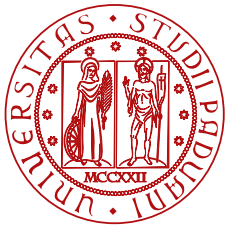

Università
degli Studi
di Padova

Dipartimento
di Ingegneria
dell'Informazione

Laurea Triennale in Ingegneria Informatica

Cover Trees: Analisi e Sperimentazione

Laureando

Guido Furlanetto

Student ID: 1187451

Relatore

Prof. Geppino Pucci

Università di Padova

Anno Accademico 2023-2024
Data di Laurea: 14/03/24

Sommario

La ricerca del vicino più vicino (Nearest Neighbor Search, NNS) rappresenta un pilastro fondamentale per applicazioni di intelligenza artificiale e sistemi di recupero informazioni, dove la rapidità e l'accuratezza sono essenziali. Una soluzione promettente al problema NNS si ottiene tramite l'impiego di Cover Trees, strutture dati ad albero innovative organizzate in livelli, dove ciascun nodo rappresenta un punto all'interno di un dataset in uno spazio metrico e ogni livello funge da "copertura" per il livello sottostante.

Lo scopo di questa tesi è l'analisi teorica e pratica delle operazioni principali legate ai Cover Trees con particolare interesse per la ricerca dei k-nearest neighbors. Viene discussa e analizzata l'architettura del Cover Tree, dimostrando come le sue proprietà intrinseche consentano una navigazione rapida dello spazio dei dati. Si analizzano i tempi d'esecuzione delle operazioni principali della struttura dati e si confrontano diverse implementazioni esistenti e accessibili pubblicamente, evidenziando le differenze in termini di prestazioni e uso delle risorse.

Il presente lavoro non solo fornisce un quadro teorico dettagliato per la comprensione dei Cover Trees ma propone anche una valutazione comparativa che serve da punto di riferimento per futuri sviluppi e applicazioni pratiche.

Indice

Capitolo 1

Introduzione

L'avvento dell'era digitale ha portato alla generazione di grandi volumi di dati in continua espansione, sfidando le capacità computazionali dei calcolatori moderni. La necessità di gestire ed elaborare queste enormi quantità di informazioni impone lo sviluppo di algoritmi avanzati che consentano di ridurre i tempi d'esecuzione. Questa esigenza è particolarmente sentita nell'apprendimento automatico (machine learning) e, più in generale, nell'intelligenza artificiale, dove si lavora con enormi dataset dinamici.

La ricerca del vicino più vicino (Nearest Neighbor Search, NNS) rappresenta una sfida cruciale in questi ambiti, fondamentale per applicazioni come la classificazione di immagini e l'analisi di sequenze genetiche. La sfida non è solamente ritrovare l'elemento più vicino secondo una data metrica di distanza, ma fare in modo che tale ricerca sia efficiente in spazi ad alta dimensionalità.

In questo contesto, i Cover Trees emergono come una soluzione promettente. Queste strutture dati si distinguono per la loro architettura particolare, che consente di condurre ricerche di prossimità con notevole efficienza. Questo lavoro di tesi si dedica a un'esplorazione approfondita dei Cover Trees, partendo dalle basi teoriche necessarie a comprendere il problema della NNS e le sue implicazioni.

Il capitolo successivo getta le fondamenta per un'analisi dettagliata, introducendo il problema della NNS e definendo i Cover Trees e i parametri che li descrivono, evidenziando la loro capacità di superare le limitazioni delle tecniche convenzionali. Nel terzo capitolo vengono introdotti gli algoritmi che regolano i Cover Trees, con spiegazioni supportate da pseudocodici, mentre nel quarto viene proposta un'analisi approfondita sui tempi di esecuzione delle operazioni per la loro gestione.

Infine, il quinto capitolo analizza tre implementazioni dei Cover Trees disponibili online, esaminando il codice e le prestazioni relative alle operazioni di inserimento, rimozione e ricerca dei k-nearest neighbors. Le conclusioni tratte offrono una prospettiva su come ulteriori tecniche possano affinare i tempi di esecuzione, ponendo le basi per futuri miglioramenti in questo campo di ricerca.

Capitolo 2

Definizioni di base

In questo capitolo vengono introdotte le definizioni di base utili per l'analisi prestazionale ed implementativa. Viene descritto il problema della ricerca nearest-neighbor e si fornisce una definizione dei concetti di dimensionalità intrinseca e delle costanti di riferimento. Fatto ciò, si presenta la struttura dati dei Cover Tree con le sue proprietà e menzionando nell'ultimo paragrafo la differenza tra rappresentazione implicita ed esplicita dei nodi, che sarà di fondamentale importanza per l'analisi dei tempi d'esecuzione.

2.1 Nearest-Neighbor Search

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 sintesi, l'obiettivo è identificare un punto $q \in S$ che minimizzi la distanza da un punto p, utilizzando una funzione di distanza d, conosciuta come metrica, in uno spazio metrico definito sull'insieme X. La complessità cresce esponenzialmente con l'aumentare dei punti in S, richiedendo un'efficace pre-elaborazione per garantire una soluzione tempestiva. La ricerca ha tradizionalmente esplorato spazi euclidei di bassa dimensione, limitazione che non contempla le esigenze di numerosi contesti applicativi. Nel campo dell'apprendimento automatico, il concetto di dimensionalità intrinseca (descritto in dettaglio nella prossima sezione) assume un'importanza cruciale, spesso superiore al numero di dimensioni spaziali del dataset. La struttura dati proposta mira a superare le limitazioni degli studi di Karger e Ruhl [5] e delle Navigating Net [7], che hanno introdotto analisi di spazi multidimensionali non esclusivamente euclidei.

Altre varianti della ricerca del vicino più prossimo includono la k-nearest-neighbors (K-NN), nota anche come batch query, che focalizza sulla ricerca dei k punti più prossimi a un dato

punto di input in S, e l'approccio alla ricerca approssimata del vicino più prossimo, che sarà dettagliatamente definito nei capitoli successivi.

2.2 Dimensionalità intrinseca

La dimensionalità intrinseca in ambito Machine Learning rappresenta il numero minimo di variabili indipendenti (dette "features" di un determinato dataset) necessarie per descrivere la variazione sottostante nei dati senza perdere informazioni significative. Di seguito vengono presentati due esempi, uno più pratico ed intuitivo e uno più teorico.

Esempio 2.1. *Individuare un luogo d'interesse di una località tramite le coordinate geografiche.*

Si immagini di avere un insieme di dati che rappresenta la posizione di un luogo d'interesse (ad esempio Palazzo Bo). In questo caso, si potrebbe avere le coordinate geografiche, latitudine, longitudine e altitudine del luogo, oltre a molte altre informazioni come il clima, il meteo giornaliero, la pressione dell'aria, l'area e il volume che occupa. La dimensionalità intrinseca qui si riferirebbe al numero minimo di caratteristiche, dette "features", necessarie per determinare efficacemente la posizione dell'edificio.

Se si dovesse indicare ad un turista la posizione corretta del Palazzo Bo basterebbe fornirgli tra tutte le caratteristiche elencate solo la latitudine e la longitudine. Aggiungere, quindi, la coordinata altitudine o le altre informazioni non porterebbe significativi miglioramenti alla comprensione della variazione nei dati. Ciò implica che, salvo casi particolari in cui in una zona ristretta vi sia un cambiamento improvviso di altitudine, le features necessarie per descrivere il problema sono solo 2 sebbene sia parte di uno spazio tridimensionale.

Nell'esempio sotto viene racchiuso questo concetto da un punto di vista algebrico indicando la dimensionalità intrinseca come le variabili linearmente indipendenti all'interno di una matrice.

Esempio 2.2. *Classificazione di documenti*

Si immagini di avere un grande set di dati che rappresenta documenti, e per ogni documento hai registrato il numero di volte che ogni parola appare. Quindi, si ha una matrice dove le righe rappresentano i documenti, le colonne rappresentano le parole, e i valori nella matrice indicano la frequenza di ogni parola in ogni documento.

Ora, potrebbe sembrare che la dimensionalità di questo spazio sia uguale al numero di parole, poiché ogni parola è una dimensione separata. Ma la dimensionalità intrinseca potrebbe essere molto più bassa. Si supponga che, in realtà, solo un piccolo insieme di parole sia veramente significativo per distinguere i documenti tra loro, mentre molte altre parole potrebbero essere presenti solo occasionalmente o in modo casuale.

In questo spazio vettoriale di parole, la dimensionalità intrinseca sarebbe il numero minimo di parole (variabili) necessarie per descrivere efficacemente le differenze essenziali tra i documenti.

Quindi, riducendo la dimensionalità intrinseca, si potrebbe trovare che un insieme molto più piccolo di parole è sufficiente per catturare la variazione principale nei documenti, semplificando notevolmente la rappresentazione dei dati e migliorando le prestazioni di algoritmi di Machine Learning su questo set di dati.

