#algorithms #study

그래프 기초

그래프란?

- 그래프는 간선, 노드, 가중치로 이루어진 자료구조.
- 이해를 돕기 위해 자료구조에서 나오는 트리를 그래프로 생각하고 특징을 뽑아보자.
  - 방향 : 트리는 간선에 방향이 있다. 최상위 노드를 기준으로 하위 노드로 이동하기 때문이다. 단, 그래프 기준으로 트리를 분석하면 방향이 없는데 이건 추후 고급 알고리즘 부분에서 배우도록 하겠다.
  - 비순환 : 트리는 부모 노드에서 자식 노드로 이동한다. 이 때, 만약 자식 노드에서 이어지는 간선이 부모 노드를 가리키게 되면 순환이 발생한다. 트리 구조는 무조건 부모 노드에서 간선이 분화되어 자식 노드로 이동하며 자식 노드의 자식이 부모 노드일 수 없다. 그렇기에 트리는 비순환 구조다.
  - 가중치 : 트리는 간선에 값을 부여하지 않는다. 즉, 간선 자체는 무료 도로고 관광지인 노드에만 값이 매겨져 있는 셈이다. 그렇기에 가중치는 0이다.
- 그래프는 간선, 노드, 가중치만 있으면 되는 자료구조이다. 방향, 순환, 가중치 값 등은 그래프를 어떻게 만드느냐에 따라 달라진다. 즉, 트리가 그래프에서 특수한 조건이 적용된 자료구조인 셈이다.

그래프 구현

- 배열(행렬)
  - 각 행은 현재 노드 위치를 나타내고 열은 현재 노드에서 간선이 뻗어서 연결된 노드를 의미한다. 그리고 각 행, 열이 속한 곳에 들어간 값은 간선의 가중치 값이다.

```
          1
          /\
         2  3
        /\  /\
       4 5  6 7
// 숫자는 노드 번호를 의미하고 가중치는 전부 1이라 가정한다. 또한, 간선에 방향이 없다.
// 노드가 총 7개이며 각 번호에 맞춰서 인덱스를 넣기 위해 아래와 같이 N + 1 배열 구성
int graph[8][8];

// 기존에 설정한 대로 그래프 만들기
void makeGraph() {
    // 간선에 방향이 없기에 1 -> 2도 가능하고 2 -> 1도 가능하다. 가중치는 1이니까 배열 값은 1로 넣는다. 간선이 연결되지 않은 노드들의 가중치 값은 당연히 0.
    graph[1][2] = 1; graph[2][1] = 1;
    graph[2][4] = 1; graph[4][2] = 1;
    graph[2][5] = 1; graph[5][2] = 1;
    graph[3][6] = 1; graph[6][3] = 1;
    graph[3][7] = 1; graph[7][3] = 1;
```

```
}

// 순회함수. 단순히 연결된 노드가 무엇인지 확인만 할 것이기에 반복문을 쓰겠다.
void rotation() {
    for (int row = 1; row < 8; i++) {
        for (int col = 1; col < 8; col++) {
            // 간선이 이어져 있으면 누구와 누가 이어져 있는 지 출력해서 알려준다.
            if (graph[row][col])
                print("노드 {row}가 {col}과 이어져 있습니다.");
        }
    }
}
```

