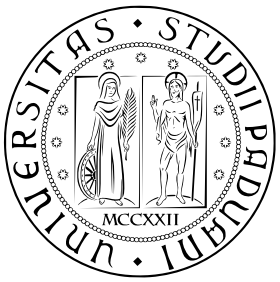

Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

Corso di Laurea Triennale in Ingegneria Informatica

Esperimenti mip per una classe di problemi di assegnamento quadratico

Relatore
Prof. Domenico Salvagnin

Laureando
Mattia Toffolon

Anno Accademico 2022/2023

Padova, XX luglio 2023

A mamma, papà e Cristian

Indice

Introduzione

La ricerca operativa, in inglese *operational research*, è una branca della matematica applicata che si occupa dell'analisi e risoluzione di complessi problemi decisionali mediante modelli matematici e metodi quantitativi. Lo scopo di questa disciplina è dunque quello di fornire supporto nell'attività decisionale in cui occorre gestire e coordinare risorse limitate sottoforma di soluzione al problema ottima, quando possibile, o ammissibile. Essa rappresenta un approccio scientifico alla risoluzione di problemi complessi che ha trovato largo successo in molti settori come l'economia, l'informatica e l'ingegneria.

Una nota branca della ricerca operativa è l'ottimizzazione. Essa si fonda sulla risoluzione di problemi di massimizzazione o minimizzazione di una data funzione, detta funzione obbiettivo, le cui variabili sono legate fra loro tramite un dato insieme di vincoli.

Questo studio verte su una particolare classe di problemi di ottimizzazione detta di assegnamento quadratico, in inglese *quadratic assignment problem* (QAP). In particolare, è stato analizzato l'andamento della complessità di risoluzione delle istanze del problema al variare della sua dimensione e di altri parametri utili alla sua generazione.

La stesura di questo elaborato è stata articolata in quattro capitoli. Inizialmente verranno brevemente esposti alcuni concetti teorici utili alla comprensione degli esperimenti svolti. Successivamente si proseguirà con le metodologie adottate per la realizzazione e risoluzione delle istanze e la presentazione dei risultati. Infine, verrà riportata una breve trattazione relativa alle conclusioni del lavoro svolto.

Tutto il codice sviluppato relativamente a questo studio è stato scritto in Python [1], linguaggio di programmazione *general-purpose* di alto livello che ha permesso di gestire ognuna delle diverse fasi in cui si è articolato lo svolgimento degli esperimenti, dalla generazione delle istanze all'elaborazione grafica dei risultati. Il codice sorgente, così come tutti i risultati delle elaborazioni, sono liberamente consultabili online nella *repository* utilizzata per gestire il controllo di versione del progetto [2].

Capitolo 1

Nozioni preliminari

In questo primo capitolo sono stati riportati alcuni concetti che rappresentano le fondamenta teoriche del lavoro svolto. Come prima cosa è stato definito formalmente il concetto di problema di ottimizzazione. Successivamente si è proseguito con una coincisa trattazione relativa a due importanti algoritmi usati nella risoluzione di problemi di ottimizzazione: *branch-and-bound* e *branch-and-cut*. Infine, è stata esposta la definizione formale di problema di assegnamento quadratico sul quale questo lavoro è stato basato. Molte delle definizioni in questo capitolo sono ispirate alle dispense del corso di *Modelli e Software per l'Ottimizzazione Discreta*, tenuto dal professore Domenico Salvagnin [3][4].

1.1 Problema di ottimizzazione

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 $x_j \in D_j$.

In generale, un problema di ottimizzazione nella forma (1.1) è intrattabile, nel senso che non esistono algoritmi efficienti (o addirittura algoritmi) per la sua risoluzione. Si rende pertanto necessario considerare casi particolari di questa formulazione che presentano struttura e proprietà particolari da poter sfruttare.

