Capitolo 1

Algoritmi approssimati

1.1 Definizioni e concetti preliminari

Molto spesso nelle applicazioni ci si trova di fronte alla necessità di risolvere problemi NP-completi e garantire l'ottimalità della soluzione richiederebbe un tempo di calcolo troppo elevato. Pensiamo, a titolo di esempio, allo scheduling dei lavori giornalieri in una data fabbrica, sarebbe inaccettabile se per calcolare una soluzione si impiegasse quasi mezza giornata.
Nasce allora la necessità di avere a disposizione *algoritmi approssimati*, la cui complessità sia generalmente bassa, purché si abbia una garanzia sull'entità dell'errore commesso. Si introducono allora le seguenti definizioni:

Definizione 1. *Sia*

- c *il valore della soluzione fornita dall'algoritmo asppprossimato,*
- c^* *il valore della soluzione ottima,*

allora chiamiamo ***fattore di approssimazione*** *quella quantità* $\rho(n)$ *tale che*

$$1 \leq \max\left\{\frac{c}{c^*}, \frac{c^*}{c}\right\} \leq \rho(n),$$

dove n *è la dimensione dell'istanza del problema. Diremo che l'algoritmo è* $\rho(n)$ *approssimato.*

Definizione 2. *Gli algoritmi tali per cui* $\rho(n)$ *è costante si dicono* ***schemi di approssimazione****.*

Se il fattore di approssimazione è unitario allora la soluzione trovata è quella ottima. In generale in uno schema di approssimazione si vuole trovare una soluzione $(1+\varepsilon)$ approssimata, con ε costante. Intuitivamente ci si aspetta che più ε è piccolo più il tempo di calcolo salga.

Definizione 3. *Se, per ogni* ε *fissato, l'algoritmo è polinomiale in* n*, allora si ha uno schema di approssimazione polinomiale (****PTAS****).*

Esempio : $\mathcal{O}(n^{\frac{2}{3\varepsilon}})$.

Definizione 4. *Se l'algoritmo è polinomiale sia in* ε *che in* n*, si ha uno schema di approssimazione pienamente polinomiale (****FPTAS****).*

Esempio : $\mathcal{O}\left(n^3\left(\frac{1}{\varepsilon}\right)^2\right)$.

1.2 Scheduling di lavori

Siano dati:

- m macchine identiche,
- n lavori,
- t_j, $j = 1, .., n$ il tempo di processamento del j-esimo lavoro,

inoltre i lavori non siano partizionabili (altrimenti il problema sarebbe banale) e non ci sia nessun vincolo di precedenza o di finestra temporale.
Indicato con t_i l'ineieme dei lavori assegnati alla macchina i, il problema consiste nell'assegnare i lavori alle macchine in modo tale da minimizzare il tempo impiegato, ovvero, detto

$$T_i = \sum_{j \in A_i} t_j$$

il tempo di utilizzo della i-esima macchina, si vuole disporre i lavori in modo da avere

$$T = \min \max_{i=1,..,m} T_i. \tag{1.1}$$

In gergo si dice che si vuole minimizzare il *makespan*.

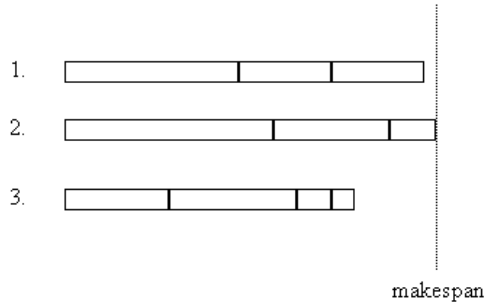

Figura 1.1: Scheduling di lavori

1.2.1 Algoritmo GREEDY1

Una prima idea che verrebbe in mente per risolvere il problema è assegnare i lavori nell'ordine di input sempre alla macchina più scarica. Formalizzando

GREEDY1(A)

```
1  for i ← 1 to m
2      do A[i] ← ∅
3         T[i] ← 0
4  for j ← 1 to n
5      do let k = arg min{T[i] i = 1, .., m}
6         A[k] ← A[k] ∪{j}
7         T[k] ← T[k]+t_j
```

Per quanto riguarda la complessità si ricava facilmente che il primo ciclo for costa $\mathcal{O}(m)$, mentre il secondo costa $\mathcal{O}(n\phi(m))$, dove $\phi(m)$ è il costo della determinazione del minimo. Il secondo termine è dunque quello che determina la complessità.

Analisi del fattore di approssimazione

Sia T^* il valore della soluzione ottima e T il valore trovato dall'algoritmo. Ovviamente sarà $T \geq T^* > 0$.
A questo punto osserviamo che

$$T^* \geq \frac{1}{m} \sum_{j=1}^{n} t_j, \tag{1.2}$$

cioè il tempo ottimo è maggiore o uguale della somma dei tempi di tutti i lavori diviso il numero delle macchine disponibili (questa sarebbe la soluzione ottima se i lavori fossero partizionabili); inoltre

$$T^* \geq \max t_k. \tag{1.3}$$

A questo punto (si veda la figura (1.2)) sia n_i la macchina più carica, e sia t_j l'ultimo lavoro assegnato alla macchina che fornisce la soluzione. Allora dopo il lavoro j se ne possono assegnare altri, ma T non verrà mai superato. Si sa inoltre che n_i prima dell'assegnazione era la macchina più scarica. A questo punto, siccome le altre macchine hanno carico maggiore di $T - t_j$, abbiamo