Sulla base di questa nozione sono state proposte due costanti *expansion constant* e *doubling constant*, in stretta relazione con la dimensione intrinseca di un dataset, calcolabili attraverso operazioni matematiche elementari.

2.2.1 Costante d'espansione

Dato un insieme S estratto da uno spazio metrico con funzione distanza d, indichiamo con $B_S(p, r)$ la palla chiusa di raggio r e centro p: $B_S(p, r) = q \in S : d(p, q) \leq r$

L'expansion constant di un insieme $S \subset X$, introdotta da Karger e Ruhl 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). Risulta quindi logico dare la seguente definizione di expansion dimension: $dim_{KR}(S) = \log c$. Sebbene esista un legame con la dimensione intrinseca dello spazio, esistono situazioni in cui la costante di espansione può assumere valori grandi anche in spazi di bassa dimensione. Prendiamo in considerazione, per esempio, uno spazio metrico euclideo unidimensionale, con l'insieme $S = \{2^i : i = 0, 1, 2, ..., n\}$. Il nostro obiettivo è identificare il valore di c più piccolo che rispetti la relazione precedentemente descritta per ogni sfera di raggio arbitrario e centro generico. In questo caso, il valore minimo di c si verifica nella situazione peggiore, ovvero quando $|B_S(2^n, 2^{n-1} - 0.5)| = 1$ e $|B_S(2^n, 2^n - 1)| = n$.

Ciononostante, per un insieme di dati tipico, privo di anomalie e con molteplici variabili, la costante di espansione si rivela un eccellente indicatore che facilita l'analisi computazionale e conserva un legame con la dimensione intrinseca del dataset.

2.2.2 Costante di raddoppio

La costante di duplicazione, impiegata nell'analisi delle Navigating Nets, offre un approccio più solido per esaminare strutture dati analoghe. La sua 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.

La dimensione di duplicazione si esprime come $dim_{KL}(S) = \log c$. A differenza della costante di espansione, la costante di duplicazione impedisce l'acquisizione di valori eccessivamente elevati per dimensioni d prestabilite. Infatti, nel contesto bidimensionale si necessitano al massimo 7 sfere di raggio $r/2$ per coprirne una di raggio r.

Nella trattazione seguente, si opterà per focalizzarsi sulla costante di espansione che è stata quella utilizzata dagli autori del lavoro [1] in cui è stato presentato il cover tree.

2.3 Cover Tree: definizione e proprietà

La struttura dati presentata da Beygelzimer, Kakade e Langford [1] 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. Si indica 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 $p, q \in C_i, d(p, q) > 2^i$.

Un esempio di albero che soddisfa ognuna delle tre invarianti viene mostrato in figura 2.1 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). Benché le regole di costruzione non precludano direttamente che un elemento identificato da un punto $p \in S$ al livello i possa avere come discendente diretto il punto p medesimo, questa condizione è implicitamente richiesta per mantenere valide le invarianti. Supponiamo, per assurdo, che esista un punto $p' \in S$, genitore di p al livello i; dato che p si trova sia in C_i che in C_{i-1}, dovrebbero essere soddisfatte contemporaneamente l'invariante di copertura, che implica $d(p, p') \leq 2^i$, e l'invariante di separazione, che afferma $d(p, p') > 2^i$, condizione chiaramente contraddittoria.

È importante notare come l'ultimo principio sia cruciale per l'introduzione di nuovi strati nella struttura. Senza di esso, ogni nuovo nodo potrebbe teoricamente essere aggiunto come discendente del nodo radice, compromettendo l'efficacia delle funzioni primarie sulla struttura. L'invariante di copertura, invece, limita la distanza massima tra i nodi discendenti e il loro antenato a 2^i, favorendo così un miglioramento significativo delle performance nell'esplorazione dell'albero, consentendo di selezionare solamente una frazione dei nodi.

Per quanto riguarda l'analisi formale della struttura, il numero di livelli è teoricamente infinito ($-\infty < i < \infty$), rendendo necessaria, nella sua implementazione pratica, la definizione di un

intervallo limitato che risulta adeguato per affrontare il problema in questione ($0 \leq i \leq 2$ nell'esempio dato).

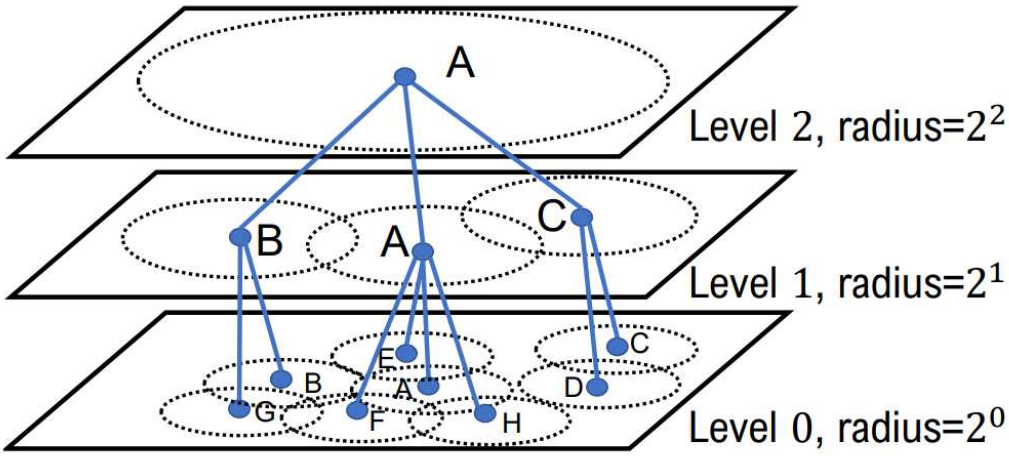

Figura 2.1 Esempio di cover tree con 8 punti distribuiti su 3 livelli. Nessun punto si trova nel cerchio di altri punti allo stesso livello (invariante di separazione) e ogni punto è almeno all'interno del cerchio di un altro punto a un livello superiore (invariante di copertura) [3].

2.4 Rappresentazione implicita ed esplicita

È fondamentale distinguere tra le forme implicita ed esplicita di rappresentazione delle strutture dati. Pur rispettando la regola della nidificazione, si evita di archiviare in memoria un numero elevato di nodi del Cover Tree, ottimizzando dunque sia l'efficienza algoritmica che l'utilizzo della memoria. Nella rappresentazione esplicita, la struttura include unicamente i nodi che possiedono almeno un figlio addizionale rispetto alla loro replica, o un genitore differente (vedi figura 2.2).

Ciò permette di non conservare in memoria sequenze di nodi identici, considerati superflui e non indispensabili, limitandosi a salvare soltanto i dettagli pertinenti alle posizioni all'interno dell'albero dove ciascun nodo è descritto in maniera esplicita.

Di seguito vengono riportate le tabelle che riassumono le complessità computazionali e lo spazio occupato dalla struttura dati con questo approccio, in paragone con le strutture dati presentate in [7] e [5].

	Cover Tree	Nav. Net	[KR02]
Spazio di costruzione	$O(n)$	$c^{O(1)}n$	$c^{O(1)}n \ln n$
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	$O(c^{16} n)$	$c^{O(1)}n \ln n$	$c^{O(1)}n \ln n$

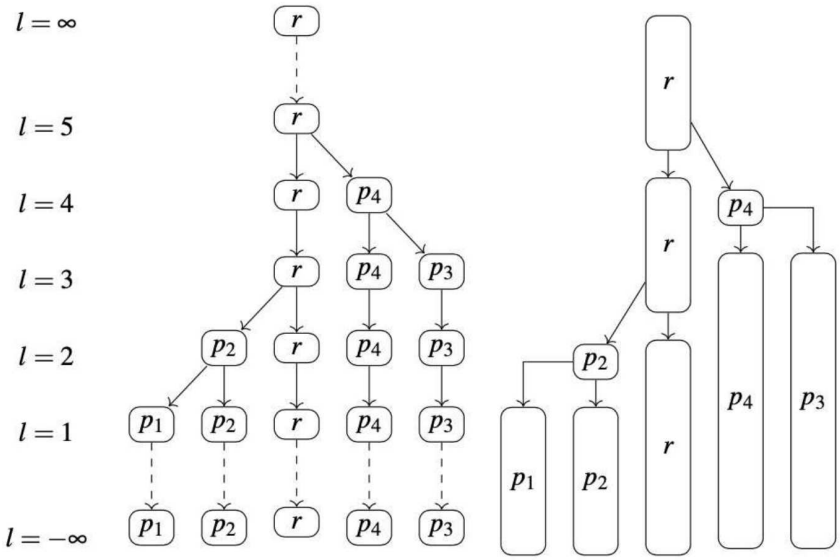

Figura 2.2 Rappresentazione implicita a sinistra ed esplicita a destra dello stesso Cover Tree

Si può osservare che 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' antenati di p che, per l'invariante (2), devono esserlo in due livelli diversi. Se q' è antenato di p al livello più basso $i-1$, allora p deve comparire anche ai livelli superiori per poter essere discendente di q, incluso lo stesso livello di q' (i). Ciò non è 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

Algoritmi fondamentali del Cover Tree

Questo capitolo esamina dettagliatamente gli algoritmi essenziali applicabili ai Cover Tree. Tra queste, si annoverano l'identificazione del nearest neighbor, nonché le procedure di inserimento e rimozione di un elemento. Ogni sezione del capitolo è dedicata alla descrizione di un algoritmo distinto, illustrandone il meccanismo operativo attraverso una sequenza di passaggi e confermandone l'efficacia mediante dimostrazioni matematiche. Si discutono altresì strategie per gestire diverse forme della ricerca di query, includendo l'approccio approssimativo alla ricerca del nearest neighbor e l'indagine sui k-nearest neighbors.

