Università degli Studi di Padova

Dipartimento di Matematica

Corso di Laurea Triennale in Matematica

Algoritmi di approssimazione per trasversali connessi minimi

Relatore:
Prof. Marco Di Summa
Correlatore:
Manuel Francesco Aprile
Laureando:
Marco Furlan
1201774

Anno Accademico 2020/2021

Alla mia famiglia

Abstract

In questa tesi parleremo di algoritmi di 2-approssimazione per trasversali minimi (VC) e più approfonditamente per trasversali connessi minimi (CVC). Tratteremo l'algoritmo di 2-approssimazione di Savage [2] dal punto di vista teorico, e discuteremo l'implementazione di tale algoritmo e di un algoritmo euristico GRASP [4]. Vedremo che in quest'ultimo con una piccola modifica si ottengono risultati migliori.

Indice

Capitolo 1

Introduzione e Nozioni di Base

1.1 Introduzione

In questa tesi trattiamo problemi di Vertex Cover (d'ora in poi VC) e di Connected Vertex Cover (d'ora in poi, CVC) che definiremo in seguito (Sezione 1.2, Nozioni di Base). I problemi di trovare una VC minima o una CVC minima sono notamente problemi NP-hard, ovvero non ammettono algoritmi polinomiali (a meno che P non sia uguale a NP). Ciò significa che, al momento, gli algoritmi conosciuti che risolvono tali problemi in modo esatto sono esponenziali. Al crescere delle dimensioni dei grafi questi algoritmi sono inutilizzabili, in quanto richiedono ore (se non giorni o anni o decenni) per fornire la soluzione. Non avendo algoritmi che sono in grado di fornire la soluzione ottimale in tempo accettabile, l'attenzione si sposta sugli algoritmi di approssimazione, ovvero algoritmi che forniscono soluzioni "buone" in tempo polinomiale.

Il problema di minimizzazione della VC è largamente studiato in informatica teorica. Si tratta di uno dei 21 problemi NP-completi di Karp, e un esempio classico di problema NP-completo (ovvero un problema in cui una soluzione può essere verificata in tempo polinomiale, ma non si conosce un algoritmo che sia in grado di trovare una soluzione in tempo polinomiale).

Il modello è applicato in svariati ambiti: per fare un esempio, una catena commerciale che vuole installare il numero minore possibile di telecamere (nodi) per coprire tutti i corridoi (archi); un altro esempio è la necessità di posizionare un servizio (ad esempio, centri di test COVID) negli aeroporti (nodi) in modo che ogni tratta (archi) sia coperta nell'aeroporto di partenza o di arrivo.

Il problema di minimizzazione della CVC è meno studiato. Tra i lavori principali sull'argomento vi sono i risultati di Carla Savage, tra cui [2] e [3], su cui ho

basato la parte teorica della tesi. Altri risultati importanti vengono dal lavoro di B.Escoffier, L.Gourvès, J.Monnot, [5]; e il lavoro di Y.Li, W.Wang, Z.Yang, [6]. Si trovano applicazioni del problema della CVC nel campo della progettazione di connessioni wireless [6], e nell'RWA (*routing and wavelength assignment*).

1.2 Nozioni di base

Introduciamo alcune definizioni di teoria dei grafi.

Def 1. *Un* ***grafo semplice*** *è una coppia ordinata* $G = (V, E)$ *dove* V *è un insieme, i cui elementi sono detti* ***vertici*** *o* ***nodi****, ed* E *è un insieme di coppie non ordinate di vertici, dette* ***archi****.*
$e = (u, v) \in E$ *è detto arco di* ***estremi*** u *e* v*, inoltre* $u, v \in V$ *sono* ***adiacenti*** *se* $(u, v) \in E$*.*

Def 2. *Un* ***cammino*** *è una coppia* (V, E) *ove* $V = \{v_1, v_2, \ldots, v_n\}$ *e* $E = \{(v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n)\}$.
v_1 *e* v_n *sono gli* ***estremi*** *del cammino. La* ***lunghezza*** *di un cammino è* $|E|$*.*

Def 3. *In un grafo G=(V,E), la* ***distanza*** *tra due vertici* $v,w \in V$*,* ***d(v,w)****, è la minima lunghezza di un cammino in G (ovvero un cammino* (V', E') *t.c.* $V' \in V, E' \in E$*) di estremi v,w.*

Def 4. *Un* ***ciclo*** *è una coppia* (V, E) *ove* $V = \{v_1, v_2, \ldots, v_n\}$ *e* $E = \{(v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n), (v_n, v_1)\}$*,* $n \geq 3$*. Un grafo* $G = (V, E)$ *è detto* ***ciclico*** *se contiene (almeno) un ciclo. (ovvero esistono due sottoinsiemi* $V' \subset V$ *e* $E' \subset E$ *tali che* (V', E') *sia un ciclo).* [1]

Def 5. *Un grafo* $G = (V, E)$ *è detto* ***connesso*** *se per ogni due vertici* $v, w \in V$ *esiste un cammino di estremi* v *e* w*. (ovvero esistono due sottoinsiemi* $V' \subset V$ *e* $E' \subset E$ *tali che* (V', E') *sia un cammino di estremi* v, w*).*

Def 6. *Dato un grafo* $G = (V, E)$ *e un sottinsieme di vertici* $S \subset V$*, chiamiamo* ***sottografo indotto*** *da* S *in* G *il grafo* $G' = (S, E')$ *ove* $E' = \{e = (u, v) \in E : u, v \in S\}$

[1] In questa tesi, il simbolo $\subset$ è un'inclusione larga, cioè corrisponde a $\subseteq$. L'inclusione stretta se necessaria sarà indicata con $\subsetneq$

Def 7. *Dato un grafo $G = (V, E)$, un* ***trasversale*** *o* ***vertex cover*** *(abbreviata* ***VC****) è un insieme di vertici $T \subset V$ tale che ogni arco ha almeno un estremo in T. Ovvero, $\forall e = (u, v) \in E, u \in T \vee v \in T$.*

Def 8. *Un* ***trasversale connesso*** *o* ***connected vertex cover*** *(abbr.* ***CVC****) è un trasversale il cui sottografo indotto è connesso.*

Def 9. *Un* ***matching*** *è un sottoinsieme $M \subset E$ di archi senza estremi in comune.*

Def 10. *Un* ***matching massimale*** *è un matching massimale per inclusione, ovvero un matching che non è contenuto in nessun altro matching.*

Def 11. *Un* ***albero*** *è un grafo connesso e aciclico (= privo di cicli).*
Equivalentemente, un albero è un grafo in cui per ogni due vertici esiste un unico cammino avente come estremi tali vertici.

Def 12. *Un* ***albero ricoprente*** *di un grafo connesso G = (V,E) è un albero T=(V,E') con E' $\subset$ E.*

Def 13. *Dato un problema di minimizzazione P, un'istanza I e un algoritmo A, detti A(I) il costo della soluzione fornita dall'algoritmo e OPT(I) il costo della soluzione ottimale, l'algoritmo A è detto di α* ***- approssimazione*** *per P se vale:*

$$\frac{A(I)}{OPT(I)} \leq \alpha \quad \forall I,$$

ove $\alpha \geq 1$.

Capitolo 2

Teoria

2.1 Approssimazione per trasversali minimi

Vediamo per iniziare un algoritmo di 2-approssimazione per trasversali minimi.

Algorithm 1 Trasversale (Vertex Cover)

Input: G=(V,E) grafo
Output: S vertex cover
1: S = Ø
2: E' = E
3: **while** E' ≠ Ø **do**
4: let e = (u,v) ∈ E' be an uncovered edge
5: S = S ⋃ {u,v}
6: remove from E' all edges with u or v as extremes
7: **end while**
8: **return** S

Funziona così: selezioniamo un arco qualunque (4). Teniamo i due vertici agli estremi (5), e scartiamo tutti gli archi coperti (6). Ripetiamo il processo fino a coprire tutti gli archi (3).

L'insieme ottenuto è ovviamente un trasversale, perché per costruzione copre ogni arco del grafo. Prima di dimostrare che l'algoritmo è di 2-approssimazione, enunciamo il seguente lemma.