$$\sum_{k=1}^{m} T_k \geq m(T - t_j),$$

cioè, sfruttando la (1.2) e sapendo che non tutti i lavori sono stati ancora assegnati,

$$T - t_j \leq \frac{1}{m} \sum_{k=1}^{m} T_k \leq \frac{1}{m} \sum_{l=1}^{n} t_l \leq T^*, \tag{1.4}$$

e, utilizzando la (1.3)

$$t_j \leq T^* \Rightarrow T = (T - t_j) + t_j \leq 2T^*,$$

quindi il fattore di approssimazione è 2. Osserviamo che il punto debole dell'algoritmo è costituito da un'istanza in cui arrivano lavori brevi tutti uguali e alla fine un lavoro molto lungo. La soluzione ottima mette il lavoro più lungo da solo mentre il nostro algoritmo è costretto a metterlo in coda a una serie di lavori brevi. Qualora ci siano $m(m-1)$ lavori di durata unitaria e un lavoro lungo m l'errore che si commette è quasi pari a due.

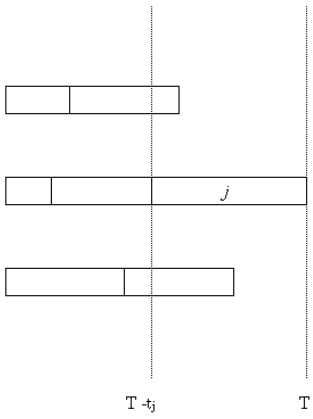

Figura 1.2: Scheduling di lavori (2)

1.2.2 Algoritmo GREEDY2

L'alternativa sarebbe considerare i lavori in ordine decrescente per durata (con un contributo complessivo di complessità pari a $\mathcal{O}(n \lg n)$ per l'ordinamento). Supponiamo $n > m$, altrimenti l'algoritmo fornisce la soluzione ottima. Essendo i lavori ordinati, $T^* \geq t_m + t_{m+1}$ e quindi

$$T^* \geq 2t_{m+1}. \tag{1.5}$$

Sia t_j la durata dell'ultimo lavoro assegnato alla macchina che fornisce la soluzione n_i, allora deve essere $j \geq m+1$ perché ci deve essere una macchina con almeno due lavori ($n \geq m$ per ipotesi), e quindi $t_j \leq t_{m+1}$. Otteniamo allora, grazie alla (1.5) e alla (1.4)

$$T = (T - t_j) + t_j \leq T^* + \frac{1}{2}T^* = \frac{3}{2}T^* \Rightarrow \rho(n) \leq \frac{3}{2}. \tag{1.6}$$

1.3 Localizzazione di centri commerciali

Vi sia un'area geografica con dislocate n città. Si vogliono costruire $k < n$ centri commerciali in modo da minimizzare la massima distanza tra ogni città e il centro più vicino. Si introduca una funzione distanza che soddisfi le seguenti proprietà: per ogni x, y, z,

- $dist(x, x) = 0$,
- $dist(x, y) = dist(y, x)$,
- $dist(x, y) + dist(y, z) \geq dist(x, z)$.

Si indica con C l'insieme delle località in cui si costruisce un centro commerciale e C^* la soluzione ottima.

Definizione 5. *C è una r-copertura se*

$$\min_{c \in C} dist(s, c) \leq r \ \ \forall s \in S,$$

Figura 1.3: Errore

con S insieme delle città.

Si osserva subito che l'idea *greedy* di mettere il primo centro nel baricentro delle città e i successivi nei punti che riducono il raggio di copertura non è un'idea buona perché, ad esempio, se le città fossero raccolte in due nuclei metropolitani, si troverebbe una soluzione molto scorretta (vedi figura (1.3)).

1.3.1 Algoritmo GREEDY1

Si supponga di conoscere a priori il raggio di copertura ottimo, si vuole un algoritmo che sbagli al massimo di $2r$. A tal proposito si decida di posizionare i centri solo in punti di S, in modo da ottenere $C \subseteq S$. Con questa scelta si raggiunge il risultato voluto. Lo pseudocodice è il seguente:

GREEDY1(r)
1 $S' \leftarrow S$
2 $C \leftarrow \emptyset$
3 **while** $S' \neq \emptyset$
4 **do** let $s \in S'$
5 $C \leftarrow C \cup \{s\}$
6 $S' \leftarrow S' \setminus \{i : dist(i, s) \leq 2r\}$
7 **if** $|C| > k$
8 **then** non esiste una soluzione con copertura r

Nel ciclo *while* di riga 3, si considerano tutte le città che si trovano a distanza minore o uguale di $2r$ dalla città s considerata (a caso) e le si rimuovono dall'insieme dei centri commerciali serviti.

Il risultato chiave è il seguente: se, al termine dell'algoritmo sono stati utilizzati un numero di centri commerciali maggiore di K (il numero di centri a disposizione), allora l'ottimo è maggiore di r. Cioè, se l'ottimo è minore o uguale di r, la soluzione trovata dall'algoritmo *greedy* ha raggio di copertura minore o uguale a $2r$.

Lemma 1. *Se $|C| > k$, allora qualsiasi soluzione ottima C^* con $|C^*| \leq k$ ha raggio di copertura maggiore di r.*

Dimostrazione. Per ipotesi si ha che C è soluzione di copertura $2r$ e C^* è la soluzione ottima.

