UNIVERSITY ”ALEXANDRU-IOAN CUZA” FROM IAȘI

FACULTY OF COMPUTER SCIENCE

BACHELOR THESIS

Implementing and Verifying the BMH Algorithm in F*

proposed by

Daniel-Antoniu Dumitru

Session: july, 2024

Scientific coordinator

Associate Professor Ștefan Ciobâcă

Contents

Introduction

One of my favourite courses in the faculty was Functional Programming. It made me see computer science from a different point of view and develop my critical and analytical thinking. For these reasons, I found the idea of making my bachelor thesis on a subject related to this course interesting.

The subject of the thesis is verifying that a F* implementation of the Boyer-Moore-Horspool algorithm is correct. The thesis is divided into 3 parts:

1. The Boyer-Moore-Horspool algorithm - this chapter describes the way the algorithm works, what problem it solves and what is the output data;
2. The F* language - here I describe the programming language I used for my thesis, the motivation of choosing it and the particular components that were useful;
3. The proof for the Boyer-Moore-Horspool algorithm - this part describes the solution I implemented and the steps I made.

Contributions

The main contributions I had are:

1. implementing a function that returns the indices where a certain value is found on a list (if there are any);
2. implementing a function for preprocessing the pattern string;
3. proving lemmas for each function, which ensure that the result is correct;
4. implementing the main algorithm;
5. specifying the main algorithm;
6. verifying the main algorithm.

Chapter 1

The Boyer-Moore-Horspool Algorithm

The Boyer-Moore algorithm is based on two heuristics: the bad character heuristic and the good suffix heuristic. Each of them can be used independently and helps in shifting the pattern such that we can find a match (if there exists one) efficiently. One peculiarity of this algorithm is that it can be faster when the size of the input data is larger[1].

The algorithm Boyer-Moore-Horspool is a simplified version of Boyer-Moore and it uses only the bad character heuristic. As input data, it takes 2 parameters: a string `text` and a string `pattern`. The algorithm returns an integer. Its value is the first position in `text` where `pattern` is found or -1 if `pattern` is not a substring of `text`. There is also the version of Boyer-Moore-Horspool where the output is a bool value: true if `pattern` is in `text` and false otherwise.

1.1 Rules for Matching

Both `text` and `pattern` are string values. The 2 parameters must fulfill 2 criteria: all of the characters in both strings are elements in the same alphabet (defined by a string with the name `alphabet`) and the length of `pattern` is less or equal than the length of `text`. The comparison between characters in `text` and characters in `pattern` starts from right to left and is made while there is an equality relation between the values in the 2 indices (or while there are characters in `pattern`). If there is a mismatch, we focus only on the character in the text where the mismatch happened (the bad character) and its appearances in the `pattern` string. There are 3 possible cases:

1. The last occurrence of the bad character in `pattern` is to the left of the mismatch - here, the shift is made such that the respective occurrence matches the character in `text`;

2. The bad character is not in `pattern` - since there cannot be any match, `pattern` is shifted such that it is positioned just after the bad character (on the next position in `text`);

3. The last occurrence of the bad character in `pattern` is to the right of the mismatch - it is ineffective to match that position with the bad character because the shift will be to the left, where we already know that the `pattern` does not match with the `text`. In this particular case, the offset is incremented by 1 in order to not miss any possible matching [1].

1.2 Preprocessing

To use the bad character heuristic, it is necessary to preprocess the pattern string. In order to do that, an array of the same length as `alphabet` is created. Let us call this array `bc`. Each index in `bc` corresponds to the character in `alphabet` stored in the same index (for example, the second index in `bc` corresponds to the second character in `alphabet`). At each index we stored the last position of the character in `pattern`. To treat the case where a value in the alphabet is not in `pattern`, all of the indices in `bc` will be initially set to -1. After that, we go through each index in `pattern`, from the first one to the last one, and we store the position of the character in `bc`. If a symbol occurs in more than one position, the last one will overwrite the other ones.

1.3 The Main Algorithm

Based on the presentation [1], the Boyer-Moore-Horspool algorithm is:

```
int Boyer-Moore-Horspool (string T) (string P) {
  int k = 0;
  int i = 0;
  int m = P.size();
  int n = T.size();
  while(k < m && i <= n - m) {
    if(T[i + m - 1 - k] == P[m - 1 - k]) k++;
    else {
      //BC is the preprocessing of the string P
      int shiftbc = m - k - 1 - BC[T[i + m - 1 - k]];
      //max(shiftbc, 1) is used for the case when the
      //last position of the bad character is at a greater
      //position than the one where we found the mismatch
      i += max(shiftbc, 1);
      k = 0;
    }
  }
  if(k == m) return i;
  else return -1;
}
```

The variable `i` represents the position in `T` where it is a possible match with `P`. The variable `k` represents the number of consecutive characters in `P` (starting with the last one) that are equal to the corresponding characters in `T`. The value is increased by 1 if `T`[`i` + `m` - 1 - `k`] = `P`[`m` - 1 - `k`]. If the value is `m` (the length of `P`), then all of the characters match, so there is a match between `P` and `T` at position `i`. If there is a mismatch between characters, the variable `shiftbc` stores the shift between the position of the mismatch and the last position of the bad character in `P`. If `shiftbc` is negative (it cannot be 0 because in that case the last position of the bad character in `P` is the position of the mismatch), then the last position is to the right of the mismatch. In this case, we increment `i` by 1, otherwise, `i` is incremented by `shiftbc`. If the value of `i` is greater than the difference between the length of `T` and the length of `P`, then `P` is not found in `T` and -1 is returned.

