

Data Structure

Practical Class - Week 13

Shortest Path

Adjacent(인접)

- 두 정점 u, v 를 이어주는 간선이 있는 경우

Connected(연결)

- 두 정점 u, v 사이에 경로가 존재하는 경우

Path(경로)

- 한 정점 u 에서 출발해 유한개의 간선을 거쳐 정점 v 로 도달할 수 있는 간선 집합

Simple Path(단순 경로)

- 한 정점 u 에서 v 로 향하는 경로들 중 같은 정점이나 간선을 중복해서 포함하지 않는 경로

Shortest Path(최단 경로)

- 한 정점 u 에서 v 로 향하는 경로들 중 간선들의 가중치 합이 최소가 되는 경로

Negative Cycle

- 한 정점 u 에서 v 로 향하는 최단경로의 가중치 합을 무한히 줄일 수 있을 때

Shortest Path Algorithm

Dijkstra

- 한 출발 정점에 대해 모든 도착 정점에 대한 최단거리 계산
- BFS + Greedy
- 모든 간선이 0이상의 가중치를 가져야만 정상적으로 동작한다
- 반복문으로 구현시 $O(V^2)$
- Heap을 사용해 구현 시 $O(E \log_2 V)$

Bellman Ford

- 한 출발 정점에 대해 모든 도착 정점에 대한 최단거리 계산
- 간선의 가중치가 음수여도 상관없다.
- Negative Cycle을 검출할 수 있다.
- 반복문으로 구현시 $O(VE)$
- SPFA알고리즘으로 성능 개선 가능

A* Algorithm

- 다익스트라 알고리즘에 각 정점에 휴리스틱 우선순위를 부여한 형태
- 특수한 경우 평균적으로 더 좋은 성능을 보여줌. (ex: 네비게이션)

Floyd Warshall Algorithm

- 그래프 g 내의 모든 두 정점 $\langle u, v \rangle$ 사이의 최단거리를 계산할 수 있다.
- $O(V^3)$ 시간이 걸린다.

Skeleton Code 1 (공통)

```
1  #pragma warning(disable: 4996)
2  #include<stdio.h>
3  #include<limits.h>
4  #include<stdbool.h>
5  #include<malloc.h>
6  #include<stdlib.h>
7
8  #define MAX_NODE_SIZE 1000    // 최대 정점의 수
9
10 #define INFINITY    123456789 // 무한대 가중치 (정점간 경로가 없는 경우)
11 #define NONE        -1        // 빈 정점 인덱스
12
13 typedef struct Graph Graph;    // 그래프 구조체
14
15 /**
16  * @brief 인접 행렬로 그래프를 표현하는 구조체
17  *        정점은 0 ~ (V-1) 사이의 인덱스를 가진다
18  */
19 typedef struct Graph{
20     int V;    // 정점의 수
21     int** W;  // 2차원 인접행렬 W[V][V]
22 };
```

Skeleton Code 2 (공통)

```
24  /**
25   * @brief      그래프 초기화 후 반환
26   *
27   * @param V      정점의 수
28   * @return Graph*  그래프
29   */
30  Graph* init_graph(const int V){
31      int i, j;
32      Graph* g = (Graph*)malloc(sizeof(Graph));
33      g->V = V;
34      g->W = (int**)malloc(sizeof(int*) * V);
35      for (i = 0; i < V; i += 1){
36          g->W[i] = (int*)malloc(sizeof(int*) * V);
37          for (j = 0; j < V; j += 1){
38              if (i == j){
39                  // 자기 자신으로 향하는 거리는 0
40                  g->W[i][j] = 0;
41              }
42              else{
43                  // 초기에는 모든 간선이 없는 것으로 표시
44                  g->W[i][j] = INFINITY;
45              }
46          }
47      }
48      return g;
49  }
```

Skeleton Code 3 (공통)

```
51  /**
52  * @brief 그래프 할당을 해제해주는 함수
53  *
54  * @param g 그래프 포인터
55  */
56  void release_graph(const Graph* g){
57      int i, j;
58      for (i = 0; i < g->V; i += 1){
59          free(g->W[i]);
60      }
61      free(g->W);
62      free(g);
63  }
```