Se esiste una soluzione ottima C^* con $|C^*| \leq k$ e raggio di copertura minore o uguale a r, ogni centro $c^* \in C^*$ appartiene a uno e un solo cerchio di raggio $2r$ centrato su una città di C, che implica $|C| \leq k$. Infatti se esistessero 2 centri c' e c'' di C tali che $dist(c', c^*) \leq r$ e $dist(c'', c^*) \leq r$ allora, grazie alla disuguaglianza triangolare, si conclude che

$$2r \geq dist(c', c^*) + dist(c'', c^*) \geq dist(c', c'') > 2r,$$

dove l'ultima disuguaglianza segue dall'algoritmo. Contraddizione. □

Un approccio per ovviare alla mancata conoscenza di r può essere effettuare una ricerca binaria tramite metodo di bisezione.

Ne segue che la complessità totale sarà moltiplicata per un fattore logaritmico.

1.3.2 Algoritmo GREEDY2

Sia

$$dist(s, C) = \min_{c \in C} dist(s, c).$$

Si propone ora un secondo algoritmo:

GREEDY2

```
1  C ← {s} (s città qualsiasi)
2  while |C| < k
3        do select s ∈ S : s = arg max{dist(s', C), s' ∈ S}
4           C ← C ∪ {s}
```

L'algoritmo lavora in questo modo: l'insime C viene inizializzato con una città arbitraria. In seguito alla soluzione viene aggiunta la città che di volta in volta è più lontana dall'insieme C.

Si ha allora il seguente

Lemma 2. GREEDY2 *fornisce una soluzione con raggio di copertura minore o uguale a* $2r$*, con* r *raggio di copertura della soluzione ottima. Cioè,* $\rho(n) = 2$.

Dimostrazione. Per assurdo. Sia s una città qualsiasi a distanza maggiore di $2r$ dal centro più vicino in C. Siano $C' \subset C$ e c' il centro che GREEDY2 aggiunge a C' ad un determinato passo. Allora c' dista almeno $2r$ da C'. Inoltre, per ipotesi, $dist(s, C) > 2r$ implica

$$dist(c', C') \geq dist(s, C') \geq dist(s, C) > 2r.$$

Si ricade allora nella dimostrazione del lemma 1. Contraddizione. □

Questo algoritmo fornisce quindi una soluzione equivalente per fattore di approssimazione a quella dell'algoritmo GREEDY1, ma prescinde dalla conoscenza a priori del raggio di copertura r.

1.4 Il problema del *Set Covering*

Si consideri un insieme $X = \{x_1, ..., x_n\}$ di categorie lavorative (metalmeccanici, autoferrotranvieri...) ed un insieme $F = \{S_1, ..., S_l\}$ di rappresentanti sindacali, ognuno dei quali puó rappresentare differenti categorie allo stesso momento. Ogni elemento F_i di F è quindi un sottoinsieme di X. Il problema del *Set Covering* consiste nel convocare il minor numero possibile di rappresentanti sindacali, in modo tale che ogni categoria lavorativa sia rappresentata da almeno un sindacalista.

Formalmente, il problema è il seguente: sia $X = \{X_1, ..., X_n\}$ un insieme assegnato, e sia $F = \{S_1, ...S_l\}$ un insieme di sottoinsiemi di X. Trovare un sottoinsieme di F, chiamato C^*, che abbia le seguenti proprietà:

- $\bigcup_{S_i \in C^*} S_i = X$
- La cardinalità di C^* sia minima.

Questo problema è $NP - Hard$. Per la sua risoluzione si può ricorrere quindi un algoritmo greedy approssimato. Una soluzione C sarà perciò tanto migliore quanto più la sua cardinalità è piccola, ovvero quando $\frac{|C|}{|C^*|}$ è piccolo.

L'algoritmo GREEDY-SET-COVER proposto lavora nel seguente modo: alla prima iterazione si prende l'insieme S_0 di cardinalità massima, ed ai passi successivi si prende l'insieme S_i che comprende il maggior numero possibile di elementi di X non ancora coperti. Nell'esempio di figura 1.4 l'insieme generato dall'algoritmo è il seguente: $C = \{S_1, S_4, S_5, S_3\}$, mentre la soluzione ottima è costituita dall'insieme $C^* = \{S_3, S_4, S_5\}$, con errore relativo pari a $\frac{4}{3}$. Lo pseudocodice è quindi il seguente:

GREEDY-SET-COVER(X, F)
1 $U \leftarrow \{X\}$
2 $C \leftarrow \emptyset$
3 **while** $U \neq \emptyset$
4 **do** scegli un $S \in F$ che massimizza $|S \cap U|$
5 $U \leftarrow U \setminus S$
6 $C \leftarrow C \cup \{S\}$
7 **return** C

1.4.1 Analisi del fattore di approssimazione

Purtroppo l'algoritmo *greedy* appena fornito non ha un fattore di approssimazione $\rho(n)$ costante: esso dipende dalla dimensione del problema in maniera logaritmica, e quindi la qualità della soluzione approssimata sarà tanto peggiore quanto più il problema è grande. L'idea della dimostrazione è quella di distribuire il costo (unitario) di ogni sottoinsieme della soluzione C sugli elementi che esso aggiunge alla copertura.

