Relazione progetto

Giovanni Palmeri

1 Strutture dati

Le strutture dati utilizzate sono principalmente 3, un **grafo non orientato**, una struttura **union-find** per la gestione di insiemi disgiunti ed una coda di priorità implementata tramite **heap**.

1.1 Heap

Si tratta di una coda di priorità nella sua verisione `minHeap`, quindi i nodi figli di un generico nodo v conterranno elementi di valore maggiore rispetto a quello contenuto nel padre. La struttura ad albero è stata implementata tramite un array `heap` con la seguente regola: il nodo padre del nodo in posizione `i` si trova in posizione `(i-1)/2`, conseguentemente i figli, rispettivamente, sinistro e destro, di un nodo in posizione `i` si troveranno in posizione `2*i+1` e `2*i+2`.

Listing 1: Heap

```
class Heap{
private:
    vector<pair<int, pair<int,
        int>>> heap;
}
```

1.1.1 Operazioni su Heap e loro complessità

Le operazioni più importati, quindi non analizziamo le funzioni che calcolano la posizone dei figli o del padre dei nodi poichè banali, sono `heapifyUp`, `heapifyDown`, `insert` e `extractMin`.

- `heapifyUp` e `insert`. `heapifyUp` serve per ristabilire la struttura del `min heap` in seguito ad un inserimento, questo perchè per inserire un nuovo elemento dobbiamo collocarlo nell'ultima posizione dell'array, che nella rappresentazione ad albero dell'`heap` corrisponderebbe ad essere inserito come foglia del cammino più a destra dell'albero, a questo punto dobbiamo trovare la corretta posizione all'interno dell'array effettuando degli scambi con il nodo padre, come descritto dal codice sottostante. La complessità di questa operazione è $O(h)$ dove h è l'altezza dell'albero, questo perchè con il massimo numero di scambi possibili tra padre e figlio porteremmo il

nodo da inserire fino alla radice e quindi avremmo percorso l'intera altezza dell'albero. Per quanto riguarda l'operazione di `insert` abbiamo capito che si basa sull'operazione `heapifyUp` pertanto anche la sua complessità sarà $O(h)$.

Listing 2: HeapifyUp

```
void Heap::heapifyUp(int ind){
    while (ind > 0 && heap[parent(ind)].first >
        heap[ind].first){
        swap(heap[ind], heap[parent(ind)]);
        ind = parent(ind);
    }
}
```

Listing 3: insert

```
void Heap::insert(pair<int, pair<int, int>> val) {
    heap.push_back(val);
    heapifyUp(heap.size() - 1);
}
```

- `heapifyDown` e `extractMin`. Anche questa operazione serve per ristabilire la struttura dell'`heap` questa volta però dopo che si è effettuata un'operazione di `extractMin`. Dovendo sia restituire che eliminare il minimo dall'`heap` dobbiamo eseguire i seguenti passi: salvare il minimo, che sappiamo essere contenuto nella radice (dato che si tratta di un `min heap`), e successivamente mettere al suo posto il contenuto di una foglia in particolare prendiamo l'elemento contenuto in ultima posizione nell'array, a questo punto non ci resta che eliminare tale foglia e ripristinare la struttura `min heap` facendo degli scambi tra padre e figlio come descritto dal codice. Anche in questo caso la complessità è $O(h)$ dato che al massimo dovremmo effettuare una quantità di scambi tale che porterebbe la nuova radice da tale a foglia, abbiamo inoltre due confronti per trovare il figlio con chiave minima che però prendono tempo lineare ($O(1)$).

