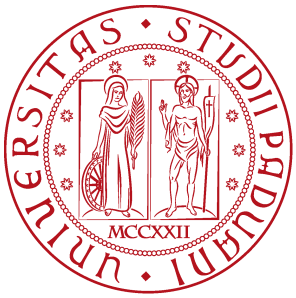
Analisi ed implementazione di Cover Tree per la ricerca di nearest neighbour in spazi multidimensionali



Thomas Rossi Mel

Dipartimento di ingegneria dell'informazione

Corso di laurea in Ingegneria Informatica

Università degli studi di Padova

Data di laurea: 19/09/2022

Relatore Ch.mo Prof. Geppino Pucci

Anno accademico 2021 - 2022

Indice

Capitolo 1

Introduzione

Negli ultimi anni, con l'aumento della quantità di dati presenti nella rete, sono nati sempre più campi dell'informatica volti all'analisi e all'utilizzo delle informazioni per la risoluzione di problemi ad alta complessità.
Uno dei principali nuovi usi delle informazioni riguarda il campo del machine learning, che ha portato ad un cambiamento radicale della vita di tutti i giorni introducendo dispositivi e tecnologie sofisticate come gli assistenti vocali o i recommendation systems, basati su dataset in continua espansione composti da milioni di dati.
Con il passare degli anni sono migliorate anche le prestazioni degli elaboratori ma, nonostante ciò, la gestione e l'utilizzo di così tante informazioni risulta difficile, e rimane fondamentale studiare tecniche e algoritmi che permettano il miglioramento delle prestazioni.
In questo documento verrà presentata una struttura dati capace di risolvere uno dei più importanti problemi degli ultimi anni con applicazioni nei campi del computer vision, database, robotica e molto altro: il nearest neighbor search (NNS).

Nel prossimo capitolo viene introdotto il problema, insieme alle varianti ed alle precedenti ricerche svolte per la sua risoluzione. Si prosegue poi con la definizione del Cover Tree, una struttura dati gerarchica ad albero che permette di effettuare query, inserimenti ed eliminazioni anche con diverse dimensioni all'interno di uno spazio multidimensionale, rendendo efficiente la risoluzione dell'NNS.
Il capitolo 3 e 4 sono interamente dedicati agli algoritmi, spiegati attraverso pseudocodici e analizzati sia nella correttezza che nella complessità computazionale.
Infine il capitolo 5 presenta tre dei progetti più popolari presenti in rete che implementano la struttura dati, con una analisi sia del codice che delle prestazioni. Vengono poi suggerite delle modifiche in grado di migliorare i tempi di esecuzione delle principali operazioni.

Capitolo 2

Definizioni preliminari

Questo primo capitolo ha come scopo la formulazione del problema della nearest-neighbour search e la definizione rigorosa del Cover Tree.
In particolare ci si sofferma sull'importanza della dimensionalità intrinseca dei dati, che rappresenta un fattore di estrema rilevanza per le prestazioni della struttura dati.
Infine, l'ultimo paragrafo, introduce un concetto fondamentale per i Cover Tree, ossia la rappresentazione implicita ed esplicita dei nodi, in grado di migliorare sia l'efficienza delle operazioni che la quantità di memoria occupata.

2.1 Il problema

La *Nearest Neighbor Search*, o NNS, è uno tra i più importanti problemi di ottimizzazione degli ultimi anni, con numerose applicazioni in machine learning, computer vision, database, reti peer-to-peer e biologia computazionale. Si può definire il problema nel seguente modo:

Dato un insieme S di n punti in uno spazio metrico (X,d) *e dato un punto* $p \in X$*, si trovi* $q \in S$ *tale da minimizzare* $d(p,q)$

In altre parole si richiede di trovare un punto $q \in S$ che sia il più vicino possibile a p secondo una certa funzione di distanza d, definita in uno spazio metrico con punti di dominio X. La presenza di centinaia di migliaia di punti all'interno di S può però portare a una notevole lentezza nella risoluzione del problema, che necessita una pre-elaborazione dei dati per essere eseguita efficientemente. La maggior parte della ricerca si è focalizzata sull'analisi di spazi metrici euclidei con dimensioni ridotte (1, 2 o 3) che, pur essendo una categoria frequente, rappresenta una forte limitazione in molti ambiti applicativi. Inoltre, soprattutto nel campo del machine learning, il concetto di *dimensionalità intrinseca* (vedi sezione 2.2) può essere anche più importante rispetto al numero di dimensioni effettive dei punti del dataset. La struttura dati che verrà presentata è, difatti, un miglioramento delle Navigating Net [1] e dei risultati presentati da Karger e Ruhl [2], lavori che considerano quest'ultimo concetto e che analizzano spazi multidimensionali non necessariamente euclidei.

Esistono inoltre delle varianti della NNS, anch'esse molto utilizzate. Un esempio è la *k-nearest-*

neighbors (K-NN) (spesso chiamata *batch query*), dove si cercano i k elementi più vicini ad un punto di input, in S. Una seconda variante è anche la *ricerca approssimata del nearest neighbour*, che verrà definita rigorosamente nei prossimi capitoli.

2.2 Dimensionalità intrinseca

Uno dei concetti più importanti, che verrà ulteriormente approfondito nel capitolo successivo, è quello della dimensionalità intrinseca. La definizione più semplice e comune di dimensionalità intrinseca consiste nel numero minimo di feature in grado di rappresentare correttamente il dataset. Per chiarire il significato, consideriamo il semplice esempio di tutti i punti a tre dimensioni x, y, z tali che $x+y+z=100$.
In figura 2.1 si ha una rappresentazione grafica delle soluzioni.

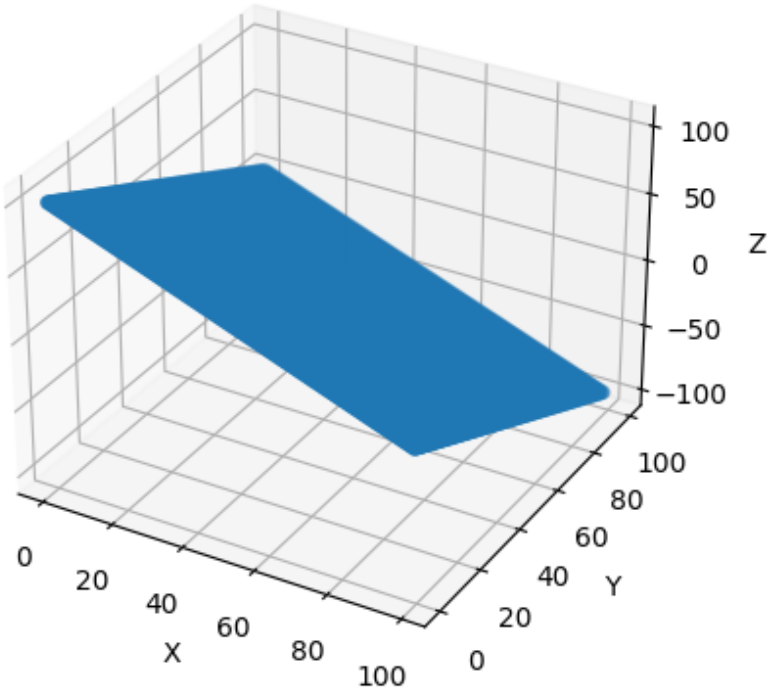


Figura 2.1. Luogo dei punti tali che $x+y+z=100$

Nonostante lo spazio sia tridimensionale, sono necessarie solo due dimensioni per rappresentarlo, in quanto la terza è univocamente determinata dalle altre due (ad esempio $z=100-x-y$); la dimensionalità intrinseca di quel luogo di punti è quindi pari a 2.
In generale, però, è difficile trovare dei dataset aventi un parametro completamente dipendente dagli altri, ma è probabile che ne sia fortemente legato; proprio per questa ragione è di estrema utilità

l'introduzione di una metrica che, non solo è in stretta relazione con la dimensionalità intrinseca dei dati, ma che è anche in grado di quantificare la dipendenza tra le feature.

Sulla base di questa nozione nascono quindi i concetti di *expansion constant* e *doubling constant*, in stretta relazione con la dimensione intrinseca di un set S di punti, calcolabili attraverso operazioni matematiche elementari.

2.2.1 Expansion constant

Prima d'introdurre la costante di espansione, diamo un'importante definizione che verrà usata anche nei prossimi capitoli. Dato un set S, indichiamo con $B_S(p,r)$ la palla chiusa di raggio r e centro p:

$B_S(p,r) = \{q \in S : d(p,q) \leq r\}$ [1]

L'*expansion constant* di un set $S \subset X$, introdotta da Karger e Ruhl [2] per analizzare le prestazioni della struttura dati da loro proposta, è definita come il minimo valore $c \geq 2$ tale che:

$|B_S(p,2r)| \leq c|B_S(p,r)|$

Per uno spazio metrico a d dimensioni, con punti distribuiti uniformemente nel piano, si ottiene una costante esponenziale in d (circa 2^d). Ciò è anche evidente considerando il rapporto dei volumi di una palla di raggio $2R$ e di una seconda palla di raggio R, che risulta esattamente pari a 2^d. Sembra quindi ragionevole dare la seguente definizione di *expansion dimension*: $dim_{KR}(S) = \log c$.
Nonostante sia evidente una correlazione con la dimensionalità intrinseca, ci sono alcuni casi che portano a una costante arbitrariamente grande anche per spazi con poche dimensioni. Consideriamo, ad esempio, lo spazio metrico euclideo monodimensionale, ed il set $S = \{2^i : i = 0,1,2,...,n\}$. Vogliamo quindi trovare c minore possibile tale che, per ogni palla di raggio e centro generico, sia soddisfatta la relazione sopra citata. Il valore minimo è $c = n$, che si ottiene con il caso peggiore possibile: $|B_S(2^n, 2^{n-1} - 0.5)| = 1$ e $|B_S(2^n, 2^n - 1)| = n$.
Nonostante ciò, per un dataset generico con molti parametri e senza casi "anomali", la expansion constant rappresenta un ottimo compromesso in grado di permettere una analisi computazionale semplice, mantenendo una correlazione con la dimensionalità intrinseca dei dati.

[1] Alcune volte verrà utilizzata la scrittura $B(p,r)$, se il riferimento a S è chiaro dal contesto.

2.2.2 Doubling constant

La doubling constant è stata utilizzata per l'analisi delle Navigating Net [1], e rappresenta una soluzione più "robusta" per l'analisi di strutture dati simili.
La definizione è la seguente:

La doubling constant è il minimo valore di c tale che ogni palla in X può essere coperta da al più c palle in X di metà raggio.

