1. **소스코드 및 주석**

**1-1. BASIC VERSION**

#include <iostream>

#include <ctime>

#include <algorithm>

#include <random>

#include <vector>

#include <queue>

#define INF 1000000000

#define NODE 10

using namespace std;

// == 플로이드 시작 ==

void floyd\_washall(int graph[NODE][NODE], int\* duration\_floyd)

{

clock\_t start, finish;

start = clock();

int dp[NODE][NODE];

for (int i = 0; i < NODE; i++)

for (int j = 0; j < NODE; j++)

dp[i][j] = graph[i][j];

// k : 거쳐가는 노드

for (int k = 0; k < NODE; k++)

// i : 출발지 노드

for (int i = 0; i < NODE; i++)

// j : 도착지 노드

for (int j = 0; j < NODE; j++)

// "i -> k , k -> j" 랑 "i -> j" 비교하여 더 작은 값을 선택

if (dp[i][k] + dp[k][j] < dp[i][j])

dp[i][j] = dp[i][k] + dp[k][j];

for (int i = 0; i < NODE; i++)

{

for (int j = 0; j < NODE; j++)

{

if (dp[i][j] == INF) cout << "-1 ";

else cout << dp[i][j] << " ";

}

cout << '\n';

}

finish = clock();

\*duration\_floyd = (double)(finish - start);

}

// == 플로이드 끝 ==

// === 다익스트라 시작===

bool visit[NODE];

int dist[NODE];

int min\_node;

int get\_small\_node() {

// 최소비용노드 탐색 함수

int min = INF;

int minpos = 0;

for (int i = 0; i < NODE; i++) {

if (dist[i] < min && !visit[i]) {

min = dist[i];

minpos = i;

}

}

return minpos;

}

void dijkstra(int start, int graph[NODE][NODE]) {

for (int i = 0; i < NODE; i++) {

dist[i] = INF;

visit[i] = false;

}

dist[start] = 0;

// pair<int, int> 큐에 넣을 자료형 → (거리, 노드번호) 형태

// vector<pair<int, int>> 내부 컨테이너(기본값도 vector임)

// greater<pair<int, int>> 정렬 기준 : 작은 값부터 꺼내기(최소 힙)

priority\_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;

pq.push({ 0, start });

while (!pq.empty()) {

int cur\_dist = pq.top().first;

int cur = pq.top().second;

pq.pop();

if (visit[cur]) continue;

visit[cur] = true;

for (int next = 0; next < NODE; next++) {

if (graph[cur][next] != INF) {

int new\_dist = dist[cur] + graph[cur][next];

if (new\_dist < dist[next]) {

dist[next] = new\_dist;

pq.push({ new\_dist, next });

}

}

}

}

}

void all\_dijkstra(int graph[NODE][NODE], int\* duration\_dijkstra) {

clock\_t start, finish;

start = clock();

for (int i = 0; i < NODE; i++)

{

dijkstra(i, graph);

for (int j = 0; j < NODE; j++)

cout << dist[j] << " ";

cout << endl;

}

cout << endl;

finish = clock();

\*duration\_dijkstra = (double)(finish - start);

}

// === 다익스트라 끝 ===

int main(void)

{

cout << "<Basic version>" << endl;

cout << "주어진 그래프에 대해서 Dijkstra(시작점 1~10변경) vs Floyd 알고리즘의 연산 시간을 측정하시오." << endl;

int graph[NODE][NODE] = {

{ 0, 15, 12, INF, INF, INF, INF, INF, INF, INF }, //서울

{ 15, 0, INF, INF, INF, 21, INF, 7, INF, INF }, // 원주

{ 12, INF, 0, 4, 10, INF, INF, INF, INF, INF }, //천안

{ INF, INF, 4, 0, 3, INF, 13, INF, INF, INF }, //논산

{ INF, INF, 10, 3, 0, INF, INF, 10, INF, INF }, //대전

{ INF, 21, INF, INF, INF, 0, INF, INF, 25, INF }, //강릉

{ INF, INF, INF, 13, INF, INF, 0, INF, INF, 15 }, //광주

{ INF, 7, INF, INF, 10, INF, INF, 0, 19, 9 }, //대구

{ INF, INF, INF, INF, INF, 25, INF, 19, 0, 5 }, //포항

{ INF, INF, INF, INF, INF, INF, 15, 9, 5, 0 } //부산

};

