최단 경로 알고리즘 - 가중치 그래프에서 간선의 가중치 합이 최소가 되는 경로는 찾는 것

- 가중치 그래프 G=(V,E)가 있다고 할 때. 모든 간선(edge)에 가중치가 있음
- 경로의 길이는 모든 간선의 가중치 합

최단 경로 문제 종류

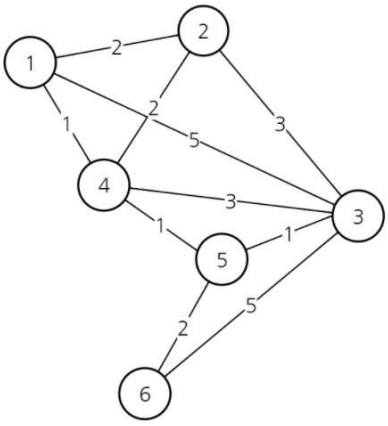
- **단일 출발 최단 경로**: 하나의 출발 노드 로부터 다른 모든 노드까지의 최단 경로를 찾는 문제 ex) 다익스트라 알고리즘 , 벨만-포드 알고리즘
- **단일 도착 최단 경로**: 다른 모든 노드로부터 하나의 목적지 노드까지 최단 경로를 찾는 문제
- **단일 쌍 최단 경로**: 단일 출발 최단 경로 문제 알고리즘을 사용
- **모든 쌍 최단 경로**: 모든 노드 쌍에 대해서 최단 경로를 찾는 문제 ex) 플로이드 알고리즘

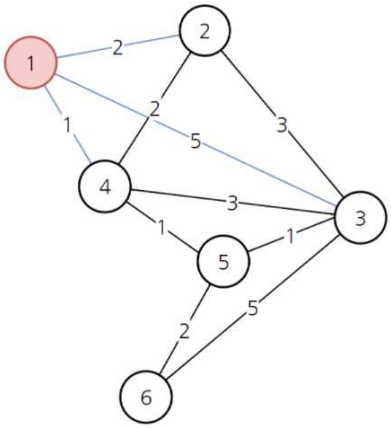
주요 알고리즘

- **BFS** : 가중치가 없거나 모든 가중치가 동일한 그래프에서 최단 경로 구하는 경우 가장 빠름
- **다익스트라 알고리즘**: 음의 가중치가 없는 경우의 최단 경로 알고리즘
- **벨만-포드 알고리즘**: 음의 가중치가 존재하는 경우의 최단 경로 알고리즘
- **플로이드-와샬 알고리즘:** 모든 정점 쌍 간의 최단 경로를 구하는 알고리즘

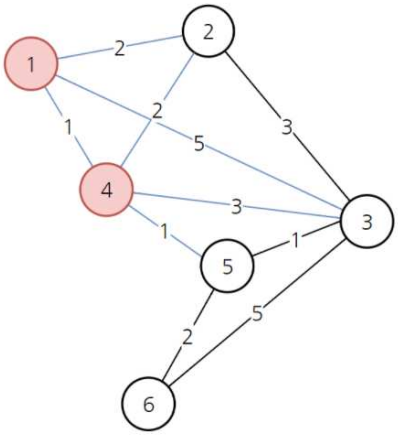
다익스트라 알고리즘

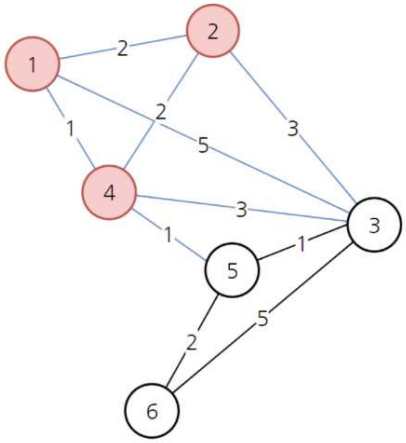
- 간선들의 가중치가 모두 0이상인 경우의 최단 경로 알고리즘
- 하나의 정점에서 다른 모든 정점으로 가는 최단 경로를 구함
- 매번 방문하지 않은 노드 중 최단 거리가 가장 짧은 노드를 선택하여 최단 거리를 구해나감 (벨만-포드 알고리즘과 다른점)





