

# 데이터 구조 설계

이기훈 교수님

Project 3

컴퓨터정보공학부

2018202076

이연걸

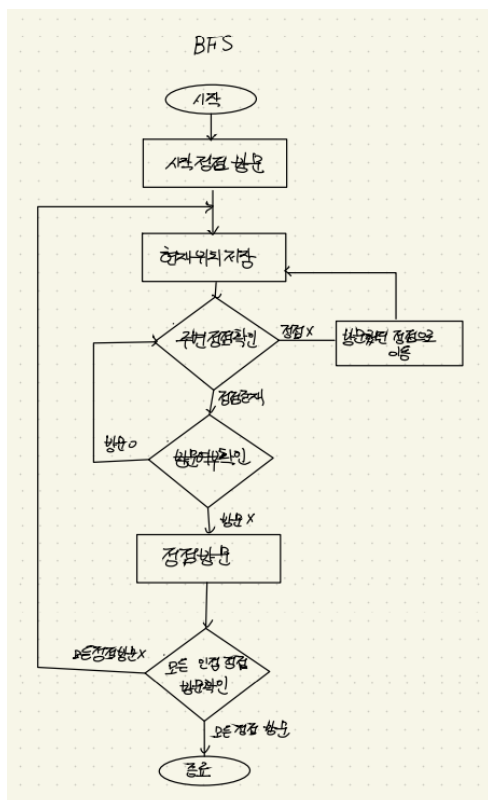
- Introduction

본 과제는 그래프를 이용한 그래프 연산 프로그램을 만드는 것이다. 그래프 정보가 저장된 텍스트 파일을 읽어서 BFS, DFS, DFS\_R, KURSKAL, DIJKSTRA, BELLMAN-FORD, FLOYD 연산을 수행한다. 본 과제를 해결하기 위해서는 각 알고리즘들의 특성을 알고 있어야 한다. BFS, DFS의 수색 방법 차이, 재귀와 반복문에 따른 구현의 차이를 알아야 하며, 최단 거리 알고리즘들의 차이를 알아야 한다. KURSKAL은 weight가 적은 edge부터 사이클 형성에 유의해 그래프에 추가해주면서 진행하고, DIJKSTRA는 시작 정점에서 종료 정점까지 이동하면서 여러 정점의 최단 경로를 파악하며 진행한다. BELLMAN-FORD는 간선의 가중치가 0일 수 있음에 유의해 가용한 edge가 늘어남에 따라 모든 정점들 간의 최단 거리를 최신화해 가며 진행한다. 마지막으로 FLOYD는 모든 경유점을 지나는 모든 시작점, 끝점을 확인해 가며 진행한다.

이런 알고리즘의 특성들을 바탕으로 전체 최단거리 정보, 혹은 지나쳤던 경로들을 기록해 최단 경로, 거리를 구한다.

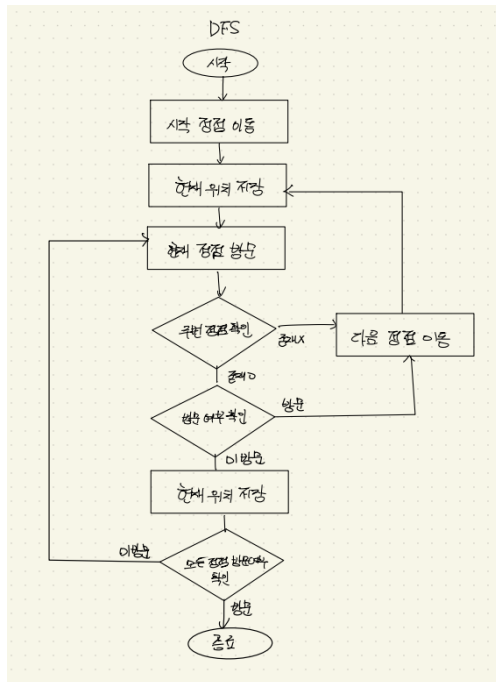
- Flowchart

1. BFS



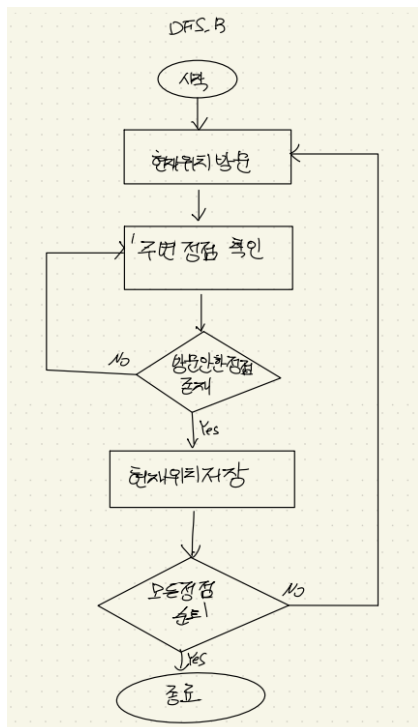
BFS는 너비 우선 탐색 알고리즘으로 주변 정점들을 하나씩 방문해 가면서 진행한다.

2. DFS



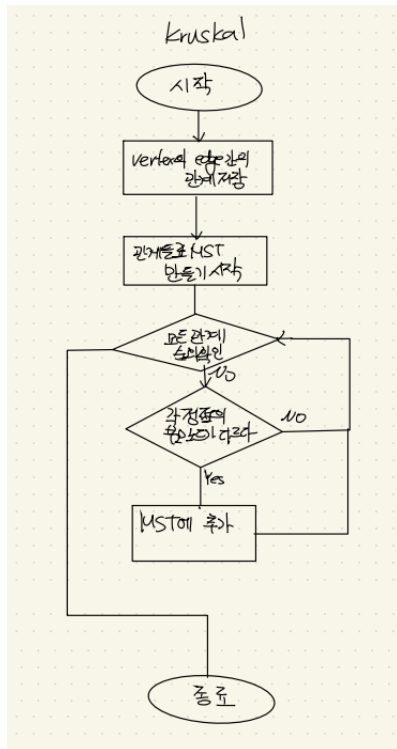
DFS는 깊이 우선 알고리즘으로 방문한 정점의 인접 정점 하나를 쫓 따라가면서 더 이상 진행할 수 있는 정점이 없다면 다시 돌아와 방문 가능한 정점을 방문하면서 진행한다.

### 3. DFS\_R



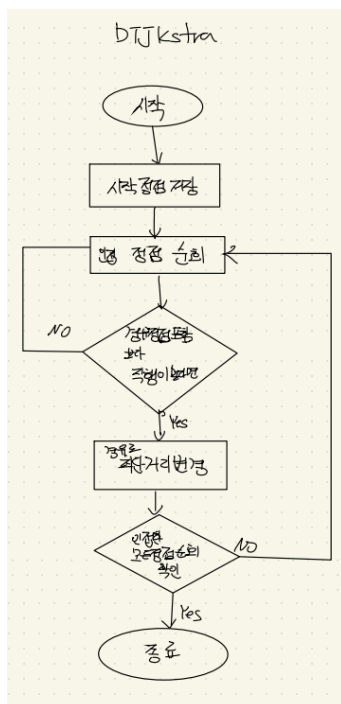
DFS\_R은 위의 DFS를 재귀함수로 구현한 것이다. 재귀 함수이므로 방문 정보를 저장할 수 있는 배열, 혹은 벡터가 필요하며 현재 위치를 방문하고 방문 가능한 주변 정점이 있다면 재귀 함수를 호출해 바뀐 현재 위치를 방문하면서 진행한다.