Skeleton Code 4 (공통)

```
65  /**
66  * @brief          해당 그래프의 최단거리 정보와 경로를 출력해주는 함수
67  *
68  * @param g         그래프 구조체
69  * @param origin     출발 노드
70  * @param D          D[v] := 출발 노드로부터 v번 노드까지의 최단 거리
71  * @param P          P[v] := 출발 노드로부터 v번 노드에 도달하기 위해 직전에 경유해야 할 정점의 번호 (최단 경로상 바로 이전 정점)
72  */
73  void print_path_info(const Graph* g, const int origin, int *D, int *P){
74      int i, j;
75      int top = 0;
76      int stack[MAX_NODE_SIZE];
77
78      for (i = 0; i < g->V; i += 1){ // 모든 정점에 대해
79          // P배열을 사용해 i번 정점까지의 최단 경로를 스택에 역순으로 저장한다
80          int cur;
81          for (cur = i; cur != NONE; cur = P[cur]){
82              stack[top++] = cur;
83          }
84
85          if (D[i] == INFINITY){
86              // 도달할 수 없는 정점이라면 무시한다
87              printf(" - Path from '%d' to '%d' (length : INFINITY) NO PATH\n", origin, i);
88          }
89          else{
90              // 해당 정점까지의 최단 경로 정보를 출력한다
91              printf(" - Path from '%d' to '%d' (length : %d) ", origin, i, D[i]);
92              printf("{ ");
93
94              // 스택에 역순으로 저장된 경로 정보를 출력한다
95              while (top > 0){
96                  printf("%d", stack[--top]);
97                  if (top > 0){
98                      printf(" => ");
99                  }
100              }
101              printf(" }\n");
102          }
103      }
104  }
```

Skeleton Code 5 (공통)

```
106  /**
107   * @brief      그래프 g에 정점 source로 부터 정점 dest로 향하는 단방향 간선을 추가한다. weigh는 가중치다
108   *
109   * @param g      그래프 구조체
110   * @param source  출발 정점 번호
111   * @param dest    도착 정점 번호
112   * @param weight  간선의 가중치
113   */
114  void add_edge(const Graph* g, const int source, const int dest, const int weight){
115      g->W[source][dest] = weight;
116  }
```


Exercise 01 - Dijkstra

```
120  /**
121   * @brief          그래프 g에 대해 다익스트라 알고리즘을 수행한 후 결과를 출력하는 함수
122   *
123   * @param g         그래프 구조체
124   * @param origin     출발 정점의 인덱스
125   */
126  void Dijkstra(const Graph* g, const int origin){
127      int i, j;    //
128      int steps;   // 반복 횟수
129
130      // visited[v] := 현재 다익스트라 탐색 과정에서 정점 v가 이미 방문 되었는지 여부 T/F
131      bool* visited = (bool*)malloc(sizeof(bool) * g->V);
132
133      // D[v] := 정점 origin부터 정점 v로 향하는 최단 경로의 길이
134      //          경로가 존재하지 않는다면 D[v] == INFINITY
135      int* D = (int*)malloc(sizeof(int) * g->V);
136
137      // P[v] := 정점 origin에서 정점 v로 향하는 최단경로 상에서 v번 정점 바로 이전 정점 번호
138      //          P[origin] = NONE
139      int* P = (int*)malloc(sizeof(int) * g->V);
140
141      for (i = 0; i < g->V; i += 1){
142          D[i] = INFINITY;
143          P[i] = NONE;
144          visited[i] = false;
145      }
146
147      D[origin] = 0;
148
149      for (steps = 1; steps <= g->V; steps += 1){
150          

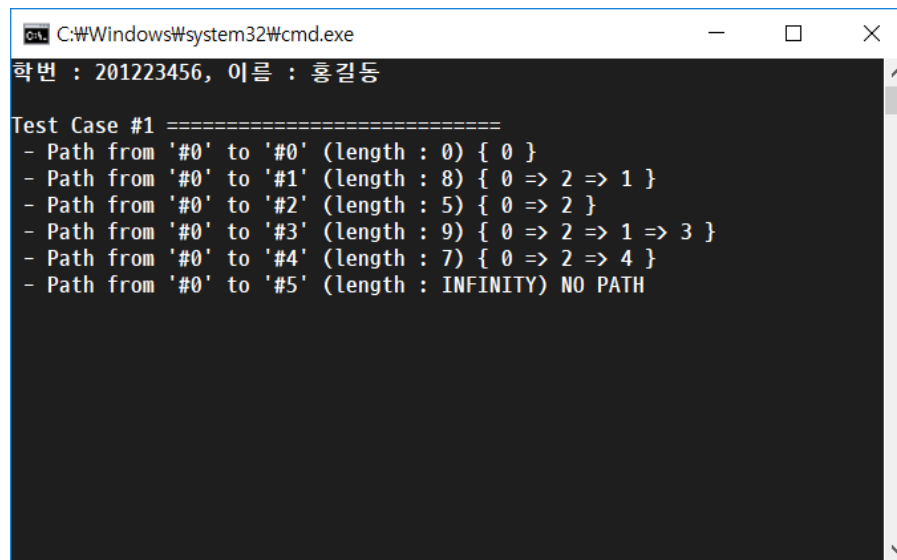
Fill your code Here


151      }
152
153      print_path_info(g, origin, D, P);
154  }
155
156 }
```

