UNIVERSITÀ DI PISA

DIPARTIMENTO DI INGEGNERIA
DELL'INFORMAZIONE

LAUREA TRIENNALE IN
INGEGNERIA INFORMATICA

Calcolo di stime certificate per il numero cromatico di un grafo

Relatore
Prof. Stefano Massei

Candidato
Alessandro Xavier Battisti

9 Ottobre 2025

A.A. 2024/2025

Sommario

1. Introduzione

Il problema affrontato in questa tesi è quello della **stima del numero cromatico di un grafo non orientato**.

1.1. Definizione del problema

Dato un grafo non orientato G, cioè un insieme di vertici e archi, si vuole assegnare a ciascun vertice un colore con l'unico vincolo che due vertici collegati da un arco (adiacenti) abbiano colori diversi **[Figura 1]**. Il numero minimo di colori necessari a colorare un grafo è detto **numero cromatico del grafo G**, indicato con $\chi(G)$.

A partire da un qualsiasi grafo, trovare il suo numero cromatico è un problema *NP-completo* (Non-deterministic Polynomial time). Ciò significa che non sono noti algoritmi con costo polinomiale in grado di calcolarlo. Inoltre, dal momento che il problema appartiene alla classe degli *NP-**completi***, qualsiasi altro problema NP è riconducibile a questo in un tempo polinomiale.

Visto che calcolare esattamente il numero cromatico può essere troppo costoso ci si accontenta di trovare un insieme di valori possibili che può assumere. Si rivela infatti necessario procedere con algoritmi euristici per trovare una *stima dall'alto*, quindi un numero maggiore o uguale a quello cercato, e una *stima dal basso* con strategie di algebra computazionale.

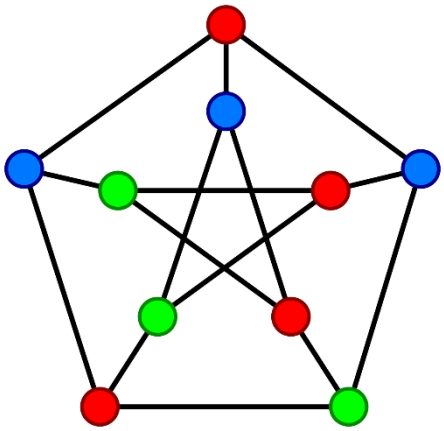


Figura 1
Esempio di colorazione di un grafo

La finalità del progetto è quella di scrivere un programma in grado di restituire la stima del numero cromatico di un grafo in ingresso.
Nelle sezioni successive sono riportati i cenni matematici necessari, le strategie per approcciare il problema, l'ambiente di sviluppo usato e la struttura del programma finale.

1.2. Storia

La colorazione dei grafi ha una lunga storia e rimane un tema aperto ancora oggi.
Nacque intorno al 1852 quando Francis Guthrie iniziò a trattare una forma primitiva del problema che riguardava grafi planari. Egli era interessato infatti alla colorazione delle mappe e intuì che quattro colori erano sempre sufficienti per fare in modo che regioni adiacenti avessero colori diversi. Chiese consulto al fratello minore Frederick il quale sottopose le osservazioni di Francis al suo insegnante di matematica all'Università di Londra, il grande matematico Augustus De Morgan, autore di molti risultati fra cui l'omonimo teorema alla base della logica booleana. Quest'ultimo menzionò poi il problema in una lettera a William Rowan Hamilton, illustre scienziato. Nacque così la formulazione del famoso **Teorema dei quattro colori**.

Molte persone tentarono di dimostrare rigorosamente questo teorema, ma senza successo. Esempio memorabile è la dimostrazione di Alfred Kempe del 1879 che gli valse numerosi titoli. Questa fu ritenuta valida per 11 anni fino a quando, nel 1890, Heawood scoprì degli errori nell'argomentazione di Kempe. Per dimostrare il teorema sarebbero serviti più di cento anni.
Nel 1976 venne finalmente proposta una soluzione da Kenneth Appel e Wolfgang Haken che ebbe grande rilevanza storica in quanto, per la prima volta, fu determinante l'uso del calcolatore per la dimostrazione di un teorema, fatto che sollevò anche dubbi di carattere filosofico.

La sfida posta da questo teorema e, successivamente, dal problema più generico della colorazione di grafi non planari, ha contribuito significativamente allo sviluppo della teoria dei grafi.
Nel 1912 George David Birkhoff introdusse per la prima volta il concetto di **polinomio cromatico** e nel 1960 Claude Berge formulò la **congettura forte del grafo perfetto** che venne enunciata soltanto nel 2002 nel **Teorema forte del grafo perfetto**.

Per quanto riguarda il problema algoritmico trattato nella tesi esso appartiene ai **21 problemi NP-completi di Karp** del 1972. Nel 1981 è stata introdotta una delle applicazioni principali, cioè l'allocazione dei registri nei compilatori, trattata sommariamente nella **sezione 6.2.3**.

2. Stima del numero cromatico

Le stime da fare sono due: una dall'alto con algoritmi euristici e una dal basso usando una strategia diversa. Se le stime coincidono allora si conclude che il valore ottenuto è proprio il numero cromatico del grafo, altrimenti ci si limita a definire un intervallo di appartenenza formato dalle stime stesse.

2.1. Stima dall'alto

La stima dall'alto è la più semplice e fa uso di un algoritmo *greedy*, ovvero un algoritmo che effettua la scelta ottima passo per passo nella speranza di raggiungere l'ottimo globale.
Lo scopo dell'algoritmo è utilizzare il minor numero di colori possibile ed è così definito:

- Si ordinano, secondo qualche logica, i vertici del grafo in una lista
- Si prende il primo vertice della lista e gli si assegna un colore
- Per ogni vertice successivo, se possibile (cioè, sulla base delle sue adiacenze) gli si assegna un colore già usato, altrimenti si sceglie un nuovo colore.

Esistono diversi algoritmi che effettuano queste operazioni, la differenza sostanziale sta nella logica con cui si ordinano i vertici nella lista.
Relativamente alla scelta dei nuovi colori, supponendo di rappresentare i colori con un intero crescente, nel momento in cui bisogna deciderne uno nuovo da assegnare (quindi nel caso in cui non sia possibile riutilizzare un colore precedente) si sceglie l'intero non ancora utilizzato minore possibile. Quindi per ogni vertice da colorare si scorrono i colori partendo da 1 e incrementando finché non se ne trova uno ammissibile.
Nel progetto sono stati utilizzati i seguenti tre algoritmi:

Welsh-Powell

Questo algoritmo prevede di ordinare i vertici in ordine decrescente rispetto al loro *grado* ossia rispetto al numero di vertici adiacenti. Quindi il vertice di grado più alto sarà colorato per primo e così a seguire **[Figura 2]**.

Euristica di Brelaz (algoritmo DSATUR)

L'algoritmo "*degree of saturation*" (DSATUR) prevede di valutare il prossimo vertice da colorare passo per passo, senza stabilire un ordinamento a priori. L'idea è cominciare dal vertice col massimo *grado di saturazione* (da cui il nome dell'algoritmo), ossia il vertice adiacente al maggior numero di colori distinti. A parità di grado di saturazione si sceglie il vertice con grado maggiore (inteso come numero di vertici adiacenti). All'inizio nessun vertice è colorato, quindi hanno tutti grado di saturazione uguale a 0. Il primo vertice è scelto quindi sulla base dei vicini e, in caso di pari merito, si può scegliere in ordine di indice o casualmente.

Ordinamento casuale

Come il nome suggerisce, si ordinano i vertici in maniera casuale e si colorano. Questo viene fatto per un certo numero di volte. Si considera come stima vera e propria il risultato che ha richiesto meno colori.

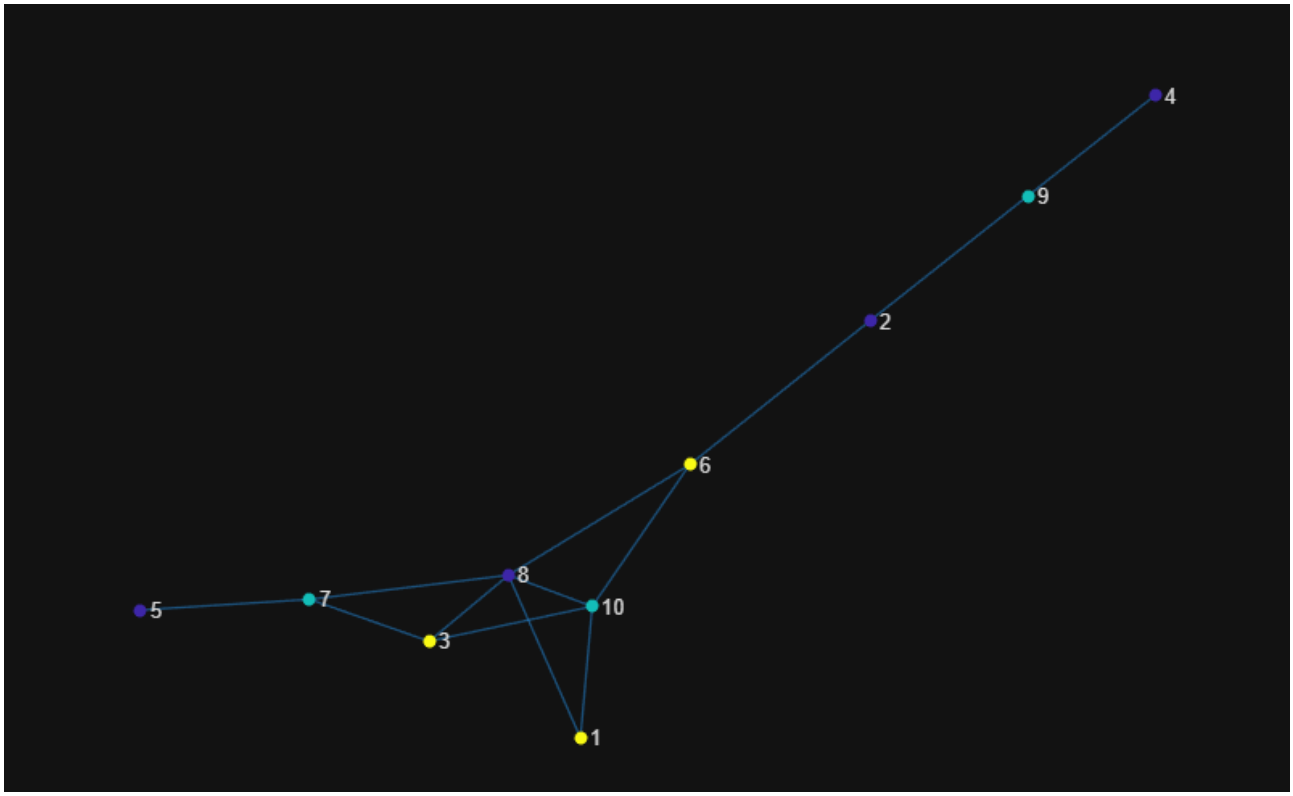


Figura 2
Esempio di grafo colorato con l'algoritmo di Welsh-Powell (3 colori)

È stata scritta, per fini sperimentali, una funzione MATLAB "*creaMat(n)*" che prende in ingresso un numero n di vertici e genera un grafo casuale non orientato di n vertici (rappresentato con la sua matrice di adiacenza, concetto spiegato all'inizio della **sezione 5.2**).

Il codice è molto semplice, l'idea è generare una matrice $n \times n$ con elementi reali tra 0 e 1, convertire in 1 quelli minori di un certo valore p e in 0 gli altri (la matrice di adiacenza ha l'elemento $(i,j) = 1$ se il vertice v_i del grafo è adiacente al vertice v_j, 0 altrimenti). Successivamente si rende la matrice simmetrica (grafo non orientato):

```
function A = creaMat(n)
    p = 0.27; % Probabilità che un'entrata della matrice diventi 1

    % Matrice casuale (parte superiore senza diagonale)
    A = rand(n) < p;
    A = triu(A, 1);

    % Rendi simmetrica e senza self-loop
    A = A + A';
end
```

Con questa funzione sono stati testati i tre algoritmi su 10.000 grafi generati casualmente di dimensione uniformemente variabile tra 16 e 27 vertici e, scartando tutti i risultati ex-aequo (circa 8000), si ricava che ciascun algoritmo trova la stima più piccola una percentuale di volte pari a:

1. DSATUR	79.5%
2. Ordinamento casuale	17.5%
3. Welsh-Powell	3%
Ex-Aequo	≈ 8000

È importante specificare che l'algoritmo DSATUR, che raggiunge più spesso il risultato migliore, è in media circa dieci volte più lento rispetto agli altri.

Testando altri 10.000 grafi con dimensioni, stavolta, tra 100 e 200 vertici (100 grafi per dimensione) i risultati ottenuti sono (sempre scartando i pari merito):

1. DSATUR	98.8%
2. Welsh-Powell	1.1%
3. Ordinamento casuale	0.1%
Ex-Aequo	≈ 800

All'aumentare della dimensione del problema diventa più evidente quanto l'algoritmo DSATUR sia migliore nella stima dall'alto rispetto agli altri due, sempre tenendo di conto della differenza di velocità d'esecuzione (analizzata meglio nella **sezione 5.3**).
Il motivo è da attribuirsi al grande numero di risultati pari merito ottenuto nel primo caso (ben dieci volte più grande del secondo). Per dimensioni più piccole, infatti, i tre algoritmi si comportano in maniera molto simile e non hanno modo di mostrare i loro vantaggi e svantaggi.

2.2. Stima dal basso

Dato un grafo G, la strategia per stimare il suo numero cromatico $\chi(G)$ prevede di:

1. Calcolare una stima dall'alto $k + 1$
2. Verificare se il grafo è colorabile con *k* colori

Se una volta colorato G con $k + 1$ colori si riesce a dimostrare che *k* non sono sufficienti, allora si è anche dimostrato che $\chi(G) = k + 1$.
Serve quindi un metodo rigoroso per verificare la ***k*-colorabilità** di un grafo. A tal proposito si rende necessaria una "codifica" matematica del problema.

