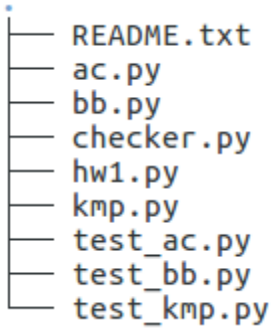
고급계산이론 HW1 보고서

2017-34165 김성국

1 프로젝트 구성

본 프로젝트의 구성은 다음과 같다.



최종 목표인 Baker-Bird 알고리즘을 구현하기 위해서 꼭 필요한 KMP 알고리즘과 AC 알고리즘도 구현하였다. 각각 bb.py, kmp.py, ac.py에 python3으로 구현하였다.

다음으로 위의 모듈을 사용하여 주어진 과제를 수행하는 프로그램을 작성하였다. hw1.py와 checker.py가 이에 해당한다.

또한 부차적으로 알고리즘 구현을 테스트하기 위하여 유닛 테스트들을 작성하였다. 파이썬의 unittest 모듈을 이용하였고, 알고리즘마다 test_<알고리즘>.py의 파일에 작성하였다.

프로그램의 자세한 실행 방법은 README.txt에 작성하였다.

2 Knuth-Morris-Pratt 알고리즘 (소스 : kmp.py)

소스 kmp.py는 Knuth-Morris-Pratt 알고리즘을 구현한다. 이는 추후에 Baker-Bird 알고리즘의 column-matching 단계에서 사용될 것이다.

구현은 class를 이용하여 구현하였다. 이는 주어진 패턴 P 에 대한 전 처리를 딱 한번만 하게하고, T에 대한 search를 여러번 할 때 전 처리된 결과물을 반복해서 사용하기 위함이다. 즉 하나의 패턴마다 instance를 생성하여 반복적으로 사용을 할 수 있다. 하나의 instance는 패턴 P 그 자체(p), 패턴의 길이(m), failure function(ff)을 instance variable로 지닌다.

Failure function의 생성은 construct()함수가 담당을 한다. 수업 슬라이드의 방식을 그대로 사용하였고 인덱스 정도만 바뀌었다.

텍스트 T에 대한 search는 search(T)함수가 담당을 한다. 텍스트 T를 입력 받아 수업 슬라이드의 방식대로 search를 진행한다. Match가 발생하면 yield를 하도록 하여 iterator로 사용되기 좋게 만들었다.

위의 search(T)와는 별도로, 입력 stream에 대해서 탐색을 진행하는 stream()함수도 구현하였다. 이는 Baker-Bird 알고리즘이 space 사용을 줄이고자, n개의 KMP 알고리즘에 각각 stream으로 입력을 주기 때문이다. 이를 위해 stream()은 generator를 사용하여 구현하였고, 이를 통해서 n개의 호출마다 search 상태를 유지할 수 있게 되었다. 동작 방식은 search(T)에서와 같이 수업 슬라이드의 방식을 사용한다.

3 Aho-Corasick 알고리즘 (소스 : ac.py)

소스 ac.py는 Aho-Corasick 알고리즘을 구현한다. 이는 추후에 Baker-Bird 알고리즘의 row-matching 단계에서 사용될 것이다.

구현은 마찬가지로 class를 이용하여 구현하였다. 패턴들 P[] 를 전 처리하여 반복적으로 사용한다. 하나의 instance는 패턴들 P[](p), 가능한 최대 state의 수(m), trie(tree), failure function(ff), output function(of)를 저장한다.

Trie, failure function, output function의 모든 생성을 construct()함수가 담당한다.

Trie는 배열을 이용하여 구현하였다. 알파벳이 a-z라는 가정이 있기 때문에 m by 26 사이즈의 배열로 trie를 구현할 수 있다. 배열에 접근할 때는 ord(char) - ord('a')를 통해서 문자를 숫자 인덱스로 변환할 수 있다. 각 패턴을 처음부터 끝까지 읽어가며 trie를 확장시켜 나간다. Edge가 없는 경우는 -1로 나타낸다. 패턴의 마지막을 읽을 때는 output function의 초기 값도 같이 설정해 준다.