1.2 Programmazione lineare intera

Uno dei casi particolari della formulazione generale di problema di ottimizzazione (1.1) è quello della 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:

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

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 la tipologia di vincoli a disposizione nel processo di formalizzazione matematica del problema, determinando una maggior difficoltà in fase di modellazione. Tuttavia, nel caso della MIP, questa restrizione non comporta un'eccessiva limitazione sul tipo di problemi formulabili secondo questo paradigma. Alcuni esempi classici di problemi risolvibili mediante MIP sono *knapsack*, problemi di *scheduling*, *minimum vertex cover* e, naturalmente, *quadratic assignment problem*. Inoltre, l'introduzione dei vincoli di linearità ed interezza comporta notevoli vantaggi nella definizione ed implementazione di algoritmi di risoluzione, due dei quali sono stati descritti più nel dettaglio nelle seguenti sezioni.

1.2.1 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 [5].

Questo algoritmo permette di gestire il problema dell'esplosione combinatoria scartando intere porzioni dello spazio delle soluzioni attraverso operazioni di *pruning*. Ciò risulta possibile quando si riesce a dimostrare che questi sottospazi non possono contenere soluzioni migliori di quelle note. Branch-and-bound implementa inoltre una strategia *divide and conquer*, che permette di ottenere la soluzione al problema ricombinando quelle

relative a partizioni del problema stesso. Viene riportata di seguito una breve descrizione dell'algoritmo.

Sia F l'insieme delle soluzioni ammissibili di un problema di minimizzazione (oppure di 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. Il costo di tale soluzione nota $z = f(\bar{x})$, detto *incumbent*, rappresenta per sua natura un limite superiore al valore della soluzione ottima.

L'algoritmo branch-and-bound prevede una fase iniziale 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*, l'algoritmo termina in quanto si è trovata una soluzione ottima del problema. Se il rilassamento dovesse risultare impossibile, è possibile anche in questo caso terminare la ricerca di una soluzione in quanto si può concludere che anche il problema di partenza è impossibile.

Invece, nel caso in cui 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 tale che:

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

Tale fase, detta di *branching*, è giustificata dal fatto che la soluzione ottima del problema è data dalla minima tra le soluzioni delle varie separazioni $F_i \in F^*$ dette figli di F.

$$min\left\{c\left(x\right) \mid x \in F\right\} = min\left\{min\left\{c\left(x\right) \mid x \in F_i\right\} \mid F_i \in F^*\right\}$$

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 sottoproblemi da processare.

L'algoritmo procede quindi con il selezionare un sottoproblema P_i dalla coda e risolverne un rilassamento. Si possono presentare quattro casi differenti:

- 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 sottoproblema.
- Se il rilassamento del sottoproblema non ammette soluzione, allora si smette di esplorare l'intero sottoalbero 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 sottoalbero associato al sottoproblema corrente, in quanto non può portare ad una soluzione migliore di quella già nota (*pruning by optimality*).
- Infine, se non è stato in alcun modo possibile scartare o concludere l'esplorazione del sottoalbero associato a P_i, è necessario eseguire nuovamente il *branching*, aggiungendo i nuovi sottoproblemi alla lista dei sottoproblemi da processare.

L'algoritmo prosegue processando sottoproblemi finché la lista di questi 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 in maniera immediata, agendo sulle condizioni che regolano *bounding* e *branching*. Per quanto riguarda il primo, la scelta più diffusa consiste nel considerare il rilassamento lineare dei sottoproblemi, rilassando dunque il vincolo di interezza. Se la soluzione del rilassamento non è intera, una possibile separazione in sottoproblemi può essere effettuata considerando la partizione:

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

Infine, è importante notare come, per costruzione, ogni soluzione trovata dall'algoritmo è migliore dell'incumbent e, di conseguenza, l'andamento dell'upper bound del problema decresce fino al raggiungimento della soluzione ottima. A differenza degli upper bound, i lower bound dei singoli sottoproblemi non hanno invece valenza globale. Nonostante ciò, è comunque possibile derivare un lower bound globale considerando il minimo tra tutti i lower bound dei sottoproblemi ancora aperti. Avere a disposizione in qualsiasi momento entrambi i bound del problema, permetta quindi di valutare la bontà della soluzione provvisoria in ogni momento.