Listing 4: heapifyDown

```
void Heap::heapifyDown(int
    ind){
    int small = ind;
    int left = left_son(ind);
    int right =
        right_son(ind);

    if (left < heap.size()
        && heap[left].first
        < heap[small].first)
        small = left;
    if (right < heap.size()
        && heap[right].first
        < heap[small].first)
        small = right;
    if (small != ind) {
        swap(heap[ind],
            heap[small]);
        heapifyDown(small);
    }
}
```

Listing 5: extractMin

```
pair<int, pair<int, int>>
    Heap::extractMin() {
    if (heap.empty()) throw
        out_of_range("Heap
        is empty");
    pair<int, pair<int,
        int>> min = heap[0];
    heap[0] = heap.back();
    heap.pop_back();
    heapifyDown(0);
    return min;
}
```

- **Altezza albero**. Sappiamo che un `heap binario` è un albero binario quasi completo, cioè completo fino al penultimo livello, con una relazione d'ordine tra i suoi nodi dipendente dalla versione di `heap`, `min heap` o `max heap`. Per queste ragioni fino al penultimo livello, la sua altezza è calcolabile come facciamo per gli alberi binari ovvero:

$$n = 1 + 2 + 2^2 + \cdots + 2^{h-1} + 2^h = \frac{2^{h+1} - 1}{2 - 1} = 2^{h+1} - 1 \tag{1}$$

dove n è il numero di nodi dell'albero binario di altezza h. Pertanto avendo un albero binario completo fino al penultimo livello avremo un numero di nodi pari a

$$2^k - 1 < n \leq 2^{h+1} - 1 \tag{2}$$

il che implica

$$h = \theta(\log n) \tag{3}$$

1.2 Union-Find

La struttura **Union-Find** è stata realizzata nella sua versione **Quick-Union**, in particolare si tratta di un semplice vettore `uf` di interi ove il contenuto di `uf[i]` corrisponde al rappresentante dell'insieme di cui fa parte `i`.

Listing 6: Union-Find

```
class QuickUnion {
private:
    std::vector<int> uf;
    std::vector<int> sz;
}
```

1.2.1 Operazioni su Union-Find e loro complessità

Le operazioni che possiamo effettuare su questa struttura sono le classiche `find` e `merge`, che corrisponde alla `union`, ed in aggiunta è presente l'inizializzazione del vettore `uf`.

- `Find`. Trattandosi della variante `QuickUnion` questa operazione prende tempo proporzionale all'altezza dell'albero rappresentante l'insieme come mostrato dalla figura 1. Pertanto avremo una complessità pari a $O(\log n)$, il perchè verrà speigato quando si parlerà dell'operazione `Merge`.

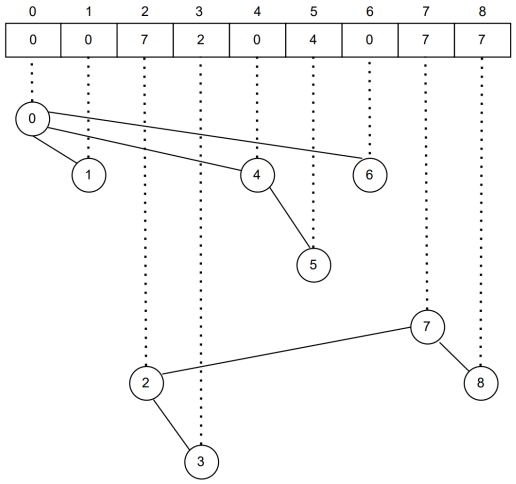

Figure 1: Union-Find Tree

Listing 7: Find

```
int QuickUnion::find(int val){
    while(uf[val] != val) val = uf[val];
    return val;
}
```

- `Connected`. Questa funzione controlla se due elementi sono contenuti nello stesso insieme, per farlo chiama l'operazione `find` su entrambi gli elementi e poi ne confronta il risultato, pertanto la sua complessità è la stessa di quella della funzione `find` ovvero $O(\log n)$.

Listing 8: Connected

```
bool QuickUnion::connected(int el1, int el2){
    int root1 = find(el1);
    int root2 = find(el2);

    return root1 == root2;
}
```

- `Costruttore`. La costruzione degli array `uf` e `sz` avviene in tempo $O(n)$, dove n è il numero di elementi che stiamo inserendo nella struttura `Union-Find`.