그 다음 failure function을 만든다. Failure function은 수업 슬라이드에서 설명된 것처럼, depth에 대해서 귀납적인 구조를 갖는다. 따라서, breadth-first search를 통해서 낮은 depth부터 순차적으로 failure function을 생성해 나간다. 즉, depth <= 1인 state의 경우 base case로써 failure function 값을 0으로 설정하고, BFS로 더 깊은 depth의 state를 만날 때마다 현재 state(curr), 이전 state(prev), 이 둘을 연결하는 알파벳(char)을 사용해 failure function의 값을 구한다. 이 때, 수업 슬라이드에서 설명하는 것처럼 output function도 같이 만들 수 있다.

KMP와 같이 여기에서도 텍스트 T에 대한 search는 search(T)함수가 담당을 한다. 텍스트 T를 입력 받아 수업 슬라이드의 방식대로 search를 진행한다. Match가 발생하면 yield를 하도록 하여 iterator로 사용되기 좋게 만들었다.

4 Baker-Bird 알고리즘 (소스 : bb.py)

소스 bb.py는 Baker-Bird 알고리즘을 구현한다. 앞서 정의한 ac.py와 kmp.py를 각각 row-matching, column-matching에 사용한다.

이 또한 class로 구현을 하였고, 2-D 패턴 P(p), P의 사이즈 m, 패턴 한 줄의 distinct row number를 알려주는 테이블(dr), P를 distinct row number로 변환한 P'(pp), Aho-Corasick instance(ac), KMP instance(kmp)를 Baker-Bird instance마다 갖도록 한다. construct()는 크게 복잡한 것 없이, distinct row number와 관련된 dr과 pp을 생성한다.

텍스트 T에 대한 search에는 search(T) 함수를 이용하는데, 기본적으로 수업 슬라이드와 같은 방식이다. 특히, 여기서 extra space의 사용을 줄이기 위해 O(n^2)의 R을 할당하지 않고, AC를 통해 R의 row가 한줄한줄 생성될 때마다 n번의 KMP search를 한 단계씩 진행 시킨다. 이처럼 n번의 search를 동시 다발적으로 각기 다른 stream에 대해서 실행하도록, 앞서 언급한 것처럼 KMP에서 generator를 사용해 stream()을 구현하였다. Generator는 자신만의 실행 상태를 유지한 채, 자신을 호출한 루틴과 실행 흐름을 주고 받을 수 있으므로 Baker-Bird 알고리즘을 구현하는데 매우 적합하다.

위의 search(T) 함수도 match가 발생하면 yield를 하도록 하여 iterator로 사용되기 좋게 만들었다.

5 HW1 (소스 : hw1.py)

소스 hw1.py는 위의 모듈들을 사용해 주어진 과제를 해결하도록 만든 프로그램의 코드이다.

파일 입출력만을 담당하며, 실질적인 알고리즘은 위의 모듈들에 맡긴다. 우선 입력 파일을 읽어 정규식 등을 통해 처리한 후, 패턴 P(p)와 텍스트 T(t)를 만든다. 그 다음 Baker-Bird 알고리즘을 실행하면서 match되는 인덱스 쌍을 얻을 때마다 형식에 맞춰 string으로 저장한 후 최종적으로 출력 파일로 내보낸다.

6 Checker (소스 : checker.py)

소스 checker.py는 위의 hw1.py 및 알고리즘 모듈들이 제대로 구현된 것인지 확인하는 프로그램의 코드이다.

Checker가 정확성을 따지는 기본적인 아이디어는 다음과 같다. 우선 2-D pattern matching 문제를 동일하게 해결하지만, 더 비효율적이고 간단한(정확성을 따지기 쉬운) 알고리즘을 구현한 뒤, 이 것의 실행 값과 Baker-Bird 알고리즘의 실행 값을 비교한다. 본 checker에서는 더 간단한 알고리즘으로 naive한 brute-force 방식을 사용한다. 즉, 4중 loop을 돌며(O(m^2*n^2)) match 되는 위치들을 찾고, 그 위치들을 set에 모아 놓았다가 읽은 위치들(이 것도 set에 저장)과 비교한다.

파일 입출력은 hw1.py와 같은 방식으로 한다.

7 결론

단순하지 않은 알고리즘을 직접 구현해 볼 수 있어서 유익한 경험이었다. 각 알고리즘마다 모듈화 하여 잘 구현한 것 같아서 더 좋은 경험이었던 것 같다. 특히, generator를 통해 extra space를 줄이는 Baker-Bird를 구현한 것은 좋은 경험이었다고 생각한다.