오픈랩2-KMP 결과 보고서

20171704 한상우

**1. KMP알고리즘이란?**

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 비효율적인 string search알고리즘를 효율적으로 실행 할 수 있게 해 주는 알고리즘이다. 일반적인 string search알고리즘과 KMP알고리즘을 비교해보면 다음과 같다   |  |  |  | | --- | --- | --- | | - | **일반적인 알고리즘(1번)** | **KMP알고리즘(2번)** | | **복잡도** | O(string\_size \* pattersize) | O(string\_size + pattersize) | | **기본적인 개념** | string의 처음부터 끝까지 살펴보며  pattern==string[i+strlen(pat)-1]을 확인해준다 | Failure fucntion을 이용해서 pattern을 분석하고 pattern search중 mismatch가 발생하면 분석 내용을 이용해 어디서부터 pattern search를 진행할지 결정한다 | | **차이점** | 1번은 string[0]에서 시작하여 string[i+strlen(pat)-1]까지 살펴 본 후 mismatch이면 string[1]에서 다시 pattern search를 해 줘야 하는 반면 2번의 경우 mismatch가 일어난 경우 failure function에서 얻은 정보를 가지고 어디서부터 pattern search를 진행 할지 정할 수 있다. 즉 1번은 search 도중에 한번 지나온 string[i]를 다시 보게되지만(moving backward) 2번의 경우에는 그렇지 않다.(search the pattern without moving backwards) | | |

**2. KMP알고리즘 분석**

|  |
| --- |
|  |
| **2-1-1. 코드 실행 결과(match)**  **string : ArtificialIntelligence**  **pat : elli** |
|  |
| **2-1-2. 코드 실행 결과(no match)**  **string : ArtificialIntelligence**  **pat : ellii** |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **2-2. 코드/알고리즘 설명**  **2-2-1. prefix와 suffix**   |  | | --- | | **suffix** | | b | | ab | | bab | | abab | | aabab | | baabab | | abaabab |  |  | | --- | | **prefix** | | a | | ab | | aba | | abaa | | abaab | | abaaba | | abaabab |   abbaabab라는 문자열에서의 prefix와 suffix의 예시이다.  즉, prefix는 p[시작]p[시작+1]…p[시작+i]로, suffix는 p[끝-i]p[끝-I+1…]p[끝]으로 볼 수 있다  **2-2-2. failure function**  abaabab라는 pattern을 입력받은 경우 다음과 같은 결과가 나온다.   |  |  |  |  | | --- | --- | --- | --- | | **pat[j]** | **j** | **case** | **failure[j]** | | a | 0 | - | -1 | | b | 1 | else | -1 | | a | 2 | if | 0 | | a | 3 | While, if | 0 | | b | 4 | if | 1 | | a | 5 | if | 2 | | b | 6 | While, if | 1 |   failure function에서는 결국 pattern을 pat[0]~pat[j]까지의 문자열로 쪼개서 prefix = suffix를 만족하는 최고길이를 찾아 failure[j]에 저장해주는 역할을 한다. 이 때 prefix와 suffix가 일치되는 만큼 이동이 가능하다  **2-2-3. pmatch**  pmatch의 while문에서는 3가지 중 하나의 작업을 실행한다.   |  |  | | --- | --- | | **if (string[i] == pat[j])** | 같을 경우 모두 index를 이동시켜 다음 char 비교 | | **else if (j == 0)** | j가 0인 케이스, string의 인덱스만 이동 | | **else** | mismatch인 경우 failure function을 이용해 pat의 인덱스만 이동 (pattern에서 prefix의 위치를 suffix의 위치로 이동) | |