Journal of The Korea Society of Computer and Information
Vol. 19, No. 7, July 2014
www.ksci.re.kr
http://dx.doi.org/10.9708/jksci.2014.19.7.131

최소신장트리를 위한 크루스칼 알고리즘의 효율적인 구현

이 주 영*

An Efficient Implementation of Kruskal's Algorithm for A Minimum Spanning Tree

Ju-Young Lee *

요 약

본 논문에서는 최소신장트리를 구하는 크루스칼 알고리즘의 효율적인 구현 방법을 제시한다. 제시하는 방법은 union-find 자료구조를 이용하며, 노드 집합을 나타내는 각 트리의 깊이를 줄이기 위해 union 연산시 루트까지의 경로에 있는 노드들의 위치를 최종 루트의 자식노드로 직접 이동하여 깊이를 줄이도록 하는 방법이다. 이 방법은 루트까지의 경로를 축소하고 노드의 레벨을 축소시킴으로써 트리의 깊이도 줄일 수 있다. 트리의 깊이가 줄어든다면 노드가 속하는 트리의 루트를 찾는 시간을 줄일 수 있게 되어 효율적인 방법이라 할 수 있다. 본 장에서 제안하는 방법을 그래프로 평가해보고 분석해 본 결과, 기존의 union() 방법이나 경로축소방법인 union2() 보다 트리의 깊이를 작게 유지함을 알 수 있다.

▸ Keywords : 최소신장트리, 가중치 그래프, 싸이클, 크루스칼 알고리즘

Abstract

In this paper, we present an efficient implementation of Kruskal's algorithm to obtain a minimum spanning tree. The proposed method utilizes the union-find data structure, reducing the depth of the tree of the node set by making the nodes in the path to root be the child node of the root of combined tree. This method can reduce the depth of the tree by shortening the path to the root and lowering the level of the node. This is an efficient method because if the tree's depth reduces, it could shorten the time of finding the root of the tree to which the node belongs. The performance of the proposed method is evaluated through the graphs generated randomly. The results showed that the proposed method outperformed the conventional method in terms of the depth of the tree.

▸ Keywords : minimum spanning tree, weighted graph, cycle, Kruskal's algorithm

•제1저자 : 이주영 •교신저자 : 이주영
•투고일 : 2014. 6. 24, 심사일 : 2014. 6. 30, 게재확정일 : 2014. 7. 23.
* 덕성여자대학교 컴퓨터학과(School of Information and Media, Duksung Women's University)

I. 서 론

가중치 그래프(weighted graph)에서의 최소신장트리(Minimum Spanning Tree)[1]를 구하는 문제는 중요한 그래프 이론 문제로서 현재 통신 네트워크의 구축 및 라우팅 등에 적용되며 많은 연구가 이루어지고 있다. 뿐만 아니라 최소신장트리를 이용하여 비디오 영상에서 문자 영역을 추출하는 연구[2]가 현재 활발히 진행되고 있으며, 보안 전송(confidential transmission), 비디오 감시(video surveillance), 군사 및 의학 분야에서 활용되는 데이터 은폐(data hiding)처리에도 최소신장트리를 적용하기도 한다[3].

신장트리는 주어진 그래프에서 노드는 모두 포함하면서 간선은 일부분을 포함하는 부분 그래프로써 싸이클(cycle)이 없는 연결된 그래프이다. 간선에 가중치(weight)가 있는 그래프에서 신장트리의 간선들의 가중치 합이 최소인 트리를 최소신장트리라고 한다.

최소신장트리를 구하는 대표적인 알고리즘은 프림(Prim) 알고리즘과 크루스칼(Kruskal) 알고리즘이 있다.

Prim 알고리즘[4,5]은 그래프에서 임의의 노드를 선택하여 트리를 만들고, 이에 연결된 간선들(트리 내의 두 정점을 연결하지 않는 간선) 중에서 가중치가 가장 작은 간선을 선택하여 트리에 추가한다. 이러한 과정을 모든 노드가 트리에 추가될 때까지 반복 수행하여 최소신장트리를 얻는 방법이다. Kruskal 알고리즘[6,7]은 모든 간선들을 오름차순으로 정렬시키고 최소 가중치를 가진 간선을 하나씩 선택하여 트리에 추가하는데, 만약 추가할 때 사이클이 발생하면 그 간선은 제외시키고 그 다음 최소 가중치 간선을 선택하여 추가하는 일을 반복하여 최소신장트리를 얻는다.

이 두 알고리즘의 효율성은 입력으로 주어진 그래프에 따라 결정되는데, 밀집(dense) 그래프의 경우 프림 알고리즘이 더 좋은 비용(시간복잡도)으로 수행되고 희소(sparse) 그래프인 경우 크루스칼 알고리즘이 더 좋은 비용으로 수행된다. 또한, 최소신장트리문제에 대한 랜덤화된 알고리즘이 Klein과 Tarjan[8]에 의해 개발되었는데 이는 $1-e^{-\Omega(m/\log^{a(1)}m)}$ 확률로 시간복잡도 O(E)에 최소신장트리를 구하는 알고리즘이다. 여기서 E는 간선(edge)의 수이다.

크루스칼 알고리즘에서 최소의 가중치를 갖는 간선에 대해 싸이클 발생 여부를 검사하여 그 간선을 최소신장트리에 첨가할지를 결정한다. 최소 가중치를 갖는 간선에 접한(incident) 두 노드들에 대해 자신이 속하는 노드집합을 나타내는 트리의 루트를 찾아서 결정한다. 두 노드가 같은 노드집합의 트리에 속하지 않는다면 싸이클은 발생되지 않는 것으로, 이 경우 두 트리를 합치게 된다. 두 트리를 합칠 때, 어느 트리를 종속시킬 것인가 결정하는 것도 수행 속도에 중요한 요소가 된다.

본 논문에서는 최소신장트리를 구하는 크루스칼 알고리즘의 효율적인 구현 방법을 제시한다.