2.2.1. Modellizzazione del problema

Un grafo $G(V, E)$ è definito dall'insieme $V = \{1, \ldots, n\}$ dei vertici e dall'insieme $E \subseteq V \times V$ degli archi, dove un elemento (i, j) ∈ E se e solo se il vertice i-esimo e il j-esimo sono collegati da un arco (il grafo non è orientato, quindi analogamente (j, i) ∈ E).
Si vuole:

- Assegnare uno dei *k* colori ad ogni vertice in V
- Assegnare colori distinti a vertici adiacenti

L'idea alla base è associare a ciascuno dei *k* colori una **radice *k*-esima dell'unità**.
Ad esempio, se si vuole testare la *k*-colorabilità con $k = 2$, i due colori che si assegneranno ai vertici saranno:

$$\sqrt[2]{1} = \begin{cases} 1 \\ -1 \end{cases}.$$

Per ogni vertice v_i in V si impone quindi:

$$v_i = \sqrt[2]{1} \quad \Rightarrow \quad v_i^2 - 1 = 0$$

ed effettivamente i due "colori" {-1, 1} sono soluzioni dell'equazione.

Nel caso in cui *k* sia generico, il vincolo di assegnare una radice dell'unità (quindi un colore) ad ogni vertice assumerà forma:

$$v_i = \sqrt[k]{1} \quad \Rightarrow \quad v_i^k - 1 = 0$$

Si ricordi che ogni polinomio di grado *k* ammette *k* radici complesse per il **Teorema fondamentale dell'algebra**.

Ora resta da modellare il vincolo dei vertici adiacenti.

Si considerino i due vertici adiacenti (v_i, v_j) e si analizzi il seguente polinomio:

$$(v_i^k - v_j^k)$$

Questo si annulla per ogni scelta di (v_i, v_j) radici *k*-esime di 1. Se si prova ad espandere:

$$(v_i^k - v_j^k) = (v_i - v_j) \cdot \sum_{L=0}^{k-1} (v_{\mathrm{i}}^{k-L-1} \cdot v_{\mathrm{j}}^L)$$

Ad esempio quando $k = 3$:

$(v_{\mathrm{i}}^3 - v_{\mathrm{j}}^3) = (v_{\mathrm{i}} - v_{\mathrm{j}}) \cdot (v_{\mathrm{i}}^2 + v_{\mathrm{i}}v_{\mathrm{j}} + v_{\mathrm{j}}^2)$

Scomposizione col falso quadrato

Il primo termine del prodotto $(v_{\mathrm{i}} - v_{\mathrm{j}})$ si annulla solo quando $v_{\mathrm{i}} = v_{\mathrm{j}}$, cosa che si vuole evitare.

Il secondo termine, cioè la sommatoria, è uguale a 0 solo quando:

- v_{i} e v_{j} sono radici *k*-esime dell'unità
- $v_{\mathrm{i}} \neq v_{\mathrm{j}}$

che sono esattamente le condizioni cercate.

Sarà quindi sufficiente imporre:

$$\sum_{L=0}^{k-1} v_i^{k-L-1} \cdot v_j^L = 0$$

Si ricordi che il grafo non è orientato, quindi basta una sola condizione per ogni arco.

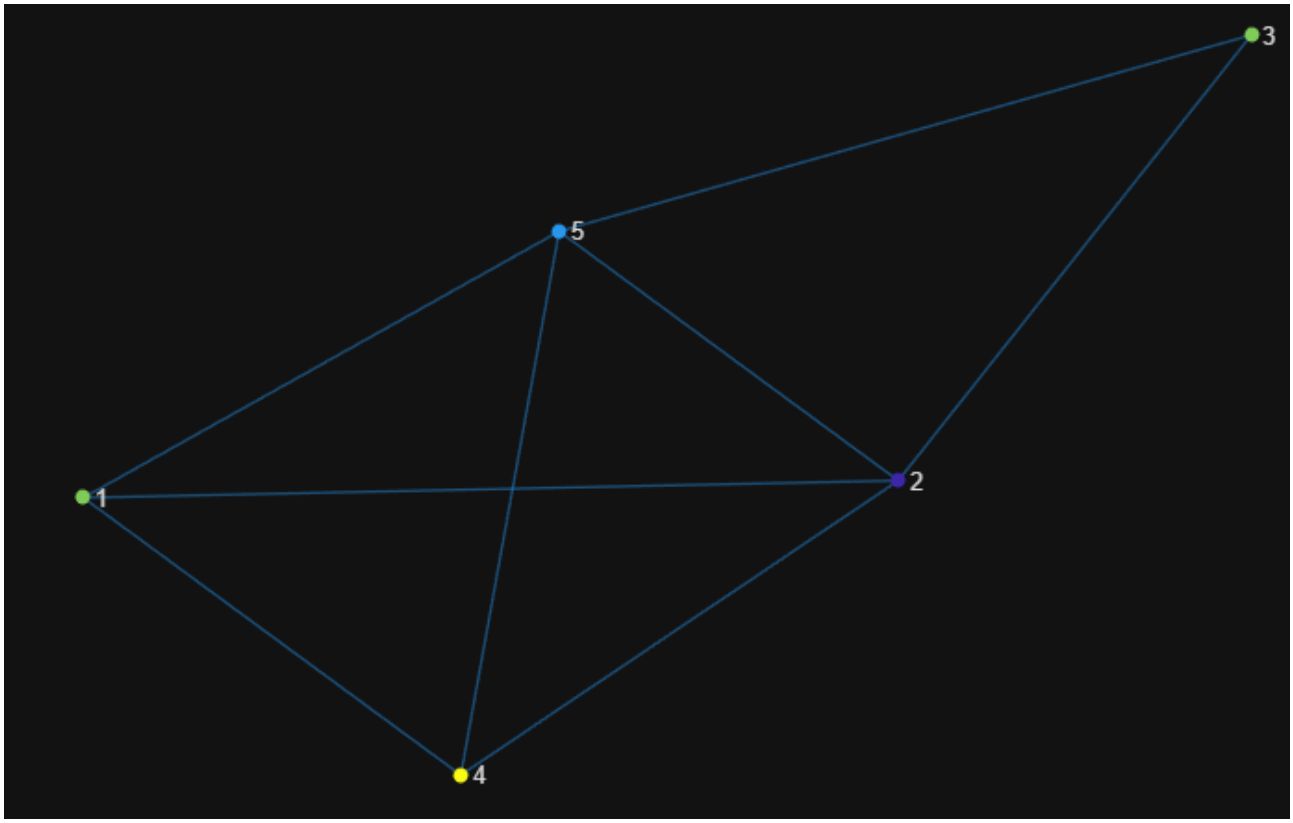
Si ottiene così ottenuto il sistema che rappresenta il problema della *k*-colorabilità:

$$\begin{cases} v_i^k - 1 = 0 & \forall v_i \in V \\ \sum_{L=0}^{k-1} v_i^{k-L-1} \cdot v_j^L = 0 & \forall (i,j) \in E. \end{cases} \quad \textbf{(1)}$$

Se questo sistema di $|V| + |E|$ equazioni e $|V|$ incognite ha soluzione allora il grafo è *k*-colorabile. Quindi, trovata una stima dall'alto $k + 1$, per verificare se $\chi(G) = k + 1$ bisogna dimostrare che il sistema **(1)** non ha soluzione. Nella prossima sezione si costruisce esplicitamente **(1)** per un grafo di piccole dimensioni.

2.2.2. Esempio di costruzione di un sistema per la *k*-colorabilità

Sia dato il grafo in figura. È stata trovata una stima dall'alto pari a 4 con tutti e tre gli algoritmi greedy. Si vuole verificare che il grafo non sia colorabile con $k = 3$ colori.



Si costruisce quindi il sistema di 3-colorabilità. Le prime *n* equazioni (*n* = |V|, pari al numero di nodi, in questo caso cinque) sono:

$$v_i^k - 1 = 0, \qquad i = 1,\ldots,5$$

Le restanti *m* equazioni (*m* = |E|, numero di archi, in questo caso 8) sono:

$$\sum_{L=0}^{k-1} v_i^{k-L-1} \cdot v_j^L = 0, \qquad \forall\,(i,j) \in E$$

Il sistema completo per la 3-colorabilità in questo grafo di 5 nodi e 8 archi è:

$$\begin{cases} v_1^3 - 1 = 0 \\ v_2^3 - 1 = 0 \\ v_3^3 - 1 = 0 \\ v_4^3 - 1 = 0 \\ v_5^3 - 1 = 0 \\ v_1^2 + v_1 v_2 + v_2^2 = 0 \\ v_2^2 + v_2 v_3 + v_3^2 = 0 \\ v_1^2 + v_1 v_4 + v_4^2 = 0 \\ v_2^2 + v_2 v_4 + v_4^2 = 0 \\ v_1^2 + v_1 v_5 + v_5^2 = 0 \\ v_2^2 + v_2 v_5 + v_5^2 = 0 \\ v_3^2 + v_3 v_5 + v_5^2 = 0 \\ v_4^2 + v_4 v_5 + v_5^2 = 0 \end{cases}$$

Come si può evincere, il sistema cresce molto rapidamente con l'aumentare del numero di vertici e archi, ma le considerazioni sulla scalabilità del problema verranno trattate meglio nella **sezione 5.3**.

Il problema che si presenta adesso è stabilire se si è in grado di dimostrare la non risolubilità di un sistema polinomiale. Trovare la soluzione di un sistema non lineare è già costoso di per sé, dimostrare che la soluzione non esiste è un compito ancora più difficile.

Nella prossima sezione viene introdotto lo strumento matematico che si sfrutta per risolvere questo problema.

2.2.3. Teorema degli zeri di Hilbert

Il Teorema degli zeri di Hilbert, o **Nullstellensatz** in tedesco (letteralmente "Teorema dei luoghi di zeri") è un risultato algebrico profondo che si può utilizzare per dimostrare la non risolubilità di un sistema polinomiale. Di seguito una versione del suo enunciato semplificata ma significativa nell'ambito di questo problema:

Sia dato un sistema di equazioni polinomiali $f_1(x) = 0, \ldots, f_s(x) = 0$, dove

$$f_i \in \mathbb{K}[x_1, \ldots, x_n]$$

con $\mathbb{K}$ campo algebrico chiuso.

Il sistema non ammette soluzione in $\mathbb{K}^n$ se e solo se esistono polinomi

$$\beta_1, \ldots, \beta_s \in \mathbb{K}[x_1, \ldots, x_n]$$

tali che:

$$\sum_{i=1}^{s} \beta_i f_i = 1 \qquad \textbf{(A)}$$

Quest'ultima identità prende il nome di "***Certificato Nullstellensatz***" di grado *d*, dove

$$d = \max\{\deg(\beta_i)\}$$

con β_i polinomi in più variabili il cui grado è definito come il massimo grado che appare tra i monomi che li compongono.

Ora il problema si riduce alla ricerca dei polinomi β_i, cioè alla **risoluzione di un sistema lineare**; Tuttavia, il teorema non ci assicura niente sul grado che questi polinomi possono assumere. La strategia sarà quella di prestabilire un grado massimo e generare iterativamente tutti i polinomi β_i con gradi che vanno da 0 al massimo.

Quando si applica questo teorema al problema della stima di $\chi(G)$ possono presentarsi due scenari possibili:

1) Si riescono a trovare polinomi β_i di grado minore o uguale al massimo prestabilito che verificano **(A)**, quindi si è riusciti a dimostrare che il sistema polinomiale della *k*-colorabilità non ha soluzione e $\chi(G) = k + 1$
2) Non si riescono a trovare i polinomi β_i entro il grado prestabilito; Quindi, non si riesce a dimostrare che il sistema è impossibile e di conseguenza non si dimostra che il grafo non è k-colorabile. A questo punto si ripete il procedimento con $k - 1$ colori cercando di dimostrare che il sistema della $(k - 1)$-colorabilità non ha soluzione e così via. In questo secondo scenario la procedura restituisce solamente una stima dal basso.

3. Logica di risoluzione

Lo scopo di questa sezione è illustrare il flusso operativo adoperato nel programma finale senza introdurre dettagli implementativi specifici.

L'oggetto di lavoro iniziale è un grafo dato in ingresso. A partire da questo, la prima operazione svolta è la stima dall'alto del numero cromatico con i tre algoritmi esposti nella **sezione 2.1**. Si manterrà come risultato il minore ottenuto a fronte dell'esecuzione dei tre algoritmi.

Indicando con $k+1$ il valore calcolato al passo precedente, si costruisce il sistema polinomiale per verificare la *k*-colorabilità. Si otterrà un sistema analogo a quello mostrato nella **sezione 2.2.2**.

Per capire se il sistema non è *k*-colorabile si cercherà di calcolare un *certificato Nullstellensatz* di grado *d*.

3.1. Calcolo dei polinomi β_i

Si affronta ora, in maniera esplicita, il problema del calcolo dei polinomi della combinazione lineare **(A)**. Per prima cosa si stabilisce un grado massimo dei polinomi β_i, in accordo a quanto anticipato nella **sezione 2.2.3**. Nel contesto di questo progetto, il grado massimo sarà **quattro**, numero scelto perché risulta quasi sempre sufficiente a calcolare i certificati cercati.

Si comincerà col generare tutti i polinomi β_i di grado 0, poi di grado 1 e così via fino al grado 4.

3.1.1. Metodo intuitivo

Una prima soluzione a cui si potrebbe pensare (ed esplorata anche in una delle prime versioni di questo progetto) prevede di generare, a fronte di $s = n + m$ equazioni polinomiali f_i, s polinomi β_i.

Più nello specifico, generare *s* polinomi di grado 0 significa definire *s* costanti $c_1, \ldots, c_s$ ed effettuare i prodotti:

$$\beta_i f_i \qquad \text{i} = 1, \ldots, \text{s}$$

ottenendo qualcosa della forma:

$$\begin{cases} c_1 f_1 \\ \cdots \\ c_s f_s \end{cases}$$

Riprendendo l'esempio della **sezione 2.2.2** si ottiene:

$$\begin{cases} c_1 v_1^3 - c_1 = 0 \\ c_2 v_2^3 - c_2 = 0 \\ c_3 v_3^3 - c_3 = 0 \\ c_4 v_4^3 - c_4 = 0 \\ c_5 v_5^3 - c_5 = 0 \\ c_6 v_1^2 + c_6 v_1 v_2 + c_6 v_2^2 = 0 \\ c_7 v_2^2 + c_7 v_2 v_3 + c_7 v_3^2 = 0 \\ c_8 v_1^2 + c_8 v_1 v_4 + c_8 v_4^2 = 0 \\ c_9 v_2^2 + c_9 v_2 v_4 + c_9 v_4^2 = 0 \\ c_{10} v_1^2 + c_{10} v_1 v_5 + c_{10} v_5^2 = 0 \\ c_{11} v_2^2 + c_{11} v_2 v_5 + c_{11} v_5^2 = 0 \\ c_{12} v_3^2 + c_{12} v_3 v_5 + c_{12} v_5^2 = 0 \\ c_{13} v_4^2 + c_{13} v_4 v_5 + c_{13} v_5^2 = 0 \end{cases}$$

Tutti i $\beta_i = c_i$ sono ignoti e bisogna imporre:

$$\sum_{i=1}^{s} \beta_i f_i = 1$$

Per ottenere come risultato il polinomio costante 1, bisogna sommare tutti i polinomi del precedente sistema tra loro, imporre a 0 tutti i coefficienti dei termini con la v e imporre a 1 il coefficiente del monomio costante. Quest'ultima equazione in questo esempio ha forma:

$$-c_1 - c_2 - c_3 - c_4 - c_5 = 1.$$

Si ottiene così un sistema lineare, dove le incognite sono i termini c_i, la cui eventuale risolubilità certifica che il grafo non è *k*-colorabile.