Nell'esempio di figura, 1.4 l'insieme S_1 aggiunge alla copertura 6 nuovi elementi, e quindi il peso di ogni singolo elemento aggiunto alla copertura da S_1 sarà $\frac{1}{6}$, il peso degli elementi aggiunti da S_4 sarà $\frac{1}{3}$ e così via. Ad ogni elemento dell'insieme X viene dunque assegnato un peso $p_j, j = 1, ..., n$.

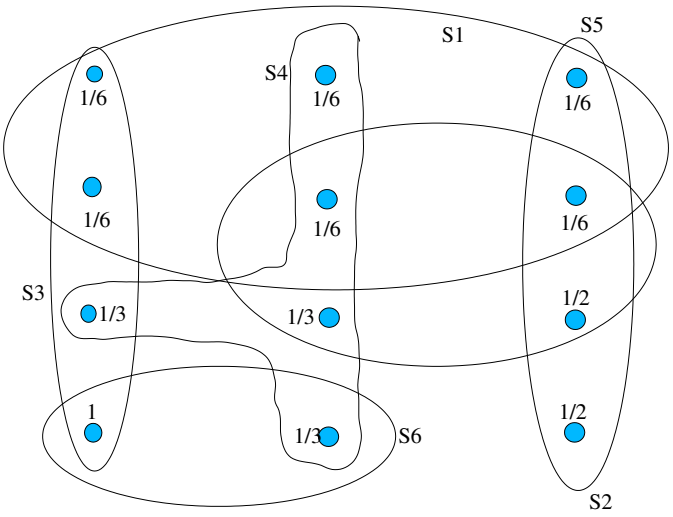

Figura 1.4: Esempio di *SetCovering*. La soluzione fornita dall'algoritmo GREEDY-SET-COVER è $C = \{S_1, S_4, S_5, S_6\}$. Nella figura è riportata anche la pesatura indotta dalla soluzione C.

Per definizione il valore della soluzione $|C|$ sarà dato da:

$$|C| = \sum_{j \in X} p_j \tag{1.7}$$

dove, essendo i l'indice dell'insieme S_i che include l'elemento j nella copertura,

$$p_j = \frac{1}{|S_i \setminus (\bigcup_{k=1}^{i-1} S_k)|} \tag{1.8}$$

Si consideri ora la soluzione ottima C^*. Essa, essendo una soluzione ammissibile, comprende tutti gli elementi di X almeno una volta. Si ha dunque che:

$$|C| = \sum_{j \in X} p_j \leq \sum_{s \in C^*} \sum_{j \in S} p_j \tag{1.9}$$

poiché uno stesso elemento di X può appartenere a più elementi di C^*. Il risultato chiave è dunque il seguente:

Lemma 3. *Sia S un qualsiasi elemento di F. Allora la pesatura indotta dalla soluzione C dell'algoritmo* GREEDY-SET-COVER *soddisfa la sequente disuguaglianza:*

$$\sum_{j \in S} p_j \leq H(|S|) \tag{1.10}$$

dove $H(x)$ è la serie armonica arrestata ad x, cioè $H(x) = \sum_{j=1}^{x} \frac{1}{j}$.

Dimostrazione. Sia $u_i = |S - (S_1 \cup ... \cup S_i)|$ la quantità di elementi di S scoperti dopo che gli insiemi $S_1, ...S_i$ sono stati scelti dall'algoritmo. La differenza $u_{i-1} -$

u_i è quindi la quantità di elementi di S scelti nell'iterazione i. u_0 è quindi $|S|$. Sia k la prima iterazione in cui $u_k = 0$, ovvero nella quale sono stati scelti gli ultimi elementi di S rimasti scoperti. Ovviamente $u_{i-1} \geq u_i$. Si noti che

$$\sum_{j \in S} p_j = \sum_{j=1}^{k} \frac{u_{i-1} - u_i}{|S_i - (S_1, ..., S_{i-1})|} \leq \sum_{j=1}^{k} \frac{u_{i-1} - u_i}{|S - (S_1, ..., S_{i-1})|} \tag{1.11}$$

poiché, se all'iterazione i l'algoritmo sceglie l'insieme S allora si ha l'uguaglianza, mentre se l'algoritmo sceglie un altro insieme, esso ricopre un numero maggiore di elementi, e quindi il denominatore con S al posto di S_i è più piccolo. Si ha dunque la seguente catena di disuguaglianze:

$$\begin{aligned}
\sum_{j \in S} p_j &\leq \sum_{j=1}^{k} \frac{u_{i-1} - u_i}{|S - (S_1, ..., S_{i-1})|} \\
&= \sum_{j=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{u_{i-1}} \\
&= \sum_{j=1}^{k} \sum_{j=u_i+1}^{u_{i-1}} \frac{1}{u_{i-1}} \\
&\leq \sum_{j=1}^{k} \sum_{j=u_i+1}^{u_{i-1}} \frac{1}{j} \\
&= \sum_{i=1}^{k} \Big(\sum_{j=1}^{u_{i-1}} \frac{1}{j} - \sum_{j=1}^{u_i} \frac{1}{j} \Big) \\
&= \sum_{i=1}^{k} (H(u_{i-1}) - H(u_i)) \\
&= H(u_0) - H(u_k) \\
&= H(u_0) - H(0) \\
&= H(u_0) \\
&= H(|S|)
\end{aligned} \tag{1.12}$$

