Q1 Implementation Idea

Threads are represented separately as two classes, class Normal and class Optimistic. Optimistic inherits Normal. They are bundled together into an ArrayList<Normal> threads, with index 1 being the normal thread and the rest optimistic. All threads interact with this threads to either read or write as needed with synchronization.

The basic idea I used is that every thread has fields

DFAState InitialState, Optional<Boolean> keepEnd

each protected by a monitor lock.

In general, an optimistic thread first attempt to see if its ending state converges no matter the initial state, if so it passes its ending state as the initial state to the next thread.

It then waits (and needs to be notified) until the previous thread passes to it its initial state. Once it knows its initial state, it computes whether the previous thread should keep end, and updates accordingly.

Note: In the special case where the current thread does not know either whether the previous thread should keep end, it waits for an answer to propagate back from the bottom of the list since the last thread never keeps end. For example, with input 1447 and 3 Optimistic Threads:

```
<terminated> q1 [Java Application] C:\Program Files\Java\jdk-14.0.2\bin\javaw.exe  (Mar. 30, 2021, 2:58:13 p.m. – 2:58:15 p.m.)
[Console output redirected to file:C:\Users\erres\OneDrive\Desktop\sequential.log]
1447

1
4
4
7
Thread ID: 2 waiting
Thread ID: 2 awoken
Thread ID: 3 waiting
Thread ID: 3 awoken
Optimisitc Thread 2 successfully terminates.
Optimisitc Thread 1 successfully terminates.
Optimisitc Thread 3 successfully terminates.
Normal Thread Successfully terminate.
_
_
_
_
Total run time 2 ms
```

Once a thread knows both the initial state and whether to keep end, it can compute its task and terminate.

Q1 Sample Tests

The following sample input is produced by a seed of 9 and has a length of 300000. When ran sequentially, the run time is above 100 ms. The whole string to too long to show on console, but the sequential execution by one thread is attached, along with other files, in a separate sequential.log.

The average sequential run time is 120.8 ms over 10 tests on a 4-core machine.

The average run time for 2 threads (1 normal, 1 optimistic) is 289 ms over 10 tests.

The average run time for 3 threads is 302 ms over 10 tests.

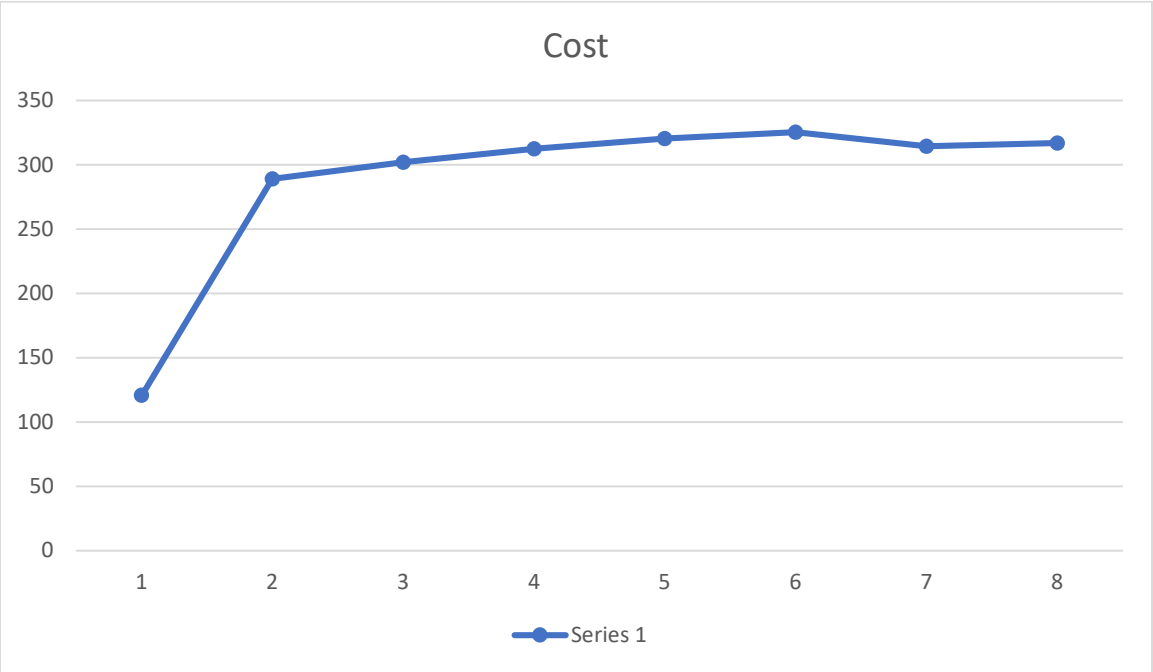
The average run time for 4 threads is 312.5 ms over 10 tests.

The average run time for 5 threads is 320.5 ms over 10 tests.

The average run time for 6 threads is 325.5 ms over 10 tests.

The average run time for 7 threads is 314.5 ms over 10 tests.

The average run time for 8 threads is 317 ms over 10 tests.



Q1 Conclusion

It seems that adding more threads don't achieve speedup, which is not very surprising given that pattern matching is sort of sequential in nature. The only concurrent part comes only as a result of converging ends states, which allows a thread to go ahead with its task individually. However, that 'going ahead of time' comes at the cost of thread coordination. For a 4-core machine, it seems that 7 or 8 threads is the right spot for multithreading.