Listing 9: Costruttore

```
    QuickUnion::QuickUnion(int n){
        sz.resize(n);
        uf.resize(n);
        for (size_t i = 0; i < n; i++){
            uf[i] = i;
            sz[i] = 1;
        }
    }
```

- `Merge`. Si basa sulla comeplessità dell'operazione `find` dato che le altre operazioni sono degli accessi all'array `uf` e `sz` che prendono tempo pari a $O(1)$, e 2 confronti oltre ai vari aggiornamenti dei valori degli array precedentemente specificati che prendono tempo pari a $O(1)$ anch'essi, in totale quindi avremo una complessità pari a $O(\log n)$. Il motivo per il quale l'altezza dell'albero, e quindi la complessità dell'operazione `find`, è $O(\log n)$ è il seguente:

  – utilizzando la versione `weighted quick-union`, aggiungendo quindi alla classica struttura `quick-union` un array `sz` per immagazzianare la quantità di nodi contenuti nell'albero radicato in `sz[i]` (radice inclusa), avremo che quando andremo a fare la `union` di 2 alberi, la radice dell'albero di peso minore diventerà figlia della radice dell'albero di peso maggiore, in questo modo tutti i nodi dell'albero di peso minore incrementeranno la loro profondità di 1, mentre i nodi dell'albero di peso maggiore manterranno inalterato questo dato. Da ciò deduciamo che la profondità di un nodo x, appartenente ad un albero T_1, incrementa di 1 quando su tale albero T_1 viene eseguita l'operazione di `union` con un altro albero T_2 la cui `size` è $|T_2| > |T_1|$. Osserviamo che per ogni `union` che facciamo la `size` dell'albero contenente

x risulterà, come minimo, doppia rispetta a quella di partenza, pertanto l'incremento della profondità del nodo x e conseguentemente dell'altezza dell'albero, è uguale al numero di volte che siamo in grado raddoppiare, tramite l'operazione `union`, la `size` dell'albero T_1 contenente x. Partendo da uno stato in cui tutti gli insiemi della struttura `union-find` contengono un solo elemento, possiamo notare come siamo in grado di raddoppiare la `size` di un insieme, e quindi dell'albero che lo rappresenta, al massimo $O(\log n)$ volte (fino ad ottenere quindi un unico insieme contenete tutti gli n elementi), dove n è il numero di elementi contenuti in tutti gli insiemi. Indicando con k il numero massimo di raddoppiamenti che possiamo fare, possiamo scrivere la seguente relazione:

$$n = 2^k \tag{4}$$

pertanto avremo che il numero massimo di `union`, e quindi anche l'altezza che l'albero può raggiungere, è pari a $O(\log n)$

Listing 10: Merge

```
void QuickUnion::merge(int s1, int s2){
    int root_s1 = find(s1);
    int root_s2 = find(s2);

    if (root_s1 == root_s2) return;
    if (sz[root_s1] < sz[root_s2]){
        uf[root_s1] = root_s2;
        sz[root_s2] += sz[root_s1];
    }else{
        uf[root_s2] = root_s1;
        sz[root_s1] += sz[root_s2];
    }
}
```

.

1.3 Grafo

Il grafo è composto da una variabile V contenente il numero di vertici del grafo e da una **lista di adiacenza** contente per ogni vertice la lista degli archi uscenti da esso nel formato $\{peso\ dell'arco, vertice\}$. Come possiamo vedere dal codice sottostante la lista di adiacenza `adjList` è implementata come una mappa che ha per chiavi gli interi che indentificano i vertici e per valore un array di `Node`, quest'ultima è una struttura creata appositamente per rapprensentare i vertici e i suoi archi uscenti, come possiamo notare infatti sono presenti due campi, uno contenente il valore del vertice ed un altro contenente una lista di puntatori a `Node` la quale rappresenta l'insieme degli archi uscenti dal vertice in questione,

come ho già detto tali archi sono presenti nel formato {*peso dell'arco*, *vertice*} in accordo al type `ad`.

