[CSE3081(2반)] 알고리즘 설계와 분석

2020학년도 2학기

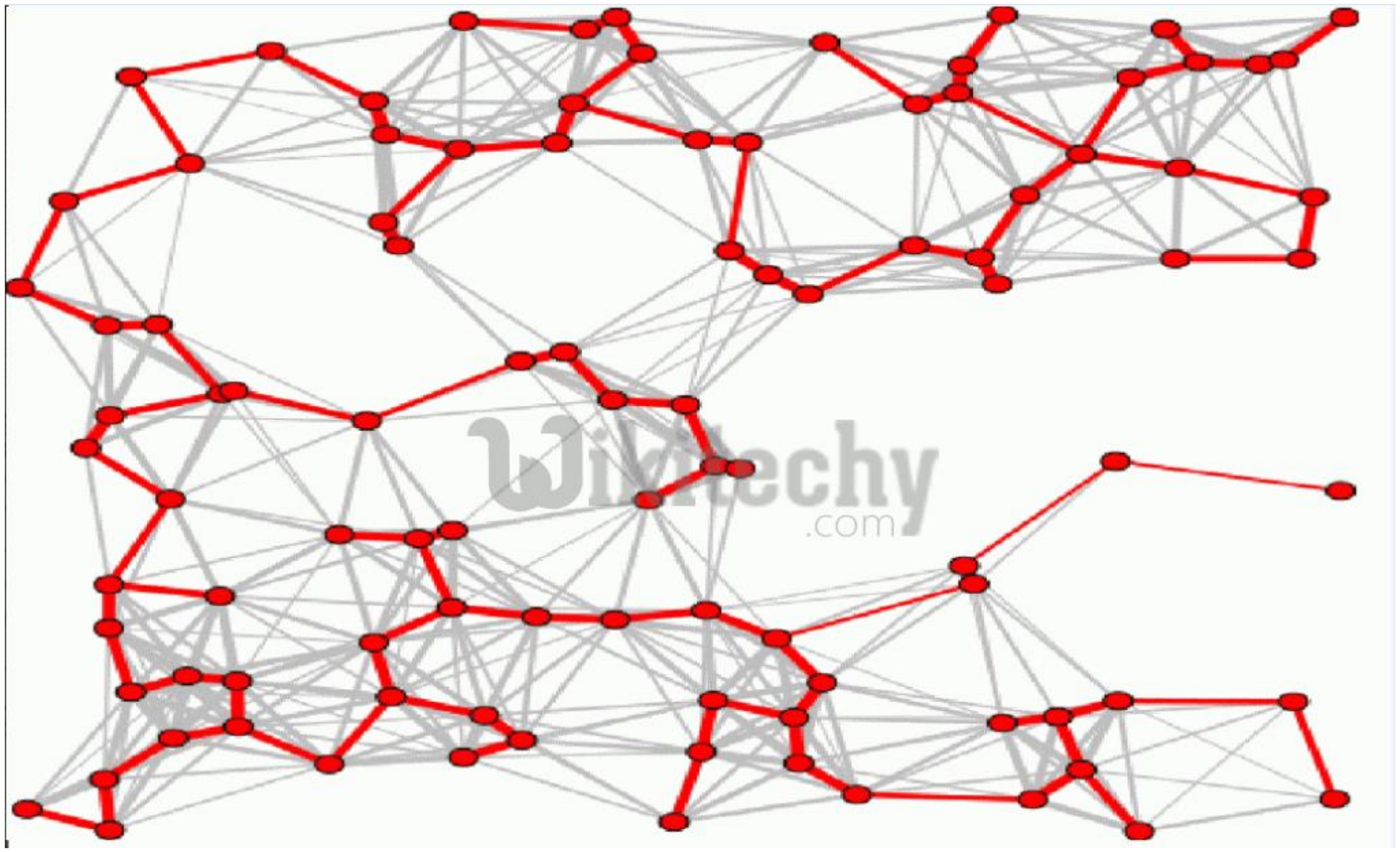
강의자료

(2020.12.01 화요일)

