자료구조및프로그래밍 HW13

C211171 최후락

2025 12 18

1 개요

지하철 노선도에는 여러 개의 호선이 존재하며, 각 호선은 여러 개의 역으로 구성되어 있다. 일부 역은 서로 다른 호선에 동시에 속해 있으며, 이 경우 해당 역에서는 환승이 가능하다. 각 역 사이를 이동하는 데에는 일정한 시간이 소요되며, 같은 호선 내에서의 이동 시간과 환승에 소요되는 시간은 서로 다르게 주어진다.

이번 과제에서는 이러한 지하철 노선 정보를 입력으로 받아, 두 개의 역이 주어졌을 때 이동에 필요한 최단 경로와 소요 시간을 계산하는 문제를 다룬다. 또한, 두 사람이 서로 다른 출발 역에서 출발할 경우, 도착 시간의 차이가 최소가 되도록 만날 수 있는 중점 역을 함께 구한다.

이를 위해 지하철 노선도를 그래프 구조로 표현하고, 각 역을 정점으로, 역 사이의 이동을 간선으로 모델링한다. 이동 시간은 간선의 가중치로 처리되며, 환승이 가능한 역의 경우 추가적인 이동 시간이 고려된다. 이와 같은 모델을 기반으로 최단 경로 탐색 알고리즘을 적용하여 문제에서 요구하는 결과를 계산한다.

- 과제를 진행한 환경
- 노선도 입력 처리
- 문제 (a): 최단 경로 탐색
- 문제 (b): 중점 역 탐색
- 출력처리
- 테스트 결과
- 어려웠던 점

2 과제를 진행한 환경

과제를 수행할때 Windows 환경에서 WSL2 기반 Ubuntu를 사용했다. 프로그램의 구현 및 실험에 사용된 시스템 사양은 다음과 같다.

- CPU: Intel® Core™ Ultra 5 226V (2.10 GHz)
- RAM: 16 GB
- 그래픽: Intel® Arc™ Graphics (내장)
- 저장장치: 477 GB SSD
- 운영체제: Windows 64-bit (x64)
- 개발 환경: WSL2 Ubuntu
- 컴파일러: g++ (C++17)

또한 아래와 같은 명령어를 사용해 실행했다.

```
g++ -O2 -std=c++17 hw13.cpp subway.cpp -o hw13
```

3 노선도 입력 처리

문제 (a)와 (b)를 해결하기에 앞서, 입력으로 주어진 데이터를 기반으로 지하철 노선도를 구성해야 한다. 입력 파일은 하나의 호선 전체를 순서대로 제공하지 않고, 서로 인접한 두 역의 연결 관계를 나열한 형태로 구성되어 있다.

따라서, 입력 데이터를 그대로 사용하기보다는, 각 역과 역 사이의 연결 정보를 그래프 구조로 변환하여 저장하는 방식이 필요하다. 각 역을 하나의 노드로, 역 사이의 이동을 간선으로 표현하여 지하철 노선도를 모델링하였다. 이때 하나의 역은 (호선 번호, 역 이름)의 쌍으로 구분하여 관리하였다.

3.1 loadStations()

loadStations() 함수는 노선도 입력 파일을 읽어 지하철 노선도를 그래프 형태로 구성하는 역할을 한다. 입력 파일의 첫 줄에는 연결 정보의 개수가 주어지며, 이후 각 줄에는 서로 연결된 두 역의 호선 번호와 역 이름이 주어진다.

각 입력 줄에 대해 두 역을 노드로 등록하고, 두 노드 사이에 이동 가능한 간선을 추가한다. 같은 호선에 속한 경우에는 이동 시간을 60초로, 서로 다른 호선에 속한 경우에는 이동 시간을 30초로 설정하였다. 이 과정을 반복하며 전체 노선도를 만든다.