1.2.2 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 [6] e ideata appositamente per la risoluzione di problemi MIP.

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

L'idea alla base di questo algoritmo è quella di "rafforzare" la formulazione associata al rilassamento lineare di ogni sottoproblema mediante la generazione di piani di taglio. I vantaggi a livello risolutivo sono molteplici, tra cui una maggior probabilità di ottenere soluzioni intere del rilassamento lineare o, in alternativa, di ottenere lower bound più stretti e pertanto più efficienti 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 banale 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 $\alpha^T x^* > \alpha_0$.

1.3 Quadratic assignment problem

I QAP (Quadratic assignment problem), conosciuti anche come *problemi di assegnamento quadratico*, si fondano sulla ricerca di un posizionamento ottimale di un dato insieme di unità anche dette *facilities*, in riferimento ai *facility location problems* di cui i QAP sono una branca, in un dato insieme di posizioni anche dette *locations*. Come provato da S. Sahni e T. Gonzalez nel 1976 [7], un problema di assegnamento quadratico è di tipo NP-difficile, ciò significa che non esistono algoritmi in grado di trovarne una soluzione in un tempo polinomiale. Inoltre, può essere risolto all'ottimo solo per instanze particolarmente piccole.

Nello specifico, il problema può essere presentato come segue: n unità devono essere assegnate ad n posizioni differenti, sapendo che a_{ij} è il flusso di informazioni che deve essere trasferito dall'unità i all'unità j e che la distanza tra le posizioni r ed s è pari a b_{rs}, si vuole trovare l'assegnamento delle unità nelle posizioni ottimo, ovvero quello che minimizza la somma dei prodotti flusso $\times$ distanza. Matematicamente, il problema si può formulare nel seguente modo:

$$\min_{\pi \in P(n)} \sum_{i=1}^{n} \sum_{j=1}^{n} a_{ij} b_{\pi_i \pi_j}$$

dove $A = (a_{ij})$ e $B = (b_{rs})$ sone due matrici $n \times n$, $P(n)$ è l'insieme di tutte le possibili permutazioni di $\{1, 2, ..., n\}$ e π_i indica la posizione dell'unità i nella permutazione $\pi \in P(n)$.

Capitolo 2

Approccio sperimentale

A seguito di una prima parte teorica introduttiva, si procede in questo capitolo con la presentazione dell'impostazione pratica che si è voluto dare alle sperimentazioni condotte. La struttura secondo la quale verranno 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 parametri
- Generazione delle istanze di problemi MIP
- Risoluzione delle istanze di problemi
- Estrazione ed analisi dei risultati

2.1 Generazione dei parametri

Il primo problema presentatosi è stato quello relativo all'individuamento dei parametri di flusso e distanza, in quanto fondamentali per la creazione di istanze del problema e conseguentemente anche per la loro risoluzione.

La prima possibilità valutata è stata la generazione di valori casuali per comporre le matrici dei parametri A e B. Questa però è stato scartata fin da subito poichè utilizzare valori di tale tipologia non permette di fornire una versione pseudo-realistica di istanze del problema ed inoltre, non garantisce alcun tipo di uniformità nella generazione delle istanze, il che non ci permette di effettuare successivamente uno studio approfondito sulla loro complessità di risoluzione.

La soluzione a questo problema è stata individuata in un metodo illustrato in un articolo del professor Éric D. Taillard del 1995 [8]. Come verrà illustrato più nello specifico nella prossima sezione, il metodo utilizzato è detto Densità di grigio, in inglese (*Density of grey*). Esso, oltre a compensare i difetti del metodo di istanziamento casuale, permette di automatizzare la creazione delle matrici dei parametri realizzando un algoritmo che richiede ai fini della generazione esclusivamente due valori: dimensione dell'istanza e densità utilizzata.