Listing 11: Grafo

```
class Graph {
private:
    int V;
    map<int, Node*> adjList;
}
```

Listing 12: Node

```
struct Node
{
    int data;
    typedef pair<int, Node*> ad;
    list<ad>* lis;

    Node(int data) : data{ data
        }, lis{ new list<ad> } {}
};
```

1.3.1 Operazioni sui grafi e loro complessità

Le operazioni effettuate sulla struttra **Graph** sono `populateGraph(int N)`, la quale inizializza il grafo con un numero di vertici passati a parametro, `copyGraph()` che crea una copia di un grafo, `Kruskal()` che implementa l'algoritmo di Kruskal, `SMST()` che calcola il *Second Minimum Spanning Tree*, `addEdge(int source, int dest, int w)` che aggiunge un arco al grafo specificando il nodo di partenza, il nodo di arrivo ed il peso dell'arco, `removeEdge(int source, int dest, int w)` che rimuove dal grafo l'arco che corrisponde ai valori specificati nei paramentri della funzione e infine `copyeEdge(int source, int dest, int w)` che copia un singolo arco, identificato dai valori passati a paramentro della funzione, nel grafo su cui è chiamata la funzione.

- `populateGraph`. Come possiamo vedere dal codice sottostante consiste di un ciclo `for` che prende tempo pari a $O(N)$, dove n è il numero di nodi da creare, quindi $O(|V|)$.

Listing 13: populateGraph

```
void Graph::populateGraph(int N) {
    for (int i = 1; i <= N; i++) adjList.insert(make_pair(i, new
        Node(i)));
}
```

- `addEdge`. Essendo un grafo non orientato se aggiungiamo un arco da una vertice v_1 ad un vertice v_2 di peso w, allora dovremmo creare anche un arco che va da v_2 a v_1 di peso w. Osserviamo che, per quanto riguarda la complessità di questa operazione, abbiamo 2 accessi diretti alla mappa `adjList` per ottenere le liste di adiacenza dei due vertici coinvolti nell'arco che vogliamo aggiungere, quindi abbiamo una complessità $O(1)$ dato che si tratta di accessi diretti. Inoltre dobbiamo aggiungere alle liste di adiacenza dei due vertici il nuovo arco, per farlo usiamo l'operazione `push_back` passandogli a parametro il peso dell'arco e il vertici di arrivo, sappiamo che

`push_back` eseguito su una lista, aggiunge alla fine di essa ciò che passiamo a paremetro della funzione stessa, pertanto si tratta di un aggiornamento di puntatori che prende tempo $O(1)$. Pertanto il costo complessivo di `addEdge` è $O(1)$.

Listing 14: addEdge

```
void Graph::addEdge(int source, int dest, int w) {
    Node* s = adjList[source];
    Node* d = adjList[dest];

    s->lis->push_back({w, d});
    d->lis->push_back({w, s});
}
```

- `CopyEdge`. Questa funzione serve a copiare un singolo arco, quindi se abbiamo un arco da v_1 a v_2 con peso w, tale arco verrà aggiunto al grafo su cui viene chiamata la funzione senza creare l'arco da v_1 a v_2 di peso w come succede per la funzione `addEdge`. Le operazioni eseguite sono identiche a quelle che troviamo in `addEdge` con la differenza che aggiungiamo il nuovo arco solamente alla lista di adiacenza del vertice di partenza dell'arco stesso, quindi anche qui avremo una complessità totale di $O(1)$.

