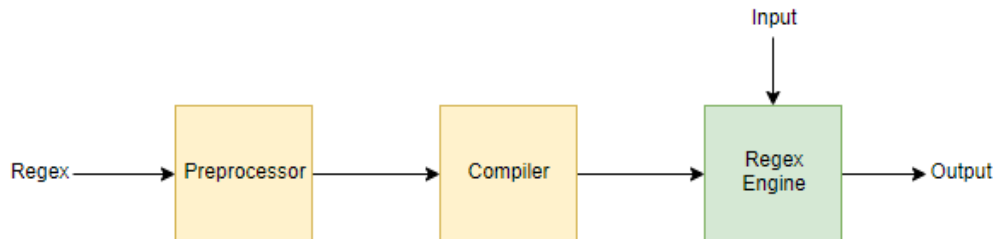


정규표현식의 엔진과 성능

정규표현식이란?

정규표현식은 특정 문자열 규칙을 가진 문자열의 집합을 찾는 언어이다.

정규표현식의 동작



모든 정규표현식은 그 정규표현식에 맞는 **Finite Automaton(FA)**로 만들 수 있다.

정규표현식을 컴파일한다는 것은 원래 정규표현식을 정규표현식 엔진이 쉽게 실행할 수 있는 **Finite Automaton**으로 변환한다는 것이다.

몇몇 구현에서는 **Preprocessor**를 이용하기도 하는데, 이는 $\{\alpha\}$ 와 같은 것을 $[a-zA-Z]$ 로 변환시켜주는 것과 같은 일들을 한다.

1. 컴파일

정규식 패턴을 **정규식 패턴 객체**로 변환하는 과정이 필요하다.

```
import java.util.regex.*;

public class RegexExample {
    public static void main(String[] args) {
        Pattern p = Pattern.compile(".s"); // .은 단일 문자를 나타냄
        Matcher m = p.matcher("as");
        boolean b = m.matches();
        boolean b2 = Pattern.compile(".s").matcher("as").matches();
        boolean b3 = Pattern.matches(".s", "as");
        System.out.println(b+" "+b2+" "+b3);
    }
}
```

위와 같이 패턴을 먼저 컴파일하여 패턴을 만들고 특정 문자열이 해당 패턴에 맞는지를 검사한다.

2. 매칭

컴파일된 패턴을 문자열에 적용시켜야 한다. 이 매칭이라는 작업은 **정규표현식의 엔진**이 시행한다.

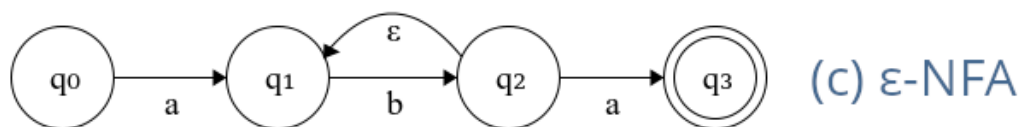
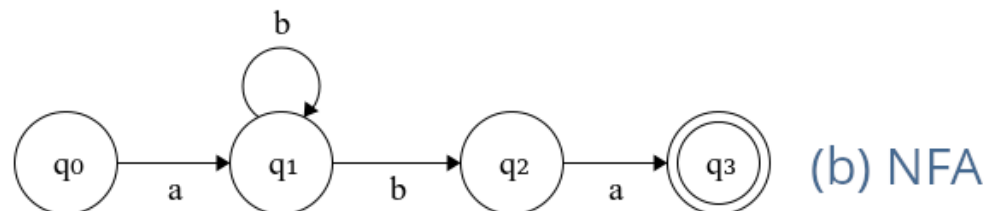
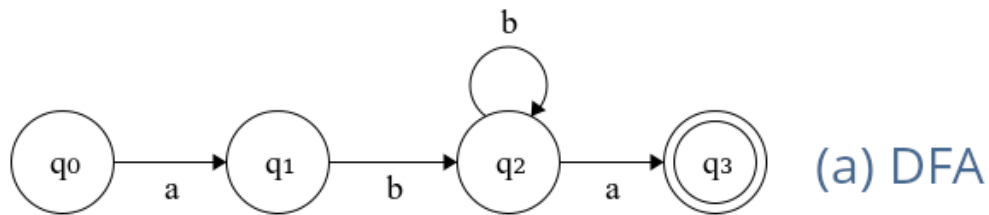
정규표현식의 엔진

