30520 조휘정 알고리즘 보고서

목차

1. 압축 포맷과 간단한 압축과정
2. 대중적인 ZIP 압축포맷과 Huffman 알고리즘
3. 간단하게 파이썬으로 Huffman 알고리즘 구현
4. 느낀점과 알게된 점
5. 출처
6. 전체 코드 및 전체 실행 결과

1. 압축 포맷과 간략한 압축과정

우리는 컴퓨터를 이용하면서 다양한 파일들을 사용하게 되는데, 이 과정속에서 다양한 압축 기술이 적용된 파일들을 이용하기도 한다. 가장 잘 알려진 압축 포맷으로는 ZIP, RAR, 7z 같은 포맷들도 있고 일반적으로는 압축 포맷이라고 잘 못 느낄 수 있지만, GIF, PNG, JPEG 등과 같은 이미지 포맷들도 사실 이미 압축이 적용된 포맷이다.

압축 포맷은 또 손실 압축 포맷과 무손실 압축 포맷으로 나누어 볼 수 있는데, 손실 압축은 압축과정에서 원본 데이터가 손실되는 포맷이다. 대표적으로는 JPEG가 있는데, 손실 압축 포맷이긴 하지만 사람의 눈으로는 그 손실의 정도를 지각하기 힘든 부분의 데이터를 버리고 압축하여 원본 데이터가 손실되는 것이므로, JPEG가 손질 압축 포맷이긴 하지만 실제로 JPEG 이미지 파일을 우리가 봐도 이질감을 느끼지 못하거나 거의 느끼지 못한다. 위에서 언급한 압축 포맷들 중 ZIP, RAR, 7z, GIF, PNG는 무손실 압축 포맷이다. 무손실 압축 포맷은 대게 데이터에서 반복되는 정보들을 줄여서 압축을 하는 포맷들이다.

예를 들어 18281828182818281828 라는 20바이트 데이터에서 1828을 a로 치환해서 압축한다고 하면, aaaaa로 나타낼수 있고 20바이트의 데이터를 aaaaa로 나타내었으므로 총 15바이트를 절약하게 된것이다. 하지만 이러한 무손실 압축 포맷은 손실 압축 포맷에 비해 압축률이 떨어지고 이미 압축된 데이터를 다시 압축한다고해서 효과가 있지는 않다.

2. 대중적인 ZIP 압축포맷과 Huffman 알고리즘

가장 대중적으로 받아들여지는 압축 포맷은 ZIP은 Deflate라는 압축 알고리즘을 사용한다고 한다. 그리고 Deflate 압축 알고리즘은 LZ77 알고리즘과 Huffman 알고리즘을 혼용하여 사용하는데, 이번에는 Huffman 알고리즘을 알아보도록 하자. Huffman 알고리즘은 직업 압축을 하는 알고리즘은 아니지만, ZIP 포맷으로 압축을 하는데 필요하다. 그 이유는 컴퓨터가 문자를 ‘0’ 또는 ‘1’로 나타내어야 압축하기 편하기 때문이다. 기본적으로 Huffman 알고리즘은 특정문자를 비트로 표한하게 되는데, 이러한 특성 때문에 가장 많이 등장하는 문자를 가장 짧은, 적은 비트수로 표현이 되어야 효율적으로 압축할 수 있게 된다. 예를들어 ‘아’라는 단어가 어떤 파일에서 500번 정도 반복되고 있고 ‘가’라는 단어가 1000번 정도 반복되고 있다고 했을 때 ‘아’를 10, ‘가’를 011로 표현했다고 하자. 그러면 Huffman 알고리즘을 거치고 난 뒤에는 10이 500번, 011이 1000번 반복되어 총 4000개 정도의 비트수로 바뀌게 된다 하지만 ‘아’를 011, ‘가’를 10으로 바꾸어 진행한다면, 011이 500번, 10이 1000번 반복되게 되어 총 비트수가 3500개 정도가 된다. 이렇듯 가장 많이 반복되는 문자를 가장 적은 비트수로 표현하여야 알고리즘이 효율적으로 작동하게 된다. 그리고 Huffman 알고리즘에는 두번째 규칙이 있는데, 어떤 한 문자를 표현한 비트 조합이 다른 문자를 표현하는 비트 조합에서 접두사로 시작되면 안된다. 예시를 이용하면 바로 이해할 수가 있는데 ‘가’를 01, ‘나’를 011로 표현하게 되면 컴퓨터는 01까지 읽었을 때 이 비트 조합이 ‘가’를 나타내는 것인지, ‘나’를 나타는 조합인지 알수가 없다. 따라서 효율과 무결성을 위해 이러한 두번째 규칙을 지켜주어야 한다.

