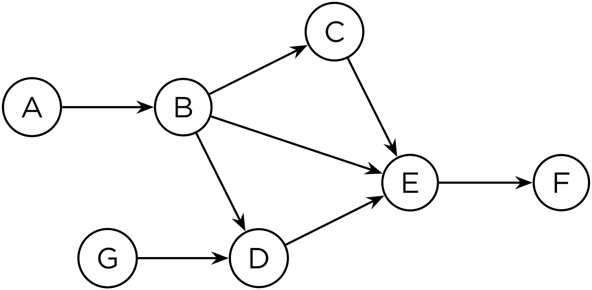
#4 최단 경로 알고리즘

2019 SCSC Summer Coding Workshop

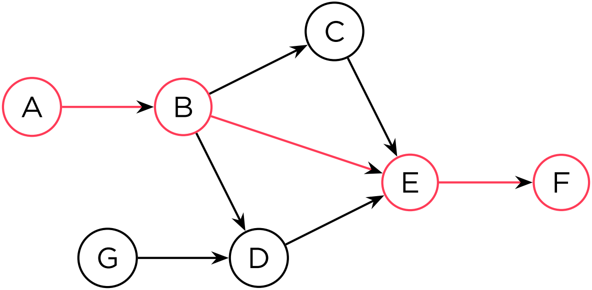
서강대학교 컴퓨터공학과 박수현

me@shiftpsh.com

그래프

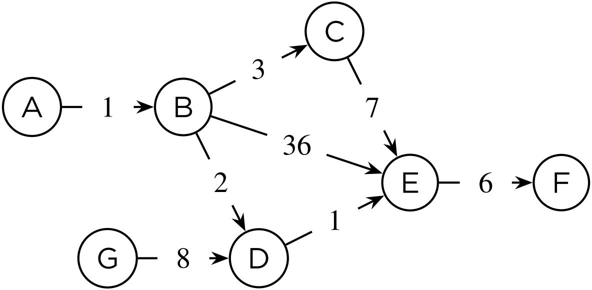


그래프



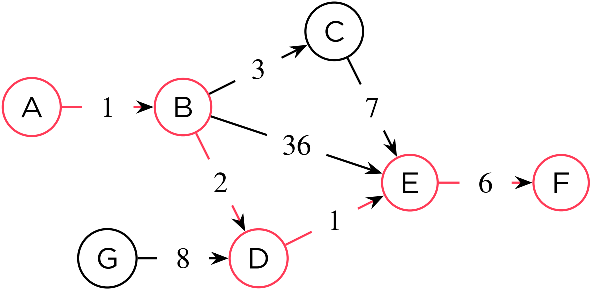
최단 경로: 모든 경로를 확인하지 않고도 BFS로 빠르게 찾을 수 있다

그래프



가중치 그래프weighted graph

그래프



이 때는 최단 경로를 BFS/DFS만으로 찾으려면 모든 경로를 다 확인해야···

최단 경로 알고리즘

한 개의 정점에서 시작해 모든 정점으로 가는 최단 경로 찾기

- 데이크스트라 알고리즘 Dijkstra's algorithm – $\mathcal{O}(\|E\| \log \|E\|)$
- 벨만–포드 알고리즘 Bellman-Ford algorithm – $\mathcal{O}(\|V\| \|E\|)$

모든 정점에서 모든 정점으로 가는 최단 경로 찾기

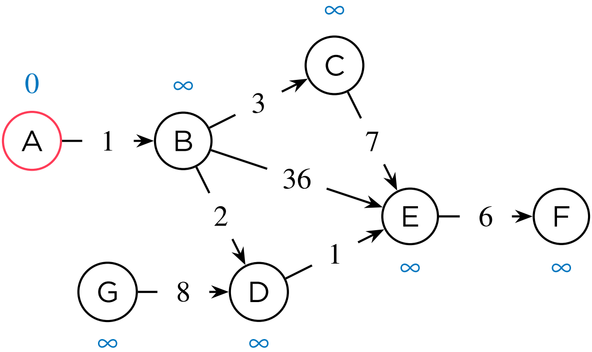
- 플로이드–와샬 알고리즘 Floyd-Warshall algorithm – $\mathcal{O}\left(\|V\|^3\right)$

