Final Project - CS110. Factor Oracles.

I. Introduction

Factor oracles are a deterministic finite-state automaton which can detect substrings (factors) of a given string. Finite-state automaton means that we can represent the algorithm of finding the substrings as a system of states, each with conditions for transitions to other states. A simple example could be a coin-vending machine, which has two states: awaiting for a bill and giving out coins. Awaiting for a bill has a transition condition of bill inserted that would move it to the second state. In the second state, inserting a bill would transition back to itself, whereas removing the coins would reset the machine into the awaiting state.

In the case of a factor oracle, we put string indices as states and allocate one more for the initial state (0). The transitions are letters that the factor oracle receives. To simply put it, as we put in letters, we should arrive through a series of transitions at the right index of the substring occurance. Each state is connected with the consecutive state (since if two letters are neighbors in a string, they will form a substring). For each letter, we also find the letters that are next to it at any point later in the string. An example is $abacj$. In the automaton of factor oracles, a will be connected with b (as explained before), but it would also be connected with c, since if we query for a and then c, we know that we're talking about the a at position 2 (counting from 0).

Factor oracles are an exciting work-in-progress because it can be built incrementally in linear time and space complexity. Compared to older techniques for string matching, such as suffix trees, factor oracles are as efficient but require less memory and are easier to implement. Elaboration on oracle building time and space complexity will be provided in rubric 4 of Part II.

One important thing to note is that factor oracles are able to detect all of the string's factors (substrings) but it can also detect the ones that are not substrings, allowing room for false positives in certain applications. The index detected by factor oracle is also less or equal to the ending position of first occurance (again, related to the problem of falsely detecting substrings that are not there). Some of the examples will be shown in rubric 5 of Part II.

II. Implementation (factor oracle)

1. States in an oracle

As mentioned earlier, states of an oracle are indices of a given string p. For each state, we need to be able to look up trasition conditions as well as add them. In the $OracleState$ class below, $supply\_fn$ (supply function) is a function that maps state i of the oracle to state j where the first occurrence of $repet_p(i)$ (longest suffix of prefix i of string p that repeats at least twice in the prefix). It can be shown that supply function will always

give a smaller value than i (which is obvious because if a suffix appeared before, then the ending positing of the first occurrence of that factor will be before the last ending where the suffix is). We can note that for $OracleState.transitions$, we use a dictionary which is a hash map that will allow us $O(1)$ transition time and $O(1)$ space taken by each state (this will be proven in rubric 4).

In [415]:
```
class OracleState:
    idx = 0
    transitions = {}
    supply_fn = 0

    def __init__(self, idx=0):
        self.idx = idx
        self.transitions = {}
        self.supply_fn = -1

    def add_transition(self, by_letter, to_idx):
        self.transitions[by_letter] = to_idx

    def transition_by(self, by_letter):
        return self.transitions[by_letter]
```

2. Oracle operations, automaton, and helper-functions

Main operations:

1. $online\_init()$ is a way to build a factor oracle "online," which means to build in an incremental fashion as we receive new letters. In this case, this method takes in a string as a whole and feeds one character after another to $add\_letter()$.
2. $add\_letter()$ takes in a new letter and builds a factor oracle from previous accumulated oracle an adding in the next character. It first connects the previous last state to the current last state since they are consecutive. Then, based on implications from Lemmas 9, 10 from Allauzen et al. (1999), we can go down the suffix path and for each of the states on the "suffix path" (a chain of supply functions starting from m, then $supply\_fn(m)$, then $supply\_fn(supply\_fn(m))$ until when the value of is 0), connect them to the last inserted state with transition condition of the last inserted letter, since letters lying on the suffix path carry the same as the suffix of the whole string p. We know that any suffix of p is connected to the newly inserted state; thus, the the factors lying on the suffix path (share the same suffix), will also be connected to the newly inserted state, as explained in Introduction.

Helper operations:

1. $traverse\_oracle()$ simulates the work of the automaton by feeding it character-by-character and returning the last state the automaton stops at.
2. $suff\_repet()$ (not implemented) describes more details about $repet_p(i)$ mentioned in rubric 1.

3. $first\_occur()$ returns the ending position of the first occurrence of a substring in constant time with a chance of false positive, as mentioned in Introduction and will be demonstrated in rubric 5.