int duration\_floyd = 0;

cout << endl;

cout << "(floyd\_washall 결과)" << endl;

floyd\_washall(graph, &duration\_floyd);

cout << "Floyd-Warshall 알고리즘 연산시간 : " << duration\_floyd << " ms" << endl;

cout << endl;

int duration\_dijkstra = 0;

cout << "(Dijkstra 결과)" << endl;

all\_dijkstra(graph, &duration\_dijkstra);

cout << "Dijkstra 알고리즘 연산시간 : " << duration\_dijkstra << " ms" << endl;

return 0;

}

**1-2. EXTEND VERSION**

#include <iostream>

#include <ctime>

#include <algorithm>

#include <random>

#include <vector>

#include <queue>

#define INF 1000000000

#define MAX\_NODE 20

using namespace std;

random\_device rd;

mt19937\_64 rng(rd());

template<typename T>

T RandomNumber(T start, T end) {

uniform\_int\_distribution<T> dist(start, end);

return dist(rng);

}

// 그래프 생성 함수

void CreateGraph(int graph[MAX\_NODE][MAX\_NODE], int NODE) {

for (int i = 0; i < NODE; i++) {

for (int j = 0; j < NODE; j++) {

graph[i][j] = (i == j) ? 0 : RandomNumber(1, 20);

}

}

}

// == 플로이드 시작 ==

void floyd\_washall(int graph[MAX\_NODE][MAX\_NODE], int\* duration\_floyd, int NODE)

{

clock\_t start, finish;

start = clock();

int dp[MAX\_NODE][MAX\_NODE];

for (int i = 0; i < NODE; i++)

for (int j = 0; j < NODE; j++)

dp[i][j] = graph[i][j];

// k : 거쳐가는 노드

for (int k = 0; k < NODE; k++)

// i : 출발지 노드

for (int i = 0; i < NODE; i++)

// j : 도착지 노드

for (int j = 0; j < NODE; j++)

// "i -> k , k -> j" 랑 "i -> j" 비교하여 더 작은 값을 선택

if (dp[i][k] + dp[k][j] < dp[i][j])

dp[i][j] = dp[i][k] + dp[k][j];

for (int i = 0; i < NODE; i++)

{

for (int j = 0; j < NODE; j++)

{

if (dp[i][j] == INF) cout << "-1 ";

else cout << dp[i][j] << " ";

}

cout << '\n';

}

finish = clock();

\*duration\_floyd = (double)(finish - start);

}

// == 플로이드 끝 ==

// === 다익스트라 시작===

bool visit[MAX\_NODE];

int dist[MAX\_NODE];

int min\_node;

void dijkstra(int start, int graph[MAX\_NODE][MAX\_NODE], int NODE) {

for (int i = 0; i < NODE; i++) {

dist[i] = INF;

visit[i] = false;

}

dist[start] = 0;

// pair<int, int> 큐에 넣을 자료형 → (거리, 노드번호) 형태

// vector<pair<int, int>> 내부 컨테이너(기본값도 vector임)

// greater<pair<int, int>> 정렬 기준 : 작은 값부터 꺼내기(최소 힙)

priority\_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;

pq.push({ 0, start });

while (!pq.empty()) {

int cur\_dist = pq.top().first;

int cur = pq.top().second;

pq.pop();

if (visit[cur]) continue;

visit[cur] = true;

for (int next = 0; next < NODE; next++) {

if (graph[cur][next] != INF) {

int new\_dist = dist[cur] + graph[cur][next];

if (new\_dist < dist[next]) {

dist[next] = new\_dist;

pq.push({ new\_dist, next });

