6.4 최소 비용 Spanning tree

weighted, undirected graph의 spanning tree의 비용은 spanning tree의 edge weight의 합이다. minimum-cost spanning tree T는 edge weight의 합이 최소가 되는 spanning tree이다. minimum-cost spanning tree는 greedy-method로서 Kruskal, Prim, Sollin 알고리즘으로 구한다. greedy-method(탐욕적 방법)은 여러 경우 중에서 하나를 선택할 때마다 그 순간에 best라고 생각되는 것을 선택(매 순간마다 best만 선택하므로 탐욕적이라 부름)하는 방식이다. 매 순간마다 best를 선택한다고 해서 최종적인 선택이 best라고 항상 보장할 수는 없다. greedy-method가 잘 작동하는 문제는 greedy choice property와 optimal substructure 조건이 만족되는 경우이다. greedy choice property 조건은 앞의 best 선택이 이후의 best 선택에 영향을 주지 않는다는 것이다. optimal substructure 조건은 문제 전체에 대한 최적해(best 선택)가 부분 문제에 대해서도 역시 최적해라는 것이다. 이러한 두 가지 조건이 만족되지 않으면 greedy-method가 최적 해를 구하지 못한다.

greedy-method에서 각 stage별로 best 선택에 의한 optimal 해를 만든다. 초기 stage에서의 best 선택은 나중 stage에서 변경할 수 없다. minimum-cost spanning tree를 만들기 위한 best 선택은 least-cost를 선택할 때 사용하는 constraints는 다음과 같다: 1) 그래프의 edge 정보만 사용해야 한다. 2) n-1 edge만을 사용해야 한다. 3) cycle을 만드는 edge는 사용해서는 안 된다.

