자료구조 (Data Structure)
10주차: 그래프의 최소 길이 트리

지난 시간 내용

- 최소길이트리: 연결선 합계가 가장 작은 트리 (MST)
- PRIM: 트리에서 나가는 연결선 중 작은 것부터 추가
- KRUSKAL: 전체 연결선 중 작은 것부터 트리 합치기

오늘의 목차

- PRIM 구현하기
- KRUSKAL 구현하기
- 그래프 출력하기

구조체 - 인접리스트

- 링크드 리스트 형태
  - 데이터: 연결선의 주소
  - 링크: 다음 인접리스트 노드의 주소

구조체 - 인접리스트

```
struct edge;

struct elist {
        struct edge *edge;
        struct elist *next;
};
```

구조체 - 노드

- 그래프 노드
  - 데이터: 문자
  - 인덱스: 그래프의 노드 리스트에서 몇번째
  - 상태: MST에 들어갔는지 여부
  - 키: MST와의 최단 연결선 길이
  - 연결선: MST에 들어가게한 연결선
  - 이웃들: 연결선 정보

구조체 - 노드

```
struct node {
        char data;
        int index;
        int status;                 /* 0: MST에 없음, 1: 있음 */
        int key;                    /* MST와 최단 연결 길이 */
        struct edge* tree_edge;          /* MST 연결선 */
        struct elist *adj;
};
```

구조체 - 연결선

- 연결선
  - 노드1: 연결선의 한쪽 노드
  - 노드2: 연결선의 다른쪽 노드
  - 모양: 화면에 표시할 모양. 선의 방향.
  - 길이: 최적화 대상
  - 종류: 0 : 확인 전
    1 : MST에서 그 노드까지 최소 길이
    2 : 최소 길이 아님

구조체 - 연결선

```
struct edge {
        struct node *node1;
        struct node *node2;
        char shape;                 /* '-', '/', '|' */
        int length;                 /* 연결선의 길이 */
        int type;                   /*  0: 체크 전
                                        1: MST 후보
                                        2: MST 아님 */
};
```

구조체 - 그래프

- 그래프
  - V: 노드 배열의 주소
  - n_V: 노드 개수
  - E: 연결선 배열의 주소
  - E_V: 연결선 개수
  - adj: 노드 인덱스 두개에 대한 연결선 이차원 배열

구조체 - 그래프

```
struct graph {
        struct node *vertexes;
        int n_vertexes;
        struct edge *edges;
        int n_edges;
        struct edge ***adj;
};
```

PRIM

- 시작:
  - 어떤 한 노드로 부터 시작

- 반복되는 내용:
  - MST에 없는 노드들 중,
    MST와 제일 가까운 노드가 MST에 추가됨
  - 추가된 노드의 이웃들 거리 업데이트

PRIM - 자료구조

- 우선순위 큐:

  - 노드들 저장

  - MST와의 거리 기준

  - 최소 거리 노드가 꺼내짐

PRIM - 자료구조

```
/* 우선순위 큐 구조체 */
struct pqueue {
        struct node *data[100];
        int capacity;
        int size;
};
```

PRIM - 자료구조

- 우선순위 큐에 전체 데이터 추가:

  - 데이터 모두 복사한다

  - size를 데이터 개수만큼 증가시킨다

PRIM - 자료구조

```
void init_pqueue(struct pqueue *pq, struct node *data,
        int n)
{
        for (int i = 0; i < n; i++)
                pq->data[i] = data[i];
        pq->size = n;
}
```

PRIM - 자료구조

- 우선순위 큐에서 데이터 꺼내기:

  - key가 최소인 데이터를 찾는다

  - 그 데이터를 삭제하고, 반환한다

PRIM - 자료구조

```
struct node *dequeue(struct pqueue *pq)
{
        /* 최소 키 노드 찾기 */
        int m = 0;
        for (int i = 1; i < pq->size; i++) {
                if (pq->data[i]->key < pq->data[m]->key)
                        m = i;
        }
        struct node *min_node = pq->data[m];
        min_node->pq_index = -1;
```

PRIM - 자료구조

```
        /* 노드 삭제하기 */
        for (int i = m; i < pq->size - 1; i++) {
                pq->data[i] = pq->data[i + 1];
                pq->data[i]->pq_index = i;
        }
        pq ->size -= 1;

        /* 노드 반환하기 */
        return min_node;
}
```

