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. [**gen**]

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. [**dna**]

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 [**gcn**], so there's a need for clever and efficient software to do this.

`scan_for_matches` [**scm**] 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 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 either a range PU or an exact 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 or an exact PU.

  The range or exact PU functions normally, 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.
  An example of each edit is shown in Figure 1.

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

- Putting a ∼ 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 ∼ feature : `p1=3...4 GG ~p1` would match the data subsequence: ”TCACGGGTGA” since ”GTGA” is the reverse complement of ”TCAC”.

- Ambiguous letters [**ambi**] 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 list above explains 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).

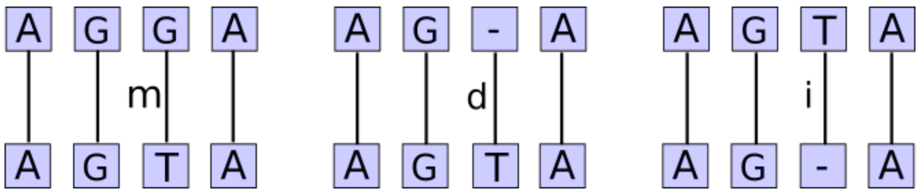

Figure 1: *The top character sequences are the patterns and the bottom the data. In the first example the G is simply substituted for a T. In the second example the pattern is AGA but matches the sub-sequence AGTA in data by insertion of a T. In the third example the pattern AGTA matches the data AGA by deleting the T.*

## 2.2 Backtracking

Finding a match between data and a PU with allowed edits often requires trying different combinations of uses of the edits, and *backtracking* is a sort of algorithm that offers this functionality. Backtracking is the foundation for string search in `scan_for_matches` and in our re-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 that makes a particular PU match, 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 2 where the pattern is

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

"CTAGACGT"

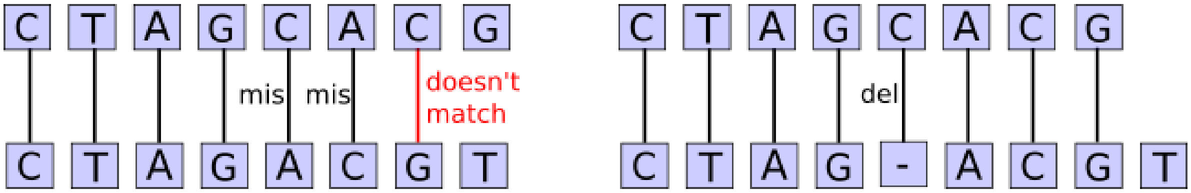

Figure 2: *In the leftmost example the* PU `AGCA` *chooses to use 2 mismatches to match the data sub-sequence* "AGAC" *and then move on the next* PU. *The next* PU *would then have to match it's letters* `CG` *with the data sub-sequence* "GT" *which it can't do without edits. The overall algorithm would 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 match data* "AGA" *and when moving forward to the next* PU `CG` *will match* `CG`

There are two different ways for a PU to have multiple different matches with a sub-sequence in data. Either it's an exact or reference PU with allowed mismatches, insertions and deletions, or it is a range PU. 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 grows in an exponential fashion with the number of PU's and the number of different matches each PU has. One could visualize this as a search tree structure, where each horizontal level of the tree represents a single PU. The different nodes in each horizontal level then each represent a different way of matching the overall pattern with data until reaching this PU. The edges from parent node to children nodes would then represent the different possibilities of match-

ing the given PU with the data. The nodes would have varying number of branches since PU's have a varying number of ways to match with a give data sub-sequence. The height of the tree would also vary 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 3 where the pattern is

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

"ACCTCTTG<more data>"

Using a deletion to match `AGC` with "ACC" turns out to be the "wrong" way to go down the tree. This path only goes down 2 levels in the tree before it becomes impossible to go further. Choosing a mismatch to match `AGC` with "ACC" and then matching `TG` with "TG" without using edits lets us further down the tree (and possibly to the last PU).