Lemma 1. *Sia G=(V,E) grafo, detti $\alpha'(G)$ la massima cardinalità di un matching e $\beta(G)$ la minima cardinalità di un trasversale, valgono le seguenti disuguaglianze:*

$$\alpha'(G) \leq \beta(G) \tag{2.1}$$

$$\beta(G) \leq 2\alpha'(G) \tag{2.2}$$

Dimostrazione. (2.1) Sia M $\subset$ E matching massimo, $|M| = \alpha'(G)$, e T $\subset$ V trasversale minimo, $|T| = \beta(G)$. Essendo T un trasversale deve coprire ogni arco E, in particolare T deve avere almeno un estremo di ogni arco di M. Essendo gli archi di M disgiunti sui vertici, concludiamo che $|M| \leq |T| \implies \alpha'(G) \leq \beta(G)$. (2.2) Consideriamo l'insieme degli estremi degli archi di M, sia T*. Allora $|T^*| = 2\alpha'(G)$. L'insieme T* è un trasversale: infatti se non lo fosse esisterebbe un arco e disgiunto sui vertici da ogni arco di M, pertanto $M' = M \bigcup \{e\}$ sarebbe matching con $|M'| > |M|$, ma M è matching massimo, assurdo. Concludiamo che T* è un trasversale, perciò la dimensione del trasversale minimo è al più $|T^*|$, cioè $\beta(G) \leq 2\alpha'(G)$ □

Teorema 1. *L'Algoritmo 1 è di 2-approssimazione.*

Dimostrazione. Gli archi selezionati dall'algoritmo formano un matching massimale M, sia $k = |M|$ la sua cardinalità: l'algoritmo fornisce un trasversale T di dimensione $|T| = 2k$. Allora, dalla disuguaglianza (2.1), abbiamo $|M| = k \leq \alpha'(G) \leq \beta(G) \implies \dfrac{|T|}{\beta(G)} \leq \dfrac{2k}{\alpha'(G)} \leq \dfrac{2k}{k} \leq 2 \implies \dfrac{|T|}{\beta(G)} \leq 2.$ □

E' facile convincersi che questo algoritmo non può essere di α-approssimazione, con $\alpha < 2$: ad esempio nel grafo in Figura 2.1 il trasversale minimo ha dimensione m, ma l'algoritmo può fornire soluzioni di dimensione $2m$ se sceglie in sequenza gli archi 1-2, 3-4, ecc., inoltre non darà mai una soluzione ottimale. E' curioso che nonostante svariati algoritmi di 2-approssimazione come questo siano noti non è ancora stato trovato un algoritmo di 1.99-approssimazione (o di α-approssimazione con $\alpha \in \mathbb{R}, \alpha < 2$) e non si sa ancora se sia possibile crearlo [1].

2.2 Approssimazione per trasversali connessi minimi

Questa sezione è dedicata a dimostrare l'esistenza di un algoritmo di 2- approssimazione per CVC (dal lavoro di Savage: [2], [3]). Il risultato principale è il Teorema 4, in cui costruiamo la CVC e dimostriamo che è al più 2 volte più grande di quella minima.

Per arrivare a dimostrare il Teorema 4 ci servono un po' di nozioni preliminari.

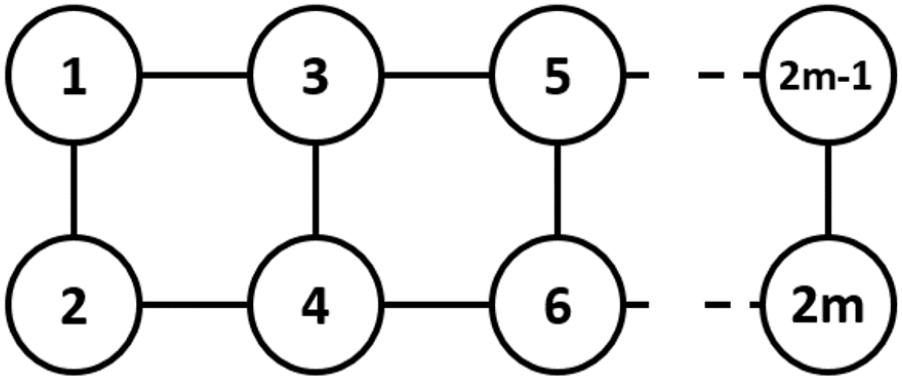


Figura 2.1: Grafo a griglia quadrata. E' un grafo bipartito, la VC minima ha m vertici.

2.2.1 Definizioni

Def 14. *Un **albero con radice** è un albero $T = (V, E, r)$ in cui un vertice arbitrario r è stato designato come **radice**. In questo contesto, le **foglie** o **vertici esterni** sono tutti i vertici di grado 1 distinti dalla radice, e i **vertici interni** sono tutti i vertici che non sono foglie.*
*In un albero con radice, definiamo la **profondità** di un vertice, $d(v)$ (**depth** of v) come la distanza (Def 3) tra la radice e il vertice. Dato un arco $e = (u, v)$, ove $d(u) = d(v) + 1$, diremo che v è **padre** di u e u è **figlio** di v. Due vertici con il padre in comune sono **fratelli**. Diremo che w è **antenato** di v se w è padre di v o, ricorsivamente, è padre di un antenato di v. Diremo che v è **discendente** di w se w è antenato di v.*

Def 15. *L'**algoritmo DFS** (depth-first search algorithm) è un noto algoritmo che permette di visitare tutti i vertici di un grafo. Vedere l'Algoritmo 2 per un'implementazione ricorsiva.*

*Un **albero ricoprente DFS** di un grafo G è un albero ricoprente (Def 12) con radice (Def 14) costruito attraverso la procedura del DFS, collegando di volta in volta il vertice visitato con i vertici adiacenti su cui esegue la chiamata ricorsiva.*

2.2.2 Matching propri

Per dimostrare il Teorema 4 è fondamentale dare la definizione di matching proprio:

Algorithm 2 DFS

```
Input: Graph G=(V,E)
Output: visits every vertex of G
 1: function DFS(G,v)
 2:     visit(v)
 3:     for w in N(v) do      % N(v) is the set of vertices adjacent to v
 4:         if visited[w] = 0 then
 5:             DFS(G,w)
 6:         end if
 7:     end for
 8: end function
 9:
10: visited[v] = 0 for all v in V
11: v = random vertex of T
12: DFS(T,v)
```

Def 16. *Un vertice è **libero rispetto a un matching** se non è estremo di alcun arco del matching.*

Def 17. *Un **matching proprio** di un albero con radice T=(V,E,r) è un matching $M \subset E$ tale che se un vertice v distinto dalla radice è libero rispetto al matching M, allora detto **f(w)** il padre di w esiste un fratello u di w tale che $(u,f(w)) \in M$.*

Qui dimostriamo due proprietà dei matching propri:

1. ogni matching proprio è massimo,
2. è possibile trovare un matching proprio in tempo $O(|V|)$.

La seconda proprietà garantisce l'esistenza di un matching proprio in ogni albero con radice.

Per dimostrare 1. ci serve la nozione di cammino M-aumentante:

Def 18. *Dato un grafo $G = (V, E)$ e un matching $M \in E$, un cammino è detto **M-alternante** se i suoi archi appartengono alternativamente a M e a $E \setminus M$. Un cammino è detto **M-aumentante** se è M-alternante e i suoi estremi sono esposti (ovvero non coperti da M).*

Un risultato noto in teoria dei grafi è:

Teorema 2. *Un matching M di G è massimo se e solo se non esiste un cammino M-aumentante in G.*

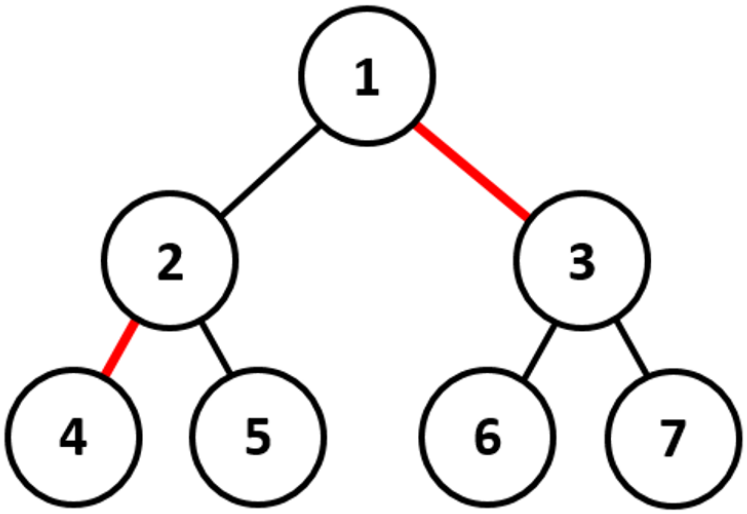


Figura 2.2: Matching massimo $\not\Rightarrow$ matching proprio.
Il matching è massimo perché i vertici (2) e (3) formano un trasversale di dimensione 2, dunque un matching può avere al più dimensione 2 (vedi Lemma 1, (2.1)). Il matching non è proprio perché i vertici (6) e (7) non ne rispettano la definizione.

