Bachelor Thesis

University of Copenhagen, DIKU

Optimized pattern matching in genomic data

Martin Westh Petersen - mqt967
Kasper Myrtue - vkl275
-
Supervisors:
Rasmus Fonseca
Martin Asser Hansen

8. June 2015

Table of contents

1 Introduction

Analysis and research of genomic data such as DNA is beneficial in a variety of fields for example medicine, where scientists are now able to identify the genes responsible for causing genetic diseases like Alzheimer's disease. [1]

DNA consists of two biopolymer strands that coil around each other forming a double helix. The two strands connect along the way, binding pairs of molecules called nucleobases. There are four different nucleobases, called guanine (G), adenine (A), thymine (T), cytosine (C). They bind to each other in complementary pairs, T binds with A and G binds with C. [3]

What holds the genetic information in DNA is the sequence of nucleobases along the strands, and thus sequencing DNA result in a long sequence of these four bases represented as just the letters A,T,G and C.

Pattern matching functionality is unavoidable when analyzing and searching for patterns in these sequences, but huge amounts of data makes it inefficient to manually find these patterns [5], so there's a need for clever and efficient software to do this.

`scan_for_matches` [6] is a piece of software that serves this purpose, but big improvements in performance can be made. On top that, the code is poorly documented, lacks version control and is hard to read and maintain.

Our goal is therefore to re-implement `scan_for_matches` to contain the useful parts, while optimizing the algorithms to improve performance, and have version control and documentation.

2 Methods

2.1 Analysis of `scan_for_matches`

2.1.1 Defining patterns in `scan_for_matches`

`scan_for_matches` provides a simple and limited domain-specific language for defining patterns to look for in a text file consisting of a specific alphabet, e.g. the four letters (A, T, C and G) representing bases in a DNA sequence.

The language revolve around a basic building block called the pattern (PU) which can take different forms and be used in combination to define an overall pattern to search for.

Each PU can either match or not match a sub-sequence of data, but it may be able to match the data in several different ways. To declare an overall match, `scan_for_matches` has to be able to match each PU in the given order with a consecutive sub-sequence in the data. It does so by trying to match the PU's one at a time going left to right with a specific starting position in data. The algorithm uses backtracking which means that whenever a

PU (call it p2) is not able to match, the algorithm goes back to the previous PU (call it p1) to try and find a different match for p1. If successful the algorithm continues to p2 again. The different way of matching p1 may have resulted in a different starting position for p2, which may enable p2 to actually find a match this time. If there was not way of finding an overall match the algorithm increments the starting position for the first PU and tries again. It continues like this until the end of the data file.

`scan_for_matches` offers some features besides the use of just PU's. These features apply to the PU's and alter their criteria for matching. The different formats of PU's and some of the extra features are described in the list below.

- An exact PU consists of a specific sequence of letters from the alphabet which will only match if the compared sub-sequence in data has the exact same letters in the exact same order as the exact PU .

  Example of exact PU : `AGGT`

- A range PU consists of 2 positive integers separated by three dots, where the first integer is less than or equal to the second. The PU matches any combination of letters from the alphabet that has a length that is equal to one of the integers or any integer in between those integers.

  Example of range PU : `4...8` which matches any letter combination of length 4,5,6,7 or 8.

- A variable can be assigned a range PU for later reference. The variable is specified by any user defined name followed by an equality symbol and then followed by a range PU.
  The range PU functions normally, and matches any combination of letters with the allowed range, but at run-time the letter combination that it matches in data is saved, and one can reference it in a later PU by simply writing the name of the variable. That is a reference PU and functions as an exact PU but uses the specific saved letter combination to match with.

  Example using variable/reference PU : `p1=4...6 ATG p1`, where the first PU is the range PU with added variable functionality, and the third being the reference PU that is linked to the first PU.

- Any exact or reference PU can be made flexible by allowing a number of specified single-letter edits; insertions, deletions and mismatches.

  A mismatch allows one letter from the PU to match one letter in the data even though they're not the same. An insertion allows for a temporary insert of one letter in the PU letter combination that matches one letter in the data, thus lengthening the PU. A deletion allows for temporarily ignoring one letter in the PU letter combination and jumping straight the next letter, thus shortening the PU.

Example using single-letter edits : `p1=12...15 ATTCC[1,0,3] p1[2,2,2]` where the exact PU is allowed 1 mismatch, 0 deletions and 3 insertions and the reference PU is allowed 2 of each.

- Putting a $\sim$ in front of a reference PU means that `scan_for_matches` tries to match the reverse complement of the letter-sequence saved in the variable. The reverse complement of a letter sequence is a sequence of same length, where it has been reversed and every letter is substituted with its complementary counterpart (A swaps with T and vice versa, G swaps with C and vice versa).

  Example using the $\sim$ feature : `p1=3...4 GG ~p1` would match the data sub-sequence: "TCACGGGTGA" since "GTGA" is the reverse complement of "TCAC".

