**알고리즘**

알고리즘 설계 과제 보고서

[도시 설계 프로그램 설계 1차]

학번: 12151402

이름: 서주원

연락처: 010-3857-5403

이메일: [seozoo12@naver.com](mailto:seozoo12@naver.com)

**1. 개요**

(1) 설계의 목적

해당 프로그램의 설계는 인하 왕국의 모든 도시들을 자동차로 왕래할 수 있도록 도로들을 포장하는데 드는 최소 비용을 계산하기 위함입니다. 최소한의 비용으로 모든 도시들을 연결할 수 있는 루트를 찾는 방법은 최소 신장 트리 찾기가 있습니다. 따라서, 최소 신장 트리 이하 MST를 찾기 위해서 해당 프로그램의 설계가 필요합니다.

(2) 요구사항

프로그램을 설계 하기 위한 요구사항들이 다음과 같이 있습니다.

1. Kruskal’s algorithm을 이용합니다.

2. Union-Find 자료구조는 연결리스트표현(linked-list representation)으로 구현합니다.

3. 두 집합의 union은 크기가 더 작거나 같은 집합을 다른 집합의 tail에 연결하는 가중치-유니온 휴리스틱(weighted-union heuristic)을 이용합니다.

4. union 되는 두 집합의 크기가 동일하다면, 두 집합 중 leader가 더 작은 집합 즉, leader 도시의 번호가 더 작은 집합으로 union합니다.

5. 그래프는 자유롭게 구현 가능하지만 Union-Find를 forest 기반으로 구현하지 않습니다.

6. 도로를 포장하는 비용이 같다면 포장하려는 도로의 양쪽 끝 두 도시의 인구수의 합이 더 큰 비포장도로를 우선적으로 개선합니다.

7. 두 도시의 인구수의 합도 같을 경우, 양쪽 끝의 도시 번호 중 더 작은 번호 기리 비교해 더 작은 도시 번호를 포함하는 비포장도로를 먼저 개선합니다.

8. STL은 vector, list, 표준 입출력, algorithm 헤더의 sort만 허용합니다.

(3) 개발 환경

언어는 c++를 사용했고 visual studio 2019버전에서 제작했습니다.

**2. 필요한 자료구조 및 기능**

(1) 필요한 자료구조

해당 프로그램을 설계하기 위해서는 다음과 같은 자료구조들이 필요합니다.

1. Edge[]

- 그래프를 형성하는데 필요한 자료구조로 도로의 양 끝에 연결된 도시 번호 2개와 도로를 재포장하는데 필요한 비용을 저장하는 Edge structure를 삽입, 삭제하는 기능이 있습니다.

2, Vertex[]

- 그래프를 형성하는데 필요한 자료구조로 도시의 번호, 이름, 인구 수. 속한 집합의 도로들의 총 비용, 크기, 위치를 저장하는 Vertex structure를 삽입, 삭제하는 기능이 있습니다.

3. Vector

- Union-Find 자료구조를 구현하기 위해서 필요한 자료구조로 List로 구현된 여러 개의 disjoint set들을 관리하기 위해서 사용됩니다. Find 연산에서 자신이 속한 집합이 몇 번째에 속해 있는지 확인할 때 사용합니다. 리스트의 검색 속도가 느리다는 점을 극복하기 위해서 사용했습니다.

- 기능

1) void push\_back(list<int>)

4. List(Linked-list)

- Union-Find 자료구조를 구현하기 위해서 필요한 자료구조 입니다. c++ <list> container를 이용했습니다. 해당 <list> container는 doubly linked list로 구현되어 있습니다.

- 기능

1) void push\_back(int city\_num)

2) void splice(iterator, list<int>)

3) int front()

5. Union-Find

- Kruskal’s algorithm을 구현하기 위해서 필요한 자료구조 입니다.

- 기능

1). int myfind(int city\_number)

2). void myunion(int set\_position1, int set\_position2, int road\_cost)

3) void move\_leader(list<int>,int set\_position)

(2) 프로그램 기능

해당 프로그램의 주요 기능은 6가지가 있습니다.

1. 특정 도시가 포함된 연결리스트의 내부 정보 파악

Query로 문자 “N”이 들어가며, 추가적으로 도시 번호 v와 정수 k를 입력해야 합니다.