데이크스트라 알고리즘

한 개의 정점에서 시작해 모든 정점으로 가는 최단 경로를 찾는 알고리즘, 단 가중치가 음수인 간선이 있으면 안 됨

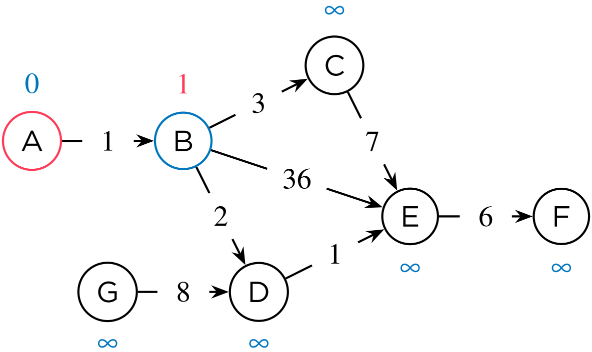
- 아직 확인하지 않은 정점들 중 시작점으로부터의 최단 거리가 가장 짧은 정점 u에 대해
- u에 인접한 정점 v들의 최단 거리를 갱신해 준다
- 그러면 u는 확인이 끝난다
- 이를 반복

데이크스트라 알고리즘



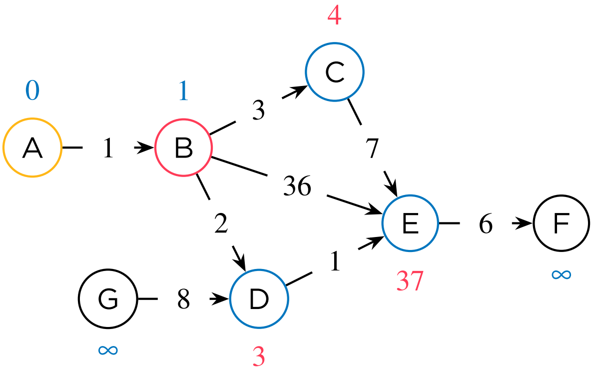
A에서 시작: A→A의 최단거리는 0

데이크스트라 알고리즘



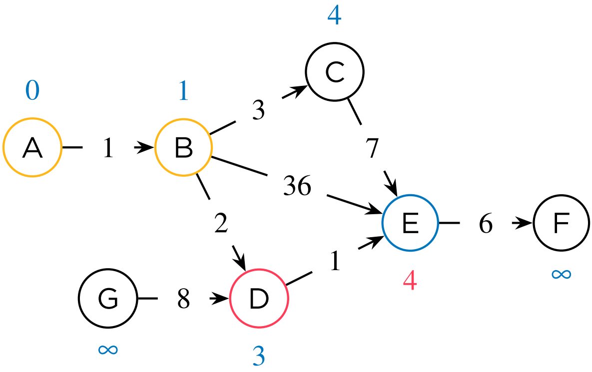
A에 인접한 B의 최단 거리 갱신 후 종료

데이크스트라 알고리즘



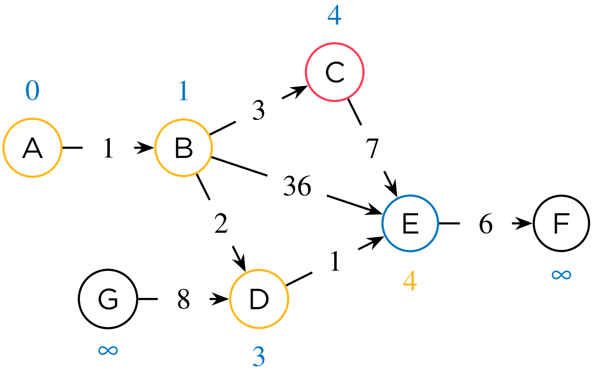
남은 노드 중 A에서의 거리가 가장 짧은 노드는 B: 인접한 C, D, E의 최단 거리 갱신 후 종료