Minimal-cost spanning tree 문제는 자료구조로서 adjacency list와 set를 사용하여 알고리즘 구현하기에 좋은 예제이다. 어떠한 자료구조를 선택하여 재사용할 것인지에 대한 실습 과제로서 소스 코드를 완성해보는 실습을 수행한다. 정렬을 위해 min-heap을 사용하고 cycle 여부를 판단하기 위하여 set의 find()를 사용하도록 소스코드 6.2을 수정한다.

```
//소스 코드 6.2: Minimal Spanning Tree
//  minimal spanning tree:: Kruskal's source code
// min heap, set 사용하여 MST 구현
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <iostream>
using namespace std;
const int HeapSize = 100;

enum Boolean { FALSE, TRUE };

template <class Type>
class Element {
public:
      Type key;
};
```

```
template <class Type>
class MaxPQ {
public:
	virtual void Insert(const Element<Type>&) = 0;
	virtual Element<Type>* DeleteMax(Element<Type>&) = 0;
};

template <class Type>
class MaxHeap : public MaxPQ<Type> {
public:
	MaxHeap(int sz = HeapSize)
	{
		MaxSize = sz; n = 0;
		heap = new Element<Type>[MaxSize + 1]; // Don't want to use heap[0]
	};
	void display( );
	void Insert(const Element<Type>& x);
	Element<Type>* DeleteMax(Element<Type>&);
private:
	Element<Type>* heap;
	int n; // current size of MaxHeap
	int MaxSize; // Maximum allowable size of MaxHeap

	void HeapEmpty( ) { cout << "Heap Empty" << "\n"; };
	void HeapFull( ) { cout << "Heap Full"; };
};

template <class Type>
void MaxHeap<Type>::display( )
{
	int i;
	cout << "MaxHeap:: (i, heap[i].key): ";
	for (i = 1; i <= n; i++) cout << "(" << i << ", " << heap[i].key << ") ";
	cout << "\n";
}

template <class Type>
void MaxHeap<Type>::Insert(const Element<Type>& x)
{
	int i;
	if (n == MaxSize) { HeapFull( ); return; }
	n++;
	for (i = n; 1; ) {
		if (i == 1) break; // at root
		if (x.key <= heap[i / 2].key) break;
		// move from parent to i
		heap[i] = heap[i / 2];
		i /= 2;
	}
	heap[i] = x;
}
```

```
template <class Type>
Element<Type>* MaxHeap<Type>::DeleteMax(Element<Type>& x)
{
	int i, j;
	if (!n) { HeapEmpty( ); return 0; }
	x = heap[1]; Element<Type> k = heap[n]; n--;

	for (i = 1, j = 2; j <= n; )
	{
		if (j < n) if (heap[j].key < heap[j + 1].key) j++;
		// j points to the larger child
		if (k.key >= heap[j].key) break;
		heap[i] = heap[j];
		i = j; j *= 2;
	}
	heap[i] = k;
	return &x;
}

class Sets {
public:
	Sets(int);
	void display( );
	void SimpleUnion(int, int);
	int SimpleFind(int);
	void WeightedUnion(int, int);
	int CollapsingFind(int);
private:
	int* parent;
	int n;
};

Sets::Sets(int sz = HeapSize)
{
	n = sz;
	parent = new int[sz + 1]; // Don't want to use parent[0]
	for (int i = 1; i <= n; i++) parent[i] = -1;  // 0 for Simple versions
}

void Sets::SimpleUnion(int i, int j)
// Replace the disjoint sets with roots i and j, i != j with their union
{
	parent[j] = i;
}

int Sets::SimpleFind(int i)
// Find the root of the tree containing element i
{
	while (parent[i] > 0) i = parent[i];
	return i;
}
```

```
void Sets::WeightedUnion(int i, int j)
// Union sets with roots i and j, i != j, using the weighting rule.
// parent[i]~=~-count[i] and parent[i]~=~-count[i].
{
	int temp = parent[i] + parent[j];
	if (parent[i] > parent[j]) { // i has fewer nodes
		parent[i] = j;
		parent[j] = temp;
	}
	else { // j has fewer nodes
		parent[j] = i;
		parent[i] = temp;
	}
}

int Sets::CollapsingFind(int i)
// Find the root of the tree containing element i.
// Use the collapsing rule to collapse all nodes from @i@ to the root
{
	int r;
	for (r = i; parent[r] > 0; r = parent[r]); // find root
	while (i != r) {
		int s = parent[i];
		parent[i] = r;
		i = s;
	}
	return r;
}

void Sets::display( )
{
	cout << "display:index= ";
	for (int i = 1; i <= n; i++) cout << " " << i;
	cout << "\n";
	cout << "display: value= ";
	for (int i = 1; i <= n; i++) cout << " " << parent[i];
	cout << "\n";
}

template <class Type> class List;
template <class Type> class ListIterator;

template <class Type>
class ListNode {
	friend class List<Type>;
	friend class ListIterator<Type>;
private:
	Type data;
	ListNode* link;
	ListNode(Type);
};
```

```
template <class Type>
ListNode<Type>::ListNode(Type Default)
{
	data = Default;
	link = 0;
}

template <class Type>
class List {
	friend class ListIterator<Type>;
public:
	List( ) { first = 0; };
	void Insert(Type);
	void Delete(Type);
private:
	ListNode<Type>* first;
};

template <class Type>
void List<Type>::Insert(Type k)
{
	ListNode<Type>* newnode = new ListNode<Type>(k);
	newnode->link = first;
	first = newnode;
}

template <class Type>
void List<Type>::Delete(Type k)
{
	ListNode<Type>* previous = 0;
	for (ListNode<Type>* current = first; current&& current->data != k;
		previous = current, current = current->link);
	if (current)
	{
		if (previous) previous->link = current->link;
		else first = first->link;
		delete current;
	}
}

template <class Type>
class ListIterator {
public:
	ListIterator(const List<Type>& l) : list(l) { current = l.first; }
	Type* First( );
	Type* Next( );
	bool NotNull( );
	bool NextNotNull( );
private:
	const List<Type>& list;
	ListNode<Type>* current;
```

```
};

template <class Type>
Type* ListIterator<Type>::First( ) {
	if (current) return &current->data;
	else return 0;
}

template <class Type>
Type* ListIterator<Type>::Next( ) {
	current = current->link;
	return &current->data;
}

template <class Type>
bool ListIterator<Type>::NotNull( )
{
	if (current) return true;
	else return false;
}

template <class Type>
bool ListIterator<Type>::NextNotNull( )
{
	if (current->link) return true;
	else return false;
}

//template <class Type>
ostream& operator<<(ostream& os, List<char>& l)
{
	ListIterator<char> li(l);
	if (!li.NotNull( )) return os;
	os << *li.First( ) << endl;
	while (li.NextNotNull( ))
		os << *li.Next( ) << endl;
	return os;
}

class Queue;

class QueueNode {
	friend class Queue;
private:
	int data;
	QueueNode* link;
	QueueNode(int def = 0, QueueNode* l = 0)
	{
		data = def;
		link = l;
	};
};
```

```
class Queue {
private:
	QueueNode* front, * rear;
	void QueueEmpty( ) { };
public:
	Queue( ) { front = rear = 0; };
	void Insert(int);
	int* Delete(int&);
	bool IsEmpty( ) { if (front == 0) return true; else return false; };
};

void Queue::Insert(int y)
{
	if (front == 0) front = rear = new QueueNode(y, 0); // empty queue
	else rear = rear->link = new QueueNode(y, 0); // update \fIrear\fR
}

int* Queue::Delete(int& retvalue)
// delete the first node in queue and return a pointer to its data
{
	if (front == 0) { QueueEmpty( ); return 0; };
	QueueNode* x = front;
	retvalue = front->data; // get data
	front = x->link;	// delete front node
	if (front == 0) rear = 0; // queue becomes empty after deletion
	delete x; // free the node
	return &retvalue;
}

class Graph
{
private:
	List<int>* HeadNodes;
	int n;
	bool* visited;

	void _DFS(const int v);
public:
	Graph(int vertices = 0) : n(vertices) {
		HeadNodes = new List<int>[n];
	};
	void BFS(int);
	void InsertVertex(int startNode, int endNode, int weight);
	void Setup( );
	void displayAdjacencyLists( );

	void DFS(int v);

};

void Graph::displayAdjacencyLists( ) {
```

```
	for (int i = 0; i < n; i++) {
		//HeadNodes[i];
		ListIterator<int> iter(HeadNodes[i]);
		if (!iter.NotNull( )) {
			cout << i << " -> null" << endl;
			continue;
		}
		cout << i;
		for (int* first = iter.First( ); iter.NotNull( ); first = iter.Next( )) {
			cout << " -> " << (*first);
		}
		cout << endl;
	}
}

void Graph::InsertVertex(int start, int end, int weight) {
	if (start < 0 || start >= n || end < 0 || end >= n) {
		cout << "the start node number is out of bound.";
		throw "";
	}
	//check if already existed.
	ListIterator<int> iter(HeadNodes[start]);
	for (int* first = iter.First( ); iter.NotNull( ); first = iter.Next( )) {
		if (*first == end) return;
	}

	HeadNodes[start].Insert(end, weight);
	HeadNodes[end].Insert(start, weight);
}

void Graph::BFS(int v)
{
	visited = new bool[n]; // visited is declared as a Boolean \(** data member of Graph .
	for (int i = 0; i < n; i++) visited[i] = false; // initially, no vertices have been visited

	visited[v] = true;
	cout << v << ",";
	Queue q;
	q.Insert(v);

	while (!q.IsEmpty( )) {
		v = *q.Delete(v);
		ListIterator<int> li(HeadNodes[v]);
		if (!li.NotNull( )) continue;
		int w = *li.First( );
		while (1) {
			if (!visited[w]) {
				q.Insert(w);
				visited[w] = true;
				cout << w << ",";
			};
			if (li.NextNotNull( ))
```

```
                        w = *li.Next( );
                else break;
            }
        }
        delete[ ] visited;
}

// Driver
void Graph::DFS(int v)
{
        visited = new bool[n]; // visited is declared as a bool \(** data member of Graph .
        for (int i = 0; i < n; i++)
            visited[i] = false; // initially, no vertices have been visited

        _DFS(v); // start search at vertex 0
        delete[ ] visited;

}

// Workhorse
void Graph::_DFS(const int v)
//  visit all previously unvisited vertices that are reachable from vertex v
{
        visited[v] = true;
        cout << v << ", ";
        ListIterator<int> li(HeadNodes[v]);
        if (!li.NotNull( )) return;
        int w = *li.First( );
        while (1) {
            if (!visited[w]) _DFS(w);
            if (li.NextNotNull( )) w = *li.Next( );
            else return;
        }
}

// The main function to construct MST using Kruskal's algorithm
void KruskalMST(struct Graph* graph)
{
        int V = graph->V;
        struct Edge result[20];  // Tnis will store the resultant MST
        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 subsets
        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(void)
{
	int select = 0, n, start, end, weight;
	int startBFSNode = 100;//the start node to BFS

	cout << "Input the total node number: ";
	cin >> n;

	/* Let us create following weighted graph */
	struct Graph graph(n);
	Graph* spanningTree = nullptr;
	while (select != '0')
	{
		cout << "\nSelect command 1: Add edges and Weight, 2: Display Adjacency Lists,
3: spanningTree, 4: Quit => ";
		cin >> select;
```