Ora dimostriamo la proprietà 1.

Lemma 2. *Ogni matching proprio è massimo.*

Dimostrazione. Sia T=(V,E,r) albero con radice, M matching proprio in T; è sufficiente provare che T non ha cammini M-aumentanti. Per assurdo, supponiamo esista un cammino M-aumentante, p, che congiunge due vertici a, b liberi rispetto a M. Sia w un antenato di a e b tale che nessun discendente proprio di w è un antenato sia di a che di b (il "minimo comune antenato"). Allora p è composto da due cammini p_a e p_b, che congiungono rispettivamente w con a e w con b. Essendo $|p|$ dispari, possiamo supporre senza perdita di generalità che $|p_a| > |p_b|$. Se $|p| = 1$, si ha che $b = f(a)$ con a,b vertici liberi rispetto a M: questo contraddice l'ipotesi che M sia un matching proprio. Se $|p| > 1$, allora $|p_a| \geq 2$. Allora sia $(a, f(a))$ che $(f(a), f(f(a)))$ appartengono a p_a, inoltre $(f(a), f(f(a))) \in M$ dato che p_a è un cammino M-alternante. Tuttavia essendo M matching proprio, esiste un vertice x fratello di a t.c. $(x, f(a)) \in M$. Chiaramente $x \neq f(f(a))$, pertanto vi sono due archi distinti di M incidenti nel vertice $f(a)$, che contraddice l'ipotesi che M sia un matching. □

Dimostriamo la proprietà 2. L'Algoritmo 3 ci permette di costruire un matching proprio (dimostreremo rigorosamente che lo è).

$free[v]$ è una lista posizionale (o un dizionario) che rappresenta se il vertice v può essere o meno aggiunto alla soluzione (1=Vero, 0=Falso). L'algoritmo chiama ricorsivamente la funzione *match* (Algoritmo 4).

Algorithm 3 Proper Matching

Input: T=(V,E,r) rooted tree
Output: M proper matching

```
1: for x ∈ V do:
2:     free[x] = 1
3: end for
4: return match(r,free)
```

Algorithm 4 match(x,free)

```
1: M = ∅
2: w = next_son(x)
3: while w ≠ NIL do
4:     S = match(w,free)
5:     M = M ⋃ S
6:     if free[w] and free[x] then
7:         add x,w to M
8:         free[x] = 0
9:     end if
10:    w = next_son(x)
11: end while
12: return M
```

Si assume che l'albero T sia rappresentato come un dizionario (o una lista posizionale) che associa a ogni vertice i suoi figli (o NIL se non ha figli). La funzione $next\_son(v)$ corrisponde di fatto al *.pop*() di Python, nel senso che ritorna un vertice figlio di v e lo elimina dall'insieme dei figli di v (per maggiori dettagli vedere il codice nel capitolo relativo).

Ora dimostreremo che l'insieme di archi M ritornato dall'algoritmo è un matching proprio.

Def 19. *Dato un albero con radice T=(V,E,r), e $x \in V$, l'**albero indotto da x** è il sottoalbero indotto (Def 6) da x e i sui discendenti, con radice x.*

Lemma 3. *Sia x un vertice di T, sia T_x l'albero indotto da x. Allora, l'Algoritmo 3 ritorna un matching proprio di T_x.*

Dimostrazione. Procediamo per induzione sul numero di vertici di T_x. Se $|V_x| = 1$, allora $match(x, free)$ ritorna correttamente M = Ø. Sia ora $|V_x| > 1$, e assumiamo che il Lemma sia vero per ogni sottoalbero di T_x con meno di $|V_x|$ vertici. Allora, $match(x, free)$ inizializza M = Ø e per ogni figlio w_i di x per ipotesi induttiva trova un matching proprio M_{w_i} di T_{w_i}, e aggiunge gli archi

di tutti gli M_{w_i} a M. Inoltre, se uno dei un dei figli di x è libero, sia w_j t.c. $free[w_j] = 1$, allora l'arco (w_j,x) viene aggiunto a M.

Sia ora $y \neq x$ un vertice di T_x libero rispetto a M. Allora esiste un figlio di x, detto w, t.c. y è in T_w. Se $y \neq w$, allora sappiamo che per ipotesi induttiva $M_w = M \cap T_w$ è un matching proprio, dunque esiste un fratello di y, sia y', t.c. $(y', f(y)) \in M_w$. Se $y = w$, cioè $f(y) = x$, allora deve esistere un fratello di y, y' tale che $(y', f(y)) \in M$; se così non fosse, l'algoritmo avrebbe aggiunto l'arco $(y, f(y))$ a M, ma y è libero rispetto a M, assurdo.

Concludiamo che M è un matching proprio. □

2.2.3 Costruzione di un trasversale minimo attraverso un matching proprio

Come abbiamo visto, i matching propri hanno due proprietà rilevanti: la prima è che ogni matching proprio è massimo; la seconda è che esiste sempre un matching proprio e si può trovare in tempo $O(|V|)$.

Sia $T = (V, E, r)$ un albero con radice, e M un matching proprio di T. Definiamo i due insiemi:
$S = \{w \in V : (w, f(w)) \in M\}$
$F = \{f(w) : w \in S\}$
Rispettivamente figli (*sons*) e padri (*fathers*). Ogni altro vertice è libero rispetto a M. Vediamo il seguente teorema:

Teorema 3. *L'insieme F è un trasversale minimo di T.*

Dimostrazione. Dato che $|F| = |M|$, se F è un trasversale allora è minimo.
Dimostriamolo: chiamiamo $\alpha'(T)$ la massima cardinalità di un matching di T, e $\beta(T)$ la minima cardinalità di un trasversale di T, per la (2.1) del Lemma 1 $\alpha'(T) \leq \beta(T)$. Ricordiamo che proprio $\implies$ massimo, dunque $\alpha'(T) = |M|$. Se F fosse un trasversale non minimo, ovvero $|F| > \beta(T)$, si avrebbe $\alpha'(T) = |M| = |F| > \beta(T) \implies \alpha'(T) > \beta(T)$, assurdo.
Ci basta dunque dimostrare che F è un trasversale.
Dimostriamolo: sia $e = (w, v)$ un arco di T, con $f(w) = v$. Allora se il vertice w è libero rispetto a M, allora $v \in F$ per definizione di matching proprio. Se $w \in S$, allora $v \in F$ per definizione di F. Se w non è libero rispetto a M, né è in S, allora è in F. Pertanto, o w o v deve essere in F. Questo vale per ogni arco $e = (w, v)$, dunque concludiamo che F è un trasversale. □

2.2.4 Approssimazione per trasversali connessi minimi

Ora siamo pronti a dimostrare il risultato principale di questa sezione, il Teorema 4:

Teorema 4. *Dato un grafo G, sia T un albero ricoprente DFS di G, allora i vertici interni di T formano un trasversale di G.*
Inoltre, la dimensione di tale trasverale è al più due volte quella minima.