È immediato vedere come questo particolare sistema lineare non ammetta soluzione, servono polinomi almeno di grado 1. Il primo di questi avrà forma (nel caso di cinque variabili v):

$$c_1 v_1 + c_2 v_2 + c_3 v_3 + c_4 v_4 + c_5 v_5 + c_6$$

3.1.2. Algoritmo NulLA

Esiste in realtà un modo migliore per cercare i polinomi β_i dove si lavora solamente con i polinomi f_i, senza bisogno di definirne di nuovi.

Per prima cosa occorre definire una base dei monomi con un certo ordinamento, nel progetto si usa l'**ordine lessicografico crescente**. Si veda subito un esempio: supponendo di avere le tre variabili $(v_1,\ v_2,\ v_3)$, la base dei monomi fino a grado 3 sarà:

$$[1,$$

$$v_3, \quad v_2, \quad v_1,$$

$$v_3^2, \quad v_2v_3, \quad v_2^2, \quad v_1v_3, \quad v_1v_2, \quad v_1^2,$$

$$v_3^3, \quad v_2v_3^2, \quad v_2^2v_3, \quad v_2^3, \quad v_1v_3^2, \quad v_1v_2v_3, \quad v_1v_2^2,$$
$$v_1^2v_3, \quad v_1^2v_2, \quad v_1^3]$$

Sono disposti a forma piramidale per una più facile comprensione ma vanno pensati come elementi di una lista.
Notare che i monomi appaiono in ordine di grado: prima quello di grado 0, poi tutti quelli di grado 1 e così via.
Ora si può spiegare meglio come funziona l'ordinamento lessicografico crescente: si considerano gli esponenti delle tre variabili per ogni monomio, ad esempio il monomio $v_1^2 \cdot v_2$ ha come esponenti delle tre variabili $(v_1,\ v_2,\ v_3)$ i valori (2, 1, 0).

Siano dati ora due monomi generici, il primo con esponenti (a_1, a_2, a_3) e il secondo con esponenti (b_1, b_2, b_3).
Il primo è maggiore del secondo (e quindi viene dopo nell'ordinamento) quando:
$(a_1, a_2, a_3) > (b_1, b_2, b_3)$ e questo avviene se $a_1 > b_1$, o se sono uguali, $a_2 > b_2$, o se sono uguali, $a_3 > b_3$.
Questo dovrebbe chiarire il motivo dell'ordine con cui appaiono i monomi in base.

Una volta che si è fissato un ordinamento sull'insieme dei monomi la matrice del sistema **(A)** risulta univocamente determinata.
In particolare, la matrice del sistema lineare conterrà nella colonna i-esima i coefficienti dei monomi che appaiono nel polinomio f_i e nella riga j-esima tutti i coefficienti del j-esimo elemento della base dei monomi. È importante notare come gli unici coefficienti che appaiono siano {-1, 0, 1} con una grande prevalenza di coefficienti uguali a 0, questa proprietà permette di ottimizzare molto il procedimento. Il termine noto sarà un vettore con primo elemento a 1 e il resto a 0.
Se questo sistema lineare ha soluzione allora il certificato esiste ed il sistema polinomiale è impossibile, altrimenti bisogna passare al grado successivo e per ottenerlo **è sufficiente moltiplicare ogni polinomio di grado immediatamente precedente per ogni variabile v_i**.
Questo procedimento prende il nome di **algoritmo NulLA** (Nullstellensatz Linear Algebra).

L'ultimo punto, relativo all'aumento del grado, è il più importante: nel caso dell'esempio nella **sezione 2.2.2** si moltiplica ognuna delle 13 equazioni per ognuna delle 5 variabili $v_1, \ldots, v_5$ ottenendo **65 nuove equazioni** da aggiungere in coda al sistema. Per aumentare nuovamente il grado si moltiplicano le 65 equazioni del passo precedente per tutte le variabili e così via fino al massimo grado prestabilito. Se quest'ultimo non è sufficiente si passa alla prova di $k - 1$ colori.
Da qui deriva la crescita esponenziale del problema, meglio trattata nella **sezione 5.3**. Di seguito uno pseudocodice che riassume quanto appena detto sul procedimento di aumento del grado:

```
input ← sistema polinomiale f iniziale, variabili v_i
grado ← 0
num_eq ← numero di equazioni iniziali
G ← matrice del sistema lineare
b ← vettore con primo elemento a 1 e tutti gli altri a 0 lungo quanto il numero
di righe di G
. . . Si estraggono i coefficienti dei monomi, si inseriscono nella matrice G e
si verifica se esiste la soluzione di Gx = b . . .

Se il sistema lineare ha soluzione allora
      Ritorna grado
Altrimenti
      grado ← grado + 1
      Moltiplica le precedenti num_eq equazioni per ogni variabile v_i e
aggiungile ad f
      num_eq ← numero delle equazioni ottenute (non il totale, solo le nuove)
```

3.1.3. NulLA su campi finiti

Esiste un'ottimizzazione per un problema più specifico di quello trattato finora, vale a dire quello della **3-colorabilità** di un grafo. Lo scopo, come il nome suggerisce, è quello di capire se un grafo dato è 3-colorabile o meno. Si tratta di una proprietà interessante da verificare per molte famiglie di grafi.

L'ottimizzazione in questione riguarda l'applicazione dell'algoritmo NulLA su ***campi finiti***.

L'idea è semplice: invece di lavorare con coefficienti in $\mathbb{R}$ (in particolare {-1, 0, 1}), con conseguenti operazioni *floating-point* e arrotondamenti, si restringe il problema al campo finito $\mathbb{F}_2$ che contiene solo gli elementi {0, 1} e dove si lavora in aritmetica modulo 2. Si risolve infatti il sistema **(A)** applicando l'eliminazione Gaussiana su $\mathbb{F}_2$.

Il vantaggio sta, oltre che nell'efficienza dei calcoli in modulo 2, nell'abbassamento del grado necessario a dimostrare la **non 3-colorabilità** per molti grafi.

Ad esempio, si può dimostrare che i grafi "Ruota dispari" (*Odd-Wheel*, **[Figura 3]**) richiedono sempre un certificato di grado 4 per attestare che non sono 3-colorabili se si lavora in $\mathbb{R}$, ma il grado si abbassa sempre ad 1 quando si lavora su $\mathbb{F}_2$, aumentando di molto l'efficienza.

È possibile estendere questo ragionamento alla *k*-colorabilità con campi finiti $\mathbb{F}_q$, dove q è primo rispetto a *k*, ma va oltre gli scopi di questo progetto.

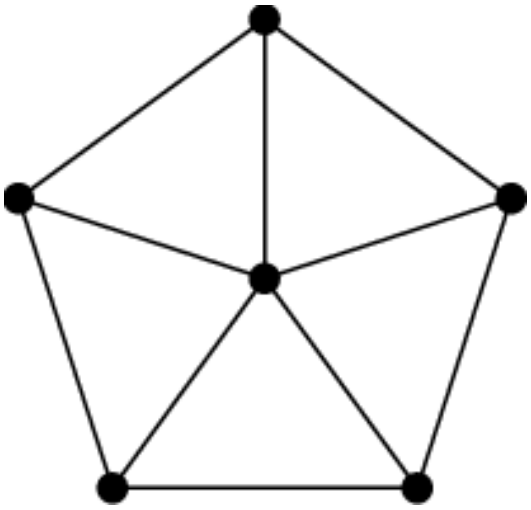


Figura 3
Un grafo "Odd-Wheel" ha un ciclo esterno formato da un numero dispari di vertici e un vertice centrale collegato a tutti gli altri detto "hub"

4. Ambiente di lavoro

Il progetto è stato scritto quasi interamente in ambiente MATLAB versione 25.1.0.2943329 (R2025a) con aggiunta dell'Add-On:

MATLAB Support for MinGW-w64 C/C++/Fortran Compiler

Quest'ultimo permette di compilare e richiamare funzioni in C++ direttamente dall'ambiente MATLAB per una maggiore efficienza.

L'uso di MATLAB è risultato essere la scelta più naturale dal momento che bisogna trattare polinomi, matrici e risoluzioni di sistemi lineari.
L'Add-On utilizzato si è rivelato utile nell'eliminare un pesante collo di bottiglia, ma verrà meglio esplicato nella sezione successiva.

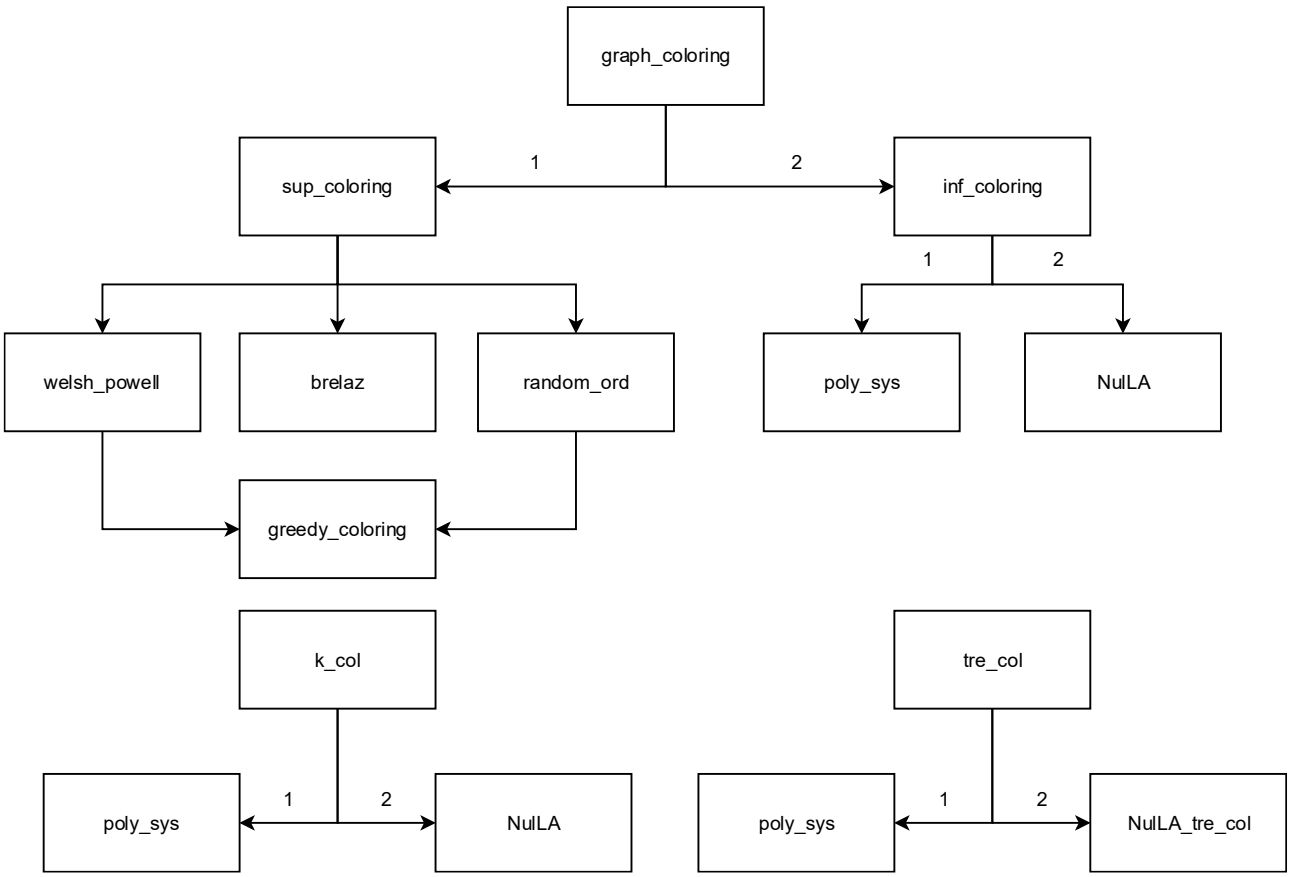
L'idea iniziale prevedeva l'uso di un ulteriore Add-On, ossia il ***Symbolic Math Toolbox***, che permette di lavorare con simboli matematici espliciti (i sistemi di equazioni scritti in questo documento sono stati generati con questo Add-On).
Serviva per implementare l'algoritmo esemplificato nella **sezione 3.1.1** ma è stato rapidamente soppiantato per motivi di efficienza.
È stato tuttavia molto utile per poter visualizzare la forma assunta dal problema e capire a fondo come organizzare il lavoro.

5. Programma sviluppato

In questa sezione si tratta per prima la struttura del programma finale, che ripercorre quanto esposto nella **sezione 3**, successivamente vengono esposti estratti di codice MATLAB e infine si affronta una trattazione sulla scalabilità del problema e sulla velocità di esecuzione a fronte di input di tipo diverso.

5.1. Struttura del programma

La struttura riflette la strategia proposta precedentemente. Il programma è composto da un insieme di funzioni MATLAB salvate in file separati. Ecco di seguito un albero che schematizza come le funzioni si richiamano tra loro:



Non sono state riportate le funzioni di supporto più specifiche, analizzate nella sezione successiva, ma solo lo scheletro principale del programma.

Ci sono tre gruppi di funzioni separati. Quello principale è in alto e come si può vedere la funzione cardine del progetto è "*graph_coloring*" che a partire da un grafo restituisce il suo numero cromatico. Per prima cosa viene calcolata la stima dall'alto con la funzione "*sup_coloring*" precisamente come spiegato nella **sezione 2.1**. Il risultato viene passato poi alla funzione "*inf_coloring*" che costruisce il sistema polinomiale con "*poly_sys*" e poi esegue l'algoritmo *NulLA* con la funzione omonima.

