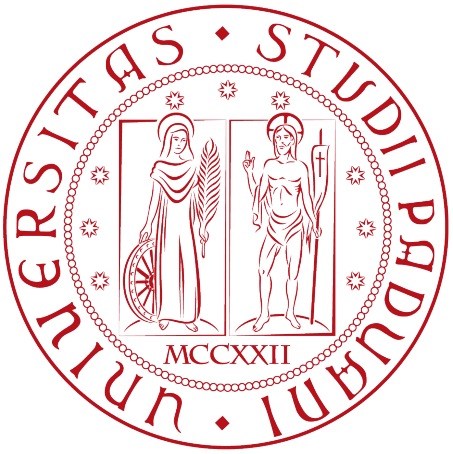
Università degli Studi di Padova

Dipartimento di Matematica "Tullio Levi-Civita"

Corso di Laurea in Informatica



Studio e implementazione di algoritmi per la stima delle frequenze di conflitti tra veicoli sincronizzati su grafi a griglia

Tesi di laurea

Relatore

Prof. Luigi De Giovanni

Laureando

Elia Scandaletti
1216751

Anno Accademico 2021-2022

Sommario

Il presente documento descrive il lavoro svolto durante il periodo di tirocinio interno, della durata di circa trecento ore, dal laureando Elia Scandaletti. Il tirocinio è stato svolto con la guida del Prof. De Giovanni nelle vesti di proponente e con la collaborazione della Prof.ssa De Francesco, facente parte del gruppo di Ricerca Operativa del Dipartimento di Matematica dell'Università di Padova. Tutor interno al Consiglio del Corso di Studio è stato il Prof. Palazzi.

Il fulcro del tirocinio è stato il Fleet Quickest Routing Problem on Grid (FQRP-G) che è un problema di instradamento su grafi attualmente oggetto di studi da parte del gruppo di Ricerca Operativa del Dipartimento.

Scopo del problema è muovere una flotta di veicoli autonomi su una griglia affinché raggiungano le loro destinazioni nel minor tempo possibile. Per far ciò bisogna evitare che i veicoli entrino in collisione tra loro, risolvendo in modo adeguato possibili conflitti sull'uso delle risorse (nodi e archi) della griglia. In letteratura sono stati definiti più tipi di conflitto, ciascuno con caratteristiche diverse, e diversi algoritmi di soluzione, la cui applicabilità è legata alla presenza di alcuni di questi tipi di conflitto.

Il tirocinio si è composto di una parte di studio e analisi teorica del FQRP-G e, successivamente, di una parte di progettazione e implementazione di algoritmi capaci di stimare empiricamente la frequenza di istanze del problema caratterizzate dalla presenza di determinati tipi di conflitto.

La parte teorica iniziale è servita per poter comprendere al meglio il dominio di applicazione del problema e lo stato dell'arte della letteratura a riguardo e, al contempo, capire la rilevanza di alcuni tipi di conflitto per gli algoritmi proposti in letteratura.

La successiva parte di progettazione e implementazione ha permesso una stima esatta e approssimata delle frequenze di particolari tipi di conflitto.

Ringraziamenti

Innanzitutto, vorrei esprimere la mia massima gratitudine al Prof. De Giovanni, relatore della mia tesi, per la pazienza, l'aiuto e il supporto fornitomi durante questi mesi di lavoro e durante la stesura della tesi.

Desidero ringraziare di cuore i miei genitori per essermi sempre stati vicini e per il sostegno che mi hanno dato in ogni momento durante gli anni di studio.

Infine, vorrei ringraziare i miei amici per i bellissimi anni passati insieme e, soprattutto, per il supporto che mi hanno dato in uno dei periodi che più ha impattato sulle nostre vite.

Padova, Luglio 2022 Elia Scandaletti

Indice

Elenco delle figure

Elenco delle tabelle

Elenco dei listati

Capitolo 1

Introduzione

In questo capitolo viene introdotto e motivato il tirocinio. Inoltre, verrà presentata la pianificazione del tirocinio, descrivendone obiettivi, contenuti formativi, attività previste e prodotti attesi. Infine, viene esposta l'organizzazione di questo documento.

1.1 Motivazioni del tirocinio

L'idea del tirocinio nasce sulla scia dei più recenti sviluppi della letteratura scientifica riguardante il Fleet Quickest Routing Problem on Grid (FQRP-G), problema introdotto e studiato in [3, 4, 5, 1, 2].

Il problema, oggetto di studio del gruppo di Ricerca Operativa del Dipartimento, si ispira alla movimentazione simultanea di un gruppo di veicoli automatizzati impiegati, ad esempio, in magazzini automatici, nei piazzali aeroportuali o nei porti.

Nel problema in esame, i veicoli sono inizialmente posti su un lato di una griglia e devono attraversarla per raggiungere la loro destinazione, posta sul lato opposto. I veicoli partono tutti contemporaneamente e si muovono a velocità costante. Per permettere l'instradamento più efficiente, si vuole che ogni veicolo segua un percorso di lunghezza minima, senza fermate intermedie. Tali percorsi vanno però scelti in modo accurato, in modo da evitare collisioni tra veicoli. Scopo del FQRP-G è minimizzare le dimensioni della griglia in modo tale che sia possibile muovere i veicoli su percorsi minimi senza collisioni e trovare tali percorsi.

Al momento, la letteratura propone varie soluzioni al problema, sia ottimali che subottimali. L'applicabilità e l'efficienza di queste soluzioni dipende in larga parte dalla presenza o meno di particolari tipi di conflitti, ovvero di particolari coppie di veicoli i cui percorsi minimi possono generare collisioni. Risulta perciò interessante uno studio per valutare le frequenze dei conflitti, distinguendo le varie tipologie.

Il tirocinio mira, in un primo momento, all'analisi teorica ed empirica dei conflitti e alla progettazione e l'implementazione di un algoritmo per il calcolo del numero di conflitti in una data istanza di FQRP-G. Successivamente, si progetteranno e implementeranno altri algoritmi che, data la dimensione di una istanza, calcolano in modo esatto e stimato la frequenza di conflitti per tipologia.

1.2 Piano di lavoro del tirocinio

Di seguito vengono riportati e descritti i contenuti formativi previsti dal tirocinio, i prodotti attesi, gli obiettivi formali e le attività previste per il raggiungimento di questi

ultimi. Successivamente, vengono motivate alcune minori variazioni del piano di lavoro avvenute in corso d'opera.

1.2.1 Contenuti formativi

Il tirocinio mira ad approfondire le conoscenze in ambito di progettazione e implementazione di algoritmi combinatori. In particolare, il tirocinio è ritenuto formativo sui seguenti aspetti:

* analisi dello stato dell'arte di un problema di matematica discreta di rilevanza applicativa;
* analisi teorica di un problema di conteggio combinatorio;
* progettazione e implementazione di algoritmi, in particolare per il calcolo empirico delle frequenze;
* valutazione delle prestazioni di algoritmi di calcolo combinatorio e analisi dei risultati ottenuti.

1.2.2 Prodotti attesi

Di seguito vengono riportati i prodotti risultanti dal tirocinio.

* Documento di descrizione e analisi dei conflitti: il documento descrive le singole tipologie di conflitti e i risultati della loro analisi teorica al fine di determinare, per quanto possibile, una valutazione analitica delle loro frequenze.
* Documento di descrizione e progettazione degli algoritmi: il documento definisce nel dettaglio, ad esempio tramite pseudocodice, gli algoritmi sviluppati, in particolare:
  - un algoritmo A1 che, data un'istanza di FQRP-G, calcola il numero di conflitti distinti per tipologia;
  - un algoritmo A2 che, data la dimensione dell'istanza, calcola in modo esatto, per enumerazione, il numero medio di conflitti per tipologia;
  - un algoritmo A3 che, data la dimensione dell'istanza, stima le frequenze dei conflitti per tipologia generando un opportuno campione casuale di istanze.

  Il documento definisce inoltre la progettazione del software che implementa gli algoritmi stessi.
* Implementazione degli algoritmi: gli algoritmi definiti saranno implementati utilizzando un linguaggio di programmazione (C++ o altro linguaggio adeguato), eventualmente utilizzando il calcolo parallelo.
* Definizione di benchmark di valutazione dei modelli di programmazione matematica: definizione di insiemi di istanze di FQRP-G sulla base dell'occorrenza di particolari pattern di conflitto.
* Documento di descrizione e analisi dei risultati computazionali: il documento riporta i risultati dei test computazionali, in termini di statistiche descrittive e confronto con i risultati analitici, e di descrizione delle performance dei modelli di programmazione matematica sul benchmark individuato.

1.2.3 Obiettivi

Ciascun obiettivo è identificato da un codice univoco formato da una lettera che ne indica la categoria e un numero di due cifre incrementale. I codici per le categorie sono:

* O per i requisiti obbligatori, vincolanti in quanto obiettivo primario richiesto dal committente;
* D per i requisiti desiderabili, non vincolanti o strettamente necessari, ma dal riconoscibile valore aggiunto;
* F per i requisiti facoltativi, rappresentanti valore aggiunto non strettamente competitivo.

Lo stage si pone i seguenti obiettivi:

* Obbligatori:

  O01 Acquisizione di competenze di progettazione e implementazione di algoritmi combinatori;

  O02 Analisi preliminare di fattibilità per la determinazione analitica delle frequenze per una o più tipologie di conflitto;

  O03 Progettazione di algoritmi per il calcolo esatto delle frequenze;

  O04 Implementazione di algoritmi per il calcolo esatto delle frequenze;

  O05 Progettazione di algoritmi per la stima delle frequenze su base campionaria;

  O06 Implementazione di algoritmi per la stima delle frequenze su base campionaria;

* Desiderabili

  D01 Parallelizzazione degli Algoritmi A2 e A3 presentati in Sezione 1.2.2;

  D02 Generazione di un benchmark di istanze di FQRP-G;

* Facoltativi

  F01 Test computazionali di modelli di programmazione matematica sul benchmark;

  F02 Definizione di funzioni analitiche per il calcolo delle frequenze dei conflitti.

1.2.4 Attività previste

La Tabella 1.1 riporta le attività previste dal tirocinio e, per ciascuna di esse, la durata stimata in ore.

1.2.5 Variazioni

In corso d'opera, a seguito di alcuni sviluppi teorici interni al gruppo di lavoro di Ricerca Operativa del Dipartimento, descritti nella Sezione 2.4.2, si è concordato col proponente di apportare alcune modifiche al piano di lavoro. In particolare, si è deciso di sostituire l'Obiettivo D01 con il seguente:

* Desiderabile

Descrizione attività	Durata in ore
Formazione su argomenti specifici dello stage e FQRP-G	32
Formazione su algoritmi su grafo	8
Formazione su modelli di programmazione matematica per FQRP-G	8
Analisi teorica dei conflitti e calcolo analitico delle frequenze	16
Studio dei metodi per la generazione di opportuni campioni casuali di istanze	24
Progettazione algoritmi per il calcolo dei conflitti e relative frequenze	72
Implementazione Algoritmo A1	10
Implementazione Algoritmo A2	10
Implementazione Algoritmo A3	20
Test preliminari	8
Studio e implementazione di possibili parallelizzazioni	32
Progettazione e implementazione interfaccia	6
Preparazione ed esecuzione test	28
Definizione di istanze benchmark per FQRP-G	6
Test computazionali dei modelli di programmazione matematica su benchmark	10
Analisi comparative e redazione documenti finali	30
Totale	320

Tabella 1.1: Preventivo delle ore previste per attività.

D03 Verifica dell'esistenza di grafi di conflitti di tipo misto che non sono foreste.

Di conseguenza, l'attività di "Studio e implementazione di possibili parallelizzazioni" è stata sostituita dall'analisi teorica del nuovo tipo di conflitti e dalle implementazioni necessarie per la loro individuazione, come dettaglieremo nel consuntivo presentato nel Capitolo 8. Inoltre, sono state modificate le definizioni degli algoritmi A2 e A3 che, non solo calcoleranno il numero medio di conflitti per tipologia, ma ne determineranno la distribuzione statistica. Ulteriore modifica riguarda i documenti prodotti. Si è scelto di unire i documenti di descrizione e analisi dei conflitti, di descrizione e progettazione degli algoritmi e di descrizione e analisi dei risultati computazionali in un'unica relazione poiché i tre argomenti sono strettamente correlati tra loro.

1.3 Organizzazione del testo

Il testo è strutturato in modo tale da ripercorrere il tirocinio nello stesso ordine in cui è stato svolto.

Il Capitolo 2 (Introduzione al problema) descrive formalmente il problema FQRP-G, richiamando lo stato dell'arte della letteratura e i risultati alla base delle attività di stage.

Il Capitolo 3 (Analisi teorica del problema) espone alcuni semplici risultati teorici ottenuti nel corso del tirocinio e riguardanti la probabilità che, in un'istanza

FQRP-G, un veicolo si muova verso destra, verso sinistra o solo in verticale, e la probabilità che un veicolo sia soggetto a un particolare tipo di conflitto.

Il Capitolo 4 (Progettazione dell'Algoritmo A1 e funzioni di supporto) espone gli algoritmi di base sviluppati nell'ambito del tirocinio e, in particolare, l'Algoritmo A1 per il calcolo del numero di conflitti in una data istanza, insieme ad alcune funzioni di supporto.

Il Capitolo 5 (Architettura e progettazione degli Algoritmi A2 e A3) presenta l'architettura scelta per il software e per gli Algoritmi A2 e A3.

Il Capitolo 6 (Implementazione) descrive l'implementazione degli algoritmi.

Il Capitolo 7 (Risultati sperimentali e confronto con risultati teorici) riporta i risultati sperimentali ottenuti e li confronta, dove possibile, con quelli teorici.

Nel Capitolo 8 (Conclusioni) viene effettuato un consuntivo e viene valutato il tirocinio nel suo insieme.

Riguardo la stesura del testo, relativamente al documento sono state adottate le seguenti convenzioni tipografiche:

* i pezzi di codice in linea sono riportati in carattere `monospaziato`;
* i termini tecnici dell'ingegneria del software vengono riportati in *corsivo*.

Capitolo 2

Introduzione al problema

In questo capitolo viene definito formalmente il FQRP-G, inoltre vengono introdotti alcuni concetti e notazioni che torneranno utili nella restante parte del documento. Infine, verrà esposto lo stato dell'arte della letteratura scientifica a riguardo.

2.1 Descrizione del problema

Di seguito viene riportata una definizione del problema basata su quella fornita in [4].

Si consideri una flotta di n veicoli V che si muove su un grafo a griglia planare $G = (N, A)$, dove N è l'insieme dei nodi e A l'insieme degli archi che possono essere orizzontali o verticali. Il grafo è costituito da n colonne e m righe o livelli. Chiameremo n la dimensione del problema, essendo, come vedremo, m limitato superiormente da n.

I veicoli sono identificati univocamente da un numero compreso tra 1 e n. Le colonne sono numerate progressivamente da 1 a n, i livelli da 1 a m. Per convenzione, si considera la colonna 1 quella più a sinistra e il livello 1 quello più in basso. Ogni nodo viene identificato dalla coppia (p, q), dove p è il suo livello e q la sua colonna.

Inizialmente, ogni veicolo $v \in V$ si trova sul nodo $(1, v)$ e deve raggiungere la sua destinazione $(m, \sigma(v))$. Ogni veicolo ha una colonna di partenza e una destinazione distinta da tutti gli altri veicoli. Per questo motivo un veicolo può essere identificato con la sua colonna di partenza e viceversa.

I veicoli devono muoversi lungo la griglia rispettando i vincoli di capacità unitaria di archi e nodi. In altre parole, due veicoli non possono muoversi sullo stesso arco in contemporanea e due veicoli non possono stare sullo stesso nodo in contemporanea. Se due o più veicoli non rispettano tali vincoli, si parla di collisione.

Al tempo zero, tutti i veicoli iniziano a muoversi sulla griglia per raggiungere la loro destinazione. I veicoli non possono mai fermarsi se non sul loro nodo di destinazione. Ogni arco richiede un'unità di tempo per essere attraversato. L'obiettivo è far arrivare tutti i veicoli a destinazione nel minor tempo possibile T.

Si noti che il tempo richiesto a ciascun veicolo $v \in V$ per raggiungere la sua destinazione, che chiameremo T_v, è il numero di passi orizzontali sommato al numero di passi verticali che deve compiere. Abbiamo dunque che $T_v = |v - \sigma(v)| + m$.

Considerato che $T = \max_{v \in V}(T_v) = \max_{v \in V}(|v - \sigma(v)|) + m$, è chiaro che l'obiettivo del problema equivale a minimizzare il numero di livelli m.

Osservazione 1. Si noti che gli unici due parametri che permettono di identificare un'istanza del problema da un'altra sono il numero di colonne della griglia e le destinazioni dei veicoli. Possiamo dunque identificare un'istanza attraverso la permutazione delle destinazioni $\sigma = \sigma(1), \sigma(2), \ldots, \sigma(n)$.

2.2 Classificazione di veicoli e conflitti

Il modo più naturale di far muovere un veicolo per minimizzarne il tempo di spostamento è lungo un percorso minimo. Essendo tale percorso su una griglia unitaria, si tratta di un percorso Manhattan, ovvero un percorso su una griglia che consente spostamenti in una sola direzione orizzontale e una sola direzione verticale [3]. Risulta evidente, quindi, che tutti i veicoli si sposteranno verso l'alto e potranno essere distinti in base alla direzione orizzontale in cui si muovono.

È dunque possibile dividere i veicoli in tre insiemi, come in [4]:

Definizione 1 (Partizionamento dei veicoli)**.**

$$\mathbb{L} = \{v \in V \mid \ \sigma(v) < v\}$$
$$\mathbb{R} = \{v \in V \mid \ \sigma(v) > v\}$$
$$\mathbb{H} = \{v \in V \mid \ \sigma(v) = v\}$$

Si osservi che i veicoli in $\mathbb{H}$ si muoveranno esclusivamente verso l'alto, mentre i veicoli in $\mathbb{R}$ e $\mathbb{L}$ muoveranno verso l'alto e, rispettivamente, verso destra o sinistra.

La scelta del percorso Manhattan su cui ciascun veicolo si muove è guidata dalla necessità che i veicoli non collidano tra loro. In particolare, per trovare dei percorsi adatti, bisogna tener conto dei conflitti esistenti.

Definizione 2 (Conflitto)**.** Si dice che due veicoli i e j sono in conflitto tra loro se esiste un cammino Manhattan di i e un cammino Manhattan di j tali che i due veicoli, se instradati senza interruzioni su tali cammini, collidono.

Esistono due tipi principali di conflitto: i conflitti di arco e i conflitti di nodo. Nei conflitti di arco, due veicoli potrebbero occupare lo stesso arco contemporaneamente. Al contrario, nei conflitti di nodo, due veicoli potrebbero occupare lo stesso nodo nello stesso momento.