Dimostrazione. Sia $L(T)$ (*leaves*) l'insieme delle foglie di T, e $NL(T)$ (*non-leaves*) l'insieme dei vertici interni di T. Dato che T è un albero ricoprente DFS, nessun arco di G congiunge due vertici di $L(T)$, dunque $NL(T)$ è un trasversale di G.
Dato un vertice $v \in T$, definiamo $\boldsymbol{sons(v)}$ come il numero di figli di v. Allora, $|L(T)| = 1 + \sum_{v \in NL(T)} (sons(v) - 1)$. (volendo si può dedurre questa uguaglianza a partire dalla più intuitiva $|L(T)| + |NL(T)| = 1 + \sum_{v \in NL(T)} sons(v)$.)
Sia ora M(T) un matching proprio (dunque massimo) di T, e sia X l'insieme di vertici liberi rispetto a M(T). Dato un vertice $w \in X$ distinto dalla radice, esiste un fratello w' di w tale che $(w', v) \in M(T)$, dunque $sons(v) \geq 2$. In generale vale $sons(v) \geq 2, \quad \forall v = f(w)$ t.c. $w \in X$; dato che al più 1 vertice di X è la radice, attraverso l'uguaglianza di prima otteniamo $|L(T)| \geq 1 + (|X| - 1) \implies |L(T)| \geq |X|$.
Dato che $|L(T)| = |T| - |NL(T)|$, e $|X| = |T| - 2|M(T)|$, la disuguaglianza diventa $|NL(T)| \leq 2|M(T)|$. In conclusione, sia $C(G)$ un trasversale minimo di G, $M(G)$ matching massimo di G; dalla stringa di disuguaglianze:

$$|M(T)| \leq |M(G)| \leq |C(G)| \leq |NL(T)| \leq 2|M(T)| \tag{2.3}$$

ricaviamo: $|NL(T)|/|C(G)| \leq 2$.

□

La disuguaglianza non può essere migliorata: un controesempio è un cammino dispari ove la radice dell'albero DFS è uno dei due vertici.

Questo teorema ci permette di costruire un algoritmo di 2-approssimazione per CVC con tempo di esecuzione $O(|V| + |E|)$: basta creare un albero ricoprente DFS e prenderne i nodi interni. Lo vedremo nel capitolo Algoritmi.

2.2.5 Corollari

La dimostrazione del Teorema 4 prova indirettamente anche il seguente:

Simbolo	Significato
G	grafo
T	albero ricoprente DFS di G
C(G)	trasversale minimo di G (*cover*)
M(G)	matching massimo di G
C(T)	trasversale minimo di T
M(T)	matching massimo ($\Longleftarrow$ proprio) di T
L(T)	vertici esterni di T (*leaves*)
NL(T)	vertici interni di T (*non-leaves*)

Tabella 2.1: Riassunto dei simboli usati in questa sezione

Teorema 5. *Se T è un albero ricoprente DFS di G, e M(T), M(G) sono matching massimi di T e G rispettivamente, allora M(G)/M(T) ≤ 2.*

Dimostrazione. Si ricava dalla stringa di disuguaglianze (2.3). □

Corollario 1. *Sia G un grafo, T un albero ricoprente DFS di G, C(G) trasversale di G e C(T) trasversale di T. Allora* $|C(G)|/|C(T)| \leq 2$.

Dimostrazione. Sia $M(T)$ matching massimo di T, allora possiamo concludere con un'altra stringa di disuguaglianze: $|C(T)| \leq |C(G)| \leq |NL(T)| \leq 2|M(T)| = 2|C(T)|$. (Per inciso, l'ultima è un'uguaglianza per proprietà degli alberi) □

Ancora una volta, la disuguaglianza non può essere migliorata: un controesempio è un grafo completo dispari K_n, per cui $|C(G)| = n - 1$ e $|C(T)| = \frac{n-1}{2}$.

Come abbiamo visto nel Teorema 5, vale $|M(G)|/|M(T)| \leq 2$, e come visto nel Teorema 4, vale $|NL(T)|/|C(G)| \leq 2$. Una terza disuguaglianza nota è $|C(G)|/|M(G)| \leq 2$, che è la (2.2) dal Lemma 1. Facciamo notare per chiarezza che valgono $1 \leq |M(G)|/|M(T)|$ e $1 \leq |NL(T)|/|C(G)|$ e $1 \leq |C(G)|/|M(G)|$ (quest'ultima dal Lemma 1 (2.1)). Riassumendo:

$$1 \leq \frac{|NL(T)|}{|C(G)|} \leq 2, \quad 1 \leq \frac{|C(G)|}{|M(G)|} \leq 2, \quad 1 \leq \frac{|M(G)|}{|M(T)|} \leq 2$$

Il seguente crea un legame tra le tre disuguaglianze appena citate:

Teorema 6. *Sia G grafo, T albero ricoprente DFS di G. Allora vale:*

$$\frac{|NL(T)|}{|C(G)|} \cdot \frac{|C(G)|}{|M(G)|} \cdot \frac{|M(G)|}{|M(T)|} \leq 2.$$

Dimostrazione. La disuguaglianza è equivalente a $\frac{|NL(T)|}{|M(T)|} \leq 2$, che è stata dimostrata nella solita stringa di disuguaglianze (2.3) dal Teorema 4. □

Questo teorema è utile se si hanno informazioni aggiuntive sul grafo, ad esempio se sappiamo che $\frac{|NL(T)|}{|C(G)|} \approx 2$, cioè la $|NL(T)|$ non è una buona approssimazione per $|C(G)|$, allora le altre due frazioni $\frac{|C(G)|}{|M(G)|}$ e $\frac{|M(G)|}{|M(T)|}$ sono ≈ 1, cioè $|C(G)|$ è una buona approssimazione per $|M(G)|$ e $|M(G)|$ è una buona approssimazione per $|M(T)|$.

Un esempio: possiamo trovare un trasversale $C'(G)$ con un algoritmo di 2-approssimazione come l'Algoritmo 1, dopodiché calcoliamo $k = \frac{|NL(T)|}{|C'(G)|}$ dove $1 \leq k \leq 2$; dato che $|C'(G)| \geq |C(G)|$ ricaviamo $k = \frac{|NL(T)|}{|C'(G)|} \leq \frac{|NL(T)|}{|C(G)|}$ e così sappiamo che $k \cdot \frac{|C(G)|}{|M(G)|} \cdot \frac{|M(G)|}{|M(T)|} \leq 2 \implies \frac{|C(G)|}{|M(G)|} \cdot \frac{|M(G)|}{|M(T)|} \leq 2/k$, da cui $\frac{|C(G)|}{|M(G)|} \leq 2/k$ e $\frac{|M(G)|}{|M(T)|} \leq 2/k$. Abbiamo trovato stime più strette per le due frazioni.

Capitolo 3

Algoritmi

In questo capitolo vediamo alcuni algoritmi per approssimare trasversali minimi e trasversali connessi minimi. L'algoritmo principale che vedremo è il GRASP-CVC, presentato nell'articolo [4].

3.1 Algoritmo di approssimazione per trasversali connessi minimi

Prima di tutto vediamo l'algoritmo la cui validità è dimostrata nel Teorema 4. L'algoritmo è molto semplice:

Algoritmo: Sia $G = (V, E)$ grafo. Attraverso la procedura del DFS (*depth-first search*) creiamo un albero ricoprente DFS $T = (V, E_T)$. I nodi interni di tale albero sono una CVC di G.

Di seguito lo pseudocodice e il codice Python.

```
import random

def CVC_via_dfs(self):
  children = {v:set() for v in graph.vertices}
  visited = {v:0 for v in graph.vertices}
  def dfs_call(self,u):
    visited[u] = 1
    for v in graph.N[u]: #neighbour (adjacent) vertices
      if visited[v] == 0:
        d[u].add(v)
        dfs_call(self,v) #recursive call

  r = random.choice(list(graph.vertices)) #root
```

Algorithm 5 CVC via DFS

Input: Graph G=(V,E)
Output: C $\subset$ V CVC

```
1: % Lines 2,3,4 are needed before calling dfs
2: initialize a positional list children[v] = ∅ for v in graph.vertices
3: inizialize a positional list visited[v] = 0 for v in graph.vertices
4: properly define a function dfs(graph,r)
5: % Now we can build the CVC
6: r = random vertex from graph.vertices
7: dfs(graph,r)
8: CVC = empty_set
9: for v in graph.vertices do
10:     if children[v] is not empty then
11:         Add v to CVC
12:     end if
13: end for
14: return CVC
```

```
14  dfs_call(self,r)
15  CVC = set()
16  for v in d:
17    if bool(d[v]): #exclude the leaves
18      CVC.add(v)
19  return CVC
```

Listing 3.1: Codice Python - CVC via DFS

Questo algoritmo ha complessità $O(|V| + |E|)$, infatti le operazioni (2) e (3) (dallo pseudocodice) sono $O(|V|)$, la dfs (7) è $O(|V|+|E|)$, il ciclo for (9) è $O(|V|)$, le operazioni restanti sono $O(1)$.