#### 4. KRUSKAL



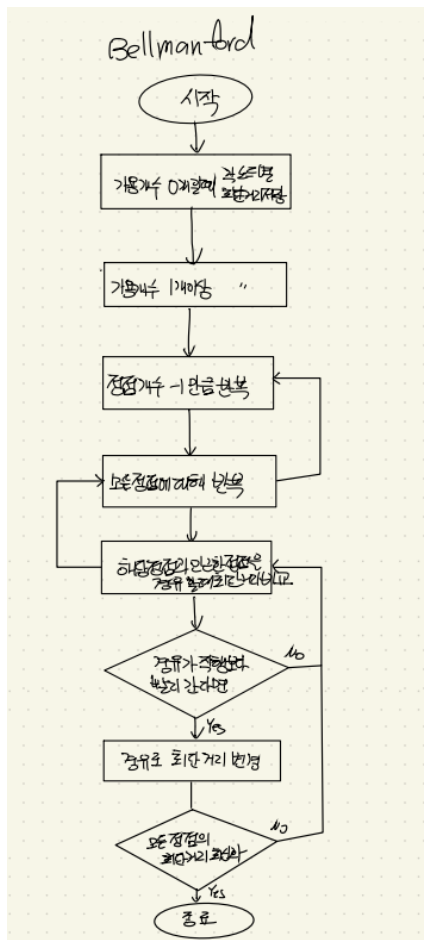
KRUSKAL은 min heap을 사용해 정점과 간선들의 정보를 저장한 후 적은 비용의 간선을 우선으로 택하며 진행한다. 중간에 부모 노드가 다른, 즉 사이클을 형성하지 않는 정점들을 MST에 저장해 가면서 모든 min heap에 저장된 정보를 추가해주며 진행한다.

#### 5. DIJKSTRA



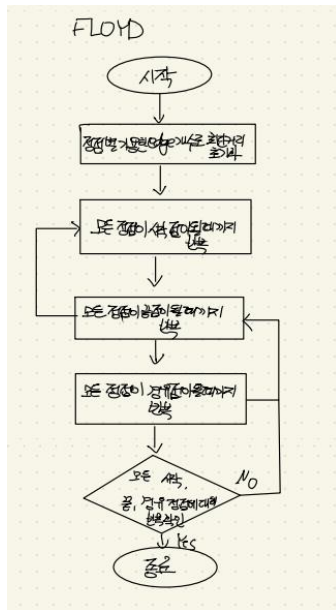
DIJKSTRA는 다이나믹 프로그래밍을 활용한 알고리즘으로 그때 그때 최선의 값을 구해 최종적으로 최선의 값, 즉 가장 적은 weight로 end vertex까지 갈 수 있는 path를 구한다. 때문에 방문하는 정점마다. 해당 정점으로 바로 오는 것과 현재 정점을 경유해서 오는 것의 차이를 비교해서 더 짧은 것을 해당 정점 까지의 weight로 설정한다.

## 6. BELLMANFORD



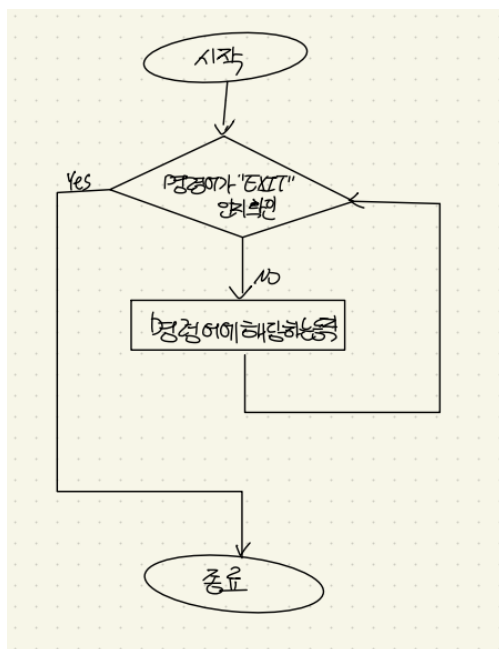
BELLMAN-FORD 알고리즘은 가용한 edge가 늘어날 때 마다 출발, 도착 정점의 최단거리를 최신회 하는 알고리즘이다. 가용한 edge가 추가되면서 현상 유지와 경유 후 도달 등의 조건이 생기는데 이 둘을 비교해 더 짧은 거리로 도달할 수 있는 것을 선택한다.

## 7. FLOYD



FLOYD 알고리즘은 시작, 경유, 도달 각각에 모든 정점들이 각각 참여하고 그렇기 때문에 한 정점에서 다른 한 정점까지의 최단거리를 모두 구할 수 있다. 때문에 3번의 반복문 중첩이 필요하다.

## 8. PROJECT



프로젝트는 command를 읽어서 해당 명령어에 해당하는 동작을 하고 EXIT 명령어가 확인되면 할당된 메모리를 해제하고 프로그램을 종료한다.

## ● Algorithm

### 1. BFS

```

30 bool BFS(Graph *graph, int vertex, ofstream &fout)
31 {
32     int cnt = 1;
33     int vis[graph->getSize() + 2] = {0}; // Check for vertex already visited
34
35     queue<int> Q;
36     Q.push(vertex);
37
38     vis[vertex] = 1;
39     fout << "startvertex: " << vertex << "\n"
40         << vertex << " -> ";
41     while (!Q.empty()) // Start BFS
42     {
43         auto cur = Q.front();
44         Q.pop();
45         for (auto edge : graph->getAdjacentEdges(cur)) // Searching Adjacent vertex
46         {
47             if (vis[edge.first]) // Already visited
48                 continue;
49             vis[edge.first] = 1; // Newly visit
50             Q.push(edge.first);
51
52             fout << edge.first;
53             if (++cnt < graph->getSize())
54                 fout << " -> ";
55         }
56     }
57     fout << "\n";
58     return true;
59 }
60

```

BFS는 너비 우선 탐색 알고리즘이다. 때문에 시작 정점을 먼저 큐에 넣고 주변 정점을 탐색과 동시에 방문한다. 큐에 먼저 들어간 정점을 먼저 확인하고 더 이상 확인할 정점이 없다면 그 다음에 들어간 정점을 기준으로 주변을 확인해 방문하는 방식의 반복이 이루어진다. 그렇기 때문에 방문한 정점과 하지 않은 정점을 구분해 주어야 하고 이것이 vis 배열을 사용한 이유다. 45번 줄을 보면 해당 점점에서 이동가능한 정점들을 확인하고 있다. 확인했다면 47번 줄에서 이전에 방문한 정점인지 확인하고 방문하지 않았다면 방문 후 큐에 집어넣는다.

위의 과정을 이동가능한 모든 정점에 대해 실행하고 알고리즘이 종료된다.

## 2. DFS & DFS\_R

```

61 bool DFS(Graph *graph, int vertex, ofstream &fout)
62 {
63     int cnt = 0;
64     int graph_size = graph->getSize();
65     int vis[graph_size + 2] = {0}; // Check for vertex already visited
66
67     stack<int> S;
68     S.push(vertex);
69
70     vis[vertex] = 1;
71     fout << "startvertex: " << vertex << "\n";
72     while (!S.empty()) // Start DFS
73     {
74         auto cur = S.top();
75         S.pop(); // A visiting vertex
76         fout << cur;
77         if (++cnt < graph_size)
78             fout << " -> ";
79         for (auto edges : graph->getAdjacentEdges(cur)) // Searching Adjacent vertex
80         {
81             if (vis[edges.first])
82                 continue;
83             vis[edges.first] = 1;
84             S.push(edges.first);
85         }
86     }
87     fout << "\n";
88     return true;
89 }
90

```

```

91 bool DFS_R(Graph *graph, vector<bool> *visit, int vertex, ofstream &fout)
92 {
93     visit[0][vertex] = true;
94     fout << vertex;
95     if (++dfs_cnt < graph->getSize())
96         fout << " -> ";
97     else // When visit all the vertex,
98         fout << "\n";
99     for (auto edges : graph->getAdjacentEdges(vertex)) // Searching Adjacent vertex
100     {
101         if (!visit[0][edges.first])
102             DFS_R(graph, visit, edges.first, fout); // Increasing depth
103     }
104 }
105

```

DFS는 깊이 우선 탐색 알고리즘이다. 현재 정점에서 주변 정점을 탐색해 한 정점을 방문하고 이 정점에서 이동가능한 다른 정점을 탐색해 그 정점으로 방문다. 더 이상 갈 정점이 없으면 그 정점, 즉 가장 최근에 도착했던 정점에서 방문할 수 있는 정점이 있는지 확인하고 있다면 방문하고 없다면 다시 돌아온다.