Exercise 01 - Example

아래 URL에서 테스트를 위한 코드들을 Copy & Paste하세요.

- <https://gist.github.com/waps12b/2049ac81e62d3ac5ccb0577074c1f313>



```
C:\Windows\system32\cmd.exe
학번 : 201223456, 이름 : 홍길동

Test Case #1 =====
- Path from '#0' to '#0' (length : 0) { 0 }
- Path from '#0' to '#1' (length : 8) { 0 => 2 => 1 }
- Path from '#0' to '#2' (length : 5) { 0 => 2 }
- Path from '#0' to '#3' (length : 9) { 0 => 2 => 1 => 3 }
- Path from '#0' to '#4' (length : 7) { 0 => 2 => 4 }
- Path from '#0' to '#5' (length : INFINITY) NO PATH
```

테스트케이스 1번에 대해 다익스트라를 수행한 결과 예시

Exercise 02 – Bellman Ford

```
119  /**
120   * @brief      그래프 g에 대해 벨만포드 알고리즘을 수행한 후 결과를 출력하는 함수
121   *
122   * @param g      그래프 구조체
123   * @param origin  출발 정점의 인덱스
124   */
125  void BellmanFord(const Graph* g, const int origin){
126      int i, j;    //
127      int steps;  // 반복 횟수
128
129      // 해당 그래프가 Negative Cycle을 가진다면 true로 변경시켜 주어야 할 변수
130      bool has_negative_cycle = false;
131
132      // D[v] := 정점 origin부터 정점 v로 향하는 최단 경로의 길이
133      //          경로가 존재하지 않는다면 D[v] == INFINITY
134      int* D = (int*)malloc(sizeof(int) * g->V);
135
136      // P[v] := 정점 origin에서 정점 v로 향하는 최단경로 상에서 v번 정점 바로 이전 정점 번호