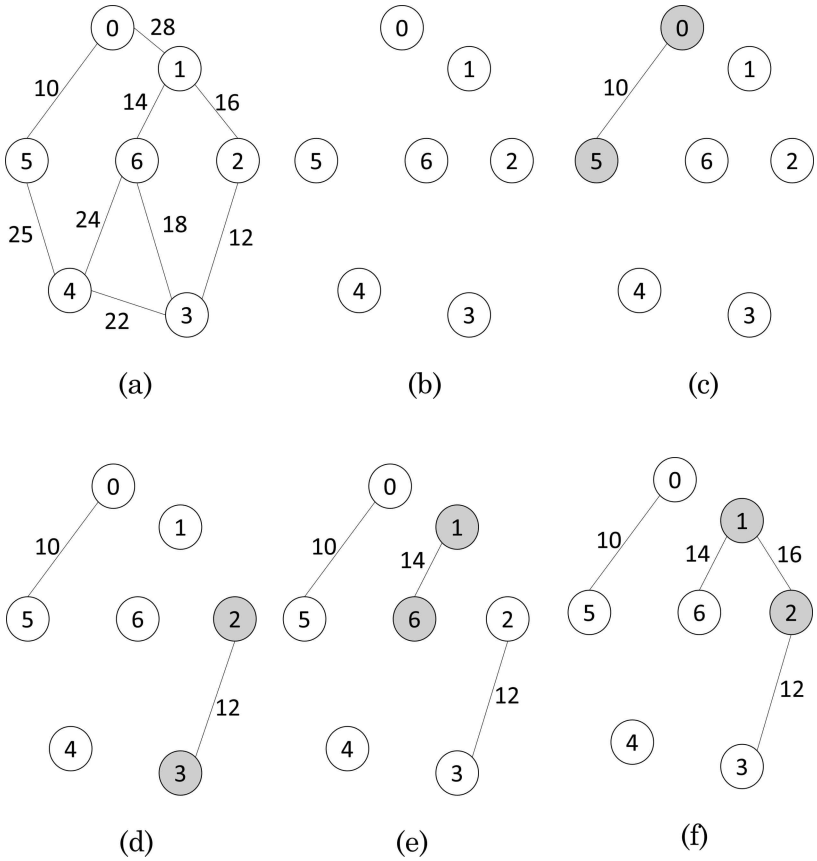
```
            switch (select) {
            case 1:
                  cout << "Add an edge: " << endl;
                  cout << "---------Input start node: ";
                  cin >> start;
                  cout << "---------Input  destination  node: ";
                  cin >> end;
                  if (start < 0 || start >= n || end < 0 || end >= n) {
                        cout << "the input node is out of bound." << endl;
                        break;
                  }
                  cout << "---------Input  weight: ";
                  cin >> weight;

                  graph.InsertVertex(start, end, weight);
                  break;
            case 2:
                  //display
                  graph.displayAdjacencyLists( );
                  break;
            case 3:
                  cout << "\nSpanningTree - Prim's algorithm: " << endl;
                  spanningTree = KruskalMST(&graph);
                  if (spanningTree) {
                        spanningTree->displayAdjacencyLists( );
                  }
                  delete spanningTree;
                  spanningTree = nullptr;
                  break;
            case 4:
                  exit(0);
                  break;
            default:
                  cout << "WRONG INPUT  " << endl;
                  cout << "Re-Enter" << endl;
                  break;
            }
      }

      KruskalMST(graph);
      system("pause");
      return 0;
}
```