- Ambiguous letters [4] are letters other than the standard letters of the used alphabet. These letters are used in exact PU's to allow matching with one of multiple of the standard letters in the alphabet. In `scan_for_matches` the letter "Y" in a PU matches either a "C" or a "T" in data, a "D" matches either "A" or "G" or "T" and "N" matches any of the standard letters. There is a letter for each different combination of the four standard letters.

  Example using ambiguous letters : `CYDTDNA` matches the sub-sequence "CCGTACA" in data.

The above are what we have found to be the core features of `scan_for_matches`, but other features are available as well. Here are some of them:

- A logical "or" between PU's or sets of PU's , e.g. "(`AGGT | CCCC`)", or "(`p1=3...6 TG p1[1,0,4] | p1=3...6 AAA p1[2,2,0]`)" allows either of the two sides of the | to match.

- Custom pairing rules can be defined, with which the user can define custom letter pairings of in the given alphabet, instead of the standard rules (A pairs with T and C with G).

2.1.2 Flow of `scan_for_matches`

`scan_for_matches` uses backtracking functionality [2] on two different levels to find the matches. One is the outer backtracking system that backtracks between PU's , and the other is a backtracking algorithm for matching a single PU if it has been allowed any mismatches, insertions or deletions.

`scan_for_matches` has an outer backtracking system that controls the flow of the outer pattern match, i.e. decides whether to move forward to the next PU if the current PU matched, or backtrack to the previous PU if there was no match. When backtracking to a previous PU the next possibility (if any) for matching is chosen and the algorithm moves forward again. This problem would normally be solved with recursion and/or loops, but

`scan_for_matches` uses GOTO statements which has both pros and cons.

