Progettazione di Algoritmi - lezione 12

Discussione dell'esercizio [MST]

Sia G un grafo pesato e connesso e sia T un MST di G.

a. Se incrementiamo di una stessa costante c il peso degli archi di G, il nuovo peso di un albero di copertura S diventa $p'(S) = p(S) + (n - 1)c$. Quindi T è ancora un MST dato che $p'(T) = p(T) + (n - 1)c \leq p(S) + (n - 1)c = p'(S)$, per ogni albero di copertura S.

b. Un arco di peso minimo di G è un arco $\{u, v\}$ tale che $p(u, v) \leq p(x, y)$ per ogni arco $\{x, y\}$ di G. Si osservi che possono esserci tanti archi di peso minimo. Supponiamo per assurdo che T non contenga nessun arco di peso minimo. Sia allora $\{u, v\}$ un arco di peso minimo di G. Siccome $\{u, v\}$ non appartiene a T, $T \cup \{u, v\}$ ha un ciclo C che contiene l'arco $\{u, v\}$. Per ipotesi T non contiene archi di peso minimo, quindi tutti gli archi di C eccetto $\{u, v\}$, essendo archi di T, devono avere un peso strettamente maggiore a $p(u, v)$. Sia $\{w, z\}$ uno di questi. Definiamo $T' = (T - \{w, z\}) \cup \{u, v\}$. Chiaramente T' è ancora un albero di copertura perché è connesso, copre n nodi ed ha esattamente $n - 1$ archi. Inoltre, risulta $p(T') = p(T) - p(w, z) + p(u, v) < p(T)$ (dato che $p(u, v) < p(w, z)$), in cotraddizione con l'ipotesi che T sia un MST. Quindi T deve necessariamente contenere almeno un arco di peso minimo.

c. Assumiamo che gli archi di G abbiano tutti pesi distinti. Supponiamo per assurdo che ci siano due MST differenti A e B di G. Sia allora $\{u, v\}$ l'arco di peso minimo tra quelli che appartengono alla differenza simmetrica di A e B, cioè $A \Delta B = (A - B) \cup (B - A)$. Supponiamo, senza perdita di generalità, che $\{u, v\}$ è in $A - B$. L'aggiunta di $\{u, v\}$ a B produce un ciclo C. Gli archi in $C - \{u, v\}$ appartengono tutti a B ma non possono appartenere anche a A, altrimenti A conterrebbe un ciclo. Quindi c'è almeno un arco $\{w, z\}$ che è in C (e quindi in B) ma non è in A. Ne segue che $\{w, z\}$ è in $A \Delta B$ e allora $p(w, z) > p(u, v)$ (perché $\{u, v\}$ ha peso minimo tra tutti gli archi in $A \Delta B$ e tutti i pesi sono diversi). Perciò $B' = (B - \{w, z\}) \cup \{u, v\}$ è un albero di copertura con peso strettamente inferiore a quello di B contraddicendo l'ipotesi che B fosse un MST. Dunque, se tutti i pesi degli archi sono distinti, c'è un unico MST.

d. Effettivamente l'algoritmo di Prim funziona correttamente anche quando G contiene archi di peso negativo. Questo può anche essere ricavato da quanto provato nel punto (a).

Algoritmo di Kruskal

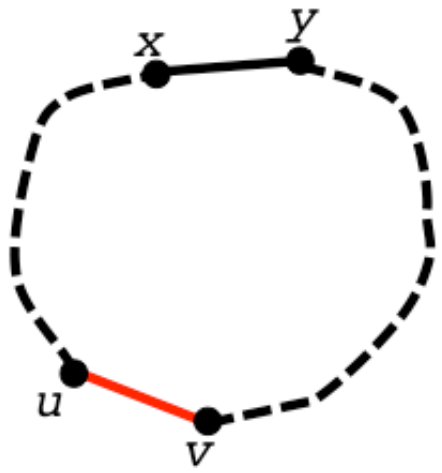
Torniamo al problema di trovare un MST. L'algoritmo di Kruskal parte dall'insieme vuoto e ad ogni passo aggiunge all'insieme un arco di peso minimo tra tutti quelli che possono essere aggiunti senza creare cicli.

```
KRUSKAL(G: grafo non diretto, pesato e connesso)
    SOL <- insieme vuoto
    WHILE esiste un arco che può essere aggiunto a SOL senza creare cicli DO
        Sia {u, v} un arco di peso minimo fra tutti quelli che possono essere aggiunti a SOL
        SOL <- SOL ∪ {{u, v}}
    RETURN SOL
```