□

Corollario 1. *Il fattore di approssimazione dell'algoritmo* Greedy-Set-Cover *è:*

$$\rho(n) = H(\max\{|S| : S \in F\}) = H(|S^*|). \tag{1.13}$$

Dimostrazione. Grazie al Lemma 3 si ha:

$$|C| = \sum_{j \in X} p_j \leq \sum_{S \in C^*} \sum_{j \in S} p_j \leq \sum_{S \in C^*} H(S) \leq |C^*| H(|S^*|) \tag{1.14}$$

e quindi

$$\frac{|C|}{|C^*|} \leq H(|S^*|) = \rho(n) \tag{1.15}$$

□

Siccome a priori la cardinalità dell'insieme più grande non è limitata e siccome la serie armonica è asintotica al logaritmo di n per n grande, nella pratica GREEDY-SET-COVER ha un fattore di approssimazione pari a $log(n)$.

Infine, l'algoritmo può essere facilmente esteso al caso in cui ogni elemento di F ha un peso (*Weighted Set Covering*): la copertura dovrà essere quella di peso complessivo minimo.

1.5 Il problema del *Vertex Cover*

Si consideri un grafo $G = (V, E)$ non orientato, con dei pesi $w_i \geq 0, \forall i \in V$ sui vertici del grafo. Si vuole trovare un insieme di nodi $S^* \subseteq V$ tale che:

- Tutti i lati del grafo sono incidenti in almeno un vertice di S^*, cioè $\forall (i, j) \in E, i \vee j \in S^*$
- $W(S^*) = \sum_{i \in S^*} w_i$ è minimo tra tutte le soluzioni ammissibili.

Ad esempio, si vuole presidiare tutte le vie di una città minimizzando il costo dei *checkpoint*: le vie sono i lati di un grafo mentre le posizioni dei *checkpoint* si scelgono tra i vertici.

A ben riflettere, questo problema è riconducibile al *Weighted-Set-Covering*, se si considera la stella uscente S_i da ciascun vertice i come insieme F e l'insieme degli archi E come insieme X.

In realtà è possibile trovare un algoritmo il cui fattore di approssimazione è costante, pari a 2, inventato *ad hoc* per il problema di *Vertex Covering*, sfruttando il fatto di lavorare su un grafo.

Il principio è il seguente: si cerca di trovare una pesatura (*pricing*) per i lati del grafo, funzione della pesatura iniziale sui vertici, che abbia leache seguenti proprietà:

- Per ogni arco, $p_e \geq 0$,
- Per ogni vertice del grafo, la somma dei pesi degli archi incidenti è minore o uguale del peso di quel vertice, cioè:$\forall i \in V, \sum_{e \in S_i} p_e \leq w_i$.

Data quindi una pesatura degli archi, la somma dei pesi sugli archi è un *lower bound* della soluzione ottima S^*. Infatti:

$$\sum_{e \in E} p_e \leq \sum_{i \in S^*} \sum_{e \in S_i} p_e \leq \sum_{i \in S^*} w_i \tag{1.16}$$

Definizione 6. *Si definisce un vertice i* pagato *se $\sum_{e \in S_i} p_e = w_i$.*

L'algoritmo proposto (VERTEX-COVER-APPROX) lavora quindi nel seguente modo: si considerano in sequenza una sola volta tutti i lati del grafo (non importa l'ordine) e, partendo dal prezzo nullo per tutti gli archi, si alzano i pesi degli archi il più possibile fino a pagare almeno un nodo incidente.

Si consideri l'esempio di figura 1.5: si scorrono gli archi in ordine lessicografico. Al primo passo si pesa l'arco $(1, 2)$ con un peso $p_{1,2} = 2$, in modo da pagare il nodo 2. Si passa poi all'arco $(1, 6)$ e si assegna un peso $p_{1,6} = 1$ in modo da pagare il nodo 1 e così via.

Per il dettaglio dei pesi assegnati dall'algoritmo si veda la tabella 1.1.

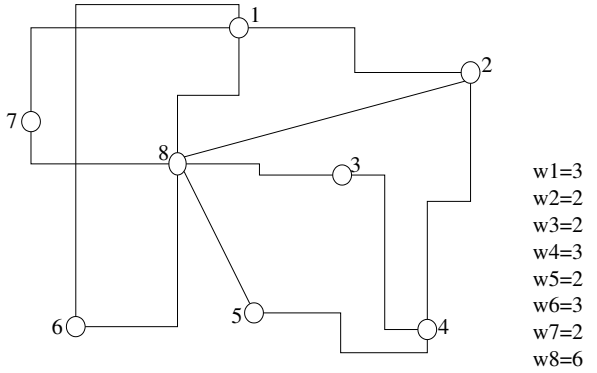

Figura 1.5: Esempio di *Vertex Covering*. A lato sono i riportati i pesi dei nodi.

arco	*peso assegnato*
(1,2)	2
(1,6)	1
(1,7)	0
(1,8)	0
(2,4)	0
(2,8)	0
(3,8)	2
(3,4)	0
(4,5)	2
(4,6)	1
(6,8)	1
(7,8)	2

Tabella 1.1: I pesi assegnati dall'algoritmo VERTEX-COVER-APPROX nell'esempio di figura 1.5

La soluzione sarà composta dall'insime dei vertici pagati $S = \{7, 1, 2, 3, 5, 4, 6\}$. Si noti che sono stati inclusi ben 7 degli 8 nodi del grafo ma è stato escluso il nodo 8, che quello con peso maggiore.