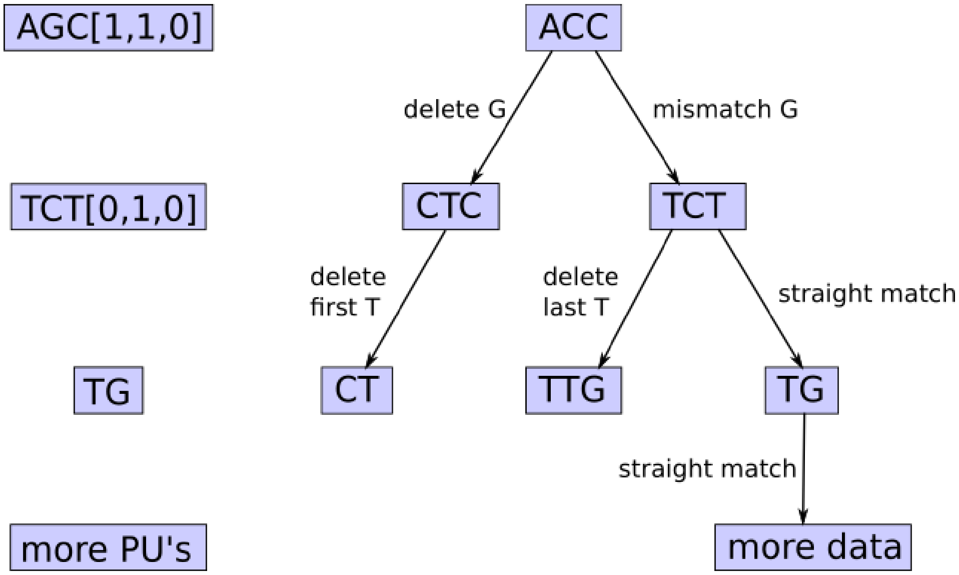

Figure 3: *The rectangles on the left show the PU's and each horizontal level of the tree represents the PU which is horizontally aligned with it. The letter combination inside each node in the tree shows the part of the data sequence that the given PU tries to match with. The text attached to each edge says what choice of use of edits was made in order to match.*

It may be that not every path down the tree reaches the last PU so even though the number of combinations that should possibly be tried grows 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.

Many of the sub-paths in a tree like this may yield the same result on the sub-path's end node, i.e. at a horizontal level in the tree there may be several identical nodes, and the children of these identical nodes will of course be identical as well, so when counting towards the number of *different* overall matches, only one of these identical branches should be considered.

Let us define the function $children(PU)$ to take the a PU as input and return the amount of possible matches with *different lengths* on any data, that the input PU can have. 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 longer) and the deletions (use of these makes the match shorter). 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.

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

Let $p$ be a pattern and $p[i]$ be the i'th PU in $p$, and let $n$ be the number of PU's in $p$, then the amount of combinations of the different matches of the PU's would be $comb(p)$

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

Consider the pattern `AGGT[0,1,1]  4...5  TTCTAA[0,2,1]` called $p1$:

$$comb(p1, n) = \prod_{i=1}^{3} children(p1[i]) =$$

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

A total of 24 leaves on the tree.
As mentioned this is the maximum amount of checks that would need to be performed in order to determine if there is a possible match at a specific starting position in the data. 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, but worst case is:

$comb(p, n)$ given a pattern $p$ and the the number of PU's $n$ in that $p$.

## 2.3 Flow of **scan_for_matches**

`scan_for_matches` uses backtracking functionality [**back**] 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.

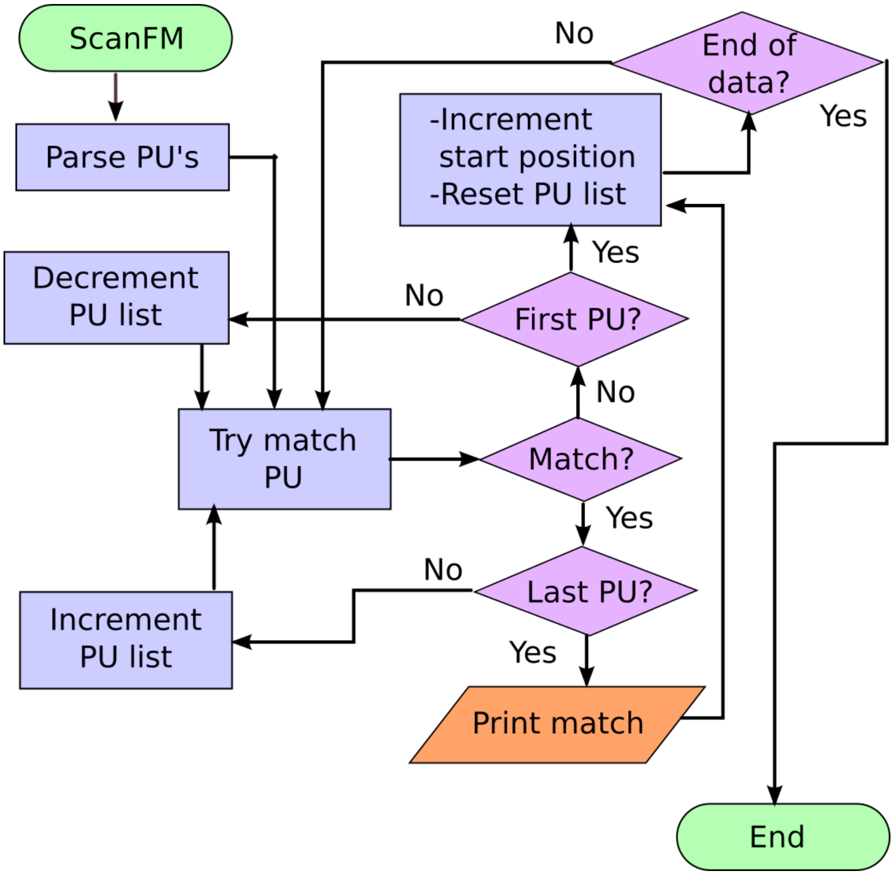

Figure 4: *Simplistic flow chart of* `scan_for_matches`. *The blue rectangles indicate a process being executed, the purple diamonds indicate decisions being made, the green rounded rectangles indicate start and end of the program and the orange parallelogram indicates output.*