Prima di tutto sinceriamoci che l'algoritmo termina sempre. Ad ogni iterazione del `WHILE` un nuovo arco viene aggiunto a `SOL` e la condizione del `WHILE` diventa falsa quando non ci sono più archi che possono essere aggiunti a `SOL` senza creare cicli, quindi vengono eseguite al più n-1 iterazioni (un grafo con al più n nodi e aciclico ha al più n-1 archi). Siccome il grafo è connesso, il numero di iterazioni è esattamente n-1, dato che se fossero di meno il sottografo indotto dagli archi in `SOL` non sarebbe connesso e quindi esisterebbe almeno un arco che si può aggiungere senza creare cicli. Sia SOL_h il valore di `SOL` al termine della h-esima iterazione e sia SOL_0 il valore iniziale. Dimostriamo che ogni soluzione parziale è estendibile ad una soluzione ottima.

Per ogni $h = 0,1,\ldots n\text{-}1$ esiste una soluzione ottima SOL^* che estende SOL_h.

Dimostrazione Procediamo per induzione su h. Per $h = 0$ è banalmente vero. Sia ora $h \geq 0$ (e $h \leq n\text{-}1$), assumendo la tesi vera per h dimostriamola per $h + 1$. Per ipotesi induttiva esiste una soluzione ottima SOL^* che estende SOL_h. Se SOL^* estende anche SOL_{h+1}, abbiamo fatto. Altrimenti, l'arco $\{u, v\}$ aggiunto a SOL_h nella $(h + 1)$-esima iterazione non appartiene a SOL^*. Cerchiamo di trasformare SOL^* in un'altra soluzione ottima che contiene SOL_{h+1}. Siccome gli archi di SOL^* formano un albero di copertura, in $SOL^* \cup \{u, v\}$ l'arco $\{u, v\}$ determina un ciclo C. L'arco $\{u, v\}$ non crea cicli con SOL_h per cui in C deve esserci almeno un altro arco $\{x, y\}$ che non appartiene a SOL_h.



Inoltre, l'arco $\{x, y\}$ non crea cicli con SOL_h dato che appartiene a SOL^* e SOL^* contiene SOL_h e non ha cicli. Quindi l'arco $\{x, y\}$ era tra gli archi che potevano essere scelti durante l'$(h + 1)$-esima iterazione. È stato scelto $\{u, v\}$ per cui deve essere che $p\{u, v\} \leq p\{x, y\}$. Definiamo $SOL^{\#} = (SOL^* - \{\{x, y\}\}) \cup \{\{u, v\}\}$. Chiaramente, $SOL^{\#}$ è una soluzione ottima che estende SOL_{h+1}.

Adesso sappiamo che la soluzione finale $SOL_{n\text{-}1}$ prodotta dall'algoritmo di Kruskal è contenuta in una soluzione ottima SOL^*. Ma sia $SOL_{n\text{-}1}$ che la soluzione SOL^* hanno lo stesso numero di archi, cioè n-1, quindi sono uguali.