위에서 정규표현식은 그에 맞는 **Finite Automaton**이 있다고 했다. 정규표현식의 엔진은 사실 많은 수가 **Finite Automata**로 구현되어 있다.

Engine type	Programs
DFA	awk (most versions), <i>egrep</i> (most versions), <i>flex</i> , <i>lex</i> , MySQL, Procmail
Traditional NFA	GNU Emacs, Java, <i>grep</i> (most versions), <i>less</i> , <i>more</i> , .NET languages, PCRE library, Perl, PHP (all three regex suites), Python, Ruby, sed (most versions), <i>vi</i>
POSIX NFA	<i>mawk</i> , Mortice Kern Systems' utilities, GNU Emacs (when requested)
Hybrid NFA/DFA	GNU awk, GNU <i>grep/egrep</i> , Tcl

각 언어의 정규표현식 구현 방법

오토마타의 종류는 크게 두 가지로 나뉜다.



Deterministic Finite Automaton (DFA)

상태와 input값이 주어지면, 그에 해당하는 상태 전이의 종류는 단 한가지다.

위의 예에서 **abba**가 input으로 주어지면, $q_0 \rightarrow q_1 \rightarrow q_2 \rightarrow q_2 \rightarrow q_3$ 의 상태 전이 하나 뿐이다.

DFA를 사용하는 엔진은 **Backtracking**을 하지 않기 때문에 **선형 시간 안에 동작한다**.

또한 **가능한 가장 긴 문자열을 매칭시킨다**.

다만 위에서 말했듯이 DFA의 상태 전이는 단 하나가 되기 때문에 아래와 같은 **Backreference**를 이용하는 패턴은 사용할 수 없다.

Backreference (역참조)

```
using System;
using System.Text.RegularExpressions;
public class Example{
    public static void Main(){
        string pattern = @"(\w)\1";
        string input = "trellis llama webbing dresser swagger";
        foreach (Match match in Regex.Matches(input, pattern))
            Console.WriteLine("Found '{0}' at position {1}.",
                match.Value, match.Index);
    }
}
// The example displays the following output:

//      Found 'll' at position 3.

//      Found 'll' at position 8.
```

```
//      Found 'bb' at position 16.

//      Found 'ss' at position 25.

//      Found 'gg' at position 33.
```

Backreference를 사용한다는 것은 앞서 일치한 부분을 다시 찾는다는 것이다.

위의 예에서는 (w)로 문자 하나를 찾는 그룹을 만들고, 그 후에 \1로 1번 그룹을 다시 찾겠다는 것이다.

결과적으로 문자 두개가 연이어 나타나는 패턴을 찾게 된다.

Nondeterministic Finite Automaton (NFA)

상태와 input값이 주어지면, 그에 해당하는 상태 전이는 여러개가 있을 수 있다.

위의 예에서 **abb**가 input으로 주어지면, q0 -> q1 -> q1-> q1 와 q0 -> q1 -> q1 -> q2 두 가지 상태 전이 결과가 있을 수 있다.

Traditional NFA

Traditional NFA는 greedy하게 모든 가능한 경우를 **Backtracking**하여 매칭시키고 매칭되는 것을 찾으면 바로 끝낸다.

DFA와 달리 Backreference를 이용할 수 있다.