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. 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 proved to work. Every variable is defined in a global scope and has to be managed as such, which is very prone to error. With every jump back and forth between the GOTO's all variables and flags must be set just right or the behavior of the program is undefined. 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 4 shows the overall flow of finding patterns in `scan_for_matches` by backtracking back and forth between the PU's. When a match is found it is printed to the user, the list of PU's is reset to start with the first PU and the starting position in data is incremented accordingly so the search for other matches can continue.

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.

### 2.3.1 Multi-level backtracking

Exact or reference PU's that has allowed mismatches, insertions or deletions are handled in a very specific way by `scan_for_matches`.Starting from left going right in the PU and data, the algorithm greedily matches as many characters as possible without using allowed edits. Every time a character doesn't match, `scan_for_matches` spends an edit and uses a stack structure to keep track of 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, deletion, insertion, 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 a deletion is tried.

By only using edits when necessary, `scan_for_matches` exploits the property of *dominance relation.* [**dom**] Dominance relation can be applied in many combinatorial problems, and allows for some of the solution space of a problem to be dismissed as it can never be an optimal solution due to a "dominant" other part of the solution space. A good example is the classic knapsack problem [**knap**]; 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

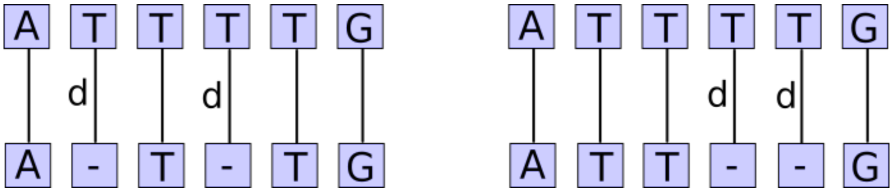

Figure 5: *The leftmost example uses its two deletions where it was not necessary, and the rightmost example shows the same match where the use of the deletions have been postponed as much as possible before being used.*

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 $I_3$) weighs less than X and is worth more than Y, then item $I_1$ can never be part of an optimal solution to this problem, since in any solution that contains $I_1$, one could substitute $I_1$ for $I_2$ and $I_3$ and get more value without breaking the weight limit, in which case the original solution containing $I_1$ was not optimal.

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 to rule out many of the permutations and speed up the search.
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 5 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 characters. The pattern is `ATTTTG[0,2,0]` and the data is "ATTG". 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, the same match with the same length will be found.

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!

Figure 6 shows an example of an overall pattern that will only match the data if unnecessary edits are used in a PU. The pattern is: `AAA[0,2,0] AT` and the data is "AAAT". Only by using an unnecessary deletion in the first PU will the second PU match.

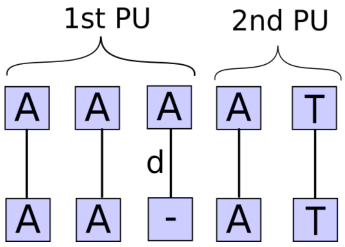

Figure 6: *The top sequences are the PU's and the bottom the data.*

## 2.4 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 7 shows the class structure of `scanfm`. The "Sequence" and "Range" classes inherit from the overall class "PUnit", and the "Reference" class inherit from the "Sequence" class, since a reference PU works exactly like a sequence PU at run-time when it has collected its letter sequence from either a sequence PU or a range PU.

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" functions 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.