In [ ]:

```
class FactorOracle:
    inp_str = ''
    oracle_sz = 0
    states = []

    def __init__(self, inp_str):
        self.reset()

        self.inp_str = " " + inp_str
        self.oracle_sz = len(self.inp_str)
        self.__online_init()

    def reset(self):
        self.inp_str = ''
        self.oracle_sz = 0
        self.states = []

    def __online_init(self):
        self.states.append(OracleState())

        for i in range(1, self.oracle_sz):
            self.__add_letter(self.inp_str[i])

    def __add_letter(self, new_letter):
        new_last_idx = len(self.states)
        self.states.append(OracleState(idx=new_last_idx)) # create state m + 1
        self.states[-2].add_transition(to_idx=new_last_idx, by_letter=new_letter) # transition from state m to m

        cur_idx = self.states[-2].supply_fn # supply_fn(m)
        # while we still have the suffixes to connect to the last inserted element
        while cur_idx > -1 and (new_letter not in self.states[cur_idx].transitions):
            self.states[cur_idx].add_transition(to_idx=new_last_idx, by_letter=new_letter)
            cur_idx = self.states[cur_idx].supply_fn

        if cur_idx == -1: # repet(i) is zero
            next_idx = 0
        else:
            next_idx = self.states[cur_idx].transition_by(new_letter)
        self.states[-1].supply_fn = next_idx

    def __str__(self):
```

```
        out = "Factor Oracle of \"{0}\":\n".format(self.inp_str[1:])
        for i in range(self.oracle_sz):
            st = self.states[i]
            out += "State " +  str(i) \
                + " Supply function " + str(st.supply_fn) \
                + " Transitions " + str(st.transitions) + '\n'
        return out

    '''
    Running the deterministic automaton and getting the output
    '''
    def traverse_oracle(self, query_str):
        cur_idx = 0 # starting with state 0
        for char in query_str:
            if char in self.state[cur_idx].transitions:
                cur_idx = self.state[cur_idx].transitions[char]
            else:
                return -1 # not found
        return cur_idx

    '''
    Function repet(i), i is a state in Oracle(int_str),
    returns the longest suffix of prefix(i) in inp_str that appears at least twice
    '''
    def suff_repet(self, orc_idx):
        prefix = self.inp_str[:orc_idx]
        pass

    '''
    Function poccur(u), u ∈ Fact(int_str),
    returns ending position of first occurrence of factor u in int_str
    '''
    def first_occur(self, factor): # poccur(u), u ∈ Fact(p)
        poccur = traverse_oracle(factor)
        if poccur == -1:
            return "Not Found"
        return poccur
```

3. Testing oracles

In [519]:
```
a = FactorOracle("abbbaab")
print(a)

b = FactorOracle("abbcabc")
print(b)

c = FactorOracle("abcjiobeamf")
print(c)

d = FactorOracle("abb")
print(d)
```

```
Factor Oracle of "abbbaab":
State 0 Supply function -1 Transitions {'a': 1, 'b': 2}
State 1 Supply function 0 Transitions {'b': 2, 'a': 6}
State 2 Supply function 0 Transitions {'b': 3, 'a': 5}
State 3 Supply function 2 Transitions {'b': 4, 'a': 5}
State 4 Supply function 3 Transitions {'a': 5}
State 5 Supply function 1 Transitions {'a': 6}
State 6 Supply function 1 Transitions {'b': 7}
State 7 Supply function 2 Transitions {}

Factor Oracle of "abbcabc":
State 0 Supply function -1 Transitions {'a': 1, 'b': 2, 'c': 4}
State 1 Supply function 0 Transitions {'b': 2}
State 2 Supply function 0 Transitions {'b': 3, 'c': 4}
State 3 Supply function 2 Transitions {'c': 4}
State 4 Supply function 0 Transitions {'a': 5}
State 5 Supply function 1 Transitions {'b': 6}
State 6 Supply function 2 Transitions {'c': 7}
State 7 Supply function 4 Transitions {}

Factor Oracle of "abcjiobeamf":
State 0 Supply function -1 Transitions {'a': 1, 'b': 2, 'c': 3, 'j': 4, 'i': 5, 'o': 6, 'e': 8, 'm': 10, 'f':
11}
State 1 Supply function 0 Transitions {'b': 2, 'm': 10}
State 2 Supply function 0 Transitions {'c': 3, 'e': 8}
State 3 Supply function 0 Transitions {'j': 4}
State 4 Supply function 0 Transitions {'i': 5}
State 5 Supply function 0 Transitions {'o': 6}
State 6 Supply function 0 Transitions {'b': 7}
```

```
State 7 Supply function 2 Transitions {'e': 8}
State 8 Supply function 0 Transitions {'a': 9}
State 9 Supply function 1 Transitions {'m': 10}
State 10 Supply function 0 Transitions {'f': 11}
State 11 Supply function 0 Transitions {}

Factor Oracle of "abb":
State 0 Supply function -1 Transitions {'a': 1, 'b': 2}
State 1 Supply function 0 Transitions {'b': 2}
State 2 Supply function 0 Transitions {'b': 3}
State 3 Supply function 2 Transitions {}
```