그러나 Backtracking을 시도하기 때문에 **같은 상태를 다른 경로로 여러번 반복하여 방문한다.**

이는 굉장히 느린 실행시간으로 이어지며, 처음 매칭되는 것만 찾고 바로 끝내기 때문에 후에 매칭될 수도 있는 것들은 찾지 못한다.

Backtracking (역추적)

검색 실패시에 정규식 엔진이 현재의 성공한 부분을 버리고 이전에 저장한 상태로 되돌아가서 다시 검색을 시도하는 것을 **Backtracking**이라고 한다.

```
using System;
using System.Text.RegularExpressions;
public class Example{
    public static void Main(){
        string input = "Essential services are provided by regular expressions.";
        string pattern = ".*(es)";
        Match m = Regex.Match(input, pattern, RegexOptions.IgnoreCase);
        if (m.Success) {
            Console.WriteLine("'es' found at position {1}",
                m.Value, m.Index);
            Console.WriteLine("'es' found at position {0}",
                m.Groups[1].Index);
        }
    }
}
//      'Essential services are provided by regular expres' found at position 0
//      'es' found at position 47
```

Backtracking은 다음과 같이 이루어진다.

1. .*(0개 이상의 임의 문자 검색)을 검색한다.

2. 다음 정규식 패턴인 e를 찾는다. 그러나 이미 .*에서 input의 끝까지 탐색했기 때문에 Backtracking을 시작한다.
3. "Essential services are provided by regular expressions." 에서 마지막 '.' 부터 뒤로 돌아가면서 e를 찾는다.
4. expressions의 e를 먼저 찾고, 그 뒤에 s가 있는지 찾아내어 검색에 성공한다.

결국 길이가 55자인 input에서 67번의 비교작업을 하였다.

POSIX NFA

Traditional NFA와 비슷하지만 다른 점은 **Backtracking으로 가능한 가장 긴 문자열을 찾는다**는 것이다. 당연히 Traditional NFA보다 느리다.

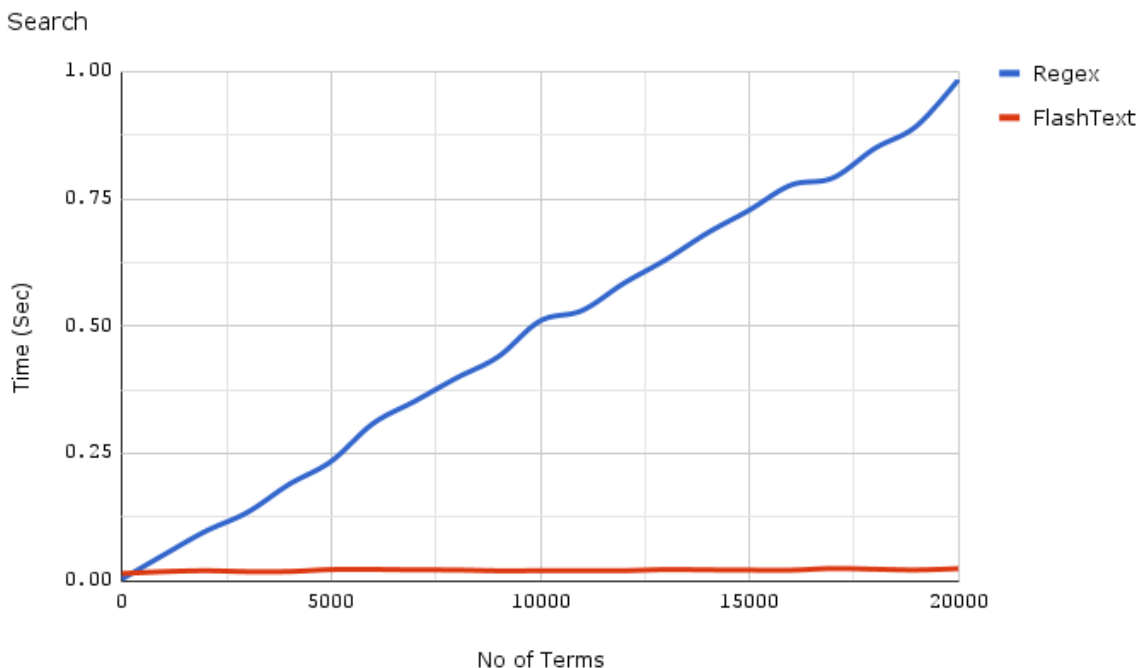
NFA 엔진을 사용하는 정규표현식의 최적화

지금까지 DFA,NFA에 대해서 알아보았다.