Anche in questo caso si può definire la doubling dimension come $dim_{KL}(S) = \log c$. Rispetto alla expansion constant, non si possono ottenere valori arbitrariamente grandi della costante per una dimensione d fissata. Come si nota dalla figura 2.2, per il caso bidimensionale saranno sempre necessarie al massimo 7 palle di raggio $r/2$ per coprirne una di raggio r.
Nonostante questo vantaggio, per i successivi capitoli verrà presa in considerazione solo la expansion constant, in quanto l'analisi ha portato a risultati migliori, soprattutto per la ricerca esatta del nearest neighbor.

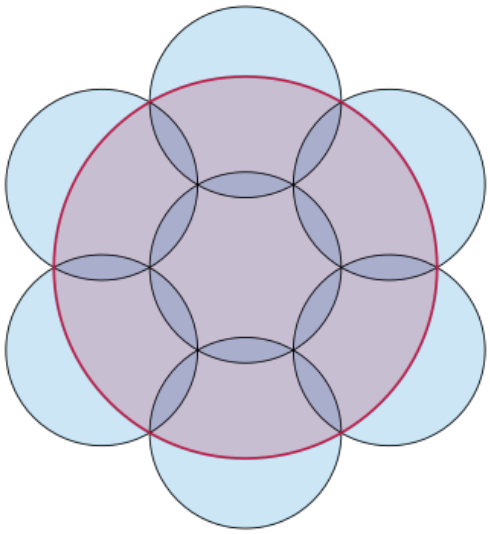


Figura 2.2. Sette dischi di raggio $r/2$ possono coprire qualsiasi disco di raggio r

2.3 Definizione di Cover Tree

La struttura dati presentata da Beygelzimer, Kakade e Langford [3] prende il nome di *Cover Tree*, e permette di eseguire operazioni di inserimento ed eliminazione di punti in uno spazio multidimensionale, e supportando efficientemente la NNS.

Un *Cover Tree* T costruito su un dataset S è un albero a livelli numerato in ordine decrescente verso le foglie, che deve soddisfare tre invarianti. Indichiamo con C_i l'insieme dei nodi al livello i:

(1) (*Nesting*) $C_i \subset C_{i-1}$.
(2) (*Covering Tree*) Per ogni punto $p \in C_{i-1}$, esiste almeno un $q \in C_i$ tale che $d(p,q) \leq 2^i$, ed esattamente un nodo q tra questi è padre di p.
(3) (*Separation*) Per ogni $p,q \in C_i$, $d(p,q) > 2^i$.

Consideriamo come esempio uno spazio monodimensionale euclideo e cinque punti di coordinate $p_1 = -4$, $p_2 = 0$, $p_3 = 2.5$, $p_4 = 4$, $p_5 = 5.5$. Un esempio di albero che soddisfa ognuna delle tre invarianti viene mostrato in figura 2.3 dove il primo livello, contenente la sola radice, equivale ad $i = 2$ e l'ultimo ad $i = 0$. Come si può notare, la condizione (1) viene rispettata, siccome ogni nuovo punto è presente anche ai livelli successivi. Inoltre ogni nodo dell'albero al livello i-esimo (a eccezione della radice) ha un padre con distanza minore o uguale a 2^{i+1}, rendendo valida la (2). Infine, la distanza tra due qualsiasi nodi del livello i è maggiore di 2^i, il che soddisfa anche la (3). Nonostante le invarianti non escludano esplicitamente che un nodo riferito a un punto $p \in S$ del livello *i*-esimo non abbia come figlio il punto p stesso, ciò è necessario affinché le invarianti restino soddisfatte. Consideriamo infatti un punto $p' \in S$ che per assurdo è padre di p al livello $i-esimo$; siccome p compare sia in C_i che C_{i-1} deve essere sia valida l'invariante di copertura tra p e p' ossia $d(p,p') \leq 2^i$, che l'invariante di separazione sempre al livello $i-esimo$ tra p e p', ossia $d(p,p') > 2^i$, che risulta impossibile.

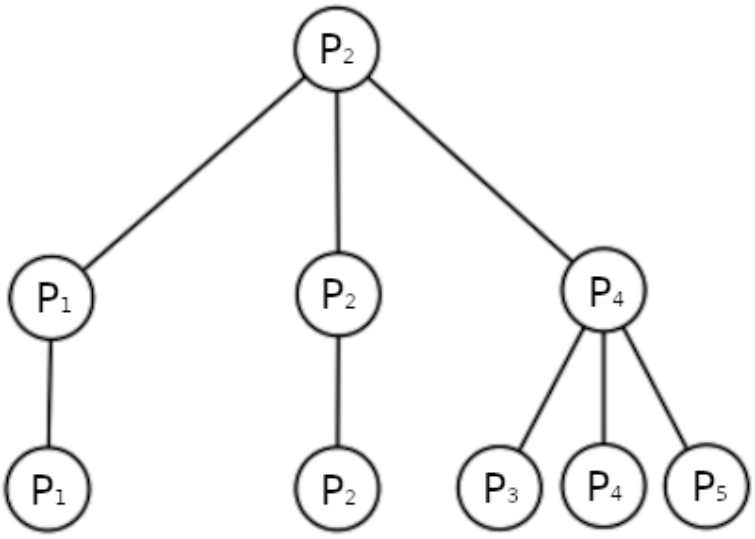


Figura 2.3. Esempio di grafo ad albero che soddisfa le tre invarianti.

Si noti come l'ultima invariante renda necessaria la creazione di nuovi livelli. Se non ci fosse, infatti, ogni nuovo elemento potrebbe essere inserito come figlio della radice, rendendo inefficienti le operazioni principali sulla struttura dati. L'invariante di copertura impone invece che i figli di un nodo non abbiano distanza maggiore di 2^i dal padre. Anche questa invariante contribuisce notevolmente ad incrementare le prestazioni delle varie operazioni, dato che la discesa dell'albero può essere effettuata selezionando solo una parte dei nodi.

Si noti inoltre che, per l'analisi teorica della struttura dati, il numero di livelli è infinito ($-\infty < i < +\infty$) e, di conseguenza, la rappresentazione fisica necessita di una restrizione del range in un intervallo che risulta sufficientemente ampio per il problema da risolvere ($0 \leq i \leq 2$ nel caso precedente).

2.3.1 Rappresentazione implicita ed esplicita

Un'importante distinzione dev'essere fatta tra rappresentazione implicita ed esplicita della struttura dati; nonostante la condizione di nidificazione debba essere vera, infatti, non è necessario memorizzare la maggior parte dei nodi del Cover Tree, migliorando quindi le prestazioni degli algoritmi e il consumo della memoria.
In particolare la rappresentazione esplicita contiene tutti i nodi che hanno almeno un altro figlio oltre alla replica del nodo stesso, oppure un padre diverso.
In questo modo, tutte le "catene" di nodi uguali non vengono salvate nella memoria, in quanto ridondanti e non essenziali; basterà infatti salvare le informazioni relative alle posizioni nell'albero in cui ogni nodo risulta rappresentato esplicitamente.
Di seguito vengono riportate le tabelle che riassumono le complessità computazionali e lo spazio occupato dalla struttura dati con questo approccio, in paragone con Nav. Net [1] e KR02 [2].

	Cover Tree	**Nav. Net**	**KR02**
Spazio occupato	$O(n)$	$c^{O(1)}n$	$c^{O(1)}n\ln n$
Tempo di costruzione	$O(c^6 n\ln n)$	$c^{O(1)}n\ln n$	$c^{O(1)}n\ln n$
Inserimento/Rimozione	$O(c^6\ln n)$	$c^{O(1)}\ln n$	$c^{O(1)}\ln n$
Query	$O(c^{12}\ln n)$	$c^{O(1)}\ln n$	$c^{O(1)}\ln n$
Batch Query	$O(c^{16}n)$	$c^{O(1)}n\ln n$	$c^{O(1)}n\ln n$

Come si può notare, il vantaggio principale rispetto alle altre due soluzioni riguarda lo spazio occupato, che risulta lineare indipendentemente dalla costante di espansione, come dimostrato dal seguente teorema:

Teorema 1. *Lo spazio occupato da un Cover Tree è* $O(n)$.

Dimostrazione. Per iniziare, dimostriamo che ogni nodo p ha al più un padre diverso da se stesso nell'albero.
Supponiamo per assurdo che esistano due nodi q e q' padri di p che, per l'invariante (2), devono esserlo in due livelli diversi. Se q' è padre di p al livello più basso $i-1$, allora p deve comparire anche ai livelli superiori per poter essere figlio di q, incluso lo stesso livello di q', i. Ciò non è però possibile in quanto non verrebbe soddisfatta l'invariante (3) tra il nodo p e q'.
Infine, siccome ogni volta che un nodo viene salvato in memoria esplicitamente questo ha un figlio oppure un padre diverso da se stesso, l'aggiunta di un nuovo elemento nel Cover Tree porta a un aumento costante della memoria (viene infatti reso esplicito e aggiunto, se già non lo è, il solo nodo padre), rendendo lo spazio occupato $O(n)$. □

Capitolo 3

Operazioni sul Cover Tree

In questo capitolo vengono presentate le principali operazioni che possono essere eseguite sui Cover Tree, come la ricerca del nearest neighbour, l'inserimento e la rimozione di un punto.
Ciascun paragrafo introduce un nuovo algoritmo, spiegando passo per passo il suo funzionamento e dimostrandone matematicamente la sua correttezza; vengono inoltre presentate delle soluzioni per le varianti del problema originale, come l'approssimazione del nearest neighbour o la ricerca dei k-nearest neighbours.

3.1 Ricerca del nearest neighbour

Il primo algoritmo che vedremo, nonché il più importante, consiste nella ricerca del nearest neighbour di un punto p in un cover tree T.
Per iniziare indichiamo con Q_i un sottoinsieme dei punti contenuti in C_i, opportunamente selezionati per lo scopo dell'algoritmo. Definiamo inoltre con $d(p,Q) := min_{q \in Q} d(p,q)$ la distanza tra p e il punto in Q più vicino a p. L'algoritmo è descritto dal seguente pseudocodice:

Algoritmo 1 Ricerca del nearest neighbour

1: **procedure** FIND-NEAREST(**point** p)
2: $\quad Q_\infty = C_\infty$
3: $\quad$ **for** $i = \infty$ to $-\infty$ **do**
4: $\quad\quad Q_{temp} = Children(q) : q \in Q_i$
5: $\quad\quad Q_{i-1} = q \in Q_{temp} : d(p,q) \leq d(p,Q_{temp}) + 2^i$
6: $\quad$ **return** $argmin_{q \in Q_{-\infty}} d(p,q)$

Lo scopo dell'algoritmo consiste nel selezionare, a ogni iterazione, i soli punti in C_i che potrebbero essere il nearest neighbour per p, oppure che potrebbero portare, con la discesa dell'albero, a trovarlo nei livelli successivi. Vediamo quindi nel dettaglio ciascun passo dello pseudocodice:

- **Passo 2** Essendo C_∞ il primo livello dell'albero, questo conterrà un solo nodo che verrà quindi considerato come candidato nearest neighbour.

