ASD2 Homework 1

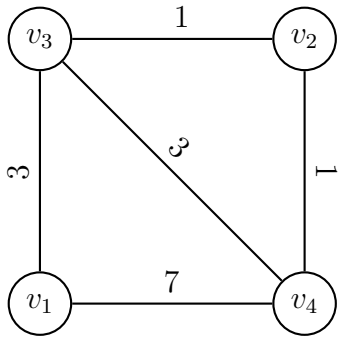
Valerio Pro

Novembre 2021

Esercizio 1

1.1

Si consideri il seguente grafo istanza del problema *Max Cut*



Per tale grafo l'*Algoritmo 1* non trova una soluzione ottima, infatti:

- al passo iniziale dell'algoritmo avremo $S_1 = \{v_1\}$ e $S_2 = \{v_2\}$
- nella prima iterazione del ciclo for il nodo v_3 è aggiunto all'insieme S_2 (perchè $w(v_3, v_2) < w(v_3, v_1)$)
- nella seconda ed ultima iterazione il nodo v_4 è aggiunto a S_2 (perchè $w(v_4, v_3) + w(v_4, v_2) < w(v_4, v_1)$)

L'algoritmo restituisce quindi il taglio (S_1, S_2) con $S_1 = \{v_1\}$ e $S_2 = \{v_2, v_3, v_4\}$ di capacità pari a 10. Il taglio di capacità massima è dato invece da $S_1 = \{v_1, v_3\}$ e $S_2 = \{v_2, v_4\}$ oppure da $S_1 = \{v_1, v_2, v_3\}$ e $S_2 = \{v_4\}$.

1.2

Il rapporto di approssimazione può essere dimostrato facendo delle semplici osservazioni su come l'algoritmo costruisce il taglio. Consideriamo innanzitutto cosa succede all'i-esima iterazione dell'algoritmo: il nodo v_i è aggiunto all'insieme S_1 se la somma dei pesi degli archi che vanno da v_i in S_1 è minore o uguale alla somma dei pesi degli archi che vanno da v_i in S_2, altrimenti v_i è inserito in S_2. L'osservazione chiave riguarda il fatto che mentre stiamo "analizzando" il nodo v_i, all'interno degli insiemi S_1 e S_2 troviamo nodi che compaiono prima nell'ordinamento rispetto v_i, ovvero nodi che hanno indice più piccolo del nodo v_i; l'algoritmo decide in che insieme inserire v_i guardando solo questi nodi con indice minore e senza considerare i restanti nodi di indice più grande.

Preso quindi il taglio (S_1, S_2) restituito dall'algoritmo, quello che possiamo dire è:

$$\forall v_i \in S_1, \sum_{\substack{v_j \in S_1 \\ (v_i,v_j)\in E \\ j<i}} w(v_i, v_j) \leq \sum_{\substack{v_j \in S_2 \\ (v_i,v_j)\in E \\ j<i}} w(v_i, v_j) \tag{1}$$

Questa disuguaglianza rispecchia il comportamento dell'algoritmo precedentemente descritto[1], lo stesso vale per tutti i nodi in S_2:

$$\forall v_i \in S_2, \sum_{\substack{v_j \in S_2 \\ (v_i,v_j)\in E \\ j<i}} w(v_i, v_j) \leq \sum_{\substack{v_j \in S_1 \\ (v_i,v_j)\in E \\ j<i}} w(v_i, v_j) \tag{2}$$

La dimostrazione continua similmente alla dimostrazione di approssimazione dell'algoritmo di *Local Search per Max Cut*. Si prosegue sommando su tutti i v_i entrambi i membri delle due disuguaglianze, ottenendo la (3) e la (4):

$$\sum_{v_i \in S_1} \sum_{\substack{v_j \in S_1 \\ (v_i,v_j)\in E \\ j<i}} w(v_i, v_j) \leq \sum_{v_i \in S_1} \sum_{\substack{v_j \in S_2 \\ (v_i,v_j)\in E \\ j<i}} w(v_i, v_j) \tag{3}$$

[1]Si osservi che non è possibile generalizzare rimuovendo il vincolo $j < i$ dalle sommatorie, l'esempio presentato nel punto 1.1, con il rispettivo taglio trovato dall'algoritmo, mostra che la disuguaglianza più generale senza $j < i$ non reggerebbe, ad esempio, per il nodo v_3

$$\sum_{v_i \in S_2} \sum_{\substack{v_j \in S_2 \\ (v_i,v_j)\in E \\ j<i}} w(v_i, v_j) \leq \sum_{v_i \in S_2} \sum_{\substack{v_j \in S_1 \\ (v_i,v_j)\in E \\ j<i}} w(v_i, v_j) \tag{4}$$

