COSC 348:
Computing for Bioinformatics

Lecture 4: Exact string searching algorithms

Lubica Benuskova

http://www.cs.otago.ac.nz/cosc348/

1

Definitions

- A *pattern* (*keyword*) is an ordered sequence of symbols.
- Symbols of the pattern and the searched text are chosen from a predetermined finite set, called an *alphabet* (Σ)
  - In general alphabet can be any finite set of symbols/letters
- In bioinformatics:
  - DNA alphabet Σ = {A,C,G,T},
  - RNA alphabet Σ = {A,C,G,U};
  - protein alphabet Σ = {A,R,N,…V} (20 amino acids)

2

Exact string searching or matching

- Much of data processing in bioinformatics involves in one way or another recognising certain *patterns* within DNA, RNA (1st assignment) or protein sequences.
- String-matching consists of finding one, or more or generally all the occurrences of a string of length *m* (called a *pattern* or *keyword*) within a *text* of the total length *n* characters.
- An example of an exact string search (match):

```
– Pat:                EXAMPLE
– Txt: HERE IS A SIMPLE EXAMPLE
```

3

Exact string search algorithms

Algorithm	Preprocessing time	Matching time
Naïve string search algorithm (brute force)	0 (no preprocessing)	average $O(n+m)$, worst $O(n\ m)$
Knuth-Morris-Pratt algorithm	$O(m)$	$O(n)$
Boyer-Moore algorithm	$O(m + \|\Sigma\|)$	$O(n/m)$, $O(n)$
Rabin-Karp algorithm	$O(m)$	average $O(n+m)$, worst $O(n\ m)$
Aho-Corasick algorithm (suffix trees)	$O(n)$	$O(m+z)$

z = number of matches

- **35 algorithms with codes at http://www-igm.univ-mlv.fr/~lecroq/string/**

4

Naïve string search (brute force)

- The most intuitive way is to slide a window of length *m* (pattern) over the text (of length *n*) from left to right one letter at a time.
- Within the window compare successive characters:

```
txt: ABCABCDABABCDABCDABDE
pat: BCD
```

5

Naïve string search (brute force)

- If there is not a copy of the whole pattern in the first *m* characters of the text, we look if there's a copy of the pattern starting at the second character of the text:

```
txt: ABCABCDABABCDABCDABDE
pat:  BCD
```

6

Naïve string search (brute force)

- If there is not a copy of the pattern starting at the second character of the text, we look if there's a copy of the pattern starting at the third character of the text, and so forth:

```
txt: ABCABCDABABCDABCDABDE
pat:   BCD
```

Naïve string search (brute force)

- until we hit a match; then we continue in the same way along the text and count number of matches.

```
txt: ABCABCDABABCDABCDABDE
pat:     BCD
```

Match !

Properties of the naïve search

- Can be used on-line (advantage)
- Usually takes $O(n+m)$ steps – not so bad
- The inner loop finds a mismatch quickly and moves on the next position quickly without going through all the *m* steps
- Worst case scenario $O(nm)$ when searching for `aaab` in `aaaaaaaaaaaaaaaaaaaaaaaaab`

Knuth–Morris–Pratt algorithm

- Integer **`i`** denotes the position within the searched **`txt`**, which is the beginning of the prospective match for `pat`
- Integer **`j`** denotes the character currently under consideration in **`pat`**
- '`-`' denotes a gap in the sequence

```
i:   01234567890123456789012
txt:ABC-ABCDAB-ABCDABCDABDE
pat:ABCDABD
j:   0123456
```

Knuth–Morris–Pratt algorithm

- Slide a sliding window of length *m* (pattern) over the text (of length *n*) from left to right.
- Within the window compare successive characters from left to right until a mismatch is hit.

```
i:   01234567890123456789012
txt:ABC-ABCDAB-ABCDABCDABDE
pat:ABCDABD
j:   0123456
```

Knuth–Morris–Pratt algorithm

- When a mismatch occurs, the pattern itself is used to determine where to jump to the next meaningful position to continue, in this case i = j = 4:

```
i:   01234567890123456789012
txt:ABC-ABCDAB-ABCDABCDABDE
pat:ABCDABD
j:   0123456
```

Knuth–Morris–Pratt algorithm

- From the next meaningful position, i.e. i = 4, we proceed in the same way;
- There is a nearly complete match `ABCDAB` when we hit a mismatch again at `pat[6]` and `txt[10]`.

```
i:  01234567890123456789012
txt:ABC-ABCDAB-ABCDABCDABDE
pat:    ABCDABD
j:      0123456
```

Knuth–Morris–Pratt algorithm

- We passed an "`AB`" which could be the beginning of a new match, so we simply reset `i = 8, j = 2` and continue matching the current character from left to right within a window.

```
i:  01234567890123456789012
txt:ABC-ABCDAB-ABCDABCDABDE
pat:        ABCDABD
j:          0123456
```

