Progettazione di algoritmi

Discussione dell'esercizio [MST dinamico]

Sia G un grafo pesato e connesso e sia T un suo MST. Aggiungiamo a G un nuovo arco a di peso p ottenendo il grafo G'. Vogliamo trovare un MST T' di G', sfruttando la conoscenza di T, senza doverlo ricalcolare daccapo.

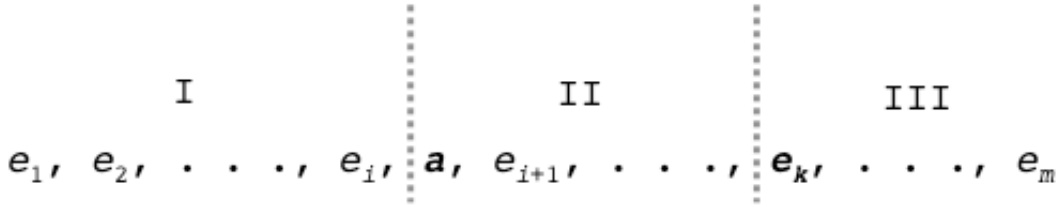
Un algoritmo che viene subito in mente consiste nel considerare $T \cup \{a\}$. Tale insieme di archi ha un ciclo $C^{\#}$ e $C^{\#}$ contiene il nuovo arco a. Sia b l'arco di peso massimo in $C^{\#}$. Se $p(b) > p(a)$, poniamo $T^{\#} = (T - \{b\}) \cup \{a\}$ altrimenti $T^{\#} = T$. Sappiamo che $T^{\#}$ è un albero di copertura, ma ha anche peso minimo in G'?. Per verificare che ciò è vero, farebbe comodo poter assumere che T sia stato costruito tramite l'algoritmo di Kruskal, potremmo così confrontare le due esecuzioni dell'algoritmo sui grafi G e G'. Affinchè l'algoritmo di Kruskal sia corretto è richiesto solamente che gli archi siano considerati in ordine di peso non descrescente. Gli archi di pari peso possono essere permutati tra loro in un modo qualsiasi senza compromettere la correttezza dell'algoritmo.

> **Dato un grafo pesato e connesso G e un qualsiasi MST T di G, ordinando gli archi in modo tale che gli archi di T siano i primi tra gli archi di pari peso (cioè, se gli archi e ed e' hanno pari peso, e è in T ed e' non è in T, allora e viene prima di e'), l'algoritmo di Kruskal produce in output proprio T.**

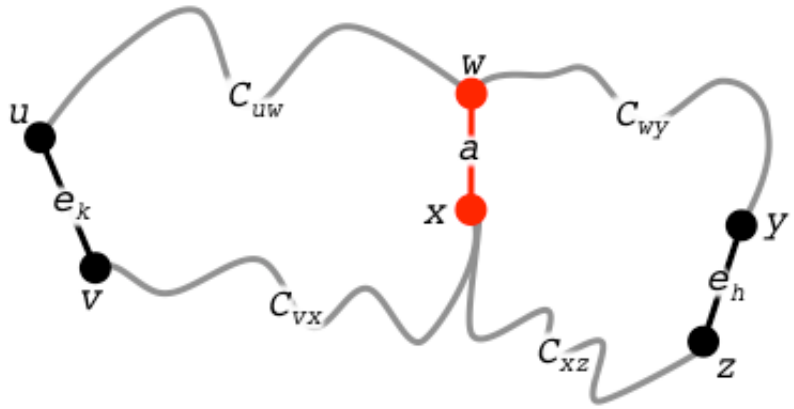
Dimostrazione Supponiamo per assurdo che eseguendo l'algoritmo di Kruskal con l'ordinamento degli archi suddetto l'MST prodotto T' è diverso da T. Sia e' il primo arco durante l'esecuzione che viene scelto in T' ma non appartiene a T. Sia C il ciclo in $T \cup \{e'\}$. Ci deve essere almeno un arco e in C che non appartiene a T', altrimenti T' conterrebbe un ciclo. Non può essere $p(e) > p(e')$, altrimenti scambiando i due archi si otterebbe un albero di copertura di peso inferiore a quello di T, il che non è possibile perché T è un MST. Non può neanche essere $p(e) < p(e')$ perchè altrimenti e sarebbe stato considerato dall'algoritmo prima di e' ed essendo e' il primo arco scelto dall'algoritmo non appartenente a T, quando l'arco e è stato considerato non avrebbe prodotto cicli e quindi sarebbe stato scelto in T' (ma e non appartiene a T'). Rimane solamente la possibilità che $p(e) = p(e')$. Ma ancora una volta l'arco e non può essere stato considerato prima di e', quindi deve essere stato considerato dopo e'. Ma questo è in contraddizione con l'ordinamento degli archi perché ci sarebbe un arco e', non in T, di pari peso di un arco e di T che nell'ordinamento viene prima dell'arco e di T.