137      //          P[origin] = NONE
138      int* P = (int*)malloc(sizeof(int) * g->V);
139
140      for (i = 0; i < g->V; i += 1){
141          D[i] = INFINITY;
142          P[i] = NONE;
143      }
144
145      D[origin] = 0;
146
147      for (steps = 1; steps <= g->V; steps += 1){
148          

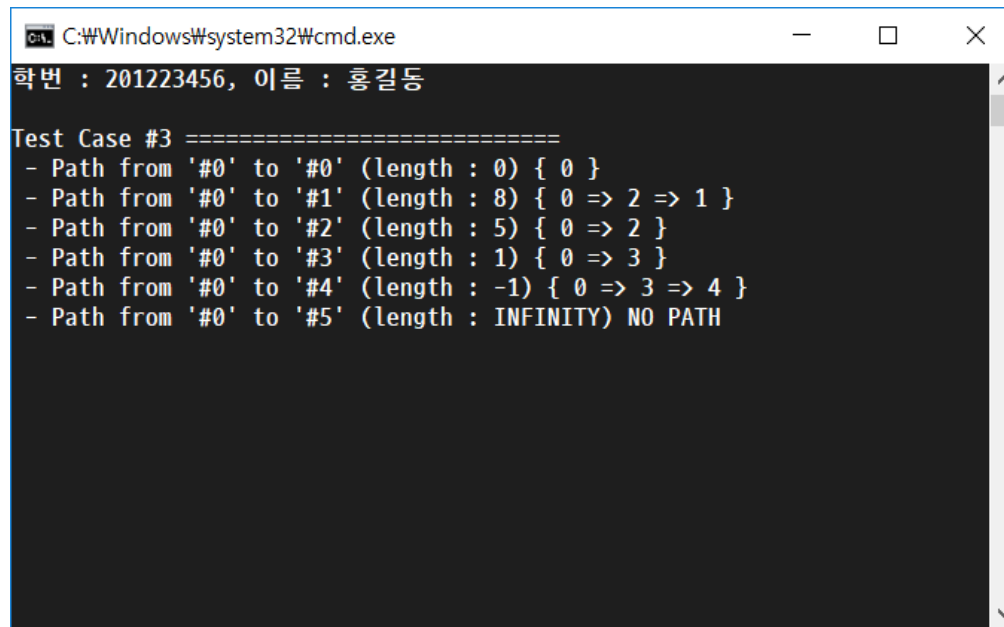
Fill your code Here


149      }
150
151
152      if (has_negative_cycle){
153          printf("This graph 'g' has negative cycle from origin node!!\n");
154          return;
155      }
156      else {
157          print_path_info(g, origin, D, P);
158      }
159  }
160 }
```

Exercise 02 - Example

아래 URL에서 테스트를 위한 코드들을 Copy & Paste하세요.

- <https://gist.github.com/waps12b/2049ac81e62d3ac5ccb0577074c1f313>

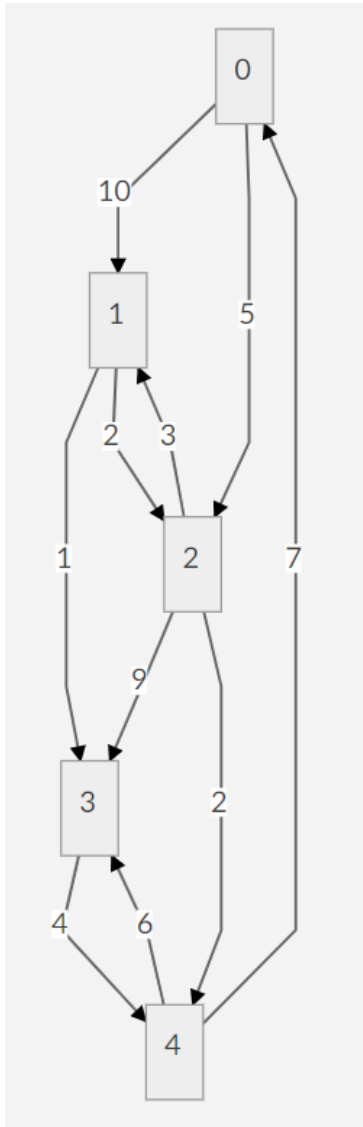


```
C:\Windows\system32\cmd.exe
학번 : 201223456, 이름 : 홍길동

Test Case #3 =====
- Path from '#0' to '#0' (length : 0) { 0 }
- Path from '#0' to '#1' (length : 8) { 0 => 2 => 1 }
- Path from '#0' to '#2' (length : 5) { 0 => 2 }
- Path from '#0' to '#3' (length : 1) { 0 => 3 }
- Path from '#0' to '#4' (length : -1) { 0 => 3 => 4 }
- Path from '#0' to '#5' (length : INFINITY) NO PATH
```