제안하는 방법은 union-find 자료구조를 이용하며, 상호배타적인 노드 집합을 트리로 표현하는데, 이는 구현이 매우 간단하고 동작 속도가 빠르기 때문이다. 최소 값을 갖는 간선에 대한 싸이클 형성 여부를 확인하기위해 find 연산을 사용하며, 두 트리를 합치기 위해 union 연산을 사용한다. 노드집합을 나타내는 트리의 깊이(depth)를 줄이기 위한 방법으로 union 연산시 루트까지의 경로에 있는 노드들의 위치를 변경시키는 경로 압축 방법을 제안한다. 즉, 루트까지의 경로에 있는 노드들을 두 트리를 합쳤을 때의 최종 루트의 자식노드로 직접 이동시킴으로써 루트까지의 경로를 축소하여 노드들의 레벨을 줄이고 트리의 깊이도 감소시킨다. 트리의 깊이가 줄어든다면 노드가 속하는 트리의 루트를 찾는 시간, union-find 연산을 빠르게 할 수 있으므로 기존의 방법에 비해 더 효율적으로 수행할 수 있다.

본 논문의 구성은 다음과 같다. 1장의 서론에 이어 2장에서는 관련 연구로서 가장 많이 사용되는 대표적인 최소신장트리 알고리즘인 중 크루스칼 알고리즘과 개념들에 대하여 살펴보고 알고리즘 구현의 효율성에 대한 문제를 살펴본다. 3장에서는 트리의 깊이를 줄여 union-find 연산을 효율적으로 하는 방법을 제시한다. 4장에서는 수행 예를 보이고, 마지막으로 5장에서 결론을 맺는다.

II. 관련 연구

그래프 G(V,E)는 노드들의 집합 V와 간선들의 집합 E로 구성되며, 트리(tree)는 싸이클(cycle)이 없는 연결된 그래프이다. 신장트리(spanning tree)는 주어진 그래프에서 노드 집합은 모두 포함하면서 간선은 부분 집합으로 일부분을 포함하여 구성되는 트리이다. 신장트리에 있는 각 노드는 적어도 하나의 간선에 연결되어 있어야 하며 간선의 수는 |V|-1개로 구성된다. 그래프의 간선에 가중치(weight)가 있는 경우, 간선의 가중치의 합이 최소인 신장트리를 최소신장트리라고 한다.

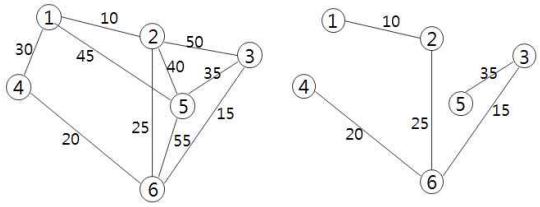
크루스칼 알고리즘은 알고리즘 설계 방법 중 탐욕적인

(greedy) 기법[9,10,11]을 사용하는데, 최소 비용 신장 트리가 사이클을 형성하지 않는 최소 비용 간선을 선택하여 트리에 포함시키는 과정을 반복함으로써 그래프의 모든 노드를 최소비용으로 연결시켜 최적 해답을 구한다. 그림 1은 크루스칼 알고리즘의 의사코드이다[9].

```
/* Input: graph G(V, E) & Output: MST T */
Sort E in increasing oder by weight w;
/* 정렬후 E = {(u1,v1),(u2,v2),...,(u|E|,v|E|)} */
T = ∅
for each v ∈ V
    CREATE-SET(v);
for ei = (ui,vi) from 1 to |E| do
    if FIND-SET(ui) ≠ FIND-SET(vi)
    {
        T = T∪ {(ui,vi)};
        UNION(ui, vi);
    }
return T
```

그림 1. Kruskal 알고리즘(의사코드)
Fig. 1. Kruskal Algorithm(Pseudo code)

그림 2는 크루스칼 알고리즘을 이용하여 얻어진 최소신장트리의 예를 보여준다.



(a) 그래프 G_1 (b) G_1의 최소신장트리

그림 2. Kruskal 알고리즘 적용 결과
Fig. 2. Result of Applying Kruskal's Algorithm

크루스칼 알고리즘을 구현하는데 있어서 최소 간선을 선택하는 방법, 간선을 첨가하였을 때 싸이클을 형성하는지 여부를 확인하는 방법, 그리고 간선을 첨가함으로써 두 개의 상호 배타적인 트리 집합을 합치는 방법 등은 주된 일이라 할 수 있다. 지금까지 사용되는 효율적인 방법은 최소 간선을 선택하기 위해서 간선들을 정렬하고 힙(heap) 등의 자료구조를 사용하여 구현한다.

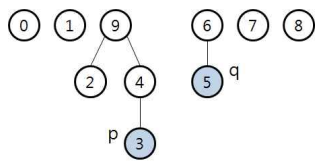
상호 배타적인 집합(disjoint-set)들로 나누어진 원소들에 대한 정보들을 조작하는 자료구조로써 연결리스크(linked list) 혹은 트리 자료구조를 사용하여 구현한다[10,11,12]. 최소 간선을 첨가할 때 싸이클을 형성하는지 여부를 확인하고 두 트리를 합치는 작업은 union과 find 등을 사용한다.

최소 가중치를 갖는 간선을 최소신장트리에 포함시킬지 고려할 때, 먼저 간선에 인접하는(incident) 양 끝 두 노드 x, y가 속하는 트리 집합의 인덱스(루트)를 찾아야 한다. 만약 그 두 노드가 서로 다른 트리에 속한다면, 싸이클이 발생하지 않는다는 의미이므로 그 간선은 최소신장트리에 포함시킬 수 있다. 그리고 노드 x, y가 속하는 트리들은 하나로 합쳐준다.

union(x, y) 연산은 x와 y가 각 각 속해있는 노드집합을 합집합으로 만든다. x가 속하는 집합은 find(x) 연산을 통하여 구한다. 위 그림 1에서 UNION과 FIND-SET이 각 각 이에 해당되는 함수이다.