PRIM - 자료구조

- 우선순위 큐에서 데이터 키를 더 작게하기:

  - 아무 일도 하지 않는다

  - (힙으로 구현할 때 힙 속성을 회복시킨다)

PRIM - 자료구조

```
void decrease_key(struct pqueue *pq)
{


}
```

PRIM 구현

- 초기값:
  - 연결선 타입 = 0,
  - 노드 상태 = 0, 키 = ∞, mst선 = 없음, pq_i = i
  - 시작점 키 = 0
  - 모든 노드 pq에 추가

- 반복되는 내용:
  - pq에서 노드 꺼내기
  - 노드를 MST에 추가
  - 노드의 이웃들 pq에 추가 또는 업데이트

PRIM - 구현

```
void prim(struct graph *g, struct node *start_node)
{
        /* 연결선 초기화 */
        for (int i = 0; i < g->n_edges; i++) {
                struct edge *e = g->edges + i;
                e->type = 0;
        }
```

PRIM - 구현

```
/* 노드 초기화 */
for (int i = 0; i < g->n_vertexes; i++) {
        struct node *n = g->vertexes + i;
        n->status = 0;
        n->key = 9;
        n->tree_edge = NULL;
        n->pq_index = i;
}
```

PRIM - 구현

```
/* 시작점 초기화 */
start_node->key = 0;

/* 우선순위 큐 만들고, 노드들 추가 */
struct pqueue _pq = {.size = 0, .capacity = 5};
struct pqueue *pq = &_pq;
init_pqueue(pq, v->vertexes, v->n_vertexes);
```

PRIM - 구현

```
/* mst에 가장 가까운 노드 u에서 반복 */
while (pq->size > 0) {

        struct node *u = dequeue(pq);
        if (u->status != 0) continue;

        u->status = 1;
```

PRIM - 구현

```
                if (v->key <= e->length) {
                        e->type = 2;
                } else {
                        v->key = e->length;
                        e->type = 1;
                        if (v->tree_edge != NULL)
                                v->tree_edge->type = 2;
                        v->tree_edge = e;
                        decrease_key(pq, v);
                }
        }
}
```

KRUSKAL

- 시작:
  - 연결선을 정렬한다
- 반복되는 내용:
  - 연결선의 두 노드의 소속 트리를 확인한다
  - 소속 트리가 다르면 합친다

KRUSKAL - 그래프 노드

- 그래프 노드
  - 데이터: 문자
  - 인덱스: 그래프의 노드 리스트에서 몇번째
  - 규모: 같은 규모 크기를 흡수하면 커짐
  - 루트: MST의 루트

KRUSKAL - 그래프 노드

```
struct node {
        char data;
        int index;
        int rank;                   /* 서브 MST 규모 */
        struct node* root;          /* 소속된 MST의 루트 */
};
```

KRUSKAL - 소속 확인

- 소속 MST의 루트인 경우:
  - 자신의 주소를 반환

- 루트가 아닌 경우:
  - 저장된 루트들을 따라가서 최종 루트를 확인
  - 최종 루트를 저장

KRUSKAL - 소속 확인

```
struct node *find_root(struct node* n)
{
        if (n->root != n) {
                n->root = find_root(n->root);
        }
        return n->root;
}
```

KRUSKAL - 합치기

- MST 두개의 루트 x, y에 대해,
- 더 큰 규모의 MST가 다른 MST를 흡수한다.
- 규모가 같을 경우 흡수한 MST의 rank가 증가한다.

KRUSKAL - 합치기

```
void union_mst(struct node* x, struct node *y)
{
        if (x->rank < y->rank) {
                x->root = y;
        } else {
                y->root = x;
                if (x->rank == y->rank)
                        x->rank += 1;
        }
}
```

KRUSKAL - 연결선 정렬

- 개수가 작아서 insertion sort 사용

- 정렬되지 않은 구간의 가장 왼쪽 데이터를

  정렬된 구간의 적절한 자리로 이동시킨다

KRUSKAL - 연결선 정렬

```
void sort_edges(struct edge **edges, int n)
{
        for (int i = 0; i < n; i++) {
                int j = i, d = edges[j]->distance;
                while (j > 0 && edges[j - 1]->distance > d) {
                        edges[j] = edges[j - 1];
                        j -= 1;
                }
                sorted[j] = edges[i];
        }
}
```

