BFS - Conecpt

BFS란?

- 시작 노드에서 가장 가까운 노드부터 먼저 방문하는 알고리즘
- DFS가 시작 노드에서 가장 깊은 곳에 있는 노드까지 들어간 다음에 하나씩 다음 노드를 봤다면 BFS는 가까운 노드를 먼저 훑어보면서 아래로 내려간다
- 예시

```
// 그래프의 인접 리스트 표현
vector<vector<int> > adj;
// start에서 시작해 그래프를 너비 우선 탐색(BFS)하고 각 정점의 방문 순서를 반환
vector<int> bfs(int start) {
        // 각 정점의 방문 여부
        vector<bool> discovered(adj.size(), false);
        // 방문할 정점 목록을 유지하는 큐
        queue<int> q;
        // 정점의 방문 순서
        vector<int> order;
        discoverd[start] = true;
        q.push(start);
        while (!q.empty()) {
                int here = q.front();
                q.pop();
                // here을 방문한다
                order.push_back(here);
                // 모든 인접한 정점을 검사한다
                for (int i = 0; i < adj[here].size(); ++i) {
                        int there = adj[here][i];
                        // 처음 보는 정점이면 방문 목록에 추가
                        if (!discovered[there]) {
```

```
                                q.push(there);
                                discovered[there] = true;
                        }
                }
        }
        return order;
}
```

- 너비 우선 탐색의 시간 복잡도 : 깊이 우선 탐색과 동일하다

최단 경로 탐색

- 너비 우선 탐색은 거의 최단 경로를 찾는 문제에서 사용된다
- 최단 경로는 출발점과 도착점 사이에 있는 간선의 수와 밀접한 연관이 있다. 이는 곧, 도착점까지 도착하는 가장 깊이가 얕은 경로를 찾으면 그게 최단 경로라는 의미다.
  - 너비 우선 탐색은 동일한 깊이에 있는 노드를 탐색하고 해당 노드를 전부 본 이후에 다음 깊이의 노드로 이동하므로 경로 즉, 가장 간선 수가 적은 곳을 찾기 쉽다
- 예시(스패닝 트리)

```
// start에서 시작해 그래프를 너비 우선 탐색하고 시작점부터 각 정점까지의 최단 거리 계산
// distance[i] = start부터 i까지의 최단 거리
// parent[i] = 너비 우선 탐색 스패닝 트리에서 i의 부모 번호, 루트인 경우 자식의 번호
void bfs2(int start, vector<int>& distance, vector<int>& parent) {
        distance = vector<int>(adj.size(), -1);
        parent = vector<int>(adj.size(), -1);
        // 방문할 정점 목록을 유지하는 큐
        queue<int> q;
        distance[start] = 0;
        parent[start] = start;
        q.push(start);
        while (!q.empty()) {
                int here = q.front();
                q.pop();
```

```
                // here의 모든 인접한 정점을 검사
                for (int i = 0; i < adj[here].size(); ++i) {
                        int there = adj[here][i];
                        // 처음 보는 정점이면 방문 목록에 집어넣는다
                        if (distance[there] == -1) {
                                q.push(there);
                                distance[there] = distance[here] + 1;
                                parent[there] = here;
                        }
                }
        }
}
```

