#include <stdio.h>

#include <stdlib.h>

#define \_CRT\_SECURE\_NO\_WARNINGS

#pragma warning(disable:4996)

#define TRUE 1

#define FALSE 0

#define MAX\_VERTICES 100

#define INF 1000L

// 그래프를 표현하는 구조체

typedef struct GraphType {

int n; // 정점의 개수

int weight[MAX\_VERTICES][MAX\_VERTICES]; // 가중치 인접 행렬

} GraphType;

int selected[MAX\_VERTICES]; // 정점이 선택되었는지 여부를 저장하는 배열

int distance[MAX\_VERTICES]; // 선택된 정점과의 거리

// 최소 dist[v] 값을 갖는 정점을 반환하는 함수

int get\_min\_vertex(int n) {

int v, i;

for (i = 0; i < n; i++) // 선택되지 않은 정점 중에서

if (!selected[i]) {

v = i;

break;

}

for (; i < n; i++) // 가장 작은 distance 값을 가진 정점을 찾음

if (!selected[i] && (distance[i] < distance[v])) v = i;

return v;

}

// Prim의 MST 알고리즘을 구현한 함수

void prim(GraphType\* g, int s) {

int i, u, v;

for (u = 0; u < g->n; u++)

distance[u] = INF; // 모든 거리 값을 무한대로 초기화

distance[s] = 0; // 시작 정점의 거리를 0으로 설정

for (i = 0; i < g->n; i++) {

u = get\_min\_vertex(g->n); // 최소 거리 정점을 선택

selected[u] = TRUE; // 선택된 정점을 표시

if (distance[u] == INF) return; // 모든 정점을 다 선택했으면 종료

printf("정점 %d 추가\n", u); // 선택된 정점 출력

for (v = 0; v < g->n; v++) // 인접한 정점들의 거리 갱신

if (g->weight[u][v] != INF && !selected[v] && g->weight[u][v] < distance[v])

distance[v] = g->weight[u][v];

}

}

int main(void) {

GraphType g = { 7,

{{ 0, 29, INF, INF, INF, 10, INF },

{ 29, 0, 16, INF, INF, INF, 15 },

{ INF, 16, 0, 12, INF, INF, INF },

{ INF, INF, 12, 0, 22, INF, 18 },

{ INF, INF, INF, 22, 0, 27, 25 },

{ 10, INF, INF, INF, 27, 0, INF },

{ INF, 15, INF, 18, 25, INF, 0 } }

};

prim(&g, 0); // 0번 정점을 시작 정점으로 Prim 알고리즘 수행

return 0;

}

2, 다익스트라

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

#define \_CRT\_SECURE\_NO\_WARNINGS

#define TRUE 1

#define FALSE 0

#define MAX\_VERTICES 100

#define INF 1000000 /\* 무한대 (연결이 없는 경우) \*/

// 그래프를 표현하는 구조체

typedef struct GraphType {

int n; // 정점의 개수

int weight[MAX\_VERTICES][MAX\_VERTICES]; // 가중치 인접 행렬

} GraphType;

int distance[MAX\_VERTICES]; // 시작 정점으로부터의 최단 경로 거리

int found[MAX\_VERTICES]; // 방문한 정점을 표시

// 방문하지 않은 정점 중에서 최소 거리를 가진 정점을 선택하는 함수

int choose(int distance[], int n, int found[]) {

int i, min, minpos;

min = INT\_MAX;

minpos = -1;

for (i = 0; i < n; i++)

if (distance[i] < min && !found[i]) {

min = distance[i];

minpos = i;

}

return minpos;

}

// 현재 상태를 출력하는 함수

void print\_status(GraphType\* g) {

static int step = 1;

printf("STEP %d: ", step++);

printf("distance: ");

for (int i = 0; i < g->n; i++) {

if (distance[i] == INF)

printf(" \* ");

else

printf("%2d ", distance[i]);

}

printf("\n");

printf(" found: ");

for (int i = 0; i < g->n; i++)

printf("%2d ", found[i]);

printf("\n\n");

