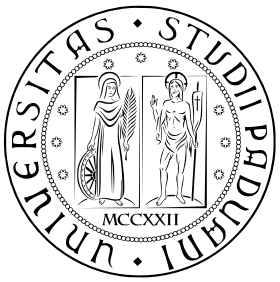


Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

Corso di Laurea Triennale in Ingegneria Informatica

Esperimenti di programmazione lineare intera per problemi di vertex cover

Relatore

Prof. Domenico Salvagnin

Laureando

Giacomo Camposampiero

Anno Accademico 2020/2021

Padova, 22 luglio 2021

Indice

Capitolo 1

Introduzione

La ricerca operativa, dall'inglese *operational research*, è una disciplina scientifica relativamente giovane, nata con lo scopo di fornire strumenti matematici di supporto ad attività decisionali in cui occorre gestire e coordinare attività e risorse limitate. La ricerca operativa permette infatti di trovare, mediante la formalizzazione matematica di un problema, una soluzione ottima o ammissibile (quando possibile) al problema stesso. Costituisce di fatto un approccio scientifico alla risoluzione di problemi complessi, che ha trovato grande applicazione in moltissimi ambiti, non ultimo quello industriale.

Una delle diverse branche che compongono la ricerca operativa è l'ottimizzazione. Quest'ultima si occupa principalmente di problemi che comportano la minimizzazione o la massimizzazione di una funzione, detta funzione obbiettivo, sottoposta ad un dato insieme di vincoli. Un problema di ottimizzazione può essere formulato come

$$\begin{aligned} &\min(or\ \max) f(x) \\ &S \\ &x \in D \end{aligned} \tag{1.1}$$

dove $f(x)$ è una funzione a valori reali nelle variabili x, D è il dominio di x e S un insieme finito di vincoli. In generale, x è una tupla $(x_1, ..., x_n)$ e D è un prodotto cartesiano $D_1 \times ... \times D_n$, e vale $x_j \in D_j$.

Un problema nella forma (1.1) è intrattabile, ovvero non esistono algoritmi efficienti (o non esiste proprio alcun algoritmo) per la sua risoluzione. Si rende quindi necessario considerare dei casi particolari di questa formulazione, i quali possiedono determinate proprietà che possono essere sfruttate nella definizione di algoritmi ad-hoc.

Nelle successive sezioni di questa introduzione verranno brevemente esposti alcuni concetti teorici rilevanti nel contesto degli esperimenti svolti. La trattazione proseguirà poi con l'esposizione dell'impostazione utilizzata nello svolgimento degli esperimenti, dei risultati ottenuti e di un conciso commento riguardo questi ultimi.

Tutto il codice sviluppato in relazione a questo elaborato è stato scritto in Python (versione 3.6.8) [1], linguaggio di programmazione *general-purpose* di alto livello, ed è liberamente consultabile online [2].

Capitolo 2

Nozioni preliminari

2.1 Programmazione lineare intera

Uno dei casi particolari della formulazione generale (1.1) a cui si è precedentemente fatto riferimento viene trattato dalla programmazione lineare intera. Un problema di programmazione lineare intera consiste nella minimizzazione (o massimizzazione) di una funzione lineare soggetta ad un numero finito di vincoli lineari, con in aggiunta il vincolo che alcune delle variabili del problema debbano assumere valori interi. In generale, il problema può quindi essere riformulato come

$$\begin{aligned} &\min cx \\ &a_i x \sim b_i \qquad i = 1, ..., m \\ &l_j \leq x_j \leq u_j \qquad j = 1, ..., n = N \\ &x_j \in \mathbb{Z} \qquad \forall j \in J \subseteq N = 1, ..., n \end{aligned}$$

Se $J = N$ si parla di programmazione lineare intera pura, altrimenti di programmazione lineare intera mista (o MIP, dall'inglese *Mixed Integer Programming*).

La programmazione lineare intera restringe quindi notevolmente il tipo di vincoli a disposizione nel processo di formalizzazione matematica del problema, determinando una maggior difficoltà nella modellazione del problema. Tuttavia, questo non comporta eccessive restrizioni, almeno per la MIP, sulle tipologie di problemi che possono essere formulati secondo questo paradigma. Alcuni esempi classici di problemi risolvibili mediante MIP sono *knapsack*, problemi di *scheduling*, *facility location* e, naturalmente, *minimum vertex cover*.

Inoltre, l'introduzione dei vincoli di linearità ed interezza comporta notevoli vantaggi nella definizione ed implementazione di algoritmi risolutivi.

2.1.1 Algoritmo Branch and Bound

L'algoritmo branch-and-bound (B&B) è un algoritmo di ottimizzazione generica basato sull'enumerazione dell'insieme delle soluzioni ammissibili di un problema di ottimizzazione combinatoria, introdotto nel 1960 da A. H. Land e A. G. Doig [3].

Questo algoritmo permette di gestire il problema dell'esplosione combinatoria mediante il *pruning* di intere porzioni dello spazio delle soluzioni, che può essere effettuato quando si riesce a dimostrare che queste ultime non contengono soluzioni migliori di quelle note. Branch-and-bound implementa inoltre una strategia *divide and conquer*, che permette di partizionare lo spazio di ricerca e di risolvere ognuna di esse separatamente. Viene riportata di seguito una breve descrizione dell'algoritmo.

Sia F l'insieme delle soluzioni ammissibili di un problema di minimizzazione (simmetrico nel caso della massimizzazione, a meno di un cambio di segno della funzione obiettivo), $c : F \to \mathbb{R}$ la funzione obiettivo e $\bar{x} \in F$ una soluzione ammissibile nota, generata mediante euristiche o mediante assegnazioni casuali. Sia $z = f(\bar{x})$ il costo di tale soluzione nota, anche detto *incumbent*, che rappresenta per sua natura un limite superiore al valore della soluzione ottima.