6.4.1 Kruskal 알고리즘

minimum-cost spanning tree을 만들기 위해 그림 6.7처럼 한 번에 하나의 edge만을 선택하여 set T에 추가한다[1]. 이를 위한 먼저 edge들의 weight를 오름차순으로 sort하

고 edge가 cycle을 만들지 않으면 T에 추가한다. n vertices를 갖는 G에 대하여 n-1 edge가 포함되면 종료된다.

```
Algorithm Kruskal //[1]참조
begin
T = Ø
while ((T contains less than n−1 edges) && ( E not empty)) {
      choose an edge(v,w) from E of lowest cost;
      delete (v,w) from E;
      if ((v,w) does not create a cycle in T) add (v,w) to T;
      else discard (v,w);
}
if (T contains fewer than n−1 edges) cout <<"no spanning tree" <<endl;
end Kruskal
```



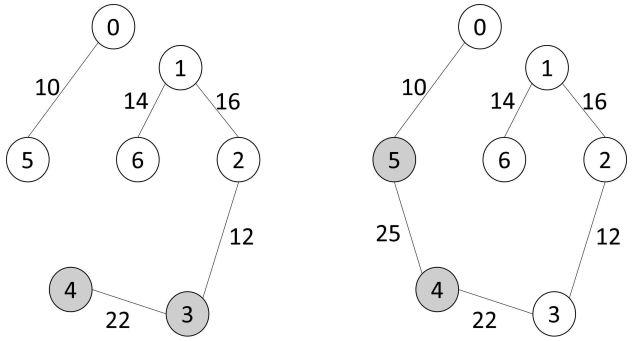


그림 6.7 Kruskal 알고리즘 적용 처리.

G = (V, E)에서 E의 lowest edge (v,w)를 선택하기 위하여 모든 edge를 minheap알고리즘으로 O(e log e)시간 복잡도로 sort한다. min heap은 next edge의 선택은 O(log e)으로 처리된다. (v,w)가 T에서 cycle을 만들지 않으면 T에 추가된다. T는 connected vertices로서 set으로 표현된다. v와 w가 T에서 connected이면 같은 set에 있게 된다.