Lo pseudocodice dell'algoritmo è il seguente:

VERTEX-COVER-APPROX$(G(V, E), W)$
1 **for each** $e \in E, p[e] \leftarrow 0$
2 $S \leftarrow \emptyset$
3 **while** $\exists (i, j) \in E$ tale che né i né j sono pagati,
$\triangleright$ Aumenta p_e fino al massimo possibile,
4 **do** $p_e \leftarrow p_e + \min\{w_j - \sum_{e \in S_i} p_e, w_j - \sum_{e \in S_j} p_e\}$
5 $S \leftarrow S \cup \{$i nodi pagati al passo 4$\}$
6 **return** S

1.5.1 Valutazione del fattore di approssimazione dell'algoritmo VERTEX-COVER-APPROX

Lemma 4. VERTEX-COVER-APPROX *ha grado di approssimazione 2.*

Dimostrazione. Siano S ed S^*, rispettivamente, la soluzione ottenuta con l'algoritmo VERTEX-COVER-APPROX e la soluzione ottima. In particolare, S^* è copertura di vertici se

$$\sum_{e \in E} p_e \leq \sum_{i \in S^*} w_i. \tag{1.17}$$

Inoltre, si ricorda che un nodo è pagato se

$$\sum_{e \in E_i} p_e = w_i, \tag{1.18}$$

dove $E_i \subseteq E$ è l'insieme degli archi incidenti nel nodo i. Al fine di dimostrare che l'algoritmo ha grado di approssimazione 2, è necessario provare che

$$\sum_{i \in S} w_i \leq 2 \sum_{e \in E} p_e.$$

Quindi si cerca una maggiorazione di $\sum_{i \in S} w_i$. Dalla (1.18), si ha che

$$\sum_{i \in S} w_i = \sum_{i \in S} \sum_{e \in E_i} p_e \leq 2 \sum_{e \in E} p_e,$$

dove la disuguaglianza è dovuta al fatto che, al limite (caso pessimo), con il termine $\sum_{e \in E_i} p_e$ si contano due volte gli archi. A questo punto, utilizzando la (1.17), la precedente diventa

$$\sum_{i \in S} w_i \leq 2 \sum_{i \in S^*} w_i.$$

Di conseguenza, $\rho = 2$. □

1.5.2 Un approccio basato sulla programmazione lineare

Si consideri la seguente formulazione di programmazione lineare intera del problema di *Vertex Cover*. Definendo la variabile binari

$$x_i = \begin{cases} 1 & \text{se } i \in S, \\ 0 & \text{altrimenti,} \end{cases}$$

la formulazione del problema diventa:

$$\min \sum_{i \in V} w_i x_i, \tag{1.19}$$

con i vincoli:

$$x_i + x_j \geq 1 \quad \forall (i,j) \in E \text{ con } i \neq j \tag{1.20}$$

$$x_i \in \{0,1\} \quad \forall i \in V. \tag{1.21}$$

Per la risoluzione di questo problema si utilizza un algoritmo euristico basato sul rilassamento lineare. In questo modo, la variabile intera x_i diventa continua, ossia $x_i \in [0,1]$. Considerata la soluzione ottima del rilassato continuo, l'algoritmo ha la seguente forma:

```
1  for each i ∈ V
2        do if x_i ≥ 1/2
3              then S ← S ∪ {i}
```

Si noti che per ogni vincolo (1.20), si garantisce che il valore di almeno una delle due variabili è almeno 1/2. Di conseguenza, il vincolo (1.20) è soddisfatto e la correttezza dell'algoritmo è assicurata, cioè almeno uno dei vertici estremi di un lato appartiene alla soluzione.

Per quanto riguarda invece la valutazione del fattore di approssimazione, si procede nel seguente modo. Innanzitutto si osservi che, definita w_{LP} la soluzione del problema rilassato,

$$w_{LP} = \min \sum_{i \in V} w_i x_i \leq w^*, \tag{1.22}$$

dove $w^* = \sum_{i \in S^*} w_i$ è il valore della soluzione ottima. Inolte, al minimo,

$$w_{LP} \geq \frac{1}{2} w, \tag{1.23}$$

dove $w = \sum_{i \in S} w_i$ è il valore della soluzione ottenuta con l'algoritmo approssimato. Questo è vero perchè al minimo il rilassamento continuo impone $x_i = 1/2 \quad \forall i \in S$. Ora, unendo la (1.22) e la (1.23), si ottiene

$$\frac{1}{2} w \leq w^* \quad \Longrightarrow \quad \rho = 2.$$

Notiamo infine che, al momento, non esiste un algoritmo approssimato per il problema della copertura di vertici con grado di approssimazione minore di 2. La ricerca di un algoritmo con questa caratteristica è un problema ancora aperto.

1.6 *PTAS* per il problema dello zaino binario

Dati n oggetti, con valore v_i e peso w_i $(i = 1, ..., n)$, ed assegnata una capacità w, si cerca un sottoinseme $S \subseteq \{1, ..., n\}$ tale che

$$\sum_{i \in S} w_i \leq w,$$

e che massimizzi $\sum_{i \in S} v_i$. Considerando il sottoinsieme $\{1, ..., i\}$, la somma dei suoi pesi è data da $\overline{w}$. Se si riesce a costruire un appropriato grafo aciclico, il problema può essere risolto in termini di cammino minimo.

