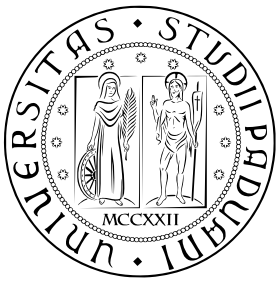

Università degli Studi di Padova

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, 19 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 di 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.

L'implementazione dei metodi per la generazione casuale di grafi è stata fatta 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 rappresentazione, diversi algoritmi per lo studio delle proprietà e il calcolo di misure in specifiche istanze di grafi.

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 $N = 1, 2, \ldots n$ 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).

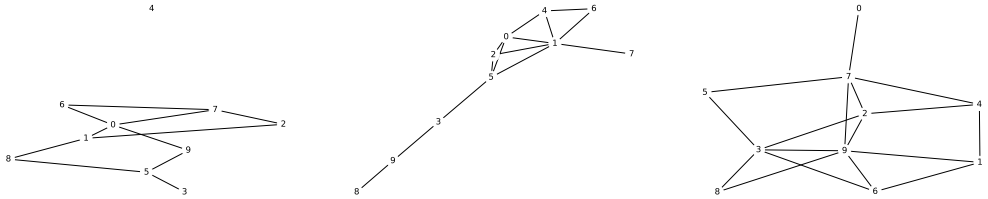

Figura 3.1: Diversi grafi $G(n, p)$ generati a partire dagli stessi parametri $n = 10$ e $p = 0.3$ con la libreria NetworkX e visualizzati graficamente con l'ausilio di un'altra libreria Python, Matplotlib [7].

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 $1 \ldots n$ 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 binomiale 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 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 [8], la cui caratteristica principale risiede nel possedere proprietà *small-world*, come ad esempio alto coefficiente di clustering globale e bassa lunghezza media dei cammini all'interno del grafo.

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, in cui ipotizzava per l'appunto un "*piccolo mondo*", composto da una rete di collegamenti tra persone relativamente breve [9]. Sulla base di queste considerazioni

3.1.3 Generazione di grafi regolari

3.1.4 Modello di Barabási-Albert

Capitolo 4

Risultati sperimentali

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] 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.

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