Può essere importante soffermarsi un attimo sulla disuguaglianza (3) per notare delle differenze con la dimostrazione di approssimazione dell'algoritmo di Local Search per Max Cut, infatti, in questo caso, nel membro a sinistra della disuguaglianza sono contati tutti gli archi in $E \cap \binom{S_1}{2}$ una e una sola volta (e non due). Intuitivamente questo accade perchè fissato un v_i nella sommatoria più esterna, andiamo a contare tutti gli archi che vanno da v_i verso nodi $v_j, \ldots, v_k$ (dello stesso insieme) di indice minore e tali archi non verranno riconsiderati quando fisseremo nella sommatoria esterna i nodi $v_j, \ldots, v_k$. Le stesse considerazioni valgono per la (4). Nel membro a destra della disuguaglianza (3) invece, non stiamo contando gli archi di tutto il taglio ma solo una porzione, la porzione complementare del taglio è contata nel membro a destra della disuguaglianza (4).

Passiamo quindi a fare gli ultimi passaggi, sommando (3) e (4) si ottiene:

$$C(E \cap \binom{S_1}{2}) + C(E \cap \binom{S_2}{2}) \leq C(S_1, S_2) \tag{5}$$

Sommando a entrambi i membri $C(S_1, S_2)$ e riscrivendo la disuguaglianza:

$$\frac{1}{2} \leq \frac{C(S_1, S_2)}{C(E)} \tag{6}$$

Ma dato che $C(E) \geq OPT(I)$[2]:

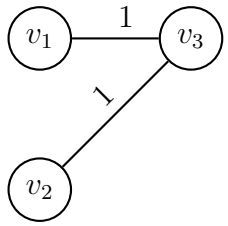
$$\frac{1}{2} \leq \frac{C(S_1, S_2)}{C(E)} \leq \frac{C(S_1, S_2)}{OPT(I)} \tag{7}$$

Si ottiene quindi il rapporto di $\frac{1}{2}$-*approssimazione* dell'*Algoritmo 1* per Max Cut. La capacità del taglio restituito dall'algoritmo è almeno pari alla metà della capacità del taglio ottimo.

[2]OPT(I) è il valore ottimo associato all'istanza I di Max Cut

1.3

Per mostrare che l'analisi dell'algoritmo fatta nel punto precedente è "stretta" e non può essere migliorata si può considerare la seguente istanza di *Max Cut* in cui il costo della soluzione trovata dall'algoritmo è esattamente pari alla metà del costo della soluzione ottima:



L' algoritmo trova il taglio indotto da $(\{v_1, v_3\}, \{v_2\})$ di capacità pari ad 1, ma il taglio ottimo ha capacità 2 ed è dato da $(\{v_1, v_2\}, \{v_3\})$.

Esercizio 2

Per questo problema si vorrebbe un algoritmo che operi in tempo polinomiale nella dimensione dell'input e che restituisca una soluzione ottima. Ho provato a ragionare su diversi approcci, soprattutto di programmazione dinamica dato che il problema sembra essere simile a *Subset Sum* (per il quale si conosce un algoritmo di programmazione dinamica pseudo polinomiale), ma non sono riuscito a definire il valore ottimo associato alla generica sottoistanza del problema. Infatti non ho trovato modo di utilizzare il rapporto $\frac{s(A)}{s(B)}$ come parametro per la definizione dell'ottimo e quindi come parametro per la costruzione e iterazione della matrice, questo perchè $\frac{s(A)}{s(B)}$ non è necessariamente un intero e non consente di individuare e selezionare precisamente le entrate della matrice. Allo stesso modo non sono riuscito a definire una riduzione a *Subset Sum* che avrebbe almeno consentito di risolvere all'ottimo il problema in tempo pseudo polinomiale.

Ho definito allora un semplice algoritmo greedy e ho poi tentato di mostrarne il rapporto di approssimazione.

Ho assunto per semplicità di avere una rappresentazione indicizzata dell'insieme X, inoltre con n verrà indicato il numero di elementi dell'insieme X. L'algoritmo greedy ordina in senso non crescente gli elementi dell'insieme X, determina poi la coppia di elementi in posizioni i, j (con $X[i] > X[j]$) con rapporto minimo e aggiunge l'elemento $X[i]$ all'insieme A e l'elemento $X[j]$ all'insieme B. Dopodichè vengono aggiunti all'insieme B tutti quegli elementi $X[k]$ (con $X[j] > X[k]$) che possono migliorare il rapporto tra le somme degli insiemi ma senza sforare il valore $X[i] = s(A)$.

