Programming Project 2: Pattern Matching

CS 181, Fall 2025

Out: Oct. 14
Due: Oct. 27, 11:59 PM

1 Task

In this assignment, you will implement the Knuth-Morris-Pratt algorithm as described in lecture. You will also implement an algorithm that constructs a deterministic finite automaton that accepts a given input string.

2 Program Specifications

2.1 Programming Language

Our Gradescope autograder is currently configured to accept solutions written in Python 3, R, and Java.

Furthermore, in addition to the usual permitted packages, students using Python will also have access to the Python library graphviz in this project to help with writing dot files. Check the graphviz website and user guide for more information regarding its installation and usage.

We recommend doing this project in your Docker container. If you would like to program in your container, navigate into your cs181-projects23 folder and run git pull. A new folder called 2_patternmatching should appear with all of the necessary stencil for this project. Your container also already has graphviz installed!

We will also attach the stencil files on the website. If you would rather program in your own virtual environment, running `pip install graphviz` or `pip3 install graphviz` should do the trick. If this doesn't work, some students on Mac have used the commands '`(sudo) port install graphviz` or `brew install graphviz` to install graphviz using Macports or homebrew respectively.

3 Knuth-Morris-Pratt Specifications

3.1 Input Format

Your Knuth-Morris-Pratt algorithm should take one argument, a path (relative or absolute) to a file containing (1) the text to be searched and (2) the pattern to be found within this text.

The first line of the input file specifies the text, and the second line specifies the pattern. For example, here are the contents of `test_cases_kmp/01.txt`:

```
ACGTGTGACTGAGAGTTGACTGAGTGGGGAGCGATGACTGATC
```

```
TGACTGAGAG
```

While parsing inputs, please be aware that these files may have newlines at the end of line 2.

3.2 Output Format

Your algorithm should return all indices at which the pattern is found. If the pattern is not contained in the text, your algorithm should return -1.

Here is an example illustrating how we expect to be able to call your shell scripts and what we expect as output.

```
> sh kmp.sh test_cases_kmp/01.txt
[5]

> sh kmp.sh test_cases_kmp/05.txt
[4, 6]
```

4 Deterministic Finite Automaton Specifications

4.1 Input Format

This script should take two arguments:

- a path (relative or absolute) to a file with a sequence for which we wish to construct a DFA
- a name for a dot output file (filename must end in '.dot')

For example, here are the contents of `test_cases_dfa/01.txt`:

```
CCT
```

and `test_cases_dfa/02.txt`:

```
ACCATATACCATCC
```

4.2 Output Format

The output for your algorithm should be a diagram in DOT format, like the one shown below (DOT file for 01.txt)

```
digraph dfa {
    0->1 [ label ="C"]
    1->2 [ label ="C"]
    2->3 [ label ="T"]
    3 [peripheries = 2]
    2->2 [ label ="C"]
}
```

Please do use the *digraph* command as above; our autograders will not accept the *graph* subcommand *edge [dir=forward].*

In DOT format, an edge with an arrow from one node to another is indicated by −>. To build the DFA, you will need to make edges from node to node as new characters are read from the input file. Additionally, you will need to draw edges back to previous nodes in the case that a mismatch is encountered (unless a mismatch takes you back to state 0, in which case you can just assume this transition and omit it from the graph). All edges must be labeled with the character (A, T, C, or G) that they correspond to. The node for the final state must be drawn with a second circle around it (see above, as done for node 3).

Here are a few examples illustrating how we expect to be able to call your shell scripts and what we expect as output.

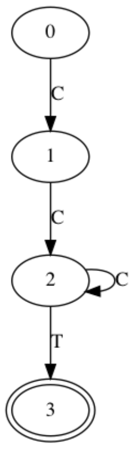
```
> sh dfa.sh test_cases_dfa/01.txt 01.dot
```

If you would like to visualize the DFA from the DOT file, you can run the following command:

```
> dot -Tpng (input_fle_name).dot -o (output_file_name).png
```