데이크스트라 알고리즘



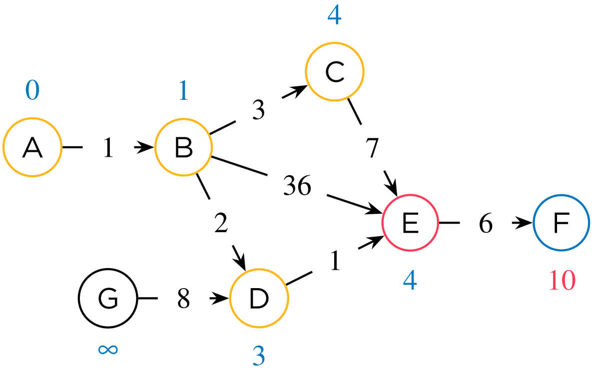
남은 노드 중 A에서의 거리가 가장 짧은 노드는 D: 인접한 E의 최단 거리 갱신 후 종료

데이크스트라 알고리즘



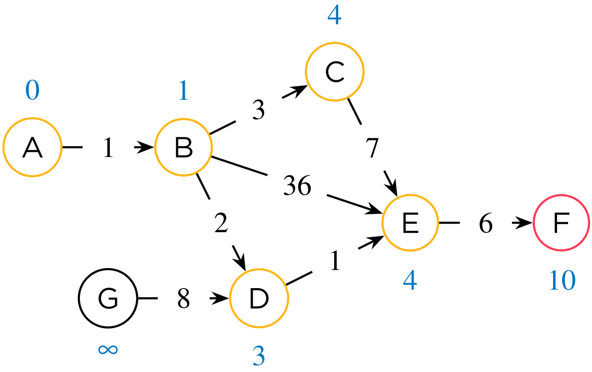
남은 노드 중 A에서의 거리가 가장 짧은 노드는 C와 E가 있는데, 아무거나 골라도 상관없다. 먼저 C를 고르고 인접한 E의 최단 거리 갱신 후 종료

데이크스트라 알고리즘



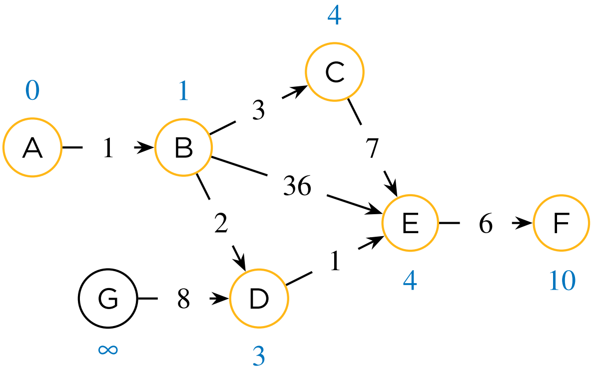
E를 고르고 인접한 F의 최단 거리 갱신 후 종료

데이크스트라 알고리즘



F는 고르긴 했으나 인접한 정점이 없으므로 바로 종료

데이크스트라 알고리즘



A에서 A–F로 가는 최단 거리가 모두 계산되었다. A→G의 경로는 없으므로 ∞

데이크스트라 알고리즘

매번 최소 거리 정점을 효율적으로 찾으려면

- 최소 힙 (우선순위 큐) 사용!
- $\big($시작점 $\to u$의 최단 거리$\big) + \big(u \to v$의 거리$\big)$ 들을 전부 최소 힙에 집어넣고, 다음에 확인할 정점을 판단하기 위해 매번 최소 힙에서 꺼낸다

데이크스트라 알고리즘

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>
#include <queue>
using namespace std;
using pii = pair<int, int>;

int inf = 987654;
int dist[20001];
vector<pii> graph[20001]; // destination, cost
```

최단 거리들을 저장하는 배열을 만들어 둔다

데이크스트라 알고리즘

```cpp
int main() {
    fill(dist, dist + 20001, inf);

    int n, m, k;
    cin >> n >> m >> k; // vertices, edges, start node index

    dist[k] = 0;

    while (m--) {
        int u, v, w; // (u -> v), cost = w
        cin >> u >> v >> w;
        graph[u].emplace_back(pii(v, w));
    }
```

최단 거리는 전부 ∞(충분히 큰 수)로 초기화해 둔다

데이크스트라 알고리즘