Algorithm 1 Greedy

Require: Un insieme $X \subset \mathbb{N}$

Ensure: Due insiemi A e B tali che $A \subset X$, $B \subset X$, $A \cap B = \emptyset$, $s(A) \geq s(B)$

```
A ← ∅
B ← ∅
min ← ∞
Ordina X in senso non crescente X[1] > X[2] > ··· > X[n]

indexA ← −1
indexB ← −1
for i = 1 to n − 1 do
    if X[i]/X[i+1] < min then
        min ← X[i]/X[i+1]
        indexA ← i
        indexB ← i + 1
    end if
end for

A ← {X[indexA]}
B ← {X[indexB]}
k ← indexB + 1
while k ≤ n do
    if s(A) ≥ s(B) + X[k] then
        B ← B ∪ {X[k]}
    end if
    k ← k + 1
end while

return (A, B)
```

Il primo ciclo *for* è utilizzato per trovare la coppia di interi di rapporto minimo, vengono memorizzati anche gli indici di tale coppia di elementi. Si sfrutta il fatto che X è ordinato e quindi basta considerare coppie adiacenti di interi. Dopo che i due elementi vengono aggiunti in A e in B si cercano altri interi più piccoli dell'elemento in B per migliorare eventualmente il costo della soluzione.

A livello di complessità l'algoritmo ha Running Time:

$$T(n) = \Theta(n \log n)$$

Infatti l'ordinamento iniziale richiede $\Theta(n \log n)$, trovare la coppia di rapporto minimo richiede $\mathcal{O}(n)$ iterazioni e il ciclo *while* richiede al più tempo $\mathcal{O}(n)$.

L'algoritmo trova sempre una soluzione ammissibile del problema, poichè restituisce due sottoinsiemi A e B non vuoti e disgiunti che rispettano il vincolo $s(A) \geq s(B)$, tuttavia non trova sempre la soluzione ottima. Di seguito un esempio di istanza sulla quale l'algoritmo *Greedy* non trova l'ottimo:

$$X = \{28, 17, 3, 2\}$$

L'algoritmo *Greedy* restituisce $A = \{3\}$ e $B = \{2\}$ con un rapporto $\frac{s(A)}{s(B)} = 1.5$, mentre la soluzione ottima è $A^* = \{28\}$ e $B^* = \{17, 3, 2\}$ con rapporto $\frac{s(A^*)}{s(B^*)} = 1.27$.

Vediamo ora cosa si può dire sulla soluzione proposta dall'algoritmo in relazione all'ottimo. Con $Cost(I, S^A(I))$ si indica il costo della soluzione *Greedy* sull'istanza I e sulla soluzione ammissibile $S^A(I)$ per quell'istanza. Per tutte quelle istanze in cui all'interno del vettore X sono presenti due interi il cui rapporto è minore o uguale a 2, allora l'algoritmo è *2-approssimante*, questo perchè:

1) $OPT(I) \geq 1$ (l'ottimo incontra questo lower bound in 1 quando ci sono due sottoinsiemi disgiunti di uguale somma);

2) $Cost(I, S^A(I)) \leq 2$, poichè esistono due interi *X[i]* e *X[j]*, con $X[i] > X[j]$, tali che $X[j] \geq \frac{X[i]}{2}$;

3) $Cost(I, S^A(I)) \geq OPT(I)$, dato che è un problema di minimo.

Si ha quindi l'approssimazione per questo tipo di istanze:

$$\frac{Cost(I, S^A(I))}{OPT(I)} \leq 2$$

Per il caso più difficile, cioè quando nell'istanza non sono presenti due interi il cui rapporto è minore o uguale a 2, non sono riuscito a dimostrare alcuna approssimazione. Ho provato a cercare controesempi per smentire la *2-approssimazione* ma non ho trovato nulla. Tuttavia ho comunque il sospetto che in questo caso l'approssimazione dipenda dagli interi all'interno di X e che quindi non è in generale a fattore costante.

Esercizio 3

Gli esercizi implementativi sono stati realizzati in Python e sono suddivisi nei vari punti 3.1, 3.2 e 3.3. Ho commentato ogni script, cercando di spiegarne al meglio il funzionamento e i contenuti. Nei paragrafi seguenti cercherò di concentrarmi principalmente solo sulle scelte più importanti fatte, al fine di non risultare troppo pedante nella spiegazione dei dettagli implementativi. Sia l'algoritmo greedy che quello esaustivo prendono in input il nome del file sul quale sono presenti i parametri (file *input.txt*).