L'algoritmo branch-and-bound esegue un'iniziale fase di *bounding* in cui uno o più vincoli del problema vengono rilassati, allargando di conseguenza l'insieme delle possibili soluzioni $G \supseteq F$. La soluzione di questo rilassamento, se esiste, rappresenta un *lower bound* alla soluzione ottima del problema iniziale. Se la soluzione di tale rilassamento appartiene a F o ha costo uguale all'attuale *incumbent*, allora l'algoritmo termina, in quanto si è trovata una soluzione ottima del problema. Se il rilassamento risulta essere impossibile, possiamo anche in questo caso terminare la ricerca di una soluzione e concludere che anche il problema di partenza è impossibile.

Nel caso in cui invece una soluzione al rilassamento esiste ma non è contenuta nell'insieme delle soluzioni ammissibili F, l'algoritmo procede con l'identificare una separazione F^* di F, ossia un insieme finito di sottoinsiemi tali che

$$\bigcup_{F_i \in F^*} F_i = F$$

Questa fase, detta di *branching*, è giustificata dal fatto che la soluzione ottima dell'intero problema è data dalla minima tra le soluzioni delle varie separazioni $F_i \in F^*$. F^* è spesso, anche se non necessariamente, una partizione dell'insieme iniziale F. A questo punto, tutti i figli di F vengono aggiunti alla coda dei sotto-problemi da processare.

L'algoritmo procede quindi con il selezionare un sotto-problema P_i dalla coda un rilassamento. Quattro sono i possibili casi:

- Se si trova una soluzione $\in F$ migliore dell'attuale incumbent, quest'ultimo viene sostituito dalla soluzione trovata e si procede con lo studio di un altro sotto-problema.
- Se il rilassamento del sotto-problema non ammette soluzione, allora si smette di esplorare l'intero sotto-albero a lui associato nello spazio di ricerca (*pruning by infeasibility*).
- Altrimenti, si confronta la soluzione trovata con il valore corrente dell'*upper-bound* dato dall'incumbent; se quest'ultimo è minore della soluzione del rilassamento trovata, è possibile anche in questo caso smettere di esplorare il sotto-albero associato al sotto-problema corrente, in quanto non può portare ad una soluzione migliore di quella che già si conosce (*pruning by optimality*).
- Infine, se non è stato possibile scartare o concludere la visita del sotto-albero associato a P_i in alcun modo, è necessario eseguire nuovamente il *branching*, aggiungendo i nuovi sotto-problemi alla lista dei sotto-problemi da processare.

L'algoritmo prosegue nella selezione di sotto-problemi finché la lista di questi ultimi non si svuota. Quando ciò avviene, la soluzione rappresentata dall'attuale *incumbent* è la soluzione ottima al problema iniziale.

Quella appena descritta rappresenta una formulazione generica dell'algoritmo B&B. Questa formulazione può essere tuttavia specializzata nella risoluzione di problemi MIP con relativa semplicità, agendo sulle condizioni che regolano *bounding* e *branching*. Nel primo caso la scelta più diffusa consiste nel rilassamento del vincolo di interezza. Se la soluzione del rilassamento non è intera, una possibile separazione in sotto-problemi può essere fatta considerando la partizione

$$x_j \leq \left\lfloor x_j^* \right\rfloor \vee x_j \geq \left\lceil x_j^* \right\rceil$$

Per costruzione, ogni soluzione trovata dall'algoritmo è migliore dell'incumbent e, di conseguenza, l'andamento dell' upper bound del problema è strettamente decrescente. I lower bound dei singoli sotto-problemi non hanno invece, al contrario degli upper bound, valenza globale. Derivare un lower bound globale è comunque possibile, considerando il

minimo tra tutti i lower bound dei sotto-problemi ancora aperti. Avendo a disposizione in ogni momento entrambi i bound del problema, è possibile quindi valutare la bontà della soluzione provvisoria in qualsiasi momento.

2.1.2 Algoritmo Branch and Cut

L'algoritmo *branch-and-cut* (B&C) rappresenta una versione migliorata dell'algoritmo branch-and-bound, introdotta nel 1987 da M. Padberg e G. Rinaldi [4] e ideata appositamente per la risoluzione di problemi MIP.

L'algoritmo branch-and-cut è un ibrido tra branch-and-bound, trattato in precedenza, e un algoritmo a piani di taglio puro, in cui la soluzione è ottenuta mediante raffinazioni successive dello spazio delle soluzione mediante la progressiva aggiunti di vincoli. Le due tecniche si rafforzano a vicenda, contribuendo al raggiungimento di prestazioni complessive superiori a quelle che otterrebbe ciascuna delle due singolarmente.

L'idea alla base di questo algoritmo è quella di "rafforzare", per ogni sotto-problema, la formulazione associata al suo rilassamento lineare mediante la generazione di piani di taglio. I vantaggi a livello risolutivo sono diversi, tra cui una maggior probabilità di ottenere soluzioni intere al rilassamento lineare o, in alternativa, di ottenere lower bound migliori e quindi più discriminanti in fase di pruning.

Nonostante l'idea alla base di questo approccio sia relativamente semplice, l'implementazione dell'algoritmo B&C è tutt'altro che scontata e richiede l'esistenza di procedure efficienti per la risoluzione del seguente problema di *separazione*: data una soluzione frazionaria x^*, trovare una diseguaglianza valida $\alpha^T x \leq \alpha_0$ (se esiste) violata da x^*, cioè tale che α^T.

2.2 Vertex cover

Il problema di *vertex cover* (anche detto di *copertura dei vertici*) è un problema di ottimizzazione combinatoria che consiste nel trovare il minimo vertex cover di un grafo, ossia il più piccolo insieme di nodi del grafo tale che almeno uno dei due vertici di ogni arco sia contenuto in questo insieme. Trovare il vertex cover minimo di un generico grafo è un problema *NP-completo*, ovvero non esistono algoritmi in grado di trovarne una soluzione in un tempo polinomiale. In questo elaborato è stata considerata la formulazione indiretta e *unweighted* del problema, in cui gli archi non sono direzionati hanno tutti peso uguale e pari ad 1.