```cpp
    priority_queue<pii, vector<pii>, greater<>> pq; // minimum heap; cost, destination
    pq.emplace(pii(0, k));

    while (pq.size()) {
        int d = pq.top().first, u = pq.top().second;
        pq.pop();
        if (dist[u] < d) continue;
        for (pii v : graph[u]) {
            if (dist[u] + v.second >= dist[v.first]) continue;
            dist[v.first] = dist[u] + v.second;
            pq.emplace(pii(dist[v.first], v.first));
        }
    }
```

최소 힙 (어렵다면 거리를 음수로 해서 넣어도 무방)

데이크스트라 알고리즘

```cpp
    priority_queue<pii, vector<pii>, greater<>> pq; // minimum heap; cost, destination
    pq.emplace(pii(0, k));

    while (pq.size()) {
        int d = pq.top().first, u = pq.top().second;
        pq.pop();
        if (dist[u] < d) continue;
        for (pii v : graph[u]) {
            if (dist[u] + v.second >= dist[v.first]) continue;
            dist[v.first] = dist[u] + v.second;
            pq.emplace(pii(dist[v.first], v.first));
        }
    }
```

시작 지점을 넣는다

데이크스트라 알고리즘

```cpp
    priority_queue<pii, vector<pii>, greater<>> pq; // minimum heap; cost, destination
    pq.emplace(pii(0, k));

    while (pq.size()) {
        int d = pq.top().first, u = pq.top().second;
        pq.pop();
        if (dist[u] < d) continue;
        for (pii v : graph[u]) {
            if (dist[u] + v.second >= dist[v.first]) continue;
            dist[v.first] = dist[u] + v.second;
            pq.emplace(pii(dist[v.first], v.first));
        }
    }
```

현재 확인하는 노드 u에 대해 힙에 들어있는 거리가 계산한 최단 거리보다 크다면 확인하지 않고 무시해버린다

데이크스트라 알고리즘

```cpp
    priority_queue<pii, vector<pii>, greater<>> pq; // minimum heap; cost, destination
    pq.emplace(pii(0, k));

    while (pq.size()) {
        int d = pq.top().first, u = pq.top().second;
        pq.pop();
        if (dist[u] < d) continue;
        for (pii v : graph[u]) {
            if (dist[u] + v.second >= dist[v.first]) continue;
            dist[v.first] = dist[u] + v.second;
            pq.emplace(pii(dist[v.first], v.first));
        }
    }
```

인접한 노드 v에 대해 $\left(\text{시작점} \to u\text{의 최단 거리}\right) + \left(u \to v\text{의 거리}\right)$를 갱신하고, 갱신되었다면 이 거리를 힙에 넣는다

데이크스트라 알고리즘

```cpp
    for (int i = 1; i <= n; i++) {
        if (dist[i] == inf) {
            cout << "INF\n";
        } else {
            cout << dist[i] << '\n';
        }
    }