Implementazione efficiente dell'algoritmo di Kruskal

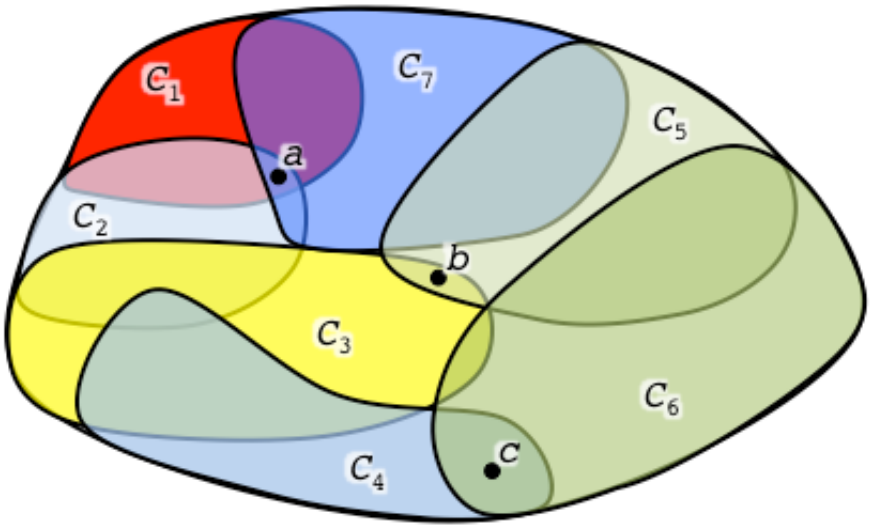
Ora che sappiamo che l'algoritmo di Kruskal è corretto, consideriamo possibili implementazioni efficienti. Osserviamo subito che quando in un'iterazione un arco è esaminato e scartato perché crea cicli non sarà successivamente riesaminato. Allora, può essere conveniente ordinare preliminarmente gli archi del grafo in ordine di peso non decrescente. Fatto ciò, per ogni arco che si presenta tramite questo ordine, si deve controllare se produce o meno cicli con gli archi in `SOL` . Per effettuare tale controllo osserviamo che `SOL` determina delle componenti connesse che coprono tutti i nodi del grafo e un arco non crea cicli con `SOL` se e solo se gli estremi dell'arco appartengono a componenti connesse differenti. Per mantenere l'informazione di queste componenti possiamo usare un vettore `CC` tale che, per ogni nodo `i` , `CC[i]` è l'etichetta che identifica la componente a cui il nodo `i` appartiene. Come etichetta di una componente possiamo usare un nodo della componente.

```
KRUSKAL(G: grafo non diretto, pesato e connesso)
    SOL <- lista vuota
    E <- array contenente gli m archi di G, per ogni arco i: E[i].u, E[i].v, E[i].p
    Ordina l'array E rispetto ai pesi in modo non descrescente
    CC <- array delle componenti
    FOR ogni nodo u DO
        CC[u] <- u      /* Inizialmente, ogni nodo è una componente a sè stante */
    FOR i = 1 TO m DO
        {u, v} <- {E[i].u, E[i].v}
        IF CC[u] <> CC[v] THEN         /* Se l'arco non crea cicli, */
            SOL.append({u, v})         /* aggiungilo alla soluzione */
            c <- CC[v]
            FOR ogni nodo w DO         /* Fondi le due componenti */
                IF CC[w] = c THEN
                    CC[w] <- CC[u]
    RETURN SOL
```

L'ordinamento degli archi richiede O($m$log$m$) tramite un algoritmo classico di ordinamento. Ogni iterazione del `FOR` sugli archi richiede un tempo che dipende dal fatto se l'arco crea o meno cicli. Se crea cicli l'iterazione richiede O(1) ma se invece non crea cicli e quindi l'arco viene aggiunto l'iterazione richiede O(n). Per fortuna però le iterazioni più onerose non sono così tante. Infatti, ce ne saranno tante quanti sono gli archi che vengono aggiunti a `SOL` , cioè, $n-1$. Quindi, il `FOR` sugli archi richiede O($m + n^2$) = O(n^2) e l'implementazione ha complessità O($m$log$m + n^2$) = O($m$log$n + n^2$). Quando il grafo non è denso questa è una complessità superiore a quella che abbiamo visto per l'algoritmo di Prim O(($n + m$)log$n$). Per ottenere una complessità di quest'ordine anche per l'algoritmo di Kruskal si può usare una struttura dati, chiamata *merge-find-set*, che permette di gestire efficientemente una collezione di insiemi disgiunti (le componenti) rispetto alle operazioni di *find* (determinare la componente di un nodo) e *merge* (fusione di due componenti).