Formalmente, il vertex cover V' di un grafo $G = (V, E)$ può essere definito come un sotto-insieme di V tale che $uv \in E \Rightarrow u \in V' \vee v \in V'$. Definita $\tau = \mid V' \mid$ la cardinalità del vertex cover, il vertex cover minimo può essere definito come

$$V'_{min} = \arg\min_{\tau} V'$$

Capitolo 3

Approccio sperimentale

Terminata la breve parentesi teorica introduttiva, si procede in questo capitolo alla presentazione dell'impostazione pratica che si è voluto dare alle sperimentazioni condotte. La struttura secondo cui saranno esposte le informazioni nei seguenti paragrafi ricalca la partizione logica alla base del codice sviluppato, pragmaticamente diviso in quattro moduli tra loro indipendenti:

- generazione dei grafi
- generazione delle istanze di problemi MIP
- risoluzione delle istanze di problemi MIP
- estrazione ed analisi dei dati

3.1 Generazione dei grafi

L'iniziale problema che è stato necessario affrontare nello svolgimento di questo lavoro è stata la generazione automatica e pseudo-casuale di grafi, indispensabile al fine di ottenere un numero sufficiente di istanze da cui estrarre informazioni statisticamente rilevanti. Quello della generazione di grafi pseudo-casuali è un problema noto in letteratura, in quanto in numerosi ambiti di ricerca, che spaziano dalla biologia molecolare allo studio delle reti di calcolatori, si rende necessaria la generazione casuale di queste strutture dati nello studio delle realtà di loro interesse. Di conseguenza, nel corso degli ultimi decenni sono stati molti i metodi proposti dalla comunità scientifica al fine di affrontare il problema. Tra questi ne sono stati selezionati quattro, sulla base delle proprietà e delle caratteristiche particolari di ciascuno di essi.

La gestione dei grafi a livello di codice è stata implementata utilizzando NetworkX, una libreria stabile e flessibile sviluppata appositamente per lo studio e la rappresentazione di grafi in Python [5]. NetworkX offre, oltre alla possibilità di rappresentare grafi, diversi algoritmi per lo studio delle proprietà e il calcolo di determinate misure su istanze di grafi fornite. Questa libreria offre inoltre un insieme di metodi che implementano un gran numero di modelli standard noti in letteratura per la generazione di grafi, tra cui naturalmente quelli utilizzati in questo lavoro.

Inoltre, tutti i grafi generati in questo lavoro sono stati salvati sotto forma di *adjacency list*, in modo tale da poter essere letti ed elaborati da altri moduli del programma a posteriori. Il salvataggio è stato svolto mediante un apposito metodo della libreria, `write_adjlist(G, path)`. In una lista di adiacenze ogni nodo è rappresentato da una stringa nel formato

```
nodo [vicino] [vicino] ... [vicino]
```

3.1.1 Modello di Erdős–Rényi

Il modello di Erdős–Rényi è un modello per la generazione di grafi casuali, detti grafi Erdős–Rényi (*ER*) o binomiali, introdotto per la prima volta nel 1959 dai matematici ungheresi Paul Erdős e Alfréd Rényi [6]. Nella formulazione $G(n, p)$ del modello, un grafo di n nodi viene costruito secondo un procedimento iterativo, in cui ogni arco tra due nodi del grafo viene formato indipendentemente dagli altri con una probabilità fissa p.

Definizione 1. Un grafo ER, anche detto $G(n, p)$, è un grafo (N,G) con $N = 1, 2, \ldots n$ insieme dei nodi e in cui la matrice delle adiacenze G = (g_{ij}) è tale per cui, per ogni coppia di nodi distinti i e j, a probabilità che $g_{ij} = 1$ è $P(g_{ij} = 1) = p$,, con $p \in [0, 1]$ fissato.

I parametri di questo modello sono quindi due, il numero di nodi del grafo n e la probabilità di generazione di ogni arco p. A parità di parametri, i grafi che si possono ottenere sono in ogni caso molti, come mostrato in Figura 3.1.

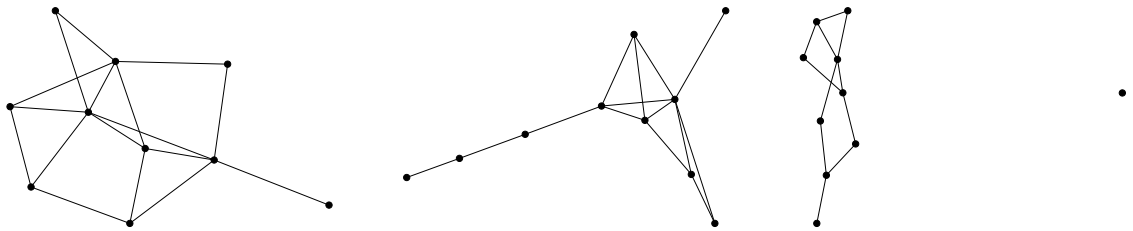


Figura 3.1: Diversi grafi $G(n, p)$ generati a partire dagli stessi parametri $n = 10$ e $p = 0.3$ con la libreria NetworkX e visualizzati graficamente con l'ausilio di un'altra libreria Python, Matplotlib [7].