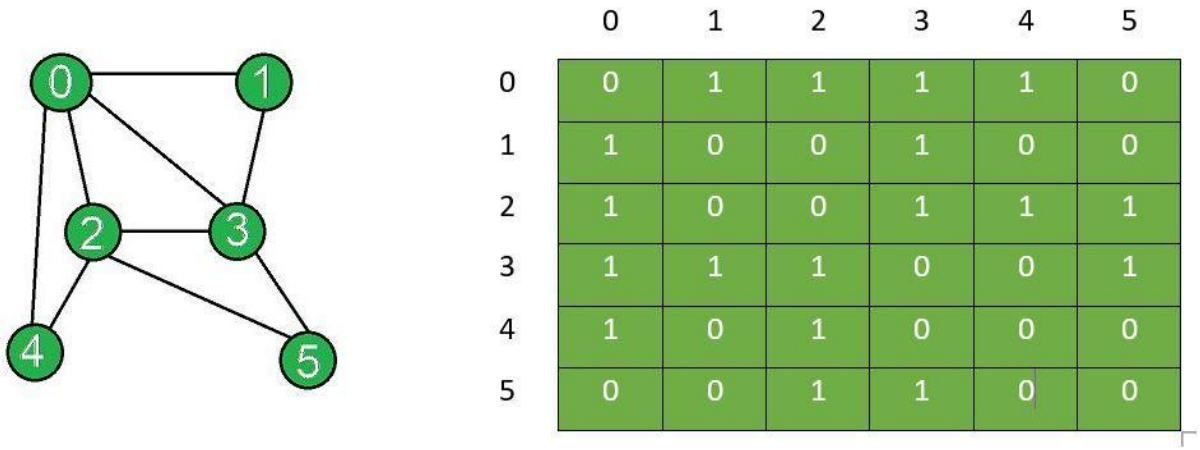
I due gruppi sottostanti, invece, servono per operazioni leggermente diverse:

- "*k_col*" prende in ingresso un grafo e un parametro *k* e attesta la *k*-colorabilità di quel grafo
- "*tre_col*" verifica la 3-colorabilità di un grafo in input sfruttando l'algoritmo NulLA su campi finiti spiegato alla **sezione 3.1.3**

5.2. Codice sorgente

Si procede ora alla spiegazione delle funzioni più importanti con aggiunta di estratti di codice ove necessario.

Molte funzioni del programma prendono in ingresso l'argomento ***"A"***. Si tratta della matrice di adiacenza del grafo dato in input, dove l'elemento $A_{i,j}$ è 1 se il vertice i è adiacente al vertice j, 0 altrimenti **[Figura 4]**. Nel programma si lavora con le matrici di adiacenza di grafi non orientati e senza *self-loop* quindi nessun vertice è mai collegato a sé stesso (la diagonale della matrice è sempre 0).



	0	1	2	3	4	5
0	0	1	1	1	1	0
1	1	0	0	1	0	0
2	1	0	0	1	1	1
3	1	1	1	0	0	1
4	1	0	1	0	0	0
5	0	0	1	1	0	0

Figura 4
Esempio di grafo con relativa matrice di adiacenza

Seguendo l'albero principale delle funzioni:

- `graph_coloring(A)`

È la funzione principale e il suo scopo è restituire la stima del numero cromatico di un grafo passato come argomento.

Il suo codice è semplice, contiene solo la chiamata ad altre due funzioni: *sup_coloring* per ottenere la stima dall'alto che passa poi ad *inf_coloring* per la stima dal basso.

Se le due stime coincidono allora si conclude che quel valore è il numero cromatico, altrimenti restituisce un intervallo di appartenenza. Si analizza ora la diramazione:

5.2.1. Funzioni per la stima dall'alto

- `sup_coloring(A)`

Il suo scopo è richiamare le funzioni *welsh_powell*, *brelaz* e *random_ord* che implementano i tre algoritmi greedy trattati nella **sezione 2.1**. Ognuna di queste tre restituisce la sua stima dall'alto e la funzione *sup_coloring* restituisce in output la più piccola (che servirà come argomento ad *inf_coloring*).

- `welsh_powell(A)`

Implementa l'algoritmo omonimo ordinando i vertici in ordine di grado e colorandoli con la funzione *greedy_coloring*. Ritorna il numero di colori utilizzato.

Osservando la **[Figura 4]** si nota come la somma degli elementi sulla riga i-esima corrisponda al grado del vertice i-esimo. È sufficiente quindi calcolare tutti i gradi, disporli in ordine decrescente e ricavare i vertici corrispondenti ad ogni grado. Su MATLAB:

```
%   Le somme delle righe di A vanno nel vettore "gradi" in modo che
%   il suo i-esimo elemento rappresenti il grado dell'i-esimo vertice
gradi = sum( A(1:end, :), 2);

%   Si ordina il vettore "gradi" in ordine decrescente e si salvano nel vettore
%   "ordine" gli indici originali degli elementi ordinati.
%
%   In questo modo il vettore "ordine" contiene i vertici in ordine
%   decrescente di grado
[~, ordine] = sort(gradi, 'descend');


%   Dopo aver ordinato i vertici si colorano (WP è il ritorno della funzione)
[WP, colori] = greedy_coloring(A, ordine);
```

- `random_ord(A)`

Si dispongono i vertici casualmente e si colorano con *greedy_coloring* per un numero predefinito di volte (in questo caso 10). Si ritorna il numero minimo di colori utilizzati tra tutte le iterazioni:

```
%   Numero di righe di A (corrisponde al numero di vertici del grafo)
n = size(A,1);

%   Numero di colorazioni casuali effettuate
iterazioni = 10;

for i = 1:iterazioni
    %   Si usa la funzione nativa di MATLAB "randperm" che restituisce un
    %   ordinamento casuale di interi da 1 ad n
    ordine_random = randperm(n);

    %   Dopo aver ordinato i vertici si colorano
    [risultati(i), colori{i}] = greedy_coloring(A, ordine_random);
end

%   "rand" è il ritorno della funzione
[rand, idx] = min(risultati);
```

- `greedy_coloring(A, ordine)`

Accetta come argomento un vettore "ordine" in cui appaiono i vertici nell'ordine in cui vanno colorati (che viene definito nelle funzioni *welsh_powell* e *random_ord*, le quali fanno uso di *greedy_coloring*).

Lo scopo è colorare i vertici in ordine usando colori già usati per i vertici precedenti, se possibile.
I colori sono codificati con numeri naturali crescenti e si fa uso di un array "*colori"* con un elemento per vertice. L'elemento i-esimo di "*colori"* rappresenta il colore del vertice i-esimo. Un vertice ancora da colorare ha colore 0.

Quando si cerca di colorare il vertice i-esimo, bisogna prima di tutto tenere traccia dei colori dei suoi vicini. Per farlo, si trovano i vicini nella riga i-esima di A e i loro colori nel vettore "*colori*". A questo punto si cerca di assegnare al vertice i-esimo il colore 1 e, se è già usato da un vicino, si incrementa il colore finché non se ne trova uno libero.
La parte del codice MATLAB che cerca di colore ogni vertice è la seguente:

```
%   Numero vertici
n = size(A,1);

%   Vettore il cui elemento i-esimo contiene il colore del vertice
%   i-esimo (un vertice non ancora colorato ha colore 0)
colori = zeros(n, 1);

%   Si scorrono gli elementi del vettore "ordine" (che ha dimensione n
%   pari al numero di vertici)
for i = 1:n

    %   L'elemento attuale di "ordine" rappresenta il vertice da
    %   colorare, quindi la prossima riga della matrice di adiacenza A
    %   da controllare
    riga = ordine(i);

    %   Array coi vertici adiacenti a quello attuale
    vicini = find(A(riga, :) == 1);

    %   Al vettore "proibiti" verranno aggiunti i colori già assegnati
    %   ai vertici adiacenti senza duplicati
    proibiti = unique(colori(vicini));

    %   Si cerca di dare il colore 1 al vertice attuale
    colore = 1;

    %   Finché il colore è nella lista dei colori proibiti si
    %   incrementa, il primo colore ammesso trovato verrà dato al
    %   vertice
    while(ismember(colore, proibiti))
        colore = colore + 1;
    end

    colori(riga) = colore;

end
```

- `brelaz(A)`

Implementa l'algoritmo DSATUR, colorando i vertici in modo simile a *greedy_coloring* ma valutando il prossimo vertice da colorare passo per passo.

Si usano strutture dati simili ai casi precedenti, con un vettore "*colori*" e un vettore "*gradi*". Si aggiunge un vettore "*sat*" con i gradi di saturazione dei vertici, ovvero il numero di colori distinti a cui sono adiacenti.

A questo punto il prossimo vertice da colorare è quello con massimo grado di saturazione e in caso di parità, quello col massimo grado (se anche qui ci sono pari merito si prende il primo della lista):

```
%   Ciclo per colorare gli n vertici
for i = 1:n

    %   Vertici ancora da colorare
    da_colorare = find(colori == 0);

    %   Tra questi si cercano i vertici (o il vertice) col massimo grado di
    %   saturazione
    max_sat = max(sat(da_colorare));

    %   "idxs" è un vettore che contiene i vertici col massimo grado di
    %   saturazione (tra quelli ancora da colorare)
    idxs = find(sat == max_sat);

    %   Dai vertici trovati si tolgono quelli già colorati (quando si
    %   cerca la saturazione massima tra vertici da colorare,
    %   potrebbero esserci vertici già colorati con quella stessa
    %   saturazione che finiscono nel vettore "idxs")
    gia_colorati = find(colori ~= 0);
    idxs = setdiff(idxs, gia_colorati);

    %   Tra i vertici trovati si sceglie quello di grado massimo per
    %   gestire i pari merito

    %   Si trova il vertice di grado massimo tra quelli trovati in idxs
    %   con una ricerca sul sotto-vettore "gradi(idxs)"
    [~, local_idx] = max(gradi(idxs));

    %   Questa istruzione serve per recuperare il vertice corretto a partire
    %   dall'indice trovato nel sottovettore "gradi(idxs)"
    next_v = idxs(local_idx);

    %   Ora "next_v" rappresenta il prossimo vertice da colorare. Si
    %   cercano i suoi vicini e i loro colori
    vicini = find(A(next_v, :) == 1);
    proibiti = unique(colori(vicini));

    %   Si colora il vertice "next_v" col primo colore disponibile
    colore = 1;

    while(ismember(colore, proibiti))
        colore = colore + 1;
    end

    colori(next_v) = colore;









    %   Si aggiornano ora i gradi di saturazione
    %
    %   Gli unici vertici interessati saranno quelli adiacenti a quello
```

```
    %   appena colorato, già presenti nell'array "vicini"

    for u = vicini
        %   Per ogni vertice adiacente bisogna contare il numero di colori
        %   distinti dei loro vicini

        %   Considerato il vicino i-esimo del nodo appena colorato, si
        %   salva il colore dei suoi vicini
        colori_vicini = colori(A(u,:) == 1);

        %   Si tengono solo i colori maggiori di 0
        colori_vicini = colori_vicini(colori_vicini > 0);

        %   Si conta il numero di colori distinti
        sat(u) = numel(unique(colori_vicini));
    end
end
```

5.2.2. Funzioni per la stima dal basso

- `inf_coloring(A, stima_alto)`

Riceve come argomento *stima_alto*, il valore ritornato da *sup_coloring*. Lo scopo è verificare se il grafo può essere colorato con $stima\_alto - 1$ colori. I valori di ritorno sono la stima dal basso ottenuta e il grado del certificato Nullstellensatz trovato.

Questa funzione è ricorsiva e l'idea è costruire il sistema di *k*-colorabilità ed eseguire l'algoritmo *NulLA*.

Se si dimostra che *k* colori non bastano allora si ritorna *k+1* come stima dal basso, altrimenti la funzione *inf_coloring* richiama sé stessa ma passandosi come parametro $stima\_alto - 1$ in modo da testare la (*k-1*)-colorabilità.

Il caso base è $stima\_alto == 1$ o $stima\_alto == 0$ e la *stima_basso* ritornata in questi casi è 1.

```
function [stima_basso, cert] = inf_coloring(A, stima_alto)

    %   Caso base
    if (stima_alto == 1 || stima_alto == 0)
        stima_basso = 1;
        cert = 0;
        return;
    end

    %   Abbiamo trovato una stima dall'alto del numero cromatico k+1,
    %   vogliamo dimostrare che k colori non bastano
    k = stima_alto - 1;

    %   Funzione che restituisce il sistema delle equazioni polinomiali che
    %   codifica il problema della colorazione di un grafo con k colori
    f = poly_sys(A, k);

    %   Bisogna cercare di dimostrare che questo sistema non ha soluzione
    %   sfruttando il Teorema di Hilbert
    cert = NulLA(A, f);

    if ~isa(cert, "string")
        stima_basso = stima_alto;
        return;

    else
        %   Se NulLA ritorna una stringa, significa che la funzione non è
        %   riuscita a dimostrare che k colori non bastano, si riprova con
        %   k-1
        [stima_basso, cert] = inf_coloring(A, (stima_alto - 1));
        return;
    end
end
```

- `poly_sys(A, k)`

Costruisce e ritorna il sistema polinomiale *f* della *k*-colorabilità per il grafo A.
L'idea adottata per la rappresentazione del sistema prevede di usare come struttura dati un vettore di *struct* con due campi: una matrice *esponenti* e un vettore *coefficienti*. Ogni elemento del vettore (quindi ogni singola struct) rappresenta un polinomio f_i.
Per ognuno, la matrice *esponenti* contiene una riga per ogni monomio che appare nel polinomio. Alla riga i-esima ci sono gli esponenti delle *n* variabili all'interno del monomio i-esimo (dove *n* è il numero di vertici del grafo).
Il vettore *coefficienti* rappresenta invece i coefficienti dei monomi, quindi avrà un elemento per ogni riga di *esponenti*. Di seguito un esempio:

$$v_1^2 + v_1 v_2 + v_2^2$$

Questo polinomio è composto da tre monomi. Supponendo che il grafo abbia tre vertici e che le variabili siano quindi $[v_1, v_2, v_3]$, la matrice *esponenti* nella struct relativa a questo polinomio avrà forma:

$$\begin{bmatrix} 2 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 2 & 0 \end{bmatrix}$$