Ricoprimento tramite Nodi

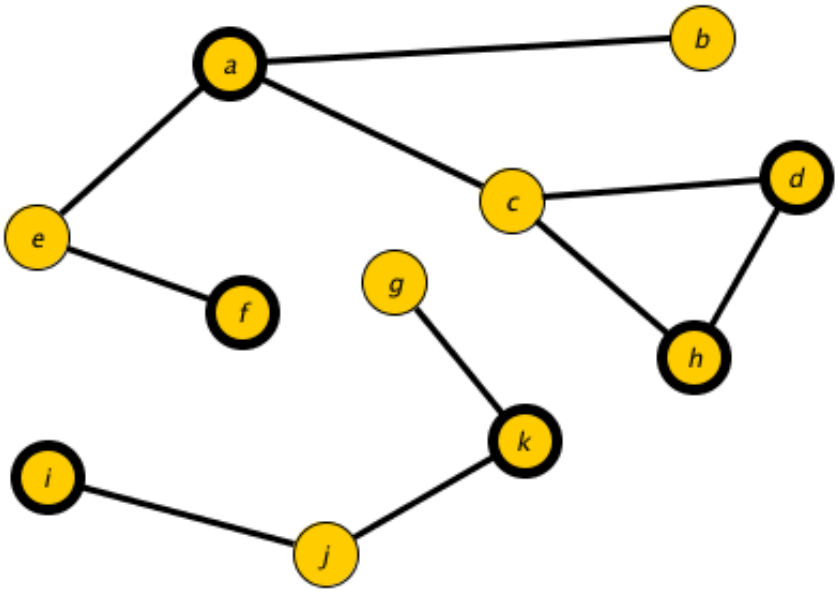
Una azienda ha bisogno di coprire un certo insieme di mansioni $m_1,\ldots,m_n$. Per ogni mansione m_i c'è un insieme di possibili candidati C_i in grado di svolgere quella mansione. Un candidato può essere in grado di svolgere più mansioni e quindi gli insiemi C_i possono essere non disgiunti. L'azienda vuole assumere un numero minimo di candidati capaci di coprire tutte le mansioni.



Ad esempio nella figura qui sopra, i candidati *a*, *b* e *c* coprono tutte e 7 le mansioni.

In termini più astratti il problema può essere formulato come segue. Dati n sottoinsiemi $C_1,\ldots,C_n$ di un insieme C (l'insieme di tutti i candidati) si vuole trovare un sottoinsieme H di C di cardinalità minima che ha intersezione non

vuota con ogni C_i. Nella letteratura questo problema è conosciuto con il nome di *Minimum Hitting Set*. Nonostante l'apparente semplicità si tratta di un problema molto difficile. Rimane difficile anche se assumiamo che la cardinalità degli insiemi C_i sia 2, che è la cardinalità più piccola affinché il problema non diventi banale. Con questa restrizione il problema può essere formulato in termini di grafi. I nodi sono gli elementi di C e per ogni C_i c'è un arco non diretto i cui estremi sono proprio gli elementi di C_i (che per assunzione ha cardinalità 2). Il problema, che è conosciuto con il nome di *Ricomprimento tramite Nodi* (in inglese *Vertex Cover*), è così definito: dato un grafo non diretto G trovare un sottoinsieme dei nodi di cardinalità minima che copre tutti gli archi di G (un nodo *copre* un arco se è uno dei suoi due estremi). Nella figura qui sotto i nodi marcati formano un ricoprimento ottimo.