- 질의 형식: “N v k”

출력 형식으로는 “x y” 또는 “no exist”가 있습니다.

- “x”: 도시 v가 포함된 linked-list의 head로부터 k번째 도시의 번호입니다.

- “y”: 도시 v가 포함된 linked-list의 head로부터 k번째 도시명입니다.

- “no exist”: v가 포함된 linked-list의 head로부터 k번째 도시 정보가 존재하지 않을 때 출력입니다.

2. 특정 도시가 포함된 연결리스트의 크기 출력

Query로 문자 “L”이 들어가며, 추가적으로 도시 번호 v를 입력해야 합니다.

- 질의 형식: “L v”

출력 형식으로는 “S”가 있습니다.

- “S”: 도시 v가 포함된 linked-list의 총 element의 개수입니다.

3. Kruskal 알고리즘에서 하나의 반복 진행

Query로 문자 “I”가 들어갑니다.

- 질의 형식: “I”

출력 형식으로는 “ID S” 또는 “not union” 또는 “ID S v D”가 있습니다.

- “ID”: 해당 반복에서 union에 성공했다면, union된 새로운 집합의 set id(leader) 입니다.

- “S”: union된 새로운 집합의 크기입니다.

- “not union”: 만약 해당 반복에서 두 집합에 대해 union을 실행할 경우 cycle이 발생하면, “not union”을 출력하여 예외처리를 합니다.

- “ID S v D”: 만약 해당 반복에서 union에 성공하였고 MST를 찾았다면, “ID S v D”(ID, S, v, D는 각각 union된 새로운 집합의 set id(leader), union된 새로운 집합의 크기, 최종 완성된 MST 집합의 leader, MST의 모든 포장도로들의 길이의 합)를 출력하고 프로그램을 종료합니다.

4. 두 도시가 같은 집합에 속하는지 확인

Query로 문자 “F”가 들어가고 추가적으로 도시 번호1 V1, 도시 번호2 V2를 입력해야 합니다.

- 질의 형식: “F V1 V2”

출력 형식으로는 “True s” 또는 “False v w”가 있습니다.

- “True s”: 두 도시가 같은 집합일 경우, “True s”(s를 집합의 leader)을 출력합니다.

- “False v w”: 두 도시가 다른 집합일 경우, “False v w”(v, w는 각각 V1, V2를 포함하는 집합의 leader)를 출력합니다.

5. 임의의 도시를 포함하는 집합의 모든 포장도로의 길이

Query로 문자 “W”가 들어가고 추가적으로 노드 번호 v를 입력해야 합니다.

- 질의 형식: “W v”

출력 형식으로는 “D”가 있습니다.

- “D”: 도시 v를 포함하는 집합의 모든 포장도로들의 길이의 합 입니다.

6. 알고리즘의 나머지 단계 연속 수행 및 프로그램 종료

Query로 문자 “Q”가 들어갑니다.

- 질의 형식: “Q”

- “Q”: MST를 생성할 때까지 Kruskal’s algorithm을 진행하라는 질의입니다.

출력형식으로는 “v D”가 있습니다.

- “v”: MST의 leader입니다.

- “D”: MST의 모든 포장도로들의 길이의 합입니다.

7. Edge 배열 sorting

- Kruskal’s algorithm을 수행하기 위해서는 Edeg가 cost를 기준으로 오름차순으로 정렬되어야 합니다.

**3. 기능별 알고리즘 명세**

(1) 각 기능에 대한 알고리즘 설명(시간 및 공간 복잡도)

1. Vertex[], Edge[]

1) 삽입, 삭제

a. Vertex, Edge 자료 형을 index를 통해서 원하는 위치에 삽입 또는 삭제합니다.

b. 단순 index를 통한 접근이므로 시간 복잡도는 O (1) 입니다.

c. 이미 할당된 배열들을 이용하기 때문에 공간복잡도는 O (1) 입니다.

- Edge 기능

1) sort(Edge, Edge)

2. Vector

- 기능

1) void push\_back(list<int>)

a. Vector의 맨 뒤에 원소를 추가합니다. Vector에 구현되어 있는 내장 함수 입니다.

b. vector의 맨 뒤 index에 접근해 삽입하므로 시간복잡도는 O (1) 입니다.

c. vector크기가 n일 때, 공간복잡도는 O (n) 입니다.