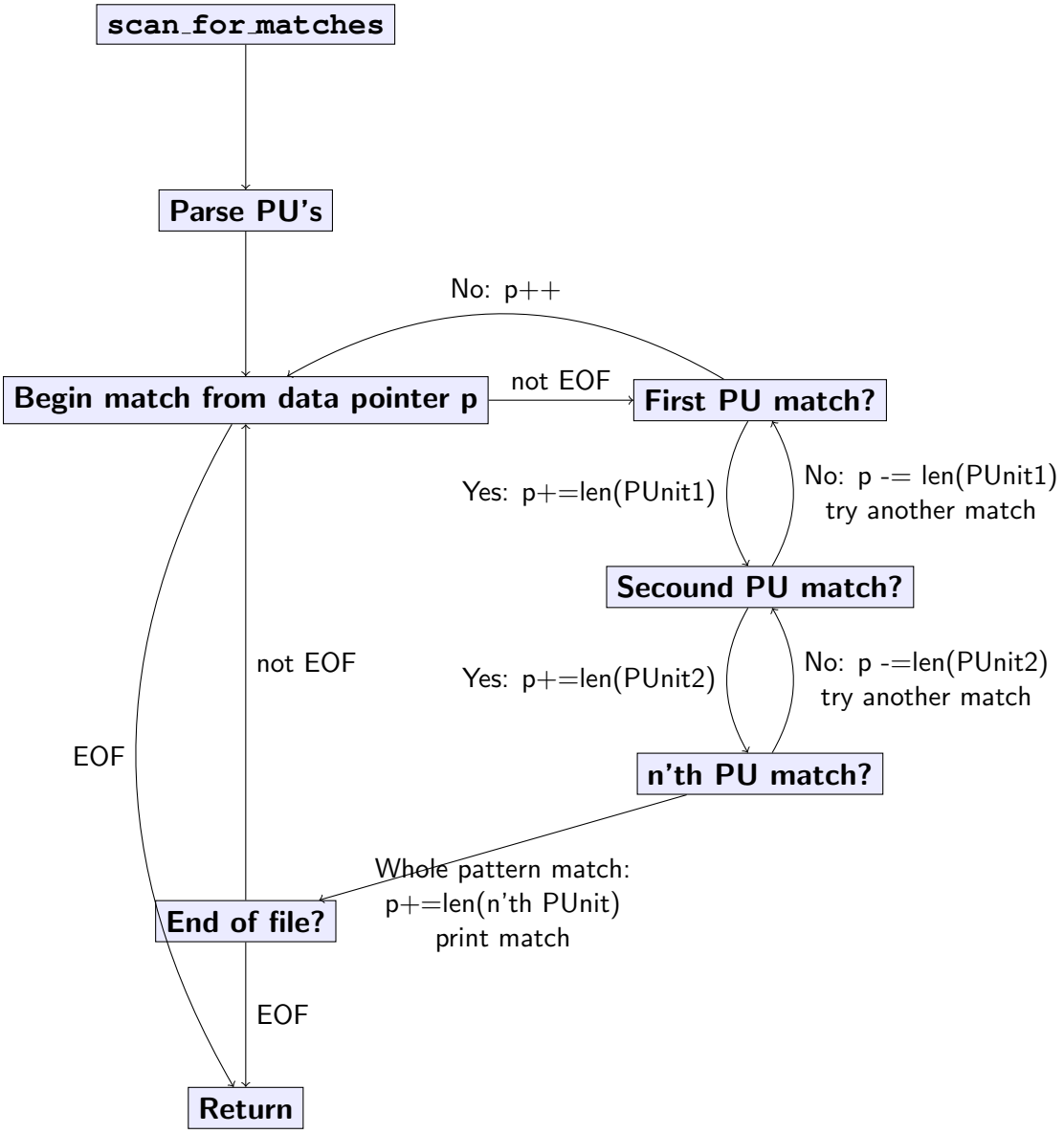

Figure 1: *Simplistic flow chart of pattern matching in* `scan_for_matches`*. p is a pointer to data and is initially set to start of data. Notice that overlapping matches will no be found.*

The use of GOTO statements avoids some of the risks associated with the use of recursion like stack overflow. How deep the recursion would go would depend on the size of the pattern, and it would therefore be hard to control and unsafe to use with respect to unwanted termination. Analyzing the possible depth of recursion and rejecting a queried search if too deep is hardly an option, as it may be necessary to find these large patterns.

Even though the use of GOTO statements uses a controllable fixed amount of memory,

it is not necessarily safe. The big con is the missing modularity and structure. Every variable is defined in a global scope and has to be managed as such. It is very easy to make a mistake when creating or maintaining that sort of code (as `scan_for_matches` is a great example of) because the responsibility is not partitioned into functions or classes that can be unit-tested and proofed to work. One variable being set wrongly due to a particular input and searching in a particular part of the data may escalate to either deliver wrong matches or just terminate.

Figure 1 shows the overall flow of finding patterns by backtracking back and forth between the PU's.

The other backtracking system happens internally in the pattern matching of a single PU if that PU has allowed mismatches, insertions and deletions. Figuring out a way to utilize the allowed edits to possibly find a match, requires trying out a lot of combinations so this part of the program is the most computationally heavy.

2.1.3 Cooperability of multi-level backtracking

`scan_for_matches` uses a very certain type of backtracking to optimize performance. Starting from left going right in the pattern and data, the algorithm greedily matches as many characters as possible without using one of the allowed edits. Every time a character doesn't match, `scan_for_matches` spends an edit and uses a stack structure to keep track on which kind of edit should be tried next at this point in the sub-sequence, in case the rest of the PU doesn't match and need to backtrack to this spot. The different edits are tried in the following order: mismatch, insertion, deletion, i.e. if a character doesn't match the character in data and this spot was approached from the left (not backtracking) then a mismatch is used. When backtracking to this spot then an insertions is tried, and if that failed as well further along in the PU the a deletion is tried.

The naive way of trying to fit a PU to a sub-sequence in data using edits would be trying out every permutation of the edits applied on the characters, even those which actually matched the data. The number of permutations would be very large, so `scan_for_matches` uses dominance relation [**dom**] to rule out many of the permutations and speed up the search. Dominance relation is a property that is used in many combinatorial problems, and allows for some of the solution space of a problem to be dismissed as an (optimal) solution due to a "dominant" other part of the solution space. A good example is the classic knapsack problem; Given a set of items that each has a weight and value assigned to them, and then a bag that can contain items up to a certain weight limit, determine how many of the different items to include in order to maximize the total value but keeping the weight under the limit. If one of the items I_1 weighs X and is worth Y, but a combination of other items (say I_2 and I_3) weight less than X and is worth more than Y, then item T can never be part of an optimal solution to this problem, since in any solution that contains T, one could substitute T for I_2 and I_3 and get more value without breaking the weight limit, in which case the original solution containing T was not optimal.

In this case dominance relation allows for greedily ruling out possible combinations of uses of mismatches, insertions and deletions that applies any of the edits on characters that could match without an edit (still looking at one character at a time from left to right). In other words, whenever `scan_for_matches` encounters a character that matched the character in data, spending a mismatch, insertion or deletion is never considered and never tried out, because the property of dominance relation promises that spending an edit here is not necessary for finding a match.

Figure 2 shows an example of a match between pattern and data that uses edits where it was not required, and how one can transform this into a match that uses its first edit at the first mismatching character.

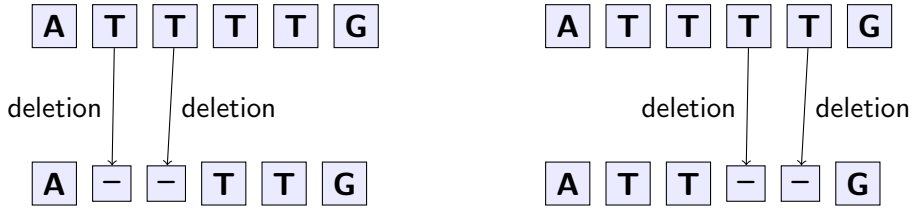

Figure 2: *The leftmost match uses its two deletions where it was not necessary, and the rightmost example shows the same match where the use of the deletions have been postponed as much as possible before being used.*