### 2.4.1 Correctness of **scanfm**

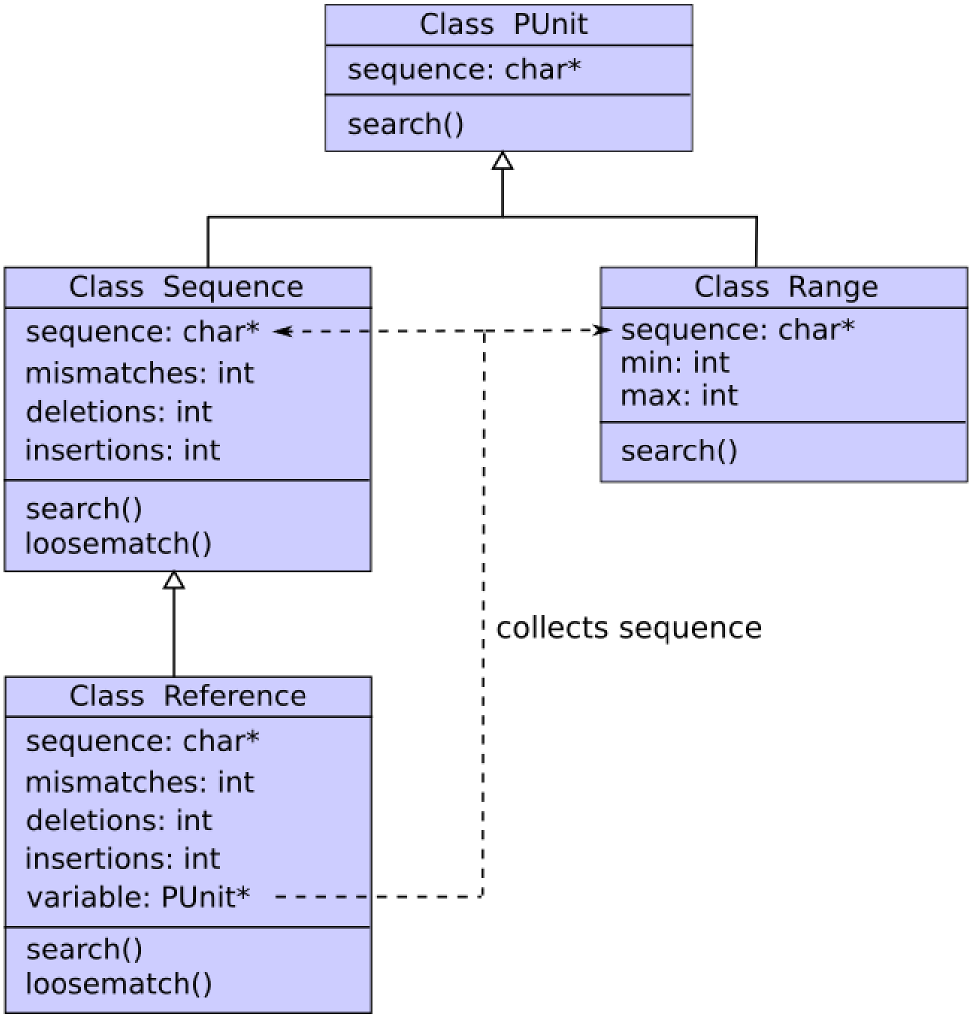

Figure 7: *A simplified class diagram which only shows the most important varibles and functions. There is one generic* PU *class called "PUnit". This class holds a lot of the common variables and flags that are needed no matter what kind of* PU. *Each class (except the "PUnit" class itself) defines the method "search", which searches for a match given a start position. The function "loosematch" is used for trying to find a match when single-character edits are allowed.*

### 2.4.2 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` actually stores 4 more bits (left of our 4-bit field) 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 way of comparing characters.

### 2.4.3 Why not Levenshtein distance?

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**]