    return 0;
}
```

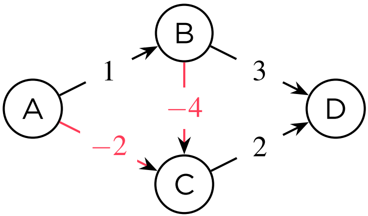
이 과정을 반복하면 dist에는 시작 노드 k로부터 각 노드에 도달하는 최단 거리들이 저장되어 있게 된다

데이크스트라 알고리즘

특징

- 최단 경로 알고리즘 중 가장 빠르다 – $\mathcal{O}(\|E\| \log \|E\|)$
- 가중치가 음수인 경로가 있으면 사용 불가능

벨만-포드 알고리즘



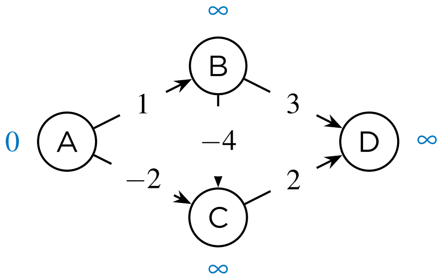
타임머신?!

벨만-포드 알고리즘

어떤 경로든 최대 $\|V\| - 1$개의 간선으로 이루어질 수 있으므로 <u>모든 간선을 $\|V\| - 1$번 확인하면서 모든 정점의 최단 거리를 갱신</u>하는 알고리즘

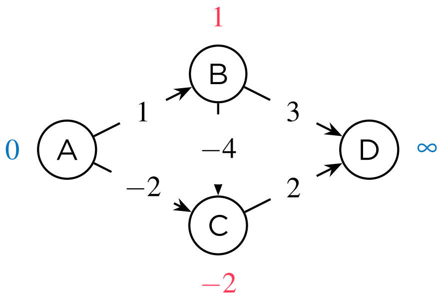
- 음수 간선에도 쓸 수 있다

벨만-포드 알고리즘



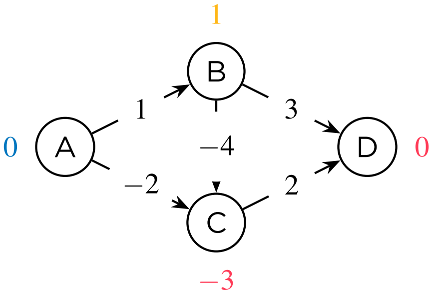
초기 상태. A에서 출발한다. 정점이 4개니까 모든 간선을 확인하는 일을 $4-1=3$번 할 예정

벨만-포드 알고리즘



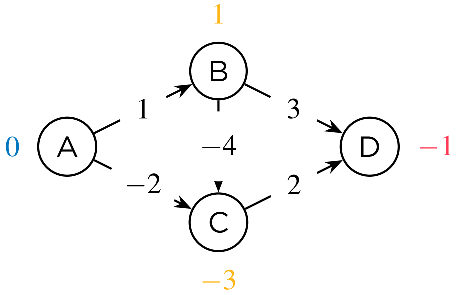
1 번째 갱신

벨만-포드 알고리즘



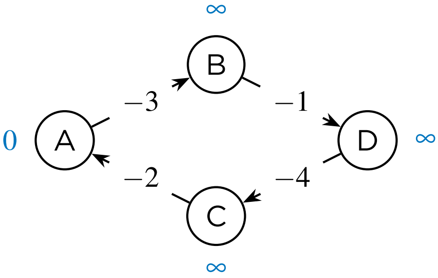
2번째 갱신

벨만-포드 알고리즘



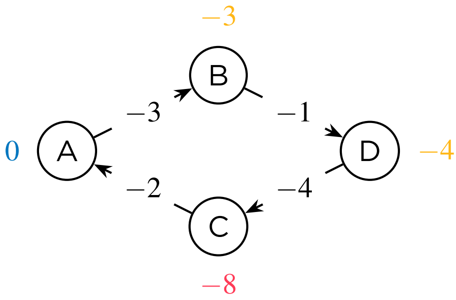
3 번째 갱신 → 끝!

벨만-포드 알고리즘



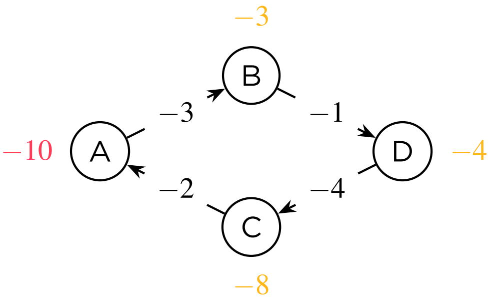
만약 이런 그래프라면? A $\to$ B $\to$ D $\to$ C를 무한 반복하면 최단 거리는 $-\infty$

벨만-포드 알고리즘



3번 업데이트

벨만-포드 알고리즘



이런 경우 최단 경로를 구성하는 노드 수가 $\infty > \|V\| - 1$ 이기 때문에,
<u>루프를 한 번 더 돌려도 최단 거리가 갱신된다</u>

이를 **음수 사이클**negative cycle 이라고 한다

벨만-포드 알고리즘

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
using pii = pair<int, int>;

int inf = 98765432;
int dist[501];
vector<pii> graph[501]; // destination, cost

int main() {
    fill(dist, dist + 501, inf);
    dist[1] = 0;
```

역시 최단 거리들을 저장하는 배열을 만들고 ∞로 초기화한다.

