Dijkstra Algorithm을 이용한 네비게이션 구현

1. 개요

- 설계 목적 : 주어진 지역정보를 이용하여 쿼리에 따라 출발지에서 목적지까지의 최단 경로 및 거리를 계산하는 프로그램을 짠다.

- 요구 사항 :
  - C++를 사용하여 구현해야 하며 서버환경에서 2초이내에 수행되어야 한다.
  - 주어진 쿼리에 맞게 출력해야 하며, 예외를 처리한다.
  - 출발지와 목적지가 주어졌을 때 경로를 출력한다.
  - 출발지와 목적지가 주어졌을 때 최단거리를 출력한다.
  - Dijkstra 알고리즘을 사용한다.
  - Tree set의 개수를 출력한다.
  - 인접 리스트 기반의 자료구조를 사용한다.
  - 표준입출력을 사용한다.
  - 주석을 달아야 한다.
- 개발 환경 : (개인) OS – Window 10(64bit)/ IDE-Visual Studio 2017
  (서버) OS – Ubuntu 18.04(64bit)/ compiler: gcc 7.3.0

- 기본 사항 : C++ 14에 맞는 문법 사용. C++ STL사용

2. 필요한 자료구조 및 기능

- 필요한 자료구조
  - 구조체 인접 리스트
  - Hash 배열,
  - Vector,
  - Priority queue(우선순위 큐)
- 기능
  - 구조체 인접 리스트는 지역 정보가 저장된 구조체들을 인접 리스트의 그래프로 구성.
  - Hash배열은 vertex들의 노드를 0부터 N까지 설정하고 값을 지역 번호와 1:1 대응 시켜주는 배열
  - Vector는 C++ STL로 다양한 체크와 변수 저장에 사용되는 배열
  - 우선순위 큐는 Fringe를 저장할 Min heap

3. 기능별 알고리즘 명세

```cpp
#include <iostream>
#include <algorithm>
#include <queue>
#include <stack>
#include <math.h>
#include <string>
#include <climits>
#include <bitset>
#include <functional>
using namespace std;
struct city {
    int cityNum;
    string cityName;
    vector< pair<int, int> > neighbor;
    int isFlood;
};
#define over 1000000
vector<int> visit;
vector<int> arr(1000000);
vector<city> list;
```

- 구조체 city는 입력 받은 도시의 이름, 번호, 침수 여부 그리고 인접한 도시의 정보를 저장하는 구조체이다.
- 구조체 내의 neighbor벡터는 2차원 벡터로 neighbor[i]는 i번째 vertex의 인접 vertex의 집합이며 인접 리스트 형태로 push back 된다.
- Visit 배열은 tree set에 삽입된 vertex를 체크하는 배열이다. Dijkstra알고리즘에서 모든 vertex는 tree에 많아야 한번만 들어가므로 0,1로 구분한다.
- arr배열은 도시 번호와 vertex의 노드 number를 1:1로 대응하는 배열이다. (배열 전체가 bijective는 아니다. cityNum -> node_index로의 bijective를 의미한다) 문제의 조건에서 지역 정보는 유일함으로 1:1을 만족한다. Arr[i] = node_index로 정의되며 i 는 지역번호이다. 지역 번호를 인덱스로 사용하는 경우보다 시간-공간 면에서 유리하다. 그 외의 값은 쓰레기 값이 들어있다.
- 구조체 배열로 입력된 도시 정보의 배열이며 여기서의 인덱스가 node_index로 arr배열과 1:1대응된다.

```
22  int dijkstra(city A, city B, vector<int> &path, int &lastDist) {
23      priority_queue< pair<int, int>, vector< pair <int, int> >, greater< pair<int, int> > > Minpq;
24      visit.assign(list.size(), 0);
25      vector<pair<int, int>> pp(list.size(), make_pair(-1, INT_MAX));
26      vector<int> dist(list.size(), INT_MAX);
27      vector<int> fringe(list.size());
28      dist[arr[A.cityNum]] = 0;
29      visit[arr[A.cityNum]] = 1;
30      int treeSize = 1;
31      int y = arr[A.cityNum];
32      for (int i = 0; i < A.neighbor.size(); i++) {
33          Minpq.push(make_pair(A.neighbor[i].second, A.neighbor[i].first));
34          pp[arr[A.neighbor[i].first]].first = arr[A.cityNum];
35          pp[arr[A.neighbor[i].first]].second = A.neighbor[i].second;
36          fringe[arr[A.neighbor[i].first]] = 1;
37      }
```