Come anticipato nell'introduzione, per realizzare tale processo è stato fatto uso del linguaggio di programmazione *Python* [1] e come supporto *NumPy* [9], una libreria open source che aggiunge supporto a grandi matrici e array multidimensionali insieme a una vasta collezione di funzioni matematiche di alto livello per poter operare efficientemente su queste strutture dati.

2.1.1 Istanze Tai*c

Le istanze Tai*C, la cui denominazione deriva dal nome del loro ideatore, sono una particolare classe di problemi di assegnamento quadratico che si fondano sull'utilizzo del metodo di Densità di grigio, il quale può essere descritto nel seguente modo.

Al fine di ottenere una tonalità di grigio di densità pari a m/n , il metodo in questione consiste nel generare una griglia rettangolare contenente $n = n_1 \times n_2$ caselle quadrate, m delle quali sono nere e $n - m$ sono bianche. Giustapponendo molte di queste griglie è possibile ottenere una superficie grigia di gradazione pari a m/n. Per ottenere la miglior qualità di colore, è necessario che le caselle nere, o quelle bianche, siano sparse uniformemente nella griglia.

A prova di ciò viene qui riportata un'applicazione del metodo realizzata con una delle soluzioni ottenute.

Figura 2.1: a sinistra la griglia relativa ad una soluzione ottima di un'istanza *Tai36c* a densità $d = 30\%$, a destra quella ottenuta accostando 1600 griglie del tipo a sinistra

Per rendere al meglio l'idea che sta alla base del metodo, è possibile paragonare le caselle nere a degli elettroni e la griglia allo spazio accessibile agli elettroni. Il posizionamente di quest'ultimi deve essere effettuato in modo tale che la somma delle intensità delle forze di repulsione elettrostatica è minimizzata.

Nei problemi considerati, le forze f_{rstu} ($r, t \in [1, n_1]$ $s, u \in [1, n_2]$) presenti tra le caselle i e j locate rispettivamente nelle posizioni della griglia di coordinate (r, s) e (t, u) sono definite come segue:

$$f_{rstu} = \max_{v,w \in \{-1,0,1\}} \frac{1}{(r - t + n_1 v)^2 + (s - u + n_2 w)^2}$$

Per ottenere una densità di grigio pari a m/n è possibile risolvere un problema di tipo QAP (assegnamento quadratico) in cui i parametri sono definiti nel seguente modo.

$$a_{ij} = \begin{cases} 1 & \text{if } i \leq m \text{ and } j \leq m \\ 0 & \text{otherwise} \end{cases} \qquad b_{ij} = b_{n_2(r-1)+s \; n_2(t-1)+u} = f_{rstu}$$

La i-esima componente ($i \leq m$) di una soluzione π, $\pi_i = \pi_{n_2(r-1)+s}$ fornisce la posizione in cui deve essere posizonata una casella nera.

Nonostante la relativa semplicità di questi problemi, molti metodi risolutivi potrebbero non funzionare correttamente a causa dell'esistenza di più soluzioni con lo stesso valore obiettivo. Tali soluzioni posso essere ottenute scambiando la posizione di due caselle nere oppure attuando rotazioni, traslazioni o simmetrie delle caselle nella griglia. Queste tre infatti, sono azioni che non mutano il valore della soluzione associata.

Questi problemi sono identificati in letteratura dalla sigla $grey$n$_1$_n$_2$_m e sono stati risolti pseudo-ottimamente fino a valori $n_1 = n_2 = 8$. Scegliendo definizioni dei parametri di distanza differenti oppure variando le scelte prese sui valori n_1, n_2 ed m è possibile ottenere moltissime varianti del problema.

