# Albero di copertura Minimo

Bellomo Andrea

A.A. 2020/2021


## 1 Albero di copertura Minimo

### 1.1 Definizioni utili

### 1.2 Teorema fondamentale

## 2 Algoritmi per il calcolo di un MST

### 2.1 Algoritmo di Kruskal

### 2.2 Algoritmo di Prim

## 3 Implementazione

### 3.1 Ulteriori scelte implementative

### 3.2 Implementazione di Kruskal

### 3.3 Implementazione di Prim

### 3.4 Diagramma UML Alg. Kruskal

### 3.5 Diagramma UML Alg. Prim

## 4 Fonti e Approfondimenti

# 1. Albero di copertura Minimo

Un classico problema di ottimizzazione su grafi consiste nel determinare un albero di copertura di peso minimo. Uno spanning tree di un grafo G è un sottografo di G che contiene tutti i vertici del grafo , ed è un albero. Da un grafo possono essere creati diversi spanning trees. Supponiamo che gli archi del grafo abbiano un costo associato (peso). Il costo di uno spanning tree sarà dato dalla somma dei costi dei suoi archi. Chiaramente, spanning tree diversi avranno costi diversi.

Uno dei problemi posti è quello di trovare lo spanning tree di costo minimo. Questo è il Minimum Spanning Tree (MST). Una classica applicazione del problema è il progetto di reti telefoniche o di calcolatori. Si deve mantenere un collegamento con vari uffici, e devono essere affittate un certo numero di linee telefoniche per garantire il collegamento. I costi delle linee sono stabiliti dalla compagnia telefonica e ovviamente dipendono dalle città che devono essere connesse. L'obbiettivo è affittare un insieme di linee telefoniche che connettono tutti gli uffici col costo totale minimo.

Possiamo modellare questo problema tramite un grafo connesso non orientato e pesato.Tale problema può quindi essere risolto trovando il Minumun Spanning Tree del grafo.

**Definizione 1.** Dato un grafo G = (V, E) non orientato e connesso con una una funzione peso w : E $\rightarrow$ R, un albero di copertura minimo di G è un sottoinsieme $T \subset E$ tale che la somma dei pesi degli archi che compongono T sia :

$$\text{w(T)} = \sum e \in T \text{ w(e) minima.}$$

Un albero di copertura è quindi un sottografo di G che contiene tutti i suoi vertici ed è sia aciclico che connesso, e inoltre $|T| = |V| - 1$ .

## 1.1 Definizioni Utili

**Definizione 2.** Un taglio (S, V \S ) di un grafo non orientato G = (V, E) è una partizione di V.

**Definizione 3.** Un arco attraversa il taglio (S, V \S) se uno dei suoi estremi si trova in S e l'altro in V\S.

**Definizione 4.** Un taglio rispetta un insieme A di archi se nessun arco di A attraversa il taglio.

**Definizione 5.** Un arco che attraversa un taglio si dice leggero se ha peso pari al minimo tra i pesi di tutti gli archi che attraversano tale taglio.

**Definizione 6.** Sia A sottoinsieme di un qualche albero di copertura minimo; un arco si dice sicuro per A se può essere aggiunto ad A e quest'ultimo continua ad essere sottoinsieme di un albero di copertura minimo.

## 1.2 Teorema Fondamentale

**Teorema 1 (fondamentale).** Sia G = (V, E, w) un grafo non orientato, connesso e pesato; sia inoltre:

- $A \in E$ contenuto in qualche albero di copertura minimo;
- (S, V \S) un taglio che rispetta A
- $(u, v) \in E$ arco leggero che attraversa il taglio.

Allora (u, v) è sicuro per A.

**Corollario 1.** Sia G = (V, E, w) un grafo non orientato, connesso e pesato; siano:

- $A \in E$ contenuto in un albero di copertura minimo;
- C componente connessa della foresta GA = (V, A);
- $(u, v) \in E$ arco leggero che connette C ad un' altra componente connessa di GA .

Allora (u, v) è sicuro per A.