}

}

}

}

}

void all\_dijkstra(int graph[MAX\_NODE][MAX\_NODE], int\* duration\_dijkstra, int NODE) {

clock\_t start, finish;

start = clock();

for (int i = 0; i < NODE; i++)

{

dijkstra(i, graph, NODE);

for (int j = 0; j < NODE; j++)

cout << dist[j] << " ";

cout << endl;

}

cout << endl;

finish = clock();

\*duration\_dijkstra = (double)(finish - start);

}

// === 다익스트라 끝 ===

int main(void)

{

cout << "<Extend version>" << endl;

cout << "랜덤 그래프에 대해서 Dijkstra vs Floyd 알고리즘의 연산 시간을 측정하시오." << endl;

for (int NODE = 10; NODE <= MAX\_NODE; NODE++) {

cout << "==============================================" << endl;

cout << "[노드의 개수가 " << NODE << "인 랜덤그래프를 생성합니다...]" << endl;;

int graph[MAX\_NODE][MAX\_NODE] = { 0, };

CreateGraph(graph, NODE);

int duration\_floyd = 0;

cout << endl;

cout << "(floyd\_washall 결과)" << endl;

floyd\_washall(graph, &duration\_floyd, NODE);

cout << "Floyd-Warshall 알고리즘 연산시간 : " << duration\_floyd << " ms" << endl;

cout << endl;

int duration\_dijkstra = 0;

cout << "(Dijkstra 결과)" << endl;

all\_dijkstra(graph, &duration\_dijkstra, NODE);

cout << "Dijkstra 알고리즘 연산시간 : " << duration\_dijkstra << " ms" << endl;

}

return 0;

}

**1-3. ADVANCED VERSION**

#include <iostream>

#include <ctime>

#include <algorithm>

#include <random>

#include <vector>

#include <queue>

#define INF 1000000000

#define MAX\_NODE 20

using namespace std;

random\_device rd;

mt19937\_64 rng(rd());

template<typename T>

T RandomNumber(T start, T end) {

uniform\_int\_distribution<T> dist(start, end);

return dist(rng);

}

// 그래프 생성 함수

void CreateGraph(int graph[MAX\_NODE][MAX\_NODE], int NODE) {

for (int i = 0; i < NODE; i++) {

for (int j = 0; j < NODE; j++) {

graph[i][j] = (i == j) ? 0 : RandomNumber(1, 20);

}

}

}

// == 플로이드 시작 ==

void floyd\_washall(int graph[MAX\_NODE][MAX\_NODE], int\* duration\_floyd, int NODE, int dp[MAX\_NODE][MAX\_NODE])

{

clock\_t start, finish;

start = clock();

for (int i = 0; i < NODE; i++)

for (int j = 0; j < NODE; j++)

dp[i][j] = graph[i][j];

// k : 거쳐가는 노드

for (int k = 0; k < NODE; k++)

// i : 출발지 노드

for (int i = 0; i < NODE; i++)

// j : 도착지 노드

for (int j = 0; j < NODE; j++)

// "i -> k , k -> j" 랑 "i -> j" 비교하여 더 작은 값을 선택

if (dp[i][k] + dp[k][j] < dp[i][j])

dp[i][j] = dp[i][k] + dp[k][j];

for (int i = 0; i < NODE; i++)

{

for (int j = 0; j < NODE; j++)

{

if (dp[i][j] == INF) cout << "-1 ";

else cout << dp[i][j] << " ";

}

cout << '\n';

}

finish = clock();

\*duration\_floyd = (double)(finish - start);

}

// == 플로이드 끝 ==

// === 다익스트라 시작===

bool visit[MAX\_NODE];

int dist[MAX\_NODE];

int min\_node;

void dijkstra(int start, int graph[MAX\_NODE][MAX\_NODE], int NODE) {

for (int i = 0; i < NODE; i++) {

dist[i] = INF;

visit[i] = false;