Listing 1: loadStations

```
bool Subway::loadStations(const std::string& file) {
  std::ifstream fin(file);
  if (!fin.is_open()) return false;

  int m;
  fin >> m;

  for (int i = 0; i < m; i++) {
    int l1, l2;
    std::string s1, s2;
    fin >> l1 >> s1 >> l2 >> s2;

    int a = getId(l1, s1);
    int b = getId(l2, s2);

    int w = (l1 == l2) ? 60 : 30;
    addEdge(a, b, w);
  }

  for (auto &kv : stationGroup) {
    const std::vector<int>& ids = kv.second;
    for (int i = 0; i < (int)ids.size(); i++) {
      for (int j = i + 1; j < (int)ids.size(); j++) {
        addEdge(ids[i], ids[j], 30);
      }
    }
```

```
        }

        return true;
    }
```

3.2 findId()와getId()

역 정보는 문자열 형태의 역 이름과 호선 번호로 주어지지만, 프로그램 내부에서는 이를 그대로 사용하기보다 정수 ID로 변환하여 관리하는 것이 효율적이다. 이를 위해 `findId()`와 `getId()` 함수를 사용하였다.

findId() 함수는 이미 등록된 역의 ID를 찾는 역할을 하며, 해당 역이 존재하지 않는 경우 -1을 반환한다. getId() 함수는 역이 아직 등록되지 않은 경우 새로운 노드를 생성하고 ID를 부여한 뒤 이를 반환한다. 이러한 방식으로 모든 역은 내부적으로 고유한 ID를 가지게 된다.

Listing 2: findId, getId

```
int Subway::findId(int line, const std::string& station)
    const {
    auto it = idMap.find(makeKey(line, station));
    if (it == idMap.end()) return -1;
    return it->second;
}

int Subway::getId(int line, const std::string& station)
   {
    std::string key = makeKey(line, station);
    auto it = idMap.find(key);
    if (it != idMap.end()) return it->second;

    int id = (int)nodes.size();
    idMap[key] = id;
    nodes.push_back({line, station});
    adj.push_back(std::vector<Edge>());
    stationGroup[station].push_back(id);
    return id;
}
```

3.3 addEdge()

addEdge() 함수는 두 역 사이의 연결 관계를 그래프의 간선으로 추가하는 역할을 한다. 지하철 노선은 양방향 이동이 가능하므로, 간선은 양방향으로 저장하였다.

간선에는 해당 구간을 이동하는 데 필요한 시간이 가중치로 함께 저장되며, 이 정보는 이후 최단 경로 탐색 과정에서 사용된다.

Listing 3: addEdge

```
1     void Subway::addEdge(int a, int b, int w) {
2       adj[a].push_back({b, w});
3       adj[b].push_back({a, w});
4     }
```

3.4 환승 처리: stationGroup 사용

실습시간에는 환승역을 하나의 노드로 구성하는 방식을 제시하고 말씀하셨으나, 이를 그대로 구현할 경우 하나의 노드 안에 호선 정보, 환승 여부, 이동 시간 등을 모두 포함해야 하므로 구조체가 과도하게 복잡해지는 문제가 있었다. 특히 환승역과 일반 역을 구분하는 것과 만약 엄청많은 호선이 겹치는 역이라면 그 한개의 역 때문에 모든 환승이 아닌역의 노드까지 무의미하게 커질 수 있다는 생각이 들었다.

따라서 과제를 하며 우선 프로그램이 정상적으로 동작하는 것을 목표로 하여, 각 호선의 역을 별도의 노드로 구성하고 같은 이름을 가지는 역들 사이를 환승을 의미하는 30초 가중치의 간선으로 연결하는 방식을 선택하였다. 이 방법은 환승을 하나의 노드로 처리한 경우와 동일한 이동 시간 계산 결과를 얻을 수 있으며, 구현 또한 어렵지 않게 해낼 수 있었다.

