최단 경로 - Concept

최단 경로 알고리즘이란?

- 그래프에서 주어진 두 정점을 연결하는 가장 짧은 경로를 찾는 것
- 무가중치 그래프는 BFS로 찾기에 해당 주제에서는 가중치 그래프에서 최단 경로를 찾는 방법을 논함

다익스트라

- 단일 시작점 최단 경로 알고리즘. 즉, 하나의 시작점에서 시작하여 다른 정점들까지의 최단 경로를 구하는 것.
- 가중치 그래프인 걸 고려하면 단순히 간선의 깊이만으로 최단 경로를 구할 수 없다. 그렇기에 간선에 대한 정보를 큐가 아닌 우선순위 큐를 이용하여 적절히 찾아나간다.
- 예시

```
// 다익스트라 최단 거리 알고리즘의 구현
// 정점의 개수
int V;
// 그래프의 인접 리스트. (연결된 정점 정보, 간선 가중치) 쌍을 담는다
vector<pair<int, int> > adj[MAX_V];
vector<int> dijkstra(int src) {
        vector<int> dist(V, INF);
        dist[src] = 0;
        priority_queue<pair<int, int> > pq;
        pq.push(make_pair(0, src));
        while (!pq.empty()) {
                int cost = -pq.top().first;
                int here = pq.top().second;
                pq.pop();
                // 만약 지금 꺼낸 것보다 더 짧은 경로를 알고 있으면 지금 꺼낸 것을 무시한다.
                if (dist[here] < cost) continue;
```

```
                // 인접한 정점들을 모두 검사한다
                for (int i = 0; i < adj[here].size(); ++i) {
                        int there = adj[here][i].first;
                        int nextDist = cost + adj[here][i].second;
                        // 더 짧은 경로를 발견하면, dist[]를 갱신하고 우선순위 큐에 넣는다
                        if (dist[there] > nextDist) {
                                dist[there] = nextDist;
                                pq.push(make_pair(-nextDist, there));
                        }
                }
        }
        return dist;
}
```

- 정당성 증명
  - 다익스트라 알고리즘이 최단 거리를 제대로 계산하지 못하는 정점 u가 존재한다고 가정
  - 시작점은 이미 최단 거리를 계산할 수 있으므로 u는 시작점이 아님.
  - 다익스트라 알고리즘을 이용하여 u를 방문할 때, u이전에 방문한 정점과 방문하지 않은 정점으로 그래프 양분 가능
  - u에서부터 방문하지 않은 정점을 따라가면 결국 시작점이 있는 특성상, 방문한 정점을 만나게 됨
  - 다익스트라 알고리즘은 방문한 정점에서 다음 정점으로 이동 시, 가장 경로가 짧은 곳을 계산한다. 그렇기에 거리가 더 긴 정점으로 이동하지 않음
  - 즉, 최단 거리로 이동하므로 결국 다익스트라 알고리즘이 찾아내는 경로가 최단 경로임
- 다익스트라 알고리즘의 단점
  - 음수 가중치 간선에 대해서는 사용할 수 없음. 시작점에서 거리가 더 먼 정점을 배제하더라도 그 정점과 연결된 다음 정점의 가중치가 음수면 선택하지 않은 경로가 더 짧을 수 있기 때문이다.
- 시간복잡도
  - O(|E|lg|V|) : 최대 간선의 개수(E) 만큼 우선순위 큐에 추가될 수 있고 우선순위 추가/제거 과정에서 lg|V| 만큼의 시간이 소요됨