}

dist[start] = 0;

// pair<int, int> 큐에 넣을 자료형 → (거리, 노드번호) 형태

// vector<pair<int, int>> 내부 컨테이너(기본값도 vector임)

// greater<pair<int, int>> 정렬 기준 : 작은 값부터 꺼내기(최소 힙)

priority\_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;

pq.push({ 0, start });

while (!pq.empty()) {

int cur\_dist = pq.top().first;

int cur = pq.top().second;

pq.pop();

if (visit[cur]) continue;

visit[cur] = true;

for (int next = 0; next < NODE; next++) {

if (graph[cur][next] != INF) {

int new\_dist = dist[cur] + graph[cur][next];

if (new\_dist < dist[next]) {

dist[next] = new\_dist;

pq.push({ new\_dist, next });

}

}

}

}

}

void all\_dijkstra(int graph[MAX\_NODE][MAX\_NODE], int\* duration\_dijkstra, int NODE) {

clock\_t start, finish;

start = clock();

for (int i = 0; i < NODE; i++)

{

dijkstra(i, graph, NODE);

for (int j = 0; j < NODE; j++)

cout << dist[j] << " ";

cout << endl;

}

cout << endl;

finish = clock();

\*duration\_dijkstra = (double)(finish - start);

}

// === 다익스트라 끝 ===

int main(void)

{

cout << "<Extend version>" << endl;

cout << "랜덤 그래프에 대해서 Dijkstra vs Floyd 알고리즘의 연산 시간을 측정하시오." << endl;

int NODE = 20;

cout << "==============================================" << endl;

cout << "[노드의 개수가 " << NODE << "인 랜덤그래프를 생성합니다...]" << endl;;

int graph[MAX\_NODE][MAX\_NODE] = { 0, };

CreateGraph(graph, NODE);

int duration\_floyd = 0;

cout << endl;

cout << "(floyd\_washall 결과)" << endl;

int dp[MAX\_NODE][MAX\_NODE] = { 0, };

floyd\_washall(graph, &duration\_floyd, NODE, dp);

cout << "Floyd-Warshall 알고리즘 연산시간 : " << duration\_floyd << " ms" << endl;

cout << endl;

int duration\_dijkstra = 0;

cout << "(Dijkstra 결과)" << endl;

all\_dijkstra(graph, &duration\_dijkstra, NODE);

cout << "Dijkstra 알고리즘 연산시간 : " << duration\_dijkstra << " ms" << endl;

int start = -1, end = -1, now = -1, pre=-1, next\_min = INF;

cout << "시작 노드 번호를 입력하세요(0 ~ " << NODE - 1<< ") : ";

cin >> start;

cout << "도착 노드 번호를 입력하세요(0 ~ " << NODE - 1 << ") : ";

cin >> end;

if (start < 0 || start >= NODE || end < 0 || end >= NODE) {

cout << "잘못된 입력입니다." << endl;

return 0;

}

cout << "다음은 플로이드 와샬 알고리즘을 이용한 최단 경로입니다." << endl;

now = start;

cout << "경로: " << now;

while (now != end) {

int next = -1;

for (int j = 0; j < NODE; j++) {

// 자기 자신및 역방향은 제외

if (j != now && dp[now][j] + dp[j][end] == dp[now][end]) {

next = j;

break;

}

}

if (next == -1) {

cout << " -> 경로 없음";

break;

}

cout << " -> " << next;

now = next;

