UNIVERSITÀ DEGLI STUDI DI PARMA
FACOLTÀ DI SCIENZE
MATEMATICHE, FISICHE e NATURALI
Corso di Laurea in Informatica

Tesi di Laurea Triennale

Analisi Comparativa di Strutture Dati per la Rappresentazione dello Spazio

Candidato:
Fabio Trabucchi

Relatore:
Prof. Alessandro Dal Palù

Anno Accademico 2007/2008

Thanks to ...

Prima di tutto voglio ringraziare la mia famiglia, i miei fratelli ma in particolor modo i miei genitori, senza di voi e senza il vostro immancabile supporto non sarei riuscito a fare nemmeno la metà delle cose che ho fatto sino ad ora nella mia vita. Solo crescendo ho capito l'importanza dei vostri amorevoli insegnamenti. Grazie di cuore.

In adolescenza ho avuto la fortuna di incontrare una persona: Cheryl. Definirla fantastica o meravigliosa non rendono abbastanza il prestigio che merita. In questi duri anni di studio (e non) mi sei sempre stata a fianco, senza mai demordere. Da te ho imparato moltissimo, più di quello che potevo immaginare. Hai saputo, e sai, regalarmi momenti indimenticabili, e solo col tempo scoprirò cos'altro ancora mi riserverai. Grazie.

Durante il mio percorso universitario ho avuto l'opportunità di incontrare molte persone e con alcune ho stretto una forte amicizia e voglio ringraziarle tutte. In maniera particolare voglio ricordare Matteo (ratto), Danilo, Dolma, Giordano, Andrea (cimmo), Irene, tutti gli informatici "nerdosi" del condominio di matematica, inclusi i poveri matematici a cui ho rotto le scatole innumerevoli volte, i miei amici biologi e gli amici di sempre, "quelli di Vigoleno", gli immancabili, sempre pronti ad offrirmi supporto e divertirsi come dei matti. Grazie per avermi sopportato (e ce ne vuole di pazienza, aimé lo so) nei momenti di crisi (e non solo), e sono stati davvero tanti, soprattutto in queste ultime settimane, ma nonostante ciò mi siete sempre stati vicini, mi avete sempre incoraggiato ad affrontare le situazioni complesse con grinta. Forse solo voi davvero riuscite a capire il significato di questo traguardo. Spero che questa stupenda amicizia non venga persa, ma di portarvi con me per tutta la vita.

In ultimo, ma di fondamentale importanza, devo ringraziare indistintamente tutti i professori incontrati in questi anni, i quali hanno profuso in me una parte della loro conoscenza.

Indice

Capitolo 1

Introduzione

L'argomento affrontato in questa tesi è quello della rappresentazione e manipolazione di dati nello spazio tridimensionale. In diverse applicazioni viene richiesta l'elaborazione di modelli spaziali come ad esempio nel videogame programming oppure in diversi campi della bioinformatica e modellistica molecolare; in questi ultimi l'intervento dell'informatica è ormai necessario per predire comportamenti di strutture organiche come DNA e proteine o per simulare interazioni tra farmaco e recettore. Problemi "classici" trovano come ambiente naturale lo spazio bidimensionale e tridimensionale, ma altri tipi di applicazioni, come i database o linguaggi di programmazione logica, hanno la necessità di elaborare informazioni in spazi multidimensionali e insiemi generici di elementi multidimensionali.

È semplice accorgersi che all'aumentare del numero di dimensioni di un problema la complessità di quest'ultimo aumenta considerevolmente, ed è per questo che l'informatica interviene cercando di ottimizzare la rappresentazione di informazioni multidimensionali studiando nuove strutture dati e algoritmi correlati.

Questa tesi cerca di fornire elementi utili per la scelta di strutture dati per rappresentare e manipolare informazioni in spazi multidimensionali tramite un'analisi comparativa di due diverse strutture dati. Nel Capitolo 2 cerchiamo di fornire le nozioni preliminari sulla rappresentazione di spazi multidimensionali ed alcuni suoi metodi di rappresentazione con le relative strutture dati; nel Capitolo 3 viene descritto lo scopo della tesi; nel Capitolo 4 vengono forniti i dettagli generali di implementazione di due strutture dati, dove poi nel Capitolo 5 ne verranno valutate le loro performance e nel Capitolo 6 verranno tratte le conclusioni su di esse.

Capitolo 2

Background

Per poter comprendere meglio lo scopo di questa tesi è necessario approfondire il concetto di spazio n-dimensionale, illustrare qualche suo metodo di rappresentazione ed inoltre, con l'aiuto di alcuni esempi, l'utilizzo in applicazioni e campi di ricerca.

2.1 Spazi, oggetti e punti n-dimensionali

Concettualmente un punto non è altro che un elemento di un generico insieme o *dominio*. Cos'è dunque un punto n-dimensionale? Per rispondere a questa domanda dobbiamo definire i concetti di spazio e oggetto n-dimensionale.

Definizione 2.1. *Sia $n \in \mathbb{N}^+$ fissato e siano $D_0, \ldots, D_{n-1}$ domini qualsiasi. Definiamo* ***spazio n-dimensionale*** *il prodotto cartesiano*

$$D_0 \times \cdots \times D_{n-1}$$

Definizione 2.2. *Definiamo* ***oggetto n-dimensionale*** *ogni sottoinsieme di uno spazio n-dimensionale.*

$$O \subseteq D_0 \times \cdots \times D_{n-1}$$

Definizione 2.3. *Definiamo* ***punto n-dimensionale*** *un qualsiasi elemento di uno spazio n-dimensionale.*

$$p \in D_0 \times \cdots \times D_{n-1}$$

Per comprendere meglio queste definizioni possiamo fare un'analogia con la terminologia dei database. La struttura base del modello relazionale è il *dominio*, definito come l'insieme dei valori che può assumere un determinato *attributo* (punto). Una *relazione di dimensione n* (oggetto n-dimensionale) è un sottoinsieme del *prodotto cartesiano di n domini* (spazio n-dimensionale). Una *tupla* (punto n-dimensionale) è un insieme di valori dei vari attributi. I moderni RDBMS applicano un processo di enumerazione sulle varie relazioni per velocizzare l'accesso ai dati, meglio conosciuto come **indicizzazione**. Quando un indice viene generato considerando solo una *chiave* di ricerca, quest'indice si dice monodimensionale, altrimenti, se l'indice viene ricavato da più chiavi di ricerca, si dice n-dimensionale [Garcia Molina H. et al., 2002]. Gli oggetti basati sul modello generico di spazio n-dimensionale, così come lo abbiamo definito, non possono essere elaborati dai calcolatori, proprio per la natura generica dei domini su cui si basa lo spazio, i quali possono anche essere domini continui. In questo caso per poter elaborare informazioni provenienti da spazi n-dimensionali qualsiasi dobbiamo cercare di

ricondurci allo spazio $\mathbb{N}^n$, ovvero si deve effettuare una **discretizzazione dello spazio**. La discretizzazione è un processo che, attraverso vari step di quantizzazione, cerca di portare modelli continui nella loro controparte discreta [Wikipedia, 2009a]. La **quantizzazione** è quel processo di "approssimazione" di un insieme di valori continui ad un insieme di valori discreti [Wikipedia, 2009b]. Un banale esempio di funzione di quantizzazione può essere la funzione *parte intera*. $\lfloor \cdot \rfloor : \mathbb{R} \longmapsto \mathbb{N}$

Matematizzando il processo di discretizzazione con una qualche funzione

$$f : \mathbb{R}^n \longmapsto \mathbb{N}^n$$

si può quindi definire lo spazio n-dimensionale discreto e limitato che prenderemo in considerazione nel resto della tesi.

Definizione 2.4. *Sia $k \in \mathbb{N}$ fissato, tale che $\forall i \in \mathbb{N}^+,\ k = 2^i$ e sia $n \in \mathbb{N}^+$ fissato, definiamo dominio discreto e limitato, in breve **dominio**, l'insieme*

$$D_k^n = \{0, \ldots, k-1\}^n \subset \mathbb{N}^n$$

Descriviamo quindi un insieme di punti in un dominio D_k^n con la proprietà caratteristica di appartenenza, ovvero

Definizione 2.5. *Dato un insieme $O \subseteq D_k^n$, si definisce la funzione*

$$\varphi_O : D_k^n \longmapsto \mathbb{B} \quad \textit{tale che} \quad \forall p \in D_k^n,\ \varphi_O(p) \iff p \in O$$

2.1.1 Spazi e oggetti geometrici

Un particolare spazio n-dimensionale è lo spazio $\mathbb{R}^n$, dove è possibile rappresentare oggetti geometrici come punti, linee, curve, superfici e solidi. I punti che descrivono un oggetto possono essere interpretati in due modi diversi: punti caratteristici e punti vettoriali; in quest'ultimo tipo di interpretazione vengono associate delle equazioni matematiche (chiamate primitive grafiche) ai punti che descrivono l'oggetto; per esempio, nello spazio $\mathbb{R}^2$, i punti dell'insieme $\{(1,1),(2,2),(1,2)\}$ possono essere interpretati, a seconda delle primitive associate, sia come punti singoli, sia come estremi di una curva, sia come vertici di un triangolo. Solitamente, quando si devono analizzare dati sperimentali, non si conosce la natura dell'oggetto, quindi non lo si può descrivere a priori mediante primitive grafiche, dunque prenderemo come naturale descrizione di un oggetto geometrico, l'interpretazione tale per cui l'insieme dei punti che lo compongono è la caratterizzazione della sua struttura nello spazio.

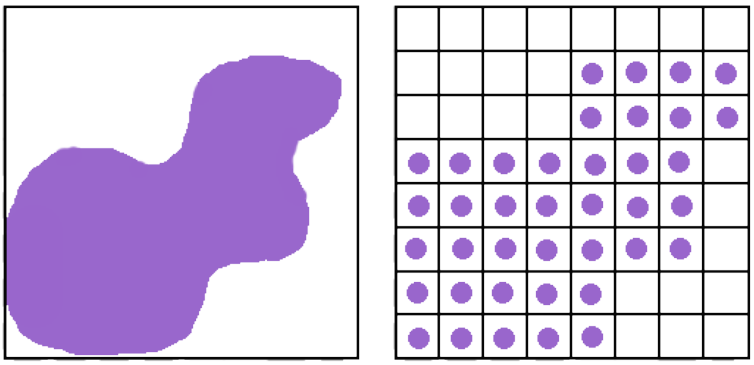

Figura 2.1: Oggetto in $\mathbb{R}^2$ discretizzato in D_8^2.

2.2 Operazioni

Definiamo ora le principali operazioni sui domini, che in seguito ci serviranno per operare sugli oggetti, utilizzando la precedente definizione 2.5.

Definizione 2.6. *Siano* $A, B \subseteq D_k^n$. *Definiamo* ***unione*** *la funzione*

$$\cup : D_k^n \times D_k^n \longmapsto D_k^n$$

$$A \cup B = \{c \in D_k^n \mid (\varphi_A(c) \ \vee \ \varphi_B(c))\}$$

Definizione 2.7. *Siano* $A, B \subseteq D_k^n$. *Definiamo* ***intersezione*** *la funzione*

$$\cap : D_k^n \times D_k^n \longmapsto D_k^n$$

$$A \cap B = \{c \in D_k^n \mid (\varphi_A(c) \ \wedge \ \varphi_B(c))\}$$

Definizione 2.8. *Siano* $A, B \subseteq D_k^n$. *Definiamo* ***differenza*** *la funzione*

$$\setminus : D_k^n \times D_k^n \longmapsto D_k^n$$

$$A \setminus B = \{c \in D_k^n \mid (\varphi_A(c) \ \wedge \ \neg\varphi_B(c))\}$$

2.3 Metodi di rappresentazione

In questa tesi restringiamo il campo di interesse della rappresentazione di oggetti in spazi bidimensionali e tridimensionali descritti da regioni di spazio come celle e volumi. Per semplicità e necessità le argomentazioni saran-

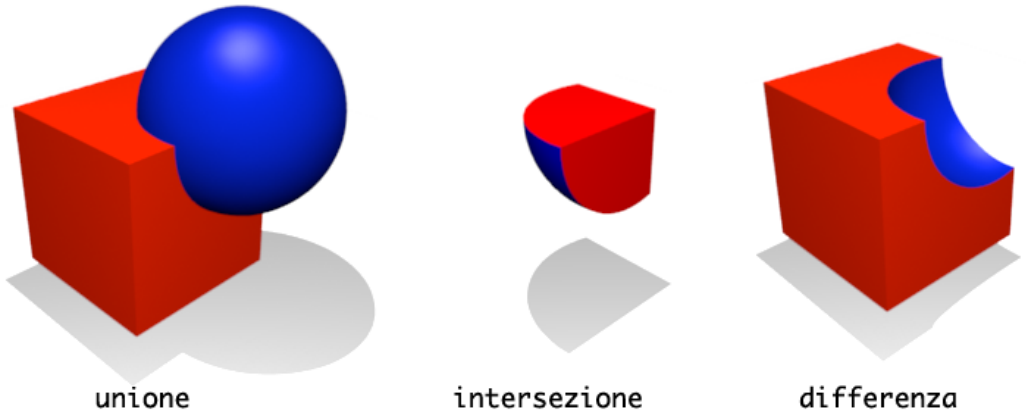

Figura 2.2: Operazioni insiemistiche nella terza dimensione.

no trattate con problemi ed esempi soprattutto bidimensionali, in modo da tenere un'analogia molto simile alle dimensioni superiori.

Preso atto che un oggetto in un dominio è un insieme di punti, quale è il modo più *efficiente* di rappresentarlo e manipolarlo? Per diversi motivi e soprattutto per diversi tipi di uso degli oggetti, si può rappresentare un insieme di punti "approssimando" lo spazio da loro occupato, ovvero rappresentando la minor area rettangolare che i punti occupano, quindi le coordinate della frontiera rettangolare che racchiude l'insieme di punti. Questa tecnica è chiamata **bounding box**. Ovviamente le operazioni su un oggetto rappresentato mediante bounding box sono estremamente efficienti, infatti si può rappresentare un oggetto composto da un numero elevato di punti con un esiguo numero di punti, mai superiori alla dimensione dell'oggetto. Questo tipo di rappresentazione è molto efficiente, ma allo stesso tempo molto inaccurata, quindi inadatta se si vuole rappresentare *puntualmente* ogni elemento dell'oggetto.

Un metodo abbastanza intuitivo per la rappresentazione puntuale di un

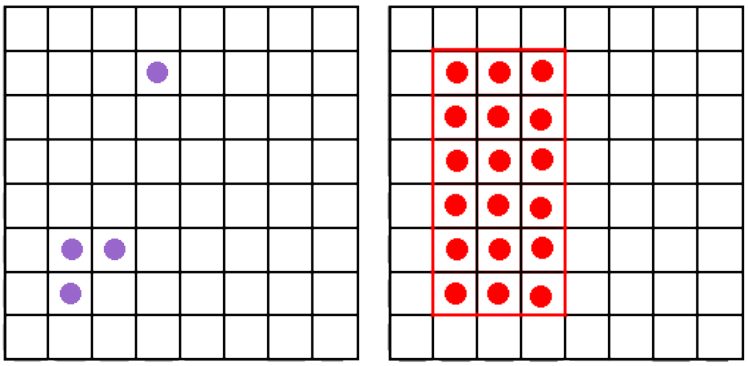

Figura 2.3: Tre punti in D_8^2 con la relativa bounding box.

oggetto è quello di usare una struttura dati lineare, come array o liste, per memorizzare l'insieme di punti. Metodo efficace se la rappresentazione dell'oggetto è fine a se stessa, ma quando bisogna supportare delle operazioni sugli oggetti, quindi sulle strutture dati, gli array e liste non hanno un comportamento efficiente. Il dominio è rappresentato con una matrice n-dimensionale a valori booleani dove ogni punto corrisponde all'indice n-dimensionale della matrice. Questa rappresentazione è accurata, ma le operazioni e lo spazio di memorizzazione sono nell'ordine di $\Omega(n)$, quindi risultano ancora troppo costose.

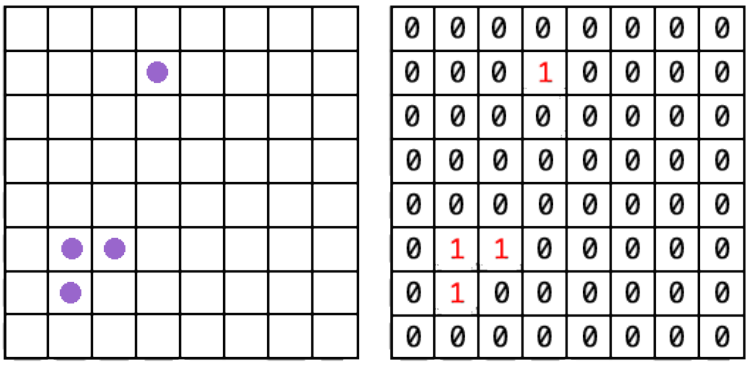

Figura 2.4: Tre punti in D_8^2 con la relativa matrice.

2.3.1 Partizionamento

Subentra quindi la necessità di *partizionare l'insieme di punti*. Si sono studiati metodi di decomposizione dello spazio tali per cui la rappresentazione dei punti comporta, in funzione della natura degli oggetti, un risparmio di spazio di memorizzazione e tempo di esecuzione delle operazioni rispetto a strutture dati classiche. Esistono dunque due metodi principali per decomporre oggetti n-dimensionali. Il primo metodo consiste nell'aggregazione dei punti in blocchi in base alla loro "prossimità". Lo svantaggio di questo metodo è che i blocchi risultano non disgiunti. Il secondo metodo si basa sulla decomposizione ricorsiva dello spazio in blocchi disgiunti con specifiche proprietà legate allo spazio [Samet H., 2005].

Quando una decomposizione è ricorsiva la k-esima decomposizione (stage k) è il risultato di una $(k-1)$-esima decomposizione (stage $k-1$). Questo significa che un blocco b ottenuto da uno stage i verrà decomposto in altri blocchi b_j. Un blocco b_j verrà a sua volta decomposto, nello stage $i+1$, con lo stesso criterio precedente. Alcune regole di decomposizione restringono le possibili proprietà dei blocchi:

- blocchi congruenti ad ogni stage
- blocchi simili in tutti gli stage
- i lati dei blocchi sono tutti uguali ad ogni stage
- i lati dei blocchi sono sempre una potenza di due

Questo tipo di decomposizione viene chiamato **regular decomposition** ed è solitamente usato per descrivere la classe di decomposizione ricorsiva che

aggrega celle aventi le stesse proprietà geometriche in blocchi simili e disgiunti tra loro. In particolare la tecnica **quadtree**, suddivide, ad ogni stage di decomposizione, un blocco in quattro (2^2) sottoblocchi fino ad arrivare, condizione di terminazione della decomposizione, al blocco contenente una singola cella; in modo analogo, nel caso tridimensionale, si adotta la tecnica **octree** che suddivide, ad ogni stage di decomposizione, un blocco in otto (2^3) sottoblocchi fino ad arrivare al blocco contenente un singolo voxel.

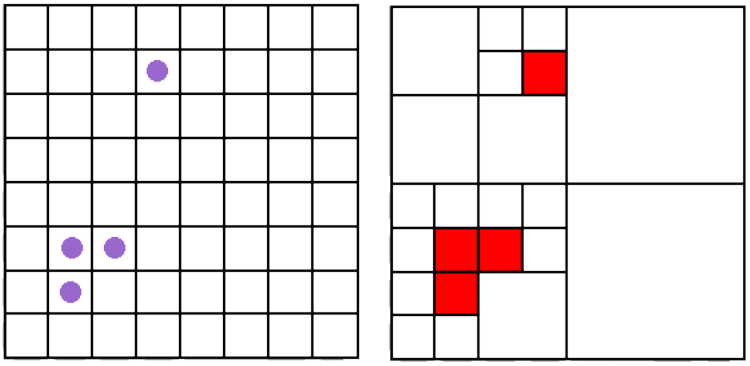

Figura 2.5: Tre punti in D_8^2 con il relativo quadtree.

Regole di decomposizione che non restringono le possibili proprietà dei blocchi rientrano nella classe **bucketing decomposition**. Questo tipo di decomposizione permette a blocchi diversi di poter descrivere porzioni di spazio simili, quindi i blocchi risultano essere bounding box non-disgiunte generando una serie di blocchi gerarchici al cui interno sono presenti un numero prefissato di punti.

2.3.2 Rappresentazione nei database

Come abbiamo cercato di far notare, la rappresentazione di oggetti n-dimensionali è strettamente legata al concetto di indicizzazione di relazioni di database,

quindi osserviamo le varie rappresentazioni che diversi tipi di database adottano per rappresentare oggetti n-dimensionali. La generazione di indici n-dimensionali non è banale, soprattutto per il fatto di dover "soddisfare" una tipologia di query non comune, in applicazioni come i GIS. I tipi di query che maggiormente vengono usate in queste applicazioni sono:

partial match queries: dati i valori per una o più dimensioni, cercare tutti i punti corrispondenti ai valori in quelle dimensioni

range queries: dato un range per una o più dimensioni, cercare una serie di punti all'interno di tali intervalli

nearest-neighbor queries: dato un punto, cercare il punto più vicino

where-am-I queries: dato un punto, vogliamo sapere, se esiste, la regione in quale si trova

Questi tipi di query possono essere eseguite anche con il solo supporto di indici monodimensionali, ma la velocità di risposta risulta compromessa e l'integrità del risultato non è sempre certa, specialmente quando si operano nearest-neighbor queries e range queries. La difficoltà sta nel cercare di rappresentare questi indici in strutture dati capaci di conferire una "proprietà spaziale" all'oggetto indicizzato [Garcia Molina H. et al., 2002, Samet H., 1999].

2.4 Strutture dati

Presentiamo ora una breve rassegna sulle struttura dati più usate per realizzare le diverse tecniche di rappresentazione di spazi n-dimensionali descritte nella sezione precedente.

2.4.1 B-tree

I B-tree sono strutture dati ad albero comunemente utilizzati nell'ambito di database e filesystem usati per indicizzare spazi mono/bidimensionali. Essi derivano dagli alberi binari di ricerca (BST), in quanto ogni chiave appartenente al sottoalbero sinistro di un nodo è di valore inferiore rispetto a ogni chiave appartenente ai sottoalberi alla sua destra; questa struttura deriva anche dagli alberi bilanciati perché tutte le foglie si trovano alla stessa distanza rispetto alla radice. Il vantaggio principale dei B-tree è che essi mantengono automaticamente i nodi bilanciati permettendo operazioni di inserimento, cancellazione e ricerca in tempi ammortizzati logaritmici nel numero di oggetti rappresentati. Le caratteristiche principali sono:

- Assegnare ad un B-tree un ordine n significa dire che i nodi interni (esclusa la radice) conterranno da n/2 a n-1 chiavi.
- Una corrispondente lista di puntatori è situata tra le chiavi per indicare dove cercare una chiave se non è nel nodo corrente. Un nodo contenente k chiavi contiene sempre k+1 puntatori. Il numero di puntatori è detto fattore di ramificazione.
- La radice contiene almeno una chiave.
- In un nodo le chiavi sono ordinate in ordine crescente.
- Ad ogni chiave è associato un puntatore ad una informazione.

Aumentando il numero di nodi figli per ogni nodo, l'altezza dell'albero si riduce, l'operazione di bilanciamento è meno richiesta e quindi l'efficienza aumenta. [Cormen T. H. et al., 2001]

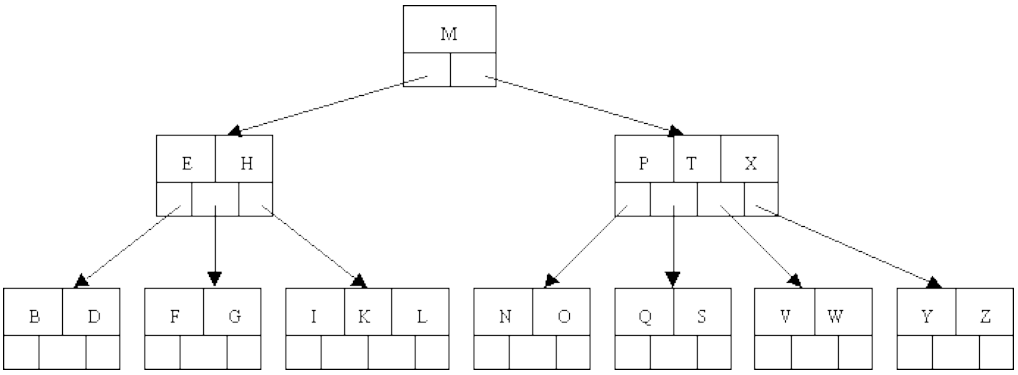

Figura 2.6: Esempio di B-tree di ordine 4.

2.4.2 R-tree

Gli R-tree sono strutture dati ad albero simili al B-tree, ma usati per indicizzare spazi n-dimensionali, ad esempio le coordinate spaziali (X, Y, Z) per dati geografici. Una richiesta di esempio che usi un R-tree potrebbe essere "Trova tutti i musei entro 2 km dalla mia posizione attuale". La struttura dati divide lo spazio in minimum bounding rectangles (MBR), infatti R-tree deriva proprio da Rectangle-tree, innestati gerarchicamente e quando possibile sovrapposti. Ogni nodo dell'R-tree ha un numero variabile di entry (fino ad un massimo predeterminato). Ogni entry che non sia un nodo foglia contiene due entità: una identifica il nodo figlio, l'altra l'MBR che contiene tutte le entry del nodo figlio. L'algoritmo di inserimento e cancellazione di entry dagli MBR assicura che elementi "vicini" siano posizionati nello stesso nodo foglia: un nuovo elemento andrà nel nodo foglia che richiede il minor numero di estensioni delle dimensioni dell'MBR. Gli algoritmi di ricerca usano gli MBR per decidere se cercare o meno nel nodo figlio del nodo corrente. In questo modo la maggior parte dei nodi non viene esplorata dagli algoritmi. Per questo motivo, come per i B-tree, ciò rende gli R-tree adatti ai database,

dove i nodi possono essere copiati in memoria solo quando necessario. Diversi algoritmi possono essere usati per dividere i nodi quando diventano troppo estesi, ovvero quando in un nodo viene aggiunto un numero di elementi che supera il limite prestabilito. Gli R-tree non garantiscono una performance ottima di caso peggiore, ma in generale si comportano molto bene con dati reali.

Nell'operazione di ricerca, simile a quella dei B-tree, l'input è un rettangolo MBR. Essa parte dal nodo radice e si estende ad ogni nodo figlio (contenente sia rettangoli MBR che puntatori ai nodi figli). Giunti al nodo foglia si hanno i veri e propri punti n-dimensionali. Per ogni MBR incontrato si verifica se esso è sovrapposto con il rettangolo di input o meno, e si continua la ricerca nel nodo figlio corrispondente se e solo se è sovrapposto. La ricerca procede quindi in un modo ricorsivo, fermandosi quando tutti gli MBR sovrapposti sono stati esplorati. Un punto viene aggiunto all'insieme di ricerca (l'output dell'algoritmo) quando si trova in un MBR che si sovrappone all'MBR query.

Per inserire un elemento, l'albero è visitato ricorsivamente dal nodo radice. Un elemento è inserito in un MBR che necessita il minor numero di estensione delle dimensioni che l'aggiunta comporterebbe. A parità di numero di estensioni, viene scelto il nodo con MBR di area minima. Trovato il nodo foglia corretto, viene aggiunto il punto vero e proprio. Se viene aggiunto un nodo ad un MBR che contiene già un numero limite di elementi, la regione (MBR) viene divisa in due regioni con un algoritmo di split. Tale algoritmo realizza le due nuove regioni tendendo a creare MBR con area minima tra tutti quelli possibili. [Cormen T. H. et al., 2001]

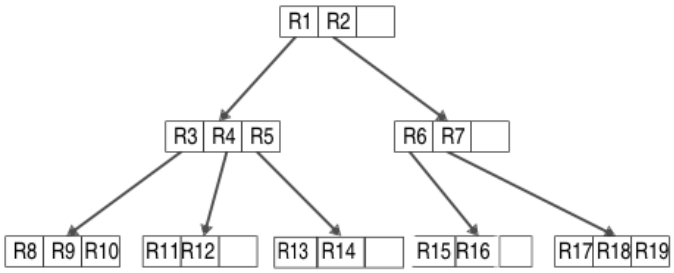

Figura 2.7: Esempio di R-tree di ordine 3.

2.4.3 Quadtree

Un quadtree è un struttura dati ad albero non bilanciata nella quale tutti i nodi interni hanno esattamente quattro nodi figli. I quadtree sono spesso usati per partizionare uno spazio bidimensionale suddividendolo ricorsivamente in quattro quadranti, comunemente denotati come NE, NW, SE, SW (Nord-Est, Nord-Ovest, Sud-Est, Sud-Ovest). Gli utilizzi comuni dei quadtree sono: rappresentazione di immagini, indicizzazione spaziale, determinazione di collisioni in due dimensioni, memorizzazione di dati sparsi. I quadtree rappresentano un insieme di punti bidimensionali partizionandolo in quattro sottoinsiemi, che possono, a loro volta venir partizionati, e così via sino ai nodi foglia. Il termine quadtree è usato per descrivere la classe di strutture dati gerarchiche aventi in comune la decomposizione ricorsiva dello spazio. La risoluzione della decomposizione, ovvero il numero di volte in cui viene attuato il processo di decomposizione, può essere fissato a priori oppure dipendente dai dati da rappresentare. L'interesse della rappresentazione quadtree è la rappresentazione di un dominio D_k^2 basandosi sulla suddivisione ricorsiva del dominio in quattro quadranti della stessa misura. Se un quadrante non contiene uniformemente uno o zero, allora sarà suddiviso in quadranti, sottoquadranti etc., finché vengono ottenuti blocchi a contenuto

uniforme [Samet H., 1988].

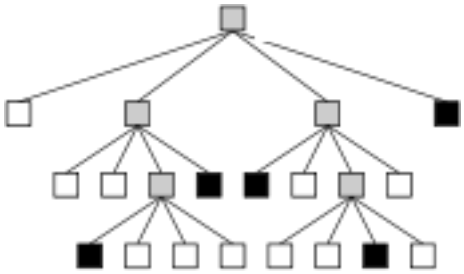

Figura 2.8: Esempio di quadtree con $k = 3$.

Nella rappresentazione ad albero, la radice corrisponde all'intero dominio. Ogni figlio di un nodo rappresenta un quadrante NE, NW, SE, SW della regione rappresentata dal nodo. I nodi foglia corrispondono a regioni contenenti solo uno o zero. Ogni nodo foglia può quindi essere WHITE (0) o BLACK (1), ogni nodo non foglia è GREY, perché può contenere sia nodi WHITE, sia nodi BLACK, sia nodi GREY. Dato un dominio D_k^2, il nodo radice ha profondità 0 mentre la profondità massima a cui un nodo foglia può arrivare è k [Samet H., 2002]

La motivazione principale dello sviluppo dei quadtree è quella di risparmiare lo spazio attraverso l'uso dell'aggregazione omogenea dei blocchi. Il caso peggiore per un quadtree di una certa profondità nel numero dei nodi necessari, è quando si ha un oggetto nel dominio con un pattern a scacchiera. Con un piccolo esempio [Dayer C.R., 1982] ha mostrato un piazzamento arbitrario di un quadrato $2^m \times 2^m$ in ogni posizione in un dominio D_k^2, con $m < k$, ciò ha richiesto una media di $O(2^{m+2} + k - m)$ nodi. Una ulteriore caratterizzazione di questo risultato è che lo spazio necessario è quantificabile in $O(p+k)$ dove p è il perimetro dei blocchi. Il risultato di [Dayer C.R., 1982] è un caso particolare del teorema di [Hunter G.M. and Steiglitz K., 1979], i

quali hanno ottenuto gli stessi risultati per poligoni semplici (senza intersezioni dei bordi e buchi). Questo risultato è stato inoltre osservato in altri esperimenti con immagini arbitrarie.

Teorema 2.1. *Il numero massimo di nodi di un quadtree corrispondente ad un poligono di perimetro p in un dominio* D_k^2 *è*

$$24 * k - 19 + 24 * p \quad ovvero \quad O(p + k)$$

[Meagher D., 1980] ha mostrato che questo teorema si adatta anche alla terza dimensione dove il perimetro è sostituito dall'area di superficie. Il perimetro e l'area di superficie corrispondono alla misura del bordo di un poliedro, quindi questo teorema va bene per qualsiasi dimensione. Inoltre i quadtree sono efficienti su operazioni insiemistiche come intersezione ed unione, infatti, considerando l'operazione di intersezione tra due quadtree, il tempo di esecuzione, nel caso peggiore, è proporzionale al tempo di visita della somma dei nodi nei quadtree.

2.4.4 ROBDD

ROBDD, acronimo di **Reduced Ordered Binary Decision Diagram**, è una struttura dati DAG (directed acyclic graph) studiata per la rappresentazione simbolica di funzioni booleane. Molti problemi di progettazione di sistemi digitali, ottimizzazione combinatoria, logica matematica, e intelligenza artificiale possono essere formulati in termini di operazioni su domini finiti. Con l'introduzione di una codifica binaria degli elementi in questi settori, questi problemi possono essere ulteriormente ridotta a operazioni su valori booleani.

Utilizzando una rappresentazione simbolica di funzioni booleane, possiamo esprimere un problema in una forma molto generale. Pertanto, un metodo efficace per la rappresentazione e la manipolazione simbolica di funzioni booleane può portare alla soluzione di una vasta classe di problemi complessi. Reduced Ordered Binary Decision Diagram [Bryant R. E., 1986] fornisce una tale rappresentazione. Questa rappresentazione è definita mediante l'imposizione di restrizioni, introdotte da [Lee C. Y., 1959, Akers S. B., 1978], sulla struttura base BDD, tali che risultino in forma canonica. Tali restrizioni e la conseguente canonicità sono stati riconosciuti da [Fortune S. et al., 1978]. Le funzioni vengono rappresentate come grafi aciclici diretti, con vertici interni corrispondenti alle variabili su cui la funzione è definita e vertici terminali classificati dai valori 0 e 1 [Bryant R. E., 1992].

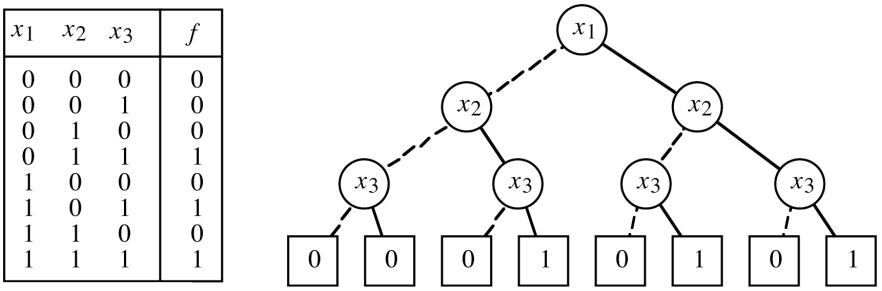

Figura 2.9: Rappresentazione di una funzione booleana mediante una tabella di verità e un albero binario di decisione. I rami tratteggiati (non tratteggiati) dell'albero denotano il caso di decisione 0 (1) della variabile corrispondente.

Un BDD rappresenta una funzione booleana come un grafo radicato aciclico. Nell'esempio di figura 2.9 viene illustrata una rappresentazione della funzione $f(x_1, x_2, x_3)$ definita dalla tabella di verità nel caso particolare in cui il grafo è in realtà un albero. Ogni vertice v non terminale è etichettato da una variabile $var(v)$ e ha due archi diretti verso i figli: $lo(v)$ (mostrato da una linea tratteggiata) corrisponde al caso in cui alla variabile è assegnato il

valore 0; $hi(v)$ (mostrato da una linea continua) corrisponde al caso in cui alla variabile è assegnato il valore 1. Ogni vertice terminale è etichettato con 0 o 1. Per una data assegnazione di valori alle variabili, il valore restituito dalla funzione è determinato dal percorso che parte dalla radice fino ad arrivare ad un vertice terminale.

Imponendo un ordine totale all'insieme di variabili, il BDD si definisce OBDD. La relazione di ordine impone che, per ogni vertice u, ed un suo vertice figlio v non terminale, le rispettive variabili devono essere ordinate $var(u) < var(v)$. Nell'albero della figura 2.9, ad esempio, le variabili sono ordinate $x_1 < x_2 < x_3$. Sul grafo OBDD si definiscono tre regole di trasformazione che non alterano la funzione rappresentata:

a) rimuovere i vertici terminali duplicati.

b) rimuovere i vertici non terminali duplicati: se i vertici non terminali u e v hanno $var(u) = var(v),\ lo(u) = lo(v),\ hi(u) = hi(v)$ si elimina uno dei due vertici e si riorientano tutti gli archi entranti del vertice eliminato sul vertice restante.

c) rimuovere test ridondanti: se un vertice non terminale v ha $lo(v) = hi(v)$ si elimina v e si riorientano tutti gli archi entranti in v nel vertice $lo(v)$.

A partire da qualsiasi OBDD si può ridurre la sua dimensione più volte applicando le regole di trasformazione arrivando dunque ad un ROBDD. Ad esempio, la figura 2.10 illustra la riduzione dell'albero mostrato in figura 2.9 a un ROBDD. Applicando la prima regola di trasformazione (a) si riducono gli otto vertici terminali a due. Applicando la seconda regola di trasforma-

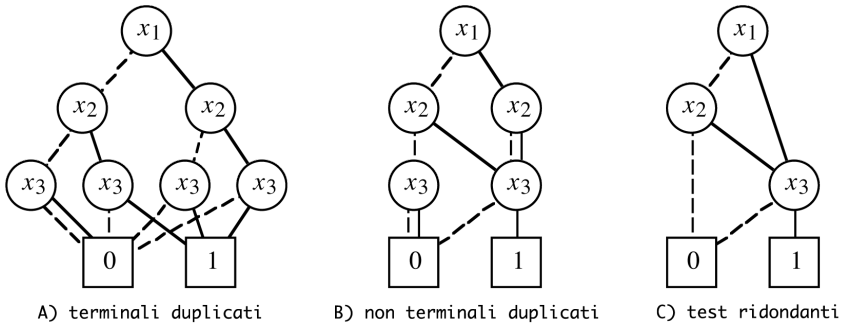

Figura 2.10: Passaggio da BDD a ROBDD.

zione (b) si eliminano due dei vertici etichettati x_3. Con l'applicazione della terza regola di trasformazione (c), si eliminano due vertici: uno di variabile x_3 e di variabile x_2. In generale, le regole di trasformazione devono essere applicate più volte fino al raggiungimento di un punto fisso, dal momento che ogni trasformazione può esporre nuove possibilità per ulteriori trasformazioni. Una rappresentazione di una funzione è canonica per un determinato ordine, quindi due ROBDD rappresentanti la stessa funzione sono isomorfi. Questa proprietà ha diverse conseguenze importanti. Come le figure 2.9 e 2.10 illustrano, possiamo costruire un ROBDD rappresentante una funzione booleana data la sua tabella di verità e di ridurre la costruzione di un albero decisionale. Questo approccio è pratico, tuttavia, solo per funzioni con un piccolo numero di variabili, dal momento che sia la tabella di verità sia l'albero decisionale hanno dimensioni esponenziali nel numero di variabili. Invece, usando questa struttura dati, la valutazione di funzioni booleane risulta più efficiente. [Bryant R. E., 1986]

In linea di principio, l'ordinamento delle variabili può essere selezionata arbitrariamente; in pratica, la selezione di un ordinamento soddisfacente è di **fondamentale** importanza per un'efficiente manipolazione simbolica. La

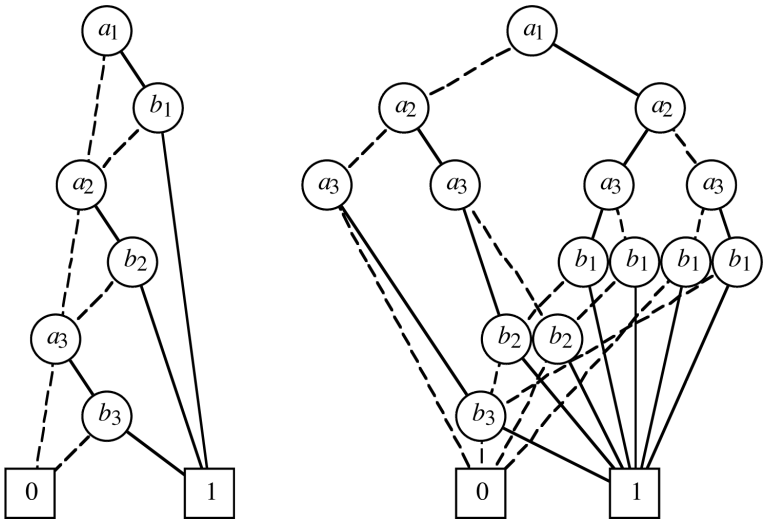

Figura 2.11: Due ROBDD che rappresentano la stessa funzione booleana, ma con ordinamenti diversi.

forma e le dimensioni di un ROBDD dipendono dall'ordinamento delle variabili. Per esempio, la figura 2.11 mostra due rappresentazioni ROBDD della funzione indicata dall'espressione $(a_1 \wedge b_1) \vee (a_2 \wedge b_2) \vee (a_3 \wedge b_3)$. Per il caso a sinistra, le variabili sono ordinate $a_1 < b_1 < a_2 < b_2 < a_3 < b_3$, mentre per il caso sulla destra sono ordinate $a_1 < a_2 < a_3 < b_1 < b_2 < b_3$. Possiamo generalizzare questa funzione per più variabili.

$$(a_1 \wedge b_1) \vee (a_2 \wedge b_2) \vee \cdots \vee (a_n \wedge b_n)$$

Usando il primo ordinamento $a_1 < b_1 < \cdots < a_n < b_n$ si genera un ROBDD con $2n$ vertici non terminali (uno per ogni variabile). Usando il secondo ordinamento $a_1 < a_2 < \ldots a_n < b_1 < b_2 < \cdots < b_n$ si genera un ROBDD con $2(2^n - 1)$ vertici non terminali. Per valori di n grandi, la differenza tra la crescita lineare del primo ordinamento contro la crescita esponenziale del secondo ordinamento ha un effetto negativo sulla conservazione e l'effi-

cienza della manipolazione degli algoritmi. Per la maggior parte delle applicazioni che utilizzano ROBDD, scegliere l'ordine delle variabili è una fase cruciale ed essenziale. Quest'ordine può essere scelto o manualmente, o da un analisi euristica del particolare sistema con cui si modella il problema [Bryant R. E., 1992].

Capitolo 3

Scopo della tesi

Dopo aver fornito una breve descrizione sugli spazi n-dimensionali e alcuni loro metodi di rappresentazione, con questa tesi si cercano di offrire, attraverso un'analisi comparativa, informazioni utili ed interessanti per una migliore scelta di strutture dati a fronte di una specifica richiesta di utilizzo di dati n-dimensionali.

Per la fase di implementazione e la successiva fase di valutazione delle strutture dati, abbiamo scelto di rappresentare insiemi di punti nel dominio tridimensionale D_k^3.

Le strutture dati candidate sono ROBDD ed OCTREE. Abbiamo scelto queste fra le altre disponibili per:

- ✓ la loro versatilità rispetto al valore della dimensione dello spazio da rappresentare;
- ✓ la loro capacità di risparmiare lo spazio di memorizzazione dei dati;
- ✓ la loro efficienza algoritmica rispetto alle operazioni insiemistiche.

Le valutazioni sperimentali si baseranno sui risultati di test condotti sulle strutture dati in termini di complessità computazionale per le operazioni di inserimento, unione, intersezione e differenza insiemistica cercando di quantificare sia lo spazio usato per rappresentare insiemi di punti, sia i tempi di esecuzione.

Per fornire elementi utili nella scelta di una struttura dati, a seconda del tipo di applicazione a cui sarà destinata, verranno eseguiti diversi tipi di test, sia in funzione del valore k nel dominio D_k^3, sia in funzione della densità di punti nel dominio, in modo da poter avere un ampio spettro di valori confrontabili per osservare i diversi comportamenti delle strutture dati e determinare le situazioni in cui si verificano i casi pessimi ed i casi ottimi.

Capitolo 4

Implementazione

In questo capitolo presentiamo il lavoro di implementazione svolto per confrontare alcune strutture dati descritte nella sezione 2.4: OCTREE e ROBDD. Nella sezione 4.1 presentiamo la classe **Point** e la classe virtuale **Cube**, necessarie per una corretta implementazione delle successive classi specializzate; nella sezione 4.2 descriviamo l'implementazione della classe derivata **BddCube** e nella sezione 4.3 descriviamo l'implementazione della classe derivata **OctreeCube** e della classe **Node**.

4.1 Point, Cube

Abbiamo creato una classe **Point** come implementazione di un punto tridimensionale. Nel nostro caso, lavorando in D_k^3, un qualsiasi valore negativo assegnato ad **x** o **y** o **z**, conferisce al punto lo stato di *vuoto*.

```
class Point {
  int x, y, z;
  Point(int X = -1, int Y = -1, int Z = -1);
  bool empty() const;
};
```

Abbiamo implementato una classe che rispecchi il dominio D_k^3 su cui abbiamo basato il nostro lavoro, una classe virtuale pura (interfaccia), chiamata **Cube** con i metodi di inserimento di un punto nel dominio, rimozione di un punto nel dominio, verifica di esistenza di un punto nel dominio, verifica di empty e full del dominio, restituzione di un vettore di punti rappresentati nel dominio.

```
class Cube {
  virtual ~Cube() {};
  virtual void insert(const Point& p) = 0;
  virtual void remove(const Point& p) = 0;
  virtual bool exist(const Point& p) const = 0;
  virtual bool empty() const = 0;
  virtual bool full() const = 0;
  virtual std::vector<Point> points() const = 0;
};
```

4.2 BddCube

Per prima cosa siamo andati alla ricerca di librerie che implementassero la struttura ROBDD trovando grandi difficoltà. Abbiamo trovato una decina di librerie disponibili in vari linguaggi, alcune tra le più "promettenti" sono:

- ABCD `http://fmv.jku.at/abcd`

- BuDDy `http://sourceforge.net/projects/buddy`
- CMU BDD `http://www-2.cs.cmu.edu/~modelcheck/bdd.html`
- JavaBDD `http://javabdd.sourceforge.net/`
- CUDD `http://vlsi.colorado.edu/~fabio/CUDD/`
- Biddy `http://lms.uni-mb.si/biddy/`

Tra tutte abbiamo scelto di usare BuDDy, valutando la semplicità d'uso e la bontà della documentazione, infatti BuDDy rispetto alle altre è quella con una documentazione accettabile, abbastanza "general purpose", semplice e intuitiva da usare con un'interfaccia C++, anche se implementata in C. Le altre librerie sono quasi tutte documentate in modo pessimo o la documentazione è inesistente, rendendo così il nostro processo di implementazione troppo laborioso, alcune sono state progettate per scopi specifici, quindi poco "general purpose" e altre ancora hanno un modo d'uso non intuitivo.

La libreria BuDDy offre la possibilità di creare oggetti `bdd` su cui compiere operazioni logiche con altri `bdd`. Abbiamo creato dunque la classe **BddCube** wrapper per BuDDy, parametrica rispetto al valore k del dominio D_k^3. Internamente la classe **BddCube** è composta dai seguenti attributi (privati):

- `side` indica il limite del dominio, quindi `side` corrisponde al k in D_k^3;
- `nVar` indica il numero di variabili per dimensione ($log_2(k)$) necessarie per costruire il `bdd` che si vuole rappresentare;
- `root` è l'oggetto `bdd` usato per la rappresentazione dei punti in D_k^3.

Abbiamo quindi implementato i metodi di inserimento, rimozione e verifica di un punto all'interno del nostro D_k^3, metodi di unione, intersezione e differenza tra vari D_k^3, inoltre abbiamo implementato il metodo `nodecount`, che restituisce il numero di nodi usati per rappresentare l'insieme di punti, ed il metodo `dot` che restituisce un grafo in formato dot della struttura interna.

```
class BddCube : public Cube {
public:
  explicit BddCube(int side = 2);
  void insert(const Point& p);
  void remove(const Point& p);
  bool exist(const Point& p) const;
  bool empty() const;
  bool full() const;
  std::vector<Point> points() const;
  int nodecount() const;
  std::string dot() const;
  BddCube& unions(const BddCube& obj);
  BddCube& intersect(const BddCube& obj);
  BddCube& diff(const BddCube& obj);
private:
  int side;
  int nVar;
  bdd root;
  const bdd convertPointToBdd(const Point& p) const;
};
```

La libreria BuDDy, essendo strutturata per un uso in C e offrendo solo un'interfaccia per un uso in C++, necessita di essere *inizializzata*, dunque in fase di istanziazione di un nuovo oggetto **BddCube** si deve inizializzare il package BuDDy settando alcuni parametri, come il numero di nodi iniziali e una dimensione della cache, usata internamente, insieme ad un garbage collector, per ottimizzare le operazioni sui vari `bdd`. Abbiamo cercato di

seguire i suggerimenti riportati nella documentazione di BuDDy e, provando diverse combinazioni dei parametri, abbiamo reso dipendenti rispetto a `side` questi parametri di *inizializzazione* del package BuDDy.

Per rappresentare dei punti tridimensionali a valori interi in un `bdd` si devono scomporre le tre componenti del punto in una serie di variabili booleane. Dato un punto $p = (x, y, z) \in D_k^3$ generiamo un insieme

$$B = \{x_0, \ldots x_{s-1}, y_0, \ldots, y_{s-1}, z_0, \ldots, z_{s-1}\} \subset \mathbb{B}^{3s}$$

$$\text{dove} \quad s = log_2(k) \ \wedge \ \forall w \in \{x, y, z\}, \ (w_0 = MSB(w), \ w_{s-1} = LSB(w))$$

imponendo un ordinamento "$<$" in B

$$x_0 < y_0 < z_0 < x_1 < y_1 < z_1 < \cdots < x_{s-1} < y_{s-1} < z_{s-1}$$

La scelta di questo ordinamento è stata guidata dai risultati di test prodotti per il confronto di diversi ordinamenti. L'ordinamento scelto è risultato essere il più efficente rispetto al numero di nodi generati nel `bdd` interno, confermando i risultati riportati da [Bryant R. E., 1992].

Ora l'insieme B, dotato di uno specifico ordinamento, è la rappresentazione mediante ROBDD del dominio D_k^3. La specificazione del tipo di ordinamento avviene nella fase di *inizializzazione* del package BuDDy, quindi una sola volta. La conversione di un punto $p \in D_k^3$ in un punto $b \in B$ è stata implementata nel metodo `convertPointToBdd`, il quale permette la traduzione di un oggetto `Point` in un oggetto `bdd` corrispondente. L'inserimento (rimozione) di un punto nel nostro `bdd root` viene effettuato mediante l'unione (differenza) di quest'ultimo con il `bdd` restituito dal meto-

do `convertPointToBdd`. Le operazioni di unione, intersezione e differenza tra due D_k^3 rappresentati mediante **BddCube**, sono rispettivamente le operazioni `OR`, `AND` e `DIFF` booleane, implementate nei metodi `operator|`, `operator&`, `operator-`, offerti dalla libreria BuDDy.

4.3 OctreeCube, Node

Nella ricerca di librerie che implementano gli OCTREE, non abbiamo trovato una situazione migliore rispetto agli ROBDD, anzi, la maggior parte delle librerie sono a pagamento, e le poche librerie "free" che si sono trovate sono pessime, sia dal punto di vista della documentazione, sia dal punto di vista strettamente implementativo, inoltre nessuna libreria forniva metodi di intersezione, unione, differenza tra i vari OCTREE. Abbiamo dunque deciso di implementare la struttura dati ex-novo. Abbiamo creato dunque la classe **OctreeCube** parametrica rispetto al valore k del dominio D_k^3. Internamente la classe **OctreeCube** è composta dai seguenti attributi (privati):

- `side` indica il limite del dominio, quindi `side` corrisponde al k in D_k^3;
- `treeHigh` indica l'altezza dell'albero OCTREE ($log_2(k)$);
- `root` è un oggetto `Node` che rappresenta la radice dell'albero OCTREE usato per la rappresentazione di un insieme di punti in D_k^3.

Abbiamo quindi implementato i metodi di inserimento, rimozione e verifica di un punto all'interno del nostro D_k^3, metodi di unione, intersezione e differenza tra vari D_k^3, inoltre abbiamo implementato il metodo `nodecount`, che restituisce il numero di nodi usati per rappresentare l'insieme di punti, ed il metodo `dot` che restituisce un grafo in formato dot della struttura interna.

```
class OctreeCube : public Cube {
public:
  explicit OctreeCube(int side = 2);
  void insert(const Point& p);
  void remove(const Point& p);
  bool exist(const Point& p) const;
  bool empty() const;
  bool full() const;
  std::vector<Point> points() const;
  int nodecount() const;
  std::string dot() const;
  OctreeCube& unions(const OctreeCube& obj);
  OctreeCube& intersect(const OctreeCube& obj);
  OctreeCube& diff(const OctreeCube& obj);
private:
  int side;
  int treeHigh;
  Node root;
};
```

La classe **Node** implementa la struttura di un nodo di un octree, ovvero un nodo ad otto figli. Internamente la classe **Node** è composta dai seguenti attributi (privati):

- `_depth` indica il livello in cui il nodo si trova nell'`OctreeCube`;
- `childrenStatus`, un vettore di booleani, indica lo stato dei nodi figli;
- `childrenPtr`, il vettore di puntatori `Node` ai nodi figli.

Un nodo di un OCTREE può assumere i seguenti tre stati:

- **WHITE:** il nodo non ha figli;
- **BLACK:** il nodo ha otto figli BLACK;

- **GREY:** il nodo ha qualche figlio BLACK oppure qualche figlio WHITE oppure qualche figlio GREY.

Nell'implementazione della classe **Node** si è scelto di rappresentare lo stato di un figlio j con questa corrispondenza:

- **WHITE:** `childrenPtr[j] = NULL and childrenStatus[j] = false`
- **BLACK:** `childrenStatus[j] = true`
- **GREY:** `childrenPtr[j] != NULL and childrenStatus[j] = false`

```
class Node {
public:
  Node(int depth = 0, bool full = false);
  Node*& operator[](int i);
  const Node* operator[](int i) const;
  bool empty() const;
  bool full() const;
  int depth() const;
  void depth(int depth);
  bool status(int i) const;
  void status(int i, bool value);
  void unions(const Node& obj, const int& treeHigh);
  void intersect(const Node& obj, const int& treeHigh);
  void diff(const Node& obj, const int& treeHigh);
private:
  int _depth;
  std::bitset<NCHILD> childrenStatus;
  std::vector<Node*> childrenPtr;
};
```

Abbiamo scelto di implementare le operazioni insiemistiche di **Octree-Cube** passandole alla classe **Node** in modo da tenere separata le due strut-

ture, ovvero la classe **OctreeCube** non necessita di conoscere come è strutturata la classe **Node**, infatti le operazioni insiemistiche tra due **OctreeCube** non sono altro che le stesse operazioni sui nodi radice dei vari oggetti. Cerchiamo di presentare le operazioni insiemistiche mediante pseudocodice:

unione In questo caso l'operazione `OR` effettuata tra due foglie, che possono assumere solo lo stato `BLACK` o `WHITE`, viene riassunta da:

- `BLACK OR (ANYTHING) = BLACK`
- `WHITE OR (ANYTHING) = (ANYTHING)`

```
unione(Node n1, Node n2)
  if (n1,n2 foglie) then
    n1 = n1 OR n2; return;
  for i := 0 to 8
    if (n1.child[i] is BLACK) then
      continue;
    if (n1.child[i] is GRAY) then
      if (n2.child[i] is WHITE) then
        continue;
      if (n2.child[i] is BLACK) then
        n1.child[i] = BLACK; continue;
      unione(n1.child[i], n2.child[i]);
    n1.child[i] = n2.child[i];
  end for; return;
```

intersezione In questo caso l'operazione `AND` effettuata tra due foglie, che possono assumere solo lo stato `BLACK` o `WHITE`, viene riassunta da:

- `WHITE AND (ANYTHING) = WHITE`
- `BLACK AND (ANYTHING) = (ANYTHING)`

```
intersezione(Node n1, Node n2)
  if (n1,n2 foglie) then
    n1 = n1 AND n2; return;
  for i := 0 to 8
    if (n1.child[i] is BLACK) then
      n1.child[i] = n2.child[i]; continue;
    if (n1.child[i] is GRAY) then
      if (n2.child[i] is WHITE) then
        n1.child[i] = WHITE; continue;
      if (n2.child[i] is BLACK) then
        continue;
      intersezione(n1.child[i], n2.child[i]);
  end for;
return;
```

differenza In questo caso l'operazione `DIFF` effettuata tra due foglie, che possono assumere solo lo stato `BLACK` o `WHITE`, viene riassunta da:

- `WHITE DIFF (ANYTHING) = WHITE`
- `BLACK DIFF WHITE = BLACK`
- `(ANYTHING) DIFF BLACK = WHITE`

```
differenza(Node n1, Node n2)
  if (n1,n2 foglie) then
    n1 = n1 DIFF n2; return;
  for i := 0 to 8
    if (n2.child[i] is WHITE) then
      continue;
    if (n2.child[i] is GRAY) then
      if (n1.child[i] is WHITE) then
        continue;
      if (n1.child[i] is BLACK) then
        n1.child[i] = complementare(n2.child[i]); continue;
      differenza(n1.child[i], n2.child[i]); continue;
    if (n1.child[i] is WHITE) then
      continue;
    n1.child[i] = WHITE;
  end for;
return;
```

Capitolo 5

Valutazioni sperimentali

Abbiamo predisposto una serie di test generando un pool di insiemi di diversa densità di punti casuali con distribuzione uniforme sul dominio. Ogni test viene ripetuto per diversi valori di k del dominio D_k^3. I vari insiemi di punti sono stati rappresentati con le due strutture dati **BddCube** e **OctreeCube** ottenendo una campionatura sia dei tempi delle operazioni fra domini, sia sullo spazio occupato in memoria. I test sono stati eseguiti su una macchina x86 con processore INTEL® Core™ 2 Duo 2.66GHz, 2 Gb di RAM. Il sistema operativo installato sulla macchina è Ubuntu Linux 8.10, con kernel 2.6.27, gcc 4.3.2.

5.1 Test sui tempi

Per ogni valore $k \in \{16, 32, 64\}$ e per diversi fattori $(1\%, 10\%, 50\%, 90\%, 99\%)$ di densità d di punti negli insiemi sul totale dei punti rappresentabili nel dominio, pari a k^3, abbiamo generato 250 insiemi di punti, di valori casuali con distribuzione uniforme nel dominio D_k^3. Dopo aver rappresentato ogni

insieme di punti in un rispettivo oggetto **BddCube** e **OctreeCube**, abbiamo calcolato i valori medi su 250 tempi di esecuzione (in secondi) di operazioni unione, intersezione e differenza ottenendo i seguenti risutati:

(a) **BddCube**

	16	**32**	**64**
1%	0.048	0.412	7.892
10%	0.396	6.98	120.276
50%	2.216	43.736	692.12
90%	2.796	56.532	854.128
99%	2.692	31.188	616.184

(b) **OctreeCube**

16	**32**	**64**	
0.028	0.26	2.12	1%
0.148	1.3	22.224	10%
0.332	2.78	260.372	50%
0.376	3.22	79.844	90%
0.364	2.712	23.136	99%

Tabella 5.1: Medie dei tempi totali, in secondi, di ogni test.

(a) **BddCube**

	16	**32**	**64**
1%	0.00	0.03	0.34
10%	0.02	0.23	2.68
50%	0.05	0.63	6.78
90%	0.02	0.28	2.98
99%	0.01	0.04	0.47

(b) **OctreeCube**

16	**32**	**64**	
0.01	0.07	0.56	1%
0.04	0.31	6.30	10%
0.05	0.45	74.38	50%
0.04	0.31	13.11	90%
0.02	0.09	0.63	99%

Tabella 5.2: Medie dei tempi, in secondi, di ogni singola operazione insiemistica.

(a) **BddCube**

	16	**32**	**64**
1%	0.04	0.34	6.88
10%	0.32	6.28	112.25
50%	2.06	41.85	671.78
90%	2.73	55.69	845.20
99%	2.66	31.08	614.76

(b) **OctreeCube**

16	**32**	**64**	
0.00	0.05	0.44	1%
0.04	0.38	3.31	10%
0.18	1.42	37.24	50%
0.26	2.28	40.52	90%
0.30	2.46	21.23	99%

Tabella 5.3: Medie dei tempi, in secondi, totali delle operazioni di inserimento.

5.2 Test sullo spazio occupato

Per stimare lo spazio occupato dalle due rappresentazioni, abbiamo calcolato i valori medi di 500 oggetti **BddCube** e **OctreeCube** considerando il numero dei nodi necessari per rappresentare l'insieme di punti, di valori casuali con distribuzione uniforme nel dominio D_k^3. Si sono effettuati test per ogni valore $k \in \{16, 32, 64, 128, 256, 512\}$ e per diversi fattori $(1\%, 10\%, 50\%, 90\%, 99\%)$ di densità d di punti negli insiemi sul totale dei punti rappresentabili nel dominio, pari a k^3, ottenendo questi risultati:

		16	**32**	**64**	**128**	**256**	**512**
1%	punti	40	327	2621	20971	167772	1342177
	bdd	130	674	3747	23190	146287	975484
	octree	75	625	5048	40442	323097	2586193
10%	punti	400	3270	26210	209710	1677720	13421770
	bdd	425	2452	15273	98963	663056	4570733
	octree	360	2936	23249	186358	1490229	11925975
50%	punti	2000	16350	131050	1048550	8388600	67108850
	bdd	728	4323	31089	187759	1114066	8451989
	octree	580	4639	37151	297235	2377739	19023241
90%	punti	3600	29430	235890	1887390	15099480	120795930
	bdd	468	2484	15280	98980	663060	4570740
	octree	406	2969	23275	186384	1490238	11925986
99%	punti	3960	32373	259479	2076129	16609428	132875523
	bdd	229	756	3788	23226	146297	975497
	octree	179	720	5118	40515	323119	2586230

Tabella 5.4: Numero dei nodi.

5.3 Valutazioni

5.3.1 Tempi di esecuzione

Prendendo in considerazione il caso più significativo dai dati riportati in tabella 5.1 in cui $k = 64$, si può notare che la struttura dati OCTREE, rappresentata da **OctreeCube**, sia significativamente più performante rispetto alla struttura dati ROBDD, rappresentata da **BddCube**, come illustrato nel grafico 5.1.

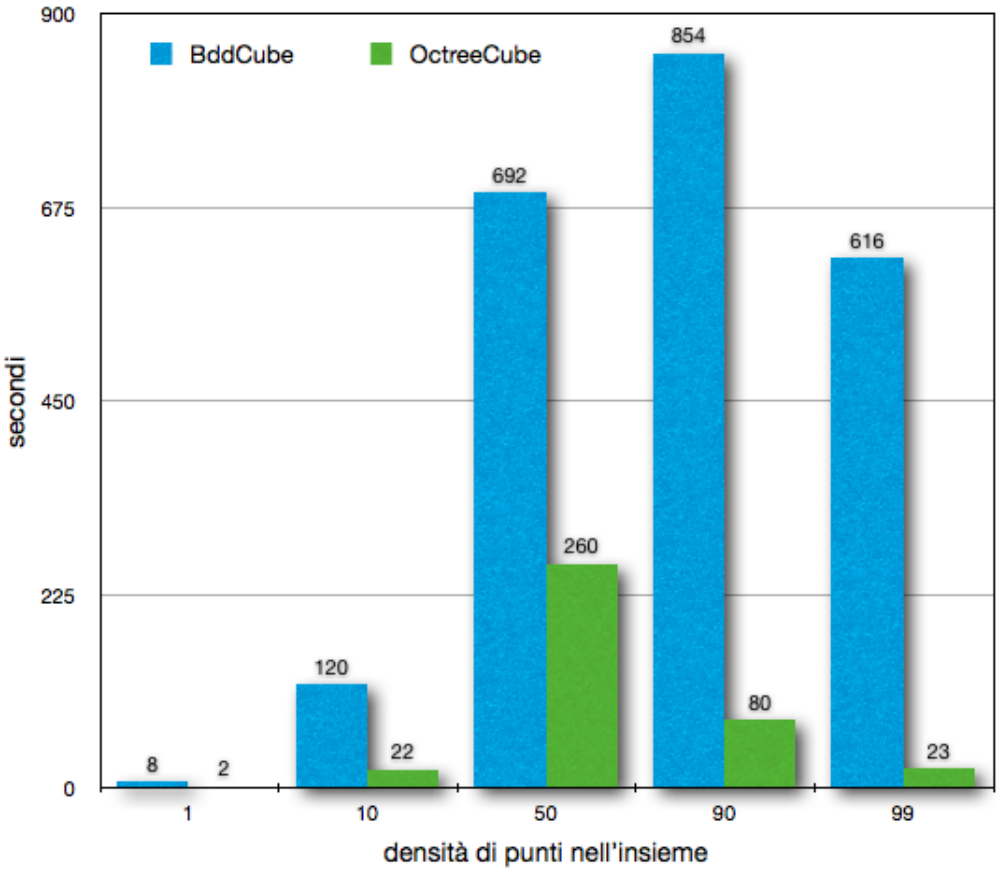

Figura 5.1: Tempi totali, in secondi, dei test sulle diverse densità con $k = 64$.

Analizzando più approfonditamente i dati parziali di ogni singola operazione insiemistica in tabella 5.2 e di inserimento in tabella 5.3, notiamo che la struttura dati **BddCube**, rispetto alla struttura dati **OctreeCube**, impiega la maggior parte del tempo totale in operazioni di inserimento, come riportato dal grafico 5.2. Questo insolito risultato è giustificabile dal fatto che la struttura dati **BddCube** effettua un'operazione di unione ad ogni operazione di inserimento, di conseguenza un'operazione interna di *rilassamento* dei nodi.

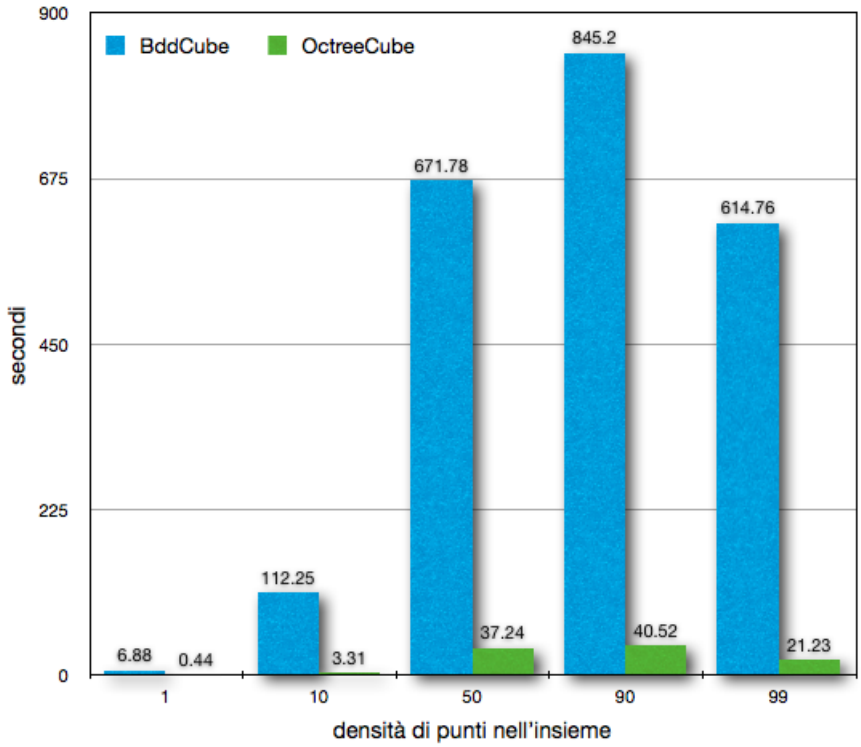

Figura 5.2: Tempi totali, in secondi, delle operazioni di inserimento nei test sulle diverse densità con $k = 64$.

Al contrario, come illustrato dal grafico 5.3, la struttura dati **OctreeCube** risulta essere significativamente meno performante rispetto alla struttura dati **BddCube** nelle operazioni insiemistiche.

Si può notare inoltre come le due strutture dati si comportano in funzione della densità di punti nel dominio che devono rappresentare. Questi risultati ci danno la possibilità di affermare che le due strutture dati trovano il loro caso pessimo quando devono operare con oggetti che occupano la metà del domino, ovvero quando si ha una densità di punti nel dominio prossima al 50%; viceversa si può affermare che le due strutture dati esprimono la loro massima efficienza quando si trovano ad operare su dati sparsi o densi, ovvero quando si ha una densità di punti nel dominio prossima all'10% o al 90%.

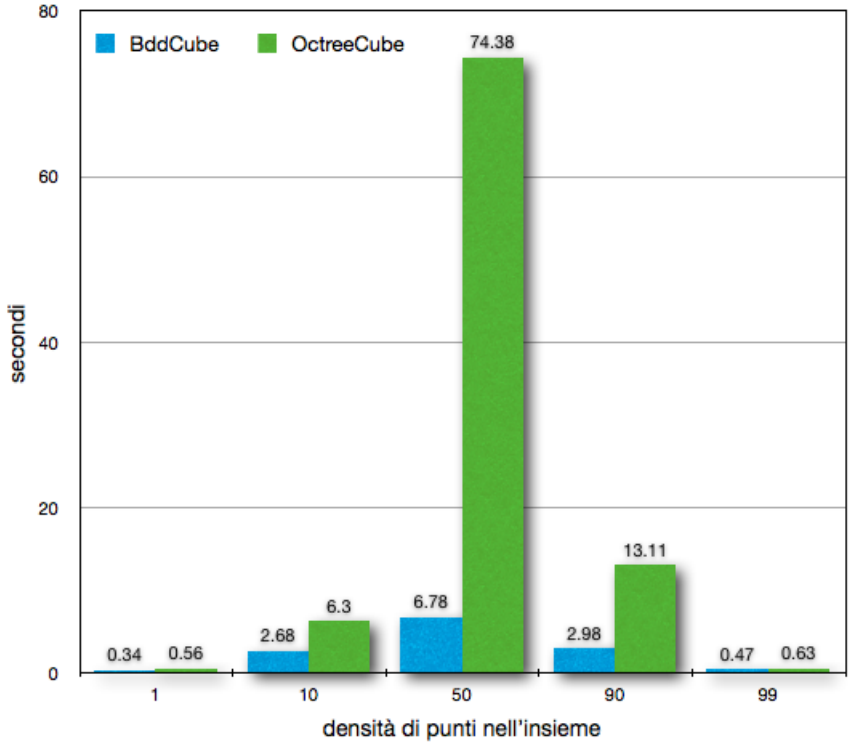

Figura 5.3: Tempi, in secondi, di una singola operazione insiemistica nei test sulle diverse densità con $k = 64$.

5.3.2 Spazio occupato

I dati relativi alla stima dello spazio occupato dalle due strutture dati riportati in tabella 5.4, possono essere riassunti considerando i casi in cui $k = 64$ e $k = 128$. Come evidenziato dai grafici 5.4 e 5.5, si può affermare che al crescere del numero dei punti da rappresentare, quindi al crescere della densità dei punti nel dominio, le due strutture dati risultano essere molto efficienti in termini di conservazione dello spazio di memorizzazione, in accordo con le precedenti valutazioni sui di tempi di esecuzione.

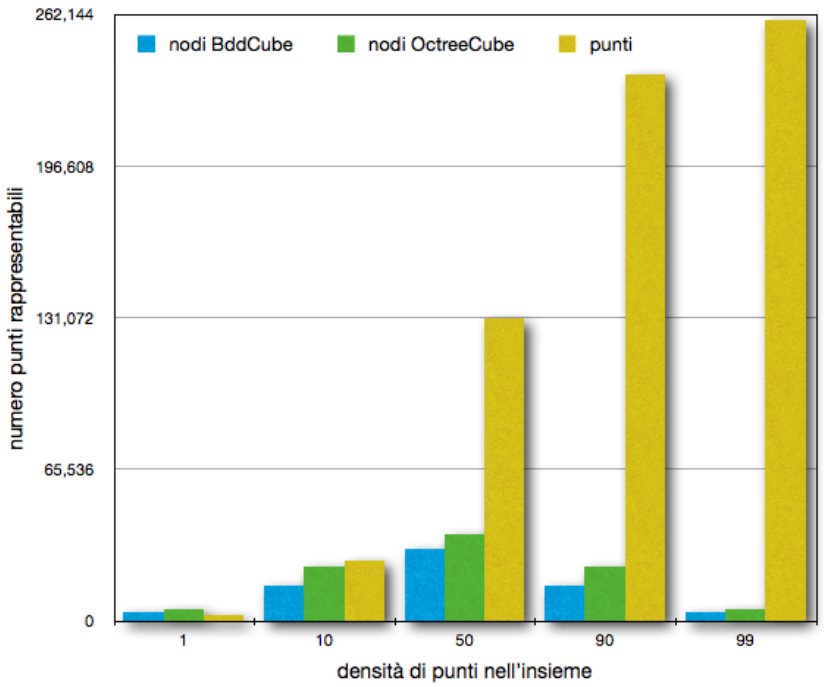

Figura 5.4: Spazio occupato con $k = 64$.

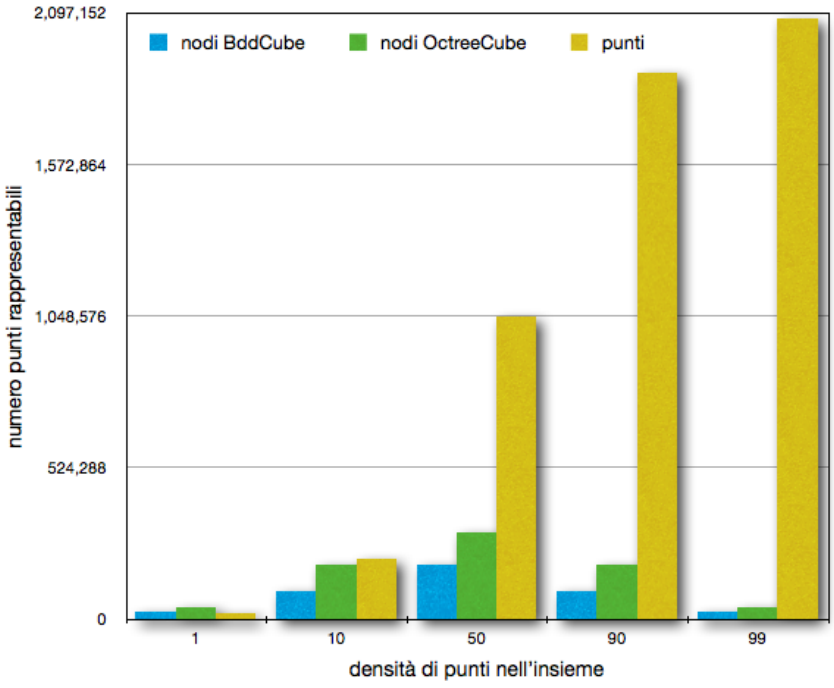

Figura 5.5: Spazio occupato con $k = 128$.

Capitolo 6

Conclusioni

Basandoci sulle valutazioni esposte precedentemente possiamo riassumere pregi e difetti delle due strutture dati confrontate in riferimento alle varie situazioni a cui sono state sottoposte.

In primo luogo si può sottolineare che, sia ROBDD, sia OCTREE, offrono una valida alternativa alle strutture dati "classiche" in termini di efficienza nella memorizzazione dello spazio utilizzato per rappresentare insiemi di punti n-dimensionali; infatti sono strutture dati *adattive*, ovvero, al contrario di un array n-dimensionale, lo spazio di memorizzazione di cui hanno bisogno è solo quello realmente utilizzato, inoltre, superata una certa soglia "critica", quando la densità di punti in un dominio è intorno al 50%, lo spazio occupato diminuisce. Questo pregio, in termini di risparmio dello spazio di memorizzazione, condiziona notevolmente il comportamento delle operazioni insiemistiche sulle strutture dati, il quale è risultato più che soddisfacente per entrambe in ogni situazione, seppur con un discreto vantaggio di ROBDD rispetto ad OCTREE.

Un ***piccolissimo prezzo*** da pagare per avere questa considerevole ef-

ficienza è dovuto alla lieve differenza tra i tempi di accesso ai singoli punti rappresentati nel dominio rispetto ad una rappresentazione mediante un array n-dimensionale, infatti, in quest'ultima struttura dati, il tempo di accesso ai singoli punti si mantiene costante nell'ordine di $O(1)$, mentre sia per ROBDD, sia per OCTREE, il tempo di accesso rimane nell'ordine di $O(log(n))$.

A questo punto una domanda sorge spontanea:

Quale delle due strutture dati è la migliore?

$\rightsquigarrow$ **Dipende dalla natura del problema a cui sono sottoposte.**

Se per esempio devo rappresentare una struttura tridimensionale che viene modificata spesso mediante unioni, intersezioni o differenze insiemistiche, conviene rappresentare la struttura usando ROBDD rispetto ad OCTREE.

Se invece devo rappresentare un insieme di tuple in un database, una relazione, in cui si inseriscono spesso nuovi elementi, la quale deve soddisfare query di tipo "esiste l'elemento (X,Y,...)", conviene rappresentare la relazione usando OCTREE rispetto a ROBDD.

Si possono sviluppare ulteriori miglioramenti soprattutto nella ricerca di metodi efficienti per l'inserimento di singoli punti nella struttura dati ROBDD, la quale in questo tipo di operazione ha trovato il proprio punto debole. La causa di ciò può essere attribuita alla libreria BuDDy, di cui non si conosce perfettamente l'implementazione, oppure al modo in cui l'inserimento è stato pensato. Un modo per rimediare a questo inconveniente può essere quello di studiare meglio le situazioni critiche dell'inserimento e inserire i punti non più singolarmente, ma cercare di inserire gruppi di punti,

in modo da ammortizzare il costo di inserimento sfruttando meglio l'operazione di unione degli ROBDD. Un'ulteriore, ma più drastica, soluzione può essere quella di implementare ex-novo una struttura dati ROBDD.

Bibliografia

[Akers S. B., 1978] Akers S. B. (1978). Binary decision diagrams. *IEEE Transactions on Computers*, C27(6):509–516.

[Bryant R. E., 1986] Bryant R. E. (1986). Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C35(6):677–691.

[Bryant R. E., 1992] Bryant R. E. (1992). Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3):3–7.

[Cormen T. H. et al., 2001] Cormen T. H., Leiserson C. E., Rivest R. L., and Stein C. (2001). *Introduction to Algorithms*. MIT Press and McGraw-Hill, 2nd edition.

[Dayer C.R., 1982] Dayer C.R. (1982). The space efficiency of quadtrees. *Computer Graphics and Image Processing*, 19(4):335–348.

[Fortune S. et al., 1978] Fortune S., Hopcroft J., and Schmidt E. M. (1978). The complexity of equivalence and containment for free single variable program schemes. *Automata, Languages and Programming, Lecture Notes in Computer Science*, 62:227–240.

[Garcia Molina H. et al., 2002] Garcia Molina H., Ullman J. D., and Widom J. (2002). *Database Systems - The Complete Book*. Prentice Hall, international edition.

[Hunter G.M. and Steiglitz K., 1979] Hunter G.M. and Steiglitz K. (1979). Operations on image using quad trees. *IEEE Transactions on Pattern Analisys and Machine Intelligence*, 1(2):145–153.

[Lee C. Y., 1959] Lee C. Y. (1959). Representation of switching circuits by binary-decision programs. *Bell System Technical Journal*, 38:985–999.

[Meagher D., 1980] Meagher D. (1980). Octree encoding: a new tecnique for the representation, manipulation, and display of arbitrary 3-d objects by computer. *Electrical and Systems Engineering*, A.1.2, D.1.5:80–110.

[Samet H., 1988] Samet H. (1988). An overview of quadtrees, octrees, and related hierarchical data structures. *Computer and System Sciences*, 40:51–68.

[Samet H., 1999] Samet H. (1999). *Handbook of Algorithms and Theory of Computation*, chapter 18. CRC Press. Multidimensional data structures `http://www.cs.umd.edu/~hjs/pubs/crchandbookchapter.pdf`.

[Samet H., 2002] Samet H. (2002). *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*. Addison-Wesley.

[Samet H., 2005] Samet H. (2005). *Handbook of Algorithms and Theory of Computation*, chapter 16. CRC Press. Multidimensional spatial data structures `http://www.cs.umd.edu/~hjs/pubs/SameCRC05.pdf`.

[Wikipedia, 2009a] Wikipedia (2009a). Wikipedia. The free enciclopedia. `http://en.wikipedia.org/wiki/Discretization`.

[Wikipedia, 2009b] Wikipedia (2009b). Wikipedia. The free enciclopedia. `http://en.wikipedia.org/wiki/Quantization`.