KRUSKAL 구현

- 초기값:
  - 연결선 타입 = 0, 오름차순 정렬
  - 노드 규모 = 0, 루트 = 자신

- 반복되는 내용:
  - 연결선의 두 노드가 소속이 같으면,
    연결선은 MST가 아님
  - 소속이 다르면
    연결선은 MST

KRUSKAL - 구현

```
void kruskal(struct graph *g)
{
        /* 연결선 초기화 및 정렬*/
        struct edge *edges[100];
        for (int i = 0; i < g->n_edges; i++) {
                edges[i] = g->edges + i;
                edges[i]->type = 0;
        }
        sort_edges(edges, g->n_edges);
```

KRUSKAL - 구현

```
void kruskal(struct graph *g)
{
        /* 노드 초기화*/
        for (int i = 0; i < g->n_vertexes; i++) {
                struct node *n = g->vertexes + i;
                n->rank = 0;
                n->root = n;
        }
```

KRUSKAL - 구현

```
        /* 반복: 최소 길이 연결선으로 트리 합치기*/
        for (int i = 0; i < g->n_edges; i++) {
                struct node *e = edges[i];
                struct node *x = find_root(e->node1);
                struct node *y = find_root(e->node2);
                if (x == y) { e->type = 2; }
                else {
                        e->type = 1;
                        union_mst(x, y);
                }
        }
}
```

그래프 출력 - printf

```
#include <stdio.h>
int main()
{
        printf("Hello, minimum spanning tree!\n");
        return 0;
}
```

그래프 출력 - 노드 출력 함수

data가 A인 노드를 출력할 때:

- 상태가 0인 경우

  A

- 상태가 1인 경우

  Ao

그래프 출력 - 노드 출력 함수

```
void print_node(struct node *v)
{
        char str[5] = {' ', v->data, ' ', ' ', '\0'};
        if (v->status != 0)
                str[2] = 'o';
        printf("%s", str);
}
```

그래프 출력 - 노드 출력 함수

```
int main()
{
        struct node v = {'A'};
        print_node(&v);
        v.status = 1;
        print_node(&v);
        return 0;
}
```

그래프 출력 - 연결선 출력 함수

모양 -, 길이 1인 연결선을 출력할 때:

- 상태가 0인 경우

  -1

- 상태가 1인 경우

  =1

- 상태가 2인 경우

  .

그래프 출력 - 연결선 출력 함수

```
void print_edge(struct edge *e)
{
        char str[5] = {' ', e->shape, '0' + e->length, ' ', '\0'};
        if (e == NULL) {
                str[1] = str[2] = ' ';
        } else {
                if (e->type == 2) {
                        str[1] = ':';
                        str[2] = str[3] = ' ';
                }
```

그래프 출력 - 연결선 출력 함수

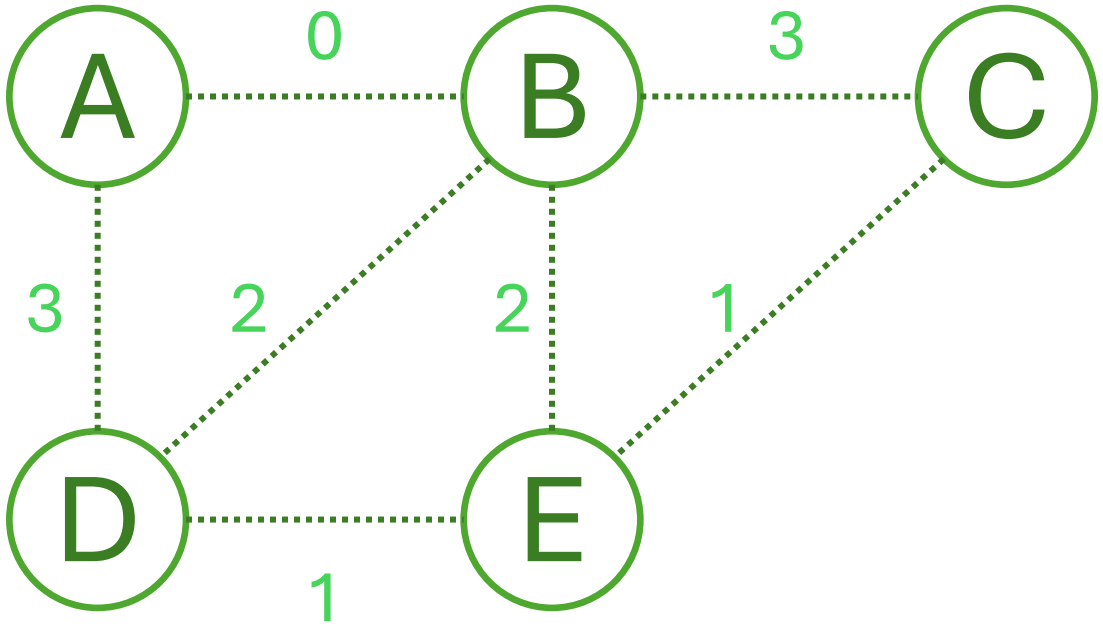
```
            if (e->type == 1) {
                    if (e->shape == '-') {
                            str[1] = '=';
                    } else {
                            str[2] = e->shape;
                            str[3] = '0' + e->length;
                    }
            }
    }
    printf("%s", str);
}
```