2.2.1 Conflitti di arco

Un conflitto di arco si verifica quando un arco orizzontale è compreso tra le colonne di partenza e di arrivo di due veicoli ed è equidistante dalle loro posizioni di partenza. Questo significa che entrambi i veicoli potrebbero passare per quell'arco e, se lo fanno, ci arriverebbero nello stesso momento. Questo implica che i due veicoli vadano in direzioni opposte e che la loro distanza orizzontale iniziale sia dispari.

Si noti che il livello in cui si trova l'arco non è influente, poiché ogni livello aggiuntivo richiede che entrambi i veicoli facciano un passo in più prima di collidere.

Segue la definizione di conflitti di arco data in [4].

Definizione 3 (Conflitto di arco)**.** Dati due veicoli i e j, questi sono in conflitto di arco se e solo se le seguenti condizioni sono rispettate:

$$\begin{cases} i \in \mathbb{R} \\ j \in \mathbb{L} \\ j - i = 2n + 1, \ n \in \mathbb{N} \\ \sigma(j) < \frac{i+j}{2} < \sigma(i) \end{cases}$$

In [4] viene anche osservato che per evitare collisioni dovute a conflitti di arco, è sufficiente far sì che i veicoli in $\mathbb{R}$ e in $\mathbb{L}$ si muovano su livelli distinti, ovvero utilizzino due insiemi di archi orizzontali distinti.

2.2.2 Conflitti di nodo

I conflitti di nodo sono stati classificati in [4] e [2] come conflitti di nodo di tipo A, B, C e misto.

Conflitti di tipo A. I conflitti di tipo A sono quei conflitti in cui un veicolo già arrivato a destinazione, e quindi fermo, può collidere con un veicolo che si sta spostando in orizzontale. Affinché questo si verifichi, è necessario che la distanza orizzontale percorsa dal veicolo fermo sia minore della distanza orizzontale percorsa, fino al momento del conflitto, dal veicolo in movimento. Inoltre, la colonna del conflitto dev'essere compresa tra le colonne di partenza e arrivo del veicolo in movimento.

Segue la definizione di conflitto di tipo A data in [4].

Definizione 4 (Conflitto di tipo A)**.** Un conflitto di nodo tra i veicoli i e j è di tipo A se e solo se rispetta le seguenti condizioni:

$$\begin{cases} \min\{j, \sigma(j)\} < \sigma(i) < \max\{j, \sigma(j)\} \\ |i - \sigma(i)| < |j - \sigma(i)| \end{cases}$$

In [4] viene fatta la seguente osservazione.

Osservazione 2. I conflitti di nodo di tipo A sono gli unici che possono coinvolgere veicoli appartenenti ad $\mathbb{H}$.

Sempre in [4], viene osservato che per evitare collisioni dovute a conflitti di tipo A è sufficiente impedire gli spostamenti orizzontali sull'ultimo livello, eventualmente aggiungendone uno.

Conflitti di tipo B. I conflitti di tipo B sono quei conflitti di nodo che avvengono tra due veicoli che potrebbero arrivare allo stesso nodo nello stesso momento, purché tale nodo non sia nella colonna di destinazione di nessuno dei due veicoli. Affinché ciò sia possibile, è necessario che i due veicoli si muovano in direzioni opposte, che la colonna del conflitto sia esattamente a metà tra le loro colonne iniziali e che entrambi i veicoli passino per la colonna interessata.

Di seguito viene riportata la definizione di conflitto di tipo B data in [4].

Definizione 5 (Conflitto di tipo B)**.** Un conflitto di nodo tra i veicoli i e j è di tipo B se e solo se rispetta le seguenti condizioni:

$$\begin{cases} i \in \mathbb{R} \\ j \in \mathbb{L} \\ j - i = 2n,\ n \in \mathbb{N} \\ \sigma(j) < \frac{i+j}{2} < \sigma(i) \end{cases}$$

Conflitti di tipo C. I conflitti di tipo C sono simili ai conflitti di tipo B, con l'unica fondamentale differenza che la colonna in cui avviene il conflitto corrisponde alla colonna di destinazione di uno dei due veicoli. Questa differenza si motiva col fatto che il veicolo che è arrivato alla sua colonna di destinazione è costretto a fare solo movimenti verticali.

Di seguito viene riportata la definizione di conflitto di tipo C data in [4].

Definizione 6 (Conflitto di tipo C)**.** Un conflitto di nodo tra i veicoli i e j è di tipo C se e solo se rispetta le seguenti condizioni:

$$\begin{cases} i \in \mathbb{R} \\ j \in \mathbb{L} \\ j - i = 2n,\ n \in \mathbb{N} \\ \sigma(j) < \sigma(i) \\ \sigma(i) = \frac{i+j}{2} \vee \sigma(j) = \frac{i+j}{2} \end{cases}$$

Si noti che per la definizione data i conflitti di tipo C non sono simmetrici. È quindi possibile distinguere i due veicoli in base al ruolo assunto nel conflitto. A questo fine, in [4], viene introdotta una notazione per indicare i conflitti di tipo C.

Notazione (Conflitto di tipo C). Nel caso in cui i e j siano veicoli in conflitto di tipo C e $\sigma(i) = \frac{i+j}{2}$, diremo che i è il soggetto del conflitto e j ne è l'oggetto. Indicheremo questo conflitto con $i(j)$.

Nel caso, invece, in cui $\sigma(j) = \frac{i+j}{2}$, diremo che j è il soggetto del conflitto e i ne è l'oggetto. Indicando il conflitto con $j(i)$.

Si noti che, come osservato in [4], un veicolo può essere soggetto di un solo conflitto di tipo C ma può essere oggetto di più conflitti di tipo C.

Sigma-adiacenza. La sigma adiacenza è una caratteristica dei conflitti di tipo C introdotta per la prima volta in [5] e che è utile per poter definire i conflitti di tipo misto.

Di seguito, riportiamo la definizione data in [2], in quanto più semplice di quella data in [5].

Definizione 7. Un veicolo j è detto sigma-adiacente se esiste un veicolo i tale che $i(j)$ e $|\sigma(i) - \sigma(j)| = 1$.

Conflitti di tipo misto. In [2], vengono introdotti i conflitti di tipo misto. Questi conflitti sono conflitti di tipo B con particolari caratteristiche. In particolare, sono conflitti di tipo B tali che i veicoli coinvolti sono al contempo soggetti e oggetti di conflitti di tipo C e sono entrambi sigma-adiacenti.

Di seguito viene riportata la definizione data in [2].

Definizione 8 (Conflitto di tipo misto)**.** Un conflitto di nodo tra i veicoli $i \in \mathbb{R}$ e $j \in \mathbb{L}$ è di tipo misto se e solo se i e j sono in conflitto di tipo B e valgono le seguenti condizioni:

$$\begin{cases} i \text{ e } j \text{ sono entrambi sigma-adiacenti} \\ \exists\ i', i'' :\ i'(i) \wedge i(i'') \\ \exists\ j', j'' :\ j'(j) \wedge j(j'') \end{cases}$$

Introduciamo, in questo documento, una nuova classe di conflitti che include, come caso particolare, i conflitti misti.

Definizione 9 (Conflitto di tipo misto rilassato)**.** Un conflitto di nodo tra i veicoli $i \in \mathbb{R}$ e $j \in \mathbb{L}$ è di tipo misto rilassato se e solo se i e j sono in conflitto di tipo B e valgono le seguenti condizioni:

$$\begin{cases} \exists\ i', i'' :\ i'(i) \wedge i(i'') \\ \exists\ j', j'' :\ j'(j) \wedge j(j'') \end{cases}$$

Si noti che la definizione di conflitto di tipo misto rilassato richiama quella di conflitto di tipo misto data in [2], tuttavia, manca la condizione sulla sigma-adiacenza di i e j.

La definizione dei conflitti misti rilassati è necessaria in quanto l'Algoritmo A1, definito nella Sezione 4, fa riferimento a questa classe più ampia di conflitti. Pertanto, nel Capitolo 7 riporteremo dati sulle frequenze dei conflitti misti rilassati, che risultano una sovrastima delle frequenze di conflitto misto.

2.3 Catene di conflitti di tipo C

Si noti che, dalla definizione di Conflitto di tipo C, un veicolo può essere sia soggetto di un conflitto di tipo C, sia oggetto di un altro conflitto di tipo C allo stesso momento. Questo porta alla definizione di catene di conflitti di tipo C [4].

Definizione 10 (Catena di conflitti di tipo C)**.** Una catena di conflitti di tipo C (o semplicemente catena di conflitti) di lunghezza k è una sequenza di veicoli $(z_1, z_2, \ldots, z_k)$ tale che $z_i(z_{i+1})$ $\forall i, 1 \leq i \leq k$.

Si noti che deve esistere anche un veicolo z_{k+1}, che non viene incluso nella catena, tale che $z_k(z_{k+1})$.

Teorema 1. [4]. *La distanza tra partenza e destinazione dei veicoli in una catena di conflitti di tipo C è strettamente crescente.*

$$i(j) \implies |i - \sigma(i)| < |j - \sigma(j)|$$

Dimostrazione. Senza perdita di generalità, assumiamo $i \in \mathbb{R}$ e $j \in \mathbb{L}$. Bisogna dunque dimostrare che $\sigma(i) - i < j - \sigma(j)$.

$$\begin{aligned} i(j) &\implies \sigma(j) < \sigma(i) = \frac{i+j}{2} \\ &\implies \sigma(i) + \sigma(j) < i + j \\ &\implies \sigma(i) - i < j - \sigma(j) \end{aligned}$$

□

Corollario 1.1. [4]. *Una catena di conflitti di tipo C non può essere ciclica.*

2.4 Grafi di conflitto

Fino ad ora sono stati definiti i conflitti in modo indipendente tra loro, tuttavia, risulta interessante vedere come i conflitti interagiscono tra loro. Vengono quindi riportate le definizioni di grafo di conflitti di tipo C, grafo di conflitti di tipo misto e grafo di conflitti di tipo misto rilassato.

2.4.1 Grafo di conflitti di tipo C

In particolare, in [4] viene introdotta la definizione di grafo di conflitti di tipo C. Si tratta di un grafo in cui ogni arco corrisponde a un conflitto di tipo C.

Di seguito la definizione data in [4].

Definizione 11 (Grafo di conflitti di tipo C)**.** Un grafo di conflitti di tipo C è un grafo orientato in cui ogni vertice è un veicolo coinvolto in un conflitto di tipo C e ogni conflitto $i(j)$ è rappresentato da un arco uscente da i ed entrante in j.

Osservazione 3. Basandosi sul Corollario 1.1 e sulla Definizione 11 di Grafo di conflitti di tipo C, è evidente che il grafo dei conflitti di tipo C è una foresta di alberi orientati verso la propria radice.

2.4.2 Grafo di conflitti di tipo misto rilassato

Sebbene non sia ancora presente in letteratura, è possibile effettuare un'operazione simile con i conflitti di tipo misto e di tipo misto rilassato.

Definizione 12 (Grafo di conflitti di tipo misto)**.** Un grafo di conflitti di tipo misto è un grafo non orientato in cui ogni nodo è un veicolo coinvolto in un conflitto di tipo misto e ogni conflitto di tipo misto tra i veicoli i e j è rappresentato da uno spigolo tra i e j.

Definizione 13 (Grafo di conflitti di tipo misto rilassato)**.** Un grafo di conflitti di tipo misto rilassato è un grafo non orientato in cui ogni nodo è un veicolo coinvolto in un conflitto di tipo misto rilassato e ogni conflitto di tipo misto rilassato tra i veicoli i e j è rappresentato da uno spigolo tra i e j.

In contemporanea al tirocinio, in maniera indipendente da esso, è emerso nel gruppo di Ricerca Operativa del Dipartimento l'ipotesi che il grafo di conflitti di tipo misto (non rilassato) fosse una foresta. Le analisi computazionali svolte durante lo stage hanno permesso di dimostrare che il grafo di conflitti di tipo misto rilassato non è gerarchico, mentre non sono emersi risultati riguardanti i grafi di tipo misto non rilassato. Infatti, le prove sperimentali supportano la seguente osservazione.

Osservazione 4. Il grafo di conflitti di tipo misto rilassato può contenere cicli, come mostra il seguente esempio.

Esempio. L'istanza di dimensione $n = 30$ rappresentata dalla permutazione

$$19, 29, 18, 16, 27, 26, 20, 6, 8, 1, 22, 21, 23, 17, 28, 12, 4, 15, 7, 11, 2, 24, 13, 3, 9, 14, 30, 5, 25, 10$$

contiene un ciclo di conflitti di tipo misto rilassato. In Figura 2.1, viene riportato il grafo dei conflitti di tipo C dell'istanza e i conflitti che costituiscono un ciclo all'interno del grafo dei conflitti di tipo misto rilassato.

Sono richieste ulteriori ricerche riguardo la possibile presenza di cicli in grafi di conflitto di tipo misto (non rilassato).

2.5 Istanza partizionata

Alcune istanze, le istanze partizionate, possono essere risolte facilmente scomponendole in istanze indipendenti più piccole e più facili da risolvere.

Per questo motivo si è scelto di concentrare le attività del tirocinio sulle istanze non partizionate, ritenute più interessanti.

Definizione 14 (Istanza partizionata)**.** Un'istanza FQRP-G è detta partizionata se è possibile dividere i veicoli di tale istanza in parti tali per cui i cammini Manhattan di due veicoli appartenenti a parti distinte usano nodi e archi distinti. Questo equivale alla seguente condizione:

$$\max(i, \sigma(i)) < \min(j, \sigma(j)), \forall\, i, j : i < j \text{ appartenenti a parti distinte.} \tag{2.1}$$

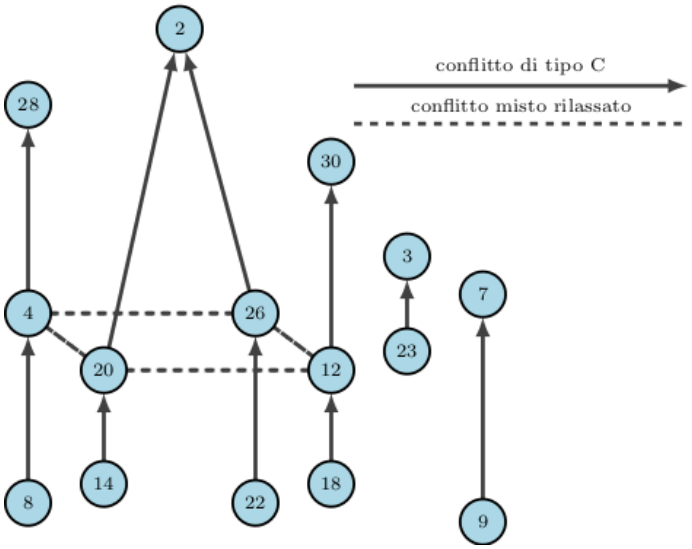


Figura 2.1: Grafo dei conflitti di tipo C e parte ciclica di un grafo dei conflitti di tipo misto rilassato.

2.6 Stato della letteratura sul FQRP-G

Ora che il problema è stato definito formalmente, viene fatta una rapida rassegna dello stato dell'arte della letteratura scientifica riguardante il FQRP-G.

Il problema FQRP-G viene introdotto per la prima volta in [3], dove viene proposto un algoritmo euristico polinomiale per la risoluzione del problema.

Successivamente, in [4], FQRP-G viene considerato nella prospettiva di determinare, dato il numero di colonne n, il minimo numero di righe che rendano il problema risolvibile. In questa direzione, vengono definite delle condizioni sufficienti affinché sia possibile, data una qualsiasi permutazione iniziale delle posizioni dei veicoli, trovare un insieme di cammini Manhattan senza collisioni. In particolare, si dimostra che è sempre possibile trovare una soluzione se si hanno a disposizione $m*$ livelli, dove $m*$ è definito nel seguente modo:

$$m* = \begin{cases} \lfloor \frac{n-1}{4} \rfloor + 3, & n \geq 3 \\ 2, & \text{altrimenti} \end{cases}$$

Tale numero deriva dalla lunghezza massima di una catena di conflitto che, sempre in [4], si dimostra essere:

$$k_{max} = \begin{cases} \lfloor \frac{n-1}{4} \rfloor + 1, & n \geq 3 \\ 0, & \text{altrimenti} \end{cases}$$

La soluzione proposta richiede che i veicoli si muovano su percorsi Manhattan semplici (i.e., i movimenti verticali avvengono solo sulla colonna di partenza o di destinazione) e che ogni livello consenta movimenti in un solo senso di marcia. In questo modo vengono evitati i conflitti di arco e di tipo B. I conflitti di tipo A vengono evitati impedendo spostamenti orizzontali sull'ultimo livello. I conflitti di tipo C

vengono evitati assicurandosi che, per ogni conflitto $i(j)$, i faccia il suo spostamento orizzontale a un livello superiore a j.

Gli stessi autori, in [5], propongono un nuovo metodo risolutivo sostenendo di poter sempre risolvere un problema di qualsiasi dimensione con 8 livelli. Tale affermazione viene confutata in [2], dove viene dimostrato che l'algoritmo è corretto solo sotto l'ipotesi che non ci siano conflitti di tipo misto, che qui vengono definiti per la prima volta. Inoltre, in quest'ultimo articolo, si evidenzia come l'algoritmo proposto sia in certi casi ambiguo e viene quindi riformulato in modo da sciogliere possibili dubbi.

Capitolo 3

Analisi teorica del problema

In questo capitolo si discuterà brevemente di alcuni risultati teorici ottenuti. Inizialmente, si discuterà della probabilità di un veicolo di appartenere agli insiemi $\mathbb{R}$, $\mathbb{L}$ e $\mathbb{H}$, secondo la Definizione 1. Successivamente, si discuterà la probabilità che un veicolo sia soggetto di un conflitto di tipo C.