DFS가 BFS와 다른 점은 BFS는 현재 정점에서 주변 정점을 탐색해 가능한 모든 정점을 방문하는데 반해 DFS는 한 정점을 우선으로 방문하고 그 정점과 이웃한 다른 정점 중 하나를 방문한다는 것이다. 이 과정을 Depth가 깊어진다고 표현하며 DFS에서는 while문과 Stack을 통해 Depth를 표현하였고 DFS\_R에서는 재귀 함수의 호출을 통해 호출이 될 때마다 Depth가 깊어지는 것을 표현하였다. 이 부분은 각각 72번, 102번에서 확인할 수 있다.

### 3. KRUSKAL



```

12 void insertion(tuple<int, int, int> &e, vector<tuple<int, int, int>> &a, int i)
13 {
14     while (i > -1 && e < a[i]) // Insertion sort using sentinel
15     {
16         a[i + 1] = a[i];
17         i--;
18     }
19     a[i + 1] = e;
20 }
21
22 void insertion_sort(vector<tuple<int, int, int>> &a, const int low, const int high)
23 {
24     for (int j = low; j <= high; j++) // Check Every Element
25     {
26         tuple<int, int, int> temp = a[j];
27         insertion(temp, a, j - 1);
28     }
29 }
30

```

```

31 void quick_sort(vector<tuple<int, int, int>> &a, const int left, const int right)
32 {
33     if (left < right)
34     {
35         if (right - left + 1 <= 6)
36             insertion_sort(a, left, right); // if segment size is lower than 7 Do insertion sort
37         else
38         {
39             int i = left;
40             int j = right + 1;
41             tuple<int, int, int> pivot = a[left];
42             do // Compare using pivot
43             {
44                 do
45                 {
46                     i++;
47                     while (a[i] < pivot);
48                 } do
49                 {
50                     j--;
51                     while (a[j] > pivot);
52                     if (i < j)
53                         swap(a[i], a[j]);
54                 } while (i < j);
55                 swap(a[left], a[j]);
56
57                 quick_sort(a, left, j - 1); // divide
58                 quick_sort(a, j + 1, right); // divide
59             }
60         }
61     }
62 }

```

본 코드는 삽입, 퀵 정렬을 구현한 것이다. Segment size에 따라서 정렬 방법이 달라진다. 6이하면 퀵 정렬이 아닌 삽입 정렬을 사용해 정렬을 완성한다. Pivot을 정해 pivot 대로 구간을 나눠가면서 quick\_sort함수를 다시 호출하기 때문에 언젠가는 segment size가 6이하가 된다. 이 전까지는 퀵소트를 사용해 divide를 하다가 그 순간이 오면 삽입 정렬을 사용해 한 번에 정렬한다.

삽입 정렬은 원소가 들어갈 공간을 만들기 위해 뒤로 한 밀어서 생긴 공간에 삽입 하기 때문에 삽입 정렬이다. 여기서는 모든 원소를 대상으로 진행하고 sentinel을 사용했기 때문에 다른 조건문이 필요하지 않다.

퀵 정렬은 pivot을 정하고 왼쪽에서부터 pivot보다 큰 값을 찾음과 동시에 오른쪽에서부터

터 pivot보다 작은 값을 찾아간다. 찾은 다음 두 수를 바꿔서 정렬시킨다. 이 과정을 pivot을 사용해 구간을 나눠가면서 반복하면 정렬된 상태를 얻을 수 있다.

```

156 bool Kruskal(Graph *graph, ofstream &fout)
157 {
158     int sum = 0;
159     int graph_size = graph->getSize();
160     int *parent = new int[graph_size];
161     map<int, int> MST[graph_size]; // Minnium Spanning Tree
162     vector<tuple<int, int, int>> edges; // The relationship between all vertex and edge
163
164     for (int i = 0; i < graph_size; i++)
165     {
166         for (auto edge : graph->getAdjacentEdges(i))
167         {
168             tuple<int, int, int> tmp = make_tuple(edge.second, i, edge.first); // Make min
169             edges.push_back(tmp);
170         }
171     }
172     for (int i = 0; i < graph_size; i++)
173         parent[i] = i;
174
175     quick_sort(edges, 0, edges.size() - 1); // sorting edges
176
177     for (int i = 0; i < graph_size && !edges.empty(); i++) // Making MST
178     {
179         tuple<int, int, int> cur = edges.front();
180         edges.erase(edges.begin());
181         if (find_par_vertex(parent, get<1>(cur)) != find_par_vertex(parent, get<2>(cur)))
182         {
183             union_par_vertex(parent, get<1>(cur), get<2>(cur));
184             MST[get<1>(cur)].insert({get<2>(cur), get<0>(cur)});
185             MST[get<2>(cur)].insert({get<1>(cur), get<0>(cur)});
186             sum += get<0>(cur);
187         }
188     }
189     for (int i = 0; i < graph_size; i++)
190     {
191         fout << "[" << i << " ] ";
192         for (auto cur : MST[i])
193             fout << cur.first << "(" << cur.second << " ) ";
194         fout << "\n";
195     }
196     fout << "cost: " << sum << "\n";
197     if (sum < 0) // If MST cannot be created
198         return false;
199     return true;
200 }

```

위는 Kruskal 알고리즘의 구현 코드다. quick\_sort 까지는 앞서 확인이 되었고 이제 정렬된 quick\_sort에서 값을 하나씩 가져와 사이클 형성에 유의하면서 MST를 만든다. 사이클은 두 노드의 부모 노드를 확인하면 알 수 있다. 각 노드는 처음에 자기 자신과 동일한 부모 노드를 갖고 있지만 서로 다른 부모 노드를 갖는 것 확인이 되면 한 노드가 다른 노드의 부모 노드를 자신의 부모 노드로 갖게 된다. 만약 두 노드가 같은 부모 노드를 공유하고 있다면 사이클을 형성하는 것이므로 MST에 추가하지 않는다.