- **Passo 3** La visita dell'albero viene eseguita di livello in livello, con l'utilizzo di un ciclo for. Si ricorda che i valori $+\infty$ e $-\infty$ vengono utilizzati al solo scopo semplificativo per lo pseudo-codice, e rappresentano rispettivamente il primo e ultimo strato dell'albero.

- **Passo 4** Dopo l'esecuzione di questa istruzione, Q_{temp} è l'insieme di tutti i figli dei nodi in Q_i (in modo che $Q_i \subseteq Q_{temp}$ per l'invariante di nidificazione); ciò permette di ottenere un nuovo set per il livello $i-1$ che includerà sicuramente l'antenato del nearest neighbour.

- **Passo 5** Non è necessario mantenere tutti i nodi presenti in Q_{temp}: i più "distanti" da p possono essere infatti rimossi. La condizione di rimozione verrà discussa nel teorema relativo alla correttezza dell'algoritmo.

- **Passo 6** Siccome il generico set Q_i contiene il nearest neighbour per il livello i, allora sicuramente $Q_{-\infty}$ conterrà il nearest neighbour assoluto, in quanto l'invariante (1) implica la presenza di tutti i punti al livello $-\infty$.

Per completare l'analisi relativa al funzionamento e alla correttezza della procedura è infine importante verificare che la condizione presente al passo 5. escluda solo i nodi che non possono essere antenati del nearest neighbour di p. Ciò viene dimostrato dal seguente teorema:

Teorema 2. *Sia T un cover tree su S, allora $Find-Nearest(p)$ ritorna il nearest neighbour del punto p in S.*

Dimostrazione. In assenza del punto 5, è facile notare come $Q_{-\infty}$ coinciderebbe con $C_{-\infty}$, includendo di conseguenza il nearest neighbour. Per dimostrare quindi la condizione di selezione, supponiamo di avere un nodo q al livello $i-1$ tale per cui $d(p,q) > d(p,Q_{temp}) + 2^i$; con questa distanza l'algoritmo rimuoverebbe q dal set Q_{temp}.
Si vuole quindi verificare che non esista un punto q' discendente di q tale per cui $d(p,q') <= d(p,Q_{temp})$, ossia con una distanza da p minore rispetto al candidato nearest neighbour in Q_{temp}. Ciò si può verificare attraverso la disuguaglianza triangolare; si ha infatti che $d(p,q') + d(q',q) \geq d(p,q) > d(p,Q_{temp}) + 2^i > d(p,Q_{temp})$, che dimostra la correttezza dell'algoritmo.

□

3.1.1 Ricerca dei k-nearest neighbours

Un'importante variante del problema è rappresentata dalla ricerca dei k elementi più vicini a q all'interno del cover tree.
In questo caso l'algoritmo subisce solo due modifiche che garantiscono comunque delle prestazioni simili al caso generale e, pertanto, non verranno fatte ulteriori analisi computazionali.

L'algoritmo modificato è il seguente:

Algoritmo 2 Ricerca dei k nearest neighbours

1: **procedure** FIND-NEAREST(**point** p, **integer** k)
2: $Q_\infty = C_\infty$
3: **for** $i = \infty$ to $-\infty$ **do**
4: $Q_{temp} = Children(q) : q \in Q_i$
5: $Q_{i-1} = q \in Q_{temp} : d(p,q) \leq Bound(Q_{temp}) + 2^i$
6: **return** $k\_argmin_{q \in Q_{-\infty}} d(p,q)$

Di seguito viene riportata un'analisi dei due punti modificati:

- **Passo 5** La condizione deve essere modificata in modo da mantenere k candidati nearest neighbours per ciascun livello. Per fare ciò è sufficiente considerare la distanza del k-esimo elemento più vicino a p in Q, restituito dalla funzione *Bound*. Il principio su cui si basa il teorema di correttezza è lo stesso e, pertanto, non verrà ulteriormente discusso.
- **Passo 6** Invece di un elemento, vengono restituiti k elementi, selezionati attraverso la nuova funzione k_argmin che opera sul set finale $Q_{-\infty}$.

L'analisi di correttezza si ottiene come come semplice variante del Teorema 2.

3.1.2 Approssimazione del nearest neighbour

Una seconda variante del problema, che permette di migliorare le prestazioni dell'algoritmo, consiste nella ricerca di un punto che sia sufficientemente vicino all'elemento considerato, ma che non sia necessariamente il nearest neighbour stesso. In particolare basta trovare un qualsiasi $q \in S$ tale che $d(p,q) \leq (1+\varepsilon)d(p,S)$, ossia con un errore accettabile sulla distanza, secondo il parametro $\varepsilon > 0$.
Di seguito viene riportato l'algoritmo modificato:

Algoritmo 3 Ricerca approssimata del nearest neighbour

1: **procedure** FIND-NEAREST(**point** p, **decimal** ε)
2: $Q_\infty = C_\infty$
3: $i = \infty$
4: **while** $2^{i+1}(1+1/\varepsilon) \leq d(p,Q_i)$ **do**
5: $Q_{temp} = Children(q) : q \in Q_i$
6: $Q_{i-1} = q \in Q_{temp} : d(p,q) \leq d(p,Q_{temp}) + 2^i$
7: $i = i - 1$
8: **return** $argmin_{q \in Q_i} d(p,q)$

Come si può notare, invece d'iterare tra tutti i livelli, il ciclo viene fermato in anticipo secondo la nuova condizione.
Il prossimo teorema dimostra che la relazione $d(p,q) \leq (1+\varepsilon)d(p,S)$ è valida quando il ciclo viene terminato, completando così l'analisi di correttezza.

Teorema 3. *Sia T un cover tree su S, allora $Find-Nearest(p,\varepsilon)$ ritorna un punto $q \in S$ tale che $d(p,q) \leq (1+\varepsilon)d(p,S)$.*

Dimostrazione. Supponiamo che l'algoritmo si concluda al livello i-esimo dopo che la condizione a linea (4) fallisce, si ha quindi che la distanza $d(p,Q_i)$ è al più $d(p,S)+2^{i+1}$. Ciò è dovuto al fatto che il punto $q \in Q_i$ più vicino a p è distante dal nearest neighbour al più 2^{i+1} per poterlo raggiungere discendendo l'albero, come visto nel Teorema 2.
La relazione $d(p,Q_i) \leq d(p,S)+2^{i+1}$ appena vista può essere combinata con la condizione presente in linea (4) $2^{i+1}(1+1/\varepsilon) \leq d(p,Q_i)$ con una sostituzione, per ottenere $2^{i+1}(1+1/\varepsilon) \leq d(p,S)+2^{i+1}$. Da quest'ultima relazione si ottiene $2^{i+1} \leq \varepsilon d(p,S)$ che, combinata con $d(p,Q_i) \leq d(p,S)+2^{i+1}$, permette di ricavare la disuguaglianza $d(p,q) = d(p,Q_i) \leq (1+\varepsilon)d(p,S)$.

□

3.2 Inserimento di un punto

Oltre all'operazione di query, la struttura dati deve prevedere una funzione d'inserimento che permetta l'aggiunta dinamica di punti. Di seguito viene riportato lo pseudocodice:

Algoritmo 4 Inserimento di un punto

```
1:  procedure INSERT(point p, cover set Q_i, level i)
2:      Q_temp = {Children(q) : q ∈ Q_i}
3:      if d(p, Q_temp) > 2^i then
4:          return "Elemento non inserito"
5:      Q_{i-1} = {q ∈ Q_temp : d(p,q) ≤ 2^i}
6:      result = Insert(p, Q_{i-1}, i-1)
7:      if result ="Elemento non inserito" and d(p, Q_i) ≤ 2^i then
8:          Scelgo q ∈ Q_i tale che d(p,q) ≤ 2^i
9:          Inserisco p in Children(q)
10:         return "Elemento inserito"
11:     return result
```

L'algoritmo genera, per ogni livello, il set Q_i contenente i soli punti di C_i in grado di essere possibili antenati di p. Ciò significa che tutti i punti $q \in C_i$ tali che $d(p,q) > 2^{i+1}$ vengono scartati, in quanto la distanza non permette a p di essere inserito in nessun sottoalbero avente q come radice.
Vediamo quindi nel dettaglio ciascun passo dello pseudocodice:

- **Passi 2-4** Come per l'operazione di query, anche per l'insert è utile creare un nuovo set Q_{temp} contenente i figli dei nodi in Q_i, che verranno poi selezionati per la creazione di Q_{i-1}; affinché un nodo venga inserito dal livello $i-2$ in poi (e, quindi, continuare la ricorsione), è necessario che la distanza tra $q \in Q_{temp}$ e p sia al più 2^i. Infatti, i nodi del livello $i-1$ presenti in Q_{temp}, possono diventare parenti di p solo se quest'ultimo risulta raggiungibile da loro discendendo l'albero, come visto nel teorema 1; oltretutto, se la condizione è falsa, si esclude anche che p possa essere aggiunto al livello $i-1$, in quanto non sarebbe verificata neppure $d(p,Q_i) \leq 2^i$, condizione necessaria per mantenere l'invariante di copertura del cover tree. Di conseguenza il fallimento della condizione al passo 3 porta a un fallimento generale dell'inserimento per i livelli inferiori ad i, terminando così la ricorsione.

- **Passi 5-7** Per propagare l'inserimento ai livelli inferiori, si genera Q_{i-1} con lo stesso principio di selezione visto nella query, rimuovendo i nodi che non permettono di raggiungere p in alcun modo. A questo punto si tenta l'inserimento di p nei livelli successivi, salvando il risultato in una nuova variabile. Se il nodo non viene aggiunto nei livelli sottostanti, si verifica che possa essere inserito al livello $i-1$, cercando un padre $q \in Q_i$ per p che renda valida l'invariante di copertura del cover tree. Se ciò avviene, p viene inserito come figlio di q e l'algoritmo si conclude con successo.

Per completare l'analisi dell'algoritmo d'inserimento, il seguente teorema ne dimostra la sua correttezza:

Teorema 4. *Sia T un cover tree su S, allora Insert$(p, C_{+\infty}, +\infty)$ genera un nuovo cover tree su $S \cup p$.*

Dimostrazione. Il primo passo consiste nel verificare che p venga sempre inserito all'interno dell'albero. Supponendo che p non sia già presente in T, è facile verificare che il caso base della ricorsione a riga 3 si verificherà necessariamente per un certo livello $i-esimo$, in quanto 2^i tende ad azzerarsi per $i \to -\infty$ e la distanza tra p ed il nearest neighbour è maggiore di zero. La condizione a riga 7, infine, dovrà essere vera per un certo $j > i$, eventualmente per $j = +\infty$ come caso peggiore, portando a un inserimento di p nell'albero.