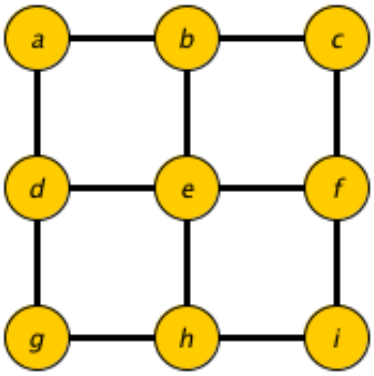


Consideriamo un semplice algoritmo greedy per il problema Ricomprimento tramite Nodi. Il primo che viene in mente consiste nello scegliere i nodi in base al numero di archi che coprono. Cioè, si sceglie sempre un nodo che massimizza il numero di archi che copre fra quelli non coperti dai nodi già scelti:

```
MAX_GRADO(G: grafo non diretto)
    C <- insieme vuoto
    WHILE ci sono archi non coperti dai nodi in C DO
        Sia u un nodo che copre il massimo numero di archi non ancora coperti da C
        C <- C u {u}
    RETURN C
```

Vediamo come si comporta su qualche grafo. Nel grafo della figura qui sotto, l'algoritmo MAX_GRADO all'inizio sceglie uno dei tre nodi *b*, *c* o *d*.

Se sceglie *b* o *d* poi sceglierà l'altro dei due e produrrà una soluzione ottima (un ricoprimento con due nodi). Ma se sceglie *c* poi dovrà necessariamente scegliere altri due nodi producendo una soluzione con tre nodi che non è ottima. Si potrebbe pensare di emendare l'algoritmo aggiungendo qualche ulteriore criterio di scelta quando ci sono più nodi che massimizzano il numero di nuovi archi che vengono coperti. Ma ciò non è possibile, infatti consideriamo il grafo seguente:



L'algoritmo MAX_GRADO sceglie all'inizio il nodo centrale *e*, poi deve necessariamente scegliere altri 4 nodi, producendo un ricoprimento di cardinalità 5. Ma i 4 nodi *b*, *d*, *f*, *h* sono un ricoprimento e quindi l'algoritmo non ritorna un ricoprimento ottimo. Il fatto che l'algoritmo MAX_GRADO non sia corretto non è sorprendente perché non si conoscono algoritmi efficienti per il problema Ricoprimento tramite Nodi e si congettura che non esistano. Per problemi così difficili è naturale cercare algoritmi efficienti che seppur non corretti producono soluzioni vicine a soluzioni ottime.

Algoritmi di approssimazione

Ci sono moltissimi problemi come Ricoprimento tramite Nodi, che hanno grande importanza sia sul piano teorico sia su quello applicativo, ma che sono computazionalmente difficili. Ovvero non si conoscono algoritmi neanche lontanamente efficienti per tali problemi. Tuttavia, spesso trovare la soluzione ottima non è l'unica cosa che interessa. Potrebbe essere già soddisfacente ottenere una soluzione che sia soltanto vicina ad una soluzione ottima e, ovviamente, più è vicina e meglio è. In realtà è proprio così per un gran numero di problemi.

Fra gli algoritmi che non trovano sempre una soluzione ottima, è importante distinguere due categorie piuttosto differenti. Ci sono gli algoritmi per cui si dimostra che la soluzione prodotta ha almeno una certa vicinanza ad una soluzione ottima. In altre parole, è garantito che la soluzione prodotta approssima entro un certo grado una soluzione ottima. Questi sono chiamati algoritmi di approssimazione e vedremo tra poco come si misura la vicinanza ad una soluzione ottima. L'altra categoria è costituita da algoritmi per cui non si riesce a dimostrare che la soluzione prodotta ha sempre una certa vicinanza ad una soluzione ottima. Però, sperimentalmente sembrano comportarsi bene. Essi sono a volte chiamati algoritmi euristici. Spesso sono l'ultima spiaggia, quando non si riesce a trovare algoritmi corretti efficienti né algoritmi di approssimazione efficienti che garantiscano un buon grado di approssimazione, rimangono soltanto gli algoritmi euristici. Per una gran parte dei problemi computazionalmente difficili non solo non si conoscono algoritmi corretti efficienti ma neanche buoni algoritmi di approssimazione. Non è quindi sorprendente che fra tutti i tipi di algoritmi, gli algoritmi euristici costituiscano la classe più ampia e che ha dato luogo ad una letteratura sterminata. Ciò è dovuto non solo ai motivi sopra ricordati ma anche, e forse soprattutto, al fatto che è quasi sempre molto più facile inventare un nuovo algoritmo o una variante di uno già esistente e vedere come si comporta sperimentalmente piuttosto che dimostrare che un algoritmo, vecchio o nuovo che sia, ha una certa proprietà (ad esempio, è corretto, garantisce un certo grado di approssimazione, ecc.). Prima di passare a descrivere più in dettaglio gli algoritmi di approssimazione, è bene osservare che la classificazione che abbiamo dato in algoritmi di approssimazione ed algoritmi euristici è una semplificazione della realtà. Esistono algoritmi che non ricadono in nessuna delle classi discusse. Ad esempio gli algoritmi probabilistici per cui si dimostra che con alta probabilità producono una soluzione ottima ma che possono anche produrre soluzioni non ottime sebbene con piccola probabilità. Un altro importante esempio sono gli algoritmi che producono sempre una soluzione ottima ma non è garantito che il tempo di esecuzione sia sempre efficiente.