서강대학교 공과대학 컴퓨터공학과
임 인 성 교수

[주제 6]
Graph Algorithms

Minimum Spanning Tree



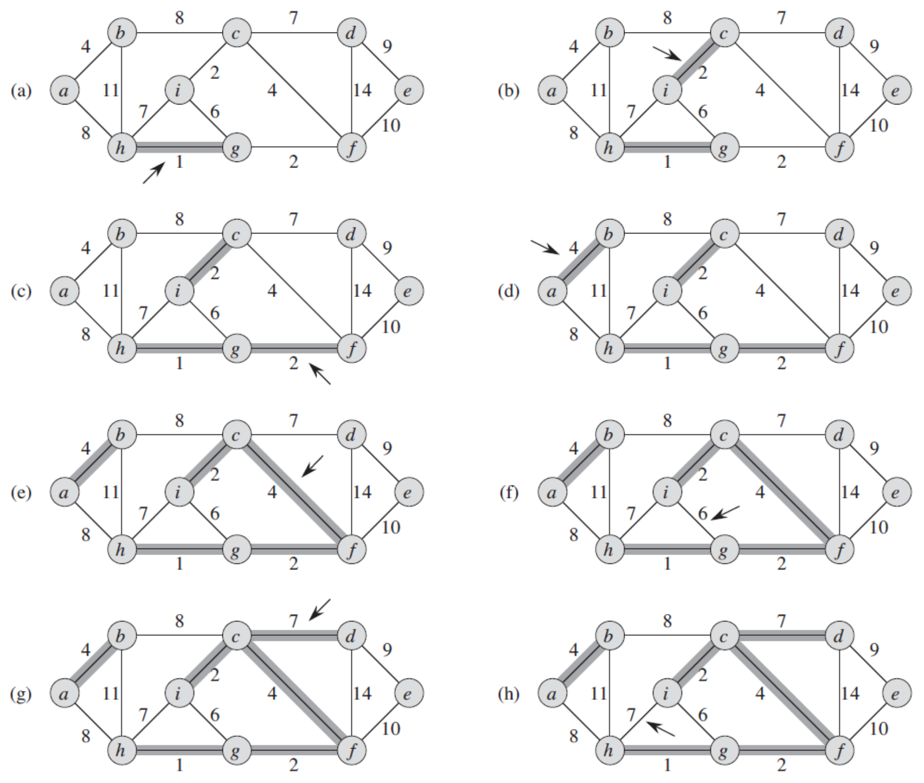
https://www.wikitechy.com/technology/applications-minimum-spanning-tree-problem/