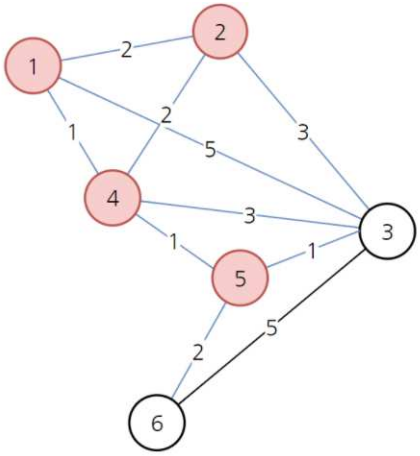
0	2	5	1	무한	무한

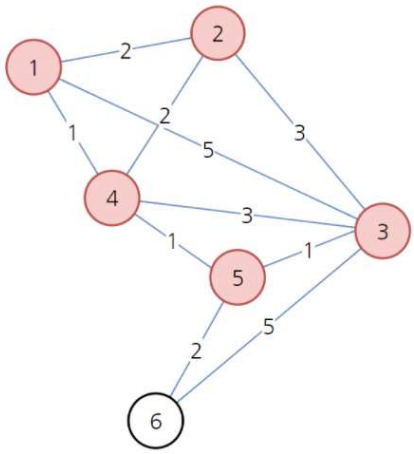




0	2	4	1	2	무한

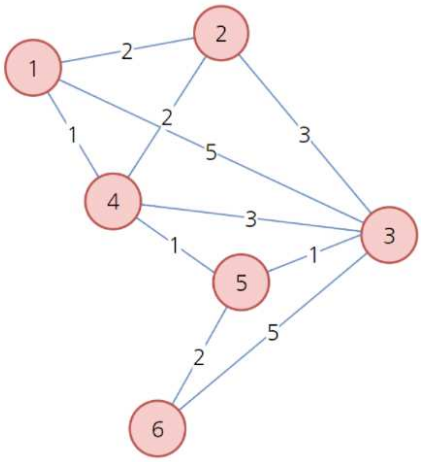
0	2	4	1	2	무한





0	2	3	1	2	4

0	2	3	1	2	4



0	2	3	1	2	4

다익스트라 코드

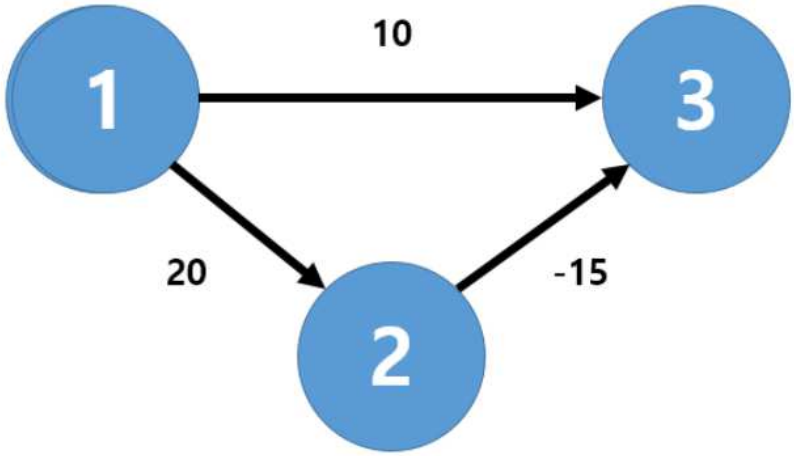
1. 출발 노드를 설정
2. 출발 노드를 기준으로 각 노드의 최소 비용 저장
3. 방문하지 않은 노드 중에서 가장 비용이 적은 노드 선택
4. 해당 노드를 거쳐 다른 노드 가는 경우를 비교하여 최소 비용 갱신
5. 계속 반복

```
int INF = 10000000;
int map[6][6] = {
    {0,2,5,1,INF,INF},
    {2,0,3,2,INF,INF},
     {5,3,0,3,1,5},
    {1,2,3,0,1,INF},
    {INF,INF,1,1,0,2},
    {INF,INF,5,INF,2,0}

};
bool check[6]; //방문한 노드 체크
int cnt = 0;

int main(void) {
    
    while (cnt != 6) {
        int idx = 0;
        int minum = INF;
        for (int i = 0; i < 6; i++) {
            if (!check[i]) {
                if (minum > map[0][i]) { //방문하지 않은 노드 중 제일 비용이 적은 노드를 선택
                    minum = map[0][i];
                    idx = i;
                }
            }
        }
        check[idx] = true;
        cnt++;
        for (int i = 0; i < 6; i++) {
            if ((map[idx][i] != 0 )&& (map[idx][i] != INF) && (!check[i])) {
                if (map[0][i] > (map[0][idx] + map[idx][i])) {
                    map[0][i] = map[0][idx] + map[idx][i];
                }
            }
        }
    }
    for (int i = 0; i < 6; i++) {
        cout << map[0][i] << " ";
    }
}
```