시작점인 1번 노드에서 1번 노드까지의 최단 거리는 0

벨만-포드 알고리즘

```cpp
    int n, m;
    cin >> n >> m;

    while (m--) {
        int u, v, c;
        cin >> u >> v >> c;
        graph[u].emplace_back(pii(v, c));
    }

    for (int _ = 1; _ < n; _++) {
        for (int u = 1; u <= n; u++) {
            if (dist[u] == inf) continue;
            for (pii v : graph[u]) {
                dist[v.first] = min(dist[v.first], dist[u] + v.second);
            }
        }
    }
```

$\|V\| - 1$ 번의 루프

벨만-포드 알고리즘

```cpp
    int n, m;
    cin >> n >> m;

    while (m--) {
        int u, v, c;
        cin >> u >> v >> c;
        graph[u].emplace_back(pii(v, c));
    }

    for (int _ = 1; _ < n; _++) {
        for (int u = 1; u <= n; u++) {
            if (dist[u] == inf) continue;
            for (pii v : graph[u]) {
                dist[v.first] = min(dist[v.first], dist[u] + v.second);
            }
        }
    }
```

(시작점을 이미 방문한) 모든 간선에 대해 최단거리 업데이트

벨만-포드 알고리즘

```cpp
    bool minus_cycle = false;
    for (int u = 1; u <= n; u++) {
        if (dist[u] == inf) continue;
        for (pii v : graph[u]) {
            if (dist[v.first] > dist[u] + v.second) {
                minus_cycle = true;
                break;
            }
        }
    }
```

루프를 한 번 더 돌렸는데 거리가 갱신된다면 음의 사이클이 있다는 뜻

벨만-포드 알고리즘

```cpp
    if (minus_cycle) {
        cout << -1;
    } else {
        for (int i = 2; i <= n; i++) {
            if (dist[i] == inf) {
                cout << -1 << '\n';
            } else {
                cout << dist[i] << '\n';
            }
        }
    }

    return 0;
}
```

벨만-포드 알고리즘

특징

- 그럭저럭 빠르다 – $\mathcal{O}(\|V\|\,\|E\|)$
- 가중치가 음수인 경로가 있어도 최단 경로를 찾을 수 있다

플로이드-와샬 알고리즘

최단 거리를 DP로 생각하면?

- $u \to v$의 최단 거리를 D_{uv}라 하자
- 그러면 $D_{uv} = \min_{k \in V} (D_{uk} + D_{kv})$가 성립

이 점에서 착안해 모든 시작점과 끝 점에 대해 최단 경로를 구해 주는 알고리즘

플로이드-와샬 알고리즘

```cpp
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

int dp[101][101];
int inf = 98765432;

int main() {
    int n, m;
    cin >> n >> m;

    for (int i = 1; i <= n; i++) {
        fill(dp[i], dp[i] + 101, inf);
        dp[i][i] = 0;
    }
```

dp[i][j]: i번 노드에서 j번 노드로 가는 최단 거리

플로이드-와샬 알고리즘

```cpp
    while (m--) {
        int u, v, c;
        cin >> u >> v >> c;

        dp[u][v] = min(dp[u][v], c);
    }

    for (int k = 1; k <= n; k++) {
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= n; j++) {
                dp[i][j] = min(dp[i][j], dp[i][k] + dp[k][j]);
            }
        }
    }
```

인접 행렬로 받는다, $i \to j$로 가는 여러 간선이 있다면 거리가 최소인 간선만 저장되도록

플로이드-와샬 알고리즘

```cpp
    while (m--) {
        int u, v, c;
        cin >> u >> v >> c;

        dp[u][v] = min(dp[u][v], c);
    }

    for (int k = 1; k <= n; k++) {
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= n; j++) {
                dp[i][j] = min(dp[i][j], dp[i][k] + dp[k][j]);
            }
        }
    }
```

$$D_{ij} = \min_{k \in V} \left(D_{ik} + D_{kj} \right)$$

플로이드-와샬 알고리즘

```cpp
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            if (dp[i][j] >= inf)
                dp[i][j] = 0;
            cout << dp[i][j] << ' ';
        }
        cout << '\n';
    }

    return 0;
}
```

문제 풀어보고, 질문하는 시간 (–17시까지)