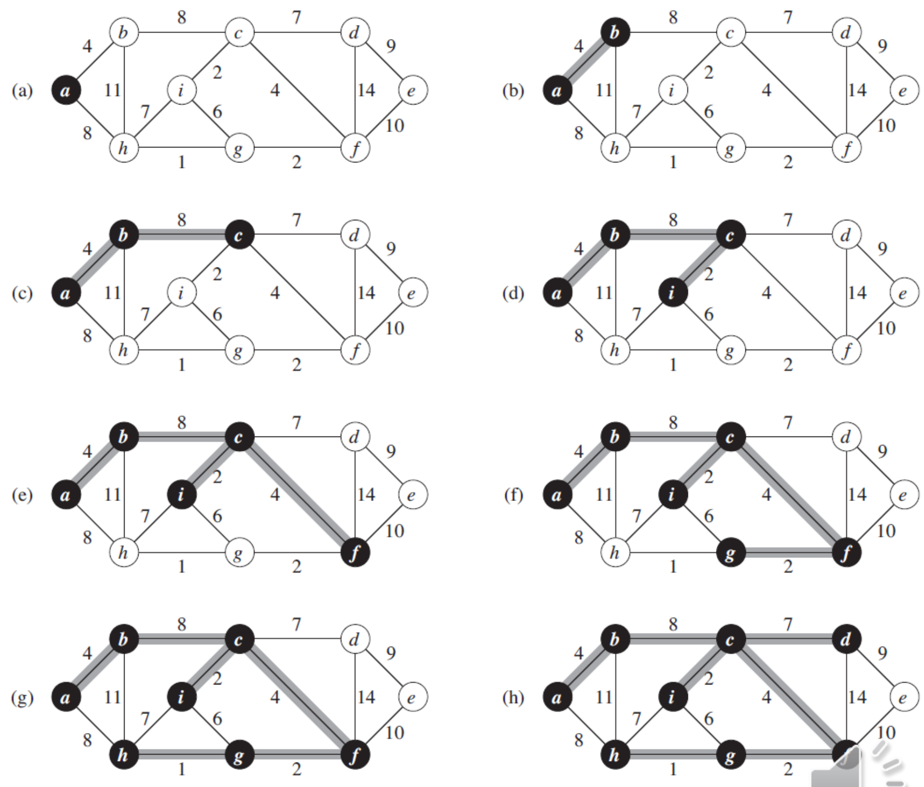
Kruskal's Algorithm vs Prim's Algorithm (Greedy!)

- **Kruskal's algorithm:** In each step, find and add **an edge of the least possible weight** that connects any two trees in the (current) forest.

- **Prim's algorithm:** In each step, find and add **an edge of the least possible weight** that connects the (current) tree to a non-tree vertex.

Courtesy of T. Cormen et al.

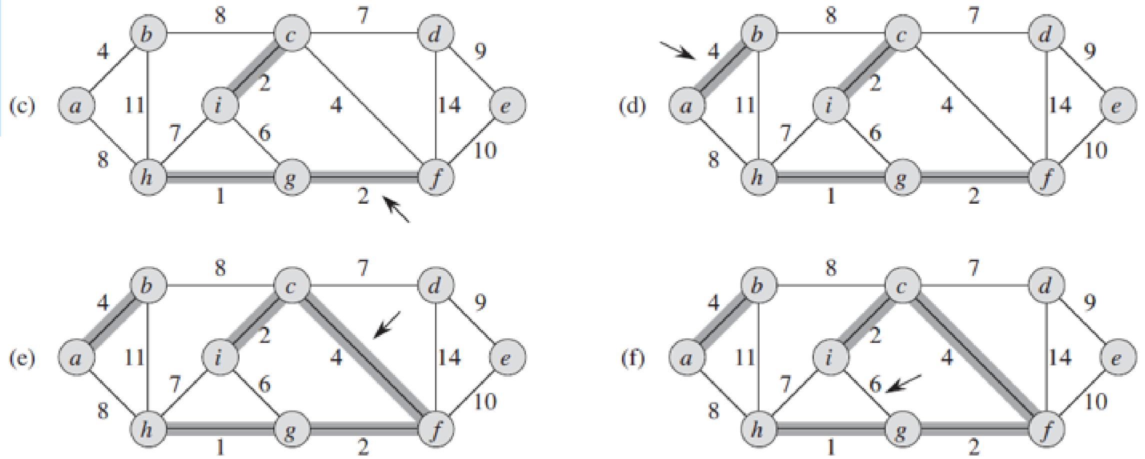




Selection of Next Edge: Kruskal's Algorithm

```
Generic-MST(G) {
    A := empty; // A: a set of edges of G
    While (A does not form a spanning tree) {
        Find and edge (u, v) that is safe for A;
        A := A ∪ { (u, v) };
    }
}
```

In each step, find and add **an edge of the least possible weight** that connects any two trees in the (current) forest.