Listing 15: copyEdge

```
void Graph::copyEdge(int source, int dest, int w){
    Node* s = adjList[source];
    Node* d = adjList[dest];

    s->lis->push_back({w, d});
}
```

- `removeEdge`. Come conseguenza del fatto che il grafo è non orientato e che quindi per ogni arco (v_1, v_2) esiste un arco (v_2, v_1), quando vogliamo eliminare un arco (v_1, v_2) dobbiamo eliminare anche l'arco (v_2, v_1). Per poter fare ciò dobbiamo ottenere le liste di adiacenza dei due vertici v_1 e v_2, come abbiamo visto prima questo possiamo farlo in $O(1)$ accedendo direttamente alla mappa contente tali liste, successivamente scorriamo tali liste alla ricerca dell'arco corrispondente ai valori passati a paramentro della funzione, la complessità di tale operazione si basa sulla lunghezza delle liste di adiacenza che nel caso peggiore, il quale si verifica quando l'arco da noi cercato si trova in coda alla lista, la lunghezza della lista di adiacenza, e quindi anche la complessità della funzione `removeEdge`, sarà uguale a $O(deg(v))$, ovvero al degree del vertice di partenza v_1, in generale, al massimo ogni vertice può avere $|V|-1$ archi, nel caso specifico del problema che dobbiamo risolvere però possono esserci molteplici archi tra 2 nodi, distinguibili solamente dal peso di questi, pertanto possiamo

affermare che la complessità di questa funzione nel caso pessimo è $O(|E|)$.

Listing 16: removeEdge

```
void Graph::removeEdge(int source, int dest, int w){
    Node* ad = adjList[source];
    for (auto it = ad->lis->begin(); it != ad->lis->end(); ++it){
        if (it->second->data == dest && it->first == w){
            ad->lis->erase(it);
            break;
        }
    }

    Node* adDest = adjList[dest];;
    for (auto it = adDest->lis->begin(); it != adDest->lis->end();
         ++it) {
        if (it->second->data == source && it->first == w) {
            adDest->lis->erase(it);
            break;
        }
    }
}
```

- `copyGraph`. Per copiare un grafo dobbiamo, prima di tutto, inizializzare un nuovo grafo utilizzando la funzione `populateGraph` che abbiamo visto avere complessità pari a $O(|V|)$, successivamente si scorrono, tramite il primo ciclo `for`, gli elementi della lista di adiacenza del grafo da cui dobbiamo copiare gli archi e i vertici in modo da inizializzare la lista di adiacenza del grafo in cui andremo ad effettuare la copia (`copiedG`) inserendo appunto tali vertici, pertanto dovendo scorrere `adjList`, che contiene $|V|$ vertici avremo che la complessità di questo ciclo è $O(|V|)$. A questo punto non ci resta che inserire gli archi, per farlo utilizziamo un secondo ciclo `for` che scorre per ogni vertice v all'interno di `adjList`, la sua lista di adiacenza, eseguendo su ogni arco incotrato la funzione `copyEdge`, sappiamo che quest'ultima operazione ha complessità pari a $O(1)$ mentre il ciclo ha complessità $O(|E|)$ se il grafo originale ha $|E|$ archi. In totale avremo qundi una complessità pari a $O(|V|)$ (inizializzazione del nuovo grafo) + $O(|V|) + O(|E|)$ (allocazioni di memoria ed inserimento dei vertici + iterezione sugli archi) = $O(|V| + |E|)$

Listing 17: copyGraph

```
Graph Graph::copyGraph(){
    Graph copiedG(V);
    for (auto pair : adjList){
        int node = pair.first;
        Node* newNode = new Node(node);
        copiedG.adjList[node] = newNode;
        for (auto edge : *(pair.second->lis)){
```

```
            int des = edge.second->data;
            int w = edge.first;
            copiedG.copyEdge(node, des, w);
        }
    }
    return copiedG;
}
```