Il secondo passo consiste nel verificare che il nuovo albero sia ancora un cover tree.
Una volta inserito p, questo viene implicitamente inserito anche nei livelli inferiori, mantenendo valida l'invariante di nidificazione. anche l'invariante di copertura è valida, in quanto la verifica è esplicita.
Infine, per verificare l'invariante di separazione, consideriamo $q \in C_{i-1}$ e distinguiamo i seguenti tre casi possibili:

- **Caso 1:** $q \in Q$ e $q \notin Q_{i-1}$. Se ciò è vero, allora q è stato scartato a riga 5. Questo può accadere solo se $d(p,q) > 2^i$ rendendo valida la separazione dei due punti.

- **Caso 2:** $q \in Q$ e $q \in Q_{i-1}$. Consideriamo la chiamata a livello i della *insert*, dove viene fatta la decisione d'inserire p. Necessariamente la chiamata $Insert(p, Q_{i-1}, i-1)$ ha restituito "Elemento non inserito", altrimenti l'if a riga 7 sarebbe falso. Affinché la chiamata al livello $i-1$ restituisca "Elemento non inserito" è necessario che la condizione a riga 3 sia verificata oppure che la condizione a riga 7 fallisca a causa della seconda clausola, essendo la prima verificata. In entrambi i casi viene ancora una volta verificata l'invariante di separazione

- **Caso 3:** $q \notin Q$. Se q non è presente in Q significa che il padre q' è stato rimosso ad un livello precedente $i' > i$, verificatasi la condizione $d(p,q') > 2^{i'}$. Con questi dati è possibile trovare un lower bound a $d(p,q)$: $d(p,q) \geq d(p,q') - \sum_{j=i'-1}^{i} 2^j$. La sommatoria indica, al più, quanto è possibile avvicinarsi al nodo p discendendo l'albero partendo da q' fino a raggiungere q, come visto nel teorema 2. Si può ora risolvere la sommatoria e imporre $d(p,q') = 2^{i'}$, come caso limite: $d(p,q) \geq d(p,q') - (2^{i'} - 2^i) = 2^{i'} - (2^{i'} - 2^i) = 2^i$.

□

3.3 Rimozione di un punto

Il procedimento di rimozione di un punto varia leggermente rispetto all'inserimento, con la particolare attenzione alla ristrutturazione dell'albero una volta rimosso ciascun nodo. L'algoritmo può essere descritto nei seguenti punti:

Algoritmo 5 Rimozione di un punto

```
1:  procedure REMOVE(point p, cover sets {Q_i, Q_{i+1}, ..., Q_∞}, level i)
2:      Q_temp = {Children(q) : q ∈ Q_i}
3:      Q_{i-1} = {q ∈ Q_temp : d(p,q) ≤ 2^i}
4:      if d(p, Q_temp) = 0 then
5:          Rimuovo p da Children(Parent(p)), da C_{i-1} e dai livelli inferiori
6:          for q ∈ Children(p) do
7:              i' = i - 1
8:              while d(q, Q_{i'}) > 2^{i'} do
9:                  Inserisco q in C_{i'} e Q_{i'}
10:                 i' = i' + 1
11:             Scelgo q' ∈ Q_{i'} tale che d(q,q') ≤ 2^{i'}
12:             Inserisco q in Children(q')
13:     else if Q_{i-1} is not Empty then
14:         Remove(p, {Q_{i-1}, Q_i, ..., Q_∞}, i - 1)
```

Anche in questo caso, come per l'inserimento, l'algoritmo genera un cover set Q_i per ogni livello contenente i soli nodi di C_i in grado di essere possibili antenati di p; una volta trovato il punto, questo deve essere rimosso e l'albero modificato. Vediamo quindi i passi principali dello pseudo-codice:

- **Passi 2-4:** La prima fase consiste nella ricerca di p, in modo analogo a quanto visto per la query, generando di volta in volta un nuovo set per il livello inferiore. Si procede con la ricorsione solo se p non è stato ancora trovato e sono rimasti dei nodi in Q_{i-1}.
- **Passi 5-6:** Una volta trovato il nodo si procede con la sua rimozione da tutti i set in cui compare e si rimuove il collegamento con il padre (Passo 5). Fisicamente, ciò comporta una eliminazione dei soli nodi espliciti dal livello $i-1$ in poi.
- **Passi 7-12:** Rimuovendo il punto p dall'albero, con tutti i nodi ad esso associati, alcuni discendenti non saranno più connessi all'albero. Chiamiamo con q un generico figlio di un nodo rimosso: allora, se esiste un nodo al livello superiore che rispetta l'invariante di copertura lo si può collegare direttamente a esso; se invece non esiste, bisogna inserire un nuovo nodo (uguale a q) al livello precedente e ripetere il procedimento risalendo l'albero. Come verrà successivamente dimostrato, ciò non viola alcuna invariante e rende l'albero un cover tree valido.
- **Passi 13-14:** Se Q_{i-1} è vuoto significa che la ricorsione deve concludersi, in quanto non sono presenti altri nodi in grado di condurre a p. Se ciò non è vero si procede con la ricorsione al livello successivo, in quanto p non è stato ancora trovato.

In questo caso, la verifica di correttezza risulta piuttosto semplice, in quanto differisce di poco rispetto agli altri algoritmi. Il seguente teorema mostra, in particolare, come la ristrutturazione dell'albero non violi le tre invarianti.

Teorema 5. *Sia T un cover tree su S, allora Remove$(p, C_{+\infty}, +\infty)$ genera un nuovo cover tree su $S - \{p\}$.*

Dimostrazione. Dopo la rimozione di p da tutti i livelli, rimangono valide le invarianti di nidificazione e separazione, mentre l'invariante di copertura non è più valida, in quanto alcuni nodi non sono collegati a nessun padre.
Per un nodo q tra questi, al generico livello $i-esimo$, si controlla se esiste un nodo q' candidato padre al livello superiore e, se ciò avviene, si collega q a q'. Questo primo caso non comporta nessuna ulteriore problematica e rende valida l'invariante di copertura per il nodo q.
Nel caso non esistesse alcun q' valido, si inserisce q al livello $i+1$ e si ripete il procedimento con il nodo aggiunto, fino a quando non si risolve l'invariante di copertura; l'inserimento, inoltre, non

viola né la condizione di nidificazione né quella di separazione.
Infatti non è possibile che esista $q'' \in C_{i+1}$ tale che $d(q'', q) \leq 2^{i+1}$ in grado di rendere invalida la condizione di separazione, siccome q'' sarebbe stato un candidato padre valido per q nell'iterazione precedente. □

3.4 Costruzione

Nella maggioranza dei casi è frequente la presenza di un dataset contenente migliaia di elementi, anzichè riceverli nel tempo.
In questi casi aggiungerne ciascuno tramite la funzione d'inserimento costituisce una grossa limitazione alle prestazioni, siccome ogni chiamata porta alla necessità di discendere nuovamente l'albero.
La funzione di costruzione vuole migliorare la complessità computazionale, basandosi su un unico set iniziale di punti. L'algoritmo è il seguente:

Algoritmo 6 Algoritmo di costruzione

```
 1: procedure CONSTRUCT(point p, point sets <Near, Far>, level i)
 2:     if Near is Empty then
 3:         return <p, Far>
 4:     else
 5:         <Self, Near> = Construct(p, Split(d(p, ·), 2^(i−1), Near), i − 1)
 6:         Aggiungo Self a Children(p_i)
 7:         while Near is not Empty do
 8:             Scelgo q in Near
 9:             <Child, Unused> = Construct(q, Split(d(q, ·), 2^(i−1), Near, Far), i − 1)
10:             Aggiungo Child a Children(p_i)
11:             <New-Near, New-Far> = Split(d(p, ·), 2^i, Unused)
12:             Aggiungo New-Far a Far e New-Near a Near
13:         return < p_i, Far >
```

Dato un punto p al livello i-esimo, la procedura crea un cover tree avente radice in p utilizzando tutti i punti contenuti nel set *Near* e un sottoinsieme dei punti contenuti in *Far*; in particolare *Near* contiene i soli punti distanti al più 2^i da p, ciascuno dei quali può essere sempre inserito nell'albero senza violare alcuna invariante, in quanto raggiungibile da p discendendo l'albero. In *Far* sono invece presenti i punti $p' \in S$ tali che $2^i < d(p, p') < 2^{i+1}$. In quest'ultimo caso non è detto che esista un nodo per cui è possibile collegare p', basti pensare al caso in cui l'albero presenta il solo nodo p.
In generale la procedura tenta d'inserire il più elementi possibili appartenenti a *Far* senza violare nessuna invariante. Per semplificare lo pseudocodice si fa uso di una funzione ausiliaria

Split(d(p,·),r,S_1,S_2,...) in grado di ritornare una coppia <*Near*, *Far*> creati a partire dai punti presenti in S_1,S_2,... che soddisfano rispettivamente $d(p,q) \leq r$ e $r < d(p,q) < 2r$.
La funzione, oltre a creare i due set, rimuove i punti selezionati da S_1,S_2,... Vediamo quindi ciascun passo importante dello pseudocodice:

- **Passi 2-4:** L'algoritmo termina quando il set *Near* è vuoto, in quanto per ogni elemento $q \in Far$ è valida la relazione $d(p,q) > 2^i$ che viola l'invariante di copertura.
- **Passi 5-6:** La costruzione dell'albero sul nodo p è ricorsiva e consiste nella creazione di più sotto alberi figli che verranno poi connessi alla radice p. Bisogna però fare particolare attenzione all'invariante di nidificazione, e gestirla separatamente; il passo 5 chiama la funzione sul figlio medesimo di p ripartendo tutti i punti ancora da inserire in due set, con la funzione split. I punti contenuti in *Far* sicuramente non potranno far parte dell'albero con radice p al livello $i-1$, data la loro distanza. La funzione ritorna il nodo p_{i-1} e un set di elementi non usati. Infine il passo 6 connette p_i a p_{i-1}.
- **Passi 7-10:** Per i restanti punti il funzionamento è analogo, viene scelto un elemento $q \in Near$ a ogni iterazione e costruito il sotto albero avente q come radice; in questo caso alcuni punti $q' \in Far$ potrebbero essere considerati per l'inserimento e rispettare l'invariante di copertura. La funzione ritorna un insieme di nodi che non sono stati usati e lo stesso nodo q, che viene poi inserito come figlio di p per connettere il sotto albero.
- **Passi 11-12:** Gli ultimi passi inseriscono i punti appartenenti al set *Unused* nei rispettivi set *Near* e *Far*, in quanto non ancora inseriti.