Consideriamo l'ordinamento degli archi di G in modo tale che l'algoritmo di Kruskal produca

in output l'MST T. Inseriamo in questo ordinamento il nuovo arco a di modo che venga immediatamente dopo gli eventuali archi di T di pari peso. Sia T' l'MST di G' prodotto dall'algoritmo di Kruskal con questo ordinamento. Se l'arco a non viene scelto, allora $T' = T$ e anche $T^{\#} = T$, perché a produce un ciclo con archi di T di peso $\leq p(a)$ e quindi $p(b) \leq p(a)$. In questo caso il nostro algoritmo è corretto. Consideriamo allora il caso in cui l'arco a viene scelto. Sia $e_1,e_2,\ldots,e_m$ l'ordinamento degli archi di G e sia $e_1,e_2,\ldots,e_i,a,e_{i+1},\ldots,e_m$ l'ordinamento degli archi di G'. Sia e_k il primo arco di T che non viene scelto nelll'esecuzione su G'. Chiaramente un tale arco esiste (altrimenti si avrebbero un ciclo) ed è incontrato dopo l'arco a. Possiamo suddividere le esecuzioni dell'algoritmo di Kruskal su G e G' in tre fasi come in figura:



Nella prima fase le scelte effettuate dalle due esecuzioni coincidono. Nella seconda fase l'unica differenza è solamente la scelta dell'arco a per G' (non possono essere aggiunti archi diversi da quelli in T perché producono cicli). Nella terza fase l'unica differenza è la scelta dell'arco e_k. Infatti, nessun arco non in T può essere scelto perché produrrebbe un ciclo e tutti gli archi in T che vengono dopo e_k sono scelti perché non producono cicli in entrambe le esecuzioni. Quest'ultima affermazione richiede una dimostrazione. Supponiamo per assurdo che ci sia un arco e_h in T, successivo a e_k, che non viene scelto nell'esecuzione di G' (supponiamo che sia il primo che non viene scelto). Sia T_k l'insieme degli archi di T scelti nelle prime due fasi, ovviamente sono gli stessi in entrambe le esecuzioni. Sia T_h l'insieme degli archi di T scelti prima di e_h, anche questi coincidono nelle due esecuzioni. Siccome e_k non è stato scelto esiste un ciclo in $T_k \cup \{a, e_k\}$. Perciò esiste sia un cammino C_{uw} in T_k da un estremo di $e_k = \{u, v\}$ a un estremo di $a = \{w, x\}$ sia un cammino C_{vx} in T_k dall'altro estremo di e_k all'altro estremo di a. Parimenti, siccome c'è un ciclo in $T_h \cup \{a, e_h\}$, esiste un cammino C_{wy} in T_h da un estremo di a ad un estremo di $e_h = \{y, z\}$ e un cammino C_{xz} dall'altro estremo di a all'altro estremo di e_h.



Allora esistono in T (siccome T_k e T_h sono sottoinsiemi di T) sia un cammino da un estremo di e_k a un estremo di e_h (la concatenazione di C_{uw} e C_{wy}) sia un cammino dall'estremo di e_k all'altro estremo di e_h (la concatenazione di C_{vx} e C_{xz}). Quindi c'è un ciclo in T, in contraddizione con il fatto che T è un albero.

Riassumendo nelle due esecuzioni le sole differenze sono l'arco a, scelto in T' ma non in T, e l'arco e_k, scelto in T ma non in T'. Allora e_k crea un ciclo in $T' \cup \{e_k\} = T \cup \{a\}$ che non può che essere il ciclo $C^{\#}$ individuato dal nostro algoritmo. Inoltre il peso dell'arco e_k è massimo tra i pesi di tutti gli archi del ciclo (perché è l'ultimo arco del ciclo ad essere considerato durante l'esecuzione ddell'algoritmo di Kruskal), quindi $p(e_k) = p(b)$. Ne segue che $p(T') = p(T) - p(e_k) + p(a) = p(T) - p(b) + p(a) = p(T^{\#})$ e l'albero prodotto dal nostro algoritmo è garantito essere un MST di G'.

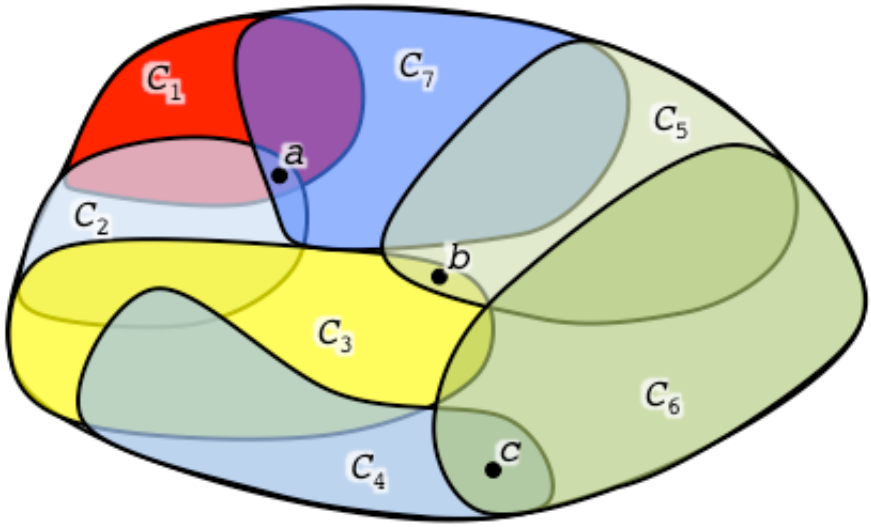
Per quanto riguarda la complessità, facendo una semplice visita (ad esempio DFS) da un estremo dell'arco a si ottengono gli archi del ciclo $C^{\#}$ e poi è sufficiente trovare tra questi l'arco di peso massimo. Quindi la complessità è $O(n)$.