Chapter 2

The F* Language

F* is a proof-oriented programming language. It can be used as a proof assistant and as a verification engine, in order to verify logical or mathematical relations. To do that, F* uses the SMT solver Z3. However, the language still needs a little help from the programmer to ensure certain statements. Another role of the language is as a compiler. It is possible to have functions with effects such as printing on the screen an output value. To see the result, we need an executable file. We cannot make an executable file in an F* file, but we can compile the F* file, translate it to OCaml and make an executable file in the OCaml file [2].

2.1 Lemmas and Refinements

A Lemma is a special function from the F* language. It has as output data the unit value and one or more statements within the value, which represents properties that we want to prove. If the proof was verified and no errors were given, then the proof is valid [2]. The lemmas can be proved by F* directly or with the help of assertions, sublemmas or induction. For the last part, the lemma is a recursive function with a base case and an inductive step. Refinements are another important component of F*. They are used to set restrictions for certain variables. For example, if you have a function with a natural number `i` as parameter and you want to ensure that `i` is in a segment [`a`,`b`] (`a` and `b` are natural numbers, with `a` less than `b`), you can set the restriction with a refinement: `(i:nat{i >= a && i <= b})`.

2.2 Useful Instructions

Besides the components described above, for proving the algorithm, 3 instructions helped me in finding a solution for an incomplete proof:

1. `assert()` - the instruction was very useful for testing what statement is needed in order for the proof to be complete or to reformulate statements or mathematical operations in a way that F* can understand;

2. `assume()` - when a certain proposition was not valid, I used `assume()` to verify if that proposition was the reason why the lemma was not provable;

3. `admit()` - for a proof by induction or for an if statement, `admit()` helped me in finding the part of the function which didn't had enough data for making the proof valid.

The only one instruction present in the proofs is assert.

Chapter 3

The Proof of the Boyer-Moore-Horspool Algorithm

3.1 Defining the Global Variables

A key element in Boyer-Moore-Horspool algorithm is the `alphabet` array. All of the characters in `text` and `pattern` must be elements in this vector. In order to define it, I created a type in F* that contains the first 2 letters in the english alphabet.

```
type english_letters =
  | A
  | B
```

After that I defined the `alphabet` array as a list that contains both of the elements in the new type created. I used a refinement type to specify that the list cannot be empty and all of the elements of type `english_letters` are in the array.

```
val alphabet : (l:list english_letters
               {(forall (x:english_letters). mem x l = true)
               /\  l <> []})
let alphabet = [A;B]
```

The next two global variables are `text` and `pattern`. Like `alphabet`, each of these variables are lists of `english_letters`. The variable `text` is a non-empty list.

```
val text : (l:list english_letters{l <> []})
let text = [A;A;A;A;B;A;A;B;A;B;A;A;A;B;A;B;B]
```

The variable pattern is a list of length less or equal than the length of text. Based on the definition, the second list can have 0 elements. If we want to assign a value to a global list l, if the type of the list has the refinement length l <= length text, F* cannot prove the query if the length of l is greater than 7.

This happens because of the fuel limit, which is set by default to 8. The limit is used for recursive functions, in order to set the maximum number of recursive calls (including the initial call of the function) to the value of fuel [2]. In our case, if the length of the global list l is at least 8, the result cannot be calculated and the inequality cannot be verified. To set the value of pattern, we will check if the length of the value that we want to set is less or equal than the length of text.

```
val pattern : (l:list english_letters
                {length l <= length text})
let pattern = if length [B;A;B;A;A;A;B;A;B] <= length text
              then [B;A;B;A;A;A;B;A;B]
              else []
```

If we have a recursive function f, the fuel limit is applied only to this function, and not to the calls of other functions in its body. This is why the above implementation works.

The functions length and mem are defined in the FStar.List.Tot.Base library. The first function, length l, returns the length of the list l and the second function, mem x l, checks if x is in the list l (x has the same type as the elements in l) [4].

3.2 Defining a Function that Stores a List of Indices

Some useful functions for lists are defined in the standard F* libraries. However, there was no function that returns the index where a certain element is stored in a list, so I implemented such a function, which returns a list with all of the indices where a value can be found, named item_indices. The main use of item_indices is for finding the last position of each value in pattern, which is the last element in the resulting list.

```
let rec item_indices (#a:eqtype) (item:a) (l:list a) (i:nat)
  : list nat
  = match l with
    | [] -> []
    | hd :: tl -> if hd = item
                  then i :: item_indices item tl (i + 1)
                  else item_indices item tl (i + 1)
```

In order to return the correct result, the parameter `i` needs to start with the value 0. In this way, `item_indices` will have for each item in the list the position where it can be found.