Algoritmo 1: Generazione di un grafo di Erdős–Rényi

Input: numero di nodi n e probabilità di generazione di ogni arco p
inizializza il grafo vuoto G;
aggiungi i nodi $1 \dots n$ a G;
foreach arco e tra due nodi del grafo **do**
  $rand$ = numero casuale $\in [0, 1]$;
  if $rand > p$ **then**
    aggiungi e a G;
  end
end
return G

Il metodo di NetworkX utilizzato nella generazione di questa tipologia di grafi è `gnp_random_graph(n, p, seed)`, che restituisce un'istanza di grafo $G(n, p)$ di dimensione `n` in cui gli archi hanno probabilità di essere generati pari a `p`, mentre il parametro `seed` regola il comportamento pseudo-casuale, in modo tale da permettere la riproducibilità degli esperimenti. Lo pseudo-codice dell'algoritmo che gestisce la generazione di questa tipologia di grafi è riportato in Algoritmo 1.

3.1.2 Modello di Steger-Wormald

Nella teoria dei grafi, un grafo è detto *regolare di grado d* quando tutti i suoi vertici hanno lo stesso grado, ossia lo stesso numero di archi incidenti., pari a d Un grafo regolare random è un grafo selezionato da $\mathcal{G}_{n,r}$, che denota lo spazio di probabilità di tutti i possibili grafi r-regolari di n vertici, in cui $3 \leq r < n$ e nr è pari. In questo spazio ogni grafo d-regolare di n nodi ha la stessa probabilità $\frac{1}{|\mathcal{G}_{n,r}|}$ di essere generato. Si tratta quindi di grafi random, le cui caratteristiche sono tuttavia significativamente diverse dal caso generale a causa del vincolo di regolarità.

Dal momento che la maggior parte dei grafi random (che possono essere generati ad esempio mediante il modello precedentemente esposto nella Sottosezione 3.1.1) non sono regolari, l'implementazione di un algoritmo imparziale per la generazione di grafi regolari casuali non è così scontata.

Un modello tipicamente utilizzato per la generazione di grafi d-regolari con d piccolo è il cosiddetto *pairing model*. Tale modello si basa sulla definizione di n insiemi, uno per ogni vertice del grafo da generare, contenenti ciascuno d elementi, tanti quanti il numero di archi che devono incidere ogni nodo. La costruzione del grafo avviene quindi selezionando iterativamente due insiemi i e j, con $i \neq j$, aggiungendo al grafo l'arco (i, j) e rimuovendo dai corrispettivi insiemi un elemento ciascuno.

In questo lavoro la generazione dei grafi regolari random è stata implementata utilizzando una versione ottimizzata del modello ad accoppiamento, proposta nel 1999 dai matematici Angelika Steger e Nick Wormald [8]. Lo pseudo-codice dell'algoritmo utilizzato nel metodo proposto da questi ultimi è riportato in Algoritmo 2. Alcuni esempi di grafi regolari random generati mediante il modello di Steger-Wormald sono riportati in Figura 3.2.

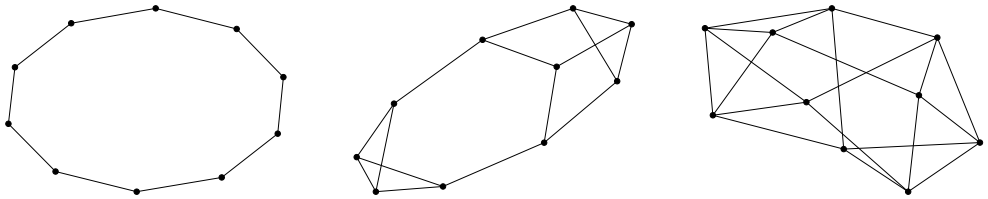


Figura 3.2: Diversi grafi regolari di dimensione 10 generati utilizzando il metodo `random_regular_graph(d, n)` della libreria NetworkX con d pari a, da sinistra verso destra, 2,3 e 4

Il metodo di NetworkX utilizzato nella generazione di questa tipologia di grafi è `random_regular_graph(d, n, seed)`, che restituisce un'istanza di grafo regolare di dimensione `n` in cui tutti i nodi hanno grado `d`. Il parametro `seed` anche in questo caso regola il comportamento pseudo-casuale della generazione, permettendo la replicabilità dell'esperimento.

Algoritmo 2: Generazione di un grafo regolare con il modello di Steger-Wormald

```
do
    Input: numero di nodi n e numero di archi incidenti per ogni nodo d
    inizializza un grafo G con n nodi e nessun vertice;
    inizializza un insieme S di coppie di nodi non adiacenti e il cui grado è al
     massimo d − 1;
    do
        assegna ad ogni coppia (u, v) in S una probabilità di essere scelta
         proporzionale a (d − d(u))(d − d(v));
        scegli una coppia (u, v) da S sulla base delle probabilità calcolate;
        aggiungi l'arco u, v a g;
    while S non è vuoto;
while G non è d-regolare;
return G
```

3.1.3 Modello di Watts-Strogatz

Il modello di Watts-Strogatz è un modello per la generazione di grafi casuali presentato nel 1998 da Duncan J. Watts e Steven Strogatz [9], la cui caratteristica principale risiede nel possedere proprietà *small-world*.

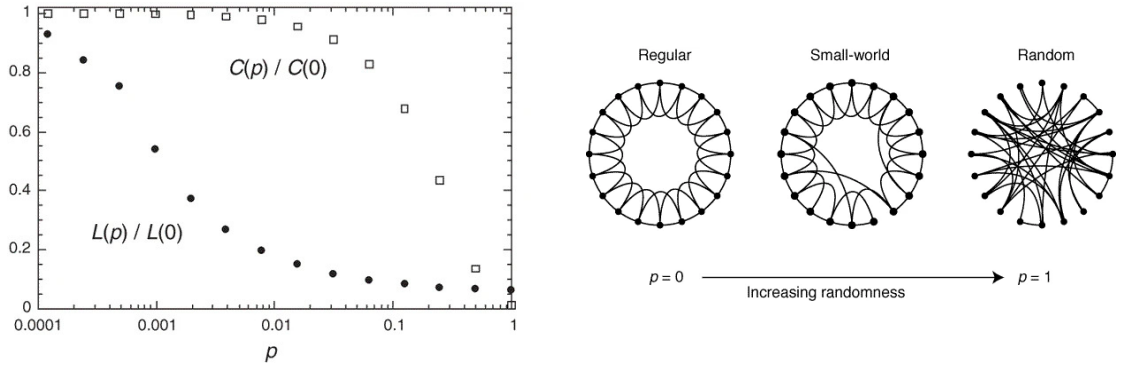
Il concetto di small-world venne introdotto per la prima volta nel 1967 dallo psicologo statunitense Stanley Milgram nella sua serie di esperimenti mirata ad esaminare la lunghezza media dei percorsi delle reti sociali tra residenti negli Stati Uniti (popolarmente conosciuti con il nome di "*six degrees of separation*"). In questi esperimenti Milgram ipotizzò l'aderenza di questi contesti sociali ad un modello a "*piccolo mondo*", composto da una rete di collegamenti tra persone relativamente breve [10].