- 배열 표현의 장단점
  - 장점
    - 배열 정의하고 순회하기 쉽다. 노드 개수만큼 배열 원소 늘리고 순회는 이중 반복문 돌리면서 가중치가 있는 간선만 조회하면 되니까.
    - 특정 노드와 연결된 간선의 가중치를 탐색하는 건 O(1)이다. 왜냐하면 열 번호가 곧 행 노드와 연결된 열 노드 번호고 해당 번호의 값을 조회하면 가중치가 몇인지 바로 알 수 있으니까.
  - 단점 : 간선이 연결되지 않은 노드들의 관계도 가중치 0으로 들어가 있기에 공간복잡도만 커짐. 그리고 순회하려면 간선이 연결되지 않은 노드들도 한 번 씩 거쳐가므로 시간복잡도 상승.
- 인접 리스트
  - 인접 리스트는 행 뒤의 열을 연결 리스트로 붙이기에 당연하게도 열의 길이가 행마다 다르다. 그렇기에 인접 리스트의 한 열에 <연결된 노드 번호, 간선 가중치 값>이 한 묶음으로 들어가야 한다.

```
// 그래프 형태는 배열에서 쓴 예시와 같다.
// 인접 리스트을 선언한다. 수도 코드이므로 실제 구현 시에는 본인이 주로 쓰는 프로그래밍 언어에서 인접 리스트를 지원하는 라이브러리가 무엇이 있는 지 찾아보기를 바란다.
list graph[8];

void makeGraph() {
    // 노드 1을 기준으로 하면 가중치 1인 간선이 각각 노드 2번과 3번에 연결되어 있다.
    graph[1].add({2, 1});
    graph[1].add({3, 1});
    // 무방향이므로 반대쪽도 연결할 것
    graph[2].add({1, 1});
    graph[3].add({1, 1});

    graph[2].add({4, 1}); graph[4].add({1, 1});
    graph[2].add({5, 1}); graph[5].add({2, 1});
    graph[3].add({6, 1}); graph[6].add({3, 1});
    graph[3].add({7, 1}); graph[7].add({3, 1});
}
```

```
void rotation() {
    // 행은 배열처럼 이어져 있으니까 그냥 순서대로 본다
    for (int row = 1; row < 8; row++) {
        // 인접 리스트 특성 상 열의 개수는 제각각이다. 각 행에 대한 열 크기 파악.
        for (int col = 1; col < graph[row].size(); col++) {
            // 배열과 달리 연결되지 않은 노드는 인접 리스트에 넣지 않기 때문에 그냥 출력
한다.
            // 해당 열에 존재하는 값 중에서 첫번째 값이 노드 번호이므로 첫번째 값을 출력
한다.
            print("{row}는 {graph[row][col].first}와 이어져 있습니다.");
        }
    }
}
```

- 인접 리스트의 장단점
  - 장점 : 인접 행렬에 비해서 공간 복잡도가 낮고 순회할 때의 시간 복잡도도 낮다.
  - 단점 : 탐색은 O(E)다. 왜냐하면 행 노드에 연결된 각 열의 인덱스 번호가 노드 번호가 아니기 때문이다. 결국, 특정 행 노드에 연결된 특정 열 노드 번호를 찾으려면 행 노드에 연결된 열 값을 전부 확인해 볼 수 밖에 없다.

그래프 탐색

- 그래프를 탐색하는 방식은 크게 깊이 우선 방식과 너비 우선 방식으로 나뉜다. 즉, 트리에서 전위 순회하듯이 가장 깊은 곳까지 내려가는 방식으로 값을 찾을 것이냐. 아니면 부모 노드 -> 자식 노드(동일한 부모를 둔 자식 노드 전부 확인) 순으로 인접한 노드를 우선적으로 탐색할 것이냐.
- 깊이 우선 탐색(DFS, depth-first search)
  - 깊이 우선 탐색은 현재 노드를 확인 -> 왼쪽 자식 노드 -> 오른쪽 자식 노드 순으로 탐색을 거친다. 즉, 트리에서 볼 수 있는 전위 순회 방식으로 탐색을 시도한다.
  - 각 연결된 트리와 노드마다 최종적으로 도달할 수 있는 깊이가 다르므로 반복문으로 들어가는 건 쉽지 않다. 그렇기에 방문 거점을 쌓아 놓고 나중에 더 파고들 경로가 없을 때 쌓아 놓은 방문 거점들 중에서 가장 최근에 방문한 거점으로 되돌아간다.
  - 즉, LIFO 구조인 스택 자료구조 방식과 깊이가 들쭉날쭉일 때 사용하는 재귀 방식이 더해진 것. 좀 더 짚어보자면 백트래킹 기법이 들어갔다 보면 된다.
    - 애초에 재귀 자체가 메모리 내에 자기 자신을 호출하며 스택을 쌓아가는 격이니 재귀도 스택 구조다.

```
// 배열 그래프.
int graph[12][12];

// 방문한 거점인 지 확인용.
bool visited[12];

void DFS(int parentNode) {
    // 방문한 노드면 빠진다.
```

```
    if (visited[parentNode])
        return ;
    // 방문하지 않았으면 지금 방문한 걸로 확인
    visited[parentNode] = true;
    for (int childNode = 0; i < 12; i++) {
        // 부모 노드와 자식 노드가 연결되어 있으면 자식 노드로 이동한다.
        if (graph[parentNode][childNode]) {
            DFS(childNode);
        }
    }
}
```