Infine, la costruzione dell'albero può iniziare con la chiamata Construct($p \in S$, $<S-\{p\}, \emptyset>$, $+\infty$), dove p rappresenta un qualsiasi punto in S, il set *Near* contiene tutti i punti (ad eccezione di p) che sicuramente verranno inseriti al di sotto di p, il set *Far* è vuoto di conseguenza e il livello iniziale è $+\infty$.
Per completare l'analisi dell'algoritmo, il seguente teorema ne dimostra la sua correttezza:

Teorema 6. *Construct($p \in S$, $<S-\{p\}, \emptyset>$, $+\infty$) ritorna un cover tree valido su S.*

Dimostrazione. La condizione di nidificazione è valida, dato l'inserimento esplicito del figlio stesso di ciascun punto al passo 5.
Anche la condizione di copertura è sempre valida, in quanto i figli del nodo p vengono scelti sempre tra i punti del set *Near* che, per definizione, non superano la distanza di 2^i da p.
Infine, per verificare la condizione di separazione al livello $i-esimo$, consideriamo due generici

punti $u, v \in S$ tali per cui $d(u,v) \leq 2^i$; affinché l'invariante venga rispettata, non possono essere presenti entrambi i punti al livello $i-esimo$. Supponiamo che, tra u e v, la prima chiamata alla funzione di costruzione avvenga per il punto u, ad un certo livello $k \geq i$; allora le chiamate immediatamente successive gestiranno i nodi $u_{i-1}, u_{i-2}, u_{i-3}...$, siccome il passo 5 impone una costruzione per profondità dell'albero. Inoltre, dato che il set *Near* relativo al livello i contiene tutti i punti di S non ancora inseriti che sono distanti al più 2^i da p, si ha che $v \in Near$, altrimenti v sarebbe già stato inserito nell'albero contraddicendo l'ipotesi iniziale.
Ciò conclude la dimostrazione, infatti tutti i punti del set *Near* vengono sempre aggiunti nel sottoalbero e, di conseguenza, in un livello sottostante ad i. □

3.5 Ricerca dei nearest neighbours per più punti

L'algoritmo di query può essere facilmente modificato per effettuare la ricerca simultanea del nearest neighbour per più punti, rappresentati da un cover tree T'.
Di fatto questa operazione può portare ad un forte vantaggio nelle prestazioni, anche considerando un pre-processing tramite funzione di costruzione per il cover tree T'; ciò accade in particolare quando le richieste di ricerca sono numerose e consecutive.
L'algoritmo modificato è il seguente:

Algoritmo 7 Ricerca dei nearest neighbours di più punti

1: **procedure** FIND-ALL-NEAREST(**query cover tree** p_j, **cover set** Q_i)
2: **if** $i = -\infty$ **then**
3: **return** $argmin_{b \in Q_{-\infty}} d(a,b)$ come nearest neighbour di a, per ogni $a \in L(p_j)$
4: **else**
5: **if** $j < i$ **then**
6: $Q_{temp} = Children(q) : q \in Q_i$
7: $Q_{i-1} = \{q \in Q_{temp} : d(p_j, q) \leq min_{q \in Q} d(p_j, q) + 2^i + 2^{j+2}\}$
8: **Find-All-Nearest**(p_j, Q_{i-1})
9: **else**
10: **Find-All-Nearest**(q_{j-1}, Q_i) per ogni $q_{j-1} \in Children(p_j)$

Il funzionamento è molto simile all'algoritmo di query per più punti del KD-tree [4], con qualche modifica; l'idea consiste nel discendere il query cover tree e, per ogni sotto albero con radice p_j, associare un cover set Q_i con tutti i nodi del livello $i-esimo$ in grado di ricondurre al nearest neighbour di ciascun elemento del sotto albero di ricerca.
Per fare ciò, ogni volta che il query cover tree viene espanso in più sotto alberi, si espande successivamente anche il cover set Q_i (se $j < i$) selezionando i soli nodi candidati nearest neighbour.

Rispetto all'algoritmo di query per un punto singolo, la condizione di selezione è stata modificata aumentando la distanza. Per analizzarla consideriamo l'esempio in figura 3.1, per uno spazio

monodimensionale, rappresentante il caso peggiore possibile che rende necessaria la condizione al passo 7.

Indichiamo con q l'elemento più vicino a p in Q_{temp} e con q' un generico punto di Q_{temp}. Indichiamo inoltre con p' il punto più lontano raggiungibile da p (verso destra) e con q'' il punto più lontano raggiungibile da q' (verso sinistra).

Come visto nei teoremi precedenti, si ha che $d(p,p') = 2^{j+1}$ e $d(q',q'') = 2^i$. Si ha inoltre che $d(p',q) = d(p,q) + 2^{j+1}$ e, affinché q' non venga rimosso da Q, si vuole avere $d(p',q'') < d(p',q)$. Sostituendo si ottiene $d(p',q'') < d(p,q) + 2^{j+1}$ ma, siccome $d(p',q'') = d(p,q') - 2^i - 2^{j+1}$, si può ricavare la disequazione $d(p,q') - 2^i - 2^{j+1} < d(p,q) + 2^{j+1}$ da cui si ottiene $d(p,q') < d(p,q) + 2^{j+2} + 2^i$.

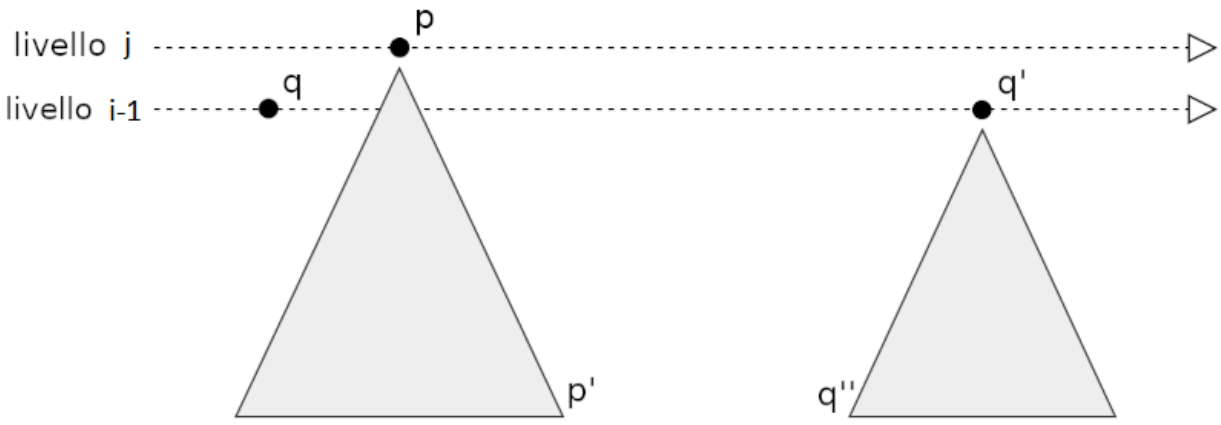


Figura 3.1. Esempio di caso limite per la selezione di un nodo in Q_{temp}

Capitolo 4

Analisi delle prestazioni

Come già anticipato, l'intera analisi prestazionale dei Cover Tree si basa sulla dimensionalità intrinseca del dataset, essendo di forte impatto sulle prestazioni degli algoritmi.
In particolare il prossimo paragrafo fornisce tre risultati che saranno di fondamentale importanza per mettere in relazione le prestazioni delle operazioni sulla struttura dati con la costante di espansione.
Infine i successivi paragrafi dimostrano analiticamente le complessità computazionali degli algoritmi di query, inserimento e rimozione.

4.1 Analisi e considerazioni preliminari

Prima di proseguire con l'analisi computazionale di ciascuna operazione, vengono introdotti i seguenti importanti lemmi che semplificano le dimostrazioni dei prossimi teoremi:

Lemma 1. *(Width Bound) Il numero di figli di un qualsiasi nodo p è limitato superiormente da c^4.*

Dimostrazione. Supponiamo che p sia al livello i-esimo; allora il numero di figli di p è limitato superiormente da $|B(p,2^i) \cap C_{i-1}|$, ossia dal numero di nodi del livello $i-1$ che, per l'invariante di copertura, sono distanti al più 2^i da p.
L'idea del teorema consiste nel trovare il numero di palle disgiunte aventi raggio 2^{i-2} che possano essere contenute in $B(p,2^{i+1})$; ciascuna di queste palle può infatti contenere un solo punto di C_{i-1}, per l'invariante di separazione applicata al livello $i-1$, e permette quindi di stabilire un upper-bound del numero di figli di p.
Si noti che viene usato $B(p,2^{i+1})$ anzichè $B(p,2^i)$ in modo da poter considerare qualsiasi palla avente centro in q, anche per nodi distanti fino a 2^i da p, ossia agli estremi.

Per un certo figlio q di p, essendo $d(p,q) \leq 2^i$, è sicuramente vero che $B(p,2^{i+1}) \subset B(q,2^{i+2})$, e ciò permette di ottenere il seguente risultato:

$$|B(p,2^{i+1})| \leq |B(q,2^{i+2})| \leq c^4 |B(q,2^{i-2})| \quad (1)$$

dove la seconda relazione deriva direttamente dalla definizione di expansion constant, applicata quattro volte di seguito.

Come già accennato, le palle $|B(q,2^{i-2})|$ con $q \in C_{i-1}$ sono tra di loro separate, come conseguenza dell'invariante di separazione. Essendo inoltre $B(q,2^{i-2}) \subset B(p,2^{i+1})$ per la distanza $d(p,q) \leq 2^i$, si può ottenere un upper bound per il numero di figli di p:

$|B(p,2^i) \cap C_{i-1}| \leq |B(p,2^{i+1}) \cap C_{i-1}| \leq \frac{B(p,2^{i+1})}{B(q,2^{i-2})} \leq c^4$

Dove l'ultima disuguaglianza si ottiene direttamente da (1).

□

Come visto nel secondo capitolo, la definizione di costante di espansione fornisce un upper bound alla crescita di una qualsiasi palla di centro p, con la variazione del raggio. Il prossimo lemma vuole invece dimostrare l'esistenza di un lower bound, che sarà utile per il calcolo dell'altezza dell'albero in funzione di c.

Lemma 2. *(Growth Bound) Per ogni punto $p \in S$ e raggio $r > 0$ se esiste un punto $q \in S$ tale che $2r < d(p,q) < 3r$, allora $|B(p,4r)| \geq (1 + \frac{1}{c^2})|B(p,r)|$.*

Dimostrazione. Siccome $d(p,q) \leq 3r$ si ha che $B(p,r) \cap B(q,3r+r)$ da cui segue la relazione

$|B(p,r)| \leq |B(q,4r)| \leq c^2|B(q,r)|$ (1)

dove l'ultima disuguaglianza è stata ottenuta applicando due volte la definizione di costante di espansione. Essendo la distanza $d(p,q) > 2r$ si ha che $B(p,r)$ e $B(q,r)$ non si intersecano, e sono contenute completamente in $B(p,4r)$. Ciò comporta la relazione $|B(p,4r)| \geq |B(p,r)| + |B(q,r)|$. Infine, sostituendo a $|B(q,r)|$ il lower bound $\frac{1}{c^2}|B(p,r)|$ trovato nella disuguaglianza (1), si ottiene $|B(p,4r)| \geq (1 + \frac{1}{c^2})|B(p,r)|$.