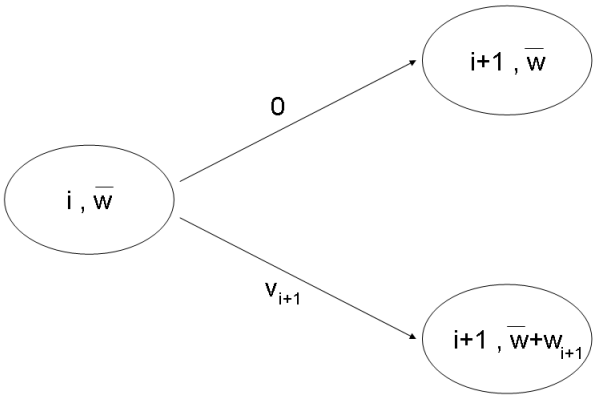

Figura 1.6: Generico stato i.

Il generico stato del grafo sarà costituito dalla coppia $(i, \overline{w})$, nella quale i rappresenta l'i-esimo oggetto e $\overline{w}$ è il peso accumulato fino a quel punto. Di conseguenza, gli stati iniziale e finale sono, rispettivamente, $(0, 0)$ e (n, w).

A questo punto il problema diventa quello di trovare il cammino minimo da $(0, 0)$ a (n, w) e la complessità dell'algoritmo che lo risolve, essendo il grafo aciclico, è $\mathcal{O}(nw)$.

In alternativa, come generico stato del grafo si può considerare la coppia $(i, \bar{v})$, dove $\bar{v}$ rappresenta il valore accumulato fino a quel punto.

In particolare, il peso di ogni arco rappresenta il costo di transizione tra i due stati.

In questo grafo il valore v è dato da $v = \sum_{i \in S} v_i$. Cercare l'albero dei cammini minimi significa raggingere le foglie dalla radice $(0, 0)$ scegliendo il cammino che ha valore massimo ma costo inferiore o uguale a w. La complessità dell'algoritmo è $\mathcal{O}(nv)$ (non polinomiale dal momento che dipende da v che è ingonita del problema), che, stimando il valore v con $v \approx nv^*$, dove $v^* = \max v_i$, diventa $\mathcal{O}(n^2 v^*)$.

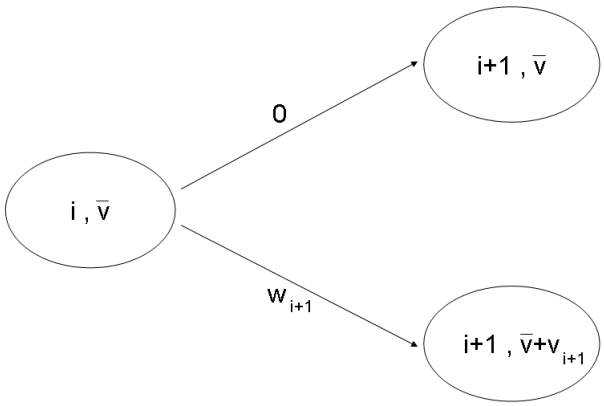

Figura 1.7: Generico stato i.

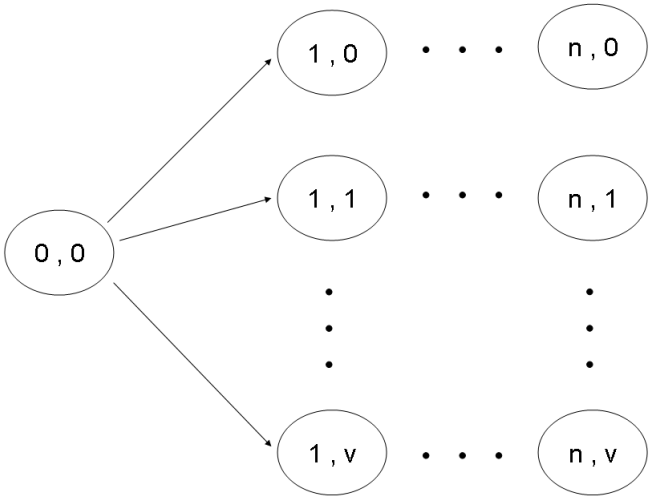

Figura 1.8: Grafo aciclico degli stati.

Una possibile strategia per ridurre la complessità dell'algoritmo e per renderlo approssimato è quella di dividere i valori v_i per una quantità fissata, ad esempio 10, e scartare le unità. In questo modo si riduce notevolmente la dimensione del grafo. Se ε è il fattore di approssimazione richiesto, si riscalano tutte le quantità rispetto a questo fattore per mezzo della trasformazione

$$v = \frac{\varepsilon}{n} v^*. \tag{1.24}$$

Di conseguenza, i valori v_i diventano

$$\tilde{v}_i = \lceil \frac{v_i}{v} \rceil v. \tag{1.25}$$

Questi valori $\tilde{v}_i$ sono detti *riposizionamento* di v_i e sono tutti multipli di v. Introduciamo inoltre i valori

$$\hat{v}_i = \lceil \frac{\tilde{v}_i}{v} \rceil, \tag{1.26}$$

$$\hat{v}^* = \lceil \frac{v^*}{v} \rceil = \lceil \frac{v^* n}{v^* \varepsilon} \rceil = \frac{n}{\varepsilon}. \tag{1.27}$$