La prima riga rappresenta v_1^2, quindi la variabile v_1 ha esponente 2 mentre v_2 e v_3 hanno esponente 0. Questo stesso procedimento è ripetuto per ogni monomio.
Il vettore *coefficienti* conterrà invece [1, 1, 1], ovvero i coefficienti dei 3 monomi.
Notare che tutti i coefficienti sono sempre uguali a 1 o -1, come nel seguente esempio:

$$v_1^3 - 1$$

Che ha *esponenti* $\begin{bmatrix} 3 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$ e *coefficienti* [1, -1]

Il monomio costante è sempre rappresentato da una riga di esponenti nulla. Il primo esempio corrisponde ad un'equazione di vincolo di colorazione per vertici adiacenti, il secondo è un'equazione di colorazione di un singolo vertice. Di seguito un'analisi parte per parte della funzione MATLAB:

Si definisce il vettore di struct che conterrà i $|V| + |E|$ polinomi pari al numero di vertici + numero di archi del grafo ($n + m$). Il numero degli archi corrisponde alla metà del numero di elementi uguali a 1 nella matrice di adiacenza in quanto il grafo è non orientato e ad ogni arco corrispondono due elementi a 1.

```
%   Numero di vertici
n = size(A,1);
%   Numero di archi (con la funzione "nnz": number of non-zero elements)
m = nnz(A) / 2;

dim_sys = n + m;

f(dim_sys) = struct("esponenti", [], "coefficienti", []);
```

Si inseriscono i primi *n* polinomi che hanno forma $v_i^k - 1$. Sono tutti definiti da due monomi, quindi *esponenti* avrà 2 righe (di cui la seconda di tutti 0) e *coefficienti* i due elementi [1, -1]:

```
%   Le prime n funzioni hanno forma "v(i)^k - 1"
for i = 1:n
    esponenti = zeros(2, n);

    %   "v(i)^k" ha esponente k in posizione i e coefficiente 1
    %   "-1" ha tutti esponenti a 0 e coefficiente -1
    esponenti(1, i) = k;
    coefficienti = [1 -1];

    f(i).esponenti = esponenti;
    f(i).coefficienti = coefficienti;
end
```

Per i seguenti *m* polinomi, quelli relativi ai colori di vertici adiacenti, bisogna ricavarsi gli archi del grafo per capire quali vertici sono collegati. Essendo il grafo non orientato la sua matrice di adiacenza è simmetrica quindi basta analizzare la parte triangolare superiore (o inferiore):

```
%   Le restanti m funzioni rappresentano le condizioni sui vertici adiacenti

%   Per poterle scrivere bisogna ricavarsi l'insieme degli archi a
%   partire dalla matrice di adiacenza

%   "riga" e "colonna" contengono gli indici di riga e di colonna degli
%   elementi diversi da 0 nella metà superiore di A (il grafo è non
%   orientato)
[riga, colonna] = find(triu(A));

%   Le righe di "archi" contengono le m coppie di vertici collegati tra loro
archi = [riga(:) colonna(:)];
```

A questo punto "*archi*" è una matrice di *m* righe (numero di archi) e due colonne che contengono, alla riga i-esima, gli indici dei vertici collegati dall'arco i-esimo. Avendo ad esempio un grafo con un vertice 1 collegato ad un vertice 2 e ad un vertice 3, "*archi*" avrà forma:

$$\begin{bmatrix} 1 & 2 \\ 1 & 3 \end{bmatrix}$$

Si inseriscono a questo punto i restanti *m* polinomi, ciascuno di *k* termini:

```
for i = (n + 1) : (n + m)

    %   Si salvano in v1 e v2 gli indici dei 2 vertici legati dall'arco
    %   attuale
    v1 = archi(i - n, 1);
    v2 = archi(i - n, 2);

    %   Per ogni arco tra 2 vertici si avrà un polinomio di k termini
    %
    %   Serve quindi una riga di "esponenti" e un elemento di
    %   "coefficienti" per monomio

    esponenti = zeros(k, n);
    coefficienti = ones(1, k);

    %   Si danno gli esponenti corretti ai k monomi
    for l = 0 : (k-1)

       esponenti(l + 1, v1) = k - 1 - l;
       esponenti(l + 1, v2) = l;

    end

    f(i).esponenti = esponenti;
    f(i).coefficienti = coefficienti;
end
```

- `NulLA(A, f)`

Questa funzione è il motore per il calcolo della stima dal basso. Prende in ingresso la matrice di adiacenza e il sistema polinomiale *f* calcolato con *poly_sys* e ritorna in uscita il grado del certificato Nullstellensatz calcolato o, se non riesce a dimostrare che *f* è impossibile, ritorna la stringa "END".

Lo scopo della funzione è costruire una matrice G con una colonna per ogni polinomio ed una riga per ogni elemento nella base dei monomi fino al grado massimo che appare nel sistema *f*. Questa matrice conterrà i coefficienti dei monomi di ogni f_i in *f*.
Successivamente si cerca una soluzione al sistema lineare

$$G \cdot c = b$$

con b vettore colonna di tutti 0 tranne il primo elemento uguale a 1 (che corrisponde al monomio costante 1). Se questo sistema lineare ha soluzione significa che esistono i polinomi β_i tali per cui

$$\sum_{i=1}^{s} \beta_i f_i = 1$$

dimostrando che il sistema polinomiale *f* non ha soluzione.

L'idea generale è eseguire un ciclo per il calcolo dei vari certificati partendo dal grado 0 e arrivando al grado massimo prestabilito (nel codice è il grado 4) espandendo ad ogni iterazione il sistema *f* e la matrice G e cercando una soluzione del sistema lineare ottenuto.
Inserire i coefficienti dei monomi di f_i in G non è un'operazione banale. Si devono infatti inserire dei valori nella colonna i-esima (si ricorda che ogni colonna corrisponde ad un polinomio in *f*) e nelle righe corrette. La parte complicata è proprio capire quali siano le righe da modificare.
Nella **sezione 3.1.2** è stato spiegato l'ordinamento lessicografico della base dei monomi ed è qui che viene utilizzato: la riga j-esima di G contiene tutti i coefficienti del j-esimo monomio della base, il quale apparirà in alcuni polinomi f_i. Ad esempio, tutti i coefficienti del monomio costante 1 saranno alla prima riga (visto che 1 è il primo elemento della base), tutti quelli di v_1 saranno alla quarta riga e così via.
Dato un monomio, si descrive ora un metodo per calcolare la sua posizione in base. A tal proposito si osservi il seguente esempio.

Sia data la base dei monomi in ordine lessicografico crescente di grado massimo 2 e formata da 3 variabili:

$$[1,$$

$$v_3, \quad v_2, \quad v_1,$$

$$v_3^2, \quad v_2v_3, \quad v_2^2, \quad v_1v_3, \quad v_1v_2, \quad v_1^2]$$

Si supponga di voler cercare la posizione del monomio v_1v_2 che in questo caso deve risultare essere 9.

Per prima cosa si osserva che è un monomio di grado 2, quindi viene dopo tutti quelli di grado inferiore (cioè, minore o uguale ad 1). Il numero di monomi di grado **minore o uguale** a *d* formati da *n* variabili è dato dal coefficiente binomiale:

$$\binom{n + d}{d}$$

In questo esempio $n = 3$ e $d = 1$ quindi il coefficiente binomiale vale 4, che torna.

Ora bisogna contare il numero di monomi di grado esattamente 2 che precedono v_1v_2 nella base (ci si aspetta di trovare il valore 4 visto che v_1v_2 è il quinto monomio di secondo grado). Si ricorda che nell'ordine lessicografico si tiene conto degli esponenti dei monomi e vale che un monomio è maggiore di un altro (quindi viene dopo in base) se:

$$(a_1, a_2, a_3) > (b_1, b_2, b_3)$$

e questo avviene quando $a_1 > b_1$, o se sono uguali, $a_2 > b_2$, o se sono uguali, $a_3 > b_3$.

Ciò significa che tutti i monomi che precedono (a_1, a_2, a_3) hanno sicuramente un valore minore per il primo esponente o per il secondo o per il terzo.

v_1v_2 ha esponenti $(1, 1, 0)$, quindi decrementando quello più a sinistra e costruendo tutti i possibili monomi di grado 2 con le restanti variabili, questi precederanno sicuramente il monomio iniziale.

Una volta decrementato fino a 0 si riporta al valore originale e si decrementa l'esponente successivo, così via fino all'ultimo. Contando ad ogni fase il numero di monomi possibili si ottiene la posizione del monomio cercato in base tra quelli di grado uguale.

Nel caso del monomio v_1v_2 si parte da $(1, 1, 0)$. Si decrementa l'esponente più a sinistra ottenendo $(0, 1, 0)$. Ora, tutti i possibili monomi di grado 2 che si possono definire col primo esponente fissato a 0 sono:
$(0, 0, 2)$, $(0, 1, 1)$, $(0, 2, 0)$, cioè v_3^2, v_2v_3 e v_2^2. Per contare quanti sono si tiene conto del fatto che il numero di monomi di grado **esattamente** *d* e formati da *n* variabili è dato dal coefficiente binomiale:

$$\binom{n + d - 1}{d}$$

In questo caso $n = 2$ (il primo esponente ha valore fissato), $d = 2$ e il coefficiente binomiale vale 3 che è proprio il numero di monomi trovati.
Il primo esponente è arrivato a 0, quindi si rifissa al suo valore originale e si decrementa il successivo ottenendo $(1, 0, 0)$. Seguendo lo stesso procedimento di prima, tutti i possibili monomi di grado 2 che si possono ottenere con i valori dei primi due esponenti fissati sono (o meglio, è):
$(1, 0, 1)$ che corrisponde a v_1v_3.
La formula del coefficiente binomiale con $n = 1$ e $d = 2$ vale effettivamente 1.
Anche il secondo esponente è stato decrementato fino a 0, quindi in teoria bisognerebbe fissare i primi due esponenti e decrementare l'ultimo. In realtà questo è scorretto perché, tralasciando innanzitutto che il terzo esponente è già a 0, decrementandolo e fissando tutti gli altri ai loro valori originali non si possono ottenere più monomi del grado cercato ma solo di grado inferiore.
Ad esempio, se si arrivasse a questo punto dell'algoritmo con $(1, 0, 1)$ dove i primi due esponenti sono già fissati, se si decrementasse il terzo si otterrebbe $(1, 0, 0)$ con tutti e tre gli esponenti fissati, ottenendo quindi solo un monomio di grado 1, già contato con il primo coefficiente binomiale mostrato. Quindi il processo è terminato e sommando i valori ottenuti $3 + 1 = 4$ si ottiene proprio il numero di monomi di grado esattamente 2 che precedono v_1v_2.
Sommando ora questo valore al numero di monomi di grado inferiore trovato precedentemente si ottiene $4 + 4 = 8$ che è proprio il numero totale di monomi che precedono v_1v_2 in base. Per la posizione effettiva è sempre sufficiente sommare 1.

Il procedimento appena spiegato è implementato tramite delle funzioni di supporto, si ritiene opportuno analizzarle prima di illustrare il codice della funzione *NulLA* vera e propria:

- `precalc_binom(max_n, max_k)`
- `double fast_nchoosek(unsigned int n, unsigned int k)`

Come si vede nell'esempio precedente, ci sono da calcolare diversi coefficienti binomiali. Esiste una funzione MATLAB chiamata *nchoosek(n,k)* che permette di calcolare il coefficiente binomiale $\binom{n}{k}$, tuttavia quando il numero di chiamate e le dimensioni degli argomenti crescono molto questa risulta essere troppo lenta. Per rendere più efficiente il programma è stata scritta la funzione *precalc_binom* che verrà richiamata una sola volta all'inizio di *NulLA* con lo scopo di precalcolare tutti i coefficienti binomiali necessari durante l'esecuzione, restituendoli all'interno di una matrice *binom_table*:

```
function binom_table = precalc_binom(max_n, max_k)

    binom_table = zeros(max_n+1, max_k+1);

    for n = 0:max_n

        for k = 0:min(n, max_k)
            binom_table(n+1, k+1) = fast_nchoosek(n, k);
        end

    end

end
```

Per migliorare ulteriormente le prestazioni si è rinunciato completamente all'uso della funzione nativa *nchoosek* e si utilizza invece una funzione C++ *fast_nchoosek*, più efficiente della versione MATLAB. È proprio qui che entra in gioco l'Add-On citato nella **sezione 4** il quale ha permesso di rimuovere uno stringente collo di bottiglia.
Di seguito la parte significativa della sua implementazione C++:

```
include "mex.h"

//  Funzione efficiente per calcolare il coefficiente binomiale C(n, k)
double fast_nchoosek(unsigned int n, unsigned int k) {
    if (k > n) return 0;

    // Sfrutta la simmetria: C(n, k) = C(n, n-k)
    if (k > n - k)
        k = n - k;

    double result = 1.0;

    for (unsigned int i = 1; i <= k; ++i) {
        result *= (n - k + i);
        result /= i;
    }

    return result;
}
```

- `posizione_monomio(esponenti, binom_table)`

Implementa l'algoritmo per trovare la posizione di un monomio nella base dei monomi. Prende come argomenti il vettore di esponenti che rappresenta il monomio cercato e la *binom_table* restituita dalla funzione *precalc_binom* per evitare di dover calcolare i coefficienti binomiali volta per volta.
Per prima cosa si calcola il numero di monomi di grado minore a quello considerato, poi si effettua la ricerca della posizione in base tra i monomi con lo stesso grado:

```
%   Numero delle variabili
    n = length(esponenti);

%   Grado del monomio passato come argomento
grado = sum(esponenti);

if(grado == 0)
    pos = 1;
    return;
end

%   Il monomio viene dopo tutti quelli di grado minore
%   ("disp" sta per displacement)

n_binom = n + grado - 1;
k_binom = grado - 1;




%   Questo controllo evita accessi inutili a binom_table
if k_binom < 0 || k_binom > n_binom

    disp = 0;
```

```
else

    disp = binom_table(n_binom + 1, k_binom + 1);

end

somma = 0;
esp = [0, cumsum(esponenti)];

for i = 1 : length(esponenti) - 1

    %   a1, a2, a3, ...
    ai = esponenti(i);
    ei = esp(i);

    var_restanti = length(esponenti) - i;

    t = grado - ai - ei;

    %   Si decrementa "ai" fino a 0
    for k = 1:ai

        %   Quando si decrementa "ai" bisogna costruire monomi con le
        %   variabili alla sua destra, quindi il grado di questi monomi
        %   sarà dato da:
        %
        %   Il grado originale ("grado") meno gli esponenti fissati
        %   (quelli a sinistra di "ai", quindi gli esponenti da 1 a i-1)
        %   meno il grado di "ai" decrementato (ai - k)

        d = t + k;

        %   Si contano tutti i monomi di grado esattamente d sulle
        %   variabili restanti

        n_binom = var_restanti + d - 1;
        k_binom = d;

        coeff = binom_table(n_binom + 1, k_binom + 1);
        somma = somma + coeff;

    end

end

pos = disp + somma + 1;
end
```

Si illustra infine il codice della funzione *NulLA* parte per parte.