#### 4. DIJKSRA

```

153 bool Dijkstra(Graph *graph, int vertex, ofstream &fout)
154 {
155     int graph_size = graph->getSize();
156     int prev[graph_size];
157
158     vector<int> dist(graph_size, BIGBIG_NUM);
159     priority_queue<iPair, vector<iPair>, greater<iPair>> pq;
160
161     for (int i = 0; i < graph_size; i++) // Show past paths
162         prev[i] = i;
163
164     dist[vertex] = 0;
165     pq.push(make_pair(0, vertex));
166     fout << "startvertex: " << vertex << "\n";
167
168     while (!pq.empty()) // Start Dijkstra
169     {
170         int via_vertex = pq.top().second; // via vertex
171         pq.pop();
172
173         for (auto edge : graph->getAdjacentEdges(via_vertex)) // Check adjacent vertex
174         {
175             int dest_vertex = edge.first;
176             int weight_vertex = edge.second;
177
178             if (dist[dest_vertex] > dist[via_vertex] + weight_vertex) // If it's faster
179             {
180                 dist[dest_vertex] = dist[via_vertex] + weight_vertex;
181                 pq.push(make_pair(dist[dest_vertex], dest_vertex));
182                 prev[dest_vertex] = via_vertex;
183             }
184         }
185     }

```

```

186     for (int i = 0; i < graph_size; i++) // Print the past path
187     {
188         if (i == vertex)
189             continue;
190         fout << "[" << i << " ] ";
191         int cur_vertex = i;
192         if (prev[cur_vertex] != cur_vertex)
193         {
194             stack<int> S;
195             while (prev[i] != vertex)
196             {
197                 S.push(prev[i]);
198                 i = prev[i];
199             }
200             fout << vertex << " -> ";
201             while (!S.empty())
202             {
203                 fout << S.top() << " -> ";
204                 S.pop();
205             }
206             fout << cur_vertex << " (" << dist[cur_vertex] << ")"
207                 << "\n";
208         }
209         else
210             fout << "x"
211                 << "\n";
212         i = cur_vertex;
213     }
214 }
215

```

DIJKSTRA는 Dynamic programming을 사용한다. 때문에 시작 정점으로부터 현재까지 도달한 정점 까지의 거리를 유지하고 있는 우선순위 큐인 pq와 시작 정점으로부터 각 정점 까지의 최단 거리를 저장하고 있는 dist 배열이 필요하다. 현재 정점에서 인접한 정점까지의 weight를 찾는다. 인접한 정점 까지의 거리가 현재 정점에서 바로 가는 것보다. 인접한 정점을 경유해서 가는 것이 더 가깝다면 pq와 dist 배열을 더 가까운 값으로 초기화 해주고 이것을 목적지에 도달할 때까지, 혹은 가능한 모든 정점을 돌 때 까지 실행한다.

인접한 정점을 경유한다는 말에서 "인접한 정점 까지의 거리 + 인접한 정점에서 목적 정점까지의 거리"가 함축되어 있다. 인접한 정점 까지의 거리는 이전에 구한 값을 활용 한다는 의미이므로 동적 프로그래밍을 알아있다.

## 5. BELLMAN-FOR

```

217 bool Bellmanford(Graph *graph, int s_vertex, int e_vertex, ofstream &fout)
218 {
219     stack<int> S;
220     int end_vertex;
221     int graph_size = graph->getSize();
222     int prev[graph_size];
223     int dist[graph_size];
224
225     map<int, int> income_vet[graph_size]; // incoming vertices
226     for (int i = 0; i < graph_size; i++)
227     {
228         prev[i] = i;
229         dist[i] = BIGBIG_NUM;
230     }
231     dist[s_vertex] = 0; // The number of edges 0
232
233     for (auto edge : graph->getAdjacentEdges(s_vertex)) // The number of s_vertex's adjacent
234     {
235         dist[edge.first] = edge.second;
236         prev[edge.first] = s_vertex;
237     }
238     for (int i = 0; i < graph_size; i++) // Find incoming vertices
239     {
240         for (auto edge : graph->getAdjacentEdges(i))
241             income_vet[edge.first].insert({i, edge.second});
242     }
243
244     for (int k = 2; k <= graph_size - 1; k++) // Number of available edges
245     {
246         for (int v = 0; v < graph_size; v++) // Check shortest distance for all vertices
247         {
248             if (income_vet[v].empty() || v == s_vertex)
249                 continue;
250             for (int w = 0; w < graph_size; w++) // A direct comparison between diesel and
251             {
252                 if (w == v)
253                     continue;
254
255                 for (auto edge : graph->getAdjacentEdges(w))
256                 {
257                     if (v == edge.first)
258                     {
259                         if (dist[v] > dist[w] + graph->getAdjacentEdges(w).find(v)->second)
260                         {
261                             prev[v] = w;
262                             dist[v] = dist[w] + graph->getAdjacentEdges(w).find(v)->second;
263                             if (dist[v] < 0) // The negative number cycle generates
264                                 return false;
265                         }
266                     }
267                 }
268             }
269         }
270
271         end_vertex = e_vertex;
272         while (prev[end_vertex] != s_vertex) // Print Shortest distance path from start to finish
273         {
274             S.push(prev[end_vertex]);
275             end_vertex = prev[end_vertex];
276         }
277         fout << s_vertex << " -> ";
278         while (!S.empty())
279         {
280             fout << S.top() << " -> ";
281             S.pop();
282         }
283         fout << e_vertex << "\n";
284         fout << "cost: " << dist[e_vertex] << "\n";
285         return true;
286     }
287 }

```

Bellman-ford 알고리즘은 현재 정점에서 갈 수 있는 정점이 아닌, 가야하는 정점으로 갈 수 있는 정점을 구해야 한다. 그렇기 때문에 238-242까지 해당 연산을 진행해 주었다. 갈 수 있는 edge가 0개, 1개 일 때는 각 정점 별 인접한 edge를 구할 수 있기 때문에 찾을 수 있다.

때문에 244번의 k=2부터 반복문을 시작했다. 가려는 정점으로 이어주는 edge가 존재하지 않을 때는 다른 정점을 택해야 한다. 가려는 정점으로 이동할 수 있다고 하더라도 현재 정점을 경유해 가는 것보다 가려는 정점으로 직접 가는 것이 더 오래 걸린다면 의미가 없다. 위는 각각 248, 258번 라인을 뜻한 것이며 이 조건들을 모두 통과했다면 도착 정점까지의 거리를 더 짧은 거리로 최신회 할 수 있다. 이동 경로들을 출력해주기 위해서 prev 배열을 사용하였다. 각 인덱스, 즉 정점마다. 경유했던 정점을 값으로 할당했기 때문에 prev 배열을 타고 이동하면 시작했던 정점까지 알 수 있다.

## 6. FLOYD

```

288 bool FLOYD(Graph *graph, ofstream &fout)
289 {
290     int graph_size = graph->getSize();
291     int dist_board[graph_size][graph_size];
292
293     for (int i = 0; i < graph_size; i++) // Initialize distance board 1
294     {
295         for (int j = 0; j < graph_size; j++)
296         {
297             if (i == j)
298                 dist_board[i][j] = 0;
299             else
300                 dist_board[i][j] = BIGBIG_NUM;
301         }
302     }
303
304     for (int i = 0; i < graph_size; i++) // Initialize distance board 2
305     {
306         for (auto edge : graph->getAdjacentEdges(i))
307             dist_board[i][edge.first] = edge.second;
308     }
309
310     for (int k = 0; k < graph_size; k++) // As the edge increases, the minimum distance is found again.
311     {
312         for (int i = 0; i < graph_size; i++)
313         {
314             for (int j = 0; j < graph_size; j++)
315             {
316                 if (i == j || j == k || i == k || dist_board[i][j] == dist_board[i][k] + dist_board[k][j])
317                     continue;
318                 dist_board[i][j] = min(dist_board[i][j], dist_board[i][k] + dist_board[k][j]); // Find smallest dist
319                 if (dist_board[i][j] < 0) // The negative number
320                     return false;
321             }
322         }
323     }
324
325     for (int i = 0; i < graph_size; i++) // Print distance board
326     {
327         if (i == 0)
328         {
329             fcout << "      ";
330             for (int k = 0; k < graph_size; k++)
331                 fcout << "[" << k << "]\t";
332             fcout << "\n";
333         }
334         for (int j = 0; j < graph_size; j++)
335         {
336             if (j == 0)
337                 fcout << "[" << i << "]\t";
338             if (dist_board[i][j] == BIGBIG_NUM)
339                 fcout << 'x' << "\t";
340             else
341                 fcout << dist_board[i][j] << "\t";
342         }
343         fcout << "\n";
344     }
345     return true;
346 }

```

FLOYD알고리즘은 임의의 정점에서 다른 임의의 정점까지의 최단거리를 전부 구할 수 있는 알고리즘이다. 때문에 3번의 반복이 필요하다. 첫번째 반복문은 모든 정점이 한번씩

시작위치에 선다는 뜻이고 두번째 반복문은 모든 정점이 한번씩 경유 되는 정점에 선다는 뜻이며 마지막 세번째 반복문은 모든 정점이 한번씩 도착 점점이 된다는 뜻이다.