3. List(Linked-list)

- 기능

1) void push\_back(int city\_num)

a. List의 맨 뒤에 데이터를 추가합니다. List container에 구현되어 있는 내장 함수입니다.

b. List의 맨 뒤 pointer에 접근해 삽입하므로 시간복잡도는 O (1) 입니다.

c. 하나의 노드를 생성하므로 공간복잡도는 O (1) 입니다.

2) void splice(iterator, list<int>)

a. Union-Find에서 두 개의 집합을 한 개의 집합으로 union하는 myunion연산을 할 때 사용하는 함수입니다. List container에 구현되어 있는 내장 함수입니다.

b. Splice함수의 1번째 parameter인 iterator에는 insert하는 list의 tail주소를 2번째 parameter인 list<int>는 삽입할 list를 넣습니다.

c. parameter로 들어온 list의 head pointer를 삽입되는 list의 tail pointer를 나타내는 iterator에 연결하기 때문에 시간 복잡도는 O (1) 입니다.

d. 이미 할당되어 있는 두개의 리스트의 pointer만 활용하기에 공간 복잡도는 O (1) 입니다.

3) int front()

a. list의 head에 위치한 원소를 반환하는 기능입니다. 즉, 각 집합의 leader 도시 번호를 반환합니다. List container에 구현되어 있는 내장 함수입니다.

b. list의 head pointer에 한 번 접근하면 되기 때문에 시간 복잡도는 O (1) 입니다.

c. list의 pointer만 활용하므로 공간 복잡도는 O (1) 입니다.

4. Union-Find

- 기능

1). int myfind(int city\_number)

a. 도시 번호가 들어오면 그 도시 번호가 속해있는 집합의 위치를 반환하는 함수입니다.

b. 도시 번호가 들어오면 이미 할당된 Vertex[city\_number] 에 접근해 해당 도시의 집합 위치를 받아옵니다.

c. 단순 index 접근이기 때문에 시간 복잡도는 O (1) 입니다.

d. 이미 할당된 city 배열을 이용하기 때문에 공간 복잡도는 O (1) 입니다.

2) void move\_leader(list<int>,int set\_position)

a. 가중치-유니온 휴리스틱(weighted-union heuristic)을 실행하는 함수입니다.

b. 요구사항에 명세한 조건대로 집합의 개수가 더 적은 집합을 집합의 개수가 더 큰 집합으로 옮깁니다. (두 집합의 개수가 같다면 leader 도시 번호가 더 큰 집합을 더 작은 집합으로 이동)

c. parameter에는 이동될 집합과 새롭게 만들어지는 집합의 위치가 들어옵니다. 이동되는 집합의 원소들이 가진 집합의 위치를 새롭게 만들어진 집합의 위치로 업데이트 합니다.

d. 이동되는 list의 원소 개수에 dependent한 시간 복잡도를 가지고 있습니다. 그래프의 전체 vertex의 개수가 n일 때, 특정 리스트의 한 원소의 갱신 횟수는 최악의 경우 O(log n)번 입니다. 그러므로 O(log(n))의 시간복잡도를 가지게 됩니다.

e. 이미 할당된 city 배열을 이용하기 때문에 공간 복잡도는 O (1) 입니다.

3). int myunion(int set\_position1, int set\_position2, int road\_cost)

a. 두 개의 서로 다른 집합을 하나의 집합으로 union하는 함수입니다.

b. 도로에 연결된 두 도시가 속해 있는 집합의 위치와 도로의 비용을 parameter로 받아 옵니다.

c. set\_position1, set\_position2 parameters를 이용해서 도로에 연결된 두 도시가 속한 집합의 리더를 가져옵니다.

d. 가져온 leader 도시의 city\_number인 leader1, leader2를 이용해서 각 집합의 크기를 비교합니다. 요구사항에 명세한 조건에 맞게 union되는 type을 결정합니다.

e. leader1의 집합 개수가 leader2의 집합 개수보다 많거나 leader1,2의 집합 개수가 같을 때, leader1이 leader2보다 작은 수라면 type은 1이고 leader2의 집합 개수가 leader1의 집합 개수보다 많거나 leader1,2의 집합 개수가 같을 때, leader2가 leader1보다 작은 수라면 type은 2입니다.