The next step I did was to verify that the output data meets certain criteria. The first one is that the length of the result is equal to the number of appearances of `item` in `l`.

```
let rec item_list_has_correct_length (#a:eqtype) (l:list a)
    (i:nat)
  : Lemma (ensures forall (item:a).
  length (item_indices item l i) = count item l)
  = match l with
    | [] -> ()
    | hd :: tl -> item_list_has_correct_length tl (i + 1)
```

This is a proof by induction. The base case of the lemma is when `l` is an empty list. The result of `item_indices item l i` is also an empty list, and the length of it is 0. The count of any item in `l` is 0, since there is no element. The equality is true. The case where the list is not empty represents the inductive step. We suppose that for every natural number `j < i` the property is true and we prove the criterion for the value `i`. The following equalities hold:

```
if hd = item then (
  length (item_indices item l i) =
  1 + length (tail (item_indices item l i)) =
  1 + length (tail (i :: item_indices item tl (i + 1))) =
  1 + length (item_indices item tl (i + 1));
  count item l = 1 + count item tl
)
else (
  length (item_indices item l i) =
  length (item_indices item tl (i + 1));
  count item l = count item tl
)
```

The equality remains valid due to the recursive call in the lemma. After a number of steps, the recursion reaches the base case, where the property is proved.

The second criterion of the output data is that each value in the resulted list needs to be in the segment [`i`, `i + length l`). For this, I made another lemma.

```
let rec item_indices_is_in_interval (#a:eqtype) (item:a)
                                    (l:list a) (i:nat) (x:nat)
  : Lemma (ensures mem x (item_indices item l i) ==>
                   i <= x && x < i + length l)
  = match l with
    | [] -> ()
    | hd :: tl -> item_indices_is_in_interval item tl (i + 1) x
```

As in the previous lemma, I used a proof by induction. Here, the property is valid for an arbitrary natural number `x`. We want to prove the implication for all possible values of `x`. To achieve this, I used the `forall_intro` function in the `FStar.Classical` library [3].

```
let item_indices_is_in_interval_forall (#a:eqtype) (item:a)
                                       (l:list a) (i:nat)
  : Lemma (ensures forall (x:nat).
           mem x (item_indices item l i) ==>
           i <= x && x < i + length l)
  = forall_intro (item_indices_is_in_interval item l i)
```

The third thing I proved is that a certain index in the list does not store the value of the item if and only if that index is not part of `item_indices item l 0`. The parameter `i` must be 0 to cover all of the indices of a list.

```
let index_not_mem_not_item_not_item_not_mem
    (#a:eqtype) (l:list a)
  : Lemma (ensures forall (item:a) (i:nat{i < length l}).
           mem i (item_indices item l 0) = false <==>
           index l i <> item)
  = index_is_not_item_is_not_mem l;
    index_is_not_mem_is_not_item l
```

For the double implication to be valid, both the direct and the reciprocal have to be valid. The direct was the first verified implication.

```
let rec index_is_not_mem_is_not_item_base (#a:eqtype)
    (l:list a) (item:a) (i:nat{i < length l})
  : Lemma (ensures mem i (item_indices item l 0) = false ==>
                   index l i <> item)
  = match l with
    | [] -> ()
    | hd :: tl ->
      match i with
      | 0 -> if hd = item
             then ()
             else item_indices_is_in_interval item tl (i + 1) i
      | _ -> index_is_not_mem_is_not_item_base tl item (i - 1);
             item_indices_one_and_item_indices_zero
               item tl i 1
```

The function `index` is another function in the `FStar.List.Tot.Base` library. It returns the n-th item in the list `l`. As a constraint, the value of `n` needs to be less than the length of `l`. Based on [4], the definition of the function is:

```
let rec index #a (l: list a) (i:nat{i < length l}) : Tot a =
  if i = 0 then hd l
  else index (tl l) (i - 1)
```

If `i` is 0, then the first element of the list is returned, otherwise a recursive call is made with the list without the first element. In order to see the result of `index l i`, `i` needs to be 0. For that reason, if the list is not empty, we distinguish 3 cases:

1. if `i = 0` and `hd = item`: in this case, 0 can be found in `item_indices item l 0`, thus the first proposition of the implication is false. For that reason, the

implication is true, no matter if the second proposition is true or false (in this case is false);

2. if `i = 0` and `hd <> item`: here, the least possible value in `item_indices item l 0` is 1, thus 0 is not part of the list;

3. if `i > 0`: this is the inductive step. A recursive call of the function is made and a lemma is used to ensure that the tail of the index list is equal with the list from the recursive call (`mem i (item_indices item tl 1)` and `mem (i - 1) (item_indices item tl 0)` are equal).

The second verified implication was the reciprocal.

```
let rec index_is_not_item_is_not_mem_base (#a:eqtype)
    (l:list a) (item:a) (i:nat{i < length l})
  : Lemma (ensures index l i <> item ==>
           mem i (item_indices item l 0) = false)
  = match l with
    | [] -> ()
    | hd :: tl ->
      match i with
      | 0 -> if hd = item
             then ()
             else item_indices_is_in_interval
                    item tl (i + 1) i
      | _ -> if hd = item then
             index_increases_with_one_when_first_is_item
               item tl i
             else
             index_increases_with_one_when_element_is_added
               item tl i hd;
             index_is_not_item_is_not_mem_base
               tl item (i - 1)
```

For both the direct and the reciprocal, I used `forall_intro_2` in order to verify the correctness of the lemmas for all possible values of `item` and `i` [3].