그림 3의 (a)는 0부터 n-1까지의 인덱스(index)를 가지고 있는 n=10개의 노드들이 union을 통하여 합쳐진 중간 과정으로 현재 6개의 서로 다른 노드집합을 나타내고 있다. 노드집합을 표현하기 위해 트리를 이용하며, 각 집합의 인덱스는 트리의 루트 노드의 인덱스로 나타낸다. 즉, 노드 0, 노드 1, 노드 7, 노드 8은 자기 자신인 한 개의 노드로 이루어진 각 각 서로 다른 네 개의 집합을 나타내며, 각 집합의 인덱스는 각 노드 인덱스인 0, 1, 7, 8로 나타낸다. 따라서 노드 2, 3, 4, 9로 구성되어 있는 집합의 인덱스는 9이며, 노드 5와 6으로 구성되어 있는 집합의 인덱스는 6이다.

그림 3의 (b)는 (a)의 트리 집합에서의 노드 i의 부모 노드를 id[i]로 표현한다. 만일, 노드 i가 루트인 경우 id[i] = i로 나타낸다. 노드 2의 부모노드는 id[2] = 9 이고, 노드 3의 부모노드는 id[3] = 4, 노드 4의 부모노드는 id[4] = 9, 노드 5의 부모노드는 id[5] = 6이다. 루트 노드 0의 부모노드 id[0] = 0, 루트 노드 9의 부모노드 id[9] = 9 등으로 나타낸다. 또한, 노드 3의 부모의 부모 노드는 id[id[3]] = id[4] = 9로써, 노드 9는 노드 2, 3, 4, 9로 구성되어 있는 집합의 루트이다. 노드 3이 속해있는 노드집합의 인덱스는 루트의 인덱스인 9가 된다.



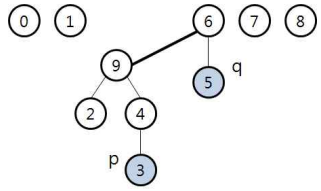
(a) 트리 집합

	0	1	2	3	4	5	6	7	8	9
id[]	0	1	9	4	9	6	6	7	8	9

(b) 배열 id[]

그림 3. 트리 집합과 부모 인덱스를 나타내는 배열 id[]
Fig. 3. Tree sets and array id[]

만일 노드 3과 5 사이에 간선 e(3,5)가 존재하여 이 간선을 신장트리에 포함시킬지 여부를 결정하고자 할 때, 간선에 접한 양 끝 노드 3과 5가 속한 노드집합의 루트를 find()연산을 통하여 구한다. 노드 3이 속한 노드집합(트리)의 루트는 9이고, 노드 5가 속한 노드집합(트리)의 루트는 6으로써 서로 다르다. 즉, 노드 3과 5는 서로 다른 노드집합에 속해있으므로 간선 e(3,5)를 신장트리에 포함시켜도 사이클을 발생하지 않으므로 신장트리에 추가시킨다. 그림 3(a)의 트리(노드집합)들은 노드 3이 속하는 트리와 노드 5가 속하는 트리를 합하는 union(3,5) 연산을 통하여 그림 4(a)와 같이 얻어진다. 배열 id[]에서는 그림 4(b)와 같이 3의 루트(즉, id[id[3]]=9)의 인덱스를 5의 루트(즉, id[5]=6)로 치환한다. 이는 노드 3의 루트가 5의 루트에 종속되는 것을 나타낸다.



(a) union(3,5) 연산 후 트리 집합

	0	1	2	3	4	5	6	7	8	9
id[]	0	1	9	4	9	6	6	7	8	6

(b) union(3,5) 연산 후 배열 id[]

그림 4. union(3,5) 연산 후 트리 집합과 배열 id[]
Fig. 4. Tree sets and array id[] after union(3,5)

그림 5는 union()과 find() 연산을 java 코드로 나타낸 것으로 여기서의 root() 함수는 앞에서 설명한 find() 함수에 해당한다[10,11].

```
private int root(int i)
{
  while (i != id[i])
     i=id[i];
  return i;
}
```

```
pubic void union(int p, int q)
{
  int i = root(p);
  int j = root(q);
  id[i] = j;
}
```

그림 5. root() 와 union() 함수
Fig. 5. root() and union() functions

어떤 노드 i의 루트를 찾을 때, id[i], id[id[i]], ... 등 반복하여 수행하다가 루트까지 도달하면 루프를 종료하고 그 값을 반환한다. 즉, 루트까지의 경로의 길이만큼 루프를 반복하여 도는 것이다.

트리의 깊이(depth)는 트리에 있는 각 노드의 레벨(level) 중 최대 값으로 정의한다. union() 연산에서 두 개의 트리를 합할 때, 합집합이 된 트리의 깊이는 이 두 개의 트리 중 적어도 어느 하나 보다는 증가된다. 최악의 경우, 트리의 깊이는 두 트리를 합할 때마다 두 트리의 깊이의 합으로 증가하여 결국 배열 id[]의 원소 수만큼 될 수 있다. 트리의 깊이가 커질수록 트리의 루트를 찾는 find() 연산 (즉, 그림 5에서의 root() 연산) 수행시간은 더 많이 소요된다.

이러한 문제를 해결하기 위해 Weighting Quick-union 방법[11]은 두 트리를 합칠 때, 트리 크기(즉, 트리에 있는 노드 수)가 더 적은 트리를 크기가 더 큰 트리의 루트에 종속시키는 방법이다. 그림 6은 이에 해당하는 union2() 함수를 나타낸 코드이며, 배열 $sz[i]$는 루트가 i인 트리에 있는 노드의 총 수를 나타낸다.

```
pubic void union2(int p, int q)
{
  int i = root(p);
  int j = root(q);

  If (i == j) { }
  else If (sz[i] < sz[j])
    {  id[i] = j;
       sz[j] += sz[i];}
  else
    {  id[j] = i;
       sz[i] += sz[j];}
}
```

그림 6. union2() 함수
Fig. 6. union2() function

물론 노드 수가 적은 트리를 노드 수가 많은 트리에 종속시켜서 트리를 합치는 방법은 트리의 깊이를 최소화하는 방법이 아닐 수 있다. 예를 들면, 그림 7의 (b)와 같이 노드 수가 더 많은 트리가 반드시 깊이가 더 크다고 할 수 없으므로 노

드 수를 비교하는 것보다 트리의 깊이를 비교해서 결정하는 것이 좋을 수도 있다. 그러나 트리의 깊이로 결정하는 경우에는 상대적으로 더 많은 노드를 포함한 트리의 깊이가 1 증가되므로 상대적으로 많은 노드의 레벨이 증가될 수 있다. 또한, 뒤에서 언급할 노드의 경로축소 방법의 구현 및 효율성을 고려해볼 때, 본 논문에서는 종속될 트리를 결정하는 기준으로 노드 수를 비교하여 결정한다.

