cont'd)

- Sort id-lists with x value as the major key and y value as a minor key.
- A linear scan of List(P) and List(Q)
  - a,b : current position of List(P), and List(Q)
  - If (a.x < b.x) a++;
  - else if (a.x > b.x) b++;
  - else if (b.y-a.y > M) a++;
  - else if (b.y-a.y < N) b++;
  - else insert a into List (R), a++;
  - until reach the end of List(P) or List(Q)

22

Algorithm B (cont'd)

- An iterative method
  - Scan the database to obtain the id-lists of all characters, and get L_1
  - Calculating the id-lists of C_2 (from L_1) and get L_2
  - Calculating the id-lists of C_3 (from L_2) and get L_3
  - Etc.

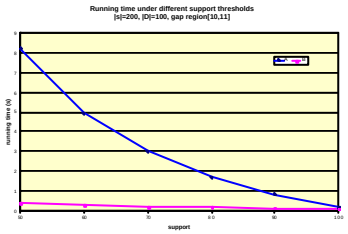
23

Performance

- Data generation
  - Download a DNA sequence from a bioinformatics site.
  - Consider a piece of it with length 200 base pairs.
    - s
  - Generate 100 derivative DNAs from s
    - With a X% similarity to s
  - D = The 100 derivative DNA sequences

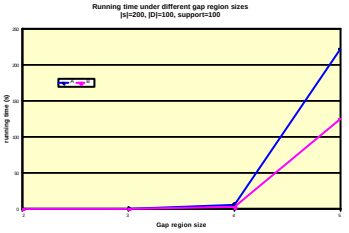
24

Performance (cont'd)



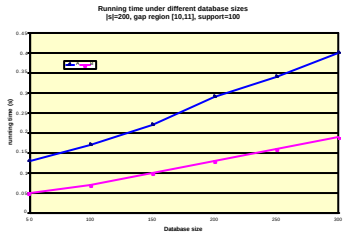
25

Performance (cont'd)



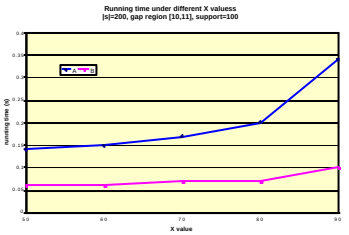
26

Performance (cont'd)



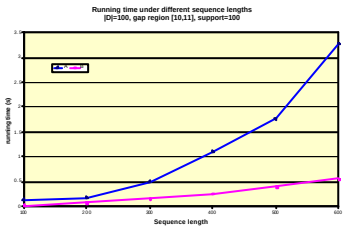
27

Performance (cont'd)



28

Performance (cont'd)



29

Future work

- Algorithms A and B are not efficient when the frequent patterns are very long.
  - They are step by step methods.
- Algorithms with a jump step may be more efficient.

30

Future work (cont'd)

- Idea:
  - L_1 -> L_2 -> L_4 -> L_8, etc.
- Goodness:
  - E.g: if all candidates of length 8 are frequent, we don't need to check patterns of length 5, 6, 7.
- Badness: trace back

31

Future work (cont'd)

- Step size
  - L_i -> L_j (C_j)
  - j is in [i+1, i+i]
  - j-i: [1, i]
  - based on current situation.

32

Future work (cont'd)

- Candidate generation (L_i -> C_j)
  - Given 2 patterns p_1, p_2 in L_i
  - If j=i+1, the suffix of p_1 should be equal to the prefix of p_2.
    - same as algorithms A and B
    - E.g: acgt and cgtt -> acgtt
  - If j=i+i, no requirement.
    - E.g: acgt and cgtt -> acgtcgtt
  - Otherwise, the length-(i+i-j) suffix of p_1 is equal to the length-(i+i-j) prefix of p_2.
    - E.g: i=6, j=9, then i+i-j=3. acgtgc and tgctac -> acgtgctac

33

Future work (cont'd)

- Support counting when step size > 1
  - Like algorithm A: not efficient
  - Like algorithm B
    - The id-list of candidates cannot be computed from its generating subpatterns.
    - acg tg c and tg ctac -> acg tg ctac
  - A mixed way
    - Make use of gap region requirement and id-lists
    - scan the part of the original sequence related

34

Future work (cont'd)

- Backward
  - E.g: when L_4 -> L_8, if not all patterns in C_8 are frequent
  - For such infrequent candidates
    - get their prefix and suffix of length 7
    - If they are not subpatterns of some patterns in L_8
      - Insert them into C_7
  - If not all patterns in C_7 are frequent
    - get their prefix and suffix of length 6
    - If they are not subpatterns of some patterns in L_7 and L_8
      - Insert them into C_6
  - Until we deal with C_5

35

Future work (cont'd)

- Analysis
  - Efficiency depends on
    - How many candidates can be known as frequent without calculating their supports (jump successfully)
    - How many more candidates are generated than algorithms A and B (jump unsuccessfully)

36

Conclusion

- Finding patterns of non-continuous characters with a given gap region from DNA sequences is a new research topic.
- Both algorithms A and B can successfully find out such patterns.
- Algorithm B is more efficient than A.
  - It uses a kind of index, while algorithm A does not.
- The two algorithms are not very efficient when the frequent patterns are very long.
- Future research: a jump method with back retrieval.

37

?

38