REPORT

lab 09

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 과목명 : | | | 컴퓨터알고리즘과실습 | | | | | | | | |
| 담당교수 : | | | 주종화 | | | | | | | 교수님 | |
| 제출일 : | 2021 | | | 년 | | 05 | 월 | | 31 | | 일 |
| 공과 | | 대학 | | | 컴퓨터공학 | | | 과 | | | |
| 학번 : | | 2016112154 | | | 이름: | | | 정동구 | | | |

1. 문제 1
2. 결과

텍스트이(가) 표시된 사진

자동 생성된 설명

위부터 sequence를 순서대로 1,2,3,4 라고 하였을 때 3번 sequence의 경우 결과가 총 두가지가 나와야 하지만 한가지만 나와서 완전하지 않다.

1. 코드

hamiltonian::hamiltonian(std::string input[], int arrsize)//constructor

{

arr\_size = arrsize;

sequence = new std::string[arr\_size];

str\_size = input[0].length();

adj\_matrix = new int\* [arr\_size];

path\_vertex = new int[arr\_size];

std::cout << "string size = " << str\_size << std::endl;

std::cout << "arr size = " << arr\_size << std::endl;

// 인접행렬 동적 할당, 스펙트럼 배열 생성, 경로 배열 초기화

for (int i = 0; i < arr\_size; i++)

{

path\_vertex[i] = -1;

sequence[i] = input[i];

adj\_matrix[i] = new int[arr\_size];

for (int j = 0; j < arr\_size; j++)

{

adj\_matrix[i][j] = 0;

}

}

}

Hamiltonian 클래스의 생성자이다. 인접행렬간 그래프를 배열로 선언한다.

hamiltonian::~hamiltonian()//destructor

{

for (int i = 0; i < arr\_size; i++)

{

delete[] adj\_matrix[i];

}

delete[] sequence;

delete[] path\_vertex;

}

Hamiltonian 클래스의 소멸자이다. 동적할당한 배열을 delete한다.

// 인접 행렬 생성

void hamiltonian::make\_adj\_matrix()

{

for (int i = 0; i < arr\_size; i++)

{

for (int j = 0; j < arr\_size; j++)

{

// i != j 이고, 출발 길이 n인 스펙트럼의 1부터 끝까지의 서브 스트링과 도착 스펙트럼의 시작부터

// n-1까지의 서브스트링이 같은지 비교후 같을 시 인접행렬에 추가.

if (i != j &&

((sequence[i]).substr(1, str\_size - 1) == (sequence[j]).substr(0, str\_size - 1)))

{

adj\_matrix[i][j] = 1;

}

}

}

}

Make\_adj\_matrix 함수로 글자 길이-1 만큼의 개수가 뒤에서 연속으로 중복 될 경우 path가 있는 것으로 보아 인접행렬을 형성한다.

// 도착한 곳에 갈곳이 있는지 없는지 체크

bool hamiltonian::check\_end(int index)

{

int cnt = 0;

for (int i = 0; i < arr\_size; i++)

{

if (adj\_matrix[index][i] == 1)

return false;

}

return true;

}

해밀턴 패스에서 더 진행 할 인접행렬이 있는지 판단한다.

// 이미 한번 왔던 곳인지 체크

bool hamiltonian::check\_path(int destination, int index)

{

if (adj\_matrix[(path\_vertex[index - 1])][destination] == 0)

return false;

for (int i = 0; i < index; i++)

{

if (path\_vertex[i] == destination)

{

return false;

}

}

return true;

}

이미 한번 왔던곳인지 확인하는 check\_path 함수이다.

bool hamiltonian::hamilton\_sub(int index)

{

// 만약 그래프의 존재하는 버텍스를 다 돌았을 경우

if (index == arr\_size)

{

// 끝인지 확인, 끝일 경우 true 반환

if (check\_end(path\_vertex[index - 1]))

return true;

else

return false;

}

// 끝이 아닐 경우

for (int v = 0; v < arr\_size; v++)

{

// 이미 한번 갔었던 곳인지 체크, 아닐 경우 동작 수행

if (check\_path(v, index))

{

path\_vertex[index] = v;

if (hamilton\_sub(index + 1) == true)

return true;

// true 반환 안됐을 경우 경로 삭제

path\_vertex[index] = -1;

}

}

return false;

}

해밀턴 패스의 핵심으로 재귀를 통해 구현하였다. 방문했던 곳이 아닌경우 path\_vertex에 추가하고 이를 재귀로 반복한다.

void hamiltonian::hamilton\_rec()

{

// 시작 인덱스 = path[0], 시작점은 반복문으로 모두 수행해봄

for (int j = 0; j < arr\_size; j++)

{

path\_vertex[0] = j;

// 경로를 찾았을 경우

if (hamilton\_sub(1) == true)

{

std::string answer = sequence[j];

std::cout << "Path Found!! " << std::endl;

std::cout << j << " ";

for (int i = 1; i < arr\_size; i++)

{

std::cout << path\_vertex[i] << " ";

answer = answer + sequence[path\_vertex[i]].substr(str\_size - 1, 1);

}

std::cout << std::endl << answer << std::endl << std::endl;

}

}

}

재귀함수를 호출하여 실행하고, 완성된 경로를 출력한다.

