PSET 5

Finite Automata

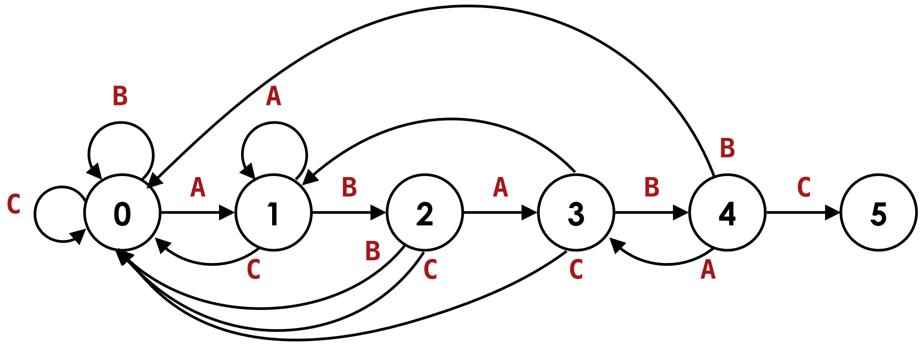
In this problem, you will learn to implement a finite automata in the Knuth-Morris Pratt string matching algorithm.

Section A: How does a finite automata work?

Before going into how to build the finite automata, we will first look at how a finite automata works. A finite automata is essentially a topological graph which guides match and mismatch transitions in the KMP algorithm. To demonstrate this, we will use the following substrings and texts respectively:

```
substring: "ABABC"
text: "ABABABAABABC"
```

The above substring produces the following finite automata:



- **vertices (state number):** how many characters we've matched so far
- **edges (next transition):** state transition depending on next character

The above can also be represented as a 2 dimensional list like so:

`dfa`

	0	1	2	3	4	5
A	1	1	3	1	3	0
B	0	2	0	4	0	0
C	0	0	0	0	5	0

Performing the KMP algorithm using the finite automata

Imagine we have matched up to the following point, this would put us currently in state 3:

A	B	A	A	A	B	A	A	B	A	B	C

All we need to do now is look at the next character, which is A, refer back to the DFA and determine the transition. In this case, **`dfa['A'][3]`** is 1, so we would go back to **state 1**.

Section B: Building the Finite Automata

Take note that for this assignment, we will assume that there are at most 256 unique characters in any substring (ASCII characters).

Step 1:

Initialise a 2D list containing **zeros** with rows equals to number of characters (256)

`pattern`

`"ABABC"`

`dfa`

	0	1	2	3	4	5
...	0	0	0	0	0	0
65	0	0	0	0	0	0
66	0	0	0	0	0	0
67	0	0	0	0	0	0
...	0	0	0	0	0	0

Note:

- **65** is the ASCII code for `'A'`, **66** for `'B'` and **67** for `'C'`, hence, the row index corresponds to the character through ASCII encoding.
- The value of **`dfa[i][j]`** essentially just represents a **match / mismatch transition** to the next state!

Step 2 - Match Transitions:

Since we already know the pattern, we can fill up the **match transitions**

Example:

If we are at state 0 (zero matches), and we encounter `'A'`, the first char in our substring, we transition to state 1, so **`dfa['A'][0]`** will be 1

`dfa`

	0	1	2	3	4	5
...	0	0	0	0	0	0
65	**1**	0	**3**	0	0	0
66	0	**2**	0	**4**	0	0
67	0	0	0	0	**5**	0
...	0	0	0	0	0	0

`pattern`

`"ABABC"`

Step 3 - Mismatch Transitions:

This is the most challenging step in the building of the DFA. However, once you understand the concept, it actually becomes rather simple!

The main idea is that if we are to mismatch at a certain character, we know that with KMP, we want to record the longest suffix of our chars matched which matches a prefix of our pattern:

A	B	A	B	A	B	A	A	B	A	B	C

For instance, if we have a mismatch at the `'A'` highlighted in yellow, we know that the last three characters (suffix) `'ABA'` is a match to a prefix of our substring, and so we would **transition to state 3**.

Now, back to our DFA, if we want to fill up some mismatch transition **`dfa[i][j]`**, all we do is:

A. **simulate** the dfa for `pattern[1:j]`

B. **simulate** the transition on **char i** to get our next state

We simulate index 1 to j because we already know from index 0, it would lead to a mismatch and we now want a **suffix**

Example 1: Let's try to fill up **`dfa['A'][1]`**

A. We simulate our dfa for `pattern[1:1] = ""`, and so we reach **state 0**.

B. Then we perform a transition on the character `'A'` **(`dfa['A'][0]`)**, which moves us to **state 1**, so **`dfa['A'][1] = 1`**

`pattern`

`"ABABC"`

`dfa`

	0	1	2	3	4	5
...	0	0	0	0	0	0
65	1	1	3	0	0	0
66	0	2	0	4	0	0
67	0	0	0	0	5	0
...	0	0	0	0	0	0

Example 2: Let's try to fill up **`dfa['A'][4]`**

A. We simulate our dfa for **`pattern[1:4] = "BAB"`**, and so we reach **state 2**.

B. Then we perform a transition on the character `'A'` **(`dfa['A'][2]`)**, which moves us **state 3**, so **`dfa['A'][2] = 3`**

pattern

"ABABC"

dfa

	0	1	2	3	4	5
...	0	0	0	0	0	0
65	**1**	**1**	**3**	0	**3**	0
66	0	**2**	0	**4**	0	0
67	0	0	0	0	**5**	0
...	0	0	0	0	0	0

Now, just do that for every mismatch transition and your dfa will be filled up!

Note: As you can see, filling mismatch transitions relies on simulating the dfa on previous states, so you should fill up the dfa column by column (state by state)

A. Starting Code

`KnuthMorrisPratt`

- The KMP algorithm using the finite automata has been implemented to help you test the functionality of your dfa

`main`

- main has been implemented for you to test your dfa & KMP algorithm
- It asks the user for a text and substring, and prints out all indices in which the substring is matched in the text, assuming the dfa function has been implemented

B. Your Task

Implement the following functions:

1. `buildDFA`

- This function takes in a **substring** and builds the appropriate finite automata for the KMP string matching algorithm
- It should return the DFA as a **2D list**, with 256 rows (number of ASCII characters) and M (length of substring) + 1 columns, which is the number of states
- You may assume that there are only 256 characters possible in this dfa
- All values in the last column (final state) should be 0
- All values in rows of chars not in the substring should be 0

Example:
`buildDFA("ABABAC")` should return

	0	1	2	3	4	5
...	0	0	0	0	0	0
65	1	1	3	1	3	0
66	0	2	0	4	0	0
67	0	0	0	0	5	0
...	0	0	0	0	0	0

Note:

- To convert a character to it's ASCII number, you can use the ord() function in python, and to convert from it, you can use the chr() function

```
ord('A') # convert char to ascii number: 65
chr(65)  # convert ascii number to char: 'A'
```

- The number of unique characters is given in a constant variable. Do make use of this to initialise your 2D list!

```
NO_OF_CHARS = 256
```

C. Sample Output

```
python dfa.py
text: BANANA
substring: ANA
substring found at index 1
substring found at index 3
```

D. Submission

To test your code, run the following command:

```
python utils/test.py
```

To submit your code, run the following command:

```
python utils/submit.py
```

A report.txt should be generated for you to view your results