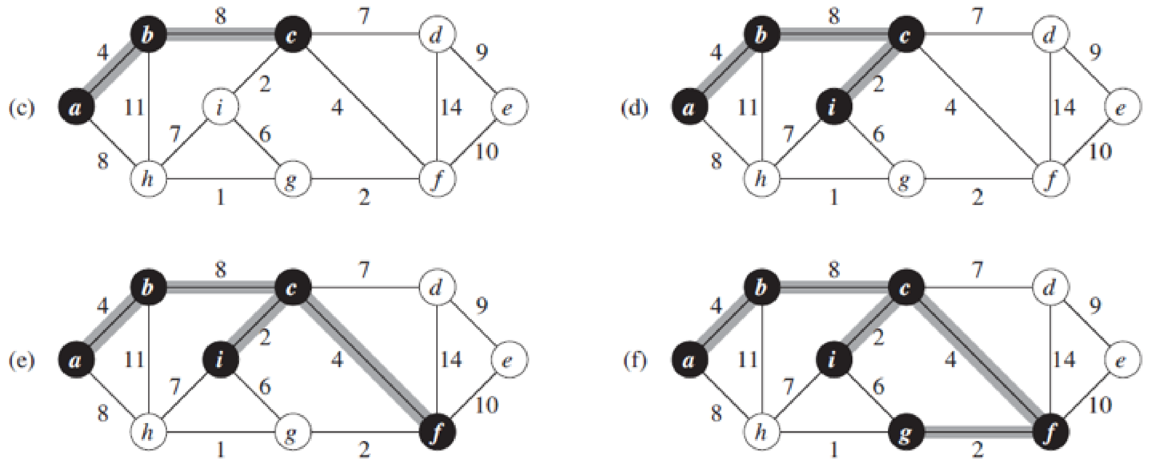
Theorem

Let G = (V, E) be a connected, undirected graph with a real-valued weight function w defined on E. Let **A** be a set of E that is included in some minimum spanning tree for G, let **(S, V-S)** be any cut of G that respects A, and let **(u, v)** be a light edge crossing (S, V-S). Then, **edge (u, v) is safe for A**.

Selection of Next Edge: Prim's Algorithm

```
Generic-MST(G) {
    A := empty; // A: a set of edges of G
    While (A does not form a spanning tree) {
        Find and edge (u, v) that is safe for A;
        A := A ∪ { (u, v) };
    }
}
```

In each step, find and add **an edge of the least possible weight** that connects the (current) tree to a non-tree vertex.



Theorem

Let G = (V, E) be a connected, undirected graph with a real-valued weight function w defined on E. Let **A** be a set of E that is included in some minimum spanning tree for G, let **(S, V-S)** be any cut of G that respects A, and let **(u, v)** be a light edge crossing (S, V-S). Then, **edge (u, v) is safe for A**.

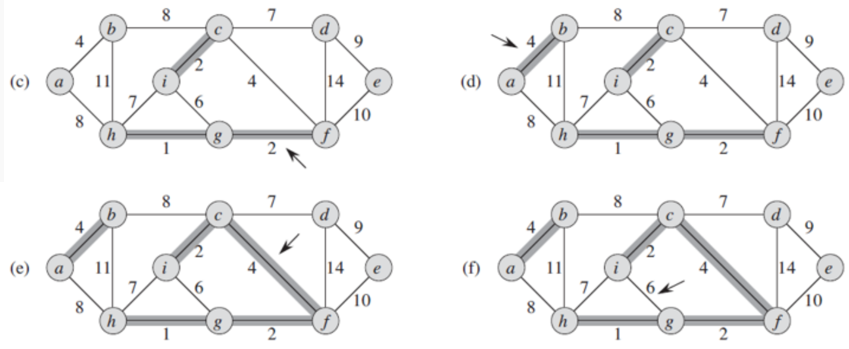
From Wikipedia

Kruskal's Minimum Spanning Tree Algorithm

- **Idea**
  - Finds an edge of the least possible weight that connects any two trees in the forest.
- **Implementation using disjoint-set data structure**

```
KRUSKAL(G):
1 A = ∅
2 foreach v ∈ G.V:
3     MAKE-SET(v)
4 foreach (u, v) in G.E ordered by weight(u, v), increasing:
5     if FIND-SET(u) ≠ FIND-SET(v):
6         A = A ∪ {(u, v)}
7         UNION(u, v)
8 return
```