보조정리 1. n개의 노드를 가진 트리 T가 union2() 함수를 이용하여 만들어진 트리라면, T의 깊이 $depth(T)$는 다음과 같다[11].

$$depth(T) \leq \log_2(n+1)$$

[증명] 귀납증명법에 의해 다음과 같이 증명된다.

n=1일 때 T의 깊이는 1이므로 $\log_2(1+1)=1$보다 작거나 같다는 것이 명백하다. $i \leq n-1$개의 노드를 가진 트리가 이 정리를 만족한다고 가정하자. 그러면 $i=n$일 때 이 정리가 참이라는 것을 보여주면 된다.

트리 T가 union2()함수에 의해 만들어졌다는 가정 하에 마지막으로 수행된 합집합 연산 union(k, j)을 고려해 보자. m 이 트리 j 의 노드수라면 k 에는 n - m 개의 노드가 있다. 여기서 모순됨이 없이 1 〈= m 〈= n/2 을 가정할 수 있다. 그러면 T 에 있는 노드들의 최고 레벨은 k 의 최고 레벨과 같거나 j 의 그것보다 1 이 크다. 전자의 경우라면 T 의 최고 레벨 log_2 (n-m) + 1〈= log_2 n +1 이 되며, 후자의 경우에는 T 의 최대 레벨 〈= log_2 m + 2 〈= $\log_2 \frac{n}{2}+2 \leq \log_2 n+1$이 되어 만족함을 알 수 있다. □

노드가 속하는 트리의 루트를 찾는 시간은 트리의 깊이에 좌우되므로, 보조정리 1에 따라 어떤 노드가 자신이 속하는 트리(노드 개수는 n)의 루트(즉, 트리인덱스)를 찾기 위한 연산은 $O(\log_2 n)$의 시간복잡도를 가진다. 따라서 트리의 깊이를 줄일 수 있다면 수행 시간도 단축될 수 있다.

다음 장에서는 경로축소(path reduction) 방법[11,13]을 이용하여, 트리의 깊이를 줄이기 위한 효율적인 구현 방법을 제안한다.

III. 제안하는 방법

본 장에서는 2장에서 설명한 크루스칼 알고리즘의 수행 시간을 줄일 수 있는 효율적인 구현 방법을 제안한다.

크루스칼 알고리즘에서는 그래프의 간선들 중 최소 값을 갖는 간선을 하나씩 선택하여 신장트리에 포함시킬 수 있는지에 대한 싸이클 발생 여부를 검사한 후 싸이클을 발생시키지 않으면 신장트리에 포함시킨다. 이때 최소 값을 갖는 간선의 인접한 두 노드들이, 형성된 노드집합의 트리들 중에서 같은 트리에 속하지 않으면 싸이클을 발생시키지 않는 것이므로 두 노드가 어느 트리에 속하는지를 먼저 찾아야 한다(그림 3과 4 참조).

각 트리의 인덱스(ID)는 그 트리의 루트 노드의 인덱스(ID)로써 나타낸다. 어떤 노드 i가 속하는 트리의 인덱스(즉, 루트 인덱스)를 구하는 함수를 find_root()로 정의한다.

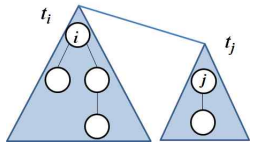
간선의 두 노드가 속하는 트리의 루트를 find_root()를 통해 구하고, 만일 같은 트리에 속하지 않으면 두 트리를 합치게 된다. 두 개의 트리를 하나의 트리로 합치는 일을 수행할 때, 어느 트리를 종속시킬 것인가 결정하는 데 있어서 트리의 노드 수를 기준으로 한다. 2장에서 설명한 것처럼 트리의 노드 수를 비교하여 노드 수가 적은 트리를 노드 수가 더 많은 트리의 루트의 자식노드가 되도록 종속시킨다.

노드 수를 기준으로 종속되는 트리를 결정하고 union() 함수를 사용하여 두 트리를 합칠 때, 합쳐진 트리의 깊이는 다음과 같은 식으로 나타낼 수 있다. 여기서 트리 t_i는 루트가 i인 노드집합을 나타내는 트리이고, 트리 t_j는 루트가 j인 트리이며, 트리 T는 두 트리들을 합하여 만들어진 트리이다.

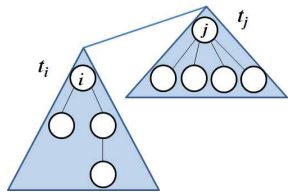
$$depth(T)=\begin{cases}\max\{depth(t_i),\ depth(t_j)+1\}, & \text{if } sz[i] \geq sz[j]\\ \max\{depth(t_i)+1,\ depth(t_j)\}, & else\end{cases}$$

두 트리의 노드 수를 비교하여 $sz[i] \geq sz[j]$이라면 그림 7의 (a)와 같이 루트 i의 자식으로 트리 t_j가 종속되어 트리 t_j에 있는 모든 노드의 레벨은 1 증가된다. 즉, 합쳐진 트리 T의 깊이는 t_i의 깊이, 혹은 t_j의 깊이보다 1 증가된 값 중 더 큰 값이 된다.

또한 $sz[i] < sz[j]$인 경우, 그림 7의 (b)와 같이 루트 j의 자식으로 트리 t_i가 종속되어 트리 t_i에 있는 모든 노드의 레벨은 1 증가된다. 즉, 합쳐진 트리 T의 깊이는 t_i의 깊이보다 1 증가된 값, 혹은 t_j의 깊이 중 더 큰 값이 된다.