Questo algoritmo è efficiente ma non accurato. Vedremo la differenza con il GRASP-CVC nel Capitolo 5, per ora ci basta sapere che nonostante l'algoritmo sia più veloce, è meno efficace nel trovare CVC piccole.

Per questo vedremo in dettaglio l'algoritmo dell'articolo [4], detto GRASP-CVC.

3.2 GRASP-CVC

L'algoritmo del GRASP-CVC (*Greedy Randomized Adaptive Search Procedures for Connected Vertex Cover problem*) è discusso nell'articolo [4]. E' costituito da due fasi: una fase di costruzione di una soluzione greedy, detta *GreedyCon-*

struction, e una fase di ricerca di soluzioni migliori vicine a quella grezza, detta *LocalSearch*. Ecco come viene definito l'algoritmo GRASP-CVC:

Algorithm 6 GRASP-CVC

Input: Connected graph G=(V,E)
Output: C ⊂ V CVC

```
1: Best_CVC = V
2: i = 0
3: max_iter = 100
4: while i < max_iter do
5:     i = i + 1
6:     CVC = G.GreedyConstruction()
7:     CVC = G.LocalSearch(CVC)
8:     if |CVC| < |Best_CVC| then
9:         Best_CVC = CVC
10:    end if
11: end while
12: return Best_CVC
```

Si inizializza la soluzione Best_CVC (1) come l'insieme di tutti i vertici, che è banalmente una CVC (altrimenti si definisce una variabile $max\_CVC = |V|$ e si procede similmente). Dopodiché si ripetono gli algoritmi *GreedyConstruction* (6) e *LocalSearch* (7) e si confronta la CVC ottenuta con la Best_CVC. Se la CVC ha cardinalità minore, si rimpiazza Best_CVC con CVC (9) e si continua. Il numero di iterazioni max_iter (3) può essere regolato in base alla dimensione del grafo: più grande e denso è il grafo, maggiori saranno i tempi di esecuzione.
E' importante notare che l'algoritmo GRASP_CVC non è deterministico, perché le funzioni *GreedyConstruction* e *LocalSearch*, che vedremo in dettaglio, hanno una componente aleatoria. Ciò garantisce una soluzione diversa a ogni iterazione (nella gran parte dei casi).

3.2.1 GreedyConstruction

L'algoritmo della *GreedyConstuction* si basa sulla funzione *score*. Dato un grafo $G = (V, E)$ e un insieme di vertici $C \subset V$ definiamo la funzione *score* che assegna un punteggio a ciascun vertice, $score_C : V \to \mathbb{Z}$,

$$score_C(v) = \begin{cases} |\{w \in N(v) : w \notin C\}| & \text{se } v \notin C \\ -|\{w \in N(v) : w \notin C\}| & \text{se } v \in C \end{cases}$$

Dove $N(v)$ è l'insieme di vertici adiacenti a v.

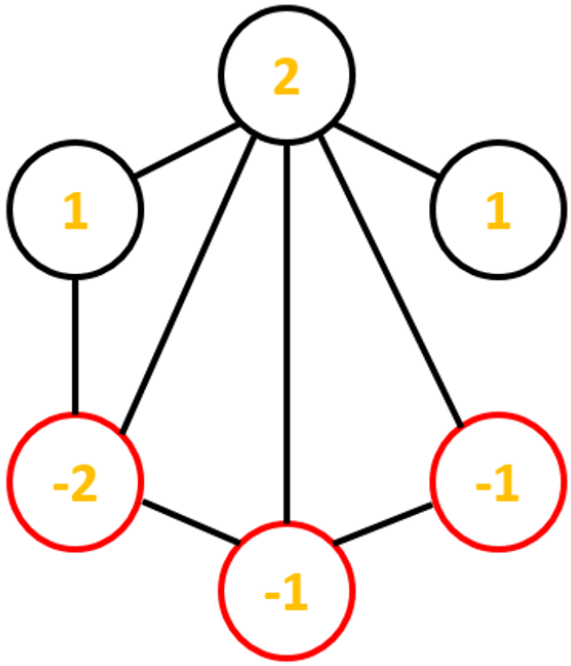


Figura 3.1: La funzione *score* (in arancione) relativa all'insieme C (in rosso)

Se siamo nel caso $v \notin C$, la funzione *score* è il numero di archi che guadagniamo aggiungendo v a C. Se siamo nel caso $v \in C$, la funzione *score* è il numero (negativo) di archi che perdiamo se togliamo v da C (Figura 3.1). In particolare, $score_{\emptyset}(v) = d(v) \quad \forall v \in V$.

Ora siamo pronti a vedere lo pseudo-codice per la *GreedyConstruction*:

Algorithm 7 GreedyConstruction

Input: Connected graph G=(V,E)
Output: C $\subset$ V CVC

1: C = $\emptyset$
2: RCL = $\{v \in V$: v has maximum degree$\}$
3: score[v] = degree[v] for every v in V
4: **while** C is not a CVC **do**
5: choose a vertex v in RCL randomly
6: C = C $\bigcup$ {v}
7: update score[u] for every u in N(v)
8: RCL = $\{u \in N(C)$: u has maximum score$\}$
9: **end while**

Inizializziamo C come insieme vuoto (1) e RCL come insieme dei vertici di grado massimo (2); RCL sta per *Restricted Candidate List* e rappresenta l'insieme dei vertici candidati ad essere aggiunti. Inizializziamo *score* (3) come $score_{\emptyset}$

e entriamo in un ciclo *while*, in cui aggiungiamo di volta in volta uno dei vertici di RCL (5-6), aggiorniamo *score* (7), e aggiorniamo RCL considerando i vertici adiacenti a C (cioè adiacenti ad almeno un vertice di C) con *score* massimo (8) e continuiamo finché C non è una CVC.

Per costruzione, C è connesso ad ogni passaggio (nel senso che C $\subset$ V induce un sottografo connesso), e per controllare se è un trasversale basta definire una variabile (ad esempio, *edges_covered*) che conta quanti archi vengono aggiunti ad ogni iterazione del ciclo while, e arrestare il ciclo quando tale variabile corrisponde al numero totale di archi nel grafo.

Vediamo la complessità di *GreedyConstruction*: (2-3) sono $O(|V|)$, il ciclo while ha al più $|V|$ iterazioni e al suo interno (7), (8) sono $O(d(v))$. Quindi l'algoritmo ha complessità $O(|V| \cdot \sum_{v \in V} d(v)) = O(|V| \cdot |E|)$.

Per vedere un'implementazione in Python e per una spiegazione più dettagliata, vedere il capitolo successivo.

3.2.2 LocalSearch

Supponiamo ora di aver eseguito la *GreedyConstruction*, e di avere C $\subset$ V CVC. E' possibile che all'interno di C vi sia un vertice v t.c. $score_C(v) = 0$; questo vuol dire che togliendo v da C, C* = C $\setminus$ {v}, otteniamo nuovamente una VC (ovvero un trasversale). Se C* è anche connessa, allora C* è una CVC con $|C^*| < |C|$, cioè è una soluzione migliore. Più in generale, avendo una soluzione C la possiamo raffinare, togliendo vertici superflui e visitando le soluzioni "vicine", nella speranza di trovare una nuova soluzione, C*, più piccola. Questo è il principio alla base dell'algoritmo *LocalSearch*.