The fourth property that ensures the correctness of the result is that a certain index in the list has the value of the item if and only if that index is part of `item_indices item l 0`. Again, I proved this and the reciprocal and I used them to verify the double implication.

```
let index_mem_item_item_mem (#a:eqtype) (l:list a)
  : Lemma (ensures forall (item:a) (i:nat{i < length l}).
          mem i (item_indices item l 0) = true <==>
          index l i = item)
  = //the direct
    index_is_mem_is_item l;
    //the reciprocal
    index_is_item_is_mem l
```

The last criterion that I verified is that `item_indices item l 0` is in ascending order. More precisely, for any two positions in the list (if there are at least two elements), the smaller position has the smaller value.

```
let rec indices_are_ordered_in_ascending_order_base
     (#a:eqtype) (l:list a) (item:a)
     (i:nat{i < length (item_indices item l 0)})
     (j:nat{j < length (item_indices item l 0)})
  : Lemma (requires i < j)
          (ensures
          (let indices = item_indices item l 0 in
           index indices i < index indices j))
  = match l with
    | [] -> ()
    | fst :: tl ->
      let l' = item_indices item l 0 in
      let l'' = item_indices item tl 1 in
      let l''' = item_indices item tl 0 in
      match i with
      | 0 -> if fst = item then (
                mem_index_element l'' (j - 1);
```

```
        item_indices_is_in_interval item tl 1
          (index l'' (j - 1))
      )
      else (
        if length l'' < 2
        then ()
        else (
          mem_index_element l' j;
          index_not_zero_not_first_element l item 0 j;
          hd_item_indices_is_min l item (index l' j) 0
        )
      )
  | _ -> item_indices_zero_is_item_indices_one_minus_one
        tl item 0;
      assert(l''' = map subtraction l'');
      if fst = item then (
        assert(index l' i = index l'' (i - 1));
        assert(index l' j = index l'' (j - 1));
        mem_index_element l'' (i - 1);
        mem_index_element l'' (j - 1);
        item_map_subtraction l'' (i - 1);
        item_map_subtraction l'' (j - 1);
        item_indices_is_in_interval item tl 1
          (index l'' (i - 1));
        item_indices_is_in_interval item tl 1
          (index l'' (j - 1));
        assert(index l''' (i - 1) =
              (index l'' (i - 1)) - 1);
        assert(index l''' (j - 1) =
              (index l'' (j - 1)) - 1);
        indices_are_ordered_in_ascending_order_base
          tl item (i - 1) (j - 1)
      )
      else (
        assert(index l' i = index l'' i);
        assert(index l' j = index l'' j);
        mem_index_element l'' i;
        mem_index_element l'' j;
        item_map_subtraction l'' i;
        item_map_subtraction l'' j;
        item_indices_is_in_interval item tl 1
          (index l'' i);
        item_indices_is_in_interval item tl 1
          (index l'' j);
        assert(index l''' i = (index l'' i) - 1);
        assert(index l''' j = (index l'' j) - 1);
        indices_are_ordered_in_ascending_order_base
          tl item i j
      )
```

Since F* complains if the parameter `i` in `index l i` is greater or equal than the length of `l`, the values of `i` and `j` have to be less than the length of `item_indices item l 0`. For the statement to be true, it is necessary that `i` be less than `j`. The `requires` proposition and the refinements of the variables assures F* that these inequalities hold in our proof.

The main problem in verifying the inequality was when I made the recursive call (case where `i` was greater than 0). In the new call, the indices list is `item_indices item tl 0`, but the initial function has the indices list `item_indices item l 0`. There was a mismatch between the inequalities:

```
//for item_indices item l 0
let indices = item_indices item l 0 in
//if first element from l is equal with item
index indices i = index (0 :: item_indices item tl 1) (i - 1)
                = index (item_indices item tl 1) (i - 1)
or
//if first element from l is not equal with item
index indices i = index (item_indices item tl 1) i
//the same equalities are valid for index indices j
```

```
//in the new call of the function
//indices = item_indices item tl 0
```

For the proof to be valid, I made an equality relation between elements in the lists `item_indices item tl 0` and `item_indices item tl 1`. The first step was writing a function that decreases by one each strictly positive natural number.

```
let subtraction (i:nat) : nat
  = if i > 0
    then i - 1
    else i
```

Then, I verified that it gives the correct result.

```
let proof_subtraction (i:nat)
  : Lemma (ensures i > 0 ==> subtraction i = i - 1)
  = ()
```

The relation between the 2 lists is that at each position `i`, the value in the first list is the value in the second list subtracted with one. To prove this, I used the `map` function in `Fstar.List.Tot.Base` library. The function `map` has 2 parameters: a function `f` is applied on each element in the second parameter, a list `l`. The demonstration is for a general case, but to fit with what we need, the value of `m` has to be set to 0.

```
let rec item_indices_zero_is_item_indices_one_minus_one
  (#a:eqtype) (l:list a) (item:a) (m:nat)
  : Lemma (ensures item_indices item l m =
          map subtraction (item_indices item l (m + 1)))
  = match l with
    | [] -> ()
    | hd :: tl ->
      item_indices_zero_is_item_indices_one_minus_one
        tl item (m + 1)
```

Another useful property is that for 2 lists, `l` and `l' = map subtraction l` (the lists have the same length), and a position `i`, the value in the position `i` of the first list is smaller by 1 than the value in the position `i` of the second list. This is used for the indices of the inequality.

```
let rec item_map_subtraction (l:list nat)
                              (i:nat{i < length l})
  : Lemma (ensures
          (let l' = map subtraction l in
           index l i > 0 ==> index l' i = (index l i) - 1))
  = match l with
    | [] -> ()
    | fst :: tl -> match i with
                   | 0 -> ()
                   | _ -> item_map_subtraction tl (i - 1)
```

Using the 2 lemmas above, we obtain the following statements:

```
//relation between the 2 lists
item_indices item tl 0 =
map subtraction (item_indices item tl 1);
//relation between the 2 indices
let indices = item_indices item tl 0 in
let indices' = item_indices item tl 1 in
index indices i = (index indices' i) - 1 &&
index indices j = (index indices' j) - 1;
//the inequality
index indices i < index indices j ==>
(index indices' i) - 1 < (index indices' j) - 1 ==>
index indices' i < index indices' j
//for (i - 1) and (j - 1) the above statements are the same
```

Since both indices in the new call are smaller by 1 than the indices in the initial function, the inequality is not affected.