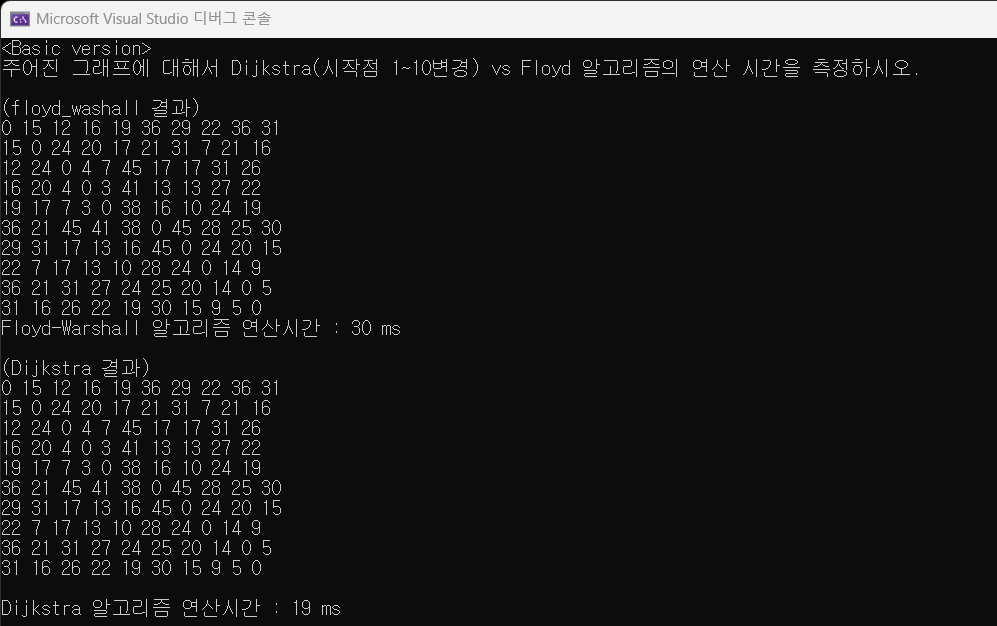
}

return 0;