- `Kruskal`. Per prima cosa si inizializza una coda di priorità heap con tutti gli archi basando la prorità di questa sui pesi degli stessi, sappiamo che la complessità del metodo `insert` della struttura `heap` è pari a $O(\log n)$ dove n è il numero degli elementi già presenti nella struttura, nel nostro caso quindi $O(\log |E|)$, tale operazione deve essere eseguita per ogni arco quindi in totale avremo $O(|E| \log |E|)$. A questo punto creiamo una struttura `Union-Find`, sappiamo che la complessità dell'inizializzazione di tale struttura è $O(|V|)$, dove mantere i vertici, dopodichè si entra nel ciclo `while` che estrarrà ripetutamente il minimo dalla coda di priorità, ovvero l'arco di costo minore, considerando che nel peggiore dei casi dovremmo estrarre tutti gli elementi dall'`heap` abbiamo che la complessità di questo blocco di codice sarà $O(|E| \log |E|)$ mentre nel caso migliore $O(|V| \log |E|)$, per ogniugno di essi controlliamo se fanno parte dello stesso insieme nella struttura `Union-Find` tramite la funzione `connected` che ha complessità $O(\log |V|)$, in base al risultato di tale funzione verrà o meno fatto il `merge` dei due vertici interessati dall'arco che stiamo esaminando, anche in questo caso la complessità è $O(\log |V|)$. In totale quindi avremo $O(|E| \log |E|)$ (inserimento degli archi nella coda di priorità) + $|V|$ (inizializzazione della struttura `union-find`) + $|E| \cdot (\log |E| + \log |V| + \log |V|)$ (numero di volte che il ciclo viene eseguito, moltiplicato per, rispettivamente, `extractMin`, `connected`, `merge`) = $O(|E| \log |E|)$.

Listing 18: Kruskal

```
pair<int, Graph> Graph::Kruskal() {
    Heap hp;
    for (auto node : adjList) {
        if (node.second->lis != nullptr) {
            for (auto adj : *(node.second->lis)) {
                hp.insert({ adj.first, { node.first,
                    adj.second->data } });
            }
        }
    }

    QuickUnion qu(V);
    Graph MST(V);

    int cost_MST = 0;
    int numEdges = 0;
```

```
    pair<int, pair<int, int>> edge;

    while (numEdges < V - 1 && !hp.isEmpty()){
        edge = hp.extractMin();
        int source = edge.second.first;
        int destination = edge.second.second;

        if (!qu.connected(source - 1, destination - 1)) {
            MST.addEdge(source, destination, edge.first);
            qu.merge(source - 1, destination - 1);
            cost_MST += edge.first;
            numEdges++;
        }
    }

    if (numEdges == V - 1) return {cost_MST, MST};
    else return {-1, MST};
}
```

- **Second Minimum Spanning Tree**. Il **SMST** di un grafo G è un albero ricoprente avente costo minimo tra tutti i possibili spanning tree di G escluso il minimum spanning tree. Il **SMST** differisce dal **MST** per un solo arco, la dimostrazione di tale affermazione verrà esposta successivamente, pertanto un metodo per poter calcolare il second minimum spanning tree è quello di calcolare inizialmente il MST e scegliere un arco alla volta da esso che verrà eliminato dal grafo G sul quale eseguiremo nuovamente l'algoritmo per calcolare il minimum spanning tree, avendo però questa volta un arco in meno. Tra tutti gli spanning tree che genereremo ci interessa il minimo il quale corrisponderà al SMST, in pratica stiamo cercando qual è l'arco di differenza tra il SMST e il MST.

Listing 19: Second minum spanning tree

```
void Graph::SMST(){
    pair<int, Graph> MST = Kruskal();
    int mincost = MST.first;
    if (mincost == -1){
        cout << "Nessuna soluzione" << endl;
        return;
        }
    Graph copiedG = copyGraph();

    int minco = INT_MAX;
    for (auto pair : MST.second.adjList){
        if (pair.second->lis != nullptr){
            for (auto adj : *(pair.second->lis)){
                int src = pair.first;
                int dest = adj.second->data;
                int w = adj.first;
```

```
                copiedG.removeEdge(src, dest, w);
                int cost = copiedG.Kruskal().first;
                if (cost != -1) minco = min(minco, cost);
                copiedG.addEdge(src, dest, w);
            }
        }
    }

    if (minco == INT_MAX) cout << "Nessun secondo miglior costo" <<
        endl;
    else cout << minco <<endl;
}
```