}

// Dijkstra 알고리즘을 구현한 함수

void shortest\_path(GraphType\* g, int start) {

int i, u, w;

for (i = 0; i < g->n; i++) { // 초기화

distance[i] = g->weight[start][i];

found[i] = FALSE;

}

found[start] = TRUE; // 시작 정점 방문 표시

distance[start] = 0;

for (i = 0; i < g->n - 1; i++) {

print\_status(g);

u = choose(distance, g->n, found); // 최소 거리 정점을 선택

found[u] = TRUE;

for (w = 0; w < g->n; w++) // 인접한 정점들의 거리 갱신

if (!found[w] && distance[u] + g->weight[u][w] < distance[w])

distance[w] = distance[u] + g->weight[u][w];

}

}

int main(void) {

GraphType g = { 7,

{{ 0, 7, INF, INF, 3, 10, INF },

{ 7, 0, 4, 10, 2, 6, INF },

{ INF, 4, 0, 2, INF, INF, INF },

{ INF, 10, 2, 0, 11, 9, 4 },

{ 3, 2, INF, 11, 0, INF, 5 },

{ 10, 6, INF, 9, INF, 0, INF },

{ INF, INF, INF, 4, 5, INF, 0 } }

};

shortest\_path(&g, 0); // 0번 정점을 시작 정점으로 Dijkstra 알고리즘 수행

return 0;

}

Floyd-Warshall 최단 경로 알고리즘

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

#define \_CRT\_SECURE\_NO\_WARNINGS

#define TRUE 1

#define FALSE 0

#define MAX\_VERTICES 100

#define INF 1000000 /\* 무한대 (연결이 없는 경우) \*/

// 그래프를 표현하는 구조체

typedef struct GraphType {

int n; // 정점의 개수

int weight[MAX\_VERTICES][MAX\_VERTICES]; // 가중치 인접 행렬

} GraphType;

int A[MAX\_VERTICES][MAX\_VERTICES]; // 최단 경로 길이를 저장하는 행렬

// 최단 경로 행렬을 출력하는 함수

void printA(GraphType\* g) {

int i, j;

printf("===============================\n");

for (i = 0; i < g->n; i++) {

for (j = 0; j < g->n; j++) {

if (A[i][j] == INF) printf(" \* ");

else printf("%3d ", A[i][j]);

}

printf("\n");

}

printf("===============================\n");

}

// Floyd-Warshall 알고리즘을 구현한 함수

void floyd(GraphType\* g) {

int i, j, k;

for (i = 0; i < g->n; i++)

for (j = 0; j < g->n; j++)

A[i][j] = g->weight[i][j]; // 초기화

printA(g);

for (k = 0; k < g->n; k++) { // 모든 정점에 대해

for (i = 0; i < g->n; i++)

for (j = 0; j < g->n; j++)

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

A[i][j] = A[i][k] + A[k][j]; // 경유지를 거친 최단 경로 갱신

printA(g);

}

}

int main(void) {

GraphType g = { 7,

{{ 0, 7, INF, INF, 3, 10, INF },

{ 7, 0, 4, 10, 2, 6, INF },

{ INF, 4, 0, 2, INF, INF, INF },

{ INF, 10, 2, 0, 11, 9, 4 },

{ 3, 2, INF, 11, 0, INF, 5 },

{ 10, 6, INF, 9, INF, 0, INF },

{ INF, INF, INF, 4, 5, INF, 0 } }

};

floyd(&g); // Floyd-Warshall 알고리즘 수행

return 0;

}

4. 위상 정렬

#include <stdio.h>

#include <stdlib.h>

#define \_CRT\_SECURE\_NO\_WARNINGS

#pragma warning(disable:4996)

#define TRUE 1

#define FALSE 0

#define MAX\_VERTICES 50

// 인접 리스트의 노드 구조체

typedef struct GraphNode {

int vertex;

struct GraphNode \*link;

} GraphNode;

// 그래프를 표현하는 구조체

typedef struct GraphType {

int n; // 정점의 개수

GraphNode \*adj\_list[MAX\_VERTICES]; // 인접 리스트

} GraphType;