Per prima cosa si definiscono le variabili utili e si richiama la funzione *precalc_binom*. Da notare l'uso dei vettori *indici_riga*, *indici_colonna* ed *elementi*. Questi serviranno come argomenti al momento della costruzione della matrice G per mezzo della funzione *sparse*. G sarà infatti formata prevalentemente da elementi 0 e, in MATLAB, le matrici di tipo *sparse* servono proprio per risparmiare memoria, salvando solamente la posizione e il valore degli elementi diversi da 0. Questa è l'ottimizzazione accennata alla fine della **sezione 3.1.2**.

```
function cert = NullA(A, f)

    %   Numero di vertici
    n = size(A, 1);

    %   Bisogna tenere traccia della dimensione del sistema polinomiale che
    %   varia ad ogni iterazione

    %   "dim_vecchia" rappresenta l'indice dell'ultimo elemento aggiunto
    %   all'iterazione precedente
    dim_vecchia = 0;

    %   "dim_nuova" rappresenta il numero di elementi aggiunti
    %   all'iterazione precedente
    dim_nuova = length(f);

    %   Vettori per costruire la matrice sparsa finale
    indici_riga = [];
    indici_colonna = [];
    elementi = [];

    %   Si cercano polinomi di grado al massimo "max_grado"
    max_grado = 4;

    binom_table = precalc_binom(n + max_grado*n, n + max_grado);
```

Comincia ora il ciclo per cercare il certificato Nullstellensatz partendo dal grado 0 e arrivando al massimo al grado *max_grado*. Come prima cosa nel ciclo si incontra una condizione *if* in cui si entra dalla seconda iterazione in poi. Dopo la prima iterazione bisognerà infatti aumentare il grado ed espandere il sistema *f* moltiplicando ogni polinomio calcolato all'iterazione precedente per ogni variabile del sistema (si ricorda che c'è una variabile per vertice, si pensi all'esempio nella **sezione 2.2.2**).

Notare che dato un polinomio rappresentato con la matrice degli esponenti, moltiplicarlo per l'i-esima variabile significa incrementare di 1 la colonna i-esima della matrice degli esponenti e lasciare invariati i coefficienti:

```
for d = 0:max_grado

    if(d > 0)

        %   Dalla seconda iterazione in poi bisogna moltiplicare ogni
        %   elemento di f calcolato all'iterazione precedente per ogni
        %   variabile v e aggiungere i risultati in coda ad f
        %
        %   Data la rappresentazione con la matrice degli esponenti,
        %   moltiplicare un polinomio per una variabile "vi" significa
        %   incrementare di 1 la colonna i-esima

        %   Ad "f" si aggiungono "n" * "dim_nuova" elementi, si
        %   preallocano
        old_length = length(f);
   f(old_length + n * dim_nuova) = struct("esponenti", [], "coefficienti", []);

        contatore = 0;

        %   Ciclo "i" per incrementare di 1 la colonna i-esima
        %   (corrisponde a moltiplicare un polinomio per "vi")

        for i = 1:n

            %   Ciclo "j" che scorre tutti gli elementi di f calcolati
            %   al passo precedente

            for j = dim_vecchia + 1 : dim_vecchia + dim_nuova

                nuovo_polinomio = f(j);
         nuovo_polinomio.esponenti(:, i) = nuovo_polinomio.esponenti(:, i) + 1;

                old_length = old_length + 1;
                f(old_length) = nuovo_polinomio;

                %   Si tiene il conto dei nuovi polinomi
                contatore = contatore + 1;
            end

        end

        dim_vecchia = dim_vecchia + dim_nuova;
        dim_nuova = contatore;

    end
```

A questo punto del programma bisogna scorrere i polinomi calcolati all'iterazione attuale (quando $d = 0$ si scorrono tutti i polinomi del sistema iniziale) e inserire i coefficienti dei loro monomi nel vettore *elementi*. Il k-esimo valore di *elementi* sarà inserito, tramite la funzione sparse, nella matrice sparsa G alla riga indicata dal k-esimo valore di *indici_riga* e alla colonna indicata dal k-esimo valore di *indici_colonna*. L'indice di riga si trova con *posizione_monomio*:

```
%   Si scorrono i polinomi di f calcolati all'iterazione attuale
for i = dim_vecchia + 1 : dim_vecchia + dim_nuova

    %   Si sta analizzando il polinomio i-esimo, quindi relativo
    %   alla colonna i-esima

    esponenti = f(i).esponenti;
    coefficienti = f(i).coefficienti;

    %   Ogni riga j-esima di "esponenti" rappresenta un monomio
    for j = 1:size(esponenti, 1)
        indice_riga = posizione_monomio(esponenti(j,:), binom_table);

        indici_riga(end + 1) = indice_riga;
        indici_colonna(end + 1) = i;
        elementi(end + 1) = coefficienti(j);

    end

end

G = sparse(indici_riga, indici_colonna, elementi, max(indici_riga), length(f));
b = sparse(1, 1, 1, size(G,1), 1);

tolleranza = 1e-14;

%   Metodo iterativo
%   Il metodo è "Least squares" e "relres" è la norma del residuo
[~, ~, relres] = lsqr(G, b, tolleranza, 1000);

if(relres < tolleranza)

    %   Il grafo non è k-colorabile.
    %   Si ritorna il grado del certificato
    cert = d;
    return;

end

    %   Se il sistema lineare non ha soluzione si incrementa il grado
    %   dei polinomi "p" con la prossima iterazione del primo for
end

%   A questo punto della funzione non sono stati trovati polinomi "p"
%   tali per cui la combinazione lineare f*p facesse 1.
%
%   Significa che non si è riusciti a dimostrare che il sistema
%   polinomiale non ha soluzione, quindi il grafo potrebbe essere
%   k-colorabile.

cert = "END";
return;
end;
```

Con questo si conclude l'analisi della funzione principale *graph_coloring*. Riprendendo l'albero mostrato alla **sezione 5.1** resta da parlare di tre funzioni:

- `k_col(A, k)`

Funzione molto semplice che tenta di colorare il grafo con k colori. Invece, quindi, di cercare una stima dal basso ricorsivamente come la funzione *inf_coloring* cerca un singolo certificato relativo alla k-colorabilità e se non lo trova conclude che il grafo è k-colorabile:

```
function ris = k_col(A, k)

    f = poly_sys(A, k);
    cert = NulLA(A, f);

    if ~isa(cert, "string")
        ris = sprintf("Il grafo non è %i-colorabile.\nGrado del certificato:
%i", k, cert);

    else

        ris = sprintf("Il grafo è %i-colorabile", k);

    end

end
```

- `tre_col(A)`

Tetsa la 3-colorabilità di un grafo tramite il NulLA su campi finiti (**sezione 3.1.3**). La funzione è quasi identica a *k_col*, con la differenza che k non è un parametro d'ingresso ma viene impostato a 3 e al posto della funzione *NulLA* si richiama *NulLA_tre_col* che è identica se non per la parte finale dove si implementa il calcolo sul campo $\mathbb{F}_2$.

```
function ris = tre_col(A)
    k = 3;
    f = poly_sys(A, k);
    cert = NulLA_tre_col(A, f);

    if ~isa(cert, "string")

        ris = sprintf("Il grafo non è %i-colorabile.\nGrado del certificato:
%i", k, cert);

    else
        ris = sprintf("Il grafo è %i-colorabile", k);
    end
end
```

- `NulLA_tre_col(A, f)`

È quasi identica alla funzione *NulLA* con l'unica differenza che per implementare il calcolo su $\mathbb{F}_2$ non si richiama la funzione *lsqr* per risolvere il sistema lineare ma bensì la funzione *gauss_mod2* per applicare l'algoritmo di gauss in modulo 2 su G e b. Successivamente si verifica l'esistenza della soluzione controllando che non siano presenti righe della forma $0 = 1$ nel sistema lineare ottenuto. Si riporta la parte del codice che varia rispetto alla funzione *NulLA*:

```
G = sparse(indici_riga, indici_colonna, elementi, max(indici_riga), length(f));

b = sparse(1, 1, 1, size(G,1), 1);

[~, b_mod2, ind] = gauss_mod2(G, b);

%   Se non ci sono righe 0 = 1 il sistema lineare ha soluzione
if ~any(b_mod2(ind:end))

    %   Il grafo non è k-colorabile.
    %   Si ritorna il grado del certificato
    cert = d;
    return;

end
```

Di seguito la funzione *gauss_mod2*:

```
%   Algoritmo di Gauss con pivoting per matrici rettangolari MxN con M > N
%   in aritmetica modulo 2

function [U, c, ind] = gauss_mod2(A, b)

    A = logical(mod(A, 2));
    b = logical(mod(b, 2));

    [m, n] = size(A);

    %   Matrice aumentata
    Ab = [A, b];

    %   Si scorrono le righe
    for i = 1:min(m,n)
        %   Se l'elemento sulla diagonale è 0 si fa pivoting
        if(Ab(i, i) == 0)
            [irow, icol] = find(Ab(i:end, i:end-1), 1);
            irow = irow + i - 1;
            icol = icol + i - 1;

            if isempty(irow)
                ind = i;
                U = Ab(:, 1:n);
                c = Ab(:, end);
                return;

            %   Pivoting
```

```
            else
                if(irow ~= i)
                    Ab([i, irow], :) = Ab([irow, i], :);
                end

                if(icol ~= i)
                    Ab(:, [i, icol]) = Ab(:, [icol, i]);
                end

            end

        end

        %   Si trovano le righe da azzerare sotto il pivot


        %   Maschera delle righe sotto il pivot con elemento non nullo
        mask = Ab(i+1:end, i) ~= 0;
        rows_to_update = find(mask) + i;

        if ~isempty(rows_to_update)

            %   Si scorre la lista delle righe con 1 sulla colonna i
            for k = 1:length(rows_to_update)

                row_idx = rows_to_update(k);

                Ab(row_idx, :) = xor(Ab(row_idx, :), Ab(i, :));

            end

        end


    end

    U = Ab(:, 1:n);
    c = Ab(:, end);
    ind = n + 1;
end
```

5.3. Scalabilità e risultati sperimentali

La complessità del calcolo della stima di un numero cromatico cresce **esponenzialmente** con l'aumentare delle dimensioni del grafo.
Si possono calcolare esplicitamente le dimensioni del sistema da risolvere in funzione di $|V|$, $|E|$ e del grado d del certificato cercato.
Dati n vertici ed m archi si definisce il numero iniziale di polinomi $z = n + m$.
Il grado massimo dei monomi che appaiono nel sistema è k, cioè il numero di colori che si sta verificando (questa osservazione servirà per il calcolo del numero di righe).

Si comincia con l'analisi del numero di colonne in funzione del grado del certificato cercato. Si avrà un numero di colonne pari al numero di polinomi del passo corrente.
Si comincia con gli z polinomi iniziali quando $d = 0$ e ogni volta che si aumenta il grado si aggiungono $z \cdot n$ polinomi. Per il calcolo del certificato di grado d, il numero di colonne del sistema lineare sarà quindi:

$$\sum_{i=0}^{d} (z \cdot n^i).$$

La matrice avrà poi una riga per ogni elemento della base dei monomi. Quest'ultima avrà una cardinalità dipendente dal numero n di variabili e dal grado massimo dei monomi presenti nel sistema polinomiale.
Quando si cerca il primo certificato ($d = 0$) il numero di righe della matrice, uguale al numero di elementi nella base dei monomi, si calcola come numero di monomi di grado minore o uguale a k in n variabili (visto nella **sezione 5.2.2**):

$$\binom{n + k}{k}.$$

Per ognuno dei passi successivi il grado massimo dei monomi viene incrementato di 1, quindi durante il calcolo del certificato di grado d il numero di righe del sistema lineare sarà pari a:

$$\binom{n + (k + d)}{(k + d)}.$$

Di seguito sono riportate delle tabelle che contengono dati sperimentali sulle dimensioni dei sistemi da risolvere (per grafi generati casualmente, sempre con la funzione *creaMat(n)* descritta nella **sezione 2.1)** in funzione del numero di vertici, numero di archi e grado del certificato cercato. Si può notare come questi valori rispettino le formule appena descritte.