- Dijkstra 알고리즘이다. 파라미터를 보면 city A는 시작점의 구조체이다. B는 도착점의 구조체이다. Path는 경로를 저장할 배열이며, lastDist는 최단경로를 받을 변수이다.
- Minpq는 우선순위 큐로 Min heap이 구현되어있는 STL이다. 인자를 살펴보면 구현하기위한 자료구조로 vector를 선택했고 greater 함수는 Min heap으로 최소값을 top에 두는 것을 의미한다. 또한 배열의 원소는 pair로 들어가는데 (dist , city number)로 들어가며 dist를 우선적으로 정렬하고 dist가 같을 시 city number가 낮은 쪽이 우선순위가 높다. 이는 조건과 일치하기 때문에 더 손볼 것은 없다.
- pp배열은 pair를 원소로 갖는 배열로 i번째 vertex의 predecessor와 source부터의 거리를 원소로 갖는다. 즉, (node num of predecessor, distance from source) 가 원소로 들어가며, spanning tree를 구성할 때 부모 노드를 기억하고 목적지 까지 다달았을 때 back tracking을 하여 경로를 구하기 위해 사용된다.
- fringe배열로 i번째 vertex(node)가 fringe에 들어왔었는지를 체크하는 배열이다. 모든 vertex는 많아야 한번 fringe에 삽입되고 삭제되므로 한 번의 체크면 충분하다. (bool로 충분)
- dist[i]배열은 시작점에서 i번째 노드까지의 거리를 업데이트 하는 배열이다. 시작점을 s라할 때 dist[s] := 0으로 한다.
- 처음 tree에 시작점이 있음으로 tree의 크기를 1로잡는다. Tree의 이웃한 vertex를 모두 fringe, Minpq(min-heap)에 삽입하고 predecessor역시 업데이트한다.

```
46      while (!Minpq.empty()) {
47          int u = arr[Minpq.top().second];
48          int w = Minpq.top().first;
49          y = pp[arr[Minpq.top().second]].first;
50          if (visit[u]) { Minpq.pop(); continue; }
51          Minpq.pop();
52          treeSize++;
53          visit[u] = 1;
54          if (dist[u] > w) { dist[u] = w; }
55          for (int i = 0; i < list[u].neighbor.size(); i++) {
56              if (!visit[arr[list[u].neighbor[i].first]]) {
57                  if (fringe[arr[list[u].neighbor[i].first]]) {
58                      if (pp[arr[list[u].neighbor[i].first]].second > list[u].neighbor[i].second + dist[u]) {
59                          Minpq.push(make_pair(list[u].neighbor[i].second + dist[u], list[u].neighbor[i].first));
60                          pp[arr[list[u].neighbor[i].first]].first = u;
61                          pp[arr[list[u].neighbor[i].first]].second = list[u].neighbor[i].second + dist[u];
62                      }
63                  }
64                  else {
65                      Minpq.push(make_pair(list[u].neighbor[i].second + dist[u], list[u].neighbor[i].first));
66                      pp[arr[list[u].neighbor[i].first]].first = u;
67                      pp[arr[list[u].neighbor[i].first]].second = list[u].neighbor[i].second + dist[u];
68                      fringe[arr[list[u].neighbor[i].first]] = 1;
69                  }
70              }
71          }
72          if (u == arr[B.cityNum]) break;
73      }
74      if (treeSize == 1 || pp[arr[B.cityNum]].first == -1) return -1;
75      int t = arr[B.cityNum];
76      while (t != -1) {
77          path.push_back(t);
78          t = pp[t].first;
79      }
80
81      lastDist = dist[arr[B.cityNum]];
82      return treeSize;
83  }
```