Un algoritmo di approssimazione per un dato problema è un algoritmo per cui si dimostra che la soluzione prodotta approssima sempre entro un certo grado una soluzione ottima per il problema. Si tratta quindi di specificare cosa si intende per "approssimazione entro un certo grado". Ci occorrono alcune semplici nozioni. Sia *P* un qualsiasi problema di ottimizzazione e sia *I* una istanza di *P*, indichiamo con *OTT*(*I*) il valore o misura di una soluzione ottima per l'istanza *I*. Ad esempio se *P* è il problema Selezione Attività ed *I* è un'istanza di tale problema (cioè, un insieme di attività specificate tramite intervalli temporali) allora *OTT*(*I*) è il numero massimo di attività di *I* che possono essere

svolte senza sovrapposizioni in un unica aula. Se *P* è il problema Minimo Albero di Copertura ed *I* è un istanza di tale problema (cioè, un grafo pesato e connesso) allora *OTT*(*I*) è il peso di un MST per *I*. Quindi *OTT*(*I*) non denota una soluzione ottima per l'istanza *I* ma soltanto il valore di una soluzione ottima. Chiaramente tutte le soluzioni ottime per una certa istanza *I* hanno lo stesso valore. Sia *A* un algoritmo per un problema *P* e sia *I* un'instanza di *P*, indichiamo con *A*(*I*) il valore o misura della soluzione prodotta dall'algoritmo *A* con input l'istanza *I*. Il modo usuale di misurare il grado di aprossimazione di un algoritmo non è altro che il rapporto fra il valore di una soluzione ottima e il valore della soluzione prodotta dall'algoritmo. Assumeremo che un algoritmo produca perlomeno una soluzione che è ammissibile. Per fare sì che il rapporto dei valori dia sempre un numero maggiore od uguale a 1, si distingue fra problemi di minimo e problemi di massimo.

Iniziamo dai problemi di massimo. Sia *P* un problema di massimo ed *A* un algoritmo per *P*. In questo caso risulta sempre $A(I) \leq OTT(I)$. Si dice che *A* approssima *P* entro un fattore di approssimazione *r* se, per ogni istanza *I* di *P*,

$$\frac{OTT(I)}{A(I)} \leqslant r$$

Analogamente per i problemi di minimo. Sia *P* un problema di minimo ed *A* un algoritmo per *P*. In questo caso risulta sempre $A(I) \geq OTT(I)$. Si dice che *A* approssima *P* entro un fattore di approssimazione *r* se, per ogni istanza *I* di *P*,

$$\frac{A(I)}{OTT(I)} \leqslant r$$

Quindi se *A* approssima *P* entro un fattore 1 ciò equivale a dire che *A* è corretto per *P*, cioè trova sempre una soluzione ottima. Se invece *A* approssima *P* entro, ad esempio, un fattore 2, ciò significa che *A* trova sempre una soluzione di valore almeno pari alla metà di quello di una soluzione ottima, se *P* è un problema di massimo, e al più pari al doppio di quello di una soluzione ottima se *P* è di minimo.