이 현산을 진행하기 전에 우리는 한 정점에서 인접한 정점들 까지의 거리를 알고 있기 때문에 최초 거리를 할당해 줄 수 있다. 어떤 노드에서 도달 불가능한 정점은 weight 값이 될 수 없는 큰 값(무한대)을 할당해 준다.

위의 BELLMAN-FORD 알고리즘과 마찬가지로 지나왔던 경로를 저장할 prev 배열을 선언해 주고 각 정점에 해당하는 배열의 인덱스에 경유 정점을 값으로 할당에 되돌아 가면서 시작 정점을 찾을 수 있게 한다.

- Result Screen

Log.txt 출력 값

log.txt

```
1  =====LOAD=====
2  SUCCESS
3  =====
4
5  =====PRINT=====
6  [0] -> (1,6) -> (2,2)
7  [1] -> (3,5)
8  [2] -> (1,7) -> (4,3) -> (5,8)
9  [3] -> (6,3)
10 [4] -> (3,4)
11 [5] -> (6,1)
12 [6] -> (4,10)
13
14 =====
15
16 =====BFS=====
17 startvertex: 0
18 0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6
19 =====
20
21 =====DFS=====
22 startvertex: 0
23 0 -> 2 -> 5 -> 6 -> 4 -> 3 -> 1
24 =====
25
26 =====DFS_R=====
27 2 -> 1 -> 3 -> 6 -> 4 -> 5
28 =====
29
30 =====KRUSKAL=====
31 [0] 2(2)
32 [1] 3(5)
33 [2] 0(2) 4(3)
34 [3] 1(5) 4(4) 6(3)
35 [4] 2(3) 3(4)
36 [5] 6(1)
37 [6] 3(3) 5(1)
38 cost: 18
39 =====
40
41 =====DIJKSTRA=====
42 startvertex: 5
43 [0] x
44 [1] x
45 [2] x
```

```

41 | =====DIJKSTRA=====
42 | startvertex: 5
43 | [0] x
44 | [1] x
45 | [2] x
46 | [3] 5 -> 6 -> 4 -> 3 (15)
47 | [4] 5 -> 6 -> 4 (11)
48 | [6] 5 -> 6 (1)
49 | =====
50 |
51 | =====BELLMANFORD=====
52 | 0 -> 2 -> 4 -> 3
53 | cost: 9
54 | =====
55 |
56 | =====FLOYD=====
57 | | [0] [1] [2] [3] [4] [5] [6]
58 | [0] 0 6 2 9 5 10 11
59 | [1] x 0 x 5 18 x 8
60 | [2] x 7 0 7 3 8 9
61 | [3] x x x 0 13 x 3
62 | [4] x x x 4 0 x 7
63 | [5] x x x 15 11 0 1
64 | [6] x x x 14 10 x 0
65 | =====
66 |
67 |

```

전체 결과는 위와 같다.

- LOAD



```

155 fgraph.getline(buf, 128);
156 graph_type = (buf[0] == 'M' ? 1 : 0); // Check Matrix or List
157 fgraph.getline(buf, 128);
158 graph_size = stoi(buf); // Find graph size
159 graph = new ListGraph(graph_type, graph_size);
160
161 if (!graph_type)
162 {
163     while (fgraph.getline(buf, 128)) // Read edges and weight
164     {
165         char *tmp;
166         if (!strchr(buf, ' '))
167         {
168             from = 0, to = 0, weight = 0;
169             from = stoi(buf);
170         }
171         else
172         {
173             tmp = strtok(buf, " ");
174             to = stoi(tmp);
175             tmp = strtok(NULL, "\n");
176             weight = stoi(tmp);
177             graph->insertEdge(from, to, weight);
178             value_checker.push_back(from);
179         }
180     }
181 }
182 fcout << "SUCCESS\n";
183 return true;
184 }
185
186 bool Manager::PRINT(ofstream &fout)
187 {
188     if (graph->printGraph(fout))
189         return true;
190     return false;
191 }

```

LOAD는 graph.txt 파일에서 정점과 edge의 weight 값을 가져와 graph라는 map에 넣어 준다. 왼쪽을 보면 두번째 값이 정점 0에서 정점 2로 이동하는데 weight가 2인 값이 할당되고 있는 것을 확인할 수 있다.

#### - PRINT

```

13 delete[] m_List;
14 }
15
16 map<int, int> ListGraph::getAdjacentEdges(int vertex)
17 {
18     return m_List[vertex];
19 }
20
21 void ListGraph::insertEdge(int from, int to, int weight)
22 {
23     m_List[from].insert({to, weight});
24 }
25
26 bool ListGraph::printGraph(ofstream &fout)
27 {
28     if (m_List.empty())
29         return false;
30     for (int i = 0; i < m_Size; i++)
31     {
32         fcout << "[" << i << " ";
33         for (auto it_ = m_List[i].begin(); it_ != m_List[i].end() && fcout << " -> "; it_++)
34         {
35             fcout << "(" << it_->first << ", " << it_->second << ") ";
36         }
37         fcout << "\n";
38     }
39     fcout << "\n";
40     return true;
41 }
42 }

```

PRINT를 보면 LOAD에서 추가한 값이 m\_List에 잘 들어가 있음을 확인할 수 있고 iterator를 활용해 m\_List의 정점 별 값을 잘 뽑아내 출력하고 있는 것을 볼 수 있다.

#### - BFS

```

26 |     else
27 |     {
28 |         par_vertex[x] = y;
29 |     }
30 |
31 | bool BFS(Graph *graph, int vertex, ofstream &fout)
32 | {
33 |     int cnt = 1;
34 |     int vis[graph->getSize() + 2] = {0}; // Check for vertex already visited
35 |
36 |     queue<int> Q;
37 |     Q.push(vertex);
38 |
39 |     vis[vertex] = 1;
40 |     fout << "startvertex: " << vertex << "\n"
41 |         << vertex << " -> ";
42 |     while (!Q.empty()) // Start BFS
43 |     {
44 |         auto cur = Q.front();
45 |         Q.pop();
46 |         for (auto edge : graph->getAdjacentEdges(cur)) // Searching Adjacent vertex
47 |         {
48 |             if (vis[edge.first]) // Already vistied
49 |                 continue;
50 |             vis[edge.first] = 1; // Newly visit
51 |             Q.push(edge.first);
52 |
53 |             fout << edge.first;
54 |             if (++cnt < graph->getSize())
55 |                 fout << " -> ";
56 |         }
57 |     }
58 |     fout << "\n";
59 |     return true;
60 | }