3. 간단하게 파이썬으로 Huffman 알고리즘 구현

구현 설명에 앞서, 사용한 라이브러리로는 쉽게 트리 구성을 할 수 있게 도와주는 anytree 라이브러리를 사용하였다.

(출처: <https://tekhartha.com/programming/2019/02/14/anytree/>)

(테스트에 사용된 파일의 이름은 test.txt이며, 파일의 내용은

assdddffffggggghhhhhhjjjjjjjuuuuuuuu 이다.)

def gen\_freq\_dict(f) -> dict:  
 line = f.readline()  
 f.close()  
 temp\_dict = dict()  
 while line:  
 temp\_dict[line[0]] = line.count(line[0])  
 line = line.replace(line[0], '')  
 return dict(sorted(temp\_dict.items(), key=lambda x: x[1]))

제일 처음에 선언한 함수이다. 함수에서 받는 매개변수 f는 파이썬에서 파일을 읽을 때 사용하는 open함수의 리턴을 받는 함수이다. gen\_freq\_dict함수는 파일을 읽어 파일에 각 문자가 어떤 빈도수만큼 파일내에 있는지를 나타내는 딕셔너리를 리턴하는 함수이다.

freq\_dict: dict = gen\_freq\_dict(open("C:/Users/Saryrn/Documents/Study/Algorithm-test/test.txt"))  
print(freq\_dict)

함수 선언 후 다음을 실행한 결과이다.



a라는 문자가 1번, s라는 문자가 2번 있다는 것과 나머지 문자들도 확인할 수 있었다.

def gen\_binary\_huffman\_tree(freq\_dict: dict) -> Node:  
 temp\_dict = {i: Node(i, freq=freq\_dict[i], isfreq=False) for i in freq\_dict.keys()}  
 print(temp\_dict)  
 while len(temp\_dict) != 1:  
 keys = list(temp\_dict.keys())  
 temp\_dict[f"freq {temp\_dict[keys[0]].freq + temp\_dict[keys[1]].freq}"] = Node(f"freq {temp\_dict[keys[0]].freq + temp\_dict[keys[1]].freq}", freq=temp\_dict[keys[0]].freq + temp\_dict[keys[1]].freq)  
 keys = list(temp\_dict.keys())  
 temp\_dict[keys[-1]].isfreq = True  
 temp\_dict[keys[0]].id = 0  
 temp\_dict[keys[1]].id = 1  
 temp\_dict[keys[0]].parent = temp\_dict[keys[-1]]  
 temp\_dict[keys[1]].parent = temp\_dict[keys[-1]]  
 temp\_dict.pop(keys[0])  
 temp\_dict.pop(keys[1])  
 temp\_dict = (  
 dict(sorted(temp\_dict.items(), key=lambda x: x[1].freq)))  
 print(temp\_dict)  
  
 return temp\_dict[list(temp\_dict.keys())[0]]

다음은 huffman 트리를 만드는 함수이다. 함수가 실행되면 gen\_freq\_dict함수에서 만든 문자들과 각 문자들의 빈도수가 들어있는 딕셔너리를 freq\_dict라는 매개변수로 받아 temp\_dict에 빈도수와 isfreq(구분을 위한 더미노드인지 확인하기위한 불린값)을 가지고 있는 Node들을 저장하게된다. 다음 출력은 세번째 줄에 있는 print(temp\_dict)의 출력이다. (출력이 길어서 사진이 작고 뒷부분이 잘려있습니다. 확인이 필요하다면 본 문서를 확대하여 확인해주세요)