Il valore $\hat{v}^*$ rappresenta la dimensione del nuovo grafo ("*grafo riscalato*"). Quindi, grazie alla (1.27), la complessità dell'algoritmo diventa polinomiale, ossia

$$\mathcal{O}(n^2 \hat{v}^*) = \mathcal{O}\left(\frac{n^3}{\varepsilon}\right). \tag{1.28}$$

KNAPSACK-APPROX(ε)
1 $v \leftarrow \frac{\varepsilon}{n} v^*$
2 **for** $i \leftarrow 1$ **to** n
3 **do** $\hat{v}_i = \lceil \frac{v_i}{v} \rceil$
4 $S \leftarrow$ KNAPSACK$(\hat{v}_i, w_i, w)$
5 **return** S

1.6.1 Valutazione del fattore di approssimazione dell'algoritmo KNAPSACK-APPROX(ε)

Siano S^* ed S, rispettivamente, una soluzione ammissibile e la soluzione fornita dall'algoritmo KNAPSACK-APPROX(ε). S^*, essendo soluzione ammissibile, soddisfa la relazione

$$\sum_{i \in S^*} w_i \leq w. \tag{1.29}$$

Inoltre valgono le seguenti

$$v_i \leq \tilde{v}_i \leq v_i + v, \tag{1.30}$$

$$\sum_{i \in S} v_i \geq v^* = \tilde{v}^*, \tag{1.31}$$

dove l'uguaglianza è motivata dal fatto che v^* è invariante alla trasformazione (1.25). Ora, limitando inferiormente e superiormente $\sum_{i \in S} \tilde{v}_i$ ed utilizzando la

(1.30), si trova

$$\sum_{i\in S} \tilde{v}_i \geq \sum_{i\in S^*} \tilde{v}_i \geq \sum_{i\in S^*} v_i, \tag{1.32}$$

$$\sum_{i\in S} \tilde{v}_i \leq \sum_{i\in S} (v_i + v) \leq \sum_{i\in S} v_i + nv. \tag{1.33}$$

Così, unendo la (1.32) e la (1.33), si ottiene

$$\sum_{i\in S^*} v_i \leq \sum_{i\in S} v_i + nv. \tag{1.34}$$

A questo punto, utilizzando la (1.24) e la (1.31), si ottiene la seguente stima per la quantità nv:

$$nv \leq \varepsilon \sum_{i\in S} v_i. \tag{1.35}$$

Di conseguenza la (1.34) diventa

$$\sum_{i\in S^*} v_i \leq \sum_{i\in S} v_i + \varepsilon \sum_{i\in S} v_i \quad \Longrightarrow \quad \rho = 1 + \varepsilon.$$

1.7 Il problema del commesso viaggiatore

Nel problema del commesso viaggiatore, dato un grafo completo non orientato $G = (V, E)$ (cioè, un grafo non orientato in cui ogni coppia di vertici è adiacente) con $n = |V|$ nodi e con una distanza intera non negativa $d(i, j)$ associata a ciascun arco $(i, j)\in E$, si deve trovare un *giro* che passi una sola volta per ogni nodo e termini nel nodo di partenza (*ciclo Hamiltoniano*) di distanza minima.

L'obiettivo è quello di mostrare che il problema del commesso viaggiatore non ammette un algoritmo approssimato con tempo polinomiale e con fattore di approssimazione ρ costante a meno che $P = NP$.

Teorema 1. *Se $P\neq NP$ e $\rho > 1$, allora non esiste alcun algoritmo approssimato con tempo polinomiale e grado di approssimazione ρ per il problema del commesso viaggiatore.*

Dimostrazione. Si supponga per assurdo che, per qualche $\rho > 1$, ci sia un algoritmo approssimato A con tempo polinomiale e fattore di approssimazione ρ. Se si riesce a mostrare che l'algoritmo A risolve istanze del problema del ciclo Hamiltoniano, si prova che questo problema è risolubile in tempo polinomiale. A questo punto, però, essendo questo problema *NP-completo*, risolverlo in tempo polinomiale implica che $P = NP$.

Sia $G = (V, E)$ un'istanza del problema del ciclo Hamiltonano. Per determinare se G contiene un ciclo Hamiltoniano utilizzando l'algoritmo approssimato A, si trasforma G in un'istanza del problema del commesso viaggiatore nel seguente modo. Sia $G' = (V, E')$ il grafo completo su V, cioè $E' = \{(i, j) : i, j\in V \text{ e } i\neq j\}$, e si assegni una distanza a ciascun arco in E' come segue:

$$d(i, j) = \begin{cases} 1 & \text{se } (i, j)\in E, \\ n\rho + 1 & \text{altrimenti.} \end{cases}$$

Le rappresentazioni di G' e d possono essere create da una rappresentazione di G in tempo polinomiale rispetto a n ed $|E|$.

Si consideri ora il problema del commesso viaggiatore (G', d). Si supponga che tale algoritmo approssimato esista e risolviamo il problema con grado di approssimazione $\rho^{'} < \rho$

G ha cicli Hamiltoniani se e solo se A restituisce un valore minore o uguale a $(n\rho^{'} + 1) + (n-1) = n\rho^{'} + n < n\rho + n$. Di conseguenza, si può utilizzare l'algoritmo A per risolvere il problema del ciclo Hamiltoniano in tempo polinomiale. Assurdo, poiché è un problema *NP-completo.* □