Prima di vedere lo pseudocodice della *LocalSearch*, dobbiamo parlare del dizionario *Change* (o volendo l'*array Change*; il dizionario è una struttura dati in Python, più dettagli nel capitolo relativo al codice). La *LocalSearch*, il cui funzionamento verrà spiegato a breve, ha la tendenza a visitare ripetutamente le stesse soluzioni, sprecando di fatto tempo e iterazioni. Per limitare questo problema implementiamo un dizionario *Change*, che rappresenta se il vertice v può essere aggiunto alla soluzione $Change[v] = 1$, oppure no $Change[v] = 0$. Il dizionario *Change* viene inizializzato e modificato secondo tre regole:

1. Inizialmente $Change[v] = 1$, cioè tutti i vertici sono candidati ad essere aggiunti alla soluzione;

2. Quando rimuoviamo v da C, poniamo $Change[v] = 0$;

3. Quando un vertice v viene rimosso da C o inserito in C, poniamo $Change[u] = 1 \quad \forall u \in N(v) \setminus C$.

Algorithm 8 LocalSearch

Input: Connected graph G=(V,E), CVC C $\subset$ V
Output: C* $\subset$ V CVC, |C*| $\leq$ |C|

```
1:  C* = C
2:  i = 0
3:  max_iter = 100
4:  while i < max_iter do
5:      if C is vertex cover then
6:          if C is connected then
7:              if |C| < |C*| then
8:                  C* = C
9:              end if
10:         else
11:             return C*
12:         end if
13:         drop a max score vertex u from C, update Change array
14:         continue
15:     end if
16:     choose an uncovered edge e randomly
17:     Choose a vertex v ∈ e such that Change[v]=1 with max score
18:     C = C ⋃ {v}, update Change array
19: end while
20: return C*
```

Ora possiamo vedere lo pseudocodice: (1) Inizializziamo C* come C, e entriamo in un ciclo *while* (4). Se C è un trasversale (5), connesso (6) e più piccolo di C* (7), allora la nuova soluzione migliore è C* = C (8). Se C è un trasversale ma non è connesso (10), abbiamo raggiunto un minimo locale rispetto alla connettività, pertanto ritorniamo C* (11). Altrimenti, proviamo a migliorare la soluzione C togliendo un vertice u di punteggio massimo da C (13), aggiorniamo *Change* secondo le regole 2) e 3), e torniamo all'inizio del ciclo *while* (14). Se C non è un trasversale (da 16 in poi), aggiungiamo vertici finché non torna ad essere un trasversale: scegliamo un arco e non coperto da C (16), scegliamo un estremo v di e con $Change[v] = 1$ (l'esistenza di un vertice $v \in e$ t.c. $Change[v] = 1$ è garantita dalla regola 3) di *Change*). Se entrambi gli estremi di e hanno *Change* uguale a 1, scegliamo quello con *score* maggiore. Aggiungiamo v a C (18) e aggiorniamo Change secondo la regola 3). Una volta eseguito il numero massimo di iterazioni *max_iter*, usciamo dal ciclo *while* e ritorniamo C* (20).

Ogniqualvolta si aggiunge o rimuove un vertice da C è raccomandabile aggiornare *score* (l'algoritmo funziona anche senza, ma dà prestazioni migliori con).

Vediamo la complessità di *LocalSearch*: (5) è $O(1)$ perché teniamo una lista degli archi non coperti *uncovered_edges*; (6) è $O(|C| + |E_C|)$ perché viene fatto via DFS (E_C sono gli archi del sottografo indotto da C); (13) è $O(|C| + d(u))$ e (18) è $O(d(v))$; il resto è $O(1)$. Sia $\Delta = \max\{d(u) : u \in V\}$, allora la complessità della *LocalSearch* è $O(|C| + |E_C| + \Delta)$.

La complessità dell'intero algoritmo GRASP-CVC è dunque $O(|V| \cdot |E| + |C| + |E_C| + \Delta)$. Se siamo nel caso in cui $|V| \cdot |E| \gg |V| + |E| > |C| + |E_C|$, e $|V| \cdot |E| \gg |V| \geq \Delta$, possiamo dire che la complessità è $O(|V| \cdot |E|)$.

3.2.3 Altre idee

L'algoritmo GRASP-CVC ha una buona efficienza e una buona prestazione, fornendo CVC piccole in tempi brevi. Lavorando sul codice ho provato a pensare ad miglioramenti o ad altre possibili soluzioni per trovare una CVC piccola in modo efficiente; le lascio come idee per chi fosse interessato:

- La prima idea è modificare gli algoritmi di *GreedyConstruction* e *LocalSearch* in modo intelligente. Ho provato le seguenti idee:
  - Quando viene rimosso v, si aggiungono tutti gli u ∈ N(v). Il nuovo insieme ottenuto è ancora una VC e l'algoritmo esplora un sottinsieme dello spazio delle soluzioni più ampio. Le prestazioni sono analoghe a quelle del *LocalSearch* originale, occasionalmente migliori.
  - Quando l'insieme è una VC disconnessa (cioè la soluzione è un minimo locale), invece di ritornare C* si aggiunge un nuovo vertice che connette l'insieme (possibilmente distinto da quello appena rimosso) e si continua con il ciclo. Questo è probabilmente efficace con grafi sparsi, che arrivano alla condizione di uscita con più facilità; tuttavia i grafi più grandi che ho usato non arrivano mai a quel punto di codice (anche per grafi come C125.9, C250.9, C500.9 che sono piuttosto sparsi, evidentemente è molto raro che si ottenga una VC disconnessa, generalmente rimouvere vertici compromette la VC prima della connettività).
  - Invece di rimuovere un vertice alla volta si possono rimuovere più vertici con *score* massimo. Anche questo esplora un sottinsieme più

grande dello spazio delle soluzioni. Le prestazioni sono analoghe, occasionalmente migliori.

- La seconda idea è di scrivere un algoritmo nuovo. L'algoritmo *GreedyConstruction* mantiene la connettività e accresce l'insieme finché non arriva ad un trasversale. Allora possiamo provare a fare il procedimento opposto, ovvero partire da un trasversale e provare a connetterlo. Un'idea può essere trovare un matching massimale (può essere fatto in $O(|V|)$) o massimo (più dispendioso), sappiamo che una VC contiene almeno un vertice per arco del matching, dunque scegliamo (in modo casuale, o in modo intelligente se possibile) un vertice per ogni arco, dopodiché connettiamo l'insieme aggiungendo "sufficientemente pochi" vertici (la parte più complicata da implementare), infine se non si ha già una VC si aggiungono vertici (ad es. con una $GreedyConstruction$ modificata, o con la $LocalSearch$).

  La mia implementazione di questo metodo aveva risultati discreti in tempi abbastanza grandi, e si affidava comunque alla $LocalSearch$ per ottimizzare la soluzione. Il processo è in generale più macchinoso del GRASP-CVC e non dà risultati migliori.

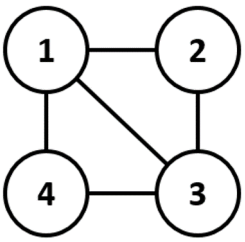
- Non sono esperto nell'ambito, ma un algoritmo di Deep Learning potrebbe avere potenziale per trovare CVC piccole in tempi efficienti. Potrebbe funzionare, a grandi linee, così: si pensa al problema come un gioco da 1 giocatore; le regole del gioco sono: nel primo turno si sceglie un vertice v_1 e si pone $C = \{v_1\}$, nei turni seguenti si sceglie un vertice v_i adiacente a (almeno un vertice di) C e si pone $C = C \cup \{v_i\}$. Il gioco termina quando C è una CVC, e si vince quando si trova una CVC piccola, possibilmente più piccola di quella precedente. Le mosse possibili possono essere rappresentate attraverso una struttura ad albero del tutto analoga a quella che gli algoritmi di Deep Learning come AlphaZero studiano per scegliere le mosse migliori a scacchi [7] [8], e si potrebbero applicare gli stessi metodi come la ricerca ad albero Monte Carlo.

Capitolo 4

Codice

In questa sezione presento il codice che ho programmato in Python, per vedere maggiori dettagli tecnici, e per renderlo disponibile se qualcuno volesse provarlo.