- 너비 우선 탐색(BFS, breadth first search)
  - 현재 노드에서 가장 가까운 노드(즉, 차수가 1인 이들)를 먼저 방문하고 이들을 다 방문했으면 가까운 노드들로 거점을 옮겨서 그 다음 노드들을 방문하는 식이다.
  - 현재 노드 -> 차수가 1인 노드 확인 -> 확인한 노드로 거점을 옮겨서 거기서 차수가 1인 노드 확인 . 차수가 가까운 노드부터 탐색 우선권이 주어지는 점에서 FIFO인 큐 자료구조로 구현을 한다.

```
// 배열 그래프
int graph[12][12];

void BFS() {
    // 거점 방문 여부 확인
    bool visited[12][12];

    // 방문 예정 노드들을 넣어둘 큐를 선언한다.
    queue q;

    // 가장 먼저 방문할 노드를 사전에 큐에 넣는다.
    q.add(1);
    // 방문할 노드가 없으면 탐색을 종료한다.
    while (!q.empty()) {
        // 큐에서 방문할 노드를 가져오고 큐에서 노드를 제거
        int curNode = q.pop();

        // 노드 방문 표시
        visited[curNode] = true;

        for (int nextNode = 0; nextNode < 12; nextNode++) {
            // 간선이 있고 방문하지 않은 노드면 방문 예정 노드로 큐에 삽입한다.
            if (graph[curNode][nextNode] && !visited[nextNode]) {
                q.add(nextNode);
            }
        }
```

```
    }
}
```

DFS와 BFS 응용

- DFS는 백트래킹 기법이 들어간 만큼 전 노드 순회하기 편하다. 그렇기에 최단 경로보다는 모든 가능한 해 찾기나 그래프에서 순환 구조를 찾을 때 쓰인다.
- BFS는 자기와 가장 가까운 노드부터 찾는 점에서 차수가 짧은 노드들을 우선 탐색한다. 그렇기에 모든 간선의 가중치가 동일한 그래프에서 특정 출발지에서 목적지까지의 최단 경로를 찾을 때 좋다. 가중치가 동일한 상태에서 최단 경로는 거쳐가는 간선의 개수가 최소한인 경로고 이는 출발지부터 목적지까지 가장 짧은 차수를 뜻하니까.

최단경로

동일하지 않은 가중치

- 가중치가 동일한 경우에는 출발지와 목적지의 차수가 짧은 것만 고려하면 되니까 BFS를 한다. 하지만 가중치가 다르면 어떻게 최단 경로를 찾을 것인가? 결국 간선의 가중치를 고려한 최단 경로 알고리즘이 필요한 것