(a) $sz[i] \geq sz[j]$ 인 경우



(b) $sz[i] < sz[j]$ 인 경우

그림 7. union() 이용한 트리의 합집합
Fig. 7. Tree combined by union()

이와 같이 union() 연산 후 트리의 깊이가 반복적으로 증가된다면, 크루스칼의 수행시간을 좌우하는 union_find 연산의 수행 속도가 느려지게 된다. 자신이 속한 노드집합을 나타내는 트리의 루트를 찾는 연산은 자신의 레벨 수만큼 반복 수행되므로, 트리의 깊이는 적을수록 효율적이다. 그러므로 union() 연산에서 트리의 깊이가 증가되지 않도록 트리를 유지하는 방법을 구한다면 크루스칼 알고리즘을 더 효율적으로 수행할 수 있을 것이다.

루트를 찾기 위한 연산을 수행하는 과정에서 루트까지 가는 경로에 있는 노드들의 레벨을 줄일 수 있다면 전체 트리의 깊이가 줄어들 수 있다. find_root() 함수는 입력된 노드가 속한 트리의 루트 인덱스를 반환하는데, 노드에서 루트까지 가는 경로에 있는 노드들을 inter_node 집합에 저장한다. 트리의 루트를 찾은 후, inter_node 집합에 있는 노드들은 루트 노드의 자식으로 이동시킨다. 즉, inter_node 집합에 있는 노드들의 id[] 값을 루트 노드의 인덱스로 변경한다.

다음으로, 두 트리를 합치는 연산에 대해 설명한다. 제안하는 방법은 두 트리를 합칠 때 트리의 노드 수를 비교하여 노드 수가 적은 트리를 노드 수가 더 많은 트리의 루트의 자식노드로 종속시키는 기본 방법은 union2()와 같다. 그러나 find_root() 연산 시 얻어진 inter_node 집합에 있는 노드들 각 각에 대해 노드 수가 더 많은 트리에다 직접 종속시켜 루트의 자식노드로 이동한다.

그림 8은 이에 해당하는 find_root()와 union3() 함수를 나타낸 의사코드이다.

```
int find_root(int i)
{
   while (i != id[i])
   {  inter_node = inter_node ∪ {i};
      i = id[i];  }
   return i;
}

void union3(int p, int q)
{  inter_node = ∅;
   i = find_root(p);
   j = find_root(q);
   if (i != j)
   {
      if (sz[i] >= sz[j])
      {  for each node k ∈ inter_node∪{j}
            id[k] = i;
         sz[i] += sz[j]; }
      else
      {  for each node k ∈ inter_node∪{i}
            id[k] = j;
         sz[j] += sz[i]; }
   }
}
```

그림 8. find_root()와 union3()의 의사코드
Fig. 8. pseudo codes for find_root() and union3()

제안하는 방법의 개략적인 설명을 위하여 i-1개의 간선이 신장트리에 이미 포함되어 있고, 현재 i번째 간선을 선택한다고 가정하자. 남은 간선들 중에서 최소 가중치를 가진 간선이 (p,q)라고 할 때, 간선 (p,q)를 신장트리에 추가할 수 있는지를 여부를 확인하고 신장트리에 추가하는 동작을 살펴보겠다.

그림 9의 (a)는 노드 0과 10을 각 각 루트로 하는 트리 t_0과 t_{10}으로 트리의 ID는 루트 인덱스로 나타내며, 같은 부분 신장트리에 속하는 노드 집합을 트리로써 나타낸 것이다. 배열 id[]에는 부모 노드에 해당하는 인덱스가 저장되어 있다. 현재 p=6, q=12인 간선 (p,q)가 최소 가중치를 갖는 간선이라고 할 때, 간선 (p,q)를 신장트리에 추가할 수 있는지를 결정하기 위해 먼저 싸이클을 형성하는지 확인해야 한다. 두 노드가 같은 트리에 속하는지 find_root() 함수를 사용하여 노드 p와 q가 속한 트리의 루트를 각 각 구하여, 루트가 같지 않으면 싸이클을 형성하지 않는 것이므로 이 간선은 신장트리에 추가할 수 있는 것이다.

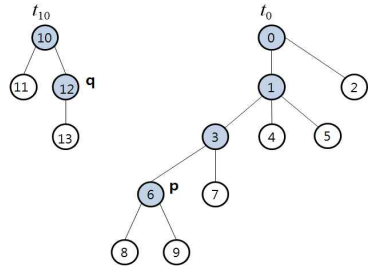
find_root() 수행하여 트리의 루트는 다음과 같이 구한다. 노드 6의 경우 id[6]= 3이고, 노드 3의 경우 id[3]= 1, 노드 1의 경우 id[1]= 0, 노드 0의 경우 id[0]= 0이므로, 노드 6이 속한 트리의 루트는 0이다. 트리의 루트까지의 경로에 있는 노드들은 inter_node 집합에 저장된다. 즉, find_root(6) 연산에서 노드 6에서 루트인 0까지의 경로에 있는 노드들은 그림 9의 (a)에서 음영을 준 노드들인 6, 3, 1, 0이다. 이 집합에 있는 노드들 각 각 id[] 값은 트리의 루

트 인덱스로 저장이 된다. 즉, 이 노드들을 루트로 하는 서브 트리가 루트의 자식으로 종속되어 노드들의 레벨이 줄어들게 된다. 노드 6의 경우 id[6]= 3이고, 노드 3의 경우 id[3]= 1, 노드 1의 경우 id[1]= 0, 노드 0의 경우 id[0]= 0이므로, 노드 6이 속한 트리의 루트는 0이다. 이 값들은 id[6]= id[3]= id[1]= 0으로 변경되어 루트의 자식으로 이동되어 레벨이 축소되고 트리의 깊이가 축소될 수도 있게 된다.

그림 9의 (b)는 find_root(6)과 find_root(12)를 수행한 후 변경된 트리 t_0과 t_{10}를 보여준다. 노드 6의 레벨은 2가 줄어들었고, 노드 3의 레벨은 1이 줄어들었다. 이와 같은 경로 축소 방법으로 각 노드들의 레벨을 줄여 노드에서 루트까지의 경로를 축소시켜 효율적으로 수행하도록 한다.