Using dominance relation `scan_for_matches` always finds a match for a PU if it exists, but when used in combination with an outer backtracking system that is not enough!
The technique described is useful only if the goal is determining if there exist a possible match or not. With an outer backtracking system it is often times required that several if not all possibilities are tried in order to find the overall match. Some PU's of the overall pattern may be required to be of a specific length in order to match the whole pattern, and the only way such a PU can obtain the required length may very well be using unnecessary edits. Trying to match a PU with allowed edits, `scan_for_matches` will always accept the first way of matching, and even if subsequent PU's fail and we backtrack to this PU again, it will simply fail.
It is therefore not guaranteed that `scan_for_matches` will find all the matches it's supposed to find. In fact every sub-sequence of data that should match with the pattern, but requires for one or more of its PU's to have a length only acquirable by using unnecessary edits, `scan_for_matches` will not find!

2.2 Backtracking

Backtracking is a commonly used technique in string searching functionality and is the foundational algorithm of the string search in `scan_for_matches` and in our re-implementation. The problem of determining a match between two strings are closely related to the Levenshtein distance problem although it has important differences.

The Levenshtein distance between two strings is defined as *the minimum amount of single-character edits required to change on string into the other.* **leve**

Figure 2 shows the available single-character edits.

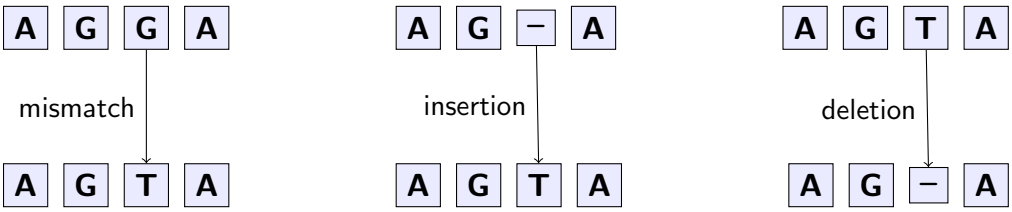

Figure 3: *The top character sequences are the patterns and the bottom the data. In the first example the G is simply substituted for a T. In the second example the pattern is AGA but matches the sub-sequence AGTA in data by insertion of a T. In the third example the pattern AGTA matches the data AGA by deleting the T*

Calculating the Levenshtein distance does not include requirements for the distribution of the different single-character edits, it just finds the minimum *sum* of them.
If the task at hand is to determine whether two strings are within a maximum allowed Levenshtein distance of each other, then simply calculating the Levenshtein distance would be enough.
If however the task is to determine whether one string can be changed into another string with a maximum amount of X mismatches, Y insertions and Z deletions one can *not* simply calculate the Levenshtein distance of the two strings and compare the number to the sum of the different edits ($X + Y + Z$) since the Levenshtein distance may not live up to the additional requirements of a maximum of either of the different edits. An example of this is shown in Figure 3.

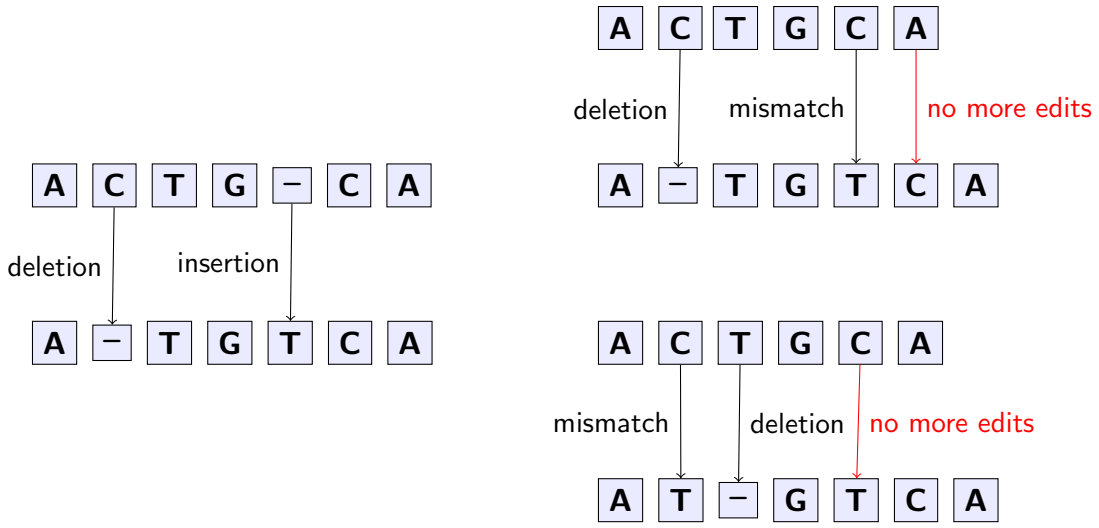

Figure 4: *The Levenshtein distance between the pattern ACTGCA and the data ATGTCA is 2 (1 deletion, 1 insertion) as the leftmost example shows. If the pattern is allowed 1 deletion and 1 mismatch, then the two strings can not match, as the rightmost examples show, even though this also would be a sum of 2 edits.*