1. 문제 2
2. 결과

텍스트이(가) 표시된 사진

자동 생성된 설명

Mississippi와 이름인 chungdonggu를 입력하여 BWT를 실행한 결과이다. $를 문자열 뒤에 추가한 후 만든 rest에서 첫번째와 마지막을 출력하고, 이를 통해 문자열을 복원한 결과가 마지막 줄에 출력되어있다.

1. 코드

void BWT::init()

{

int length = pattern\_str.length();

std::string temp = pattern\_str;

for (int j = 0; j < length; j++)//한칸씩 밀어서 생성한 문자열을 rest에 pushback한다.

{

rest.push\_back(pattern\_str);

for (int i = 0; i < length; i++)

{

pattern\_str[i] = temp[(i + 1) % length];

}

temp = pattern\_str;

}

}

초기화 하는 함수로 문자열을 한칸씩 밀어 rest를 생성한다.

void BWT::sortRest()

{

int length = rest[0].length();

std::sort(rest.begin(), rest.end());//사전순 정렬

for (int i = 0; i < length; i++)

{

first += rest[i][0];

last += rest[i][rest[i].length() - 1];

}

std::cout <<"first: " <<first << std::endl << "last: "<<last << std::endl;

last\_tuples.resize(length);

std::vector<char> char\_set;

char tuple = first[0];

int num = 0;

int j = 0;

for (int i = 0; i < length;)//첫번째 rest를 통해 중복되어있는 문자의 순서를 매긴다. pair을 이용하고 이를 first\_tuples에 push back 한다.

{

if (tuple == '$')

{

first\_tuples.push\_back(std::pair<char, int>(tuple, num));

char\_set.push\_back(tuple);

tuple = first[i + 1];

j = i + 1;

i++;

continue;

}

if (first[i] == tuple)

{

num++;

first\_tuples.push\_back(std::pair<char, int>(tuple, num));

char\_set.push\_back(tuple);//마지막 rest도 순서를 매기기 위해 만든 char\_set에 tuple push\_back

i++;

}

else

{

j += num;

tuple = first[j];

num = 0;

}

}

Rest를 정렬하고 중복되는 문자를 순서대로 번호를 매기는 과정이다. 복원을 하기 위해 첫번째 rest와 마지막 rest에 번호를 매긴다.

void BWT::decode()//문자열 복원.

{

int length = 0;

int index = 0;

result\_str += last\_tuples[0].first;

while (result\_str.length()< pattern\_str.length()-1)

{

for (int i = 0; i < pattern\_str.length(); i++)

{

if (first\_tuples[i] == last\_tuples[index])

{

index = i;

if (last\_tuples[index].first == '$')continue;

result\_str += last\_tuples[index].first;

}

}

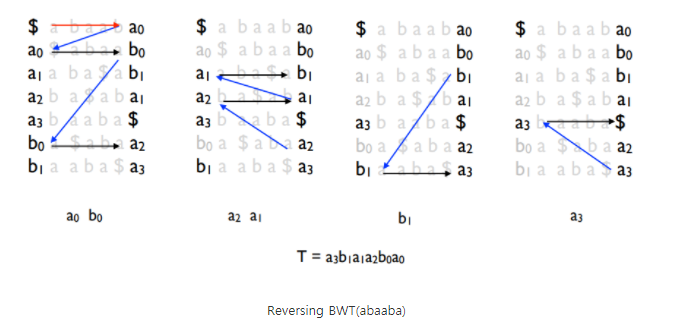
}

std::reverse(result\_str.begin(), result\_str.end());

std::cout << result\_str << std::endl;

}

문자열을 복원하는 함수이다. 복원 과정은 아래 그림과 같은 과정을 거쳤다.

우선 first와 last rest를 그림과 같이 두었다고 가정할 경우 first의 $문자에서 출발한다. 동일한 인덱스의 last rest로 넘어 간 후 해당 index의 문자와 순번과 일치하는 index를 first rest를 찾는다. 해당 과정을 반복해서 모든 문자를 탐색하고 이를 역순으로 하면 문자열이 복원된다.

1. 문제 3
2. 결과

텍스트이(가) 표시된 사진

자동 생성된 설명

길이 30 개수 40000개의 read를 만들었을 때 결과

총 343초가 소요되었고 96.201%의 일치율을 보인다.

텍스트이(가) 표시된 사진

자동 생성된 설명

길이 60 개수 30000개의 short read를 만들었을 때 결과

총 258초가 소요되었고 98.2671%의 일치율을 보인다.