f. Vertex[leader1], Vertex[leader2]에 접근해서 각 집합의 road\_cost를 불러와서 현재 이 도로의 cost를 합한 total\_cost를 계산합니다.

g. type이 1이라면 leader2의 집합에 move\_leader()함수를 실행하고 splice()함수를 이용해서 leader2의 집합을 leader1의 집합으로 연결시켜줍니다. leader1이 새로운 집합의 leader가 되었으므로 Vertex[leader1]에 합쳐진 집합의 크기와 비용을 update해줍니다. 새로 형성된 집합의 leader 도시의 번호를 나타내는 res에 leader1을 저장합니다.

h. type이 2라면 type1과 반대의 leader로 똑같이 실행합니다.

i. res 변수를 반환합니다.

j. 해당 기능은 (1)-3-2)의 splice 기능과 (1)-4-2)의 move\_leader 기능, 대입,비교 연산을 사용합니다. Vertex의 총 개수가 n이라면, Splice는 O (1), move\_leader는 O(log(n)), 대입, 비교 연산은 O(1) 이므로 해당 기능의 시간복잡도는 move\_leader 기능에 dependent하며 O(log(n)) 입니다.

k. 이미 할당되어 있는 배열과 리스트들을 이용하므로 공간복잡도는 O (1) 입니다.

(2) 프로그램 기능

1. 특정 도시가 포함된 연결리스트의 내부 정보 파악

Query로 문자 “N”이 들어가며, 추가적으로 도시 번호 v와 정수 k를 입력해야 합니다.

a. 도시 번호 v와 정수 k를 입력 받습니다.

b. (1)-4-1)의 myfind 기능을 이용해서 도시 번호 v의 도시가 속한 집합의 위치를 가져옵니다.

c, 3-3)의 front 기능을 이용해서 도시가 속한 집합의 leader 도시의 번호를 가져옵니다.

d. 만약, 정수 k가 도시 번호가 v인 도시가 속한 집합의 크기보다 같거나 크다면 “no exist” 를 출력합니다.

e. 만약, 정수 k가 도시 번호가 v인 도시가 속한 (집합의 크기/2) 보다 작다면, iterator를 집합의 head로 잡고 k번 만큼 tail쪽으로 이동해서 해당 list의 k번 째 위치한 도시의 번호를 가져옵니다.

f. 만약, 정수 k가 도시 번호가 v인 도시가 속한 (집합의 크기/2) 보다 크다면, iterator를 집합의 tail로 잡고 (집합의 크기 – k) 만큼 head쪽으로 이동해 해당 list의 k번 째 위치한 도시의 번호를 가져옵니다.

g. x에 도시의 번호를 대입하고 city[x]에 접근해서 해당 도시의 이름을 y에 대입해 출력 형식에 맞게 출력합니다.

h. 리스트는 순차 접근을 해야하므로 head에서 원하는 위치까지 가거나 tail에서 원하는 위치까지 가야합니다. 그래프의 vertex가 총 n개라면, 해당 기능의 시간 복잡도는 O (n) 입니다.

i. 이미 할당된 리스트를 탐색하는 것이므로 공간 복잡도는 O (1) 입니다.

2. 특정 도시가 포함된 연결리스트의 크기 출력

Query로 문자 “L”이 들어가며, 추가적으로 도시 번호 v를 입력해야 합니다.

a. 도시 번호를 v를 입력 받습니다.

b. (1)-4-1)의 myfind 기능을 이용해서 도시 번호 v의 도시가 속한 집합의 위치를 가져옵니다.

c. (1)-3-3)의 front 기능을 이용해서 도시 번호 v의 도시가 속한 집합의 leader 도시의 번호를 가져옵니다.

d. city[leader\_city\_num]에 접근해서 해당 집합의 크기를 가져와 S에 대입합니다.

e. 출력 형식에 맞게 출력합니다.

f. myfind, front 기능 모두 O(1)의 시간복잡도를 가지고 대입 연산만 하므로 시간 복잡도는 O(1)입니다.

g. 이미 할당된 배열과 list들에 접근하므로 공간 복잡도는 O (1) 입니다.

3. Kruskal 알고리즘에서 하나의 반복 진행

Query로 문자 “I”가 들어갑니다.