// 그래프 초기화 함수

void graph\_init(GraphType \*g) {

int v;

g->n = 0;

for (v = 0; v < MAX\_VERTICES; v++)

g->adj\_list[v] = NULL;

}

// 정점 삽입 함수

void insert\_vertex(GraphType \*g, int v) {

if (((g->n) + 1) > MAX\_VERTICES) {

fprintf(stderr, "그래프: 정점의 개수 초과");

return;

}

g->n++;

}

// 간선 삽입 함수, v를 u의 인접 리스트에 삽입

void insert\_edge(GraphType \*g, int u, int v) {

GraphNode \*node;

if (u >= g->n || v >= g->n) {

fprintf(stderr, "그래프: 정점 번호 오류");

return;

}

node = (GraphNode \*)malloc(sizeof(GraphNode));

node->vertex = v;

node->link = g->adj\_list[u];

g->adj\_list[u] = node;

}

#define MAX\_STACK\_SIZE 100

typedef int element;

typedef struct {

element stack[MAX\_STACK\_SIZE];

int top;

} StackType;

// 스택 초기화 함수

void init(StackType \*s) {

s->top = -1;

}

// 공백 상태 검출 함수

int is\_empty(StackType \*s) {

return (s->top == -1);

}

// 포화 상태 검출 함수

int is\_full(StackType \*s) {

return (s->top == (MAX\_STACK\_SIZE - 1));

}

// 삽입 함수

void push(StackType \*s, element item) {

if (is\_full(s)) {

fprintf(stderr, "스택 포화 에러\n");

return;

}

else s->stack[++(s->top)] = item;

}

// 삭제 함수

element pop(StackType \*s) {

if (is\_empty(s)) {

fprintf(stderr, "스택 공백 에러\n");

exit(1);

}

else return s->stack[(s->top)--];

}

// 위상 정렬을 수행하는 함수

int topo\_sort(GraphType \*g) {

int i;

StackType s;

GraphNode \*node;

// 모든 정점의 진입 차수를 계산

int \*in\_degree = (int \*)malloc(g->n \* sizeof(int));

for (i = 0; i < g->n; i++) // 초기화

in\_degree[i] = 0;

for (i = 0; i < g->n; i++) {

GraphNode \*node = g->adj\_list[i]; // 정점 i에서 나오는 간선들

while (node != NULL) {

in\_degree[node->vertex]++;

node = node->link;

}

}

// 진입 차수가 0인 정점을 스택에 삽입

init(&s);

for (i = 0; i < g->n; i++) {

if (in\_degree[i] == 0) push(&s, i);

}

// 위상 순서를 생성

while (!is\_empty(&s)) {

int w;

w = pop(&s);

printf("정점 %d -> ", w); // 정점 출력

node = g->adj\_list[w]; // 각 정점의 진입 차수를 변경

while (node != NULL) {

int u = node->vertex;

in\_degree[u]--; // 진입 차수를 감소

if (in\_degree[u] == 0) push(&s, u);

node = node->link; // 다음 정점

}

}

free(in\_degree);

printf("\n");

return (i == g->n); // 반환값이 1이면 성공, 0이면 실패

}

int main(void) {

GraphType g;

graph\_init(&g);

insert\_vertex(&g, 0);

insert\_vertex(&g, 1);

insert\_vertex(&g, 2);

insert\_vertex(&g, 3);

insert\_vertex(&g, 4);

insert\_vertex(&g, 5);

// 정점 0의 인접 리스트 생성

insert\_edge(&g, 0, 2);

insert\_edge(&g, 0, 3);

// 정점 1의 인접 리스트 생성

insert\_edge(&g, 1, 3);

insert\_edge(&g, 1, 4);

// 정점 2의 인접 리스트 생성

insert\_edge(&g, 2, 3);

insert\_edge(&g, 2, 5);

// 정점 3의 인접 리스트 생성

insert\_edge(&g, 3, 5);

// 정점 4의 인접 리스트 생성

insert\_edge(&g, 4, 5);

// 위상 정렬 수행

topo\_sort(&g);

// 동적 메모리 반환 코드 생략

return 0;

}