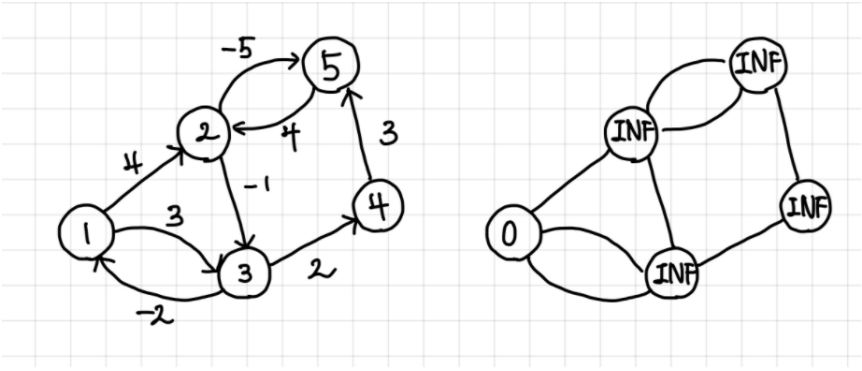
벨만-포드 vs 다익스트라



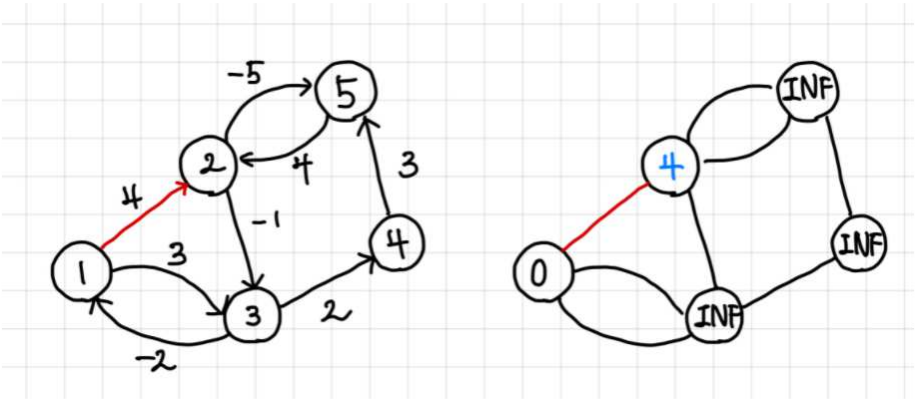
- 1번에서 3번 노드로 가는 최단 거리를 구하려고 함
- 다익스트라 알고리즘으로 풀게 되면 방문하지 않은 노드 중 최단 거리가 짧은 노드를 선택하므로 1->3 번의 최단경로 10 이 됨
- 가중치가 음수가 있는 경우 다익스트라론 최단 거리 찾을 수 없음 ->벨만-포드 알고리즘
- 벨만-포드는 매번마다 모든 간선 전부를 확인