4 문제 (a): 최단 경로 탐색

문제 (a)는 입력으로 주어진 두 역 사이를 이동할 때, 소요 시간이 최소가 되는 경로와 그 시간을 구하는 것이다. 지하철 노선도는 역을 정점으로, 역 사이의 이동을 간선으로 표현한 가중치 그래프로 볼 수 있으며, 각 간선의 가중치는 이동에 소요되는 시간이다.

그래프에서 같은 호선 내에서 인접한 역 사이의 이동은 60초, 환승이 필요한 경우는 30초의 비용이 들도록 가중치를 설정하였다.

이와 같이 모든 간선의 가중치가 음수가 아닌 조건에서 최단 경로를 구해야 하므로, 다익스트라 알고리즘을 사용하여 문제를 해결하였다.

과제설명 파일에 있는 다익스트라 알고리즘은 시작 정점으로부터 다른 모든 정점까지의 최단 거리를 효율적으로 계산할 수 있는 알고리즘이다. 한 출발 역에서 모든 역까지의 최소 이동 시간이 필요하므로, 여러 알고리즘 중에 다익스트라 알고리즘을 골랐다.

4.1 dijkstra()

dijkstra() 함수는 시작 역을 기준으로 모든 역까지의 최단 이동 시간을 계산하는 역할을 한다. 각 역까지의 현재까지 알려진 최소 시간을 dist 배열에 저장하고, 경로 복원을 위해 이전에 방문한 역 정보를 prev 배열에 저장한다.

우선순위 큐를 사용하여 아직 방문하지 않은 역 중에서 현재까지의 이동 시간이 가장 짧은 역을 먼저 선택하도록 하였다. 선택된 역에서 인접한 역으로 이동했을 때 더 짧은 시간이 계산되는 경우, 해당 값을 갱신하는 방식으로 알고리즘을 진행한다.

Listing 4: dijkstra

```
1   void Subway::dijkstra(int start, std::vector<int>& dist,
        std::vector<int>& prev) {
2     int n = (int)nodes.size();
3     dist.assign(n, INF);
4     prev.assign(n, -1);
5
6     using P = std::pair<int,int>;
7     std::priority_queue<P, std::vector<P>, std::greater<P
          >> pq;
8
9     dist[start] = 0;
10    pq.push({0, start});
11
12    while (!pq.empty()) {
13      auto [d, u] = pq.top();
14      pq.pop();
15
16      if (d != dist[u]) continue;
```

```
17
18          for (auto &e : adj[u]) {
19            int v = e.to;
20            int nd = d + e.w;
21            if (nd < dist[v]) {
22              dist[v] = nd;
23              prev[v] = u;
24              pq.push({nd, v});
25            }
26          }
27        }
28      }
```

4.2 recover()

최단 경로 탐색이 완료된 후에는 prev 배열에 저장된 이전 역 정보를 이용하여 실제 이동 경로를 복원한다. 도착 역에서 시작 역까지 거꾸로 추적한 뒤,이를 다시 역순으로 정렬하여 출력에 사용할 경로를 생성한다.

환승으로 인해 동일한 역 이름이 연속해서 등장할 수 있으므로, 출력 시에는 연속으로 중복되는 역 이름은 하나만 출력하도록 처리하였다.

Listing 5: recover

```
1     void Subway::recover(int start, int end,
2     const std::vector<int>& prev,
3     std::vector<std::string>& outNames) {
4       outNames.clear();
5
6       std::vector<int> ids;
7       int cur = end;
8       while (cur != -1) {
9         ids.push_back(cur);
10        if (cur == start) break;
11        cur = prev[cur];
12      }
13      std::reverse(ids.begin(), ids.end());
14      if (ids.empty() || ids.front() != start) return;
15
16      std::string last = "";
17      for (int id : ids) {
18        const std::string& nm = nodes[id].station;
19        if (outNames.empty() || nm != last) {
20          outNames.push_back(nm);
21          last = nm;
22        }
23      }
24    }
```

4.3 shortestPath()

shortestPath() 함수는 문제 (a)의 전체 흐름을 담당하는 함수이다. 입력으로 받은 출발 역과 도착 역을 내부 ID로 변환한 뒤, 다익스트라 알고리즘을 실행하여 최단 이동 시간을 계산한다.