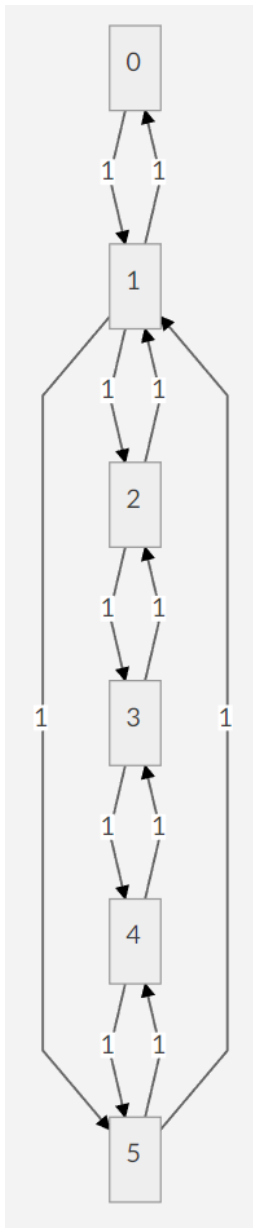
테스트케이스 3번에 대해 벨만포드를 수행한 결과 예시

Appendix - 테스트케이스 1



```
Graph* test_case_1(){
    Graph* g = init_graph(6);
    printf("Test Case #1 ===== \n");
    add_edge(g, 0, 1, 10);
    add_edge(g, 0, 2, 5);
    add_edge(g, 1, 2, 2);
    add_edge(g, 1, 3, 1);
    add_edge(g, 2, 1, 3);
    add_edge(g, 2, 3, 9);
    add_edge(g, 2, 4, 2);
    add_edge(g, 3, 4, 4);
    add_edge(g, 4, 0, 7);
    add_edge(g, 4, 3, 6);
    return g;
}
```