1. 코드

ACTG::ACTG(int k\_,int n\_)//생성자

{

n = n\_;

k = k\_;

for (int i = 0; i < 1000000; i++)

{

ref\_DNA\_seq += random();

}

my\_DNA\_seq = ref\_DNA\_seq;

}

ACTG클래스의 생성자로, short\_read의 길이 k와 개수 n을 생성시에 입력받고, 레퍼런스 DNA를 1000000길이로 생성하고 my\_DNA도 똑같이 초기화 한다.

void ACTG::init()//short read를 생성한다.

{

for (int i = 0; i < my\_DNA\_seq.length(); i += k)

{

my\_DNA\_seq[i + rand() % k] = random();

my\_DNA\_seq[i + rand() % k] = random();

}

std::random\_device rd;

std::mt19937 gen(rd());

std::uniform\_int\_distribution<int> dis(0, my\_DNA\_seq.length()-k-1);

for (int i = dis(gen); i < my\_DNA\_seq.length(); i = dis(gen))

{

std::string read\_str = "";

for (int j = 0; j < k; j++)

{

read\_str += my\_DNA\_seq[i + j];

}

short\_read.push\_back(read\_str);

if (short\_read.size() == n)

break;

}

}

My\_DNA와 short read를 생성하는 과정으로 k킬이당 0~2개씩 기존 레퍼런스 dna와 다르게 만든다. 이후 랜덤한 위치부터 k길이만큼의 문자열을 my\_DNA로부터 잘라 short\_read vector에 push\_back한다.

char ACTG::random()

{

std::random\_device rd;

std::mt19937 gen(rd());

std::uniform\_int\_distribution<int> dis(0, 3);

return actg[dis(gen)];

}

메르센 트위스트를 이용한 난수생성기로 ACTG중 하나의 문자열을 return한다.

void ACTG::restore()

{

restore\_seq = ref\_DNA\_seq;

int mismatch;

for (int i = 0; i < short\_read.size(); i++)

{

for (int j = 0; j < 1000000; j++)

{

mismatch = 0;

for (int x = 0; x < k; x++)

{

if (short\_read[i][x] != ref\_DNA\_seq[j + x])

mismatch++;

if (mismatch >= 2)

break;

if ((x == k - 1) && mismatch < 2)

{

for (int y = 0; y < k; y++)

{

restore\_seq[j + y] = short\_read[i][y];

}

}

}

}

}

}

DNA sequence를 복원하는 과정으로 short\_read를 brute force방식으로 ref\_DNA와 비교하여 mismatch가 2이하일경우 일치하는것으로 판단하고 일치하는 위치에 short\_read를 넣는다. 해당 방식을 모든 short\_read에 대해 반복하여 DNA를 복원한다.

void ACTG::compare()

{

int miss = 0;

for (int i = 0; i < 1000000; i++)

{

if (restore\_seq[i] != my\_DNA\_seq[i])

{

miss++;

}

}

std::cout << "일치율 : " << ((double)(1000000-miss) / (double)1000000) \* 100 << "%\n";

std::cout << "불일치 문자 개수 : " << miss << std::endl;

}

MY\_DNA와 복원된 restore\_seq를 비교하는 함수로 일치율을 백분율로 나타낸다.

반복을 줄인 방식

-일반적인 rand()함수와 나머지 연산자를 통해 랜덤을 계산하면 중복이 심하게 나온다. 이는 나머지 연산자가 가지는 문제점과 rand()함수가 특정 시드에 의해 생성되는 의사 난수이기 때문이다. 이를 해결하기 위해 균일하게 결과가 나오는 의사난수 생성기를 이용하였다. 모던 C++부터는 메르센트위스트를 이용한 의사난수 생성기를 지원한다. 이를 사용하면 난수가 균일하게 생성되기 때문에 중복이 확연하게 줄어든다. Short\_read를 생성할 때 난수생성기를 이용하여 랜덤한 위치에서 시작하여 길이만큼 자르는 과정을 하였는데 메르센트위스트를 이용한 난수생성기가 균일한 결과를 만들기 때문에 높은 일치율이 나올 수 있었다.

길이 k와 n을 동적으로 입력

-ACTG 클래스를 생성할 때 생성자가 k와n을 입력받아 초기화 할 수 있게 하였다. 상단 코드 참조.

걸린 시간과 missmatch허용 개수

* 길이 30, 40000개의 short read는 총 343초가 소요되고 약 96퍼센트의 일치율이 나왔고 길이60,30000개의 short read는 총 258초와 약 98퍼센트의 일치율이 나왔다. 두 경우 모두 2개의 missmatch를 허용하였다.

전체 소스코드 : [algorithm/Week10/Week10 at master · dsaf2007/algorithm (github.com)](https://github.com/dsaf2007/algorithm/tree/master/Week10/Week10)