그 다음 While문은 freq가 가장 낮은 2개의 노드를 선택하여(temp\_dict속 노드들은 freq값에 따라 오름차순으로 정렬되어있다.)더미 노드를 만든후 그 더미 노드의 자식으로 설정하는 과정을 진행한다. 그 과정에서 어떤 부모의 왼쪽 자식인 경우에는 id값을 0으로 설정하고 오른쪽 자식인 경우에는 id값을 1로 설정한다. 그리고 이 과정을 While문의 조건을 통하여 temp\_dict의 길이가 1이 아닐때까지, 즉 temp\_dict안에 루트 노드만 남을때까지 진행한다. 그리고 gen\_binary\_huffman\_tree는 만들어진 트리의 루트 노드를 리턴하게 된다.

root: Node = gen\_binary\_huffman\_tree(freq\_dict)  
root.id, root.isfreq = -1, -1  
  
for pre, fill, node in RenderTree(root):  
 print(f"{pre}{node.name}: {node.freq}")

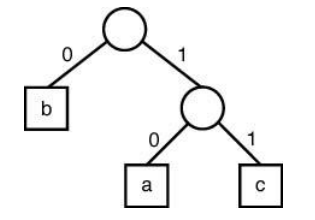
이 코드에서 for문에서는 트리의 구조를 출력하게되며

각 노드가 가진 문자와 빈도수를 출력하게 된다. 다음은 해당 코드를 실행시킨 결과이다.

텍스트, 스크린샷, 도표, 폰트이(가) 표시된 사진

자동 생성된 설명

이렇게 만들어진 트리에서 문자를 표현할 비트를 결정해주어야 한다. 기본적으로 Huffman 알고리즘에서는 트리의 가지 왼쪽은 0, 그리고 오른쪽은 1로 설정하여 비트를 결정하게 된다.

(출처:

<https://hojung-testbench.tistory.com/entry/c-Huffman-Algorithm-%ED%8C%8C%EC%9D%BC-%EC%95%95%EC%B6%95-%EC%95%8C%EA%B3%A0%EB%A6%AC%EC%A6%98>)

예를 들어 다음과 같은 트리에서는 b문자를 0, a문자를 10,. c문자를 11로 비트를 결정하게된다.

이번에 구현한 알고리즘에서는 이과정을 아까 설명한 gen\_binary\_huffman\_tree함수에서 노드들에 id값을 설정하는 것으로 트리의 가지 왼쪽은 0을 오른쪽은 1로 설정하였다.

def gen\_huffman\_final\_dict(root: Node) -> dict:  
 huffman\_final\_dict = dict()  
 for pre, fill, node in RenderTree(root):  
 if not node.isfreq:  
 string = ""  
 for n in node. Path:  
 if n.id != -1:  
 string += str(n.id)  
 huffman\_final\_dict[node.name] = string  
 return huffman\_final\_dict

이 함수는 트리에 있는 노드들마다 for문을 돌려서 해당 노드 문자의 최종 비트를 구한후 그 정보가 들어있는 딕셔너리를 리턴하는 함수이다.

print(gen\_huffman\_final\_dict(root))

다음은 해당 코드를 실행한 출력이다.



이렇게 하여 Huffman 알고리즘을 파이썬으로 간단하게 구현해보았다.