A questo punto è possibile sfruttare le definizioni degli elementi che caratterizzano questo modello, per realizzare degli algoritmi che generino la matrice dei flussi A e quella delle distanze B sulla base dei soli tre parametri richiesti, le dimensioni n_1 e n_2 (si ricorda queste sono in relazione con la dimensione dell'istanza n nel seguente modo: $n = n_1 \times n_2$) e la densità di grigio utilizzata d.

Qui di seguito sono riportati gli pseudocodici degli algoritmi in questione insieme a quello delle eventuali funzioni che sono state utilizzate all'intero di essi.

Algorithm 1 Matrix A generation

function A-GENERATOR(n, d)
    $*$ *sia* $A = [a_{ij}]$ $*$
    $m \leftarrow round\,(n \cdot (d/100))$
    for all $i \in [0, n)$ **do**
        for all $j \in [0, n)$ **do**
            if $i < m$ or $j < m$ **then**
                $a_{ij} \leftarrow 1$
            else
                $a_{ij} \leftarrow 0$
    return A

Algorithm 2 Matrix B generation

function FORCE(n_1, n_2, r, s, t, u)
    $max \leftarrow 0$
    for all $v \in \{-1, 0, 1\}$ **do**
        for all $w \in \{-1, 0, 1\}$ **do**
            $m \leftarrow 1 \,/\, \left((r - t + n_1 v)^2 + (s - u + n_2 w)^2\right)$
            $max \leftarrow \max\,(max, m)$
    return max

function B-GENERATOR(n_1, n_2)
    $*$ *sia* $B = [b_{ij}]$ *con valori inizializzati a zero* $*$
    $scale \leftarrow 100000$
    for all $r \in [0, n_1)$ **do**
        for all $s \in [0, n_2)$ **do**
            for all $t \in [0, n_1)$ **do**
                for all $u \in [0, n_2)$ **do**
                $i \leftarrow n_2 \cdot (r - 1) + s$
                $j \leftarrow n_2 \cdot (t - 1) + u$
                if $b_{ij} \neq 0$ **then**
                continue
                else
                $b_{ij} \leftarrow \text{round}\,(\text{Force}\,(n_1, n_2, r, s, t, u) \cdot scale)$
                $b_{ji} \leftarrow b_{ij}$
    return B

Come si può intuire osservando lo pseudocodice del primo algoritmo, nella sua costruzione è stata presa la decisione di esprimere la densità d in punti percentuali, ovvero $d \in [0, 100]$. Da qui il perchè nel calcolo del valore di m è necessaria la divisione per 100.

Relativamente al secondo algoritmo invece, sono necessarie alcune puntualizzazioni. Alla riga n°10 si può notare come il valore della distanza calcolata tramite la funzione *Force* non avviene direttamente ma vengono effettuate delle operazioni intermedie. Come prima cosa il valore ritornato dalla funzione viene scalato di un fattore *scale* pari a 100000 e successivamente viene arrotondato all'intero più vicino. In tale modo è stato possibile confrontare le matrici ottenute, nello specifico la *Tai64c* e la *Tai256c*, con quelle messe a disposizione dal sito web *QAPLIB* [10], una libreria online contenente diverse tipologie di istanze di prolemi di assegnamento quadratico tra cui la *Tai*c*. Quest'operazione ha permesso in fase di costruzione dell'algoritmo di garantirne la correttezza. All riga n°21 invece, si sfrutta la proprietà di simmetria della matrice B per risparmiare metà dei calcoli dei valori di distanza, riducendo così la complessità temporale dell'algoritmo. Si ricorda infatti che, date due unità qualsiasi, la distanza della prima dalla seconda è la medesima della seconda dalla prima.

Come si può notare sono state utilizzate due funzioni di cui non è stato riportato lo pseudocodice, *max* e *round*. Questo perchè sono due funzioni elementari messe a disposizione della libreria standard di *Python* [1] di cui il funzionamento è noto. La prima richiede due valori come parametri e ritorna il massimo tra questi mentre la seconda ritorna l'arrotondamento all'intero più vicino del numero fornito come parametro.