DFA는 선형 시간안에 탐색이 보장되지만 Backreference를 사용할 수 없었고,

NFA는 Backreference를 사용 가능하지만 Backtracking을 그 방법으로 이용하여 시간이 오래 걸릴 수 있었다.

만약 키워드의 개수가 많아진다면 정규표현식을 쓰는 방식에 따라 엄청난 성능 차이를 볼 수 있다.

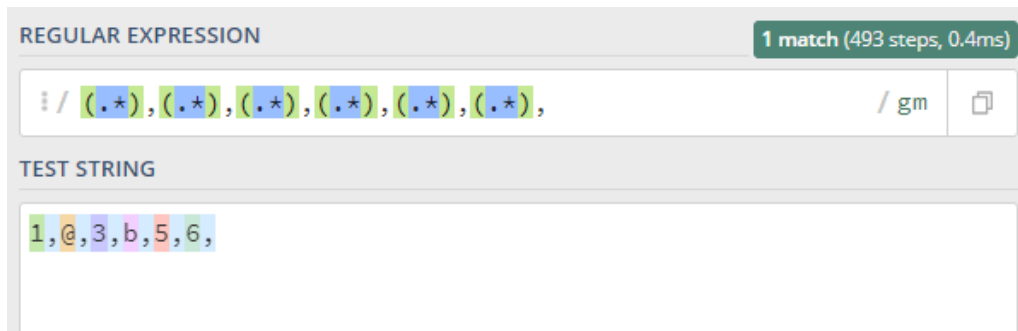


정규표현식과 FlashText를 이용한 문자열 매칭 시간 차이

위에서 NFA를 사용하는 정규표현식의 특징을 알았다.

그 특징을 이용하여 정규표현식의 최적화를 시도해볼 수 있다.

1. Greedy Quantifier vs Lazy Quantifier

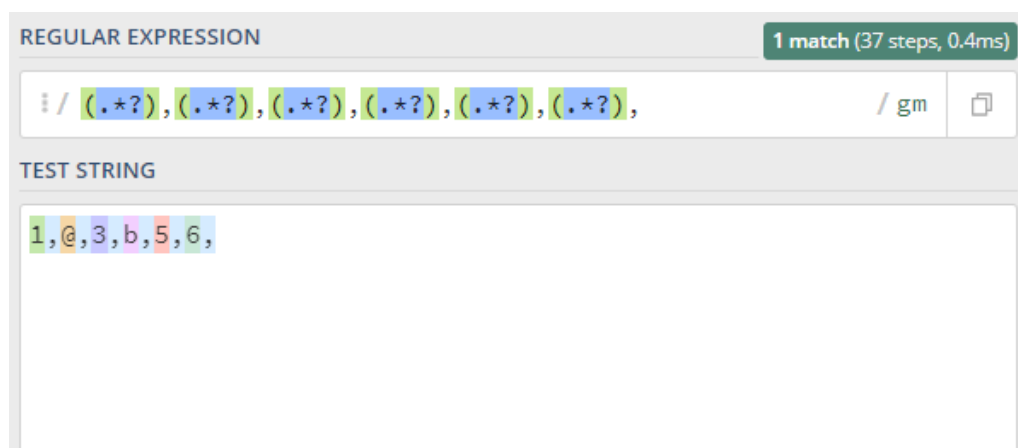


Greedy Quantifier 이용

<https://regex101.com/>에서 측정한 횟수이다.