Numero di vertici $\|V\|$	Numero di archi $\|E\|$
10	19

Grado del certificato (3 colori)	Dimensioni sistema lineare (righe × colonne)
0	286×29
1	1001×319
2	3003×3219
3	8008×32219
4	19448×322219

Numero di vertici $\|V\|$	Numero di archi $\|E\|$
12	18

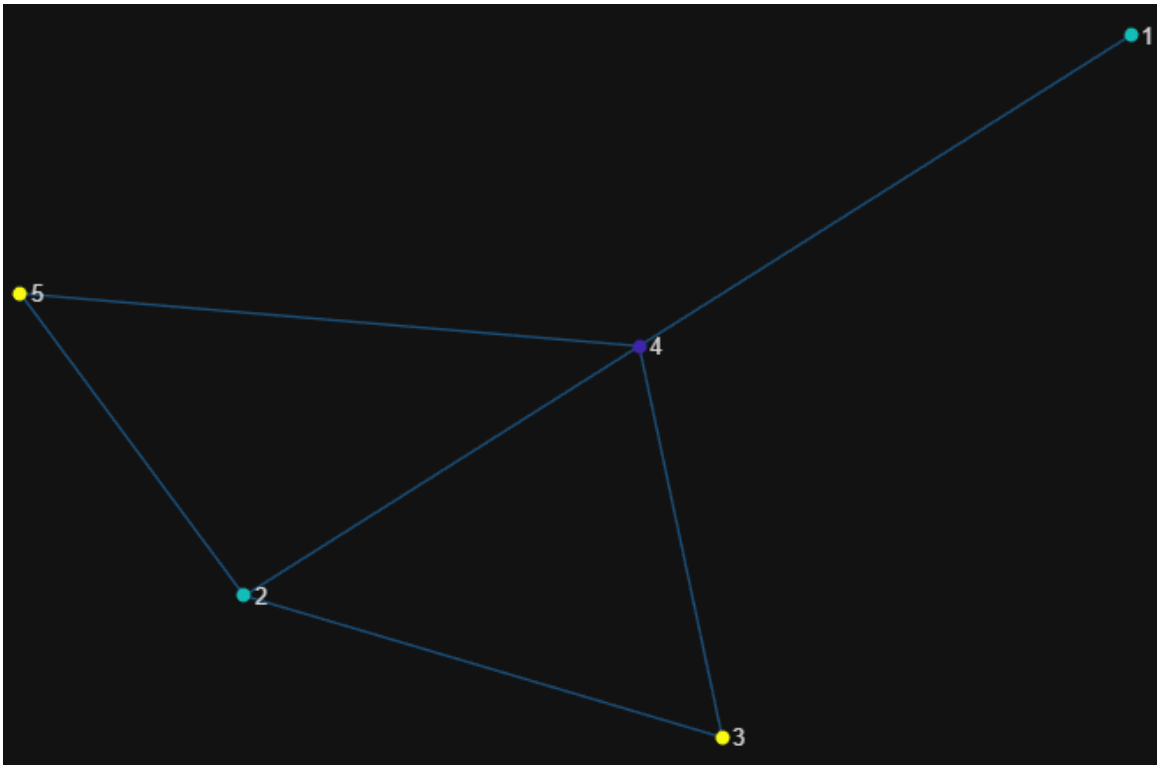
Grado del certificato (3 colori)	Dimensioni sistema lineare (righe × colonne)
0	455×30
1	1820×390
2	6188×4710
3	18564×56550
4	50388×678630

Numero di vertici $\|V\|$	Numero di archi $\|E\|$
15	27

Grado del certificato (3 colori)	Dimensioni sistema lineare (righe × colonne)
0	816×42
1	3876×672
2	15504×10122
3	54264×151872
4	170544×2278122

Si vogliono ora analizzare le prestazioni del programma relativamente al calcolo del numero cromatico di grafi dati.

Siano dati come esempio questo grafo di cinque vertici generato casualmente e la sua matrice di adiacenza (la stima dall'alto calcolata in questo caso è uguale a 3):



0	0	0	1	0
0	0	1	1	1
0	1	0	1	0
1	1	1	0	1
0	1	0	1	0

Il suo numero cromatico, calcolato con *graph_coloring*, risulta essere proprio 3.

Provando, tramite la funzione *k_col*, a dimostrare che 3 colori non bastano, ci si aspetta che la funzione cerchi un certificato fino al grado massimo senza successo (dal momento che il grafo è 3-colorabile). In questo modo possiamo osservare per ogni grado del certificato come variano le dimensioni del sistema f e della matrice G:

Grado certificato (3 colori)	Dimensione sistema *f* (numero di polinomi)	Dimensioni matrice G (righe × colonne)
0	11	56×11
1	66	126×66
2	341	252×341
3	1716	462×1716
4	8591	792×8591

Anche per un grafo molto piccolo il problema assume grandi dimensioni. Queste dipendono in generale da tre fattori principali:

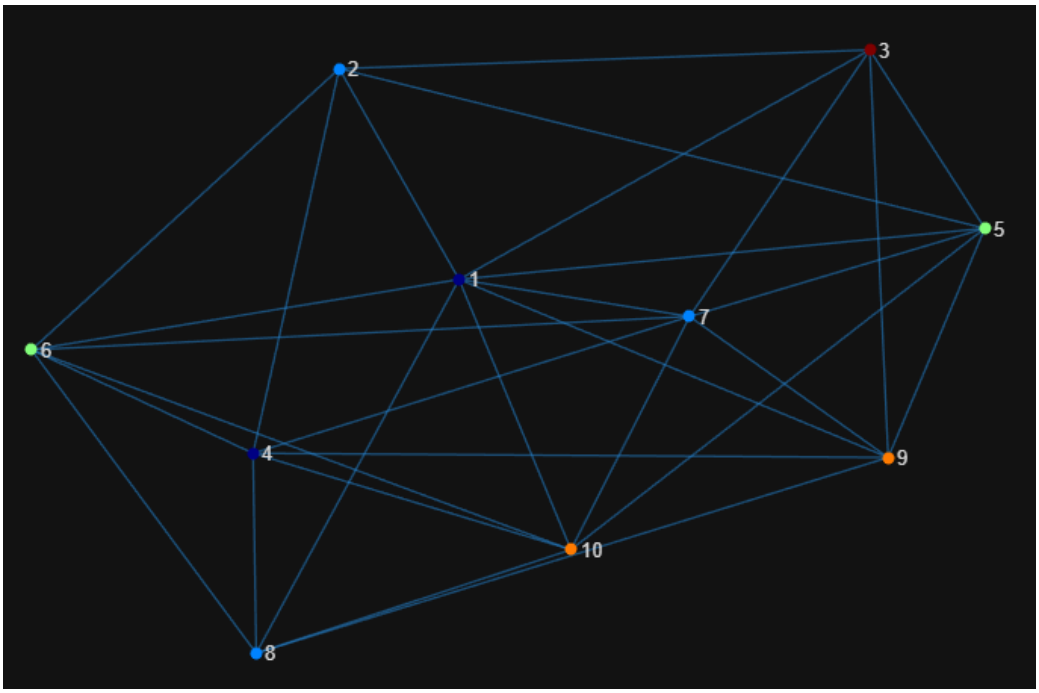
- Numero di vertici
- Numero di archi (assieme al numero di vertici determina la dimensione di *f*)
- Grado del certificato (determina il numero di righe di G)

I tempi di esecuzione sono i seguenti:

Funzione	Tempo d'esecuzione (secondi)
`welsh_powell`	`0.001023`
`brelaz`	`0.006132`
`random_ord`	`0.002009`
`graph_coloring`	`0.028572`
`k_col`	`0.469922`

Per quanto riguarda le funzioni per la stima dall'alto, i loro tempi sono sempre comparabili a quelli ottenuti in questo esempio, indipendentemente dalle dimensioni del grafo trattato.
Invece, come si può evincere dalle dimensioni che assumono *f* e G, *graph_coloring* e *k_col* rallentano esponenzialmente all'aumentare del numero di vertici.
Anche per questo semplice esempio *k_col* è 10 volte più lenta di *graph_coloring* dal momento che deve calcolare certificati fino a grado massimo (mentre graph_coloring, quando prova a colorare il grafo con 2 colori, si ferma al certificato di grado 1 per dimostrare che il sistema della 2-colorabilità non ha soluzione).
Si riportano ora le stesse informazioni per due grafi casuali di 10 e 20 vertici rispettivamente.

Grafo di 10 vertici con stima dall'alto = 5



0	1	1	0	1	1	1	1	1	1
1	0	1	1	1	1	0	0	0	0
1	1	0	0	1	0	1	0	1	0
0	1	0	0	0	1	1	1	1	1
1	1	1	0	0	0	1	0	1	1
1	1	0	1	0	0	1	1	0	1
1	0	1	1	1	1	0	0	1	1
1	0	0	1	0	1	0	0	1	1
1	0	1	1	1	0	1	1	0	0
1	0	0	1	1	1	1	1	0	0

Grado certificato (4 colori)	Dimensione sistema *f* (numero di polinomi)	Dimensioni matrice G (righe × colonne)
0	40	1001×40
1	440	3003×440
2	4440	8008×4440
3	44440	19448×44440
4	444440	43758×444440

La funzione non riesce a dimostrare che 4 colori non bastano quindi *inf_coloring* richiama sé stessa ricorsivamente per provare con 3 colori, ottenendo:

Grado certificato (3 colori)	Dimensione sistema f (numero di polinomi)	Dimensioni matrice G (righe × colonne)
0	40	286×40
1	440	1001×440
2	4440	3003×4440
3	44440	8008×44440
4	444440	19448×444440

Viene trovato un certificato di grado 4 dimostrando che 3 colori non bastano, quindi la stima dal basso vale 4. Di conseguenza si conclude che il numero cromatico è compreso tra 4 e 5.

Funzione	Tempo d'esecuzione (secondi)
welsh_powell	0.002146
brelaz	0.007692
random_ord	0.002695
graph_coloring	40.909666
k_col	18.615912
tre_col	0.242391

Si nota, con questo grafo, una proprietà interessante.

Mentre *k_col* richiede un tempo d'esecuzione comparabile a quello di *graph_coloring* (ci mette meno secondi perché a *k_col* è stato passato il parametro $k = 3$, quindi si salta la prova della 4-colorabilità), la funzione *tre_col* impiega un tempo estremamente minore a concludere che il grafo non è 3-colorabile.

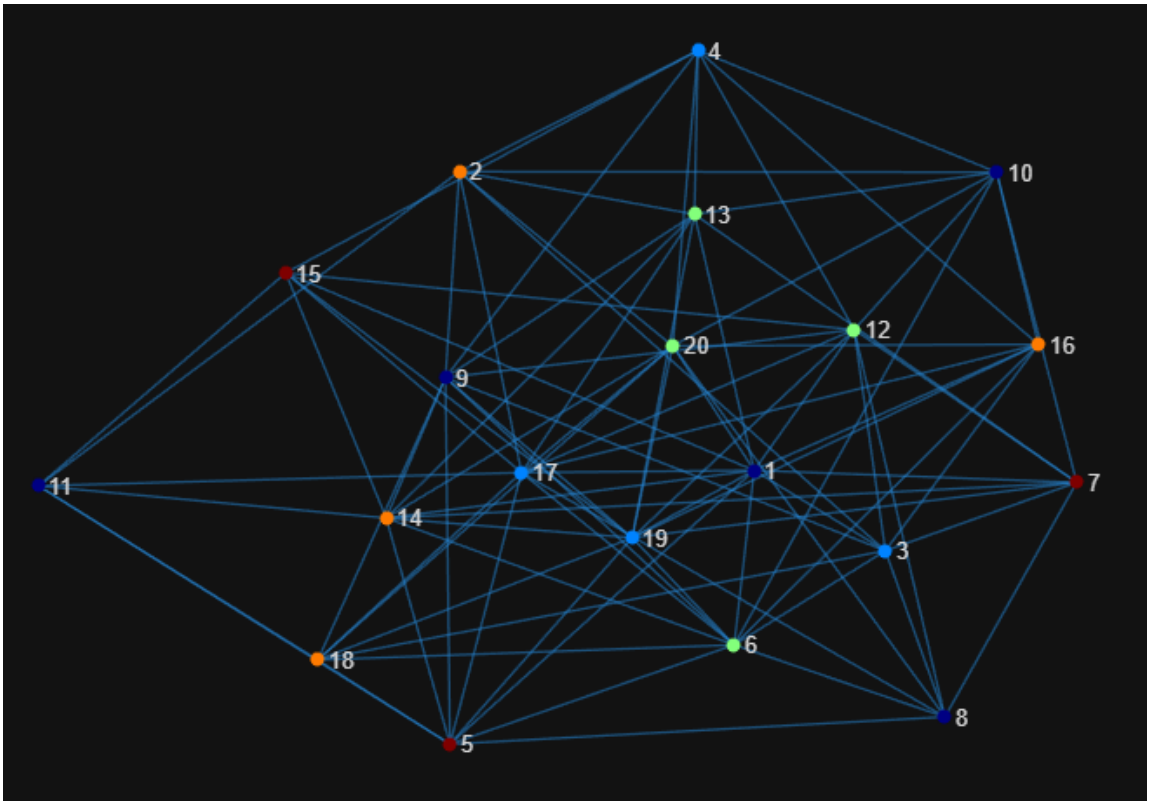
Viene infatti alla luce, con questo e molti altri esempi, il vantaggio dell'algoritmo NulLA su campi finiti (**sezione 3.1.3**) il quale richiede un certificato di **grado 1** piuttosto che 4 per attestare la non 3-colorabilità del grafo. Dovendo quindi risolvere un sistema di dimensioni molto inferiori impiega meno.

Quando si hanno grafi per cui va testata la 3-colorabilità, se il grafo è tale per cui *tre_col* necessita di un certificato di grado 1 (assicurato solo per i grafi Odd-Wheel) allora può convenire utilizzarla al posto di *k_col* a cui si passa $k = 3$.

Va però considerata una cosa: si risparmia tempo dal momento che si calcola un certificato di grado minore ma bisogna richiamare l'algoritmo di Gauss sulla matrice G, che per matrici estremamente grandi e sparse può essere molto dispendioso.

Per ogni grafo da 3-colorare bisogna quindi valutare il *tradeoff* di tempo guadagnato per il grado minore del certificato e tempo speso per eseguire *gauss_mod2*.

Grafo di 20 vertici con stima dall'alto = 5



0	0	1	0	1	1	1	0	0	0	0	1	1	1	1	1	1	0	1	1
0	0	1	1	0	0	0	0	1	1	1	0	1	0	0	0	1	0	0	1
1	1	0	0	0	1	1	1	1	0	0	1	0	0	0	1	0	1	0	0
0	1	0	0	0	0	0	0	1	1	0	1	1	0	1	1	0	0	0	1
1	0	0	0	0	1	0	1	1	0	1	0	0	1	0	0	1	1	1	0
1	0	1	0	1	0	0	1	1	1	0	0	0	1	0	1	1	1	1	0
1	0	1	0	0	0	0	1	0	1	0	1	1	1	0	0	0	0	1	0
0	0	1	0	1	1	1	0	0	0	0	1	0	0	0	0	1	0	0	1
0	1	1	1	1	1	0	0	0	0	0	1	1	1	0	0	0	1	1	0
0	1	0	1	0	1	1	0	0	0	0	1	1	0	0	1	0	0	0	1
0	1	0	0	1	0	0	0	0	0	0	0	0	1	1	0	1	1	0	0
1	0	1	1	0	0	1	1	1	1	0	0	0	0	1	0	1	0	1	0
1	1	0	1	0	0	1	0	1	1	0	0	0	1	0	0	1	0	1	0
1	0	0	0	1	1	1	0	1	0	1	0	1	0	1	0	0	0	1	1
1	0	0	1	0	0	0	0	0	0	1	1	0	1	0	0	1	0	1	0
1	0	1	1	0	1	0	0	0	1	0	0	0	0	0	0	1	0	1	1
1	1	0	0	1	1	0	1	0	0	1	1	1	0	1	1	0	1	0	1
0	0	1	0	1	1	0	0	1	0	1	0	0	0	0	0	1	0	1	1
1	0	0	0	1	1	1	0	1	0	0	1	1	1	1	1	0	1	0	1
1	1	0	1	0	0	0	1	0	1	0	0	0	1	0	1	1	1	1	0

Grado certificato (4 colori)	Dimensione sistema *f* (numero di polinomi)	Dimensioni matrice G (righe × colonne)
0	111	10626×111
1	2331	53130×2331
2	46731	230230×46731
3	934731	888030×934731
4	18694731	3108105×18694731

Anche in questo caso non è stata trovata soluzione a nessuno di questi sistemi lineari, quindi si prova con 3 colori:

Grado certificato (3 colori)	Dimensione sistema *f* (numero di polinomi)	Dimensioni matrice G (righe × colonne)
0	111	1771×111
1	2331	10626×2331
2	46731	53130×46731
3	934731	230230×934731
4	18694731	888030×18694731

Serve un certificato di grado 4 per dimostrare che 3 colori non bastano, trovando un numero cromatico compreso da 4 e 5.