양방향 탐색

- BFS를 응용한 탐색 방법
- 시작 정점에서 시작하는 정방향 탐색과 목표 방향에서 올라오는 역방향 탐색을 동시에 하면서 둘이 가운데에 만나면 종료하는 것
- 가령, 이진 트리에서 BFS를 적용하면 depth = n이라 할 때, BFS는 각 depth마다 2^n 개의 정점을 확인해야 한다. 깊이가 32이면 2^32이기에 마지막 깊이에서만 약 22억개의 정점을 방문한다. 하지만 역방향 탐색을 동시에 진행하면 2^16 + 2^16이므로 12만개다. 연산량의 차이가 느껴지는가?
- 예시 : 15- 퍼즐
  - 4 x 4의 격자에 딱 한 칸만 비어있고 나머지 15칸은 전부 숫자가 들어 있다. 각 숫자칸은 상하좌우로 한 칸 씩 움직일 수 있으나 다른 숫자칸이 존재하면 움직일 수 없다. 즉, 빈칸이 있는 곳만 숫자칸이 들어갈 수 있다. 그리고 숫자칸이 빈칸을 차지하면 원래 그 숫자칸이 존재하던 칸이 빈칸이 된다. 이 때, 가장 적은 움직임으로 숫자를 순서대로 배치하시오.

```
// 15-퍼즐 문제의 상태를 표현하는 클래스
class State {
        // 인접한 상태들의 목록을 반환
        std::vector<State> getAdjacent() const;
        // map에 State를 넣기 위한 비교 연산자
        bool operator < (const State& rhs) const;
```

```
        // 종료 상태와 비교하기 위한 연산자
        bool operator == (const State& rhs) const;
};
// x의 부호를 반환한다
int sgn(int x) { if (!x) return 0; return x > 0 ? 1 : -1; }
// x의 절대값을 1 증가
int incr(int x) { if (x < 0) return x - 1; return x + 1; }
// start에서 finish까지 가는 최단 경로의 길이를 반환
int bidirectional(State start, State finish) {
        // 각 정점까지의 최단 경로의 길이를 저장한다.
        std::map<State, int> c;
        // 앞으로 방문할 정점들을 저장
        std::queue<State> q;
        // 시작 상태와 목표 상태가 같은 경우는 예외로 처리
        if (start == finish) return 0;
        q.push(start); c[start] = 1;
        q.push(start); c[finish] = -1;
        // 너비 우선 탐색
        while (!q.empty()) {
                State here = q.fron();
                q.pop();
                // 인접한 상태들을 검사한다.
                std::vector<State> adjacent = here.getAdjacent();
                for (int i = 0; i < adjacent.size(); i++) {
                        std::map<State, int>::iterator it = c.find(adjacent[i]);
                        if (it == c.end()) {
                                c[adjacent[i]] = incr(c[here]);
                                q.push(adjacent[i]);
                        }
                        // 가운데에서 만나 경우
                        else if (sgn(it->second) != sng(c[here]))
                                return abs(it->second) + abs(c[here]) - 1;
                }
        }
        // 답을 찾지 못한 경우
```

```
        return -1;
}
```

점점 깊어지는 탐색

- 너비 우선 탐색(BFS)의 특징은 메모리를 많이 잡아먹는 것이다. 깊이 우선 탐색(DFS)는 플래그를 통해서 가는 정점만 확인하고 바로 빠져나오기에 별도로 메모리를 더 잡아먹지 않는다. 하지만 너비 우선 탐색(BFS)은 동일 깊이에 있는 노드들을 후보군에 별도로 저장하고 방문을 해야 하기에 메모리가 더 필요하다. 최단 경로 탐색에 대한 너비 우선 탐색(BFS)의 이점을 가져가면서 동시에 메모리를 어떻게 줄일 것인가?
- 위 사항에서 나온게 점점 깊어지는 탐색(IDS)이다. 점점 깊어지는 탐색은 임의의 깊이 제한(l, limit)을 두고 해당 깊이까지 깊이 우선 탐색(DFS)을 시도한다. 거기서 못 찾으면 l을 1씩 늘려가면서 찾아나간다. 이렇게 하면 너비 우선 탐색(BFS)보다 연산량이 좀 늘어나지만 메모리 효율은 늘릴 수 있다.
- 예시 : 15- 퍼즐

```
class State;
int best;

void dfs(State here, const State& finish, int steps) {
        // 지금까지 구한 최적해보다 더 좋을 가능성이 없으면 버린다
        if (steps >= best) return;
        // 목표 상태에 도달한 경우
        if (here == finish) { best = steps; return; }
        // 인접 상태들을 깊이 우선 탐색으로
        std::vector<State> adjacent = here.getAdjacent();
        for (int i = 0; i < adjacent.size(); ++i)
                dfs(adjacent[i], finish, steps + 1);
}
// 점점 깊어지는 탐색
int ids(State start, State finish, int growthStep) {
        for (int  limit = 4; ; limit += growthStep) {
                best = limit + 1;
                dfs(start, finish, 0);
                if (best <= limit) return best;
```

```
        }
        return -1;
}
```