To verify the inequality between any 2 variables i and j, we need to get rid of the requires statement. The function move_requires in the FStar.Classical library does that, creating an implication where the first proposition is the requires statement and the second proposition is the ensures statement [3].

```
let indices_are_ordered_in_ascending_order_implication
    (#a:eqtype) (l:list a) (item:a)
    (i:nat{i < length (item_indices item l 0)})
    (j:nat{j < length (item_indices item l 0)})
  : Lemma (ensures i < j ==>
          (let indices = item_indices item l 0 in
           index indices i < index indices j))
  = move_requires
    (indices_are_ordered_in_ascending_order_base l item i) j
```

In the end, I used `forall_intro_2` to make the lemma valid for any two values `i` and `j`, both of them less than the length of `item_indices item l 0`.

```
let indices_are_ordered_in_ascending_order
    (#a:eqtype) (l:list a) (item:a)
  : Lemma (ensures forall
          (i:nat{i < length (item_indices item l 0)})
          (j:nat{j < length (item_indices item l 0)}).
           i < j ==>
          (let indices = item_indices item l 0 in
           index indices i < index indices j))
  = forall_intro_2
    (indices_are_ordered_in_ascending_order_implication l item)
```

3.3 Function for Value of a Position in `bc`

In order to make the `bc` list, we need a function that computes, for any character `c`, its last position in a list `l`. The function `new_or_old` does that. If `item_indices c l 0` is not an empty list, the output is the last item in `item_indices` list (which is the last position of `c` in `l`), otherwise, the returned value is -1 (`c` is not part of `l`).

```
let new_or_old (#a:eqtype) (c:a) (l:list a)
  : int
  = if item_indices c l 0 <> []
    //last returnes the last value from the list
    then last (item_indices c l 0)
    else -1
```

If the value returned is -1, then the value of the character `c` is not in any position in the list. A lemma for an arbitrary natural number `i` (less than the length of `l`) proves this.

```
let rec new_or_old_not_item_base (#a:eqtype) (l:list a)
        (item:a) (i:nat{i < length l})
  : Lemma (ensures new_or_old item l = -1 ==>
                   index l i <> item)
  = match l with
    | [] -> ()
    | hd :: tl ->
      match i with
      | 0 -> ()
      | _ -> new_or_old_not_item_base tl item (i - 1);
             not_new_or_old_empty_list l item;
             empty_list_length_zero l item;
             length_zero_count_zero l item;
             count_zero_mem_false l item;
             mem_false_not_item l item i
```

For the proof, I used sublemmas for proving implications, starting from the relation `new_or_old item l = -1` and ending at the inequality `index l i <> item`.

```
new_or_old item l = -1 ==> item_indices item l 0 = [];
item_indices item l 0 = [] ==>
  length (item_indices item l 0) = 0;
length (item_indices item l 0) = 0 ==> count item l = 0;
count item l = 0 ==> mem item l = false;
mem item l = false ==> index l i <> item
```

After that, I made the lemma valid for any value `i`.

```
let new_or_old_not_item (#a:eqtype) (l:list a) (item:a)
  : Lemma (ensures forall (i:nat{i < length l}).
           new_or_old item l = -1 ==>
           index l i <> item)
  = forall_intro (new_or_old_not_item_base l item)
```

If the value returned is not -1, then it is the position in the list where the value of the character `c` is found and at any greater position a value that is not equal with the one from the character `c` is found.

```
let new_or_old_not_empty_list_correct_item (#a:eqtype)
    (l:list a) (item:a)
  : Lemma (requires item_indices item l 0 <> [] &&
             last (item_indices item l 0) < length l)
            (ensures new_or_old item l <> - 1 ==>
             index l (last (item_indices item l 0)) = item /\
             (forall (i:nat{i > (last (item_indices item l 0))
             && i < length l}). index l i <> item))
  = new_or_old_not_empty_list l;
    not_empty_list_mem_is_true l;
    mem_index_is_item_gives_correct_result l item
```

The steps I took for proving the lemma are:

1. I verified that if the result of `new_or_old` is not -1, then the list `item_indices item l 0` is not empty and the last element is part of it;

2. I used the Lemma `index_is_mem_is_item` to ensure that the last element is a position where we can find the value of item in the list `l`;

3. I proved, via the fact that the indices list is in ascending order, that the last value in the indices list has the biggest value;

4. I used the previous statement, together with the previously proved property `mem i (item_indices item l 0) = false ==> index l i <> item` for all values of `i`, to prove that for every value greater than the last one in the indices list, if it is a position from the `l` list, then the value stored is not equal with `item`;

5. I combined the results in 2 and 4.

For our problem, `new_or_old` takes as parameters a character in `alphabet` and the `pattern` list.

