너비우선탐색

NEXTERS

너비 우선 탐색 (1)

한 정점으로부터 연결된 정점들을 먼저 방문한 후 그 정점에서 다음 단계의 정점들을 반복해서 방문한다. 정점 목록으로 큐를 사용하면 적합하다.

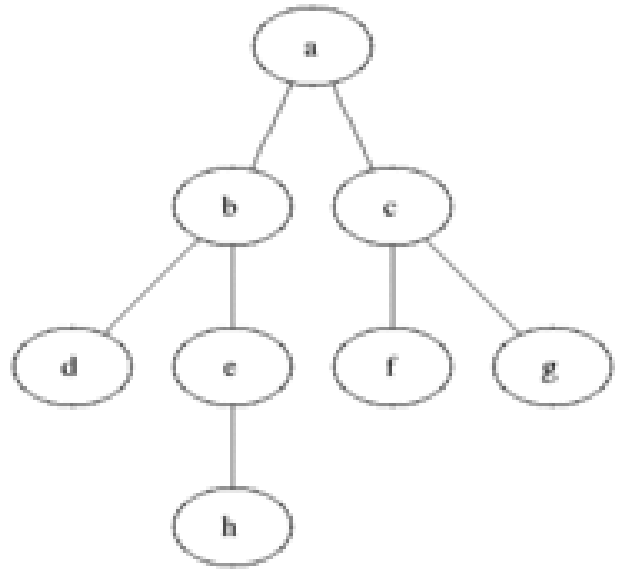
모든 정점을 한 번씩 방문하고 정점을 방문할 때마다 인접한 모든 간선을 검사하기 때문에 O(V+E)시간이 걸린다. 인접행렬은 O(V^2)

너비 우선 탐색 (2)

너비 우선 탐색 과정

a -> b -> c -> d ->e -> f -> g -> h

- 한 정점에서 연결된 정점들을 차례대로 방문 한다.



너비 우선 탐색 (3)

깊이 우선 탐색과 달리 발견과 방문이 같지 않음

아래와 같은 상태 변화를 가지게 됨

1. 아직 발견되지 않은 상태
2. 발견 되었지만 아직 방문되지 않은 상태
3. 방문된 상태

너비우선탐색 (4)

장점

- 출발노드에서 목표노드까지의 최단 길이 경로를 보장한다.

단점

- 경로가 매우 길 경우에는 탐색 가지가 급격히 증가함에 따라 보다 많은 기억 공간을 필요로 하게 된다.

너비 우선 탐색 구현 (1)

```
// 그래프의 인접 리스트 표현
vector < vector<int> ad;
// start에서 시작해 그래프를 너비 우선 탐색하고 각 정점의 방문 순서를 반환한다.
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;
    discovered[start] = true;
    q.push(start);
```

너비 우선 탐색 구현 (2)

```
    while (!q.empty()) {
    int here = q.front();
    q.pop();
    // here를 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 집어넣는다.
        if (!discovered[there]) {
        q.push(there);
        discovered[there] = true;
        }
    }

}

return order;

}
```

너비 우선 탐색과 최단 거리

그래프의 구조에 관련된 다양한 문제를 푸는 데 사용되는 깊이 우선 탐색과 달리, 너비 우선 탐색은 대개 딱 하나의 용도로 사용됩니다. 바로 그래프에서의 **최단 경로 문제** 를 푸는 것입니다.

각 정점으로부터 트리의 루트인 시작점으로 가는 경로가 실제 그래프 상에서의 최단 경로임을 알 수 있습니다. 시작점으로부터의 최단 경로는 너비 우선 탐색 스패닝 트리에서 루트로 가는 경로이므로, 각 정점이 부모의 번호를 갖고 있게 하면 쉽게 최단 경로를 계산할 수 있습니다.

최단 경로 구현 (1)

```
// start에서 시작해 그래프를 너비 우선 탐색하고 시작점부터 각 정점까지의
// 최단 거리와 너비 우선 탐색 스패닝 트리를 계산한다.
// distance[i] = start로부터 i까지의 최단 거리
// parent[i] = 너비 우선 탐색 스패닝 트리에서 i의 부모의 번호. 루트인 경우 자신의 번호
void bfs2(int start, vector<int>& distance, vector<int>& parent) {
distance = vector<int>(adj.size(), -1);
parent = vector<int>(adj.size(), -1);
queue<int> q; // 방문할 정점 목록을 유지하는 큐
distance[start] = 0;
parent[start] = start;
q.push(start);
```

최단 경로 구현 (2)

```
while (!q.empty()) {
int here = q.front();
q.pop();
// here의 모든 인접한 정점을 검사한다.
    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;
            }
        }
    }
```

최단 경로 구현 (3)

```
// v로부터 시작점까지의 최단 경로를 계산한다.
vector<int> shortestPath(int v, const vector<int>& parent) {
vector<int> path(1, v);
    while (parent[v] != v) {
        v = parent[v];
        path.push_back(v);
    }
reverse(path.begin(), path.end());
return path;
}
```

문제 – Sorting Game

배열의 연속된 부분 구간 뒤집기 연산을 이용해 정렬하기 위한 최소의 뒤집기 연산을 구한다.

가능한 모든 구간을 뒤집으면서 그래프를 만들어 나간다.

✓문제

https://algospot.com/judge/problem/read/SORTGAME

✓풀이코드

https://github.com/Nexters/algorithmStudy/blob/master/seokjoong/Chapter29/SortingGame.cpp

문제 – 하노이 탑

4개의 기둥이 있는 하노이 탑 문제를 푸는 문제

변할 수 있는 모든 경우를 계산 하면서 끝 상태를 찾아 낸다.

✓풀이코드

https://github.com/Nexters/algorithmStudy/blob/master/seokjoong/Chapter29/Hanoi4.cpp