1) void kruskal(char ch)

Kruskal’s algorithm을 진행하는 함수입니다.

a. road 배열에서 Edge 하나를 가져와서 해당 Edge의 양 끝 도시의 번호인 v1, v2 그리고 도시의 비용 cost를 가져옵니다.

b. (1)-4-1)의 myfind 기능을 이용해서 도로 양 끝의 도시들이 속한 집합의 위치를 불러옵니다.

c. 두 집합의 위치가 같다면 두 도시는 같은 집합에 속해 있다는 뜻이므로 cycle을 형성하지 않기 위해서 union 하지 않고 “not union” 을 출력합니다.

d. 두 집합의 위치가 다르다면 (1)-4-3)의 myunion 기능을 이용해서 두 집합을 union 해줍니다.

e. union을 했으므로 union을 몇 번 했는지 확인하는 union\_cnt를 증가 시킵니다.

f. ID에 새롭게 형성된 집합의 leader 도시 번호를 넣고 city[ID]에 접근해 집합의 크기를 대입합니다.

g. 만약, ch값이 ‘I’일 때, union\_cnt값이 (N-1) 이면 MST를 찾았다는 뜻이므로 “ID S v D” 의 형식으로 출력합니다. 그렇지 않다면 “ID S” 형식으로 출력합니다.

h. 만약, ch값이 ‘Q’일 때, union\_cnt값이 (N-1) 이면 “ID D” 형식으로 출력합니다.

i. 그래프의 vertex가 n개라고 할 때, myfind는 O (1), myunion은 O(log(n))의 시간복잡도를 가지므로 해당 기능의 시간 복잡도는 O(log(n))입니다.

j. 이미 할당된 배열과 리스트를 이용해서 연산을 하기 때문에 공간 복잡도는 O (1) 입니다.

k. (2)-3-1)의 kruskal을 한 번만 진행하므로 O(log(n))의 시간 복잡도를 가지고 O (1)의 공간 복잡도를 가집니다.

4. 두 도시가 같은 집합에 속하는지 확인

Query로 문자 “F”가 들어가고 추가적으로 도시 번호1 V1, 도시 번호2 V2를 입력해야 합니다.

a. 도시 번호를 v1, v2를 입력 받습니다.

b. (1)-4-1)의 myfind 기능을 이용해서 도시 번호 v1, v2의 도시가 속한 집합의 위치를 가져옵니다.

c. 두 도시가 속한 집합의 위치가 같다면 (1)-3-3)의 front 기능을 이용해서 s에 집합의 leader 도시 번호를 삽입하고 “True s”의 형식으로 출력합니다.

d. 두 도시가 속한 집합의 위치가 다르다면 (1)-3-3)의 front 기능을 이용해 v, w에 두 집합의 leader 도시 번호를 각각 삽입하고 “False v w”의 형식으로 출력합니다.

e. myfind, front 기능만을 사용하므로 시간 복잡도는 O (1) 입니다.

f. 이미 할당된 배열과 리스트를 이용하므로 공간 복잡도는 O (1) 입니다.

5. 임의의 도시를 포함하는 집합의 모든 포장도로의 길이

Query로 문자 “W”가 들어가고 추가적으로 노드 번호 v를 입력해야 합니다.

a. 도시 번호를 v를 입력 받습니다.

b. (1)-4-1)의 myfind 기능을 이용해서 도시 번호 v의 도시가 속한 집합의 위치를 가져옵니다.

c. (1)-3-3)의 front 기능을 이용해서 도시 번호 v의 도시가 속한 집합의 leader 도시 번호를 가져옵니다.

d. city[leader\_city\_num]에 접근해 해당 집합의 모든 포장도로들의 길이의 합을 D에 저장합니다.

e. 형식에 맞게 출력합니다.

e. myfind, front 기능만을 사용하므로 시간 복잡도는 O (1) 입니다.

f. 이미 할당된 배열을 이용하므로 공간 복잡도는 O (1) 입니다.

6. 알고리즘의 나머지 단계 연속 수행 및 프로그램 종료

Query로 문자 “Q”가 들어갑니다.