3.4 Verifying the Correctness of the Updated bc

The last step of preprocessing the pattern list is to update the positions in the bad character array. I made a general function which does the preprocessing of an arbitrary list p (of type english_letters) given an arbitrary list a as alphabet (of type english_letters). For each value hd in a, new_or_old hd p is applied.

```
let rec update_bc (a:list english_letters)
                  (p:list english_letters)
  : list int
  = match a with
    | [] -> []
    | hd :: tl -> (new_or_old hd p) :: update_bc tl p
```

One important thing about the output of update_bc is to have the same length as the length of a. Thus, each integer in the result has a correspondent in a.

```
let rec update_bc_length_is_initial_list_length
    (l:list english_letters) (p:list english_letters)
    : Lemma (ensures length (update_bc l p) = length l)
    = match l with
      | [] -> ()
      | hd :: tl -> update_bc_length_is_initial_list_length
                    tl p
```

The other criteria that needs to be fulfilled by the output list are proved with the help of the properties in new_or_old:

1. If on a position i the value is -1, then the value in the position i of a is not found in p;

   ```
   let rec update_bc_for_minusone_base
           (l:list english_letters)
           (p:list english_letters)
           (i:nat{i < length l &&
                  i < length (update_bc l p)})
           (j:nat{j < length p})
   ```

```
  : Lemma (ensures
          (let l' = update_bc l p in
           index l' i = -1 ==>
           index p j <> index l i))
  = update_bc_length_is_initial_list_length l p;
    match l with
     | [] -> ()
     | fst :: tl ->
       match i with
       | 0 -> assert(update_bc l p = (new_or_old fst p)
                   :: update_bc tl p);
              assert(index (update_bc l p) i =
                   new_or_old fst p);
              assert(fst = index l i);
              new_or_old_not_item_base p fst j;
              assert(new_or_old fst p = -1 ==>
                   index p j <> fst)
       | _ -> update_bc_for_minusone_base
              tl p (i - 1) j
```

2. If on a position `i` the value is `x <> -1`, then, for the value in the position `i` of `a`, `x` is the last position in `p`;

```
let rec update_bc_for_nat_base (l:list english_letters)
        (p:list english_letters)
        (i:nat{i < length l &&
               i < length (update_bc l p)})
        (j:nat{j < length p})
  : Lemma (ensures
          (let l' = update_bc l p in
           index l' i = j ==> index p j = index l i /\
            (forall (i':nat{i' > j && i' < length p}).
             index p i' <> index l i)))
  = update_bc_length_is_initial_list_length l p;
    match l with
    | [] -> ()
    | fst :: tl ->
      match i with
      | 0 -> assert(update_bc l p = (new_or_old fst p)
                    :: update_bc tl p);
             assert(index (update_bc l p) i =
                    new_or_old fst p);
             assert(fst = index l i);
             if j = new_or_old fst p then (
               if item_indices (index l i) l 0 <> []
               then (
                 item_indices_is_in_interval (index l i)
                   l 0
                   (last (item_indices (index l i) l 0));
                 new_or_old_not_empty_list_correct_item
                   p fst
               )
               else ()
             )
             else ()
      | _ -> update_bc_for_nat_base tl p (i - 1) j
```

3. Any value in the returned list is either -1 or a natural number, less than the length of p.

```
let rec update_bc_has_values_in_interval
          (l:list english_letters)
          (p:list english_letters)
          (i:nat{i < length (update_bc l p)})
    : Lemma (ensures (let l' = update_bc l p in
             index l' i = -1 ||
             (index l' i >= 0 && index l' i < length p)))
    = match l with
      | [] -> ()
      | fst :: tl ->
```

```
            if i = 0 then
            new_or_old_return_values p fst
            else update_bc_has_values_in_interval tl p (i - 1)
```

The updated bc list which will be used in the main algorithm is the result of the function `update_bc alphabet pattern`.

```
let final_bc : list int
  = update_bc alphabet pattern
```

Since the criteria have been proved for 2 arbitrary lists `a` and `p`, to make them valid for `final_bc` we just have to use the lemmas with the parameters `alphabet` and `pattern`.

3.5 Specifying the Main Algorithm

To specify the main algorithm, I made a function that checks if a substring `p` is part of a string `t`. To be possible, the length of `p` needs to be less or equal to the length of `t`.

```
let rec belongs (t:list english_letters)
        (p:list english_letters{length p <= length t})
        (i:nat)
  : bool
  = match p with
    | [] -> true
    | hd :: tl -> if i < length t && hd = index t i
                  then belongs t tl (i + 1)
                  else false
```

The Boyer-Moore-Horspool algorithm has 4 parameters:

1. `t` - a list of type `english_letters`, represents the text string;
2. `p` - a list of type `english_letters` with the length less or equal than the length of `t`, is the pattern string;

3. `k` - a natural number with the value less or equal than the length of `p`, used to store the number of characters that match between `t` and `p`;

4. `i` - a natural number, used to store the starting position of a possible match between `t` and `p`.

To ensure that the function is total, one parameter needs to decrease at each recursive call. Since the starting values of `k` and `i` are 0 and none of them decreases (either `k` is increased by 1 or `i` is increased by a shift value), the decreasing values are `length t- length p + 1 - i` and `length p - k`. To prove the termination of the algorithm, the two values needs to be specified in the `decreases` statement.