□

Il terzo ed ultimo lemma fornisce un importante risultato sulla profondità esplicita di un punto dell'albero, mettendola in relazione con la costante di espansione.
La profondità esplicita di un punto p viene definita come il numero di nodi espliciti nel percorso tra il nodo radice e il livello più basso in cui p è esplicito.

Lemma 3. *(Depth Bound) La profondità esplicita di un nodo $p \in S$ è $O(c^2 \log n)$*

Dimostrazione. Consideriamo il set $S_i = \{q \in S : 2^{i+1} \leq d(p,q) < 2^{i+2}\}$. Vogliamo dimostrare che per un certo punto p, il set S_i contiene al più quattro antenati espliciti di p, e che si trovano nei set C_i, C_{i+1}, C_{i+2} e C_{i+3}. Per fare ciò, dimostriamo innanzitutto che se un nodo $q \in S_i$ è antenato di p, allora $q \in C_i$. Considerando un nodo q antenato di p che appare in C_j, sicuramente sarà vera la relazione $d(p,q) < 2^{j+1}$ per far sì che p sia raggiungibile da q discendendo l'albero. Essendo q appartenente ad S_i si ha che $2^{i+1} \leq d(p,q)$ ma ciò è possibile solo se $j \geq i$, di conseguenza qualsiasi nodo riferito al punto q, antenato di p, si troverà per la proprietà di nidificazione anche dal livello

C_i in poi.
Possiamo ora considerare gli antenati univoci di p appartenenti ad S_i e dimostrare che sono al più quattro. Dato che qualsiasi $q \in S_i$ antenato di p deve trovarsi ad un livello superiore o uguale ad i, possiamo considerare i soli set C_i, C_{i+1}, C_{i+2}.... Supponiamo ora che esistano due antenati q', q'' diversi da p in C_i, uno dei quali che si trova necessariamente anche ad un livello superiore; se entrambi appartenessero ad S_i si avrebbe che $d(q', q'') < 2^{i+2} + 2^{i+2} = 2^{i+3}$.
La condizione di separazione del livello $i+3$ impone però che i due punti siano distanti almeno 2^{i+3} e ciò comporta che ci possa essere al massimo un punto in C_{i+3} con le caratteristiche cercate. Di conseguenza S_i contiene un numero di antenati univoci di p ai soli livelli C_i, C_{i+1}, C_{i+2} e C_{i+3} e, pertanto, in numero costante.
L'ultimo passo per dimostrare la profondità esplicita dell'albero consiste nel trovare il numero di anelli S_i intorno a p che riescano a coprire tutti i punti in S. Per fare ciò sarà utile il lower bound ottenuto dal lemma 2, che necessita però di un secondo punto q per essere applicato. Consideriamo quindi il punto q più vicino a p e poniamo $r = \frac{d(p,q)}{2}$ in modo da poter applicare il growth bound e trovare la relazione $|B(p,4r)| \geq (1+\frac{1}{c^2})|B(p,r)| = 1+\frac{1}{c^2}$. Consideriamo ora il prossimo punto q' tale che $d(p,q') > 8r$ e applichiamo nuovamente il growth bound con $r' = \frac{d(p,q')}{2}$ per ottenere la relazione $|B(p,4r')| \geq (1+\frac{1}{c^2})|B(p,r')| \geq (1+\frac{1}{c^2})|B(p,4r)| \geq (1+\frac{1}{c^2})^2$. Procedendo ricorsivamente in questo modo stiamo coprendo tutto lo spazio di punti S con anelli di raggio sempre maggiore; vogliamo quindi continuare fino a quando non è valida la relazione $|B(p,r)| \geq n$, ossia pari al numero di punti in S. Dopo k applicazioni del growth bound si ottiene un lower bound pari a $(1+\frac{1}{c^2})^k$ che posto uguale ad n implica $k = \frac{\log n}{log(1+\frac{1}{c^2})}$ iterazioni massime. I set di punti associati ad ogni iterazione possono appartenere al massimo a quattro set S_i diversi, portando ad un numero di anelli pari a $O(\frac{\log n}{log(1+\frac{1}{c^2})})$. Quest'ultimo risultato, una volta sviluppato asintoticamente e applicate le proprietà della notazione O-grande, può essere riscritto come $O(c^2 \log n)$, essendo $c \geq 2$.
Il numero di antenati espliciti di p in ogni S_i è costante e ciò completa la dimostrazione.

□

4.2 Analisi computazionale

Ora che abbiamo tutti gli strumenti necessari all'analisi computazionale del Cover Tree, possiamo procedere con il seguente teorema relativo alle prestazioni dell'algoritmo di query:

Teorema 7. *Per un punto p ed un set S con costante di espansione c, il nearest neighbour $q \in S$ di p può esser trovato in tempo $O(c^{12} \log n)$.*

Dimostrazione. Consideriamo Q^* l'ultimo set Q_i con almeno un nodo esplicito; siccome la profondità massima di un nodo nell'albero è pari a $k = O(c^2 \log n)$, il numero di iterazioni del passo

3 può essere al più $k|Q^*| \leq k max_i|Q_i|$. Ciò si può notare considerando le sole iterazioni in cui il set Q_i è esplicito ed in cui i percorsi espliciti per raggiungere ciascun nodo in Q^* hanno il minor numero di nodi in comune possibile.
Ad ogni iterazione vengono individuati i punti di Q_i espliciti in tempo $O(max_i|Q_i|)$, attraverso una ricerca lineare; considerando anche il numero di iterazioni massime, la complessità totale è quindi pari a $O(k max_i|Q_i|^2)$.
Al passo 4 avviene una espansione per i soli nodi espliciti presenti in Q_i e, pertanto, ha senso considerare questa istruzione globalmente, indipendentemente dal ciclo for. In particolare il numero di nodi espliciti totali considerati sono $O(k max_i|Q_i|)$, come visto ad inizio dimostrazione; di conseguenza la complessità computazionale relativa al passo 4 è pari a $O(k c^4 max_i|Q_i|)$, dove il fattore c^4 deriva dall'espansione di ogni nodo nei suoi figli, secondo il lemma 1. I passi rimanenti impiegano un tempo di esecuzione inferiore rispetto al passo 4, e ciò porta ad una complessità finale dell'algoritmo di query pari a $O(k c^4 max_i|Q_i| + k max_i|Q_i|^2)$ dato dalla somma dei due risultati.
Infine, per concludere la dimostrazione, è sufficiente verificare che $max_i|Q_i| \leq c^5$.

Per fare ciò consideriamo il set Q_{i-1} generato all'iterazione i-esima e definiamo $d = d(p,Q)$; allora possiamo riscrivere il set nel seguente modo: $Q_{i-1} = \{q \in Q : d(p,q) \leq d + 2^i\} = B(p, d+2^i) \cap Q \subseteq B(p, d+2^i) \cap C_{i-1}$. Ciò risulta vero poichè Q è un sottoinsieme di C_{i-1} generato al livello i.

Procediamo suddividendo il problema in due casi più semplici da risolvere individualmente, ossia $d > 2^{i+1}$ e $d \leq 2^{i+1}$.
Consideriamo il primo caso, da cui si può ricavare la seguente relazione: $|B(p, d+2^i)| \leq |B(p, 2d)| \leq c^2|B(p, \frac{d}{2})|$ dove la prima disuguaglianza si ottiene direttamente dalla relazione $d > 2^{i+1}$ e la seconda applicando la definizione di costante di espansione. Ricordando che da qualsiasi punto $q \in C_{i-1}$ ci si può avvicinare a p di una distanza inferiore a 2^i, segue la disuguaglianza $d \leq d(p,S) + 2^i$; Quest'ultima, combinata con l'assunzione $d > 2^{i+1}$, permette di ricavare $d(p,S) \geq d - 2^i > 2^i$. Ciò significa, siccome $\frac{d}{2} < 2^i$, che $B(p, \frac{d}{2}) = \{p\}$ da cui si ricava per sostituzione $|B(p, d+2^i)| \leq c^2$ e $|Q_{i-1}| \leq c^2$ che dimostra il primo caso.

Per il secondo caso consideriamo un punto $q \in C_{i-1}$ e proseguiamo, come per il lemma 1, nel trovare il numero di palle disgiunte di raggio 2^{i-2} contenute in $B(p, d+2^i+2^{i-2}) \cap C_{i-1}$. Ciascuna palla può infatti coprire un solo punto di $B(p, d+2^i) \cap C_{i-1}$, e l'aggiunta di 2^{i-2} permette di coprire anche i punti più esterni. Anche la seguente disuguaglianza è analoga al primo lemma, e permette di ottenere un upper bound per $|Q_{i-1}|$: $|Q_{i-1}| \leq |B(p, d+2^i) \cap C_{i-1}| \leq |B(p, d+2^i+2^{i-2})| \leq |B(q, 2(d+2^i)+2^{i-2})| \leq |B(q, 2^{i+2}+2^{i+1}+2^{i-2})| \leq |B(q, 2^{i+3})| \leq c^5|B(q, 2^i-2)|$.
In entrambi i casi si ha $|Q_{i-1}| \leq c^5$, che dimostra il teorema. □

Il prossimo teorema analizza la complessità computazionale delle operazioni di inserimento e rimozione:

Teorema 8. *L'inserimento e la rimozione in un cover tree richiedono tempo di esecuzione* $O(c^6 \log n)$.

Dimostrazione. Per dimostrare la complessità computazionale relativa all'inserimento, consideriamo un nodo q presente nei cover set Q_i, Q_{i-1}, Q_{i-2}; se un secondo nodo q' fosse presente in Q_i, questo non potrebbe apparire anche in Q_{i-2} in quanto la condizione di separazione del livello i-esimo impone $d(q,q') > 2^i$ e il passo 5 dell'algoritmo impone che la distanza massima tra due punti qualsiasi in Q_{i-2} sia al massimo $2^{i-1} + 2^{i-1} = 2^i$. Indichiamo quindi con $k = c^2 log|S|$ la profondità esplicita di un punto e calcoliamo il numero massimo di cover set espliciti possibili, in relazione con k. Dati quattro nodi, di cui uno sempre presente in Q_i, Q_{i-1}, Q_{i-2}, si può massimizzare il numero di cover set espliciti nel caso in cui i primi tre nodi siano espliciti ciascuno in uno dei tre livelli, ed il quarto nodo non lo sia. In questo caso, estendendo il ragionamento a tutto l'albero, si ottengono $3k$ cover set diversi; si deve però aggiungere un fattore k relativo al quarto nodo che può essere esplicito in altri k livelli dell'albero.