3.1 Ricerca del Nearest Neighbor

Il primo algoritmo, nonché il più importante, consiste nella ricerca del nearest neighbor 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 Neighbor

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

L'obiettivo principale dell'algoritmo è identificare, durante ciascuna iterazione, i punti in C_i che sono candidati ad essere nearest neighbor per il punto p, o che potrebbero, attraversando l'albero, condurci a identificarlo nei livelli inferiori. Esaminiamo pertanto con precisione ogni fase dello pseudocodice, facendo riferimento ai numeri delle righe di codice:

2. Data la radice dell'albero in C_∞, si parte da un unico nodo, considerandolo come possibile nearest neighbor.

3. L'esplorazione dell'albero procede un livello alla volta, attraverso un ciclo for, dove gli estremi ∞ e $-\infty$ facilitano la descrizione dello pseudocodice, indicando rispettivamente il livello iniziale e finale.

4. A questo punto, Q_{temp} rappresenta l'insieme dei discendenti diretti dei nodi in Q_i (garantendo che $Q_i \subseteq Q_{temp}$ grazie all'invariante di nidificazione), formando così un nuovo insieme per il livello $i-1$ che includerà necessariamente l'antenato del nearest neighbor.

5. Non è obbligatorio conservare ogni nodo in Q_{temp}: quelli più distanti da p di 2^i possono essere esclusi. Il criterio di esclusione sarà illustrato nel teorema che attesta la validità dell'algoritmo.

6. Poiché il set Q_i specifico include il nearest neighbor per quel livello, di conseguenza $Q_{-\infty}$ conterrà il nearest neighbor definitivo, dato che l'invariante (1) assicura la presenza di tutti i punti al livello $-\infty$.

Per approfondire l'analisi del funzionamento e l'accuratezza dell'algoritmo, è cruciale confermare che il criterio menzionato al punto 5 permetta di scartare solo i nodi non pertinenti come antenati del nearest neighbor di p. Tale affermazione è corroborata dal teorema seguente:

Teorema 2. *Sia T un cover tree su S, allora Find-Nearest(p) ritorna il nearest neighbor 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 neighbor.

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 $d(p,q') \leq d(p,Q_{temp})$, ossia con una distanza da p minore rispetto al candidato nearest neighbor in Q_{temp}. Ciò si può verificare attraverso la disuguaglianza triangolare; si ha infatti che $d(p,q')+d(q',q) \geq d(p,q) > d(p,Q_{temp})+2^i$ da cui si ottiene $d(p,q') > d(p,Q_{temp})+2^i-d(q',q) > d(p,Q_{temp})$ poiché $d(q',q) \leq 2^i$, che dimostra la correttezza dell'algoritmo. □

3.1.1 Ricerca dei k-nearest neighbors

Una variante significativa del problema principale è l'identificazione dei k punti più prossimi al punto q all'interno del Cover Tree. Questa variante introduce due sole modifiche nell'algoritmo, le quali mantengono le performance comparabili a quelle dell'approccio standard. Di conseguenza, non si procederà con ulteriori valutazioni della performance computazionale.

L'algoritmo modificato è il seguente:

Algoritmo 2 Ricerca dei k-nearest neighbors

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:

5. Per garantire il mantenimento di k vicini più prossimi in ogni strato dell'albero, si richiede un aggiustamento della condizione di selezione. Questo può essere realizzato valutando la distanza del k-esimo vicino più prossimo a p, utilizzando il valore fornito dalla funzione *Bound*. La logica sottostante alla prova di correttezza rimane invariata e per questo motivo non sarà oggetto di ulteriori approfondimenti.

6. A differenza del caso precedente, qui si procede alla restituzione di k elementi, che vengono selezionati mediante l'applicazione della nuova funzione *k_argmin* all'insieme $Q_{-\infty}$.

La dimostrazione di correttezza per questa variazione segue direttamente dal Teorema 2, adattandosi alla nuova logica implementativa.

3.1.2 Approssimazione del nearest neighbor

Un'altra variante di questo problema, che contribuisce a ottimizzare le performance dell'algoritmo, si focalizza sulla determinazione di un punto che, pur essendo vicino all'elemento in esame, non deve obbligatoriamente rappresentare il nearest neighbor. È sufficiente individuare un certo $q \in S$ per cui $d(p,q) \leq (1+\varepsilon)d(p,S)$, dove l'errore sulla misura di distanza rimane entro un margine definito dal parametro $\varepsilon > 0$.

Di seguito viene riportato l'algoritmo modificato:

Algoritmo 3 Ricerca approssimata del nearest neighbor

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** $argmin_{q \in Q_{-\infty}} d(p,q)$

Come osservato, il processo evita iterazioni complete tra i livelli, interrompendosi anticipatamente in base al nuovo criterio specifico. Il seguente teorema stabilisce che, all'interruzione

del ciclo, si mantiene la condizione $d(p, q) \leq (1 + \epsilon)d(p, S)$, confermando quindi la validità dell'analisi di correttezza.

Teorema 3. *Sia T un cover tree su S, allora* Find-Nearest(p, ε) *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 2^{i+1}. Ciò è dovuto al fatto che il punto $q \in Q_i$ più vicino a p è distante dal nearest neighbor 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 $2^{i+1} + 2^{i+1}(1 + 1/\varepsilon) \leq d(p, S) + 2^{i+1}$. 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

In aggiunta alla funzionalità di ricerca, è fondamentale che la struttura dati supporti un meccanismo per l'inserimento, consentendo così l'aggiunta progressiva di nuovi elementi. Il processo è illustrato attraverso il seguente 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:      else
6:          Q_{i-1} = q ∈ Q_temp : d(p, q) ≤ 2^i
7:          result = INSERT(p, Q_{i-1}, i − 1)
8:          if result = "Elemento non inserito" and d(p, Q_i) ≤ 2^i then
9:              Scelgo q ∈ Q_i tale che d(p, q) ≤ 2^i
10:             Inserisco p in Children(q)
11:             return "Elemento inserito"
12:     return result
```

Per ogni livello, l'algoritmo costruisce l'insieme Q_i, selezionando esclusivamente quei punti in C_i che possono essere considerati antenati potenziali di p. Di conseguenza, i punti $q \in C_i$ per cui $d(p, q) > 2^{i+1}$ sono esclusi poiché la loro distanza impedisce l'inclusione di p in qualsiasi sottoalbero con q come radice. Esaminiamo pertanto dettagliatamente ogni fase dello pseudocodice:

2-4. Nella fase iniziale, si cerca p creando progressivamente un nuovo insieme per il livello successivo, similmente a quanto fatto per la query. La ricerca procede ricorsivamente finché p non viene individuato e finché rimangono nodi in Q_{i-1}.

5-6. Identificato il nodo, si procede eliminandolo da tutti gli insiemi dove appare e si interrompe il legame con il nodo padre. Questo comporta l'eliminazione dei nodi espliciti a partire dal livello $i - 1$.

7-12. La rimozione di p e dei nodi correlati dall'albero può causare la disconnessione di alcuni discendenti. Per un discendente q di un nodo rimosso, se esiste un nodo a un livello superiore che mantiene l'invariante di copertura, esso può essere collegato direttamente; altrimenti, si inserisce un nuovo nodo (corrispondente a q) al livello inferiore e si ripete il processo risalendo l'albero. Questa procedura non contravviene a nessuna invariante e assicura che l'albero rimanga un cover tree valido.

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. La nostra analisi inizia con la dimostrazione che ogni elemento p non presente si inserisca correttamente in T. Se p non è parte di T, per costruzione, esiste un livello i per il quale la base della ricorsione si applica, dato che 2^i diminuisce tendendo a zero mentre $i \to -\infty$, e la distanza minima tra p e il suo nearest neighbor è positiva. Il criterio per l'inserimento a riga 7 diviene valido per qualche $j > i$, potenzialmente all'infinito, assicurando l'aggiunta di p.

Successivamente, si verifica che l'aggiunta di p preservi la struttura di cover tree, confermando così la validità delle invarianti di struttura. L'inserimento di p ai vari livelli garantisce la conservazione dell'invariante di nidificazione, e analogamente si mantiene l'invariante di copertura.

Per assicurare l'invariante di separazione, si esaminano tre scenari per $q \in C_{i-1}$:

1. $q \in Q$ e $q \in Q_{i-1}$. 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.

2. $q \in Q$ e $q \in Q_{i-1}$. si consideri 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.

3. $q \notin Q$. Se q non è presente in Q significa che il padre q' è stato rimosso ad un livello precedente $i' > i$, verificata si la condizione $d(p, q') > 2^i$. Con questi dati è possibile trovare un lower bound a $d(p, q)$: $d(p, q) \geq d(p, q') - \sum_{j=i'}^{i-1} 2^j$. 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:             Scegli 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)
```

Analogamente alla procedura di inserimento, l'algoritmo elabora un insieme di copertura Q_i ad ogni livello, includendo esclusivamente i nodi di C_i potenzialmente antenati di p. Dopo aver individuato il punto da eliminare, si procede con la modifica dell'albero. I passaggi chiave dello pseudocodice sono delineati di seguito:

2-4: La ricerca di p segna la prima fase, creando progressivamente un insieme per il livello successivo in maniera simile a quella adottata per la query. La ricorsione persiste finché p resta introvabile e Q_{i-1} non si svuota.