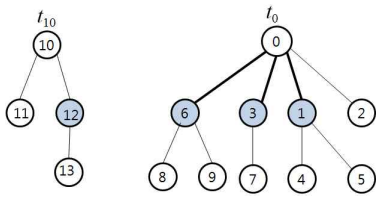
그림 9의 (c)는 union3()을 수행한 후의 두 트리가 하나의 트리로 합쳐진 결과를 보여준다. 그림 9의 (b)에서의 트리 t_0의 노드 수는 10개이고, 트리 t_{10}의 노드 수는 4개이므로 $sz[0] \geq sz[10]$으로써 트리 t_0의 자식으로 트리 t_{10}가 종속된다. find_root(12) 수행 시 구한 집합 inter_node ={10, 12}에 있는 노드 10과 12의 id[] 값을 트리 t_0의 루트 노드의 인덱스인 0으로 변경한다. 즉, id[10] = id[12] = 0이 되어 노드 10과 12는 그림 9의 (c)와 같이 t_0의 루트 노드의 자식으로 이동하게 된다.

이러한 경로축소방법을 통해 노드의 레벨을 축소시킴으로써 합쳐진 트리의 깊이도 줄일 수 있다. 트리의 깊이가 작아진다면 노드가 속하는 트리의 루트를 찾는 시간은 트리의 깊이에 비례하므로 제안하는 방법은 효율적인 방법이라 할 수 있다.

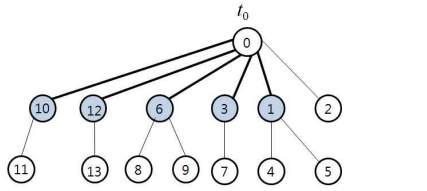


	0	1	2	3	4	5	6	7	8	9	10	11	12	13
id	0	0	0	1	1	1	3	3	6	6	10	10	10	12

(a) 최소가중치간선(p,q)일 때의 p와 q가 속한 트리 및 id[]



(b) find_root(6)와 find_root(12) 수행 후



	0	1	2	3	4	5	6	7	8	9	10	11	12	13
id	0	0	0	0	1	1	0	3	6	6	0	10	0	12

(c) union3(0,10) 수행 후 합쳐진 트리와 배열 id[]

그림 9. 제안하는 경로축소 과정
Fig. 9. Process of Tree being combined by proposed method

IV. 알고리즘 적용성 평가

본 장에서는 그림 10에 있는 그래프에 대해 알고리즘 적용성을 평가한다.

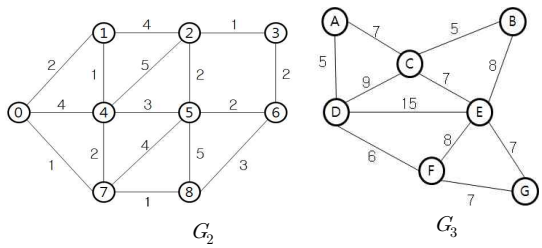


그림 10. 알고리즘 평가에 사용된 그래프
Fig 10. Graphs for Evaluation of Algorithm

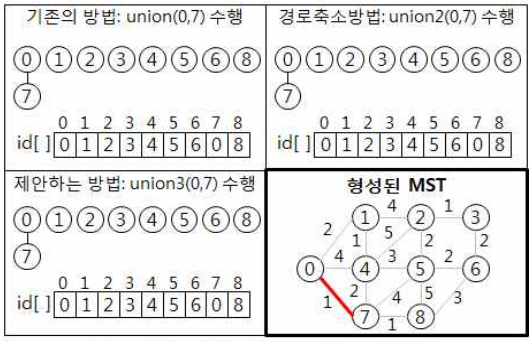
그림 11은 그래프 G_2에 대해 알고리즘을 적용한 수행 과정을 단계별로 (a)~(h)에 나타낸 것이다. 각 단계 그림에서 기존의 방법인 union()과 경로축소 알고리즘을 사용한 개선된 방법인 union2(), 그리고 본 논문에서 제안하는 방법인 union3()을 수행한 결과를 왼쪽 상단, 오른 쪽 상단, 왼쪽 하단에 각 각 보여준다. 오른 쪽 하단의 그래프는 최소신장트리

가 형성되는 단계를 보여준다.

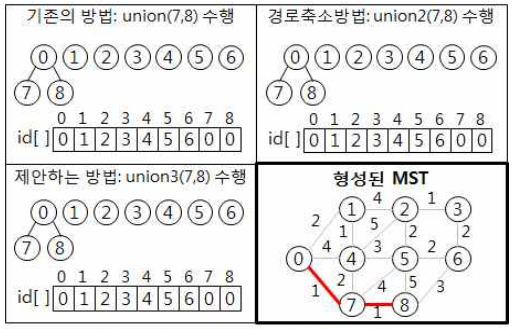
그림 10(e)에서 현재 최소 값을 갖는 간선(4,7)을 선택하여 싸이클 형성 여부를 판단한다. 노드 4가 속하는 트리의 루트는 1이고 노드 7이 속하는 트리의 루트는 0이다. 즉 루트가 서로 다르므로 간선(4,7)은 최소신장트리에 첨가할 수 있으며, 노드 4가 속하는 트리와 노드 7이 속하는 트리는 각 각 union(), union2(), union3() 함수를 사용하여 합친다. 각 방법으로 합쳐진 트리는 왼쪽 상단, 오른 쪽 상단, 왼쪽 하단에 각 각 보여주고, 오른 쪽 하단에는 간선(4,7)를 첨가한 부분 최소신장트리를 보여준다. 제안하는 방법을 이용하여 두 트리를 합친 결과의 트리에서는 그 깊이가 증가되지 않고 원래의 깊이를 유지하고 있다.

그림 10(g)는 간선(3,6)을 최소 간선으로 선택하여 노드 3이 속하는 트리의 루트 2와 노드 6이 속하는 트리의 루트 5가 서로 다르므로 역시 간선(3,6)도 최소신장트리에 첨가할 수 있다. 노드 3이 속하는 트리와 노드 6이 속하는 트리는 각 각 union(), union2(), union3() 함수를 사용하여 합친다. 각 방법에 의해 합쳐진 트리의 결과에서 보듯이 제안하는 방법은 기존의 다른 두 방법과는 달리 두 트리를 합친 결과로 만들어진 트리에서 깊이가 증가되지 않도록 하는 방법으로 효율적인 방법이라 할 수 있다.