Tornando al problema Ricoprimento tramite Nodi, l'algoritmo MAX_GRADO che abbiamo visto non è corretto. Abbiamo trovato un'istanza per cui l'algoritmo produce una soluzione con 5 nodi mentre la soluzione ottima ha 4 nodi. Da ciò possiamo dedurre che il fattore di approssimazione di MAX_GRADO è maggiore od uguale a $5/4 > 1$. Ma potrebbe essere peggiore. In effetti per ogni numero *R*, non importa quanto grande, si possono trovare grafi per cui l'algoritmo MAX_GRADO sbaglia di un fattore superiore a *R*. Quindi MAX_GRADO non garantisce nessun fattore di approssimazione costante. Ciò non toglie che potrebbero comunque esistere algoritmi efficienti che garantiscono una buona approssimazione. Uno di questi si basa su una semplice osservazione. Se in un grafo *G* ci sono *h* archi fra loro disgiunti allora un qualsiasi ricoprimento di *G* deve avere almeno *h* nodi. Questo è ovvio, un qualsiasi ricoprimento deve coprire gli *h* archi ma essendo essi disgiunti occorrono almeno *h* nodi per coprirli. Un algoritmo che costruisce un ricoprimento in modo strettamente legato alla contemporanea costruzione di un insieme di archi disgiunti produrrebbe un ricoprimento che non può essere troppo lontano da un ricoprimento ottimo. L'idea allora è molto semplice. Ad ogni passo si sceglie un arco fra quelli non ancora coperti ed entrambi gli estremi sono aggiunti all'insieme di copertura. Così facendo si garantisce che tutti gli archi scelti sono fra loro disgiunti. La descrizione dell'algoritmo è immediata:

```
ARCHI_DISGIUNTI(G: grafo non diretto)
    C <- insieme vuoto
    WHILE ci sono archi non coperti da nodi in C DO
        Sia {u, v} un arco non coperto da nodi in C
        C <- C ∪ {u} ∪ {v}
    RETURN C
```

Da quanto sopra detto è chiaro che l'algoritmo produce sempre un ricoprimento di cardinalità al più doppia rispetto a quella di un ricoprimento ottimo. Quindi l'algoritmo ARCHI_DISGIUNTI garantisce un fattore di approssimazione 2. Non si conoscono a tutt'oggi algoritmi efficienti che garantiscano un fattore di approssimazione migliore di 2. Per il problema da cui eravamo partiti, cioè, Minimum Hitting Set, la situazione è anche peggiore. Non si conoscono algoritmi efficienti che garantiscano un fattore di approssimazione migliore di un fattore che è logaritmico nella

dimensione dell'istanza. Però si può dimostrare che l'adattamento dell'algoritmo MAX_GRADO per il problema suddetto garantisce un fattore di approssimazione che è logaritmico nella dimensione dell'istanza. Quindi alla fine quell'algoritmo non è da buttare via.

Esercizio [file]

Si hanno n file di dimensioni $s_1,\ldots,s_n$ e un disco di capacità C. Purtroppo $s_1 + \cdots + s_n > C$, quindi non possiamo memorizzare tutti i file sul disco. Vogliamo trovare un sottoinsieme dei file che può essere memorizzato sul disco e che massimizza lo spazio usato. Più precisamente, indicando i file con gli interi $1,2,\ldots,n$, e per ogni sottoinsieme X di $\{1,2,\ldots,n\}$ denotiamo con $S(X)$ la somma delle dimensioni dei file in X, allora vogliamo trovare un sottoinsieme F di $\{1,2,\ldots,n\}$ tale che (1) $S(F) \leq C$ e (2) per ogni sottoinsieme di file X con $S(X) \leq C$, si ha $S(X) \leq S(F)$. Descrivere un algoritmo (greedy) per questo problema e studiarne il fattore di approssimazione.