5-6: Identificato il nodo, ne segue la rimozione da ogni insieme in cui figura, insieme alla cancellazione del legame col nodo padre. Questo comporta l'eliminazione dei nodi espliciti a partire dal livello $i - 1$.

7-12: Eliminando p e i relativi nodi collegati, certi discendenti risulteranno disconnessi. Se un nodo a un livello superiore soddisfa l'invariante di copertura, può essere collegato direttamente; altrimenti, si inserisce un nuovo nodo corrispondente a q al livello inferiore, ripetendo il processo verso l'alto. Questa operazione non contravviene alle invarianti, preservando la validità del cover tree.

13-14: Se Q_{i-1} risulta vuoto, la ricorsione si arresta, mancando nodi che possano ricondurre a p. Diversamente, si prosegue al livello superiore finché p non viene rinvenuto.

Questa procedura di verifica mantiene una struttura simile ad altri algoritmi, facilitandone la correttezza. Un teorema specifico dimostra come tale ristrutturazione mantenga le tre invarianti fondamentali.

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

Spesso ci si trova di fronte a dataset che contengono migliaia di punti dati già disponibili, piuttosto che acquisirli progressivamente. In tali scenari, l'inserimento individuale di ogni elemento tramite la funzione dedicata può significativamente ridurre l'efficienza del sistema, dato che richiede di attraversare l'albero da capo ad ogni nuovo inserimento. L'obiettivo della funzione di costruzione è dunque quello di ottimizzare la complessità computazionale partendo da un insieme unificato di punti iniziali. L'algoritmo proposto opera come segue:

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:              Scegli 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⟩
```

Consideriamo un punto p al livello i. Questo processo inizializza un cover tree con radice in p, includendo tutti i nodi in $Near$ e un subset di quelli in Far. Qui, $Near$ comprende i punti a distanza non maggiore di 2^i da p, ognuno dei quali può essere integrato nell'albero rispettando le invarianti, poiché accessibile tramite una discesa dal nodo p. Al contrario, Far racchiude i punti p' e l'insieme S, dove $2^i < d(p, p') \leq 2^{i+1}$. In questa situazione, non è garantito che p' possa essere collegato, come nel caso in cui l'albero consista unicamente nel nodo p.

La strategia mira a inserire il maggior numero di elementi da Far, mantenendo le invarianti intatte. Utilizziamo la funzione **Split**$(d(p, \cdot), r, S_1, S_2, ...)$ per dividere i punti in $S_1, S_2, ...$ in due insiemi, $\langle Near, Far \rangle$, seguendo $d(p, q) \leq r < d(p, q) < 2r$ e rimuovendo i punti selezionati da $S_1, S_2, ...$

2-4. L'esecuzione si conclude quando $Near$ si svuota, dato che ogni $q \in Far$ implica $d(p, q) > 2^i$, infrangendo l'invariante di copertura.

5-6. La costruzione dell'albero da p procede in modo ricorsivo, generando sottalberi che si agganceranno alla radice p. Durante questa fase, è cruciale preservare l'invariante di nidificazione, affrontata separatamente; al punto 5 applica la funzione al diretto discendente di p, redistribuendo i punti in $Near$ e Far mediante split. I punti in Far non rientreranno nell'albero radicato in p a livello $i - 1$ a causa della loro distanza maggiore. La funzione restituisce il nodo P_{i-1} e un insieme di elementi non impiegati. Il punto 6 collega p_i a p_{i-1}.

7-10. Per gli altri punti, il procedimento è simile: in ogni iterazione si seleziona un $q \in Near$ per costruire un sottalbero con q come radice; qui, alcuni $q' \in Far$ potrebbero essere adatti all'inserimento, rispettando l'invariante di copertura. La funzione fornisce un set di nodi inutilizzati e lo stesso nodo q, che viene poi aggiunto come discendente di p.

11-12. Gli ultimi passi includono i punti di $Unused$ nei set $Near$ e Far per quelli non ancora inseriti.

La costruzione dell'albero inizia con Construct$(p \in S, < S - \{p\}, \varnothing >, +\infty)$, dove p è un punto qualunque in S, $Near$ raccoglie tutti i punti (eccetto p) da inserire sotto p, Far rimane vuoto e il livello di partenza è $+\infty$.

Per validare l'algoritmo, il seguente teorema ne conferma la correttezza:

Teorema 6. *Construct$(p \in S, < S - \{p\}, \varnothing >, +\infty)$ 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 $u_{i-1}, u_{i-2}, u_{i-3}, \ldots$, 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 comporta la fine della dimostrazione, infatti tutti i punti del set $Near$ vengono sempre aggiunti nel sottoalbero e, quindi in un livello sottostante ad i. □

3.5 Ricerca dei nearest neighbor per più punti

La procedura per eseguire interrogazioni è stata adattata per permettere la ricerca contemporanea del nearest neighbor per un insieme di punti attraverso l'utilizzo di un cover tree, indicato con T'. Questo adattamento può significativamente migliorare le prestazioni, soprattutto se si utilizza una funzione di pre-elaborazione per la costruzione di T'. Tale miglioramento è evidente in scenari con una alta sequenzialità e volume delle interrogazioni. La versione adattata dell'algoritmo è presentata di seguito:

Algoritmo 7 Ricerca dei nearest neighbors di più punti

1: **procedure** FIND-ALL-NEAREST(**query cover tree** p_j, **cover set** Q_i)
2: **if** $i = -\infty$ **then**
3: **return** $argmin_{q \in Q_{-\infty}} d(a, b)$ come nearest neighbor 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: $Q_{i-1} = \{q \in Q_{temp} : d(p_j, q) \leq \min_{q \in Q_d}(p_j, q) + 2^i + 2^{i+2}\}$
8: FIND-ALL-NEAREST(p_j, Q_{i-1})
9: **else**
10: FIND-ALL-NEAREST(q_{j-1}, Q_i) per ogni $q_{j-1} \in Children(p_j)$

La strategia adottata ricorda quella dell'algoritmo di query per più punti del KD-tree, benché presenti alcune differenze significative. Il procedimento prevede di percorrere il query cover tree e, associando, per ogni sottoalbero con nodo radice p_j, un insieme di copertura Q_i che include i nodi al livello i capaci di identificare il nearest neighbor per ogni punto del sottoalbero esaminato.

In questa operazione, all'espansione del query cover tree in più sottoalberi corrisponde l'ampliamento di Q_i (quando $j < i$), selezionando esclusivamente i nodi potenzialmente più vicini.

Per il processo di query relativo a un punto singolo, la logica di selezione è stata adeguata per incrementare la distanza di valutazione. Per analizzare questa logica si modifica la condizione di scelta al passo 7.

Definiamo q come il punto più prossimo a p dentro Q_{temp}, e q' come un punto qualunque di Q_{temp}. Definiamo altresì p' come il punto più distante raggiungibile da p (verso destra) e q'' come il punto più distante da q (verso sinistra).

Secondo quanto stabilito da teoremi precedenti, risulta che $d(p, q') = 2^{i+1}$ e $d(q', q'') = 2^i$. Inoltre, $d(p', q) = d(p, q) + 2^{i+1}$. Per evitare l'eliminazione di q' da Q_i, è necessario che $d(p', q') < d(p, q)$. Dalle sostituzioni, si deduce che $d(p', q'') < d(p, q) + 2^{i+1}$. Tuttavia, dato che $d(p', q'') = d(p, q') - 2^i - 2^{i+1}$, si arriva alla disuguaglianza $d(p, q) > d(p, q) + 2^{i+2} + 2^i$.

Capitolo 4

Analisi dei tempi di esecuzione

Come precedentemente accennato, l'analisi delle prestazioni dei Cover Tree è fortemente influenzata dalla dimensionalità intrinseca del dataset, la quale gioca un ruolo cruciale nell'efficacia degli algoritmi. Il seguente paragrafo illustra tre risultati chiave che stabiliscono un collegamento critico tra le prestazioni delle operazioni effettuate sulla struttura dati e la costante di espansione del dataset. Nei paragrafi successivi, vengono fornite dimostrazioni analitiche riguardanti le complessità computazionali associate alle operazioni di query, inserimento e cancellazione.

4.1 Analisi preliminare per la query

Si enunciano e spiegano 3 lemmi riguardo alcune proprietà strutturali dei cover tree.

Lemma 4.1 (*Width Bound*)**.** *Il numero di figli di qualunque nodo p è limitato da* c^4.

Dimostrazione. L'idea è quella di confezionare tante piccole palle disgiunte di raggio 2^{i-2} all'interno della palla $B(p, 2^{i+1})$. Ogni piccola palla può coprire al massimo un punto in C_{i-1}, e ciò impone un limite al numero di figli del nodo p.

Spiegazione passo per passo della dimostrazione:

1. Si considera un nodo p che si trova al livello i dell'albero.

2. Il numero di figli di p è limitato dalla cardinalità dell'intersezione tra la palla $B(p, 2^i)$ e il set C_{i-1}, che è sicuramente limitata dalla cardinalità dell'intersezione tra $B(p, 2^{i+1})$ e C_{i-1} essendo a raggio doppio.

3. La chiave dell'intera dimostrazione sta nel conteggio del numero di palle disgiunte di raggio 2^{i-2} che possono essere inserite all'interno della palla $B(p, 2^{i+1})$. Questo numero è limitato poiché ognuna di esse può coprire al massimo un punto in C_{i-1} (spiegazione dettagliata nel punto 5).

4. Per qualsiasi figlio q di p, la palla $B(p, 2^{i+1})$ è contenuta nella palla $B(q, 2^{i+2})$. Questo deriva dalla distanza tra p e q, che è limitata da 2^i. Questa inclusione implica che $|B(p, 2^{i+1})| \leq |B(q, 2^{i+2})| \leq c^4|B(q, 2^{i-2})|$. L'ultima limitazione è intrinseca nella definizione di costante di espansione sul raddoppio del raggio (in questo caso 4 raddoppi, da 2^{i-2} a 2^{i+2}) del raggio della palla.

5. Ogni palla $B(q, 2^{i-2})$ è disgiunta da tutte le altre palle $B(q', 2^{i-2})$ per q' in C_{i-1}. Questo è dovuto al fatto che i punti in C_{i-1} sono separati da almeno 2^{i-1} per la regola di separazione dei nodi dello stesso livello di un cover tree.

6. Poiché ogni palla $B(q, 2^{i-2})$ è contenuta in $B(p, 2^{i+1})$, il numero di palle disgiunte che possono essere collocate intorno ai figli q e quindi all'interno di $B(p, 2^{i+1})$ è limitato da $|B(p, 2^{i+1})|/|B(q, 2^{i-2})| \leq c^4$.