3.1

Nell'implementazione dell'algoritmo greedy per Load Balancing ho utilizzato una coda con priorità per selezionare ad ogni passo la macchina con carico minimo, la quale verrà reinserita subito dopo ma chiaramente con carico maggiore. All'interno della coda, la priorità di una macchina i è quindi data dal carico totale L_i assegnato alla macchina stessa, inizialmente tale carico sarà pari a 0 per ogni macchina.

3.2

Ho implementato l'algoritmo esaustivo utilizzando la tecnica *Branch-And-Bound*, i punti principali sono i seguenti:

- *Modellazione del Problema:* il generico problema di Load Balancing è costituito dal numero di macchine m (informazione statica che non cambia mai), un dizionario che associa ad ogni macchina una lista dei tempi dei task correntemente schedulati e un altro dizionario che associa ad ogni task ancora da schedulare il suo tempo di esecuzione;
- *Espansione in SottoProblemi:* dato il problema corrente P, per ogni task che deve essere ancora schedulato in P si generano m sottoproblemi, assegnando il task ad ognuna delle m macchine. Quindi per ogni problema si hanno al più $n*m$ sottoproblemi;
- *Lower Bound:* il lower bound che ho utilizzato è molto semplice e consiste del carico più grande L_h tra le m macchine. Se il carico più grande L_h supera il best corrente, allora il sottoproblema può essere scartato.

Il best iniziale è dato dalla soluzione restituita dall'algoritmo greedy, l'algoritmo esaustivo migliorerà (se possibile) tale soluzione.

L'algoritmo considera tutte le possibili assegnazioni dei task alle macchine, tagliando le ramificazioni che non possono portare ad una soluzione migliore di quella corrente, tuttavia c'è un problema che non sono riuscito a risolvere: per come ho definito l'espansione in sottoproblemi, nei casi in cui ci sono nell'istanza più task con stessi tempi di esecuzione, possono venire a crearsi sottoproblemi che sono "identici" tra loro rispetto le assegnazioni dei task alle

macchine e rispetto i tempi di esecuzione dei task ancora da schedulare. Per alcune istanze, ottimizzare questo aspetto avrebbe consentito di analizzare meno sottoproblemi.

L'algoritmo in generale non è molto veloce, rendere leggermente più complessa l'istanza porta ad avere un'esplosione nel numero di sottoproblemi considerati.

3.3

Ho provato a stimare empiricamente il rapporto di approssimazione attraverso una funzione che, dato un certo numero intero t, esegue t volte l'algoritmo esaustivo del punto precedente su istanze generate randomicamente ad ogni iterazione. Vengono sommati tutti i rapporti di approssimazione ottenuti e divisi per il numero t di iterazioni, ottenendo quindi una certa stima. Sono stati ottenuti i seguenti risultati per diversi valori di *t, m, n* e *task time*.

Stime Rapporto di Approssimazione		
Numero di Iterazioni	**Parametri Input**	**Stima**
t = 50	$m \in [2,4]$, $n \in [5,7]$, $task\ time \in [1,250]$	1.006776
t = 100	$m \in [2,3]$, $n \in [4,6]$, $task\ time \in [1,250]$	1.003611
t = 500	$m = 2$, $n = 5$, $task\ time \in [1,250]$	1.015431
t = 1.000	$m = 2$, $n = 5$, $task\ time \in [1,250]$	1.014493
t = 10.000	$m = 2$, $n = 5$, $task\ time \in [1,250]$	1.013213

Ho voluto mantenere $m \geq 2$ e $n > m$ perchè l'algoritmo greedy ottiene sempre la soluzione ottima in tutte quelle istanze in cui abbiamo una sola macchina o il numero di macchine supera il numero di task. Dato che queste istanze non possono fornire informazioni sul rapporto di approssimazione ho preferito escluderle.

Purtroppo c'è la limitazione che l'algoritmo di ricerca esaustiva non è molto efficiente e non si comporta bene su istanze leggermente più complicate di quelle usate per calcolare le stime. Ho dovuto restringere i test a istanze molto semplici e addirittura a "parametri fissi" per i test con più iterazioni. Credo che queste limitazioni restringano di molto la possibilità di stimare il vero rapporto di approssimazione dell'algoritmo greedy. Le stime calcolate per i diversi valori di t mostrano che l'algoritmo greedy è molto vicino all'ottimo per istanze semplici. Non si può dire nulla di più preciso sul rapporto di approssimazione avendo escluso istanze più complicate.