2.2.1 Complexity of backtracking

Let's look at the implications backtracking has on performance of an implementation. An implementation similar to `scan_for_matches` that allows complex patterns to be defined by separable PU's would often require more than just finding *one* match for a particular PU with allowed single-character edits. One possible combination of the single-character edits used to find a match for one particular PU may result in a valid match of the whole pattern, where other combinations for that particular PU may prevent a whole match from being possible.
It is therefore necessary for our particular PU to be able to find more than one way to make a valid match (if these are possible). Consider the example in Figure 4 where the pattern is

`CT AGCA[2,0,1] TT`, i.e. 2 mismatches and one deletion allowed for the second `PUnit`, and the data is

`CTAGAGGT`.

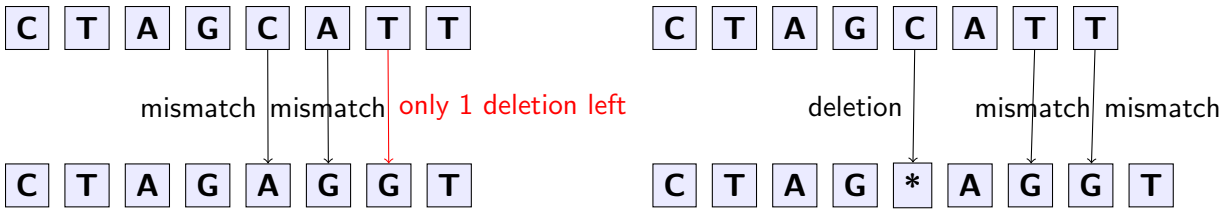

Figure 5: *In the leftmost example the* PU `AGCA` *chose to use 2 mismatches to match the data sub-sequence* `AGAG` *and then move on the next* PU. *The next* PU *would then have to match it's letters* `TT` *with the data sub-sequence* `GT` *which it can't do with its remaining deletion. The overall algorithm would have to backtrack the the second* PU *to try another possible match (if any) to see if that would make a difference. The rightmost example shows how using a deletion to delete the C in* `AGCA` *would also match data* `AGA` *and when moving forward to the next* PU `TT` *will actually match* `GG` *by using the two remaining mismatches.*

For two different reasons can a PU have multiple ways of matching with the data. Either it's a PU with allowed mismatches, insertions and deletions like discussed, or it is a range PU which allows matches of all the lengths included in the interval given (`7...9` would allow 3 possible matches).

Whenever backtracking to one of these, another of the possibilities will be chosen and the algorithm continues to the subsequent PU again. The amount of checks that the algorithm would need to perform in order to try out all possibilities follow normal combinatorics, and would therefore grow exponentially. One could visualize this with a search tree structure, where each node is a possibility for a single PU. The nodes would have varying number of branches as not all PU's have the same number of ways to reach a match. The height of the tree may vary as well, because which branch is chosen may affect what possibilities the subsequent PU has, and the final PU may not be reached with every combination.
Consider the example in Figure 5 where the pattern is