벨만-포드

- 다익스트라와 동일하게 단일 시작점 최단 경로 알고리즘. 단, 다익스트라와 달리 가중치가 음수인 간선을 판별할 수 있다

- 시작점에서 각 정점까지 가는 최단 거리의 상한을 적당히 예측하고 예측 값과 실제 거리 사이의 오차를 반복적으로 줄여나가는 방식.
- 동작 과정
  - dist[v] <= dist[u] + w(u, v)
  - 시작점에서 u까지 가는 최단 거리(dist[u])에 u에서 v로 가는 경로의 가중치를 더한값은 시작점에서 v까지 가는 최단 거리(dist[v])보다 크거나 같다는 특성을 이용하여 찾는다.
- 예시

```
// 정점의 개수
int V;
// 그래프의 인접 리스트
vector<pair<int, int> > adj[MAX_V];
// 음수 사이클이 있을 경우 빈 배열 반환
vector<int> bellmanFord(int src) {
        // 시작점을 제외한 모든 정점까지의 최단 거리 상한을 INF로 둔다.
        vector<int> upper(V, INF);
        upper[src] = 0;
        bool updated;
        // V번 순회한다
        for (int iter = 0; iter < V; iter++) {
                updated = false;
                for (int here = 0; here < V; here++) {
                        int there = adj[here][i].first;
                        int cost = adj[here][i].second;
                        // (here, there) 간선을 따라 완화를 시도한다
                        if (upper[there] > upper[here] + cost) {
                                // 성공
                                upper[there] = upper[here] + cost;
                                updated = true;
                        }
                }
                // 모든 간선에 대해 완화가 실패했을 경우 V-1 번도 돌 필요 없이 곧장 종료한다
                if (!updated) break;
        }
```

```
        // V번째 순회에서도 완화가 성공했다면 음수 사이클이 있다.
        if (updated) upper.clear();
        return upper;
}
```

플로이드

- 모든 쌍 간의 최단 거리를 구하는 알고리즘
- 어느 점에서 어느 점까지의 최단 거리를 경유점을 통해서 구하는 방식
  -
- 예시

```
// 정점의 개수
int V;
// 그래프의 인접 행렬 표현
// adj[u][v] = u에서 v로 가는 간선의 가중치. 간선이 없으면 아주 큰 값을 넣음.
int adj[MAX_V][MAX_V];
int C[MAX_V][MAX_V][MAX_V];
void allPairShortPath1() {
        // C[0]를 초기화
        for (int i = 0; i < V; i++)
                for (int j = 0; j < V; j++)
                        if (i != j)
                                C[0][i][j] = min(adj[i][j], adj[i][0] + adj[0][j]);
                        else
                        C[0][i][j] = 0;
        // C[k - 1]이 있으면 C[k]를 계산할 수 있다
        for (int k = 1; k < V; ++k)
                for (int i = 0; i < V; ++i)
                        for (int j = 0; j < V; ++j)
```

```
                                C[k][i][j] = min(C[k - 1][i][j], C[k - 1][i][k] + C[k - 1][k][j]);
}
```

- 시간 복잡도 : O(|V|^3)
- 슬라이딩 윈도우 기법을 이용한 메모리 절약

```
int V;
int adj[MAX_V][MAX_V];
void floyd() {
        for (int i = 0; i < V; i++) adj[i][j] = 0;
        for (int k = 0; k < V; k++)
                for (int i = 0; i < V; i++)
                        for (int j = 0; j < V; j++)
                                adj[i][j] = min(adj[i][j], adj[i][k] + adj[k][j]);
}
```

- 실제 경로 계산

```
int V;
int adj[MAX_V][MAX_V]
// via[u][v] = u에서 v까지 가는 최단 경로가 경유하는 점 중 가장 번호가 큰 정점
// -1로 초기화
int via[MAX_V][MAX_V]
// 플로이드의 모든 쌍 최단 거리 알고리즘
void floyd() {
        for (int i = 0; i < V; i++) adj[i][v] = 0;
        memset(via, -1, sizeof(via));
        for (int k = 0; k < V; k++)
                for (int i = 0; i < V; i++)
                        for (int j = 0; j < V; j++)
                                if (adj[i][j] > adj[i][k] + adj[k][j]) {
                                        via[i][j] = k;
                                        adj[i][j] = adj[i][k] + adj[k][j];
```

```
                                }
        }
}
// u에서 v로 가는 최단 경로를 계산해 path에 저장한다
void reconstruct(int u, int v, vector<int>& path) {
        // 기저사례
        if (via[u][v] == -1) {
                path.push_back(u);
                if (u != v) path.push_back(v);
        }
        else {
                int w = via[u][v];
                reconstruct(u, w, path);
                path.pop_back();
                reconstruct(w, v, path);
        }
}
```