2.2 Generazione delle istanze di problemi MIP

Terminata la generazione delle matrici dei parametri, costituenti il *dataset* grezzo alla base di questo lavoro, è stato necessario ricavare a partire da ciascuna coppia di matrici A e B il corrispondente problema di programmazione lineare intera di assegnamento quadratico. Per fare questo è stato inizialmente necessario ricavare una modellazione algebrica del problema stesso, definendo variabili, vincoli e funzione obiettivo con cui rappresentarlo formalmente. Come sarà possibile vedere successivamente, a tale modello è stato necessario apportare diverse modifiche per renderlo prima corretto e poi anche ottimizzato. 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.

2.2.1 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 [11]. Un modello algebrico è composto dai seguenti elementi:

- *insiemi*, che racchiudono gli elementi del sistema permettendo l'indicizzazione delle grandezze trattate nel problema
- *parametri*, ossia quantità costanti e definite nella formulazione del problema stesso, che solitamente sono delle proprietà relative agli insiemi definiti precedentemente
- *variabili*, ossia le grandezze incognite del sistema, su cui l'algoritmo può agire (nei limiti imposti dai vincoli 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, piuttosto che definire una strategia con cui eseguire la ricerca della soluzione stessa. Un modello di programmazione matematica potrebbe essere quindi paragonato ad un linguaggio dichiarativo, che descrive il risultato che si vuole ottenere, piuttosto che ad un linguag-

gio procedurale, che indica invece la sequenza di passi da percorrere per arrivare a quel risultato.

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.

//DA CAMBIARE !!

La modellazione algebrica non è stata, 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

$$\begin{aligned} & min\left(\sum_{v \in V} x_v\right) \\ & x_u + x_v \geq 1 \qquad \forall (u,v) \in E, u \in V, v \in V \\ & x_v \in 0,1 \qquad \forall v \in V \end{aligned} \tag{2.1}$$

2.2.2 Linearizzazione del modello

prova

2.2.3 Semplificazione del modello

prova

2.2.4 Modellazione nel software

prova

2.3 Risoluzione delle istanze di problemi

prova

2.4 Estrazione ed analisi dei risultati

prova

Bibliografia

[1] Guido Van Rossum e Fred L Drake Jr. *Python tutorial*. Centrum voor Wiskunde en Informatica Amsterdam, The Netherlands, 1995.

[2] https://github.com/mattia-toffolon/QAP.

[3] Domenico Salvagnin. *Cenni di Programmazione Lineare Intera*. Università degli Studi di Padova, 2020.

[4] Domenico Salvagnin. *Introduzione all'ottimizzazione discreta*. Università degli Studi di Padova, 2018.

[5] 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.

[6] 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.

[7] Sartaj Sahni e Teofilo Gonzalez. «P-complete approximation problems». In: *Journal of the ACM* 23.3 (1976), pp. 555–565. DOI: 10.1145/321958.321975. URL: https://www.researchgate.net/publication/220432228_P-complete_approximation_problems_J_ACM_233_555-565.

[8] Éric D. Taillard. «Comparison of iterative searches for the quadratic assignment problem». In: *Location Science* 3.2 (1995), pp. 87–105. DOI: https://doi.org/10.1016/0966-8349(95)00008-6. URL: https://www.sciencedirect.com/science/article/pii/0966834995000086.

[9] C.R. Harris, K.J. Millman e S.J. van der Walt. «Array programming with NumPy». In: *Nature* 585 (2020), pp. 357–362. DOI: https://doi.org/10.1038/s41586-020-2649-2. URL: https://www.nature.com/articles/s41586-020-2649-2.

[10] R.E. Burkard et al. *QAPLIB - A Quadratic Assignment Problem Library*. `https://www.opt.math.tugraz.at/qaplib/inst.html`. 2002.

[11] Laura Brentegani Luigi De Giovanni. *Modelli di Programmazione Lineare*. Università degli Studi di Padova.