Come possiamo vedere abbiamo inzialmente l'esecuzione dell'algoritmo di `Kruskal` e di `copyGraph` per effettuare una copia del grafo G, la complessità di tali operazioni sappiamo essere pari a $O(|E|\log|E|)$ e $O(|E|+|V|)$, il contenuto del ciclo `for` più interno esegue l'eliminazione dell'arco scansionato tramite `removeEdge`, l'esecuzione dell'algoritmo di `kruskal` sul nuovo grafo con un arco in meno ed infine il ripristino dell'arco all'interno del grafo tramite l'operazione `addEdge`, queste operazioni hanno complessità, rispettivamente $O(|E|)$, $O(|E|\log|E|)$ e $O(1)$, il resto sono dei confronti che prendono tempo $O(1)$ quindi non vengono considerate nell'analisi. Le ultime 3 operazioni vengono eseguite per ogni arco presente nel **MST** quindi $2V-2$ volte, pertanto la complessità totale dell'algoritmo risulta essere

$$|E|\log|E|+|E|+|V|+(V)*(|E|+|E|\log|E|)=O(|V||E|\log|E|) \quad (5)$$

- **Correttezza algoritmo**. Per dimostrare che l'algoritmo calcola effettivamente il second minimum spanning tree ci basta dimostrare che, come dicevamo prima, il SMST differisca dal MST per un solo arco, per fare ciò dimostreremo che se rimpiazziamo 2 o più archi del MST allora non riusciremo ad ottenere un SMST. Sia quindi T un minimum spanning tree di un grafo G, e supponiamo che esista un second minimum spanning tree T' che differisce da T per 2 o più archi, ciò significa che nell'insieme $T-T'$ ci saranno almeno 2 archi. Sia (u,v) l'arco di peso minore tra tutti quelli in $T-T'$, per definizione di spanning tree, se aggiungessimo tale arco a T' otterremmo un ciclo c, indichiamo con (x,y) l'arco che chiude il ciclo. Possiamo affermare che $w(x,y)>w(u,v)$, se infatti per assurdo assumiamo che $w(x,y)<w(u,v)$, aggiungendo (x,y) a T otterremmo un ciclo c' il quale conterrà un arco (u',v') anch'esso in $T-T'$. L'insieme $T''=T-\{(u',v')\}\cup\{(x,y)\}$ formerà uno spanning tree nel quale dovremmo avere $w(u',v')<w(x,y)$ altriementi T'' sarebbe uno spanning tree di peso minore di T (il minum spanning tree), pertanto avremo che $w(u',v')<w(x,y)<w(u,v)$ che contraddice

la scelta di (u, v) come arco in $T - T'$ di peso minimo. Appurato che $w(x, y) > w(u, v)$ possiamo continuare dicendo che l'insieme di archi $T' - \{(x, y)\} \cup \{(u, v)\}$ è uno spanning tree ed il suo peso è minore di $w(T')$ ma comunque diverso da $w(T)$, pertanto T' non era un second-best minimum spanning tree.

2 Conclusioni

In definitiva per poter risolvere il problema assengato dobbiamo eseguire la funzione `SMST` su ogni grafo fornito in input, a ciò sommiamo i costi computazionali derivanti dalla costruzione di questi grafi. Per ogniuno di essi dobbiamo infatti eseguire `populateGraph` per inserire i vertici, e `addEdge` per poter inserire gli archi. Se indichiamo con `numGraphs` il numero di grafi contenuti nel file di input, la complessità totale del codice per risolvere, risulterà uguale a $O(\texttt{numGraphs} \cdot (|V| + |E| + |E||V| \log |E|)) = O(\texttt{numGraph}(|E||V| \log |E|))$.