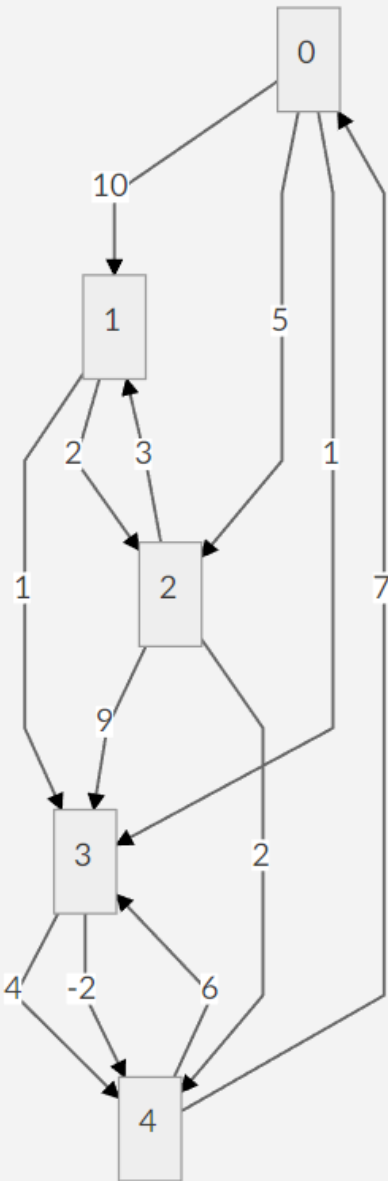
Appendix - 테스트케이스 2



```
Graph* test_case_2(){
    Graph* g = init_graph(6);
    printf("Test Case #2 ===== \n");
    add_edge(g, 0, 1, 1);
    add_edge(g, 1, 2, 1);
    add_edge(g, 2, 3, 1);
    add_edge(g, 3, 4, 1);
    add_edge(g, 4, 5, 1);
    add_edge(g, 5, 1, 1);

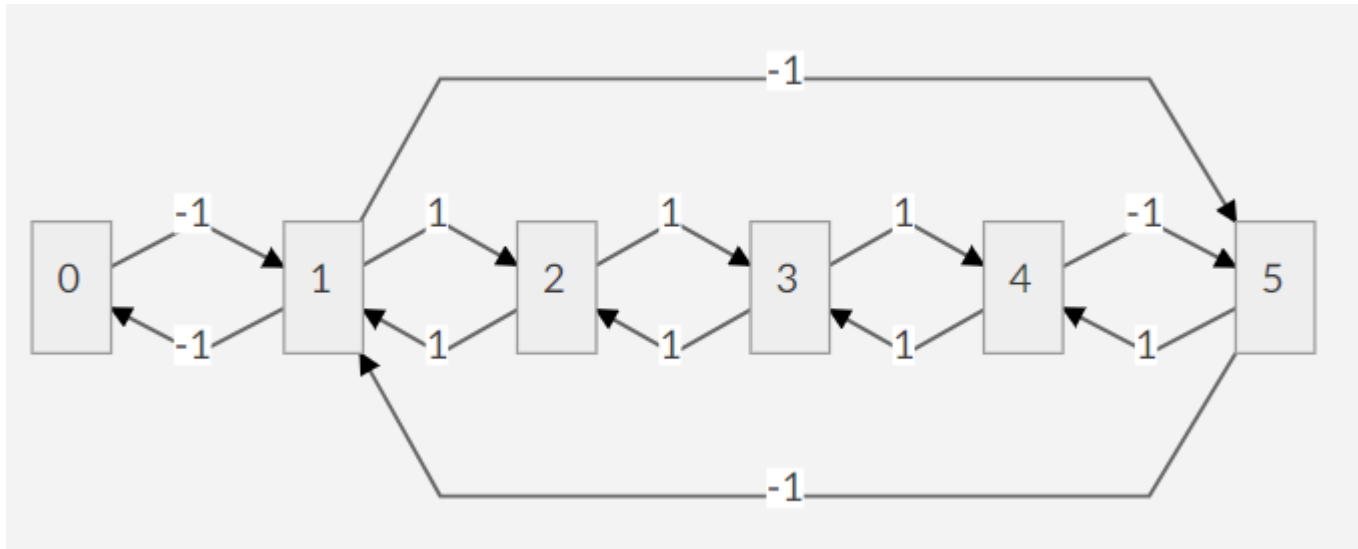
    add_edge(g, 1, 0, 1);
    add_edge(g, 2, 1, 1);
    add_edge(g, 3, 2, 1);
    add_edge(g, 4, 3, 1);
    add_edge(g, 5, 4, 1);
    add_edge(g, 1, 5, 1);
    return g;
}
```

Appendix - 테스트케이스 3



```
Graph* test_case_3(){
    Graph* g = init_graph(6);
    printf("Test Case #3 ===== \n");
    add_edge(g, 0, 1, 10);
    add_edge(g, 0, 2, 5);
    add_edge(g, 1, 2, 2);
    add_edge(g, 1, 3, 1);
    add_edge(g, 2, 1, 3);
    add_edge(g, 2, 3, 9);
    add_edge(g, 2, 4, 2);
    add_edge(g, 3, 4, 4);
    add_edge(g, 4, 0, 7);
    add_edge(g, 4, 3, 6);
    add_edge(g, 0, 3, 1);
    add_edge(g, 3, 4, -2);
    return g;
}
```

Appendix - 테스트케이스 4



```
Graph* test_case_4(){
    Graph* g = init_graph(6);
    printf("Test Case #4 ===== \n");
    add_edge(g, 0, 1, -1);
    add_edge(g, 1, 2, 1);
    add_edge(g, 2, 3, 1);
    add_edge(g, 3, 4, 1);
    add_edge(g, 4, 5, -1);
    add_edge(g, 5, 1, -1);

    add_edge(g, 1, 0, -1);
    add_edge(g, 2, 1, 1);
    add_edge(g, 3, 2, 1);
    add_edge(g, 4, 3, 1);
    add_edge(g, 5, 4, 1);
    add_edge(g, 1, 5, -1);
    return g;
}
```