MST: Kruskal & Prim

1115 이정우

목차

1 알고리즘 개요

1.1 Kruskal 알고리즘

Kruskal 알고리즘이란, 가중치가 있는 무방향 그래프에서 최소 신장 트리(MST)를 찾는 대표적인 방법으로, 모든 간선을 가중치 오름차순으로 정렬한 뒤, 가장 비용이 낮은 간선부터 하나씩 선택하면서 사이클이 생기지 않도록 Union-Find로 관리한다. 이 과정을 거치면 모든 정점을 최소 비용으로 연결할수 있으며, 불필요한 간선이 제외되어 최적의 연결 구조를 보장한다.

1.2 Prim 알고리즘

Prim 알고리즘은 그래프에서 최소 신장 트리(MST)를 찾는 방법 중 하나로, 시작 정점을 기준으로 매번 현재 트리에 연결된 간선들 중 가장 가중치가 작은 간선을 선택하여 새로운 정점을 추가하는 방식으로 진행된다. 즉, 트리에 속하지 않은 정점 중에서 이미 선택된 정점과 연결된 가장 작은 간선만 골라 확장해 나가며, 모든 정점을 포함할 때까지 반복한다.

2 자료구조 상세

2.1 변수와 ADT

2.1.1 Kruskal 변수

- `parent: any[size]`
  - 각 원소가 가진 대푯값을 표시하는 배열이다.

- `pq: tuple({int, int, int})`
  - 두 정점과 그 사이의 거리를 저장하는 우선순위 큐 배열이다.

2.1.2 Kruskal ADT

- `my_union() -> int`
  - 두 노드를 병합하는 함수이다.
- `find() -> int`
  - 노드의 대푯값을 반환한다.

2.1.3 Prim 변수

- `parent: any[size]`
  - 각 원소가 가진 대푯값을 표시하는 배열이다.

- `pq": tuple({int, int, int})`
  - 두 정점과 그 사이의 거리를 저장하는 우선순위 큐 배열이다.

2.1.4 Prim ADT

- `my_union() -> void`
  - 두 노드를 병합하는 함수이다.
- `find() -> int`
  - 노드의 대푯값을 반환한다.

2.1.5 Exception

- 특별한 예외 상황은 존재하지 않는다.

2.2 구현 상세 및 핵심 코드

2.2.1 `my_union()`

a와 b를 매개변수로 받은 뒤 두 노드의 대푯값을 찾는 연산 및 사이클 검사를 하는 함수이다. 두 노드의 뿌리가 같아 사이클이 된다면 False, 아니면 True를 반환한다.

```
int my_union(int a, int b){
  int RootX = find(a);
  int RootY = find(b);
  if(RootX == RootY){
      return 0;
  }
  else if(RootX<RootY){
      parent[RootY] = RootX;
  }
  else{
```

```
        parent[RootX] = RootY;
    }
    return 1;
}
```

2.2.2 find()

노드의 대푯값을 반환하는 기능을 가진 함수이다. 노드 a의 대푯값이 자신을 가르키지 않으면 계속해서 재귀적으로 자신의 뿌리를 찾고 그렇지 않으면 최종값은 그 노드의 대푯값이 되므로 그 값을 반환한다.

```
int find(int a){
    if(parent[a]==a)return a;
    return parent[a] = find(parent[a]);
}
```

2.3 복잡도 분석

- 위 함수를 바탕으로 Union-Find를 하게 된다면 처음에는 O(n)의 시간 복잡도를 가지지만 계속해서 parent의 값이 갱신되므로 경로를 찾는 연산 횟수가 낮아지기 때문에 나중에는 O(1)의 시간복잡도를 가지게 된다. 이를 아커만 함수 역함수로 부르며, O(a(n))으로 표현된다.

2.4 복잡도 분석

- 위 함수를 바탕으로 Union-Find를 하게 된다면 처음에는 O(n)의 시간 복잡도를 가지지만 계속해서 parent의 값이 갱신되므로 경로를 찾는 연산 횟수가 낮아지기 때문에 나중에는 O(1)의 시간복잡도를 가지게 된다. 이를 아커만 함수 역함수로 부르며, O(a(n))으로 표현된다.

2.5 전체 코드

```
#include <stdio.h>

int parent[1000001]={0,};

int find(int a){
    if(parent[a]==a)return a;
    return parent[a] = find(parent[a]);
}
```

```
void my_union(int b, int c){
    int RootX = find(b);
    int RootY = find(c);
    if(RootX!=RootY){
        if(RootX>RootY)parent[RootX] = parent[RootY];
        else parent[RootY] = parent[RootX];
    }
}

int main(){
    int n, m;
    scanf("%d %d", &n, &m);
    for(int i=1; i<=n; i++){
        parent[i] = i;
    }
    for(int i=1; i<=m; i++){
        int a, b, c;
        scanf("%d %d %d", &a, &b, &c);
        if(a==0){
            my_union(b, c);
        }
        else{
            if(find(b)==find(c))printf("YES\n");
            else printf("NO\n");
        }
    }
    return 0;
}
```

3 활용 방안

1. 서버/네트워크 연결 체크
    - Union-Find를 활용하여 연결된 네트워크를 빠르고 쉽게 확인할수있다.

4 인상깊었던 점

이전에는 DFS, BFS, 다익스트라, 플로이드 워셜, 벨만-포드 등 최단경로, 즉 정점과 정점 사이의 최단거리를 초점에 뒀다면 이번에는 이 노드가 다른노드와 연결되있는지 확인하는 자료구조라 새로웠다. 또한 단순히

같은 경로에 있다는것이 아니라 집합이라는 수학 개념을 활용하여 자료구조를 정의했다는것이 인상깊었다.

5 참고문헌

유니온 파인드(Union-Find)