Calculating the Levenshtein distance can be done quite efficiently, at least faster than our implementation calculates every possible match given the specific three amounts of single-character edits. One could think that a faster implementation would be just summing the number of allowed mismatches, insertions and deletions, calculating the Levenshtein distance between data sub-sequence and PU and then comparing to see if the allowed sum was exceeded. This would not be correct since 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 9.

```
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 8: *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.*

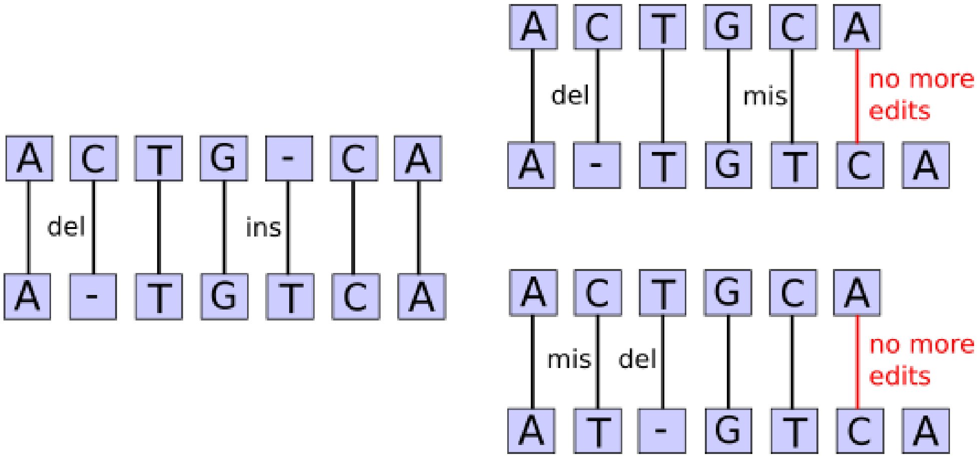

Figure 9: *The Levenshtein distance between the pattern ACTGCA and the data ATGTCA is 2 (1 deletion, 1 insertion) as the leftmost example shows. If the pattern is allowed 1 deletion and 1 mismatch, then the two strings can not match, as the rightmost examples show, even though this also would be a sum of 2 edits.*

### 2.4.4 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 the sequence PU with the most letters is the PU that is probably found least times in a big data file. 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. As Figure 10 shows, this optimization presents a potential huge increase in running time of the program.

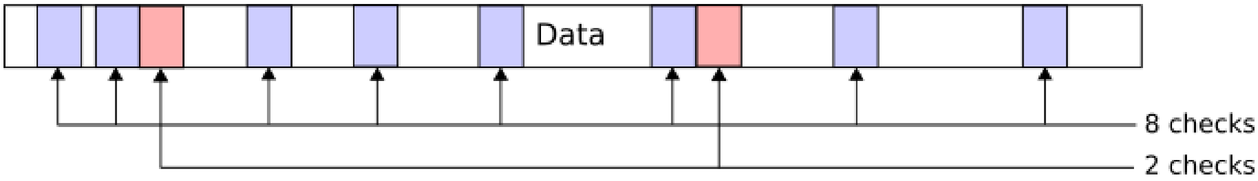

Figure 10: *A pattern that consists of two* PU's *where the second (say* CTCGAATAG[1,0,0] *) is more rare in the data than the first* PU *(say(*GGT*). Searching for the second* PU *first needs three checks (with three additional check afterwards to verify the whole pattern) while choosing the look for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

### 2.5 Clean code

Writing clean code is meant as writing understandable code, that provides overview of the programs structure. This is necessary when dealing with code that needs to be updated, further developed, checked for correctness, and debugged.

Clean code has been valued when creating scanfm , by dividing responsibilities into classes, and splitting those classes into suitable methods. This being said in cases where dividing into different methods started to cost running time, the fastest solution has been chosen. This has lead to a larger number of if statements where it can be more difficult to understand the larger picture of certain pieces algorithms.

The structure of the PU's has been constructed in a way that it should be possible without to much understanding of the other individual PU's code to create a new PU . This is so further updating and development should be as easy as possible.

Correctness can be a very strong ally when creating code that needs to be used scientifically, but also just in cases where correct outputs are important. Splitting the responsibilities into smaller classes can make it easier to find faults in the code. It can also be used to create unit tests for each method in the class.

Unit tests in scanfm would further help provide a certainty of correctness to the program, but because of the complexity of outputs and inputs in scanfm creating unit tests would be very time consuming and has not been done currently.

## 3 Correctness Test

In order to show that each PU works and returns the correct and expected results, they need to be tested individually and in specific combinations. This is done with the use of equivalence classes, this means proving that the edge scenarios of each PU and average scenarios returns the expected result, and by doing so providing a by statistically solid foundation that the PU is correct. In order to create the largest number of tested features with the least needed tests, the correctness groups has been constructed to create the largest equivalence classes possible.

| Feature grouping | Example | Explained | Result |
|---|---|---|---|
| Simple sequence | ”ACCG” | A sequence of bases only consisting of basic bases | Empty sequence returns unexpected results |
| MID sequence | ”GGGT[2,1,1]” | A sequence og bases with the possibility of mismatches, insertions, and deletions | Pass |
| Ambiguous base sequence | ”MMNN” | A sequence consisting of only ambiguous bases | Ambiguous base ”U” does not function |
| Ambiguous and basic bases | ”MSGGTM” | A sequence consisting of both ambiguous and basic bases | Ambiguous base ” does not function |