# 2. Algoritmi per il calcolo di un MST

L'algoritmo generico per la costruzione di un albero di copertura minimo è di tipo greedy. Dato un grafo G = (V, E) non orientato e connesso con una una funzione peso w : E $\rightarrow$ R, l'algoritmo agisce su un insieme $A \in T$ (con T MST di G) al quale ad ogni passo viene aggiunto un arco (u,w) che permette ad A di restare un sottoinsieme del MST finale, tale arco verrà chiamato arco sicuro.

**- Pseudo-codifica**

GENERIC$_M ST()$

1. A$\leftarrow$0
2. **While** A != T do
3. (u,v)$\leftarrow$ArcoSicuro(A)

4. **return** A

**Descrizione**
Dopo la riga 1. L' insieme A soddisfa banalmente l'invariante di ciclo. Il ciclo nel blocco 2 aggiunge solo archi sicuri e quindi conserva l'invariante. Tutti gli archi aggiunti ad A si trovano in un albero di connessione minimo, quindi l'insieme A restituito nella riga 5 deve essere un albero di connessione minimo. I due algoritmi per gli alberi di connessione minimi (Kruskal e Prim) sono elaborazioni del metodo generico.

## 2.1 Algoritmo di Kruskal

L'algoritmo sfrutta il Corollario 1 e mantiene, istante per istante, una foresta contenuta in qualche MST. Per poter gestire gli insiemi disgiunti che rappresentano le varie componenti connesse della foresta, occorre usare una struttura dati appropriata. La tecnica utilizzata consiste nel partizionare l'insieme V in k gruppi tali che:

1. 1<i<k Vi = V
2. i != j  Vi ∩ Vj = 0

rappresentando ogni gruppo con un elemento del gruppo stesso. Devono essere permesse le seguenti operazioni:

1. **makeSet (Elemento a)** Crea un insieme formato da un solo elemento, a, ed associa all'insieme l'etichetta ' a'.
2. **union (Insieme A, Insieme B)** Unisce gli elementi degli insiemi A e B in un unico insieme che etichetta con 'A'. I vecchi insiemi A e B saranno cancellati.
3. **find (Elemento a)** Restituisce l'etichetta dell'insieme contenente l'elemento a.

**- Pseudo-codifica**

KRUSKAL MST()

1. A←0
2. per ogni u ∈ V do
3. **makeSet(u)**
4. ordina gli archi di E in modo non decrescente rispetto al peso
5. per ogni (u, v) ∈ E do
6. **if**( **findSet (u)** != **findSet (v)**) **then**

7. A = A ∪ (u, v)
8. **union(u , v)**
9. **return** A

**Descrizione**

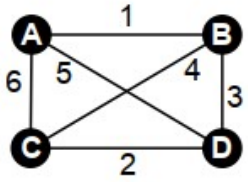
Le righe 1-3 inizializzano l' insieme A come un insieme vuoto e creano V alberi, uno per ogni vertice. Il ciclo ,esamina gli archi ordinati in ordine non decrescente, e controlla per ogni arco(u,v), se le estremità u e v appartengono allo stesso albero; in caso affermativo, l' arco (u,v) non può essere aggiunto senza generare un ciclo, quindi l'arco viene scartato.Nel caso in cui i due vertici appartengono ad alberi differenti, l arco (u,v) viene aggiunto ad A nella riga 7 e i vertici dei due alberi vengono fusi nella riga 8.

**Complessità**

Il costo computazionale dell'algoritmo è nel caso peggiore **O(V*E)** dove E è il numero di archi ed V il numero di vertici. È possibile diminuire il costo computazionale dell'algoritmo sino ad ottenere **O(log(V)* E)**

**Esempio Svolgimento Algoritmo**

Consideriamo il seguente grafo :

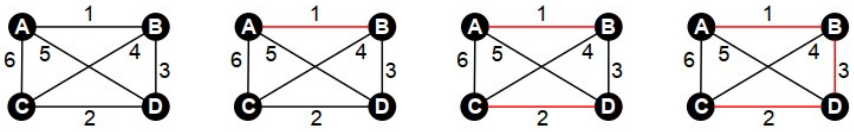



Gli archi sono ordinati in modo non decrescente rispetto al peso : AB, CD, BD, BC, AD, AC. Inizialmente tutti i nodi sono isolati. Consideriamo nell' ordine gli archi che vengono selezionati:

1. AB: poiché i nodi A e B non sono ancora collegati, l'arco viene selezionato;
2. CD: i nodi C e D non sono nella stessa componente, dunque l'arco viene selezionato;
3. BD: i nodi B e D non sono nella stessa componente, quindi l'arco viene scelto (il MST è ora pronto);
4. BC: i nodi B e C stanno già nella stessa componente, dunque l'arco non viene selezionato;

5. AD: i nodi A e D stanno già nella stessa componente, perciò l'arco viene scartato;

6. AC: i nodi A e C stanno già nella stessa componente, quindi l'arco non viene scelto.




## 2.2 Algoritmo di Prim

L'algoritmo di Prim è fondato direttamente sul Teorema 1 Tale algoritmo mantiene, istante per istante, un albero contenuto in qualche MST. L'algoritmo, inoltre, richiede in ingresso un nodo s ∈ V che sarà la radice dell'albero di copertura. Viene utilizzata una struttura dati del tipo coda a min-priorità per gestire l'insieme dei vertici Q non ancora inclusi nell'albero di copertura. L'insieme v\Q contiene i vertici già inseriti nell'albero.
Per ciascun vertice u ∈ Q vengono tenuti aggiornati i seguenti attributi:

1. **key[u]** è la chiave di priorità: se u è adiacente a un vertice dell'albero V\Q il valore della chiave key[u] è pari al minimo tra i pesi di tutti gli archi che collegano u ai nodi dell'albero V\Q, altrimenti il valore di key[u] è infinito.

2. **p[u]** memorizza il predecessore di u nell'albero generato dall'algoritmo.

Si può ricostruire l'albero finale utilizzando i nodi e le informazioni sui predecessori:

$$A = \left\{ (u, p[u]) \in E : u \in (V \backslash s) \right\}$$

**- Pseudo-codifica**

PRIM MST()

1. Q: coda di priorita contenente tutti i vertici in V
2. per ogni u ∈ Q do
3. key[u] = inf
4. key[s] = 0
5. p[s] = NIL
6. while (Q != 0) do

7. u = exstractMin(Q)
8. per ogni v ∈ Adj[u] do
9. if(v ∈ Q  w(u, v) ¡ key[v]) then
10. key[v] = w(u,v)
11. p[v] = u
12. **return** $A = \left\{ (u, p[u]) \in E : u \in (V \setminus s) \right\}$

**Descrizione**
Le righe 1-5 impostano la chiave di ciascun vertice a infinito (ad eccezione della redice s la cui chiave è impostata a 0 ; in questo modo la radice sarà il primo nodo ad essere elaborato), assegnano al padre di ciascun vertice il valore NIL e inizializzano la coda di min priorità Q in modo che contenga tutti i vertici. L'algoritmo conserva la seguente inviriante di ciclo composta da tre parti :

1. $A = \left\{ (v, p[v]) : v \in (V - \left\{ s \right\} - Q) \right\}$.
2. I vertici già inseriti nell' albero di connessione minimo sono quelli che appartengono a V - Q.
3. Per ogni vertice v ∈ Q , se p[v] != NIL, allora key[v] < inf e key[v] è il peso di un arco leggero (v,p[v]) che collega v a qualche vertice che si trova già nell' albero di connessione minimo.