}

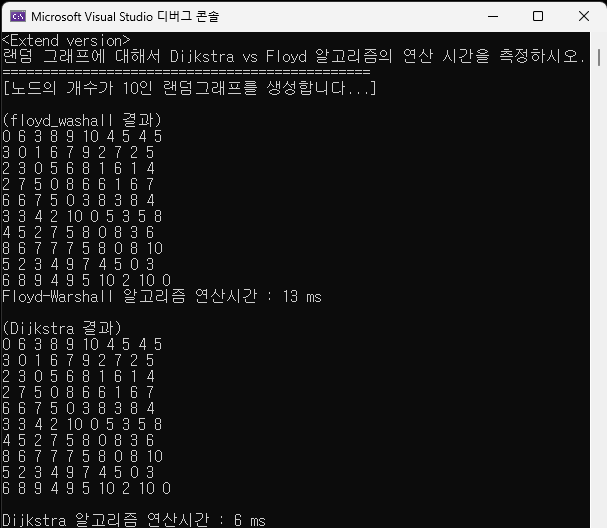
1. **출력 결과**

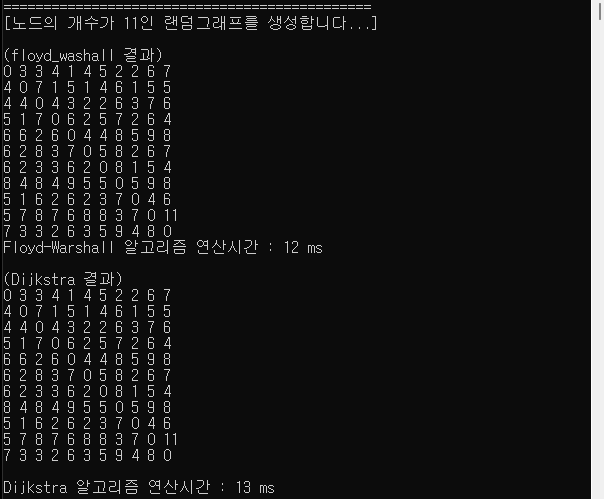
**2-1. BASIC VERSION**

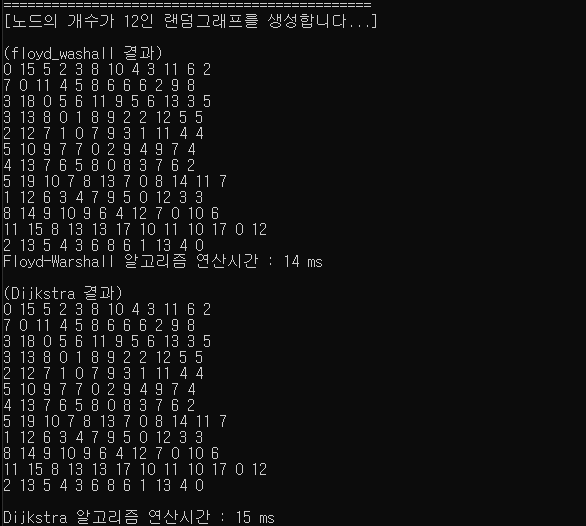
****

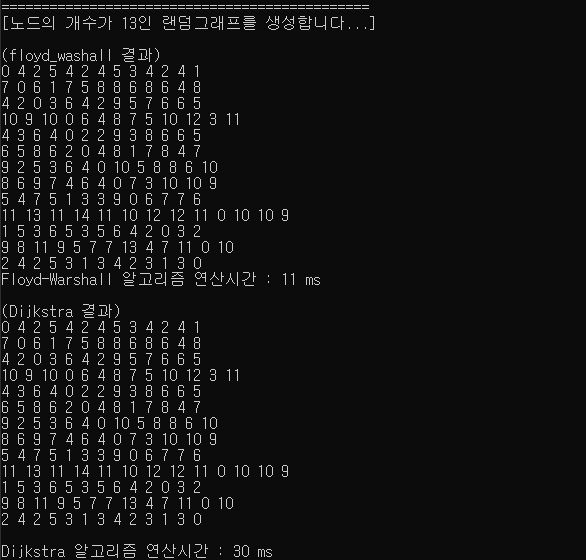
**2-2. EXTEND VERSION**

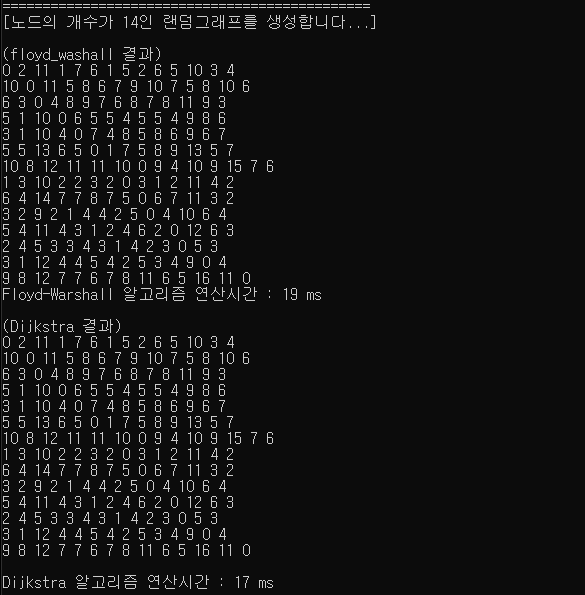
*10개의 랜덤 그래프를 만든다음 플로이드와 다익스트라를 비교하였습니다 그리고 노드를 20까지 한개씩 늘리면서 출력했습니다*