Watts e Strogatz, partendo dalle osservazioni di Milgram, verificarono che diversi esempi di grafi che rappresentano contesti reali, tra cui il sistema nervoso del verme nematode fasmidario *Caenorhabditis elegans*, la rete elettrica degli Stati Uniti occidentali e un grafo rappresentante le collaborazioni tra diversi attori di film, mostrano alcune proprietà comuni molto importanti. Sulla base di queste osservazioni proposero quindi un modello di rete denominato *small-world network*, caratterizzato da:

- lunghezza media del cammino minimo tra due nodi molto bassa
- tendenza al raggruppamento in cluster all'interno della rete, che si traduce in regolarità e località del grafo

Nessuno dei principali metodi di generazione di grafi casuali garantiva tuttavia la generazione di grafi con tali caratteristiche. I grafi $G(n, p)$ infatti presentano caratteristiche

simili a reti small-world, ma non hanno un alto coefficiente di clusterizzazione. Al contrario, i grafi regolari presentano un alto coefficiente di clusterizzazione ma lunghezza media dei cammini minimi generalmente elevata. Per questa ragione, Watts e Strogatz proposero un nuovo modello di generazione di grafi casuali, basato su modifiche iterative di un grafo regolare ad anello mediante il *rewiring* di archi del grafo, controllato dal parametro p (che modula la regolarità o, viceversa, la casualità del grafo prodotto, come mostrato in Figura 3.3b). Queste modifiche consentono, per opportuni valori di p, la conservazione di un alto livello di clustering locale ma, allo stesso tempo, abbattono la lunghezza del cammino minimo tra nodi del grafo mediante l'aggiunta di collegamenti casuali. Un andamento di queste due proprietà al variare del valore di p è riportato in Figura 3.3a. Lo pseudo-codice di questo metodo è riportato in Algoritmo 3.



(a) Clustering globale C e lunghezza media del cammino minimo tra nodi L al variare della probabilità di rewiring p

(b) Regolarità del grafo in funzione della probabilità di rewiring p

Figura 3.3: Rappresentazioni grafiche che evidenziano caratteristiche proprie del modello di generazione proposto da Watts-Strogatz [9]

Il metodo di NetworkX utilizzato nella generazione di questa tipologia di grafi è `watts_strogatz_graph(n, k, p, seed)`, che restituisce un'istanza di grafo Watts-Strogatz di dimensione `n` i cui nodi sono inizialmente organizzati in una struttura ad anello dove ogni nodo è collegato con i suoi `k//2+1` vicini sinistri e `k//2+1` vicini destri ed in cui la probabilità che ciascun nodo sia "ricollegato" con un altro nodo diverso da uno dei suoi vicini attuali è `p`. Anche in questo caso il parametro `seed` regola il comportamento casuale della generazione, permettendo così di ottenere la riproducibilità dei risultati.

Questo modello di generazione presenta tuttavia diverse limitazioni. La principale tra queste riguarda il grado dei nodi che, a differenza della maggior parte dei grafi che rappresentano modelli reali, denota una varianza media molto bassa tra i nodi di uno stesso grafo. Inoltre, questo modello di generazione implica l'utilizzo di grafi con un numero fisso di nodi e, di conseguenza, non può essere utilizzato nella simulazione di realtà la cui dimensione è variabile nel tempo.

Algoritmo 3: Generazione di un grafo di Watts-Strogatz

Input: numero di nodi n, numero collegamenti iniziali con i vicini k e probabilità di fare *rewiring* di ciascun arco p

inizializza un grafo ad anello G con n nodi;
foreach nodo m appartenente al grafo **do**
  aggiungi un arco tra m e i suoi $k/2$ vicini destri e $k/2$ vicini sinistri;
end
foreach arco *e=(u,v)* tra due nodi del grafo **do**
  $rand$ = numero casuale $\in [0, 1]$;
  if $rand > p$ **then**
    w = nodo diverso da v scelto casualmente con probabilità uniforme;
    $e = (u, w)$;
  end
end
return G

3.1.4 Modello di Barabási-Albert

Tutti i modelli visti fino a questo momento mancano di due aspetti fondamentali che caratterizzano invece i grafi reali. Per prima cosa, viene sempre assunta la presenza di un numero fisso di vertici nel grafo. Al contrario, molte delle delle reti nel mondo reale sono composte da un insieme di vertici che solitamente tende a crescere nel tempo. Alcuni esempi di grafi che hanno questo comportamento sono il World Wide Web, che cresce esponenzialmente con l'aggiunta di nuove pagine Web, e la letteratura scientifica, che cresce anch'essa nel tempo con la pubblicazione di nuovi articoli.

In secondo luogo, nessuno dei modelli trattati in precedenza tiene conto del cosiddetto *preferential attachment*, fenomeno per cui i nodi che si aggiungono alla rete sono solitamente più propensi a creare connessioni con nodi che hanno già un gran numero di connessioni con altri nodi. Ritornando agli esempi già citati del WWW e della letteratura scientifica, è più probabile che una nuova pagina Web includa link a pagine Web più conosciute e già referenziate da molte altre pagine, così come è molto probabile che un nuovo paper scientifico includa riferimenti ad altri paper influenti e già citati in molti altri lavori.