4.1 Come rappresentare un grafo



(a) Grafo

```
{    1:{2,3,4},
     2:{1,3},
     3:{1,2,4},
     4:{1,3}
}
```

(b) Grafo implementato via dizionario

```
[    {1,2},
     {1,3},
     {1,4},
     {2,3},
     {3,4}
]
```

(c) Grafo implementato via lista di archi

	(1)	(2)	(3)	(4)
(1)	0	1	1	1
(2)	1	0	1	0
(3)	1	1	0	1
(4)	1	0	1	0

(d) Grafo implementato via matrice di incidenza

I grafi hanno tre implementazioni comuni:

1. La prima e la più comune è implementarli come lista posizionale (*positional list*) di liste concatenate (*linked lists*), in cui si associa ad ogni vertice i suoi vertici adiacenti. Questo può essere fatto in Python sfruttando i dizionari.

2. La seconda è implementarli come lista di archi. E' un'implementazione svantaggiosa per operazioni come trovare i vertici adiacenti, vantaggiosa in altri casi.

3. Infine, un grafo può essere implementato attraverso una matrice di incidenza.

4.2 Algoritmo per matching propri

Vediamo implementato l'algoritmo per trovare matching propri. d è il dizionario che rappresenta l'albero; in questo codice è inizializzato come l'albero in Figura 2.2.

```
d = {1:{2,3},2:{4,5},3:{6,7},4:set(),5:set(),6:set(),7:set()}
V = set(d.keys())

def proper_matching(r):
  free = {}
  for x in V:
    free[x] = 1
  M = match(r,free)
  return M

def next_son(x):
  if d[x]:
    return d[x].pop()
  else:
    return None

def match(x,free):
  M = []
  w = next_son(x)
  while not w == None:
    M.extend(match(w,free))
    if free[w] and free[x]:
      M.append({x,w})
      free[x] = 0
    w = next_son(x)
  return M

M = proper_matching(1) #1 e' la radice
print(M)
#risultato: [{2,4},{3,6}]
```

Listing 4.1: Codice Python - Matching Proprio

4.3 GRASP-CVC

I due algoritmi *GreedyConstruction* e *LocalSearch* sono definiti come metodi di una classe Python *Graph*, i cui oggetti sono grafi con i seguenti attributi:

- `self.N` è il dizionario che rappresenta il grafo. Pertanto, in questo codice, `self.N[v]` $= N(v)$, e `len(self.N[v])` $= |N(v)|$ grado di v.
- `self.V` è insieme dei vertici del grafo. `len(self.V)` $= |V|$
- `self.E` è l'insieme degli archi del grafo. `len(self.E)` $= |E|$

4.3.1 GreedyConstruction

```
def GreedyConstruction(self):
  #inizializziamo score con i gradi dei vertici
  score = {v:len(self.N[v]) for v in self.V}
  #creiamo l'insieme RCL dei vertici di grado massimo
  max_deg = 0
  for v in self.V:
    if score[v] > max_deg:
      max_deg = score[v]
      RCL = {v}
    elif score[v] == max_deg:
      RCL.add(v)
  C = set()
  neighbourhood = set()
  edges_covered = 0
  m = len(self.E)
  while edges_covered != m:
    v = random.choice(list(RCL))
    C.add(v)
    new_neighbours = self.N[v]-C
    neighbourhood.update(new_neighbours)
    neighbourhood.discard(v)
    #aggiorniamo edges_covered
    for w in self.N[v]:
      edges_covered += not w in C
    #aggiorniamo score
    for w in new_neighbours:
      score[w] -= 1
    #aggiorniamo RCL con i vertici con score massimo
    max_score = -1
    for w in neighbourhood:
```

```
31        if score[w] > max_score:
32          max_score = score[w]
33          RCL = {w}
34        elif score[w] == max_score:
35          RCL.add(w)
36    return C
```

Listing 4.2: GreedyConstruction

4.3.2 LocalSearch

Il codice della LocalSearch è abbastanza lungo e complicato, per avere un'intuizione su come funzioni l'algoritmo consiglio di guardare lo pseudo-codice; se invece si è interessati nell'implementazione, eccolo qua.

```
1  def LocalSearch(self,C):
2    #inizializziamo Change
3    Change = {v:1 for v in self.V}
4    #inizializziamo score
5    score = {v:(2*(not v in C)-1)*len(self.N[v]-C) for v in self.V}
6    uncovered_edges = []
7    C2 = C.copy()
8    is_vertex_cover = True
9    is_connected = True
10   max_iterations = 100
11   for i in range(max_iterations):
12     if is_vertex_cover:
13       if is_connected:
14         if len(C) < len(C2):
15           C2 = C.copy()
16       else:
17         return C2
18       #creiamo RCL con i vertici con score massimo
19       max_score = float('-inf')
20       for w in C:
21         if score[w] > max_score:
22           max_score = score[w]
23           RCL = {w}
24         elif score[w] == max_score:
25           RCL.add(w)
26       u = random.choice(list(RCL))
27       C.remove(u)
28       score[u] = -score[u]
29       Change[u] = 0
30       #aggiorniamo Change
```

```
31        for v in self.N[u]-C:
32          Change[v] = 1
33        #aggiorniamo score
34        for v in self.N[u]:
35          score[v] = score[v] - (2*(not v in C)-1)
36        #aggiorniamo uncovered_edges
37        for v in self.N[u]-C:
38          uncovered_edges.append({u,v})
39        #aggiorniamo is_vertex_cover
40        is_vertex_cover = not len(uncovered_edges)
41        #aggiorniamo is_connected
42        is_connected = self.is_connected_subgraph(C)
43        continue
44      e = random.choice(uncovered_edges)
45      v1,v2 = e
46      if (Change[v1], Change[v2]) == (1,0): u = v1
47      elif (Change[v1], Change[v2]) == (0,1): u = v2
48      else:
49        if score[v1] >= score[v2]: u = v1
50        else: u = v2
51      C.add(u)
52      score[u] = -score[u]
53      #aggiorniamo Change
54      for v in self.N[u]-C:
55        Change[v] = 1
56      #aggiorniamo score
57      for v in self.N[u]:
58        score[v] = score[v] + (2*(not v in C)-1)
59      #aggiorniamo uncovered_edges
60      for v in self.N[u]-C:
61        uncovered_edges.remove({u,v})
62      #aggiorniamo is_vertex_cover
63      is_vertex_cover = not len(uncovered_edges)
64      #aggiorniamo is_connected
65      is_connected = self.is_connected_subgraph(C)
66    return C2
```

Listing 4.3: LocalSearch

Per aggiornare `is_connected` si ricorre al metodo `self.is_connected_subgraph(C)`, che ritorna Vero se il sottografo indotto da C è connesso, Falso altrimenti. Può essere programmato via DFS (*depth-first search*) o BFS (*breadth-first search*).

Capitolo 5

Risultati

In questo capitolo mettiamo a confronto i miei risultati con quelli dell'articolo [4]. Non ho inserito i risultati del mio GRASP-CVC perché sono identici a quelli dell'articolo; tuttavia ho messo quelli del mio algoritmo GRASP-CVC* di cui parlerò in dettaglio a breve. Ho selezionato 17 grafi dall'archivio DIMACS (*Center for Discrete Mathematics and Theoretical Computer Science*), un noto archivio di grafi usati anche nell'articolo; i grafi che ho scelto hanno un numero di vertici che varia dai 200 agli 800. Definiamo la **densità** di un grafo come il rapporto tra il numero di archi presenti nel grafo e il numero di archi massimo, cioè $\frac{|E|}{\binom{|V|}{2}} = \frac{2|E|}{|V|(|V|-1)}$.

Il computer con cui ho calcolato i risultati ha sistema operativo Windows 10, processore i5 7th gen, ram 8gb.

5.1 GRASP-CVC vs DFS-CVC

Nella Tabella 5.1 confrontiamo il GRASP-CVC da [4] con il DFS-CVC (Algoritmo 5). I risultati e i tempi del GRASP-CVC sono presi da [4]; i risultati e i tempi del DFS-CVC sono ottenuti eseguendo 1000 iterazioni e selezionando il risultato migliore. Ho deciso di eseguire diverse iterazioni anche per il DFS-CVC perché la scelta della radice è casuale e può portare a risultati diversi, e l'algoritmo è comunque molto veloce dunque ce lo possiamo permettere. MVC è la dimensione della VC minima, che è nota in questi grafi; segnaliamo in *corsivo* quando la CVC è uguale alla MVC ($\implies$ è CVC minima).