To make it clearer, the first factor oracle of $abbbaab$ would look like this:

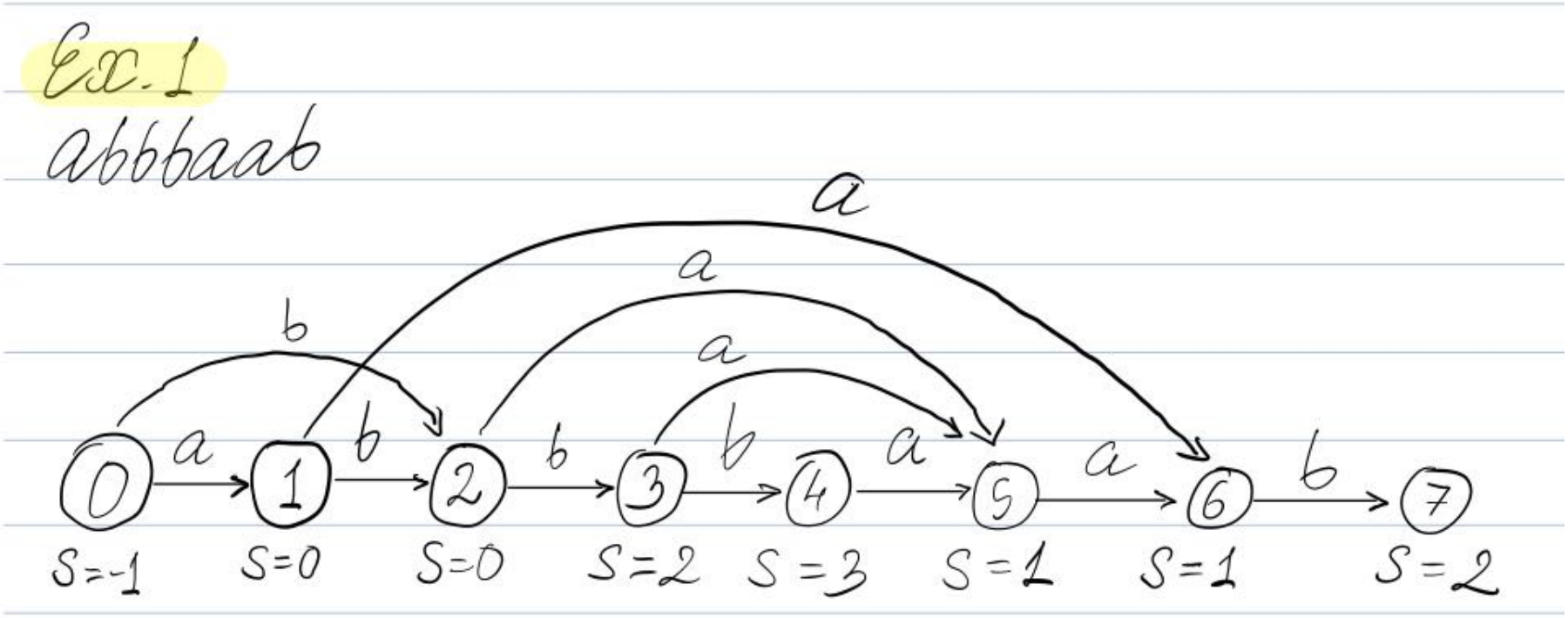

Figure 1. Visualization of factor oracle on string "abbbaab". Letters s under each state indicate the supply function value (exactly defined) of that state.

4. Complexity analysis

First and foremost, we will introduce Lemma 6 from Allauzen et al. (1999), which states that the number of transitions we have in the graph is no more than $2m - 1$ where m is the size of the word we are build factor oracle from.