다익스트라 알고리즘

- 다익스트라는 1959년에 나온 가중치가 동일하지 않은 경우에 최단 경로를 구하는 알고리즘이다.
- 동일하지 않은 가중치를 가진 그래프에서 최단 경로란 무엇인가? 결국, 출발지부터 목적지까지 거쳐간 간선의 가중치합이 가장 작은 경로다. 그렇기에 다익스트라 알고리즘은 아래와 같은 가정을 한다.
    - 현재 노드에서 가장 가중치 합이 적은 노드를 택한다.
    - 만약, 가중치 합이 적은 노드가 이미 더 가중치 합이 작은 경로를 가지고 있으면 다른 노드를 택한다.
    - 위와 같이 비용이 최소인 노드만 골라 가면 결국 출발지에서 목적지까지 최소 비용을 들인 경로가 나올 것이다.
- 당연히 위 글로는 머리에서 알고리즘이 그려지지 않을 것이니 아래를 보자.

```
// 아래와 같이 배열 그래프를 조성했다. 행은 현재 노드를 뜻하고 열은 해당 노드와 연결 여부를 나타내는(0이면 연결 X, 아니면 연결 O) 동시에 가중치 값이 들어간다.
  A B C D E
A 0 4 4 0 1
B 0 0 6 0 0
C 0 0 0 8 0
D 5 2 0 0 0
E 0 0 2 0 0

// A에서 D로 가는 최단 경로를 찾아보자. 다익스트라 알고리즘의 가정을 고려하여 각 노드마다 최소 비용과 해당 최소 비용일 때 직전 노드를 기록하는 배열을 구성한다.
// A의 최소 비용이 0인 이유는 출발점이기 때문이다. 각 노드마다 최소 비용을 계산할 때 다시
```

A로 돌아오면 최단 경로를 어떻게 구하겠는가?
// 현재 방문하지 않은 노드 중에 최소 비용 노드는 A니까 A부터 시작한다.
        A B C D E
최소 비용 0 X X X X
직전 노드 A X X X X
// 각 노드의 방문 여부를 점검하기 위한 방문 점검 배열도 구성하자. 현재 A를 방문했으므로 A는 방문 확인 표시
A B C D E
O X X X X

// A를 기점으로 이동할 수 있는 노드는 B, C, E이다. B, C, E 전부 거쳐간 적이 없기에 최소 비용이 A에서 거쳐간 비용이므로 A에서 각 노드로 거쳐가는 가중치 비용을 기입한다. 그리고 방문하지 않은 노드 중에서 현재 최소 비용이 가장 작은 E를 택한다.
A B C D E
0 4 4 X 1
A A A X A
// 이 때 E를 방문 했으므로 방문 표시한다
A B C D E
O X X X O

// E를 기점으로 이동할 수 있는 노드는 C 뿐이다. E에서 C로 가는 비용은 1 + 2 = 3이므로 현재 A에서 C로 거쳐가는 비용보다 작다. 수정하자.
A B C D E
0 0 3 X 1
A A E X A
// 그럼 이제 방문하지 않은 노드 중에서 최소 비용이 가장 적은 노드는 B다. B 방문 표시하자
A B C D E
O O X X O

// B를 기점으로 이동할 수 있는 노드는 C 뿐이다. 현재 C 노드의 최소 비용은 3이며 B에서 C로 이동하는 비용은 6이다. 기존 최소 비용이 더 작으므로 갱신하지 않는다.
A B C D E
0 0 3 X 1
A A E X A
// 이제 방문하지 않은 노드 중에서 최소 비용이 가장 적은 노드는 C다. C 방문 표시하자.
A B C D E
O O O X O

// C를 기점으로 방문할 수 있는 노드는 D 하나다. D는 지금까지 방문할 수 있는 경로가 없으므로 C에서 D로 가는 비용(3 + 8 = 11)을 넣어주자.
A B C D E
0 0 3 11 1
A A E C A
// 현재 방문하지 노드 중에서 가장 최소 비용이 낮은 노드는 D다. D를 방문 표시하자.
A B C D E
O O O O O

// D를 기점으로 이동할 수 있는 노드는 A와 B다. A와 B의 최소 비용은 0이기에 D에서 A나 B로 가는 비용보다 낮다. 그렇기에 갱신하지 않는다.
A B C D E

```
0 0 3 11 1
A A E C  A

// 이제 모든 노드를 점검했으므로 경로와 비용을 산출하자.
D로 가는 최단 경로 비용 : 11
최단 경로 : D - C - E - A
```

- 각 노드에서 최선의 선택(최소 비용인 경로를 고른다)을 하여 경로를 고르는 알고리즘이기에 한 가지 특징을 기억할 필요가 있다.
  - 경로 비용은 올라가면 올라갔지. 절대로 내려가면 안 된다. 다른 방문하지 않은 노드에서 갑자기 방문한 노드로 이동하는 데 비용이 감소하면 방문한 노드를 다시 방문해야 하기 때문이다. 그렇기에 음의 가중치가 있는 그래프에서는 사용할 수 없다.
- 시간복잡도 : O(V^2), 우선순위 큐를 적용하면 O(E*logV)