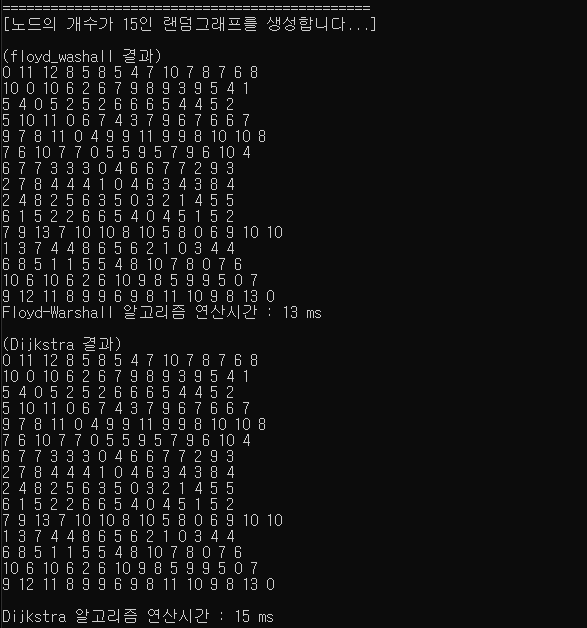
****

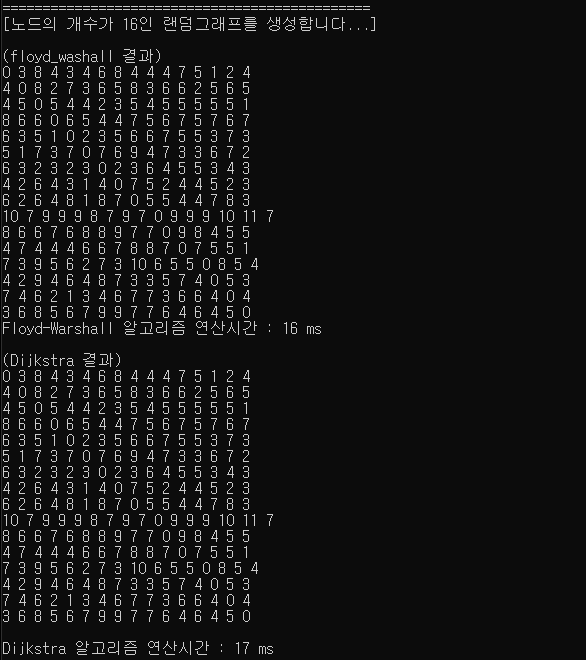
****

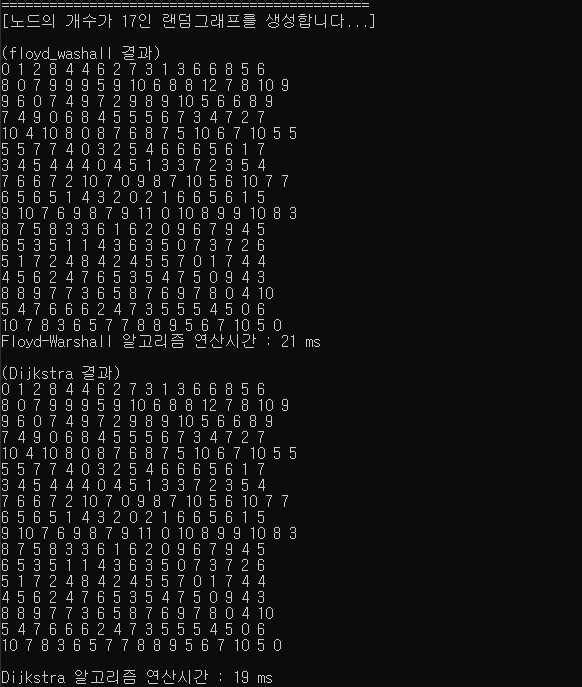
****

****

****

****

****

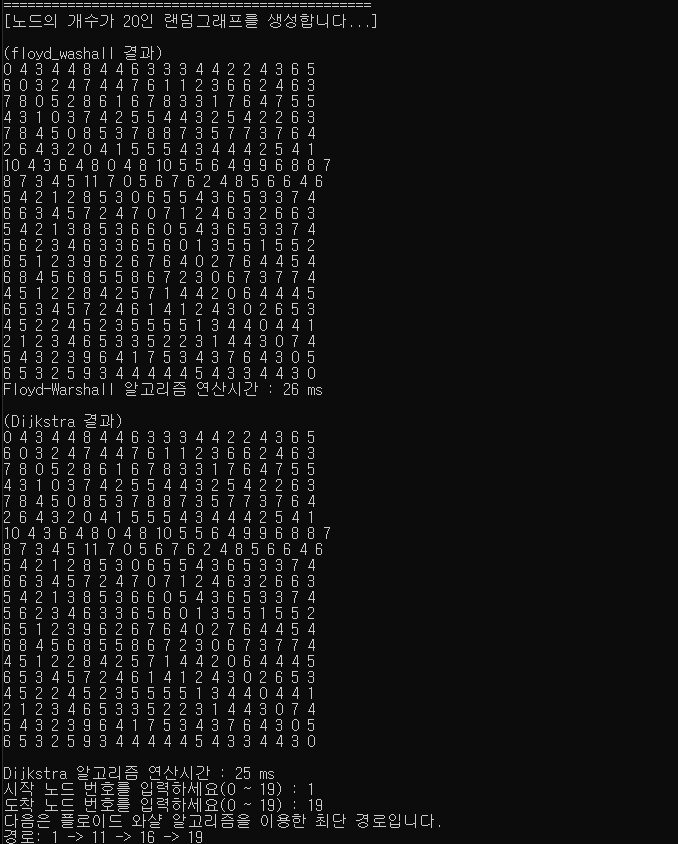
****

****

****

****

**2-3. ADVANCED VERSION**

****

****

1. **고찰**

**3-1. BASIC VERSION**

1. **우선순위큐를 사용하여 다익스트라 알고리즘의 시간복잡도를 줄였기에 다익스트라가 더 빠를 것이라고 예측하였다**
2. **여러번 ‘플로이드 vs 다익스트라’의 시간 복잡도를 비교하였다**
3. **결과**
   1. **실제로 다익스트라가 대부분 조금 빠르게 출력되었다**
   2. **대신, 플로이드의 코드가 압도적으로 짧고 간단하였다**

**3-2. EXTEND VERSION**

1. **우선순위큐를 사용하여 다익스트라 알고리즘의 시간복잡도를 줄였기에 다익스트라가 더 빠를 것이라고 예측하였다**
2. **그리고, 노드의 수가 증가할 수록 더 빠를 것이라고 예측하였다**
3. **결과 : 노드에 따른 (플로이드 - 다익스트라)의 시간복잡도**

단위 : ms

|  |  |  |  |
| --- | --- | --- | --- |
|  | **플루이드** | **다익스트라** | **(플루이드 - 다익스트라)** |
| **10** | **13** | **6** | **7** |