이후 recover() 함수를 통해 이동 경로를 복원하고, 역 이름만을 포함하는 경로를 결과로 반환한다. 최종적으로는 계산된 최단 이동 시간을 초 단위로 반환한다.

Listing 6: shortestPath

```
1     int Subway::shortestPath(int l1, const std::string& s1,
          int l2, const std::string& s2, std::vector<std::
          string>& pathStations) {
2       pathStations.clear();
3
4       int start = findId(l1, s1);
5       int end   = findId(l2, s2);
6       if (start == -1 || end == -1) return -1;
7
8       std::vector<int> dist, prev;
9       dijkstra(start, dist, prev);
10
11      if (dist[end] >= INF) return -1;
12
13      recover(start, end, prev, pathStations);
14      return dist[end];
15    }
```

5 문제 (b): 중점 역 탐색

문제 (b)는 서로 다른 두 출발 역에서 출발한 두 사람이 이동할 때, 도착 시간의 차이가 최소가 되도록 만날 수 있는 역을 찾는 문제이다. 단순히 총 이동 시간이 가장 짧은 역을 선택하는 것이 아니라, 두 사람의 도착 시간 차이를 우선적으로 고려해야 한다는 점에서 문제 a와는 다른 기준이 필요하다.

이를 해결하기 위해, 각 출발 역을 시작점으로 하여 모든 역까지의 최단 이동 시간을 각각 계산한 뒤, 모든 역을 대상으로 두 시간 값을 비교하는 방식을 사용하였다.

중점 역을 선택할 때는 다음과 같은 우선순위를 적용하였다.

1. 두 사람의 도착 시간 차이 $|T_1 - T_2|$가 최소인 역
2. 위 조건이 같은 경우, 두 사람 중 더 늦게 도착하는 시간 $\max(T_1, T_2)$가 최소인 역
3. 위 두 조건도 같은 경우, 역 이름의 사전순이 가장 앞서는 역

5.1 findMidStation()

findMidStation() 함수는 문제 b의 중점 역을 탐색하는 역할을 한다. 먼저 두 출발 역 각각을 시작점으로 하여 다익스트라 알고리즘을 실행하고, 모든 역까지의 최단 이동 시간을 계산한다.

이후 모든 역을 순회하면서, 각 역에 대해 두 사람의 이동 시간을 비교한다. 앞서 정의한 세 가지 기준에 따라 가장 적합한 역을 하나 선택하고, 해당 역 이름과 두 사람의 도착 시간을 결과로 반환한다.

처음에는 전수조사를 하지 않고 어떤 방식으로 구현이 가능한지 생각해보았으나 중점 역의 후보는 특정 조건으로 미리 제한할 수 없다는 결론에 도달해 최단 경로 계산 이후에는 모든 역을 대상으로 전수 조사를 수행하였다. 역의 개수가 많지 않은 문제 조건에서는 이 방식이 구현이 간단하면서도 충분히 효율적이라고 판단하였다.