벨만-포드 알고리즘

- 다익스트라 알고리즘처럼 노드에서 노드까지 최소 비용을 고려하는 건 동일하다. 대신에 매 단계마다 간선의 가중치를 재확인해서 갱신하므로 음의 가중치에서도 사용할 수 있다.
- 이번에도 세부 동작 방식을 보면서 이해해 보자.

```
// 아래와 같이 배열 그래프를 조성했다.
   A  B  C  D  E
A  0  4  3  0 -6
B  0  0  0  5  0
C  0  2  0  0  0
D  7  0  4  0  0
E  0  0  2  0  0

// A에서 D로 가는 최단 경로를 찾아 보자. 이번에는 A부터 시작해서 E까지 순차적으로 방문한다.
         A  B  C  D  E
최소 비용  0  X  X  X  X
직전 노드  A  X  X  X  X

// A에서 갈 수 있는 노드는 B, C, E이고 전부 비용이 max 값보다 작으니까 B, C, E의 최소 비용 갱신
A  B  C  D  E
0  4  3  X  -6
A  A  A  X  A

// 이제 갱신한 비용을 기준으로 B에서 갈 수 있는 노드를 확인하고 비용을 비교한다.
A  B  C  D  E
0  4  3  9  -6
A  A  A  B  A

// 이제 C 노드 기준으로 탐색한다.
A  B  C  D  E
```

```
0  4  3  9  -6
A  A  A  B  A

// 이제 D 노드 기준으로 탐색한다.
A  B  C  D  E
0  4  3  9  -6
A  A  A  B  A

// 이제 E 노드 기준으로 탐색한다
A  B  C  D  E
0  4  -4 9  -6
A  A  E  B  A

// 이제 A ~ E까지 순차적으로 최소 비용을 찾는 행위를 V(노드 개수) - 1회 반복한다
// 2회
A  B  C  D  E
0  -2 -4 9  -6
A  C  E  B  A
//3회
A  B  C  D  E
0  -2 -4 3  -6
A  C  E  B  A
// 4회
A  B  C  D  E
0  -2 -4 3  -6
A  C  E  B  A
// 마지막으로 음의 순환이 벌어지는 경우를 찾기 위해 한 번 더 반복
// 4회와 동일한 값이 나오므로 음의 순환이 없음

// 최단 경로와 최소 비용
최단 경로 : D - B - C - E - A
최소 비용 : 3
```

- 시간복잡도 : O(V*E)
- 벨만 포드 알고리즘의 특징
  - 다익스트라가 노드를 중심으로 최선의 선택을 하면 벨만-포드는 간선을 중심으로 여러번 간선을 왕복하며 최단 경로를 찾는다.
  - 모든 간선을 찾아가며 최소 비용을 찾는 행위를 V - 1회를 하는 이유는 비순환 그래프에서 간선의 개수는 정점의 개수 - 1을 넘지 못하기 때문이다.
  - 마지막에 한 번 더 반복하는 이유는 비순환 그래프를 가정하고 V - 1회를 한 건데 1회를 더했을 때 경로값의 변동이 있으면 이는 반복할 때마다 음의 값 간선에 연결된 경로가 순환하고 있는 것이다. 그렇기에 음의 순환이 발생하면 경로가 계속 짧아질 것이므로 최단 경로를 구하는 의미가 없다.

풀어보자

- 미로 탈출, https://school.programmers.co.kr/learn/courses/30/lessons/159993

- 게임 맵 최단 거리, https://school.programmers.co.kr/learn/courses/30/lessons/1844
- 배달, https://school.programmers.co.kr/learn/courses/30/lessons/12978
- (난이도 상) 양과 늑대, https://school.programmers.co.kr/learn/courses/30/lessons/92343
- (난이도 상) 경주로 건설, https://school.programmers.co.kr/learn/courses/30/lessons/67259