`AGC[1,0,1] TCT[0,0,1] TG <more PUnits>` and the data

`ACCTCTTG<more data>`

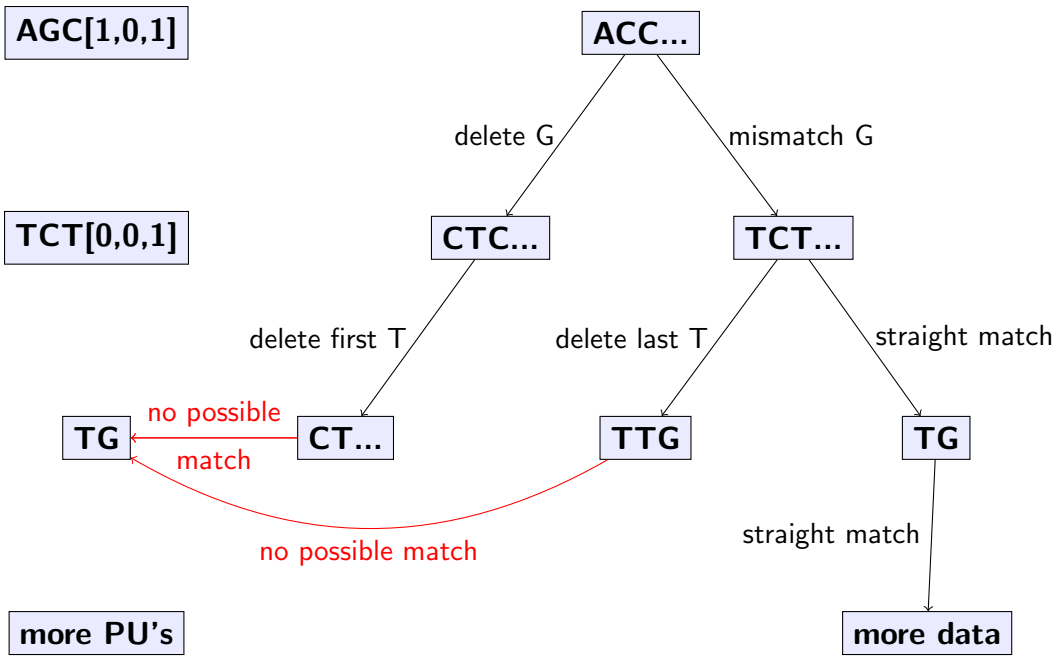

Figure 6: *Each node shows the relevant data that needs to match the* `PUnit` *on the left side of the tree, and each edge between nodes shows the combination chosen to match that data. The first* PU *has two ways of matching it's data. Say the G is deleted to match with data* `AC`*. The second* PU *has only one way of matching with data* `CTC` *that is by deleting the first T. The third* PU *can not match the data* `CT`*. If mismatching with the G for the first* PU *was chosen instead, the second* PU *would have two possible ways of matching with the data* `TCT`*. If the second T is deleted, the third* PU *would again have no way of matching with data* `TT` *but if no edits are used and the straight match is chosen, the third* PU *would be able to do a straight match, and the tree would continue to branch downwards to the subsequent* PU's.

So even though the number of combinations that should possibly be tried outgrows exponentially, it can often times be determined before reaching the bottom of the tree whether that particular way down is possible or not and if not, the combination does not have to be checked all the way to the end.

The worst case running time would still be exponential since it is possible for a pattern that no possible choice of match for any PU would prevent any of the following choices, i.e. the tree would have a homogeneous height, and all the nodes would have their respective maximum amount of child nodes.

Let us define the function $children(input)$ to return the amount of possible matches with *different lengths* on any data, that the input PU has. One should only distinguish between possible matches of different lengths since matches of same length would yield the same results in their child branches, so only one set of these child branches should be considered. In case of mismatches insertions and deletions, the number of characters a match can differ is determined by the sum of the insertions (use of these makes the match shorter) and the

deletions (use of these makes the match longer). The last option is using none of the insertions or deletions which result in yet another match length.

If it's a range PU then the number of different lengths are given by the size of the interval.

$$children(input) = \begin{cases} (input.max - input.min) + 1 & \text{if } input = Range \\ input.insertions + input.deletions + 1 & otherwise \end{cases}$$

Let *punits* be an array of PU's in a given pattern p indexed in the same order as p, n be the number of PU's in *punits*, and assume worst case where each node has its maximum number of edges:

$$real\_number\_of\_edges(punits[i]) = children(punits[i]) \quad \text{for all } i = \{1...n\}$$

Then the amount of combinations of the different matches of the PU's would be $comb(p)$

$$comb(p) = \prod_{n=1}^{n} children(punits[i])$$

Consider the pattern `AGGT[0,1,1] 4...5 TTCTAA[0,2,1]`.

$children(AGGT[0,1,1]) \cdot children(4...5) \cdot children(TTCTAA[0,2,1]) = 3 \cdot 2 \cdot 4 = 24$

A total of 24 leaves on the tree.
As mentioned this is the maximum amount of checks that would need to be performed. Even if a tree has a homogeneous height, the probability of checking the correct branch lastly is low. The average actual running time of an algorithm that uses these principles would therefore be significantly better.

2.3 Design

Our re-implementation of `scan_for_matches` called `scanfm` is developed in C++ rather than C, which was used to develop `scan_for_matches`. We have used C++ classes instead of C structs as the base structure for the PU's.
Figure 6 shows the class structure of `scanfm`.