| Range | ”4...8” | A range of jumps in the data | Range with 0 does not terminate |
| Simple reference | ”p1=3...5 p1” | A reference being made and then searched for | Ranges with 0 results in wrong found matches |
| Complementary reference | ”p1=4...7 ~p1” | A reference being made and then the complementary is searched for | Ranges with 0 results in wrong found matches |
| MID reference | ”p1=5...6 p1[2,1,3]” | A reference being made and then searched for with the possibility for mismatches, insertions and deletions | Ranges with 0 results in wrong found matches |
| MID complementary reference | ”p1=3...8 ~p1[1,2,1]” | A reference being made and then the complementary is searched for with the possibility of mismatches, insertions and deletions | Ranges with 0 results in wrong found matches |
| Multiple PU’s | ”ACCGMG[1,1,1] 2...8 ACC” | Multiple PU’s in a single pattern | Multiple errors described below |

## 3.1 Found errors

When performing the correctness tests certain errors emerged. None of the found errors where alone enough to argue that the program is not correct, but if future development ever occurs they would need attention. In order to explain what these errors does in a search and how they could be fixed, they are put into categories:

### 3.1.1 Empty sequences

If a empty sequence occurs in a pattern, the parser should return a error, this does not happen. This should be changed in the parser, if 2 spaces occurs without anything in between a error message should be written explaining empty sequences are not allowed.

### 3.1.2 Ranges from 0 to 0

Ranges that does not move the pattern will alone cause a never ending loop. This is not a pattern that can be used to find a meaningful match and should not be allowed by the scanfm. A potent solution would then be to return a error.

### 3.1.3 Ranges from 0

Range from 0 without a PU after results in a never ending loop. This should not be the case, if a the last PU of a pattern is a range from 0, a parse error should occur, since it does not give a meaningful solution since everything reaching that PU is true. Another way of handling the error could be to always return successful in the event it reaches the range from 0.

### 3.1.4 References with ranges from 0

In the event a reference contains a range from 0, scanfm returns wrong results, this can not be fixed by placing a error because it is a valid pattern given by the user. In order to fix this the reference PU needs to be able to handle ranges that are 0.

# 4 Benchmark comparison

# 5 Appendics

## 5.1 Correctnes test results

### 5.1.1 Sequence

To create exhaustive equivalence classes for the PU sequence, there are multiple scenarios needed to be proven. A sequence can be exact by having no mismatches, insertions or deletions, it can be loose fitting by having mismatches, insertions and deletions or it can be combined with ambiguous bases. The equivalence class for the exact sequence, needs a test at 0, the smallest possible length and one with multiple bases:

| Case | Pattern | Data | Expected Result | Result |
|---|---|---|---|---|
| Empty string | ” ” | ACGTNNN | suiting error message | No matches |
| Single base | ”A” | ANNN | Match at 1 ”A” | Match at 1 ”A” |
| Multiple bases | ”ACCGT” | ACCGTNNN | Match at 1 ”ACCGT” | Match at 1 ”ACCGT” |

The equivalence class for the loose sequence, with mismatches, insertions and deletions has more edge scenarios. A case with 0 is needed, but in this class there are multiple scenarios for the smallest possible amount of mismatches, insertions and deletions, since these are not equivalent they are all needed to be tested. Also the combinations of the mismatches, insertions and deletions needs to be tested with both the smallest possible scenarios and average cases. Finally the scenarios with all of the possible mismatches, insertions and deletions being used needs to be tested with both the smallest scenario and a average case. With deletions and mismatches there are also the scenario where they exceed the pattern length. This also needs to be tested.

| Case | Pattern | Data | Expected Result | Result |
|---|---|---|---|---|
| MID 0 | ”ACGT[0,0,0]” | ACGTNNN | Match at 1 ”ACGT” | Match at 1 ”ACGT” |
| MID 1 miss | ”ACC[1,0,0]” | ACGNNN | Match at 1 ”ACG” | Match at 1 ”ACG” |
| MID 1 del | ”ATG[0,1,0]” | AGNNN | Match at 1 ”AG” | Match at 1 ”AG” |