그래프 출력 - 연결선 출력 함수

```
int main()
{
        struct edge e = {NULL, NULL, '-', 1};
        print_edge(&e);
        e.type = 1;
        print_edge(&e);
        e.type = 2;
        print_edge(&e);
        return 0;
}
```

그래프 출력 - 그래프 내용

- 모든 노드에 대해 print_node() 호출
- 모든 연결선에 대해 print_edge() 호출



그래프 출력 - 그래프 내용 출력 함수

```
void print_graph(struct graph *g)
{
        /* 노드 출력 */
        for (int i = 0; i < g->n_vertexes; i++) {
                print_node(g->vertexes + i);
                printf(", ");
        }
        printf("\n");
```

그래프 출력 - 그래프 내용 출력 함수

```
    /* 연결선 출력 */
    for (int i = 0; i < g->n_edges; i++) {
        printf("[");
        print_node(g->edges[i].node1);
        print_edge(g->edges + i);
        print_node(g->edges[i].node2);
        printf("], ");
    }
    printf("\n");
}
```

그래프 출력 - 그래프 내용 출력 함수

```
int main()
{
        struct graph g;
        struct node n[5] = {
                {'A', 0}, {'B', 1}, {'C', 2}, {'D', 3}, {'E', 4}
        };
        g.vertexes = n;
        g.n_vertexes = 5;
        g.n_edges = 7;
```

그래프 출력 - 그래프 내용 출력 함수

```
    struct edge edges[] = {
            {n + 0, n + 1, '-', 0}, {n + 1, n + 2, '-', 3},
            {n + 0, n + 3, '|', 3}, {n + 1, n + 3, '/', 2},
            {n + 1, n + 4, '|', 2}, {n + 2, n + 4, '/', 1}
            {n + 3, n + 4, '-', 1}
    };
    g.edges = edges;
    print_graph(&g);
    return 0;
}
```

그래프 출력 - 인접 행렬

- 인접 행렬

  - 두 노드에 대한 연결선 주소를 저장한 행렬

  - 전체 연결선 리스트로부터 만들 수 있음

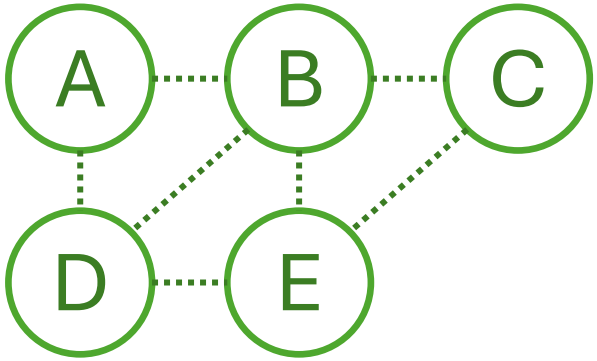
그래프 출력 - 인접 행렬

```
void make_adj_matrix(struct graph *g)
{
        for (int i = 0; i < g->n_edges; i++) {
                struct edge *e = g->edges + i;
                int n1 = e->node1->index;
                int n2 = e->node2->index;
                g->adj[n1][n2] = g->adj[n2][n1] = e;
        }
}
```