4. 느낀점과 알게된 점

이번에 다양한 압축 포맷들에 대해 조사하고 구현해보는 과정에서 평소에 쓰던 여러가지 파일들이 내가 모르는 사이에 다양한 알고리즘들을 통하여 압축되고 있다는 사실을 알게 되어 흥미로웠다. 파이썬으로 알고리즘을 구현할 때는 과정이 잘 진행될때마다 쾌감을 느껴서 기분이 정말 좋았던 것 같다. 한 가지 아쉬운 점으로는 알고리즘을 구현하면서 빈도수가 높은 문자를 가능한 짧은 비트수로 정하는 것에는 성공적이지 못해서 아쉬웠다. 나중에 시간이 될 때 한번 수정해보고싶다.

5. 출처

무손실 압축 포맷:

<https://namu.wiki/w/%EB%AC%B4%EC%86%90%EC%8B%A4%20%EC%95%95%EC%B6%95%20%ED%8F%AC%EB%A7%B7>

손실 압축 포맷:

<https://namu.wiki/w/%EC%86%90%EC%8B%A4%20%EC%95%95%EC%B6%95%20%ED%8F%AC%EB%A7%B7>

anytree 라이브러리에 관한 설명:

<https://tekhartha.com/programming/2019/02/14/anytree/>

Huffman 알고리즘:

<https://hojung-testbench.tistory.com/entry/c-Huffman-Algorithm-%ED%8C%8C%EC%9D%BC-%EC%95%95%EC%B6%95-%EC%95%8C%EA%B3%A0%EB%A6%AC%EC%A6%98>

6. 전체 코드 및 전체 실행 결과

from anytree import Node, RenderTree  
  
  
def gen\_freq\_dict(f) -> dict:  
 line = f.readline()  
 f.close()  
 temp\_dict = dict()  
 while line:  
 temp\_dict[line[0]] = line.count(line[0])  
 line = line.replace(line[0], '')  
 return dict(sorted(temp\_dict.items(), key=lambda x: x[1]))  
  
  
def gen\_binary\_huffman\_tree(freq\_dict: dict) -> Node:  
 temp\_dict = {i: Node(i, freq=freq\_dict[i], isfreq=False) for i in freq\_dict.keys()}  
 print(temp\_dict)  
 while len(temp\_dict) != 1:  
 keys = list(temp\_dict.keys())  
 temp\_dict[f"freq {temp\_dict[keys[0]].freq + temp\_dict[keys[1]].freq}"] = Node(f"freq {temp\_dict[keys[0]].freq + temp\_dict[keys[1]].freq}", freq=temp\_dict[keys[0]].freq + temp\_dict[keys[1]].freq)  
 keys = list(temp\_dict.keys())  
 temp\_dict[keys[-1]].isfreq = True  
 temp\_dict[keys[0]].id = 0  
 temp\_dict[keys[1]].id = 1  
 temp\_dict[keys[0]].parent = temp\_dict[keys[-1]]  
 temp\_dict[keys[1]].parent = temp\_dict[keys[-1]]  
 temp\_dict.pop(keys[0])  
 temp\_dict.pop(keys[1])  
 temp\_dict = (  
 dict(sorted(temp\_dict.items(), key=lambda x: x[1].freq)))  
 print(temp\_dict)  
  
 return temp\_dict[list(temp\_dict.keys())[0]]  
  
  
def gen\_huffman\_final\_dict(root: Node) -> dict:  
 huffman\_final\_dict = dict()  
 for pre, fill, node in RenderTree(root):  
 if not node.isfreq:  
 string = ""  
 for n in node.path:  
 if n.id != -1:  
 string += str(n.id)  
 huffman\_final\_dict[node.name] = string  
 return huffman\_final\_dict  
  
  
freq\_dict: dict = gen\_freq\_dict(open("C:/Users/Saryrn/Documents/Study/Algorithm-test/test.txt"))  
print(freq\_dict)  
root: Node = gen\_binary\_huffman\_tree(freq\_dict)  
root.id, root.isfreq = -1, -1  
  
for pre, fill, node in RenderTree(root):  
 print(f"{pre}{node.name}: {node.freq}")  
  
print(gen\_huffman\_final\_dict(root))

텍스트, 스크린샷, 디스플레이, 소프트웨어이(가) 표시된 사진

자동 생성된 설명