| **11** | **12** | **13** | **-1** |
| **12** | **14** | **15** | **-1** |
| **13** | **11** | **30** | **-19** |
| **14** | **19** | **17** | **2** |
| **15** | **13** | **15** | **-2** |
| **16** | **16** | **17** | **-1** |
| **17** | **21** | **19** | **2** |
| **18** | **22** | **21** | **1** |
| **19** | **25** | **24** | **1** |
| **20** | **26** | **29** | **-3** |

1. **결과 : 노드에 따른 (플로이드 - 다익스트라)의 시간복잡도의 피어슨 상관계수 분석**
   1. **노드 수가 증가함에 따라 플로이드가 더 빠르다는 경향성을 띌 것이라 예측하였음**
   2. **하지만 상관계수가 0.014으로 거의 상관이 없다는 결과가 나왔음**
   3. **각각 랜덤그래프로 조사하였기에 랜덤의 영향이 너무 컸던 것이라고 판단**
   4. **다만, 플로이드가 평균 약-1.27ms만큼 빠르다는 것은 확인되었음**

**3-3. ADVANCED VERSION**

* **이미 만들어져있는 경로값들을 바탕으로 현재값과 역방향을 제외하고 최단 경로를 출력하였다**
* **결과, 대부분 중간 경로없이 바로 가는 것이 빠르다는 결과**
* **모든 경로값을 0~20의 랜덤값을 하였기에 바로 가는것이 빠르다고 출력된것이 아닌지 생각해보았음**