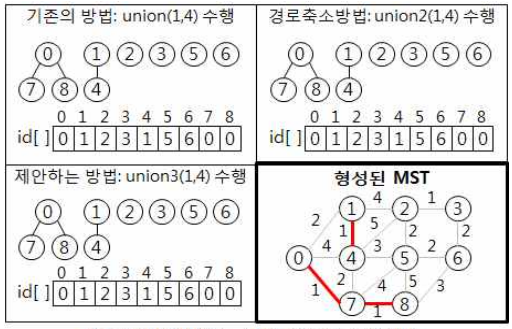
그림 10(h)에서는 간선(4,5)을 최소 간선으로 선택하였으며, 노드 4가 속하는 트리의 루트 0과 노드 5가 속하는 트리의 루트 2가 서로 서로 다르므로 간선(4,5)는 최소신장트리에 첨가할 수 있다. 이 경우, 제안하는 방법도 기존의 다른 두 방법과 같이 두 트리를 합친 결과로 만들어진 트리에서 깊이가 1 증가되었다. 그러나 다른 방법과는 달리, 종속되는 트리(즉, 노드 개수가 적은 트리)에 있는 노드들의 레벨이 모두 1 증가되지는 않는다. 즉, 노드 5의 경우 트리가 합쳐질 때 직접 루트의 자식으로 변경되어 레벨이 증가되지 않게 되어 효율적이다.



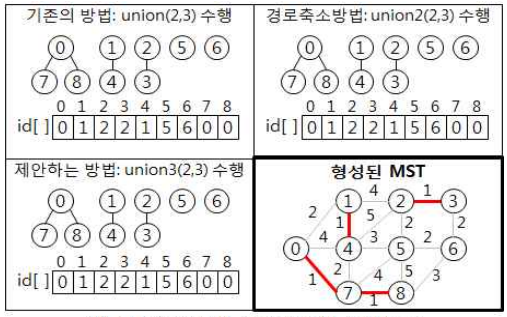
(a) 간선 (0,7) 첨가 시 각 방법의 수행 비교



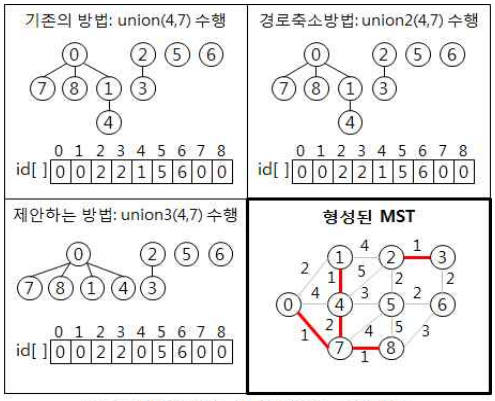
(b) 간선 (7,8) 첨가 시 각 방법의 수행 비교



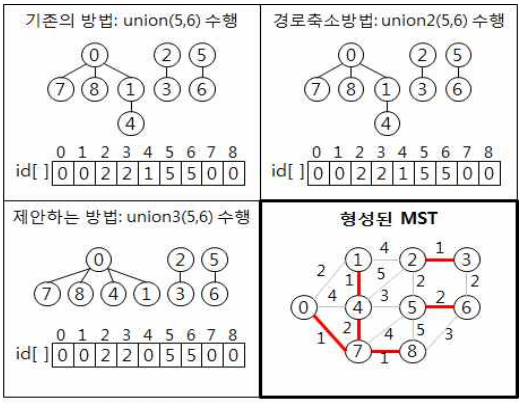
(c) 간선 (1,4) 첨가 시 각 방법의 수행 비교



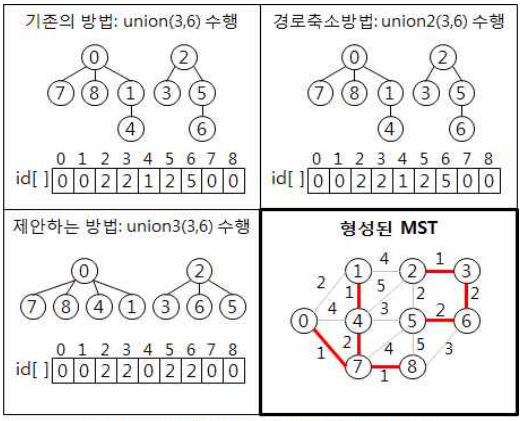
(d) 간선 (2,3) 첨가 시 각 방법의 수행 비교



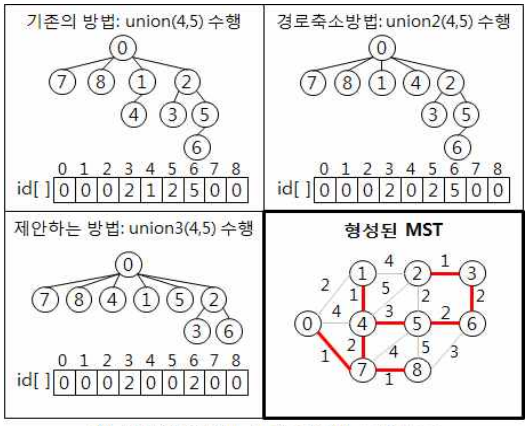
(e) 간선 (4,7) 첨가 시 각 방법의 수행 비교



(f) 간선 (5,6) 첨가 시 각 방법의 수행 비교



(g) 간선 (3,6) 첨가 시 각 방법의 수행 비교



(h) 간선 (4,5) 첨가 시 각 방법의 수행 비교

그림 11. 그래프 G_2에 각 알고리즘 적용한 단계별 과정
Fig 11. Each Algorithm Applying to Graph G_2

그래프 G_3에 대해서는 편의상 수행과정을 생략하고, 마지막 간선 첨가 후의 노드집합을 나타내는 트리들과 알고리즘 수행 결과 얻은 최소신장트리들을 그림 12에서 보여준다.

그림 12에서 보듯이 기존의 두 방법의 수행 결과, 노드집합을 나타내는 최종 트리의 깊이는 3이고, 노드 B, E, D, F의 레벨은 2이며, 노드 C, G의 레벨은 3이다. 반면 제안하는 방법으로 수행한 결과, 최종 트리의 깊이는 2, 루트를 제외한 6개의 노드들이 레벨 2로 기존의 union() 방법이나 경로축소방법인 union2() 보다 트리의 깊이가 더 적다.