Knuth–Morris–Pratt algorithm

- This search fails immediately, as the `pat` does not contain a gap, so we return to the beginning of `pat`, by resetting `j = 0`, and begin searching at `i = 11` in the text.

```
i:  01234567890123456789012
txt:ABC-ABCDAB-ABCDABCDABDE
pat:        ABCDABD
j:          0123456
```

Knuth–Morris–Pratt algorithm

- So we have returned to the beginning of `pat` and begin searching at `i = 11`, resetting `j = 0`.
- Once again we immediately hit upon a match "`ABCDAB`" but the next character, '`C`', does not match the final character '`D`' of the `pat`.

```
i:  01234567890123456789012
txt:ABC-ABCDAB-ABCDABCDABDE
pat:           ABCDABD
j:             0123456
```

Knuth–Morris–Pratt algorithm

- Thus we set `i = 15`, to start at the two-character string "`AB`", set `j = 2`, in the `pat`, and continue matching from the current position.
- This time we are able to complete the match, whose first character is at `txt[15]`.

```
i:  01234567890123456789012
txt:ABC-ABCDAB-ABCDABCDABDE
pat:               ABCDABD
j:                 0123456
```

Match !

Properties of Knuth-Morris-Pratt algorithm

- Can be used on-line (advantage) like naïve search but it's substantially improved.
- Time to find match is only $O(n)$ with $O(m)$ preprocessing time.
- Partial match table should allow not to match any letter of txt more than once.
- Can be modified to search for multiple patterns in a single search.

Boyer-Moore algorithm

- is a particularly efficient string searching algorithm, and it has been the *standard benchmark* for the practical string searching
- BM algorithm holds a window containing `pat` over `txt`, much as the naïve search does. This window moves from left to right, however, its improved performance is based around two clever ideas:
  1. Inspect the window *from right to left*.
  2. Recognize the possibility of large shifts in the window without missing a match.

Boyer-Moore algorithm

```
pat: EXAMPLE
txt: HERE-IS-A-SIMPLE-EXAMPLE
```

- By fetching the S underlying the *last* character of the `pat` we learn:
  - We are not standing on a match (because S isn't `E`).
  - We wouldn't find a match even if we slid the pattern right by 1 (because S isn't `L`), by 2 (because S isn't `P`), etc.

Boyer-Moore algorithm

```
pat: EXAMPLE
txt: HERE-IS-A-SIMPLE-EXAMPLE
```

- Since **S** doesn't occur in the pattern at all, we can slide the pattern to the right ***by its own length*** without missing a match.
- This shift can be pre-calculated for every letter and stored in a table. This table is called a *bad character shift table*.

Boyer-Moore algorithm

```
pat:        EXAMPLE
txt: HERE-IS-A-SIMPLE-EXAMPLE
```

- Focus your attention on the right end of the pattern. `E` is not P, `L` is not P, but `P`= P so let us shift the `pat` to the right to align it with the P in the `txt`:

```
pat:          EXAMPLE
txt: HERE-IS-A-SIMPLE-EXAMPLE
```

Boyer-Moore algorithm

```
pat:          EXAMPLE
txt: HERE-IS-A-SIMPLE-EXAMPLE
```

- We have discovered that `MPLE` occurs in the `txt`, let us put it in front of the `pat` like this:

```
pat:      MPLEEXAMPLE
txt: HERE-IS-A-SIMPLE-EXAMPLE
```

Boyer-Moore algorithm

```
pat:      MPLEEXAMPLE
txt: HERE-IS-A-SIMPLE-EXAMPLE
```

- Now we can shift the pattern all way down to align this discovered occurrence in the `txt` with its last occurrence in the pattern (which is partly imaginary), i.e.:

```
pat:             MPLEEXAMPLE
txt: HERE-IS-A-SIMPLE-EXAMPLE
```

Boyer-Moore algorithm

- There are only seven terminal substrings of the pattern, so we can pre-compute all these shifts too and store them in a table. This is sometimes called the *good suffix shift* table.
- In general, if the algorithm has a choice of more than one shifts, then it takes the largest one.

```
pat:             MPLEEXAMPLE

txt: HERE-IS-A-SIMPLE-EXAMPLE
```

Boyer-Moore algorithm

```
pat:             MPLEEXAMPLE

txt: HERE-IS-A-SIMPLE-EXAMPLE
```

- We've aligned the MPLE but focus on the end of the pattern. `E` is not `P`, `L` is not `P`, but `P=P` so let us shift the `pat` to the right to align it with the `P` in the `txt`:

```
pat:                  EXAMPLE

txt: HERE-IS-A-SIMPLE-EXAMPLE
```

Match !

Boyer-Moore algorithm: properties

- Observe that we have found the pattern without looking at all of the characters.
- Its speed derives from the fact that it can determine all occurrences of `pat` within `txt` without examining too many characters in `txt`.
- In fact, its average performance is $O(n / m)$, that is, it gets faster as the pattern gets longer.
- We say the algorithm is "sublinear" in the sense that it generally looks at fewer characters than it passes.