Greedy Quantifier를 이용하였을 때 이 표현식은 왼쪽에서 오른쪽으로 탐색하면서 매칭에 실패하면 Backtracking을 시도한다. **매칭되는 가장 긴 패턴을 찾아낸다.**

`(.)*(.)`, 패턴을 찾기 위해 493번의 비교를 하였다.



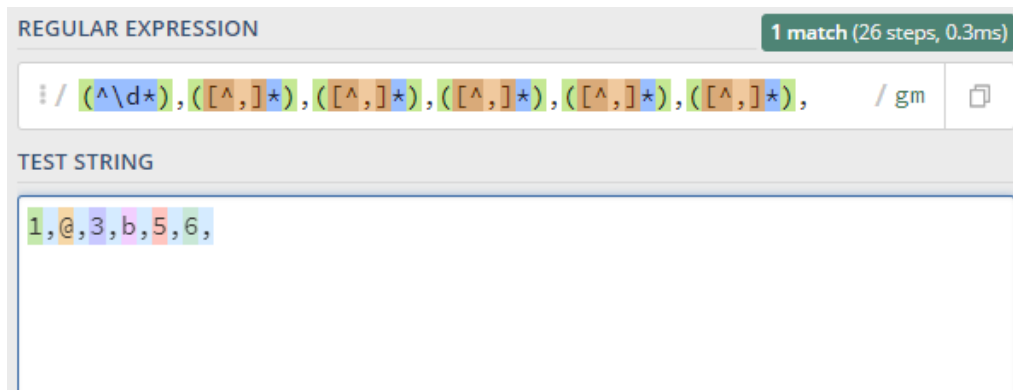
Lazy Quantifier 이용

Lazy Quantifier는 Greedy Quantifier의 뒤에 ?만 붙여주면 된다. Lazy Quantifier는 현재 매칭이 되는 정규식 이후에 다른 정규식이 있는지 먼저 체크한다. **매칭되는 가장 짧은 패턴들을 찾아낸다.**

똑같은 패턴을 찾는데 37번의 비교를 하였다.

2. 위치지정자 사용

위치지정자는 어떤 문자나 그룹이 문자열의 어느 부분에 있는지 등을 체크할 수 있다.



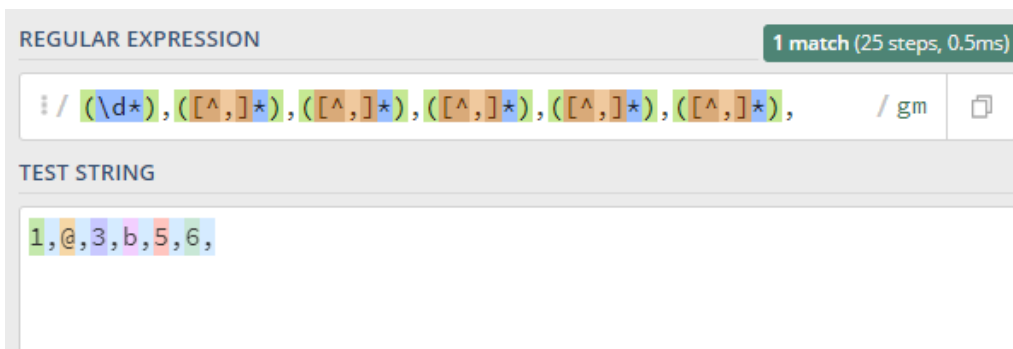
위치지정자 사용

^는 그룹의 시작 부분에 다음에 나오는 문자나 식과 일치하는지 판단한다. 이 경우 (^\\d*)로 첫 그룹의 시작 문자는 숫자인 것으로 지정했다.

^를 이용하여 그 위치에 맞는 표현식이 있는지 없는지 빠르게 판단할 수 있다.