Questo dimostra che il numero di figli di p è limitato da c^4, come richiesto dal lemma. In sostanza, il lemma stabilisce una stretta limitazione al numero di figli di un nodo nel cover tree, il che è cruciale per l'analisi dei tempi di esecuzione e la comprensione della struttura dell'albero. □

La costante di espansione determina un limite superiore per l'accrescimento di una sfera centrata in p al variare del suo raggio (vedi capitolo 2). Attraverso il lemma seguente, ci proponiamo di stabilire un limite inferiore. Questa scoperta si rivela cruciale per dedurre l'altezza dell'albero cover in relazione alla costante c.

Lemma 4.2 (*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 $|B(p, 4r)| \geq (1 + \frac{1}{2})|B(p, r)|$.*

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 $B(p, r) \cap B(q, r)$ 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 $\frac{1}{2}|B(p, r)|$ trovato nella disuguaglianza (1), si ottiene $|B(p, 4r)| \geq (1 + \frac{1}{2})|B(p, r)|$. □

Il lemma finale rivela un'osservazione cruciale riguardante la profondità esplicita di un punto all'interno dell'albero, stabilendo una connessione diretta con la costante di espansione. Definiamo la profondità esplicita di un punto p come la quantità di nodi espliciti presenti nel cammino che collega il nodo radice al più basso livello dove p risulta essere esplicito.

Lemma 4.3 (*Depth Bound*)**.** *La profondità esplicita di un nodo $p \in S$ è $O(c^2 \log n)$.*

Dimostrazione. Consideriamo il set $S_i = \{q \in S : 2^{i+1} < d(p, q) < 2^{i+2}\}$. 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, un 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 $|B(p,4r)| \geq (1+\frac{1}{2})|B(p,r)| = 1 + \frac{1}{c^2}$. 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 $|B(p,4r')| \geq (1+\frac{1}{2})|B(p,r')| > (1+\frac{1}{2})|B(p,4r)| = (1+\frac{1}{c^2})^2$. 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 $c > 2$. Il numero di antenati espliciti di p in ogni S_i è costante e ciò completa la dimostrazione. □

4.2 Analisi delle prestazioni algoritmiche

Avendo ora a disposizione tutte le risorse necessarie per l'analisi delle prestazioni algoritmiche del Cover Tree, siamo in grado di esaminare il seguente teorema, che illustra le capacità prestazionali dell'algoritmo di ricerca:

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

Dimostrazione. Poniamo l'attenzione su Q^*, identificato come l'ultimo insieme Q_i contenente nodi espliciti. Dato che la profondità massima per ogni nodo è $k = O(c^2 \log n)$, il totale delle iterazioni per il passo 3 non supera $k|Q_i|$, limitato superiormente da $k \max_i |Q_i|$. Tale osservazione emerge analizzando le iterazioni dove Q_i risulta esplicito, minimizzando il numero di nodi in comune nei percorsi per raggiungere ogni nodo in Q^*.

Durante ogni iterazione, l'identificazione dei punti espliciti in Q_i richiede un tempo $O(\max_i |Q_i|)$ attraverso una ricerca sequenziale. Tenuto conto del massimo numero di iterazioni, la complessità complessiva si attesta su $O(k \max_i |Q_i|^2)$. Il quarto passo si concentra sull'espansione dei nodi espliciti in Q_i, un'operazione che, per sua natura, prescinde dal ciclo for, considerandola nel suo complesso. Di conseguenza, il numero totale di nodi espliciti presi in esame è $O(k \max_i |Q_i|)$, coerentemente con quanto esposto all'inizio; pertanto, la complessità computazionale del passo 4 si traduce in $O(kc^4 \max_i |Q_i|)$, dove il termine c^4 deriva dall'ampliamento di ogni nodo ai suoi figli, come postulato dal lemma 1. I passaggi successivi si svolgono in un lasso temporale minore rispetto al quarto, risultando in una complessità finale per la query dell'algoritmo di $O(kc^4 \max_i |Q_i| + k \max_i |Q_i|^2)$, somma dei due contributi. Per concludere la dimostrazione, verifichiamo che $\max_i |Q_i| \leq c^5$.

Esaminiamo Q_{i-1}, generato all'i-esima iterazione, e definiamo $d = d(p, Q)$. Possiamo riformulare Q_{i-1} come $Q_{i-1} = \{q \in Q : d(p, q) \leq d + 2^i\} = B(p, d + 2^i) \cap Q \subseteq B(p, d + 2^i) \cap C_{i-1}$, dato che Q rappresenta un sottoinsieme di C_{i-1} al livello i.

Dividendo l'analisi in due scenari, $d > 2^{i+1}$ e $d \leq 2^{i+1}$, affrontiamo separatamente ciascun caso.

Nel primo scenario, otteniamo: $|B(p, d + 2^i)| \leq |B(p, 2d)| \leq c^2 |B(p, \frac{d}{2})|$, dove la prima disuguaglianza deriva da $d > 2^{i+1}$ e la seconda applica la definizione di costante di espansione. Dato che da qualsiasi $q \in C_{i-1}$ possiamo avvicinarci a p di meno di 2^i, segue che $d \leq d(p, S) + 2^i$. Con $d > 2^{i+1}$, si deduce che $d(p, S) \geq d - 2^i > 2^i$. Questo implica che $B(p, \frac{d}{2}) = \{p\}$, portandoci a concludere che $|B(p, d + 2^i)| \leq c^2$ e quindi $|Q_{i-1}| \leq c^2$, chiarendo il primo scenario.

Nel secondo scenario, prendiamo un punto $q \in C_{i-1}$ e procediamo a calcolare il numero di sfere disgiunte di raggio 2^{i-2} che si inseriscono in $B(p, d + 2^i + 2^{i-2}) \cap C_{i-1}$. Ogni sfera copre al massimo un punto di $B(p, d + 2^i) \cap C_{i-1}$, e l'incremento di 2^{i-2} assicura la copertura dei punti più lontani. Seguendo un ragionamento analogo al primo lemma, arriviamo a stabilire un limite superiore per $|Q_{i-1}|$: $|Q_{i-1}| \leq |B(p, d + 2^i + 2^{i-2})| \leq |B(q, 2^{i+3})| \leq c^5 |B(q, 2^{i-2})|$.

In entrambi i casi, confermiamo che $|Q_{i-1}| \leq c^5$, completando la dimostrazione. □

Il seguente teorema si addentra nell'esame della complessità computazionale legata alle 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. Analizziamo la complessità dell'operazione di inserimento. Presupponendo l'esistenza di un elemento q nei set Q_i, Q_{i-1}, eQ_{i-2}, se un altro elemento q' risiede in Q_i, non potrà figurare in Q_{i-2} data la regola di separazione a tale livello che stabilisce $d(q, q') > 2^i$. La condizione imposta dallo step 5 dell'algoritmo garantisce che la distanza massima tra due elementi qualunque in Q_{i-2} sia limitata a 2^i. Definendo $k = c^{2 \log |S|}$ come la profondità esplicita di un punto, determiniamo il numero massimo possibile di set Q_i espliciti in relazione a k. Con quattro elementi, uno dei quali presente nei set Q_i, Q_{i-1}, Q_{i-2}, il numero di set Q_i espliciti si massimizza se i tre elementi sono distintamente presenti in ciascuno dei tre livelli considerati, e il quarto in nessuno di questi. Questo scenario, esteso a tutto l'albero, porta a identificare $3k$ set Q_i distinti, aggiungendo un ulteriore fattore k per il quarto elemento potenzialmente esplicito in altri livelli dell'albero.

Ogni passaggio dello pseudocodice non eccede la complessità di $O(\max_i |Q_i|)$. Il prodotto tra il numero di livelli contenenti set Q_i espliciti, proporzionale a k, e la complessità sopra citata, risulta in $O(k \max_i |Q_i|)$. Ogni set Q_i raggruppa elementi in S non distanti oltre 2^i da p, e seguendo il lemma iniziale, questo implica un limite massimo di c^4 elementi per set. La complessità computazionale finale è quindi $O(c^6 \log n)$.

Il processo di rimozione segue una logica simile, includendo fasi aggiuntive per la ristrutturazione dell'albero dopo l'eliminazione di un elemento. La fase di risalita dell'albero mantiene una complessità di $O(k \max_i |Q_i|)$, calcolata con lo stesso approccio utilizzato per l'inserimento. La complessità complessiva rimane quindi $O(c^6 \log n)$. □

Ci limiteremo a presentare gli enunciati degli algoritmi di costruzione e di ricerca batch, focalizzando l'analisi implementativa sulle funzioni fondamentali.
Nel documento originale di Beygelzimer et al.[1], gli autori forniscono uno schema dimostrativo per questi algoritmi, le cui dimostrazioni sono facilmente riconducibili a quelle precedentemente esaminate per le operazioni di inserimento, rimozione e ricerca del nearest neighbor.

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

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

Infine, è il caso di segnalare che recentemente sono state proposte analisi alternative basate sull'implementazione iterativa degli algoritmi di inserimento e rimozione che si basano su una definizione "estesa" dei Cover Tree originali. Il lavoro appena descritto viene presentato nel paper [9].

Capitolo 5

Esplorazione e valutazione di implementazioni esistenti