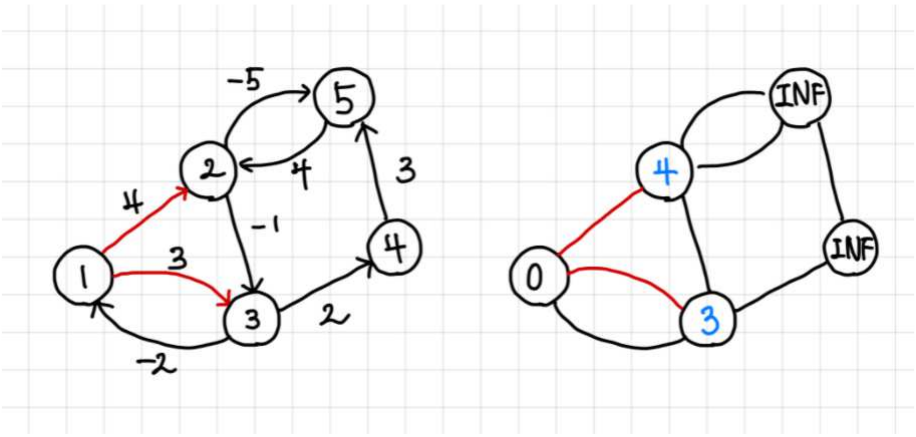
벨만-포드 알고리즘

- 간선들의 가중치가 음수가 포함되어 있는 경우의 최단 경로 알고리즘
- 매번마다 모든 간선을 확인하면서 모든 노드간의 최단 거리를 구해나감
- 음의 사이클이 나오면 최단 거리 값이 계속 작아져 무한히 갱신되므로 음의 사이클이 존재하는지 확인 해야함
- 음의 사이클 판단 방법: n-1 번 순회 한 후 , 한 번 더 순회를 했을 때, 한 노드라도 최단 거리가 변한 다면 음의 사이클이 존재함

