Bachelor Thesis

Optimized pattern matching in genomic data

Report

Martin Westh Petersen - mqt967
Kasper Myrtue - vkl275

20. April 2015

Indholdsfortegnelse

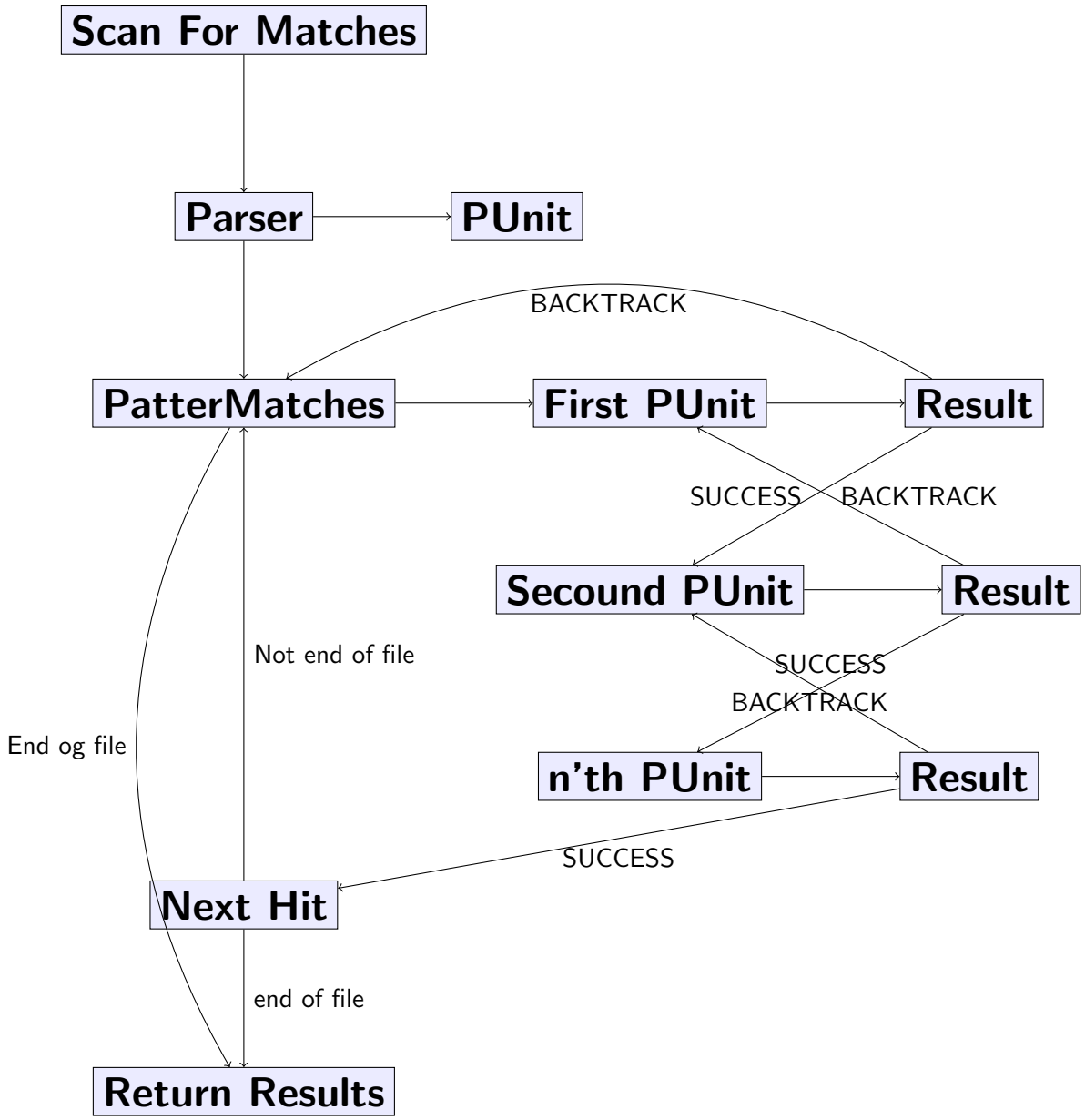
1 Analysis

2 Loose_match

All possibilities are tried out using mismatches, insertions, deletions in that order to accept chars that don't match. When all mismatches, insertions and deletions are used, and we still encounter a mismatch, the stack is popped, to try a different order of the mismatches, insertions and deletions tried.

2.1 Scan For Matches

In order to understand what is good from Scan for matches we needed to understand some code segments and the overall structure of the program, simply put we came to understand this flow in the code:



This shows how Scan_For_Match uses a backtrack algorithm to search trough the data. When ever a PUnit fails it calls BACKTRACK, a case in the function patternMatch in Scan_For_Match.