3. 문자열 부정 사용

^를 []안에 쓰면 []안에 있는 문자열이 표현식과 일치하지 않은 경우만을 탐색한다.



문자열 부정 사용

이 경우 [^,]을 사용했는데, 다음에 ,가 아닌 문자가 오는 그룹을 만드는 표현식으로 ([^[,]*)를 사용했다.

마치며..

처음부터 정규표현식을 공부하려던 것은 아니었다. '웹 개발자를 위한 대규모 서비스를 지탱하는 기술'을 읽다가 정규표현식 성능 이슈 개선 부분에서 NFA로 동작한다는 이야기가 나왔고, 정규표현식의 내부 동작원리가 궁금해졌다.


코딩테스트에서 String을 처리하는 문제가 자주 나오곤 한다. 이때 정규표현식을 알면 깔끔하게 풀리는 문제들이 몇몇 있었다. 그 때마다 '정규표현식에 대해서 공부해야겠다'라는 갈증이 쌓였던 것 같다.

내가 주력으로 사용하는 Java의 경우 Traditional-NFA를 사용하며 같은 패턴을 찾더라도 어떻게 정규표현식을 설정하는지에 따라 최적화의 여지가 많다는 것을 알게 되었다.

References


정규표현식_성능_백트래킹

정규표현식의 성능을 어떻게 하면 개선할 수 있을까? 정리하면 아래 4가지를 유의해야 합니다. 정규식은 왼쪽에서 오른쪽으로 탐색을 하는데 100%매칭되지 않으면 다시 뒤로 되돌아가면서 매칭을 시도합니다. 이를 백트래킹이라고 합니다. 우선 예제를 통해 백

 <http://sword33.egloos.com/7294056>

Regex was taking 5 days to run. So I built a tool that did it in 15 minutes.


by Vikash Singh Regex was taking 5 days to run. So I built a tool that did it in 15 minutes.dia057 | UnsplashWhen developers work with text, they often need to clean it up first. Sometimes it's by replacing keywords. Like replacing "Javascript"

 <https://www.freecodecamp.org/news/regex-was-taking-5-days-flashtext-does-it-in-15-minutes-55f04411025f/>



.NET 정규식의 역행 검사


역추적은 정규식 패턴에 선택적인 수량자 또는 교체 구문 이 포함되어 있고 정규식 엔진 이 일치 항목을 계속 검색하기 위해 이전에 저장한 상태로 되돌아갈 때 발생합니다. 역추적은 정규식 성능의 핵심입니다. 역추적을 사용하면 식의 성능과 유연성을 높일 수 있

 <https://docs.microsoft.com/ko-kr/dotnet/standard/base-types/backtracking-in-regular-expressions>



DFA vs NFA engines: What is the difference in their capabilities and limitations?


Asked I am looking for a non-technical explanation of the difference between DFA vs NFA engines, based on their capabilities and limitations. james.garriss 11.6k 6 6 gold badges 77 77 silver badges 95 95 bronze badges asked Oct 20 '10 at 13:38 blunders blunders

 <https://stackoverflow.com/questions/3978438/dfa-vs-nfa-engines-what-is-the-difference-in-their-capabilities-and-limitations>



.NET 정규식의 역참조 구문

역참조는 문자열 내에서 반복된 문자 또는 부분 문자열을 식별하는 편리한 방법을 제공합니다. 예를 들어 입력 문자열에 임의의 부분 문자열이 여러 번 포함되어 있으면 첫 번째 발생을 캡처링 그룹과 일치시킨 다음 역참조를 사용하여 부분 문자열의 후속 발생을

 <https://docs.microsoft.com/ko-kr/dotnet/standard/base-types/backreference-constructs-in-regular-expressions>



Regex Engines

A regular expression describes a search pattern that can be applied on textual data to find matches. A regex is typically compiled to a form that can be executed efficiently on a computer. The actual search operation is performed by

 <https://devopedia.org/regex-engines>