Proprio sulla base di queste osservazioni i fisici Albert-László Barabási e Réka Albert proposero nel 1999 un nuovo modello per la generazione di grafi casuali [11]. Tale metodo è basato su di un meccanismo di *preferential attachment*, in cui i nodi vengono aggiunti uno ad uno al grafo e la probabilità di creazione di un arco tra il nuovo nodo i e uno di

quelli esistenti j dipende strettamente dal grado di quest'ultimo, secondo la relazione

$$p_i = \frac{k_i}{\sum_{j \in J} k_j}$$

dove J è l'insieme dei nodi già presenti nel grafo durante l'iterazione in cui viene aggiunto il nodo i. Lo pseudo-codice del modello di generazione è riportato in Algoritmo 4.

Il modello di Barabási-Albert ha quindi la capacità di generare un grafo con pochi nodi, detti *hub*, che presentano un grado molto elevato se comparato a quello della maggior parte dei restanti nodi del grafo. Un'altra caratteristica molto importante di questo modello è la capacità di generare *scale-free networks*, ovvero grafi per cui la relazione tra numero di nodi e numero di connessioni è di tipo esponenziale negativo. Per questa tipologia di reti la distribuzione dei nodi di diverso grado non è una classica distribuzione a campana, bensì risulta essere la stessa per qualsiasi valore di scala, come riportato in Figura 3.4.

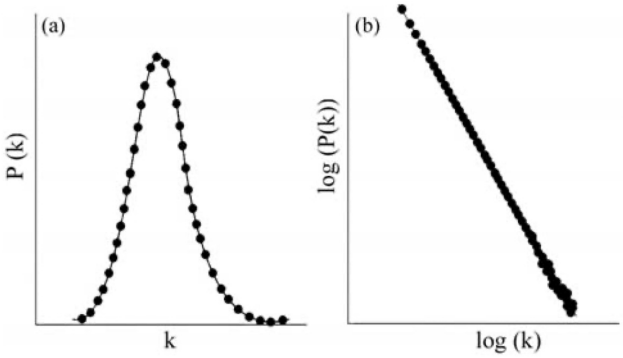


Figura 3.4: Confronto tra la distribuzione del grado dei nodi in un grafo $G(n, p)$ (a sinistra)e in un grafo scale-free (a destra) [12]

Il metodo di NetworkX utilizzato nella generazione di questa tipologia di grafi è `barabasi_albert_graph(n, m, seed`, che restituisce un'istanza di grafo Barabási-Albert di dimensione `n` in cui ogni nuovo nodo viene collegato nel procedimento iterativo a `m` nodi esistenti. Anche in questo caso il parametro `seed` regola il comportamento casuale della generazione, agendo in particolare sulla scelta dell'insieme di archi da creare ad ogni iterazione, permettendo così la replicabilità dell'esperimento.

Sebbene descriva per certi versi grafi reali, grazie alla sua natura scale-free, il modello di generazione proposto da Barabási ed Albert non riesce a replicare l'alto indice di clustering globale caratteristico di queste reti (cosa che, al contrario, viene ricreata perfettamente dal modello di Watts-Strogatz). Per questo motivo sia il modello di Watts-Strogatz che quello di Barabási-Albert devono essere considerati come parzialmente realistici.

Algoritmo 4: Generazione di un grafo di Barabási-Albert

Input: numero di nodi n e numero di archi m da creare tra ogni nuovo nodo ed i nodi esistenti

inizializza un grafo G con m nodi;
$target$ = lista contenente gli m nodi;
r = lista di nodi inizialmente vuota;
while ci sono ancora nodi da aggiungere **do**
    aggiungi un nuovo nodo al grafo;
    genera un arco tra il nuovo nodo e ogni nodo in $target$;
    aggiungi il nuovo nodo m volte e tutti i nodi di $target$ a r;
    $target$ = sottoinsieme casuale di r;
end
return G

3.2 Generazione delle istanze di problemi MIP

Terminata la generazione dei grafi, costituenti il *dataset* grezzo alla base di questo lavoro, è stato necessario tradurre ciascuno di essi nella corrispondente formulazione in forma di problema di programmazione lineare intera. Per fare questo si è inizialmente dovuta stilare una modellazione algebrica del problema stesso, definendo variabili, vincoli e funzione obiettivo con cui rappresentarlo formalmente. Solo in un secondo momento è stato quindi possibile procedere con l'implementazione del modello nel software e la costruzione delle singole istanze, mediante l'ausilio di un'apposita libreria Python.

3.2.1 Modellazione algebrica

Un *modello algebrico* di un problema di programmazione lineare intera è un modello che descrive le caratteristiche della soluzione ottima al problema mediante relazioni matematiche. Un modello algebrico è composto dai seguenti elementi:

- *insiemi*, che raggruppano gli elementi del sistema permettendo l'indicizzazione delle grandezze trattate nel problema
- *parametri*, ossia quantità costanti e definite dalla formulazione del problema stesso, che solitamente sono delle proprietà degli insiemi precedentemente definiti
- *variabili*, ossia le grandezze incognite del sistema, su cui l'algoritmo può agire (nei limiti dei vincoli imposti dal loro dominio) per ottimizzare il valore della soluzione
- *vincoli*, ossia relazioni matematiche che permettono di verificare l'ammissibilità delle soluzioni e, di conseguenza, definire quali assegnazioni alle variabili sono accettabili nel contesto del problema e quali invece non lo sono

- *funzione obiettivo*, che permette la traduzione di una soluzione del problema in un valore numerico, rendendone possibile di conseguenza l'ottimizzazione

Un modello algebrico di un problema di programmazione lineare permette quindi la definizione in forma dichiarativa delle caratteristiche della soluzione cercata. Durante la fase di modellazione formale del problema viene persa parte della potenza descrittiva caratteristica del linguaggio naturale, a vantaggio però della possibilità di utilizzare algoritmi particolarmente efficienti nella risoluzione dello stesso, che possono essere applicati solo quando il problema è definito in questa forma.