그래프 출력 - 인접 행렬 출력 함수

[A - B], [B - C], [A | D], [B / D], [B | E], [C / E], [D - E] 에서

	A	B	C	D	E
A		-		|	
B	-		-	/	|
C		-			/
D	|	/			-
E		|	/	-	



그래프 출력 - 인접 행렬 출력 함수

```
void print_adj_matrix(struct graph *g)
{
        /* 첫번째 행: 노드 출력 */
        printf("  ");
        for (int i = 0; i < g->n_vertexes; i++) {
                printf("%c ", g->vertexes[i].data);
        }
        printf("\n");
        for (int i = 0; i < g->n_vertexes; i++) {
                /* 첫번째 열: 노드 출력 */
```

그래프 출력 - 인접 행렬 출력 함수

```
            printf("%c ", g->vertexes[i].data);
            for (int j = 0; j < g->n_vertexes; j++) {
                if (g->adj[i][j] == NULL)
                    printf("  ");
                else
                    printf("%c ", g->adj[i][j]->shape);
            }
            printf("\n");
        }
}
```

그래프 출력 - 인접 행렬 출력 함수

```
int main() {
        /* 이전 내용에 이어서 */
        struct edge *mat[6][6] = { 0 };
        struct edge **rows[6] = {mat[0], mat[1], mat[2],
                mat[3], mat[4], mat[5]};
        g.adj = rows;
        make_adj_matrix(&g);
        print_adj_matrix(&g);
        return 0;
}
```

그래프 출력 - 인접 리스트

- 인접 리스트

  - 인접 리스트는 각 노드의 연결선들 정보

- 링크드 리스트 추가 방법

  - 제일 앞에 추가

  - 순서 유지를 위해 역순으로 추가한다

그래프 출력 - 인접 리스트

```
struct elist *new_elist(struct edge *e, struct elist *n)
{
        struct elist *el = malloc(sizeof(struct elist));
        if (el == NULL) return NULL;
        el->edge = e;
        el->next = n;
        return el;
}
```

그래프 출력 - 인접 리스트

```
void make_adj_list(struct graph *g)
{
        for (int i = g->n_edges - 1; i >= 0; i--) {
                struct edge *e = g->edges + i;
                struct node *u = e->node1, *v = e->node2;
                u->adj = new_elist(e, u->adj);
                v->adj = new_elist(e, v->adj);
        }
}
```

그래프 출력 - 인접 리스트

- 링크드 리스트 삭제 방법

  - 제일 앞의 노드를 지우고,

  - 다음 노드로 이동한다

그래프 출력 - 인접 리스트

```
void free_adj_list(struct graph *g)
{
        for (int i = 0; i < g->n_vertexes; i++) {
                while (g->vertexes[i].adj != NULL) {
                        struct elist *l = g->vertexes[i].adj;
                        g->vertexes[i].adj = l->next;
                        free(l);
                }
        }
}
```

그래프 출력 - 인접 리스트 출력 함수

[A - B], [B - C], [A | D], [B / D], [B | E], [C / E], [D - E] 에서

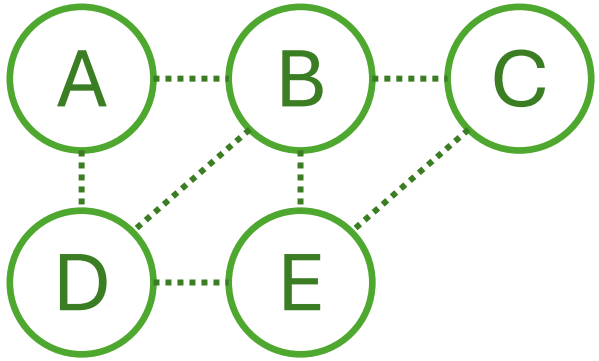
A: B D

B: A C D E

C: B E

D: A B E

E: B C D



그래프 출력 - 인접 리스트 출력 함수

```
void print_adj_list(struct graph *g)
{
        for (int i = 0; i < g->n_vertexes; i++) {
                struct node *n = g->vertexes + i;
                printf("%c: ", n->data);

                struct elist *e = n->adj;
                while (e != NULL) {
```