```

**Locals:**

- edge: {...}
  - first: 1
  - second: 6
- \_for\_range: {...}
  - \_for\_begin: {first = 1, second = ...}
  - \_for\_end: {first = 2, second = ...}
- cur: 0
- cnt: 1
- vis: [9]
  - [0]: 1
  - [1]: 0
  - [2]: 0
  - [3]: 0
  - [4]: 0
  - [5]: 0
  - [6]: 0
  - [7]: 0
  - [8]: 0
- Q: {...}
  - [0]: 0
- graph: 0x55555589480
  - m\_Type: false
  - m\_Size: 7
  - vertex: 0
  - fout: {...}

```

26 |     else
27 |     {
28 |         par_vertex[x] = y;
29 |     }
30 |
31 | bool BFS(Graph *graph, int vertex, ofstream &fout)
32 | {
33 |     int cnt = 1;
34 |     int vis[graph->getSize() + 2] = {0}; // Check for vertex already v
35 |
36 |     queue<int> Q;
37 |     Q.push(vertex);
38 |
39 |     vis[vertex] = 1;
40 |     fout << "startvertex: " << vertex << "\n"
41 |         << vertex << " -> ";
42 |     while (!Q.empty()) // Start BFS
43 |     {
44 |         auto cur = Q.front();
45 |         Q.pop();
46 |         for (auto edge : graph->getAdjacentEdges(cur)) // Searching Ad
47 |         {
48 |             if (vis[edge.first]) // Already vistied
49 |                 continue;
50 |             vis[edge.first] = 1; // Newly visit
51 |             Q.push(edge.first);
52 |
53 |             fout << edge.first;
54 |             if (++cnt < graph->getSize())
55 |                 fout << " -> ";
56 |         }
57 |     }
58 |     fout << "\n";
59 |     return true;
60 | }

```

**Locals:**

- edge: {...}
  - first: 1
  - second: 6
- \_for\_range: {...}
  - \_for\_begin: {first = 1, second = ...}
  - \_for\_end: {first = 2, second = ...}
- cur: 0
- cnt: 1
- vis: [9]
  - [0]: 1
  - [1]: 1
  - [2]: 0
  - [3]: 0
  - [4]: 0
  - [5]: 0
  - [6]: 0
  - [7]: 0
  - [8]: 0
- Q: {...}
  - [0]: 1
  - [1]: 0
- graph: 0x55555589480
  - m\_Type: false
  - m\_Size: 7
  - vertex: 0
  - fout: {...}

첫번째 사진을 보면 아직 방문하지 않은 현재 주변 정점의 방문 여부를 true로 바꾸고 있고 두번째 사진을 보면 현재 주변 정점을 방문해 큐에 집어넣고 있는 것을 확인할 수 있다.

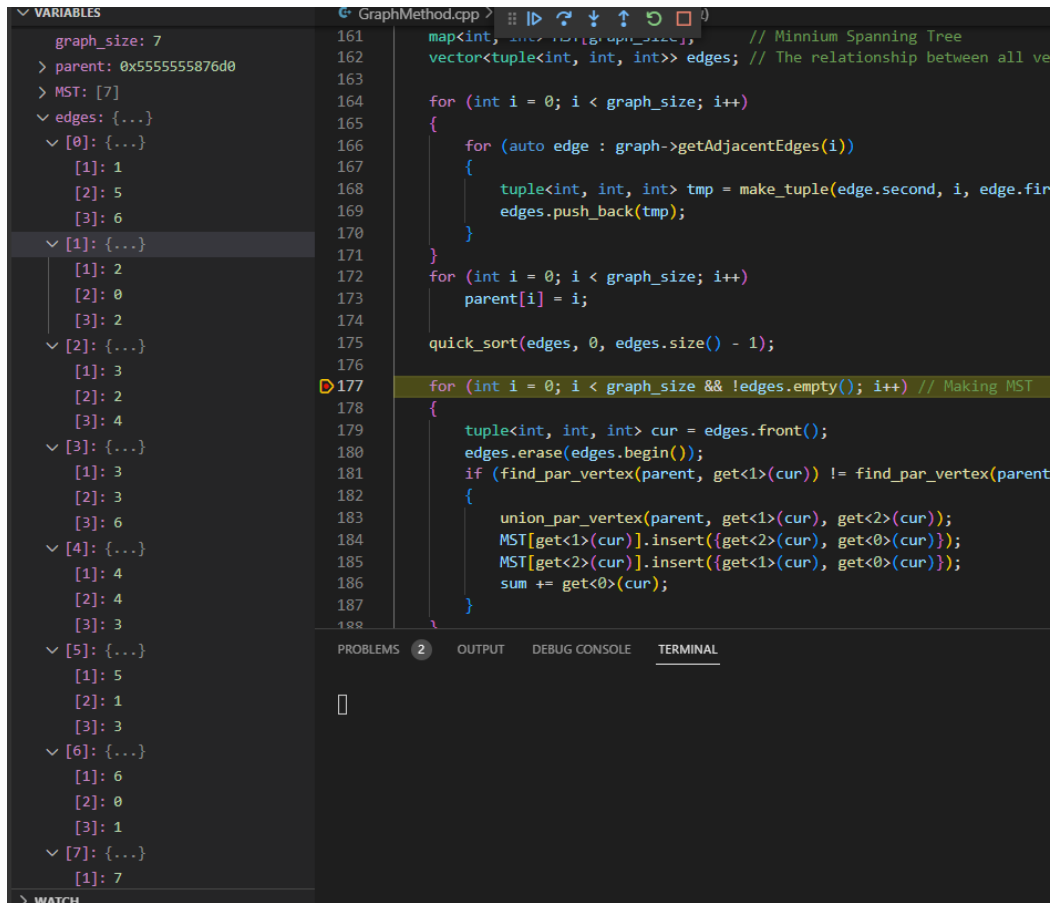
- DFS

74, 75번 줄을 보면 pop한 지점부터 탐색을 시작하는 것을 확인할 수 있다. BFS와 마찬가지로 주변 정점을 방문 배열인 vis에 추가하지만 탐색은 pop한 순간부터 시작되기 때문에 fout을 76번 줄에 선언해 주었음을 확인할 수 있다.

## - DFS\_R

첫번째, 두번째 그림을 보면 깊어진 depth가 다시 알아지는 것을 알 수 있다.

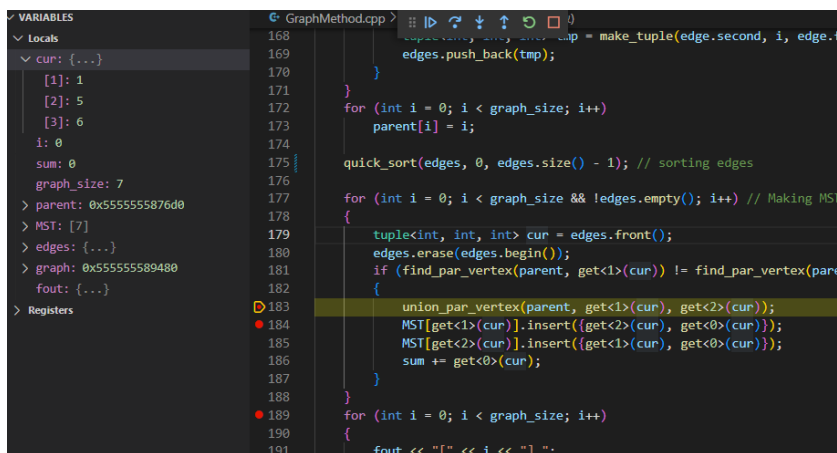
## - KRUSKAL



```
graph_size: 7
> parent: 0x555555876d0
> MST: [7]
  > edges: {...}
    > [0]: {...}
      [1]: 1
      [2]: 5
      [3]: 6
    > [1]: {...}
      [1]: 2
      [2]: 0
      [3]: 2
    > [2]: {...}
      [1]: 3
      [2]: 2
      [3]: 4
    > [3]: {...}
      [1]: 3
      [2]: 3
      [3]: 6
    > [4]: {...}
      [1]: 4
      [2]: 4
      [3]: 3
    > [5]: {...}
      [1]: 5
      [2]: 1
      [3]: 3
    > [6]: {...}
      [1]: 6
      [2]: 0
      [3]: 1
    > [7]: {...}
      [1]: 7
```

```
161 map<int, int, tuple<int, int, int>> edges; // Minnium Spanning Tree
162 vector<tuple<int, int, int>> edges; // The relationship between all ve
163
164 for (int i = 0; i < graph_size; i++)
165 {
166     for (auto edge : graph->getAdjacentEdges(i))
167     {
168         tuple<int, int, int> tmp = make_tuple(edge.second, i, edge.fir
169         edges.push_back(tmp);
170     }
171 }
172 for (int i = 0; i < graph_size; i++)
173     parent[i] = i;
174
175 quick_sort(edges, 0, edges.size() - 1);
176
177 for (int i = 0; i < graph_size && !edges.empty(); i++) // Making MST
178 {
179     tuple<int, int, int> cur = edges.front();
180     edges.erase(edges.begin());
181     if (find_par_vertex(parent, get<1>(cur)) != find_par_vertex(pare
182     {
183         union_par_vertex(parent, get<1>(cur), get<2>(cur));
184         MST[get<1>(cur)].insert({get<2>(cur), get<0>(cur)});
185         MST[get<2>(cur)].insert({get<1>(cur), get<0>(cur)});
186         sum += get<0>(cur);
187     }
188 }
```