La modellazione algebrica del problema non è, nel caso del problema analizzato in questo lavoro, particolarmente onerosa. Per prima cosa sono stati identificati gli insiemi del problema, ovvero

$$V \text{ : insieme dei vertici del grafo}$$
$$E \text{ : insieme degli archi del grafo}$$

Successivamente, è stato necessario identificare i parametri del problema. Avendo studiato la formulazione *unweighted* del problema di vertex cover, in cui tutti i vertici hanno un peso associato $c_v = 1$, non è stato necessario definire alcun parametro. Infine sono state definite le variabili del problema, indicizzate dall'insieme dei vertici V

$$x_v = \begin{cases} 1 & \text{se il vertice } v \in V \text{ è compreso nel vertex cover} \\ 0 & \text{altrimenti} \end{cases}$$

In questo caso il vincolo imposto dal problema è uno solo, ossia che almeno uno dei due vertici collegati da un arco sia presente nel vertex cover, per ognuno degli archi del grafo. Questo vincolo può essere rappresentato in forma dichiarativa mediante le variabili e gli insiemi precedentemente definiti come

$$x_u + x_v \geq 1 \quad \forall (u, v) \in E, u \in V, v \in V$$

La definizione della funzione obiettivo $f(x)$, che dovrà essere minimizzata dal risolutore, viene a questo punto naturale

$$f(x) = \sum_{v \in V} x_v$$

Il modello algebrico del problema di programmazione lineare del minimo vertex cover può

quindi essere complessivamente definito come

$$\begin{aligned} &min\,(\textstyle\sum_{v\in V} x_v) \\ &x_u + x_v \geq 1 \qquad \forall (u,v) \in E, u \in V, v \in V \\ &x_v \in 0,1 \qquad \forall v \in V \end{aligned} \tag{3.1}$$

3.2.2 Modellazione nel software

Al fine di tradurre le istanze di grafo generate con NetworkX in istanze di problemi MIP nella forma (3.1) che potessero essere risolte da CPLEX è stata utilizzata un'altra libreria Python, Pyomo [13][14]. Si tratta anche in questo caso di una libreria open-source, stabile e largamente utilizzata per formulare, risolvere ed analizzare modelli per l'ottimizzazione.

Una caratteristica molto importante di questa libreria, che è stata sfruttata in questo lavoro, è la separazione che viene mantenuta tra modello e dati. In particolare, Pyomo permette la definizione di un modello astratto, chiamato `AbstractModel`, che è definito secondo una struttura molto simile al modello algebrico e in cui non è richiesto che sia specificato nessun dato del problema. La definizione di questi ultimi sarà necessaria solo nel momento in cui si andrà a creare un'istanza concreta del modello stesso, chiamata `ConcreteModel`.

Le istanze generate in questo modo sono state salvate in file `lp`, formato utilizzato da CPLEX per il salvataggio di istanze di problemi di programmazione lineare in forma algebrica.

3.3 Risoluzione delle istanze di problemi MIP

La risoluzione delle istanze di problemi di vertex cover precedentemente create è stata svolta utilizzando il software IBM ILOG CPLEX Optimization Studio versione 12.9 [15]. Si tratta di una suite sviluppata dall'azienda francese ILOG, di proprietà di IBM, che fornisce strumenti per la modellazione e la risoluzione di problemi di ottimizzazione e che rappresenta di fatto lo stato dell'arte nell'ambito della programmazione lineare. CPLEX prende il nome dal metodo del simplesso (*simplex method*) implementato in C, sebbene al giorno d'oggi la suite sia arrivata a comprendere diversi algoritmi addizionali utilizzati nel campo della programmazione matematica e diverse altre interfacce verso altri ambienti e linguaggi diversi dal C.

Proprio una di queste interfacce, che permette di far dialogare software Python con il risolutore C, è stata utilizzata al fine di automatizzare il processo di risoluzione delle nume-

rose istanze di problema. Si è infatti sviluppato un apposito script che permette di leggere i vari file `.lp` contenenti le istanze MIP e risolverli sequenzialmente, implementando in questo modo un processo di risoluzione di tipo *batch*.

Gli output prodotti da CPLEX nella risoluzione di ciascuna istanza sono stati memorizzati su disco in file separati (uno per ciascuna istanza di problema), in modo tale da permettere in un secondo momento l'estrazione dei dati ritenuti interessanti mediante l'utilizzo di un parser sviluppato ad-hoc. Oltre agli output prodotti dal risolutore, per ogni istanza del problema sono stati memorizzati nel medesimo file anche due informazioni aggiuntive non comprese nell'output standard del risolutore, ovvero il gap tra lower bound del problema e soluzione trovata, utile nel caso in cui il problema vada in *time limit* per valutare l'ottimalità della soluzione trovata, e il valore della funzione obiettivo terminata la risoluzione, corrispondente alla cardinalità dell'insieme vertex cover trovato come soluzione.