- Fringe가 빌 때 까지 계속해서 루프를 돈다. 최악의 경우 도착지점이 제일 마지막에 pop 될 수 있다. 이 경우에는 한 점에서 모든 경로의 최단 경로를 탐색하는 것이 한 점과 특정 다른 점과의 최단 경로를 탐색하는 것보다 어렵지 않다.
- U는 fringe에서 pop할 vertex의 노드 번호다. W는 시작점부터 그 노드까지의 거리이다. Y는 그 노드의 predecessor이다.
- If(visit[u]){...}이 부분은 decreaseKey() 함수를 분리한 것인데, priority queue의 경우 iterator를 사용한 임의의 access가 불가능하기 때문에 node업데이트를 위해서는 업데이트할 노드를 찾을 때까지 pop하고 또 pop한 원소를 저장할 배열까지 필요하다. 이런 과정을 효율적으로 하기 위해서, priority queue에는 항상 최소 원소만 top으로 오기 때문에 중복된 노드에 대해서도 최적의 값을 pop한다는 점을 이용해, priority queue를 업데이트 하는 것이 아니라 predecessor만 업데이트 하는 방식을 이용하였다. 즉 priority queue에는 노드를 중복해서 넣는 것이다. 결국 tree set에는 그 노드에 대한 최적의 값이 삽입되고 dijkstra알고리즘에서는 트리에는 그 노드가 많아야 한번 들어감으로, tree에 이미 있는데 priority queue에서 pop되는 노드 정보는 최적이 아님을 알 수 있음으로 그냥 버린다. 원래

decreaseKey를 노드를 업데이트 하는 식으로 한다면 $O(\log n)$ 이지만 $O(1)$에 수행 하게끔 할 수 있다.

- fringe에서 pop된 node는 tree set에 삽입된것이므로 tree의 사이즈를 1키워준다.
- 삽입된 노드에 대해서 인접한 vertex를 fringe에 삽입하는데, fringe에 이미 있는 경우라면, 그 노드까지의 거리를 비교하여 predecessor를 업데이트한다. (pp배열을 바꾸는 것이다.) Fringe에 없는 경우는 fringe에 삽입하고 fringe에 삽입되었음을 체크한다.
- 만약 pop한 요소가 목적지라면 더 이상 tree를 확장하지않고 루프를 종료한다.
- 만약 tree의 크기가 1이거나 목적지의 predecessor가 없는 경우는 경로가 없는 경우이다.
- 도착한 점에서 predecessor를 읽기 시작하여 시작 노드까지 간다. 읽어 들인 노드는 path배열에 저장한다. 시작 노드의 predecessor는 -1로 세팅 되어있다.
- 모든 수행을 마친 후에는 dist배열에는 시작점에서 i 까지의 최소 거리가 저장되어 있음으로 dist[목적지 노드 번호]를 거리에 저장한다.
- 트리의 사이즈를 함수의 리턴값으로 준다.
- 노드들이 우선 순위 큐에 삽입되고 삭제되는 연산만 일어남으로 최악의 경우 수행 시간은 $O(\mathrm{mlog}\, n)$ 이다. M: edge수 N: 노드 수

```
77  int main() {
78      ios::sync_with_stdio(false);
79      cin.tie(0);
80      cout.tie(0);
81
82      int area_num;
83      string area_name;
84      int flood;
85
86      int n, m, q;
87      cin >> n >> m >> q;
88
89
90      list.resize(n);
91
92      int dist, area_1, area_2;
93
94      for (int i = 0; i < n; i++) {
95          cin >> area_num >> area_name >> flood;
96          list[i].cityName = area_name;
97          list[i].cityNum = area_num;
98          list[i].isFlood = flood;
99          arr[area_num] = i;
100     }
101     for (int i = 0; i < m; i++) {
102         cin >> area_1 >> area_2 >> dist;
103         if (list[arr[area_1]].isFlood == 0 && list[arr[area_2]].isFlood == 0) {
104             list[arr[area_1]].neighbor.push_back(make_pair(area_2, dist));
105             list[arr[area_2]].neighbor.push_back(make_pair(area_1, dist));
106         }
107     }
```