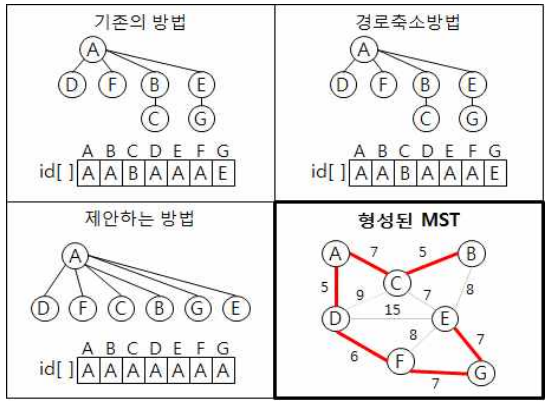


그림 12. 그래프 G_3의 각 알고리즘 적용 결과
Fig 12. Algorithm Applying to Graph G_3

본 장에서 제안하는 방법을 그래프로 평가해보고 분석해 본 결과, 기존의 union() 방법이나 경로축소방법인 union2() 보다 트리의 깊이를 작게 유지함을 알 수 있다. 이는 두 트리를 합치는 union 연산시 루트까지의 경로에 있는 노드들의 위치를 최종 루트의 자식노드로 직접 이동시킴으로써 노드들의 레벨을 줄이고 트리의 깊이도 감소시키는 방법이기 때문이다.

V. 결 론

본 논문에서 최소신장트리를 구하는 크루스칼 알고리즘의 효율적인 구현 방법을 제시하였다.

크루스칼 알고리즘은 그래프에서 최소의 가중치를 갖는 간선을 차례로 선택하여 싸이클 발생 여부를 검사한 후 싸이클이 발생되지 않으면 신장트리에 포함시킨다. 이때 최소 가중치를 갖는 간선에 접한(incident) 두 노드들에 대해 각 각 어느 트리에 속하는지 트리의 루트를 구한다. 두 노드가 노드집합을 나타내는 트리들 중에서 같은 트리에 속하지 않다면 싸이클은 발생되지 않는 것이며, 이 경우 두 트리를 합치게 된다. 두 트리를 합칠 때, 어느 트리를 종속시킬 것인가 결정

하는 데 있어서 트리의 노드 수를 기준으로 한다.

제안하는 방법은 union-find 자료구조를 이용하며, 노드 집합을 나타내는 트리의 깊이를 줄이기 위해 서브트리들을 루트의 자식노드로 직접 이동시킴으로써 루트까지의 경로를 축소하고 노드의 레벨을 감소시킴으로써 트리의 깊이도 줄일 수 있다. 트리의 깊이가 줄어든다면 노드가 속하는 트리의 루트를 찾는 시간을 줄일 수 있게 된다.

제안한 방법들을 실제 그래프에 적용한 결과 기존의 방법에 비해 노드들의 레벨과 트리의 깊이도 감소시킬 수 있었으며, 이는 노드가 많아질수록 그 효과가 클 것으로 분석된다. 트리의 깊이가 줄어든다면 노드가 속하는 트리의 루트를 찾는 시간, union-find 연산을 빠르게 할 수 있으므로 기존의 방법에 비해 제안하는 방법은 크루스칼 알고리즘을 더 효율적으로 수행할 수 있다.

참고문헌

〔1〕 Wikipedia, "Minimum Spanning Tree," http://en.wikipedia.org/wiki/Minimum_spanning_tree, Wikimedia Foundation, Inc., 2010.

〔2〕 B. Chun and Y. Kim, "A Method for Character Segm entation using MST," Journal of the Korea Society of Computer and Information, Vol.11, No.3, pp.73-78, July 2006.

〔3〕 P. Amat, W. Puech, S. Druon, and J. P. Pedeboy, "Lossless 3D Steganography Based on MST and Connectivity Modification," Image Communication, Vol.25, No.6, pp.400-412, July 2010.

〔4〕 R. C. Prim, "Shortest Connection Networks and Some Generalisations," Bell System Technical Journal, vol. 36, pp. 1389-1401, Nov. 1957.

〔5〕 Wikipedia, "Prim's Algorithm," http://en.wikipedia.org/wiki/Prims_algorithm, Wikimedia Foundation, Inc., 2010.

〔6〕 J. B. Kruskal, "On the Shortest Spanning Subtree and The Traveling Salesman Problem," Proceedings of the American Mathematical Society, Vol 7, No. 1, pp. 48-50, Feb, 1956.

〔7〕 Wikipedia, "Kruskal's Algorithm," http://en.wikipedia.org/wiki/Kruskal_algorithm, Wikimedia Foundation, Inc., 2010.

〔8〕 D. R. Karger, P. N. Klein, and R. E. Tarjan, "A randomized linear-time algorithm to find minimum spanning trees," Journal of the ACM (JACM) Vol 42, Issue 2, pp. 321-328, March 1995.

〔9〕 T. H. Cormen, C. E. Leiserson, R. L. Rivest, and Clifford Stein, "Introduction To Algorithms," Second Edition, The MIT Press, McGraw-Hill, 2002.

〔10〕 Michael T. Goodrich and Roberto Tamassia, "Data Structures and Algorithms in Java," Fourth Edition, John Wiley & Sons, Inc., 2006.

〔11〕 R. Sedgewick and K. Wayne, "Algorithms," Fourth Edition, Addison-Wesley, 2011.

〔12〕 E. Horowitz, and S. Sahni, "Fundamentals of Data Structures in C," Anderson-Freed, 2008.

〔13〕 Jongman Goo, "Strategy of Algorithm Problem Solving," insight, 2012.

저 자 소 개

이 주 영(Lee, Ju-Young)

1984: 이화여자대학교 수학과 학사.

1991: The George Washington Univ. 컴퓨터학과 석사.

1996: The George Washington Univ. 컴퓨터학과 박사.

1996~현재 : 덕성여자대학교 컴퓨터학과 교수.

관심분야: 알고리즘, 분산/병렬처리, HCI, 그래프 이론

Email : jylee@duksung.ac.kr