5.2 GRASP-CVC vs GRASP-CVC*

5.2.1 GRASP-CVC*

Parliamo dell'algoritmo GRASP-CVC*. L'algoritmo GRASP-CVC* è la mia versione dell'algoritmo del GRASP-CVC. E' di fatto identico al GRASP-CVC che abbiamo visto nell'Algoritmo 6, ad eccezione di poche linee di codice:

1. dove si aggiorna *score* nella *LocalSearch*,

   ```
   for v in self.N[u]:
       score[v] = score[v] ± (2*(not v in C)-1)
   ```

   si sostituisce con:

   ```
   for v in self.N[u]:
       score[v] = score[v] ± (2*(v in C)-1)
   ```

2. quando nella *LocalSearch* si aggiunge un estremo di un arco scoperto, se *Change* è 1 per tutti e due i vertici si sceglie uno dei due casualmente.

Un risultato del tutto empirico (vedere i risultati in **grassetto** nella Tabella 5.2) è che il GRASP-CVC* sia migliore del GRASP-CVC, con lo stesso tempo di esecuzione (vedasi sezione 5.2.3).

5.2.2 Il miglioramento del GRASP-CVC*

Il GRASP-CVC* deve gran parte del suo miglioramento a 1. (2. è meno rilevante).

Il fatto che il GRASP-CVC* ottenga risultati migliori è evidente, ma il perché non è altrettanto chiaro. In effetti, questo algoritmo è nato per un mio errore nella scrittura del codice, avevo dimenticato di inserire il "not". Il dizionario *score*, a fine processo, contiene alcuni valori completamente diversi da quelli dati dalla funzione *score*.

Ecco cosa (plausibilmente) succede: l'algoritmo GRASP-CVC, per come è strutturato, ritornerà sempre una CVC come soluzione: qualsiasi modifica venga fatta all'insieme di partenza C verrà tenuta se C è una CVC minore, e scartata altrimenti. Il ruolo di *score* in questo processo è quello di favorire la rimozione di vertici meno utili e l'aggiunta di vertici più utili a trovare una CVC minore. Per questo non è davvero importante che il dizionario *score* mantenga i valori dati dalla funzione *score*, ma è prioritario che assuma valori che favoriscono la scoperta di una soluzione migliore. A quanto pare, questo è quello che succede nel GRASP-CVC*.

5.2.3 Confronto con GRASP-CVC

Nella Tabella 5.2 confrontiamo il GRASP-CVC con il GRASP-CVC*. Ancora una volta i risultati e i tempi del GRASP-CVC vengono dall'articolo [4], mentre i risultati e i tempi del GRASP-CVC* sono ottenuti con 1000 iterazioni e selezionando il risultato migliore. Inoltre, dato che è nota la dimensione della VC minima di questi grafi (MVC nella tabella), ho aggiunto una condizione di uscita nel caso in cui la CVC avesse tale dimensione (in questo caso risultati e tempi sono in *corsivo*, e la CVC è banalmente minima).

Infine parliamo dei tempi di esecuzione: l'algoritmo dall'articolo [4] sembra essere molto più veloce del mio GRASP-CVC*, e in effetti è anche più veloce del mio GRASP-CVC: gli algoritmi GRASP-CVC* e GRASP-CVC hanno lo stesso tempo di esecuzione in quanto le differenze tra i due sono minime e non creano differenze significative nei tempi di esecuzione.

Dunque l'algoritmo dell'articolo (5000 iterazioni) è in media più veloce del mio algoritmo (1000 iterazioni). Questo è dovuto senza alcun dubbio a maggiori prestazioni del processore e del linguaggio di programmazione (nell'articolo usano C++), altrimenti non mi spiegherei come il loro GRASP-CVC che è $O(|V| \cdot |E|)$ batte occasionalmente il mio DFS-CVC che è $O(|V| + |E|)$. Inoltre, data l'oscillazione dei tempi di esecuzione dell'algoritmo dell'articolo, immagino sia stata applicata qualche ottimizzazione aggiuntiva.

Concludendo, a parità di implementazione, linguaggio e prestazioni del pc, il mio GRASP-CVC* avrebbe gli stessi tempi di esecuzione del GRASP-CVC dell'articolo [4] (perché appunto le differenze tra i due algoritmi sono minime), mantenendo risultati migliori. Nella Tabella 5.2 evidenziamo in **grassetto** i risultati migliori della GRASP-CVC*.

Le tabelle. ρ è la densità, MVC è la dimensione della VC minima.

Graph	$\|V\|$	ρ	MVC	GRASP-CVC	time	DFS-CVC	time
brock200-2	200	0.5	188	190	0.04	198	1.67
brock200-4	200	0.34	183	184	1.4	196	1.17
brock400-2	400	0.25	371	376	6.2	395	4.01
brock400-4	400	0.25	367	376	10.29	394	3.98
brock800-2	800	0.34	776	780	35.04	795	19.14
brock800-4	800	0.35	774	780	174	795	19.42
C125-9	125	0.1	91	*91*	0.01	111	0.21
C250-9	250	0.1	206	207	2.38	236	0.73
C500-9	500	0.09	443	448	12.05	484	2.89
DSJC500-5	500	0.49	487	*487*	2.42	497	10.25
gen200-p0-9-44	200	0.1	156	164	0.06	187	0.55
gen200-p0-9-55	200	0.1	145	156	0.06	187	0.5
gen400-p0-9-55	400	0.1	345	358	72.48	388	1.99
gen400-p0-9-65	400	0.1	335	354	18.81	386	1.67
gen400-p0-9-75	400	0.1	325	357	2.8	382	1.98
hamming8-4	256	0.36	240	*240*	0.02	255	1.98
keller4	171	0.35	160	*160*	3.93	169	0.78

Tabella 5.1: GRASP-CVC vs DFS-CVC

Graph	$\|V\|$	ρ	MVC	GRASP-CVC	time	GRASP-CVC*	time
brock200-2	200	0.5	188	190	0.04	**189**	60.0
brock200-4	200	0.34	183	184	1.4	184	36.63
brock400-2	400	0.25	371	376	6.2	**375**	132.52
brock400-4	400	0.25	367	376	10.29	**375**	132.05
brock800-2	800	0.34	776	780	35.04	780	620.04
brock800-4	800	0.35	774	780	174	**779**	614.08
C125-9	125	0.1	91	*91*	0.01	*91*	*0.01*
C250-9	250	0.1	206	207	2.38	***206***	*1.08*
C500-9	500	0.09	443	448	12.05	**445**	97.22
DSJC500-5	500	0.49	487	*487*	2.42	*487*	*9.27*
gen200-p0-9-44	200	0.1	156	164	0.06	**160**	22.15
gen200-p0-9-55	200	0.1	145	156	0.06	***145***	*0.2*
gen400-p0-9-55	400	0.1	345	358	72.48	**351**	68.87
gen400-p0-9-65	400	0.1	335	354	18.81	**353**	69.47
gen400-p0-9-75	400	0.1	325	357	2.8	**348**	68.89
hamming8-4	256	0.36	240	*240*	0.02	*240*	*0.06*
keller4	171	0.35	160	*160*	3.93	*160*	*0.27*

Tabella 5.2: GRASP-CVC vs GRASP-CVC*

Bibliografia

[1] Subhash Khot, Oded Regev, *Vertex cover might be hard to approximate to within 2-epsilon*, 2007

[2] C.Savage, *Depth-First Search and the Vertex Cover Problem*, 1982

[3] C.Savage, *Maximum Matchings and Trees*, 1980

[4] Y.Zhang, J.Wu, L.Zhang, P.Zhao, J.Zhou, M.Yin, *An Efficient Heuristic Algorithm for Solving Connected Vertex Cover Problems*, 2018

[5] B.Escoffier, L.Gourvès, J.Monnot, *Complexity and approximation results for the connected vertex cover problem in graphs and hypergraphs*, 2010

[6] Y.Li, W.Wang, Z.Yang, *The connected vertex cover problem in k-regular graphs*, 2019

[7] *How Does AlphaZero Play Chess?* `https://www.chess.com/article/view/how-does-alphazero-play-chess`

[8] *What's Inside AlphaZero's Chess Brain?* `https://www.chess.com/article/view/whats-inside-alphazeros-brain`