3.6 Implementing the Main Algorithm

The implementation of the main algorithm in F* is:

```
let rec boyer_moore_horspool (t:list english_letters)
        (p:list english_letters{length p <= length t})
        (k:nat{k <= length p}) (i:nat)
  : Tot (result:int{result >= -1})
  (decreases %[(if i < length t - length p + 1 then
                length t - length p + 1 - i else 0);
                length p - k])
  = let m = length p in
    let n = length t in
    if k = m then i
    else if i > n - m then -1
    else (
      if index t (i + m - 1 - k) = index p (m - 1 - k)
      then boyer_moore_horspool t p (k + 1) i
      else (
        update_bc_length_is_initial_list_length alphabet p;
        mem_last_list (item_indices (index t (i + m - 1 - k))
                      alphabet 0);
        item_indices_is_in_interval_forall alphabet 0
        (index t (i + m - 1 - k));
      let shiftbc = m - k - 1 -
        index (update_bc alphabet p) (last (item_indices
        (index t (i + m - 1 - k)) alphabet 0)) in
      let value = i + (maximum 1 shiftbc) in
      boyer_moore_horspool t p 0 value
    )
  )
```

If `p` is not a substring of `t`, then `boyer_moore_horspool` returns -1, otherwise, it returns the first position in `t` where we can find `p`. To verify this, I checked 3 properties about the algorithm:

1. If, for `boyer_moore_horspool t p k i`, the last `k` indices in `p` and `t` are equal and the indices in this function are also equal, then the recursive call of the function is `boyer_moore_horspool t p (k + 1) i` and the last `(k + 1)` indices in `p` and `t` are equal (it is possible that the equality between the indices in `boyer_moore_horspool t p k i` to be false and the returned value to be the same as the output of `boyer_moore_horspool t p (k + 1) i`);

2. If, for all natural numbers `j` less than an arbitrary natural number `i` (which is less or equal than `length t - length p`), `boyer_moore_horspool t p 0 j` is not equal to `j`, then none of these `j` values are the positions where we can find the substring `p` in `t`;

3. If, for `boyer_moore_horspool t p k i`, `i` is incremented by `shiftbc` (or by 1), then for all values `i'` in the segment [`i`, `i + maximum shiftbc 1`), `i'` is not the position in `t` where the substring `p` is found(in other words, `belongs t p i' = false`). The function `belongs t p i` is false if there exists a natural number `j` in the segment [`i`, `i + length p`) such that `index t j <> index p (j - i)`. We know that none of the natural numbers in the segment [0, `i - 1`] are the position where we can find `p` in `t`, the last `k` characters in `p` are equal to characters in `t` and `t[i + m - 1 - k] <> p[m - 1 - k]`. Since `k` is reset to 0, not all of the characters in `p` match with characters in `t` and the first string is shifted. If `p` is shifted by 1, the only possible value of `i'` is `i`, where we already know that `belongs` statement is false. Otherwise,

we use the properties from `item_indices` and `update_bc`. Let us have `a = last (item_indices (index t (i + m - 1 - k)) alphabet 0)` and `b = index (update_bc alphabet p) a`. Since `b` is an element in the list `update_bc alphabet p`, its value is either -1 or a value in the segment $[0,$ `length p`$)$. In both cases, `a` is the index in `alphabet` where the bad character is stored (so its equal to `index t (i + m - 1 - k)`). If `b` is equal to -1, then no character in `p` is equal to the bad character. The shift is made to the position that is just after the position of the bad character. For all of the positions `i'`, there will be a mismatch. If `b` is not -1, then it is the last position of the bad character and for all values from the segment (`b`, `length p`), the bad character is not found. For all of the positions `i'`, the difference `i' - i` is in the segment $[0,$ `shiftbc`$)$, so the inequality `length p - 1 - k - (i' - i) > length p - 1 - k - shiftbc` is true. Since 1, `k` and `(i' - i)` are positive numbers, the result `length p - 1 - k - (i' - i)` is less than `length p`. The relation `length p - 1 - k - (i' - i)` can be written as `i + length p - 1 - k - i'`. This, combined with the property of `belongs` described above, proves that this case is valid.