Infine, ciascuno step dello pseudocodice non supera la complessità computazionale di $O(max_i|Q_i|)$ e, moltiplicando il numero di livelli con cover set espliciti (proporzionale a k) a questo risultato, si ottiene $O(k max_i|Q_i|)$. Ciascun cover set Q_i contiene punti in S non più distanti di 2^i da p e, come visto nel primo lemma, ciò comporta un massimo di c^4 nodi; moltiplicando questi risultati si ottiene la complessità computazionale $O(c^6 \log n)$.

L'algoritmo di rimozione è simile, con l'aggiunta di alcuni step volti alla ristrutturazione dell'albero una volta rimosso un nodo. Risalire l'albero ha un costo computazionale di $O(k max_i|Q_i|)$, dimostrabile con lo stesso ragionamento dell'inserimento. Il costo non è maggiore per gli altri step, e ciò rende la complessità dell'algoritmo pari a $O(c^6 \log n)$.

□

Teorema 9. *(Solo enunciato) L'algoritmo di costruzione di un cover tree su un set S richiede tempo di esecuzione* $O(c^6 n \log n)$.

Teorema 10. *(Solo enunciato) L'algoritmo di ricerca batch richiede tempo di esecuzione* $O(c^{16} n)$.

Capitolo 5

Analisi e confronto delle implementazioni più utilizzate

In questo ultimo capitolo vengono analizzate le prestazioni delle tre implementazioni più popolari della struttura dati, scritte nei linguaggi Python (*ngeiswei-CoverTree* [5]), C++ (*DNCrane-CoverTree* [6]) e Java (*loehndorf-CoverTree* [7]). In particolare vengono confrontate le prestazioni dell'algoritmo di query su quattro diversi dataset, e proposti dei miglioramenti in grado di incrementare l'efficienza di inserimento, rimozione e ricerca.

5.1 Risultati sperimentali

Per l'analisi sperimentale ognuno dei tre progetti è stato testato con quattro dataset che variano in dimensionalità intrinseca, numero di punti e dimensioni.
Per valutare le prestazioni della query, limitando il più possibile le differenze dei tempi di esecuzione dovuti ai linguaggi di programmazione ed alla macchina su cui vengono eseguiti i programmi, è stato utilizzato il fattore di speedup; nello specifico è stato misurato il tempo necessario alla ricerca lineare del nearest neighbour per ciascun punto del dataset e, successivamente, rapportato con il tempo necessario alla ricerca del nearest neighbour attraverso la funzione $Find-Nearest$ presente in ciascuna libreria.
In entrambi i casi l'output di ogni query sarà il punto stesso di ricerca, con una distanza pari a zero.
Infine gli speedup finali sono stati calcolati come la media degli speedup su 5 test separati, per ottenere dati più precisi e facilmente confrontabili. I risultati vengono mostrati nella tabella e in figura 5.1.

Implementazione	**ionosphere.data**	**optdigits.tra**	**pendigits.tra**	**letter.data**
Ngeiswei-CoverTree	0.71	0.64	2.461	1.222
DNCrane-CoverTree	0.5	2.114	3.61	2.822
Loehndorf-CoverTree	0.77	1.019	2.425	2.758

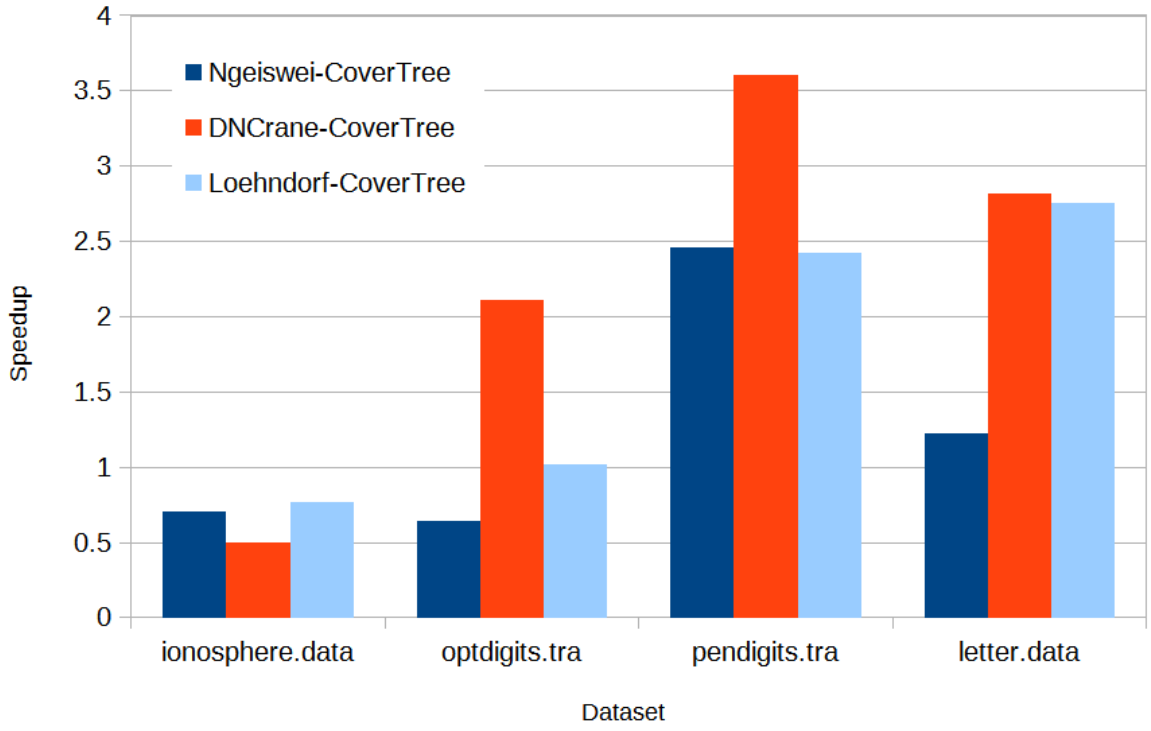


Figura 5.1. Speedup di ciascuna implementazione per l'operazione di query

Come si può notare, la soluzione in C++ ha un leggero vantaggio soprattutto nel caso di insiemi di punti con dimensionalità intrinseca elevata e con molti elementi (*pendigits.tra* e *letter.data*), mentre l'implementazione in Python ha, mediamente, le prestazioni peggiori. Queste differenze non sono dovute alla complessità computazionale delle tre strutture dati, in quanto molto simili tra di loro, ma ad ulteriori scelte implementative che verranno mostrate in seguito.
Le prestazioni sono comunque inferiori rispetto al progetto originale, dove lo speedup raggiunge valori molto più elevati in tutti e quattro i dataset.
Nei prossimi paragrafi verranno studiati le tre soluzioni e suggerite delle modifiche in grado di migliorare la loro efficienza.

5.2 Analisi delle implementazioni

La prima implementazione descritta è ngeiswei-CoverTree. Il funzionamento è molto semplice, e si basa sulla creazione di un solo oggetto di classe nodo per ciascun punto, che contiene le sue coordinate e una mappa che associa ad ogni livello una lista di eventuali nodi figli. La memoria utilizzata è $O(n)$ dato che la mappa non contiene mai il nodo medesimo, che può invece essere restituito con la chiamata alla funzione $getChildren(level)$ insieme ai restanti punti.
Le operazioni principali sono state implementate esattamente come visto negli pseudocodici, con-

siderando ciascun nodo implicito come se fosse esplicito. Non sono presenti particolari ottimizzazioni per migliorare le prestazioni. Sono invece presenti alcune istruzioni non necessarie, come la creazione di un nuovo set Q ad ogni step dell'algoritmo di query o il calcolo della radice quadrata per la distanza.

La seconda implementazione, avente le prestazioni migliori, è DNCrane-CoverTree. Il funzionamento è simile alla soluzione in Python, con la differenza che la funzione $getChildren(level)$ non ritorna mai il nodo stesso. L'operazione di query cambia leggermente, con l'utilizzo di un solo set Q che viene riutilizzato aggiungendo dei nodi figli o rimuovendo i punti con una elevata distanza. Anche in questo caso non sono presenti particolari ottimizzazioni. Uno svantaggio rispetto alla struttura precedente è l'utilizzo delle mappe ordinate (con albero di ricerca bilanciato) per mantenere i figli di un particolare nodo in un determinato livello; ciò non comporta un grosso cambiamento prestazionale dati i pochi livelli presenti, ma può essere risolto facilmente con le unordered map.

Infine, la terza ed ultima implementazione è loehndorf-CoverTree. In questo caso viene istanziato un nuovo oggetto di classe nodo anche nel caso in cui quest'ultimo possa essere rappresentato implicitamente, risultando nella perdità della linearità di memoria. Nello specifico viene aggiunto un nodo singolo all'inserimento di un punto e, qualora venisse richiesto il punto stesso ad un livello inferiore, verrebbe istanziato un nuovo oggetto. Le operazioni della struttura dati hanno un funzionamento analogo a ngeiswei-CoverTree, con la creazione di nuovi set per ogni livello in cui si discende l'albero, che può peggiorare le performance in livelli con pochi nodi figli.
Nonostante ciò, le prestazioni sono comunque migliori dell'implementazione in Python, e questo è dovuto principalmente all'utilizzo di una base pari a 1.2. Si possono quindi confrontare gli speedup usando la base 2 (figura 5.2).

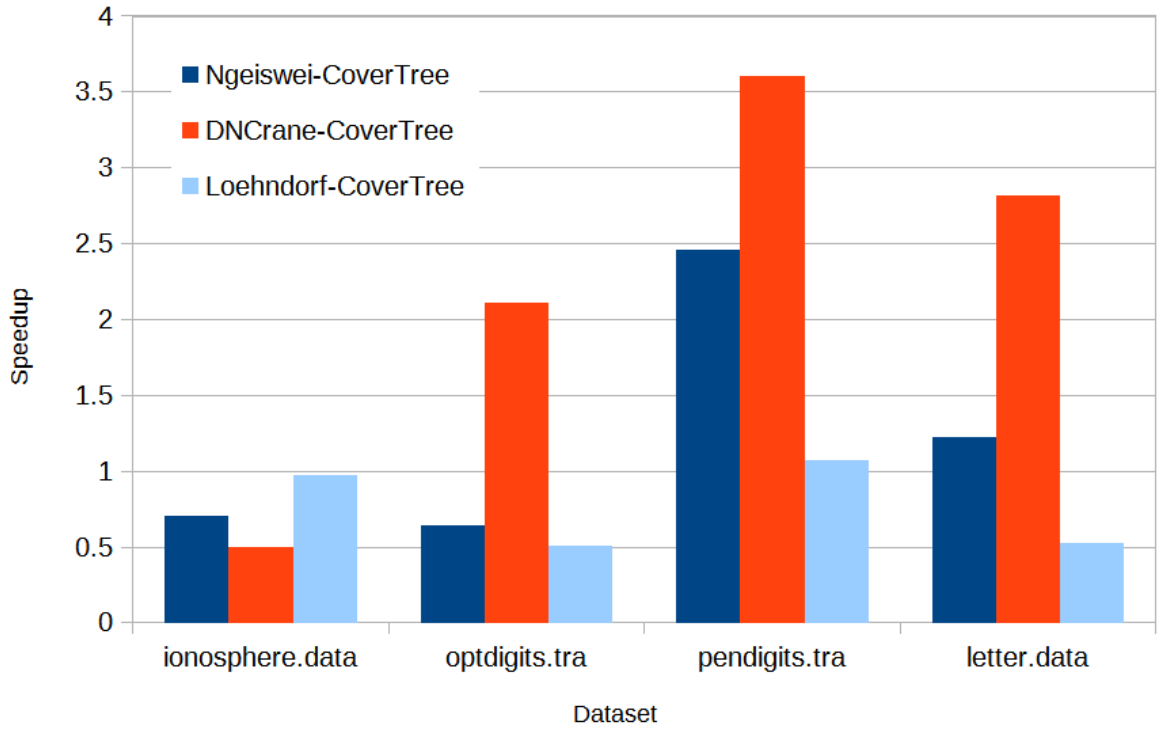


Figura 5.2. Speedup di ciascuna implementazione per l'operazione di query, con base pari a 2