Per testare l'efficacia della struttura dati in questo ultimo capitolo vengono analizzate tre implementazioni popolari di cover tree sulla performance per gli algoritmi di inserimento, rimozione e ricerca k-nearest neighbor.
Questi valori di benchmark vengono confrontati rispetto all'implementazione originale di John Langford [8] fornendo in input quattro diversi dataset.
Le implementazioni utilizzate sono *ngeiswei-CoverTree* [6], *patVarilly-CoverTree* [10] entrambe scritte in linguaggio Python e *DNCrane-CoverTree* [2] costruito in C++ come il lavoro originale.

5.1 Specifiche hardware e software

Tutti i dati presentati nelle prossime sezioni sono stati collezionati tramite un elaboratore con le seguenti specifiche tecniche:

- CPU AMD Ryzen 7-5800H 3.2GHz (4.4Ghz Turbo) quad-core
- GPU NVIDIA GeForce RTX 3050
- RAM 16 GB

Mentre gli ambienti di sviluppo utilizzati sono:

- Google Colab con Python 3.12.2
- VS Code con C++ 20

5.2 Datasets

Come abbiamo dimostrato in precedenza le operazioni di inserimento, rimozione e query richiedono un determinato tempo d'esecuzione che si vuole testare con diversi tipi di datasets. In questo studio si utilizzano nel complesso 4 tipi diversi di dataset che variano per dimensionalità intrinseca e numero di punti.

- *bupa.data* - piccolo dataset numerico da 345 punti con bassa dimensione pari a 7;
- *pima.data* - dataset numerico composto da 768 punti con bassa dimensione pari a 9;
- *ionosphere.data* - piccolo dataset alfa-numerico composto da 351 punti con alta dimensione pari a 35;
- *pendigits.tra* - dataset numerico composto da 7494 punti con alta dimensione pari a 51;

5.3 Descrizione e confronto delle implementazioni

Analizzando le quattro implementazioni dei Cover Trees (*ngeiswei*, *patvarilly*, *DNCrane* e *JLangford*), possiamo osservare diverse strategie adottate per la gestione delle strutture dati e le operazioni di base come inserimento, rimozione e ricerca dei nearest neighbors. Di seguito, vengono analizzate le caratteristiche principali di ciascuna implementazione, evidenziando punti di forza e possibili aree di miglioramento.

L'implementazione ngeiswei-CoverTree si basa sulla creazione di un oggetto nodo per ogni punto con informazioni sui nodi figli mappati per livello. Implementa gli pseudocodici fedelmente per le operazioni principali presentando una versione snella e flessibile. Si utilizza inoltre, attraverso dizionari e liste, un approccio più diretto e semplificato nella gestione dei nodi, con una rappresentazione esplicita che evita ridondanze e potenzialmente riduce l'uso della memoria. Infatti, la memoria utilizzata è $O(n)$ dato che la mappa non contiene mai lo stesso nodo.
Un aspetto importante di ngeiswei-CoverTree è l'introduzione del concetto di supporto per il parallelismo nell'inserimento e nella ricerca, suggerendo una maggiore attenzione alla scalabilità e alle prestazioni in contesti di elaborazione ad alta intensità di dati.
Questa struttura non è ottimizzata in quanto potrebbe ridurre l'overhead della memoria e migliorare le prestazioni evitando di creare nuovi set o strutture dati temporanee ad ogni passo dell'algoritmo di query. Un altro aspetto critico è la gestione delle distanze e in particolare del calcolo futile della radice quadrata per la distanza.

La seconda implementazione che si considera è patvarilly-CoverTree. Rappresenta una versione evoluta di *ngeiswei-CoverTree* e si basa anche sul paper [4].
La classe CoverTree inizializza la funzione di distanza, la dimensione delle foglie e la base utilizzando dizionari. Il parametro leafsize determina il numero di punti in ogni nodo foglia e può influenzare il bilanciamento tra il tempo di costruzione dell'albero e il tempo di que-

ry. Analizzando la costruzione dell'albero (_build method) si nota che segue l'algoritmo di costruzione batch descritto da Beygelzimer [1] utilizzando una funzione ricorsiva (construct) per organizzare i punti in una struttura gerarchica. Dato che la profondità dell'albero aumenta logaritmicamente rispetto al numero di punti nel dataset, la costruzione dell'albero su set di dati molto grandi potrebbe potenzialmente causare un superamento del limite di ricorsione di Python, specialmente se l'albero diventa molto profondo a causa della distribuzione dei dati o della scelta dei parametri (come la base dell'albero).
Nella funzione di query (query method) invece si utilizza una coda di priorità per esplorare lo spazio dei punti. Inoltre, presenta strategie di ottimizzazione, come inserendo un approccio di valutazione pigra utilizzando le classi _lazy_child_dist e _lazy_heir_dist per calcolare le distanze tra i nodi figli e i loro antenati, riducendo il costo computazionale per mantenere aggiornate queste distanze durante la costruzione e la modifica dell'albero. Effettua anche una distinzione tra nodi interni (_InnerNode) e foglie (_LeafNode) che permette di trattare in modo efficiente i casi in cui un nodo contiene un numero di punti inferiore alla soglia leafsize, raggruppandoli per ridurre la profondità dell'albero e velocizzare le query.
Per migliorare le prestazione è utile considerare la possibilità di parallelizzare la costruzione dell'albero e le query. Molti dei calcoli, specialmente nella costruzione dell'albero, sono indipendenti e potrebbero beneficiare dell'esecuzione concorrente. Un'altra possibilità è cercare di implementare gli algoritmi di inserzione e cancellazione tramite un approccio iterativo menzionato precedentemente che fa riferimento al lavoro [9]. A differenza di ngeiswei-CoverTree sembra più focalizzato su applicazioni specifiche che richiedono alta precisione e ottimizzazione delle prestazioni.

La terza implementazione, DNCrane-CoverTree, è simile alle soluzioni in Python, con la differenza che la funzione getChildren(level) non ritorna mai il nodo stesso.
L'operazione di query cambia leggermente, riutilizza un solo set Q aggiungendo dei nodi figli o rimuovendo i punti con un'elevata distanza. Non sono presenti particolari ottimizzazioni.
Uno svantaggio rispetto alle strutture in Python 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 adottando le unordered map.
Come le altre implementazioni potrebbe beneficiare di un'esplorazione parallela o di tecniche di approssimazione per accelerare le query su dataset di grandi dimensioni.

Si conclude con l'implementazione degli autori del paper originali che segue fedelmente la teoria dei cover tree, con attenzione particolare alla scalabilità.
Utilizza un approccio basato sulla divisione in batch per gestire efficacemente grandi volumi di dati.
Data la sua già elevata efficienza su dataset di grandi dimensioni, ulteriori ottimizzazioni potrebbero riguardare l'uso di tecniche di calcolo parallelo o distribuito. Anche esplorare algoritmi di bilanciamento dinamico per i cover tree potrebbe aiutare a mantenere prestazioni ottimali anche

in presenza di dati che cambiano nel tempo.
L'unico aspetto negativo risiede nella complessità dell'implementazione che potrebbe rendere difficili ulteriori ottimizzazioni o adattamenti a casi d'uso specifici.

5.4 Esiti delle prove sperimentali

In questa sezione si analizzano i valori dei test relativi alle operazioni di inserimento, rimozione e k-nearest neighbors. Verranno forniti in input i quattro dataset prima menzionati, ad ogni implementazione. Successivamente si eseguiranno 100 test a partire dalle implementazioni in Python (*ngeiswei-coverTree* e *patvarilly-CoverTree*) e passando a quelle in C++ (*DNCrane-coverTree* e *JLangford-CoverTree*) per ogni operazione ricavandone una media. Verranno confrontati prima i valori di inserimento, poi rimozione e nearest neighbor, seguendo l'ordine dei dataset in *"bupa.data", "pima.data", "ionosphere.data" e pendigits.tra"*.
Per quanto riguarda l'inserimento e la rimozione non è stato possibile effettuare il test dei metodi all'interno della classe covertree di patvarilly-CoverTree perché non implementati. Perciò, senza modificare la struttura, per puri scopi analitici ci limiteremo ad utilizzare le altre tre.
Nel test dei k-nearest neighbors invece si considerano risultati ottenuti manipolando il valore di k che rappresenta il numero di punti più vicini a quello di query. I valori che sono stati utilizzati sono k=1, 5, 10.