- 매 단계 forest를 어떻게 관리할 것인가?
- 두 tree를 어떻게 병합할 것인가?
- 매 단계 (u, v)를 어떻게 선택할 것인가?



- **Complexity**
  - Sort the edges by weight: O(E log E)
  - Process the edges until a tree is built: O(E log V)
  - ➢ O(E log E + E log V) = O(E log V) ← why?

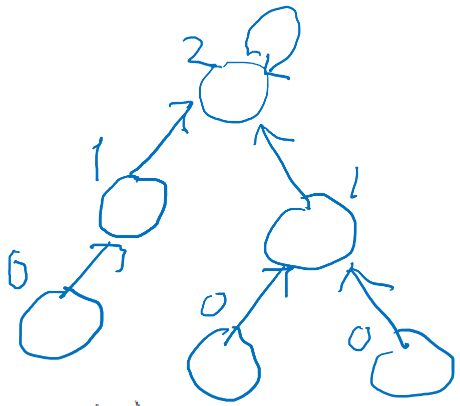
$0 \le E \le O(V^2)$

$\log E \le \log V^2$

```cpp
// C++ program for Kruskal's algorithm to find Minimum Spanning Tree
// of a given connected, undirected and weighted graph
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// a structure to represent a weighted edge in graph
struct Edge {
    int src, dest, weight;
};

// a structure to represent a connected, undirected and weighted graph
struct Graph {
    // V-> Number of vertices, E-> Number of edges
    int V, E;

    // graph is represented as an array of edges. Since the graph is
    // undirected, the edge from src to dest is also edge from dest
    // to src. Both are counted as 1 edge here.
    struct Edge* edge;
};

// Creates a graph with V vertices and E edges
struct Graph* createGraph(int V, int E) {
    struct Graph* graph = (struct Graph*) malloc(sizeof(struct Graph));
    graph->V = V;
    graph->E = E;

    graph->edge = (struct Edge*) malloc(graph->E * sizeof(struct Edge));

    return graph;
}

// A structure to represent a subset for union-find
struct subset {
    int parent;
    int rank;
};

// A utility function to find set of an element i
// (uses path compression technique)
int find(struct subset subsets[], int i) {
    // find root and make root as parent of i (path compression)
    if (subsets[i].parent != i)
```

An implementation of the Kruskal's algorithm
from http://www.geeksforgeeks.org/greedy-algorithms-set-2-kruskals-minimum-spanning-tree-mst/