3.1 Probabilità di un veicolo di appartenere a $\mathbb{R}/\mathbb{L}/\mathbb{H}$

Teorema 2. *Dato un problema di dimensione n e un veicolo i, si ha:*

$$P(i \in \mathbb{H}) = \frac{1}{n}$$

$$P(i \in \mathbb{R}) = P(i \in \mathbb{L}) = \frac{n-1}{2n}$$

Dimostrazione. Iniziamo osservando che gli insiemi $\mathbb{R}$, $\mathbb{L}$ ed $\mathbb{H}$ sono una partizione dell'insieme dei veicoli. Da questo segue che

$$P(i \in \mathbb{R}) + P(i \in \mathbb{L}) + P(i \in \mathbb{H}) = 1 \tag{3.1}$$

Inoltre, per la simmetria della definizione di $\mathbb{R}$ e $\mathbb{L}$, si ha

$$P(i \in \mathbb{R}) = P(i \in \mathbb{L}) \tag{3.2}$$

Calcolare $\mathbb{H}$ è banale, considerato che un solo valore di $\sigma(i)$ consente che i appartenga ad $\mathbb{H}$. Si ha

$$P(i \in \mathbb{H}) = \frac{1}{n} \tag{3.3}$$

Da (3.1), (3.2) e (3.3), si ottiene

$$P(i \in \mathbb{R}) = P(i \in \mathbb{L}) = \frac{n-1}{2n}$$

□

3.2 Probabilità di un veicolo di essere soggetto di conflitto di tipo C

Teorema 3. *Dato un problema di dimensione n e un veicolo i appartenente ad esso, la probabilità che i sia soggetto di un conflitto è:*

$$P(\exists j : i(j)) = \frac{1}{n^2(n-i)} \left\lfloor \frac{n-i}{2} \right\rfloor \left(\left\lfloor \frac{n+1}{2} \right\rfloor \left\lfloor \frac{n-1}{2} \right\rfloor - i(i-1) \right).$$

Dimostrazione. Per simmetria della definizione di $\mathbb{R}$ e $\mathbb{L}$, si ha

$$P(\exists j : i(j) \mid i \in \mathbb{R}) = P(\exists j : i(j) \mid i \in \mathbb{L}). \tag{3.4}$$

Inoltre, per la Definizione 6 di Conflitto di tipo C e la definizione di $\mathbb{H}$ (vedi Definizione 1), si ha

$$P(\exists j : i(j) \mid i \in \mathbb{H}) = 0. \tag{3.5}$$

Essendo $\mathbb{R}$, $\mathbb{L}$ e $\mathbb{H}$ partizione dell'insieme dei veicoli, da (3.4) e (3.5), otteniamo

$$\begin{aligned} P(\exists j : i(j)) &= 2 \times P(\exists j : i(j) \mid i \in \mathbb{R}) \\ &= 2 \times P(\exists j : i(j) \mid i \in \mathbb{L}). \end{aligned} \tag{3.6}$$

Scomponiamo la condizione $\exists j : i(j)$ in condizioni più piccole. Basandoci sulla definizione di Conflitto di tipo C, si ha

$$\exists j : i(j) \iff \begin{cases} j = 2\sigma(i) - i & \text{(3.7a)} \\ 1 \leq j \leq n & \text{(3.7b)} \\ \begin{cases} \sigma(j) < \sigma(i) & \text{se } i \in \mathbb{R} \\ \sigma(j) > \sigma(i) & \text{se } i \in \mathbb{L} \end{cases} & \text{(3.7c)} \\ \begin{cases} j \in \mathbb{L} & \text{se } i \in \mathbb{R} \\ j \in \mathbb{R} & \text{se } i \in \mathbb{L} \end{cases} & \text{(3.7d)} \end{cases}$$

Da notare che (3.7a) e (3.7c) implicano (3.7d). Si assuma il caso $i \in \mathbb{R}$; il caso $i \in \mathbb{L}$ è analogo. Per ipotesi, $i < \sigma(i)$, di conseguenza, per la (3.7a) $j = 2\sigma(i) - i > \sigma(i) + i - i = \sigma(i)$. Insieme alla (3.7c), $j > \sigma(i) > \sigma(j) \therefore j \in \mathbb{L}$. Da ciò si evince anche che $j \geq 1$ ($j \leq n$ nel caso $i \in \mathbb{L}$).

Si noti anche che, per le probabilità condizionate in (3.6), possiamo considerare solo la prima (o la seconda) condizione di (3.7c) e (3.7d).

Infine, si noti che la (3.7a) può semplicemente essere usata per sostituire j o, alternativamente, può essere portata fuori dal sistema e usata come vincolo.

Possiamo quindi scrivere che

$$\exists j : i(j) \iff \begin{cases} j \leq n \\ \sigma(j) < \sigma(i) \end{cases} \tag{3.8}$$

con $i \in \mathbb{R}$ e $j = 2\sigma(i) - i$.

Per la (3.8), si può affermare che

$$\begin{aligned}
&P(\exists j : i(j) \mid i \in \mathbb{R}) \\
&\quad = P(\exists J : i(j) \mid i < \sigma(i)) \\
&\quad = P(j \leq n \wedge \sigma(j) < \sigma(i) \mid i < \sigma(i)) \\
&\quad = \frac{P(j \leq n \wedge \sigma(j) < \sigma(i) \wedge i < \sigma(i))}{P(i < \sigma(i))} \\
&\quad = \frac{P(\sigma(j) < \sigma(i) \mid j \leq n \wedge i < \sigma(i)) \times P(j \leq n \wedge i < \sigma(i))}{P(i < \sigma(i))} \\
&\quad = \frac{P(\sigma(j) < \sigma(i) \mid j \leq n \wedge i < \sigma(i)) \times P(j \leq n \mid i < \sigma(i)) \times P(i < \sigma(i))}{P(i < \sigma(i))} \\
&\quad = P(\sigma(j) < \sigma(i) \mid j \leq n \wedge i < \sigma(i)) \times P(j \leq n \mid i < \sigma(i)).
\end{aligned} \tag{3.9}$$

con $j = 2\sigma(i) - i$.

Procediamo ora a calcolare il valore di ciascun fattore della (3.9).

$P(j \leq n \mid i < \sigma(i))$ Sostituendo j, otteniamo

$$j = 2\sigma(i) - i \leq n \implies \sigma(i) \leq \frac{n+i}{2}.$$

Per definizione del problema, $\sigma(i) \leq n$. Calcoliamo la probabilità come il rapporto tra casi favorevoli e casi possibili. I casi possibili sono $n - i$ (i.e. $i < \sigma(i) \leq n$). I casi favorevoli sono $\left\lfloor \frac{n-i}{2} \right\rfloor$ (i.e. $i < \sigma(i) \leq \frac{n+i}{2}$).

Da ciò

$$P(j \leq n \mid i < \sigma(i)) = \frac{\left\lfloor \frac{n-i}{2} \right\rfloor}{n-i}. \tag{3.10}$$

$P(\sigma(j) < \sigma(i) \mid j \leq n \wedge i < \sigma(i))$ Poiché il valore di $\sigma(j)$ è casuale e uniformemente distribuito, i casi possibili sono n, mentre quelli favorevoli sono $\sigma(i) - 1$ (i.e. $1 \leq \sigma(j) < \sigma(i)$). Dunque $P(\sigma(j) < \sigma(i)) = \frac{\sigma(i)-1}{n}$.

Al fine di rimuovere la dipendenza da $\sigma(i)$, bisogna risolvere una sommatoria per tutti i possibili valori di $\sigma(i)$. Dalle condizioni $j \leq n \wedge i < \sigma(i)$ sappiamo che $i < \sigma(i) \leq \frac{n+i}{2}$, per quanto visto al punto precedente.

$$
\begin{aligned}
P(\sigma(j) < \sigma(i) \mid j \leq n \wedge i < \sigma(i)) &= \sum_{k=i+1}^{\lfloor \frac{n+1}{2} \rfloor} \frac{k-1}{n} \times P(\sigma(i) = k) \\
&= \sum_{k=i+1}^{\lfloor \frac{n+1}{2} \rfloor} \frac{k-1}{n} \times \frac{1}{n} \\
&= \frac{1}{n^2} \sum_{k=i+1}^{\lfloor \frac{n+1}{2} \rfloor} k-1 \\
&= \frac{1}{n^2} \sum_{k=i}^{\lfloor \frac{n-1}{2} \rfloor} k \\
&= \frac{1}{n^2} \left(\sum_{k=1}^{\lfloor \frac{n-1}{2} \rfloor} k - \sum_{k=1}^{i-1} k \right) \\
&= \frac{1}{n^2} \left(\frac{\lfloor \frac{n+1}{2} \rfloor \lfloor \frac{n-1}{2} \rfloor}{2} - \frac{i(i-1)}{2} \right) \\
&= \frac{1}{2n^2} \left(\left\lfloor \frac{n+1}{2} \right\rfloor \left\lfloor \frac{n-1}{2} \right\rfloor - i(i-1) \right).
\end{aligned}
\tag{3.11}
$$

Mettendo insieme (3.9), (3.10) e (3.11), otteniamo che

$$
\begin{aligned}
P(\exists j : i(j) \mid i \in \mathbb{R}) &= \frac{1}{2n^2} \left(\left\lfloor \frac{n+1}{2} \right\rfloor \left\lfloor \frac{n-1}{2} \right\rfloor - i(i-1) \right) \times \frac{\lfloor \frac{n-i}{2} \rfloor}{n-i} \\
&= \frac{1}{2n^2(n-i)} \left\lfloor \frac{n-i}{2} \right\rfloor \left(\left\lfloor \frac{n+1}{2} \right\rfloor \left\lfloor \frac{n-1}{2} \right\rfloor - i(i-1) \right).
\end{aligned}
\tag{3.12}
$$

Considerando ora (3.6) e (3.12), segue che

$$
P(\exists j : i(j)) = \frac{1}{n^2(n-i)} \left\lfloor \frac{n-i}{2} \right\rfloor \left(\left\lfloor \frac{n+1}{2} \right\rfloor \left\lfloor \frac{n-1}{2} \right\rfloor - i(i-1) \right).
$$

□

Capitolo 4

Progettazione dell'Algoritmo A1 e funzioni di supporto

In questo capitolo viene proposto l'Algoritmo A1, non prima di aver introdotto alcune funzioni di supporto. Gli Algoritmi A2 e A3 verranno discussi nel Capitolo 5.

Gli obiettivi del tirocinio richiedono l'implementazione di tre algoritmi denominati:

A1 per il calcolo del numero di conflitti, per tipo, in una data istanza;

A2 per il calcolo esatto del numero medio di conflitti, per tipo, in funzione della dimensione del problema e la determinazione della loro distribuzione statistica;

A3 per il calcolo stimato del numero medio di conflitti, per tipo, in funzione della dimensione del problema e la determinazione della loro distribuzione statistica.

Mentre A1 è stato implementato come algoritmo indipendente, l'implementazione di A2 e A3 è stata ottenuta seguendo un'architettura che imita una *collection pipeline*. Come spiegato in [9], questa architettura consiste in un insieme di passi attraverso i quali passano dei dati che vengono trasformati. L'architettura utilizzata viene approfondita nel Capitolo 5 e lì gli Algoritmi A2 e A3 verranno esposti.

Per questo motivo, in questo capitolo verranno presentate alcune funzioni di supporto (funzioni che vengono utilizzate da altri algoritmi) e l'Algoritmo A1.

4.1 Convenzioni utilizzate

Di seguito riportiamo alcune convenzioni che sono state utilizzate all'interno dello pseudocodice:

* `function` indica che si sta dichiarando una funzione;
* `return` indica il valore ritornato dalla funzione;
* `set` indica che si sta creando una nuova variabile;
* `:=` indica un'assegnazione;
* `as` indica il tipo di una variabile, laddove sia necessario specificarlo;
* `assert` indica una condizione che è sempre vera in quel punto del codice; può essere usato per indicare delle precondizioni, come, ad esempio, alle righe 2 e 3 del Listato 4.1;

Listato 4.1: Funzione che verifica un conflitto di arco.

```
function checkArcConflict(instance, vehicleA, vehicleB) {
    assert(1 <= vehicleA < vehicleB <= instance.size)
    assert((vehicleB - vehicleA) % 2 == 1)
    set sigma := instance.sigma

    return (
        vehicleA < sigma(vehicleA) && vehicleB > sigma(vehicleB)
        && sigma(vehicleB) < (vehicleA + vehicleB)/2
        && sigma(vehicleA) > (vehicleA + vehicleB)/2
    )
}
```

* `swap` è una procedura che scambia i valori delle variabili passate come suoi argomenti.

Gli operatori aritmetici, logici e di confronto assumono lo stesso significato che hanno in C++ [7].

4.2 Funzioni di supporto

Di seguito vengono riportate alcune funzioni di supporto utilizzate. Inizialmente, vengono esposte funzioni capaci di verificare se due veicoli sono in conflitto di arco, di tipo A e di tipo B (Sezioni 4.2.1, 4.2.2, 4.2.3). Successivamente, viene data una funzione che, dato un veicolo, verifica se è soggetto di un conflitto di tipo C e, se vero, restituisce il veicolo oggetto di tale conflitto, secondo le definizioni date nella Sezione 4.2.4.

La successiva funzione verifica se due veicoli appartengono o meno alla stessa catena di conflitti di tipo C (Sezione 4.2.5).

Infine, vengono presentate una funzione che verifica se un'istanza è partizionata (Sezione 4.2.6) e un algoritmo per la costruzione di istanze partizionate (Sezione 4.2.7).

4.2.1 Verifica un conflitto di arco

La funzione riportata nel Listato 4.1 accetta in input un'istanza del problema `instance` che contiene la dimensione del problema `instance.size` e la funzione sigma `instance.sigma` (che rappresenta la permutazione σ), oltre a due veicoli `vehicleA` e `vehicleB`.

La funzione ritorna `true` se i due veicoli sono in conflitto di arco secondo la Definizione 3, `false` altrimenti.

Si noti che la funzione assume come precondizione che `vehicleA < vehicleB`, inoltre assume che la distanza tra i due veicoli sia dispari. Queste due precondizioni soddisfano la terza condizione della Definizione 3.

La condizione di ritorno verifica le rimanenti condizioni.

4.2.2 Verifica un conflitto di tipo A

La funzione riportata nel Listato 4.2 accetta in input un'istanza del problema `instance` che contiene la dimensione del problema `instance.size` e la funzione sigma `instance.sigma` (che rappresenta la permutazione σ), oltre a due veicoli `vehicleA` e `vehicleB`.

Listato 4.2: Funzione che verifica un conflitto di tipo A.

```
1  function checkAConflict(instance, vehicleA, vehicleB) {
2      assert(1 <= vehicleA < vehicleB <= instance.size)
3
4      set diff := abs(vehicleA - sigma(vehicleA))
5          - abs(vehicleB - sigma(vehicleA))
6      if (diff == 0) {
7          return false
8      }
9      if (diff < 0) {
10      swap(vehicleA, vehicleB)
11     }
12     assert(abs(vehicleA - sigma(vehicleA))
13         < abs(vehicleB - sigma(vehicleA)))
14
15     set sigma := instance.sigma
16     if (vehicleB > sigma(vehicleB)) {
17         set lower := sigma(B)
18         set upper := vehicleB
19     } else {
20         set lower := vehicleB
21         set upper := sigma(B)
22     }
23     return (lower < sigma(vehicleA) < upper)
24 }
```

La funzione ritorna `true` se i due veicoli sono in conflitto di tipo A secondo la Definizione 4, `false` altrimenti.

Si noti che la funzione assume come precondizione che `vehicleA < vehicleB`.

Dalla riga 6 alla riga 13, ci si assicura che i veicoli `vehicleA` e `vehicleB` corrispondano rispettivamente ai veicoli i e j della Definizione 4, inoltre ci si assicura che la seconda condizione della definizione stessa sia verificata.

Le successive righe, invece, verificano che venga soddisfatta la prima condizione della definizione.

4.2.3 Verifica un conflitto di tipo B

La funzione riportata nel Listato 4.3 accetta in input un'istanza del problema `instance` che contiene la dimensione del problema `instance.size` e la funzione sigma `instance.sigma` (che rappresenta la permutazione σ), oltre a due veicoli `vehicleA` e `vehicleB`.

La funzione ritorna `true` se i due veicoli sono in conflitto di tipo B secondo la Definizione 5, `false` altrimenti.

Si noti che la funzione assume come precondizione che `vehicleA < vehicleB`, inoltre assume che la distanza tra i due veicoli sia pari. Queste due precondizioni soddisfano la terza condizione della Definizione 5.

La condizione di ritorno verifica le altre tre condizioni della definizione.

Listato 4.3: Funzione che verifica un conflitto di tipo B.

```
1  function checkBConflict(instance, vehicleA, vehicleB) {
2      assert(1 <= vehicleA < vehicleB <= instance.size)
3      assert((vehicleB - vehicleA) % 2 == 0)
4
5      set sigma := instance.sigma
6
7      return (
8          vehicleA < sigma(vehicleA) && vehicleB > sigma(vehicleB)
9          && sigma(vehicleB) < (vehicleA + vehicleB)/2
10         && sigma(vehicleA) > (vehicleA + vehicleB)/2
11     )
12 }
```

4.2.4 Ottiene il veicolo oggetto di un conflitto di tipo C.

La funzione riportata nel Listato 4.4 accetta in input un'istanza del problema `instance` che contiene la dimensione del problema `instance.size` e la funzione sigma `instance.sigma` (che rappresenta la permutazione σ), oltre al veicolo `vehicle` soggetto del conflitto che si vuole verificare.

Se `vehicle` è soggetto di un conflitto di tipo C, secondo la Definizione 6, allora la funzione ritorna il veicolo oggetto di tale conflitto; altrimenti ritorna il valore speciale `nullVehicle`.

Alle righe 5 e 6 viene controllata una condizione sufficiente per escludere la presenza di un conflitto sulla base dell'Osservazione 2.

Successivamente, `target` viene costruito in modo tale da soddisfare la terza e la quinta condizione della definizione. Viene anche verificato che `target` sia un veicolo valido.

La condizione alla riga 11 distingue i casi in cui `vehicle` corrisponda a i o j nella Definizione 6. In entrambi i casi, vengono verificate le rimanenti condizioni della definizione dalle condizioni alle linee 12 e 16.

4.2.5 Verifica se due veicoli appartengono alla stessa catena di conflitti

La funzione riportata nel Listato 4.5 accetta in input un'istanza del problema `instance` che contiene la dimensione del problema `instance.size` e la funzione sigma `instance.sigma` (che rappresenta la permutazione σ), oltre a due veicoli `vehicleA` e `vehicleB`.

La funzione ritorna `true` se i due veicoli appartengono alla stessa catena di conflitti di tipo C (vedi Definizione 10), `false` altrimenti.

Dalla riga 6 alla riga 11, la funzione si assicura che se i veicoli appartengono alla stessa catena, il veicolo `vehicleA` precede il veicolo `vehicleB`. Per farlo confronta le distanze orizzontali che ciascun veicolo deve percorrere. Vedi Teorema 1.

Successivamente, il ciclo fa avanzare `vehicleA` lungo la catena a cui appartiene fino a che una delle seguenti condizioni si verifica:

* `vehicleA == nullVehicle`, significa che `vehicleA` ha percorso tutta la catena a cui appartiene senza mai trovare `vehicleB`, altrimenti il ciclo sarebbe terminato prima con la condizione `diff == 0`;

Listato 4.4: Funzione che ottiene il veicolo oggetto di un conflitto di tipo C.

```
function getCConflict(instance, vehicle) {
    assert(1 <= vehicle <= instance.size)
    set sigma := instance.sigma

    if (vehicle == sigma(vehicle)) {
        return nullVehicle
    }

    set target := 2*sigma(vehicle) - vehicle
    if(1 <= target <= instance.size) {
        if (vehicle < sigma(vehicle)) {
            if (sigma(target) < sigma(vehicle)) {
                return target
            }
        } else {
            if (sigma(target) > sigma(vehicle)) {
                return target
            }
        }
    }
    return nullVehicle
}
```

* `diff >= 0`, ci sono due casi possibili:
  - `vehicleA == vehicleB`, significa che i due veicoli appartengono alla stessa catena;
  - `vehicleA != vehicleB`, significa che i due veicoli non appartengono alla stessa catena, poiché questo violerebbe il Teorema 1.