We have found certain structures he uses to solve different functionality problems, in order to harvest these structures to achieve the same fast runtime in the base cases of the program (e.g. exact match for bases or the like) we need to understand them:

2.2 Ambiguous bases

In Scan_For_Match in order to compare strings of bases he needs to be able to compare with bases that can mean different other bases, (e.g. n is way to write (A or C or T or G), and m is a way of writing (A or C)) this is done by converting all bases to words of 4 bit. This

makes it possible to use bit wise and to compare bases.
in order to do this we need to define the different bases as 1 bit being true (e.g. A = 0001, C = 0010 ....) this makes it possible for ambiguous bases to say which bases it consists of. In Scan_For_Match he has made a conversion table for this:

```
int build_conversion_tables()
{
    int the_char;

    for (the_char=0; the_char < 256; the_char++) {
        switch(tolower(the_char)) {
          case 'a': punit_to_code[the_char] = A_BIT; break;
          case 'c': punit_to_code[the_char] = C_BIT; break;
          case 'g': punit_to_code[the_char] = G_BIT; break;
          case 't': punit_to_code[the_char] = T_BIT; break;
          case 'u': punit_to_code[the_char] = T_BIT; break;
          case 'm': punit_to_code[the_char] = (A_BIT | C_BIT); break;
          case 'r': punit_to_code[the_char] = (A_BIT | G_BIT); break;
          case 'w': punit_to_code[the_char] = (A_BIT | T_BIT); break;
          case 's': punit_to_code[the_char] = (C_BIT | G_BIT); break;
          case 'y': punit_to_code[the_char] = (C_BIT | T_BIT); break;
          case 'k': punit_to_code[the_char] = (G_BIT | T_BIT); break;
          case 'b': punit_to_code[the_char] = (C_BIT | G_BIT | T_BIT); break;
          case 'd': punit_to_code[the_char] = (A_BIT | G_BIT | T_BIT); break;
          case 'h': punit_to_code[the_char] = (A_BIT | C_BIT | T_BIT); break;
          case 'v': punit_to_code[the_char] = (A_BIT | C_BIT | G_BIT); break;
          case 'n': punit_to_code[the_char] = (A_BIT | C_BIT | G_BIT | T_BIT); break;
          default:
            punit_to_code[the_char] = 0;
            break;
        }
        if (punit_to_code[the_char] & A_BIT)
            punit_to_code[the_char] |= T_BIT << 4;
        if (punit_to_code[the_char] & C_BIT)
            punit_to_code[the_char] |= G_BIT << 4;
        if (punit_to_code[the_char] & G_BIT)
            punit_to_code[the_char] |= C_BIT << 4;
        if (punit_to_code[the_char] & T_BIT)
            punit_to_code[the_char] |= A_BIT << 4;
    }
...
```

This makes him able to use bit wise and to determine a match in the following way:

```
#define KnownChar(C)  (punit_sequence_type == PEPTIDE ? 1 : known_char[(C)])

#define Matches(C1,C2) (punit_sequence_type == PEPTIDE ? \
                        ((C1 == C2) || (C2 == 'X')): \
                        (KnownChar((C1) & 15) && ((((C1) & 15) & ((C2) & 15)) == (C1 & 15)↩
                            )))
```

In the code segment above C1 is a character from the data and C2 is a character from the pattern given. In the case where it is not PEPTIDE (PEPTIDE being not DNA) he starts by checking that the data character is a allowed base, if this is true and the two characters being bit wise "and" compared results in the same word taken from the data, it returns true.

When he grabs the two characters to compare, he doesn't just get 4bits in order to make sure it is only the 4bits he is interested in, that gets compared he bit wise "and" them with 15. This makes all other than the last 4 bits go to 0 because 15 is 4 ones in end of the Bite.

The smart result of this comparing algorithm is that you only need to compare once even though the pattern can have multiple different ambiguous bases. It will always return the base from the data if the ambiguous base contains the base from the data. example:

- The base from the data is: A (0001)
- The ambiguous base from the pattern is: D (A or T or G) 1101
- The bit wise "and" operation: 0001 & 1101 = 0001
- The result is then compared with A again to see if the compare was successful

In order to use this he has converted both the data and the pattern with the conversion_table.

3 Design

3.1 Optimization

3.1.1 Skip length

At first the pattern is matched against the database sequence as usual with different orders of mismatches, insertions and deletions if necessary. This particular match (with a specific starting char in the DB-sequence SR) continues until the N chars from SR to SR + N in the DB-string are read, and the different letters counted, where N is the length of the pattern. The letters in the pattern are also counted. We now have 2 lists. One contains the different letters and number of occurrences in the pattern (patlist), and one contains the number of occurrences of letters of the DB-string in interval (SR, SR + len(pattern)) (datlist). The length of patter is increased with allowed

Say there is not match in this first run. Normally we would increment the SR counter and try again, but instead we increment SR, and update the datlist with the new letters. If the number of a specific letter in the pattern exceeds the number of allowed mismathces + insertions

3.2 Program structure

4 Program structure

A list of actions our program should do in order to execute a pattern search:

- Read and parse the input line. E.g. "scanFM 'ATTGCCCC[0,1,2]' 'data.txt'". Possibility of writing "$->$ 'output.txt'" which results in the matches not being displayed in the terminal but written the the specified file.

- Parse the pattern into units and save them as different types (objects of different classes that inherit from a common PUNIT class), e.g. EXACT_PUNIT, AMBI_PUNIT etc.
- Choose the order of which to search for the patterns and create a state that readies for this search, for example a list of the punits in correct order with some way of keeping track of the different positions the punits have to with respect to each other.
- Search for the punits in the order chosen by simply calling a ".search()" method for each PUNIT-object. The PUNIT-object's search method invoked is unique for each different type of PUNIT, and returns either True or False. If the search for each PUNIT returns True the match is saved.
- The saved matches are either displayed in the terminal or written in a file, depending on the call of scanFM.

Types of PUNITs that we need

- EXACT - A PUNIT of this type consists of either of the letters 'A', 'C', 'G' and 'T' or any number of the wildcards.
-