5.4.1 Test per l'inserimento

Per il test sull'inserimento vengono analizzati i valori di speedup rispetto all'implementazione originale del cover tree di Beygelzimer e Langford, calcolati come rapporto tra i tempi d'esecuzione di inserimento generati da JLangford-CoverTree [8] e quelli delle due implementazioni.

Di seguito vengono riportati i valori dei risultati ottenuti in una tabella con i tempi medi effettivi di inserimento per ogni punto e poi rappresentati sotto forma di speedup in un istogramma mostrato in figura 5.1. Dopo di ciò viene dedicato uno spazio alle considerazioni sui dati ottenuti.

	bupa.data	pima.data	ionosphere.data	pendigits.tra
ngeiswei-CoverTree	6.28e-4	8.60e-4	3.22e-4	4.56e-3
DNCrane-CoverTree	2.67e-5	2.70e-5	3.40e-5	1.21e-4
JLangford-CoverTree	2.10e-7	3.21e-7	5.18e-7	1.21e-6

Qui sotto vengono riportati i valori di speedup rispetto all'implementazione originale che verranno poi riportati nell'istogramma sotto.

	bupa.data	pima.data	ionosphere.data	pendigits.tra
ngeiswei-CoverTree	3.35e-4	3.73e-4	1.61e-4	2.66e-4
DNCrane-CoverTree	7.85e-3	1.19e-2	1.52e-2	1.00e-2

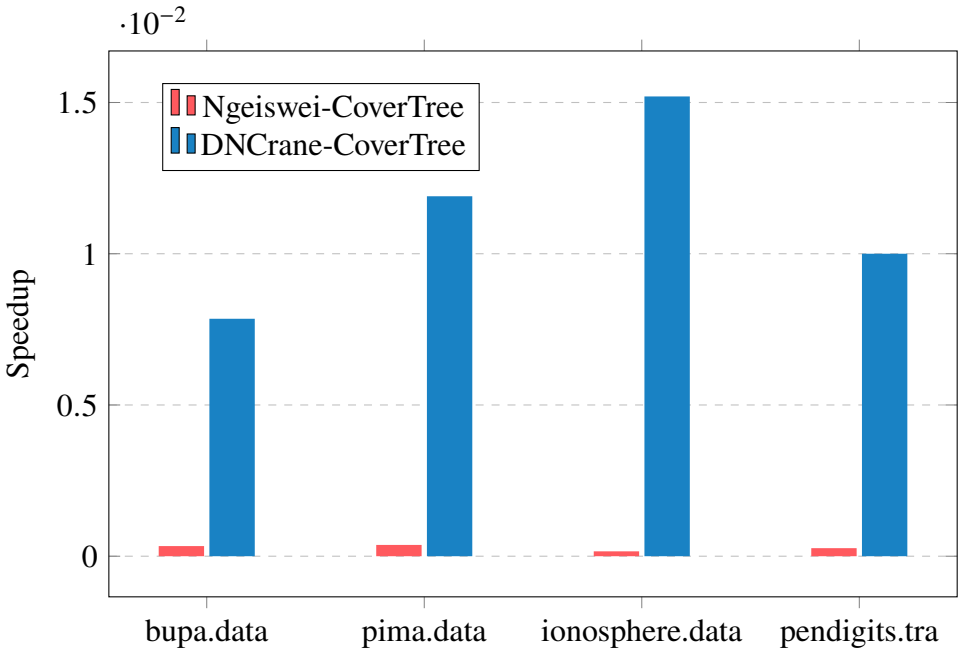

Figura 5.1 Speedup delle implementazioni per l'operazione di inserimento

Le conclusioni che si possono trarre guardando la tabella e l'istogramma è che la soluzione in C++ ha un vantaggio considerevole rispetto a quella in python in termini di speedup (tra le 20-100 volte più veloce), ma comunque entrambe le implementazioni risultato inefficienti rispetto a quella originale (supera l'ordine delle centinaia per valore di speedup rispetto a quella in c++). Nel caso di insiemi di punti con dimensionalità intrinseca elevata e con molti elementi (ionosphere.data, pendigits.tra) l'implementazione in Python ha le prestazioni peggiori confermando la difficoltà a gestire dataset di quella portata. Queste differenze non sono dovute alla complessità computazionale delle tre strutture dati, in quanto molto simili tra di loro, ma alle scelte implementative menzionate in precedenza.

5.4.2 Test per la rimozione

Come nel test sull'inserimento anche per la rimozione vengono analizzati i valori di speedup rispetto all'implementazione originale del cover tree di Beygelzimer e Langford.
Di seguito vengono riportati i valori dei risultati ottenuti in una tabella e poi rappresentati in un istogramma mostrato in figura 5.2. Dopo di ciò viene dedicato uno spazio alle considerazioni sui dati ottenuti.

Di seguito vengono riportati i valori dei risultati ottenuti in una tabella con i tempi medi effettivi di rimozione per ogni punto e poi rappresentati sotto forma di speedup in un istogramma mostrato in figura []. Dopo di ciò viene dedicato uno spazio alle considerazioni sui dati ottenuti.

	bupa.data	pima.data	ionosphere.data	pendigits.tra
ngeiswei-CoverTree	1.34e-4	5.89e-4	1.55e-3	2.58e-3
DNCrane-CoverTree	1.41e-5	1.43e-5	2.10e-5	4.42e-5
JLangford-CoverTree	1.37e-7	1.95e-7	2.41e-7	5.84e-7

Qui sotto vengono riportati i valori di speedup rispetto all'implementazione originale che verranno poi riportati nell'istogramma sotto.

	bupa.data	pima.data	ionosphere.data	pendigits.tra
ngeiswei-CoverTree	1.02e-3	3.31e-4	1.55e-4	2.26e-4
DNCrane-CoverTree	9.68e-3	1.36e-2	1.15e-2	1.32e-2

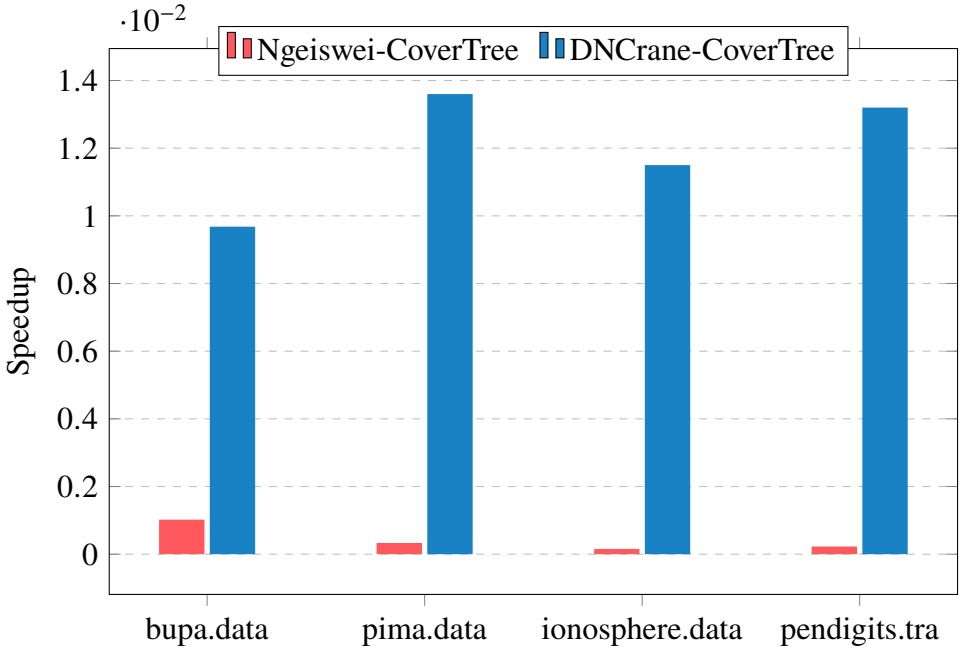

Figura 5.2 Speedup delle implementazioni per l'operazione di rimozione

Guardando la tabella e l'istogramma si nota una situazione simile al test di inserimento; la soluzione in C++ ha un vantaggio considerevole rispetto a quella in Python in termini di speedup. Si notano in generale miglioramenti su entrambe le implementazioni e per ogni dataset come valori di speedup, ma comunque entrambe le implementazioni risultano inefficienti rispetto a quella originale. L'implementazione in Python ha nuovamente le prestazioni peggiori confermando la semplicità della struttura dati implementata.

5.4.3 Test per i k-NearestNeighbors

Nel test sui k-nearest neighbors vengono analizzati i valori di speedup rispetto ad una ricerca naive/brute force senza utilizzo del cover tree. Questi risultati di speedup sono calcolati come rapporto tra i tempi d'esecuzione della ricerca lineare e quella tramite cover tree. In entrambi i casi l'output di ogni query sarà il punto stesso di ricerca, con una distanza pari a zero.
Di seguito vengono riportati i valori dei risultati sperimentali ottenuti dal test delle implementazioni nella ricerca nearest neighbor in una tabella e poi rappresentati in un istogramma mostrato in figura 5.3. Dopo di ciò viene dedicato uno spazio alle considerazioni sui dati ottenuti.

I dati sono suddivisi in tre tabelle in base al valore del parametro k:

- Caso con k=1:

	bupa.data	pima.data	ionosphere.data	pendigits.tra
ngeiswei-CoverTree	2.01	3.23	1.55	5.29
patvarilly-CoverTree	3.93	7.70	3.19	19.01
DNCrane-CoverTree	0.44	0.87	2.06	6.96
JLangford-CoverTree	0.79	2.37	1.66	11.73

- Caso con k=5:

	bupa.data	pima.data	ionosphere.data	pendigits.tra
ngeiswei-CoverTree	1.28	1.97	1.01	2.88
patvarilly-CoverTree	2.38	4.80	2.34	11.79
DNCrane-CoverTree	0.32	0.63	1.18	3.42
JLangford-CoverTree	0.58	1.32	0.79	5.18

- Caso con k=10:

	bupa.data	pima.data	ionosphere.data	pendigits.tra
ngeiswei-CoverTree	1.10	1.73	0.96	2.50
patvarilly-CoverTree	1.84	4.37	2.12	10.78
DNCrane-CoverTree	0.28	0.57	1.09	2.61
JLangford-CoverTree	0.60	1.20	0.74	4.33

Tramite l'istogramma si possono raggruppare tutti i dati e visualizzare un pattern all'aumentare del parametro k.

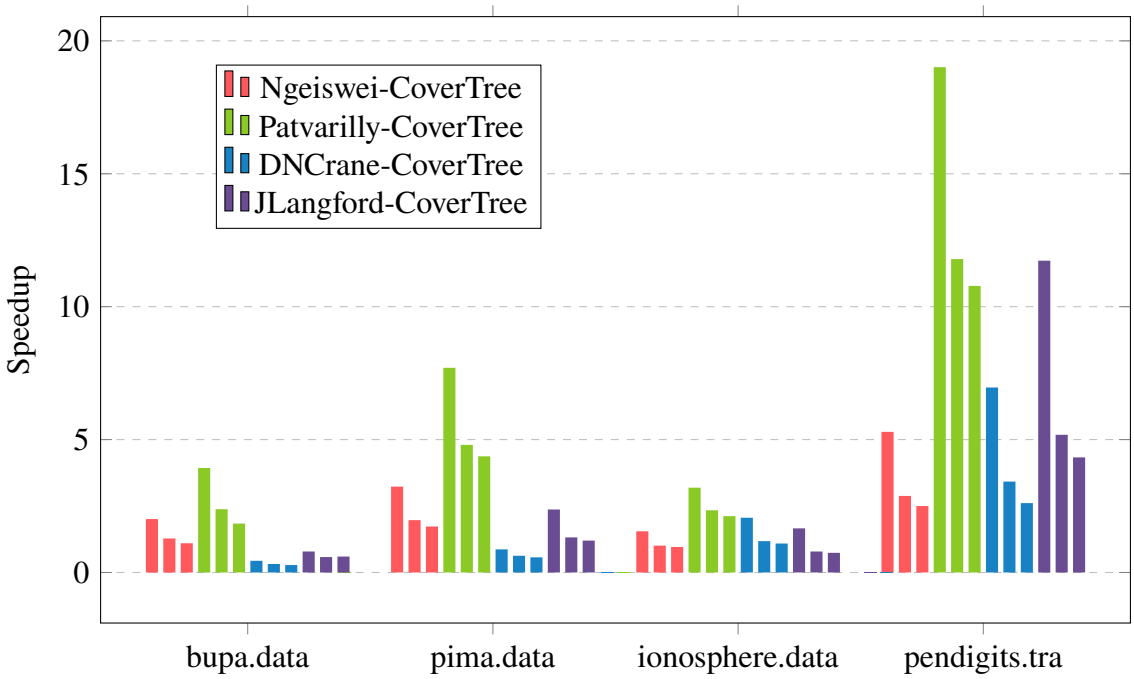

Figura 5.3 Speedup delle implementazioni per l'operazione di query con k=1,5,10

Il risultato più promettente delle prove sperimentali risiede nella ricerca k-nearest neighbor dove l'implementazione patvarilly-CoverTree come si nota risulta molto più performante in media su tutti i dataset anche rispetto a quello originale. L'ottimizzazione e le accortezze apportate alla costruzione del cover tree hanno mostrato discreti miglioramenti nelle prestazioni. L'unico problema di quest'implementazione è che con dataset molto grandi (>20000 punti) e con alta dimensionalità intrinseca, riporta l'errore di "max recursion depth exceeded" suggerendo che l'implementazione potrebbe beneficiare di un'ottimizzazione nella gestione della ricorsione o di un approccio iterativo alternativo per ridurre l'impatto della ricorsione profonda. Per le altre implementazioni in media, con dataset piccoli si presenta un vantaggio sulle prestazioni utilizzando l'implementazione in python, mentre con dataset con elevata dimensionalità, fatta eccezione per il caso di patvarilly-CoverTree, le implementazioni in C++ sono migliori.
Seguendo i suggerimenti proposti in precedenza, considerando di introdurre la parallelizzazione e un approccio iterativo agli algoritmi di inserimento e rimozione si riuscirebbe a migliorare ulteriormente le prestazioni producendo valori di speedup competitivi.

Bibliografia

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

[2] D. Crane. *Cover-Tree*. `https://github.com/DNCrane/Cover-Tree`. 2011.

[3] Yan Gu et al. “Parallel cover trees and their applications”. In: *Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures*. 2022, pp. 259–272.

[4] Mike Izbicki e Christian Shelton. “Faster cover trees”. In: *International Conference on Machine Learning*. PMLR. 2015, pp. 1162–1170.

[5] David R Karger e Matthias Ruhl. “Finding nearest neighbors in growth restricted metrics”. In: *Proceedings of the 34th Annual ACM Symposium on Theory of Computing (STOC)*. 2002, pp. 741–750.

[6] T. Kollar. *PyCoverTree*. `https://github.com/ngeiswei/PyCoverTree`. 2008.

[7] Robert Krauthgamer e James Lee. “Navigating nets: Simple algorithms for proximity search”. In: *Proceedings of the 15th Annual Symposium on Discrete Algorithms (SODA)*. 2004, pp. 791–801.

[8] J. Langford. *Cover-Tree*. `https://hunch.net/~jl/projects/cover_tree/cover_tree.tar.gz`. 2006.

[9] Paolo Pellizzoni, Andrea Pietracaprina e Geppino Pucci. “Fully dynamic clustering and diversity maximization in doubling metrics”. In: *Algorithms and Data Structures Symposium*. 2023, pp. 620–636.

[10] P. Varilly. *CoverTree*. `https://github.com/patvarilly/CoverTree`. 2012.