4.2.6 Verifica se un'istanza è partizionata

La funzione riportata nel Listato 4.6 accetta in input un'istanza del problema `instance` che contiene la dimensione del problema `instance.size` e la funzione sigma `instance.sigma` che rappresenta la permutazione σ.

Come dimostrato nel seguito, la funzione determina se un'istanza è o meno partizionata. La dimostrazione è divisa in due parti. Nella prima si dimostra che la condizione (2.1) data nella Definizione 14 di Istanza partizionata è equivalente alla condizione (4.1) definita dal seguente Teorema 4. Successivamente, il Teorema 5 dimostra che l'algoritmo verifica la condizione equivalente (4.1).

Teorema 4 (Condizione equivalente alla definizione di istanza partizionata)**.** *Un'istanza è partizionata sse*

$$\exists\, k < n : \sigma(i) \leq k, \forall\, i \leq k. \tag{4.1}$$

Dimostrazione. Verifichiamo che la condizione (4.1) è sufficiente affinché l'istanza sia partizionata, dimostrando che la partizione $\{[1 \dots k], [k+1 \dots n]\}$ rispetta la condizione (2.1). Ovvero, dimostriamo che ogni veicolo in una delle due partizioni non

Listato 4.5: Funzione che verifica se due veicoli appartengono alla stessa catena di conflitti.

```
function checkSameCChain(instance, vehicleA, vehicleB) {
    assert(1 <= vehicleA <= instance.size)
    assert(1 <= vehicleB <= instance.size)
    set sigma := instance.sigma

    set diff := abs(vehicleA - sigma(vehicleA))
        - abs(vehicleB - sigma(vehicleB))
    if (diff > 0) {
        swap(vehicleA, vehicleB)
        diff := -diff
    }

    while (vehicleA != nullVehicle && diff < 0) {
        vehicleA := getCConflict(vehicleA)
        if (vehicleA != nullVehicle)
            diff := abs(vehicleA - sigma(vehicleA))
                - abs(vehicleB - sigma(vehicleB))
    }

    return vehicleA == vehicleB
}
```

può incrociare il percorso di un veicolo nell'altra partizione, formalmente:

$$\sigma(i) \leq k, \forall\, i \leq k \implies \max(i, \sigma(i)) \leq k, \forall\, i \leq k. \tag{4.2}$$

Appare evidente, per il principio dei cassetti, che le prime k destinazioni saranno riservate ai primi k veicoli. Di conseguenza, $\sigma(j) > k, \forall\, k < j \leq n$. Da ciò, si avrà

$$\sigma(j) > k, \forall\, j : k < j \leq n \implies \min(j, \sigma(j)) > k, \forall\, j > k. \tag{4.3}$$

Da (4.2) e (4.3), otteniamo che

$$\max(i, \sigma(i)) <= k < \min(j, \sigma(j)), \forall\, i, j : 1 \leq i \leq k \wedge k < j \leq n.$$

Da ciò otteniamo che la partizione $\{[1 \dots k], [k+1 \dots n]\}$ soddisfa la condizione (2.1).

Dimostriamo ora che la condizione è anche necessaria. Supponiamo per assurdo che l'istanza sia partizionata e, al contempo, $\nexists\, k < n : \sigma(i) \leq k, \forall\, i \leq k$. È possibile riscrivere la seconda assunzione come

$$\exists\, i \leq k : \sigma(i) > k, \forall\, k < n. \tag{4.4}$$

Poiché l'istanza è partizionata, ci sono almeno due parti. Poniamo k uguale all'estremo inferiore della seconda parte. Dalla condizione (2.1) sappiamo che

$$\max(i, \sigma(i)) < \min(j, \sigma(j)), \forall\, i, j : i < j \text{ appartenenti a parti distinte.}$$

Possiamo dunque scegliere $j = k$. Sostituendo nella (2.1), otteniamo

$$\max(i, \sigma(i)) < \min(k, \sigma(k)), \forall\, i < k. \tag{4.5}$$

Notiamo che non è più necessaria la condizione che i e k siano in parti distinti poiché $i < k$ e k è il minimo della sua parte.

È evidente ora come (4.4) e (4.5) siano in contrasto tra loro. Da ciò si arriva all'assurdo.

Concludiamo dunque che la condizione (4.1) è equivalente alla condizione (2.1) derivante dalla definizione di istanza partizionata.

□

Teorema 5 (Dimostrazione di correttezza dell'algoritmo proposto)**.** *L'algoritmo proposto nel Listato 4.6 ritorna* `true` *se l'istanza passata in input è partizionata,* `false` *altrimenti.*

Dimostrazione. Per dimostrare la correttezza dell'algoritmo proposto, verrà trovata una condizione invariante rispetto al ciclo `while` che, combinata con la condizione di uscita, permetterà di dimostrare la correttezza del valore ritornato. In particolare, verrà dimostrato che:

* l'invariante è vera prima della prima iterazione;
* l'invariante è vera al termine di ogni iterazione;
* il ciclo termina in un numero finito di passi;
* l'invariante, unita alla condizione di uscita, fornisce una condizione tale per cui il valore ritornato è corretto.

L'invariante del ciclo è la seguente:

$$\begin{cases} k \leq n & \text{(4.6a)} \\ k \leq maxSigma & \text{(4.6b)} \\ maxSigma = \max\limits_{\forall\, i \leq k}(\sigma(i)) & \text{(4.6c)} \\ \nexists\, k' < k : \sigma(i) \leq k',\ \forall\, i \leq k' & \text{(4.6d)} \end{cases}$$

Consideriamo $\max(\emptyset) = 1$. Questa assunzione non crea problemi poiché 1 è l'estremo inferiore dell'insieme dei possibili valori che possono essere presi in considerazione.

L'invariante è vera prima della prima iterazione. Per dimostrarlo è sufficiente sostituire k con 0 e $maxSigma$ con 1. In questo modo tutte le condizioni sono verificate. In particolare la condizione (4.6d) è rispettata perché $\nexists\, k' < 0$.

L'invariante è vera alla fine di ogni iterazione. È sufficiente dimostrare che, in un qualsiasi ciclo, se l'invariante è vera all'inizio, sarà vera anche alla fine. Per indicare il valore delle variabili all'inizio dell'iterazione, applichiamo il pedice $_v$; mentre con il nome delle variabili stesse indichiamo il loro valore alla fine dell'iterazione. Indicheremo con $4.6a_v$, $4.6b_v$, $4.6c_v$ e $4.6d_v$ le condizioni assunte per vere all'inizio dell'iterazione.

Poiché siamo entrati nel ciclo, è lecito assumere come ipotesi che la condizione di ingresso sia rispettata:

$$\begin{cases} k_v < n & \text{(4.7a)} \\ k_v < maxSigma_v & \text{(4.7b)} \end{cases}$$

Osserviamo che:

1) n e σ sono costanti;

2) $k = k_v + 1$;

3) $maxSigma \geq maxSigma_v$;

4) $maxSigma = \max(maxSigma_v, \sigma(k))$.

Analizziamo le condizioni una alla volta.

Condizione (4.6a):

$$\begin{aligned} & k_v < n && \because 4.7a \\ \implies\ & k \leq n && \because \textit{Oss.2} \end{aligned}$$

Condizione (4.6b):

$$\begin{aligned} k_v < maxSigma_v & && \because 4.7b \\ \leq maxSigma & && \because \textit{Oss.3} \\ \implies\ & k \leq maxSigma && \because \textit{Oss.2} \end{aligned}$$

Condizione (4.6c):

$$\begin{aligned} maxSigma_v &= \max_{\forall\, i \leq k_v}(\sigma(i)) && \because 4.6c_v \\ &= \max_{\forall\, i < k}(\sigma(i)) && \because \textit{Oss.2} \\ maxSigma &= \max(maxSigma_v, \sigma(k)) && \because \textit{Oss.4} \\ &= \max(\max_{\forall\, i < k}(\sigma(i)), \sigma(k)) \\ &= \max_{\forall\, i \leq k}(\sigma(i)) \end{aligned}$$

Condizione (4.6d):

$$\begin{aligned} & \nexists\, k' < k_v : \sigma(i) \leq k', \forall\, i \leq k' && \because 4.6d_v \\ maxSigma_v > k_v & && \because 4.7b \\ \implies\ & \max_{\forall\, i \leq k_v}(\sigma(i)) > k_v && \because 4.6c_v \\ \implies\ & \exists\, i \leq k_v : \sigma(i) > k_v \\ \implies\ & \neg(\sigma(i) \leq k_v, \forall\, i \leq k_v) \\ \implies\ & \nexists\, k' = k_v : \sigma(i) \leq k', \forall\, i \leq k' \\ \left.\begin{array}{l} \nexists\, k' < k_v : \sigma(i) \leq k', \forall\, i \leq k' \\ \nexists\, k' = k_v : \sigma(i) \leq k', \forall\, i \leq k' \end{array}\right\} & \implies \nexists\, k' \leq k_v : \sigma(i) \leq k', \forall\, i \leq k' \\ \implies\ & \nexists\, k' < k : \sigma(i) \leq k', \forall\, i \leq k' && \because \textit{Oss.2} \end{aligned}$$

Il ciclo termina Considerato che a ogni iterazione il valore di k aumenta di 1 e considerato che una condizione sufficiente di uscita dal ciclo è $k \geq n$, allora è evidente che il ciclo termina in un numero finito di iterazioni, al più n.

L'invariante, con la condizione di uscita, implica che il valore ritornato è corretto. Il ciclo termina quando la sua condizione non è più soddisfatta, ovvero quando

$$k \geq n \vee k \geq maxSigma. \tag{4.8}$$

Dobbiamo dimostrare che la condizione negata, insieme all'invariante che, per quanto dimostrato, sarà vera anche al termine dell'ultima iterazione, permette di ritornare sempre il valore corretto. In particolare vogliamo dimostrare che

$$k < n \iff \exists\, k < n : \sigma(i) \leq k, \forall\, i \leq k.$$

Si noti che la parte destra dell'implicazione è esattamente la condizione (4.1), mentre la parte sinistra è il valore ritornato dall'algoritmo. Distinguiamo i casi $k < n$ e $k \geq n$.

Caso $k < n$: In questo caso, dev'essere necessariamente

$$\begin{cases} k \geq maxSigma, & \because \text{ 4.8} \\ k \leq maxSigma, & \because \text{ 4.6b} \\ maxSigma = \max\limits_{\forall\, i \leq k}(\sigma(i)), & \because \text{ 4.6c} \end{cases}$$

Da ciò deriva che

$$\begin{aligned} k &= maxSigma \\ &= \max_{\forall\, i \leq k}(\sigma(i)) \\ &\implies k \geq \sigma(i), \forall\, i \leq k. \end{aligned}$$

Si può quindi affermare che $k < n \implies \exists\, k < n : \sigma(i) \leq k, \forall\, i \leq k$.

Caso $k \geq n$: In questo caso, dev'essere necessariamente

$$\begin{cases} k \leq n, & \because \text{ 4.6a} \\ \nexists\, k' < k : \sigma(i) \leq k', \forall\, i \leq k', & \because \text{ 4.6d} \end{cases}$$

Da ciò deriva che

$$k \geq n \implies k = n \implies \nexists\, k' < n : \sigma(i) \leq n, \forall\, i \leq n.$$

Si può quindi affermare che $\neg(k < n) \implies \nexists\, k < n : \sigma(i) \leq k, \forall\, i \leq k$.

Considerando quanto dimostrato nei due casi precedenti, segue che

$$k < n \iff \exists\, k < n : \sigma(i) \leq k, \forall\, i \leq k.$$

In altri termini, il valore ritornato dall'algoritmo indica correttamente se l'istanza in input è partizionata o meno.

□

Corollario 5.1. *La complessità computazionale dell'algoritmo proposto nel Listato 4.6 è $O(n)$, dove n è la dimensione dell'istanza.*

Dimostrazione. Questo segue naturalmente dalla dimostrazione del Teorema 5, in particolare dal fatto che il ciclo termina in al più n iterazioni, e ogni iterazione può essere eseguita in tempo costante. □

Listato 4.6: Funzione che verifica se un'istanza è partizionata.

```
function isPartitioned(instance) {
    set sigma = instance.sigma
    set n = instance.size

    set k = 0
    set maxSigma = 1
    while (k < n && k < maxSigma) {
        k += 1
        maxSigma = max(maxSigma, sigma(k))
    }
    return k < n
}
```

4.2.7 Algoritmo per la generazione di istanze non partizionate

Questo algoritmo, anche detto metodo di generazione a intervalli, serve a generare in modo casuale delle istanze non partizionate data la loro dimensione n.

Non esiste una prova che questo algoritmo generi le istanze con una distribuzione uniforme, ovvero che ogni possibile istanza non partizionata abbia la stessa probabilità di essere generata. Questo argomento verrà ulteriormente trattato nella Sezione 7.3.3.

L'algoritmo prevede di:

1. scegliere una destinazione casuale per il veicolo 1 tale che $1 < \sigma(1) \leq n$;
2. considerare $[1, \sigma(1)]$ come massimo intervallo non partizionato;
3. scegliere casualmente se il prossimo veicolo va verso destra o verso sinistra;
4. se il veicolo va verso destra:
   (a) scegliere una destinazione $\sigma(k)$ al di fuori del massimo intervallo non partizionato;
   (b) scegliere una partenza k non ancora scelta nel massimo intervallo non partizionato;
   (c) considerare $[1, \sigma(k)]$ come massimo intervallo non partizionato;
5. se il veicolo va verso sinistra:
   (a) scegliere una partenza k al di fuori del massimo intervallo non partizionato;
   (b) scegliere una destinazione $\sigma(k)$ non ancora scelta nel massimo intervallo non partizionato;
   (c) considerare $[1, k]$ come massimo intervallo non partizionato;
6. se il massimo intervallo non partizionato coincide con $[1, n]$, procedere al prossimo punto; altrimenti tornare al punto 3);
7. associare in modo casuale uniforme le rimanenti partenze con le rimanenti destinazioni.

Si noti che, al punto 3, è possibile adottare varie strategie per la scelta della direzione del veicolo. Quella più intuitiva è considerare ciascuna direzione equiprobabile.

Questa scelta deriva dal fatto che, a ogni iterazione, all'esterno del massimo intervallo non partizionato ci sono tante partenze libere quante destinazioni libere. Analogamente, questa proprietà vale anche all'interno dell'intervallo. Inoltre, questa scelta è compatibile con il Teorema 2, in cui si dimostra che i veicoli hanno pari probabilità di andare verso destra o di andare verso sinistra.

Si noti che, al punto 4a, poiché il percorso di ogni veicolo appartiene al massimo intervallo non partizionato, scegliere una destinazione esterna a tale intervallo garantisce che tale destinazione sia libera. Lo stesso ragionamento si può fare per la scelta della colonna di partenza al punto 5a.

4.3 Algoritmo per il calcolo del numero di conflitti in una data istanza

Lo scopo Algoritmo A1, riportato nel Listato 4.7, è di contare quanti conflitti di ciascun tipo ci sono in una data istanza. Considerato che i conflitti avvengono tra coppie di veicoli, viene naturale costruire l'algoritmo basandosi su due cicli annidati e, per ogni coppia di veicoli, verificare se sono in un conflitto di qualche tipo o meno. Tuttavia, è possibile porre delle condizioni al fine di evitare controlli inutili su coppie di veicoli che non possono essere in conflitto.

Di seguito viene spiegato l'algoritmo più dettagliatamente.

Conteggio dei conflitti di arco e di tipo A e B. Il controllo della presenza di questi tipi di conflitto avviene all'interno dei cicli annidati (linee 21-37), viene dunque controllata ogni coppia di veicoli. Si noti che, grazie alla condizione verificata alla linea 22, ogni coppia viene considerata una volta, in conformità con le precondizioni degli algoritmi di cui alle Sezioni 4.2.1, 4.2.2 e 4.2.3. Inoltre, la condizione alla riga 26 e la sua negazione alla riga 31 garantiscono le rimanenti precondizioni degli algoritmi di cui alle Sezioni 4.2.3 e 4.2.1.

Conteggio dei conflitti di tipo C. Al contrario, i conflitti di tipo C vengono verificati nel ciclo esterno (linee 13-38). In questo modo viene effettuato un controllo per ogni veicolo (linee 14-19), che è comunque sufficiente poiché è possibile risalire al veicolo oggetto di un conflitto partendo dal veicolo soggetto di tale conflitto.

Conteggio dei conflitti di tipo misto rilassato. Restano da contare i conflitti di tipo misto rilassato. Per fare ciò, durante i precedenti cicli, si è tenuto traccia di:

* veicoli soggetto di conflitti di tipo C con l'array `outgoingCConflict`;
* veicoli oggetto di conflitti di tipo C con l'array `incomingCConflict`;
* conflitti di tipo B con la lista `potentialMixedConflicts`.

Così facendo, nell'ultimo ciclo (linee 40-50), è facile verificare le condizioni di un conflitto di tipo misto rilassato, secondo la Definizione 9.

Osserviamo che, sebbene sia stata inserita nell'implementazione utilizzata per i risultati riportati nel Capitolo 7, la condizione alla linea 46 del Listato 4.7 non è necessaria poiché ridondante. Infatti, due veicoli in conflitto di tipo B, non potranno mai appartenere alla stessa catena di conflitti di tipo C, come osservato in [2].

Listato 4.7: Algoritmo A1 per il calcolo del numero di conflitti in una data istanza.

```
function getConflictCount(instance) {
    set outgoingCConflict := [false, ..., false] as array[1,
        instance.size]
    set incomingCConflict := [false, ..., false] as array[1,
        instance.size]
    set potentialMixedConflicts := [] as list
    set conflictCount := {
        arcType: 0,
        AType: 0,
        BType: 0,
        CType: 0,
        mixedType: 0
    }

    foreach (vehicle in instance.vehicles) {
        set CConflict := getCConflict(instance, vehicle)
        if (CConflict != nullVehicle) {
            conflictCount.CType += 1
            outgoingCConflict[vehicle] := true
            incomingCConflict[CConflict] := true
        }

        foreach (otherVehicle in instance.vehicles) {
            if (otherVehicle > vehicle) {
                if(checkAConflict(instance, vehicle, otherVehicle
                    )) {
                    conflictCount.AType += 1
                }
                if ((otherVehicle - vehicle) % 2 == 0) {
                    if (checkBConflict(instance, vehicle,
                        otherVehicle)) {
                        conflictCount.BType += 1
                        potentialMixedConflicts.push((vehicle,
                            otherVehicle))
                    }
                } else {
                    if(checkArcConflict(instance, vehicle,
                        otherVehicle)) {
                        conflictCount.arcType += 1
                    }
                }
            }
        }
    }

    foreach (conflict in potentialMixedConflicts) {
        if (
            incomingCConflict[conflict[0]] &&
            outgoingCConflict[conflict[0]] &&
            incomingCConflict[conflict[1]] &&
            outgoingCConflict[conflict[1]] &&
            !checkSameCChain(instance, conflict[0], conflict[1])
        ) {
            conflictCount.mixedType += 1
        }
    }

    return conflictCount
}
```

Capitolo 5

Architettura e progettazione degli Algoritmi A2 e A3

In questo capitolo, viene presentata l'architettura a pipeline utilizzata per gli Algoritmi A2 e A3 e la loro progettazione.

5.1 Collection pipeline

Scopo dell'Algoritmo A2 è calcolare in modo esatto, per enumerazione, il numero medio di conflitti per tipologia e la loro distribuzione statistica in funzione della dimensione del problema. Scopo dell'Algoritmo A3 è, invece, stimare il numero medio di conflitti per tipologia e la loro distribuzione statistica in funzione della dimensione del problema, generando un opportuno campione casuale di istanze.

Si noti che le operazioni svolte da entrambi gli algoritmi sono le stesse:

* generare delle istanze;
* estrarre informazioni da queste istanze;
* aggregare i dati raccolti in modo statistico.

Tuttavia, ciascuno di questi passi può essere implementato in vari modi. In particolare, l'Algoritmo A2 dovrà generare tutte le istanze di una data dimensione, mentre l'Algoritmo A3 dovrà generarne un campione casuale. Questo ha suggerito la necessità di separare le tre operazioni per poterle combinare a piacere e, dunque, di utilizzare una *collection pipeline*.

L'architettura a *collection pipeline*, di seguito solo *pipeline*, mira a implementare in modo flessibile gli Algoritmi A2 e A3. Come viene descritto in [9], questa architettura consiste in una serie di passi successivi in cui dei dati vengono trasformati. Ciascun passo viene implementato da un componente, ciascuno dei quali prende l'output del precedente come input. Per sua natura, i componenti della *pipeline* possono essere combinati in qualsiasi modo, purché l'output di uno e l'input del successivo siano compatibili.

In particolare, ciascuna delle operazioni citate sopra viene svolta da un diverso passo della *pipeline*. Avremo, dunque, una *pipeline* con tre passi, ciascuno implementato da un componente.

Il compito del primo componente, che chiameremo `Generator`, è generare un campione di istanze. Il secondo componente, che chiameremo `Counter`, prenderà queste

istanze e ne conterà i conflitti. L'ultimo componente, che chiameremo `Aggregator`, accetterà i conteggi dei conflitti e li aggregherà in modo da ritornare un dato consolidato unico che sarà anche l'output dell'algoritmo.

Poiché, come si è detto, ogni passo della *pipeline* può essere implementato in vari modi, non possiamo fare assunzioni su come sia rappresentata un'istanza, un conteggio o quale sia la statistica effettuata dall'ultimo componente. Per questo motivo, è necessario introdurre dei parametri di tipo nella *pipeline*. In particolare, sono necessari i seguenti parametri di tipo:

`instance_t`: il tipo dell'istanza, deve prevedere un valore speciale nullo;

`count_t`: il tipo del conteggio su un'istanza;

`result_t`: il tipo del risultato statistico.

Di seguito, i componenti della *pipeline*, `Generator`, `Counter` e `Aggregator`, vengono descritti uno per volta. Per ciascuno vengono riportati i metodi della loro interfaccia pubblica, seguendo la notazione Unified Modeling Language (UML) per i metodi [12].

5.1.1 Generator

L'interfaccia `Generator` presenta due metodi:

`next(): instance_t` ritorna la successiva istanza generata;

`finished(): bool` indica se il generatore si è esaurito.

5.1.2 Counter

L'interfaccia `Counter` presenta un metodo:

`count(instance: instance_t): count_t` ritorna il conteggio dei conflitti relativi all'istanza passata come argomento.

5.1.3 Aggregator

L'interfaccia `Aggregator` presenta due metodi:

`aggregate(count: count_t): void` tiene traccia del conteggio ottenuto da una singola istanza;

`result(): result_t` ritorna il risultato aggregato di tutti i conteggi singoli di cui ha precedentemente tenuto traccia.

5.2 Algoritmi A2 e A3

Come notato all'inizio della Sezione 5.1, gli Algoritmi A2 e A3 sono molto simili tra loro e l'unica differenza è il modo in cui generano il campione di istanze su cui poi operano. In particolare, l'Algoritmo A3 genera un campione di istanze di una data dimensione, mentre l'Algoritmo A2 genera tutte le istanze non partizionate di una data dimensione.

Risulta evidente, quindi, come l'Algoritmo A2 possa essere considerato un caso particolare dell'Algoritmo A3. In particolare, nell'Algoritmo A2, il campione di istanze dell'Algoritmo A3 non è casuale bensì include tutte le istanze di una certa dimensione una e una volta sola.

Per questo motivo, l'Algoritmo A2 chiamerà l'Algoritmo A3, che viene quindi presentato per primo.

5.2.1 Algoritmo A3

La Figura 5.1 riporta il diagramma di sequenza dell'Algoritmo A3. Nella figura, si vede come i componenti della *pipeline* vengano prima costruiti (messaggi da 1 a 3) e successivamente passati all'Algoritmo A3 (messaggio 4) come parametri.

Nel dettaglio, l'Algoritmo A3 riceve come parametri:

`limit`: il numero massimo di istanze nel campione;

`gen`: il generatore che utilizza;

`cnt`: il contatore che utilizza;

`agg`: l'aggregatore che utilizza.

Successivamente, l'algoritmo A3 esegue i seguenti passi:

1. verifica se `limit` è stato raggiunto (in figura è il limite del **loop**);
2. verifica se `gen` è esaurito (in figura è la guardia del **loop**);
3. se il controllo al passo 1 o il controllo al passo 2 sono risultati positivi, allora va al passo 8, altrimenti procede al passo successivo;
4. ottiene un'`instance_t` da `gen` (messaggi 5 e 6);
5. passa l'`instance_t` ottenuta a `cnt` e ottiene un `count_t` (messaggi 7 e 8);
6. passa il `count_t` ad `agg` affinché ne tenga traccia (messaggio 9);
7. torna al passo 1;
8. ottiene un `result_t` da `agg` (messaggi 10 e 11);
9. ritorna il `result_t` ottenuto (messaggio 12).

5.2.2 Algoritmo A2

La Figura 5.2 riporta il diagramma di sequenza dell'Algoritmo A2. In figura, i messaggi da 1 a 3 servono a creare i componenti della *pipeline*, analogamente a quello che accade in Figura 5.1. Successivamente, il messaggio 4 rappresenta l'invocazione, da parte di un generico chiamante, dell'Algoritmo A2 che a sua volta fa subito riferimento all'Algoritmo A3. Infine, col messaggio 5, l'Algoritmo A2 ritorna il risultato ottenuto da A3 al chiamante.

In particolare l'Algoritmo A2 sfrutta il fatto che l'Algoritmo A3 termina quando il `Generator` è esaurito o ha analizzato `limit` istanze.

L'Algoritmo A2 invoca quindi l'Algoritmo A3 con `limit = +inf`, in questo modo l'unica condizione di uscita dell'Algoritmo A3 è che il `Generator` sia esaurito poiché

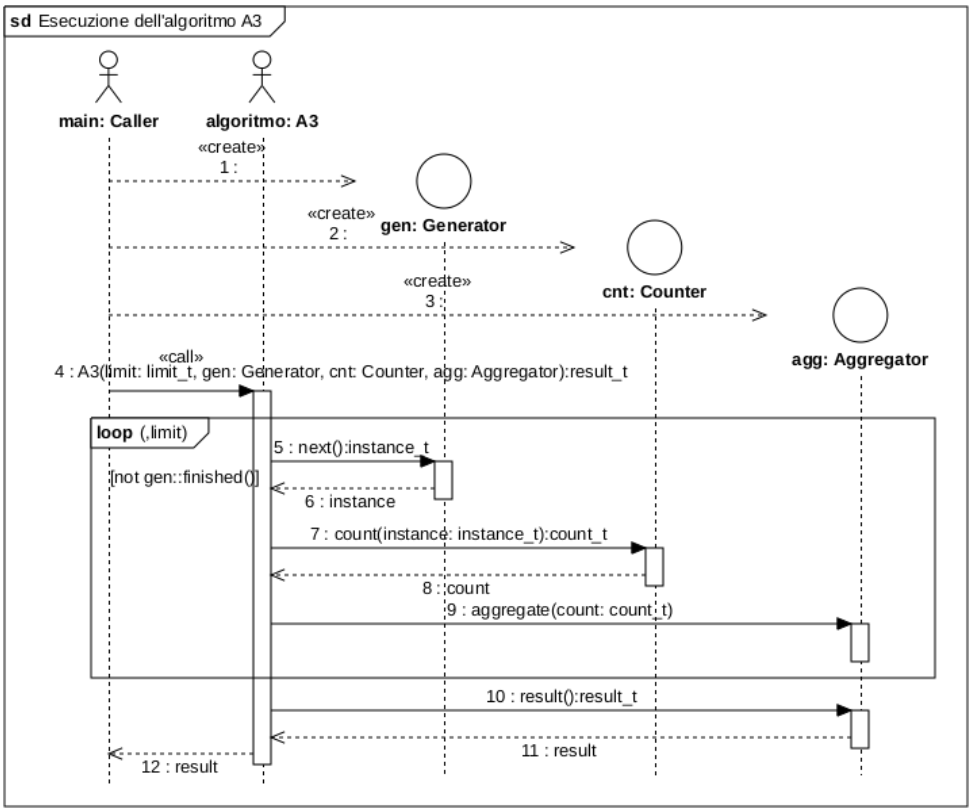


Figura 5.1: Diagramma di sequenza dell'Algoritmo A3.

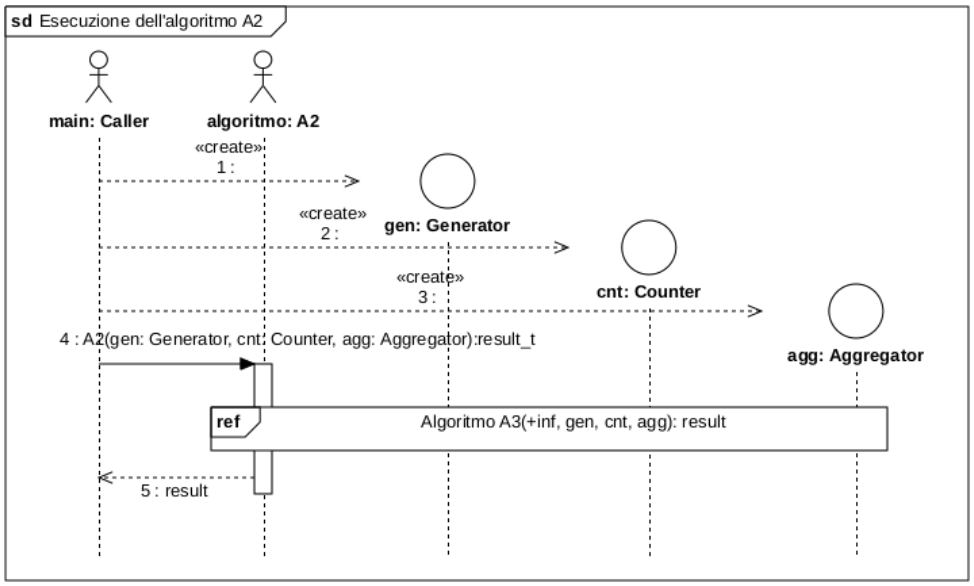


Figura 5.2: Diagramma di sequenza dell'Algoritmo A2.

non è possibile che un algoritmo operi su un numero infinito di istanze. In particolare, la condizione al passo 1 di A3 non verrà mai verificata.

Si noti che è possibile che alcuni generatori producano un numero potenzialmente illimitato di istanze e non si esauriscano mai. In questo caso, l'Algoritmo A2 non terminerà mai. La responsabilità della terminazione è lasciata all'utilizzatore dell'algoritmo.

Capitolo 6

Implementazione

In questa sezione vengono descritte le tecnologie utilizzate per lo sviluppo del software. Successivamente, viene descritta l'implementazione dell'architettura descritta nel Capitolo 5.

6.1 Tecnologie utilizzate

L'implementazione dell'architettura presentata nel Capitolo 5 ha richiesto l'utilizzo di diverse tecnologie relative ai linguaggi di programmazione, ai linguaggi per la stesura dei documenti, all'ambiente di sviluppo (Integrated Development Environment - IDE). Inoltre, per facilitare lo sviluppo, è stato utilizzato un sistema di versionamento. Di seguito vengono riportate le specifiche tecnologie.

6.1.1 Linguaggi

Linguaggi di programmazione. Per lo sviluppo del software è stato utilizzato il linguaggio C++, versione C++11 [14], il cui logo è indicato in Figura 6.1.

Il compilatore utilizzato è g++11 [15] della Gnu Compiler Collection (GCC, vedi Figura 6.2 per il logo).

La scelta del linguaggio C++ è dovuta alla sua potenziale efficienza intrinseca, oltre al fatto che tale linguaggio è stato consigliato dal proponente. Un'alternativa che è stata considerata ma scartata dal proponente è il linguaggio Rust [21], noto per la sua efficienza e per la sua sicurezza a livello di gestione della memoria.

Figura 6.1: Logo C++

Figura 6.2: Logo GCC

Documenti. Per la stesura dei documenti è stato utilizzato il linguaggio LaTeX [18] (vedi Figura 6.3 per il logo). È da sottolineare l'importante ruolo svolto dalla libreria LaTeX pgfplots [20] utilizzata per la creazione dei grafici.

La scelta del linguaggio LaTeX è dovuta al fatto che il linguaggio si presta molto bene alla stesura di documenti scientifici e alla creazione di grafici.

Figura 6.3: Logo LaTeX

Strumenti di build. Per effettuare le build del codice, è stato utilizzato il linguaggio Make. In particolare, è stata usata l'implementazione GNU Make [19].

Script di supporto. Sono stati realizzati i seguenti due script di supporto:

`run.sh` : costruisce il software e lo esegue nella configurazione descritta nella sezione 7.1.

`convert.sh` : converte i dati ottenuti dall'esecuzione del software in un formato leggibile dalla libreria LaTeX pgfplots [20].

Entrambi gli script utilizzano il linguaggio Bash [13] (vedi Figura 6.4 per il logo).

Figura 6.4: Logo Bash

6.1.2 Sistema di versionamento

Il software di versionamento utilizzato è git [16] (vedi Figura 6.5 per il logo).

L'utilizzo di git è avvenuto basandosi sulla piattaforma GitHub [17] (vedi Figura 6.6 per il logo).

Tali scelte derivano da precedenti esperienze e dalla confidenza avuta con questi strumenti.

Figura 6.5: Logo git

Figura 6.6: Logo GitHub

6.1.3 Integrated Development Environment

L'Integrated Development Environment (IDE), utilizzato sia per la scrittura del codice software sia per la stesura dei documenti, è Visual Studio Code [23] (vedi Figura 6.7 per il logo).

Analogamente a quanto detto per il sistema di versionamento, la scelta dell'IDE è legata a ragioni di esperienza e comodità.

Figura 6.7: Logo VS Code

6.2 Implementazione della pipeline

Nella descrizione dell'architettura della Collection pipeline nella Sezione 5.1, i componenti della *pipeline* sono stati descritti in termini di metodi pubblici. Questo suggerisce l'utilizzo di interfacce, tuttavia, queste entità non esistono in C++ e si è perciò dovuto ricorrere a classi astratte con metodi virtuali puri.

Sempre per quanto detto nel Capitolo 5, queste classi astratte sono templatizzate.

In Figura 6.8, viene rappresentato il diagramma delle classi della *pipeline* per come è stato implementato. Si noti che, nel diagramma, gli Algoritmi A2 e A3 sono rappresentati come metodi statici di omonime classi astratte, tuttavia, nel codice essi sono funzioni globali, non metodi. Questo è dovuto a una limitazione del linguaggio UML [22] che non consente la rappresentazione di funzioni globali.

6.3 Tipi e classi di supporto

Di seguito vengono descritti i tipi e le classi di supporto all'implementazione delle componenti concrete della *pipeline*.

Per la descrizione dei metodi pubblici e privati, verrà utilizzata la sintassi del C++ [11].

6.3.1 vehicle_t

Questo tipo è un alias per un intero unsigned di 32 bit.

Rappresenta un veicolo con un intero tra 1 e 2^{32}. Il valore 0 è un valore speciale assegnato al veicolo nullo `nullVehicle`. È possibile, quindi, rappresentare al massimo $2^{32} - 1$ veicoli.

Si ricorda che, come detto nella Sezione 2.1, veicoli e colonne coincidono di fatto nella definizione del problema FQRP-G. Per questo motivo, questo tipo può essere usato anche per rappresentare una colonna.

6.3.2 count_t

Questo tipo è un alias per un intero unsigned a 64 bit.

Viene utilizzato per contare qualcosa, solitamente conflitti. È pensato in modo tale che sia sufficientemente grande per contare tutti i conflitti di un'istanza. Poiché un'istanza può contenere al massimo $2^{32} - 1$ veicoli (si veda la Sezione 6.3.1), il numero massimo di conflitti possibili è $2^{64} - 2^{33} + 1 < 2^{64}$.

6.3.3 Instance

Questa classe rappresenta un'istanza del problema.

Presenta i seguenti metodi:

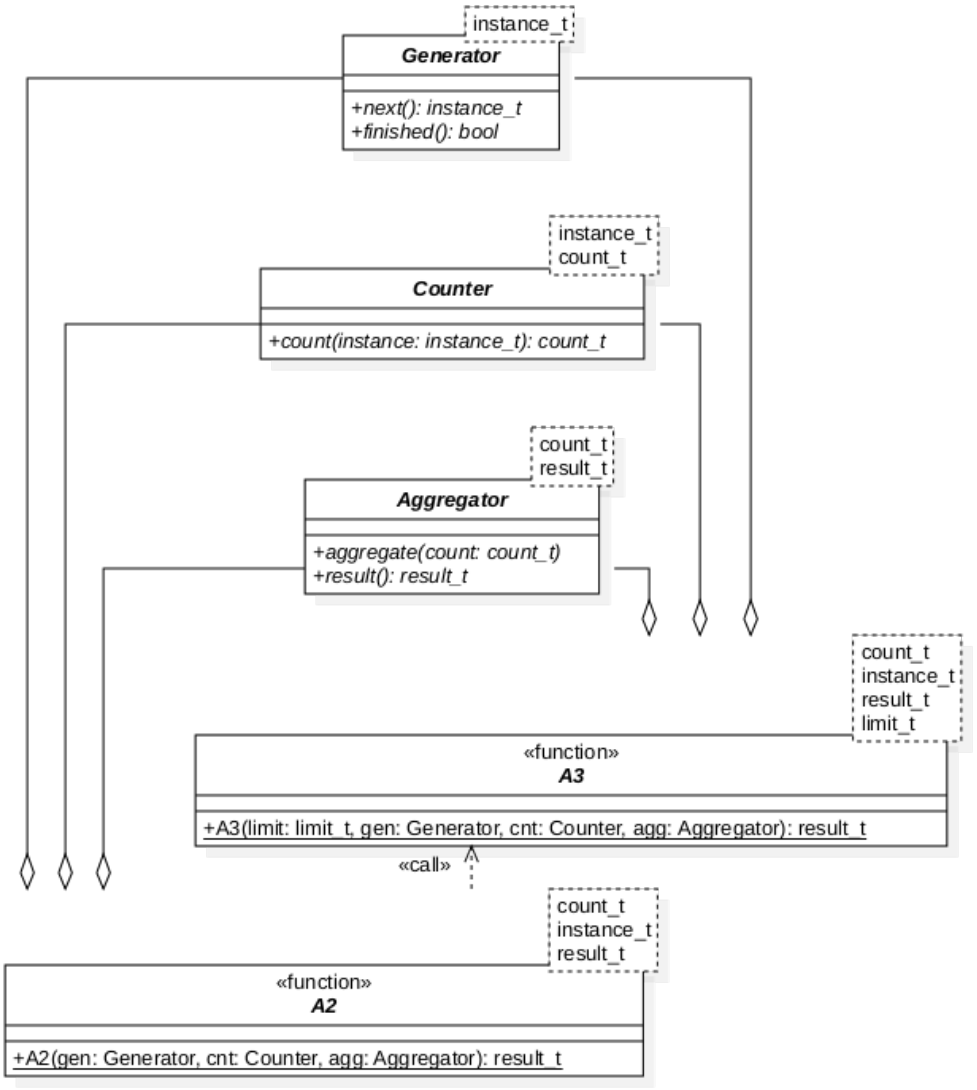


Figura 6.8: Diagramma delle classi astratte coinvolte nella pipeline.

`vehicle_t size() const`: ritorna la dimensione del problema rappresentato;

`vehicle_t sigma(vehicle_t vehicle) const`: ritorna la colonna di destinazione del veicolo `vehicle`;

`explicit operator bool() const`: conversione esplicita al tipo `bool` che verifica se un'istanza è valida o meno.

6.3.4 distribution

Questa classe rappresenta la distribuzione statistica di una variabile discreta, in particolare di `vehicle_t`.

Presenta i seguenti metodi pubblici:

`void add(vehicle_t v, size_t freq = 1)`: registra `freq` volte il campione `v`;

`long double avg() const`: ritorna la media dei campioni registrati ponderata sulla loro frequenza;

`vehicle_t min() const`: ritorna il minimo campione registrato;

`vehicle_t max() const`: ritorna il massimo campione registrato;

`vehicle_t most_freq() const`: ritorna il campione più frequente;

`size_t size() const`: ritorna il numero di campioni registrati;

`long double quantile(double part) const`: ritorna il quantile di ordine `part` dei campioni registrati;

`size_t get(vehicle_t v) const`: ritorna la frequenza del campione `v`.

La classe ha i seguenti membri privati:

`vector<size_t> _freqs`: vettore tale per cui `_freqs[v]` è uguale alla frequenza di `v`;

`size_t _sum`: somma di tutti i campioni registrati;

`size_t sample`: numero di campioni registrati;

`vehicle_t _min`: campione minimo registrato;

`vehicle_t _max`: campione massimo registrato;

`vehicle_t _most_freq`: campione più frequente;

`size_t _highest_freq`: frequenza massima di un campione.

6.4 Implementazione delle classi concrete

Nelle seguenti sottosezioni, verranno presentate varie implementazioni delle componenti della Collection pipeline.

Si noti che nei nomi di alcune variabili legate ai grafi di conflitti di tipo misto rilassato vengono utilizzati i termini `forest` e `tree` nonostante non sia detto che tali grafi siano delle foreste. Questo uso deriva dalla congettura citata nella Sezione 2.4.2, nonostante non sia certo che sia corretta i termini sono stati adottati ugualmente.

6.4.1 Generator

Di seguito verranno presentati quattro tipi di generatori. Il primo è un generatore esaustivo, ovvero che genera tutte le possibili istanze data la dimensione del problema. Il secondo e il terzo producono un numero potenzialmente illimitato di istanze (eventualmente anche ripetute) in modo casuale. L'ultimo, invece, serve ad applicare un filtro ad un altro generatore. Successivamente, nell'ultima sottosezione, vengono presentati alcuni filtri implementati.

La Figura 6.9 riporta il diagramma delle classi dei generatori. Per comodità espositiva, nel grafico sono riportate solo le interfacce pubbliche. I campi e metodi privati sono descritti nelle singole sezioni.

ExhaustiveGenerator

Questo generatore, data la dimensione del problema, genera tutte le possibili istanze, dato il numero di veicoli. Per farlo, deve generare tutte le permutazioni di una certa dimensione. A questo fine, implementa l'algoritmo di Heap [10].

A differenza dell'esempio di implementazione presente in [10] che genera tutte le permutazioni una dopo l'altra, è stato necessario introdurre la possibilità di sospendere l'esecuzione dopo la generazione di ogni permutazione e di poterla riprendere successivamente. Questa necessità è dovuta al fatto che non è possibile produrre tutte le istanze per analizzarle successivamente, ma bisogna analizzarle man mano che vengono generate, per evitare di esaurire rapidamente la memoria disponibile.

Per raggiungere questo obiettivo, ci si è basati sulla versione iterativa dell'algoritmo, disponibile in [10]. Questo algoritmo utilizza le variabili `A`, `c` e `i`. `A` è un array che contiene l'ultima permutazione generata, `c` è un array e `i` un intero che contengono indicazioni sulla permutazione a cui si è arrivati e su come generare le successive. Il loro scopo viene approfondito in [10].

Tutte le variabili interne alla funzione (`A`, `c` e `i`) sono state rese membri privati della classe e per segnalare la creazione di una nuova istanza si è usato un buffer `buf` interno alla classe.

A causa della natura stessa del buffer, è stato necessario introdurre il metodo privato `cacheNext()` che implementa l'algoritmo e popola il buffer se è possibile generare nuove istanze, altrimenti lo lascia vuoto.

Il metodo `next()` dovrà quindi preoccuparsi di svuotare il buffer quando ritorna un'istanza.

La classe ha i seguenti membri privati:

`vector<vehicle_t> perm`: corrisponde alla variabile `A` dell'algoritmo originale;

`vector<vehicle_t> buf`: buffer interno alla classe;

`vector<vehicle_t> c`: corrisponde alla variabile `c` dell'algoritmo originale;

`vehicle_t i`: corrisponde alla variabile `i` dell'algoritmo originale.

La classe implementa il seguente metodo privato:

`vector<vehicle_t> cacheNext()`: se `buf` è vuoto, ci salva la prossima permutazione da ritornare, altrimenti non fa nulla.

RandomGenerator

Questo generatore, data la dimensione del problema, crea istanze casuali in modo tale che ciascuna abbia la stessa probabilità di essere generata. Per farlo, deve generare una permutazione casuale di una certa dimensione. A questo fine, implementa l'algoritmo di Fisher-Yates [8] all'interno del metodo `next()`.

La classe ha i seguenti membri privati:

`vector<vehicle_t> base`: vettore contenente una permutazione valida e che viene utilizzato come permutazione di partenza per l'algoritmo;

`uniform_real_distribution<double> random_real`: rappresenta una distribuzione uniforme in $[0; 1[$;

`random_device source`: fonte di numeri pseudocasuali.

La classe implementa il seguente metodo privato:

`vehicle_t next_random(vehicle_t min, vehicle_t max)`: ritorna un veicolo casuale nell'intervallo $[min; max[$, utilizza i membri privati `random_real` e `source`.

IntervalGenerator

Questo generatore, data la dimensione del problema, crea istanze pseudocasuali non partizionate (vedi Definizione 14). A questo fine, il metodo `next()` implementa l'Algoritmo per la generazione di istanze non partizionate proposto nella Sezione 4.2.7. L'algoritmo è anche detto metodo a intervalli, da cui il nome del generatore.

Il metodo introduce il rischio della possibile presenza di bias che rendano la distribuzione delle istanze non omogenea, come discusso nella Sezione 7.3.3.

La classe ha i seguenti membri privati:

`vehicle_t size`: indica la dimensione del problema;

`uniform_real_distribution<double> random_real`: rappresenta una distribuzione uniforme in $[0; 1[$;

`random_device source`: fonte di numeri pseudocasuali.

La classe implementa i seguenti metodi privati:

`vehicle_t random_in_interval(vehicle_t min, vehicle_t max)`: ritorna un veicolo casuale nell'intervallo $[min; max[$, utilizza i membri privati `random_real` e `source`;

`vehicle_t pop_random(vector<vehicle_t> &vec)`: estrae e ritorna un elemento casuale dal vettore `vec`.

FilteredGenerator

Questo generatore, applica un filtro a un altro generatore. Per farlo, accetta nel costruttore un `Generator` e un `Filter` (vedi Sezione 6.4.1) e sfrutta il generatore per la generazione di istanze che ritorna solo se soddisfano le condizioni determinate dal filtro.

Per far ciò, è stato necessario usare un buffer `buf` interno alla classe. Inoltre la classe deve mantenere i riferimenti al generatore e al filtro ricevuti.

A causa della natura stessa del buffer, è stato necessario introdurre il metodo privato `cacheNext()` che chiede istanze al generatore interno fino a che non è esaurito o viene trovata un'istanza che soddisfa il filtro. A questo punto, l'istanza viene salvata nel buffer `buf`.

Il metodo `next()` dovrà quindi preoccuparsi di svuotare il buffer quando ritorna un'istanza.

Questa classe ha un parametro di tipo `instance_t` che deve essere lo stesso dei parametri `Generator` e `Filter`.

La classe ha i seguenti membri privati:

`Generator<raw_t> &generator`: il generatore interno che produce le istanze;

`Filter<raw_t> &filter`: il filtro che viene applicato;

`vector<vehicle_t> buf`: buffer interno alla classe.

La classe implementa il seguente metodo privato:

`vector<vehicle_t> cacheNext()`: se `buf` è vuoto, ci salva la prossima permutazione da ritornare, altrimenti non fa nulla.

Filter

La classe `Filter` è una classe astratta con un parametro di tipo `instance_t` che indica il tipo di istanza che accetta in input.

Non possiede membri e ha un solo metodo pubblico astratto puro:

`bool operator()(const instance_t &instance)`: ritorna `true` se e solo se l'istanza viene accettata dal filtro.

Di fatto, questo metodo implementa il filtro.

Di seguito, vengono riportate tre implementazioni di `Filter`.

IsPartitioned Accetta solo le istanze partizionate (vedi Definizione 14). Applica l'algoritmo di cui alla Sezione 4.2.6.

ExcludeSymmetric Per ogni coppia di istanze tali che un'istanza è simmetrica all'altra, ne accetta solo una. Per ottenere ciò, confronta $\sigma(1)$ e $\sigma(n)$, dove n è la dimensione del problema.

Not Il template di classe `Not`, a differenza degli altri filtri, non deriva direttamente da `Filter` ma accetta un parametro di tipo `filter_t`, solitamente un filtro, da cui deriva. Il suo `operator()(const instance_t &instance)`, dove `instance_t` è il tipo filtrato da `filter_t`, ritorna la negazione di quello della classe `filter_t`.

6.4.2 Counter

Nel corso del tirocinio è stato implementato un solo contatore, che incapsula l'Algoritmo A1 come descritto nella Sezione 4.3. La corrispondente classe, chiamata `ConflictCounter`, viene descritta di seguito. Successivamente, viene descritta la classe `Logger` che viene utilizzata da `ConflictCounter` per generare il benchmark di cui parla l'Obiettivo D02 (vedi Sezione 1.2.3).

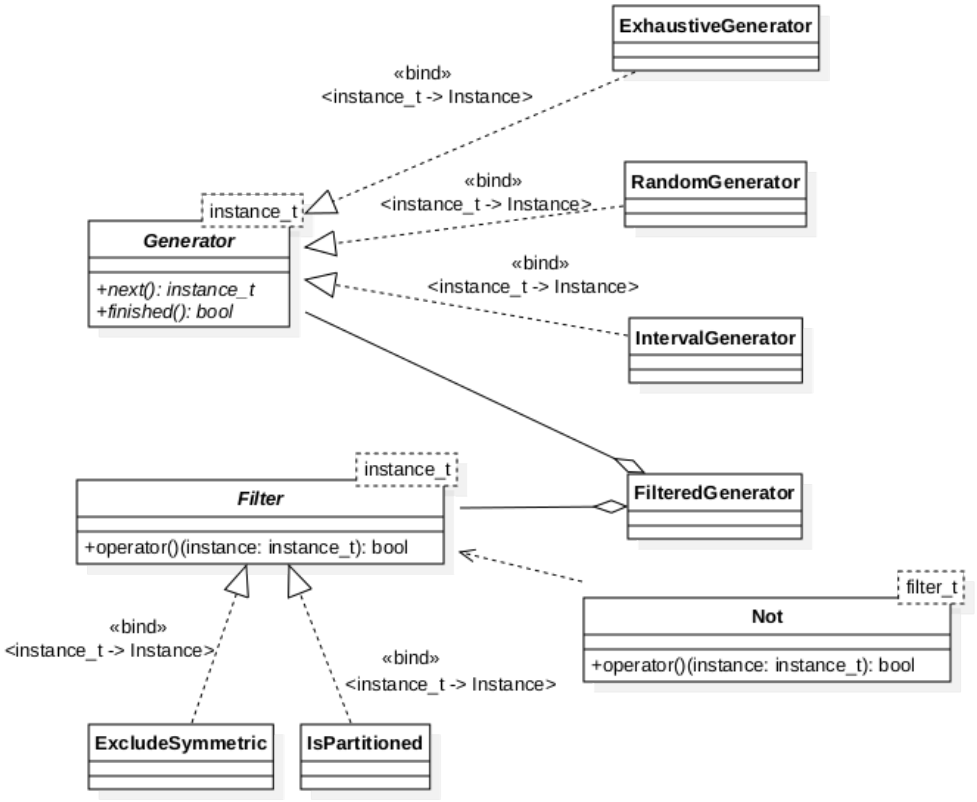


Figura 6.9: Diagramma delle classi che implementano l'interfaccia Generator.

ConflictCounter

Questa classe è l'unica implementazione di `Counter` realizzata all'interno del tirocinio.

Il suo metodo `count` implementa l'Algoritmo A1 per il calcolo del numero di conflitti in una data istanza, con alcune piccole differenze in ciò che ritorna. In particolare, non vengono solamente contati i conflitti di tipo C e di tipo misto, bensì vengono riportate anche informazioni sul Grafo di conflitti di tipo C (vedi Definizione 11) e sul Grafo di conflitti di tipo misto rilassato (vedi Definizione 13).

Il tipo ritornato è la struttura `conflictCount` che possiede i seguenti campi pubblici:

`count_t arcType`: numero dei Conflitto di arco;

`count_t AType`: numero dei Conflitto di tipo A;

`count_t BType`: numero dei Conflitto di tipo B;

`c_graph_info_t c_graph_info`: informazioni sul grafo dei conflitti di tipo C;

`mixed_graph_info_t mixed_forest_info`: informazioni sul grafo dei conflitti di tipo misto rilassato.

L'algoritmo non si limita a contare i conflitti di tipo C, ma costruisce il Grafo di conflitti di tipo C (vedi Definizione 11) e ne restituisce le seguenti proprietà attraverso altrettanti campi pubblici della struttura `c_graph_info_t`:

`size_t arcs_num`: numero di conflitti;

`size_t tree_num`: numero di componenti connesse del grafo;

`size_t chain_num`: numero di catene di conflitti;

`size_t max_length`: lunghezza massima di una catena di conflitti;

`size_t vehicles_num`: numero di veicoli coinvolti.

Per quel che riguarda i conflitti di tipo misto rilassato, l'algoritmo non costruisce completamente il Grafo di conflitti di tipo misto rilassato (vedi Definizione 13) ma ne identifica le componenti connesse e restituisce le seguenti proprietà attraverso altrettanti campi pubblici della struttura `mixed_graph_info_t`:

`bool is_a_forest`: indica se il grafo è una foresta o meno;

`size_t edges_num`: numero di conflitti;

`size_t tree_num`: numero di componenti connesse del grafo;

`size_t max_tree_size`: dimensione massima di una componente connessa;

`size_t nodes_num`: numero di veicoli coinvolti.

Oltre a estrarre le informazioni dall'istanza, la classe `ConflictCounter` permette la creazione di un benchmark attraverso l'uso della classe `Logger`.

La classe viene costruita passandole un vettore di `Logger` che verranno eseguiti al termine dell'algoritmo di conteggio.

La classe ha il seguente membro privato:

`vector<Logger> loggers`: vettore di loggers utilizzati per la creazione del benchmark.

Logger

La classe definisce il tipo `condition_t` come puntatore a una funzione che riceve `(const Instance &, const conflictCount &)` in input e ritorna un `bool`.

La classe `Logger` riceve tramite il costruttore un `condition_t test` e, opzionalmente, un `ostream &os` che di default è lo standard output.

La classe ha i seguenti membri privati:

`condition_t test`: il test che decide se l'istanza va registrata;

`ostream &os`: lo stream in cui vengono registrate le istanze.

La classe implementa il seguente metodo privato:

`void operator()(const Instance &inst, const conflictCount &c)`: quando viene invocata passa `inst` e `c` alla funzione `test` e se questa ritorna `true`, scrive `ins` nello stream `os`.

6.4.3 Aggregator

Nel corso del tirocinio sono stati implementati due tipi di aggregator:

* `AverageAggregator`: tiene traccia di media, minimi e massimi di quasi tutti i valori di `conflictCount` (non considera `is_a_forest`);
* `DistAggregator`: tiene traccia della distribuzione statistica di ogni valore che si trova in un `conflictCount`.

Poiché `AverageAggregator` si è dimostrata una classe più limitata di `DistAggregator` e in grado di restituire solo un sottoinsieme delle informazioni ritornate da quest'ultima, il suo sviluppo è stato interrotto e non verrà perciò presentata.

DistAggregator

La classe `DistAggregator` fa largo uso della classe `distribution`. In particolare, utilizza una `distribution` per ogni valore contenuto in un `conflictCount`.