3.7 Proving the Correctness of the Main Algorithm

These 3 statements helped me in verifying if the output data in the algorithm is correct for all cases.

```
let rec bmh_gives_correct_result
      (t:list english_letters)
      (p:list english_letters{length p <= length t})
      (k:nat{k <= length p})
      (i:nat{i <= length t - length p})
   : Lemma (requires
            (forall (i':nat{i' < i}). belongs t p i' = false)
            /\ (forall (k':nat{k' < k}).
            index t (i + length p - 1 - k') =
            index p (length p - 1 - k')))
            (ensures
          (let x = boyer_moore_horspool t p k i in
           x <> -1 ==> belongs t p x = true))
          (decreases %[(if i < length t - length p + 1 then
           length t - length p + 1 - i else 0);
           length p - k])
= let m = length p in
  if k = m then (
    assert(forall (k':nat{k' < m}).
           index t (i + m - 1 - k') = index p (m - 1 - k'));
    indices_are_equal_belongs_true_base t p i;
    convert_from_x_to_j p i
  )
  else (
   if index t (i + m - 1 - k) = index p (m - 1 - k) then (
     zero_to_k_then_all_indices_are_equal t p k i;
     bmh_gives_correct_result t p (k + 1) i
    )
    else (
      update_bc_length_is_initial_list_length alphabet p;
      mem_last_list (item_indices (index t (i + m - 1 - k))
                    alphabet 0);
      item_indices_is_in_interval_forall alphabet 0
          (index t (i + m - 1 - k));
      let shiftbc = m - k - 1 - index
         (update_bc alphabet p) (last (item_indices
         (index t (i + m - 1 - k)) alphabet 0)) in
      let value = i + (maximum 1 shiftbc) in
      if value > length t - m then ()
      else (
        bmh_belongs_false_if_less_than_shiftbc_forall
          t p k i;
        bmh_gives_correct_result t p 0 value
      )
    )
    )

let rec bmh_gives_minus_one_belongs_false
    (t:list english_letters)
    (p:list english_letters{length p <= length t})
    (k:nat{k <= length p})
    (i:nat{i <= length t - length p})
  : Lemma (requires
          (forall (i':nat{i' < i}). belongs t p i' = false)
          /\ (forall (k':nat{k' < k}).
          index t (i + length p - 1 - k') =
          index p (length p - 1 - k')))
          (ensures
          (let x = boyer_moore_horspool t p k i in
           x = -1 ==>
           (forall (y:nat{y <= length t - lenght p}).
           belongs t p y = false)))
          (decreases %[(if i < length t - length p + 1 then
           length t - length p + 1 - i else 0); length p - k])
  = let m = length p in
    if k = m then ()
    else (
     assert(k < m);
     if index t (i + m - 1 - k) = index p (m - 1 - k) then (
        zero_to_k_then_all_indices_are_equal t p k i;
        bmh_gives_minus_one_belongs_false t p (k + 1) i
      )
      else (
        update_bc_length_is_initial_list_length alphabet p;
        mem_last_list (item_indices (index t (i + m - 1 - k))
                       alphabet 0);
        item_indices_is_in_interval_forall alphabet 0
           (index t (i + m - 1 - k));
        let shiftbc = m - k - 1 - index
```

```
          (update_bc alphabet p) (last (item_indices
          (index t (i + m - 1 - k)) alphabet 0)) in
        let value = i + (maximum 1 shiftbc) in
        bmh_belongs_false_if_less_than_shiftbc_forall
          t p k i;
        if value > length t - m then ()
        else bmh_gives_minus_one_belongs_false
             t p 0 value
      )
    )
```

The proofs are for arbitrary parameters. For our algorithm to be correct, the input values are `text` for `t`, `pattern` for `p`, 0 for `k` and 0 for `i`. The above 2 lemmas are used to verify the properties for these inputs.

```
let bmh_gives_correct_result_for_text_and_pattern ()
  : Lemma (ensures
          (let x = boyer_moore_horspool text pattern 0 0 in
           x <> -1 ==> belongs text pattern x = true))
  = assert(0 <= length text - length pattern);
    bmh_gives_correct_result text pattern 0 0


let
bmh_gives_minus_one_belongs_false_for_text_and_pattern
() : Lemma (ensures
          (let x = boyer_moore_horspool text pattern 0 0 in
          x = -1 ==> (forall
          (y:nat{y <= length text -length pattern}).
          belongs text pattern y = false)))
  = bmh_gives_minus_one_belongs_false text pattern 0 0
```

With these proofs, the verification of Boyer-Moore-Horspool is done. The algorithm gives the desired output for any case.

Conclusions

There are several directions for further research on this subject. The bad character heuristic has various versions and can be optimized, giving a different way of solving the problem and the need of verifying if the new methods are correct and complete. The good suffix heuristic can also be used, independently of the bad character heuristic. I started to implement and verify the good suffix heuristic, but I didn't end the proof of it yet.

F* was very helpful for my thesis. The lemmas had a crucial role in making the proofs and the refinements helped me in setting certain restrictions and criteria on data types. Another useful element was the `assert` instruction, which I used to see if the language recognizes a proposition as being valid. One thing that F* did not have but would have been useful is a function that returns the index (or the indices) where an item is stored in a list and the proofs associated with this function.

It was interesting to learn a proof-oriented programming language because it was a new concept, different compared to what we did in general in the 3 years of faculty. It helped me in improving my critical and analytical thinking and seeing the problems in a different point of view. The proof of the main algorithm was the most interesting part, but at the same time the most complicated part. The proofs were more complex and had more statements and previous lemmas.

As statistics, the practical part of the thesis has 1198 lines of code, 112 lemmas (including 6 `move_requires` lemmas and 34 `forall_intro` lemmas), 85 `assert` instructions, 0 `assume` instructions, 0 `admit` instructions and an execution time of 67376 ms (if all of the code is in the same file).

Using F*, I proved that my implementation of the Boyer-Moore-Horspool is correct, meaning that, for any 2 strings `text` and `pattern` with characters in the same alphabet, it computes the first position in the `text` where we can find the `pattern`, or -1, if the `pattern` is not in the `text`.

Bibliography

[1] Ștefan Ciobâcă. Boyer-moore algorithm.

[2] Nikhil Swamy, Guido Martinez, and Aseem Rastogi. *Proof-Oriented Programming in F*.*

[3] https://fstarlang.github.io/docs/fstar.classical.fsti.html.

[4] https://github.com/fstarlang/fstar/blob/master/ulib/fstar.list.tot.base.fst.