Taking a look at the method $add\_letter()$, we see that there is a while loop that goes down the suffix path from the last state. At first glance, we can say that the time complexity for this loop is $O(m)$ because the states in suffix path are strictly decreasing as explained in rubric 2. However, the time complexity is actually $O(1)$ because if it were scaling with m, we would be able to grab larger suffixes, proportionally to m, meaning that we would have an $O(m)/O(m)$ steps in the suffix path, which is $O(1)$.

Knowing that $add\_letter()$ runs at $O(1)$ and looking at $online\_init()$, we can see a loop that runs in $O(m)$. Therefore, we can confirm that the building time of a factor oracle is indeed $O(m)$.

From a space perspective, since we have no more than $2m - 1$ transitions and exactly m states, the total space consumed by a factor oracle is $O(m)$.

5. Odd cases of factor oracles

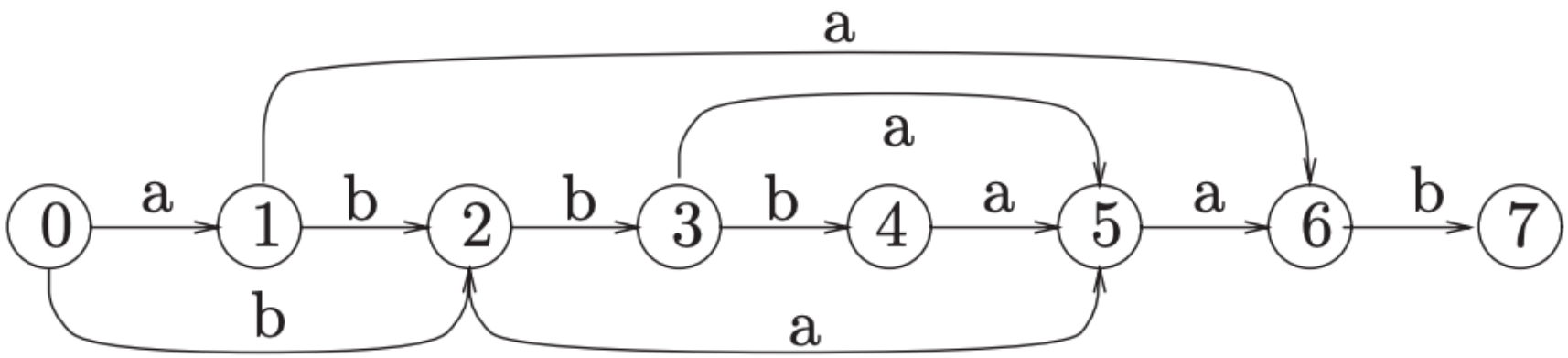

Figure 2. Factor aba is detected in the oracle, but it is actually not a substring of the string. Figured adapted from Allauzen et al. (1999).

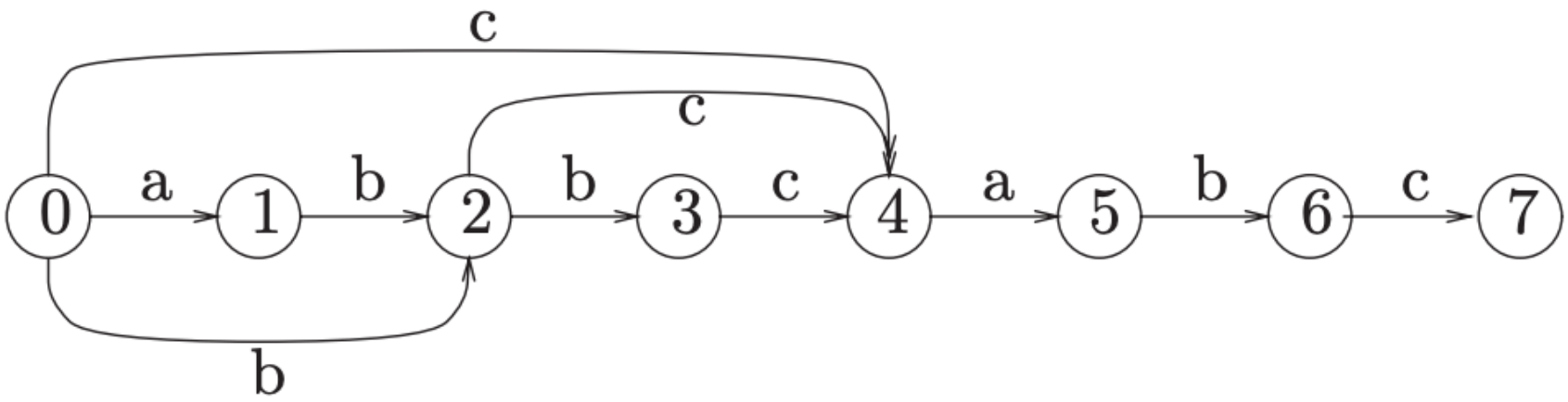

Figure 2. Factor abc is detected at ending position 4, whereas its actual ending position is 7. Figured adapted from Allauzen et al. (1999).