| MID 1 ins | ”ACC[0,0,1]” | ATCCNNN | Match at 1 ”ATCC” | Match at 1 ”ATCC” |
| MID 2 | ”AGGT[1,1,0]” | ACTNNN | Match at 1 ”ACT” | Match at 1 ”ACT” |
| MID 2 | ”ACGT[1,0,1]” | AGTGTNNN | Match at 1 ”AGTGT” | Match at 1 ”AGTGT” |
| MID 2 | ”ATTGT[0,1,1]” | ATCGTNNN | Match at 1 ”ATCGT” | Match at 1 ”ATCGT” |
| MID 3 | ”AACGT[1,1,1]” | CAGGTNNN | Match at 1 ”CAGGT” | Match at 1 ”CAGGT” |
| MID miss multiple | ”ACCT[2,0,0]” | ATTTNNN | Match at 1 ”ATTT” | Match at 1 ”ATTT” |
| MID del multiple | ”AGTTT[0,2,0]” | AGTNNN | Match at 1 ”AGT” | Match at 1 ”AGT” |
| MID ins multiple | ”ACGT[0,0,2]” | ACTTGTNNN | Match at 1 ”ACTTGT” | Match at 1 ”ACTTGT” |
| MID multiple | ”ACTTT[1,2,0]” | GCTNNN | Match at 1 ”GCT” | Match at 1 ”GCT” |
| MID multiple | ”TCGAT[3,0,2]” | CGCTTATNN | Match at 1 ”CGCTTAT” | Match at 1 ”CGCTTAT” |
| MID multiple | ”TCGGT[0,2,3]” | GGGGGTNNN | Match at 1 ”GGGGGT” | Match at 1 ”GGGGGT” |
| MID all | ”ATTCCCTT[2,2,1]” | TTCCGTANNN | Match at 1 ”TTCCGTA” or ”TTCCGTAN” | Match at 1 ”TTCCGTAN” |
| MID more del | ”ATTC[0,6,0]” | CCCCCCNNN | A unknown number of matches not equal to 0 [1] | Match at 1, 2, 3, 4, 5, and 6 ”C” |
| MID more mis | ”ATTCG[8,0,0]” | GGGGTNNN | Match at 1 ”GGGGT” | Match at 1 ”GGGGGT” |

In order to create a equivalence class for ambiguous bases in a exact sequence PU , the same scenarios as a exact sequence is needed with the addition that all ambiguous bases needs to be shown to work individually.

| Case | Pattern | Data | Expected Result | Result |
|---|---|---|---|---|
| Single ambiguous base | ”R”(R is A or G) | ATTGNNN | Match at 1 ”A” and 4 ”G” | Match at 1 ”A” and 4 ”G” |
| Ambiguous bases | ”RRGGWW” | AAGGTANNN | Match at 1 ”AAGGTA” | Match at 1 ”AAGGTA” |
| All ambiguous bases | ”UMMRRWWS SYYKKBBB DDDHHHVV VNNNN” | TACAGATC GCTGTCGT AGTACTAC GACTGNNN | Match at 1 | no Match |
| All ambiguous bases except u | ”MMRRWWS SYYKKBBB DDDHHHVV VNNNN” | ACAGATC GCTGTCGT AGTACTAC GACTGNNN | Match at 1 | Match at 1 |

In order to make sure that ambiguous bases work correctly with the defined bases, a equivalence class is needed to show that the smallest mixed scenario and a average scenario works correctly.

| Case | Pattern | Data | Expected Result | Result |
|---|---|---|---|---|
| Smallest mix | ”AM” | ACAA | Match at 1 ”AC” and 3 ”AA” | Match at 1 ”AC” and 3 ”AA” |
| Average mix | ”TTATMTNYY” | TTATATGTC | Match at 1 ”TTATATGTC” | Match at 1 ”TTATATGTC” |

### 5.1.2 Range

Range by itself does not make room for a large difference in possibilities, in order to create a equivalence class only a few scenarios are needed, the smallest possible cases, a 0 case and a average case.

| Case | Pattern | Data | Expected Result | Result |
|---|---|---|---|---|
| Range from 0...0 | ”0...0” | ACCNNN | Suiting error | Never ending loop |
| Smallest range | ”0...1” | ACCNNN | Match at all bases | Never ending loop |
| Smallest range without 0 | ”1...1” | AGTNNN | Match at 1, 2, 3, 4, 5 and 6 | Match at 1, 2, 3, 4, and 5 |
| Standard case | ”4...9” | AAAAAAAA AANNN | Match at 1 and 5 | Match at 1 ”AAAA” |

### 5.1.3 Reference

Reference can’t be tested and proved correct as a individual PU since it needs to refer to another PU to function. In order to make sure the reference PU works correctly the equivalence class needs to obtain references with and without another PU between the refereed PU and the reference PU . Both a 0, edge and average case is needed for both scenarios.

| Case | Pattern | Data | Expected Result | Result |
|---|---|---|---|---|
| 0 reference with | ”p1=0...0 2...4 p1” | ACCCNNN | Match at 1 ”ACCC” | Match at 1 ”ACCCN” |
| Smallest reference with | ”p1=1...1 2...2 p1” | ACCANNN | Match at 1 ”ACCA” | Match at 1 ”ACCA” |
| Standard case with | ”p1=3...5 2...2 p1” | CCCCAACC CCNNN | Match at 1 ”CCCCAACCCC” | Match at 1 ”CCCCAACCCC” |
| 0 reference without | ”p1=0...0 p1” | TTGTANNN | Undefined | Match at 1 ”T”, 2 ”T”, 3 ”G”, 4 ”T”, 5 ”A”, 6 ”N”, 7 ”N”, and 8 ”N” |
| smallest reference without | ”p1=1...1 p1” | AANNN | Match at 1 ”AA” | Match at 1 ”AA” |
| Standard case without | ”p1=3...5 p1” | CACCCACC NNN | Match at 1 ”CACCCACC” | Match at 1 ”CACCCACC” |

In order to show correctness of complementary reference PU a equivalence class needs to contain the same scenarios as the equivalence class showing the simple reference scenario.

| Case | Pattern | Data | Expected Result | Result |
|---|---|---|---|---|
| Complementary 0 reference | ”p1=0...0 2...2 ~p1” | AAAANNN | Match at 1 , 2, 3 and 4 | Match at 1 ”AAA” |
| Complementary smallest reference | ”p1=1...1 2...2 ~p1” | ACCTNNN | Match at 1 ”ACCT” | Match at 1 ”ACCT” |
| Complementary standard case | ”p1=3...5 2...2 ~p1” | AATTCCAA TTNNN | Match at 1 ”AATTCCAATT” | Match at 1 ”AATTCCAATT” |
| Complementary 0 reference without | ”p1=0...0 ~p1” | AATNNN | Undefined behaviour | Match at 1 ”A”, 2 ”A”, 3 ”T”, 4 ”N”, 5 ”N”, and 6 ”N” |
| Complementary smallest reference withour | ”p1=1...1 ~p1 | ATNNN | Match at 1 ”AT” | Match at 1 ”AT” |
| Complementary reference after assignment | ”p1=3...3 ~p1” | AAATTTNN N | Match at 1 ”AAATTT” | Match at 1 ”AAATTT” |

Reference with mismatches, insertions and deletions equivalence class has the same basic steps as the simple reference scenario, but because MID are all ready tested in the sequence, there is no difference in the reference and we don’t need to test with every single case of mismatches, insertions and deletions. Instead the use of the standard scenario is used.

| Case | Pattern | Data | Expected Result | Result |
|---|---|---|---|---|
| Reference MID 0 | ”p1=0...0 2...2 p1[1,2,1]” | AACTGNNN | Undefined | Match at 1 ”AAC” |
| Reference MID smallest | ”p1=1...1 2...2 p1[1,0,0]” | AACCNNN | Match at 1 ”AACC” | Match at 1 ”AACC” |
| Reference MID | ”p1=3...4 2...2 p1[2,1,1]” | AAAATTCG TANNN | Match at 1 ”AAAATTCG’ or ”AAAATTCGT or ”AAAATTC” | Match at 1 ”AAAATTC” |
| Reference MID 0 without | ”p1=0...0 p1[1,1,1]” | ACTNNN | Undefined | Match at 1 ”A”, 2 ”C”, 3 ”T”, 4 ”N”, 5 ”N”, and 6 ”N” |
| Reference MID smallest without | ”p1=1...1 p1[2,1,2]” | ACGTNNN | Match at 1 ”AC” or ”ACG” or ”ACGT” or ”A” | Match at 1 ”ACGT” |
| Reference MID without | ”p1=3...5 p1[2,1,1]” | ACTTACNNN | Match at 1 ”ACTTAC” or ”ACTTA” or ”ACTTACN” | Match at 1 ”ACTTAC” |

Complementary reference with mismatches, insertions and deletions, in this case we make the same assumptions as above that mismatches, insertions and deletions are proved to be correct in the individual cases from the sequence test. Because of that we test with the same scenarios as the simple reference tests, to obtain a exhaustive equivalence class.

| Case | Pattern | Data | Expected Result | Result |
|---|---|---|---|---|
| Reference MID complementary 0 | ”p1=0...0 2...3 ~p1[1,2,1]” | ACGTGNNN | Undefined | Match at 1 ”ACG” |
| Reference MID complementary smallest | ”p1=1...1 2...3 ~p1[0,1,0]” | ACGGGNNN | Match at 1 ”ACG” | Match at 1 ”ACG” |
| Reference MID complementary | ”p1=3...4 2...3 ~p1[1,0,1]” | ACCTTGCC TNNN | Match at 1 ”ACCTTGCCT” | Match at 1 ”ACCTTGCCT” |
| Reference MID complementary 0 without | ”p1=0...0 ~p1[0,1,2] | ACCTNNN | Undefined | Match at 1 ”AC” |
| Reference MID complementary smallest without | ”p1=1...1 ~p1[0,0,2] | CTGCNNN | Match at 1 ”CTG” | Match at 1 ”CTG” |
| Reference MID complementary without | ”p1=3...4 ~p1[1,0,0]” | AAATACTT NNN | Match at 1 ”AAATACTT” | Match at 1 ”AAATACTT” |

In order to make a exhaustive equivalence class for multiple PU’s or multiple exhaustive equivalence classes for PU’s a great number of different combinations and scenarios would be needed. Instead the following tests performed gives a statistically approximation to this by looking at the scenarios that are most likely to produce false results.

| Case | Pattern | Data | Expected Result | Result |
|---|---|---|---|---|
| Multiple exacts | ”ACGT TGCA” | ACGTTGCANNN | Match at 1 ”ACGTTGCA’ | Match at 1 ”ACGTTGCA” |
| Range MID | ”4...5 TTTT[1,1,1]” | AAAAATTCGNNN | Match at 1 ”AAAAATTC’ or ”AAAAATT” | Match at 1 ”AAAAATT” |
| Backtracking MID | ”TTTT[0,1,1] TTA” | TTTTTAANNN | Match at 1 ”TTTTTA” | Match at 1 ”TTTTTA” |
| Multiple references | ”p1=2...3 p2=2...2 p1 p2” | TTTCCTTTCCNNN | Match at 1 ”TTTCCTTTCC” | Match at 1 ”TTTCCTTTCC” |
| Mulitple complementary reffenreces | p1=3...3 p2=2...3 2...3 ~p2 ~p1” | AAACCGTTTNNN | Match at 1 ”AAACCGTTTCGGTTT” | Match at 1 ”AAACCGTTTCGGTTT” |
| 0 refference with bounding PU’s | ”AATCC p1=0...1 0...3 p1 TTG” | AATCCTTGG | Match at 1 | Not terminating |
| Range from 0 | ”ATC 0...3 TTC” | ATCTTCNNN | Match at 1 ”ATCTTC” | Match at 1 ”ATCTTC” |
| reference with to many deletion | ”p1=2...4 ATTT p1[0,6,0] TTT” | TTTATTTNN] | No match | No matches |
| sequence with more deletions than bases | ”TTTT ACC[0,6,0] TT” | TTTTACNNN | no matches | No matches |