Ottiene ciò grazie al suo unico membro `dist_count cumulative` i cui membri pubblici sono:

`distribution arcType`: distribuzione del numero di Conflitto di arco;

`distribution AType`: distribuzione del numero di Conflitto di tipo A;

`distribution BType`: distribuzione del numero di Conflitto di tipo B;

`c_graph_dist_t c_graph_dist`: distribuzione delle informazioni sui Conflitto di tipo C;

`forest_dist_t mixed_forest_dist`: distribuzione delle informazioni sui Conflitto di tipo misto rilassato.

La struttura `dist_count` ha anche un metodo pubblico:

`size_t sample_size() const`: ritorna il numero di istanze raccolte.

La struttura `dist_count` implementa il seguente metodo pubblico:

`size_t sample_size() const`: ritorna il numero di istanze raccolte.

La struttura `c_graph_dist_t` ha i seguenti membri pubblici che riguardano i conflitti di tipo C:

`distribution arcs_num`: distribuzione del numero di conflitti;

`distribution tree_num`: distribuzione del numero di componenti connesse del grafo;

`distribution chain_num`: distribuzione del numero di catene di conflitti;

`distribution max_length`: distribuzione della lunghezza massima di una catena di conflitti;

`distribution vehicles_num`: distribuzione del numero di veicoli coinvolti.

La struttura `c_graph_dist_t` implementa il seguente metodo pubblico:

`void add(const c_graph_info_t &v, size_t freq = 1)`: registra `freq` volte le informazioni contenute in `v` per ciascuno dei suoi membri.

La struttura `forest_dist_t` ha i seguenti membri pubblici che riguardano i conflitti di tipo misto rilassato:

`distribution is_forest`: distribuzione di Bernoulli che indica se il grafo è una foresta;

`distribution edges_num`: distribuzione del numero di conflitti;

`distribution tree_num`: distribuzione del numero di componenti connesse del grafo;

`distribution max_tree_size`: distribuzione della dimensione massima di una componente;

`distribution nodes_num`: distribuzione del numero di veicoli coinvolti.

La struttura `forest_dist_t` implementa il seguente metodo pubblico:

`void add(const mixed_graph_info_t &v, size_t freq = 1)`: registra `freq` volte le informazioni contenute in `v` per ciascuno dei suoi membri.

Capitolo 7

Risultati sperimentali e confronto con risultati teorici

In questo capitolo vengono esposti e commentati i risultati sperimentali ottenuti. In particolare, vengono descritte le configurazioni delle pipeline utilizzate, vengono esposti i limiti dei dati e, successivamente, i risultati ottenuti. Infine, verranno descritti i criteri con cui è stato generato il benchmark.

7.1 Configurazione della pipeline

I risultati esposti in questo capitolo sono stati ottenuti con quattro diverse *pipeline*. Di seguito, vengono descritte in termini di componenti usate (generatore, filtri, contatore e aggregatore), dimensione delle istanze analizzate e algoritmo usato.

***Pipeline* PL1: Generatore esaustivo con filtri.** Lo scopo principale di questa *pipeline* è ottenere dei dati esaustivi sulle istanze del problema di dimensioni più piccole. Il vincolo sulla dimensione è dovuto al fatto che, per grandi dimensioni, il numero delle istanze da analizzare sarebbe troppo elevato. Infatti, in tempi ragionevoli rispetto al contesto del tirocinio (500 ore), questa *pipeline* è riuscita a ottenere risultati solo fino alla dimensione 14.

Generatore: `ExhaustiveGenerator`;

Filtri: `ExcludeSymmetric`, `Not<IsPartitioned>`;

Contatore: `ConflictCounter`;

Aggregatore: `DistAggregator`;

Dimensione istanze: da 1 a 15;

Algoritmo: A2.

***Pipeline* PL2: Generatore esaustivo senza filtri.** Lo scopo di questa *pipeline* è di confronto con la *Pipeline* PL1, per verificare gli effetti dei filtri applicati a quest'ultima. Anche in questo caso, a causa del limitato tempo a disposizione, i risultati ottenuti si fermano ai problemi di dimensione 14.

Generatore: `ExhaustiveGenerator`;

Filtri: nessuno;

Contatore: `ConflictCounter`;

Aggregatore: `DistAggregator`;

Dimensione istanze: da 1 a 15;

Algoritmo: A2.

***Pipeline* PL3: Generatore casuale con filtri.** Lo scopo di questa *pipeline* è di ottenere dati su problemi più grandi di quelli analizzati dalle *Pipeline* PL1 e PL2. Per questo motivo implementa l'Algoritmo A3.

In questa *pipeline*, dalla dimensione 16 in poi, sono stati introdotti dei `Logger` per la generazione del benchmark. Il loro utilizzo viene approfondito nella Sezione 7.4.

Generatore: `RandomGenerator`;

Filtri: `Not<IsPartitioned>`;

Contatore: `ConflictCounter`;

Aggregatore: `DistAggregator`;

Dimensione istanze: da 1 a 50;

Algoritmo: A3;

Numero di istanze: 10^8 per ogni dimensione.

***Pipeline* PL4: Generatore a intervalli con filtri.** Questa *pipeline* serve principalmente a verificare se il metodo di generazione a intervalli introduce bias nella distribuzione delle istanze. Tale verifica viene affrontata nella Sezione 7.3.3.

Generatore: `IntervalGenerator`;

Filtri: `Not<IsPartitioned>`;

Contatore: `ConflictCounter`;

Aggregatore: `DistAggregator`;

Dimensione istanze: da 1 a 50;

Algoritmo: A3;

Numero di istanze: 10^8 per ogni dimensione.

7.2 Considerazioni generali

Prima di procedere con l'esposizione dei risultati ottenuti, è bene capire quanto questi siano significativi. In particolare, quanto i risultati ottenuti con le *Pipeline* PL3 e PL4 abbiano rilevanza statistica.

È facile vedere come il numero di possibili istanze di dimensione n sia $n!$, valore che cresce molto in fretta. Al contrario, per ragioni di tempo di calcolo, le dimensioni del campione sono limitate a 10^8. La Tabella 7.1 illustra il rapporto tra campione e tutte le possibili istanze in funzione della dimensione del problema. Si vede chiaramente come già per dimensioni relativamente basse (appena sopra quelle esaminate in modo esaustivo), il campione copre una percentuale estremamente ridotta di casi. È importante considerare ciò quando si valutano i risultati ottenuti.

Inoltre, essendo le istanze generate in modo indipendente una dall'altra, è possibile che alcune vengano ripetute, per quanto la probabilità che ciò accada è ridotta già a partire da dimensioni del problema pari a 15.

n	$n!$	Dimensione del campione	Percentuale coperta
11	$3.99 \cdot 10^7$	$1 \cdot 10^8$	100%
12	$4.79 \cdot 10^8$	$1 \cdot 10^8$	20.88%
13	$6.23 \cdot 10^9$	$1 \cdot 10^8$	1.61%
14	$8.72 \cdot 10^{10}$	$1 \cdot 10^8$	0.11%
15	$1.31 \cdot 10^{12}$	$1 \cdot 10^8$	$7.65 \cdot 10^{-3}$%
20	$2.43 \cdot 10^{18}$	$1 \cdot 10^8$	$4.11 \cdot 10^{-9}$%
25	$1.55 \cdot 10^{25}$	$1 \cdot 10^8$	$6.45 \cdot 10^{-16}$%
30	$2.65 \cdot 10^{32}$	$1 \cdot 10^8$	$3.77 \cdot 10^{-25}$%
40	$8.16 \cdot 10^{47}$	$1 \cdot 10^8$	$1.23 \cdot 10^{-38}$%
50	$3.04 \cdot 10^{64}$	$1 \cdot 10^8$	$3.29 \cdot 10^{-55}$%

Tabella 7.1: Percentuale di copertura del campione rispetto a tutte le istanza possibili in funzione della dimensione del problema.

7.3 Analisi dei risultati ottenuti

In questa sezione vengono riportati e analizzati i risultati sperimentali ottenuti dalle *pipeline* descritte nella Sezione 7.1.

Tra i molteplici risultati ottenuti, vengono qui riportati solo quelli ritenuti più interessanti dal punto di vista di future ricerche. Per ulteriori analisi, si rimanda alla relazione prodotta durante il tirocinio [6]. In particolare, verranno analizzati i dati riguardanti la lunghezza massima delle catene di conflitti di tipo C (Sezione 7.3.1), la frequenza dei conflitti di tipo misto rilassato (Sezione 7.3.2) e la validità del metodo di generazione a intervalli.

7.3.1 Lunghezza massima delle catene di conflitti tipo C

Come riportato nella Sezione 2.6, in [5] viene dimostrato che la lunghezza massima di una catena di conflitti di tipo C in un'istanza di dimensione n è:

$$k_{max} = \begin{cases} \lfloor \frac{n-1}{4} \rfloor + 1, & n \geq 3 \\ 0, & \text{altrimenti} \end{cases}$$

La Figura 7.1 confronta questo risultato teorico con i risultati sperimentali ottenuti rispettivamente dalle *Pipeline* PL1, PL2, PL3 e PL4, descritte nella Sezione 7.1.

In figura, in ciascun grafico, è presenta una linea spezzata rossa, chiamata k_{max}, che indica la lunghezza massima delle catene di conflitti prevista dalla teoria in funzione della dimensione del problema. Subito sotto questa linea, è presente una matrice colorata che rappresenta la distribuzione delle lunghezze massime trovate sperimentalmente.

Se in un'istanza di dimensione n è stata trovata una catena di lunghezza massima k, allora il rettangolo posto all'incrocio della colonna n e della riga k è colorato. Il colore del rettangolo rappresenta qualitativamente la frequenza delle catene di lunghezza massima k in istanze di dimensione n. Nel dettaglio, il colore rappresenta un numero nell'intervallo (0;1]. Tale numero è ottenuto assegnando, per ogni colonna, 1 ai rettangoli che rappresentano la lunghezza più frequente relativamente alla dimensione del problema rappresentata dalla colonna stessa, e agli altri rettangoli un numero che è il rapporto tra la frequenza rappresentata dal singolo rettangolo e la frequenza della lunghezza più frequente. In questo modo, ogni colonna avrà uno o più rettangoli

gialli che indicano le lunghezze più frequenti e il colore degli altri rettangoli indica, qualitativamente, come le lunghezze sono distribuite per quella dimensione del problema, con i colori più scuri associati alle lunghezze meno frequenti.

Come previsto, i dati sperimentali risultano tutti inferiori al limite teorico. In tutti i grafici si può notare come per i valori fino a 25 circa, il massimo sperimentale segue il massimo teorico. Per valori più grandi, quando il campione raccolto diventa molto piccolo rispetto a tutte le possibili istanze del problema, il divario diventa più evidente. Questo è dovuto al fatto che è improbabile che vengano trovate istanze con catene lunghe. Basti osservare che, già a dimensione 25, il campione di 10^8 di istanze usato dalle *Pipeline* PL3 e PL4 copre solamente lo $6 \times 10^{-16}\%$ delle possibili permutazioni, e il fatto che non siano state trovate istanze vicine al massimo teorico non è più sorprendente.

7.3.2 Frequenza dei conflitti di tipo misto rilassato

Visti i risultati teorici in [2], è interessante calcolare la frequenza dei conflitti di tipo misto. Nel dettaglio, è interessante stimare la frequenza con cui si presentano tali conflitti per capire fino a che punto si possa applicare l'algoritmo proposto in [5] e rivisto in [2] che, si ricorda, non è applicabile in presenza di conflitti di tipo misto. Se la frequenza dei conflitti dovesse risultare sufficientemente bassa, si potrebbe applicare l'algoritmo descritto e, solo nei rari casi in cui si verifichino conflitti di questo tipo, applicare algoritmi diversi.

Prima di procedere con l'esposizione dei dati, si ricorda che, come anticipato nella Sezione 2.2.2, i dati esposti riguardano i conflitti di tipo misto rilassato che è un sovrainsieme dei conflitti di tipo misto definiti in [2]. Quindi, bisogna considerare che la probabilità che si verifichi un conflitto di tipo misto (non rilassato) è più bassa di quelle qui riportate.

Nella Figura 7.2, descritta di seguito, vengono riportati solo i risultati delle *Pipeline* PL3 e PL4, poiché le *Pipeline* PL1 e PL2 hanno rilevato solo poche istanze con almeno un conflitto di tipo misto rilassato e il grafico sarebbe illeggibile. Questi risultati sono riportati nella Tabella 7.2, descritta successivamente.

La Figura 7.2 riporta la funzione di ripartizione della probabilità che si verifichino un certo numero di conflitti di tipo misto rilassato. In particolare, data la dimensione $n*$ del problema, la linea spezzata relativa a ciascuno dei valori di n riportati indica, in ordinata, la probabilità che si verifichino fino al numero di conflitti di tipo misto rilassato riportato in ascissa.

In primis, si vuole evidenziare come i due grafici nella Figura 7.2 siano tra loro identici. Questo verifica che, dal punto di vista dei conflitti misti rilassati, il generatore a intervalli (*Pipeline* PL4) non introduce bias osservabili. Inoltre, si noti che nel grafico sono state riportate le funzioni di ripartizione solo per alcune dimensioni, al fine di evitare un sovraffollamento dello stesso.

Il dato più interessante, come detto, è la probabilità che non ci sia alcun conflitto di tipo misto non rilassato. Vediamo che, come è facile aspettarsi, tale probabilità diminuisce all'aumentare di n, la dimensione del problema. In ogni caso, la probabilità che non ci sia alcun conflitto di tipo misto rilassato resta sopra il 90% (pari al 93%) anche con $n = 50$. Di conseguenza, la probabilità che non si verifichi alcun conflitto di tipo misto non rilassato dev'essere ancora maggiore.

Se poi viene considerata la probabilità che ci siano due o più conflitti, questa è nettamente sotto lo 0.1%.

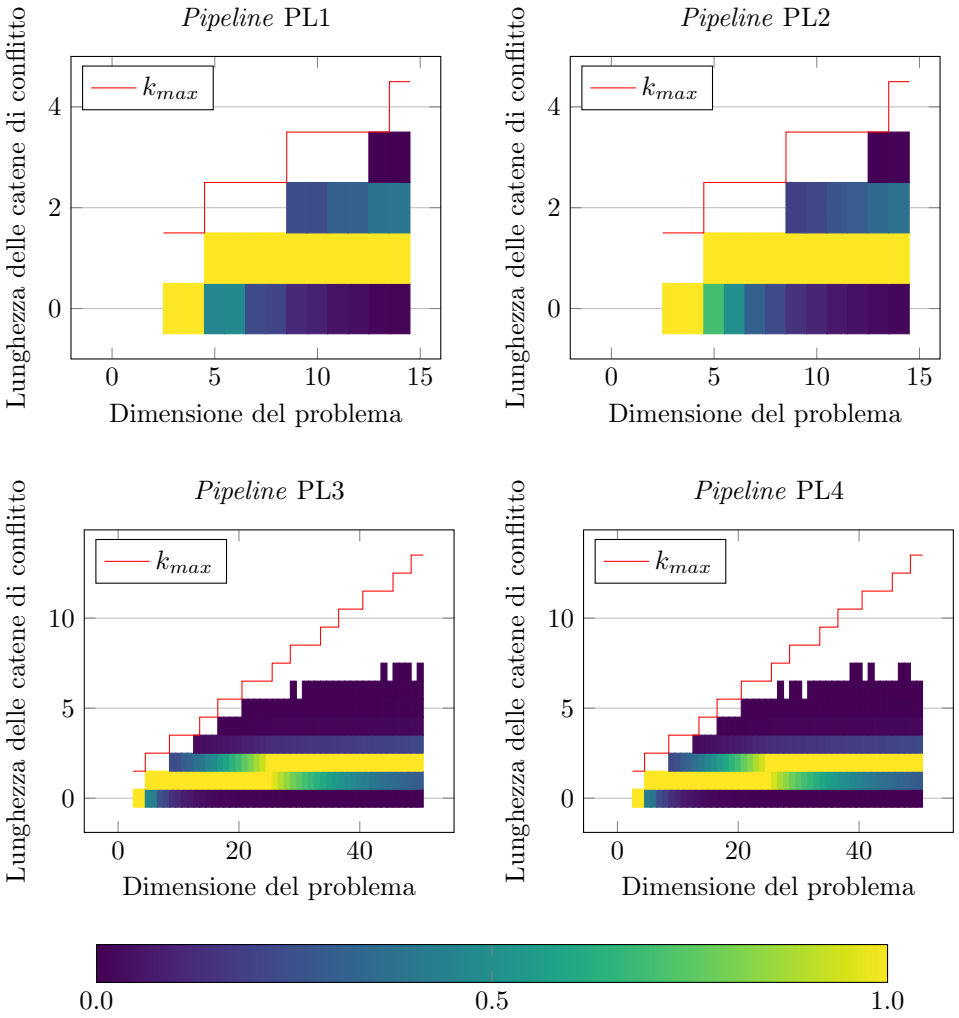


Figura 7.1: Confronto della lunghezza massima teorica delle catene di conflitti di tipo C con i corrispondenti dati sperimentali trovati. Tale confronto viene fatto per ogni pipeline descritta nella Sezione 7.1.

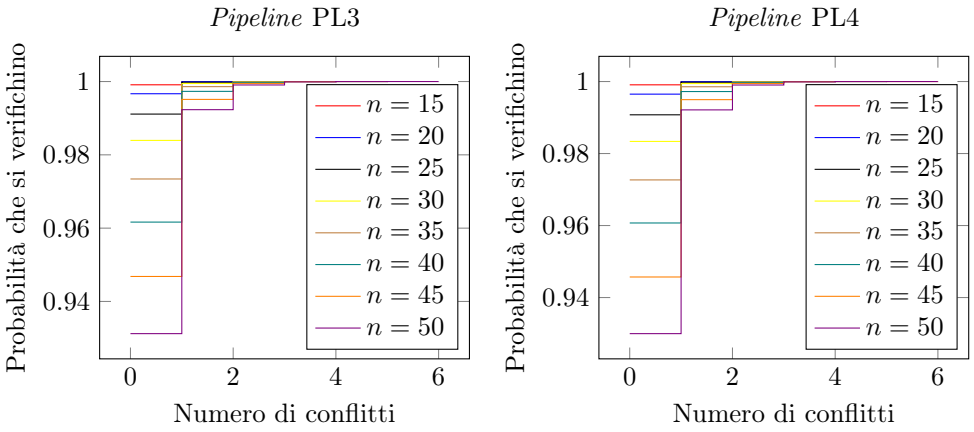


Figura 7.2: Funzione di ripartizione della probabilità che si verifichino fino a un certo numero di conflitti di tipo misto rilassato.

		Numero di conflitti		Probabilità		
Pipeline	dim istanza	0	1	0	1	Campione
PL1	14	43567666776	21478608	99.95%	$4.93 \cdot 10^{-2}$%	43589145384
PL2	14	87145531416	32759568	99.96%	$3.76 \cdot 10^{-2}$%	87178290984

Tabella 7.2: Numero di conflitti di tipo misto rilassato rilevati dalle Pipeline PL1 e PL2.