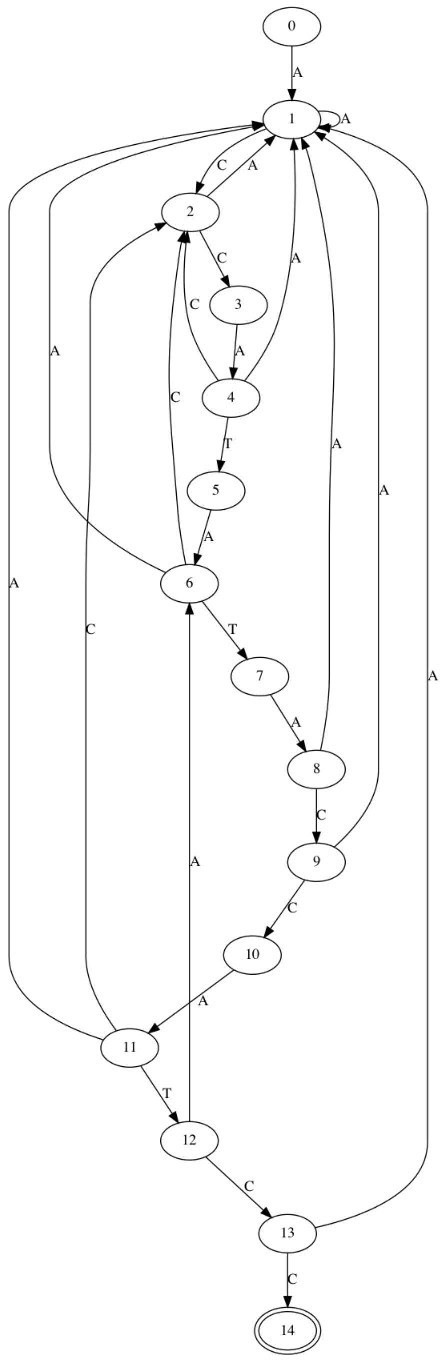
Alternatively, you can use this online Graphviz visualizer.

Let's take the DOT file we showed earlier. Using the command or website above, you should get the following graph:



Here is another example:

```
> sh dfa.sh test_cases_dfa/02.txt 02.dot
> dot -Tpng 02.dot -o 02.png
```



5 Application

COVID-19 is caused by the virus SARS-CoV-2, which is often referred to as simply “the coronavirus”. However, SARS-CoV-2 is actually only one of many different coronaviruses that exist, and several of these coronaviruses have previously caused other prominent diseases in humans (e.g. SARS, MERS). These viruses are grouped together mainly based on their characteristic appearance, which includes a spiky exterior somewhat resembling a crown (hence “corona”, which is Latin for “crown” or “wreath”). On the genomic level, coronaviruses also share specific motifs called transcription regulatory sequences (TRS) that activate subgenomic mRNA transcription in the cytoplasm. Coronavirus TRS can be divided into three sequence blocks: one “core sequence” (CS) and two flanking sequences (5’ and 3’). Coronaviruses are generally split into three main groups, each of which has a distinct consensus CS:

- Group 1 coronavirus strains most frequently have the hexamer core sequence 5’-CTAAAC-3’
- Group 2 coronavirus strains most frequently have either the heptameric sequence 5’-TCTAAAC-3’ or the hexamer sequence 5’-ACGAAC-3’.
- Group 3 coronavirus strains most frequently have the octamer core sequence 5’-CTTAACAA-3’.

Based on the sequences of CS motifs found in coronavirus groups 1, 2, and 3, apply your Knuth-Morris-Pratt algorithm to compare the frequencies of the above motifs in the SARS-CoV, MERS, and SARS-CoV-2 genomes. We have provided you with all three of these reference genomes in the `data` directory. (**Note:** consider similarities between motifs!)

Please report and submit the following in a PDF file called application.pdf:

- A reproducible description of your workflow. You should provide enough detail for us to trace your steps exactly from start to finish. Please tell us if you wrote any new scripts, created or edited input files, or saved outputs to a file. If you run any commands in the shell, document these. The point is that it should be clear exactly how you generate outputs from data.
- The coronavirus group (1, 2, or 3) that the given genomes are most likely a part of, based on your findings.
- Explain your reasoning. In order to receive full credit, make sure to support your conclusions with data regarding the relative frequencies of these motifs within and amongst the given genome sequences.

6 README

You must include a README file. Include the following information:

- A description of any known bugs
- Anything you want the TAs to know about your project

7 Handin

- To hand in this assignment, upload your shell scripts along with all files needed to run your code on Gradescope. If you have any subdirectories or folders that you would like to preserve in your handin, you will need to compress your submission into a zip file and upload the zip file to Gradescope. However, to ensure that the autograder runs your handin properly, make sure that all shell scripts are present at the root of your handin; in other words, do not place your shell scripts inside any folders.
- When you hand in your solution, our autograder will immediately run on your submission. We have made the test cases provided in the project handout immediately visible to you so that you can ensure your solution runs properly with the autograder. If you fail the autograder and cannot determine why, notify a TA and we will help you to diagnose the problem. If your final submission is not compatible with our autograder, it will be very difficult for us to give you credit for this assignment.

8 Grading

We will grade your handin using pre-generated test cases with a certain number of points allocated per test case. Test edge cases extensively! Handins should not throw exceptions on any valid inputs or edge cases.