- Main 함수이다.
- Ios::sync_with_stdio(false)는 cout과 cin의 입출력을 c의 표준입출력과 동기화하는 작업을 false로 세팅하는 것이다. 이 과정이 다소 느리기 때문이다.
- Cin.tie(0), cout.tie(0)은 i/o stream이 수행될 때 입출력 버퍼를 지우고 수행되는데 이 상태를 tied라하고 이 과정이 느리기 때문에 untied해준다. multithread작업과 같이 입출력이 동시에 일어나는 경우만 아니라면 크게 문제될 것은 없다.
- N은 노드의 수, m은 edge의 수,q는 쿼리의 수다.
- 각각의 정보를 받고 hashing도 해준다.
- Edge를 연결해줄 때 해당 vertex의 침수 여부를 확인하고 침수되어 있다면 해당 vertex로 가는 경로는 없다고 봐도 무방하다.

```
108     char c;
109     for (int i = 0; i < q; i++) {
110         int ans, num_tree; vector<int> path;
111         cin >> c >> area_1 >> area_2;
112         num_tree = dijkstra(list[arr[area_1]], list[arr[area_2]], path, ans);
113         if (c == 'A') {
114             if (ans>over || num_tree == -1) cout << "None\n";
115             else {
116                 cout << num_tree << " " << ans << " " << list[arr[area_1]].cityName << " "
117                     << list[arr[area_2]].cityName << "\n";
118             }
119         }
120         else if (c == 'B') {
121             if (ans>over || num_tree == -1) cout << "None\n";
122             else {
123                 cout << num_tree << " ";
124                 for (int i = path.size() - 1; i >= 0; i--)
125                     cout << list[path[i]].cityNum << " ";
126                 cout << "\n";
127             }
128         }
129     }
130
131 }
```

- 'A' 쿼리에 대해서는 tree의 사이즈와, 최단 거리, 출발지 이름, 목적지 이름을 출력한다.
- 'B' 쿼리에 대해서는 tree의 사이즈와, 최단 경로를 번호로 순서대로 출력한다. Path에 back tracking을하면서 저장했음으로 배열의 뒤부터 출력한다
- 두 경우 모두 경로가 over 일 경우 경로가 없다고 가정하며, 또 tree size가 -1인 경우 역시 경로가 없어서 -1이 리턴된것임으로 "None"을 출력한다.

4. 평가 및 개선 방향

- 구현한 알고리즘의 장점
  - Hash 배열을 이용하여 지역 번호에 맞춰 배열을 설계하지않아 그 만큼 시간과 메모리를 절약할 수 있었다.
  - Decreasekey() 를 $O(1)$ 에 수행하였다.
  - Decreasekey() 가 줄어든 만큼 push pop은 증가했다.

- 구현한 알고리즘의 단점

Hash를 사용한만큼 노드 번호와 지역 번호 등 다양한 Notation으로 다소 복잡한 코드가 되었다.

- 향후 개선 방향

코드의 변수 명 설정과 Notation이 여러 번 중첩되는 경우에는 주석 처리로 헷갈리는 것을 조금은 완화 할 수 있을 것이다. 또한 fringe를 저장할 자료구조를 피보나치 힙을 사용한다면 연산속도를 조금 더 개선할 수 있다.

5. 기타 (선택)

- 구현 및 개발에 있어 특이 사항

처음에는 decreaseKey를 priority queue에서 모두 팝하면서 업데이트 하는 식으로 했는데, 이럴 경우 시간 초과가 나게 된다. 그래서 따로 predecessor를 업데이트 해둘 배열을 만들어서 체크하고자 했다.