Sia in questo esempio che nel precedente si può notare come la riduzione di un colore (da 4 a 3) provochi uno sfasamento nella crescita del numero di righe della matrice G permettendo di ottenere quindi sistemi più piccoli.

Funzione	Tempo d'esecuzione (secondi)
`welsh_powell`	`0.038116`
`brelaz`	`0.067042`
`random_ord`	`0.013705`
`graph_coloring`	`2007.090361`
`k_col`	`1133.824965`
`tre_col`	`2.291621`

Come prima, *k_col* richiede un tempo comparabile a *graph_coloring* (richiamano la stessa funzione *NulLA*) ma salta il test di 4-colorabilità.

Con quest'ultimo esempio risulta estremamente evidente quanto il problema cresca rapidamente e quanto la funzione *tre_col* sia utile per valutare la 3-colorabilità (quando è effettivamente sufficiente un certificato di grado 1 in $\mathbb{F}_2$ e quando la funzione *gauss_mod2* non fa da collo di bottiglia), richiedendo un tempo ben mille volte inferiore all'algoritmo base.

6. Note conclusive

In quest'ultima sezione vengono affrontati due spunti conclusivi relativi ad eventuali migliorie applicabili al programma e ad applicazioni pratiche della stima del numero cromatico.

6.1. Possibili miglioramenti

L'ambiente MATLAB si è rivelato adatto per quanto riguarda la stesura e la ricerca della soluzione del problema. Offre infatti numerose funzioni native e Add-On aggiuntivi per trattare facilmente tutte le operazioni matematiche complesse (risoluzione di sistemi, equazioni, eccetera).

Questa non si rivela tuttavia la scelta migliore relativamente alla velocità d'esecuzione. Per quanto sia ottimizzato, quello di MATLAB resta un linguaggio interpretato e come si può vedere dai dati raccolti nella **sezione 5.3** il programma rallenta molto rapidamente all'aumentare della dimensione del grafo. Probabilmente la scrittura dell'intero codice in un linguaggio compilato come C++ avrebbe concesso prestazioni migliori.
Non bisogna tuttavia sottovalutare la crescita esponenziale della complessità del problema e gli inevitabili rallentamenti che ne conseguono. Non a caso la stesura di algoritmi efficienti per la ricerca del numero cromatico è tutt'oggi oggetto di ricerca.

Analizzando ora il lavoro svolto in MATLAB, esiste un'ottimizzazione possibile per quanto concerne le funzioni *poly_sys* e *NulLA*.
Nel progetto è stata scelta come struttura dati per rappresentare i polinomi un vettore di *struct* con campi *esponenti* e *coefficienti*.
Il primo è una matrice che può essere definita *sparse* vista la frequente prevalenza di elementi a 0 mentre del secondo si può fare direttamente a meno dal momento che i coefficienti che appaiono sono sempre noti. Infatti, nell'algoritmo di base appaiono solamente valori {-1, 0, 1} mentre nell'algoritmo NulLA su campi finiti appaiono solo i valori {0, 1}.

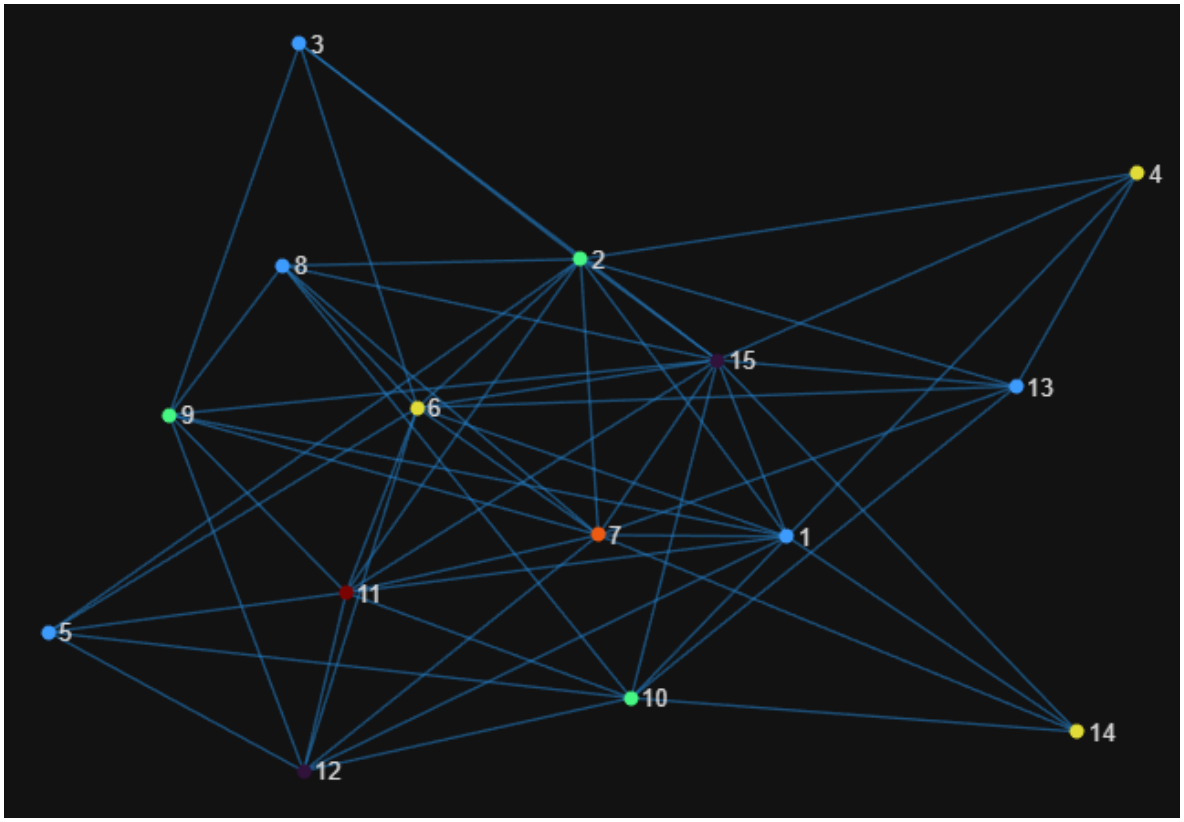
È stata scritta una diversa versione di *poly_sys* che usa come struttura dati un vettore di *cell*. Su MATLAB le *cell* possono contenere dati di qualsiasi tipo, comprese matrici. Quindi il sistema polinomiale è rappresentato da un semplice vettore di matrici *sparse* che contengono gli esponenti dei monomi che compongono ogni polinomio. La funzione NulLA è stata poi adattata a trattare questo diverso tipo di struttura dati.

Il problema è che, nonostante si risparmi effettivamente memoria, i tempi di esecuzione peggiorano dal momento che accedere e modificare elementi di matrici *sparse* è un'operazione molto lenta. Questo viene fatto ogniqualvolta si accede alle varie matrici con gli esponenti.

Optando per una soluzione ibrida, dove la funzione *poly_sys* lavora col vettore di *cell* ma senza rendere *sparse* le matrici di esponenti, i tempi di esecuzione si riducono così come l'utilizzo di memoria.

Questo non comporta variazioni significative nell'algoritmo NulLA su campi finiti (cioè, sull'esecuzione della funzione *tre_col*) ed è invece più apprezzabile quando si richiama la funzione *greedy_coloring* come illustrato dai seguenti dati.

A partire da questo grafo di 15 nodi con numero cromatico compreso tra 4 e 6:




```
0   1   0   1   0   1   1   0   1   1   1   1   0   1   1
1   0   1   1   1   1   1   1   0   0   1   0   1   0   1
0   1   0   0   0   1   0   0   1   0   0   0   0   0   1
1   1   0   0   0   0   0   0   0   0   0   0   1   0   1
0   1   0   0   0   1   0   0   0   1   1   1   0   0   0
1   1   1   0   1   0   1   1   0   0   1   1   1   0   1
1   1   0   0   0   1   0   1   1   0   1   1   1   1   1
0   1   0   0   0   1   1   0   1   1   0   0   0   0   1
1   0   1   0   0   0   1   1   0   0   1   1   0   0   1
1   0   0   0   1   0   0   1   0   0   1   1   1   1   1
1   1   0   0   1   1   1   0   1   1   0   1   0   0   1
1   0   0   0   1   1   1   0   1   1   1   0   0   0   0
0   1   0   1   0   1   1   0   0   1   0   0   0   0   1
1   0   0   0   0   0   1   0   0   1   0   0   0   0   1
1   1   1   1   0   1   1   1   1   1   1   0   1   1   0
```

La funzione *graph_coloring* non riesce a dimostrare che 5 o 4 colori non bastano. Dimostra invece che 3 colori non sono sufficienti con un certificato di grado 4.
Il sistema più grande da risolvere è quello per il certificato di grado 4 della 5-colorabilità.
Di seguito sono riportate le sue dimensioni assieme al tempo totale di esecuzione con l'implementazione trattata nella **sezione 5.2.2**:

Max dimensione *f*	Numero di byte	Tempo totale
3851111	3076549573	450.063710

Applicando ora le modifiche appena discusse si ottengono i seguenti risultati:

Max dimensione *f*	Numero di byte	Tempo totale
3851111	2479898520	320.653989

Questo miglioramento viene misurato sia per piccoli che per grandi grafi, risultando più evidente al crescere del numero di vertici.

Precedentemente è stata mostrata la versione più lenta per una maggiore chiarezza di codice e semplicità di spiegazione.

6.2. Applicazioni pratiche

Il calcolo del numero cromatico, per quanto possa sembrare un problema fine a sé stesso, ha in realtà molte applicazioni reali. Di seguito se ne elencano alcune.

6.2.1. Colorazione mappe

Questo è il problema introdotto nella **sezione 1.2** e costituisce la motivazione storica dello studio della colorazione dei grafi e ha contribuito allo sviluppo della ricerca sulla teoria di questi ultimi.
Data una cartina politica si vuole assegnare un colore distinto a regioni o stati adiacenti per una maggiore chiarezza, problema risolto appunto dal **Teorema dei quattro colori**.
Esso recita che per una qualsiasi superficie piana divisa in regioni connesse, 4 colori sono sufficienti a colorare ogni regione in modo che regioni adiacenti abbiano colori diversi.

6.2.2. Schedulazione

Supponendo di avere compiti che vanno assegnati ad intervalli temporali diversi, spesso si presenta il vincolo di non poter assegnare compiti diversi ad uno stesso intervallo perché, ad esempio, questi richiedono una risorsa comune.
Si può modellare questo problema con un grafo, definendo un vertice per ogni compito e archi che collegano compiti in conflitto tra loro.
Il numero cromatico del grafo così definito rappresenta il "tempo di completamento" minimo, cioè il tempo ottimale per finire tutti i compiti senza conflitti.
Si può applicare per schedulare processi, organizzare voli aerei, gestire l'allocazione di ampiezze di banda alle stazioni radio e molto altro.

6.2.3. Allocazione risorse limitate

Anche i problemi di allocazione di risorse si possono modellare con i grafi.
Ad esempio, una tecnica di ottimizzazione dei compilatori prevede di assegnare le variabili più usate ai registri veloci del processore. Idealmente, ciascuna di queste variabili vede assegnatosi un registro. Se si definisce un grafo con un vertice per variabile e con un arco tra variabili usate nello stesso momento, il suo numero cromatico k indica che servono k registri per immagazzinare tutti i possibili insiemi di variabili richieste nello stesso momento.

6.2.4. Partizionamento di dati

Molti algoritmi di *clustering* modellano il problema con un grafo da colorare.

Ad esempio, si supponga di avere un insieme di operai da raggruppare in squadre. Alcuni operai non possono essere messi nella stessa squadra di altri per un qualche motivo di conflitto. Similmente alle applicazioni precedenti, si definisce un grafo con un vertice per operaio e un arco tra ogni coppia di operai che non possono essere messi in squadra insieme. A questo punto ogni colore distinto rappresenta una nuova squadra. Il numero cromatico del grafo rappresenta il numero minimo di squadre.

7. Ringraziamenti

Per la realizzazione di questa tesi si ringrazia in primo luogo il relatore Prof. Stefano Massei per la grande disponibilità e la pazienza dimostrate.

In secondo luogo, ringrazio i miei genitori per il supporto fornitomi in questi anni di studio.

Ci tengo infine a ringraziare la mia fidanzata e i miei amici stretti, i quali mi hanno dato continuamente manforte e hanno contribuito ad alleggerire il mio intero percorso universitario.

8. Bibliografia

De Loera, J. A., Hemmecke, R., & Köppe, M. (2013). *Algebraic and Geometric Ideas in the Theory of Discrete Optimization.* Society for Industrial and Applied Mathematics.

De Loera, J. A., Lee, J., Malkin, P. N., & Margulies, S. (2008). Hilbert's Nullstellensatz and an Algorithm for Proving Combinatorial Infeasibility. 1-10.

Colorazione dei grafi. (2025, Marzo 23). Tratto da Wikipedia: https://it.wikipedia.org/wiki/Colorazione_dei_grafi

Teorema dei quattro colori. (2024, Novembre 22). Tratto da Wikipedia: https://it.wikipedia.org/wiki/Teorema_dei_quattro_colori