La Tabella 7.2 è limitata a n=14, il numero minimo di veicoli che ha permesso di osservare uno, ed un solo, conflitto di tipo misto rilassato in una singola istanza.

La tabella riporta per le *Pipeline* PL1 e PL2, nell'ordine, la dimensione dell'istanza, il numero di istanze trovate in cui ci sono 0 e 1 conflitti di tipo misto rilassato, la probabilità che un'istanza abbia 0 o 1 conflitti di tipo misto rilassato e il numero di istanze generato dalla *pipeline*.

Si vuole mettere in evidenza come la probabilità esatta che avvenga un conflitto, per istanze di dimensione 14, è dell'ordine dei decimillesimi. Naturalmente, questa probabilità è più alta per la *Pipeline* PL1, poiché essa esclude tutte le istanze partizionate che, per loro natura, hanno meno conflitti.

7.3.3 Validità del metodo a intervalli per la generazione delle istanze

In questa sezione, si vuole verificare se il metodo a intervalli introduce dei bias statistici nella generazione delle istanze. L'ipotesi è stata introdotta insieme alla definizione del metodo nella Sezione 4.2.7.

Per effettuare il controllo, verranno confrontati alcuni dati prodotti dalla *Pipeline* PL3, la cui distribuzione è certamente omogenea (grazie alle caratteristiche dell'algoritmo di Heap [10]) con quelli analoghi prodotti dalla *Pipeline* PL4.

Si noti che è possibile fare questa verifica perché la *Pipeline* PL3 utilizza dei filtri che rendono il dominio delle sue istanze uguale a quello della *Pipeline* PL4.

Un primo confronto è stato visto nella Figura 7.2 e, come anticipato, l'osservazione dei grafici riportati non suggerisce la presenza di bias, visto che i grafici lì presentati risultano essere identici.

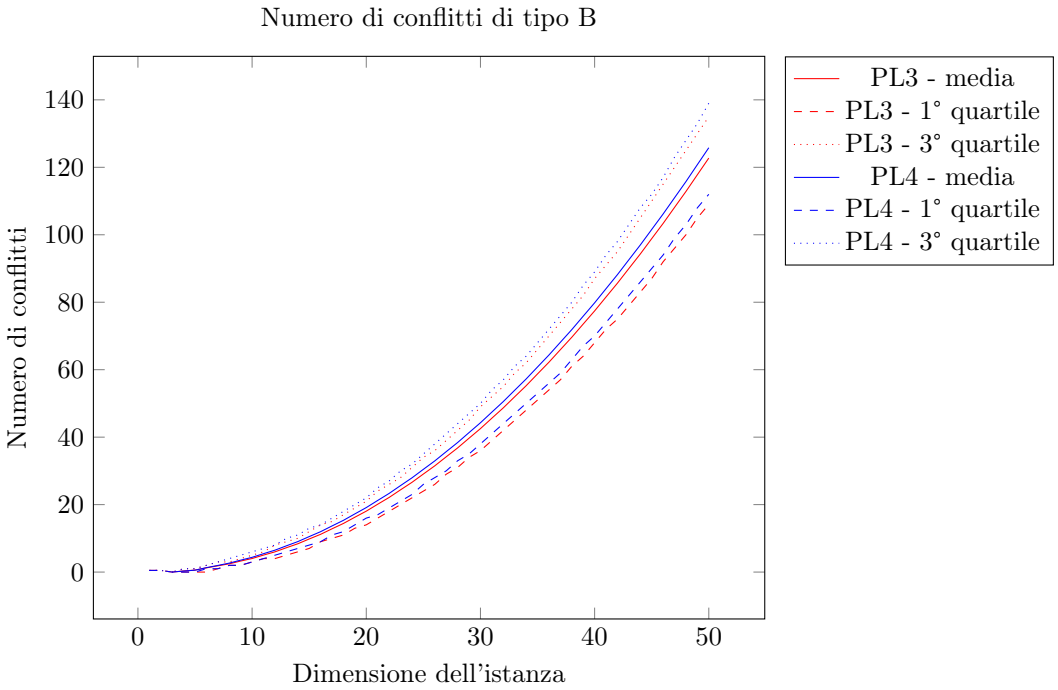


Figura 7.3: Numero di conflitti di tipo B. Confronto di media e quartili tra le Pipeline PL3 e PL4.

Vedremo altri due confronti nelle Figure 7.3 e 7.4.

La Figura 7.3 mette a confronto la distribuzione del numero di conflitti di tipo B trovati utilizzando i due metodi di generazione delle istanze. In particolare, confronta, tra le due *Pipeline* PL3 e PL4, la media e il primo e terzo quartile della distribuzione in funzione della dimensione dell'istanza.

Si può notare che il numero di conflitti trovati col metodo a intervalli è generalmente più alto. Nonostante possa trattarsi di un caso, risulta estremamente improbabile che ciò si sia verificato per tutte le dimensioni delle istanze.

La Figura 7.4 riporta un grafico analogo, solo che riguarda i conflitti di tipo C.

Nonostante la differenza sia impercettibile per la maggior parte delle dimensioni del problema, per alcune di queste è evidente che il metodo a intervalli genera globalmente un maggior numero di conflitti di tipo C.

Possiamo concludere, da questi esempi, che il metodo di generazione a intervalli introduce un bias, nonostante questo sia molto piccolo.

Al momento non è nota con certezza la sua causa. Un'ipotesi è che tale bias sia dovuto al fatto che, al passo 3 dell'algoritmo di generazione a intervalli (vedi Sezione 4.2.7), non venga considerata la possibilità che il veicolo di cui si sta scegliendo la direzione appartenga ad $\mathbb{H}$. Riteniamo necessarie ulteriori ricerche per spiegare il bias.

7.4 Generazione dei benchmark

Uno degli scopi del tirocinio è la generazione di un benchmark di istanze interessanti che potessero poi essere studiate. Per far ciò, sono stati introdotti i `Logger` (vedi

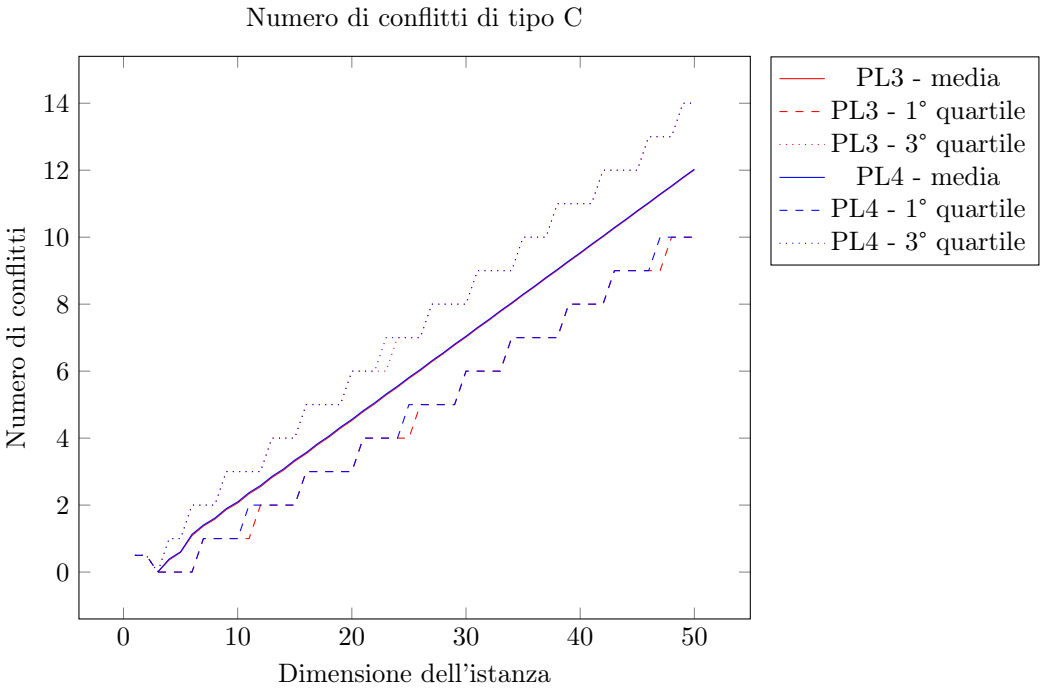


Figura 7.4: Numero di conflitti di tipo C. Confronto di media e quartili tra le Pipeline PL3 e PL4.

Sezione 6.4.2). Nella configurazione della *Pipeline* PL3 (vedi Sezione 7.1) sono stati utilizzati i seguenti tre `Logger`:

`nonMixedTreeLogger`: salva tutte le istanze contenenti grafi di conflitti misti ciclici;

`longCChainLogger`: salva tutte le istanze aventi una catena di conflitti di tipo C lunga almeno il 90% del massimo teorico definito in [4];

`mixedChainLogger`: salva tutte le istanze aventi una componente connessa del grafo dei conflitti di tipo misto rilassato il cui numero di conflitti superi il 50% del numero di conflitti di tipo B presenti nella stessa istanza.

Questi `Logger` sono stati applicati su un campione di 10^8 istanze per ciascuna delle dimensioni da 16 a 50. Il `nonMixedTreeLogger` ha prodotto un benchmark di 53648 istanze; il `longCChainLogger` ha prodotto un benchmark di 361097 istanze e il `mixedChainLogger` ha prodotto un benchmark di 37301 istanze.

Capitolo 8

Conclusioni

In questo capitolo vengono tracciate le conclusioni del tirocinio. Viene effettuato un consuntivo delle attività svolte e degli obiettivi raggiunti, che viene confrontato con quanto preventivato nel Capitolo 1. Successivamente, viene data una valutazione personale sul tirocinio.

8.1 Consuntivo finale

8.1.1 Attività svolte

La Tabella 8.1 riporta le attività previste e, per ciascuna, la durata preventivata, come da Tabella 1.1, e la durata effettiva.

Si può notare che, sebbene il numero totale di ore sia uguale, queste sono ripartite tra le varie attività in modo diverso da quanto preventivato. La differenza più evidente si trova in "Studio e implementazione di possibili parallelizzazioni". Questo è dovuto al fatto che l'Obiettivo D01 è stato sostituito, come descritto nella Sezione 1.2.5. Facciamo notare che, in ogni caso, il precedente obiettivo legato all'implementazione parallela degli algoritmi ha avuto impatto nelle attività svolte, perché parte del tempo impiegato per il progetto e l'implementazione, è comunque stato dedicato ad assicurarsi che sia facile rendere il codice parallelizzabile in eventuali sviluppi futuri.

Un'altra differenza si trova in "Analisi comparative e redazione documenti finali" ed è dovuta, almeno in parte, ad alcune difficoltà avute con la realizzazione dei grafici da inserire nella relazione di fine tirocinio.

Infine, i "Test computazionali dei modelli di programmazione matematica su benchmark" non sono stati svolti poiché erano previsti come ultima attività e non è rimasto tempo per effettuarla. In compenso, la "Definizione di istanze benchmark per FQRP-G" ha richiesto più tempo del previsto, perché è stato difficile trovare dei `Logger` significativi tali che il benchmark generato fosse né troppo grande né troppo piccolo.

Facciamo altresì notare che la progettazione degli Algoritmi A2 e A3 come *pipeline*, ha permesso di risparmiare alcune ore di implementazione rispetto alle previsioni, soprattutto per quel che riguarda l'Algoritmo A2, essendo completamente basato sull'Algoritmo A3.

8.1.2 Raggiungimento degli obiettivi

La Tabella 8.2 riporta gli obiettivi inizialmente proposti e ne viene indicato il grado di raggiungimento.

Descrizione attività	Durata prevista	Durata effettiva
Formazione su argomenti specifici dello stage e FQRP-G	32	36
Formazione su algoritmi su grafo	8	8
Formazione su modelli di programmazione matematica per FQRP-G	8	4
Analisi teorica dei conflitti e calcolo analitico delle frequenze	16	24
Studio dei metodi per la generazione di opportuni campioni casuali di istanze	24	24
Progettazione algoritmi per il calcolo dei conflitti e relative frequenze	72	80
Implementazione Algoritmo A1	10	12
Implementazione Algoritmo A2	10	4
Implementazione Algoritmo A3	20	15
Test preliminari	8	12
Studio e implementazione di possibili parallelizzazioni	32	8
Progettazione e implementazione interfaccia	6	12
Preparazione ed esecuzione test	28	20
Definizione di istanze benchmark per FQRP-G	6	16
Test computazionali dei modelli di programmazione matematica su benchmark	10	0
Analisi comparative e redazione documenti finali	30	45
Totale	320	320

Tabella 8.1: Consuntivo delle ore divise per attività.

L'Obiettivo O02 è da considerarsi parzialmente raggiunto, così come l'Obiettivo F02, poiché non è stato possibile determinare la frequenza dei conflitti in modo analitico ma sono stati ottenuti solo risultati parziali e, insieme al proponente, non è stato ritenuto opportuno continuare con le indagini.

L'Obiettivo F01 non è stato raggiunto in quanto, come esposto nella Sezione 8.1.1, non è stato possibile completare la relativa attività per mancanza di tempo.

Tutti gli altri obiettivi sono stati raggiunti, eccezion fatta per l'Obiettivo D01 che è stato sostituito, come spiegato nella Sezione 1.2.5.

Obiettivo	Descrizione	Stato
O01	Acquisizione di competenze di progettazione e implementazione di algoritmi combinatori	Raggiunto
O02	Analisi preliminare di fattibilità per la determinazione analitica delle frequenze per una o più tipologie di conflitto	Parzialmente raggiunto
O03	Progettazione di algoritmi per il calcolo esatto delle frequenze	Raggiunto
O04	Implementazione di algoritmi per il calcolo esatto delle frequenze	Raggiunto
O05	Progettazione di algoritmi per la stima delle frequenze su base campionaria	Raggiunto
O06	Implementazione di algoritmi per la stima delle frequenze su base campionaria	Raggiunto
D01	Parallelizzazione degli Algoritmi A2 e A3	Sostituito da D03
D02	Generazione di un benchmark di istanze di FQRP-G;	Raggiunto
D03	Verifica dell'esistenza di grafi di conflitti di tipo misto che non sono foreste	Raggiunto
F01	Test computazionali di modelli di programmazione matematica sul benchmark	Non raggiunto
F02	Definizione di funzioni analitiche per il calcolo delle frequenze dei conflitti	Parzialmente raggiunto

Tabella 8.2: Stato di raggiungimento degli obiettivi del tirocinio.

8.2 Valutazione personale

A conclusione del tirocinio, ritengo di esser soddisfatto del lavoro svolto. Ho appreso competenze in ambito di Ricerca Operativa e ho avuto modo di studiare da vicino un problema che impatta il mondo reale. Mi è anche stato possibile mettere in pratica gli studi svolti su algoritmi e strutture dati, ampliando, in alcuni casi, le conoscenze in questi ambiti.

Sono contento della scelta del tirocinio perché mi ha dato modo di interagire con un gruppo di ricerca del Dipartimento che si è mostrato molto disponibile. Questa inclusione mi ha stimolato molto e avrei voluto fare molto di più di quello che ho fatto, completando tutti gli obiettivi. In particolare, mi ha coinvolto molto la parte analitica, come si vede anche dal consuntivo delle ore, e mi sarebbe piaciuto poterla approfondire di più. In ogni caso, i commenti ricevuti durante lo stage e alla fine dello stesso sembrano confermare un contributo del lavoro svolto nell'avanzamento dello studio sul FQRP-G.

Bibliografia

Riferimenti bibliografici

[1] Giovanni Andreatta, Carla De Francesco e Luigi De Giovanni. «Algorithms for Smooth, Safe and Quick Routing on Sensor-Equipped Grid Networks». *Sensors* 21.24 (2021), p. 8188 (cit. a p. 1).

[2] Giovanni Andreatta, Carla De Francesco e Luigi De Giovanni. «Revising a note on solving the fleet quickest routing problem on grid graphs». Working Paper. 2022 (cit. alle pp. 1, 9–11, 14, 29, 52).

[3] Giovanni Andreatta, Luigi De Giovanni e Giorgio Salmaso. «Fleet quickest routing on grids: A polynomial algorithm». *International Journal of Pure and Applied Mathematics* 62.4 (2010), pp. 419–432 (cit. alle pp. 1, 8, 13).

[4] Marisa Cenci, Mirko Di Giacomo e Francesco Mason. «A note on a mixed routing and scheduling problem on a grid graph». *Journal of the Operational Research Society* 68.11 (2017), pp. 1363–1376 (cit. alle pp. 1, 7–11, 13, 56).

[5] Mirko Di Giacomo, Francesco Mason e Marisa Cenci. «A note on solving the Fleet Quickest Routing Problem on a grid graph». *Central European Journal of Operations Research* 28.3 (2020), pp. 1069–1090 (cit. alle pp. 1, 10, 14, 51, 52).

[6] Elia Scandaletti. «Relazione di tirocinio». 2022 (cit. a p. 51).

Siti web consultati

[7] *Expressions - cppreference.com*. cppreference.com. URL: https://en.cppreference.com/w/cpp/language/expressions#Operators (cit. a p. 20).

[8] *Fisher-Yates Algorithm*. Wikipedia. URL: https://en.wikipedia.org/wiki/Fisher-Yates_shuffle (cit. a p. 43).

[9] Martin Fowler. *Collection Pipeline*. URL: https://martinfowler.com/articles/collection-pipeline/ (cit. alle pp. 19, 31).

[10] *Heap's Algorithm*. Wikipedia. URL: https://en.wikipedia.org/wiki/Heap's_algorithm (cit. alle pp. 42, 54).

[11] *Non-static member functions - cppreference.com*. cppreference.com. URL: https://en.cppreference.com/w/cpp/language/member_functions (cit. a p. 39).

[12] *UML Operation*. uml-diagrams.org. URL: https://www.uml-diagrams.org/operation.html#method (cit. a p. 32).

Strumenti utilizzati

[13] *bash*. GNU. URL: https://www.gnu.org/software/bash/ (cit. a p. 38).

[14] *C++11*. URL: https://isocpp.org/ (cit. a p. 37).

[15] *GCC*. GNU. URL: https://www.gnu.org/software/gcc/ (cit. a p. 37).

[16] *git*. URL: https://git-scm.com/ (cit. a p. 38).

[17] *GitHub*. GitHub. URL: https://github.com/ (cit. a p. 38).

[18] *LaTeX*. URL: https://www.latex-project.org/ (cit. a p. 37).

[19] *Make*. GNU. URL: https://www.gnu.org/software/make/ (cit. a p. 38).

[20] *pgfplots*. URL: https://ctan.org/pkg/pgfplots/ (cit. alle pp. 37, 38).

[21] *Rust*. URL: https://www.rust-lang.org/ (cit. a p. 37).

[22] *Unified Modeling Language (UML)*. OMG. URL: https://www.uml.org/ (cit. a p. 39).

[23] *Visual Studio Code*. Microsoft. URL: https://code.visualstudio.com/ (cit. a p. 38).