Rabin-Karp algorithm: hashing

- uses the naïve search method (i.e. sliding window) and substantially speeds up the testing of equality of the pattern to the substrings in the text by using *hashing*.
- It is used for *multiple pattern matching* (in addition to single pattern matching), because it has the unique advantage of being able to find any one of k strings in $O(n)$ time on average, regardless of the magnitude of k.
- The key to performance is the efficient computation of hash values of the successive substrings of the text.

Rabin-Karp algorithm – *hashing*

- A hash function converts every string into a numerical value, called its *hash value (code, sum),* using for instance the ASCII value of characters.
  - For example, `hash('hello') = 5`.
- Algorithm exploits the fact that if two strings are equal, their hash values are also equal (there might be so-called hash collisions, though, that must be checked for letter by letter).
- All we have to do is to compute the hash value of the pattern we're searching for, and then look for substrings with the same hash value within the text (and then check letter by letter).
- Different variants of the algorithm compute hash values in different ways (adding, multiplying, etc.).

Rabin-Karp algorithm: properties

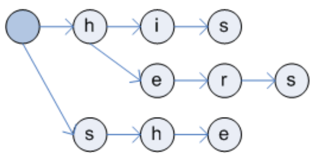
- One popular and effective hash function treats every substring as a number in some base, the base being usually a large prime.
  - For example, if the substring is "hi" and the base b = 101, then hash('hi') = 'h'*b^1 + 'i'*b^0 = 104*101+105*1 = 10,609
- Rabin-Karp is inferior for single pattern searching to Boyer-Moore algorithm because of its slow worst case behaviour.
- However, Rabin-Karp is an algorithm of choice for *multiple pattern search*.
  - That is, if we want to find many fixed length patterns in a text, say of length k, we can create a simple variant of Rabin-Karp that checks whether the hash of a given string in the text belongs to a set of hash values of patterns we are looking for.

Aho-Corasick algorithm

- Used for multiple pattern matching tasks
- Decription from the article and code by Tomas Petricek at http://www.codeproject.com/KB/recipes/ahocorasick.aspx
- The algorithm consists of two parts:
- The first part is the building of the tree from keywords/patterns you want to search for, and the second part is searching the text for the keywords using the previously built tree (finite state machine, FSM).
  - FSM is a deterministic model of behaviour composed of a finite number of states and transitions between those states

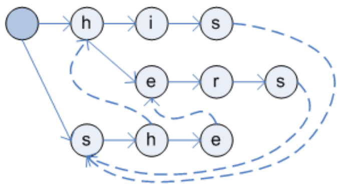
Aho-Corasick algorithm

- In the first phase of the tree building, keywords are added to the tree. (The root node is used only as a place holder and contains links to other letters.)
- Links created in this first step represents the *goto function*, which returns the next state when a character is matching.
  - Example of the tree for keywords: **`his, hers, she`**



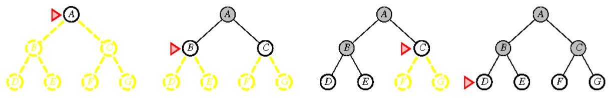
Aho-Corasick algorithm

- The fail function is used when a character is not matching.
- For example, in the text **`shis`**, the failure function is used to exit from the **`she`** branch to **`his`** branch after the first two characters (because the third character is not matching).



Aho-Corasick algorithm

- During the second phase, the BFS (breadth first search) algorithm is used for traversing through all the nodes.
  - At each stage, the node to be expanded is indicated by a marker
  - In general all the nodes are expanded at a given depths before any nodes at the next level are expanded



Help: Find the tutorial on efficient string search with suffix trees written by Mark Nelson at http://marknelson.us/1996/08/01/suffix-trees/

Aho-Corasick algorithm

- Assume that generalised suffix tree has been built for the set of patterns $D = \{S_1, S_2,..., S_K\}$ of total length $n = | n_1 | + | n_2 | + ... + | n_K |$. All patterns have the same alphabet. You can search for patterns in such a way that:
  - Check if a pattern P of length m is a substring in $O(m)$ time.
  - Find the first occurrence of the patterns $P_1,...,P_q$ of total length m as substrings in $O(m)$ time.
  - Find all z occurrences of the patterns $P_1,...,P_q$ of total length m as substrings in $O(m + z)$ time.

Conclusions

- Although data are memorized in various ways, text remains the main form to exchange information.
- String-matching is a very important subject in the wider domain of text processing (i.e. keyword search), not just bioinformatics.
- In bioinformatics, the patterns in strands of DNA, RNA and proteins, have important biological meaning, e.g. they are promoters, enhancers, operators, genes, introns, exons, etc.
- Often these meaningful patterns undergo *mutations* at some points, therefore we include in the patterns the so-called *wildcards*, to replace some of the characters (as in the assignment).