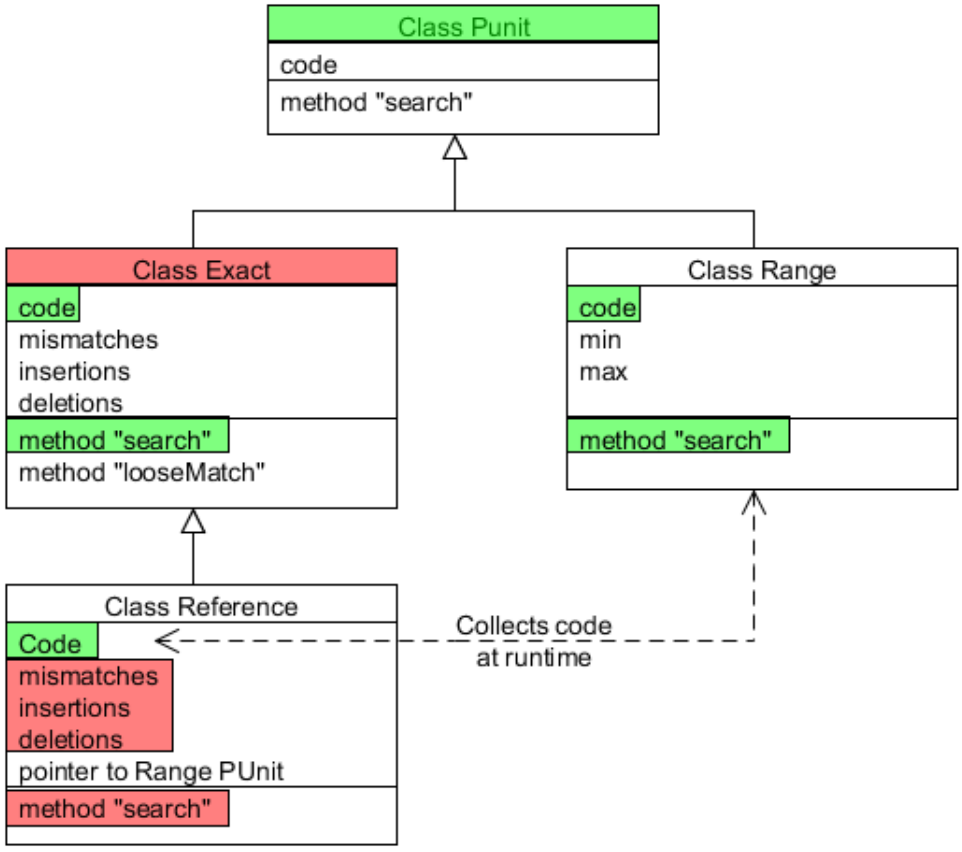

Figure 7: *There is one generic* PU *class called "Punit". This class holds a lot of the common variables and flags that are needed no matter what kind of* PU. *The "Range" class inherits from the "Punit" class and holds its own information, for instance the interval of the range. The "Exact" class also inherits from the "Punit", and lastly the "Reference" class inherits from the "Exact" class. Each class (except the "Punit" class itself) defines the method "search", which searches for a match given a start position and some extra information. A "Reference" object collects the letters it's supposed to search for at runtime from the referenced "Range"*

The control flow of `scanfm` is similar in idea to `scan_for_matches` but very different in implementation. While `scan_for_matches` uses a case/switch and GOTO statements, `scanfm` uses an outer loop to call the "search" function of the PU's and then either forward to the subsequent PU on success or backtrack to the previous PU.
Passing parameters between the calls and returns of the "search" methods sets the criteria for the particular search, for instance the search for the first PU where the "search" function is told to search and increment data until a match is
found.
Figure 7 shows a simplistic model of the flow in `scanfm`.