위에서 설명한 쿼, 삽입 정렬을 사용해 오름차순으로 잘 정렬이 되어 있는 것을 확인할 수 있다. 이제 front()를 사용해 가장 weight 가 낮은 edge부터 뽑아서 MST를 만들 수 있다.



```
168 tuple<int, int, int> tmp = make_tuple(edge.second, i, edge.f
169 edges.push_back(tmp);
170 }
171 }
172 for (int i = 0; i < graph_size; i++)
173     parent[i] = i;
174
175 quick_sort(edges, 0, edges.size() - 1); // sorting edges
176
177 for (int i = 0; i < graph_size && !edges.empty(); i++) // Making MST
178 {
179     tuple<int, int, int> cur = edges.front();
180     edges.erase(edges.begin());
181     if (find_par_vertex(parent, get<1>(cur)) != find_par_vertex(pare
182     {
183         union_par_vertex(parent, get<1>(cur), get<2>(cur));
184         MST[get<1>(cur)].insert({get<2>(cur), get<0>(cur)});
185         MST[get<2>(cur)].insert({get<1>(cur), get<0>(cur)});
186         sum += get<0>(cur);
187     }
188 }
189 for (int i = 0; i < graph_size; i++)
190 {
191     fout << "[" << i << " ] ";
```

Weight가 가장 낮은 것부터 MST에 추가되는 모습을 볼 수 있다.

```

168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200

```

MST가 잘 만들어진 것을 확인할 수 있다.

## - DIJKSTRA

```

155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182

```

```

155 int prev[graph_size];
156
157 vector<int> dist(graph_size, BIGBIG_NUM);
158 priority_queue<Pair, vector<Pair>, greater<Pair>> pq;
159
160 dist[vertex] = 0;
161 pq.push(make_pair(0, vertex));
162 fout << "startvertex: " << vertex << "\n";
163
164 for (int i = 0; i < graph_size; i++) // Show past paths
165     prev[i] = i;
166 while (!pq.empty()) // Start Dijkstra
167 {
168     int via_vertex = pq.top().second; // via vertex
169     pq.pop();
170
171     for (auto edge : graph->getAdjacentEdges(via_vertex)) // Check adjacent edges
172     {
173         int dest_vertex = edge.first;
174         int weight_vertex = edge.second;
175
176         if (dist[dest_vertex] > dist[via_vertex] + weight_vertex) // If current distance is greater
177         {
178             dist[dest_vertex] = dist[via_vertex] + weight_vertex;
179             pq.push(make_pair(dist[dest_vertex], dest_vertex));
180             prev[dest_vertex] = via_vertex;
181         }
182     }
183 }
184 for (int i = 0; i < graph_size; i++) // Print the past path
185 {

```

첫번째, 두번째 사진을 보면 아직 방문하지 않았던 정점을 이어주는 것을 확인할 수 있다.

```

183 }
184 for (int i = 0; i < graph_size; i++) // Print the past path
185 {
186     if (i == vertex)
187         continue;
188     fout << "[" << i << " ";
189     int cur_vertex = i;

```

위의 사진을 보면 도달 불가능한 0, 1, 2 정점을 제외하고는 최단거리 최신했가 완료된 것을 확인할 수 있다.

- BELLMANFORD

```

VARIABLES
  Locals
    edge: {...}
    __for_range: {...}
    __for_begin: {first = 3, second = 5}
    __for_end: {first = 1, second = 0}
    w: 1
    v: 3
    k: 2
    S: {...}
    end_vertex: 2
    graph_size: 7
    prev: [7]
    dist: [7]
      [0]: 0
      [1]: 6
      [2]: 2
      [3]: 214748364
      [4]: 214748364
      [5]: 214748364
      [6]: 214748364
    lead_up_vertex: [7]
    graph: 0x555555589480
    s_vertex: 0
    e_vertex: 3
    fout: {...}
  Registers

GraphM
  236 for (int i = 0; i < graph_size; i++) // Find leading up vertices
  237 {
  238   for (auto edge : graph->getAdjacentEdges(i))
  239     lead_up_vertex[edge.first].insert({i, edge.second});
  240 }
  241
  242 for (int k = 2; k <= graph_size - 1; k++) // Number of available edges
  243 {
  244   for (int v = 0; v < graph_size; v++) // Check shortest distance for v
  245   {
  246     if (lead_up_vertex[v].empty() || v == s_vertex)
  247       continue;
  248     for (int w = 0; w < graph_size; w++) // A direct comparison between v and w
  249     {
  250       if (w == v)
  251         continue;
  252       for (auto edge : graph->getAdjacentEdges(w))
  253       {
  254         if (v == edge.first)
  255         {
  256           if (dist[v] > dist[w] + graph->getAdjacentEdges(w).weight())
  257           {
  258             prev[v] = w;
  259             dist[v] = dist[w] + graph->getAdjacentEdges(w).weight();
  260             if (dist[v] < 0) // The negative number cycle is detected
  261               return false;
  262           }
  263         }
  264       }
  265     }
  266   }
  267 }
  268
  269 end_vertex = e_vertex;
  270 while (prev[end_vertex] != s_vertex) // Print Shortest distance path

```

```

VARIABLES
  Locals
    edge: {...}
    __for_range: {...}
    __for_begin: {first = 3, second = 5}
    __for_end: {first = 1, second = 0}
    w: 1
    v: 3
    k: 2
    S: {...}
    end_vertex: 2
    graph_size: 7
    prev: [7]
    dist: [7]
      [0]: 0
      [1]: 0
      [2]: 0
      [3]: 1
      [4]: 4
      [5]: 5
      [6]: 6
    lead_up_vertex: [7]
    graph: 0x555555589480
    s_vertex: 0
    e_vertex: 3
    fout: {...}
  Registers

GraphM
  236 for (int i = 0; i < graph_size; i++) // Find leading up vertices
  237 {
  238   for (auto edge : graph->getAdjacentEdges(i))
  239     lead_up_vertex[edge.first].insert({i, edge.second});
  240 }
  241
  242 for (int k = 2; k <= graph_size - 1; k++) // Number of available edges
  243 {
  244   for (int v = 0; v < graph_size; v++) // Check shortest distance for v
  245   {
  246     if (lead_up_vertex[v].empty() || v == s_vertex)
  247       continue;
  248     for (int w = 0; w < graph_size; w++) // A direct comparison between v and w
  249     {
  250       if (w == v)
  251         continue;
  252       for (auto edge : graph->getAdjacentEdges(w))
  253       {
  254         if (v == edge.first)
  255         {
  256           if (dist[v] > dist[w] + graph->getAdjacentEdges(w).weight())
  257           {
  258             prev[v] = w;
  259             dist[v] = dist[w] + graph->getAdjacentEdges(w).weight();
  260             if (dist[v] < 0) // The negative number cycle is detected
  261               return false;
  262           }
  263         }
  264       }
  265     }
  266   }
  267 }
  268
  269 end_vertex = e_vertex;
  270 while (prev[end_vertex] != s_vertex) // Print Shortest distance path

```

위의 두 사진은 v까지 도달하기 위한 최단 경로를 구하는 과정이다. 3번 정점으로 가는데 1번 정점을 통해 가는 것이 가장 빠르고 1번 정점을 가기 위해서는 0번 정점을 거쳐야 하는데 이것 또한 prev 배열에 저장되어 있음을 알 수 있다.