Se l'arco a invece di essere aggiunto viene eliminato (assumendo che il grafo rimanga connesso) possiamo pensare ad un semplice algoritmo che evita il ricalcolo dell'MST. Se l'arco a non è nell'MST T di G, allora T è anche un MST per il grafo G' modifcato. Se invece a è in T, gli archi di T che rimangono costituiscono due componenti (o sottoalberi) T_1 e T_2. Un algoritmo che viene in mente subito consiste nell'aggiungere un arco di peso minimo b tra tutti quelli che uniscono T_1 e T_2 (cioè hanno un estremo in T_1 e l'altro in T_2). Quindi questo algoritmo costruisce l'albero $T^{\#} = (T - \{a\}) \cup \{b\}$. La correttezza può essere dimostrata in modo molto simile a come abbiamo fatto per l'algoritmo precedente. La complessità è un po' più elevata, infatti possiamo determinare T_1 e T_2 in $O(n)$ e poi però dobbiamo esaminare tutti gli archi in tempo $O(m)$, in totale $O(n + m)$.

Ricoprimento tramite Nodi

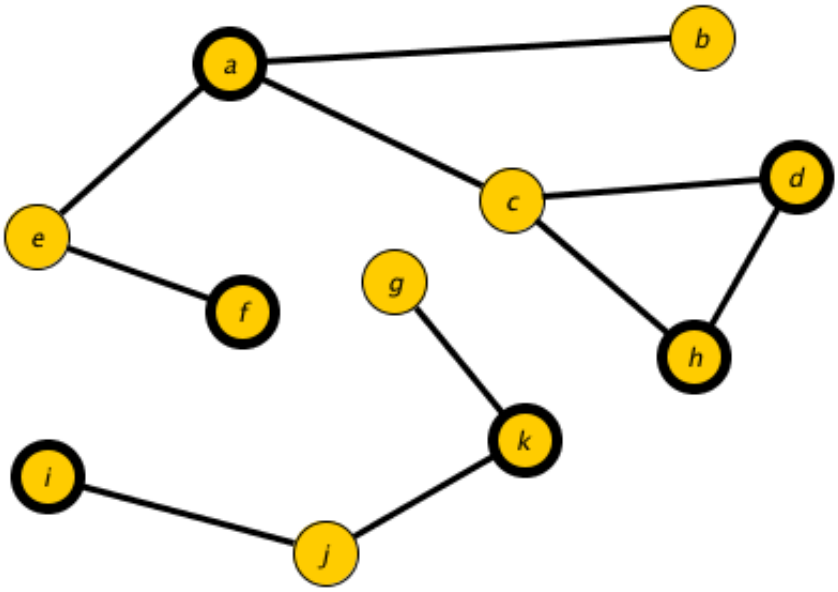
Una azienda ha bisogno di coprire un certo insieme di mansioni $m_1,\ldots,m_n$. Per ogni mansione m_i c'è un insieme di possibili candidati C_i in grado di svolgere quella mansione.

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 svolgere 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, …,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). Più precisamente, 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.

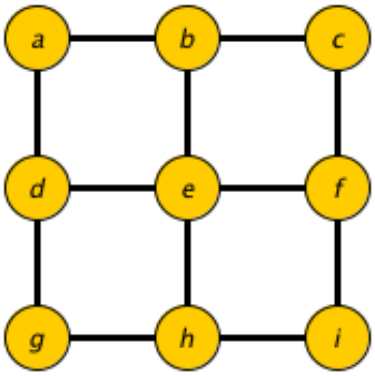


Consideriamo un semplice algoritmo greedy per il problema Ricomprimento tramite Nodi. Il primo che viene in mente consiste nello scegliere i nodi in base al numero di archi che coprono. Cioè, si sceglie sempre un nodo che massimizza il numero di archi che copre fra quelli non coperti dai nodi già scelti:

```
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* 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

$$\textbf{per ogni istanza } I \textbf{ di } P, \; OTT(I)/A(I) \leq r.$$

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

$$\textbf{per ogni istanza } I \textbf{ di } P, \; A(I)/OTT(I) \leq r.$$

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 almeno pari 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 pari a 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, denotiamo i file con gli interi 1,2,…,n, e per ogni sottoinsieme X di {1,2,…,n} denotiamo con $S(X)$ la somma delle dimensioni dei file in X, allora vogliamo trovare un sottoinsieme F di {1,2,…,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.