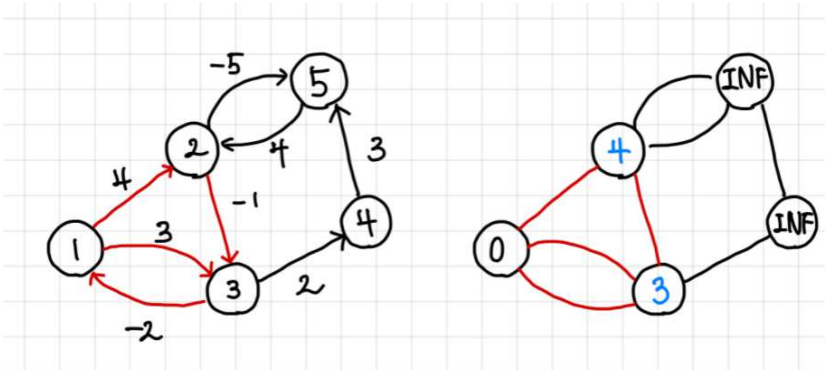


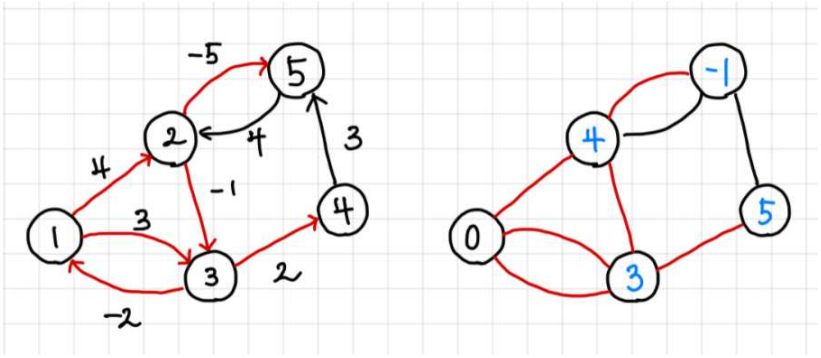
- 1번부터 시작



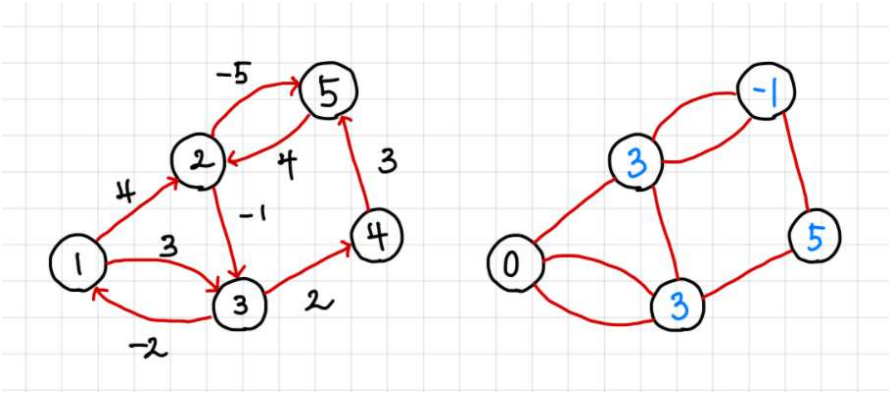


- 1번 노드 관련 갱신: 1->2 , 1->3



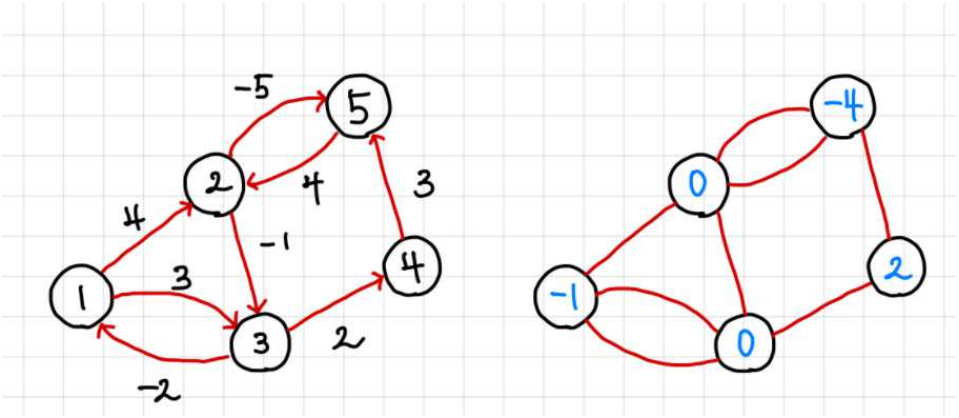


- 2번, 3번 노드 관련 갱신: 2->5 , 2->3, 3->1 , 3->4



- 5번, 4번 노드 관련 갱신 : 5->2 , 4->5

- 1회 순회 끝
- 4(v-1)회 순회 후 결과



벨만-포드 알고리즘 코드

1. d[v]=INF 가 아닌 노드(한 번이라도 계산된 정점) 는 모두 탐색하여 갱신
2. N-1번 반복 순회
3. N번째 순회 결과와 N-1번째 순회 결과가 달라졌는지 확인

```
int INF = 1000000;
vector <pair<pair<int, int>, int>> v; // <시작점, 도착점, 비용>
int d[6];
void Bellman_Ford() {
    d[1] = 0;
    for (int i = 1; i <= 4; i++) { //N-1번 반복
        for (int j = 0; j < v.size(); j++) {
            int start = v[j].first.first;
            int end = v[j].first.second;
            int cost = v[j].second;
            if (d[start] ==  INF) {
                continue;
            }
            if (d[end] > d[start] + cost) {
                d[end] = d[start] + cost;
            }
        }

    }
    for (int j = 0; j < v.size(); j++) {
        int start = v[j].first.first;
        int end = v[j].first.second;
        int cost = v[j].second;
        if (d[start] == INF) {
            continue;
        }
        if (d[end] > d[start] + cost) {
            cout << "음의 사이클이 존재합니다";
            return;
        }
    }
}
int main(void) {
    v.push_back(make_pair(make_pair(1, 2), 4));
    v.push_back(make_pair(make_pair(1, 3), 3));
    v.push_back(make_pair(make_pair(2, 5), -5));
    v.push_back(make_pair(make_pair(2, 3), -1));
    v.push_back(make_pair(make_pair(3, 4), 2));
    v.push_back(make_pair(make_pair(3, 1), -2));
    v.push_back(make_pair(make_pair(4, 5), 3));
    v.push_back(make_pair(make_pair(5, 2), 4));
}
```