3.4 Estrazione ed analisi dei risultati

L'estrazione dei dati dai file di testo contenenti gli output del risolutore è stata anch'essa implementata mediante un semplice script dedicato, sviluppato in linguaggio Python. Lo script permette, in maniera sequenziale, di leggere tutti i file contenenti gli output del risolutore in una specifica cartella e produrre, a partire da questi ultimi, un file CSV (*Comma-Separated Values*, formato di file basato su file di testo molto utilizzato nell'importazione ed esportazione di una tabella di dati) per ciascuna delle classi di grafo analizzate. Un esempio della struttura di uno dei quattro file CSV ottenuti come output è riportato in Figura 3.5.

	name	n	p	seed	time	ticks	sol_nodes	gap	time_lim	edges	cnnct_cmp	avg_clust	std_dev_clust
0	gnp_0000	100	0.1	1	0.44	99.51	69.0	0.0	False	508	1	0.113342	0.053480
1	gnp_0001	100	0.1	2	0.45	100.66	68.0	0.0	False	484	1	0.097705	0.047745
2	gnp_0002	100	0.1	3	0.54	138.64	69.0	0.0	False	487	1	0.083568	0.043205

Figura 3.5: Esempio di struttura di un file CSV contenente i risultati della risoluzione di tutti i problemi di vertex cover di una certa tipologia di grafi, in questo caso corrispondente ai grafi $G(n, p)$.

Come si può notare dalla figura precedente, per ognuna delle istanze sono stati salvati diversi dati sulla risoluzione e sul grafo associato al problema di ottimizzazione. Una breve spiegazione di ciascuno di essi è stata riportata in Tabella 3.1.

name	codice univoco associato a ciascuna delle istanze di grafo generate
parametri generazione del grafo	combinazione dei parametri con cui è stata generata l'istanza di grafo; a diverse tipologie di grafo corrispondono diversi insiemi di parametri
seed	seme random utilizzato nella generazione del grafo
time	tempo impiegato dal risolutore a trovare una soluzione ottima o, in alternativa, il TIME_LIMIT imposto
ticks	misura effettuata da CPLEX di quanto il risolutore ha impiegato a risolvere l'istanza del problema
sol_nodes	cardinalità del vertex cover individuato, che corrisponde al valore della funzione obiettivo al termine dell'ultima iterazione
gap	gap di ottimalità tra soluzione trovata e lower bound al valore della soluzione
time_limit	valore booleano che specifica se il risolutore è andato in TIME_LIMIT o meno
edges	numero di archi dell'istanza di grafo
cnnct_cmp	numero di componenti connesse dell'istanza di grafo
avg_clust	indice di clustering globale medio del grafo
avg	

Tabella 3.1: Informazioni estratte per ogni istanza di problema di vertex cover risolta.

L'analisi dei dati estratti è stata infine svolta con il supporto di un Jupyter Notebook [16], applicazione web open-source che permette di coniugare in un unico documento testo ed esecuzione di codice.

Capitolo 4

Risultati sperimentali

4.1 Grafi $G(n, p)$

4.2 Grafi regolari

4.3 Grafi di Watts-Strogatz

4.4 Grafi di Barabási-Albert

Capitolo 5

Conclusioni

Ringraziamenti

Bibliografia

[1] https://www.python.org/downloads/release/python-368/.

[2] https://github.com/giacomocamposampiero/bachelor-thesis.

[3] A. H. Land e A. G. Doig. «An Automatic Method of Solving Discrete Programming Problems». In: *Econometrica* 28.3 (1960), pp. 497–520. ISSN: 00129682, 14680262. URL: http://www.jstor.org/stable/1910129.

[4] M. Padberg e G. Rinaldi. «Optimization of a 532-city symmetric traveling salesman problem by branch and cut». In: *Operations Research Letters* 6.1 (1987), pp. 1–7. ISSN: 0167-6377. DOI: https://doi.org/10.1016/0167-6377(87)90002-2. URL: https://www.sciencedirect.com/science/article/pii/0167637787900022.

[5] Aric A. Hagberg, Daniel A. Schult e Pieter J. Swart. «Exploring Network Structure, Dynamics, and Function using NetworkX». In: *Proceedings of the 7th Python in Science Conference*. A cura di Gaël Varoquaux, Travis Vaught e Jarrod Millman. Pasadena, CA USA, 2008, pp. 11–15.

[6] P. Erdös e A. Rényi. «On Random Graphs I». In: *Publicationes Mathematicae Debrecen* 6 (1959), p. 290.

[7] J. D. Hunter. «Matplotlib: A 2D graphics environment». In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55.

[8] A. STEGER e N. C. WORMALD. «Generating Random Regular Graphs Quickly». In: *Combinatorics, Probability and Computing* 8.4 (1999), pp. 377–396. DOI: 10.1017/S0963548399003867.

[9] Duncan J. Watts e Steven H. Strogatz. «Collective dynamics of 'small-world' networks». In: *Nature* 393.6684 (1998), pp. 440–442. DOI: 10.1038/30918.

[10] Stanley Milgram. «The Small-World Problem». In: *Psychology Today* 1.1 (1967), pp. 61–67.

[11] Albert-László Barabási e Réka Albert. «Emergence of Scaling in Random Networks». In: *Science* 286.5439 (ott. 1999), pp. 509–512. ISSN: 1095-9203. DOI: 10.1126/science.286.5439.509. URL: http://dx.doi.org/10.1126/science.286.5439.509.

[12] L. D. Costa, F. Rodrigues e A. Cristino. «Complex networks: The key to systems biology». In: *Genetics and Molecular Biology* 31 (2008), pp. 591–601.

[13] Michael L. Bynum et al. *Pyomo–optimization modeling in python*. Third. Vol. 67. Springer Science & Business Media, 2021.

[14] William E Hart, Jean-Paul Watson e David L Woodruff. «Pyomo: modeling and solving mathematical programs in Python». In: *Mathematical Programming Computation* 3.3 (2011), pp. 219–260.

[15] IBM, ILOG CPLEX Optimization Studio 12.9.0. https://www.ibm.com/docs/en/icos/12.9.0.

[16] Thomas Kluyver et al. «Jupyter Notebooks - a publishing format for reproducible computational workflows». In: *Positioning and Power in Academic Publishing: Players, Agents and Agendas*. A cura di Fernando Loizides e Birgit Scmidt. Netherlands: IOS Press, 2016, pp. 87–90. URL: https://eprints.soton.ac.uk/403913/.