- 질의 형식: “Q”

- “Q”: MST를 생성할 때까지 Kruskal’s algorithm을 진행하라는 질의입니다.

a. (2)-3-1)의 kruskal 기능을 MST가 발견될 때 까지 실행됩니다.

b. 그래프의 vertex의 개수가 n개이고 edge의 개수가 m개 일 때, dense한 그래프일 경우에는 edge의 개수 만큼인 O(m)만큼 sparse한 그래프일 경우에는 vertex의 수 n만큼 (2)-3-1)의 kruskal 기능을 수행해야하므로 O(nlog(n)+m)입니다. 따라서 전체적인 시간복잡도는 O(nlog(n)+m)입니다.

c. (2)-3와 같이 공간 복잡도는 O (1) 입니다.

7. Edge 배열 sorting

- Edge 배열을 요구사항에 명세한 기준대로 sorting을 진행합니다.

a. Edge structure를 입력 받아 Edge 배열에 삽입합니다.

b. c++ <algorithm> 헤더의 sort 기능을 사용합니다.

c. 기준에 맞게 sorting 하기 위해서 compare 함수를 만들어 줍니다.

d. sort(Edge.begin(), Edge.end(), compare)를 이용해 Edge를 조건에 맞게 sorting합니다.

e. 정렬하는 개체가 n개 일 때, <algorithm> 헤더의 sort는 nlog(n)의 수행 시간을 가집니다. Edge의 개수가 m개 일 때, 해당 기능의 시간 복잡도는 mlog(m)입니다.

f. Edge 배열 만큼의 공간을 사용하기 때문에 공간 복잡도는 O (m) 입니다.

**4. 인터페이스 및 사용법**

(1) 간단한 사용법 설명

본 프로그램은 다음과 같이 사용할 수 있습니다.

1. 프로그램을 실행하면 첫 번 째 줄에는 도시들의 개수 N(2<=N<=50,000), 도로들 사이에 존재하는 비포장도로의 개수 M(1<=M<=500,000), 질의의 수 q(1<=q<=25,000)를 공백으로 구분해서 입력합니다.

2. 이후 N개의 줄을 통해 도시 정보가 “도시 번호, 도시명, 도시의 인구수”과 같이 한 줄에 각 필드를 공백으로 구분해 입력합니다.

- 도시 번호: 1~999,999 사이의 정수 (기준 키. 유일함)

- 도시명: 알파벳 20글자 이내 (예: “HongGilDong”)

- 도시의 인구수: 1~1,999,999 사이의 정수

3. 이후 M개의 줄을 통해 비포장도로의 정보가 “도시 1, 도시 2, 두 도시 간의 도로 길이”와 같이 한 줄에 각 필드를 공백으로 구분해 입력합니다.

- 도시 1: 도시 번호

- 도시 2: 도시 번호

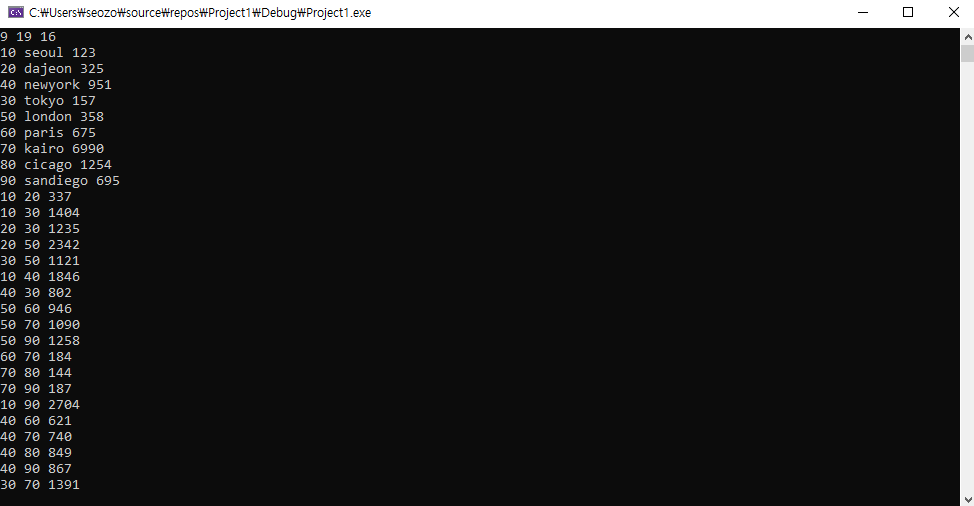
- 두 도시 간의 도로 길이: 1~10,000 사이의 정수

4. 이후 q개의 줄을 통해서 질의가 들어갑니다. q개의 질의에 대해서는 “2. 필요한 자료구조 및 기능” 부분의 ‘기능’ 부분을 참고하시면 좋을 것 같습니다.