0번 정점에서 3번 정점 까지의 경로를 구하고 싶다면 path[3]의 결과인 1을 path[1]을 해서 0을 구해서 역순으로 출력하면 가능하다.

```

S: {...}
end_vertex: 2
graph_size: 7
prev: [7]
  [0]: 0
  [1]: 0
  [2]: 0
  [3]: 4
  [4]: 2
  [5]: 2
  [6]: 5
dist: [7]
  [0]: 0
  [1]: 6
  [2]: 2
  [3]: 9
  [4]: 5
  [5]: 10
  [6]: 11
lead_up_vertex: [7]
graph: 0x55555589480
s_vertex: 0
e_vertex: 3
fout: {...}
Registers

243 {
244   for (int v = 0; v < graph_size; v++) // Check shortest distance for
245   {
246     if (lead_up_vertex[v].empty() || v == s_vertex)
247       continue;
248     for (int w = 0; w < graph_size; w++) // A direct comparison betw
249     {
250       if (w == v)
251         continue;
252       for (auto edge : graph->getAdjacentEdges(w))
253       {
254         if (v == edge.first)
255         {
256           if (dist[v] > dist[w] + graph->getAdjacentEdges(w).t
257           {
258             prev[v] = w;
259             dist[v] = dist[w] + graph->getAdjacentEdges(w).t
260             if (dist[v] < 0) // The negative number cycle ge
261               return false;
262           }
263         }
264       }
265     }
266   }
267 }
268
269 end_vertex = e_vertex;
270 while (prev[end_vertex] != s_vertex) // Print Shortest distance path fr
271 {
272   S.push(prev[end_vertex]);
273   end_vertex = prev[end_vertex];
274 }
275 fout << s_vertex << " -> ";
276 while (!S.empty())
277 {
278   fout << S.top() << " -> ";
279   S.pop();

```

위의 결과를 확인하면 prev를 통해 최단 경로를 추적할 수 있음을 확인할 수 있다.

## - FLOYD

```

Locals
j: 2
i: 1
k: 0
graph_size: 7
dist_board: [7]
  [0]: 0
  [1]: 6
  [2]: 2
  [3]: 214748364
  [4]: 214748364
  [5]: 214748364
  [6]: 214748364
  [1]: 0
  [2]: 214748364
  [3]: 5
  [4]: 214748364
  [5]: 214748364
  [6]: 214748364
  [2]: 0
  [3]: 214748364
  [4]: 3
  [5]: 8
  [6]: 214748364
> [3]
> [4]
> [5]

296   dist_board[i][j] = 0;
297   else
298     dist_board[i][j] = BIGBIG_NUM;
299   }
300 }
301
302 for (int i = 0; i < graph_size; i++) // Initialize distance board 2
303 {
304   for (auto edge : graph->getAdjacentEdges(i))
305     dist_board[i][edge.first] = edge.second;
306 }
307
308 for (int k = 0; k < graph_size; k++) // As the edge increases, the minimum distance is
309 {
310   for (int i = 0; i < graph_size; i++)
311   {
312     for (int j = 0; j < graph_size; j++)
313     {
314       if (i == j || j == k || i == k || dist_board[i][j] == dist_board[i][k] + dist_board[k][j])
315         continue;
316       dist_board[i][j] = min(dist_board[i][j], dist_board[i][k] + dist_board[k][j]);
317       if (dist_board[i][j] < 0)
318         return false;
319     }
320   }
321 }
322
323 for (int i = 0; i < graph_size; i++) // Print distance board
324 {
325   if (i == 0)
326   {
327     fout << " ";
328     for (int k = 0; k < graph_size; k++)
329       fout << "[" << k << "]\t";
330     fout << "\n";
331   }
332   for (int j = 0; j < graph_size; j++)
333   {

```



위의 사진을 확인하면 1번 vertex에서 2번 vertex로 갈 때 직행과 1번 -> 0번 -> 2번으로 경유해서 가는 값을 비교한다. 당연히 차이가 없기 때문에 값이 유지되고

경유해서 갈 때 더 빠르게 갈 수 있다면 이렇게 값을 재할당 하는 것을 확인할 수 있다.

- Consideration

이번 과제는 지난 과제들에 비해 무난한 편이었지만 몇 가지 난관이 있었다.

첫번째는 ofstream이었다. Manager.cpp와 GraphMethod.cpp는 파일이 다르기 때문에 다른 파일간 데이터 공유를 위한 작업이 필요 하다. 처음에는 extern 사용해 다른 파일에서 해당 파일의 전역 변수를 참조할 수 있게 하려고 했다. 이는 동일한 fout을 넘겨주어 파일 입출력에 혼란을 주지 않기 위함이었었는데 class의 멤버 변수인 fout을 넘길 수 없어서 각 함수에 fout을 call by reference로 넘겨주어서 해결할 수 있었다.

ofstream인 fout은 주소 값으로 넘기지 않는다면 오류를 띄운다. 다른 주소를 갖지만 동일한 파일에 대해 ofstream을 연다면 충돌이 발생해 컴파일러가 미리 알려주는 것으로 보인다.

두번째는 벨만 포드 알고리즘이다. 이 알고리즘은 현 vertex에서 A라는 다른 vertex로 넘어갈 때 인접한 것을 현 vertex를 기준으로 검사하지 않는다. 모든 vertex에서 A vertex로 이동 가능한 vertex 만을 기준으로 경유 또는 시작 정점을 정하기 때문에 따로 각 정점 별 들어오는 edge를 저장한 배열 혹은 벡터를 만들어주어야 했다. 제공된 getAdjacentEdge를 역으로 사용해 쉽게 해결할 수 있었다.

다른 하나는 DFS\_R을 구현할 때 마지막으로 방문하면 "->" 문자를 띄우지 말아야 한다. 이 부분을 어떻게 구현할지 고민을 많이 했는데 첫번째는 전역 변수를 사용하는 것이었다. 두번째는 함수에 몇 개의 정점을 방문했는지 저장하는 변수를 만드는 것인데 두번째 방법이 더 깔끔해 보였지만 효율을 생각했을 때 값이 계속 복사되는 것 보다는 그냥 전역 변수 한 개를 계속 참조하는 것이 낫다고 생각해서 첫번째 방식으로 구현하였다.

알고리즘에 대한 공부를 먼저 하고 시작한다면 의외로 금방 해결할 수 있는 과제였다. 다익스트라와 크루스칼 알고리즘이 헛갈렸는데 이번 기회에 정리할 수 있었다. 하나 아쉬운 점은 Manager.cpp에서 strtok의 사용을 줄이고 싶었는데 코드 길이가 길어져 tok를 사용한 점이다. 이 부분을 함수로 구현한다면 더 안정적으로 동작할 것으로 예상된다.