0	3	9	16
1	8	6	7
2	8	12	27



```
        subsets[i].parent = find(subsets, subsets[i].parent);

    return subsets[i].parent;
}

// A function that does union of two sets of x and y
// (uses union by rank)
void Union(struct subset subsets[], int x, int y) {
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);

    // Attach smaller rank tree under root of high rank tree
    // (Union by Rank)
    if (subsets[xroot].rank < subsets[yroot].rank)
        subsets[xroot].parent = yroot;
    else if (subsets[xroot].rank > subsets[yroot].rank)
        subsets[yroot].parent = xroot;

    // If ranks are same, then make one as root and increment
    // its rank by one
    else {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
}

// Compare two edges according to their weights.
// Used in qsort() for sorting an array of edges
int myComp(const void* a, const void* b) {
    struct Edge* a1 = (struct Edge*)a;
    struct Edge* b1 = (struct Edge*)b;
    return a1->weight - b1->weight;
}

// The main function to construct MST using Kruskal's algorithm
void KruskalMST(struct Graph* graph) {
    int V = graph->V;
    struct Edge *result; // This will store the resultant MST
    result = (struct Edge *) malloc((V-1)*sizeof(struct Edge));

    int e = 0; // An index variable, used for result[]
    int i = 0; // An index variable, used for sorted edges

    // Step 1: Sort all the edges in non-decreasing order of their weight
    // If we are not allowed to change the given graph, we can create a copy of
    // array of edges
    qsort(graph->edge, graph->E, sizeof(graph->edge[0]), myComp);

    // Allocate memory for creating V ssubsets
    struct subset *subsets = (struct subset*) malloc(V * sizeof(struct subset));

    // Create V subsets with single elements
    for (int v = 0; v < V; ++v) {
        subsets[v].parent = v;
        subsets[v].rank = 0;
    }

    // Number of edges to be taken is equal to V-1
    while (e < V - 1) {
        // Step 2: Pick the smallest edge. And increment the index
        // for next iteration
        struct Edge next_edge = graph->edge[i++];

        int x = find(subsets, next_edge.src);
        int y = find(subsets, next_edge.dest);

        // If including this edge does't cause cycle, include it
        // in result and increment the index of result for next edge
        if (x != y) {
            result[e++] = next_edge;
            Union(subsets, x, y);
        }
        // Else discard the next_edge
    }

    // print the contents of result[] to display the built MST
    printf("Following are the edges in the constructed MST₩n");
    for (i = 0; i < e; ++i)
        printf("%d -- %d == %d₩n", result[i].src, result[i].dest, result[i].weight);
    return;
}

// Driver program to test above functions
int main() {
    /* Let us create following weighted graph
             10
       0--------1
       |  ₩     |
    6|    5W    | 15
     |       W |
     2--------3
         4       */
    int V = 4; // Number of vertices in graph
    int E = 5; // Number of edges in graph
    struct Graph* graph = createGraph(V, E);

    // add edge 0-1
    graph->edge[0].src = 0;
    graph->edge[0].dest = 1;
    graph->edge[0].weight = 10;

    // add edge 0-2
    graph->edge[1].src = 0;
    graph->edge[1].dest = 2;
    graph->edge[1].weight = 6;

    // add edge 0-3
    graph->edge[2].src = 0;
    graph->edge[2].dest = 3;
    graph->edge[2].weight = 5;

    // add edge 1-3
    graph->edge[3].src = 1;
    graph->edge[3].dest = 3;
    graph->edge[3].weight = 15;

    // add edge 2-3
    graph->edge[4].src = 2;
    graph->edge[4].dest = 3;
    graph->edge[4].weight = 4;

    KruskalMST(graph);

    return 0;
}
```

Prim's Minimum Spanning Tree Algorithm

- **Idea**
  - In each step, find and add **an edge of the least possible weight** that connects the (current) tree to a non-tree vertex.
- **Algorithm**

```
Given $G = (V, E)$,
Begin with a tree $T^0 = (V^0, E^0)$ where $V^0 = \{v_1\}$ and $E_0 = \{\}$.
repeat { // $T^i = (V^i, E^i) \rightarrow T^{i+1} = (V^{i+1}, E^{i+1})$
    Select a vertex $v$ in $V - V^i$ that is nearest to $V^i$.
    // Let $v$ is from the edge $(u, v)$, where $u$ in $V^i$.

    Update $T$ in such a way that
        $V^{i+1} = V^i + \{v\}$, and $E^{i+1} = E^i + \{(u, v)\}$.
until (an MST is found)
```

- **A key issue in implementation**
  - Tree vertices와 non-tree vertices들을 어떻게 관리할 것인가?
  - Tree vertices와 non-tree vertices들 간의 최소 비용 edge를 어떻게 (효율적으로) 찾을 것인가?

From Prof. Kenji Ikeda's Home Page