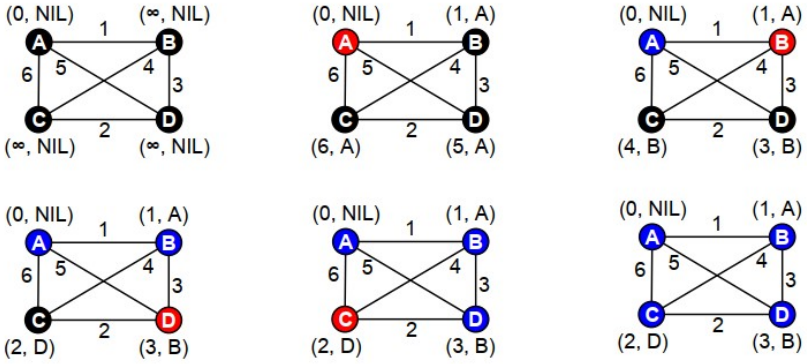
La riga 7 identifica un vertice u ∈ Q incidente su un arco leggero che attraversa il taglio (V - Q , Q )(ad accezzione della prima iterazione, nella quale u = r). Quando il vertice u viene eliminato dall' insieme Q , viene aggiunto all' insime V - Q dei vertici dell' albero, quindi (u, p[u]) viene aggiunto ad A. Il ciclo nelle righe 8 - 11 aggiorna il campo 'key' e 'p' di qualsiasi vertice di v adiacente ad u, ma che non appartiene all' MST. L' aggiornamento conserva la terza parte dell' invariante di ciclo.

**Complessità**

La complessità dell'algoritmo di Prim dipende dall'implementazione della struttura dati ausiliaria utilizzata per contenere i nodi. Se implementata con una coda di priorità e assumendo di complessità costante il test di presenza o meno nella coda il tempo totale di esecuzione dell'algoritmo è di O(V *log(V)+E*log(V)) dove log(V)*V è il tempo necessario per le operazioni di estrazione dalla coda ed E*log(V)è il tempo necessario per scorrere le liste delle adiacenze e compiere l'assegnazione.Quindi il tempo totale di esecuzione dell'algoritmo di Prim è **O(E*log(V))**.

**Esempio Svolgimento Algoritmo**

Per ciascun nodo, è indicata la coppia (chiave, predecessore) associata ad ogni iterazione del ciclo; i nodi marcati in nero sono quelli ancora presenti in Q, il nodo rosso è quello estratto nell'iterazione considerata e quelli blu sono i nodi già estratti da Q ed esaminati




L'albero, ricostruito a partire dall'ultima figura, è quello costituito dagli archi AB, BD e CD.

# 3. Implementazione

Si sceglie di utilizzare un approccio orientato agli oggetti, scomponendo il problema in due file, uno specializzato nell'algoritrmo di Kruskal e l'altro nell'algoritmo di Prim, questo tipo di approccio consente di rendere i due algoritmi indipendenti e riutilizzabili. Il singolo algoritmo è sviluppato utilizzando la tecnica di programmazione ad oggetti, rendendo la classi importabili in progetti esterni e riutilizzabili. I due algoritmi non sono implementati nella stessa classe grafo per mantenere le due logiche separate e non sovraccaricare la classe con comportamenti diversi, malgrado essi svolgono lo stesso compito . Questo tipo di approccio rende piu facile la lettura dei singoli codici.

## 3.1 Implementazione di Kruskal

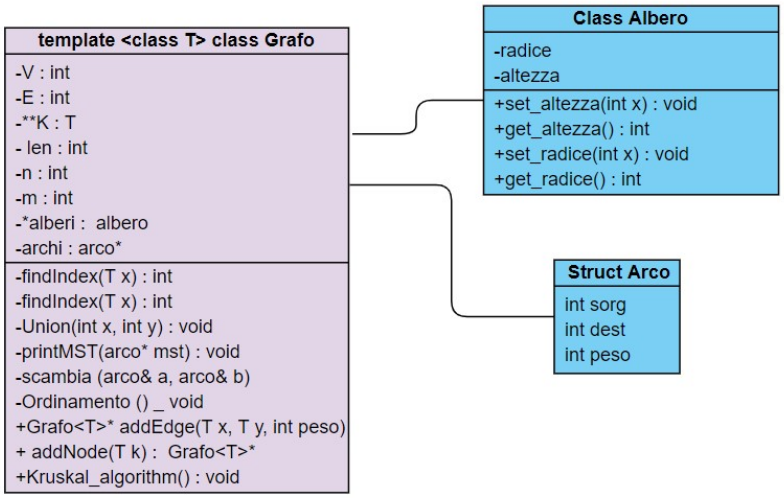
Si sceglie di utilizzare una struttura "arco", per via del funzionamento dell'algoritmo in maniera tale da mantenere il collegamento tra due nodi e il suo peso.
Viene implementata la classe "albero" per mantenere gli insiemi di nodi che successivamente formeranno l'albero di connessione minima.
In fine è definita la classe "grafo" che implementa l'algoritmo di Kruskal e fa uso delle classi precedentemente citate. Essa implementa tutte le procedure di gestione come private e permette di gestire il grafo come un array di tipo "arco". Rappresentare il grafo come un array di archi ci permette di eseguire operazioni come l' ordinamento in maniera piu semplice. All' interno della classe Grafo troviamo le operazioni di Union-Find per lavorare sugli elementi della classe Albero.