그래프 출력 - 인접 리스트 출력 함수

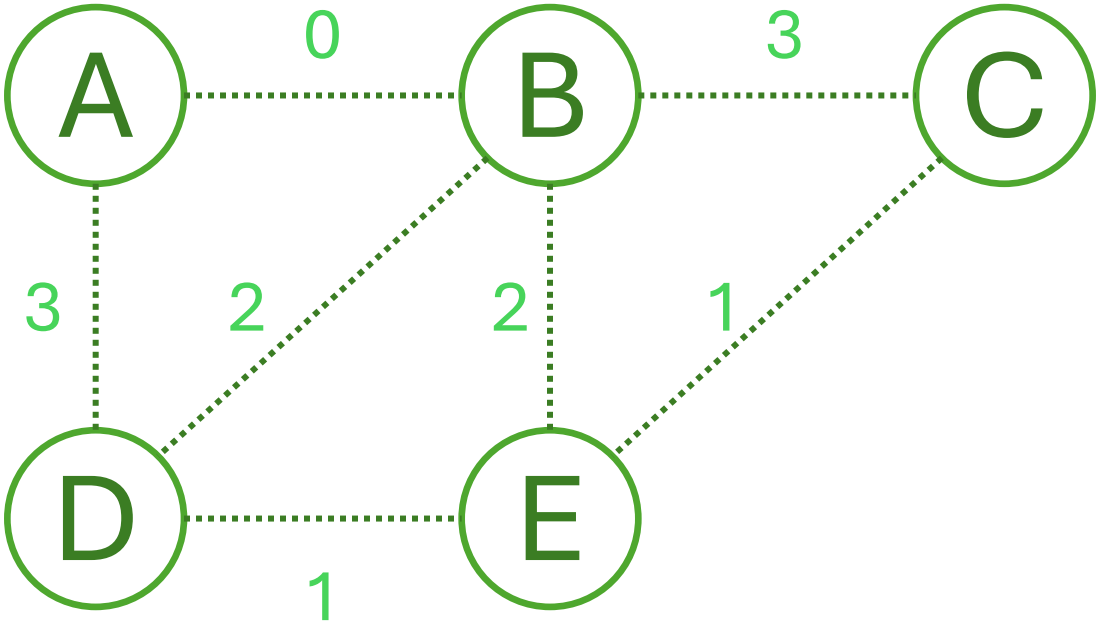
```
                struct node *near = e->edge->node1;
                if (near == n) {
                        near = e->edge->node2;
                }
                printf("%c ", near->data);
                e = e->next;
        }
        printf("\n");
    }
}
```

그래프 출력 - 인접 리스트 출력 함수

```
int main() {
        /* 이전 내용에 이어서 */
        make_adj_list(&g);
        print_adj_list(&g);

        free_adj_list(&g);
        return 0;
}
```

그래프 출력 - 3열 그래프



A -0 B -3 C

|3 /2 |2 /1

D -1 E

그래프 출력 - 3열 그래프

```
void print_graph_3col(struct graph *g)
{
        int n_vertexes = g->n_vertexes;
        /* 첫번째 행 */
        for (int i = 0; i < 3 && i < n_vertexes; i++) {
                print_node(g->vertexes + i);
                if (i + 1 < n_vertexes)
                        print_edge(g->adj[i][i + 1]);
        }
        printf("\n");
```

그래프 출력 - 3열 그래프

```
for (int row = 1; row <= n_vertexes / 3; row++) {
        int i;
        for (i = row * 3; i < row * 3 + 2
                        && i < n_vertexes; i++) {
                print_edge(g->adj[i][i - 3]);
                print_edge(g->adj[i][i - 2]);
        }
        if (i < n_vertexes)
                print_edge(g->adj[i][i - 3]);
        printf("\n");
```

그래프 출력 - 3열 그래프

```
            for (i = row * 3; i <= row * 3 + 2
                                && i < n_vertexes; i++) {
                    print_node(g->vertexes + i);
                    if (i + 1 < n_vertexes)
                            print_edge(g->adj[i][i + 1]);
            }
            printf("\n");
    }
}
```

그래프 출력 - 3열 그래프

```
int main() {
        /* 이전 내용에 이어서 */

        print_graph_3col(&g);

        free_adj_list(&g);
        return 0;
}
```

요약

- PRIM MST 구현
- KRUSKAL MST 구현