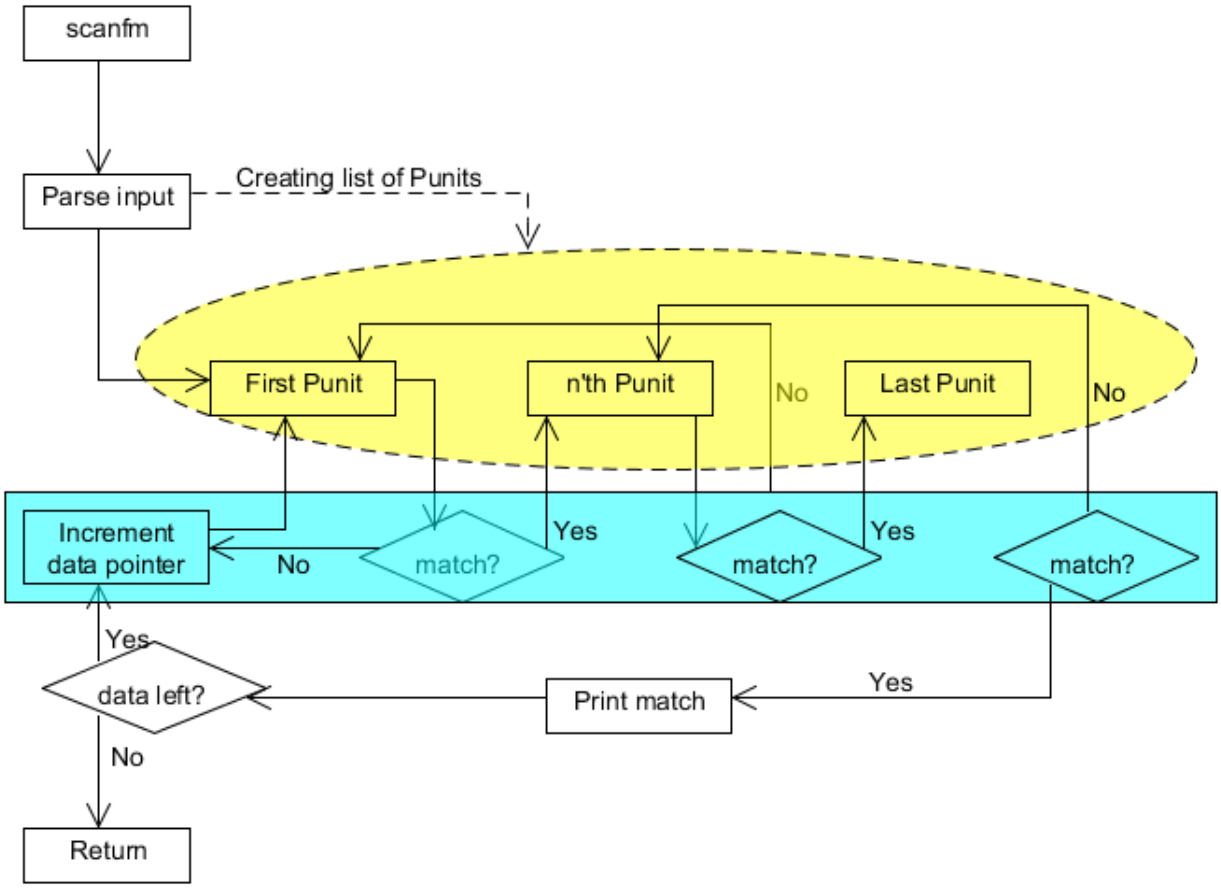

Figure 8: *The yellow ellipsis indicates the list of Punit object. The cyan rectangle marks where most of the "search" method functionality is happening.*

2.3.1 Optimizing character comparison using bitwise operations

One of the great tricks used in `scan_for_matches` is the use of bitwise operations in performance critical parts of the code. Comparing single characters is the most used operations and maximizing the speed of this particular operation means everything for the performance of the program. Figure 8 shows how `scan_for_matches` transforms every normal C character into a custom 4-bit code. These 4 bits are used to represent each of the four letters (A,T,C and G) by having a 1 in one of the slots and 0 in the other three. The letter G for example is represented as 0100. All possible permutations of combining these letters are also represented by a permutation of the bit field, e.g. the letter V is written in a PU represents either A,C or G and therefore has the bit field 0111 since A is 0001 and C is 0010.
Prior to searching, the data input is of course also transformed to 4-bit representations. When comparing two characters `scan_for_matches` uses the bitwise AND operator and if the result is not 0000 the characters matches.

scan_for_matches also put the 4 bits left of our bit field to use. The complementary letter of the letter represented in the rightmost bit field is stored in the leftmost bit field, e.g. G would look like 0010 0100 although only on of the 4-bit fields would be used in a comparison. This leftmost bit field is used when comparing a complementary reference PU instead of transforming back and forth between the letters every time a complementary PU is encountered.

We are using this exact technique in scanfm to achieve the same optimal of comparing characters.

```
int build_conversion_tables()
{
    int the_char;

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

Figure 9: *The function that prepares the conversion table that is used for converting both data and* PU's. *The last 8 or so lines of code is where the left 4 bits is being set to the complementary of the letter represented in the rightmost 4 bits.*

2.3.2 Optimizing order of PU matching

A possible optimization that scan_for_matches does *not* exploit, is choosing an order of PU's to be searched for. Some PU's are more exclusive and rare to find in data than others.

Specifically an exact PU with the most letters is the PU that statistically would be found the least times in random data. If starting the search by identifying this rarest PU the search would save a lot of instructions. `scan_for_matches` starts every search with the first PU in the pattern, and it is therefore often guaranteed to run slower than if it used the other approach.

Figure 10: *A pattern that consists of two* PU's *where the second (say* `CTCGAATAG[1,0,0]` *) is more rare in the data than the first* PU *(say(*`GGT`*). Searching for the second* PU *first needs three checks (with three additional check afterwards to verify the whole pattern) while choosing the look for the first* PU *needs 6 checks (with 6 following checks to verify whole pattern).*

As Figure 9 shows, this optimization presents a potential huge increase in running time of the program.

3 Results

References

[1] Adnan A. "DNA Sequencing: Method, Benefits and Applications". In: (2010). Accessed online 03-May 2015. URL: http://www.biotecharticles.com/Genetics-Article/DNA-Sequencing-Method-Benefits-and-Applications-248.html.

[2] Matuszek D. *Backtracking*. Accessed online 03-May 2015. 2002. URL: http://www.cis.upenn.edu/~matuszek/cit594-2012/Pages/backtracking.html.

[3] Nelson D. L. and Cox M.M. *Principles of Biochemistry*. 6th ed. Macmillan Higher Education, 2013, pp. 281–297. ISBN: 978-1-4641-0962-1.

[4] Borodowsky M. and McIninch J. "Recognition of genes in DNA sequence with ambiguities". In: (1993).

[5] McCloskey P. *Genome research creating data that's too big for IT*. Accessed online 03-May 2015. 2013. URL: http://gcn.com/Articles/2013/04/18/Genome-research-creating-data-too-big-for-IT.aspx.

[6] *Scan For Matches*. Accessed online 03-May 2015. 2010. URL: http://blog.theseed.org/servers/2010/07/scan-for-matches.html.