Listing 7: findMidStation

```
1    bool Subway::findMidStation(int l1, const std::string&
         s1, int l2, const std::string& s2, std::string& meet,
          int& bigT, int& smallT) {
2      meet = "";
3      bigT = -1;
4      smallT = -1;
5
6      int aStart = findId(l1, s1);
7      int bStart = findId(l2, s2);
8      if (aStart == -1 || bStart == -1) return false;
9
10     std::vector<int> distA, prevA, distB, prevB;
11     dijkstra(aStart, distA, prevA);
```

```
12        dijkstra(bStart, distB, prevB);
13
14        std::map<std::string, int> minA, minB;
15
16        for (int i = 0; i < (int)nodes.size(); i++) {
17          if (distA[i] < INF) {
18            if (!minA.count(nodes[i].station) || distA[i] <
                  minA[nodes[i].station])
19            minA[nodes[i].station] = distA[i];
20          }
21          if (distB[i] < INF) {
22            if (!minB.count(nodes[i].station) || distB[i] <
                  minB[nodes[i].station])
23            minB[nodes[i].station] = distB[i];
24          }
25        }
26
27        int bestDiff = INF, bestMax = INF;
28        std::string bestName = "";
29
30        for (auto &kv : minA) {
31          const std::string& name = kv.first;
32          if (!minB.count(name)) continue;
33
34          int ta = kv.second;
35          int tb = minB[name];
36          int diff = (ta > tb) ? ta - tb : tb - ta;
37          int mx   = (ta > tb) ? ta : tb;
38
39          if (diff < bestDiff ||
40          (diff == bestDiff && mx < bestMax) ||
41          (diff == bestDiff && mx == bestMax && name <
                bestName)) {
42            bestDiff = diff;
43            bestMax = mx;
44            bestName = name;
45          }
46        }
47
48        if (bestName == "") return false;
49
50        meet = bestName;
51        int ta = minA[bestName];
52        int tb = minB[bestName];
53        if (ta >= tb) { bigT = ta; smallT = tb; }
54        else { bigT = tb; smallT = ta; }
55
56        return true;
57      }
```

6 출력처리

과제를 채점 시 출력 결과를 기준 출력과 비교하는 diff 방식으로 평가한다고 언급되어 있으므로, 출력 형식이 정확히 일치하는 것이 중요하다. 따라서 문제에서 요구한 순서와 형식을 그대로 유지하여 출력하도록 구현하였다.

최단 경로의 경우, 경로에 포함된 역 이름을 한 줄에 하나씩 출력한 뒤, 마지막 줄에 총 소요 시간을 (분:초) 형식으로 출력한다. 중점 역 또한 역 이름을 먼저 출력하고, 이후 두 사람의 도착 시간을 동일한 형식으로 출력하도록 하였다.

6.1 formatTime()

시간 출력 형식을 맞추기 위해 초 단위로 계산된 시간을 (분:초) 형태의 문자열로 변환하는 formatTime() 함수를 별도로 작성하였다.
초가 한 자리 수인 경우에는 앞에 0을 붙여 항상 동일한 출력 형식을 유지하도록 하였다.

Listing 8: formatTime

```
1     std::string Subway::formatTime(int sec) {
2       int m = sec / 60;
3       int s = sec % 60;
4
5       std::string res = std::to_string(m) + ":";
6       if (s < 10) res += "0";
7       res += std::to_string(s);
8
9       return res;
10    }
```

6.2 main()

main() 함수는 프로그램의 전체 실행 흐름을 담당한다. 노선도 파일과 입력 파일을 받아, 노선도 파일을 통해 지하철 그래프를 구성한 뒤 입력 파일로부터 출발 역과 도착 역 정보를 읽어온다.

이후 문제 a에 해당하는 최단 경로 탐색을 수행하고, 경로와 총 소요 시간을 출력한다. 다음으로 문제 b에 해당하는 중점 역 탐색을 수행하여, 중점 역 이름과 두 사람의 도착 시간을 문제에서 요구한 형식에 맞게 출력한다.

Listing 9: main

```
1     int main(int argc, char* argv[]) {
2       if (argc != 3) return 1;
3
4       Subway sw;
5       if (!sw.loadStations(argv[1])) return 1;
6
7       ifstream fin(argv[2]);
```

```
        int l1, l2;
        string s1, s2;
        fin >> l1 >> s1 >> l2 >> s2;


        vector<string> path;
        int total = sw.shortestPath(l1, s1, l2, s2, path);
        for (string &x : path) cout << x << "\n";
        cout << Subway::formatTime(total) << "\n";


        string meet;
        int bigT, smallT;
        sw.findMidStation(l1, s1, l2, s2, meet, bigT, smallT);
        cout << meet << "\n";
        cout << Subway::formatTime(bigT) << "\n";
        cout << Subway::formatTime(smallT) << "\n";

        return 0;
    }
```