## 3.2 Implementazione di Prim

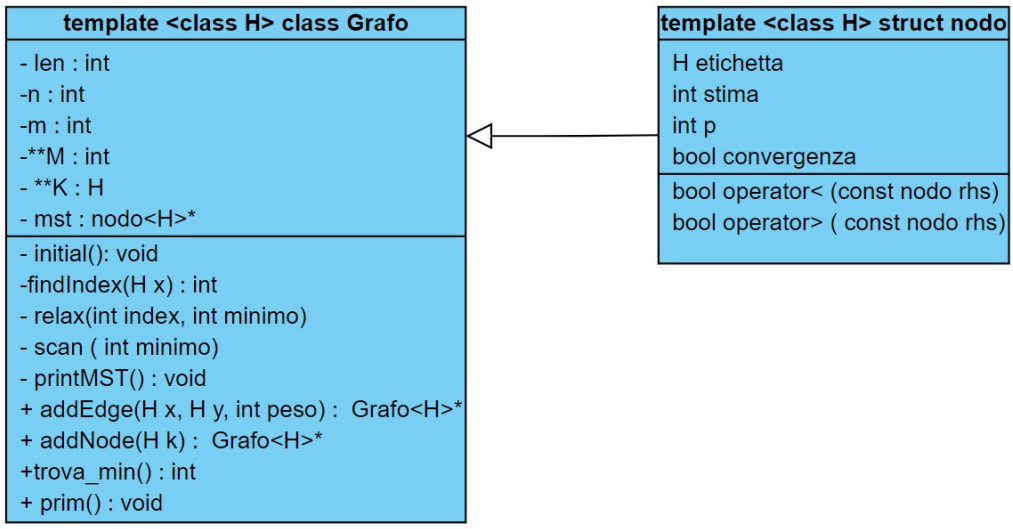
Si sceglie di utilizzare una struct "nodo", che tiene conto di variabili utili al funzionamento dell'algoritmo come : la stima di distanza , il predecessore , e una variabile bool che indica se il nodo è andato a convergenza.
Viene implementata la classe MGraph per mantenere i nodi che successivamente vengono utilizzati dal metodo prim per trovare l'albero di connessione minima. A differenza della classe "Grafo" dell' algoritmo di Kruskal , il grafo , verrà rappresentato con una matrice di adiacenza. In seguito alla chiamata di initial(), all' interno della procedura prim(), verrà creato un array di tipo "nodo" che conterrà tutti i nodi con i valori settati. L' algoritmo di prim lavorerà interamente sull' array di tipo nodo.
Le procedure di gestione in MGraph sono definite come private, e ad esse sono delegate vari compiti rendendo il metodo prim molto conciso e simile allo pseudo codice visto. La coda di min priorità è sostituita da una procedura

*trovaMin* che si occuperà di trovare il vertice che ha stima minima non ancora andato a convergenza.

## 3.3 Diagramma UML Alg. Kruskal



## 3.4 Diagramma UML Alg. Prim



## Fonti e Approfondimenti

*Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, "Introduction to Algorithms, Third Edition" 2010, McGraw-Hill.*
*"Minimum spanning tree", Wikipedia.*
*Prof. Cantone Domenico, corso di laurea magistrale*, "Minimum spanning tree" A.A. 2018/19.