III. Applying factor oracles in practice

1. BOM algorithm

Backward oracle matching is

In [647]:
```
def backward_oracle_matching(word, text):
    fo = FactorOracle(word[::-1]) # initializing factor oracle on mirror image of word
    w_len = len(word)
    t_len = len(text)
    matched_pos = []

    cur_pos = 0 # beginning of the comparison window
    while cur_pos <= t_len - w_len: # while not last window position
        cur_idx = 0
        right_pt = w_len - 1
        # we run the characters through the reversed oracle and note when it's a mismatch
        while text[cur_pos + right_pt] in fo.states[cur_idx].transitions:
            cur_idx = fo.states[cur_idx].transition_by(text[cur_pos + right_pt])
            right_pt -= 1

        # if we run till the end, we have all the characters in the window matched
        if right_pt == -1:
            matched_pos.append(cur_pos)
            right_pt = 1
        cur_pos += right_pt + 1

    return matched_pos
```

In [646]:
```
backward_oracle_matching(word="cd", text="abfecd")
```

Out[646]:
```
[4]
```

2. Testing efficiency on real DNA sequences

We will test against Python's standard substring positions finding function which supposedly runs in $O(mn)$ time. Our algorithm runs on average in $O(nlog_{|\sum|}(m)/m)$ where $\sum$ is the alphabet (Conjecture 1 in Allauzen et al. (1999)).

Disclaimer: Python's finding function runs faster probably because the loops are implemented on C :)

In [658]:
```
import time

chr1_file = open("sequence.fasta", "r")
seq = chr1_file.read()
hyp_gene = "ATCGTGAGGCCAT" # hypothetical gene

start = time.time()
seq.count(hyp_gene)
end = time.time()
print("Python gene matching time:", end - start, "s")

start = time.time()
backward_oracle_matching(hyp_gene, seq)
end = time.time()
print("BOM gene matching time:", end - start, "s")
```

```
Python gene matching time: 0.6311228275299072 s
BOM gene matching time: 33.142821073532104 s
```

3. Turbo-BOM (BOM combined with KMP)

Allauzen et al. (1999) also mention the modified version of BOM that, instead of having the worst case performace of $O(mn)$, can actually be linear in the worst case. To do so, they apply KMP (Knuth-Morris-Pratt) algorithm to make the movement of the window faster without rechecking the values.

IV. Conclusion and discussion

In this paper, we have gone through the specifics, benefits, and downsides of factor oracles. We have observed how it can be a promising technology for fast string matching in big texts (such as genome sequences). There are more works to be done, such as the analysis of the false positive cases and how to avoid them.

Appendix

1. Bibliography

Allauzen, C., Crochemore, M., & Raffinot, M. (1999). Factor Oracle: A New Structure for Pattern Matching. Retrieved from https://www.cs.upc.edu/~marias/teaching/bom.pdf (https://www.cs.upc.edu/~marias/teaching/bom.pdf)

Crochemore, M., Ilie, L., & Seid-Hilmi, E. (n.d.). The structure of Factor Oracles. Retrieved from https://hal-upec-upem.archives-ouvertes.fr/hal-00619689/document (https://hal-upec-upem.archives-ouvertes.fr/hal-00619689/document)

G. (2009). Homo sapiens chromosome 1, GRCh37 primary reference assembly. Retrieved from https://www.ncbi.nlm.nih.gov/nuccore/CM000663.1?report=fasta (https://www.ncbi.nlm.nih.gov/nuccore/CM000663.1?report=fasta)

2. HC Applications

#algorithms: I decomposed the oracle building algorithm and assembled the full working building algorithm with the same complexity from the pseudocode provided in the research paper.
#critique: I determined what the potential pitfalls of factor oracles are and demonstrated them in clear examples.
#scienceoflearning: I applied the technique called "active recall" by reading the paper and then, not looking back, explaining to myself the lemmas that I retained. This way, I was able to understand the mechanics of factor oracles in a deeper way and that has helped me implement this idea.

In [ ]: