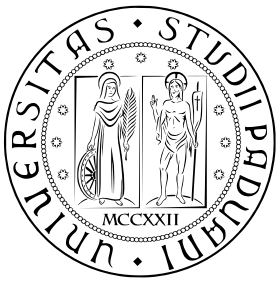

Università degli Studi di Padova

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, 18 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:

$$\begin{array}{ll} \min cx & \\ a_i x \sim b_i & i = 1, ..., m \\ l_j \leq x_j \leq u_j & j = 1, ..., n = N \\ x_j \in \mathbb{Z} & \forall j \in J \subseteq N = 1, ..., n \end{array} \tag{1.2}$$

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.

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

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*$\text{n}_1\text{\_n}_2\text{\_m}$ 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°20 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, facendo uso 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*, i quali 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.

Come si vedrà nel prosieguo della trattazione, nel caso del problema analizzato in questo lavoro la modellazione algebrica non è stata di immediata risoluzione.

Per prima cosa sono stati identificati gli insiemi del problema, ovvero

$$\begin{aligned} I &: \text{insieme delle posizioni } (\textit{locations}) \\ U &: \text{insieme delle unità } (\textit{facilities}) \end{aligned}$$

Per quanto riguarda l'identificazione dei parametri, quest'operazione è già stata trattata nella sezione precedente e rappresenta le fondamenta su cui l'istanze del problema vengono costruite. Facendo uso degli algoritmi appositamente costruiti è possibile ottenere tutti i parametri necessari in forma matriciale del tipo:

$$\begin{aligned} A = [a_{uv}] \quad & \text{con } a_{uv} : \text{flusso dall'unità } u \text{ all'unità } v \\ B = [b_{ij}] \quad & \text{con } b_{ij} : \text{distanza dalla posizione } i \text{ alla posizione } j \end{aligned}$$

Per quanto concerne le variabili del problema, queste sono indicizzate dall'insieme delle unità I e da quello delle posizioni U e la loro definizione è la seguente:

$$x_{iu} = \begin{cases} 1 & \text{se l'unità } u \in U \text{ è stata assegnata alla posizione } i \in I \\ 0 & \text{altrimenti} \end{cases}$$

È stato inoltre necessario definire due vincoli per assicurare la correttezza della soluzione trovata. Il primo garantisce che ogni unità venga assagnata ad esattamente una posizione e il secondo invece, che ogni posizione sia asseganta ad esattamente un'unità.

$$\begin{aligned} \sum_{i \in I} x_{iu} = 1 \quad & \forall u \in U \\ \sum_{u \in U} x_{iu} = 1 \quad & \forall i \in I \end{aligned}$$

Infine, la funzione obiettivo $f(x)$, che dovrà essere minimizzata dal risolutore, viene definita come segue.

$$f\,(x) \;=\; \min \sum_{i \in I} \sum_{u \in U} \sum_{j \in I} \sum_{v \in U} a_{uv} \cdot b_{ij} \cdot x_{iu} \cdot x_{jv}$$

Il modello algebrico del problema di assegnamento quadratico ricavato, può dunque essere complessivamente definito nel seguente modo:

$$\begin{array}{ll} \min \sum_{i \in I} \sum_{u \in U} \sum_{j \in I} \sum_{v \in U} \; a_{uv} \cdot b_{ij} \cdot x_{iu} \cdot x_{jv} & \\ \sum_{i \in I} \; x_{iu} = 1 & \forall \; u \in U \\ \sum_{u \in U} x_{iu} = 1 & \forall \; i \in I \\ x_{iu} \in \{0, 1\} & \forall \; i \in I, \; u \in U \end{array} \tag{2.1}$$

2.2.2 Linearizzazione del modello

Come è possibile notare, il modello algebrico ricavato precedentemente (2.1) non rispetta completamente i canoni della formulazione dei problemi di programmazione lineare intera (1.2). Ai fini della risoluzione del problema, è dunque necessario ricondurre il modello in oggetto alla formulazione *MIP* apportando alcune modifiche necessarie.

Nello specifico il problema risiede nella funzione obiettivo. Come lascia intuire il nome, la programmazione lineare intera esige un funzione obiettivo lineare, mentre in quella del modello in questione è presente il prodotto fra incognite ($x_{iu} \cdot x_{jv}$) che la rende non lineare.

Il problema è stato risolto effettuando un'operazione di linearizzazione. Tale processo consiste nell'introduzione di una variabile y_{iujv} che vada a sostituire il prodotto fra variabili, rendendo così lineare la funzione obiettivo.

Tale variabile è stata definita nel seguente modo:

$$y_{iujv} \;=\; x_{iu} \cdot x_{jv}$$

Tuttavia, affinchè questa variabile sia consistente con i valori delle variabili x_{iu} e x_{jv} che la definiscono è stato necessario introdurre tre ulteriori vincoli.

$$\begin{array}{ll} y_{iujv} \;\leq\; x_{iu} & \forall \; i, j \in I, \; u, v \in U \\ y_{iujv} \;\leq\; x_{jv} & \forall \; i, j \in I, \; u, v \in U \\ y_{iujv} \;\geq\; x_{iu} + x_{jv} - 1 & \forall \; i, j \in I, \; u, v \in U \end{array}$$

I vincoli sopra elencati garantiscono che la variabile y_{iujv} assuma valore 1 se e solamente se entrambe le variabili x_{iu} e x_{jv} assumono valore 1. Nel caso in cui anche solo una di esse dovesse assumere valore 0, anche y_{iujv} assumerebbe valore 0.

Pragmaticamente, la variabile qui introdotta indica se l'unità u è stata assegnata alla posizione i e contemporaneamente l'unità v è stata assegnata alla posizione j o meno.

Tenendo conto delle nuove modifiche e l'aggiunta dei nuovi vincoli, il modello complessivo corrisponde al seguente.

$$\begin{aligned}
&\min \textstyle\sum_{i\in I}\sum_{u\in U}\sum_{j\in I}\sum_{v\in U}\ a_{uv}\cdot b_{ij}\cdot y_{iujv} \\
&\textstyle\sum_{i\in I}\ x_{iu} = 1 \qquad \forall u \in U \\
&\textstyle\sum_{u\in U} x_{iu} = 1 \qquad \forall i \in I \\
&y_{iujv} \le x_{iu} \qquad \forall\ i,j \in I,\ u,v \in U \\
&y_{iujv} \le x_{jv} \qquad \forall\ i,j \in I,\ u,v \in U \\
&y_{iujv} \ge x_{iu} + x_{jv} - 1 \qquad \forall\ i,j \in I,\ u,v \in U \\
&x_{iu},\ y_{iujv} \in \{0,1\} \qquad \forall\ i,j \in I,\ u,v \in U
\end{aligned} \tag{2.2}$$

A questo punto della trattazione è doveroso effettuare un'importante osservazione. Come si può notare la variabile principale presente nella funzione obiettivo, ovvero quella introdotta in questa sezione, è indicizzata da quattro elementi tutti appartenenti a insiemi caratterizzati dalla medesima cardinalità, che per praticità indicheremo con la lettera n.

Ciò implica che dato n, numero totale delle posizioni e delle unità, vanno prese in esame n^4 variabili differenti e, come è facilmente intuibile, questo comporta pesanti conseguenze a livello di complessità computazionale nella risoluzione del problema. Si riportano qui di seguito alcuni esempi che evidenziano la questione.

$$\begin{aligned}
n = 4 \quad &\Rightarrow \quad \text{n° variabili} = 4^4 = 256 \\
n = 9 \quad &\Rightarrow \quad \text{n° variabili} = 9^4 = 6561 \\
n = 16 \quad &\Rightarrow \quad \text{n° variabili} = 16^4 = 65536
\end{aligned}$$

Nonostate la potenza computazionale non indifferente dell'hardware che è stato messo a dispozione per effettuare gli esperimenti, questo rapporto tra numero di variabili e cardinalità degli insiemi non permette di spingersi molto in avanti con la dimensione delle istanze nelle sperimentazioni. Senza apportare alcuna semplificazione al modello, non sarebbe stato possibile raccogliere dati a sufficienza per realizzare uno studio approfondito.

2.2.3 Semplificazione del modello

È possibile notare che l'ultima versione del modello (2.2) è valida per qualunque problema di assegnamento quadratico. Tuttavia, come è stato mostrato nella sezione 2.1, per generare i valori dei parametri di flusso e distanza è stata presa la decisione di considerare una classe di problemi QAP specifica, la classe Tai*C.

Un modo che potesse semplificare il modello, riducendo quindi il numero di variabili rispetto alla dimensione delle istanze, senza alterarne la validità, è stato trovato sfruttando le caratteristiche della classe di istanze considerata.

Nello specifico, sono state utilizzate le proprietà della matrice dei flussi $A = (a_{ij})$.
Si ricorda che il singolo parametro a_{ij} è definito nel seguente modo:

$$a_{ij} = \begin{cases} 1 & \text{se } i \leq n_1 \text{ e } j \leq n_1 \\ 0 & \text{altrimenti} \end{cases}$$

dove $n_1 = n \cdot d \leq n$ è il numero di caselle nere nella griglia di generazione.

Di conseguenza, la struttura della matrice A è la seguente.

$$A = \begin{bmatrix} 1 & 1 & . & 1 & 0 & . & 0 \\ 1 & . & . & 1 & 0 & . & 0 \\ . & . & . & . & . & . & . \\ 1 & 1 & . & 1 & 0 & . & . \\ 0 & 0 & . & 0 & 0 & . & 0 \\ . & . & . & . & . & . & . \\ 0 & 0 & . & . & 0 & . & 0 \end{bmatrix}$$

Si nota che solo i valori appartenenti al quadrante $n_1 \times n_1$ in alto a sinistra hanno valore unitario, mentre tutti gli altri hanno valore nullo. È quindi possibile definire un nuovo metodo di indicizzazione delle unità sulla base del loro flusso. Data l'unità $i \in I$:

$$m_i = \begin{cases} 1 & \text{se } i \leq n_1 \\ 2 & \text{altrimenti} \end{cases} \qquad m_i \in M$$

A questo punto esistono esclusivamente due tipologie di unità. Pertanto è possibile ridurre la matrice dei flussi alla seguente matrice 2×2:

$$A = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$$

Conseguentemente, muta anche la definizione delle variabili del problema.

$$x_{im} = \begin{cases} 1 & \text{se l'unità di categoria } m \text{ è stata assegnata alla posizione } i \in I \\ 0 & \text{altrimenti} \end{cases}$$

Infine, per garantire la correttezza del modello è necessario aggiungere i seguenti vincoli che garantiscono la corrispondenza tra la vecchia e la nuova matrice dei flussi.

$$\begin{aligned} &\textstyle\sum_{i \in I} x_{im} = n_1 && m = 1 \\ &\textstyle\sum_{i \in I} x_{im} = n - n_1 && m = 2 \end{aligned}$$

È facilmente dimostrabile che il secondo vincolo è rindondante in quanto è garantito dal primo. Per questo motivo verrà omesso nelle versioni successive del modello algebrico.

Trascurando momentaneamente la linearizzazione, il modello comprensivo delle ultime modifiche è il seguente.

$$\begin{aligned} &\min \textstyle\sum_{i \in I} \sum_{m \in M} \sum_{j \in I} \sum_{p \in M} a_{mp} \cdot b_{ij} \cdot x_{im} \cdot x_{jp} \\ &\textstyle\sum_{i \in I} x_{im} = n_1 && m = 1 \\ &x_{im} \in \{0, 1\} && \forall\, i \in I,\ m \in M \end{aligned}$$

A questo punto si può notare come a dare un contributo non nullo alle sommatorie siano solamente i prodotti relativi alle coppie di unità appartenenti entrambe alla categoria $m = 1$, per le quali infatti si ha $a_{mp} = a[1, 1] = 1$. Negli altri casi il flusso a valore nullo azzera il prodotto rendendo l'intero addendo nullo.

Per questo motivo è possibile semplificare ulteriormente il modello omettendo i valori di flusso e l'indicizzazione delle variabili rispetto alle unità.

In tal modo si ottiene il seguente risultato:

$$x_i = \begin{cases} 1 & \text{se un'unità di categoria 1 è stata assegnata alla posizione } i \in I \\ 0 & \text{altrimenti} \end{cases}$$

e per quanto riguarda il modello,

$$\begin{aligned} &\min \textstyle\sum_{i \in I} \sum_{j \in I} b_{ij} \cdot x_i \cdot x_j \\ &\textstyle\sum_{i \in I} x_i = n_1 \\ &x_i \in \{0, 1\} \qquad \forall\, i \in I \end{aligned}$$

Linearizzando il modello precedente con le stesse modalità illustrate nella sezione 2.2.2, è possibile ottenere la versione finale del modello algebrico del problema.

$$\begin{aligned} &\min \textstyle\sum_{i \in I} \sum_{j \in I} b_{ij} \cdot y_{ij} \\ &\textstyle\sum_{i \in I} x_i = n_1 \\ &y_{ij} \leq x_i \qquad \forall\, i, j \in I \\ &y_{ij} \leq x_j \qquad \forall\, i, j \in I \\ &y_{ij} \geq x_i + x_j - 1 \qquad \forall\, i, j \\ &x_i,\, y_{ij} \in \{0, 1\} \qquad \forall\, i, j \in I \end{aligned} \tag{2.3}$$

In questo caso, a differenza del modello nella versione (2.2), le variabili della funzione obiettivo y_{ij} sono indicizzate su soli due elementi, entrambi appartenti allo stesso insieme di cardinalità n. Ciò implica che dato il numero di possibili posizioni n, cardinalità di I, verranno prese in esame n^2 variabili differenti e non più n^4. È stato dunque ottenuto un guadagno di due ordini di grandezza sul numero di variabili, garantendo così la possibilità di risolvere, tramite l'hardware e i software messi a disposizione, istanze del problema di grandezza sufficientemente elevata da poterne realizzare uno studio.

2.2.4 Modellazione nel software

Affinchè le istanze generate tramite gli algoritmi presentati nella sezione 2.1.1 potessero essere tradotte in istanze di problemi MIP nella forma (2.3) risolvibili dal software CPLEX, è stato fatto uso di un'ulteriore libreria Python, Pyomo [12][13]. Si tratta anche in questo caso di una libreria open-source, stabile e largamente utilizzata per formulare, risolvere ed analizzare modelli per problemi di ottimizzazione.

Per la definizione del modello, questa libreria mette a disposizione due alternative. La prima consiste nel realizzare un modello astratto (`AbstractModel`) che separa modello e

dati, rispecchiando così maggiormente il modello algebrico, al fine di poter fornire i dati per la creazione delle istanze in un secondo momento. La seconda invece, consiste in un modello concreto (`ConcreteModel`) che, a differenza del primo, richiede immediatamente i dati poichè necessari alla creazione del modello stesso. Per questo lavoro si è optato per l'utilizzo di `ConcreteModel` poichè la generazione dei parametri non è particolarmente onerosa dal punto di vista computazionale, soprattutto se messa a confronto con la risoluzione delle istanze vera e propria. La scelta presa prevede dunque di generare i parametri richiesti ad ogni esecuzione di codice.

Questo procedimento è stato realizzato implementando gli algoritmi di generazione dei parametri come funzioni all'interno di uno script Python apposito. Tale file è stato poi importato nello script principale, che si occupa della generazione delle istanze e della loro risoluzione tramite CPLEX, e le sue funzioni sono state utilizzate nella definizione dei parametri del problema.

2.3 Risoluzione delle istanze di problemi

La risoluzione delle istanze di problemi di assegnamento quadratico generate come illustrato precedentemente, è stata effettuata facendo uso del software IBM ILOG CPLEX Optimization Studio [14]. Si tratta di una suite sviluppata dall'azienda francese ILOG, acquisita da IBM nel 2009, 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 diverse altre interfacce verso altri ambienti e linguaggi diversi dal C e diversi algoritmi addizionali utilizzati nel campo della programmazione matematica.

Nella risoluzione delle istanze è stato necessario interagire con il software risolutore. Tuttavia per effettuare ciò si è deciso di non utilizzare l'interfaccia Python offerta dalla suite CPLEX ma quella fornita dalla libreria Pyomo, ottimizzando così l'uso dei pacchetti importati.

L'interazione con il risolutore non è stata necessaria solo per avviare la risoluzione e successivamente acquisirne i risultati, ma anche per modificarne gli attributi. Dato che lo studio delle istanze è basato sull'osservazione dei tempi medi di ricerca della soluzione ottima, è stato necessario risolvere le stesse istanze più volte. Per come però sono strutturati gli algoritmi di risoluzione, senza variare alcun parametro il software produce la stessa soluzione in tempi pressoché identici, poichè questi hanno inizio sempre dallo

stesso punto di partenza ed effettuano poi le stesse scelte lungo il precorso di ricerca. Per ovviare a questo problema, tramite l'interfaccia offerta da Pyomo è stato modificato ad ogni risoluzione di istanza il parametro di CPLEX chiamato *RandomSeed*. Tale parametro, a cui può essere assegnato un valore non negativo qualunque compreso nell'intervallo [0,BIGINT], determina le prime scelte nella ricerca della soluzione ed in tal modo fa sì che ci sia diversità nei tempi di risoluzione e occasionalmente nella soluzione trovata, qualora l'istanza del problema presa in esame dovesse ammettere più soluzioni ottime.

Come visto nelle sezioni precedenti, la generazione di un'istanza Tai*C è basata su una coppia di valori (n, d) dove n è la dimensione dell'istanza e d la densità di grigio utilizzata. Nella realizzazione di questo studio i parametri delle istanze prese in esame appartengono ai seguenti insiemi.

$$n \in \{9, 16, 25, 36, 42, 45, 49\}$$
$$d \in \{10, 20, 30, 40, 50, 60, 70, 80, 90\}$$

Per ogni possibile coppia di valori, i tempi medi di risoluzione sono stati calcolati su 10 risoluzioni diverse relative a 10 valori del parametro *RandomSeed* differenti.

Gli output prodotti dal risolutore e le soluzioni da esso trovate sono stati memorizzati in appositi file. In tal modo è stato poi possibile analizzare le performance del risolutore e dare una rappresentazione grafica alle soluzioni.

2.4 Estrazione ed analisi dei risultati

Lo script Python che gestisce la creazione delle istanze ammette anche la loro risoluzione multipla secondo diversi *RandomSeed*. Questo ha permesso di ricavare i tempi medi risoluzione per istanza già in fase di esecuzione, dovendo così estrapolare un solo valore dagli output del risolutore.

Come però si vedrà in seguito, l'andamento dei tempi ha andamento esponenziale rispetto alla dimensione delle istanze. A causa di ciò, oltrepassata una certa soglia non è più stato possibile effettuare risoluzioni multiple ma si è dovuto effettuare risoluzioni singole, estrapolarne i dati e calcolare le medie solo successivamente.

Tutto ciò ammesso che la soluzione ottima all'istanza considerata sia ottenibile dal risolutore. A causa delle limiti fisici dell'hardware, la potenza computazionale offerta ha permesso di ottenere soluzioni solo entro certi valori del parametro n. Per quelli maggiori, è stato invece estrapolato l'*optimality gap* del risolutore al *time limit* di due

ore impostato per l'esecuzione. Il gap di ottimalità, in inglese *optimality gap*, indica la differenza in percentuale tra la miglior soluzione conosciuta e il *lower bound* del problema in un dato istante. Rappresenta pertanto un indice di vicinanza alla soluzione ottima.

Quest'insieme di dati è stato ricavato manualmente a partire dagli output dello script principale che come visto, si occupa dell'interazione con CPLEX e del calcolo dei tempi medi. Tali dati sono stati poi salvati in foglio di calcolo *Google Sheets* [15] dedicato.

Per quanto riguarda invece le soluzioni trovate, queste sono state salvate nel formato

$$\begin{array}{rl} i & x_i \\ i+1 & x_{i+1} \\ \dots & \dots \end{array}$$

dove i è l'indice della posizione e x_i indica nella posizione specificata è stata posizionata un'unità a flusso unitario, in file con estensione *.txt* dal nome

$$n\{\mathrm{n}\}\_\{\mathrm{n}_1\}\_\{\mathrm{n}_2\}\_d\{\mathrm{d}\}.txt$$

dove n_1 e n_2 sono le dimensioni della matrice generatrice dei valori di distanza fra posizioni (si ricorda che $n = n_1 \times n_2$).

È stato poi realizzato un terzo script ad-hoc per ricevere questi file come argomento, effettuarne il parsing ed utilizzare i dati relativi alle soluzioni ottime per realizzare delle rappresentazioni grafiche tramite le librerie Python open-source *Matplotlib* [16], che permette di creare visualizzazioni dati statiche, dinamiche e interattive, e *Seaborn* [17], che offre un'interfaccia ad alto livello per tracciare grafici statistici informativi ed attraenti.

Capitolo 3

Risultati sperimentali

Conclusa la presentazione delle scelte adottate in fase di impostazione delle sperimentazioni, si procede in questo capitolo con l'esposizione dei risultati ottenuti.

Per effettuare la risoluzione delle diverse istanze, è stato utilizzato un cluster messo a disposizione dal Dipartimeto di Ingegneria dell'Informazione dell'università. Nello specifico, la macchina in questione presenta le seguenti specifiche tecniche:

- Intel(R) Xeon(R) CPU E5-2623 v3 @ 3.00GHz quad-core
- 16GB RAM
- Linux Fedora 37
- Python 3.10.6
- IBM ILOG CPLEX Optimization Studio versione 22.1

Per garantire la compatibilità tra l'ultima versione del software CPLEX installata (22.1) e quella di Python, è stato fatto uso di *pyenv* [18], un'utility per Linux e MacOS che permette di tenere all'interno dello stesso sistema operativo differenti versioni dell'interprete Python. In questo modo è stato possibile utilizzare la versione Python richiesta da CPLEX (3.10) e non quella predefita del cluster (3.11).

Inoltre, sono state installate localmente tutte le librerie che vengono utilizzate all'interno dei vari script, che si ricordano essere: *Numpy*, *Pyomo*, *Matplotlib* e *Seaborn*.

3.1 Tempi medi di risoluzione

I tempi medi, estrapolati e memorizzati come illustrato nelle sezioni precedenti, sono riportati nelle seguenti tabelle indicizzate sulla dimensione delle istanze e sui valori di densità di grigio utilizzati.per generare quest'ultime.

I tempi sono riportati nel formato *hh:mm:ss.00*, dove *hh* indica il numero di ore impiegate, *mm* i minuti e *ss.00* i secondi arrotondati alla seconda cifra dopo la virgola.

Dimensione dell'istanza n

Densità di grigio d	9 (3x3)	16 (4x4)	25 (5x5)
10%	00:00:00.17	00:00:00.12	00:00:00.28
20%	00:00:00.14	00:00:00.10	00:00:00.27
30%	00:00:00.05	00:00:00.12	00:00:00.34
40%	00:00:00.04	00:00:00.21	00:00:01.59
50%	00:00:00.04	00:00:00.23	00:00:02.59
60%	00:00:00.06	00:00:00.22	00:00:02.46
70%	00:00:00.05	00:00:00.13	00:00:00.68
80%	00:00:00.05	00:00:00.12	00:00:00.48
90%	00:00:00.03	00:00:00.13	00:00:00.32

Dimensione dell'istanza n

Densità di grigio d	36 (6x6)	42 (7x6)	45 (9x5)
10%	00:00:00.59	00:00:00.84	00:00:01.08
20%	00:00:00.85	00:00:05.03	00:00:12.99
30%	00:00:13.00	00:04:56.35	00:15:13.03
40%	00:01:06.86	00:26:51.01	01:21:36.74
50%	00:00:18.40	00:06:13.01	00:49:02.35
60%	00:01:37.72	00:36:49.48	01:51:47.95
70%	00:01:10.83	00:27:21.45	01:37:33.22
80%	00:00:06.37	00:00:36.44	00:02:38.68
90%	00:00:00.77	00:00:01.30	00:00:05.66

Tramite uno script *Python* che fa uso di alcuni comandi delle librerie *NumPy* e *Matplotlib*, questi dati sono stati utilizzati per creare un grafico 3D ed uno 2D. Tali grafici sono stati realizzati per permettere di comprendere al meglio i risultati degli esperimenti ottenuti e di notare facilmente qual'è l'andamento del tempo di risoluzione delle istanze rispetto ai valori assunti dai parametri di generazione n e d.

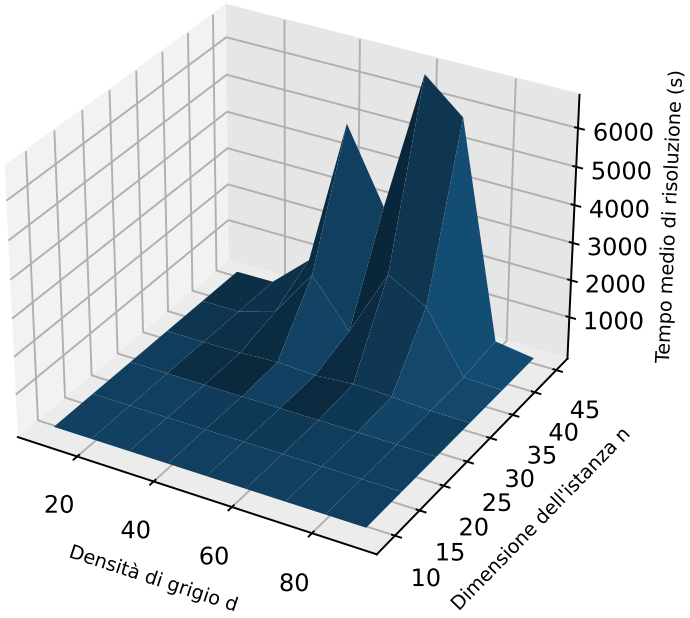

Figura 3.1: Grafico 3D dei tempi di risoluzione delle istanze Tai*c testate

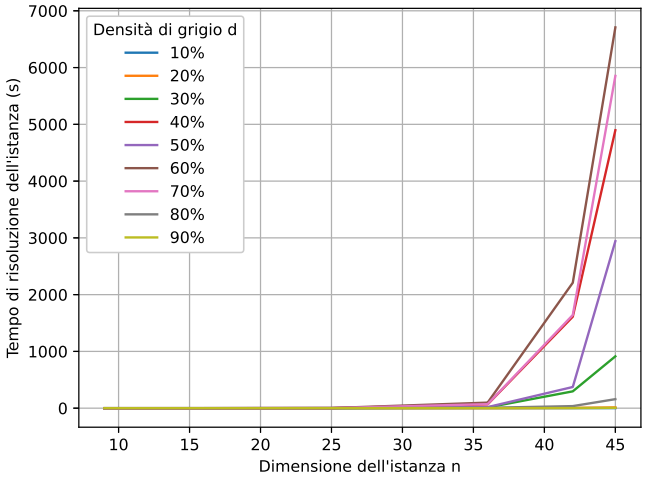

Figura 3.2: Grafico 2D dei tempi di risoluzione delle istanze Tai*c testate

Come si evince dai grafici, il comportamento assunto dei tempi di risoluzione rispetto alla dimensione dell'istanza è generalmente esponenziale, indipendentemente dal valore di densità di grigio considerato. Ciò che muta da una curva dei tempi rispetto all'altra è la velocità con cui i valori divergono. Nello specifico si nota come la crescita dei tempi per le densità intermedie 40%, 60% e 70% è maggiore rispetto a quella per $d = 50\%$, che a sua volta è notevolmente maggiore rispetto a quella per le densità 10% e 90%.

È possibile dunque affermare che a parità di dimensione n, le istanze Tai*c che richiedono un maggior tempo di risoluzione sono quelle generate da un valore di densità di grigio d prossimo, ma non corrispondente, al 50%, mentre quelle che ne richiedono meno sono quelle generate da valori di d prossimi allo 0% o al 100%.

Questo risultato corrisponde con quello ipotizzabile limitandosi ad osservare la teoria. Infatti, per densità prossime allo 0% il numero di unità da assegnare alle posizioni è ridotto ed invece per densità prossime al 100%, il numero di posizioni in cui non deve essere asseganta un'unità è esiguo. Per questo motivo si suppone che in ambo i casi il costo computazionale richiesto per trovare la soluzione ottima sia ridotto. Diversamente accade per densità vicine al 50%, valore per il quale il numero di posizioni essegnate ad un'unità è simile a quello delle posizioni libere. È pertanto comprensibile che si abbia il picco di complessità per tali valori.

La dimensione 45 (9×5) è stata l'ultima per la quale è stato possibile ricavare, al time limit di 2 ore, un tempo di risoluzione medio per ogni valore di densità di grigio. Difatti, per le istanze di dimensione 49 (7×7) sono stati ottenuti i seguenti dati:

Densità di grigio d

		10%	20%	30%	40%	50%
n	49(7x7)	00:00:01.19	00:00:43.57	00:51:50.19	12.04%	1.17%

Densità di grigio d

		60%	70%	80%	90%
n	49(7x7)	6.39%	3.02%	00:17:24.36	00:00:08.39

In queste tabelle compaiono valori di tipo diverso. Dato un valore di densità d, se nella relativa cella compare un tempo, esso corrisponde al tempo medio di esecuzione per d, mentre dove compare una percentuale significa che per quel d non è stato possibile calcolare il tempo cercato e pertanto è stato riportato il valore dell'*optimality gap* al *time limit* impostato di due ore.

3.2 Soluzioni trovate

Capitolo 4

Conclusioni

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.

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

[13] 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.

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

[15] Nancy Conner. *Google Apps: The Missing Manual: The Missing Manual.* " O'Reilly Media, Inc.", 2008, pp. 115–185.

[16] John D. Hunter. «Matplotlib: A 2D Graphics Environment». In: *Computing in Science and Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55. URL: https://www.computer.org/csdl/magazine/cs/2007/03/c3090/13rRUwbJD0A.

[17] Michael L Waskom. «Seaborn: statistical data visualization». In: *Journal of Open Source Software* 6.60 (2021), p. 3021. URL: https://joss.theoj.org/papers/10.21105/joss.03021.pdf.

[18] Moshe Zadka. «Installing Python». In: *DevOps in Python: Infrastructure as Python*. Springer, 2022, pp. 1–6. URL: https://link.springer.com/chapter/10.1007/978-1-4842-7996-0_1#Sec2.