Università degli Studi di Udine
Facoltà di Scienze Matematiche Fisiche e Naturali
Corso di Laurea Triennale in Informatica

Tesi di Laurea

Un applicativo per la generazione automatica di istanze di basi di dati

Candidato:
Daniele Canciani

Relatore:
dott. Nicola Vitacolonna

Anno Accademico 2011/2012

Università degli Studi di Udine
Via delle Scienze, 206
33100 Udine
Italia

Alla mia famiglia per avermi supportato.

Ringraziamenti

Vorrei ringraziare la mia famiglia che mi ha supportato in tutti questi anni di studi, il mio relatore dott. Nicola Vitacolonna per avermi assistito con le sue competenze durante la stesura della tesi e i miei colleghi per avermi supportato e aiutato nei momenti di bisogno.

Indice

Elenco degli algoritmi

Elenco delle figure

Introduzione

Una parte importante nel ciclo di sviluppo di un sistema software è ricoperta dal testing [1, Cap. 4]; infatti ogni ciclo di sviluppo del software include, in uno o più passaggi, il concetto di testing.

Il testing ha lo scopo di trovare e risolvere il maggior numero possibile di errori che si possono commettere in fase di progettazione e realizzazione di un sistema software; fermo restando che non è possibile trovare tutti gli errori ma solo dimostrarne la presenza [2, pag. 16], è quindi necessario scegliere con attenzione i test da effettuare in modo da coprire la maggioranza delle casistiche.

Nel testing delle basi di dati, per verificare la bontà delle interrogazioni, bisogna creare delle istanze ad-hoc: operazione che manualmente può richiedere molto tempo perché a una base di dati è sempre associato un insieme di vincoli d'integrità, spesso molto complessi, che qualunque istanza valida deve rispettare. Inoltre, secondo alcune stime [1], i costi di un progetto software possono, a seconda del ciclo di sviluppo utilizzato, arrivare anche al 50% del costo dell'intero progetto. Per evitare un incremento dei costi si rende quindi necessaria l'automatizzazione di alcune fasi del test: una di queste può essere la creazione di istanze di basi di dati da usare nei test stessi.

L'obiettivo di questa tesi è studiare e realizzare un applicativo per la generazione automatica di istanze di basi di dati. Applicativi del genere esistono, ma in generale non sempre riescono a gestire in maniera automatica nemmeno i vincoli più comuni che si possono trovare nelle basi di dati; in particolare i vincoli di integrità referenziale non sono supportati da tali software.

Tale applicativo quindi, una volta impostati alcuni dati per la connessione alla base di dati, genererà automaticamente un'istanza della stessa con dei dati pseudo-casuali; sarà quindi compito dell'addetto alla fase di test eseguire il sistema software per trovare tutti i possibili errori o in generale problemi (come le prestazioni) nell'esecuzione delle interrogazioni.

La presente tesi è composta da cinque capitoli:

- nel primo verranno richiamati alcuni dei concetti fondamentali del modello relazionale, del linguaggio SQL, dell'Information Schema e della teoria del

calcolo delle probabilità;

- nel secondo verrà invece esposto il metodo lineare congruenziale, uno dei primi metodi per la generazione di numeri pseudo-casuali, seguito da alcuni importanti risultati teorici;
- nel terzo saranno descritti gli algoritmi principali per il campionamento casuale di numeri;
- nel quarto sarà discussa la trattazione dei vincoli di una base di dati e verrà presentato l'applicativo oggetto della tesi;
- nel quinto e ultimo capitolo si esporranno una serie di risultati provenienti dalle sperimentazioni sugli algoritmi e sull'applicativo.

Contributi originali dell'autore sono contenuti nei Capitoli 4 in cui sono descritte le implementazioni degli algoritmi presentati nel Capitolo 3 e l'applicativo oggetto della tesi e nel Capitolo 5 dove sono state studiate alcune caratteristiche degli algoritmi e dell'applicativo nel suo insieme.

Particolare attenzione verrà riservata al campionamento di numeri pseudo-casuali perché, come vedremo, presenta problematiche che sono accentuate nel caso in cui si voglia generare dati senza ripetizioni e alla gestione dei vincoli di una base di dati perché, come vedremo, presentano problematiche di non banale risoluzione.

1
Preliminari

Questo capitolo iniziale presenterà alcuni dei concetti fondamentali che verranno discussi nel seguito.

1.1 Basi di dati

La basi di dati tradizionali utilizzano il modello relazionale, nel quale i dati sono rappresentati come relazioni. Presenteremo ora le principali caratteristiche del modello relazionale rimandando per gli approfondimenti a [3].

Una relazione può essere vista informalmente come una tabella contenente dei dati organizzati in colonne; tutti i valori della colonna devono avere lo stesso tipo.

Definizione 1.1 (**Schema di relazione** e **attributi**)**.** Uno **schema di relazione** R, indicato con $R(A_1, \ldots, A_n)$ è costituito da un nome di relazione R e da un elenco di attributi $A_1, \ldots, A_n$. Ogni **attributo** A_i possiede un proprio **dominio**.

Definizione 1.2 (**Tupla**)**.** Una **tupla**, o n-upla, è una funzione che associa a ciascun elemento dell'insieme di attributi $A_1, \ldots, A_n$ dei valori $t_1, \ldots, t_n$. Un elemento della tupla può contenere un valore appartenente al proprio dominio, che è l'insieme dei valori che possono essere assunti da A_i.

Definizione 1.3 (**Istanza di relazione**)**.** Un'**istanza di relazione** è l'insieme delle tuple che compaiono nello schema di relazione.

Definizione 1.4 (**Grado**)**.** Il **grado** di una relazione è il numero di attributi presenti nello schema di relazione.

Esempio 1. Una base di dati per la gestione delle carriere degli studenti avrà presumibilmente il seguente schema di relazione:

STUDENTE(nome, cognome, matricola, indirizzo, telefono)

che contiene gli attributi che contraddistinguono lo studente e ha grado 5. Una sua possibile istanza è:

STUDENTE				
nome	cognome	matricola	indirizzo	telefono
Mario	Rossi	27384	via Giuseppe Verdi 2	234234
Luigi	Bianchi	23749	via Gabriele d'Annunzio 34	456456
Giuseppe	Russo	34759	piazza dell'Unità d'Italia 56	894805

Finora abbiamo descritto solo una singola relazione, ma in una base di dati ci sono molte relazioni e le tuple in esse contenute sono legate in vari modi. Comunemente ci sono molte restrizioni, dette **vincoli d'integrità**, che limitano i valori ammissibili della base di dati. Questi vincoli possono essere suddivisi in tre categorie:

- **vincoli intrinseci al modello dei dati**;
- **vincoli basati sullo schema** esprimibili direttamente negli schemi;
- **vincoli di integrità semantici o regole aziendali** che non possono essere espressi direttamente nello schema e che vanno quindi specificati e realizzati dai programmi applicativi.

I vincoli del primo tipo si vengono a creare nella fase di creazione dello schema e rappresentano, ad esempio il dominio e formato degli attributi, il fatto che una tupla non possa occorrere più volte in una tabella e così via. I vincoli del secondo tipo, quelli che andremo a specificare ora, sono quelli che si possono specificare sullo schema. L'ultimo tipo di vincoli invece è troppo complesso da gestire sullo schema e quindi viene gestito dall'applicativo che opera sulla base di dati.

Come abbiamo visto una relazione è definita come un insieme di tuple, come per gli elementi di un insieme anche le tuple devono essere distinte. Questo impone che una relazione non possa avere due tuple identiche, cioè che coincidono sui valori di tutti gli attributi. Questo vincolo prende il nome di **superchiave**. Più precisamente una superchiave è un qualunque insieme di attributi X tali che nessuna istanza corretta dello schema contiene due tuple che coincidono su X. Ogni relazione contiene almeno una superchiave, che è banalmente l'intero insieme degli attributi. In generale una relazione può avere molte superchiavi.

Generalmente, tuttavia, conviene fare riferimento a un concetto più forte, quello di **chiave**, definito nel seguente modo:

- due tuple distinte non posso avere valori uguali su tutti gli attributi della chiave (**vincolo di unicità**);

- non è possibile togliere dalla chiave alcun attributo mantenendo valido il vincolo di unicità (**minimalità**).

In base alle considerazioni precedentemente fatte, è chiaro che ogni schema di relazione ha almeno una chiave e, in generale, più insiemi di attributi possono avere le caratteristiche di una chiave. Ogni chiave prende il nome di **chiave candidata**. Convenzionalmente una di queste è scelta dal progettista e assurge al ruolo di **chiave primaria** della relazione.

Esiste anche un secondo tipo di vincolo che può coinvolgere uno o più attributi, ed è quello di **chiave esterna** (o **vincolo di integrità referenziale**). Questo tipo di vincolo viene utilizzato per mantenere coerenti le tuple di due relazioni diverse, o i valori di attributi diversi in una stessa relazione, e impone che i valori degli attributi della **relazione referenziante** siano presenti nella **relazione riferita**.

Esempio 2. Consideriamo il seguente schema di relazione composto dalle due relazioni:

- IMPIEGATO(nome, cognome, stipendio, codice fiscale, dipartimento)

- DIPARTIMENTO(nome, numero, manager)

Le due relazioni avranno presumibilmente una chiave primaria rispettivamente su *codice fiscale* e su *numero*. Inoltre la relazione *DIPARTIMENTO* avrà un vincolo di integrità referenziale sull'attributo *manager* che si riferisce all'attributo *codice fiscale* della relazione *IMPIEGATO*; questo perché assumiamo che il manager sia a sua volta un impiegato. Stesso vale per il vincolo di integrità referenziale che sussiste fra l'attributo *dipartimento* della relazione *IMPIEGATO* e l'attributo *nome* della relazione *DIPARTIMENTO*: un impiegato può afferire solo ad un dipartimento valido (cioè quelli contenuti nella relazione *DIPARTIMENTO*).

Le nozioni di (schema di) tupla e relazione, assieme ai vincoli d'integrità, definiscono la parte strutturale del modello relazionale. La definizione del modello è completata da un insieme di operatori per la manipolazione di relazioni. Nel modello relazionale è possibile effettuare le seguenti operazioni sulle relazioni:

- operazione di **proiezione**, $\pi_X(R)$: permette di selezionare dalla relazione R solo gli attributi X;

- operazione di **selezione**, $\sigma_\alpha(R)$: permette di selezionare dalla relazione R solo le tuple che soddisfano l'espressione booleana α;

- operazione di **join**, $R \bowtie_\alpha S$: permette di unire le tuple delle relazioni R ed S sulla base della condizione booleana α;
- operazione di **prodotto cartesiano**, $\times$: permette di unire tutte le tuple delle relazioni R ed S senza considerare alcuna condizione.

Tutte le operazioni sono applicate a istanze di relazioni e producono una nuova istanza di relazione; si possono quindi concatenare a piacere per ottenere "interrogazioni" di complessità arbitraria.

Esempio 3. Considerando lo schema di relazione dell'esempio precedente, si supponga di voler trovare:

- il nome e cognome degli impiegati (senza lo stipendio):

$$\pi_{nome,cognome}(IMPIEGATO)$$

- il nome e cognome degli impiegati il cui stipendio è maggiore di 30000 €:

$$\sigma_{stipendio>30000}(IMPIEGATO)$$

- il nome e cognome degli impiegati che afferiscono ad un dato dipartimento:

$$\pi_{nome,cognome}(IMPIEGATO \bowtie_{dipartimento=nomeDip} DIPARTIMENTO)$$

- il nome e cognome degli impiegati che afferiscono ad un dato dipartimento:

$$\pi_{nome,cognome}(\sigma_{dipartimento=nomeDip}(IMPIEGATO \times DIPARTIMENTO))$$

1.2 SQL

SQL è un linguaggio per interrogare e gestire basi di dati. Si tratta, cioè, allo stesso tempo di un **linguaggio di definizione dei dati** (o **DDL**, **Data Definition Language**), basato su operazioni che permettono ad esempio la creazione, modificazione, e distruzione di tabelle, e di un **linguaggio di manipolazione dei dati** (o **DML**, **Data Manipulation Language**), basato sull'uso di costrutti chiamati **query** (o **interrogazioni**). La maggior parte dei DBMS (ovvero dei sistemi software per la gestione di una base di dati) implementano una versione dello standard SQL; implementazioni che non sempre aderiscono completamente agli standard.

Presenteremo ora le principali caratteristiche di questo linguaggio rimandando per gli approfondimenti a [3]. Ci limiteremo a considerare soltanto i costrutti usati nell'implementazione descritta nel Capitolo 4.

Introduciamo ora un semplice schema di base di dati che utilizzeremo in tutti gli esempi di questa sezione formato dalle seguenti tabelle:

- IMPIEGATO: con attributi nome, cognome, **codice fiscale**, stipendio e *dipartimento*;
- DIPARTIMENTO: con attributi **numero**, nomeDip e *manager*.

in cui in grassetto sono riportate le chiavi primarie e in corsivo le chiavi esterne. Una possibile istanza di questo schema è:

IMPIEGATO				
nome	cognome	codice fiscale	stipendio	dipartimento
Mario	Rossi	MRI...	40000	1
Luigi	Bianchi	LGI...	30000	1
Giuseppe	Russo	GSP...	50000	2

DIPARTIMENTO		
numero	nome	manager
1	amministrazione	GSP...
2	presidenza	GSP...

Introduciamo anche una notazione per gli schemi e i vincoli che tornerà utile negli esempi del Capitolo 4:

- le relazioni sono denotate con $R(A_1, \ldots, A_n)$ dove R è il nome della tabella e i vari A_i sono i suoi attributi;
- il vincolo di chiave primaria è indicato con $PK : (A_1, \ldots, A_n)$ dove i vari A_i sono gli attributi che compongono la chiave;
- un vincolo di unicità è indicato con $UNI : (A_1, \ldots, A_n)$ dove i vari A_i sono gli attributi coinvolti;
- un vincolo di chiave esterna è indicato con $CE : (A_1, \ldots, A_n) \rightarrow S(B_1, \ldots, B_n)$ dove ciascun A_i si riferisce al corrispondente B_i della tabella S.

In SQL un'interrogazione è formata da tre blocchi principali detti **clausole**:

- **SELECT**: specifica gli attributi (o le funzioni) che verranno restituite dall'interrogazione, l'operatore "$*$" indica che si vuole che vengano restituiti tutti gli attributi;
- **FROM**: specifica le relazioni a cui si vuole accedere;
- **WHERE**: specifica la condizione booleana che andrà a selezionare solo alcune tuple dalle relazioni contenute nella clausola *FROM*.

a cui si possono apporre altre tre *clausole* facoltative:

- **GROUP BY**: consente di raggruppare le tuple sulla base di una condizione;
- **HAVING**: permette di selezionare solo alcune delle partizioni ottenute da una *GROUP BY*;
- **ORDER BY**: permette di ordinare le tuple del risultato sulla base dei valori di uno o più attributi.

Osservazione 1. Delle clausole presentate solo la *SELECT* e la *FROM* sono obbligatorie, mentre tutte le altre sono facoltative. Nella clausola *SELECT* è possibile usare la parola chiave *DISTINCT* per eliminare i duplicati dal risultato.

Vediamo ora due semplici interrogazioni:

Esempio 4. Recuperare il nome e cognome degli impiegati il cui stipendio è maggiore di 30000€:

```
SELECT nome, cognome
FROM IMPIEGATO
WHERE stipendio > '30000'
```

Il risultato dell'interrogazione eseguita sull'istanza presentata sopra è:

nome	cognome
Mario	Rossi
Giuseppe	Russo

Esempio 5. Determinare le fasce di stipendio degli impiegati:

```
SELECT DISTINCT stipendio
FROM IMPIEGATO
```

Il risultato dell'interrogazione eseguita sull'istanza presentata sopra è:

stipendio
30000
40000
50000

Per fondere più tabelle assieme sono disponibili due approcci:

- utilizzare il prodotto cartesiano e poi nella clausola *WHERE* inserire una condizione, detta **condizione di join**, che andrà a selezionare solo le tuple richieste;
- utilizzare uno dei costrutti espliciti per il **join** fra tabelle come il *JOIN* o il *NATURAL JOIN* in cui la **condizione di join** viene specificata nella clausola *FROM*.

Vediamo due interrogazioni che utilizzano rispettivamente il prodotto cartesiano e l'operatore di *JOIN*:

Esempio 6. Recuperare il nome e cognome degli impiegati che afferiscono ad un dato dipartimento:

```
SELECT nome, cognome
FROM IMPIEGATO, DIPARTIMENTO
WHERE dipartimento = numero AND nomeDip = 'nome_dipartimento'
```

Il risultato dell'interrogazione eseguita sull'istanza presentata sopra è (supponendo di voler richiedere i dati del dipartimento "amministrazione"):

nome	cognome
Mario	Rossi
Luigi	Bianchi

Esempio 7. Recuperare il nome e cognome degli impiegati che afferiscono ad un dato dipartimento:

```
SELECT NOME, COGNOME
FROM  (IMPIEGATO JOIN DIPARTIMENTO ON DIP = DNUMERO)
WHERE DIP = 'nome_dipartimento'
```

Il risultato dell'interrogazione eseguita sull'istanza presentata sopra è (supponendo di voler richiedere i dati del dipartimento "amministrazione"):

nome	cognome
Mario	Rossi
Luigi	Bianchi

SQL mette a disposizione i seguenti meccanismi per la definizione dei dati:

- il dominio dei dati che può essere standard o definito dell'utente;
- definizione e modifica dello schema e delle tabelle;
- definizione dei vincoli;

Andremo ora a trattare in dettaglio la definizione dei vincoli, per il resto si rimanda a [3]; in SQL si possono definire i seguenti vincoli:

- **default** cioè il valore che l'attributo assumerà qualora non sia stato impostato un valore;
- **vincoli intrarelazionali**, cioè quei vincoli che coinvolgono solo la tabella in cui sono stati definiti;
- **vincolo interrelazionali**, cioè quei vincoli che coinvolgono più tabelle.

I vincoli intrarelazionali si dividono in:

- **PRIMARY KEY**: la chiave primaria, ve ne può essere una sola;
- **NOT NULL**: specifica che un dato insieme di attributi non ammette valori nulli;
- **UNIQUE**: specifica che un dato insieme di attributi non può contenere valori duplicati.

I vincoli interrelazionali si dividono in due categorie:

- **FOREIGN KEY**: applicabile a un attributo oppure ad un insieme di attributi, rappresenta il vincolo di chiave esterna;
- **CREATE ASSERTION**: permette di creare vincoli arbitrariamente complessi; la maggior parte dei DBMS attuali però non li implementano.

Per specificare vincoli interrelazionali di tipo arbitrariamente complessi si possono usare i **trigger**, che sono delle procedure che vengono attivate in maniera automatica al verificarsi di un dato evento. I trigger sono tipicamente scritti in un linguaggio proprietario del DBMS e vengono salvati all'interno di esso.

1.3 Information Schema

L'**Information Schema** consiste in una serie di viste (una sorta di tabelle virtuali) che contengono informazioni sugli oggetti definiti all'interno del database. Alcuni DBMS mantengono inoltre dei propri **cataloghi di sistema** che assolvono alle stesse funzioni. La principale differenza fra l'Information Schema e i cataloghi di sistema è che i primi sono definiti dallo standard SQL mentre i secondi sono definiti dal progettista del DBMS.

Nell'Information Schema, quanto nei cataloghi di sistema, si possono trovare diverse viste che permettono, ad esempio, di:

- ottenere informazioni sulle tabelle quali il loro nome e i vincoli che sono stati inseriti;
- ottenere le informazioni sugli attributi come il loro nome e dominio;
- ottenere informazioni sui vincoli.

Queste viste vengono modificate unicamente dal DBMS e sono quindi accessibili in sola lettura dagli utenti della base di dati.

Elenchiamo le principali viste dell'Information Schema utilizzate nelle interrogazioni della Sezione 4.3.6.

- TABLES: contiene tutte le tabelle e le viste definite nella base di dati; gli attributi principali sono:
  - table_name: il nome;
  - table_schema: il nome dello schema che contiene la tabella;
  - table_type: indica il tipo della tabella.
- TABLE_CONSTRAINTS: contiene tutti i vincoli che sono associati alle tabelle della base di dati; gli attributi principali sono:
  - table_name: il nome della tabella;
  - table_schema: il nome dello schema che contiene la tabella;
  - constraint_type: indica il tipo di vincolo, può assumere i seguenti valori: *CHECK*, *FOREIGN KEY*, *PRIMARY KEY*, oppure *UNIQUE*;
  - constraint_name: il nome del vincolo.

- KEY_COLUMN_USAGE: contiene le colonne a cui sono sottoposti dei vincoli di unicità, di chiave esterna o di integrità referenziale; gli attributi principali sono:
  - table_name: il nome della tabella;
  - table_schema: il nome dello schema che contiene la tabella;
  - column_name: il nome della colonna a cui è sottoposto il vincolo;
  - constraint_name: il nome del vincolo;
  - ordinal_position: la posizione della colonna all'interno della tabella;
  - position_in_unique_constraint: la posizione della colonna riferita dal vincolo di chiave esterna nella tabella riferita.
- COLUMNS: contiene le informazioni su tutte le colonne delle tabelle e delle viste ad esclusione delle colonne dei cataloghi di sistema; gli attributi principali sono:
  - table_name: il nome della tabella;
  - table_schema: il nome dello schema che contiene la tabella;
  - column_name: il nome della colonna;
  - data_type: il tipo di dato della colonna;
  - column_default: indica l'eventuale valore di default che possiede quella colonna;
  - is_nullable: indica se quella colonna ammette valori nulli.

e le principali tabelle del catalogo di sistema di PostgreSQL utilizzate nelle interrogazioni della Sezione 4.3.6:

- PG_CLASS: contiene tutte le tabelle e le viste definite nella base di dati; gli attributi principali sono:
  - relname: il nome della tabella o della vista.
- PG_CONSTRAINTS: contiene tutti i vincoli che sono associati alle tabelle della base di dati; gli attributi principali sono:
  - conname: il nome;
  - contype: il tipo che può assumere molti valori, fra cui: 'f' (foreign key), 'p' (primary key) oppure 'u' (unique);

  - conrelid: la tabella su cui è definito il vincolo;
  - confrelid: se il vincolo è di tipo 'f' indica la tabella a cui si riferisce la chiave esterna.

- PG_INDEX: contiene le informazioni riguardo gli indici sulla tabella; gli attributi principali sono:
  - indisprimary: vale *true* sull'indice che rappresenta il vincolo di chiave primaria della tabella.

- PG_ATTRIBUTE: contiene le informazioni su tutte le colonne delle tabelle; gli attributi principali sono:
  - attname: il nome dell'attributo;
  - attnum: la posizione della colonna all'interno della tabella;
  - attrelid: il nome della tabella che contiene l'attributo.

Per ulteriori informazioni si rimanda al sito web contenente la documentazione di PostgreSQL riguardo l'Information Schema [4] e i cataloghi di sistema [5].

1.4 Richiami di probabilità

In questa sezione verranno accennate alcune delle nozioni fondamentali del calcolo di probabilità e statistico e saranno descritte due distribuzioni di probabilità rilevanti nel contesto degli algoritmi presentati nel Capitolo 3. Il materiale presentato si basa principalmente sul contenuto di [6] e [7] a cui si rimanda per gli approfondimenti.

Nella teoria delle probabilità ci si limita a considerare solamente fenomeni aleatori (casuali) dove, con il termine aleatorio si fa riferimento a fenomeni o esperimenti che possono dar luogo a una pluralità di esiti, il cui insieme è noto a priori, e tali che l'esito della singola prova futura non sia completamente prevedibile. Alcuni esperimenti aleatori sono:

- il lancio di una moneta;
- il lancio di un dado;
- la misurazione dell'altezza di una classe di individui scelti a caso.

Definizione 1.5 (**Spazio campionario** e **evento elementare**)**.** L'insieme di tutti i possibili risultati di un esperimento aleatorio è chiamato **spazio campionario** e

viene indicato con Ω. Il singolo risultato prende il nome di **evento elementare**. Un **evento** è un sottoinsieme di eventi elementari.

Due casi particolari di eventi sono: l'**evento certo** Ω e l'**evento impossibile** $\emptyset$; il primo è l'evento che si realizza sempre, il secondo non si realizza mai.

Dopo aver effettuato l'esperimento uno e uno solo degli eventi elementari si sarà verificato. Lo spazio campionario è detto **discreto** se è costituito da un insieme finito o infinito numerabile di punti, **continuo** se è un insieme infinito di cardinalità più che numerabile.

In accordo alla visione frequentista del calcolo delle probabilità assumiamo che gli esperimenti siano ripetibili.

La nozione di probabilità è stata assiomatizzata da Kolmogorov, un matematico russo vissuto nel 1900.

Definizione 1.6 (**Assiomi di Kolmogorov**)**.** Si dice **distribuzione di probabilità** sugli eventi definiti a partire da uno spazio campionario Ω, una qualunque funzione P che abbia le seguenti proprietà:

- a ogni evento casuale A corrisponde una misura di probabilità $P(A)$ compresa fra i valori 0 e 1.
- la probabilità dell'evento certo è 1;
- la probabilità dell'unione di un numero finito (o infinito numerabile) di eventi mutualmente esclusivi è pari alla somma delle singole probabilità.

Da qui in avanti considereremo soltanto spazi campionari finiti in cui gli eventi elementari sono equiprobabili. Sotto tali ipotesi, possiamo dare una definizione di probabilità, denominata **classica**, in quanto storicamente fu una delle prime definizioni, spesso attribuita al matematico francese Pierre Simon Laplace.

Definizione 1.7 (**Probabilità classica**)**.** La **probabilità** di un evento è il rapporto fra il numero dei casi favorevoli all'evento e il numero dei casi possibili.

Non è difficile verificare che la Def. 1.6 è consistente con la Def. 1.7; in particolare, la probabilità classica ha le seguenti proprietà:

- la probabilità di un evento aleatorio è un numero compreso fra 0 e 1;
- la probabilità dell'evento certo è 1, quella dell'evento impossibile è 0;

Questa definizione non è sempre sufficiente per descrivere la probabilità di un fenomeno, ma nel contesto della presente tesi lo è. Per completezza riportiamo le problematiche di questa definizione:

- non tratta i casi in cui gli eventi non sono equiprobabili. Gli eventi devono avere quindi la stessa probabilità;
- suppone un numero finito di eventi e quindi non è utilizzabile se lo spazio campionario è infinito.

Passiamo ora a definire due distribuzioni di probabilità che useremo nel seguito: la distribuzione uniforme discreta e la distribuzione ipergeometrica. Per entrambe possiamo operare con la definizione classica di probabilità.

La distribuzione discreta uniforme, ad esempio, modella il lancio di un dado equilibrato (non truccato) le cui facce sono numerate da 1 a 6 e ogni faccia ha probabilità 1/6 di presentarsi.

Definizione 1.8 (**Distribuzione discreta uniforme**)**.** La distribuzione discreta uniforme attribuisce ad ogni evento elementare la stessa probabilità. La probabilità del verificarsi di un evento, A denotata con $P(A)$, è:

$$P(A) = \frac{|A|}{|\Omega|}$$

dove $|\Omega|$ è la cardinalità, che si assume finita, dello spazio campionario.

Esempio 8. Consideriamo di voler calcolare la probabilità che nel lancio di un dato esca il numero 3:

$$P = \frac{1}{6} = 0{,}1\overline{6}$$

Il cui valore è di poco inferiore al 17%.

Esempio 9. Consideriamo di voler calcolare la probabilità che nel lancio di un dato esca un numero pari:

$$P = \frac{3}{6} = 0.5$$

Il cui valore è esattamente 50%. Notiamo che ora i casi favorevoli sono tre (faccia 2, 4 o 6) e non uno solo. Abbiamo quindi considerato un *evento* "esce un numero pari" che è la composizione di tre *eventi elementari* "esce il numero 2", "esce il numero 4" e "esce il numero 6".

La distribuzione ipergeometrica descrive l'estrazione senza reinserimento di alcune palline, perdenti o vincenti, da un'urna.

Definizione 1.9 (**Distribuzione ipergeometrica**)**.** La distribuzione ipergeometrica attribuisce, nel contesto dell'estrazione, ad ogni evento elementare la stessa probabilità. Tale probabilità varierà dopo dopo ogni estrazione. La probabilità del verificarsi di un evento, indicata con $P(A)$, è:

$$P(A) = \frac{\binom{h}{k}\binom{n-h}{r-k}}{\binom{n}{r}}$$

dove:

- la cardinalità di A è n e si vuole estrarre un sottoinsieme B di k elementi;
- $\binom{h}{k}$ è il numero di possibili estrazioni di k elementi tra gli h di B;
- $\binom{n-h}{r-k}$ è il numero di possibili estrazioni fra i restanti $r-k$ elementi tra gli $n-h$ non in B;
- $\binom{n}{r}$ è il numero di possibili estrazioni di r elementi da A.

Esempio 10. Supponiamo di voler estrarre 5 palline da un'urna contenente 10 palline di cui 7 sono bianche e 3 sono nere; calcoliamo la probabilità che, estraendo un campione a caso di 5 palline 3 di esse siano bianche.

$$P = \frac{\binom{7}{3}\binom{10-7}{5-3}}{\binom{10}{5}} = \frac{105}{252} = 0{,}41\overline{6}$$

Il cui valore è di poco inferiore al 42%.

2

Generazione di dati pseudo-casuali

2.1 Introduzione

Il concetto di numero **pseudo-casuale** è fondamentalmente distinto dal concetto di numero (realmente) casuale. Infatti, mentre un numero casuale è il risultato di un esperimento aleatorio (si veda la Sezione 1.4), un numero pseudo-casuale è generato in modo algoritmico da un calcolatore.

Formalmente un numero casuale deriva dall'osservazione di un singolo risultato di una variabile casuale. I numeri pseudo-casuali sono invece calcolati da un algoritmo deterministico che produce sequenze che statisticamente **appaiono** casuali (cioè hanno le stesse caratteristiche delle sequenze casuali).

Affinché una sequenza di numeri possa essere considerata pseudo-casuale deve avere le seguenti proprietà statistiche:

- i valori devono essere distribuiti secondo una ben definita distribuzione di probabilità, come ad esempio la distribuzione uniforme;
- gli elementi devono essere fra loro indipendenti, cioè la probabilità di estrarre due elementi dev'essere il prodotto delle singole probabilità.

Esempio 11. Ad esempio la sequenza:

$$0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 \text{ sull'intervallo } [0, 10]$$

ha una distribuzione uniforme sui valori ma non soddisfa la seconda proprietà: infatti ogni valore è il successore del precedente (determinato il primo il secondo ha probabilità 1 di venir estratto).

Altra proprietà fondamentale è il **periodo**, ovvero il numero di elementi dopo i quali la sequenza si ripete. In generale il periodo dev'essere il più lungo possibile.

Gli algoritmi per il calcolo di sequenze pseudo-casuali quindi devono essere studiati in modo da soddisfare tutte queste proprietà. Bisogna tenere presente che, conoscendo l'algoritmo usato e alcuni parametri, è sempre possibile ricostruire l'intera sequenza di numeri generati. Il primo valore della successione viene detto **seed** e deve essere scelto con particolare cautela perché influenza le caratteristiche dell'intera sequenza generata. Elenchiamo ora alcuni metodi della scelta del seed **sconsigliati**:

- il valore 0 può creare problemi con alcuni tipi di successioni che andrebbero a generare solo lo 0;
- non usare valori successivi ad un seed già usato: questo creerebbe una forte correlazione fra le due successioni di valori;
- non selezionare il seed in maniera casuale (ad esempio usando il tempo in millisecondi corrente): questo rende impossibile la replicazione della sequenza e non garantisce di ottenere sequenze diverse;
- non usare come seed l'ultimo valore di un'altra sequenza: causa una forte correlazione fra le due successioni di valori.

La maggior parte degli algoritmi che vengono utilizzati in pratica si basa sul **metodo lineare congruenziale** che è presentato nella Sezione 2.2.

Per semplicità e senza tema d'ambiguità d'ora in poi per numeri casuali intenderemo sempre numeri pseudo-casuali generati dal calcolatore.

2.2 Metodo lineare congruenziale

Molti dei metodi utilizzati per la generazione di numeri casuali nascono dal seguente approccio introdotto da D. H. Lehmer, che prende il nome di **metodo lineare congruenziale**. La sequenza di numeri casuali si ottiene dalla seguente successione:

$$X_{n+1} = (aX_n + c) \bmod m$$

dove la scelta dei seguenti valori è fondamentale per ottenere una sequenza che appaia realmente casuale:

- il valore X_0 cioè il primo valore della successione;
- il moltiplicatore a;

- l'incremento c;
- il modulo m.

Esempio 12. Fissati $m = 10$ e $X_0 = c = a = 7$ si ottiene la seguente sequenza:

$$7, 6, 9, 0, 7, 6, 9, 0, \ldots$$

È immediato constatare che tale sequenza non presenta le necessarie proprietà dei numeri pseudo-casuali: si tratta infatti di un gruppo di 4 cifre che si ripete all'infinito. Questo gruppo è il **periodo** e in questo caso ha lunghezza 4.

Una buona sequenza dovrà avere quindi un periodo lungo, cosa che si ottiene scegliendo in modo opportuno le costanti che appaiono nella successione.

Presentiamo subito alcuni casi "limite" sulla scelta dei valori:

- per motivi di efficienza il metodo in origine prevedeva $c = 0$; per ottenere periodi più lunghi però si utilizza con $c \neq 0$;
- i casi in cui $a = 0$ o $a = 1$ producono delle sequenze che sono palesemente non casuali e quindi assumeremo $a>2$;

Definiamo inoltre $b = a - 1$ e presentiamo una nuova successione che rappresenta il valore $(n + k)$-esimo nei termini nel valore n-esimo:

$$X_{n+k} = (a^k X_n + (a^k - 1)c/b) \bmod m$$

Il nostro obiettivo è ora quello di trovare dei buoni valori per definire in maniera soddisfacente la successione del metodo lineare congruenziale.

2.2.1 Scelta del modulo

Nella scelta del modulo m vanno tenute in considerazione alcune cose:

- deve essere sufficientemente grande in quanto rappresenta il numero massimo di elementi nel periodo;
- essere un numero tale per cui la computazione $(aX_n + c) \bmod m$ sia efficiente.

Esempio 13. Supponiamo di dover generare una sequenza binaria. Non è una buona idea scegliere $m = 2$ perché in tal caso si ottiene una cattiva sequenza, che ha questa forma:

$$0, 1, 0, 1, \ldots$$

Dobbiamo quindi scegliere un m grande in modo tale da generare una sequenza che abbia un periodo più lungo (non 2 come nel caso qui sopra).

2.2.2 Scelta del moltiplicatore

Il periodo, come già visto, è limitato superiormente dal valore di m; con una scelta oculata del valore di a possiamo raggiungere questa limitazione e ottenere dei periodi di lunghezza relativamente alta (purché, ovviamente, m sia sufficiente grande).

Teorema 1. *Il metodo lineare congruenziale definito da m, a, c e X_0 ha un periodo di lunghezza m se e solo se:*

- *c ed m sono primi tra loro;*
- *$b = a - 1$ è un multiplo di p per ogni numero primo p che divide m;*
- *b è un multiplo di 4 se anche m lo è.*

La dimostrazione del Teorema 1 può essere trovata in [8].

2.2.3 Scelta della potenza

La potenza di una sequenza lineare congruenziale di periodo massimo è definita come il più piccolo intero s tale che:

$$b^s \equiv 0 \mod m$$

Osservazione 2. Tale numero s esiste sempre perché b è il multiplo di ogni numero primo che divide m.

In generale una potenza alta è una condizione necessaria ma non sufficiente per ottenere una successione casuale. Tale numero può essere quindi utilizzato per verificare se una successione è "cattiva" ma **non** se è "buona".

2.3 Funzioni per la verifica

Ottenere una successione il cui periodo è alto non è sufficiente per dimostrare che genera una sequenza di numeri realmente casuali.

Per questo viene in aiuto la statistica. L'applicazione di una serie di test statistici può convincerci di avere a che fare con una sequenza considerabile casuale o meno. Ovviamente non può dimostrare nulla perché se anche la successione considerata passasse i test $T_1, \ldots, T_n$ nulla garantisce che passi anche un eventuale test T_{n+1}.

Analizzeremo ora una serie di test che si sono rilevati efficienti per studiare le proprietà di casualità di una sequenza di numeri. Idealmente una successione

pseudo-casuale dovrebbe passarli tutti, ma nella pratica ciò non sempre è ottenibile; spesso, perciò, si accettano come pseudo-casuali anche sequenze che non superano alcuni test statistici di casualità.

2.3.1 Test Chi-quadro

Il test **chi-quadro** (o chi-quadrato) si basa sula *variabile casuale* chi-quadro, indicata con χ^2, che descrive la somma dei quadrati di alcune variabili aleatorie indipendenti aventi distribuzione normale.

Si supponga di avere n osservazioni indipendenti, che nel nostro caso saranno i valori della sequenza generata dall'algoritmo, che ricadono in una fra k categorie. Chiamiamo Y_s il numero di osservazioni che ricadono nella categoria s e p_s la probabilità che un'osservazione ricada nella categoria s. Per la teoria dei grandi numeri ci si aspetta che:

$$Y_s \approx n \cdot p_s \text{ per n sufficientemente grande}$$

Si vuole quindi valutare quanto "lontani" si è dai valori attesi definendo:

$$V = \frac{(Y_1 + n \cdot p_1)^2}{n \cdot p_1} + \ldots + \frac{(Y_k + n \cdot p_k)^2}{n \cdot p_k} = \sum_{0<s\leq k} \frac{(Y_s + n \cdot p_s)^2}{n \cdot p_s}$$

In Figura 2.1 troviamo la tabella dei valori che assume la variabile casuale χ^2. La lettura della tabella avviene in questo modo:

- si legge la riga in cui $v = k - 1$ dove k rappresenta il numero di gradi di libertà della variabile χ^2;
- si comparano i valori della riga con quello di V. Ad esempio con $v = 8$ il fatto che al 99% ($p = 0.99$) la variabile valga 20.09 significa che $V < 20.09$ nel 99% dei casi e che ci aspettiamo valori di $V > 20.09$ solo nell'1% dei casi: ottenere un valore di V pari a 30 per esempio è molto sospetto.

Veniamo ora alla domanda più importante: qual è un buon valore per V? Secondo Knuth [8] tale valore:

- è buono se meno dell'1% dei valori si discosta da V;
- è sospetto se fra l'1 e il 5% dei valori si discostano da V;
- è molto sospetto se fra l'5 e il 10% dei valori si discostano da V;

Se ripetendo il test almeno tre volte con valori di n differenti si ottengono almeno due casi "sospetti" allora la sequenza considerata non è casuale.

	$p = .01$	$p = .05$	$p = .25$	$p = .50$	$p = .75$	$p = .95$	$p = .99$
$\nu = 1$	0.00016	0.00393	0.1015	0.4549	1.323	3.841	6.635
$\nu = 2$	0.02010	0.1026	0.5753	1.386	2.773	5.991	9.210
$\nu = 3$	0.1148	0.3518	1.213	2.366	4.108	7.815	11.34
$\nu = 4$	0.2971	0.7107	1.923	3.357	5.385	9.488	13.28
$\nu = 5$	0.5543	1.1455	2.675	4.351	6.626	11.07	15.09
$\nu = 6$	0.8720	1.635	3.455	5.348	7.841	12.59	16.81
$\nu = 7$	1.239	2.167	4.255	6.346	9.037	14.07	18.48
$\nu = 8$	1.646	2.733	5.071	7.344	10.22	15.51	20.09
$\nu = 9$	2.088	3.325	5.899	8.343	11.39	16.92	21.67
$\nu = 10$	2.558	3.940	6.737	9.342	12.55	18.31	23.21
$\nu = 11$	3.053	4.575	7.584	10.34	13.70	19.68	24.73
$\nu = 12$	3.571	5.226	8.438	11.34	14.84	21.03	26.22
$\nu = 15$	5.229	7.261	11.04	14.34	18.25	25.00	30.58
$\nu = 20$	8.260	10.85	15.45	19.34	23.83	31.41	37.57
$\nu = 30$	14.95	18.49	24.48	29.34	34.80	43.77	50.89
$\nu = 50$	29.71	34.76	42.94	49.33	56.33	67.50	76.15

Figura 2.1: Distribuzione probabilità χ^2 - Fonte [9]

2.3.2 Test spettrale

Il test spettrale è il test principe per i metodi lineari congruenziali in quanto un buon metodo lo passerà mentre un cattivo metodo tenderà a fallirlo nella maggioranza dei casi [8].

Il test spettrale esamina l'intero periodo di un generatore lineare congruenziale e riesce quindi a valutare la proprietà di casualità a livello globale. Per lavorare con il test spettrale assumiamo di avere una sequenza di numeri U_n definita da un metodo lineare congruenziale il cui periodo è m.

L'idea del test è di considerare tutti i possibili sottoinsiemi che contengono il periodo m in uno spazio a t-dimensioni:

$$(U_n, U_{n+1}, \ldots, U_{n+t-1})$$

Sotto l'assunzione di avere un periodo di lunghezza m si può esprimere questo insieme nel seguente modo:

$$\frac{1}{m} \cdot \{(X, s(X), s(s(X)), \ldots, s^{t-1}(X)) \mid 0 \leq X \leq m\}$$

dove $s(X)$ rappresenta la successione del metodo lineare congruenziale.

Con un finito numero di punti in uno spazio finito si possono rappresentare tutti i punti con un numero finito di rette parallele (spazio bidimensionale), un numero finito di piani paralleli (spazio tridimensionale) e in generale con un numero finito di iperpiani $(t-1)$-dimensionali (spazio t-dimensionale).

Diciamo che $1/v_2$ è la massima distanza fra due linee, $1/v_3$ la massima distanza fra due piani e, in generale, $1/v_t$ la massima distanza fra iperspazi t-dimensionali. Chiamiamo ora v_2 la precisione bidimensionale del generatore di numeri casuali, v_3 la precisione tridimensionale del generatore di numeri casuali, e, in generale, v_t la precisione t-dimensionale del generatore di numeri casuali.

Sperimentalmente si è notato che:

- la precisione di una sequenza casuale generata da un buon metodo è uguale in tutte le dimensioni;
- la precisione del periodo di una sequenza casuale generata da un buon metodo decresce con l'aumentare delle dimensioni.

Il test spettrale compara tutte queste accuratezze per valutare la bontà di un generatore di numeri casuali. Tralasciando tutte le basi teoriche, che possono essere approfondite su [8], ci limitiamo a citare il seguente, importante, risultato:

$$v_t = min\{\sqrt{X_1^2, \ldots, X_t^2} \mid X_1 + a \cdot X_2 + \ldots + a^{t-1} \cdot X_t = 0 \mod m\}$$

3

Algoritmi per il campionamento

3.1 Introduzione

Nell'applicativo presentato in questa tesi vengono utilizzati diversi algoritmi per la generazione dei dati pseudo-casuali. Questi algoritmi sono stati scelti sulla base di due esigenze:

- generare dati di diverso tipo, il solo tipo numerico non è infatti sufficiente;
- generare dati con o senza ripetizioni.

Il primo requisito è determinato dal fatto che una base di dati non contiene solo tipi di dato numerico, il secondo requisito deriva dai vincoli che una base di dati può avere: vincoli che possono imporre l'unicità dei valori presenti in una o più colonne. In assenza di tali vincoli, si può (e si deve) generare i dati in modo che ammettano ripetizioni.

In questo capitolo prenderemo in esame tutti gli algoritmi utilizzati nell'applicativo, dividendoli in tre categorie: per la generazione di dati con ripetizioni, per la generazione di dati senza ripetizioni e per generare a partire da numeri interi dati con tipi diversi (*data mapping*). Per ognuno daremo cenni delle dimostrazioni di correttezza e complessità e ne elencheremo le principali proprietà.

3.2 Generazione con ripetizioni

3.2.1 Generazione di un numero intero

Per la generazione di un numero intero ci appoggiamo alle funzioni offerte dalle librerie del linguaggio scelto, in questo caso Java. Tali funzioni vengono implementate con il metodo lineare congruenziale presentato nella Sezione 2.2 avente distribuzione uniforme. [18]

3.2.2 Algoritmo per il campionamento con ripetizioni

In questa sezione presenteremo un semplice algoritmo per la generazione di dati casuali senza il vincolo di unicità. Tale algoritmo prende in input due valori numerici: uno rappresenta la grandezza dell'intervallo da cui prendere i numeri, l'altro invece la quantità di numeri da generare.

Il funzionamento è molto semplice: si tratta di generare pseudo-casualmente m numeri casuali appartenenti all'intervallo $[1, n]$ e di inserirli nel vettore. Lo pseudo-codice è il seguente:

Input : quantità di numeri da generare m e massimo numero generabile n
Output: vettore contenente dati casuali con ripetizioni S
initialize S;
for $i \leftarrow 0$ **to** m **do**
  insert $rand(1, n)$ in S;
end
return S;

Algoritmo 1: Generazione dati pseudo-casuali

La complessità dell'algoritmo è $O(m)$, nell'ipotesi in cui la funzione rand() abbia complessità $O(1)$. Lo spazio di lavoro è $O(m)$ se si tiene in memoria il vettore dei numeri generati di grandezza m. Tuttavia, poiché non è necessario ricordare quali numeri sono stati in precedenza generati (dal momento che sono consentite ripetizioni), è possibile implementare l'algoritmo in spazio $O(1)$ eliminando il vettore S e producendo in output ciascun numero man mano che viene generato. [1]

La correttezza dell'algoritmo discende in modo ovvio dall'ipotesi che la funzione primitiva rand() generi sequenze pseudo-casuali di interi uniformemente distribuiti nell'intervallo specificato.

3.3 Generazione senza ripetizioni

Gli algoritmi per la generazione dei dati senza ripetizioni sono più delicati, infatti si rischia di commettere degli errori grossolani. Si prenda ad esempio un banale approccio che consiste nel generare un dato, controllare se è già presente e scartarlo in caso affermativo. Un metodo di questo tipo prende il nome di *trial-and-error* e porta ad un algoritmo che non gode della proprietà di terminazione forte; in altre parole l'algoritmo potrebbe non terminare mai la sua esecuzione.

[1] Lo spazio occupato dall'output rimane ovviamente $O(m)$.

Input : intervallo $[1..n]$, quantità da generare m
Output: vettore di numeri casuali distinti di dimensione m
initialize S;
while *lenght of* S *is* m **do**
  $t \leftarrow rand(1, j)$;
  if t *is not in* S **then**
    insert t in S;
  end
end
return S;

Algoritmo 2: Generazione dati distinti con approccio naïve

In questa sezione presenteremo degli algoritmi appositamente pensati per generare dati senza ripetizioni, mantenendo nel contempo la proprietà di terminazione forte.

3.3.1 Algoritmo di Floyd

L'algoritmo di Floyd [12] prende in input un vettore vuoto e inserisce al suo interno valori generati in modo pseudo-casuale senza mai inserire un valore già presente. L'algoritmo opera seguendo i seguenti passi:

- inizializza il vettore S;
- itera, usando un indice j, sugli ultimi m valori in $[0, n]$ generando, in ciascuna iterazione, un numero casuale $t \in [1, j]$;
- se t non è presente in S lo inserisce, altrimenti inserisce j; in entrambi i casi l'indice j viene incrementato.

Lo pseudo-codice è il seguente:

```
Input  : intervallo [1..n], quantità da generare m
Output: vettore di numeri casuali distinti di dimensione m
initialize S;
for j ← n − m + 1 to n do
    t ← rand(1, j);
    if t is not in S then
        insert t in S;
    else
        insert j in S;
    end
end
return S;
```

Algoritmo 3: Floyd

Facendo riferimento all'Algoritmo di Floyd:

- n è un intero che rappresenta il massimo numero generabile;
- m è il numero di elementi che si vuole generare; ogni elemento sarà all'interno dell'intervallo $[1, n]$;
- S è il vettore di lunghezza n, inizialmente vuoto, che conterrà i numeri generati.

La complessità dell'algoritmo dipende dall'implementazione della ricerca di un elemento nel vettore; tre possibili soluzioni sono:

- effettuare una ricerca lineare di complessità $O(m)$ ottenendo una complessità totale di $O(m^2)$;
- utilizzando una struttura dati, come un albero di ricerca, si può effettuare una ricerca che ha complessità $O(\log\ m)$ ottenendo una complessità totale di $O(m \log\ m)$.
- usando una hash per implementare l'insieme, si può ottenere un tempo d'esecuzione medio di O(m).

Per la generazione del singolo numero pseudo-casuale si possono utilizzare le funzioni offerte delle librerie del linguaggio che hanno complessità $O(1)$. Solitamente queste funzioni vengono implementate utilizzando il metodo lineare congruenziale descritto nella Sezione 2.2

La complessità in spazio risulta essere di $O(m)$, infatti bisogna mantenere costantemente in memoria le informazioni relative ai numeri già presenti.

Per dimostrare la correttezza dell'Algoritmo di Floyd, bisogna dimostrare due cose:

- che la sequenza generata non contiene ripetizioni;
- che la sequenza è generata in accordo a una distribuzione uniforme discreta.

La verifica dell'unicità è semplice e segue dal fatto che la generazione del dato avviene in modo casuale, e che se tale dato è già presente non viene inserito; in tal caso viene inserito un dato appartenente all'insieme $[j, n]$ che sicuramente non è presente. La dimostrazione di uniformità della distribuzione è più complessa ed è conseguenza del risultato seguente.

Teorema 2. *L'algoritmo di Floyd presenta una distribuzione di probabilità uniforme dei dati, in cui ogni permutazione ammissibile del vettore di input ha probabilità:*

$$P(N) = \frac{1}{\binom{n}{m}}$$

dove $\binom{n}{m}$ *è il numero di permutazioni di cardinalità* m *di un insieme composto da* n *elementi.*

Dimostrazione. La dimostrazione procede per induzione valutando la probabilità P_j che l'elemento k non sia stato scelto dall'algoritmo al tempo T_j tale che:

$$n - m \leq T_j \leq n$$

La probabilità nel caso base, cioè quando siamo al tempo T_{n-m} vale:

$$P_{n-m} = 1$$

in quanto al primo passo non sono stati scelti elementi e quindi siamo certi che k non sia stato scelto. Trattiamo ora i casi in cui $T_j > n - m$ e calcoliamo la probabilità che un fissato valore k sia stato scelto dall'Algoritmo 3. Distinguiamo due casi, a seconda della posizione di k:

- $k \leq n - m$: la probabilità di scegliere k al tempo T_j è pari a $1/j$, quindi la probabilità di **non** scegliere k vale:

  $$1 - \frac{1}{j}$$

  Tale probabilità vale nell'ipotesi che la funzione rand() generi un numero con distribuzione di probabilità uniforme. Ricordiamo che la funzione rand(), in

ciascuna iterazione, seleziona a caso un numero nell'intervallo $[1, j]$. Dato che la scelta in ciascuna iterazione è indipendente dalle altre si può concludere che la probabilità di **non** scegliere k in nessuna iterazione (quindi al tempo T_n) è:

$$P_n = \prod_{j=n-m+1}^{n} P_j = \prod_{j=n-m+1}^{n} \frac{j-1}{j} =$$

$$= \frac{n-m}{n-m+1} \cdot \frac{n-m+1}{n-m+2} \cdot \dots \cdot \frac{n-2}{n-1} \cdot \frac{n-1}{n} = \frac{n-m}{n}$$

la probabilità di scegliere k al tempo T_n vale dunque:

$$1 - P_n = \frac{m}{n}$$

- $k > n-m$: la probabilità di non aver scelto k fino al tempo T_{k-1} in questo caso vale 1. Questo perché al generico tempo T_j sono state eseguite $j-(n-m)$ iterazioni. In particolare al tempo T_{n-m+1} è stata fatta una sola iterazione e al tempo T_n sono state fatte tutte le m iterazioni. Al tempo T_{k-1} sono state eseguite pertanto $k-1-(n-m)$ iterazioni che hanno scelto $k-1-(n-m)$ valori tutti necessariamente minori di k. Ne segue che, dopo la scelta effettuata al tempo T_{k-1} indipendentemente dal ramo dell'algoritmo seguito in ciascuna iterazione nell'intervallo $[1, k-1]$ sono rimasti:

$$k-1-(k-1-(n-m)) = n-m \text{ valori non scelti}$$

Al tempo T_j con $j = k$ la probabilità di **non** scegliere k è la probabilità di eseguire il ramo "then" dell'Algoritmo 3 ed è quindi la probabilità di scegliere un elemento che non è già stato scelto nell'intervallo $[1, k]$ che sia diverso da k. Tale probabilità è precisamente:

$$P_k = \frac{n-m}{k}$$

La probabilità di non aver scelto k fino al tempo T_k è data da $P_1 \cdot P_2 \cdot \dots \cdot P_k = P_k$ per l'indipendenza delle scelte, dato che $P_1, \dots, P_{k-1}$ sono tutte pari a 1. Dal tempo T_{k+1} in poi si ricade nel caso precedente, perciò si ottiene la formula seguente. Dato che ciascuna iterazione è indipendente dalle altre si può concludere che la probabilità di **non** scegliere k in nessuna iterazione è:

$$P_n = \frac{n-m}{k} \cdot \prod_{j=k+1}^{n} \frac{j-1}{j} =$$

$$= \frac{n-m}{k} \cdot \frac{k}{k+1} \cdot \frac{k+1}{k+2} \cdot \dots \cdot \frac{n-2}{n-1} \cdot \frac{n-1}{n} = \frac{n-m}{n}$$

La probabilità di scegliere k entro il tempo T_n vale dunque:

$$1 - P_n = \frac{m}{n}$$

In entrambi i casi abbiamo dimostrato che la probabilità di scegliere un elemento è:

$$P = \frac{m}{n}$$

che è in accordo ad una distribuzione uniforme. □

Esempio 14. Si vogliano generare 3 numeri nell'intervallo $[1, 10]$; l'algoritmo di Floyd procede nel seguente modo:

- inizialmente $j = n - m + 1 = 10 - 3 + 1 = 8$;
- calcola un numero casuale t nell'intervallo $[1, 8]$ e controlla se è già presente nel vettore, in tal caso inserisce $j = 8$, altrimenti inserisce t; in ogni caso j viene incrementato diventando 9;
- calcola un numero casuale t nell'intervallo $[1, 9]$ e controlla se è già presente nel vettore, in tal caso inserisce $j = 9$, altrimenti inserisce t; in ogni caso j viene incrementato diventando 10;
- calcola un numero casuale t nell'intervallo $[1, 10]$ e controlla se è già presente nel vettore, in tal caso inserisce $j = 10$, altrimenti inserisce t; in ogni caso j viene incrementato diventando 11;
- 11 è maggiore di 10 e quindi l'algoritmo termina.

3.3.2 Algoritmo di Knuth

L'algoritmo di Knuth [10] [11] [12] prende in input solamente due valori interi: uno rappresenta la grandezza dell'intervallo da cui scegliere casualmente i numeri, l'altro invece la quantità di numeri da generare.

Input : intervallo $[1..n]$, quantità da generare m
Output: vettore di numeri casuali distinti di dimensione m
initialize S;
for $i \leftarrow 0$ **to** n **do**
  $t \leftarrow rand(1, n)$;
  if $(t < m)$ **then**
    insert i in S;
    $m \leftarrow m - 1$;
  end
end
return S;

Algoritmo 4: Knuth

L'algoritmo di Knuth ha complessità lineare nella grandezza dell'intervallo dei numeri ammissibili, quindi $O(n)$. Rispetto all'algoritmo di Floyd, però può essere implementato in spazio di lavoro costante eliminando il vettore S e producendo in output i valori man mano che sono generati.

In virtù della sua complessità quest'algoritmo risulta migliore qualora m si avvicini a n. Al contrario se $m \ll n$ la complessità $O(m \cdot \log m)$ dell'algoritmo di Floyd favorisce quest'ultimo.

Anche questo algoritmo presenta una distribuzione di probabilità uniforme dei dati, in cui ogni permutazione ammissibile del vettore di input ha probabilità:

$$P(N) = \frac{1}{\binom{n}{m}}$$

dove $\binom{n}{m}$ è il numero di permutazioni di cardinalità m di un insieme composto da n elementi.

Osservazione 3. L'algoritmo genera una sequenza di dati ordinata.

3.3.3 Algoritmo R

L'algoritmo R [14] appartiene alla classe degli algoritmi *"reservoir"*, e si rivela particolarmente efficace perché non richiede la conoscenza della cardinalità del range di valori N da cui si vogliono estrarre n numeri in modo casuale.

L'algoritmo fu proposto in quanto, un tempo, con l'utilizzo dei supporti di memorizzazione su nastro effettuare un'intera scansione del nastro per trovare tale N risultava in pratica spesso inaccettabile. Si rivela tuttora utile qualora il range dei dati non stia in memoria (o nei buffer riservati al programma) e allora dovremo

caricare e scaricare dalla memoria porzioni del file di input; operazione che su un disco tradizionale (di tipo magnetico) è costosa come lo è, in misura inferiore, anche su un disco a stato solido. Campi di applicazione moderni possono essere ritrovati nel contesto delle basi di dati: infatti, effettuare la lettura sequenziale di un'intera tabella di grandi dimensioni può essere in certi casi eccessivamente oneroso.

In generale un algoritmo di tipo *"reservoir"* è definito nel seguente modo: al primo passo legge i primi n record del file e li carica in un "reservoir". A partire dal record $(t+1)$-esimo, per $t \geq n$, tutti gli altri record vengono processati in modo sequenziale. Ad ogni passo, è mantenuto l'invariante per cui il "reservoir" rappresenta un campione casuale di cardinalità n dei primi $t+1$ record del file considerati fino a quel momento.

Più precisamente, l'algoritmo R ha le seguenti caratteristiche:

- quando il $(t+1)$-esimo record viene processato, per $t \geq n$, gli n candidati formano un sottoinsieme casuale dei primi $t+1$ record;
- l'elemento $t+1$ (sempre con $t \geq n$) entra a far parte del *"reservoir"* con probabilità
$$P = \frac{n}{t+1}$$
- quando un elemento è selezionato per entrare nel *"reservoir"*, va a sostituire uno degli n elementi correntemente presenti, scelto in modo casuale (con distribuzione di probabilità uniforme).

L'Algoritmo 5 riporta lo pseudo-codice della procedura appena delineata. Facendo riferimento all'Algoritmo R:

- $eof()$ è una funzione che restituisce "true" quando si è arrivati alla fine del file;
- $readNextRecord(x)$ legge il prossimo record dal file e lo memorizza nel *"reservoir"* in posizione x;
- $skipRecord(x)$ salta i successivi x record del file.

Input : file, quantità da generare n
Output: vettore contenente n elementi scelti casualmente dal file
initialize S;
for $i \leftarrow 1$ **to** n **do**
  readNextRecord(S[i]);
end
$t \leftarrow n$;
while ***not*** *eof()* **do**
  $t \leftarrow t + 1$;
  $M \leftarrow rand(1, t - 1)$;
  if $M<n$ **then**
    readNextRecord(S[M]);
  end
  skipRecord(1);
end
return S;

Algoritmo 5: Algoritmo R

La complessità dell'algoritmo è $O(N)$, dove N è la lunghezza del file, in quanto bisogna scandire interamente l'intero file una volta e l'operazione di lettura di un record può essere implementata in tempo costante $O(1)$.

In [14] è possibile trovare un'implementazione in tempo $O(n + n\log(N/n))$.

3.3.4 Algoritmo di Fisher-Yate-Durstenfeld

L'ultimo algoritmo per il campionamento senza ripetizioni è in realtà un algoritmo di *shuffling*, che permette cioè di effettuare una permutazione casuale degli elementi di un vettore. Benché non sia un algoritmo di generazione di dati, si può dire che mantiene la proprietà di unicità dell'insieme di partenza; ovviamente se l'insieme di partenza gode di tale proprietà.

L'algoritmo proposto è una variazione dell'algoritmo di Fisher-Yate [15] [16] e prende in input un vettore, restituendo in output una permutazione casuale dello stesso. Lo pseudo-codice è il seguente:

Input : vettore di elementi S
Output: permutazione del vettore S
initialize S;
for $k \leftarrow n-1$ **to** 1 **do**
  $r \leftarrow rand(1, k)$;
  swap $S[k]$ with $S[r]$;
end
return S;

Algoritmo 6: Fisher-Yate-Durstenfeld

La complessità dell'algoritmo è $O(n)$, infatti le due operazioni di generazione di un numero casuale e di scambio di un vettore si possono realizzare con complessità di $O(1)$. Lo spazio di memoria utilizzato è simile a quello dell'algoritmo di Knuth in quanto si memorizza un unico vettore di dimensione n e poche altre variabili temporanee.

3.4 Generazione di dati non numerici

In quest'ultima sezione presenteremo diversi algoritmi che permettono di ottenere da un numero altri tipi di dato. In altre parole effettuano il cosiddetto *data mapping* fra due tipi, di cui il primo è un numero.

3.4.1 Conversione da interi a stringhe

Prenderemo ora in esame un semplice algoritmo per la conversione di un numero in una stringa. L'algoritmo si basa sulla rappresentazione in base 26, che genera un linguaggio contenente le lettere dell'alfabeto Inglese maiuscole. L'algoritmo prende in input un valore numerico e restituisce in output la sua rappresentazione in base 26. Lo pseudo-codice è il seguente:

Input : numero da convertire
Output: stringa che rappresenta il numero
String *conv*;
repeat
  $rm \leftarrow num \ mod \ 26$;
  $conv \leftarrow (rm + \text{'A'}) + conv$;
  $num \leftarrow (num - rm) \div 26$;
until $num > 0$;
return *conv*;

Algoritmo 7: Data-mapping $int \rightarrow String$

La complessità dell'algoritmo è $O(\log_{26} num)$ dove num è il numero da convertire. Siccome l'algoritmo viene applicato ad ogni elemento del vettore, la complessità risultante risulta di $O(n \log_{26} num)$ dove n è la dimensione del vettore di elementi da convertire.

La complessità in spazio risulta costante in quanto non vi è la necessità di creare altre variabili oltre a quelle presenti.

La proprietà di unicità dei dati generati è mantenuta, in quanto la trasformazione fra basi può essere vista come l'applicazione di una funzione biettiva. Da questo segue la proprietà di correttezza dell'algoritmo.

Questo algoritmo genera stringhe la cui lunghezza è determinata dal numero di input; qualora sia necessario generare stringhe di lunghezza fissata n è sufficiente generare un numero intero i tale che:

$$26^{n-1} < i < 26^n - 1$$

Ottenendo in tal modo una stringa la cui lunghezza è esattamente n.

3.4.2 Conversione da interi a date

Prendiamo ora in esame un algoritmo che genera delle date nel formato SQL di cui presentiamo subito lo pseudo-codice:

Input : numero rappresentante i millisecondi
Output: stringa che rappresenta una data in formato SQL
return $date \leftarrow Date(num)$;

Algoritmo 8: Data-mapping $int \rightarrow Date$

L'algoritmo prende in input num, un numero rappresentante il numero di millisecondi che sono passati dal 1 Gennaio del 1970 ad una data che può essere anche nel futuro e, grazie alla funzione $Date(num)$, restituisce in output una data nel formato SQL. Il formato SQL per le date è "AAAA-MM-DD" dove:

- *AAAA* è l'anno scritto su quattro cifre;
- *MM* e *DD* sono rispettivamente il mese e il giorno scritti su due cifre.

Per la generazione di un insieme di date possiamo generare un vettore di numeri interi con gli algoritmi di *Floyd* o di *Knuth*; tale vettore verrà poi iterativamente elaborato per generare le date nel formato corretto.

La proprietà di distribuzione uniforme delle date viene garantita dagli algoritmi di *Floyd* e *Knuth* mentre la proprietà di unicità può essere garantita generando dei numeri che riescano a discriminare un giorno dall'altro. Dato che un giorno ha 86.400.000 millisecondi è sufficiente generare numeri distanziati di 86.400.000 fra loro. Per ottenere questo è sufficiente generare un numero che rappresenta il giorno e moltiplicarlo per 86.400.000. Così facendo con gli algoritmi di *Floyd* o di *Knuth* generano dei giorni che saranno poi convertiti in data.

3.4.3 Conversione da interi a floating point

Prendiamo ora in esame alcuni metodi per la generazione di numeri floating point casuali distinti.

I numeri reali sono infiniti e quindi non sono rappresentabili nel calcolatore (a meno di non avere infinita memoria). Per questo motivo sono state introdotte varie rappresentazioni per i numeri reali che utilizzano una quantità finita di memoria (bit). Una di queste rappresentazioni è quella mediante i numeri floating point, descritta nello standard IEEE 754 [17] e che comprende quattro formati: singola precisione (32 bit), doppia precisione (64 bit), quadrupla precisione (128 bit) e "mezza" precisione (16 bit).

Il formato a singola precisione, di gran lunga quello attualmente più utilizzato, rappresenta un numero razionale (e approssima i numeri reali) con un numero a 32 bit i cui bit sono divisi in tre gruppi:

- il primo bit più significativo rappresenta il segno;
- i bit dal 2 all'8 rappresentano l'esponente e del numero;
- gli ultimi 23 bit (dal 9 al 32) rappresentano la mantissa m del numero.

Un possibile approccio per la generazione di numeri floating point consiste nell'utilizzo dell'algoritmo naïve (vedi l'Algoritmo 3.1) avendo cura di non effettuare la conversione in numero intero ma di tenere il numero floating point che viene generato dalla funzione *rand* (nell'intervallo $[0, 1)$). L'algoritmo presenta però il solito problema: non gode della proprietà di terminazione forte.

Un altro approccio consiste nel generalizzare uno degli algoritmi presentati precedentemente in modo che generi numeri floating point; l'algoritmo di *Floyd* si presta bene allo scopo. Ad esempio è possibile riscriverlo in modo tale che prenda gli elementi da un insieme, contenente numeri floating point, di grandezza n.

Sotto l'assunzione di inserire nell'insieme **tutti** i numeri floating point possiamo quindi derivare che l'algoritmo manterrà la proprietà di distribuzione uniforme dei dati estratti. L'algoritmo, in altre parole, genererà un indice (inteso come indice a vettore) che permetterà di ottenere l'n-esimo elemento dell'insieme. L'unico problema è dovuto al costo computazionale e allo spazio da riservare in memoria per salvare **tutti** i numeri floating point: i floating point non sono infiniti come i reali, perché il calcolatore lavora con una precisione finita (ad esempio 32 o 64bit), ma non è in generale possibile (né sarebbe efficiente) mantenere in modo esplicito un vettore di tutti i valori rappresentabili in un a data architettura.

Per completezza riportiamo la pseudo-codice della variante dell'algoritmo di *Floyd* che genera casualmente una sequenza di valori scelti da un insieme arbitrario invece che da un intervallo del tipo $[1..n]$.

Input : insieme IN da cui estrarre i valori, quantità da generare m
Output: vettore di numeri casuali distinti di dimensione m
initialize S;
for $j \leftarrow n - m + 1$ **to** n **do**
  $t \leftarrow rand(1, j)$;
  if $IN[t]$ *is not in* S **then**
    insert $IN[t]$ in S;
  else
    insert $IN[j]$ in S;
  end
end
return S;

Algoritmo 9: Floyd - variante per insiemi non tipizzati

Un ultimo approccio sfrutta le caratteristiche della rappresentazione IEEE di un numero floating point. Per generare un numero floating point si può operare nel seguente modo:

- si genera un numero intero con gli algoritmi di *Floyd* o di *Knuth*;
- si converte il numero appena generato nella rappresentazione binaria;
- a partire da quest'ultima si costruisce il numero floating point corrispondente.

Esempio 15. Sia 2437823 un numero intero generato con l'algoritmo di *Floyd*. La sua rappresentazione in binario è:

0 0000000 00100101001100101010111111

Il numero floating point corrispondente è così caratterizzato:

- il suo segno è positivo in quanto il primo bit più significativo è 0;
- l'esponente è il minimo possibile (-39 nella rappresentazione a 32 bit) in quanto i 7 bit che lo rappresentano valgono tutti 0;
- la mantissa è 0.5812222957611084

La cui rappresentazione è $3.416118E-39$.

Generando n interi possiamo quindi ottenere n numeri floating point. Tali numeri saranno distinti in quanto ad ogni rappresentazione binaria (quindi ad ogni intero) corrisponde uno e uno solo numero floating point.

4

Implementazione

4.1 Introduzione

In questo capitolo verranno trattate le tematiche di gestione dei vincoli di una base di dati, verranno descritte tutte le classi realizzate nell'applicativo e infine si daranno alcune informazioni sulle modalità di esecuzione del programma.

Introduciamo due notazioni che ci serviranno nel capitolo e un esempio di schema relazionale che illustra come le interazioni tra i vincoli siano in generale tutt'altro che banali.

Definizione 4.1 (**Dominio di un attributo**)**.** Il **dominio effettivo** di un attributo A rispetto a una data istanza di relazione r, denotato con $dom^r_{\text{eff}}(A)$ (o semplicemente $dom_{\text{eff}}(A)$ quando r è chiara dal contesto), è dato da:

$$\{\, t(A) \mid t \in r \,\}$$

ed è cioè l'insieme dei valori assunti da $t(A)$ per ciascuna tupla t di r.

Definizione 4.2 (**Dominio di X nell'istanza r**)**.** Il dominio effettivo (in r) di una k-upla di attributi $(X_1, \ldots, X_k)$ è definito come:

$$dom^r_{\text{eff}}(\bar{X}) = \{\, (t(X_1), \ldots, t(X_k)\,) \mid t \in r \,\}$$

Esempio 16. Consideriamo il seguente schema di relazione:

- R(A,B,C,D,E) a cui sono applicati i seguenti vincoli: PK:(A,B), UNI:(B,C,D), CE:(A,B,C)→S(A,B,C) e CE:(C,D,E)→T(C,D,E);
- S(A,B,C) a cui è applicato un vincolo PK:(A,B,C);
- T(C,D,E) a cui è applicato un vincolo PK:(C,D,E).

Graficamente può essere rappresentato come nella Figura 4.1

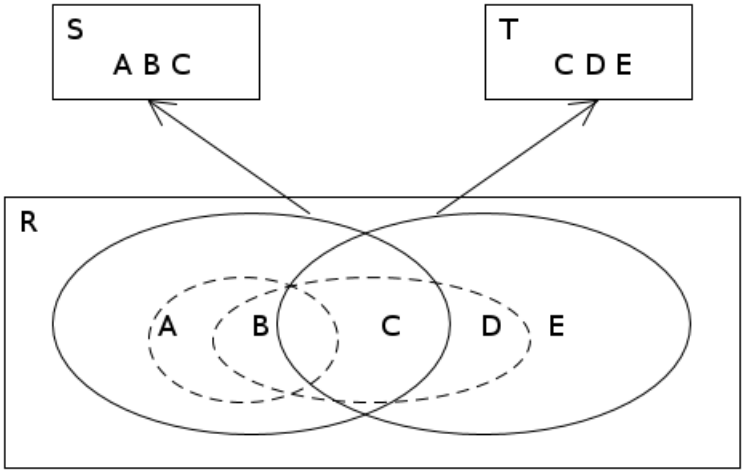

Figura 4.1: Uno schema di relazione

Nella figura, i vincoli di chiave esterna sono rappresentati con un ellisse, quelli di unicità con un ellisse tratteggiato e i vincoli di integrità referenziale fra le tabelle sono rappresentati con una freccia.

4.2 Gestione dei vincoli

Analizzeremo ora i tipi di vincoli che si possono trovare in una base di dati; cominceremo dai vincoli più semplici fino ad arrivare a vincoli con complessità arbitraria. Nella presente tesi ci limiteremo a considerare due soli tipi di vincoli: **unicità** e **chiave esterna** (integrità referenziale). Non prenderemo in considerazione né vincoli di valor non nullo (**not null**) né vincoli determinati da clausole "constraint... check" o vincoli basati su "create assertions" o trigger.

Sotto queste restrizioni il vincolo di **chiave primaria** può essere trattato come se fosse un vincolo di unicità; ricordiamo infatti che la chiave primaria può essere vista come un vincolo di **unicità** associato ad un vincolo **not-null**.

4.2.1 Vincoli di unicità

I vincoli di unicità possono essere specificati sia su un singolo attributo A sia su insieme di attributi X. Nel primo caso si richiede che i valori di A in una data istanza siano tutti distinti, nel secondo invece si richiede che le (sotto-)tuple su X in una data istanza siano distinte.

Se gli unici vincoli associati a uno schema di relazione sono vincoli d'unicità, la generazione di un'istanza casuale è semplice, infatti:

- i dati da inserire nel singolo attributo si possono generare con un algoritmi di campionamento senza ripetizioni, come l'algoritmo di Floyd 3.3.1 o di Knuth 3.3.2;
- i dati da inserire nelle tuple si possono generare con opportune generalizzazioni degli algoritmi di Floyd 3.3.1 e Knuth 3.3.2 che generino n-uple distinte invece di singoli valori.

A titolo di esempio descriviamo brevemente una generalizzazione dell'algoritmo di Floyd.

Supponiamo di voler generare un'istanza di uno schema $R(A_1, \ldots, A_m, B_1, \ldots, B_n)$ sottoposto ad un vincolo UNI: $(A_1, \ldots, A_m)$. I passi che l'algoritmo deve compiere per generare in modo distinto tuple di grandezza n sono:

- per $1 \leq i \leq m-1$, generare casualmente i valori di A_i scegliendoli da $dom_{\text{eff}}(A_i)$;
- ordinare i primi $m-1$ attributi in ordine lessicografico ottenendo così un partizionamento sui valori di tali attributi;
- per ogni partizione determinata al passo precedente si genera casualmente senza ripetizioni i valori di A_m scegliendoli da $dom_{\text{eff}}(A_m)$;
- infine, completare le tuple generando casualmente (con ripetizioni) i valori per ciascun B_j, con $1 \leq j \leq n$.

Esempio 17. Consideriamo di dover generare 5 triple di valori (per semplicità i valori sono di tipo numerico). Secondo quanto detto l'algoritmo procederà nel seguente modo:

- genererà in modo casuale i valori per i primi due attributi:

S		
1	2	-
2	1	-
1	3	-
2	3	-
1	2	-

- ordinerà in modo lessicografico i valori creando un partizionamento sugli stessi:

S		
1	2	-
1	2	-
1	3	-
2	1	-
2	3	-

- infine, per ogni partizione, genererà in modo distinto i valori del terzo attributo:

S		
1	2	1
1	2	2
1	3	1
2	1	1
2	3	1

Nel caso in cui si hanno più vincoli di unicità si distinguono tre sotto casi:

- i vincoli sono a due a due disgiunti;
- i vincoli sono applicati sullo stesso insieme di attributi;
- i vincoli sono applicati su insiemi di attributi differenti la cui intersezione è non vuota.

I primi due casi sono banali, infatti, rispettivamente nel primo i vincoli vengono trattati in modo indipendente mentre nel secondo i vincoli "collassano" in uno solo in quanto, di fatto, sono lo stesso vincolo.

Osservazione 4. D'ora in poi assumiamo, senza perdita di generalità, che più vincoli di unicità non possano essere specificati sullo stesso insieme di attributi.

Nel terzo caso bisogna invece considerare che, per ogni vincolo, esisteranno degli attributi che sono coinvolti in più vincoli di unicità. Formalmente dati due vincoli applicati sugli attributi X e Y si ha $X = X' \cup Z$ e $Y = Y' \cup Z$, dove X' e Y' sono gli insiemi di attributi che partecipano ad un solo vincolo di unicità e Z è l'insieme di attributi che partecipa ad entrambi i vincoli d'unicità.

L'approccio per la generazione consiste nel generare in modo univoco gli attributi di Z garantendo in questo modo il soddisfacimento di entrambi i vincoli di unicità; i rimanenti attributi di X e Y, cioè X' e Y' possono essere generati con l'Algoritmo 3.3.2.

La generalizzazione a casi in cui esistono più di due vincoli è banale, infatti per ogni vincolo esisterà sempre un sottoinsieme di attributi che partecipa a tutti i vincoli. Basterà generare generare in modo unico tale sottoinsieme.

Esempio 18. Consideriamo il seguente schema: INDIRIZZO(via, provincia, CAP) in cui sono applicati i vincoli di PK:(via, CAP) e UNI:(via, provincia). In tal caso verranno generati in modo unico dei valori per "via" garantendo così il soddisfacimento di entrambi i vincoli di unicità.

Come visto nell'esempio sopra però l'approccio proposto è restrittivo in quanto una "via" può essere presente in più città diverse (in cui il CAP e la provincia siano diversi).

4.2.2 Vincoli di chiave esterna

I vincoli di chiave esterna, come i vincoli di unicità, possono essere specificati sia su singoli attributi sia su insiemi di attributi. In entrambi i casi nella tabella riferita dal vincolo deve esistere un vincolo di unicità a livello di attributo o di insiemi di attributi.

Anche in questo caso la generazione di dati casuali che rispettino il vincolo d'integrità è relativamente semplice da gestire. Sia CE:$(X) \rightarrow S(Y)$ un vincolo d'integrità referenziale associato a un dato schema R. Allora, per ciascuna tupla t di un'istanza di R i valori necessari per costruire $t[X]$ devono essere recuperati dal dominio effettivo (si veda la Definizione 4.1) di Y in s, dove s è un'istanza (che assumiamo essere già stata creata) sullo schema S.

Si noti che non è necessario costruire esplicitamente il dominio effettivo di Y: per recuperare i dati da S è sufficiente generare opportuni indici alle righe di S [1] (dove per indice si intende un indice numerico i che denota l'i-esima riga della tabella S e non una struttura di indicizzazione di una base di dati).

Nel caso in cui si hanno più vincoli di chiave esterna si distinguono tre sotto casi:

- i vincoli sono a due a due disgiunti;
- i vincoli sono applicati sullo stesso insieme di attributi;
- i vincoli sono applicati su insiemi di attributi differenti la cui intersezione è non vuota.

[1] Sfruttiamo il fatto che, a differenza delle relazioni del modello relazionale, in SQL esiste un ordinamento implicito delle righe di una tabella, per cui ha senso parlare dell'"i-esima riga" di una tabella.

Il primo caso è banale, infatti basta considerare ogni vincolo in modo indipendente.

Il secondo caso invece richiede di istanziare valori che siano presenti in tutte le tabelle riferite da ciascun vincolo. Formalmente dati due vincoli di chiave esterna su una tabella R, CE:$(X_1, \ldots, X_n)\rightarrow$S$(V_1, \ldots, V_n)$ e CE:$(Y_1, \ldots, Y_n)\rightarrow$T$(W_1, \ldots, W_n)$ tali che $X = Y$ è sufficiente generare i valori prendendoli dall'intersezione dei domini effettivi di V e W, rispettivamente rispetto a un'istanza di S e a un'istanza di T. In tal modo ci assicuriamo di inserire dei valori che sono presenti in entrambe le tabelle riferite.

Nel terzo caso bisogna invece considerare che, per ogni vincolo, esisteranno degli attributi che sono coinvolti in più vincoli di chiave esterna. Formalmente dati due vincoli di chiave esterna su una tabella R, CE:$(X_1, \ldots, X_n)\rightarrow$S$(V_1, \ldots, V_n)$ e CE:$(Y_1, \ldots, Y_n)\rightarrow$T$(W_1, \ldots, W_n)$, in cui $X = X' \cup Z$ e $Y = Y' \cup Z$ dove:

- X' e Y' sono gli insiemi di attributi che partecipano ad un solo vincolo di chiave esterna;
- Z è quel sottoinsieme di attributi, in comune fra le due tabelle, che partecipa ad entrambi i vincoli di chiave esterna;
- V e W sono gli attributi riferiti dalle due chiavi esterne rispettivamente delle tabelle S e T.

Per la generazione dei dati si può operare nel seguente modo:

- mediante un'operazione di *JOIN* si determina il dominio effettivo J degli attributi $X_1, \ldots, X_n$ e $Y_1, \ldots, Y_n$:

  $$J = (\pi_{V_1,\ldots,V_n}(S)) \bowtie_{X'=Y'} (\pi_{W_1,\ldots,W_n}(T))$$

  così facendo si determinano tutti gli attributi che sono presenti sia nella tabella S sia nella tabella T e tali che le tuple coincidano sugli attributi di X' e Y'.

- si generano i valori partendo dal dominio effettivo J;

La generalizzazione a casi con più di due vincoli è banale, l'unica differenza è che avremo la necessità di fare rispettivamente un'intersezione e un *JOIN* su più di due tabelle.

4.2.3 Un solo vincolo di unicità e di chiave esterna su un singoletto

Consideriamo il caso in cui si abbia un solo vincolo per tipo e ciascun vincolo è applicato ad un singolo attributo; si distinguono due sotto casi:

- i due vincoli sono applicati a due attributi distinti;
- i due vincoli sono applicati allo stesso attributo.

Il primo caso è di semplice gestione perché i vincoli operano su insiemi disgiunti di attributi, ci si può perciò ricondurre ai casi discussi nelle Sezioni 4.2.1 e 4.2.2. Ciascun vincolo verrà trattato in modo indipendente dall'altro.

Per illustrare in che modo possiamo trattare il secondo caso ci serviamo di un esempio:

Esempio 19. Consideriamo il seguente schema: R(A,B,C) con un vincolo PK:(A) e CE:(A)→S(D), S(D,E,F) con un vincolo PK:(D). Viene quindi impostato un vincolo di unicità e di integrità referenziale sugli stessi attributi.

Un possibile approccio consiste nel recuperare i dati dell'attributo riferito al più una volta; i due vincoli sono rispettati, infatti:

- il vincolo di chiave esterna è garantito in quanto i valori sono stati presi fra quelli presenti nell'attributo riferito;
- il vincolo di unicità viene garantito perché i valori presi al passo precedente sono stati presi al più una volta, e nella tabella riferita sussiste un vincolo di unicità su quell'attributo (condizione necessaria per poter definire la chiave esterna).

4.2.4 Un solo vincolo di unicità e di chiave esterna su insiemi arbitrari

Iniziamo ora a trattare casi più generali in cui si ammettono vincoli su insiemi di cardinalità arbitraria pur ammettendo solo un vincolo per tipo; si distinguono cinque sotto casi (si veda la Figura 4.2):

a. i due insiemi risultano essere disgiunti;
b. i due insiemi sono identici;
c. il vincolo di unicità è incluso propriamente nel vincolo di chiave esterna;

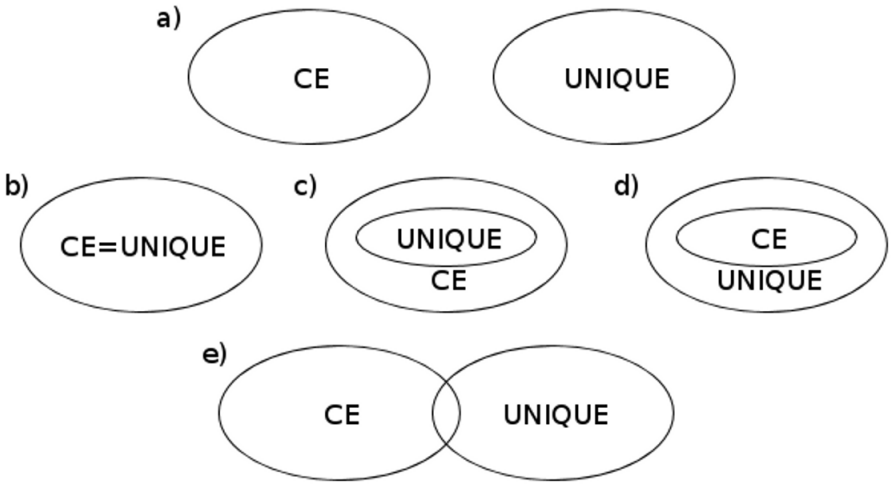

Figura 4.2: Tipologie di vincoli: un vincolo per tipo

d. il vincolo di chiave esterna è incluso propriamente nel vincolo di unicità;

e. i due insiemi presentano un'intersezione non vuota ma non sono contenuti uno nell'altro.

I primi due casi sono simili a quelli già trattati:

a) i vincoli possono essere trattati in modo indipendente con gli approcci visti nelle Sezioni 4.2.1 e 4.2.2;

b) bisogna recuperare, dalla tabella riferita, ogni tupla al più una volta con l'approccio visto nella Sezione 4.2.3.

Il terzo caso è invece fra i più delicati perché il vincolo d'integrità referenziale restringe il dominio dei valori ammissibili a quello dei valori presenti nelle tuple riferite. Per questo motivo il numero di tuple inseribili nella tabella potrebbe essere molto basso.

Per esporre meglio il concetto ci serviamo del seguente esempio:

Esempio 20. Consideriamo il seguente schema: R(A,B,C) con vincoli PK:(A,B) e CE:(A,B,C)→S(D,E,F), S(D,E,F) con un vincolo PK:(D,E,F). Un'istanza valida per S è:

S		
D	E	F
1	1	1
1	1	2
1	1	3
1	1	4

Come si può notare il dominio effettivo degli attributi (C, D) di S è $\{(1, 1)\}$. La tabella R potrà quindi contenere un solo record.

L'algoritmo deve, durante la generazione dei dati del vincolo di unicità, prendere in considerazione i valori presenti negli attributi riferiti e generare, a partire da questi, dei valori distinti.

Il quarto caso è invece più semplice da gestire; un possibile approccio può eseguire le seguenti operazioni:

- si determinano innanzitutto i domini effettivi per gli attributi sottoposti al vincolo di chiave esterna;
- si passa a considerare gli attributi coinvolti nel vincolo d'unicità ma non nel vincolo d'integrità referenziale (ve n'è certamente almeno uno): i valori per le n-uple di attributi, partecipanti al vincolo d'unicità ma non al vincolo d'integrità referenziale, possono essere generati applicando la generalizzazione dell'Algoritmo di Floyd discussa nella Sezione 4.2.1.

Infine, per il caso e), si procede come nel caso precedente, infatti esiste almeno un attributo nel vincolo di unicità che non è sottoposto ad alcun vincolo di chiave esterna.

Osservazione 5. Nella generazione dei dati unici si possono generare in modo unico anche gli attributi che compongono la chiave esterna, generando invece in modo standard (con ripetizioni) gli altri attributi; le proprietà richieste sono ugualmente soddisfatte.

Osservazione 6. Un'euristica che consente una maggiore flessibilità nella generazione dei dati consiste nel generare in modo unico quella parte dell'insieme degli attributi sottoposti a dei vincoli di chiave esterna che ha il dominio (effettivo) di cardinalità maggiore; in questo modo si possono generare più record per quella tabella.

Esempio 21. Consideriamo il seguente schema: R(A,B,C) con vincoli PK:(A,B) e CE:(B,C)→S(E,F), S(D,E,F) con vincolo PK:(E,F). Un'istanza valida per S è:

S		
D	E	F
1	1	1
2	1	2
3	1	3
4	1	4

Se i valori della tabella R fossero generati imponendo il vincolo di unicità sulla chiave esterna avremo che potremo inserire un solo valore; ottenendo questa istanza:

R		
A	B	C
1	1	1

Ma se invece fossero generati imponendo il vincolo di unicità sull'attributo A si potranno generare più record perché il vincolo di unicità su (A, B) viene garantito dall'unicità di A e quindi B può contenere anche valori tutti uguali; si può ottenere quindi un'istanza con più record simile a questa:

R		
A	B	C
1	1	1
2	1	2
3	1	2
4	1	3

4.2.5 Più vincoli di unicità e di chiave esterna su insiemi arbitrari

Trattiamo ora il caso più generale di tutti in cui ammettiamo un numero arbitrario di vincoli sottoposti ad un numero arbitrario di attributi. Un esempio di questo caso è stato proposto all'inizio del presente capitolo (si veda Esempio 16).

Vediamo ora come ci possiamo ricondurre ai casi precedenti trattando, nell'ordine, prima i vincoli di chiave esterna e poi quelli di unicità.

Un possibile approccio opera nel seguente modo:

- si trovano i domini effettivi degli attributi $X_1, \ldots, X_n$ coinvolti nei vincoli di chiave esterna come visto nella Sezione 4.2.2;
- ora si possono "ignorare" i vincoli di chiave esterna in quanto i domini effettivi degli attributi sono determinati in modo da rispettare tali vincoli;
- ora si opera come visto nella Sezione 4.2.1 per generare i dati unici **partendo** dai domini effettivi trovati al primo passo.

Esempio 22. Riferendoci allo schema presentato all'inizio del capitolo (vedi Esempio 16) analizziamo i passi con cui si procede nella generazione dei dati.

- i valori per le tabelle S e T vengono generati come descritto nella Sezione 4.2.1 in quanto non sono presenti vincoli di integrità referenziale. Supponendo che ogni attributo abbia dominio nei numeri interi delle istanze valide sono:

  Per la tabella S:

S		
A	B	C
1	2	3
1	3	2
2	1	4

  Per la tabella T:

T		
C	D	E
2	3	2
3	4	2
4	4	3
4	5	3
5	5	4

- ora bisogna calcolare i domini effettivi degli attributi della tabella R su cui sussistono due vincoli di integrità referenziale; domini che, in questo caso, sono i valori delle tabelle S e T. Come descritto nella Sezione 4.2.2 troviamo i possibili valori che possono assumere le tuple per verificare i due vincoli:

A	B	C	D	E
1	2	3	4	2
1	3	2	3	2
2	3	4	4	3
2	1	4	5	3

- infine selezioniamo le tuple in modo da soddisfare i due vincoli di unicità sugli attributi (A, B) e (B, C, D) ottenendo la seguente istanza:

R				
A	B	C	D	E
1	2	3	4	2
1	3	2	3	2
2	1	4	5	3

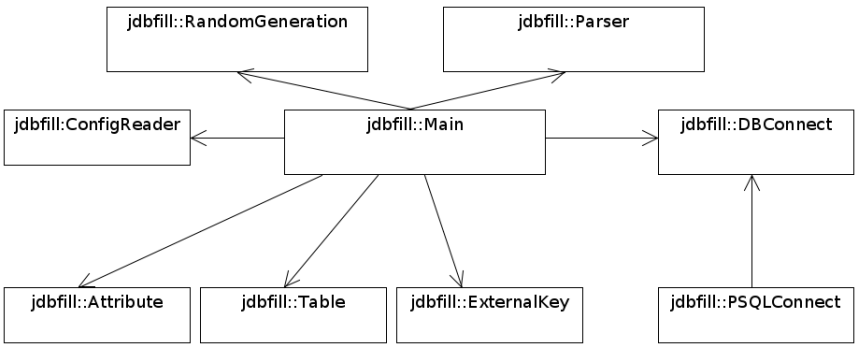

Figura 4.3: Diagramma UML

4.3 Classi

Gli algoritmi presentati nelle Sezioni 3.3.1, 3.3.2, 3.3.4, 3.4.1, 3.4.2 e 3.4.3 sono stati implementati in un programma Java che popola una base di dati con valori generati casualmente. La struttura del programma è sintetizzata nel diagramma UML della Figura 4.3. Nel dettaglio sono state realizzate le seguenti classi:

- *Main*: il punto di partenza del programma;
- *Table*: rappresenta una tabella della base di dati;
- *Attribute*: rappresenta un attributo della base di dati;
- *ForeignKey*: rappresenta una chiave esterna;
- *DBConnect*: definisce un'interfaccia pubblica per la connessione ad una base di dati;
- *PSQLConnect*: specializza *DBConnect* e implementa i metodi per la connessione ad una base di dati *Postgres*;
- *RandomGeneration*: implementa alcuni degli algoritmi descritti nel Capitolo 3;
- *ConfigReader*: implementa la gestione del file di configurazione;
- *Parser*: implementa i metodi per il *parsing* delle opzioni.

Nelle sezioni seguenti andremo a descrivere ciascuna delle classi del diagramma UML in Figura 4.3 presentando per ognuna i metodi principali. Per una documentazione approfondita si rimanda alla documentazione allegata al codice sorgente.

4.3.1 Main

La classe Main è il punto di partenza dell'applicativo; contiene i metodi per:

- effettuare, nell'ordine appropriato, le operazioni richieste per il popolamento della base di dati;
- i metodi per la generazione dei dati che andranno a richiamare opportunamente i metodi della classe *RandomGeneration*;
- alcuni metodi di supporto, come ad esempio: stampa dell'aiuto utente e stampa di informazioni di debug.

Le operazioni "tipo" svolte dal metodo di lavoro sono, nell'ordine, le seguenti:

- connessione alla base di dati;
- reperimento delle informazioni sulla base di dati (mediante interrogazioni sul *Information Schema*);
- generazione dei dati da inserire;
- cancellazione dei dati preesistenti (previa conferma da parte dell'utente);
- inserimento dei nuovi dati nella base di dati.

Vista la complessità nella gestione di alcuni vincoli l'applicativo è in grado di funzionare solo sotto le seguenti condizioni:

- in presenza di vincoli d'integrità referenziale e d'unicità che operano su attributi comuni, il vincolo di chiave esterna **deve** essere incluso nel vincolo di unicità;
- a uno stesso attributo **non** possono essere associati più vincoli di integrità referenziale;
- ogni vincolo di unicità **deve** avere almeno un attributo che non sia coinvolto in un vincolo di chiave esterna.

4.3.2 Table

La classe Table (Figura 4.4) rappresenta una **tabella** SQL. Ogni tabella è identificata dal proprio nome e contiene:

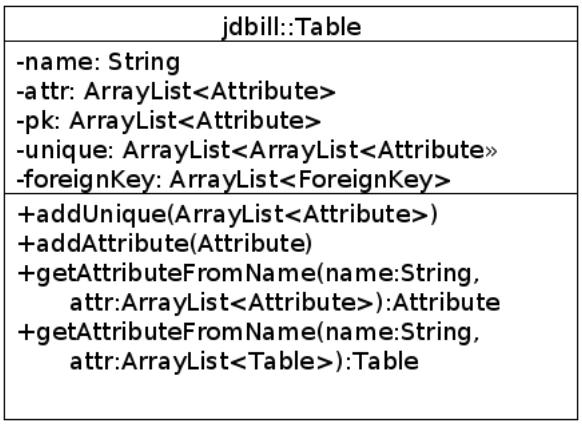

Figura 4.4: Diagramma UML di Table

- una lista degli attributi che la compongono;
- la chiave primaria implementata come una lista di attributi;
- gli eventuali vincoli di unicità implementati come una lista di liste di attributi;
- gli eventuali vincoli di chiave esterna implementati come una lista di vincoli di chiave esterna "elementari". Oltre ai metodi per accedere ai campi ("get-set") troviamo due metodi che consentono:
  - dato il nome di una tabella di ottenere il suo riferimento a oggetto;
  - dato il nome di un attributo della tabella di ottenere il riferimento all'oggetto.

4.3.3 Attribute

La classe Attribute (Figura 4.5) rappresenta un **attributo**, identificandone:

- il nome e il tipo;
- gli eventuali vincoli: **not null**, **unique** e **default**;
- i valori che sono stati generati dall'applicativo.

Sono quindi presenti gli usuali metodi per accedere ai campi ("get-set").

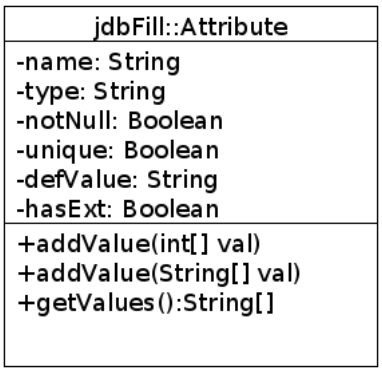

Figura 4.5: Diagramma UML di Attribute

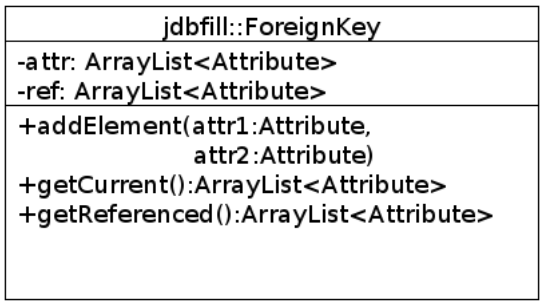

Figura 4.6: Diagramma UML di ForeignKey

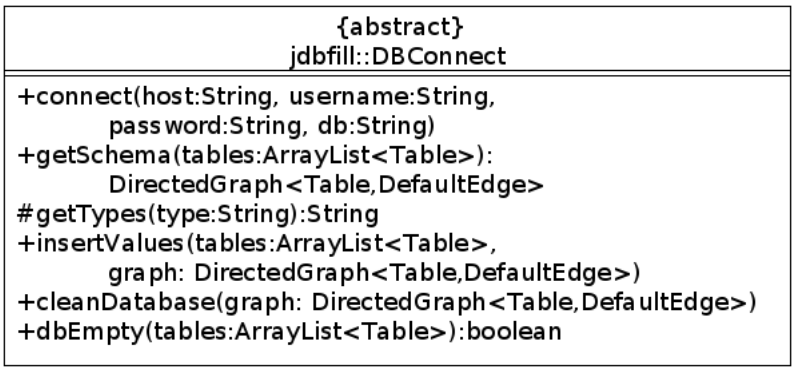

Figura 4.7: Diagramma UML di DBConnect

4.3.4 ForeignKey

La classe ForeignKey (Figura 4.6) rappresenta un vincolo di **chiave esterna**; ogni chiave esterna è composta da una lista di attributi che si riferisce ad un'altra lista di attributi di un'altra tabella. Sono quindi presenti gli usuali metodi per inserire degli elementi alla chiave esterna e per ricavare la lista di attributi coinvolti e la lista di attributi riferiti.

4.3.5 DBConnect

La classe astratta DBConnect (Figura 4.7) implementa i metodi di base per la connessione e l'interazione con la base di dati. I metodi che dovranno essere implementati dalle classi che estenderanno DBConnect sono:

- *connect(host, username, password)*: per la connessione a uno specifico DBMS;
- *getSchema(tables)*: inserisce in una lista di tabelle tutte le informazioni sulla base di dati da popolare;
- *getTypes(type)*: effettua un mapping fra i tipi della base di dati e quelli standard di Java;
- *insertValue(tables, graph)*: inserisce nella base di dati i dati generati nell'ordine dettato dall'ordinamento topologico del grafo rappresentante le dipendenze di integrità;
- *cleanDatabase(tables, graph)*: elimina tutti i dati dalla base di dati nell'ordine dettato dall'inverso dell'ordinamento topologico del grafo;

- *dbEmpty(tables)*: verifica se la base di dati contiene dei dati.

Introduciamo ora un semplice schema di base di dati che utilizzeremo in tutti gli esempi della prossima sezione formato dalle seguenti tabelle:

- IMPIEGATI: con attributi nome, cognome, **codice fiscale** e stipendio;
- DIPARTIMENTI: con attributi **numero**, nomeDip e *manager*.

in cui in grassetto sono riportate le chiavi primarie e in corsivo le chiavi esterne. Sono presenti i seguenti vincoli:

- su IMPIEGATI: UNI:(nome, cognome)
- su DIPARTIMENTI: UNI:(nome) e CE:(manager)→IMPIEGATI(codice fiscale)

4.3.6 Interrogazioni

Riportiamo ora le interrogazioni che ci consentono di ottenere le informazioni sulla base di dati. Le interrogazioni sono state implementate nella classe PostgreSQL.

Per ottenere tutte le tabelle dell'utente:

```
SELECT Table_Name
FROM INFORMATION_SCHEMA.TABLES
WHERE TABLE_TYPE = 'BASE TABLE'
      AND Table_Name NOT LIKE 'pg_%'
      AND Table_Name NOT LIKE 'sql_%'
```

L'interrogazione seleziona il nome di tutte le tabelle presenti nella vista *TABLES* dell'Information Schema (vedi Sezione 1.3), ad esclusione delle tabelle di sistema che vengono ignorate grazie alla clausola *NOT LIKE*.

Esempio 23. Nello schema considerato l'interrogazione restituirà:

IMPIEGATI
DIPARTIMENTI

Per ogni tabella otteniamo la lista dei suoi attributi:

```
SELECT table_name, column_name
FROM information_schema.columns
WHERE table_schema <> 'information_schema'
      AND table_schema <> 'pg_catalog'
```

L'interrogazione seleziona le colonne table_name e column_name della tabella *columns* dell'Information Schema (vedi Sezione 1.3) ottenendo in questo modo i nomi di tutti gli attributi; i tipi degli attributi si possono ottenere mediante una chiamata alla libreria per la gestione della connessione alla base di dati (oppure aggiungendo nella clausola *SELECT* l'attributo "data_type"). Le due condizioni nella clausola *WHERE* servono a escludere dal risultato le colonne (e le tabelle) degli schemi di sistema.

Se si vuole conoscere gli attributi di una sola tabella è sufficiente aggiungere un ulteriore vincolo nella clausola *WHERE*:

```
SELECT table_name, column_name
FROM information_schema.columns
WHERE table_schema <> 'information_schema'
      AND table_schema <> 'pg_catalog'
      AND table_name = 'name'
```

Esempio 24. Nello schema considerato l'interrogazione restituirà:

Per impiegati:

IMPIEGATI	nome
IMPIEGATI	cognome
IMPIEGATI	codice fiscale
IMPIEGATI	stipendio

Per dipartimenti:

DIPARTIMENTI	numero
DIPARTIMENTI	nomeDip
DIPARTIMENTI	manager

Per ogni tabella otteniamo la **chiave primaria**:

```
SELECT pg_attribute.attname
FROM pg_index, pg_class, pg_attribute
WHERE pg_class.oid = 'table_name'::regclass
      AND indrelid = pg_class.oid
      AND pg_attribute.attrelid = pg_class.oid
      AND pg_attribute.attnum = any(pg_index.indkey)
      AND indisprimary
```

L'interrogazione seleziona il nome degli attributi che concorrono alla chiave primaria della tabella. Ciò si ottiene attraverso la concatenazione di tre tabelle (vedi Sezione 1.3):

- *pg_index* che contiene le informazioni sugli indici;
- *pg_class* che contiene la lista degli identificativi (gli OID);
- *pg_attribute* che contiene le informazioni sugli attributi quale in nome.

L'operatore *::regclass* permette di passare dall'OID al nome della tabella.

Esempio 25. Nello schema considerato l'interrogazione restituirà: Per impiegati:

codice fiscale

Per dipartimenti:

numero

La seguente interrogazione permette di ottenere tutti i vincoli di **chiave esterna**:

```
SELECT pg.conname, pg.conkey, cl.relname, pg.confkey, cl2.relname
FROM pg_constraint as pg, pg_class as cl, pg_class as cl2
WHERE pg.contype='f' and cl.oid=pg.conrelid and cl2.oid=pg.confrelid
```

L'interrogazione effettua un *JOIN* fra tre tabelle (di cui due sono la stessa) in modo da ottenere per ogni tipo di vincolo *FOREIGN KEY* le due tabelle coinvolte. Viene quindi restituito un insieme che rappresenta gli attributi coinvolti nel vincolo di chiave esterna.

Esempio 26. Nello schema considerato l'interrogazione restituirà:

nome_vincolo	{3}	DIPARTIMENTI	{3}	IMPIEGATI

In cui il secondo e terzo valore rappresentano rispettivamente la posizione degli attributi coinvolti nel vincolo e la tabella su cui è sottoposto il vincolo, mentre il quarto e quinto valore rappresentano rispettivamente la posizione degli attributi riferiti dal vincolo e la tabella riferita dal vincolo.

Queste due interrogazioni servono rispettivamente per ottenere i vincoli di **unicità** a livello di attributo e di insieme di attributi:

```
SELECT DISTINCT tc.table_name, kcu.column_name
FROM information_schema.table_constraints AS tc
     JOIN information_schema.key_column_usage AS kcu
     ON tc.constraint_name = kcu.constraint_name
WHERE constraint_type='UNIQUE'
```

Esempio 27. Nello schema considerato l'interrogazione restituirà:

DIPARTIMENTI	nomeDip

```
SELECT pg.conkey, cl.relname
FROM pg_constraint as pg, pg_class as cl
WHERE pg.contype='u' and cl.oid=pg.conrelid
```

Esempio 28. Nello schema considerato l'interrogazione restituirà:

{1,2}	IMPIEGATI

In cui {1,2} rappresentano gli indici degli attributi coinvolti nel vincolo di unicità (in tal caso "nome" e "cognome").

Queste due ultime interrogazioni ricavano i vincoli di **not null** e **default** (attualmente non implementati):

```
SELECT table_name, column_name, column_default
FROM information_schema.columns
WHERE column_default <> '' and table_schema <> 'pg_catalog'
```

```
SELECT table_name, column_name
FROM information_schema.columns " +
WHERE is_nullable = 'NO' and table_schema <> 'pg_catalog'
```

Entrambe le interrogazioni sono simili, infatti selezionano il nome della tabella e dell'attributo dalla vista *COLUMNS* dell'Information Schema andando a selezionare solo quelle tuple che hanno un valore di default (e in tal caso lo restituiscono) oppure che non ammettono valori nulli. L'istruzione "table_schema <> pg_catalog" serve ad escludere dal risultato i vincoli che appartengono al catalogo di sistema.

Esempio 29. Nello schema considerato entrambe le interrogazioni non restituiranno nulla in quanto non ci sono ne valori di default ne vincoli not null.

4.3.7 RandomGeneration

La classe RandomGeneration (Figura 4.8) implementa gli algoritmi descritti nel Capitolo 3, ossia:

- gli Algoritmi 3.3.1 e 3.3.2 per il campionamento di numeri senza ripetizioni;

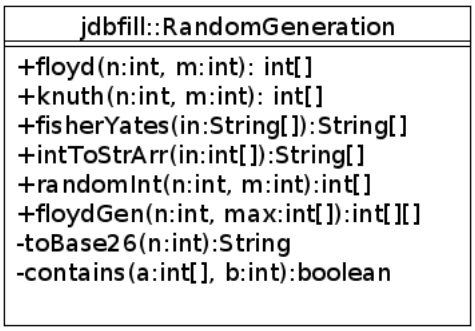

Figura 4.8: Diagramma UML di RandomGeneration

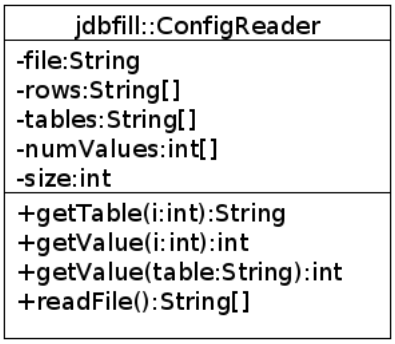

Figura 4.9: Diagramma UML di ConfigReader

- l'Algoritmo 3.3.3 per effettuare una permutazione casuale degli elementi di un insieme;
- una generalizzazione dell'algoritmo di *Floyd* per generare tuple distinte.

Sono stati inoltre implementati:

- algoritmi per il campionamento di numeri con ripetizioni;
- algoritmi per effettuare il *data-mapping* fra il tipo intero ed altri tipi (es. stringhe); descritti nella Sezione 3.4
- metodi di supporto agli algoritmi della classe (es. *contains* per *Floyd*).

4.3.8 ConfigReader

La classe ConfigReader (Figura 4.9) implementa i metodi per la gestione del file di configurazione. Il file, creato dall'utente, contiene le informazioni sul numero di

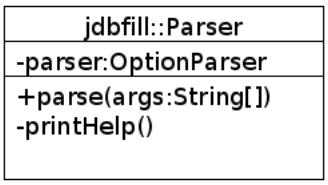

Figura 4.10: Diagramma UML di Parser

record che si vogliono inserire in ciascuna tabella. I controlli di consistenza dell'input vengono relegati ai metodi di generazione dei dati. Il programma attua ad esempio le seguenti verifiche:

- se una tabella ha un vincolo di chiave esterna verso una seconda tabella e si vuole che i dati generati siano unici bisogna verificare che la seconda tabella contenga almeno tanti record quanti se ne vogliono generare;
- se si richiede di generare, sempre in modo univoco, più dati di quello che il dominio effettivo dell'attributo permette non si può procedere e bisogna restituire un errore.

4.3.9 Parser

La classe Parser (Figura 4.10) implementa i metodi per il *parsing* delle opzioni a riga di comando.

4.4 Note per l'esecuzione

Una volta compilato l'applicativo dai sorgenti, utilizzando lo script fornito, è possibile eseguirlo con il seguente comando:

```
java -jar jdbfill.jar --server <hostname> --db <name> --username <name>
                      [--password <pwd>] --type <dbtype> [--reset]
                      [--input-file <filepath>]
```

Le opzioni disponibili sono:

- `--server <hostname>`: l'indirizzo del server che ospita la base di dati;
- `--db <name>`: il nome della base di dati che si vuole utilizzare;
- `--username <name>`: il nome utente dell'amministratore della base di dati;

- `--password <pwd>`: la password dell'utente;
- `--type`: il tipo della base di dati a cui ci si sta connettendo ('postgresql');
- `--reset`: forza la cancellazione di tutti i dati presenti nella base di dati (all'utente verrà richiesta un'ulteriore conferma);
- `--input-file <filepath>`: il percorso del file di input;
- `--help`: stampa a video la lista delle opzioni disponibili.

Qualora si selezioni l'opzione *reset* l'applicativo chiederà un'ulteriore conferma all'utente in quanto l'operazione di cancellazione dei record dalla base di dati è **distruttiva**. Se l'utente non specifica un file di input l'applicativo tenta di generare in modo automatico 100 record per tabella.

5

Sperimentazioni

In questo capitolo si riporteranno alcuni risultati sperimentali sull'esecuzione dell'applicativo descritto nel Capitolo 4. I test sono stati condotti nel seguente ambiente operativo:

- computer portatile con processore dual-core avente frequenza di 2.4 Ghz e 4 GB di memoria RAM;
- sistema operativo ArchLinux (versione kernel 3.6) 64 bit;
- versione dell'ambiente Java 1.7;
- versione del software statistico "R" [19] 2.15.1;
- versione della libreria di test "dieharder" [20] 3.31.1.

Osservazione 7. Si vuol far notare che l'applicativo non supporta il multi-threading e quindi utilizza realmente uno solo dei due core a disposizione.

Le sperimentazioni effettuate ricadono in tre categorie:

- analisi dei tempi di esecuzione degli algoritmi implementati nell'applicativo;
- test statistici sulle proprietà di casualità degli algoritmi implementati nell'applicativo;
- test di popolamento di basi di dati.

5.1 Analisi degli algoritmi

Il test sugli algoritmi mira a verificare sperimentalmente le complessità degli stessi, che ricordiamo essere:

- per l'Algoritmo di Floyd 3.3.1 di $O(m^2)$;
- per l'Algoritmo di Knuth 3.3.2 di $O(n)$;
- per l'Algoritmo per il campionamento di dati (con ripetizioni) 3.2.2 di $O(m)$;
- per l'Algoritmo per il campionamento di dati senza ripetizioni 3.3 di $O(m^2)$;

in cui n rappresenta il range di valori da cui si vanno ad estrarre m valori.

Valuteremo inoltre anche i tempi di esecuzione della generalizzazione dell'Algoritmo di Floyd discussa nella Sezione 4.2.1.

La procedura per la misurazione dei tempi d'esecuzione, in millisecondi, esegue un dato algoritmo per t volte; la ripetizione per t volte ha lo scopo di:

- rilevare tempi inferiori al minimo tempo misurabile (inferiore ad alcune decine di millisecondi);
- ridurre l'interferenza dovuta a eventuali fattori esterni quali il carico del sistema o l'intervento di politiche di recupero memoria ("garbage collection") su più misurazioni.

Nelle Figure da 5.1 a 5.14 riportiamo due serie di grafici in cui rispettivamente:

- fissiamo n a:

  $$\{1.000.000, 10.000.000, 100.000.000, 1.000.000.000\}$$

  e variamo m fra i seguenti valori:

  $$\{1.000, 2.500, 5.000, 7.500, 10.000, 25.000, 50.000, 75.000, 100.000\}$$

- fissiamo m a:

  $$\{1.000, 10.000, 100.000\}$$

  e variamo n fra i seguenti valori:

  $$\{1.000.000, 2.500.000, 5.000.000, 7.500.000, 10.000.000\}$$

  $$\{10.000.000, 25.000.000, 50.000.000, 75.000.000, 100.000.000\}$$

  $$\{100.000.000, 250.000.000, 500.000.000, 750.000.000, 1.000.000.000\}$$

Al termine degli stessi faremo alcune importanti considerazioni sull'uso degli algoritmi proposti.

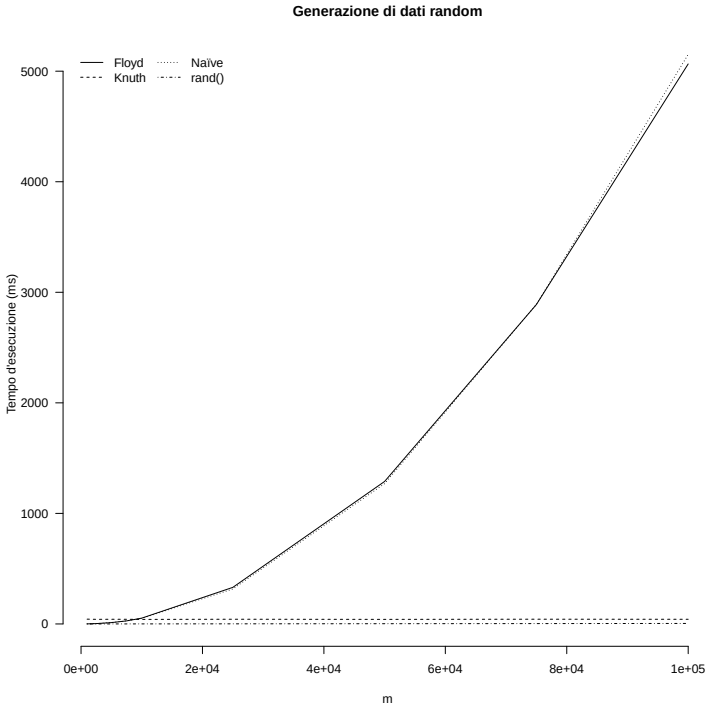

Figura 5.1: Tempi di esecuzione fissato $n = 1.000.000$ e variando m

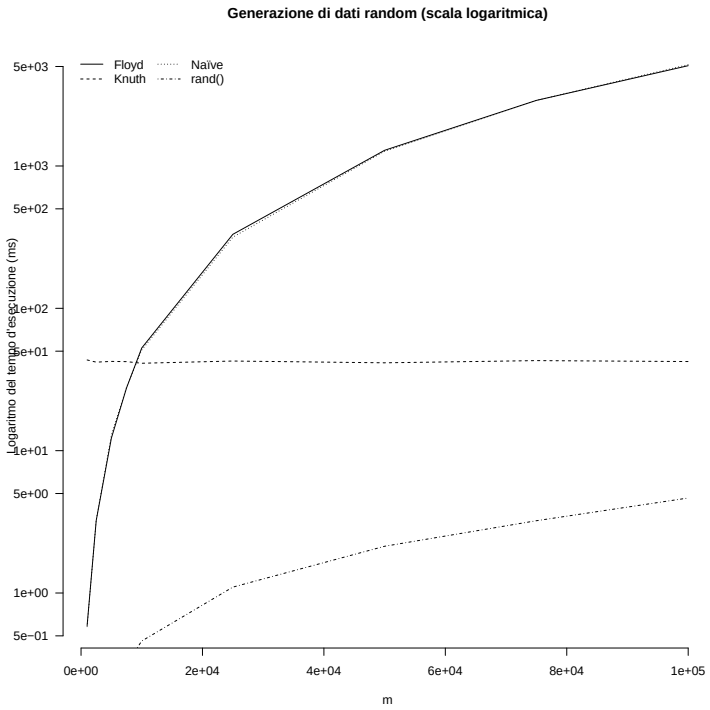

Figura 5.2: Tempi di esecuzione fissato $n = 1.000.000$ e variando m (scala logaritmica)

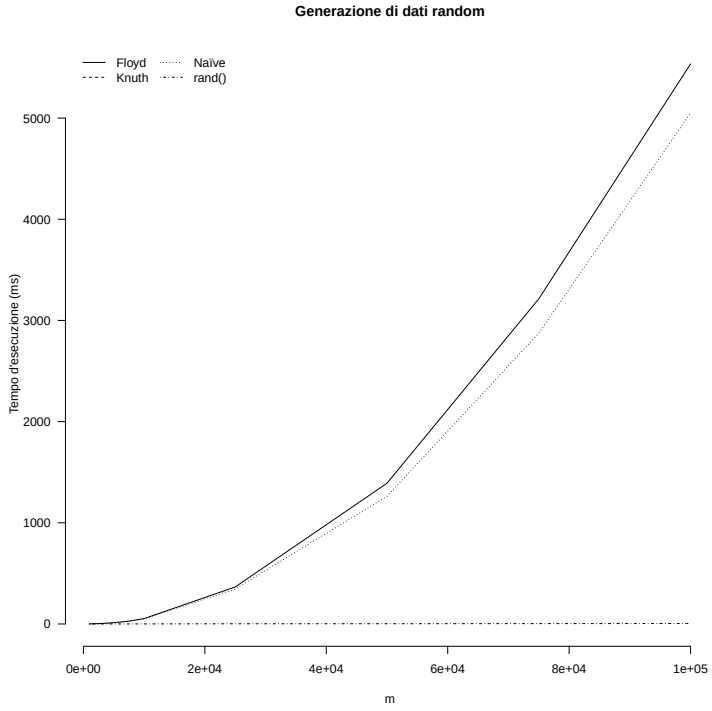

Figura 5.3: Tempi di esecuzione fissato $n = 10.000.000$ e variando m

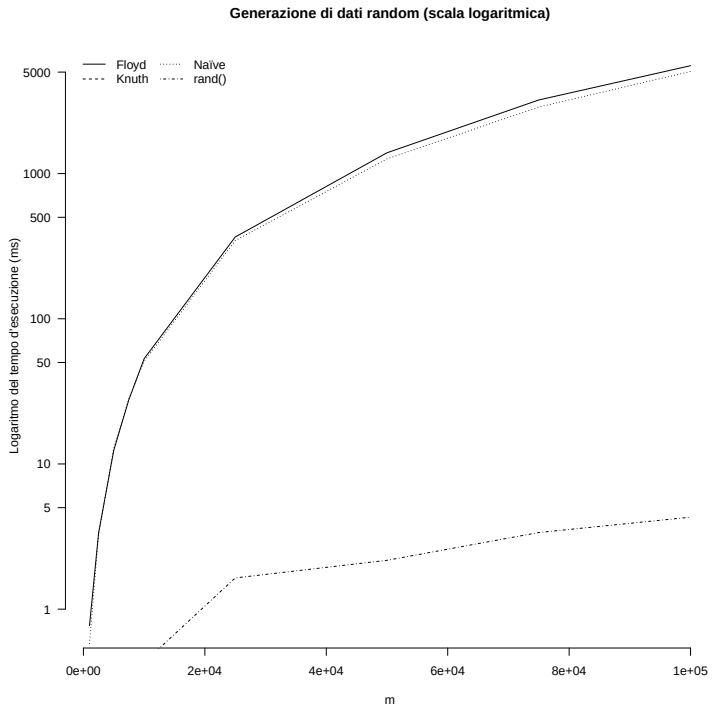

Figura 5.4: Tempi di esecuzione fissato $n = 10.000.000$ e variando m (scala logaritmica)

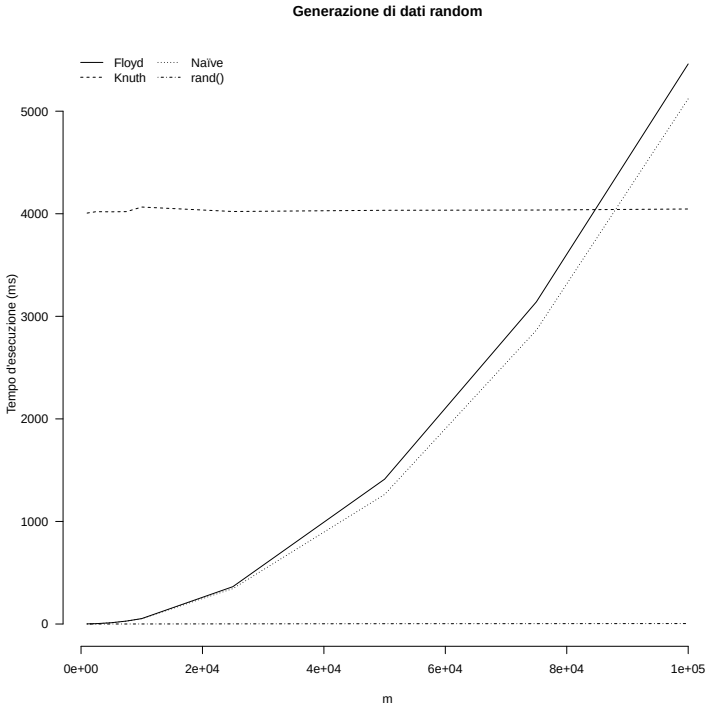

Figura 5.5: Tempi di esecuzione fissato $n = 100.000.000$ e variando m

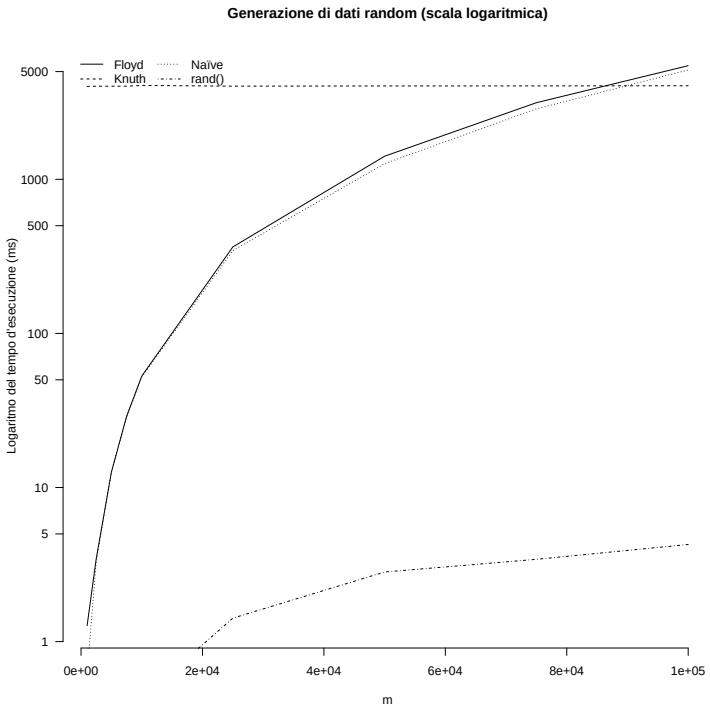

Figura 5.6: Tempi di esecuzione fissato $n = 100.000.000$ e variando m (scala logaritmica)

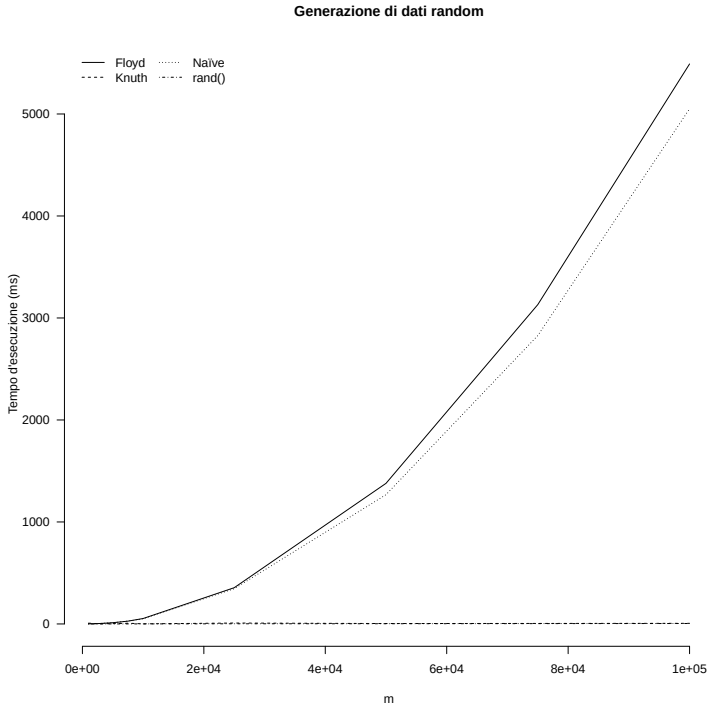

Figura 5.7: Tempi di esecuzione fissato $n = 1.000.000.000$ e variando m

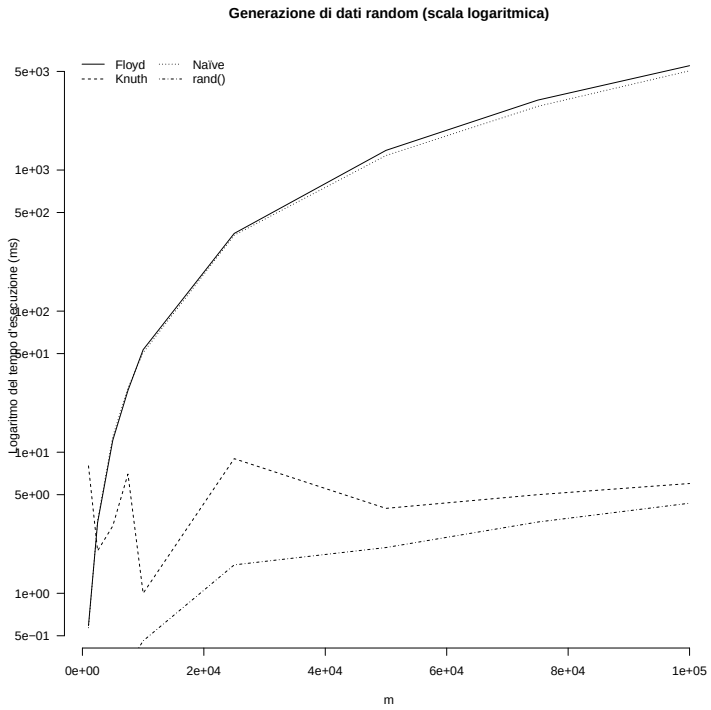

Figura 5.8: Tempi di esecuzione fissato $n = 1.000.000.000$ e variando m (scala logaritmica)

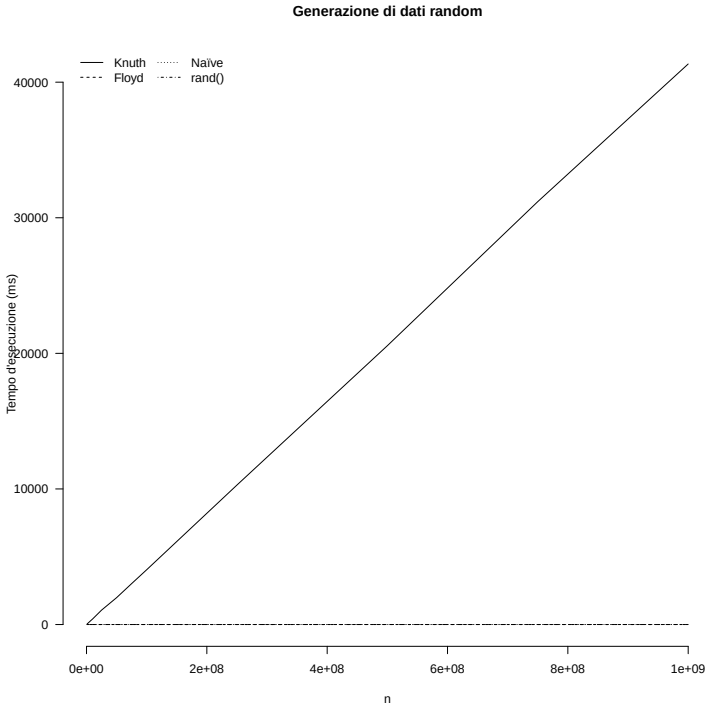

Figura 5.9: Tempi di esecuzione fissato $m = 1.000$ e variando n

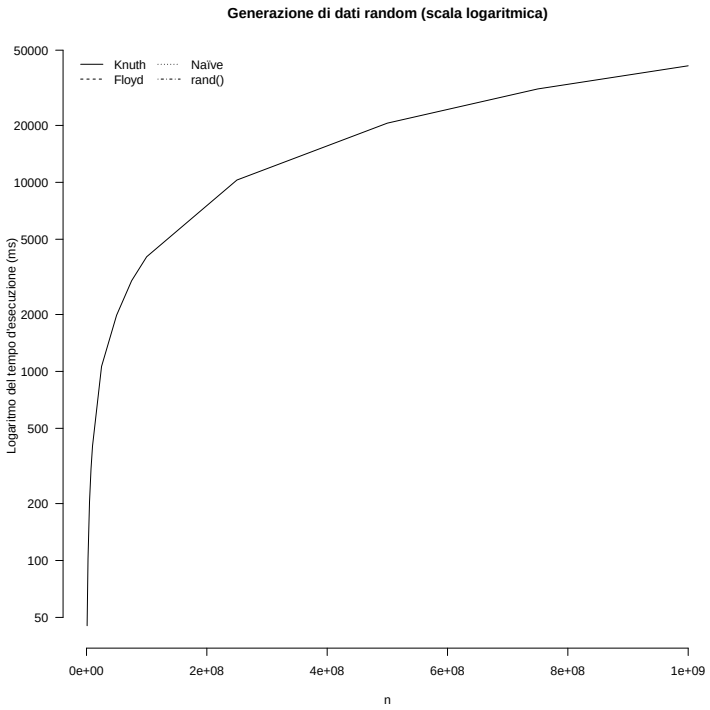

Figura 5.10: Tempi di esecuzione fissato $m = 1.000$ e variando n (scala logaritmica)

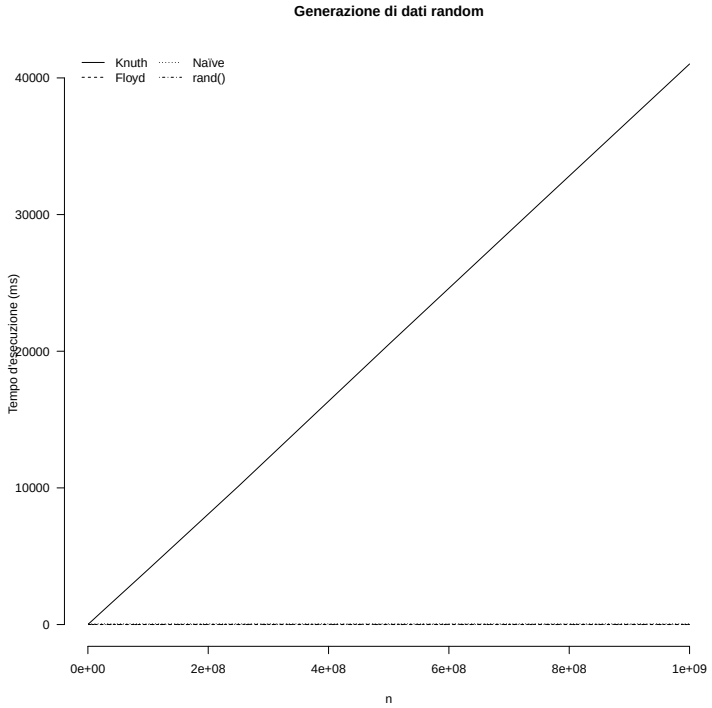

Figura 5.11: Tempi di esecuzione fissato $m = 10.000$ e variando n

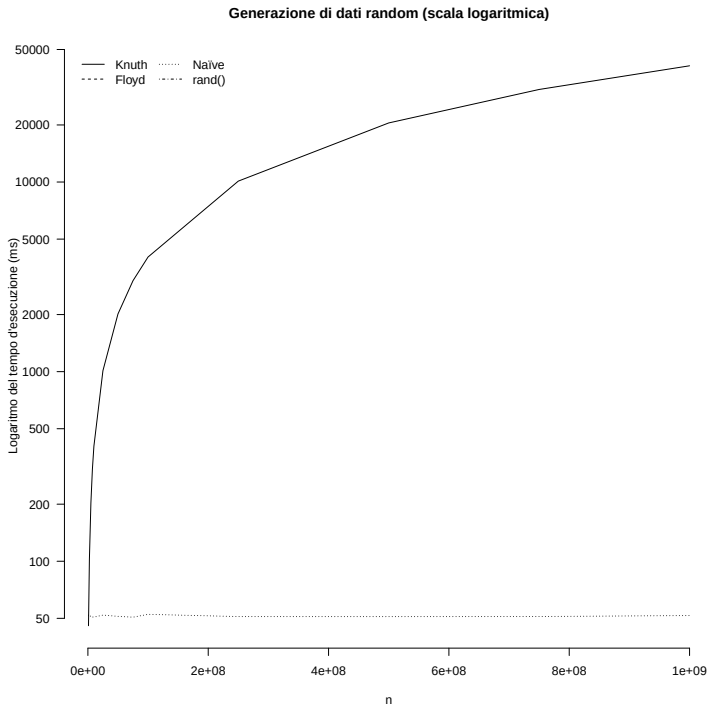

Figura 5.12: Tempi di esecuzione fissato $m = 10.000$ e variando n (scala logaritmica)

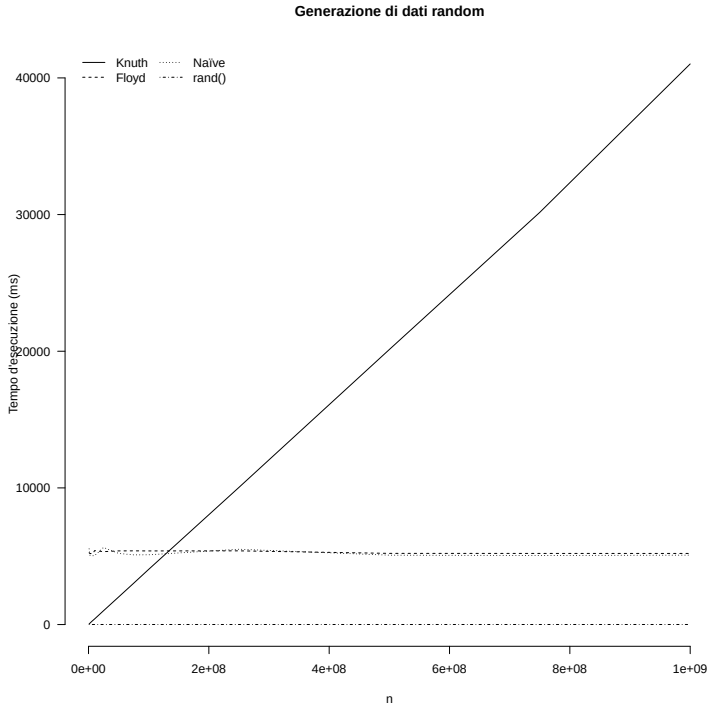

Figura 5.13: Tempi di esecuzione fissato $m = 100.000$ e variando n

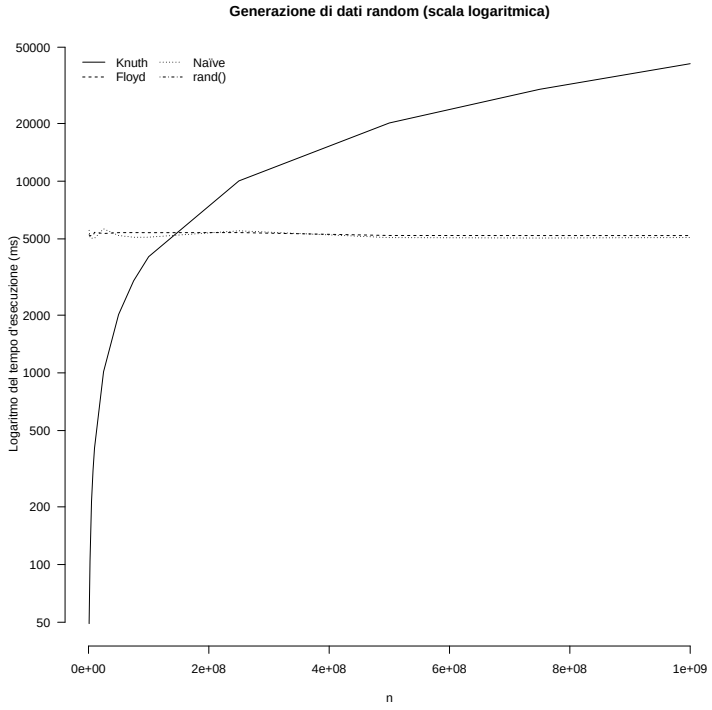

Figura 5.14: Tempi di esecuzione fissato $m = 100.000$ e variando n (scala logaritmica)

Come si può notare:

- l'Algoritmo di Knuth ha una complessità lineare nel range di possibili valori n; si vedano i Grafici 5.9, 5.11 e 5.13 in cui è stato fissato m e fatto variare n;
- l'Algoritmo di Floyd ha una complessità quadratica nel numero di numeri da generare m; si vedano i Grafici 5.1, 5.3, 5.5 e 5.7 in cui è stato fissato n e fatto variare m;
- l'Algoritmo per il campionamento con ripetizioni ha una complessità lineare nel numero di numeri da generare m; si vedano i Grafici 5.9, 5.11 e 5.13 in cui è stato fissato m e fatto variare n;
- l'Algoritmo per il campionamento senza ripetizioni ha una complessità quadratica nel numero di numeri da generare m; si vedano i Grafici 5.1, 5.3, 5.5 e 5.7 in cui è stato fissato n e fatto variare m;

Osservazione 8. Preme far notare che l'algoritmo per il campionamento con ripetizioni richiede un tempo inferiore agli altri proprio in virtù della mancanza di controllo sui valori precedentemente generati.

Sempre dai grafici si può notare che:

- l'Algoritmo di Floyd e quello per il campionamento senza ripetizioni impiegano un tempo costante se viene fissato un m e si varia n; questo è particolarmente vantaggioso se dobbiamo generare pochi dati (poche centinaia di migliaia) in un range molto elevato;
- il tempo d'esecuzione dell'Algoritmo di Knuth è indipendente da m. Tale algoritmo opera perciò in tempo costante se viene fissato n e si varia m, rendendolo svantaggioso se dobbiamo generare pochi record in un range molto elevato; per contro però se si devono generare milioni di dati opera in tempi sempre inferiori agli altri due algoritmi;
- l'Algoritmo di campionamento con ripetizioni, in virtù della sua estrema semplicità, opera in tempi praticamente ininfluenti ai fini pratici in tutti i test effettuati (nel caso peggiore si sono misurati tempi di 4-5 ms).

Da queste considerazioni possiamo fornire alcuni ambiti di utilizzo per gli algoritmi:

- l'Algoritmo di Floyd è adatto qualora si debba generare pochi record (qualche centinaio di migliaia) in range molto elevati: questo avviene nel caso della generazione di numeri floating point o stringhe che hanno necessariamente bisogno di un range di circa 2 miliardi (per poter generare stringhe di lunghezza superiore a 3-4 caratteri o floating point distribuiti in modo uniforme);

- l'Algoritmo di Knuth è adatto qualora si debba generare molti record in range relativamente piccoli (qualche milione): questo avviene nel caso della generazione di numeri interi in intervalli limitati, date oppure stringhe la cui lunghezza non deve superare 2-3 caratteri (si prenda ad esempio il tipo "char" che è, di fatto, una stringa avente un solo carattere).

Osservazione 9. L'Algoritmo naive per il campionamento senza ripetizioni non è di utilità pratica perché non gode della proprietà di terminazione forte. Oltretutto ha tempi di esecuzione comparabili con quelli dell'Algoritmo di Floyd che invece gode di tale proprietà.

Infine, valutiamo i tempi di esecuzione della generalizzazione dell'Algoritmo di Floyd che permette di calcolare i valori per N record composti da n colonne in cui ogni colonna i può assumere valori nell'intervallo $[0, max_i]$. Per tracciare i grafici abbiamo misurato, per cinque valori di $n = \{5, 10, 15, 20, 25\}$, i tempi di esecuzione dell'algoritmo nella generazione di N record in cui N assume valori:

$$\{1000, 2500, 5000, 7500, 10000, 20000, 40000, 60000\}$$

$$\{80000, 100000, 200000, 400000, 600000, 800000, 1000000\}$$

Sono stati quindi tracciati due grafici, in cui rispettivamente si sono impostati $max_i = 100$ e $max_i = 10.000$; questo per valutare l'influenza dell'Algoritmo di Floyd nella generazione dei dati dell'ultima colonna. Infatti si ricorda che l'algoritmo opera costruendo le prime $n-1$ colonne con ripetizioni, ordinandole in modo lessicografico per crearne delle partizioni e infine per ciascuna partizione utilizza l'algoritmo di Floyd per generare i valori dell'ultima colonna.

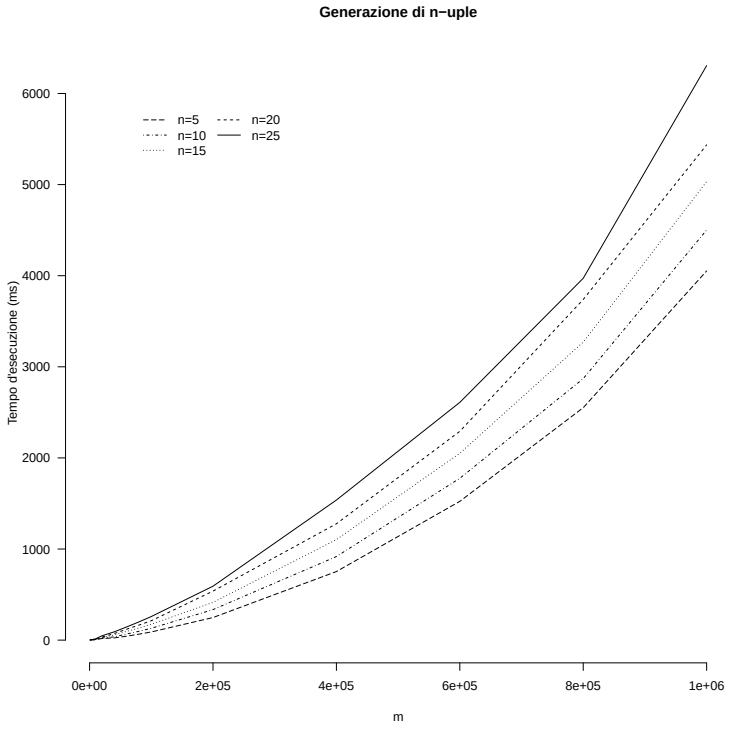

Figura 5.15: Tempi di esecuzione per Floydgen fissato $max_i = 100$ e variando il numero di colonne e il numero di record

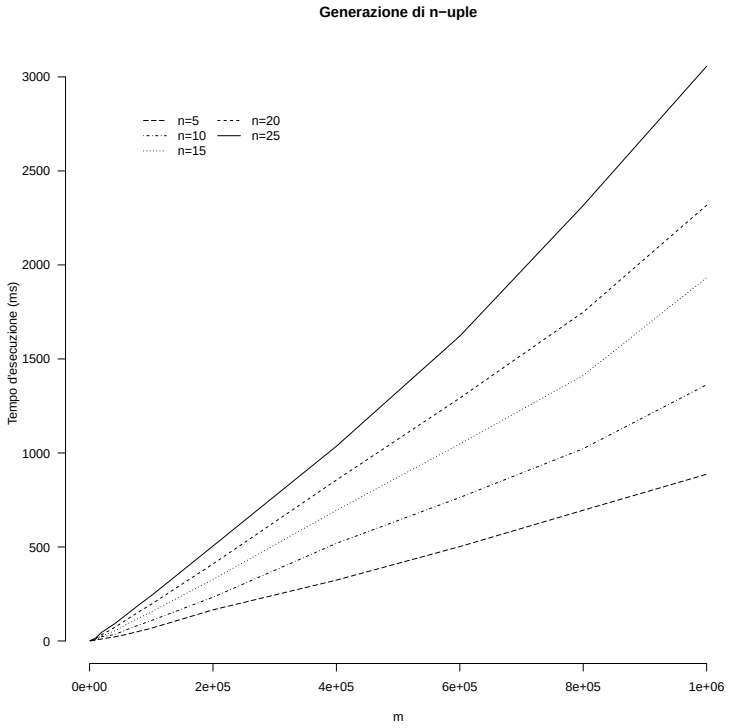

Figura 5.16: Tempi di esecuzione per Floydgen fissato $max_i = 1.000$ e variando il numero di colonne e il numero di record

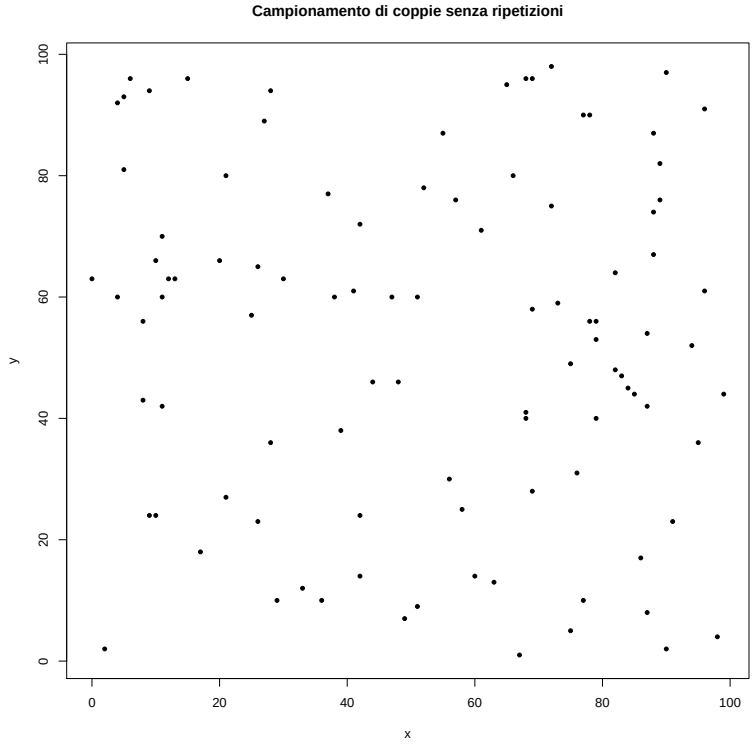

Figura 5.17: Scatter-plot per Floydgen generando 100 record

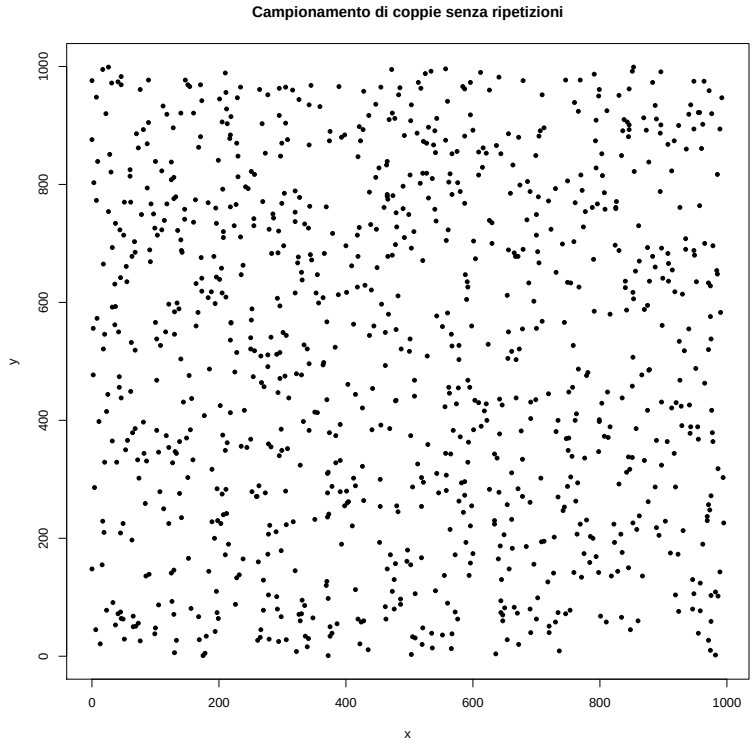

Figura 5.18: Scatter-plot per Floydgen generando 1000 record

Come si può notare:

- nel Grafico 5.15 con $max_i = 100$ gli andamenti delle cinque serie di dati seguono una funzione quadratica; questo è dovuto al fatto che le partizioni create dalle prime $n-1$ colonne sono poche ma sono costituite da un numero relativamente alto di elementi e quindi per N elevati l'Algoritmo di Floyd deve generare molti numeri (parametro m relativamente elevato). Abbiamo visto (Figura 5.1) che l'algoritmo di Floyd esibisce un comportamento quadratico dei tempi d'esecuzione, in funzione di m.

- nel Grafico 5.16 con $max_i = 10.000$ invece gli andamenti tendono a seguire una funzione lineare; questo è dovuto al fatto che le partizioni create dalle prime $n-1$ colonne sono molte e costituite da un numero relativamente basso di elementi e quindi anche per N elevati l'Algoritmo di Floyd non dovrà generare molti numeri (parametro m relativamente piccolo);

- si può notare come l'algoritmo riesca a generare in tempi brevi (circa 3 secondi) un milione di record per 25 colonne;

- dagli scatter-plot 5.17 e 5.18 si può vedere come i dati generati appaiano ben distribuiti nello spazio dei possibili valori; per dimostrare che la distribuzione dei valori è uniforme bisogna sottoporre l'implementazione ad una serie di test statistici.

5.2 Esperimenti di popolamento di basi di dati

Lo scopo degli esperimenti di popolamento è verificare che l'applicativo riesca, nonostante le sue limitazioni, a popolare una base di dati reale. Riportiamo quanto detto nella Sezione 4.3.1 riguardo le limitazioni dell'implementazione proposta:

- in presenza di vincoli d'integrità referenziale e d'unicità che operano su attributi comuni, il vincolo di chiave esterna **deve** essere incluso nel vincolo di unicità;

- a uno stesso attributo **non** possono essere associati più vincoli di integrità referenziale;

- ogni vincolo di unicità **deve** avere almeno un attributo che non sia coinvolto in un vincolo di chiave esterna.

Per eseguire i test sono stati reperiti in rete alcuni schemi di basi di dati già popolati, i cui domini degli attributi e le configurazioni dei vincoli siano compatibili con le restrizioni d'uso dell'applicativo. Dopo di che si è provveduto a importare nel DBMS PostgreSQL gli schemi di tali basi di dati e a farle popolare dall'applicativo.

La prima base di dati contiene informazioni sugli acquisti di un'azienda; il suo schema, in cui sono stati omessi attributi poco interessanti ai nostri fini, è:

- CATEGORIE(**categoria**, nome_categoria)
- ORDINI_CLIENTI(ID_cliente, ID_ordine, ID_prodotto)
- CLIENTI(**ID_cliente**, nome, cognome, indirizzo, città, regione, stato, email, telefono, tipo_carta_credito, carta_credito, scadenza_carta_credito , nome_utente, password, età, sesso, reddito)
- INVENTARIO(**ID_prodotto**, quantità_magazzino, n_vendite)
- ORDINI(**ID_ordine**, data, ID_cliente, totale_netto, tasse, totale)
- LINEE_ORDINI(ID_linea_ordine, ID_ordine, ID_prodotto, quantità, data)
- PRODOTTI(**ID_prodotto**, categoria, titolo, autore, prezzo, speciale)
- RIORDINI(ID_prodotto, data_minimo, quantità_minima, data_riordine, quantità_riordine, data_arrivo)

Al solito in grassetto riportiamo le chiavi primarie, che in questo schema non sempre sono presenti. Facciamo notare che è uno schema legale ma che è consigliabile definire sempre la chiave primaria. Sullo schema sussistono inoltre i seguenti vincoli:

- CE: ORDINI_CLIENTI(ID_cliente)→ CLIENTI(ID_cliente)
- CE: ORDINI(ID_cliente)→ CLIENTI(ID_cliente)
- CE: LINEE_ORDINI(ID_ordine)→ ORDINI(ID_ordine)

Non vi sono altri vincoli di unicità al di fuori di quelli definiti delle chiavi primarie.

La seconda base di dati contiene informazioni sui nutrienti; il suo schema, in cui sono stati omessi attributi poco interessanti ai nostri fini, è:

- DATA_SRC (**datasrc_ID**, autori, titolo, anno, rivista, volume, numero)
- DATASRCLN (**ndb_no, nutr_no, datasrc_ID**)

- DERIV_CD (**deriv_cd**, descrizione)
- FD_GROUP (**fd_cd**, descrizione)
- FOOD_DES (**ndb_no**, fdgrp_cd, descrizione, nome, produttore)
- FOOTNOTE (ndb_no, numero, tipo, nutr_no, footnt_txt)
- NUT_DATA (**ndb_no, nutr_no**, valore, src_cd, deriv_cd))
- NUTR_DEF (**nurt_no**, unità, nome, descrizione)
- SRC_CD (**src_cd**, descrizione)
- WEIGHT (**ndb_no, seq**, totale, descrizione, peso, num_data, std_dev)

Al solito in grassetto riportiamo le chiavi primarie, che in questo schema non sempre sono presenti. Facciamo notare che è uno schema legale ma che è consigliabile definire sempre la chiave primaria. Sullo schema sussistono inoltre i seguenti vincoli:

- CE: DATASRCLN(datasrc_id)→ DATA_SRC(datasrc_id)
- CE: DATASRCLN(ndb_no, nutr_no)→ NUT_DATA(ndb_no, nutr_no)
- CE: FOOD_DES(fdgrp_cd)→ FD_GROUP(fdgrp_cd)
- CE: FOOTNOTE(ndb_no)→ FOOD_DES(ndb_no)
- CE: FOOTNOTE(nutr_no)→ NUTR_DEF(nutr_no)
- CE: NUT_DATA(deriv_cd)→ DERIV_CD(deriv_cd)
- CE: NUT_DATA(ndb_no)→ FOOD_DES(ndb_no)
- CE: NUT_DATA(nutr_no)→ NUTR_DEF(nutr_no)
- CE: NUT_DATA(src_cd)→ SRC_CD(src_cd)
- CE: WEIGHT(ndb_no)→ FOOD_DES(ndb_no)

Non vi sono altri vincoli di unicità al di fuori di quelli definiti delle chiavi primarie.

La generazione dei dati è avvenuta senza particolari problemi se non la necessità di inserire in ciascuna tabella un numero di record diverso da quello della tabella originaria. Infatti l'applicativo, come visto nella Sezione 4.2.4, genera sempre i dati per le chiavi esterne in modo unico e questo impone che la tabella referenziata abbia

più record di quella su cui sussiste il vincolo. Questa è una limitazione in quanto nella realtà è possibile avere istanze in cui le tabelle riferite hanno meno record.

Per entrambe le basi di dati si sono generate decine di migliaia di record, operazione che ha richiesto molto tempo; tempo in gran parte speso nell'inserimento dei record nelle tabelle; questa inefficienza è da imputarsi alla modalità di inserimento: viene inserito un record alla volta. Per migliorare l'efficienza degli inserimenti sarà quindi necessario salvare i record in un file e inserirli direttamente nella base di dati attraverso appositi comandi.

Presentiamo ora due grafici che mostrano le distribuzioni dei valori presenti nelle istanze reali per due attributi:

- "n_vendite" (di tipo intero) della tabella *INVENTARIO* del primo schema proposto (Grafico 5.19);
- "rivista" (di tipo stringa) della tabella *DATA_SRC* del secondo schema proposto (Grafico 5.20).

Presentiamo anche un grafico (Grafico 5.21) che mostra la distribuzione dei dati per l'attributo "a" di un ipotetico schema con due tabelle R(**id**,a) e S(**id**) in cui al solito in grassetto sono riportate le chiavi primarie e su cui sussiste un vincolo di integrità referenziale CE: R(a)→ S(id).

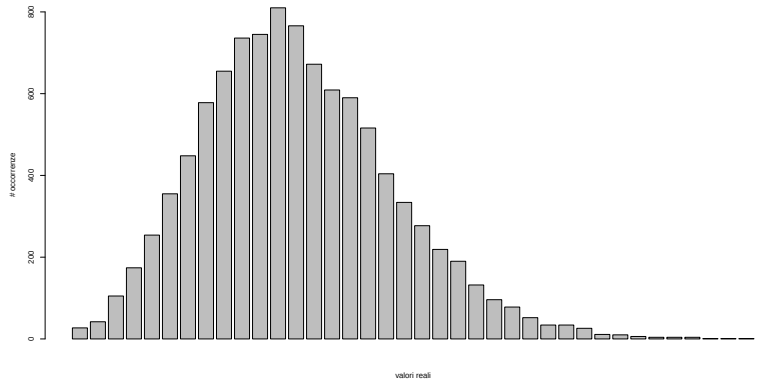

Figura 5.19: Distribuzione dati istanza reale di tipo intero

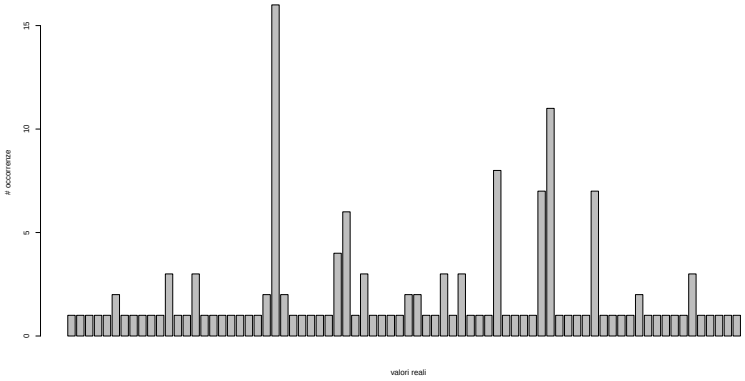

Figura 5.20: Distribuzione dati istanza reale di tipo stringa

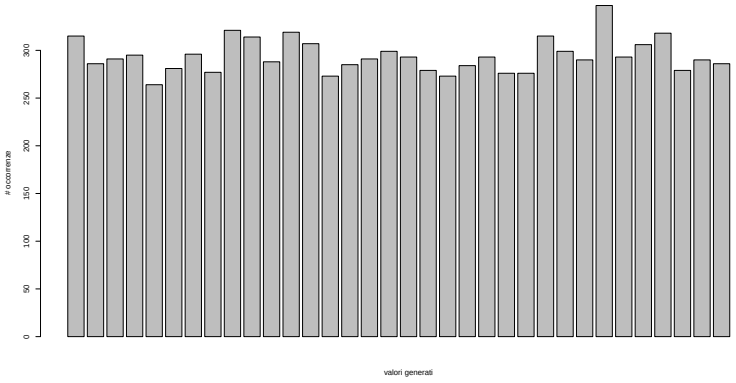

Figura 5.21: Distribuzione dati istanza generata di tipo intero

Da questi grafici si può notare come i dati delle istanze reali non seguano la distribuzione uniforme. Si consideri, ad esempio, il Grafico 5.19 che rappresenta le vendite degli oggetti nell'inventario; la sua distribuzione non è chiaramente uniforme: cosa abbastanza normale dato che ci saranno sicuramente degli oggetti che vendono più di altri. Come si può quindi notare dal Grafico 5.21 l'applicativo genererà sempre i valori casuali distribuiti con una distribuzione uniforme, anche se questo non rappresenta sempre la realtà.

5.3 Test statistici

Per i test statistici sugli algoritmi di campionamento è stato utilizzato uno specifico tool chiamato "dieharder" [20]. Prima di presentare i risultati citiamo alcune importanti considerazioni sul suo uso contenute nel manuale:

- l'utilizzo dell'applicativo con file di input è sconsigliato in quanto tale applicativo è nato per testare gli algoritmi e non le sequenze da essi generate;
- in virtù di quanto detto sopra le sequenze che devono essere passate in input devono contenere un numero molto elevato (a seconda dei test anche diverse decine di miliardi) di valori; questo per non incorrere in problemi di periodicità della sequenza: infatti se un test richiede più dati di quelli presenti nel file di input quest'ultimo viene semplicemente riletto causando una chiara anomalia statistica;
- il test è nato per verificare la proprietà di casualità dei numeri; dato che le macchine attuali sono deterministiche non possono essere del tutto casuali;
- ogni test dovrebbe avere un suo specifico file di input la cui grandezza dipende dallo specifico test e può arrivare a superare le decine di GB;
- è relativamente normale che un test fallisca e questo può succedere anche con alcuni generatori hardware (solitamente migliori di quelli software) di numeri pseudo-casuali.

Diamo anche una breve spiegazione, sempre tratta dal manuale, del **p-value**. Il p-value è il valore restituito dai singoli test della suite e per comprenderne il funzionamento è necessario introdurre anche il concetto di **ipotesi nulla**.

L'ipotesi nulla per un test sui generatori di numeri casuali è: "Il generatore è perfetto, e per ogni scelta del seed produce una sequenza infinita di numeri distinti

che possiedono tutte le proprietà statistiche dei numeri casuali". Come già detto, teoricamente, quest'ipotesi è falsa per tutti i generatori in quanto sono periodici (seppur questo periodo può essere molto alto); in linea pratica però possono esistere dei generatori che producono sequenze non distinguibili da quelle casuali. Il p-value rappresenta proprio questo e cioè: rappresenta la probabilità di avere una sequenza realmente casuale assumendo vera l'ipotesi nulla.

Osservazione 10. Il p-value **non** è la probabilità che l'ipotesi nulla sia vera. Quindi consente di fare una delle due seguenti asserzioni: dichiarare falsa l'ipotesi nulla o non poterla dichiarare falsa (il che è **diverso** dal dire che è vera).

Detto questo presentiamo i risultati dei test statistici applicati all'Algoritmo di Knuth 3.3.2 e all'Algoritmo di campionamento con ripetizioni 3.2.2. Alla luce delle considerazioni sopra l'Algoritmo di Floyd 3.3.1 e l'Algoritmo di campionamento senza ripetizioni 3.3 non possono essere testati a causa del loro elevato tempo di esecuzione necessario per generare quella decina di milioni di valori necessari a completare il test con una certa approssimazione.

I test effettuati dalla suite sono:

```
#=============================================================================#
#      dieharder version 3.31.1 Copyright 2003 Robert G. Brown       #
#=============================================================================#
Installed dieharder tests:
 Test Number                         Test Name         Test Reliability
=============================================================================
  -d 0                          Diehard Birthdays Test            Good
  -d 1                             Diehard OPERM5 Test            Good
  -d 2                  Diehard 32x32 Binary Rank Test            Good
  -d 3                    Diehard 6x8 Binary Rank Test            Good
  -d 4                          Diehard Bitstream Test            Good
  -d 5                                   Diehard OPSO         Suspect
  -d 6                                Diehard OQSO Test         Suspect
  -d 7                                 Diehard DNA Test         Suspect
  -d 8              Diehard Count the 1s (stream) Test            Good
  -d 9                Diehard Count the 1s Test (byte)            Good
  -d 10                       Diehard Parking Lot Test            Good
  -d 11      Diehard Minimum Distance (2d Circle) Test            Good
  -d 12      Diehard 3d Sphere (Minimum Distance) Test            Good
  -d 13                           Diehard Squeeze Test            Good
  -d 14                              Diehard Sums Test      Do Not Use
  -d 15                              Diehard Runs Test            Good
  -d 16                            Diehard Craps Test            Good
  -d 17                  Marsaglia and Tsang GCD Test            Good
  -d 100                              STS Monobit Test            Good
  -d 101                                 STS Runs Test            Good
  -d 102                STS Serial Test (Generalized)            Good
  -d 200                    RGB Bit Distribution Test            Good
```

```
-d 201         RGB Generalized Minimum Distance Test           Good
-d 202                      RGB Permutations Test              Good
-d 203                        RGB Lagged Sum Test              Good
-d 204             RGB Kolmogorov-Smirnov Test Test            Good
-d 205                       Byte Distribution                 Good
-d 206                                DAB DCT                  Good
-d 207                      DAB Fill Tree Test                Good
-d 208                    DAB Fill Tree 2 Test                Good
-d 209                      DAB Monobit 2 Test                Good
```

Diamo una breve descrizione di alcuni test:

- *Birthday*: sceglie alcuni punti in un intervallo con una distribuzione esponenziale; deve il suo nome al paradosso del compleanno;

- *Parking Lot*: piazza in maniera casuale i valori in quadrati di dimensione 100×100, se c'è una sovrapposizione ritenta; se dopo 12000 tentativi il numero di "parcheggi" trovati è entro una distribuzione normale il test è considerato passato;

- *Minimum Distance*: piazza in modo casuale 8000 punti in un quadrato di dimensione 10000×10000 e trova la minima distanza fra le coppie; il test è passato se questa distanza è distribuita in modo esponenziale;

- *Craps*: gioca per 200000 volte il gioco del "craps" (un gioco di dadi) e conta le vittorie e il numero di lanci; il test passa se questi valori seguono una ben particolare distribuzione.

Osservazione 11. Le distribuzioni di probabilità menzionate sopra sono da intendersi relative ai risultati dei singoli test e non relative alla distribuzione dei dati delle sequenze. Quindi, ad esempio, il risultato del test Parking Lot dovrà seguire una distribuzione normale: solo se questa distribuzione è entro certi parametri (propri del singolo test) dieharder restituirà "PASSED".

L'algoritmo di Knuth non riesce a passare alcun test, probabilmente, a causa del fatto che genera i dati in modo ordinato. Per questo motivo al termine della generazione della sequenza è stata creata una permutazione della stessa con l'Algoritmo di Fisher-Yate-Durstenfeld 3.3.4. In questo modo si è perso l'ordinamento dei valori senza modificare gli stessi.

Per l'algoritmo di Knuth con permutazione casuale ad opera dell'algoritmo Fisher-Yate-Durstenfeld i risultati sono:

```
#=============================================================================#
#      dieharder version 3.31.1 Copyright 2003 Robert G. Brown       #
#=============================================================================#
   rng_name    |           filename             |rands/second|
     file_input|                        /tmp/out|  4.23e+06  |
#=============================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================#
   diehard_birthdays|   0|       100|     100|0.00000000|  FAILED
      diehard_operm5|   0|   1000000|     100|0.78955951|  PASSED
  diehard_rank_32x32|   0|     40000|     100|0.00000000|  FAILED
    diehard_rank_6x8|   0|    100000|     100|0.36027588|  PASSED
   diehard_bitstream|   0|   2097152|     100|0.00000000|  FAILED
        diehard_opso|   0|   2097152|     100|0.38991386|  PASSED
         diehard_dna|   0|   2097152|     100|0.00000000|  FAILED
diehard_count_1s_str|   0|    256000|     100|0.00000000|  FAILED
diehard_count_1s_byt|   0|    256000|     100|0.00000000|  FAILED
 diehard_parking_lot|   0|     12000|     100|0.00000000|  FAILED
    diehard_2dsphere|   2|      8000|     100|0.00000000|  FAILED
    diehard_3dsphere|   3|      4000|     100|0.00000000|  FAILED
     diehard_squeeze|   0|    100000|     100|0.00000000|  FAILED
        diehard_sums|   0|       100|     100|0.00000000|  FAILED
        diehard_runs|   0|    100000|     100|0.46844992|  PASSED
        diehard_runs|   0|    100000|     100|0.69698039|  PASSED
       diehard_craps|   0|    200000|     100|0.00000000|  FAILED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.00000000|  FAILED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.00000294|   WEAK
         sts_monobit|   1|    100000|     100|0.00000000|  FAILED
            sts_runs|   2|    100000|     100|0.00000000|  FAILED
          sts_serial|   1|    100000|     100|0.00000000|  FAILED
          sts_serial|   2|    100000|     100|0.00000000|  FAILED
          sts_serial|   3|    100000|     100|0.00000000|  FAILED
          sts_serial|   3|    100000|     100|0.00000000|  FAILED
          sts_serial|   4|    100000|     100|0.00000000|  FAILED
          sts_serial|   4|    100000|     100|0.00000000|  FAILED
          sts_serial|   5|    100000|     100|0.00000000|  FAILED
          sts_serial|   5|    100000|     100|0.00000000|  FAILED
          sts_serial|   6|    100000|     100|0.00000000|  FAILED
          sts_serial|   6|    100000|     100|0.00000000|  FAILED
          sts_serial|   7|    100000|     100|0.00000000|  FAILED
          sts_serial|   7|    100000|     100|0.00000000|  FAILED
          sts_serial|   8|    100000|     100|0.00000000|  FAILED
          sts_serial|   8|    100000|     100|0.00000000|  FAILED
          sts_serial|   9|    100000|     100|0.00000000|  FAILED
          sts_serial|   9|    100000|     100|0.00000000|  FAILED
          sts_serial|  10|    100000|     100|0.00000000|  FAILED
          sts_serial|  10|    100000|     100|0.00000000|  FAILED
          sts_serial|  11|    100000|     100|0.00000000|  FAILED
          sts_serial|  11|    100000|     100|0.00000510|   WEAK
          sts_serial|  12|    100000|     100|0.00000000|  FAILED
```

```
          sts_serial|  12|    100000|     100|0.00108883|   WEAK
          sts_serial|  13|    100000|     100|0.00000000|  FAILED
          sts_serial|  13|    100000|     100|0.00158197|   WEAK
          sts_serial|  14|    100000|     100|0.00000000|  FAILED
          sts_serial|  14|    100000|     100|0.00254250|   WEAK
          sts_serial|  15|    100000|     100|0.00000000|  FAILED
          sts_serial|  15|    100000|     100|0.21617348|  PASSED
          sts_serial|  16|    100000|     100|0.00000000|  FAILED
          sts_serial|  16|    100000|     100|0.05276778|  PASSED
         rgb_bitdist|   1|    100000|     100|0.00000000|  FAILED
         rgb_bitdist|   2|    100000|     100|0.00000000|  FAILED
         rgb_bitdist|   3|    100000|     100|0.00000000|  FAILED
         rgb_bitdist|   4|    100000|     100|0.00000000|  FAILED
         rgb_bitdist|   5|    100000|     100|0.00000000|  FAILED
         rgb_bitdist|   6|    100000|     100|0.00000000|  FAILED
         rgb_bitdist|   7|    100000|     100|0.00000000|  FAILED
         rgb_bitdist|   8|    100000|     100|0.00000000|  FAILED
         rgb_bitdist|   9|    100000|     100|0.00000000|  FAILED
         rgb_bitdist|  10|    100000|     100|0.00000000|  FAILED
         rgb_bitdist|  11|    100000|     100|0.00000000|  FAILED
         rgb_bitdist|  12|    100000|     100|0.00000000|  FAILED
rgb_minimum_distance|   2|     10000|    1000|0.00000000|  FAILED
    rgb_permutations|   2|    100000|     100|0.92892113|  PASSED
    rgb_permutations|   3|    100000|     100|0.98621795|  PASSED
    rgb_permutations|   4|    100000|     100|0.15458416|  PASSED
    rgb_permutations|   5|    100000|     100|0.85828591|  PASSED
      rgb_lagged_sum|   0|   1000000|     100|0.00000000|  FAILED
     rgb_kstest_test|   0|     10000|    1000|0.00000000|  FAILED
     dab_bytedistrib|   0|  51200000|       1|0.00000000|  FAILED
             dab_dct| 256|     50000|       1|0.00000000|  FAILED
        dab_filltree|  32|  15000000|       1|0.66277120|  PASSED
        dab_filltree|  32|  15000000|       1|0.38684479|  PASSED
       dab_filltree2|   0|   5000000|       1|0.00000000|  FAILED
       dab_filltree2|   1|   5000000|       1|0.00000000|  FAILED
        dab_monobit2|  12|  65000000|       1|1.00000000|  FAILED
```

Per l'algoritmo di campionamento con ripetizioni i risultati sono:

```
#=================================================================#
#      dieharder version 3.31.1 Copyright 2003 Robert G. Brown       #
#=================================================================#
   rng_name    |           filename             |rands/second|
     file_input|                        /tmp/out|  4.23e+06  |
#=================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=================================================================#
   diehard_birthdays|   0|       100|     100|0.00000050|  FAILED
      diehard_operm5|   0|   1000000|     100|0.79776900|  PASSED
  diehard_rank_32x32|   0|     40000|     100|0.00000000|  FAILED
    diehard_rank_6x8|   0|    100000|     100|0.98719677|  PASSED
   diehard_bitstream|   0|   2097152|     100|0.00000000|  FAILED
        diehard_opso|   0|   2097152|     100|0.64076595|  PASSED
         diehard_dna|   0|   2097152|     100|0.00000000|  FAILED
```

```
diehard_count_1s_str|   0|    256000|     100|0.00000000|  FAILED
diehard_count_1s_byt|   0|    256000|     100|0.00000000|  FAILED
 diehard_parking_lot|   0|     12000|     100|0.00000000|  FAILED
    diehard_2dsphere|   2|      8000|     100|0.00000000|  FAILED
    diehard_3dsphere|   3|      4000|     100|0.00000000|  FAILED
     diehard_squeeze|   0|    100000|     100|0.00000000|  FAILED
        diehard_sums|   0|       100|     100|0.00000000|  FAILED
        diehard_runs|   0|    100000|     100|0.28753672|  PASSED
        diehard_runs|   0|    100000|     100|0.14714992|  PASSED
       diehard_craps|   0|    200000|     100|0.00000000|  FAILED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.00000000|  FAILED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.00366457|    WEAK
         sts_monobit|   1|    100000|     100|0.00000000|  FAILED
            sts_runs|   2|    100000|     100|0.00000000|  FAILED
          sts_serial|   1|    100000|     100|0.00000000|  FAILED
          sts_serial|   2|    100000|     100|0.00000000|  FAILED
          sts_serial|   3|    100000|     100|0.00000000|  FAILED
          sts_serial|   3|    100000|     100|0.00000000|  FAILED
          sts_serial|   4|    100000|     100|0.00000000|  FAILED
          sts_serial|   4|    100000|     100|0.00000000|  FAILED
          sts_serial|   5|    100000|     100|0.00000000|  FAILED
          sts_serial|   5|    100000|     100|0.00000000|  FAILED
          sts_serial|   6|    100000|     100|0.00000000|  FAILED
          sts_serial|   6|    100000|     100|0.00000000|  FAILED
          sts_serial|   7|    100000|     100|0.00000000|  FAILED
          sts_serial|   7|    100000|     100|0.00000000|  FAILED
          sts_serial|   8|    100000|     100|0.00000000|  FAILED
          sts_serial|   8|    100000|     100|0.00000000|  FAILED
          sts_serial|   9|    100000|     100|0.00000000|  FAILED
          sts_serial|   9|    100000|     100|0.00000000|  FAILED
          sts_serial|  10|    100000|     100|0.00000000|  FAILED
          sts_serial|  10|    100000|     100|0.40229854|  PASSED
          sts_serial|  11|    100000|     100|0.00000000|  FAILED
          sts_serial|  11|    100000|     100|0.02481280|  PASSED
          sts_serial|  12|    100000|     100|0.00000000|  FAILED
          sts_serial|  12|    100000|     100|0.79438611|  PASSED
          sts_serial|  13|    100000|     100|0.00000000|  FAILED
          sts_serial|  13|    100000|     100|0.66255419|  PASSED
          sts_serial|  14|    100000|     100|0.00000000|  FAILED
          sts_serial|  14|    100000|     100|0.89897993|  PASSED
          sts_serial|  15|    100000|     100|0.00000000|  FAILED
          sts_serial|  15|    100000|     100|0.05691246|  PASSED
          sts_serial|  16|    100000|     100|0.00000000|  FAILED
          sts_serial|  16|    100000|     100|0.63401930|  PASSED
         rgb_bitdist|   1|    100000|     100|0.00000000|  FAILED
         rgb_bitdist|   2|    100000|     100|0.00000000|  FAILED
         rgb_bitdist|   3|    100000|     100|0.00000000|  FAILED
         rgb_bitdist|   4|    100000|     100|0.00000000|  FAILED
         rgb_bitdist|   5|    100000|     100|0.00000000|  FAILED
         rgb_bitdist|   6|    100000|     100|0.00000000|  FAILED
         rgb_bitdist|   7|    100000|     100|0.00000000|  FAILED
         rgb_bitdist|   8|    100000|     100|0.00000000|  FAILED
         rgb_bitdist|   9|    100000|     100|0.00000000|  FAILED
```

```
         rgb_bitdist|  10|    100000|     100|0.00000000|  FAILED
         rgb_bitdist|  11|    100000|     100|0.00000000|  FAILED
         rgb_bitdist|  12|    100000|     100|0.00000000|  FAILED
rgb_minimum_distance|   0|     10000|    1000|0.00000000|  FAILED
    rgb_permutations|   5|    100000|     100|0.57929941|  PASSED
      rgb_lagged_sum|   0|   1000000|     100|0.00000000|  FAILED
     rgb_kstest_test|   0|     10000|    1000|0.00000000|  FAILED
     dab_bytedistrib|   0|  51200000|       1|0.00000000|  FAILED
             dab_dct| 256|     50000|       1|0.00000000|  FAILED
        dab_filltree|  32|  15000000|       1|0.52186635|  PASSED
        dab_filltree|  32|  15000000|       1|0.93640282|  PASSED
       dab_filltree2|   0|   5000000|       1|0.00000000|  FAILED
        dab_monobit2|  12|  65000000|       1|1.00000000|  FAILED
```

Entrambe le serie di test sono state effettuate con delle sequenze di 170.000.000 di valori nel range $[1, 1.000.000.000]$.

Come si può notare entrambi gli algoritmi riescono a passare solo alcuni dei test; questo è dovuto principalmente alle considerazioni fatte a inizio sezione, e cioè:

- per ogni test bisognerebbe usare un file di input in cui valori contenuti siano diversi;
- alcuni dei test hanno bisogno di più di 170.000.000 di valori e quindi dovendo rileggere il file i test hanno trovato un'anomalia statistica (il periodo è ben chiaro dato che il file, e quindi i valori in esso contenuti, è stato riletto più volte).

Per migliorare la metodologia di test bisognerebbe far analizzare a "dieharder" le implementazioni degli algoritmi e non solo i dati da essi generati. Per far ciò bisogna implementare gli algoritmi in modo che non salvino i valori in un file, ma li restituiscano direttamente in output; ma questo è possibile solamente con l'algoritmo di campionamento con ripetizioni in quanto l'algoritmo di Knuth richiede un successivo "shuffle" dei valori mentre l'algoritmo di Floyd e di campionamento senza ripetizioni hanno bisogno di conoscere ad ogni iterazione i valori precedentemente generati.

A titolo di esempio riportiamo i risultati sugli stessi test di due algoritmi noti ed implementati all'interno della suite "dieharder": il *RANDU* basato sul metodo lineare congruenziale ma notoriamente noto per le sue scarse qualità e il *Mersenne-Twister* anch'esso basato sul metodo lineare congruenziale ma che viene invece considerato uno dei migliori algoritmi per la generazione di numeri casuali.

Osservazione 12. I seguenti test **non** vengono eseguiti a partire da un file di input e quindi non hanno limitazioni sulla lunghezza delle sequenze generate. Al contrario

per ovvi motivi un file non può superare una certa dimensione. Per cercare di mettere in pari condizioni queste due serie di test è stato imposto, ove possibile (alcuni test della suite hanno dei valori minimi imprescindibili), l'uso dello stesso seed e un numero massimo di valori pari a 170.000.000 (come la grandezza del file). Nonostante questi accorgimenti le due serie di test **non** sono comparabili e vogliono quindi solamente dare un'idea di come si comporta un cattivo e un buon generatore di numeri casuali.

I risultati per l'algoritmo *RANDU*:

```
#=============================================================================#
#            dieharder version 3.31.1 Copyright 2003 Robert G. Brown          #
#=============================================================================#
   rng_name    |rands/second|
         randu|  2.07e+08  |
#=============================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment|    Seed
#=============================================================================#
   diehard_birthdays|   0|       100|     100|0.00003503|   WEAK   |3983253652
      diehard_operm5|   0|   1000000|     100|0.00000000|  FAILED  |3562955932
  diehard_rank_32x32|   0|     40000|     100|0.47973015|  PASSED  |1999856219
    diehard_rank_6x8|   0|    100000|     100|0.00000000|  FAILED  |3322892330
   diehard_bitstream|   0|   2097152|     100|0.00000000|  FAILED  |2738959134
        diehard_opso|   0|   2097152|     100|0.00000000|  FAILED  |2480420456
        diehard_oqso|   0|   2097152|     100|0.00000000|  FAILED  |1480264086
         diehard_dna|   0|   2097152|     100|0.00000000|  FAILED  |3839950855
diehard_count_1s_str|   0|    256000|     100|0.00000000|  FAILED  | 636294344
diehard_count_1s_byt|   0|    256000|     100|0.00000000|  FAILED  | 215195896
 diehard_parking_lot|   0|     12000|     100|0.80220017|  PASSED  |3244558421
    diehard_2dsphere|   2|      8000|     100|0.00000000|  FAILED  |  61264736
    diehard_3dsphere|   3|      4000|     100|0.00000000|  FAILED  |3400066210
     diehard_squeeze|   0|    100000|     100|0.00000810|   WEAK   |2335401112
        diehard_sums|   0|       100|     100|0.78694491|  PASSED  |4024982193
        diehard_runs|   0|    100000|     100|0.00021163|   WEAK   |1878020234
        diehard_runs|   0|    100000|     100|0.00042247|   WEAK   |1878020234
       diehard_craps|   0|    200000|     100|0.00000000|  FAILED  |2095956071
       diehard_craps|   0|    200000|     100|0.00000000|  FAILED  |2095956071
 marsaglia_tsang_gcd|   0|  10000000|     100|0.00000000|  FAILED  |3611822092
 marsaglia_tsang_gcd|   0|  10000000|     100|0.00000000|  FAILED  |3611822092
         sts_monobit|   1|    100000|     100|0.00000000|  FAILED  |3950636698
            sts_runs|   2|    100000|     100|0.00000000|  FAILED  |2955688416
          sts_serial|   1|    100000|     100|0.00000000|  FAILED  |3532966120
          sts_serial|   2|    100000|     100|0.00000000|  FAILED  |3532966120
          sts_serial|   3|    100000|     100|0.00000000|  FAILED  |3532966120
          sts_serial|   3|    100000|     100|0.00000000|  FAILED  |3532966120
          sts_serial|   4|    100000|     100|0.00000000|  FAILED  |3532966120
          sts_serial|   4|    100000|     100|0.00000000|  FAILED  |3532966120
          sts_serial|   5|    100000|     100|0.00000000|  FAILED  |3532966120
          sts_serial|   5|    100000|     100|0.00000000|  FAILED  |3532966120
          sts_serial|   6|    100000|     100|0.00000000|  FAILED  |3532966120
          sts_serial|   6|    100000|     100|0.00000000|  FAILED  |3532966120
```

```
          sts_serial|   7|    100000|     100|0.00000000|  FAILED  |3532966120
          sts_serial|   7|    100000|     100|0.00000000|  FAILED  |3532966120
          sts_serial|   8|    100000|     100|0.00000000|  FAILED  |3532966120
          sts_serial|   8|    100000|     100|0.00000000|  FAILED  |3532966120
          sts_serial|   9|    100000|     100|0.00000000|  FAILED  |3532966120
          sts_serial|   9|    100000|     100|0.00000000|  FAILED  |3532966120
          sts_serial|  10|    100000|     100|0.00000000|  FAILED  |3532966120
          sts_serial|  10|    100000|     100|0.00000000|  FAILED  |3532966120
          sts_serial|  11|    100000|     100|0.00000000|  FAILED  |3532966120
          sts_serial|  11|    100000|     100|0.00000000|  FAILED  |3532966120
          sts_serial|  12|    100000|     100|0.00000000|  FAILED  |3532966120
          sts_serial|  12|    100000|     100|0.00000000|  FAILED  |3532966120
          sts_serial|  13|    100000|     100|0.00000000|  FAILED  |3532966120
          sts_serial|  13|    100000|     100|0.00000000|  FAILED  |3532966120
          sts_serial|  14|    100000|     100|0.00000000|  FAILED  |3532966120
          sts_serial|  14|    100000|     100|0.00000000|  FAILED  |3532966120
          sts_serial|  15|    100000|     100|0.00000000|  FAILED  |3532966120
          sts_serial|  15|    100000|     100|0.00000000|  FAILED  |3532966120
          sts_serial|  16|    100000|     100|0.00000000|  FAILED  |3532966120
          sts_serial|  16|    100000|     100|0.00000000|  FAILED  |3532966120
         rgb_bitdist|   1|    100000|     100|0.00000000|  FAILED  |1702344236
         rgb_bitdist|   2|    100000|     100|0.00000000|  FAILED  |2132261300
         rgb_bitdist|   3|    100000|     100|0.00000000|  FAILED  |3193000911
         rgb_bitdist|   4|    100000|     100|0.00000000|  FAILED  |2176693828
         rgb_bitdist|   5|    100000|     100|0.00000000|  FAILED  |1755693687
         rgb_bitdist|   6|    100000|     100|0.00000000|  FAILED  | 481527562
         rgb_bitdist|   7|    100000|     100|0.00000000|  FAILED  |3262875243
         rgb_bitdist|   8|    100000|     100|0.00000000|  FAILED  |2264619179
         rgb_bitdist|   9|    100000|     100|0.00000000|  FAILED  |3055574385
         rgb_bitdist|  10|    100000|     100|0.00000000|  FAILED  |3202864995
         rgb_bitdist|  11|    100000|     100|0.00000000|  FAILED  |4197746056
         rgb_bitdist|  12|    100000|     100|0.00000000|  FAILED  |3102543168
rgb_minimum_distance|   2|     10000|    1000|0.00000000|  FAILED  |4074049312
rgb_minimum_distance|   3|     10000|    1000|0.00000000|  FAILED  |1411960387
rgb_minimum_distance|   4|     10000|    1000|0.00000000|  FAILED  |1592390445
rgb_minimum_distance|   5|     10000|    1000|0.00000000|  FAILED  |1158637792
    rgb_permutations|   2|    100000|     100|0.60505358|  PASSED  |2678627860
    rgb_permutations|   3|    100000|     100|0.98566896|  PASSED  |3362537243
    rgb_permutations|   4|    100000|     100|0.71066648|  PASSED  |3800652294
    rgb_permutations|   5|    100000|     100|0.00085558|   WEAK   |1355545639
      rgb_lagged_sum|   0|   1000000|     100|0.74134291|  PASSED  |2368844497
      rgb_lagged_sum|   1|   1000000|     100|0.71506737|  PASSED  |2222606451
      rgb_lagged_sum|   2|   1000000|     100|0.54568895|  PASSED  | 438630258
      rgb_lagged_sum|   3|   1000000|     100|0.37799869|  PASSED  |1936380074
     rgb_kstest_test|   0|     10000|    1000|0.03087051|  PASSED  |4188244966
     dab_bytedistrib|   0|  51200000|       1|0.00000000|  FAILED  | 146912374
             dab_dct| 256|     50000|       1|0.00000000|  FAILED  |  17815971
        dab_filltree|  32|  15000000|       1|0.00000000|  FAILED  |1023673106
        dab_filltree|  32|  15000000|       1|0.00000000|  FAILED  |1023673106
       dab_filltree2|   0|   5000000|       1|0.00000000|  FAILED  | 885195848
       dab_filltree2|   1|   5000000|       1|0.00000000|  FAILED  | 885195848
        dab_monobit2|  12|  65000000|       1|1.00000000|  FAILED  |2221575748
```

I risultati per l'algoritmo *Mersenne-Twister*:

```
#=============================================================================#
#            dieharder version 3.31.1 Copyright 2003 Robert G. Brown          #
#=============================================================================#
   rng_name    |rands/second|
        mt19937|  6.82e+07  |
#=============================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment|    Seed
#=============================================================================#
   diehard_birthdays|   0|       100|     100|0.15691373|  PASSED  |3642940582
      diehard_operm5|   0|   1000000|     100|0.89209837|  PASSED  |2603812874
  diehard_rank_32x32|   0|     40000|     100|0.36623745|  PASSED  |3532331161
    diehard_rank_6x8|   0|    100000|     100|0.72722989|  PASSED  |2680061769
   diehard_bitstream|   0|   2097152|     100|0.32226251|  PASSED  |1955527274
        diehard_opso|   0|   2097152|     100|0.21334675|  PASSED  | 332604184
        diehard_oqso|   0|   2097152|     100|0.21401414|  PASSED  |4162655506
         diehard_dna|   0|   2097152|     100|0.92594763|  PASSED  | 819122301
diehard_count_1s_str|   0|    256000|     100|0.34710510|  PASSED  |  28030129
diehard_count_1s_byt|   0|    256000|     100|0.60226321|  PASSED  |2657482734
 diehard_parking_lot|   0|     12000|     100|0.00427583|   WEAK   |1214873005
    diehard_2dsphere|   2|      8000|     100|0.03752543|  PASSED  | 801075081
    diehard_3dsphere|   3|      4000|     100|0.35771622|  PASSED  |4093632106
     diehard_squeeze|   0|    100000|     100|0.54940248|  PASSED  |3649323645
        diehard_sums|   0|       100|     100|0.33423309|  PASSED  | 641594423
        diehard_runs|   0|    100000|     100|0.99712756|   WEAK   |2760366064
        diehard_runs|   0|    100000|     100|0.88368799|  PASSED  |2760366064
       diehard_craps|   0|    200000|     100|0.80727075|  PASSED  |3526518609
       diehard_craps|   0|    200000|     100|0.41800538|  PASSED  |3526518609
 marsaglia_tsang_gcd|   0|  10000000|     100|0.18743851|  PASSED  |1895740674
 marsaglia_tsang_gcd|   0|  10000000|     100|0.42054003|  PASSED  |1895740674
         sts_monobit|   1|    100000|     100|0.48028241|  PASSED  |1721722993
            sts_runs|   2|    100000|     100|0.46789232|  PASSED  |3265004420
          sts_serial|   1|    100000|     100|0.32828481|  PASSED  |3663289047
          sts_serial|   2|    100000|     100|0.84794324|  PASSED  |3663289047
          sts_serial|   3|    100000|     100|0.80560187|  PASSED  |3663289047
          sts_serial|   3|    100000|     100|0.67574441|  PASSED  |3663289047
          sts_serial|   4|    100000|     100|0.98391690|  PASSED  |3663289047
          sts_serial|   4|    100000|     100|0.98549281|  PASSED  |3663289047
          sts_serial|   5|    100000|     100|0.55635093|  PASSED  |3663289047
          sts_serial|   5|    100000|     100|0.53438854|  PASSED  |3663289047
          sts_serial|   6|    100000|     100|0.20895150|  PASSED  |3663289047
          sts_serial|   6|    100000|     100|0.49134622|  PASSED  |3663289047
          sts_serial|   7|    100000|     100|0.98448886|  PASSED  |3663289047
          sts_serial|   7|    100000|     100|0.84748433|  PASSED  |3663289047
          sts_serial|   8|    100000|     100|0.78123523|  PASSED  |3663289047
          sts_serial|   8|    100000|     100|0.20731768|  PASSED  |3663289047
          sts_serial|   9|    100000|     100|0.56785019|  PASSED  |3663289047
          sts_serial|   9|    100000|     100|0.88991439|  PASSED  |3663289047
          sts_serial|  10|    100000|     100|0.90343701|  PASSED  |3663289047
          sts_serial|  10|    100000|     100|0.50252082|  PASSED  |3663289047
          sts_serial|  11|    100000|     100|0.25166630|  PASSED  |3663289047
          sts_serial|  11|    100000|     100|0.47487227|  PASSED  |3663289047
```

```
          sts_serial|  12|    100000|     100|0.26013879|  PASSED  |3663289047
          sts_serial|  12|    100000|     100|0.82259884|  PASSED  |3663289047
          sts_serial|  13|    100000|     100|0.51977793|  PASSED  |3663289047
          sts_serial|  13|    100000|     100|0.27384744|  PASSED  |3663289047
          sts_serial|  14|    100000|     100|0.24392404|  PASSED  |3663289047
          sts_serial|  14|    100000|     100|0.39761170|  PASSED  |3663289047
          sts_serial|  15|    100000|     100|0.17380974|  PASSED  |3663289047
          sts_serial|  15|    100000|     100|0.01834678|  PASSED  |3663289047
          sts_serial|  16|    100000|     100|0.41494073|  PASSED  |3663289047
          sts_serial|  16|    100000|     100|0.63284066|  PASSED  |3663289047
         rgb_bitdist|   1|    100000|     100|0.06794053|  PASSED  |4075949950
         rgb_bitdist|   2|    100000|     100|0.98223023|  PASSED  |1045135078
         rgb_bitdist|   3|    100000|     100|0.86805032|  PASSED  | 962660349
         rgb_bitdist|   4|    100000|     100|0.57272834|  PASSED  |2080835554
         rgb_bitdist|   5|    100000|     100|0.51362926|  PASSED  |3636550209
         rgb_bitdist|   6|    100000|     100|0.58437030|  PASSED  |2339734528
         rgb_bitdist|   7|    100000|     100|0.33034005|  PASSED  | 833566226
         rgb_bitdist|   8|    100000|     100|0.38829012|  PASSED  |1423196437
         rgb_bitdist|   9|    100000|     100|0.26903841|  PASSED  | 208467549
         rgb_bitdist|  10|    100000|     100|0.98806364|  PASSED  |4104194637
         rgb_bitdist|  11|    100000|     100|0.99155300|  PASSED  |1675791541
         rgb_bitdist|  12|    100000|     100|0.22367957|  PASSED  |  81823830
rgb_minimum_distance|   2|     10000|    1000|0.33565459|  PASSED  |3744636979
rgb_minimum_distance|   3|     10000|    1000|0.01212138|  PASSED  | 863716780
rgb_minimum_distance|   4|     10000|    1000|0.23672712|  PASSED  |2053923506
rgb_minimum_distance|   5|     10000|    1000|0.23428353|  PASSED  | 161393791
    rgb_permutations|   2|    100000|     100|0.58796560|  PASSED  |1183166591
    rgb_permutations|   3|    100000|     100|0.25235805|  PASSED  |2899153179
    rgb_permutations|   4|    100000|     100|0.60493400|  PASSED  | 213298405
    rgb_permutations|   5|    100000|     100|0.69517742|  PASSED  |1324860137
      rgb_lagged_sum|   0|   1000000|     100|0.65137305|  PASSED  |2872547827
      rgb_lagged_sum|   1|   1000000|     100|0.37722577|  PASSED  |3075655624
      rgb_lagged_sum|   2|   1000000|     100|0.75172765|  PASSED  |1893328463
      rgb_lagged_sum|   3|   1000000|     100|0.53107454|  PASSED  |2317200347
     rgb_kstest_test|   0|     10000|    1000|0.89357197|  PASSED  |1620727420
     dab_bytedistrib|   0|  51200000|       1|0.37074378|  PASSED  |3067641205
             dab_dct| 256|     50000|       1|0.75014885|  PASSED  |1851467258
        dab_filltree|  32|  15000000|       1|0.77487156|  PASSED  |4167281360
        dab_filltree|  32|  15000000|       1|0.79705635|  PASSED  |4167281360
       dab_filltree2|   0|   5000000|       1|0.77944933|  PASSED  |2842275071
       dab_filltree2|   1|   5000000|       1|0.38953587|  PASSED  |2842275071
        dab_monobit2|  12|  65000000|       1|0.64186877|  PASSED  |2507360244
```

Come si può vedere l'algoritmo *RANDU* fallisce la maggior parte dei test in modo similare agli algoritmi della presente tesi; per contro l'ottimo algoritmo di *Mersenne-Twister* riesce a superare praticamente tutti i test della suite a pieni voti (a parte due "WEAK").

Alla luce di tutte le considerazioni i due algoritmi testati non si possono ritenere dei validi generatori di numeri **pseudo-casuali**, ma nel nostro contesto ciò non è un problema in quanto non abbiamo la necessità di avere dei generatori con forti

proprietà di casualità; al contrario in alcuni ambiti, ad esempio in crittografia, tali proprietà sono necessarie e quindi si dovranno usare algoritmi migliori come quello di Mersenne-Twister.

Osservazione 13. Gli algoritmi per il campionamento proposti fanno affidamento, per la generazione del singolo numero casuale, sulle librerie dei linguaggi in cui sono implementati. Chiaramente tali funzioni influenzano anche le proprietà di casualità delle sequenze generate dagli algoritmi; è lecito aspettarsi quindi che utilizzando all'interno degli algoritmi delle funzioni migliori sia possibile ottenere risultati migliori nei test della suite "dieharder".

Conclusioni

Riassumiamo brevemente gli argomenti discussi nel corso della presente tesi.

Nel Capitolo 1 sono stati presentati alcuni concetti relativi alle basi di dati, come il modello relazionale, SQL e l'Information Schema e alcuni concetti di base del calcolo delle probabilità necessari per comprendere meglio le considerazioni e le dimostrazioni del Capitolo 3.

Nel Capitolo 2 è stato presentato il metodo lineare congruenziale, uno dei primi metodi (tuttora in uso), per la generazione di numeri pseudo-casuali. Come abbiamo verificato sperimentalmente nella Sezione 5.3 esistono vari algoritmi che si appoggiano a questo metodo: alcuni con deboli proprietà di casualità (RANDU e quello tipico delle librerie dei linguaggi) mentre altri possiedono forti proprietà di casualità (come il Mersenne-Twister); preme notare che comunque nell'ambito di questa tesi le proprietà di casualità hanno un'importanza secondaria e quindi gli algoritmi implementati dalle librerie dei linguaggi sono più che sufficienti.

Nel Capitolo 3 è stato presentato lo stato dell'arte degli algoritmi per il campionamento con e senza ripetizioni.

Nel Capitolo 4 si è invece discussa la trattazione delle possibili combinazioni di vincoli d'integrità e vincoli d'unicità che una base di dati può ammettere e si è infine presentato l'applicativo per la generazione automatica di istanze di basi di dati. L'applicativo riesce a inferire e soddisfare in modo completamente automatico i principali vincoli di integrità associati ad una base di dati; caratteristica che non è stata sempre ritrovata nei software emersi da una ricerca in rete.

Nel Capitolo 5 sono stati presentati alcuni risultati sperimentali che hanno permesso di trarre alcune interessanti conclusioni riguardo agli algoritmi presentati nel Capitolo 3 e riguardo ai possibili sviluppi futuri della tesi. Riportiamo per comodità alcune delle più importanti considerazioni emerse durante la fase di sperimentazione:

- l'Algoritmo di Floyd 3.3.1 nonostante la sua complessità quadratica (nella quantità di numeri da generare) risulta, in certi casi, migliore dell'Algoritmo di Knuth 3.3.2 che ha invece una complessità lineare (nella grandezza del range da cui estrarre i numeri);

- per contro, proprio a causa della sua complessità, l'Algoritmo di Floyd è in certi casi inutilizzabile a causa del divergere dei suoi tempi d'esecuzione;
- grazie ad un uso ragionato degli algoritmi è possibile generare un numero quasi arbitrario di dati casuali in tempi ragionevoli;
- a causa del meccanismo utilizzato per gli inserimenti nella base di dati, però, inserire già un centinaio di migliaia di record può richiedere (in funzione della velocità di rete e della potenza/carico del server) tempi molto alti;
- a causa di alcune semplificazioni nella gestione dei vincoli che una base di dati può avere l'applicativo non sempre è in grado di generare con successo un'istanza e, in certi casi, è necessario scendere a compromessi; i due problemi sono:
  - non vengono gestiti tutti i vincoli (si veda la Sezione 4.3.1 per le limitazioni d'uso);
  - nel caso di vincoli di chiave esterna non è sempre possibile generare il numero richiesto di record sulle tabelle (si veda la Sezione 5.2).

A tal proposito gli sviluppi futuri di questa tesi potrebbero riguardare: un miglioramento nella gestione dei vincoli di una base di dati, un miglioramento delle prestazioni nella generazione e nel successivo inserimento degli stessi nella base di dati e infine quello di consentire all'utente di scegliere una distribuzione di probabilità dei dati generati che sia diversa da quella uniforme.

Un ulteriore sviluppo potrebbe riguardare l'eventuale aggiunta di nuovi algoritmi per il campionamento e l'implementazione dell'algoritmo di Floyd mediante strutture dati migliori (si vedano a tal proposito le considerazioni sulla sua complessità nella Sezione 3.3.1).

Infine sarebbe interessante continuare le sperimentazioni in modo da valutare le prestazioni dell'applicativo nella generazione di istanze di maggiori dimensioni; per ottenere ciò però bisognerebbe o implementare algoritmi paralleli oppure disporre di potenze di calcolo molto superiori. Ad esempio, per poter effettuare simulazioni realistiche su istanze casuali di certi sistemi OLAP[1], bisognerebbe essere in grado di generare tabelle contenenti anche diversi TB di dati e miliardi di record.

[1] acronimo per On-Line Analytical Processing: si tratta di sistemi software per l'analisi di grandi quantità di dati. Questa è la componente principale dei sistemi di Data Warehousing utilizzati dalle aziende per le proprie ricerche di mercato.

Bibliografia

[1] Ian Sommerville: *Ingegneria del Software*. Ottava Edizione, Pearson Italia, 2007.

[2] Edsger W. Dijkstra: *Software Engineering Techniques*. Editors: J. N.Buxton and B. Randell. Rome, Italy, 27th to 31st October 1969.

[3] Ramez A. Elmasri, Shamkant B. Navathe: *Sistemi di basi di dati. Fondamenti*. Sesta Edizione, Pearson Italia, 2011.

[4] Information Schema in PostgreSQL: http://www.postgresql.org/docs/9.2/static/information-schema.html

[5] Cataloghi di sistema in PostgreSQL: http://www.postgresql.org/docs/9.2/static/catalogs.html

[6] Paolo Vidoni: *Calcolo delle Probabilità e Statistica*. Dipartimento di Scienze Economiche e Statistiche, Università degli Studi di Udine, 2001.

[7] Sheldon M. Ross: *Calcolo delle probabilità*. Seconda edizione, Apogeo, 2007.

[8] Donald Knuth: *The Art of Computer Programming. Seminumerical Algorithms*. 3rd Edition, Addison-Wesley, 1998.

[9] Dan Biebighauser: *Testing Random Number Generators*. University of Minnesota, 2000.

[10] C. T. Fan, Mervin E. Mullera, Ivan Rezuchaa: *Development of sampling plans by using sequential (item by item) selection techniques and digital computers. Journal of the American Statistical Association*. Volume 57 number 268, 1962.

[11] Terence G. Jones: *A note on sampling a tape file. Communications of the ACM*. Volume 5 number 6, 1962.

[12] Joe Bentley: *Programming Pearls*. 2nd Edition, Addison-Wesley, 2000.

[13] Joe Bentley: *A Sample of Brilliance. Communications of the ACM*. Volume 30 number 9, 1987.

[14] Jeffrey S. Vitter: *Random Sampling with a Reservoir. ACM Transactions on Mathematical Software*. Volume 11 number 1, 1985.

[15] Richard Durstenfeld: *Algorithm 235: Random permutation. Communications of the ACM*. Volume 7 number 7, 1964.

[16] Ronald A. Fisher, Frank Yates: *Statistical Tables for biological, agricultural and medical research*. Edinburgh: Oliver and Boyd, 1938.

[17] IEEE 754: http://ieeexplore.ieee.org/servlet/opac?punumber=4610933

[18] Documentazione Java SE 7: http://docs.oracle.com/javase/7/docs/api/java/util/Random.html

[19] Software statistico R: http://www.r-project.org/

[20] Software per il test di generatori di numeri casuali dieharder: http://www.phy.duke.edu/~rgb/General/dieharder.php