Implementazione	**ionosphere.data**	**optdigits.tra**	**pendigits.tra**	**letter.data**
Ngeiswei-CoverTree	0.71	0.64	2.461	1.222
DNCrane-CoverTree	0.5	2.114	3.61	2.822
Loehndorf-CoverTree	0.978	0.511	1.075	0.528

5.3 Soluzione alternativa

Come visto nel paragrafo precedente, tutte e tre le soluzioni implementano le operazioni della struttura dati senza distinguere la rappresentazione implicita dei nodi da quella esplicita, portando ad una riduzione notevole delle prestazioni.

In figura 5.3 vengono riportate, per ciascun dataset, le percentuali di livelli aventi cover set impliciti, calcolato come media delle percentuali relative a ciascuna operazione di query. Questi risultati sono stati ottenuti con il progetto Loehndorf-CoverTree ma, essendo la struttura dell'inserimento pressoché identica ai progetti in Ptyhon e C++, le percentuali risultano simili. I grafici mostrano in particolare una percentuale di iterazioni non necessarie che possono raggiungere anche il 50%. Di fatto tutti i teoremi visti che analizzano le prestazioni del Cover Tree non considerano i livelli

dove tutti i nodi sono impliciti, e sembra quindi ragionevole modificare le tre implementazioni di conseguenza.

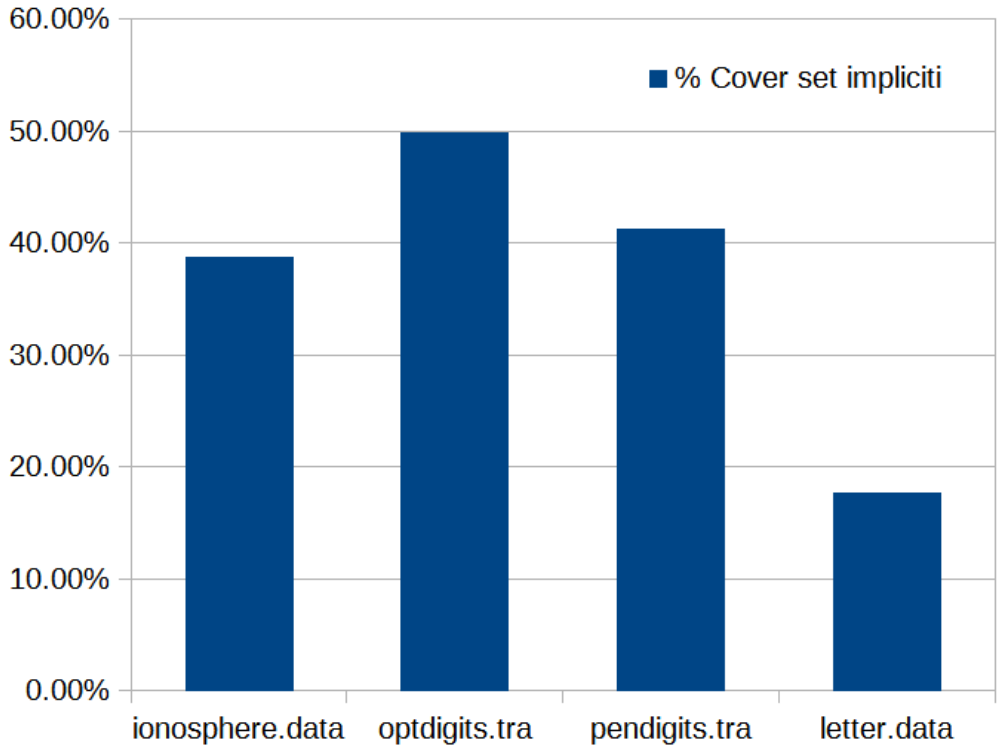


Figura 5.3. Media delle percentuali di cover set impliciti calcolati su ogni query

	ionosphere.data	**optdigits.tra**	**pendigits.tra**	**letter.data**
Percentuale	38.8%	49.9%	41.3%	17.7%

Consideriamo ora DNCrane-CoverTree (con le prestazioni migliori) come base a cui apportare le modifiche. Una possibile soluzione consiste nell'uso di una unordered map per ogni oggetto di classe nodo dove, per ogni livello in cui esso è esplicito, si associano il livello inferiore e superiore in cui il punto è ancora esplicito (se esistono).
Di seguito vengono riportate le principali modifiche da apportare all'algoritmo di query:

- Ciascun set Q_i, oltre a memorizzare i nodi di interesse per il livello i-esimo, deve memorizzare per ciascun punto anche il primo livello j inferiore ad i in cui è presente un nodo esplicito ad esso associato. Mantenere questa dato permette, di iterazione in iterazione, di accedere velocemente alla mappa e aggiornare le informazioni in Q_i in tempo $O(1)$ ammortizzato.

- Ad ogni passo iterativo i dell'algoritmo si calcola il prossimo livello j in cui è presente un cover set esplicito; ciò si ricava come massimo tra i livelli associati a ciascun nodo in Q_i. Successivamente viene calcolato direttamente Q_j e si ignorano le iterazioni da $i-1$ a $j+1$.

Per quanto riguarda l'operazione di inserimento, il principio di funzionamento è simile; di seguito vengono riportare le principali modifiche da apportare all'algoritmo:

- Per ogni set Q_i si memorizza, per ciascun punto, il primo livello j inferiore ad i in cui è presente un nodo esplicito ad esso associato. Aggiornando questa informazione di set in set, si può ricavare velocemente il prossimo livello avente un cover set esplicito.
  Rispetto all'algoritmo di query, non si possono sempre ignorare tutti i livelli da $i-1$ a $j+1$: potrebbe infatti esistere un cover set implicito contenente un candidato padre di p. Al passo ricorsivo di livello i, quindi, si calcola anche il primo livello k inferiore ad i tale per cui la condizione $d(p,Q_i) > 2^k$ del passo 3 risulta verificata (ricordando che tutti i set dal livello $i-1$ a j sono uguali). Possiamo quindi distinguere i seguenti casi:

  1) $k < j$**:** In tal caso l'algoritmo ritornerebbe *"Elemento non inserito"* solo per un livello inferiore a j. Si può ricorrere direttamente al livello j, dove, in ogni caso, esiste sempre un nodo candidato padre che soddisfa la condizione al passo 7 (opposta a quella del passo 3).

  2) $k >= j$**:** In tal caso l'algoritmo ritornerebbe *"Elemento non inserito"* al livello k, ed al livello $k+1$ può sempre essere inserito p in quanto la condizione al passo 7 risulta soddisfatta. Si può quindi ricorrere direttamente al livello $k+1$.

  Si noti infine che, per il caso singolo $k = i-1$, è necessario proseguire normalmente con il livello $i-1$ per evitare una ricorsione infinita.

- Una volta inserito il punto, deve essere gestito il caso in cui il padre non fosse un nodo già esplicito. Per fare ciò è utile associare ad ogni punto $q \in Q_i$ anche il livello superiore $l > i$ contenente un nodo esplicito relativo a q. Per l'inserimento di p come figlio di un nodo q del livello i è quindi sufficiente aggiornare correttamente i valori della mappa dei livelli j ed l, e aggiungere una Entry i avente associato come livello inferiore j e come superiore l.

L'operazione di rimozione, invece, è composta da una prima parte di ricerca di un punto che può essere modificata come visto per l'operazione di query. La seconda parte consiste invece nella rimozione effettiva del punto e dalla ristrutturazione dell'albero.
Di seguito vengono riportare le principali modifiche da apportare a quest'ultima parte dell'algoritmo:

- Indichiamo con $q \in C_i$ il nodo $Parent(p)$ ottenuto al passo 5 dell'algoritmo; allora la rimozione

di p potrebbe, in alcuni casi, rendere implicito il nodo q. Ciò si può risolvere rimuovendo la Entry relativa al livello $i-esimo$ dalla mappa di q, e aggiornando le informazioni del livello inferiore e superiore in cui il nodo padre risulta ancora esplicito. Si noti che questa operazione richiede di mantenere in Q_i le stesse informazioni viste per l'operazione di inserimento, al fine di eseguire le modifiche in $O(1)$.

- Consideriamo un qualsiasi nodo $q \in Q_{i-2}$ selezionato al passo 6 dell'algoritmo, e supponiamo che venga aggiunto come figlio di un nodo q' presente al livello j, al passo 12. Di conseguenza q verrebbe inserito dal livello $j-1$ al livello $i-1$, ed è quindi sufficiente aggiornare la mappa di q per il livello $j-1$ in cui risulta esplicito, e la mappa del nodo q' per il livello j.

Bibliografia

[1] R. Krauthgamer and J. Lee. Navigating nets: Simple algorithms for proximity search, *Proceedings of the 15th Annual Symposium on Discrete Algorithms* (SODA), 791–801, 2004.

[2] D. Karger and M. Ruhl. Finding nearest neighbors in growth restricted metrics, *Proceedings of the 34th Annual ACM Symposium on Theory of Computing* (STOC), 741–750, 2002.

[3] Beygelzimer, Alina, Kakade, Sham, and Langford, John. Cover trees for nearest neighbor. In *Proceedings of the 23rd International Conference on Machine Learning*, pp. 97–104, New York, NY, USA, 2006.

[4] J. Friedman, J. Bentley, and R. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3(3): 209–226, 1977.

[5] T. Kollar, PyCoverTree, (2008), GitHub repository, https://github.com/ngeiswei/PyCoverTree

[6] D. Crane, Cover-Tree, (2011), GitHub repository, https://github.com/DNCrane/Cover-Tree

[7] N. Löhndorf, covertree, (2013), GitHub repository, https://github.com/loehndorf/covertree