(2) 실행 화면 캡쳐 포함

본 프로그램의 실행 화면입니다.

1. 사용법 1, 2, 3번에 해당하는 내용입니다.



2.

1) 사용법 4번의 Query “I”, “F”, “L”, “W”, “N”, “Q”의 기능을 캡쳐 했습니다.

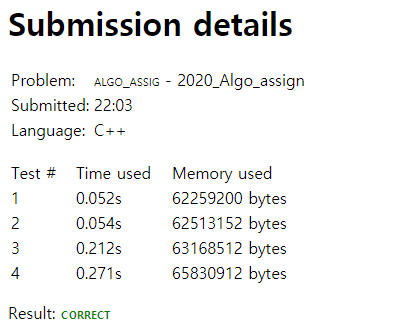


2) 사용법 4번의 “I” 기능에 대한 추가 출력 화면입니다.



**5. 평가 및 개선 방향**

**- 수행**



(1) 구현 및 개발에 있어 특이 사항

- Vertex 배열을 도시 번호의 범위만큼 지정했고, Edge를 배열로 도로의 개수 범위만큼 지정 해서 Vertex와 Edge 배열 자료구조를 생성했습니다.

- 조건 2를 맞추기 위해서 compare 함수를 따로 생성해서 sorting할 때, 여러 조건에 맞게 sorting이 될 수 있게 했습니다.

- union-find 구조에서 서로 다른 집합이 union될 때, 기존 집합이 있던 공간에 대한 처리가 없기 때문에 union-find 자료구조는 vertex의 크기만큼 메모리를 차지해서 union이 되어 집합이 줄어도 계속해서 vertex 크기만큼의 메모리를 차지합니다.

(2) 본 결과의 장점

- linked-list를 이용해 구현했기 때문에 삽입, 삭제 연산 시, pointer만 조정해 주면 되기 때문에 다른 자료구조보다 삽입, 삭제가 빠릅니다. 이 부분을 이용해서 myunion을 구현했습니다.

- weighted union heuristic을 이용해서 union될 때, 리스트의 각 원소들의 leader 정보를 update하는 수행 시간이 기존 O(n)에서 O(log(n))으로 줄었습니다.

- 각 리스트들을 담은 구조로 vector를 이용했기 때문에 list 즉, 집합에 O(1)만에 탐색해 접근이 가능합니다.

- vertex 배열을 도시 번호 범위만큼 지정해서 생성했기 때문에, 도시 번호를 배열의 index로 바로 접근해 도시에 대한 정보를 O(1) 시간 만에 받아 올 수 있습니다.

- spares한 그래프일 때, Prim algorithm의 O(n^2)보다 빠른 O(nlog(n)+m)의 시간 복잡도를 가집니다.

(3) 본 결과의 단점

- dense한 그래프일 때, m의 개수가 n^2의 근사하게 될 경우에는 Prim algorithm과 같은 O(n^2)의 시간 복잡도를 가집니다.

- 각 집합이 list로 구성되어 있기 때문에 탐색을 진행할 때, 순차 접근을 이용해야 하기 때문에 집합 내의 원소 탐색 기능은 O(n)의 시간 복잡도를 가집니다.

- vertex 배열을 도시 번호 범위만큼 지정해서 생성했기 때문에, 들어오는 vertex의 수가 적어도 도시 번호 범위만큼의 vertex 배열을 생성합니다. Edge 배열 또한 마찬 가지로 kruskal’s algorithm이 가장 효과를 볼 수 있는 spare한 그래프에서 공간 낭비가 심할 수 있습니다.

(4) 향후 개선 방향

- Vertex와 Edge를 저장하는 자료구조를 공간을 정적으로 할당하는 배열로 만들지 않고 동적으로 할당하는 자료 구조인 vector를 이용해서 생성해 Edge가 적을 때의 공간 복잡도를 줄일 수 있습니다.

- union-find에서 union을 하고 나서 splice된 집합이 있던 메모리를 삭제하는 자료구조를 이용해서 구현해 공간 낭비를 줄일 수 있도록 개선해야 합니다.