7 테스트 결과

7.1 입력 예시

다음은 테스트에 사용한 입력 파일의 일부이다.

```
6
1 A 1 B
1 B 1 C
2 A 2 D
2 D 2 C
1 B 2 B
1 C 2 C
```

7.2 실행 결과

위 입력을 사용하여 프로그램을 실행한 결과는 다음과 같다.

```
A
B
C
2:00
B
1:00
1:00
```

코드를 작성한 의도와 과제명세서의 출력예시에 적합하게 출력이 나오는 것을 확인할 수 있다.

8 어려웠던 점

과제를 진행하면서 가장 먼저 고민했던 부분은 입력으로 주어진 데이터를 어떻게 지하철 노선도 형태로 구성할 것인가였다.
입력 파일은 하나의 호선을 순서대로 제공하지 않고, 서로 인접한 두 역의 연결 관계를 한 줄씩 나열하는 방식이었기 때문에, 이를 그대로 그래프로 변환하는 방법을 고민하게 되었다. 결과적으로 입력 파일의 각 줄을 그래프에서 하나의 간선으로 생각하고, 노드 간의 연결을 하나씩 추가해 나가는 방식으로 구현하였다.

역 정보를 내부적으로 관리하는 과정에서도 시행착오가 있었다. 처음에는 한 호선을 통째로 하나의 배열로 묶어 관리하는 방법을 떠올렸으나, 환승 처리나 다른 호선과의 연결을 고려했을 때 현실적으로 구현이 어렵다는 것을 알게 되었다. 이후 호선 번호와 역 이름을 함께 사용하여 각 역을 고유하게 식별하는 방식이 필요하다는 점을 깨달았고, 이를 위해 findId()와 getId() 함수를 사용하여 문자열 정보를 내부 ID로 변환해 관리하였다. 이 과정에서 관련 자료를 찾아보며 아이디어를 참고하였다.

환승역 처리 또한 구현 과정에서 가장 어려웠던 부분 중 하나였다. 환승역을 하나의 노드로 구성하는 방식은 구조적으로 깔끔할 수 있으나, 구현 과정에서 처리해야 할 정보가 많아지고 로직이 복잡해지는 문제가 있었다. 이에 대해서는 앞선 섹션에서 설명한 바와 같이, 각 호선의 역을 별도의 노드로 두고 같은 이름을 가지는 역들 사이를 환승 간선으로 연결하는 방식으로 해결하였다.

최단 경로를 구한 뒤 경로를 복원하는 과정에서도 문제가 발생하였다. 환승역을 호선별로 분리하여 노드로 구성했기 때문에, 경로 복원 시 동일한 역 이름이 연속해서 출력되는 현상이 있었다. 이를 해결하기 위해 recover() 함수에서 연속으로 중복되는 역 이름은 하나만 출력하도록 수정하였다.

중점 역을 찾는 문제의 경우,초기에는 전수 조사를 사용하지 않고 보다 효율적인 방법을 찾고자 고민하였다. 그러나 두 출발 지점으로부터의 이동 시간을 동시에 고려해야 하며, 후보 역을 사전에 제한할 수 있는 명확한 기준을 찾기 어려웠다. 결과적으로 모든 역을 대상으로 조건을 비교하는 전수 조사 방식이 가장 확실하다고 판단하여 이를 사용하였다.

마지막으로 출력 처리 부분에서도 시행착오가 있었다. 과제 채점 방식이 출력 결과를 기준 출력과 비교하는 diff 방식이라는 점을 확인한 뒤, 출력 형식이 조금이라도 어긋나면 오답이 될 수 있다는 것을 알게 되었다. 이를 방지하기 위해 시간 출력을 전담하는 formatTime() 함수를 별도로 작성하여, 항상 동일한 형식의 출력이 이루어지도록 처리하였다.